DBpia

주기억장치 데이터베이스를 위한 동시성 제어 관리자의 설계 및 구현

정회원 김 상 욱*, 장 연 정*, 김 윤 호*, 김 진 호**, 이 승 선***, 최 완***

Design and Implementation of a Concurrency Control Manager for Main Memory Databases

Sang-Wook Kim*, Yeon-Jeong Jang*, Yun-Ho Kim*, Jin-Ho Kim**, Seung-Sun Lee***, Wan Choi*** *Regular Members*

요 약

본 논문에서는 주기억장치 DBMS(main memory DBMS: MMDBMS)를 위한 동시성 제어 관리자의 설계 및 구현에 관하여 논의한다. MMDBMS는 디스크 기반 DBMS와는 달리 주기억장치 액세스만으로 데이터 검색 및 갱신을 수행하므로 전체 수행 비용 중 동시성 제어 관리자의 수행 비용이 차지하는 비중은 매우 크다. 따라서 효율적인 동시성 제어 관리자의 개발은 MMDBMS의 성능에 큰 영향을 미치게 된다. 본 연구에서 개발된 동시성 제어 관리자는 이단계 라킹 규약을 기반으로 하며, 다음과 같은 특징을 갖는다. 첫째, 락의 단위를 주기억장치의 물리적인 할당 단위인 파티션으로 설정함으로써 응용 분야의 특성 분석을 통하여 동시성과 락 관리 비용을 유연하게 조정할 수 있다. 둘째, 락에 관한 정보를 파티션 내부에서 직접 관리함으로써 락 관리 비용을 크게 줄일 수 있다. 셋째, 시스템 데이터의 물리적 일관성 유지를 위한 수단으로서 래치를 제공한다. 개발된 래치는 공유 모드와 배제 모드를 모두 지원하며, CPU 이용률의 극대화를 위하여 Bakery 알고리즘과 Unix의 세마포어 기능을 결합하는 방법을 사용한다. 넷째, 락에 의한 교착 상태의 해결을 위하여 락 대기 정보를 기반으로 시스템의 교착 상태 여부를 주기적으로 검출하는 기능을 제공한다. 본 논문에서는 트랜잭션 테이블의 상호 배제, 인덱스 혹은 시스템 카탈로그의 상호 배제, 실시간 응용의 지원 등 실제 구현에서 발생하는 중요한 이슈들에 관해서도 아울러 논의한다.

ABSTRACT

In this paper, we discuss the design and implementation of a concurrency control manager for a main memory DBMS(MMDBMS). Since an MMDBMS, unlike a disk-based DBMS, performs all of data update or retrieval operations by accessing main memory only, the portion of the cost for concurrency control in the total cost for a data update or retrieval is fairly high. Thus, the development of an efficient concurrency control manager highly accelerates the performance of the entire system. Our concurrency control manager employs the 2-phase locking protocol, and has the following characteristics. First, it adapts the partition, an allocation unit of main memory, as a locking granule, and thus, effectively adjusts the trade-off between the system concurrency and locking cost through the analysis of applications. Second, it enjoys low locking costs by maintaining the lock information directly in the partition itself. Third, it provides the latch as a mechanism for physical consistency of system data. Our latch supports both of the shared and exclusive modes, and maximizes the CPU utilization by combining the Bakery algorithm and Unix semaphore facility. Fourth, for solving the deadlock problem, it periodically examines whether a system is in a deadlock state using lock waiting information. In addition, we discuss various issues arising in development such as mutual exclusion of a transaction table, mutual exclusion of indexes and system catalogs, and realtime application supports.

* 강원대학교 컴퓨터정보통신공학부, ** 강원대학교 전자계산학과, *** 한국전자통신연구원 실시간 DBMS 팀
논문번호 : 99166-0424, 접수일자 : 1999년 4월 24일

www.dbpia.co.kr

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 사용하며, 이 결과 데이터를 검색 및 갱신할 때 디스크 액세스로 인하여 응답 시간이 지연되는 디스크 기반 DBMS(disk-based DBMS)의 문제를 근본적으로 해결한다[10][8][12][18]. 최근, 반도체 기술의 급격한 발전에 의한 주기억장치 가격의 하락으로 인하여 MMDBMS를 실시간 응용 분야에 실제로 적용하는 사례들이 점차 증가하고 있다.

본 논문에서는 MMDBMS를 위한 동시성 제어 관리자 개발에 관하여 논의하고자 한다. 동시성 제어 관리자는 다수의 트랜잭션들에 의하여 동시에 액세스되는 데이터베이스의 일관성을 보장해 주는 DBMS의 서브 컴포넌트이다[21]. 현재까지 알려진 동시성 제어 기법들은 크게 이단계 라킹 규약(two-phase locking protocol)[21], 타임 스탬프 순서화 기법(time-stamp ordering scheme)[4], 낙관적 기법(optimistic method)[5], 다중 버전 기법(multiple version method)[9] 등으로 분류된다. 이단계 라킹 규약은 데이터를 액세스하기 전에 반드시 락을 먼저 획득하도록 함으로써 동시성을 제어하는 방법이며, 그 실용성으로 인하여 현재 상용 시스템에서 널리 사용되고 있다[7].

디스크 기반 DBMS에서는 데이터베이스가 디스크에 저장되어 있으므로 데이터 검색 및 갱신을 위해서 다수의 디스크 액세스가 요구된다. 반면, 동시성 제어를 위한 연산은 주기억장치 내에서 수행되므로 이 비용은 디스크 액세스 비용과 비교하면 상대적으로 매우 작다. 따라서 전체 데이터 검색 및 갱신 비용 중 동시성 제어 관리자의 수행 비용이 차지하는 비중은 미미하다. 반면, MMDBMS내에서는 몇 번의 주기억장치 액세스만으로 데이터 검색 및 갱신을 수행할 수 있으므로 전체 비용 중 동시성 제어 관리자의 수행 비용이 차지하는 비중은 매우 크다. 따라서 MMDBMS를 위한 효율적인 동시성 제어 관리자의 개발은 전체 시스템 성능에 큰 영향을 미치게 된다.

본 논문에서는 현재 ETRI 실시간 DBMS 팀과 강원대학교 정보통신공학과 데이터 및 지식공학 연구실이 공동으로 개발 중인 차세대 MMDBMS Tachyon을 위한 동시성 제어 관리자의 설계 및 구현에 관하여 논의하고자 한다1). Tachyon은 Unix 환경을 기반으로 하며, 데이터베이스 전체를 Unix의 공유 주기억장치(shared memory)[3]내에서 관리한다. 또한, 실시간 응용을 지원할 수 있도록 다수의 트랜잭션들을 고속으로 처리하며, 마감 시간(deadline)의 개념을 효과적으로 지원하는 것을 주요 목표로 한다.

이단계 라킹 규약을 기반으로 본 연구에서 개발된 동시성 제어 관리자는 다음과 같은 특성을 갖는다. 첫째, 락의 단위를 주기억장치의 할당 단위인 파티션(partition)으로 선정함으로써 지나친 동시성 저하나 락 처리 비용의 증가를 방지하며, 파티션의 크기를 응용 분야의 특성에 맞추어 설정하도록 함으로써 트랜잭션의 동시성과 락 처리의 비용을 유연하게 조정할 수 있다. 둘째, 락 정보의 관리를 위하여 기존의 디스크 기반 DBMS에서 널리 사용하던 해쉬 구조를 사용하지 않고 파티션 내에서 락을 직접 관리하도록 함으로써 락의 처리 비용을 극소화한다. 셋째, 시스템 데이터의 물리적인 일관성 유지를 위하여 공유 모드와 배제 모드를 모두 지원하는 래치(latch)[15][20]를 지원한다. 개발된 래치는 Bakery 알고리즘[Sil98]과 Unix의 세마포어[3]의 특성을 혼합함으로써 임계 구역(critical section)을 최소화하고, CPU의 이용률을 극대화한다. 넷째, 이단계 라킹 규약에서 필연적으로 발생되는 교착 상태(deadlock)[21]를 해결하기 위하여 대기 그래프(wait-for-graph)[6][13]를 이용하여 교착 상태를 검출하는 기능을 지원한다. 이 밖에도 본 논문에서는 시스템 카탈로그와 인덱스의 상호 배제, 트랜잭션 테이블의 상호 배제, 실시간 요구 사항의 지원 등 동시성 제어 관리자의 개발 시에 발생하는 실질적인 이슈에 대한 해결책도 함께 제시한다.

본 논문의 구성은 다음과 같다. 제 2장에서는 개발된 동시성 제어 관리자에서 선정한 락의 단위와 그 선정 배경에 대하여 논의한다. 제 3장에서는 MMDBMS를 위한 새로운 락 관리 기법의 개발에 관하여 논의한다. 제 4장에서는 시스템 데이터의 물

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

www.dbpia.co.kr

리적인 일관성 보장을 위한 효과적인 래치의 관리 기법에 관하여 논의한다. 제 5장에서는 대기 그래프를 기반으로 하는 교착 상태 검출기의 개발에 관하여 논의한다. 제 6장에서는 본 주제와 연관된 기타 구현 이슈에 관하여 언급한다. 제 7장에서는 결론을 내리고, 향후 연구 방향을 제시한다.

Ⅱ. 락 단위의 선정

락 단위(lock granularity)란 락의 대상이 되는 데이터의 크기를 의미하며, 락 단위의 선정에 따라 시스템 동시성 및 락 처리 비용이 결정된다[21]. 락의 단위로서 데이터베이스, 세그먼트, 페이지, 레코드 등을 사용할 수 있다. 락의 단위가 큰 경우에는 처리 비용 측면에서는 효과적이지만, 시스템 동시성 측면에서는 효과적이지 못하다. 반면, 락의 단위가 작은 경우에는 시스템 동시성 측면에서는 매우 효과적이지만, 락 처리 비용 측면에서는 효과적이지 못하다[18].

디스크 기반 DBMS에서는 주기억장치 액세스만을 유발하는 락의 획득 및 반환 비용이 디스크 액세스를 유발하는 데이터 검색 및 갱신 비용과 비교하여 매우 작다. 따라서 락 처리 비용의 크기를 고려하는 대신 시스템 동시성을 극대화시키는 방법을 주로 사용한다. 이 결과, 현재 대부분의 상용 DBMS에서 가장 널리 사용하는 락의 단위는 레코드이다.

반면, MMDBMS에서는 데이터 검색 및 갱신이 주기억장치 액세스만을 요구하므로 그 비용이 매우 작으며, 따라서 락의 획득 및 반환을 위한 비용이 상대적으로 크게 부각된다. 또한, MMDBMS에서 트랜잭션들은 매우 빠른 속도로 처리되므로 락을 획득하는 시간이 매우 짧으며, 이 결과, 트랜잭션들 간의 락 충돌(lock conflict) 발생 가능성이 작아진다. 따라서 디스크 기반 DBMS에서와 같이 락의 단위를 작게 함으로써 얻어지는 장점이 상실된다[18].

이와 같은 관찰을 통하여 MMDBMS에서 보다 큰 락 단위를 사용하고자 하는 연구가 시도된 바 있다. 참고 문헌[12]에서는 데이터베이스 전체를 락의 단위로 선정하는 방식을 제안하였다. 이러한 방식을 사용하는 경우, 락 처리를 위한 비용이 크게 줄어든다. 또한, 모든 트랜잭션은 단 하나의 락만을 획득하므로 교착 상태를 검출하기 위한 부가적인 작업들도 제거할 수 있다. 반면, 데이터의 읽기와 쓰기를 시도하는 트랜잭션들은 병행 수행이 불가능하므로 시스템 동시성이 크게 저하된다.

본 연구에서는 이와 같이 레코드나 데이터베이스와 같은 극단적인 크기의 락 단위를 MMDBMS에서 사용하는 것은 적절하지 않음을 파악하고, 우선 이 두 종류의 락 단위들 사이에 위치하는 세그먼트와 파티션을 락 단위의 후보로 고려하였다. 세그먼트(segment)란 같은 종류의 레코드들을 저장하는 논리적인 단위로서 관계 DBMS에서의 릴레이션과 대응된다. 파티션(partition)은 레코드들을 저장하는 고정된 크기의 물리적인 저장 단위로서 세그먼트의 구성 요소이다. 이것은 주기억장치 할당의 단위이며, 디스크 기반 DBMS에서의 페이지와 같은 역할을 한다.

Starburst[19]에서는 세그먼트 단위의 락을 사용하고 있다. 그러나 트랜잭션들이 같은 세그먼트내의 레코드들을 함께 액세스하는 경우 데이터베이스를 락의 단위로 선정하는 방식과 같은 심각한 동시성 저하의 문제가 발생할 수 있다. 또한, 세그먼트의 크기는 저장된 레코드의 수에 따라 달라지므로 시스템 동시성의 예측이 어려워진다. 현재, Starburst에서는 레코드 단위의 락과 세그먼트 단위의 락을 응용의 특성에 의하여 동적으로 변환시키는 방안을 사용하고 있다. 그러나 락 처리 시 매번 상황을 점검해야 하는 별도의 비용을 요구하며, 이것은 MMDBMS의 성능 저하의 원인이 된다.

이와 같은 관찰을 통하여 본 연구에서는 락의 단위로서 주기억장치 할당의 단위인 파티션을 선정하였다2). 이 결과, 데이터베이스나 세그먼트를 락의 단위로 사용하는 경우에 발생할 수 있는 심각한 동시성 저하의 문제가 해결되며, 레코드를 락의 단위로 사용하는 경우에 발생하는 큰 락 처리 비용 문제가 해결된다. 또한, 파티션은 그 크기 조정이 용이하므로 동시성 향상이 요구되는 경우에는 파티션의 크기를 축소하고, 반대로 락 처리의 비용 절감이 요구되는 경우에는 파티션의 크기를 확대함으로써 동시성과 락 처리 비용을 유연하게 조정할 수 있다. 즉, 해당 응용 분야에서 감수할 수 있는 락 처리 비용을 분석함으로써 최적의 동시성을 지원하는 파티션의 크기를 도출할 수 있다.

2) 디스크 기반 DBMS에서는 높은 동시성과 낮은 락 처리의 비용을 위하여 다중 단위 라킹(multiple granularity locking)을 지원하기도 한다[21]. 그러나 MMDBMS에서 이러한 다중 단위 라킹을 지원하는 경우, 락 처리의 비용이 심각하게 증가된다. 따라서 본 연구에서는 이러한 다중 단위 라킹의 개념을 별도로 도입하지 않고, 파티션을 유일한 락 단위로 사용하였다.

www.dbpia.co.kr

Ⅲ. 락 관리 기법

본 장에서는 MMDBMS에 적합한 락 관리 기법을 제안한다. 먼저, 제 3.1절에서는 기존의 디스크 기반 DBMS의 락 관리 기법을 설명하고, 이를 MMDBMS에 직접 적용하는 경우의 문제점을 지적한다. 제 3.2절에서는 제안하는 새로운 락 관리 기법에 관하여 기술하고, 그 장단점에 관하여 논의한다.

3.1. 디스크 기반 DBMS에서의 락 관리 기법

기존에 개발된 대부분의 디스크 기반 DBMS에서는 락에 관한 정보를 관리하기 위하여 해싱을 사용한다[21]. 그림 3.1은 해싱을 기반으로 하는 락 관리 기법의 기본 구조를 나타낸 것이다. 락의 대상이 되는 데이터에 해싱 함수를 적용함으로써 해쉬 테이블내의 엔트리를 찾게 된다. 각 해쉬 엔트리는 락 헤더들의 리스트를 가리키며, 각 락 헤더는 락 요청들의 리스트를 가리킨다. 같은 리스트에 속한 락 헤더들은 해쉬 충돌에 의한 해쉬 체인을 의미하며, 같은 리스트에 속한 락 요청들은 같은 데이터에 대하여 다수의 트랜잭션들이 락을 획득하고자 하는 것을 의미한다. 락 헤더 블럭과 락 요청 블럭은 새로운 락이 요청되는 경우 각각의 풀 내에서 동적으로 할당되며, 락이 반환되는 경우 각각의 풀로 다시 반환된다. 또한, 각 트랜잭션은 자신의 락 요청 블럭들을 트랜잭션 락 리스트 형태로 관리한다.

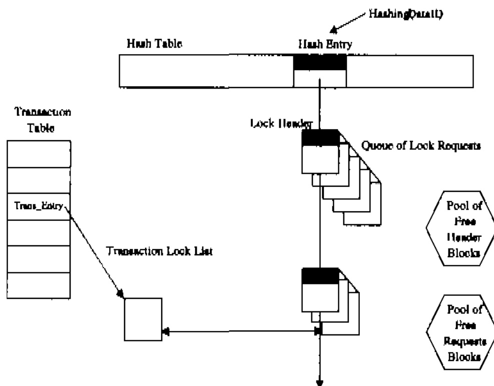

그림 3.1. 해싱을 기반으로 하는 디스크 기반 DBMS의 락 관리 기법의 기본 구조

이와 같이 기존 디스크 기반 DBMS의 해쉬 기반 락 관리 기법이 가지는 가장 큰 장점은 획득되거나 요청된 락에 관한 정보만을 동적으로 관리하므로 불필요한 주기억장치의 낭비를 방지할 수 있다는 것이다. 그러나 이 기법을 MMDBMS에 그대로 적용하는 경우 다음과 같은 문제점들을 발생시킨다.

첫째, 해쉬 구조를 사용하는 데서 발생하는 오버헤드이다. 해싱은 해싱 함수의 수행과 해쉬 체인의 관리 등 상당한 CPU 시간을 요구하는 높은 비용의 연산이다. 디스크 기반 DBMS에서 해쉬 기반 락 관리 기법을 사용한 이유는 디스크 액세스 비용과 비교하면 주기억장치 연산만을 요구하는 해싱의 비용을 무시할 수 있었기 때문이다. 그러나 MMDBMS에서는 모든 데이터가 주기억장치 내에 상주하므로 이러한 해싱의 비용이 실제 데이터 검색 및 갱신 비용보다도 커질 수 있다.

둘째, 해쉬 체인의 상호 배제를 위한 오버헤드이다. 해쉬 체인은 시스템 내에서 수행중인 다수의 트랜잭션들이 함께 액세스하는 공유 데이터에 해당된다. 즉, 하나의 해쉬 체인을 다수의 트랜잭션들이 읽거나 쓰게되는 경우가 발생하므로 이에 대한 상호 배제가 요구된다. 디스크 기반 DBMS에서는 이러한 상호 배제 비용도 무시할 수 있었으나, MMDBMS에서는 이러한 비용이 상대적으로 크게 부각된다.

셋째, 락 헤더 블럭들의 동적 관리를 위한 오버헤드이다. 기존의 해쉬 기반 락 관리 기법에서는 획득되거나 요청된 락에 대해서만 정보를 유지하므로 락 헤더 블럭들이 동적으로 관리된다. 따라서 락 헤더 블럭이 요청되거나 반환되는 경우 매번 락 헤더 블럭의 풀에서 할당되거나 반환되어야 한다. 또한, 락 헤더 블럭의 풀은 공유 데이터에 해당되므로 이에 대한 상호 배제가 요구된다. 이러한 두가지 비용은 MMDBMS의 성능을 저하시키는 원인이 된다.

3.2. 제안하는 락 관리 기법의 기본 전략

본 절에서는 MMDBMS에 적합한 락 관리 기법의 기본 전략을 설명하고, 그 장단점에 관하여 논의한다.

제안하는 기법에서는 해쉬 기반 락 관리 기법과는 달리 락의 대상이 되는 데이터 내부에 직접 락 정보를 관리하는 방법을 사용한다. 본 개발에서 채택한 락의 단위는 파티션이므로 락에 관한 정보를 가지는 락 헤더가 각 파티션 내부에 존재한다. 그림 3.2는 제안하는 락 관리 기법의 기본 구조를 나타낸 것이다. 락 헤더가 해당 파티션 내부에서 관리되므로 해쉬 테이블과 락 헤더 블럭의 풀이 사용되지 않음을 볼 수 있다.

이와 같이 락 헤더를 파티션 내부에서 관리함으

www.dbpia.co.kr

로써 전술하였던 해쉬 함수 및 해쉬 체인에서 발생하였던 오버헤드, 해쉬 체인의 상호 배제를 위한 오버헤드, 그리고 락 헤더 블럭들의 동적인 관리를 위한 오버헤드 등 기존의 해쉬 기반 락 관리 기법을 MMDBMS에 그대로 적용하는 경우의 문제점들이 자연스럽게 해결된다.

요청되거나 획득된 락과 대응되는 락 헤더를 동적으로 관리하는 해쉬 기반 기법과는 달리 제안된 기법에서는 모든 파티션내에 락 헤더를 정적으로 관리한다. 이 결과, 사용되지 않는 락 헤더들을 위한 주기억장치 공간이 일부 낭비된다. 그러나 파티션을 락의 단위로 사용하는 경우의 이러한 저장 공간의 낭비는 미미하며3), 특히 MMDBMS 전체의 성능 개선 효과를 고려할 때 이러한 오버헤드는 감수할 만 하다.

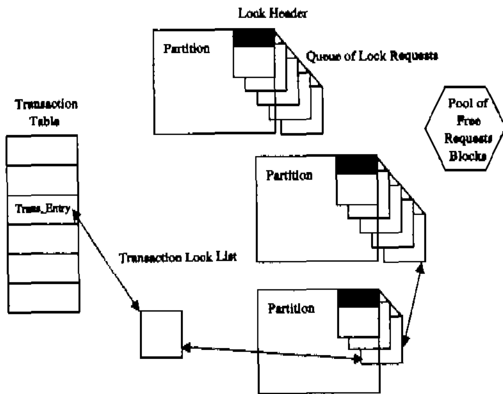

그림 3.2. 제안하는 락 관리 기법의 기본 구조

3.3. 제안하는 락 관리 기법을 위한 자료 구조

본 절에서는 그림 3.2와 관련하여 제안하는 락 관리 기법을 실제로 구현하는데 필요한 주요 자료 구조에 대하여 설명한다.

먼저, LockHeader는 각 파티션 내에 존재하는 락 헤더를 의미하며, 해당 파티션에 락이 걸려있는지의 여부를 판단할 수 있도록 한다. LockHeader는 latch, requestQueue, grantedMode, waiting 등으로 구성된다. latch는 이 LockHeader의 물리적인 일관성 보장을 위한 상호 배제의 수단으로서 사용되며, 이에 관한 자세한 내용은 제 4장에서 보다 자세히 다룬다. requestQueue는 이 파티션에 요청된 락에 관한 정보를 가지는 LockRequest들의 리스트를 가리키는 포인터이다. grantedMode는 현재 이 파티션에 락이 걸려 있는 경우, 걸려 있는 락의 모드가 공유 모드(shared mode)[21]인지 혹은 배제 모드(exclusive mode)인가를 나타낸다. 공유 모드는 해당 데이터를 단순히 읽으려는 다수의 트랜잭션들의 액세스를 허용하며, 쓰려는 트랜잭션의 액세스는 허용하지 않는다. 반면, 배제 모드는 해당 데이터를 쓰려는 단 하나의 트랜잭션의 액세스만을 허용한다. waiting은 이 파티션에 걸려있는 락이 반환되기를 기다리며 대기하고 있는 다른 트랜잭션이 존재하는지를 나타낸다.

LockRequest는 락 요청 블럭을 의미하며, 트랜잭션이 파티션에 락을 요청하여 획득한 경우나 대기 중인 경우 이에 관한 정보를 관리한다. 락 요청 블럭의 풀 내에서 관리되는 LockRequest는 락의 요청시 풀로부터 할당되며, 락의 반환시 다시 풀로 반환된다. 할당된 LockRequest는 LockHeader의 requestQueue가 가리키는 리스트상에 연결된다. LockRequest는 requestQueueNext, requestQueuePrev, status, grantedMode, convertingMode, transEntry, next, prev 등으로 구성된다.

requestQueueNext와 requestQueuePrev는 락 요청들을 이중 연결 리스트로 유지하기 위한 포인터들이다. status는 요청된 락에 관한 상태 정보로서 획득(granted), 대기(waiting), 그리고 변환(converting) 중의 한 값을 갖는다. 이들 중 변환은 해당 트랜잭션이 한 파티션에 대하여 공유 모드의 락을 획득한 후 다시 같은 파티션에 배제 모드의 락을 요청하였을 때, 해당 파티션이 다른 트랜잭션에 의하여 공유 모드의 락이 걸려 있는 경우에 발생한다. 이 경우에는 일반 대기와는 다른 의미로 대기하는 것이므로 별도의 변환 상태임을 기록하게 된다.

grantedMode는 락의 모드를 나타내며, 공유 모드 혹은 배제 모드 중 한 값을 갖는다. convertingMode는 status가 변환인 경우, 변환하고자 하는 모드의 값을 갖는다. transEntry는 이 락을 요청한 트랜잭션의 정보가 저장된 트랜잭션 테이블 내의 엔트리에 대한 포인터를 나타낸다. 끝으로 next와 prev는 이 락을 요청한 트랜잭션이 요청한 다른 락에 대한 포인터로서 그림 3.2에서 나타난 트랜잭션 락 리스트를 구성하기 위하여 사용된다. 이것은 트랜잭션의 종료시 이 트랜잭션의 수행을 위하여 요청하였던 락들을 모두 반환하기 위하여 필요하다.

TransEntryTable은 현재 수행중인 트랜잭션에 관

3) 현재, 개발된 동시성 제어 관리자에서 사용되는 락 헤더의 크기는 28 바이트이다. 보편적으로 사용하는 8K 바이트의 파티션을 고려하면, 락 헤더가 차지하는 비중은 0.3%에 불과하다.

www.dbpia.co.kr

한 각종 정보들이 저장되어 있는 트랜잭션 테이블을 의미한다. 따라서 트랜잭션이 새로 생성될 때마다 transEntryTable내의 빈 엔트리가 할당되며, 트랜잭션이 종료되면 이 엔트리는 반환된다. transEntryTable는 다양한 요소들로 구성되지만 동시성 제어와 직접 관련된 정보는 lockRequestList, latchWaitingList, lockWaitForList, semID, latchMode 등이다. lockRequestList은 이 트랜잭션이 요청한 락들이 연결된 리스트를 가리키는 포인터이다. latchWaitingList는 같은 래치를 잡기 위해 기다리는 트랜잭션들의 연결 리스트를 구성하기 위하여 사용되는 포인터로서 제 4장에서 자세히 설명한다. lockWaitForList는 락 획득을 위하여 이 트랜잭션이 대기 중인 다른 트랜잭션들의 리스트를 가리키는 포인터이며, 제 5장에서 설명하는 교착 상태 검출을 위하여 사용된다.

락 혹은 래치를 대기해야 하는 트랜잭션을 대기 상태(sleep state)로 만들거나 대기 상태의 트랜잭션을 준비 상태(ready state)로 만들기 위하여 본 개발에서는 유닉스의 세마포어를 사용하였다. semID는 각각의 트랜잭션을 위하여 할당된 세마포어의 식별자를 의미한다. latchMode는 해당 트랜잭션이 획득하기를 원하는 래치의 모드가 공유 모드인지 혹은 배제 모드인지를 표현한다.

Ⅳ. 래치 관리 기법

본 장에서는 래치 관리를 위하여 본 개발에서 사용한 기법을 소개한다. 먼저, 제 4.1절에서는 래치와 관련된 기본 개념으로서 래치를 정의하고, 락과 비교한 특성을 설명한다. 제 4.2절에서는 래치의 효과적인 관리를 위하여 고려해야 할 개발 목표와 그 의미를 제시한다. 제 4.3절에서는 본 연구에서 채택한 래치의 관리 기법을 제시하고, 장단점에 관하여 논의한다.

4.1. 기본 개념

래치(latch)란 다수의 트랜잭션들에 의하여 공유되는 시스템 데이터의 물리적인 일관성(physical consistency)을 보장해 주기 위한 수단이다[15][20]. 래치는 사용자 데이터의 논리적인 일관성(logical consistency)을 보장하기 위하여 사용되는 락과 그 특성이 유사하다. 공유 데이터를 액세스하고자 하는 트랜잭션은 반드시 래치를 획득해야 하며, 액세스를 끝낸 트랜잭션은 래치를 즉시 반환함으로써 이 래치의 획득을 원하며 대기 중이던 다른 트랜잭션이 진행될 수 있도록 해야 한다.

래치는 락과는 다른 다음과 같은 특성을 가진다[15][20]. (1) 이단계 락킹 규약을 사용하는 경우 한번 획득된 락은 트랜잭션의 종료 시점까지 유지되어야 한다. 반면, 래치는 트랜잭션이 공유 데이터를 액세스하는 짧은 기간 동안에 한하여 유지되며, 액세스가 끝난 후에는 즉시 반환된다. (2) 관리하는 비용 측면에서 래치는 락의 약 10% 정도로서 상대적으로 작은 비용으로 관리된다. (3) 래치는 대기 그래프를 기반으로 하는 교착 상태 검출기에 의하여 검출되지 않는다. 따라서 락과 래치 혹은 래치와 래치 간의 교착 상태의 발생 여부를 별도로 점검하여야 한다.

래치는 공유 데이터에 하나씩 할당된다. 예를 들어, 제 3.3절에서 언급한 lockHeader도 락을 획득하려는 다수의 트랜잭션들에 의하여 동시에 액세스될 수 있다. lockHeader가 가리키는 lockRequest들의 연결 리스트를 탐색하려는 트랜잭션과 이 연결 리스트 내에 새로운 lockRequest를 삽입하는 트랜잭션이 별도의 제어 작업 없이 동시에 수행되는 경우, 이 리스트는 물리적으로 일관되지 못한 상태로 될 수 있다. 모든 트랜잭션이 공유 데이터인 lockHeader를 액세스하기 직전 래치를 걸고, 액세스 직후에는 래치를 반환하도록 규정함으로써 이러한 문제를 방지할 수 있다. lockHeader내에 존재하는 latch 필드(제 3.3절 참조)가 이와 같이 래치를 걸기 위한 용도로서 사용된다.

4.2. 개발 목표

본 절에서는 개발하고자 하는 래치의 관리 기법의 구현 목표를 제시한다.

첫째, 공유 모드(shared mode)와 배제 모드(exclusive mode)를 모두 지원하는 것이다. 구현을 단순화하기 위하여 두 모드의 구별 없이 배제 모드와 같은 의미의 한 모드만을 제공하는 래치도 고려할 수 있다. 그러나 이 경우, 다수의 읽으려는 트랜잭션들의 동시 수행을 허용하지 않으므로 동시성이 저하되는 문제가 발생한다.

둘째, CPU 수행 시간의 낭비를 최소화하는 것이다. 한 트랜잭션 A가 다른 트랜잭션 B에 의하여 이미 획득된 래치를 대기하는 경우, B가 해당 래치를 반환하였는가의 여부를 파악하기 위하여 빈번하게 이를 점검한다면 심각한 CPU 시간의 낭비를 초래할 것이다. MMDBMS에서는 CPU 수행 시간이 성

www.dbpia.co.kr

능에 가장 큰 영향을 미치므로 이러한 CPU 수행 시간의 낭비를 방지해야 한다.

셋째, 래치 개수에 제한을 가하지 않는 것이다. 만일, 래치를 구현할 때 컴퓨터 하드웨어나 운영체제 등 기반 시스템에서 개수에 제한을 가하는 자원을 이용하는 경우, 사용 가능한 래치 개수의 제한을 초래하게 된다. 그러나 전술한 바와 같이, 래치는 각각의 공유 데이터마다 하나씩 할당되므로 공유 데이터가 많은 데이터베이스 환경에서는 할당되어야 하는 래치의 개수가 매우 많다. 따라서 이와 같이 래치의 개수를 제한하는 방법은 배제되어야 한다.

넷째, 래치의 관리 비용을 최소화하는 것이다. MMDBMS에서는 데이터 액세스 비용이 매우 작으므로 큰 래치 관리 비용은 전체 시스템의 성능을 저하시키는 효과를 가져온다. 따라서 전체 래치 관리 비용을 극소화시키는 전략이 필요하다.

4.3. 채택한 래치 기법

MMDBMS에서는 디스크 기반 DBMS와 비교하여 제 4.2절에서 제시한 설계 목표들과 관련된 래치의 효율성이 전체 시스템 성능에 미치는 영향이 매우 크다. 따라서 이러한 목표들을 달성할 수 있는 효과적인 래치 관리 기법의 개발이 요구된다. 본 절에서는 본 개발에서 채택한 기법에 관하여 기술한다4).

그림 4.1은 래치 관리를 위한 자료 구조를 나타낸다. 이것은 공유하고자 하는 각 데이터의 내부에 포함된다. 개발된 래치는 공유 모드와 배제 모드를 모두 지원한다. latchMode는 트랜잭션이 어떤 모드로 이 래치를 획득하였는가를 나타낸다. numHolders는 이 래치를 동시에 획득한 트랜잭션들의 수를 나타내며, 공유 모드를 제공하기 위하여 필요하다. waitingList는 현재 이 래치의 획득을 원하여 대기하고 있는 트랜잭션들의 리스트를 가리킨다. 리스트 내의 각 요소는 transEntryTable 내의 엔트리를 의미하며, 이 리스트는 엔트리내의 latchWaitingList 필드를 이용하여 구성된다.

트랜잭션이 래치를 새롭게 획득하고자 할 때, 원하는 모드가 latchMode와 호환되지 않는 경우에는

4) 참고 문헌[22]에서는 디스크 기반 DBMS인 BADA-II에서 채택한 래치의 구현 방안에 관하여 상세하게 다루고 있다. 본 연구에서는 이 논문을 통하여 얻은 주요 아이디어를 래치 개발에 활용하였음을 밝히고자 한다. 본 논문에서는 해결 방안와 동기와 원리에 대하여 기술의 초점을 맞추었다.

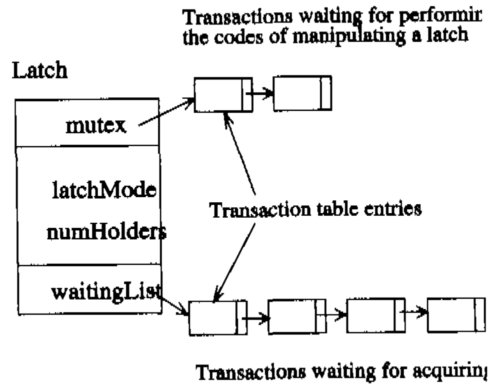

그림 4.1. 래치를 위한 자료 구조

해당되는 트랜잭션 엔트리를 이 래치의 waitingList 필드가 가리키는 리스트 내에 삽입한 후, Unix의 세마포어[3]를 사용하여 대기 상태가 되도록 한다. 트랜잭션이 래치를 반환함으로써 numHolders가 0이 되면, 다시 Unix의 세마포어를 사용하여 waitingList가 가리키는 첫 번째 트랜잭션을 대기 상태에서 깨워준다. 즉, 제안하는 기법에서는 Unix의 세마포어가 상호 배제 수단이 아니라 바쁜 대기(busy waiting)의 방지를 위한 동기화의 수단으로서 사용된다. 이와 같이, 래치를 획득하지 못하고 대기 중인 트랜잭션을 대기 상태에 놓음으로써 CPU 시간의 불필요한 낭비를 방지할 수 있다.

래치의 관리를 위하여 사용되는 latchMode, numHolders, waitingList 역시 다수의 트랜잭션이 동시에 액세스할 수 있는 공유 데이터이다. 이 공유 데이터는 액세스하는 트랜잭션에 의하여 반드시 변경된다. 따라서 이 공유 데이터를 액세스하는 코드를 임계 구역(critical section)으로 설정함으로써 특정 순간에는 오직 하나의 트랜잭션만이 이들을 액세스하도록 보장해야 한다. 이러한 임계 구역 문제를 해결하기 위한 방법으로서 다음의 두 가지 방법들의 적용이 가능하다.

먼저, 바쁜 대기를 이용하는 방법을 고려할 수 있다. 즉, 운영 체제의 전통적인 해결 방법인 Dijkstra 알고리즘[1] 혹은 Bakery 알고리즘[2] 등을 이용하는 것이다. 각 트랜잭션은 임계 구역 내에 들어가기 직전과 임계 구역에서 나온 직후 이 알고리즘을 수행한다. 이 방법은 정확성이 검증된 코드를 쉽게 만들 수 있으며, 이식성이 뛰어나다는 것이 장점이다.

그러나 다수의 트랜잭션들이 임계 구역에 들어가고자 하는 경우, 임계 구역 안으로 진입한 단 하나의 트랜잭션 외의 모든 트랜잭션들은 CPU 시간을

www.dbpia.co.kr

소모하는 바쁜 대기를 계속하게 된다. 이 결과, CPU 시간의 불필요한 낭비를 초래하게 된다. 이러한 CPU 시간의 낭비는 임계 구역의 크기가 커질수록 더욱 심각해진다. 또한, 이 방법은 제한된 대기 특성(bounded waiting)을 만족하지 못한다는 단점도 갖는다. 즉, 임계 구역 내에 진입하려는 트랜잭션들이 바쁜 대기 상태에서 독립적으로 진입을 시도하므로 그들 중 어떤 트랜잭션이 먼저 진입할 것인가는 운영 체제의 상태에 따라 매번 달라지게 된다. 따라서 개발한 기법에서는 이러한 문제점으로 인하여 바쁜 대기를 사용하지 않는다.

또 다른 방법은 운영 체제인 Unix가 상호 배제 기법으로서 제공하는 세마포어를 이용하는 것이다. 이 방법은 시스템 콜(system call)의 형태로 제공되는 기능을 쉽게 사용할 수 있으며, Unix가 세마포어를 획득하지 못한 트랜잭션을 대기 상태로 놓았다가 획득한 경우에 다시 수행 상태로 제어해 주므로 바쁜 대기로 인한 CPU 시간의 낭비를 방지할 수 있다.

그러나 Unix의 세마포어는 사용자 모드에서 커널 모드로의 문맥 교환(context switch)을 요구하므로 래치의 처리 비용이 커지며, 이 결과, 전체 시스템의 성능 저하를 초래할 수 있다. 또한, 각 래치마다 하나의 세마포어를 할당해야 하는데, Unix에서는 동시에 사용 가능한 세마포어의 개수를 제한하고 있어 결국 사용 가능한 래치의 개수가 제한된다. 따라서 개발한 기법에서는 이러한 문제를 고려하여 이 방법을 채택하지 않는다.

본 연구에서는 바쁜 대기 방법과 Unix 세마포어를 이용한 방법을 결합함으로써 임계 구역 문제를 효과적으로 해결하는 방법을 사용한다. 여기서 Unix의 세마포어는 상호 배제의 수단이 아닌 동기화의 수단으로서 사용된다.

그림 4.1에서 mutex는 래치 처리를 위한 임계 구역 문제를 해결하기 위하여 도입된 필드로서 현재 트랜잭션이 임계 구역내에 들어가 있는지의 여부를 나타내는 필드이다. 트랜잭션이 현재 들어가 있지 않는 경우, 이 값은 NULL이다. 임계 구역에 들어가고자 하는 트랜잭션은 이 mutex 필드를 참조하여 다른 트랜잭션이 임계 구역내에 들어가 있지 않으면, mutex 필드가 해당 트랜잭션의 엔트리를 가리키도록 하고 임계 구역으로 들어간다. 만일, 이미 mutex 필드가 다른 트랜잭션의 엔트리를 가리키고 있으면, 임계 구역으로 들어갈 수 없음을 의미하므로 mutex가 가리키는 리스트 내에 자신의 트랜잭션 엔트리를 삽입한 후, Unix가 제공하는 세마포어를 사용하여 대기 상태로 들어간다. 임계 구역에서 빠져 나온 트랜잭션은 자신의 트랜잭션 엔트리를 mutex가 가리키는 리스트에서 빼낸 후, Unix의 세마포어를 사용하여 리스트 내의 첫 번째 트랜잭션을 대기 상태에서 깨워준다. 이와 같이, 임계 구역으로의 진입을 원하는 트랜잭션을 대기 상태에 놓음으로써 CPU 시간의 불필요한 낭비를 방지할 수 있으며, 요구되는 세마포어의 개수도 래치의 대상이 되는 공유 데이터의 개수가 아니라 동시에 수행이 가능한 트랜잭션의 수만큼 만 요구된다.

그러나 새롭게 도입한 mutex가 가리키는 리스트도 공유 데이터이므로 이에 대한 상호 배제가 필요하다. 즉, 이 리스트를 관리하는 코드가 임계 구역으로서 보호되어야 한다는 것이다. 본 연구에서는 이를 위하여 Bakery 알고리즘을 이용한 바쁜 대기 방식을 채택하였다. mutex 필드와 관련된 임계 구역은 충분히 작은 부분이므로 바쁜 대기로 인한 CPU 시간의 낭비가 미미하다. 현재 C 언어로 개발된 코드에서 래치 관리를 위한 전체 코드의 크기는 255 라인이며, 이 임계 구역의 크기는 21 라인이다. 따라서 두 개 이상의 트랜잭션이 mutex가 가리키는 리스트상에서 대기하는 상황은 실제로 거의 발생하지 않는다.

제안하는 래치 관리 기법이 제 4.2절에서 제시한 개발 목표를 만족하는가를 점검해 보자. (1) 공유 모드와 배제 모드를 함께 지원한다. (2) 바쁜 대기를 이용하는 임계 구역은 단지 mutex가 가리키는 리스트의 조작 뿐이므로 이로 인한 CPU 시간의 낭비를 극소화할 수 있다. (3) 세마포어를 상호 배제의 수단이 아닌 동기화의 수단으로만 사용하므로 사용될 래치의 개수에 제한을 가하지 않는다. (4) 또한, 트랜잭션이 대기 상태로 들어가는 경우에 한하여 세마포어 연산이 수행되므로 문맥 교환으로 인한 래치 관리 비용의 증가를 막을 수 있다.

V. 교착 상태 검출 기법

본 장에서는 교착 상태의 검출과 관련된 이슈에 관하여 설명한다. 제 5.1절에서는 교착 상태 검출과 연관된 기본 개념을 설명하고, 제 5.2절에서는 교착 상태 검출 및 해결 방안에 관하여 간략히 설명한다. 제 5.3절에서는 교착 상태 검출 과정에서 동시성을 보다 향상시키기 위한 개선 전략에 관하여 논의한다.

www.dbpia.co.kr

5.1. 기본 개념

교착 상태(deadlock)란 두 개 이상의 트랜잭션들이 락을 획득한 상태에서 서로 다른 트랜잭션이 획득한 락이 반환되기를 무한정 기다리는 현상이다[6][21][13]. 교착 상태는 이단계 라킹 규약을 이용한 동시성 제어 기법에서 흔히 발생할 수 있으며, 시스템의 성능을 급격히 떨어뜨리는 심각한 문제를 야기 시킬 수 있다. 따라서 DBMS는 트랜잭션들이 교착 상태에 빠져 있는트랜잭션들을 찾아냄으로써 교착 상태 문제를 해결해야 한다. 교착 상태 검출기(deadlock detector)는 트랜잭션들이 교착 상태에 빠져 있는가를 주기적으로 검출해 준다.

트랜잭션 Ti가 모드 modei로 획득하고자 하는 락 L을 다른 트랜잭션 Tj가 이미 모드 modei와 충돌(conflict)되는 모드 modej로 획득한 경우, 트랜잭션 Ti는 더 이상 수행을 계속하지 않고 트랜잭션 Tj가 락 L을 반환할 것을 대기하게 된다. 이단계 라킹 규약에서 모든 트랜잭션들은 획득하였던 락들을 종료 시점까지 반환하지 않으므로 트랜잭션 Ti가 락 L을 대기한다는 것은 이미 이를 획득하고 있는 트랜잭션 Tj의 종료를 대기한다는 의미가 된다. 락 대기 정보(lock waiting information)는 락과 연관하여 각각의 트랜잭션이 어떤 트랜잭션의 종료를 대기하고 있는가를 표현하는 정보이므로 교착 상태를 검출하기 위한 중요한 정보가 된다.

락 대기 정보는 제 3.3절에서 설명한 transEntry-Table에서 관리된다. 전술한 바와 같이 lockWait-ForList는 각 트랜잭션이 대기하는 다른 트랜잭션의 transID를 가지는 요소들의 리스트를 가리키는 포인터를 의미한다. 리스트 내에 여러 개의 요소가 존재하는 이유는 한 트랜잭션이 배제 모드로 락을 요청한 경우, 이미 여러 개의 트랜잭션이 공유 모드로 해당 락을 획득한 경우가 발생 가능하기 때문이다. 예를 들어, 트랜잭션 T1이 락 L1을 배제 모드로 획득하려고 할 때, 이미 트랜잭션 T4, T6, T8이 이 락 L1을 공유 모드로 획득한 상태이면, T1과 대응되는 트랜잭션 엔트리의 lockWaitForList는 세 개의 요소 T4, T6, T8로 구성되는 리스트를 가리키게 된다.

일반적으로 이러한 락 대기 정보 요소들은 미 사용중인 요소들의 집합으로 구성되는 풀 내에서 동적으로 관리된다. 트랜잭션이 락을 대기하게 되면 이 풀로부터 할당되어 lockWaitForList는 리스트에서 사용되고, 대기하던 트랜잭션이 락을 획득하게 되면 리스트에서 제거되어 풀로 반환된다. 이와 같이, 락 대기 정보는 락 획득을 요청한 트랜잭션이 대기하는 경우와 대기하던 트랜잭션이 해당 락을 획득하는 경우 변경된다.

5.2. 교착 상태의 검출 및 해결

교착 상태의 검출을 위하여 가장 널리 사용되는 것은 대기 그래프(wait-for-graph)를 이용한 방식이다[6][13]. 대기 그래프 방식은 먼저 lockWaitForList에서 관리하는 락 대기 정보를 기반으로 트랜잭션 상호 대기 관계를 표현하는 대기 그래프를 구성한 후, 대기 그래프 상에 싸이클(cycle)이 존재하는가의 여부를 점검함으로써 교착 상태를 검출한다. 즉, 대기 그래프 상에 싸이클이 존재하면, 그 싸이클을 형성하는 트랜잭션들 간의 교착 상태가 존재하는 것이다. 본 연구에서도 참고 문헌 [13]의 방식을 기반으로 교착 상태 검출기를 개발하였다.

교착 상태 검출기는 DBMS내에서 주기적으로 수행되며, 그 검출 주기는 시스템의 부하에 따라 적절하게 조정된다. 교착 상태가 검출되면 여기에 참여하는 트랜잭션들 중 희생자(victim)를 선정하여 철회시킴으로써 교착 상태를 해결한다[5]). 또한, 교착 상태를 검출하여 해결한 직후 교착 상태 검출기를 재차 수행시킨다. 이것은 시스템 내에 두 개 이상의 교착 상태가 발생한 경우와 한 희생자 철회에도 시스템 내에 여전히 교착 상태가 남아있는 경우를 대비하기 위한 것이다. 희생자 트랜잭션을 선정하는 기준으로 우선 순위, 수행된 시간, 수행할 시간, 획득한 락의 수, 획득할 락의 수, 기록된 로그 레코드(log record)의 수 등을 사용할 수 있으나, 실시간 응용의 지원을 감안하여 마감 시간에 기반한 트랜잭션의 우선 순위를 우선적으로 고려하고 있다.

5.3. 동시성 개선 방안

트랜잭션은 락을 대기하는 경우와 대기 후 락을 획득하는 경우, 락 대기 정보 요소를 갱신하게 되고, 주기적으로 수행되는 교착 상태 검출기는 이 락 대기 정보를 참조하여 교착 상태 여부를 결정하게 된다. 따라서 이 락 대기 정보는 병행 수행되는 교착 상태 검출기와 트랜잭션들이 동시에 읽고 쓰는

5) 현재 개발중인 MMDBMS는 아직 회복 관리자를 포함하고 있지 않다. 따라서 이러한 철회 부분은 아직 구현되어 있지 않은 상태이다. 그러나 이것은 회복 관리자가 기록한 로그 레코드들을 역순으로 액세스함으로써 쉽게 처리할 수 있다.

www.dbpia.co.kr

공유 데이터이다. 이러한 공유 데이터를 별도의 제어 없이 다수의 프로세스들의 액세스를 허용한다면, 잘못된 수행이 발생 가능하다. 따라서 공유 데이터인 락 대기 정보의 물리적인 일관성을 보장하기 위한 상호 배제 기능이 필요하다.

락 대기 정보의 상호 배제를 위한 가장 단순한 방법의 하나는 락 대기 정보에 하나의 래치를 할당하여 놓은 후, 락 대기 정보를 액세스할 때마다 래치[15][20]를 획득하도록 하는 것이다. 락 대기 정보가 액세스되는 경우는 (1) 교착 상태 검출기가 대기 그래프를 형성하기 위하여 전체 락 대기 정보를 읽는 경우, (2) 트랜잭션이 다른 트랜잭션의 종료를 대기하게 되어 자신의 락 대기 정보를 갱신하는 경우, (3) 트랜잭션의 종료로 인한 락 반환의 결과, 이 트랜잭션을 기다리던 다른 트랜잭션의 락 대기 정보를 갱신하게 되는 경우 등이 있다. 따라서 이러한 세 가지 종류의 작업 수행되기 직전 래치를 획득하고, 작업이 완료된 후 래치를 반환으로써 물리적인 일관성을 보장할 수 있다.

그러나 이와 같이 단순한 방식은 다음과 같은 두 가지 문제점을 유발시킨다.

첫째, 래치 처리를 위한 오버헤드의 문제이다. 각 트랜잭션은 락을 대기하거나 대기 후 락을 획득할 때마다 락 대기 정보의 갱신을 요구하므로 모든 트랜잭션은 이러한 경우 매번 래치를 획득해야 한다. 이러한 락 대기 및 락 대기 후 획득은 하나의 트랜잭션의 수행에서도 매우 빈번하게 발생하게 된다. 래치를 처리하는 비용은 락을 처리하는 비용과 비교하여 작으나, 이와 같은 빈번한 래치의 처리는 전체 트랜잭션의 수행 성능을 저하시킨다. 특히, 본 연구 개발에서 대상으로 하는 MMDBMS에서는 데이터 액세스가 매우 빠른 속도로 처리되므로 이러한 래치 처리를 위한 오버헤드는 시스템 전체 성능에 더욱 큰 영향을 미친다.

둘째, 시스템 동시성 저하의 문제이다. 교착 상태 검출기는 래치를 획득한 상태에서 락 대기 정보를 기반으로 교착 상태를 검출하게 되므로, 교착 상태 검출기가 수행되는 동안 다른 트랜잭션들은 락 대기 정보를 전혀 액세스할 수 없게 된다. 최악의 경우, 교착 상태 검출기가 수행될 때, 전체 트랜잭션들이 모두 수행을 정지할 가능성도 존재한다. 따라서 시스템 동시성이 크게 저하되며, 이 결과 트랜잭션 응답 시간이 떨어진다. 특히, MMDBMS와 같은 고성능 시스템에서 이러한 동시성의 저하 문제는 전체 시스템 성능 저하의 직접적인 원인이 된다.

제안하는 기법의 기본적인 아이디어는 다음과 같다. 락 대기 정보 요소 풀은 모든 트랜잭션들이 공유하는 정보이므로 이의 상호 배제를 위하여 이미 래치가 존재한다. 즉, 각 트랜잭션이 자신의 락 대기 정보에 새로운 요소를 넣거나 존재하던 요소를 풀에 반환하는 경우에는 반드시 이 래치를 획득해야만 한다. 제안하는 기법에서는 락 대기 정보 요소 풀에 할당되어 있는 이 래치를 락 대기 정보를 위한 상호 배제 수단으로서 사용한다. 이것은 락 대기 정보를 위한 상호 배제를 목적으로 하는 별도의 래치 처리 오버헤드를 유발시키지 않도록 하기 위한 것이다.

트랜잭션이 락 대기 정보를 갱신하고자 하는 경우에는 이미 락 대기 정보 요소를 할당받기 위하여 풀에 래치를 걸고 있을 것이므로 락 대기 정보의 상호 배제를 위한 별도의 래치 처리 작업은 불필요하다. 한편, 교착 상태 검출기가 락 대기 정보를 참조하기 위해서는 풀에 할당된 이 래치를 획득해야 한다. 그러나 교착 상태 검출기의 수행 횟수는 트랜잭션 수행 횟수와 비교하여 매우 적으므로 이러한 오버헤드는 상대적으로 미미하다. 교착 상태 검출기가 이 래치를 사용할 수 있도록 하기 위하여 교착 상태 검출기와 대응되는 엔트리를 트랜잭션 테이블 내에 할당한다.

트랜잭션 테이블의 각 엔트리가 가리키는 락 대기 정보 리스트에 정해진 numStaticElements 만큼의 락 대기 정보 요소를 시스템 초기화 시에 미리 할당시킨다. 이는 트랜잭션이 래치를 획득하는 수를 줄임으로써 동시성을 극대화하기 위한 것이다. 즉, 트랜잭션이 다른 트랜잭션들의 종료를 대기하게 될 때, 자신이 대기하는 트랜잭션들의 수가 numStaticElements 미만인 경우에는 풀로부터 락 대기 정보 요소를 할당받을 필요가 없으며, 자신이 대기하는 다른 트랜잭션의 수가 numStaticElements를 초과하는 경우에 한하여 락 대기 정보 요소를 추가적으로 할당받게 된다. 따라서 이러한 락 대기 정보 요소의 사전 할당 방식을 사용함으로써 교착 상태 검출기와 트랜잭션들간의 래치 경쟁을 극소화 할 수 있으며, 이 결과 동시성은 극대화된다6).

6) 물론, 교착 상태 검출기와 트랜잭션들이 미리 할당되어 있는 락 대기 정보 요소 값을 함께 액세스하는 경우가 발생한다. 그러나 락 정보 요소 리스트의 형태 변화 없이 원자적인 요소 값만을 읽고 쓰는 이러한 경우에는 상호 배제가 반드시 보장되어야 하는 것은 아니다.

www.dbpia.co.kr

Ⅵ. 동시성 제어에 관련된 기타 이슈

본 절에서는 제안된 동시성 기법을 실제 시스템 내에 통합하는 경우에 발생하는 기타 구현 이슈에 관하여 설명한다. 제 6.1절에서는 인덱스와 시스템 카탈로그의 동시성 제어에 관하여 논의하고, 제 6.2절에서는 트랜잭션 테이블에 관한 상호 배제 방안에 관하여 기술한다. 제 6.3절에서는 실시간 응용의 효과적인 지원 방안에 관하여 논의한다.

6.1. 인덱스와 시스템 카탈로그를 위한 동시성 제어

인덱스(index)란 세그먼트 내의 레코드들을 빠르게 액세스하기 위한 부가적인 정보이다. MMDBMS에서는 T 트리(T-tree)와 해싱이 널리 사용된다[11]. 시스템 카탈로그(system catalog)란 세그먼트와 인덱스에 관한 부가적인 정보를 유지하는 다수의 시스템 고유 세그먼트들이다. 여기에서 관리되는 정보로는 세그먼트 식별자, 세그먼트 내의 레코드 수, 세그먼트 내의 레코드가 가질 수 있는 최대 크기, 세그먼트 내의 레코드가 갖는 애트리뷰트의 수, 세그먼트 내의 파티션의 수, 세그먼트의 첫 파티션의 주소, 세그먼트에 속하는 인덱스의 수, 인덱스의 식별자, 인덱스가 구성된 세그먼트 및 속성, 인덱스의 시작 주소 등이 있다[DeW85].

인덱스와 시스템 카탈로그도 다수의 트랜잭션들이 동시에 액세스할 수 있는 공유 데이터에 해당되므로 이에 대한 동시성 제어가 요구된다. 만일, 이들에 대하여 일반 사용자 데이터와 같이 해당 파티션을 단위로 하는 이단계 라킹 기법을 적용하는 경우, 매우 심각한 동시성 저하 문제가 발생한다. 인덱스나 시스템 카탈로그는 모든 트랜잭션들이 액세스하는 데이터이므로 트랜잭션 종료시까지 락을 계속 잡고 있는 경우, 트랜잭션들을 순차적으로 수행하는 결과를 초래하기 때문이다.

따라서 본 연구에서는 이를 해결하기 위하여 각각의 인덱스 및 시스템 카탈로그마다 고유의 래치를 둠으로써 동시성을 제어하고자 한다. 각 트랜잭션은 인덱스 및 시스템 카탈로그를 액세스하기 직전 래치를 걸고, 이에 대한 액세스가 끝나면 이 래치를 반환한다. 인덱스와 시스템 카탈로그에 대하여 래치 만으로 동시성 제어가 가능한 이유는 이들은 사용자 데이터의 관리를 위하여 필요한 부가적인 메타 데이터(meta data)이므로 사용자가 그 논리적인 내용을 알지 못하기 때문이다. 이와 같은 메타 데이터에 대해서는 단순히 물리적인 일관성만 보장하면 되므로 래치의 적용이 가능하다. 이 결과, 인덱스 및 시스템 카탈로그의 물리적 일관성이 보장되며, 짧은 시간 동안만 래치를 걸게 되므로 동시성을 극대화 시킬 수 있다.

6.2. 트랜잭션 테이블의 관리

트랜잭션이 수행될 때 트랜잭션 테이블 transEntryTable 내의 자신의 엔트리를 빈번하게 참조한다. 해당 엔트리를 참조할 때마다 transEntryTable를 탐색한다면, CPU 수행 시간이 낭비될 것이다. 이를 해결하기 위하여 각 트랜잭션의 수행이 시작될 때 transEntryTable 내에 자신의 엔트리를 확보한 후 이 엔트리에 대한 포인터를 유지함으로써 이후의 엔트리 참조시에는 transEntryTable의 탐색 없이 바로 자신의 엔트리를 참조할 수 있도록 하였다.

한 트랜잭션이 다른 트랜잭션의 엔트리를 참조하는 경우는 발생하지 않으나, 트랜잭션이 시작될 때에는 transEntryTable를 탐색함으로써 빈 엔트리 공간을 찾아 해당 엔트리를 등록해야 한다. transEntryTable도 다수의 트랜잭션이 함께 액세스하는 공유 데이터이므로 이에 대한 상호 배제가 필요하다.

다른 공유 데이터와 마찬가지로 래치를 이용한 transEntryTable의 상호 배제도 고려할 수 있다. 그러나 래치는 해당 트랜잭션이 transEntryTable에 등록된 후에만 사용이 가능하다. 이것은 래치 관리를 위한 latchWaitList 등의 정보가 transEntryTable 내에 존재하기 때문이다. 따라서 래치를 이용한 transEntryTable의 상호 배제는 실효성이 없다.

본 연구 개발에서는 Unix의 세마포어를 이용한 상호 배제를 채택하였다. 즉, 트랜잭션 테이블을 위한 고유의 Unix 세마포어를 하나 할당해 놓고, 트랜잭션의 수행 시작 시점에 빈 엔트리를 찾기 위하여 트랜잭션 테이블을 탐색하기 직전에 이 세마포어를 건다. 탐색을 통하여 빈 엔트리를 찾으면, 해당 트랜잭션에 대한 정보를 이 엔트리에 넣고 세마포어를 풀어준다. 트랜잭션이 종료할 때에도 이와 동일한 작업을 수행한다. 물론, 전술한 바와 같이 Unix 세마포어가 큰 오버헤드를 가지는 연산이나, transEntryTable에 대한 상호 배제는 트랜잭션 수행시에 빈번하게 발생하는 것이 아니고, 시작과 종료 시점에만 한번씩 요구되므로 그 영향은 미미하다.

6.3. 실시간 응용의 지원

MMDBMS의 주요 응용 분야의 하나는 실시간 시스템이다. 실시간 시스템이란 마감 시간을 가지는

www.dbpia.co.kr

실시간 태스크7)를 처리하기 위한 시스템으로 정의된다[14][17]8). 개발 중인 MMDBMS가 실시간 시스템을 지원하기 위해서는 마감 시간 개념을 수용해야 한다.

마감 시간과 관련된 가장 핵심적인 기능은 실시간 태스크가 마감 시간내에 완료될 수 있도록 수행 순서를 조정하는 실시간 스케쥴링(real-time scheduling)[14]이다. Unix를 포함한 대부분의 범용 운영 체제에서는 마감 시간 개념을 가지고 있지 않으므로 실시간 스케쥴링 기능은 지원하지 않는다.

본 연구실에서는 범용 운영 체제인 Unix상에서 마감 시간을 고려하는 실시간 스케쥴링을 지원하는 기법을 제안한 바 있다[23]. 제안된 기법은 시스템내에서 수행 준비 상태(ready-to-run state)의 태스크만을 스케쥴링의 대상으로 하는 Unix의 특성[3]을 활용함으로써 스케쥴링 데몬(sheduling daemon)이라는 특수한 태스크가 Unix상에서 수행되는 실시간 태스크들 중 마감 시간이 가장 임박한 하나만을 수행 준비 상태로 만들고, 그 외의 다른 실시간 태스크들은 모두 대기 상태로 만드는 방식을 사용한다. 이 결과, Unix는 항상 스케쥴링 데몬이 수행 준비 상태로 만든 유일한 태스크만을 스케쥴링 대상으로 하게 된다. 따라서 실질적으로 시분할 방식의 Unix 스케쥴링 전략을 바이패스할 수 있으며, 이 결과 스케쥴링 데몬의 전략에 의하여 모든 실시간 태스크들을 스케쥴링 할 수 있다. 개발 중인 Tachyon은 실시간 응용 분야의 지원을 위하여 이러한 실시간 스케쥴링 데몬 방식을 이용하여 트랜잭션 스케쥴링을 수행하므로 실시간 요구 조건을 만족시킬 수 있다.

마감 시간이 반영되어야 하는 두 번째 기능은 대기 리스트에서 우선 순위를 조정하는 것이다. 제 2장과 제 3장에서 기술한 바와 같이, 트랜잭션이 락 혹은 래치의 획득을 요청하였을 때, 다른 트랜잭션이 이미 해당 락 혹은 래치를 획득한 상태인 경우에는 리스트 내에서 대기하게 된다. 본 개발에서는 마감 시간을 기준으로 하는 우선 순위 큐(priority-based queue)의 형태로 대기 리스트를 구현하였다. 즉, 기존의 대기 리스트내에 새로운 트랜잭션이 추가로 대기하는 경우, 새로운 트랜잭션의 마감 시간을 고려하여 대기 리스트내의 적절한 위치에 삽입하도록 함으로써 실시간 요구 조건을 만족시키도록 하였다.

마감 시간이 반영되어야 하는 세 번째 기능은 우선 순위 역전을 해결하는 것이다. 우선 순위 역전(priority inversion)[16]이란 트랜잭션이 락 혹은 래치를 획득한 낮은 우선 순위의 트랜잭션을 이 락 혹은 래치를 획득하고자 하는 높은 우선 순위의 트랜잭션이 대기하게 됨으로써 실시간 스케쥴링의 효과를 저해하는 현상이다9). 본 개발에서는 우선 순위 상속을 이용하여 이러한 문제를 해결하였다. 우선 순위 상속(priority inheritance)[16]이란 우선 순위 역전이 발생하는 경우, 낮은 우선 순위의 트랜잭션이 대기하게 되는 트랜잭션의 높은 우선 순위를 상속받도록 하는 방법이다. 이 외에도 낮은 우선 순위의 트랜잭션을 철회한 후, 높은 우선 순위의 트랜잭션을 먼저 처리해 주는 방법(high priority)과 조건부 재실행(conditional restart)[17]도 적용할 수는 있으나, 데이터베이스 환경에서는 트랜잭션 철회 및 재수행의 비용이 큰 비중을 차지한다는 점을 고려하여 이들은 채택 대상에서 제외하였다.

Ⅶ. 결 론

주요 저장 매체로서 주기억장치를 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 MMDBMS Tachyon을 위한 동시성 제어 관리자의 개발에 관하여 논의하였다. 동시성 제어 관리자는 다수의 트랜잭션들에 의하여 동시에 액세스되는 데이터베이스의 일관성을 보장해 주는 DBMS의 서브 컴포넌트이다. MMDBMS는 주기억장치 액세스만으로 데이터 검색 및 갱신을 수행하므로 디스크 기반 DBMS와는 달리 데이터 검색 및 갱신 비용 중 동시성 제어 관리자의 수행 비용이 차지하는 비중은 매우 크다. 따라서 효율적인 동시성 제어 관리자의 개발은 MMDBMS의 전체 성능에 큰 영향을 미치게 된다.

---

7) 트랜잭션은 DBMS가 관리하는 작업의 단위이며, 태스크는 운영 체제가 관리하는 작업의 단위이다.

8) 실시간 시스템은 하드 실시간 시스템과 소프트 실시간 시스템으로 분류된다. 본 연구에서는 이들 중 소프트 실시간 시스템을 연구 대상으로 한다.

9) 태스크 혹은 트랜잭션의 우선 순위는 다양한 기준을 적용하여 할당할 수 있으나, 실시간 시스템에서는 마감 시간의 임박 정도를 우선 순위 할당의 기준으로 사용한다.

www.dbpia.co.kr

본 연구에서 개발된 동시성 제어 관리자의 주요 특징은 다음과 같다.

첫째, 락의 단위를 주기억장치의 물리적인 할당 단위인 파티션으로 설정하였다. 이 결과, 지나친 동시성 저하나 락 관리 비용의 증가를 방지할 수 있다. 또한, 파티션의 물리적인 크기를 자유롭게 설정할 수 있으므로 응용 분야의 특성 분석을 통하여 동시성과 락 관리 비용을 유연하게 조정할 수 있다.

둘째, 디스크 기반 DBMS에서 락 관리를 위하여 널리 사용하던 해쉬을 사용하지 않고, 파티션 내부에 락에 관한 정보를 직접 관리하는 방법을 사용하였다. 이 결과, 락 관리를 위한 CPU 수행 비용을 크게 줄이는 효과를 가져온다.

셋째, 시스템 데이터의 물리적 일관성 유지를 위한 수단으로서 래치 관리 기법을 개발하였다. 개발된 래치 관리 기법은 공유 모드와 배제 모드를 모두 지원하며, Bakery 알고리즘과 Unix의 세마포어 기능을 결합함으로써 CPU의 이용률을 극대화하였다.

넷째, 교착 상태 해결을 위하여 교착 상태 검출기를 개발하였다. 개발된 교착 상태 검출기는 락 대기 정보를 기반으로 대기 그래프를 구성함으로써 시스템이 교착 상태에 빠져 있는가를 주기적으로 검출한다. 또한, 락 대기 정보의 효과적인 상호 배제 기법을 제안함으로써 교착 상태 검출기와 트랜잭션들 간의 상호 간섭을 최소화하였다.

이 외에도, 인덱스 혹은 시스템 카탈로그의 상호 배제, 트랜잭션 테이블의 상호 배제, 실시간 응용의 지원 등 실제 구현에서 발생하는 중요한 이슈들에 대해서도 다루었다. 개발된 동시성 제어 관리자는 현재 ETRI 실시간 DBMS 팀에서 개발하고 있는 차세대 실시간 MMDBMS의 일부로서 사용 중에 있다. 향후 연구 방향으로는 개발된 동시성 제어 관리자에서 채택한 각각의 세부 기법들이 디스크 기반 DBMS에서 채택되었던 기존의 기법들과 비교하여 얼마만큼의 성능 개선 효과를 가지는가를 분석적 및 실험적으로 검증하는 것을 고려하고 있다.

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 지원사업에 의한 결과입니다.

참 고 문 헌

[1] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Comm. of the ACM*, Vol. 8, No. 9, p. 569, 1965.

[2] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," *Comm. of the ACM*, Vol. 17, No. 8, pp. 453-455, 1974.

[3] M. J. Bach, *The Design of the Unix Operating System*, Prentice-Hall, 1980.

[4] P. Bernstein and N. Goodman, "Timestamp-Based Algorithms for Concurrency Control in Distributed Systems," *In Proc. Intl. Conf. on Very Large Data Bases*, pp. 285-300, VLDB, 1980.

[5] H. Kung and J. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems*, Vol. 6, No. 2, pp. 213-226, 1981.

[6] R. Agrawal, M. Carey, and D. DeWitt, "Deadlock Detection is Cheap," *ACM SIGMOD Record*, Vol. 13, No. 2, pp. 19-34, 1983.

[7] M. Carey and M. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems," *In Proc. Intl. Conf. on Very Large Data Bases*, pp. 107-118, VLDB, 1984.

[8] 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.

[9] C. Papadimitriou and P. Kanellakis, "On Concurrency Control by Multiple Versions," *ACM Trans. on Database Systems*, Vol. 9, No. 1, pp. 89-99, 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*, pp. 294-303, Aug. 1986.

www.dbpia.co.kr

[12] H. Garchia-Molina and K. Salem, "High Performance Transaction Processing with Memory Resident Data," *In Proc. Intl. Workshop on High Performance Transaction Systems*, Dec. 1987.

[13] B. Jiang, "Deadlock Detection is Really Cheap," *ACM SIGMOD Record*, Vol. 17, No. 2, pp. 2-13, 1988.

[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] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. on Computers*, Vol. 39, No. 9, pp. 1175-1185, 1990.

[17] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *ACM Trans. on Database Systems*, Vol. 17, No. 3, pp. 513-560, Sept. 1992.

[18] 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.

[19] T. J. Lehman et al., "An Evaluation of Starburst's Memory Resident Storage Component," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 4, No. 6, pp.. 555-566, Dec. 1992.

[20] 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.

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

[22] Y. C. Park and K. H. Kim, "The Design and Implementation of Exclusive Latches and Shared Latches in BADA-II DBMS," *Database Review*, Vol. 11, No.2, pp.111-133, 1995.

[23] 김 상욱, 김 대웅, 김 진호, 이 승선, 최 완, "Unix 환경에서 실시간 스케쥴링을 지원하기 위한 새로운 접근 방안," 한국정보과학회 논문지(B), Vol. 26, No. 2, pp. 176-188, 1999년 2월.

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 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 데이터베이스, 트랜잭션 관리, 데이터 마이닝, 멀티미디어 정보 검색, 공간 데이터베이스/GIS

장 연 정(Yeon-Jeong Jang)

2000년 2월 : 강원대학교 정보통신공학과 졸업(학사)
<주관심 분야> DBMS, 실시간 주기억장치 데이터베이스, 데이터 마이닝

김 윤 호(Yun-Ho Kim)

1999년 2월 : 강원대학교 정보통신공학과 졸업(학사)
1999년 3월~현재 : 강원대학교 대학원 컴퓨터정보통신학과 석사과정
<주관심 분야> DBMS, 실시간 주기억장치 데이타베이스, 데이터 웨어하우징, 데이터 마이닝

김 진 호(Jin-Ho Kim)

1982년 2월 : 경북대학교 전자공학과(전산전공) 졸업(학사)
1985년 2월 : 한국과학기술원 전산학과 졸업(석사)
1990년 2월 : 한국과학기술원 전산학과 졸업(박사)
1990년 8월~현재 : 강원대학교 전자계산학과 부교수
<주관심 분야> 주기억장치 실시간 데이터베이스, 실시간 스케쥴링, 트랜잭션 관리, 데이터 웨어하우징, 이력 데이터베이스

www.dbpia.co.kr

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 2월 : 경북대학교 전자공학과(전산전공) 졸업(학사)
1983년 2월 : 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

www.dbpia.co.kr