Database System 2020-2

Final Report

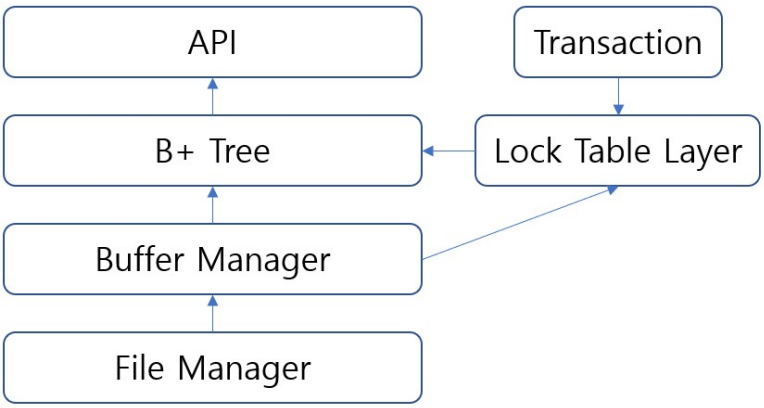
ITE2038-11801

2016027638

함재형

Table of Contents

Overall Layered Architecture



,

전체적인 DBMS의 구조는 위와 같습니다. Disk -> File Manager -> Buffer Manager -> B+ Tree -> API계층 순서로 상호작용을 합니다.

우선 Disk에 직접 접근하여 전반적인 DBMS의 페이지들을 관리하는 File Manager Layer가 가장 하위 계층입니다. 하나의 페이지는 1024byte 크기의 할당된 공간으로, File Manager Layer는 Disk에서 새로운 파일을 만들거나 읽고, Buffer Manager에서 페이지를 Disk에 write또는 Disk에서 read하라고 할 경우 이것을 담당하고, 파일에 필요한 페이지들이 부족할 경우 Free Page들을 만들어서 상위 계층에 전달하기도 합니다.

File Manager의 상위 계층인 Buffer Manager는 읽은 페이지들을 메모리 버퍼에 일시적으로 저장하여 접근속도를 향상시키는 cache 역할을 합니다. 제공하는 API 중 init_db()에 버퍼의 개수를 전달하면 그 숫자만큼의 버퍼를 생성할 수 있습니다.

Buffer Manager의 상위 계층인 B+ Tree는 B+ Tree 구조로 페이지를 저장하고 관리합니다. 한 개의 leaf_page에 들어가는 레코드는 31개이며, 한 개의 internal page에 들어가는 페이지(노드)는 249개입니다. API에서 db_find(), db_insert(), db_delete(), db_update()의 입력을 받으면 B+ Tree 구조를 기반으로 주어진 데이터를 찾고 삽입하고 지우고 업데이트 합니다.

B+ Tree의 상위 계층인 API는 유저가 직접 사용할 수 있는 함수들을 제공합니다. 제공하는 함수의 리스트는 다음과 같습니다.

int open_table(char * pathname):

pathname을 인자로 받아 파일의 unique한 table_id를 리턴

int init_db(int num_buf):

버퍼의 개수를 인자로 받아 버퍼를 생성 후 DBMS 동작에 필요한 것들을 초기화

int trx_begin():

트랜잭션 테이블에서 새로운 unique_id를 할당하여 리턴

int trx_commit(int trx_id):

트랜잭션을 종료하고 0을 리턴

int db_find(int table_id, int64_t key, char * ret_val, int trx_id):

테이블 id와 key, 트랜잭션 id와 문자열을 받아 DBMS에서 해당 레코드를 찾고, 찾았으면 문자열에 담고 0을 리턴, 실패시 -1 리턴

int db_insert(int table_id, int64_t key, char * value, int trx_id):

테이블 id와 key, 문자열레코드, 트랜잭션 id를 받아 DBMS에 write한 뒤 0을 리턴

int db_delete(int table_id, int64_t key):

테이블 id와 key를 받아 DBMS에서 찾아서 지운다. 못찾으면 -1을 리턴. 멀티 쓰레드는 지원하지 않습니다.

int db_update(int table_id, int64_t key, char * values, int trx_id):

테이블 id와 key, 트랜잭션 id, 문자열 레코드를 받아 새로운 문자열로 업데이트

int close_table(int table_id):

table_id를 인자로 받아 해당 테이블을 닫고 0을 리턴

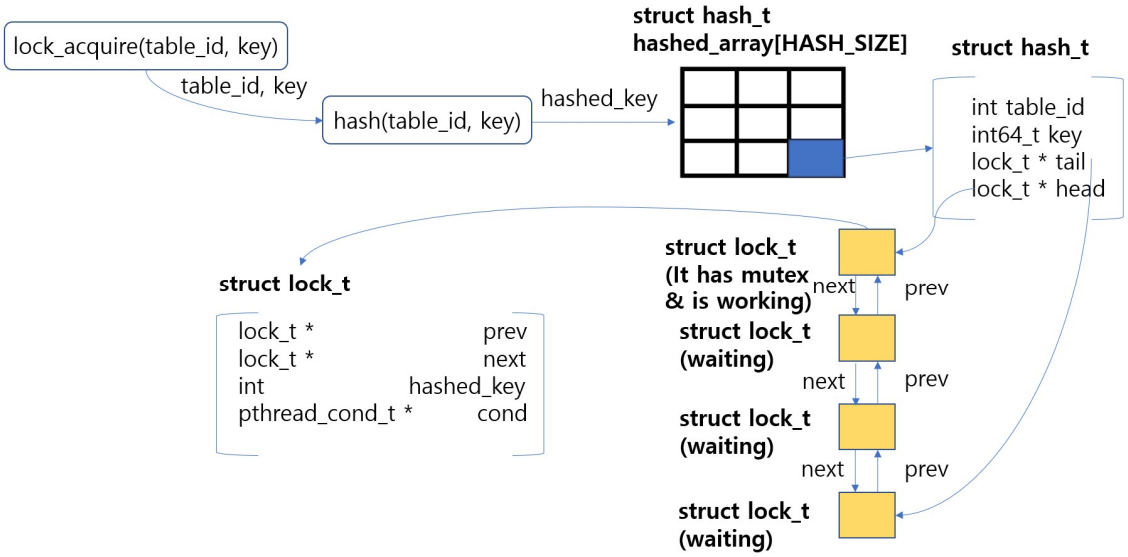
int shutdown_db():

DBMS의 버퍼 메모리를 free하고 0을 리턴

Concurrency Control Implementation

멀티 쓰레드 환경에서는 B+ Tree Layer와 Buffer Manager Layer 사이에 Lock Table Layer가 관여하여 Critical Section을 보호하는 역할을 합니다. 페이지를 찾을 때 항상 Lock Table이 관여합니다.

Lock Table의 구조는 다음과 같습니다.



B+ Tree Layer에서 페이지에 접근하기 전에 항상 lock을 얻어야 접근할 수 있습니다. B+ Tree Layer에서 특정 레코드에 접근하려고 할 때, 그것을 찾기 위해 접근해야 하는 모든 페이지들의 접근권한이 필요하게 됩니다. 따라서 그 페이지들의 접근 권한을 얻기 위해 페이지 lock을 요청할 경우 해시함수를 계산하여 해시 테이블의 해당 값에 lock object를 만들어 list로 줄 세웁니다. Lock object들은 자신에게 접근권한이 올 때까지 줄지어 기다리게 됩니다.

Crash-Recovery Implementation

Crash-Recovery는 구현되어 있지 않습니다.

In-depth Analysis

1. Workload with many concurrent non-conflicting read-only transactions.

많은 수의 non-conflicting read-only transaction이 동시에 수행될 경우에는 데드락이나 conflict문제는 없지만 read-only에 필요 없는 lock을 acquire/release하는 오버헤드가 발생하게 되므로 read-only에 성능을 최적화하기 위해서는 lock이 구현된 부분을 삭제해주면 좋을 것입니다.

또한 본 DBMS에서는 해시테이블에서 감당할 수 있는 테이블의 최대 개수가 100개이므로 100개보다 많은 테이블을 열면 segmentation 오류가 발생합니다. 이 문제는 해시테이블을 작은 고정된 크기로 사용하기 때문에 발생합니다. 따라서 해시테이블의 크기를 늘리는 것이 약간의 해결책이 될 수 있습니다.

그리고 또 다른 문제점은 DB의 버퍼의 개수를 작게 잡고 한 번에 많은 수의 테이블과 페이지에 접근하는 경우, lock을 얻고 버퍼에 올라와 있는 페이지 수가 버퍼의 수와 같아져 더 이상 읽기를 수행할 수 없다는 점입니다. 이 경우 본 DBMS에서는 에러가 발생하여 종료됩니다. 이 경우 버퍼에도 lock object list와 마찬가지의 방법으로 doubly-linked-list로 대기열을 만들어 버퍼를 대기시키고 버퍼에 빈 자리가 생기면 올릴 수 있도록 디자인하여 해결할 수 있습니다.

2. Workload with many concurrent non-conflicting write-only transactions.

많은 수의 non-conflicting write-only transaction이 동시에 수행되고 Crash가 발생하는 경우 본 DBMS에서는 Crash-Recovery를 지원하지 않기 때문에 수행하던 트랜잭션이 모두 종료되고 Recovery를 할 수 없게 되어 버립니다. 이 경우 버퍼에 dirty 상태로 되어 있고 Disk에 write되지 않은 것들은 모두 수행하지 않은 상태가 됩니다.

지금 구현되지 않은 상태이지만, 앞으로 No-force & Steal 기반 Crash-Recovery를 구현한다고 가정한다면 Crash가 발생하고 Recovery를 할 때 commit된 트랜잭션의 Redo와 commit되지 않은 트랜잭션의 Undo Operation으로 인한 오버헤드가 발생합니다. 트랜잭

션이 생성되고 오랫동안 flush나 commit을 하지 않은 경우 해당 오버헤드가 매우 커지게 됩니다. 그리고 Recovery를 하는 도중에 Crash가 다시 발생할 경우 처음부터 다시 Recovery를 해야 하므로 이를 보완하는 디자인으로 Redo-History Algorithm을 사용할 수 있습니다. Winner와 Loser 관련 없이 모든 트랜잭션에 대해 Redo를 한꺼번에 수행하고, 그 다음 Undo를 수행하며 각 트랜잭션에 Compensate Log를 발급하여 Undo가 어디까지 수행되었는지 기억하게 합니다. 이 디자인을 사용하면 Recovery 도중 Crash가 발생하여도 했던 Undo를 또 다시 하지 않고, Crash 발생 지점부터 다시 Undo Recovery를 할 수 있습니다.