Database System 2020-2
Final Report

ITE2038-11800

2016025205

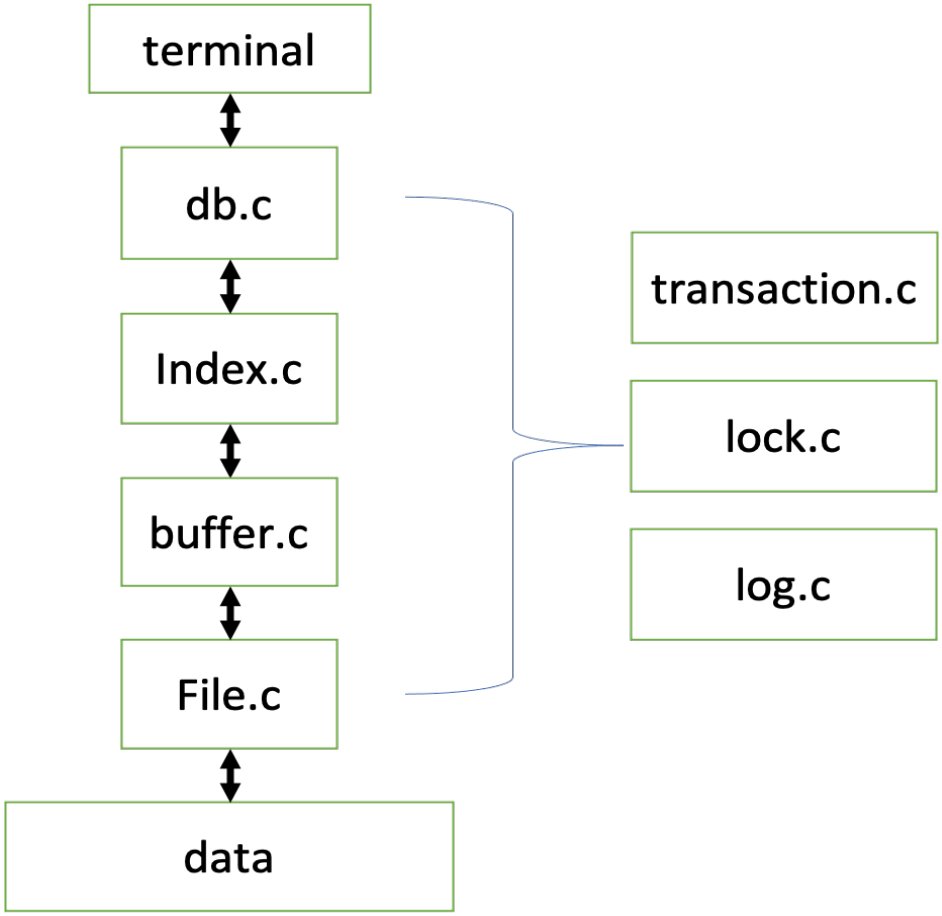
김범진

목차

1. Overall Layered Architecture

왼편에 layer들은 db의 operation과 관련된 layer, 서로 자신의 위 아래의 layer하고만 정보 전달 가능

오른편에 layer들은 db의 ACID를 만족시키기 위한 layer들 전체 layer들과 직접 정보 전달 가능



1. db.c

    DB의 API를 담당하는 부분.

2. index.c

    B+tree 인덱스 구조를 만들고 관리하는 부분

3. buffer.c

    파일 IO와 메모리 사이의 속도 차이를 줄여주기 위한 버퍼 부분

4. file.c

파일 IO를 직접적으로 담당하는 부분

5. transaction.c

여러개의 트랜잭션을 발급 및 관리 하는 부분

6. lock.c

DB의 record의 접근할 수 있는 권한을 얻는 lock을 관리하는 부분

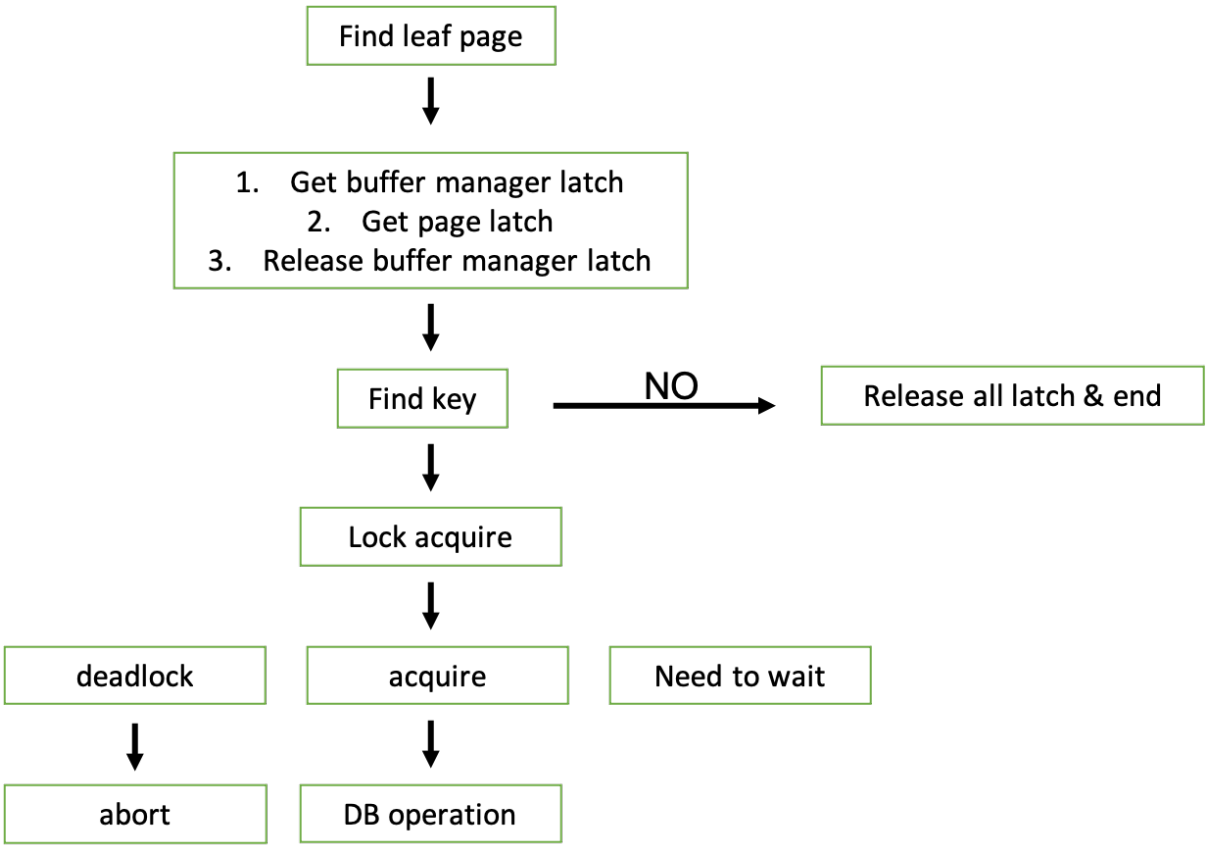
7. log.c

Crash & Recovery를 위한 log를 담당하는 부분

2. Concurrency Control Implementation

Concurrency Control은 lock.c와 transaction.c 에서 관리하게 됩니다. 먼저 conflict serializability를 위해서 strict 2PL을 이용하였습니다. Strict 2PL로 transaction 간 consistency를 보장합니다.

- Work Flow



- 구현

Transaction.c

Trx_begin() : 트랜잭션 ID를 발급하는 API

Trx_commit() : 트랜잭션을 정상종료하는 API

각각의 트랜잭션들은 ID를 통해서 hash로 관리하게 됩니다.

Lock.c

```
struct list{

    int table_id;
    int64_t key;
    int lock_num;
    pagenum_t pagenum;
    pthread_mutex_t *mutex;

    list *link;

    lock_t *head;
    lock_t *tail;
};
```

```
struct lock_t {
    pthread_cond_t *cond;
    list *pointer;

    lock_t *next;
    lock_t *prev;

    char *stored;
    bool get;

    int lock_mode;
    int trx_id;
    uint64_t prev_LSN;
    lock_t *trx_next;
};
```

struct list 는 record에 매달리는 lock들을 관리하는 구조체. Table id와 key를 통해 hash로 관리.

struct lock_t 는 실제로 매달리는 lock

Lock_acqurie() : record의 메다는 lock을 결정 상황에 따라 acquired, wait, deadlock 결정

Lock_release() : 달린 lock을 풀어주고 기다리는 다음 lock을 깨워주는 API

Lock에는 shared와 exclusive 두가지 모드가 있으며 shared는 읽기만 exclusive는 읽기 쓰기 모두 가능한 lock이고 lock_mode 변수를 통해 관리합니다.

이를 구현하기 위해서 lock_t의 condition variable 변수 cond와 boolean 변수 get을 통해 구현하였습니다.

먼저 Boolean 변수를 통해 mutex를 얻은 상태인지 아닌지를 논리적으로 구분해줍니다.

만약 mutex를 얻기 위해 대기해야한다면 condition variable을 통해서 잠에 들게 되고 앞선 lock이 release 될 때 해당 lock에 대한 condition variable에 시그널을 보내 깨워줍니다.

이 때 lost wake-up problem 발생을 막기 위해서 잠에 들기 전 해당 트랜잭션에 달린 mutex를 획득하고 mutex를 unlock하면서 잠에 드는 general design에 대해 설명해주셨는데 저는 이 부분도 lost wake-up problem이 발생할 수 있다고 생각하여서 transaction이 시작될 때부터 해당 transaction mutex를 lock 해줍니다. 그러면 이 mutex가 unlock 될 때는 lock을 얻기 위해 기다리면서 잠이 들 때 혹은 Commit or abort 될 때 뿐이므로

lost wake-up problem을 더 명확히 해결해줄 수 있다고 생각하였습니다.

- 보완할 점

위에 work flow에 나온 deadlock 인 경우는 deadlock detection을 제대로 구현하지 못해서 code상으로는 막아 두었습니다.

3. Crash-Recovery Implementation

Crash-recovery는 log.c 에서 관리합니다. Recovery를 위해서는 redo-history algorithm을 이용해 구현하려고 하였습니다. 저는 start offset을 이용하여 recovery 프로그램을 작성하였습니다. Prev LSN을 위하여 트랜잭션을 관리하는 구조체에 LSN 변수를 추가하여 마지막 LSN을 저장할 수 있도록 하였습니다. 또 Compensate Log를 위해서 lock에도 LSN을 추가하였고 트랜잭션의 마지막 LSN을 lock에 저장한 다음 로그를 작성하면서 트랜잭션의 마지막 LSN을 업데이트 해줍니다. 저는 Log list를 linked list로 구현하여 log 버퍼가 가득차면서 해야 하는 log flush를 구현하지 않았습니다. linked list에 head와 tail을 접근할 수 있는 변수를 두어 log flush는 앞에서부터 log 추가는 뒤에서부터 할 수 있도록 구현하였습니다. 또한 LSN durability를 위해 flush한 후 맨 마지막 로그를 head의 저장해서 durability를 유지합니다. 이 때 중복으로 log가 flush되면 안되기 때문에 항상 head의 다음 log부터 flush 합니다.

- analysis

로그 파일을 처음부터 읽어갑니다. 이 때 중요한 로그의 타입과 로그 사이즈만을 읽습니다. 타입이 begin이면 임시 list에 추가합니다. 타입이 commit 이나 rollback이면 winner list에 추가합니다. 파일을 끝까지 읽은후 winner list와 임시 list를 비교하면서 임시 list에 있지만 winner list에 없는 trx id를 loser list에 추가하여 줍니다. 또한 lsn의 durability를 위해서 마지막으로 읽은 LSN과 size를 log linked list head의 저장합니다.

- redo

다시 로그 파일을 처음부터 읽어갑니다. type이 begin, commit, rollback 이면 해당하는 log message를 출력합니다. type이 update나 compensate라면 로그에 적힌 table_id 를 통해서 테이블을 열고 페이지 번호를 통해 page를 읽어옵니다. 이 때 consider redo를 위해 page의 적힌 LSN과 log의 LSN을 비교합니다. 만약 page의 적힌 LSN이 log의 LSN보다 크거나 같다면 consider redo 입니다. 아니라면 log의 적힌 new data를 offset을 통해서 해당 page위치에 적어줍니다. 그리고 만약 update를 한 트랜잭션이 loser라면 undo를 위해서 log를 만들고 undo를 위한 log list에 달아줍니다. log를 만드는 이유는 compensate log를 발급하기 위함입니다. redo crash flag가 설정되어 있다면 해당 log_num에서 redo를 그만하고 아니라면 파일의 끝까지 읽어가면서 redo를 진행합니다.

- undo

redo에서 만든 log list를 통하여 undo를 진행합니다. 트랜잭션의 id가 큰 것부터 작은 순으로 진행합니다. list의 log가 update라면 compensate log를 log의 정보들을 이용해 log를 발급합니다. 이 때 next undo LSN은 prev_LSN 이 됩니다. log의 table id와 페이지번호를 이용해 page를 읽어온다음 log의 old data를 page에 적어줍니다. type 이 compensate라면 next undo LSN을 이용해서 log list를 이동하여 LSN이 next undo LSN과 같아지는 지점까지 이동합니다. undo crash flag가 되어있다면 해당하는 log num에서 멈추고 아니라면 모든 loser list를 다 돌면서 undo를 진행합니다.

- 보완할 점

Crash-recovey 부분은 완성도가 매우 낮습니다. 코드 최적화도 안되어 있고 Test시 잘 동작하지 않는 부분이 더 많습니다. 이 부분은 제가 이 파트에 대한 이론적인 지식도 많이 부족하고 시간도 많이 부족하여 기간 내에 완성하지 못했지만 방학이나 모든 시험이 끝난 후 다시 recovey파트를 완성도있게 만들어볼 계획입니다.

4. In-depth Analysis

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

먼저 성능 문제가 발생할 가능성은 한 페이지에 있는 여러 레코드에 계속 접근할 때 발생할 가능성이 있습니다. 왜냐하면 record에 lock을 달기위해 먼저 page latch를 잡게 되는데 다수의 페이지에 접근하게 된다면 기다리는 시간이 줄어들어 성능 측면에서 문제점이 발생할 가능성이 낮지만 한 페이지에 있는 여러 레코드들의 반복적으로 접근하게 된다면 page latch를 잡기 위해 대기하는 시간이 길어져 문제점이 발생할 수 있습니다.

이를 해결하기 위해서는 트랜잭션을 시작하는 API인 trx_begin()을 실행할 때 입력 인자로 모드를 설정할 수 있게 하면 됩니다. Transaction의 mode를 read only mode와 read-write mode 두가지로 선택해서 세팅합니다. 이 때 read only mode의 트랜잭션은 수정이 없으니 수정으로 인한 DB의 ACID를 해칠 가능성이 없습니다. 따라서 record에 접근할 때 page latch를 잡지 않고 page에 접근하여 record를 확인한다면 해당하는 문제에 따른 성능 저하 측면을 해결할 수 있습니다. 또한 read only 트랜잭션이 update를 할 수 없도록 update API 내부에서도 이를 확인하는 작업을 수행하면 correctness 문제를 해결할 수 있습니다.

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

3에서 작성한 것처럼 제 코드의 crash-recovery는 미완성이기 때문에 advanced problem을 해결할 단계는 아닌 것 같습니다. 먼저 basic한 problem을 다 충족시키고 나서 advanced problem을 해결하는 것이 좋은 방향인 것 같습니다.

다만 어떤 문제가 생길지 예상해 본다면

1. Log buffer write & flush time
2. Winner loser analysis time
3. Redo time

이 생길 것 같습니다. 먼저 log buffer write 부분의 개선 방안을 생각해보면 지금의 코드는 버퍼가 한 개이기 때문에 수 많은 Transaction이 log write을 시도하기 위해서 한 줄로 서서 기다려야합니다. 이를 해결하기 위해서 버퍼의 갯수를 늘려서 transaction이 한 줄이 아니라 n-way로 설 수 있으면 대기시간을 많이 줄일 수 있을 것 같습니다.

다른 방법으로는 2-3번 문제를 해결하기 위해서 log 파일을 분화해보면 좋을 것 같습니

다. 지금은 여러 type의 log가 한 파일에 있어서 파일을 처음부터 끝까지 읽는 동작을 반복해야 하지만 log file을 disk에 write할 때 begin, commit rollback은 log A, update, compensate는 log B 이런 방식으로 log의 타입 별로 나눠서 disk에 적게 되면 파일을 불필요하게 read하는 시간이 줄어들어 recovery 시간을 줄일 수 있을 것 같습니다. Analysis pass 에서는 log A만 읽어서 한 파일에 있는 것 보다 더 빠른 시간에 winner와 loser를 판별할 수 있을 것 같고 redo 나 undo pass 에서는 log B만 읽어서 역시 불필요한 read 를 줄여 수행시간을 줄이는 것이 가능할 것 같습니다.