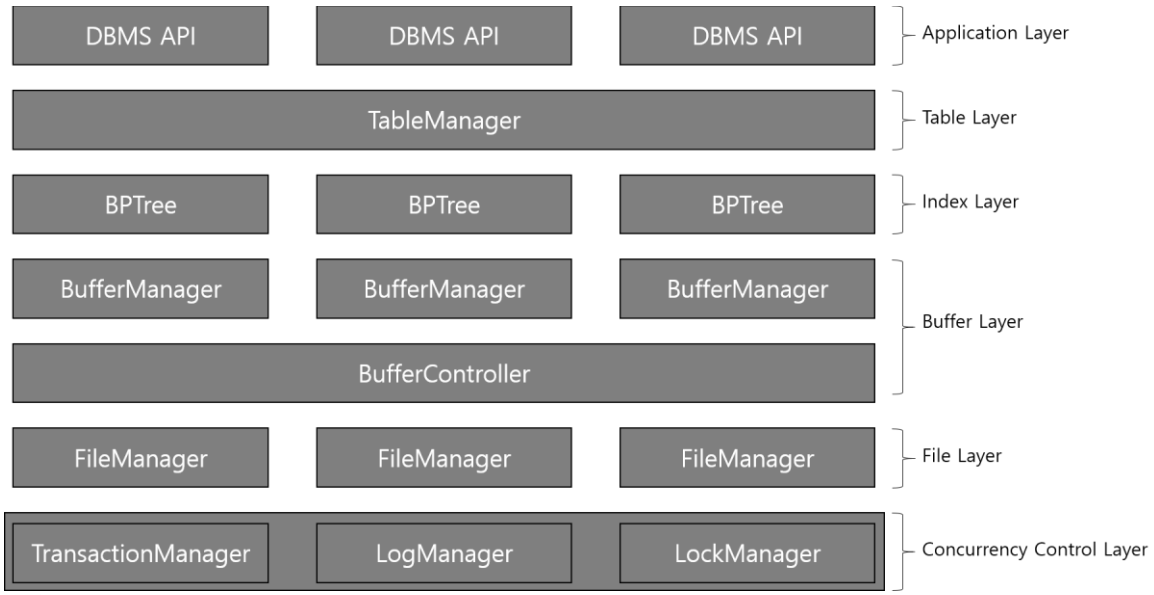
Database System 2020-2
Final Report

ITE2038-11801
2019027192
김현수

Table of Contents

1. Overall Layered Architecture



계층 구조를 그림으로 나타내면 위와 같습니다.
각 layer에 대한 설명은 다음과 같습니다.

1. Application Layer

library(libbpt.a)를 link한 프로그램에서 제가 구현한 DBMS를 이용할 수 있도록 제공하는 API 계층입니다.
제공하는 API는 다음과 같습니다.

int init_db(int buf_num);
project6 이전의 프로젝트들을 위한 legacy API 입니다. “default.log” 파일명으로 log file을 생성하고, “msg.txt” 파일명으로 log message file을 생성합니다.

int init_db(int buf_num, int flag, int log_num, char* log_path, char* logmsg_path);
dbms를 초기화 하는 함수입니다. 초기화한다는 특성상 DBMS의 전체 layer에 영향을 줍니다.
BufferController의 buffer array size를 buf_num으로 설정합니다.
과제 명세에서 요구한 대로 recovery redo/undo crash를 구현했습니다.

int open_table(char* pathname);
pathname의 db file을 이용해 table을 엽니다. table layer를 호출합니다.

int db_insert(int table_id, int64_t key, char* value);
db에 insert합니다. table layer를 호출합니다.

int db_find(int table_id, int64_t key, char* ret_val, int trx_id);
find 합니다. table layer를 호출합니다.
concurrency control을 지원합니다.

int db_update(int table_id, int64_t key, char* values, int trx_id);
update 합니다. table layer를 호출합니다.
concurrency control을 지원합니다.

int db_delete(int table_id, int64_t key);
delete 합니다. table layer를 호출합니다.

int close_table(int table_id);
특정 table을 닫습니다. table layer를 호출합니다.

int shutdown_db();
RAM에 존재하는 모든 정보를 stable stroage에 flush 하고, DBMS를 종료합니다.
그 특성상 전체 layer에 영향을 줍니다.

int trx_begin();
transaction을 begin하고, id를 발급받습니다. concurrency layer의
TransactionManager를 호출합니다.

int trx_abort(int trx_id_);
특정 transaction을 abort 합니다. concurrency layer의 TransactionManger를
호출합니다.

int trx_commit(int trx_id);
특정 transaction을 commit 합니다. concurrency layer의 TransactionManger를
호출합니다.

2. Table Layer

여러 개의 table을 열어 관리할 수 있게 하는 계층입니다. 구현 상 한번에 열 수 있는 table에는 딱히 제약이 존재하지 않지만, 과제 명세를 지키기 위해 10개로 제한했습니다.
table layer에서는 바로 아래의 계층인 index layer를 호출합니다.

3. Index Layer

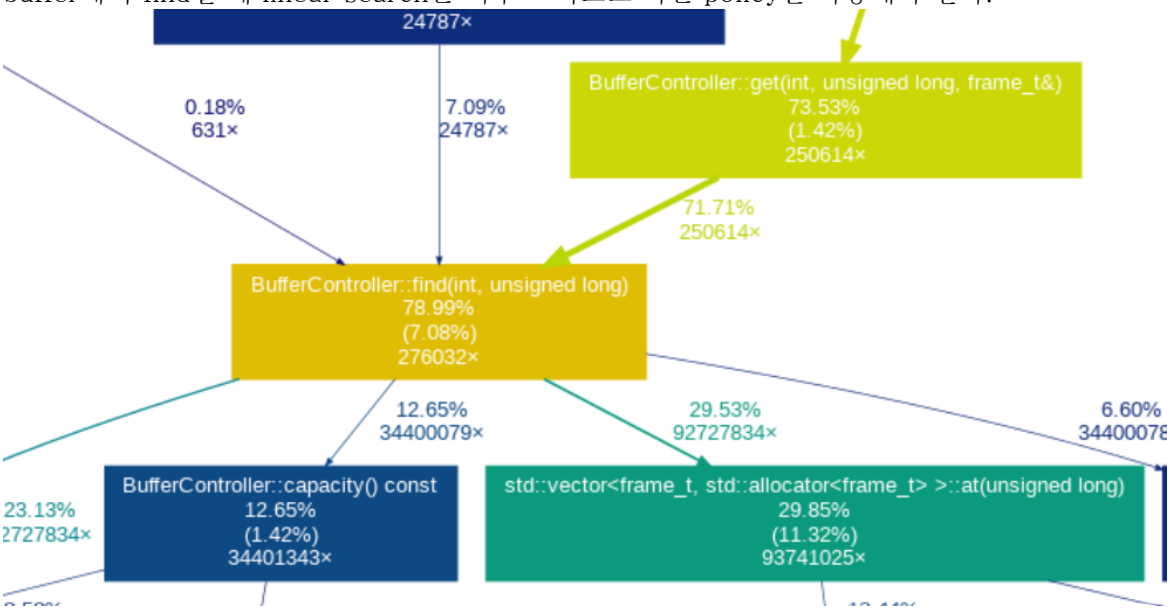
B+ Tree를 구현한 계층입니다.
db file에 File Layer를 통해 직접 접근하지 않고, Buffer Layer를 통해 접근하게 구현했습니다.
concurrency control을 위해, concurrency control 계층의 LockManager, LogManager를 호출합니다. 이 두 클래스와 BufferController는 latch로 보호되어야 하기 때문에 2PL locking protocol로 latch를 걸고, LockManager를 이용해 record lock 역시 사용합니다.

4. Buffer Layer

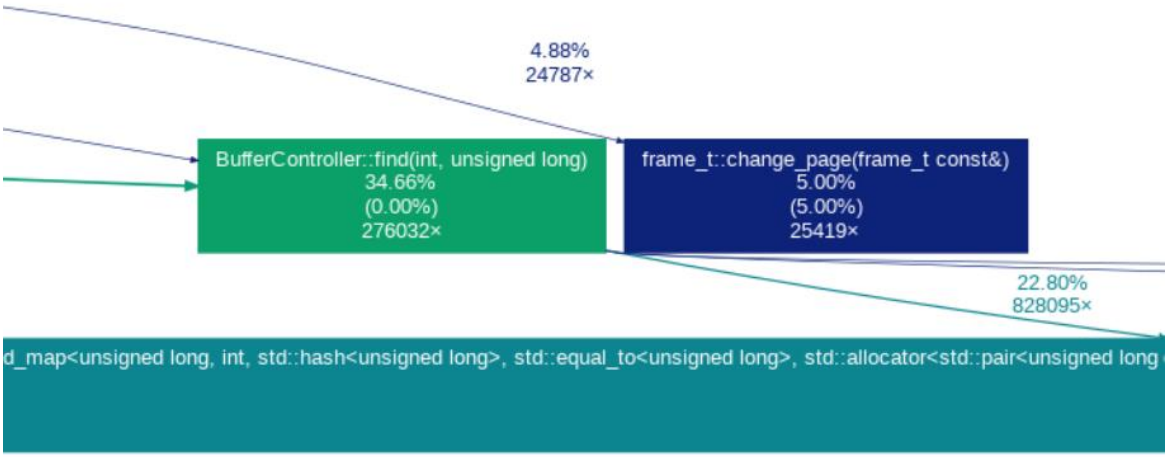
File Layer를 이용해 db 파일에 접근하는 작업은 비용이 큽니다. 그래서 중간에 page buffer 역할을 하는 계층을 만들었습니다.
buffer의 구현에서 달성해야 했던 목표들은 다음과 같습니다.

1. buffer가 가득 차면 LRU policy로 victim을 찾아 free하고, 해당 위치에 load해야 한다.
   double linked list로 frame들을 묶고, frame을 사용할 때마다 double linked list의 맨 앞에 해당 frame을 위치시키는 방식으로 LRU policy, MRU policy가 가능하게 구현했습니다.
2. 사용중인 frame의 pin은 1 이상이여야 한다. 사용되지 않는 frame의 pin은 0이어야 한다.
   Concurrency Control Layer를 구현하며, frame의 pin 없이 frame latch 만으로도 pin을 대체할 수 있게 되었습니다. 하지만, 기존 계층(Buffer layer)의 변경을 최소화 하기 위해 pin 역시 남겨 두었습니다.
   scoped_node_latch, scoped_node_latch_shared 등에서 pin을 조작해, 이 목표를 달성합니다.

3. buffer에서 find할 때 linear search는 너무 느리므로 다른 policy를 사용해야 한다.



linear search를 사용할 경우, buffer의 크기가 1만일 경우, single thread 테스트에서 전체 실행시간의 79%를 linear search를 하는데 사용하게 됩니다. 심지어 이 경우에는 buffer의 크기를 증가시킬수록 실행시간도 늘어난다는 최악의 결과를 보여주었습니다.
따라서 find에 사용하기 위해 std::unordered_map을 사용하기로 했습니다. file_id와 pagneum이 매우 작은 범위의 정수이므로, hash table 방식인 std::unordered_map을 사용하면 O(1)에 find가 가능합니다.
단, std::unordered_map은 thread-unsafe 하므로, find 함수는 buffer controller latch로 보호되어야만 합니다.



std::unordered_map을 사용한 이후에는 병목이 크게 감소한 모습입니다.

4. buffer의 size가 capacity 보다 작아 새 frame을 alloc 할 때, linear search는 너무 느리므로 다른 policy를 사용해야 한다.
   buffer의 size가 capacity 보다 작다면, buffer에서 invalid한 frame을 찾아 해당 위치에 load해야 합니다. 이 invalid한 frame은 buffer에 무작위로 흩어져 있으므로, naive하게 구현하면 buffer 전체를 linear search 해야 합니다.
   하지만 linear search는 3번에서 밝혔듯이 매우 느리기 때문에, 3번 에서와 비슷하게 frame alloc에 사용될 수 있는 frame들을 std::unordered_map에 넣어놓고, alloc 할 때 빼서 사용하는 방식을 사용했습니다. 이 역시 buffer controller latch로 보호되어야만 합니다.

5. File Layer
   db file에 접근하는 계층입니다. buffer 관리를 buffer layer에서 하므로, pwrite 이후 항상 fsync 하도록 구현했습니다.

6. Concurrency Control Layer
   Concurrency Control을 제공하는 계층입니다. Transaction 관리를 하는 TransactionManager, record lock 기능을 제공하는 LockManager, logging 기능을 제공하는 LogManager 이렇게 3개의 클래스로 구성되어 있습니다.
   자세한 설명은 밑의 2. Concurrency Control Implementation에 적혀 있습니다.

2. Concurrency Control Implementation

concurrency control를 위한 기법은 크게 2가지로 나뉩니다.

1. latch
   multithreading 상황에서 메모리의 배타적 접근을 통해 자료구조를 보호
2. lock
   multithreading 상황에서 record의 접근을 제어해 ACID속성을 준수하고, 데이터베이스를 보호

1. latch

latch는 multithreading 상황에서 메모리에 위치한 자료구조를 보호하는 장치를 말합니다.
latch는 STL의 std::mutex, std::shared_mutex, std::recursive_mutex, std::unique_lock 등을 이용해 구현했습니다.
dbms를 구현하면서 std::unordered_map과 같은 STL을 사용했는데, STL은 thread-unsafe 합니다. 따라서 다수의 thread가 STL에 접근하면 잘못된 동작을 하게 됩니다.
이러한 현상을 막기 위해 std::mutex와 std::unique_ptr을 이용해 critical section을 만드는데, 이를 latch라고 합니다.
주로 사용되는 latch는 다음과 같습니다.

1. BufferController latch
   page buffer를 보호하는 latch입니다. buffer 자체는 크기가 고정된 std::vector 이므로, 서로 다른 element에 대한 read/write 자체는 thread-safe 합니다. 하지만, LRU list를 update하거나, file id/pagenum을 buffer index와 1대1 대응시키기 위해 std::unordered_map을 사용하는 것은 thread-unsafe 합니다.
   따라서 LRU list, std::unordered_map에 접근할 때는 BufferController latch를 이용해 자료구조를 보호합니다.
2. TransactionManager latch
   transaction id와 transaction 객체를 1대1 대응시키기 위해 std::unordered_map을 사용하므로, thread-unsafe 합니다. 따라서 TransactionManager latch를 이용해 자료구조를 보호합니다.
3. Transaction latch
   transaction이 record lock을 획득하기 위해 wait하기 전에, record lock을 획득하고 있던 transaction이 record lock을 release하고 다른 transaction에 signal을 준다면, Lost Wakeup이 발생합니다. 이 문제를 방지하기 위해, Transaction latch를 이용해 wait-> signal 보냄->signal 받음 과정이 순서대로 이루어지도록 합니다.
4. page latch
   하나의 page에는 다수의 record가 저장되어 있습니다. record lock은 하나의 record만을 보호합니다. 만약 page latch가 없다면, 서로 다른 record가 수정된 두 page가 buffer에 write 되는 과정에서 race condition이 발생해 lost update 문제가 발생할 수 있습니다. 이를 방지하기 위해 page latch를 사용합니다.
   만약 lock_wait을 위해 잠깐 page latch를 풀었을 경우, 다시 page latch를 획득한 이후 해당 페이지를 다시 buffer에서 불러와야 합니다.

2PL locking protocol을 준수하기 위해, 모든 latch는 BPTree 클래스에서 한번에 획득되고, 한번에 해제됩니다.

2. lock

lock은 record의 접근을 제어하는 장치를 말합니다.
제 구현의 경우, lock은 concurrency layer의 LockManager 클래스에서 담당합니다.
lock은 2가지 종류로 나뉩니다.

1. shared lock (SLock)
2. exclusive lock (XLock)

SLock과 XLock은 다음과 같은 속성을 가지고 있습니다.

1. SLock과 XLock은 transaction이 commit/abort 될 때 release 된다.
2. 여러 transaction은 하나의 record에 다수의 SLock을 걸 수 있다.
3. SLock이 걸려있는 record에는 다른 transaction이 XLock을 걸 수 없고, SLock을 건 transaction이 commit/abort 될 때까지 대기해야 한다.
4. XLock이 걸려있는 record에는 다른 transaction이 SLock, XLock을 걸 수 없고, XLock을 건 transaction이 commit/abort 될 때까지 대기해야 한다.
5. 요청한 순서대로 처리된다 (FIFO)

1~4까지의 속성은 pthread_rwlock 같은 표준 라이브러리의 rwlock과 비슷합니다. 하지만, 퍼준 라이브러리의 rwlock은 무작위 순서이고, SLock과 XLock은 FIFO라는 차이점이 있습니다. 이는 isolation을 위해 꼭 필요한 부분입니다.

제가 SLock과 XLock을 구현한 방식은 다음과 같습니다.

1. 만약 트랜잭션이 아무런 lock도 갖고 있지 않다면, lock_acquire를 호출한다.
2. 만약 트랜잭션이 lock을 갖고 있다면, 기존의 lock과 요청받은 lock의 mode를 비교해 결정한다.

2-1. 만약 트랜잭션이 이미 획득하고 있는 lock이 요청받은 lock보다 더 강하거나 같다면, lock_acquire를 호출하지 않는다. (XLock을 획득하고 있다면 XLock, SLock / SLock을 획득하고 있다면 SLock만)
2-2. 만약 요청받은 lock이 이미 획득하고 있는 lock보다 강하다면 (SLock일때 XLock 요청), lock_upgrade를 호출해 list에 XLock을 추가한다. (conversion deadlock 감지를 위해 SLock은 그대로 둡니다)

하나의 트랜잭션이 SLock을 획득하고 있다면, 다른 트랜잭션은 XLock를 획득하기 위해서는 해당 트랜잭션이 commit 되기까지 대기해야 합니다.
또한 XLock을 획득하고 있다면, 다른 트랜잭션은 해당 트랜잭션이 commit 되기까지 R1에 접근할 수 없습니다.
따라서 이미 SLock을 획득했다면, 다른 트랜잭션의 상태와 상관없이 항상 R1을 읽을 수 있고, XLock을 획득했다면 항상 R1을 읽고 쓸 수 있습니다. 그래서 2-1와 같이 구현했습니다.
SLock을 획득했는데, 쓰기도 하고 싶어 XLock을 획득하려 하는 경우, 아래와 같은 conversion deadlock을 감지할 수 있어야 합니다. (호출 순서입니다)
S1 X2 X1
그래서 저는 S1은 그대로 두고, X1을 lock list 맨 뒤에 넣는 방식으로 lock_upgrade를 구현했습니다. 이 경우 X2가 S1을 기다리고, X1이 X2를 기다리는 conversion deadlock을 제대로 감지할 수 있게 됩니다. 그래서 2-2와 같이 구현했습니다.

SLock과 XLock은 deadlock이 생길 수 있는 구조입니다. 따라서 lock_acquire 또는 lock_upgrade 할 때마다 deadlock detection을 진행해야 합니다.

deadlock detection에서는 lock list들을 모두 순회하며, 트랜잭션이 어떤 트랜잭션이 끝나기를 대기하고 있는지를 나타내는 wait-for-graph를 만듭니다.

이 함수는 lock_acquire나 lock_upgrade에서 새로운 Lock이 추가될 때 호출되는 함수이니, 데드락을 일으키는 lock cycle에는 lock 발급을 요청한 트랜잭션이 항상 포함되어 있습니다 그러므로 lock 발급을 요청한 트랜잭션을 시작으로, dfs 탐색을 하며 방문 기록을 visited에 저장합니다. 만약 방문했던 노드에 한 번 더 방문한다면 cycle이 있다는 의미가 됩니다.

waif-for-graph는 directed graph이므로, dfs로 cycle을 탐색해야 합니다. project5 제출 당시 bfs로 cycle을 탐색하게 했는데, 이로 인해 deadlock을 감지하지 못하는 문제가 생겼었습니다. project6에서는 수정했습니다.

3. Crash-Recovery Implementation

crash recovery를 위한 단계는 크게 5단계로 나뉩니다.

1. log
   wal에 따라, log를 기록하는 단계입니다.
2. analysis
   recovery가 발생했을 때, log 파일을 분석해 winner, loser transaction을 구분하는 단계입니다.
3. redo
   log를 따라 모든 연산을 redo해 durability를 보장하는 단계입니다.
4. undo
   loser transaction을 undo해 atomicity를 보장하는 단계입니다.
5. truncate
   recovery가 성공적으로 완료된 부분의 log는 다음 recovery에서 뛰어넘도록 해 recovery 속도를 개선시키는 단계입니다.

코드 구현 관련은 모두 gitlab의 project6 문서에 포함되어 있으니, 코드 없이 설명을 작성하도록 하겠습니다.

1. log 단계

log 단계의 경우, 제 구현에서 LogManager가 맡고 있습니다. LogManager의 메소드를 호출해 log를 발급받을 수 있습니다. 로그가 발급되는 상황은 다음과 같습니다.

1. transaction이 begin 될 때 (TransactionManager::begin)
2. transaction이 commit 될 때 (TransactionManager::commit)
3. transaction이 abort되어 rollback 될 때 (TransactionManager::abort, deadlock detected)
4. update 함수를 통해 특정 key에 해당하는 record를 update 할 때 (BPTree::update)
5. update log를 따라 undo를 진행할 때 (LogManagert::rollback, LogManager::recovery)

위와 같이 5가지 상황에서 각각 BEGIN, COMMIT, ROLLBACK, UPDATE, COMPENSATE 로그가 발급됩니다.

atomicity와 durability를 위해, log 단계는 write-ahead logging (WAL)을 따라 진행됩니다. 위의 5가지 상황에서 WAL을 따르는 방식은 다음과 같습니다.

1. transaction이 begin 될 때 (TransactionManager::begin)
   BEGIN log를 log buffer에 append한 이후에 trx id를 반환
2. transaction이 commit 될 때 (TransactionManager::commit)
   COMMIT log를 log buffer에 append하고, 모든 log를 flush 한 이후에 함수 종료
3. transaction이 abort되어 rollback 될 때 (TransactionManager::abort, deadlock detected)
   ROLLBACK log를 log buffer에 append 한 이후에 rollback을 진하고, 모든 log를 flush
4. update 함수를 통해 특정 key에 해당하는 record를 update 할 때 (BPTree::update)
   UPDATE log를 log buffer에 append 한 이후에 update를 진행
5. update log를 따라 undo를 진행할 때 (LogManagert::rollback, LogManager::recovery)
   COMPENSATE log를 LogBuffer에 append 한 이후에 undo를 진행

위와 같이 log를 작성할 때 WAL을 따를 뿐만 아니라, page buffer에 있는 내용을 stable storage에 위치한 db 파일로 옮기기 전에 log buffer를 flush합니다.

1. commit, rollback 이후에 log buffer를 flush 한다
2. buffer page에 있는 page를 db file에 write 하기 전에, 해당 page의 page_lsn보다 이전 log들을 log buffer에서 flush 한다.

앞에서 언급한 과정들을 통해 ACID중 atomicity와 durability를 보장할 수 있습니다. 그 이유는 다음과 같습니다.

1. 동작 중인 dbms의 RAM, log file, db file 이렇게 세 위치에서 transaction의 상태는 running, committed/aborted로 나눌 수 있다.
2. WAL에 의해, db file 속 transaction의 상태가 committed/aborted라면 log file 역시 committed/aborted임이 보장된다.
3. 2번에 의해, 만약 RAM속 log 작성 -> commit/abort 단계에서 CRASH가 발생했다면, log file에서 해당 transaction의 상태는 committed/aborted이고 db file에서 해당 transaction의 상태는 running이다. 이 경우 recovery의 redo 과정을 통해 db file 속 transaction의 상태를 committed, aborted로 바꿔줘 durability를 보장한다.
4. page를 write하기 전에 page_lsn보다 이전 log들을 log buffer에서 flush한다는 성질 덕분에, ‘db file속 transaction의 상태가 running이다’와 ‘log file속 transaction의 상태가 running’이다는 biconditional임이 보장된다.
5. 따라서 recovery의 undo 과정을 통해 log file과 db file의 running transaction (loser transaction)들을 undo 함으로써 atomicity를 보장한다.

2. analysis 단계

log buffer를 순회하며 winner/loser transaction을 구분하는 단계입니다.
제가 구현한 analysis 단계의 의사 코드는 다음과 같습니다.

```
winners, losers = dynamic array
trx_table = map
foreach log in stable_log_file:
        trx = log.trx
        type = log.type
        if trx not in trx_table and type==BEGIN:
                losers.push(trx)
        if trx in trx_table and type in (ROLLBACK, COMMIT):
                losers.erase(trx)
                winners.push(trx)
```

analysis는 log file을 앞에서부터 순회하며 진행됩니다. BEGIN을 만나면 해당 log의 트랜잭션을 losers에 넣습니다.
만약 계속 순회하다 ROLLBACK이나 COMMIT을 만난다면, 해당 log의 트랜잭션을 losers에서 꺼내 winners에 넣습니다.

위와 같이 구현하면, BEGIN만 있는 트랜잭션은 losers에 들어가고, ROLLBACK이나 COMMIT도 있는 트랜잭션은 winners에 들어가게 됩니다. winner/loser transaction의 정의에 따라 트랜잭션들이 잘 구분되었다는 의미입니다.

3. redo 단계

log buffer를 순회하며 redo해 durability를 보장하는 단계입니다.
제가 구현한 redo 단계의 의사 코드는 다음과 같습니다.

```
foreach log in stable_log_file:
        type = log.type
        if type in (UPDATE, COMPENSATE):
                page = buffer.load(log.file, log.pagenum)
                if page.page_lsn >= log.lsn:
                        consider-redo
                        return
                page.page_lsn = log.lsn
                page = log.new_image
                buffer.save(page)
```

5가지의 log type 중에서, redo해야 하는 type은 UPDATE와 COMPENSATE입니다.
page_lsn을 비교해, 현재 redo 중인 log의 lsn보다 더 최신이라면 redo를 할 필요가 없으니 consider-redo를 합니다.
만약 현재 redo 중인 log의 lsn이 page_lsn보다 최신이라면 redo가 필요하므로, redo를 진행합니다.

4. undo 단계

log buffer를 역순으로 순회하며 undo해 atomicity를 보장하는 단계입니다.
제가 구현한 undo 단계의 의사 코드는 다음과 같습니다.

```
pq = PriorityQueue
foreach loser in losers:
        pq.push(trx_table[loser]) // loser 트랜잭션의 마지막 lsn

while (!pq.empty()):
        log = pq.pop() // lsn이 가장 최신인 로그를 pop 한다
        type = log.type
        if type == BEGIN:
                buffer.append(ROLLBACK(log))
        else if type == COMPENSATE:
                pq.push(log.next_undo_lsn)
        else if type == UPDATE:
                buffer.append(COMPENSATE(log))
                page = buffer.load(log.file, log.pagenum)
                page.page_lsn = log.lsn
                page = log.old_image
                buffer.save(page)
                pq.push(log.prev_lsn)
```

log의 prev_lsn과, COMPENSATE log의 next_undo_lsn은 transaction 단위입니다. undo는 lsn 기준 내림차순으로 이루어 져야 하기 때문에, 우선순위 큐를 이용했습니다. 우선순위 큐에는 undo가 필요한 log들의 lsn이 저장됩니다.
loser transaction들의 마지막 lsn을 우선순위 큐에 넣습니다. 이제 우선순위 큐에서 가장 최신의 lsn을 꺼내고, undo를 진행해 주면 됩니다.
ARIES 알고리즘을 사용하므로, consider-undo는 발생하지 않습니다. (모든 log를 redo 하므로)
UPDATE log는 먼저 COMPENSATE log를 append 하고, undo를 진행합니다. 그 다음 해당 log의 prev_lsn을 우선순위 큐에 넣습니다.

COMPENSATE log는 undo할 필요가 없습니다. (redo에서 이미 진행했으므로)
next_undo_lsn을 우선순위 큐에 넣어줍니다. recovery 도중 CRASH가 났을 때 이미 undo가 진행된 UPDATE들의 undo를 뛰어넘게 해줍니다.

5. truncate 단계

recovery가 완료된 log들은 다음 recovery에서 고려하지 않아도 됩니다. 또한, lsn은 계속 유지되지만 transaction은 매번 1부터 시작한다는 성질 때문에 analysis가 복잡해질 수도 있으므로, recovery 이후에 log들을 truncate 하는 것은 꼭 필요합니다.
특정 시점으로 되돌아가기(reset) 기능이 필요해질 수도 있으므로, log file에 존재하는 log를 실제로 삭제하는 것은 피해야 합니다.
따라서, 다음 recovery는 어느 lsn 부터 시작해야 한다는 정보를 log file의 맨 앞부분에 넣는데, 이 과정을 저는 truncate 단계라고 명명했습니다.
원래 log file의 lsn0에는 log record가 저장되지만, recovery의 truncate 단계 이후에는 12byte의 헤더가 위치하게 됩니다.
위 과정을 통해 실제로 truncate를 수행한 것은 아니지만, 헤더를 읽어 다음 recovery에서 truncate가 된 것 같은 효과를 낼 수 있습니다. 또한 추후에 만약 reset 기능을 구현한다면, 삭제되지 않고 log file에 남아있는 log record들을 이용하면 됩니다.

lsn0을 헤더로 덮을 수 있는 이유는, lsn0은 begin transaction1임이 보장되기 때문입니다. 그 이유는 다음과 같습니다.

1. transaction id는 1부터 시작한다.
2. COMMIT, ROLLBACK, UPDATE, COMPENSATE log는 BEGIN된 트랜잭션에 대해서만 발급 가능하다. 따라서 모든 트랜잭션의 첫번째 log는 BEGIN log이다.
3. 따라서 첫번째 트랜잭션의 첫번째 log는 begin transaction1이다.
4. lsn0에는 첫번째 트랜잭션의 첫번째 log가 위치한다.

따라서 구현하기에 따라 lsn 0의 begin record (28byte)는 헤더로 덮어버려도 recovery나 restore가 가능해집니다. (위 특성을 이용해 lsn 0이 transaction 1의 begin log임을 유추할 수 있으므로)
하지만 과제 명세상 log file에 저장되는 log record의 형식을 맞춰야 하므로, 28 byte를 모두 덮어버릴 수는 없습니다. 하지만 제 로그 헤더는 12 byte이고, 이는 일반적으로 문제가 되지 않습니다. 그 이유는 다음과 같습니다.

저와 동일하게 log size를 log record의 맨 뒤에 위치시킨 학생의 경우, lsn 0의 로그 레코드는 항상 아래와 같게 됩니다.

```
|==4 byte==|
+----------+----------+----------+----------+
|        lsn 0        |     prev lsn -1     |
+----------+----------+----------+----------+
| trxid 1  | type 0   | size 28  |
+----------+----------+----------+
```

여기에 12byte의 헤더를 작성하면 아래와 같게 됩니다.

```
|==4 byte==|
+----------+----------+----------+----------+
|0xffeeaaff|      start lsn      | reserved |
+----------+----------+----------+----------+
| trxid 1  | type 0   | size 28  |
+----------+----------+----------+
```

trxid, type, size를 변경하지 않습니다. lsn과 prev_lsn만 오염되게 됩니다. 로그 레코드에 접근했다는 사실 자체가 해당 레코드의 lsn을 이미 프로그램이 알고 있다는 의미이므로 lsn은 무결성 검사라도 진행하지 않는 한 사용될 일이 없고, prev_lsn은 begin log에서 사용되지 않는 수치입니다.

따라서 제 프로그램에서 recovery가 진행되었다 하더라도, 다른 학생분의 구현체에서 analysis, redo, undo 모두 정상적으로 진행될 수 있습니다. (lsn의 검사하거나, prev == -1 임을 이용하는 코드가 있다거나 할 경우 문제가 생길 수도 있긴 합니다.)

4-1. Workload with many concurrent non-conflicting read-only transactions.

non-conflicting read-only transactions는 항상 find 함수만을 호출합니다. find 함수에서 성능 측면의 문제가 생길 법한 부분은 다음과 같습니다.

1. record lock
2. latch

1. record lock 관련 성능 측면의 문제

non-conflicting 일 경우 모든 트랜잭션이 서로 다른 record에 접근하므로, find 함수에서 lock_acquire 함수를 호출한 이후 트랜잭션의 상태가 항상 acquire임이 보장됩니다. (abort, wait은 불가능합니다). 즉, 사실상 문제에서 가정한 non-conflicting 상황에서는 lock_acquire 함수를 호출하지 않아도 된다는 의미입니다.

따라서 Optimistic Concurrency Control 방식을 사용한다면, Pessimistic Concurrency Control로 구현해 놓은 지금보다 더 효율적일 수 있습니다.

하지만 이는 이 문제에서 가정한 특수한 상황만을 고려한 것입니다. 4-2 부분에서 설명하겠지만, 현재 제 구현상 rollback은 비용이 높습니다. (log buffer flush로 인해) 따라서, Optimistic Concurrency Control 방식으로 구현을 변경한다면 non-conflicting 상황에 한해 성능이 좋아지겠지만, 일반적인 상황에서는 잦은 rollback으로 인해 비효율적입니다. 따라서 이 부분은 성능 측면의 문제로 보지 않겠습니다.

2. latch 관련 성능 측면의 문제

non-conflicting는 서로 다른 record에 접근한다는 의미이므로, page latch는 성능 측면에서의 문제를 발생시킬 수 있습니다. 예를 들어, read-only transaction 여러 개가 동시에 하나의 page에 접근하는 것은 이상한 상황이 아니지만, 하지만 project5에서 제가 구현했던 page latch의 구조는 하나의 페이지에 하나의 transaction만이 접근할 수 있도록 제한하고 있었습니다. 저는 이 부분이 비효율적이라 생각해 project6에서 page latch를 read/write lock(이하 rwlock)으로 변경했습니다.

rwlock은 다음과 같은 특성을 가집니다. Lock Manager의 shared/exclusive와 비슷합니다.

1. 다수의 read를 허용한다.
2. read 중이면 write를 대기시킨다.
3. write 중이면 read와 write를 대기시킨다.

dbms의 find는 read-only 입니다. 따라서 page에 read lock을 걸면 됩니다. 이 경우 다수의 find가 해당 페이지에 접근할 수 있습니다. dbms의 update는 read-write 입니다. 따라서 page에 write lock을 걸면 됩니다. 이 경우 project5에서의 구현과 같이 해당 페이지에는 하나의 transaction만이 접근할 수 있게 되어 안전합니다.

위와 같이 저는 read-only transaction에서 생길 수 있는 성능 문제를 최소화하고자 노력했습니다. 하지만 project6 구현 과정에서 문제가 생겨 사실상 이 노력이 무용지물 되었습니다.

project6을 구현하기 전에, general lock design을 먼저 구현했습니다. 하지만 이 과정에서 제가 실수로 page latch를 건 이후에 buffer latch를 해제하지 않도록 구현했고, recovery를 구현하면서 이걸 잊어버리고 말았습니다. (예전 project5 구현에서는 이 부분이 잘 처리되어 있습니다)

이 경우, find 함수나 update 함수 전체가 buffer latch로 보호되는 critical section이 됩니다. lock_acquire나 transaction abort 등은 buffer latch로 보호되어야 하는 작업들이니 상관 없지만, find 함수에서 key에 해당하는 record를 찾는 작업이나 update 함수에서 key에 해당하는 record를 읽고, 쓰는 작업은 page latch만으로도 충분히 보호될 수 있는 작업입니다.

page latch만으로도 충분히 보호될 수 있는 작업들을 더 큰 단위인 buffer latch로 보호해 버리는 것은 성능 손실을 가져옵니다. 즉, 이 부분을 그대로 두면 page latch를 rwlock으로 구현한 것이 무용지물이 됩니다.

이를 해결하기 위해서는 다음과 같은 조건을 지켜야 합니다.

1. page latch를 잠근 이후에 buffer latch를 해제해야 한다. (불필요한 latch를 줄여 concurrency을 통한 성능 향상을 얻기 위해)
2. page latch를 잠근 이후에 buffer latch를 할당하는 것은 피해야 한다. (2PL protocol을 준수해 dead latch를 방지하기 위해)

project6으로 제출했던 제 find 함수의 의사 코드는 다음과 같습니다.
find(key, ret, trxid):
1. key에 해당하는 leaf node를 찾는다.
2. record lock 관련 처리를 한다. (acquire, abort, wait 등)
3. leaf node에서 key에 해당하는 record를 찾는다.
4. record를 읽어와 ret에 복사한다.

위 코드에서 latch로 반드시 보호되어야 하는 영역은 다음과 같습니다.

find(key, ret, trxid):
(buffer latch lock)
1. key에 해당하는 leaf node를 찾는다.
(page latch lock)
2. record lock 관련 처리를 한다. (acquire, abort, wait 등)
(buffer latch unlock)
3. leaf node에서 key에 해당하는 record를 찾는다.
4. record를 읽어와 ret에 복사한다.
(page latch unlock)
현재 제 구현에서 3번, 4번 코드는 buffer manager에 접근하지 않습니다. 따라서 다른 부분의 구조 변경 없이 3번 코드의 바로 위에 buffer latch unlock 코드를 삽입해 주면 끝입니다.

그러면 기존 구현과는 달리 read-only transaction의 경우 3번, 4번 작업을 concurrent하게 진행할 수 있게 됩니다. (3번, 4번 작업은 rwlock으로 보호되므로)
따라서 non-conflicting read-only transaction의 성능을 향상시킬 수 있게 됩니다.

update 함수는 read-only transaction에서 호출되지는 않지만, dbms 전체의 효율성과 관련된 주제이니 update 함수의 변경 역시 다루겠습니다. update 함수는 find 함수와는 달리, 현재 구조로는 조건 1과 2를 동시에 만족시키는 것이 불가능하므로, 약간의 수정을 가해야 합니다.

project6으로 제출했던 제 update 함수의 의사 코드는 다음과 같습니다.

update(key, value, trxid):
(buffer latch lock)
1. key에 해당하는 leaf node를 찾는다
(page latch lock)
2. record lock 관련 처리를 한다. (acquire, abort, wait 등)
3. leaf node를 버퍼에서 복사해온다.
4. leaf node의 key에 해당하는 record를 update 한다.
5. 로그를 작성한다.
6. leaf node를 버퍼에 넣는다.

여기서 latch로 반드시 보호해야 하는 범위는 아래와 같습니다.

update(key, value, trxid):
(buffer latch lock)
1. key에 해당하는 leaf node를 찾는다
(page latch lock)
2. record lock 관련 처리를 한다. (acquire, wait 등)
(buffer latch unlock)
3. leaf node를 버퍼에서 복사해온다.
4. leaf node의 key에 해당하는 record를 update 한다.
5. 로그를 작성한다.
6. leaf node를 버퍼에 넣는다.
(page latch unlock)

제 현재 구현에서는 3번에서 BufferController::get을, 6번에서 BufferController::put을 호출합니다. 이 두 함수에서는 buffer manager의 LRU list를 변경하는 작업을 진행합니다. 또한, 파라미터로 받은 file id와 pagenum을 이용해 buffer에서 적절한 frame을 찾거나 db 파일에서 load 합니다. 이 때 file id와 pagenum을 buffer index와 대응시키기 위해 std::unordered_map 자료구조를 사용하는데, 해당 자료구조는 thread-safe 하지 않습니다. 따라서 현재 구현상 3번과 6번을 위해서는 buffer latch가 필수적입니다.
만약 3번과 6번을 보호하기 위해 해당 위치에만 buffer latch를 걸면, 위에서 언급했던 2번 조건에 어긋나 dead latch가 발생할 수 있습니다. 따라서, 3번과 6번에서 LRU list의 변경이 일어나지 않게 하고, std::unordered_map 자료구조를 사용하지 않게 구현을 변경해야 합니다.
보고서를 적으면서 어차피 1번에서 LRU list를 업데이트하니 3번과 6번에서는 업데이트를 할 필요가 없고, page latch로 인해 eviction이 발생하지 않음이 보장되므로 1번 작업에서 알아낸 frame의 index를 이용해 바로 buffer에 접근하면 std::unordered_map 자료구조를 사용하지 않아도 된다는 점을 깨달았습니다.
위에 작성한 대로 구현을 변경한 결과 buffer latch로 보호되는 영역을 1번과 2번으로 줄일 수 있었습니다. 3번부터 6번까지의 작업을 concurrent하게 진행할 수 있게 되었으므로, 성능 향상이 있을 거라 기대할 수 있습니다.

4-2. Workload with many concurrent non-conflicting write-only transactions.

non-conflicting write-only transaction은 update 함수를 반복적으로 호출합니다. update 함수에서 성능 측면의 문제가 발생할 만한 부분은 다음과 같습니다.

1. record lock
2. log manager bottleneck
3. redo/undo
4. commit, log buffer flush

1. record lock 관련 성능의 문제

3번 문단에서 다루었던 것처럼 record lock은 성능 측면에서의 문제점이 없다고 가정하도록 하겠습니다.
또한 update 함수는 page latch에 wrtie-lock을 걸어야만 합니다. 따라서 find 함수와는 달리 하나의 page에 오직 하나의 트랜잭션만이 접근할 수 없습니다. 이 부분은 dbms의 correctness를 위한 것이므로 성능 측면에서의 문제점이 없다고 가정하도록 하겠습니다.
buffer latch는 3번 문단에서 다루었던 것처럼 꼭 필요한 부분만 보호하도록 했습니다. 따라서 buffer latch 역시 성능 측면에서의 문제점이 없다고 가정하도록 하겠습니다.
따라서 update에서 성능 측면의 문제가 발생할 만한 부분은 update log를 처리하는 부분일 가능성이 큽니다.

2. log manager bottleneck 관련 성능의 문제

과제 명세에 나온 것처럼 append와 flush를 하나의 mutex로 보호할 경우, log manager가 bottleneck으로 동작할 가능성이 있습니다.
그래서 저는 log buffer의 element마다 각각의 mutex를 갖도록 했습니다. 이 mutex는 해당 element에 append 될 때 lock되고, append가 종료됐을 때 unlock 됩니다.
append는 atomic 하게 증가하는 buffer_tail으로 element를 정합니다. 따라서 append 함수끼리는 element mutex가 중복되지 않으므로, 많은 append가 concurrent하게 동작할 수 있습니다.
그리고 flush할 때는, flush의 대상인 element 전체에 대해 mutex lock을 얻습니다. element mutex는 append가 종료됐을 때 unlock되므로, 이는 element들의 작성이 완료될 때까지 flush가 대기한다는 것을 의미합니다. 이 방법으로 작성이 완료되지 않은 log가 flush 되는 것을 방지합니다.
이 방식은 Log Buffer 전체를 하나의 mutex로 보호하는 방법보다 효율적입니다. 그 이유는 다음과 같습니다.

log가 buffer에 작성되는 시간을 T, mutex로 인한 overhead 시간을 V, buffer의 log 하나를 stable storage로 옮기는 데 걸리는 시간을 S라고 하겠습니다.
N개의 log가 동시에 작성된 다음 flush 된다 가정했을 때, 두 경우의 작업 소요 시간을 비교해 보면 다음과 같습니다.

1. Log Buffer 전체를 하나의 mutex로 보호할 경우
    append 소요 시간: T * N + V * N = (T + V) * N
        N개의 log를 buffer에 순차적으로 작성: T * N
    N번의 append마다 Log Buffer 전체를 mutex로 보호: V * N
        flush 소요 시간: S * N + V
        N개의 log를 stable storage에 순차적으로 작성: S * N
        flush 함수 전체를 Log Buffer의 mutex로 보호: V
2. 제 구현 방식대로 할 경우
    append 소요 시간: 2 * N * V + T
        N개의 log를 buffer에 동시에 작성: T
        N번의 append 마다 함수와 array의 특정 element를 mutex로 보호: 2 * N * V
    flush 소요 시간: S * N + (N + 1)V
    N개의 log를 stable storage에 순차적으로 작성: S * N
        flush 함수 전체를 flush 함수 전용 mutex로 보호: V
        N개의 element를 mutex로 보호: V * N

총 소요 시간을 비교해 보면 다음과 같습니다.
방법1 시간 - 방법2 시간 =
    TN + VN + SN + V - 2NV - T - SN - VN - V
    TN - 2NV - T
방법1 시간 > 방법 2 시간이기 위해서는...
    방법1 시간 - 방법2 시간 > 0
    TN - 2NV - T > 0
    T(N - 1) > 2NV

따라서 T(N-1)이 2NV보다 크다면 제 방법이 기존 방법보다 효율적이게 됩니다.
N이 큰 숫자라 가정하면, N == N-1이라 할 수 있고, 따라서 T > 2V, log가 buffer에 작성되는 시간이 mutex로 인한 overhead 시간보다 2배 이상으로 길다면 제 방식이 더 효율적이게 된다고 볼 수 있습니다.
저는 최대 296 byte를 덮어 씌우는 연산의 수행 시간(T)가 mutex 하나를 lock, unlock 하는 시간(V)보다는 훨씬 클 거라 생각해서 제 방식으로 구현했습니다. gitlab에 업로드 되어 있는 project6 코드는 제 방식대로 구현되어 있습니다.
하지만 만약 T < 2V라면, buffer_latch를 mutex에서 atomic한 bool 타입으로 바꾸고, spinlock을 이용해 V를 줄이는 방법을 사용해 볼 수 있습니다. 만약 이래도 T < 2V 라면 Log Buffer 전체를 하나의 mutex로 보호하는 방법이 제 방법보다 더 효율적일 것입니다. (아직 프로파일러를 이용해 검증해 보지는 않았습니다)

위와 같이 구현한 덕분에 여러 append가 concurrent하게 동작하게 했습니다. 하지만 여기에 flush가 추가된다면 더는 concurrent하지 않게 됩니다. 왜냐하면 flush가 동작하는 동안 새로 추가되는 append는 flush가 끝날 때까지 대기해야 하고, 하나의 flush만이 동작할 수 있기 때문입니다.
flush는 stable storage와 연관된 연산이니 비용이 매우 큽니다. 만약 여러 개의 flush가 concurrent하게 동작하게 만든다면 log buffer로 인한 병목을 줄일 수 있겠지만, 이는 로그가 lsn에 대해 오름차순으로 로그 파일에 저장되어야 하기 때문에 불가능합니다.
하지만 flush와 append는 concurrent하게 동작할 수 있습니다. flush는 현재까지 buffer에 존재하는 log들을 stable storage로 옮기는 함수이므로, flush 이후에 append가 일어나는 것이 가능해야 합니다. 하지만 현재 제 구현에서는 불가능합니다. 그 이유는 다음과 같습니다.
log buffer가 할당 받은 전체 메모리를 barray, 그 중 buffer_head와 buffer_tail 범위로 정의되는 공간을 buffer라고 하겠습니다.

1. buffer에는 log가 lsn 기준 오름차순으로 정렬되어 있어야 한다. 왜냐하면 buffer를 앞부터 순회하며 stable storage에 기록하기 때문이다.
2. buffer는 연속된 메모리에 위치해야 한다. 왜냐하면 순회를 해야 하기 때문이다.
3. barray는 단순 array 자료구조 이므로, flush가 끝나면 buffer_head와 buffer_tail을 다시 0번부터 시작해야 한다. 만약 이러지 않는다면 언젠가 buffer가 barray의 끝 부분을 넘어가게 된다.

barray가 [0, L] 범위일 때, buffer가 [S, E] (S <= E <= L) 범위라고 가정하겠습니다. 만약 이 상황에서 flush 함수가 호출된다면, [S, E] 영역을 flush하기 시작할 것입니다. flush가 종료되면 S와 E를 0으로 만듭니다.
그런데 만약 flush 중에 append가 가능하자면, 위에서 언급했던 조건들 때문에 E+ 1에 append가 되어야 합니다. 즉 flush 중에 append가 발생한다면, flush 이후의 buffer가 [E+ 1, E+ ??]가 됩니다. S와 E가 계속 증가할 수밖에 없고, 언젠가 S와 E가 L을 초과하게 될 수밖에 없습니다. 따라서 지금 구현에서는 flush와 append가 concurrent하게 동작하는 것이 불가능합니다.
물론 S와 E가 L을 초과했을 때만, 모든 append를 잠시 멈추고 flush를 진행해 buffer_head와 buffer_tail을 0으로 초기화하게 하는 방법도 있습니다. 하지만 이 방법은 [0, L] 범위의 barray 전체를 효율적으로 사용하지 못하고 flush를 진행해야 한다는 단점이 있습니다.
그래서 barray에 buffer를 circle queue 형태로 위치시키도록 코드를 수정하면 어떨까 하는 생각이 들었습니다. append는 flush와 상관없이 circle queue에 계속해서 log를 추가합니다. flush는 circle queue에서 head부터 호출 당시의 last_lsn인 last_lsn_capture까지를 stable storage에 flush합니다.
barray가 가득 찰 경우 append는 공간이 생기기를 대기해야 합니다. 따라서 매 append 마다 barray.size < buffer.size+ 1인지 확인해야 합니다. 만약 buffer가 barray를 가득 채웠다면, 현재 flush 중인 스레드의 존재 여부에 따라 2가지 정책을 실행할 수 있습니다.

1. 만약 buffer가 barray를 가득 채웠을 때 flush 중이라면, flush가 끝남과 동시에 buffer.size가 줄어드니 대기하던 append들이 다시 작업을 시작할 수 있습니다.
2. 만약 buffer가 barray를 가득 채웠을 때 flush 중이지 않다면, 이 상황을 유발한 append 함수에서 전체 buffer에 대해 flush를 호출하면 됩니다. 이러면 1번 상황과 같이 다른 append들은 전체 buffer가 flush 되기까지 대기하다, flush가 끝나면 append합니다.

circle queue를 이용하는 방법이 기존의 방법보다 더 효율적인 이유는 다음과 같습니다.

barray의 크기를 B, mutex로 인한 overhead 시간을 V, buffer의 log 하나를 stable storage로 옮기는 데 걸리는 시간을 S라고 하겠습니다.

기존 방법
append 소요 시간: 2 * N * V + T
flush 소요 시간: S * N + (N + 1)V
총 소요 시간: 3NV + T + SN + V
circle queue를 사용하는 방법
append 소요 시간: 2 * N * V + T + aN
circle queue로 인한 오버헤드를 a라 하면, 총 N번의 오버헤드가 존재
flush 소요 시간: S * N + (N + 1)V + b
circle queue로 인한 오버헤드를 b라 하면, 총 1번의 오버헤드가 존재
총 소요 시간: SN + NV + V + b (flush 소요시간과 같음)
S는 stable storage에 접근하는 작업의 시간이므로, flush 소요 시간이 매우 큼
append와 flush가 동시에 동작하므로 append 시간은 무시할 수 있음

두 방법을 비교해 보면, 2NV > b 일때 circle queue를 사용하는 방법이 더 효율적입니다.
flush 함수에서는 circle queue의 처음부터 끝가지를 순회하는데, 이 경우 circle queue로 인한 overhead는 사실상 0입니다. circle queue를 array로 구현했기 때문에, array를 순회하는 것과 차이가 없기 때문입니다. 물론 array의 마지막 element에서 처음 element로 이동하는 경우 몇 번의 캐시 미스 등의 오버헤드가 있을 수 있지만, 2NV보다는 작을 거라 추측할 수 있습니다.

요약하자면, log를 append 할 때 log manager 전체에 latch를 걸지 않고 단계를 좀더 세분화해 append가 concurrent하게 동작하도록 구현해 성능을 향상을 기대할 수 있게 했습니다. 그리고 제 dbms에 구현하지는 않았지만, flush와 append가 concurrent하게 동작할 수 있는 방법을 제시했고, 더 효율적임을 보였습니다.

3. redo/undo 관련 성능의 문제

많은 non-conflicting write-only transaction이 실행될 때, redo/undo 역시 약간의 비효율성이 있습니다. 현재 제 구현에서는 recovery를 physical logging으로 진행하고 있고, 아무런 최적화 없이 redo를 진행하고 있습니다.
극단적인 상황을 가정하겠습니다. 많은 transaction이 key=1인 레코드에 접근해 N번의 update 연산을 했다고 가정하겠습니다. 이를 현재 구현으로 recovery 하면 N번의 redo, N번의 undo가 필요합니다.
여러 번의 update는 한번의 update로 변환될 수 있습니다. 즉, 하나의 레코드에 대해 2N번의 page update가 수행되는 것은 비효율적입니다. 만약 redo, undo의 변경 사항을 추적해 한번의 page update만 일어나게 한다면 더욱 효율적일 것입니다.

4. commit, log buffer flush 관련 성능의 문제

많은 non-conflicting write-only transaction이 실행된다면, transaction들이 많이 commit되게 됩니다. 현재 제 구현에서는 transaction이 commit 될 때마다, log buffer을 stable storage로 flush합니다. 즉, 현재 구현에서는 log buffer의 flush가 최소한 transaction의 개수만큼 실행된다는 의미입니다.
log buffer를 flush하면, log buffer 속의 log를 stable storage로 flush 한 이후에 fsync까지 합니다. 이 과정은 비용이 매우 높은 작업입니다. 많은 transaction이 실행될 때, flush는 성능 문제를 일으킬 수 있습니다.
만약 non-conflicting write-only transaction가 많다면, commit을 바로 처리하는 것보다, commit queue를 만들어 여러 commit을 묶어서 처리하는 것이 성능에 유리할 수도 있습니다.
commit 요청이 들어오면 바로 log buffer를 flush해 commit을 반영하는 게 아니라, commit 요청을 queue에 넣고 대기시킵니다. (이 때 commit을 요청한 트랜잭션 역시 대기합니다)
이제 적절한 주기로 commit queue에 저장된 commit들을 동시에 처리하고, 한 번의 flush를 진행합니다. 이렇게 구현하면 transaction을 commit하고 잠시 대기해야 한다는 단점이 있지만, flush 호출 횟수가 줄어들며 생긴 성능 향상이 대기 시간을 넘어선다면 더 효율적이게 될 것입니다.
위 방법은 서비스 환경에 따라 효율적일 수도, 비효율적일 수도 있으니 db_init에서 commit queue 기능의 사용 여부를 옵션으로 받을 수 있도록 구현해야 할 것입니다.