Database System 2020-2

Final Report

ITE2038-11801

2019039843

유태환

Table of Contents

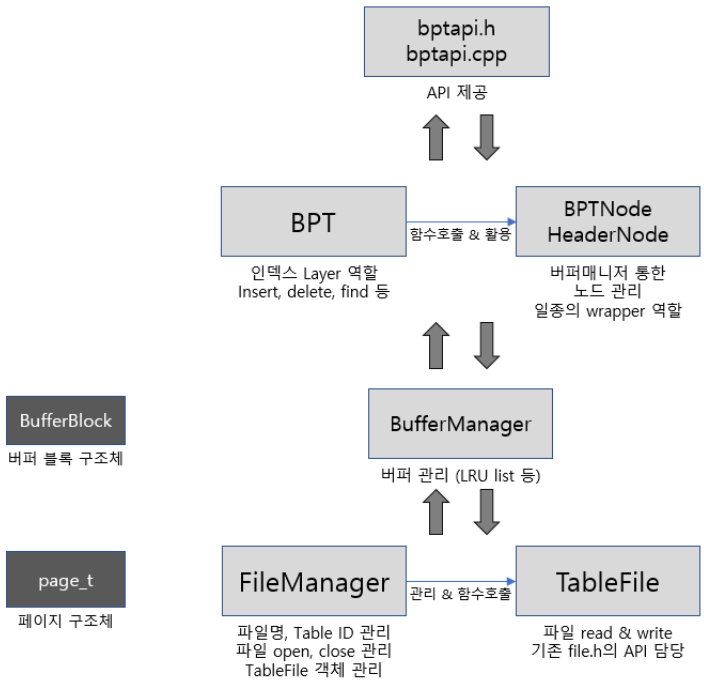
Overall Layered Architecture

1. 소스코드

이 프로젝트에서 프로젝트를 구성하는 주요 소스코드 파일들과 그에 정의된 클래스/구조체는 아래와 같다.

소스코드 파일	구조체 & 클래스
page.cpp (page.h)	record_t, child_t, page_t
file.cpp (file.h)	TableFile
file_manager.cpp (file_manager.h)	FileManager
buffer_manager.cpp (buffer.h, buffer_manager.h)	BufferManager
bpt_node.cpp (bpt_node.h, bpt_node_impl.hpp)	HeaderNode, BPTNode
bpt.cpp (bpt.h)	BPT
lock_manager.cpp (lock_manager.h, lock.h)	lock_t, lock_list_t, LockManager
trx_manager.cpp (trx_manager.h)	TrxLog, trx_t, TransactionManager
log_manager.cpp (log_manager.h, log.h)	CommonLog, UpdateData, CompensateData, LogBufferFrame, LogManager

2. 그림으로 나타낸 구조



Concurrency Control과 Crash Recovery를 담당하는 부분인 lock_manager.cpp(.h), trx_manager.cpp(.h), log_manager.cpp(.h) 등을 제외한다면, 해당 프로젝트의 layer architecture을 그림으로 나타낸다면 위와 같다.

여기에 **Transanction Manager**와 **Lock Manager**가 추가되어 트랜잭션 별로 Index Layer의 Update와 Find를 수행할 수 있도록 하고, Concurrent한 transaction들을 Control하기 위해서 Update와 Find를 수행할 때 Lock을 먼저 획득하도록 하는 작업을 하도록 하였다. 이러한 과정은 Index Layer의 함수들에서 Transaction ID를 파라미터로 받아 Transaction Manager와 Lock Manager들의 함수들을 호출하도록 하여 구현하였다.

3. 자세한 구조

A. 개요

Layer	대응하는 매니저 클래스
Index Management	BPT
Buffer Management	BufferManager
Disk Space Management	FileManager

현재 구현된 프로젝트에서는 Query Optimization과 Relation Layer를 제외한 DBMS의 Index Layer와 그 이하 레이어가 구현되어있다. 각 레이어마다 담당하는 Manager 클래스를 두었으며, BufferManager와 FileManager 클래스의 인스턴스는 전체에 하나만 존재할 수 있도록 싱글턴 패턴으로 구현하였다.

기본적으로 이에 해당하는 Manager 클래스들은 그 자신에 해당하는 Layer에 맞닿아있지 않은 Layer에 접근(해당 Layer의 Manager 클래스의 함수를 호출하는 등)하지 않도록 하였다.

Concurrency Control & Recovery	대응하는 매니저 클래스
Concurrency Control	LockManager, TransactionManager
Recovery	LogManager

이외에 Concurrency Control과 Crash Recovery를 담당하는 싱글턴 Manager 클래스들을 두었다.

B. FileManager

역할 및 타 레이어와의 관계

FileManager은 TableFile 객체들을 관리하는 클래스이다.

FileManager의 함수들을 호출하는 레이어는 기본적으로 바로 윗 레이어인 Buffer Management Layer의 BufferManager가 되며, 그 요청에 따라 여러 파일들을 열고, 이 파일들에 페이지를 읽고 쓰는 역할을 한다.

관련 구조체 & 클래스

TableFile은 file descriptor을 멤버 변수로 가지면서, 해당 file descriptor와 system call을 활용해서 파일에 페이지 단위로 read, wirte를 수행하는 함수들을 제공하는 클래스이다. 이 프로젝트에서는 Multi-Table을 지원하며, Table - File이 1 : 1로 대응되는 방식을 취하고 있으므로, 테이블 1개당 TableFile 객체 1개를 생성하여, 이 객체들을 FileManager에서 관리하는 방식으로 Multi-Table을 구현하였다.

작동 상세

상위 레이어에서 테이블을 열겠다는 요청이 들어오면 FileManager은 해당하는 파일을 연 후, TableFile 객체를 생성하고, 이에 Table ID를 부여하여 다시 상위 레이어로 돌려 보내게 된다. 이후에는 상위 레이어로부터 페이지를 읽거나 쓰고자 하는 요청이 Table ID와 Page Number가 함께 파라미터로 전달되는데, FileManager은 해당 Table ID에 대응하는 TableFile 객체를 찾아 실제로 파일에 읽거나 쓰는 작업을 수행하게 된다.

C. BufferManager

역할 및 타 레이어와의 관계

BufferManager은 페이지를 담는 버퍼를 관리하고, Eviction 정책 등에 따라 버퍼를 관리하는 클래스이다.

BufferManager의 함수들을 호출하는 레이어는 기본적으로 바로 윗 레이어인 Index Manager 단이며, BufferManager에서는 다시 FileManager의 함수들을 호출하여 물리적 파일로부터 페이지를 가져오고, 다시 페이지를 쓰는 작업을 수행한다.

관련 구조체 & 클래스

```
struct BufferBlock {
    page_t frame_;
    uint32_t table_id_;
    pagenum_t pagenum_;
    bool is_dirty_;
    std::mutex page_lock_;
    int next_block_;
    int prev_block_;
};
```

버퍼는 Linked List 자료 구조를 채택하였고 내부적으로 **BufferBlock 구조체**의 동적 할당된 배열로 구현하였으며, 버퍼 초기화 시에 요청받은 크기로 동적할당하였다.
BufferBlock은 실제 페이지인 프레임, Table ID, Page Number 데이터를 담고 있으며, 버퍼의 올바른 작동을 위한 Page Latch와, Linked List를 구현하기 위한 previous, next pointer (c/c++의 pointer를 사용한 것이 아닌 배열에서의 index), 페이지에 반영되어야하는지 아닌지를 확인하기 위한 is dirty 플래그를 추가적으로 담고 있는 구조체이다.

작동 상세

1. **Eviction**
   Eviction 정책은 기본적으로 LRU 정책을 따른다. 상위 레이어에서 어떠한 페이지를 사용하겠다는 요청이 들어오면, 해당하는 버퍼 블록의 위치를 제일 링크드 리스트의 맨 앞으로 옮겨주고 해당 페이지에 Page Latch를 걸어준다.
   버퍼에 해당 페이지가 없으면 파일에서 읽어오되, 버퍼에 자리가 없다면 LRU에 따라 링크드 리스트에서 제일 뒤에 있는 버퍼 블록을 먼저 Eviction 해주어야 한다. 이때, Page Latch를 먼저 획득하려고 시도하여 만약 다른 트랜잭션 등에서 사용중인 페이지를 Evict하지 않도록 하며, is dirty 플래그가 true라면 해당 페이지가 버퍼에 올라와있는 동안 변경이 일어났다는 의미이므로 파일에 써준 후에 요청받은 페이지를 파

일로부터 읽어와 버퍼에 올려준다. 이 과정에서 파일에 읽고 쓰는 작업은 **모두 FileManager를 통해서 이루어진다**.

2. **Page Latch**
   Page Latch는 크게 두 가지 역할을 하고 있다고 볼 수 있다.
   첫째로 현재 사용 중인 page가 사용 도중에 evict 되지 않도록 막아주고, 둘째로 메모리, 즉 버퍼에 올라와있는 페이지에 동시에 수정을 가해지는 경우를 막는다.
   Page Latch가 걸리는 시점은 상위 레이어로부터 페이지의 사용 요청이 들어왔을 때이며, Page Latch가 풀리는 시점은 상위 레이어로부터 페이지 사용이 끝났다는 것이 알려질 때이다.

D. BPT

역할 및 타 레이어와의 관계

BPT는 Index Management Layer의 역할을 하는 클래스로, B+ Tree의 자료구조에 맞추어 record를 탐색, 삽입, 삭제, 수정 등의 작업을 한다.

BPT 클래스는 insert, delete, find, update 등의 API 단의 함수들에서 호출되어 작업을 수행하며, 이러한 작업에 따라 record를 삽입, 삭제, 탐색하는 과정 등에서 페이지를 수정하거나 노드를 삭제, 생성하는 등의 작업을 수행한다. 이 때 페이지를 가져오고 수정하는 작업 등은 BufferManager를 통하여 수행하게 된다.

관련 구조체 & 클래스

BufferManager에 접근하고자 하는 페이지를 사용 요청하고, 작업을 마친 후 사용이 끝났음을 알리는 작업 등을 수월하게 하기 위하여 **HeaderNode와 BPTNode**라는 일종의 wrapper 클래스를 정의하였다.

BPTNode는 Leaf 페이지와 Internal 페이지 등의 B+ Tree의 노드 역할을 하는 page를 담당하게 하였고, HeaderNode는 root page number, free page number 등의 정보를 담고 있는 page를 담당하게 하였다.
BPTNode의 경우에는 템플릿을 활용해서 Internal node와 Leaf node를 담당하는 경우를 구분하도록 하였다.

C++의 특성을 활용하여, 생성자가 호출되어 객체가 생성될 때 생성자의 파라미터로 받은 Table ID와 Page Number을 통해서 버퍼 매니저로부터 해당 페이지를 가져오고, 명시적으로 해당 페이지의 사용이 끝났음을 알리는 함수가 호출되거나, 소멸자가 호출될 때 BufferManager에 페이지의 사용이 끝났음을 알리도록 하였다.

또한, is dirty 플래그를 알맞게 활성화시켜주기 위해서 getter와 setter를 정의하여 데이터를 접근할 때에 이들을 통하여 읽거나 수정할 수 있게 하였다. getter에서는 그저 해당하는 데이터를 반환하였지만 setter의 경우는 파라미터로 들어온 값으로 해당 데이터를 수정하되 is dirty 플래그를 true로 설정하였다.
이에 대한 예시는 아래와 같다.

```
template <NodeType T>
pagenum_t BPTNode<T>::parent() {
    return node_->page_.node_page_.header_.parent;
}

template <NodeType T>
void BPTNode<T>::parent(const pagenum_t parent_n) {
    node_->page_.node_page_.header_.parent = parent_n;
    is_changed_ = true;
}
```

이처럼 **실제로 BufferManager에 함수를 호출하고 데이터를 주고받는 역할은 BPTNode와 HeaderNode에서 담당**하였으며, BPT 클래스는 이 둘을 활용하여 BufferManager와 상호작용하는 방식으로 구현하였다. 즉, Index Layer과 Buffer Management Layer 사이에서 주고받는 데이터의 wrapper 역할을 하였으며, BPT 클래스에서는 Page Latch나 is dirty 플래그 등에 신경쓰지 않아도 되도록 하는 역할을 부여하였다. 이를 통해 Buffer Manager가 크게 수정되는 일이 생긴다고 하더라도 웬만해서는 **실제 BPT 클래스에서 상관하지 않을 수 있도록** 하는 것이 설계 목표였다.

작동 상세

BPT에서 담당하는 연산에는 크게 Insert, Delete, Find, Update가 있으며, 각각에 맞추어서 페이지를 탐색하거나, 페이지에 수정을 가하는 등의 역할을 수행하도록 하였다.

이러한 연산들은 기본적으로 일반적인 B+ Tree에서와 같이 작동하도록 구현되었으나, Delete의 경우 Merge에 의한 트리의 변형을 최소화하기 위하여 Delayed Merge를 통해서 구현하였다. 이렇게 페이지에 접근할 때에는 BPTNode, 또는 HeaderNode를 통해서만 접근할 수 있도록 하였다.

E. API

결과적으로 제공하는 API는 아래와 같다.

```
int init_db(int num_buf, int flag, int log_num, char* log_path, char*
logmsg_path);

int shutdown_db();

int open_table(char* pathname);

int close_table(int table_id);

int db_insert(int table_id, int64_t key, char* value);

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

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

int db_delete(int table_id, int64_t key);

int trx_begin(void);

int trx_commit(int trx_id);

int trx_abort(int trx_id);
```

이 중에서 Transaction과 관련되어 TransactionManager의 함수들을 호출하는 방식으로 구현된 trx_begin, trx_commit_, trx_abort를 제외하면 모두 BPT의 함수들을 호출하는 방식으로 구현되었다.

이렇듯 API단에서 BPT의 함수들을 호출하면, BPT는 BPTNode, HeaderNode를 활용해서 BufferManager의 함수들을 호출하고, 다시 BufferMananger은 FileManager의 함수들을 호출하여 그 결과가 레이어를 다시 거슬러 올라와 API를 호출한 사용자에게 전달되는 방식으로 Layered Architecture가 구현되었다.

Concurrency Control Implementation

해당 프로젝트는 Concurrency Control 기법이 적용되어 Concurrent한 여러 transaction이 실행될 수 있게 하였다. 이러한 Concurrency Control을 위한 기본 기능은 TransactionManager와 LockManager에 구현하였다.

1. Record Lock의 구현

Conflict 한 여러 transaction이 정상적으로 작동할 수 있도록 하기 위하여 Record Lock을 구현하였다. 어떠한 record에 대해 Find, 또는 Update 하고자 하는 transaction은 해당 record에 대해 lock을 획득하려는 시도를 먼저 한 후, 해당 record에 대해서 lock을 정상적으로 획득하면 마저 Find 또는 Update를 진행할 수 있도록 하였다.

2. Transaction Mananger & Lock Manager

A. LockManager

역할 및 타 레이어와의 관계

LockMananger은 Lock Table을 통해 transaction들의 Record Lock을 관리하는 싱글턴 클래스이다.

LockMananger은 BPT에서의 요청에 따라 lock을 acquire, 또는 wait 해주고, transaction이 Commit 될 때 Strict 2PL 정책에 따라 lock release 요청이 들어오면 해당 lock을 release 하고, 이때 acquire될 수 있는 lock을 깨워주는 역할을 한다.

관련 구조체 & 클래스

```
struct lock_t {
    lock_t* prev_;
    lock_t* next_;
    lock_list_t* sentinel_;
    bool can_wake_;
    bool is_wake_;
    LockMode lock_mode_;
    lock_t* next_trx_lock_;
    lock_t* prev_trx_lock_;
    int trx_id_;
};
```

Lock을 관리하기 위하여 lock_t라는 구조체를 정의하였다.
해당 Lock이 속해있는 Transaction ID를 가지고 있도록 하였다.
Lock Table에서 Lock들을 Record 마다의 링크드 리스트로 관리하므로 이를 위해 prev, next, sentinel 포인터를 가지고 있도록 했으며, 해당 Lock이 wait 상태가 되었을 때, 이후에 깰 수 있는 지 확인하는 플래그(is_wake_와 can_wake_ 중에 하나만 있어도 되나 구현상의 실수로 현재 코드에 남아있다)를 가지게 하였다.
또한, 해당 Lock이 Shared Lock인지, Exclusive Lock인지를 담고 있는 Lock Mode 멤버변수와 Transaction Table에서 transaction 마다의 또다른 링크드 리스트로 관리하므로 이를 위해 next trx lock, prev trx lock 포인터를 가지고 있도록 하였다.

또한, Record마다의 lock_t 링크드 리스트를 관리하기 위하여 lock_list_t 구조체를 정의하였다.
Lock Manager이 가지고 있는 Lock Table은 내부적으로 <Table ID, Key> Pair를 키로 가지고, 이 lock_list_t를 값으로 가지는 해시 맵으로 구현되어있다. lock_list_t 내에 std::condition_variable 멤버를 가지게 하여, 후에 std::condition_variable::notify_all()을 통해 wait 상태의 lock을 깨울 때에 원하는 Record의 링크드 리스트 내에서만 notify 될 수 있게 하였다.

이러한 lock_list_t의 변경 작업을 보호하기 위해서 **lock table latch**를 걸어주게 된다

B. TransactionManager

역할 및 타 레이어와의 관계

TransactionMananger은 말 그대로 트랜잭션을 관리해주는 Mananger 클래스로, 싱글턴 패턴을 적용하여 구현하였다.

API로 제공된 함수가 호출되어 Transaction Begin 요청이 들어오면 새로운 Transaction ID를 부여하여 반환해주고, Commit 요청이 들어오면 Strict 2PL 정책에 따라 해당 transaction이 가지고 있는 lock을 모두 release하는 역할을 하며, Lock Mananger에 의해 새로운 lock이 추가 요청 때마다 Wait For Graph를 확인하여 transaction 간의 데드락을 확인 후, 데드락이 발생하였다면 Abort 및 Rollback 작업을 해주도록 하였다.

관련 구조체 & 클래스

```
struct trx_t {
    int trx_id_;
    int64_t prev_lsn_;

    lock_t* head_;
    lock_t* tail_;

    std::vector<TrxLog> log_;
};
```

Transaction을 관리하기 위하여 trx_t라는 구조체를 정의하였다.
Transaction ID와 해당 transaction이 가지고 있는 Lock들을 링크드리스트 형태로 관리하기 위해서 head와 tail을 가지고 있도록 하였다. 이외에 Crash Recovery 구현을 위한 last LSN과 Abort시의 Rollback 및 Crash Recovery를 위한 로그 기록 또한 가지고 있도록 하였다.

작동 상세

새로운 transaction을 시작하겠다는 요청을 받으면 Transaction Manager은 새 Transaction ID를 부여해 반환해주고, trx_t 객체를 생성해 Transaction Table에 추가해준다. 또한 인접 행렬로 구현한 Wait For Graph에 새로운 transaction에 해당하는 행과 열을 추가해준다.

Transaction Commit 요청을 받으면 LockManager의 함수를 호출하여 해당 트랜잭션에 속한 Lock을 모두 Release하고, 위에서와는 반대로 trx_t 객체를 Transaction Table에서 삭제해주고, Wait For Graph 인접 행렬에서도 해당 행, 열을 삭제한다.

Transaction Abort 요청을 받으면 trx_t에 저장되어있는 로그대로 먼저 Rollback해주는 작업을 한다. Rollback이 완료되면 위의 Commit과 마찬가지로 transaction 데이터를 삭제해준다.

LockMananger에서 lock acquire 작업 시에 TransactionManager로 Lock 추가 요청을 하게 된다. 이때 TransactionManager은 파라미터로 받은 Lock의 wait 상태를 받아 Wait For Graph를 갱신하고 DFS를 통해 사이클을 찾게 된다.
사이클을 찾으면 Deadlock이 감지된 것으로 판단하여 Abort를 진행한다.

이러한 작업 중에 TransactionManager 내부의 멤버가 변경될 일이 생기면 이를 보호하기 위해 **transaction table latch**를 걸어주게 된다.

3. BPT와 Concurrency Control

BPT에서 Find 및 Update를 진행할 때에 먼저 Record Lock을 획득하는 시도를 하여 같은 Record에 접근하는 서로 다른 두 transaction이 올바르게 작동하도록 한다.

Find의 경우에는 Shared Lock 타입의 Lock을 acquire하고자 Lock Manager에 요청하게 되고, Udate의 경우에는 Exclusive Lock 타입의 Lock을 요청하게 된다. 이후에 정상적으로 Lock이 획득되었다면 하고자 한 작업을 진행하고, 획득하지 못한 경우 즉, Deadlock 등의 이유로 Abort가 발생한 경우에는 Abort 되었다는 의미의 return을 해주게 된다.

4. BufferManager와 Concurrency Control

BPTNode(또는 HeaderNode)에서 BufferManager에 페이지 사용을 요청할 때 위의 Overall Layered Architecture 파트에서 설명했던 것과 같이 버퍼에서 Eviction이 발생할 수도 있고, LRU 정책에 따라 링크드 리스트에서의 수정이 발생할 수도 있다. 이러한 상황은 동시에 여러 transaction이 접근할 경우 잘못된 수정이나 결과가 발생할 수 있으므로 보호해주어야 한다. 따라서 이를 Buffer Manager Latch를 걸어줌으로써 해결하였다.

또한, 역시 위에서 설명했던 것과 같이 페이지 사용요청이 들어왔을 때 해당 Page Latch 역시 여러 이유에 의해 걸어주게 되는데 그 이유 중의 하나가 여러 transaction이 동시에 해당 버퍼 프레임에 수정을 가하는 경우 잘못된 결과가 발생할 수 있으므로 이를 보호하기 위함이다.

Crash-Recovery Implementation

현재 최종적인 구현 상황으로는 Transaction Begin, Transaction Commit, Transaction Rollback, Update, Compensate 등 상황에 맞게 각 타입의 로그를 발급해주도록 하였고, 이러한 로그를 담는 로그 버퍼와 로그 파일을 읽고 쓰는 등의 관리를 하는 로그 매니저를 구현하는 것까지 완성하였다. 아래는 이에 대한 설명이다.

1. Log의 구현

A. Log 블록

로그 블록을 세가지로 나누자면 transaction이 Begin, Commit, Rollback시에 발급해주는 로그, Update 시에 발급해주는 로그, Rollback, 또는 Undo 시에 발급해주는 Compensate 로그가 있다. 이때 공통적으로 포함되는 데이터를 CommonLog 구조체로 묶고, UpdateLog와 CompensateLog에 필요한 추가적인 데이터를 따로 UpdateData와 CompensateData 구조체로 묶어서 관리하였다.

```
struct CommonLog {
    uint32_t log_size_;
    int64_t lsn_;
    int64_t prev_lsn_;
    int32_t trx_id_;
    LogType type_;
};
struct UpdateData {
    uint32_t table_id_;
    pagenum_t pagenum_;
    uint32_t offset_;
    uint32_t data_length_;
    char old_image_[120];
    char new_image_[120];
};
```

CompensateData는 UpdateData에 Next Undo LSN이 포함된 것과 같다.

B. Log File 구조

Log File은 DB Initalize 시에 읽기 시작할 LSN을 저장하는 LSN offset이 담긴 8byte의 header가 존재한다. 다만 LSN은 이 8byte를 무시하고, 0부터 시작하게 된다. DB가 정상적으로 shutdown되거나 복구되면, 이 offset 값이 log file의 맨 마지막(정확히는 Header 만큼의 byte 수를 빼야함)으로 바뀌게 된다.

B. LogManager

역할 및 타 레이어와의 관계

LogManager은 BPT와 같은 Index Layer, 또는 Transaction Manager에서 Begin, Commit, Rollback시에 로그를 생성해 LogManager에 파라미터로 넘기면, 이를 받아서 LSN을 부여하고 Log Buffer에 넣게 된다. 이 LSN을 return 하면 LogManager을 호출한 측에서 이를 다시 page sequence number나 transaction의 last LSN에 반영하게 된다.

또한, Commit, Rollback(TransactionManager), 또는 Page Buffer의 eviction 시(BufferManager)에 Log를 모두 파일에 반영할 수 있도록 Log Buffer를 모두 Log File에 쓰도록 하는 force 함수를 제공한다.

관련 구조체 & 클래스

```
struct LogBufferFrame {
    CommonLog log_main_;
    UpdateData update_log_;
    CompensateData compensate_log_;
};
```

Log Buffer의 프레임은 위와같이 정의하였다. Begin, Commit, Rollback 로그는 log_main_만 사용하고, Update 로그는 log_main_, update_log_, Compensate 로그는 log_main_, compensate_log_ 두 가지를 사용하게 된다.

작동 상세

LogManager가 initialize 될 때 parameter로 들어온 파일명을 열고, 헤더를 읽은 후에 헤더에 있는 값으로 현재 LSN을 설정해준다.

LogManager에 로그 발급 요청을 하게 되면, LSN 값을 부여해주고, 파라미터로 들어온 로그 데이터 값과 호출된 함수를 통해서 타입을 구분하고, 타입에 따라 다음에 들어오는 로그에 부여해 줄 LogManager 멤버 LSN 값에 더해준다. 예를 들어 Update Log 블록의 경우 그 크기가 288byte이므로 해당 크기만큼 현재 LogManager 멤버 LSN값에 288을 더해준다.

로그 버퍼에 있는 모든 로그를 파일에 써주는 force 함수를 제공하는데, 로그 블록을 파일에 쓸 때는 먼저 공통적인 CommonLog를 써준 후에 타입에 따라서 UpdateData나 CompensateData를 추가적으로 써주는 방식으로 구현하였다.

2. BPT와 Crash-Recovery

BPT에서 Update 수행할 때에 LogManager의 함수를 호출하여 Update 로그를 발급하고 반환받은 LSN 값으로 해당 트랜잭션의 last LSN과 해당 Page의 page sequence number에 적용해주게 된다.

3. BufferManager와 Crash-Recovery

BufferManager에서 eviction이 발생할 때, Write Ahead Logging 을 수행해주기 위해서 eviction이 발생한 버퍼 블록의 페이지가 수정된 내역이 로그 버퍼에 있는지 확인하고, 만약 있다면 LogManager의 함수를 호출하여 로그 버퍼를 모두 파일에 반영해주도록 하였다.

4. TransactionManager와 Crash-Recovery

Transaction이 Begin, Commit 될 때에 각 로그를 발급하고, 만약 Abort가 발생한다면, Rollback을 수행해주면서 이에 맞추어 Compensate 로그들을 모두 발급해주도록 하였다. 이때에도 ARIES 알고리즘의 Undo와 비슷하게 구현하였으며 Next Undo Sequece Number 또한 부여해주었다. Rollback이 모두 수행된 이후에는 Rollback 로그를 발급하였다. Commit, Rollback 로그 발급 이후에는 Durability 유지를 위하여 곧바로 LogManager의 force 함수를 호출하였다.

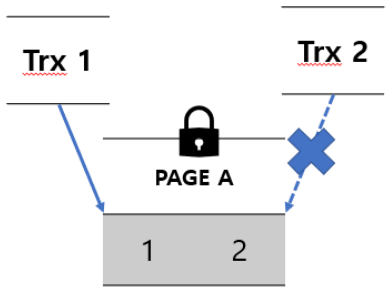
In-depth Analysis

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

A-1. 문제점 1

먼저 생각해 볼 수 있는 문제는 매우 많은 수의 non-conflict 한 read-only transaction 들이 있을 때, 이 중에 여러 Transaction이 동시에 한 페이지에 접근하는 경우이다. read-only transaction 들이라고 하였으니, read 작업인 find 연산을 한 page에 있는 여러 record에 동시에 여러 transaction들이 접근하려고 한다고 생각해보자.

아래 그림이 그 예시가 될 수 있다.



Transaction 1이 Page A의 record 1에 접근하기 위해 먼저 Page A의 Page Latch를 걸어주게 된다.
동시(또는 살짝 늦게)에 Transaction 2가 Page A의 record 2에 접근하고 싶어서 Page A의 Page Latch를 얻고자 하지만 이미 Transaction 1이 걸어주었기 때문에 Transaction 1이 풀어주기 전까지는 접근할 수 없어 기다려야 한다.

만약 매우 많은 수의 non-conflict 한 read-only transaction들이 밀려들어 한 페이지에 상당한 수의 transaction이 접근하려고 한다면 Page Latch를 순차적으로 획득하려고 시도하는 데에 시간이 들 것이라는 점을 생각해 볼 수 있을 것이다.

A-2. 디자인 1

1-A-1에서 제기한 문제점을 해결하는 디자인으로는 read 작업을 수행하고자 하는 transaction이 record lock을 획득하고자 할 때는 shared lock을 획득하고자 한 것처럼 Page Latch를 잡고자 할 때 Shared Latch를 잡게 하는 것이 있다. 즉, read-only transaction 끼리는 동시에 page latch를 잡을 수 있도록 하는 것이다.

이를 위해서는 std::shared_mutex를 사용하면 될 것이다.
https://en.cppreference.com/w/cpp/thread/shared_mutex
위의 문서에 따르면 shared_mutex를 latch로 두면 한 쓰레드가 lock()을 통해서 exclusive latch를 걸어준 상황에서는 다른 쓰레드가 latch를 잡을 수 없고, 한 쓰레드가 lock_shared()를 통해서 shared latch를 걸어준 상황에서는 다를 쓰레드가 lock_shared()를 통해서 동시에 latch를 걸어줄 수 있으나 lock()을 통해서 latch를 잡는 것이 불가능하다.

현재 프로젝트에서 Page Latch는 아래와 같이 std::mutex로 선언되어있다.

```
std::mutex page_lock_;
```

이를 아래와 같이 수정하고,

```
std::shared_mutex page_lock_;
```

read(이 프로젝트에서는 Find) 작업을 수행하고자 할 때는 page_lock_.lock_shared()를 통해 Page Latch를 잡고, write(이 프로젝트에서는 Update) 작업을 수행하고자 할 때는 page_lock_.lock()을 통해 Page Latch를 잡는다면, read 작업을 하고자 하는 transaction들은 같은 페이지에 대해서 동시에 Page Latch를 잡을 수 있어 병렬적인 처리가 가능하다. 따라서 성능 면에서의 이득을 볼 수 있을 것이다.

이렇게 하여도 괜찮은 이유는 Page Latch를 잡는 목적을 보면 알 수 있다.
먼저 사용중인 페이지가 eviction 되지 않도록 하기 위한 목적이 있는데, 이는 버퍼 매니저에서 eviction 이전에

exclusive하게 latch를 잡고자 하는 시도를 먼저 한다면, 동시에 여러 transaction이 shared latch를 잡는다고 해도 잘못될 것이 없으므로 괜찮다.
또한, 동시에 여러 transaction이 같은 페이지에 수정을 가해 문제가 생기는 경우를 방지하기 위한 목적이 있는데, shared latch를 동시에 잡을 수 있는 transaction은 read 작업을 위해 접근하는 transaction이므로 문제가 없다.

B-1. 문제점 2

생각해볼 수 있는 또다른 성능 상의 문제점은 불필요한 Transaction Begin, Transaction Commit 로그를 남기는 과정에 있다. read-only transaction의 경우 DB에 변경을 가한 것이 없으므로 Transaction Begin, Transaction Commit 로그만을 남기게 되는데, 이러한 상황에서는 저 두 로그를 없앤다고 해도 후의 Recovery 작업에 영향이 없으므로 DB의 Atomicity와 Durablity에 영향을 끼치지 않는다.

하지만, 이렇게 불필요한 로그를 남기는 과정에서 수행시간이 긴 file i/o system call이 발생하고, 이러한 수행시간의 지연은 트랜잭션이 많을 수록 누적된다. 이에 따라 성능상의 문제점이 생기게 된다.

B-2. 디자인 2

1-B-1에서 제기한 문제를 해결하는 방법은 Commit(또는 Abort) 시에 해당 transaction이 read-only transaction이라고 생각된다면 Transaction Begin과 Commit 로그를 남기지 않는 것이다.

transaction이 read-only transaction 이라고 판단할 수 있는 근거는 아래의 trx_t에 있는 transaction rollback을 위한 로그 멤버이다.

```
std::vector<TrxLog> log_;
```

log_에는 write가 이루어질 때만 새로 로그가 추가되도록 구현하였으므로 만약 log_의 크기가 0이라면, 해당 transaction이 read-only라고 할 수 있다.

이렇게 readl-only인지 판단한 이후에는 Commit log를 발급하지 않아도 된다고 생각할 수 있다. 다만, 이미 Begin Log 가 파일에 반영된 이후라면 Commit Log를 발급하여야 하며, Begin Log가 아직 버퍼에 남아있다면 해당 Begin Log가 파일에 반영되지 않도록 방지해주어야한다.

이를 위해서 로그의 타입에 BEGIN_REDUNDANT를 추가한다.

만약 transaction이 read-only라고 판단된다면 먼저 log buffer에 해당 transaction의 Begin log가 남아있는지 확인하고, 남아 있는 경우에는 commit 로그를 발급하지 않고 해당 Begin log의 타입을 BEGIN_REDUNDANT로 바꾸어준다. 만약 남아있지 않는 경우에는 Commit 로그를 발급하고 평소와 같이 LogManager의 force 함수를 호출하여 파일에 반영해주도록 한다.

만약 Abort되는 경우에도 Rollback 로그를 발급하는 과정에서 위와 같이 하면 된다.

LogManager에서 로그 버퍼를 로그 파일에 쓰는 과정에서는 BEGIN_REDUNDANT 로그는 건너 뛰어야한다.

이러한 디자인이 가능한 이유는 위의 1-B-1에서 말했다시피 read-only transaction에서는 BEGIN과 COMMIT(또는 ROLLBACK) 로그는 없더라도 무관하기 때문이다.

```
void BufferManager::erase_buffer_block(int target_idx) {
    ...
    if (buffer_[target_idx].is_dirty_) {
        set_page_to_file(target_idx);
    }
    ...
}

void BufferManager::set_page_to_file(int target_idx) {
    if (LogManager::instance().isPageExistInLog(target_idx)) {
        LogManager::instance().force();
    }
    ...
}
```

또한 이것이 성능의 향상을 불러오는 이유 중의 하나는 위 코드를 보면 알 수 있듯이 현재 구현에서 read 작업만 수행된 페이지 (즉 is dirty 플래그가 false인 경우)는 페이지 버퍼에서 evict될 때 로그 버퍼의 flush(force 함수)를 유발하지 않기 때문이다.

따라서 많은 수의 non-conflict read-only transaction들이 몰려서 실행되고 있는 상황에서는 비교적 작은 페이지 버퍼를 가지고 있더라도 Begin Log가 계속해서 로그 버퍼에 남아있을 확률이 높을 것이고, 따라서 위와 같이 구현하였을 때 '*불필요한 Begin, Commit 로그가 발급되어 파일에 write되는*' 불필요한 file i/o가 감소하여 성능 향상을 노려볼 수 있을 것이다.

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

A-1. 문제점

많은 수의 non-conflicting write-only transaction들이 실행될 때의 문제는 잦은 File I/O System Call의 발생이다.

잦은 System Call이 끼치는 성능상의 문제점은 아래의 실험을 통해서 확인해 볼 수 있다.

```
void test1() {
    int fd = open("test1", O_RDWR | O_CREAT | O_SYNC, 0666);

    test_t t;

    lseek(fd, 0, SEEK_SET);
    for (int i = 0; i < 10000; i++) {
        write(fd, &t, sizeof(test_t));
        fsync(fd);
    }

    close(fd);
}
```

```
void test2() {
    int fd = open("test2" , O_RDWR | O_CREAT | O_SYNC, 0666);

    test_t t[10000];

    lseek(fd, 0, SEEK_SET);
    write(fd, t, sizeof(test_t) * 10000);
    fsync(fd);

    close(fd);
}
```

이와 같이 테스트 코드를 작성하였다. test_t는 int64_t 멤버 하나만을 갖는 구조체이며, 따라서 test1과 test2 모두 총 80KB를 파일에 쓰는 코드이다. 다만 test1에서는 10000번에 나누어 write를 수행하였고, test2에서는 한번에 write를 수행하였다.

이들을 9번씩 수행하여 실행시간을 비교해보면 아래와 같다.

test1 (ms)	test2 (ms)
2257	1
2193	0
2228	0
2120	0
2399	0
2403	0
2083	0
2048	0
2182	0

이를 통해서 같은 크기만큼 파일에 쓴다고 하더라도 File I/O System Call 횟수를 줄이는 것이 성능에 매우 큰 영향을 줄 수 있음을 알 수 있다.

그러나, 많은 수의 write-only transaction들이 실행된다면 매우 넉넉한 크기의 페이지 버퍼를 가지고 있어 eviction이 일어나지 않는다고 하더라도, 빈번한 transaction commit이 발생하여 로그 버퍼를 파일에 쓰는 File I/O가 자주 일어나게 된다.

더욱 더 심각한 문제는 현재 구현에서 LogManager의 force 함수의 경우 로그 블록(log vector element) 하나마다 write system call을 호출해주고 있다는 것이다. 이러한 빈번한 File I/O System Call이 성능 상의 문제를 가져오고 있다.

A-2. 디자인

먼저 수정해야할 것은 force 함수로, force 함수가 호출되면 vector, 즉, 로그 버퍼에 담긴 모든 로그 블록을 담을 수 있는 크기의 char 배열을 선언하고, 해당 배열에 로그 블록들을 복사한 후에 그 배열을 한꺼번에 file에 write 해주는 방식으로 수정하여야 한다. 이를 통해서 File I/O System Call 횟수를 줄일 수 있다.

또 다른 고려할 만한 사항은 Commit으로 인해 force 함수가 호출되는 횟수를 줄여보는 것이다. 많은 수의 write-only transaction이 실행되면서 잦은 Commit으로 인해 잦은 로그 버퍼 flush가 일어나는데, 이 또한 여러 transaction을 한꺼번에 commit을 함으로써 file에 write를 하는 횟수를 줄여볼 수 있을 것이다.

이와 비슷한 기능이 MySql에 구현되어있는데, 아래 링크가 MySql 서버인 Percona의 문서 중 Group Commit에 대해 설명된 내용이다.
https://www.percona.com/doc/percona-server/5.5/performance/binary_group_commit.html

이를 위해서, TransactionManager에서 Commit 요청 큐를 관리하는 식으로 구현할 수 있다.
Commit 요청이 들어오면, 해당 요청을 Commit 요청 큐에 쌓아두었다가 일정 숫자의 Commit이 쌓이면 한꺼번에 꺼내어 각 Commit 요청을 수행, 로그 발급 후 모든 Commit이 끝나면 한번만 force 함수를 호출하는 방식으로 구현해 볼 수 있을 것이다. 다만, 이런 식으로 구현하면 한 transaction이 먼저 끝나야 다른 transaction이 끝나게 되는 관계에 의하여 원하는 Commit 요청 수만큼 쌓이지 못할 수도 있다. 따라서 timeout을 도입하여 timeout이 발생하면 원하는 수만큼 쌓이지 않았더라도 쌓여있는 Commit 요청을 수행하도록 할 수 있겠다.

다만 이러한 디자인의 단점은 많은 수의 Commit 요청이 쌓여있는 상황에서 Crash가 발생한다면 많은 수의 Transaction이 loser가 되는 상황이 벌어질 수 있을 것이다.

기타 문제 및 보완점

Log Buffer Latch를 걸기 위해 기다리는 시간 또한 성능 상의 문제를 가져올 수 있다.

이를 해결하기 위해서 로그 버퍼를 크기가 정해진 char 배열로 구현한 후, 다음 Log Block이 들어오게 될 위치를 저장하는 멤버 변수를 새로 추가한다. Log Buffer에 Log Block을 추가하겠다는 요청이 들어오면, 해당 멤버 변수의 값을 변경하는 순간만을 Log Buffer Latch로 보호하더라도 다음 Log Block이 알맞은 위치에 들어올 수 있다. 이를 통해서 Log Buffer Latch가 걸려있는 순간을 최소화할 수 있을 것이다.

다만, Log Buffer에 새로운 Log Block이 들어오는(복사되는) 와중에 flush가 발생하면 안되므로, 이는 별도의 Latch로 보호하여야 할 것이다.