# Database System 2020-2
# Final Report

Class Code(ITE2038-11800)
Student number : 2017029743
Name : 배병재

Table of Contents

# Overall Layered Architecture


-dbms 계층구조-

## 1. File Mangement and Disk Space Management :

데이터베이스 파일을 관리하는 layer이다. DB파일을 disk로부터 연결해서 관리하는 layer이며, 동시에 DB파일을 File Mangere API를 통해 관리한다. 파일 내부를 pagenation을 통해 페이지로 구획한다. 우리는 이 페이지를 통해 데이터에 접근해서 read\write를 하게 된다. API는 다음과 같다.

1) struct page_t : page의 레이아웃과(구성)과 page_num을 만들어야한다. -> PAGE의 종류별로 따로 구현한다.(헤더페이지, 리프페이지, 인터널페이지)

2) pagenum_t file_alloc_page(): free페이지 리스트로부터 page를 할당한다.
-> free page가 없을 시 새로운 페이지를 생선한다.

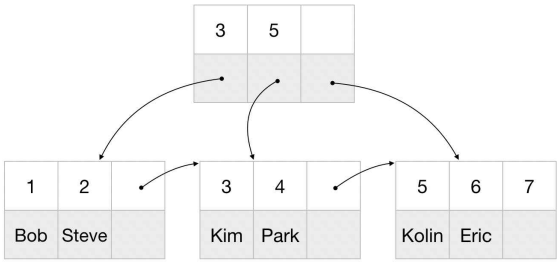
3) void file_free_page(pagenum_t pagenum): free페이지 리스트로부터 페이지를 free로바꾼다.
-> free page list의 가장 처음에 배치하도록 만든다.

4) void file_read_page(pagenum_t pagenum,page_t* dest): 페이지구조로(dest) 부터 페이지를 읽는다.
-> pagenum이라는 오프셋을 통해 파일에 원하는 위치에 페이지를 읽는다.

5) void file_write_page(pagenum_t pagenum,const page_t *src): in-memory page를 disk-page로 만든다.
-> pagenum 오프셋을 이용하여 페이지를 파일에 작성한다.

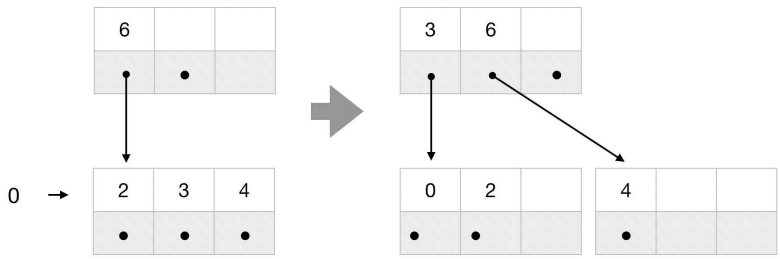
## 2. Index Management :

파일 내부에서 offset을 통해 원하는 페이지에 접근하고, 그 페이지 내부에서도 offset을 통해 원하는 key에 접근하게 된다. 이때 파일 내부에서 페이지를 관리하는 구조를 B+트리 구조로 관리하게 된다.

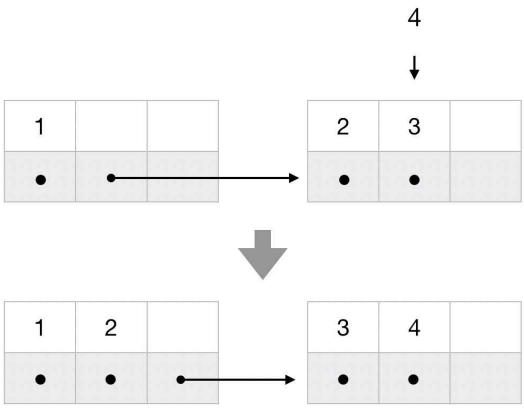



-B+트리-

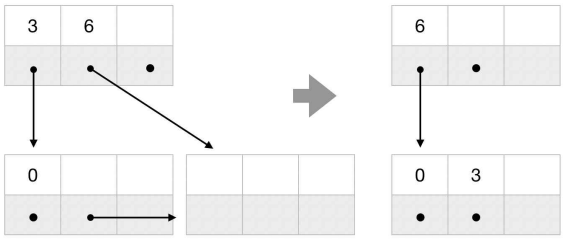
B+트리를 통해 정렬된 Key-Value Pair에 대한 효율적인 삽입, 검색, 삭제를 지원한다. 각각의 노드는 최대 노드 개수 B를 기준으로 B-1개의 키와 B개의 자식 노드를 가질 수 있다. ISAM과 다르게 내부의 키 개수와 노드 관리를 위한 split과 merge, redistribution을 활용한 balancing 작업이 있다.
insert하는 과정에서 split이 발생한다. split은 다음과 같다.




delete하는 과정에서 merge와 redistribution이 발생한다.




-redistribution-




-merge-

이 b+트리를 활용해서 index management는 데이터베이스에서
삽입,삭제,탐색을 하게 된다. 관련 API는 다음과 같다.

1) int open_table(char* pathname): Open existing data file using 'pathname' or create one if not existed. using file manager api.

- fopen을 이용해서 file을 열고, 유니크한 테이블 id를 배열로 저장

2) int db_insert(int key, char* value): insert input 'key/value(record)' to data file at the right place.

- 메모리 기반의 node를 page로 전부 바꾼다.

3) int db_find(int key,char* ret_val): Find the record conataining input 'key'and return matching 'value' or 'null' if not found.

- 메모리 기반을 페이지로 바꾼다.

4) int db_delete(int key) : Find the matching record and delete it if found.

-내부에 delayed merge구현

## 3. Buffer-Mangement :

Buffer-Manager는 파일 read\write를 바로 진행하는 것이 아닌 버퍼라는
cache를 통해 read/write를 진행함으로써 On-Disk B+ Tree 오퍼레이션을 가속화 할 수 있다.

Buffer-Manager은 요청에 따라 주어진 파일에서 Page를 버퍼에 올려 반환한다. 이 페이지 프레임은 버퍼 매니저가 종료하거나, LRU를 통한 Page evict에 의해 다른 페이지 프레임으로 교체되지 않는 이상 메모리를 유지하여 사용자가 메모리에서 연산을 진행하는 듯한 환경을 제공한다.

Buffer-Manager를 통해 Disk-Space-Manger와 File and Index Manager가 바로 연결되지 않고, Buffer-Manager를 거치게 된다. 따라서 속도 향상을 할 수 있다. 관련 API는 다음과 같다.

1). write_buf(buffer* frame):
   1-1 인자로 받은 frame에 dirty를 체크하고 dirty일 경우 file_api를 통해 write해준다.

2). get_frame(int table_id,pagenum_t page) :
   2-1 버퍼 리스트에서 table_id와 page에 해당하는 프레임을 가져온다.
   2-2 이에 해당하는 프레임을 PINNED해준다.
   2-3 만약 해당하는 페이지가 없을 경우 LRU_POLICY 정책에 따라 HEAD->PREV부터 HEAD에서 먼 페이지들을 탐색한다.
   2-4 이때 프레임이 PINNED 되어 있지 않은 것을 버리고 원하는 페이지를 읽어온다.

3) put_frame(buffer* frame, int is_dirty) :
   3-1 사용을 한 frame을 LRU-POLICY에 따라 HEAD->NEXT로 옮겨주는 작업을 한다.
   3-2 이때 UNPINNED로 바꿔주고 인자로 입력받은 DIRTY 값에 따라 is_dirty를 수정 해준다.

LRU-POLICY를 페이지 교체 정책으로 사용한다. 가장 사용한지 오래된 페이지가 evict의 대상인 정책이다. 이처럼 페이지 교체 정책에 따라 페이지를 Disk에 쓰는 주기가 달라지게 되고, 이것이 속도의 영향을 미치게 된다.

## 4. Query Optimizer :

주어진 쿼리를 Relational Algebra와 hash, sort-merge, join등에 데이터를 다룸으로써 쿼리의 연산속도를 빠르게 하는 layer이다.

쿼리 옵티마이저는 쿼리 수행을 위해 후보군이 될만한 실행 계획을 검색하고, 이 실행계획들의 비용을 비교한다. 그 중 최저비용을 갖는 실행계획을 선택하고 그 실행계획으로 쿼리를 실행해서 하위 layer를 통해 원하는 operation을 실행하게 된다.

# Concurrency Control Implementation

트랜잭션의 ACID의 속성을 가진다. 이 중 동시성 제어에 해당하는 Concurrency Control은 Consistency와 Isolation의 속성을 만족시키게 한다.

## 1. Consistency & Isolation :

Consistency은 주어진 값을 여러 TRX이 수정하고자 할 때 위협을 받으며, 이때 일어날 수 있는 문제는 크게 3가지로 나뉜다.

1) lost update:
    동시에 두 write operation이 발생하면, 하나의 write operation이 손실될 수 있다.
2) inconsistent read:
    수정될 값을 읽을 때 하나의 개체에 대해 읽어온 두 값이 다를 수 있다.
3) dirty read:
    아직 confirm 되지 않은 임시 값을 읽을 때 Undo 과정에서 값이 손실될 수 있다.

Isolation은 동시에 여러 트랜잭션이 처리될 때, 특정 트랜잭션이 다른 트랜잭션에서 변경하거나 조회하는 데이터를 볼 수 있도록 허용할지 말지를 결정하는 것.

## 2. Lock Manager :

TRX의 ACID 속성을 보장하기 위해 시스템 전체에 대한 접근을 Lock을 통해 제어하는 장치이다.

1) Accessibility

Lock은 TRX이 Database, Table, Page, Record 등 대상 개체에 접근하는 것을 제한하며, TRX이 요청한 권한과 대상 개체에 접근 중인 TRX에서 접근 안전성을 고려하여 대기 혹은 접근의 결정을 내린다.

권한은 크게 Shared와 Exclusive로 나뉘며, 개체를 수정하지 않아 다른 TRX과 접근을 공유할 수 있는 경우 Shared, 개체에 대한 추가 처리가 이루어져 다른 TRX과 접근을 공유할 수 없는 경우 Exclusive 권한을 요청하게 된다.
Shared 권한을 요청한 TRX은 서로 접근을 공유할 수 있으며, Exclusive 권한을 요청한 경우 다른 TRX이 접근할 수 없다. Lock은 이러한 권한과 접근 상태를 통해 접근 안전성을 고려한다.

2) Deadlock
서로 다른 TRX이 상대방이 소유한 개체에 대한 접근을 기다릴 때 두 TRX은 무한정 서로를 기다리게 되는데, 이를 Deadlock이라고 한다. Lock Manager은 주기적으로 혹은 매번 Lock을 처리할 때마다 TRX 사이에 Lock Cycle이 존재하는지 확인하고, 존재할 경우 Cycle을 풀기 위해 Abort 할 TRX을 지정하여 실행을 취소해야 한다.

## 3. Design :

1) Lock Structure

```
typedef struct lock_t {
        int key;
        int flag;
        pthread_cond_t cond;
        struct lock_t* next;
        struct lock_t* prev;
        struct hash_entry* sentinel;
        int lock_mode;
        int trx_id;
        struct lock_t* next_trx_ptr;
        struct lock_t* prev_trx_ptr;
}lock_t;
```

Lock의 기본적인 구조에 s,x모드의 구분을 위한 lock_mode와 트랜잭션 구분을 위한 trx_id 그리고 트랜잭션의 락을 확인하기 위한 next_trx_ptr등이 있다.

```
typedef struct hash_entry{
        int num;
        int f_trx_id;
        int check;
        int exclusive;
        int table_id;
        int64_t key;
        lock_t* head;
```

```
        lock_t* tail;
}hash_enrty;
```

레코드의 전반적인 locking을 관리하고 있다. 하나의 레코드에 하나의 트랜잭션만 접근할 시, wait을 안하게 하기 위해, int check와 acquired 된 락 중에 x모드가 있는지 체크하기 위한 exclusive등을 활용한다.

2) Transaction structure

```
typedef struct prev_record{
    int table_id;
    int64_t key;
    char value[120];
}prev_record;
typedef struct trx{
    int trx_id;
    struct trx* trx_next;
    lock_t* lock_next;
    vector<prev_record>record;
    int deadlock;
}trx;
```

트랜잭션은 생성과 즉시 TRX ID를 부여받으며, Operation 중에는 Lock Manager와 통신하며 작업에 필요한 Lock을 획득, 사용자가 trx_commit() 메소드를 호출하기 전까지 획득한 lock을 보존하는 역할을 한다. 이 과정에서 trx_commit()까지 정상 호출되면 lock을 release 하고, 그 전에 deadlock이 발생한 경우 abort 하게 된다.
이때, prev_record는 트랜잭션이 abort될 때, undo를 위한 구조체로, update를 할 때, 이전의 레코드 내용을 저장해 abort될 때, 원래대로 돌려준다.

3) Deadlock detection
Deadlock detection은 Lock의 prev 포인터와 Transaction이 가지고 있는 lock list를 통해 가능하다.
하나의 TRX을 선택하고, 소유하고 있는 lock을 순회한다. 해당 lock이 가리키는 page ID의 lock 리스트에서 그 lock이 wait하고 있을 시, 그 lock이 현재 기다리고 있는 lock 뿐만 아니라 향후 기다리게 될 lock까지 전부 체크해서 그래프를 생성해준다.이는 Graph 탐색 문제이므로 dfs를 통해 사이클을 탐색해준다.

4) Buffer Pin and page_latch
Buffer에서 사용하던 pin을 mutex인 page_latch로 수정해준다. page_latch를 잡기 위해서는 buffer_latch를 잡아야 하고, 레코드 락을 잡기전에 page_latch를 풀어줘야 데드락 방지가 가능하다. 레코드 락을 잡아주고는 다시 page_latch를 잡아준다.

5) Buffer Manager

Buffer manager에 대해서 쓰레드의 접근을 제한하기 위한 buffer_latch를 활성화 해준다. 이때, LRU_list의 수정은 버퍼가 페이지를 얻어올때, 한번 실시해준다. buffer_latch->page_latch의 구조를 지키기 위해 이러한 디자인이 필요하다. 이렇게 하지 않을 경우 데드락이 발생할 수 있어 이러한 디자인을 고려하였다. 또한 버퍼풀이 꽉차있고, 모든 버퍼페이지를 사용중이여서 원하는 페이지를 가져올 수 없을때는 기다리지 않고 NULL을 리턴하도록 핸들링하였다.

6) Strict 2-Phase-Lock

2PL 구현을 위해서 추후 TRX Commit 혹은 Abort후에 Lock을 Release 하는 방식으로 구현하였다.

# Crash-Recovery Implementation

Crash-Recovery는 트랜잭션의 특성중 Atomicity와 Durability를 만족하게 하는 방법이다. 크게 3가지 pass로 구현된다.

## 1. Naive Design

로그 파일을 만들어서 트랜잭션의 내역을 저장해서 사용하는 구조이다.

1) 변경된 페이지 구조체

```
typedef struct page_t2 {
    pagenum_t Parent_page_number;
    int is_Leaf;
    int Number_of_Keys;
    char page[104];
    pagenum_t sibling_page_number;
    record_t record[31];
   element Page_Lsn;
        -> 이것을 추가하여 페이지에 접근한 최근 lsn에 대하여 알 수 있다.
}leaf_page_t;
```

2) 일반적인 로그 레코드이다.

```
typedef struct log_record{
        int log_size;
        element lsn;
        element prev_lsn;
        int trx_id;
        int type;
}log_record;
```

3) 업데이트,clr 레코드이다.

```
typedef struct up_log_record{
        int log_size;
        element lsn;
        element prev_lsn;
        int trx_id;
        int type;
        int table_id;
        pagenum_t page_num;
        int offset;
        int data_length;
        char old_image[120];
        char new_image[120];
        element next_undo_lsn;
}up_log_record;
```

## 2. Analyze

트랜잭션을 winner와 loser 트랜잭션으로 나누는 과정이다.
트랜잭션이 commit되거나 abort 되었을 경우가 winner 트랜잭션이고
중간에 system crash로 인해 멈췄을 경우 loser 트랜잭션이다.

## 3. Redo

redo는 기본적으로 모든 트랜잭션들은 redo해준다.
간단하게 commit 된거는 commit 로그 abort된거는 rollback 로그로써 redo를 진행해준다.
모든 트랜잭션에 대해서 로그파일을 처음부터 순회하면서 redo를 해준다.
redo 작업이 끝나게 되면, 데이터베이스 상태는 장애 발생 시점의 상태와 같게 된다.
중간에 redo_crash가 일어날 경우 이전에 한번 redo가 진행되었던것은 가장 최근 page lsn을 기준으로
해당 로그의 lsn과 page lsn을 비교함으로써 판단할 수 있다.
결과적으로 page lsn이 해당 로그의 lsn보다 같거나 크면 복구가 필요치 않다.
이러한 consider_redo를 진행하게 해야한다.
consider_redo를 통해 fast recovery가 가능하게 된다.
그리고 버퍼에서 page_evict가 일어나게 되면, 그때 로그버퍼에서 로그파일로 쓰게 된다.

## 4. Undo

undo는 loser 트랜잭션들을 마치 일어나지 않은 것처럼 원래 상태로 돌려주는 것이다.
기본적으로 undo는 역방향으로 탐색하면서 undo 복구가 필요한 로그들에 대해서 undo 복구를 수행한다.
undo pass는 redo pass 이후에 진행이 된다. 최초의 undo pass는 순서대로 undo를 진행하게 된다.
undo를 수행하고 나면 해당 undo작업에 대한 clr을 발급한다. redo전용 로그로써,
undo를 하고 난 이후에 다시 undo를 하는 행위가 일어나지 않기 위해 필요하다.
clr은 이전 로그 레코드 위치를 undo로그의 이전 로그를 가리키도록 한다. 이런식으로 이전 로그를 계속 탐색한다.
근데 이때 undo_crash로 인해 undo중 fail하게 될 경우, clr을 통해 이미 undo 된 것들은 스킵하고, 새로 undo해 줄 것들을 복구한다.

# In-depth Analysis

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

현재의 디자인은 모든 요청에 대하여 locking을 하게 되고 2pl방식을 사용하기에, 대량의 non-conflictiong read-only transaction이 실행된다면, 충돌이 없는 상황에서 쓸데 없이 locking을 하게 되어 시간과 공간의 낭비가 발생할 수 있습니다. read의 처리량이 줄어들게 될 것이다. 크게 2가지 방법을 생각해볼 수 있을 것이다.

1) OCC-Optimistic Concurrency Control
낙관적 동시성 제어라고 불린다. 이것은 사용자들이 같은 데이터를 동시에 수정하지 않을 것이라고 가정한다.  따라서 데이터를 읽을때는 Lock을 설정하지 않는다. 하지만 데이터를 수정하는 시점에 다른 사용자에 의해 값이 변경되었는지 반드시 검사해야 한다. 그리고 변경되었는지 확인하는 시점부터(val) write가 끝나는 시점까지 어떠한 트랜잭션도 접근할 수 없게 된다. 즉, val-write phase는 고립되어 보장된다. 우리는 그 중 FOCC 방식을 활용한다.

기존의 레코드 단위 locking을 하던 것이 변하게 될 것이다.
3개의 phase로 나뉘게 되고 read-validation-write로 나뉜다. 먼저 수행할 관련 레코드들을 전부 읽어서 카피를 해서 따로 클라이언트의 workspace에 저장한다. 그런다음 내부의 workspace에서 말그래도 읽기만 할 수도 있고, 쓰기 작업을 할 수 있다. private workspace에서 모두 작업을 한다. 그런다음 validation 단계를 거친다.
commit 하기전에 확인하는 것인데, 이때, 이미 현재 실행중인 트랜잭션들을 대상으로 그 트

랜잭션들의 read_set과 현 트랜잭션의 write_set을 비교하여 conflicting이 일어나는지 체크해야 한다. 그런다음 문제가 없게 된다면, write를 통해 데이터베이스에 install하게 된다. 만약 충돌이 얼아나게 된다면, abort시키면 된다. 혹은 validation을 후에 다시 실행해도 된다.

하지만 이 경우 read-only 트랜잭션이므로, 충돌의 위험이 전혀없으며 validation의 과정이 필요하지 않게 된다. 왜냐하면 현 트랜잭션의 write_set이 비교의 대상이기 때문이다.

2) Read-Replica
dbms에 디자인이 아닌 외부의 서버를 이용한다. 원본 데이터베이스가 위치하는 서버를 마스터라고 하고, 그 원본을 복제한 서버를 슬레이브라고 한다. 원본 데이터베이스는 쓰기만 하게 된다. 그리고 슬레이브에 있는 데이터베이스는 read만 하게 되는데, 슬레이브에 있는 데이터베이스에는 locking하는 과정을 제거해주면 더 빠르게 read가 가능할 것이다. 원본 데이터베이스에만 쓰기 작업을 해주고, 이것을 슬레이브에 있는 데이터베이스에 복사를 해주는식으로 한다.

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

충돌이 없는 write-only 트랜잭션만이 있다는 것은 결국은 locking의 필요성이 없다는 것이다. 즉 아무런 제약없이 병렬적으로 수행할 수 있을 것이다. 이때 crash-recovery 단계에서도 마찬가지 일 것이다. redo나 undo를 할 때 모든 트랜잭션에 대해 순차적으로 로그파일을 탐색하면서 redo와 undo를 할 필요가 없게 된다. 현재의 디자인으로 진행하게 될 경우, 쓸데없는 속도의 저하가 생길 수 있다. 많은 수의 트랜잭션의 로그를 전부 순회하면서 redo와 undo를 해주는 것은 시간낭비일 것이다. 기존의 디자인대로가 아닌 redo와 undo를 병렬적으로 빠르게 할 수 있을 것이다.

analyze 단계에서 해당하는 변경된 레코드 개수 만큼 로그파일을 순회하면서 자신에게 해당하는 레코드 관련 로그들을 모두 redo해주는 작업을 한다. 즉 병렬적으로 redo 쓰기 작업을 진행하기에 훨씬 빠르게 할 수 있다. 이것에 대한 근거는 non-conflicting이기 때문이다. 그런 다음 undo를 진행할 때도 마찬가지로 loser에만 해당하는 트랜잭션들의 변경된 레코드로 병렬적으로 undo를 진행하면 된다. 이것에 근거는 마찬가지로 non-conflicting일 이다.
즉, 레코드 단위 redo 및 undo를 진행하면 된다.