

Database System 2020-2

Final Report

ITE2038-11800

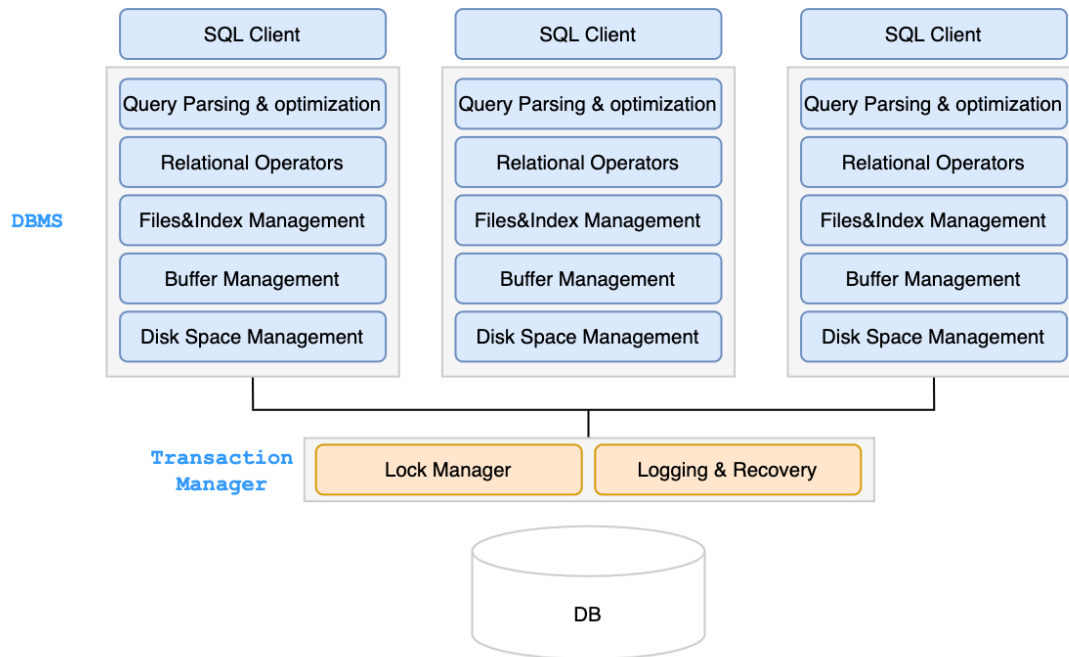
2016024902

Seryung Yoon

Table of Contents

Overall Layered Architecture	3p.
Concurrency Control Implementation	8p.
Crash-Recovery Implementation	13p.
In-depth Analysis	16p.

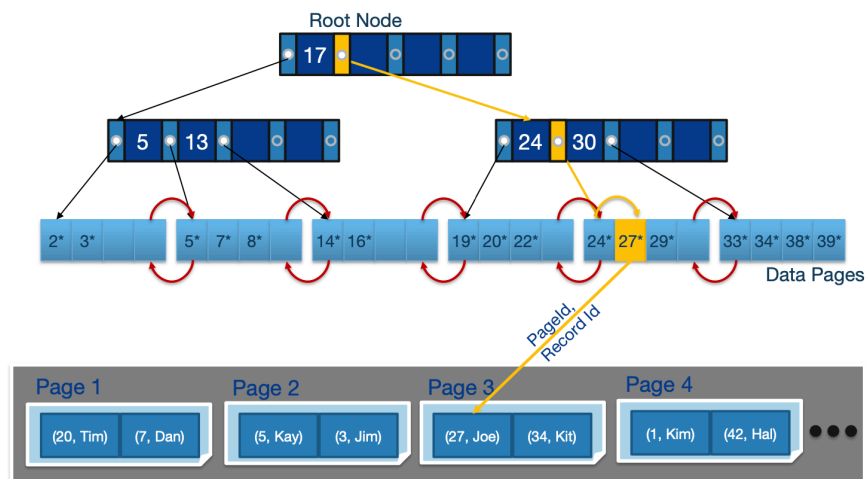
Overall Layered Architecture



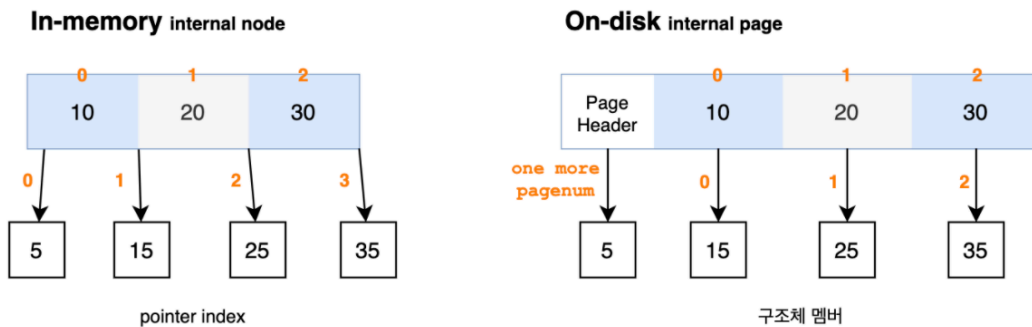
데이터베이스 관리 시스템(DBMS)이란 데이터를 효과적으로 이용할 수 있도록 보관하는 소프트웨어로, 사용자에게 파일 열기 및 데이터의 삽입과 삭제, 변경, 검색 API를 제공한다. 사용자가 API Services를 호출하면 on-Disk b+ tree 방식으로 작동하며, 페이지(4KB) 단위로 disk I/O를 수행한다.

본 DBMS에서 구현한 파트는 크게 5부분으로 이루어진다. 각 계층에서는 자신이 맡은 기능만 수행하며, 다음 단계 레이어의 일을 호출하거나 넘길 때는 API만 호출한다. 따라서 다른 계층에서 무슨 일을 하는지 알 필요가 없다.

1) Files & Index Management Layer

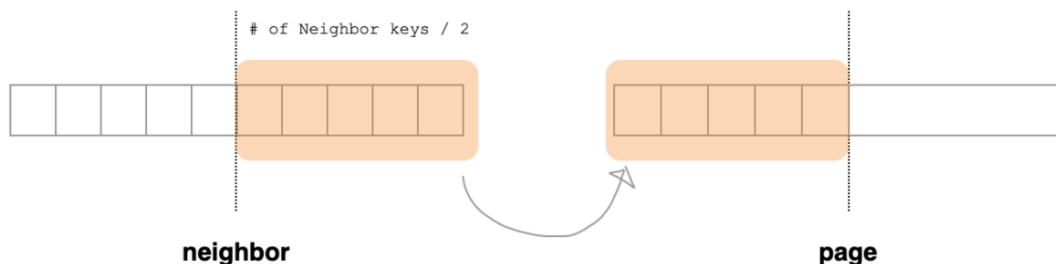


보다 효율적인 구조를 위해 b+ tree Index 방식을 차용하여 insert와 delete를 효율적으로 수행한다. B+ tree란 key에 의해 식별되는 레코드의 효율적인 삽입, 검색, 삭제를 통해 정렬된 데이터를 표현하기 위한 트리 자료구조이다. b+ tree는 data entry를 leaf에 저장하고, root와 internal page에는 어디에 특정 rid가 저장되어 있는지를 알려주는 key가 저장되어 있어 routing 역할을 한다. 또한 leaf page는 linked list 형태로 서로 연결되어 있으며, 오름차순으로 정렬되어 있어 순차적인 탐색에 매우 유리하다.



기존 in-memory b+ tree는 node 구조체를 선언하여 child node를 가리키는 데 pointer를 이용한 반면, 본 프로젝트에서 구현한 on-disk b+ tree는 page 구조체 안의 pagenum 멤버를 통해 child page에 접근하는 차이가 있다. 특히, 각 트리에서 leftmost page를 가리키는 방식에 가장 큰 차이가 있다. Pointer에서는 0번째가 가리키던 node가 leftmost node였지만, 본 프로젝트에서는 internal page의 가장 앞에 위치한 128byte 짜리 Page header 내부에 속한 li_pagenum(one more pagenum) 멤버가 leftmost page의 page number이다.

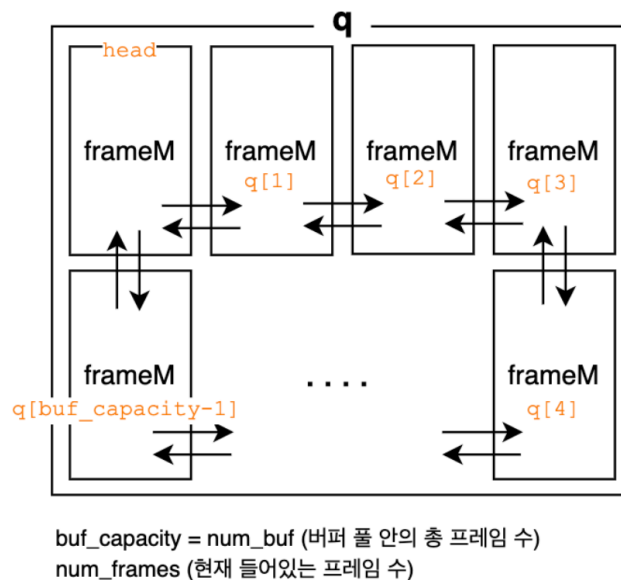
B+ tree에서 트리의 구조가 변경이 일어나는 경우는 merge와 split이 있다. 그 중 데이터를 삭제하면서 발생하는 redistribute는 split될 확률을 서로 비슷하게 만들어준다는 장점이 있지만, page의 내용이 변경되므로 disk상의 page를 두 번 써야 할 일이 발생할 가능성이 있어 이 점을 고려해서 사용해야 한다. 본 프로젝트에서는 tree structure modification을 줄이기 위해 페이지 내의 key가 하나도 없을 경우에만 merge를 수행하도록 구현하였다. 그러나 leaf page에서 key가 하나도 없어 coalesce(delayed merge)가 발생하는 경우에 속하면서 neighbor page가 가득 차 있는 상태였다면, 이후 이 이웃 페이지에 insert operation이 한 번이라도 들어오게 되면 결과적으로 merge-split이 발생하여 2번의 트리 구조 변경이 발생하게 된다. 이런 경우에는 structure modification을 줄이기 위해 redistribution이 필요하다고 판단하였다.



따라서 internal page에서 key가 존재하지 않아 delayed merge를 해야 하는 경우에 속하면서, capacity를 만족하지 못하여 merge를 수행하지 못하는 상황에서만 예외적으로 redistribute를 수행하도록 구현하였다. 기존의 redistribution은 이웃 페이지의 key를 한 개만 가져오지만, 이후 이 페이지에 delete operation이 들어와 또다시 structure modification이 일어나는 경우를 방지하기 위해 (neighbor page의 key 개수/2)개씩 가져오도록 하여 성능을 향상시켰다.

2) Buffer Management Layer

DB의 콘텐츠는 디스크에 저장되어 있고, 디스크의 속도는 메모리에 비해 매우 느리다. B+ tree index를 이용해 disk I/O가 $\log_F B$ 로 줄었으나 index layer와 disk layer의 속도 차는 여전히 존재한다. Buffer Management Layer는 이 속도차를 완충시키고 DBMS의 효율성을 높여 성능을 결정하는 데 중요한 역할을 하는 부분이다.



Frame manager 구조체 frameM 안에는 header page, general page를 담고 있는 union형 구조체 page_t와 해당 페이지의 meta data를 담고 있다. DBMS를 초기화할 때 인자로 받은 num_buf의 수만큼 프레임이 생성되며, 각 프레임은 이전/다음 프레임의 인덱스를 나타내는 int형 변수 prev/next를 담고 있어 circular queue처럼 동작한다.

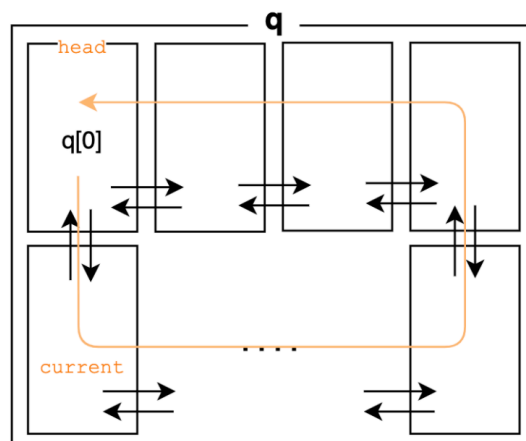
Table list는 (최대 테이블 개수+1)개만큼의 구조체 배열로, 총 10개의 hash table을 사용한다. Table id는 1부터 시작하므로 연산의 편의성을 위해 table list[1]부터 사용하기 위해 max_table의 값으로 11을 정의하였다. 해당 페이지가 버퍼에 올라와 있는지의 여부를 확인하는 hash table은 C++ STL library 중 map을 활용하였다.

기존 index layer에서 호출하던 disk space layer 함수 file_read/write_page()의 역할과 같이 buffer 내의 페이지를 읽고 쓰기 위해 buf_read/write_page() 함수를 선언하였다. 버퍼 레이어에서는 테이블이 닫히거나 flush가 발생할 경우에만 file write가 실행되어야 하므로, buf_write_page()에서도 file_read_page()를 호출한다. buffer가 비어 있을 경우에만 file_write_page()를 호출하여 disk I/O를 줄였다. 또한, 버퍼와

페이지의 값을 수정하자마자 바로 `buffer_latch`, `page_latch` 를 해제함으로써 다른 트랜잭션이 delay 없이 바로 해당 버퍼/페이지에 접근할 수 있도록 구현했다.

b+ tree 구조와 `buffer/page/record latch` 에 따라 특정 레코드에 접근하는 순서는 다음과 같다.

1	global buffer latch 획득
2	B+ tree 의 root page 에 속하는 버퍼 풀 페이지 탐색 (버퍼에 없다면, 디스크로부터 읽어오기)
3	buffer latch 해제
4	2 와 3 을 반복하여 찾고자 하는 레코드가 담긴 leaf page 까지 탐색 + page latch 획득
5	Lock manager latch 획득 + record lock 획득 시도
6-1	레코드 lock acquire 에 성공한 경우: lock manager latch 해제
6-2	레코드 lock acquire 에 실패한 경우: transaction manager latch 획득
	Transaction latch 획득 + transaction manager latch 해제
	Transaction latch 를 제외한 나머지 latch 모두 해제하기(page latch, lock manager latch 등)
	해당 transaction latch sleep 시키기 => 앞의 락이 해제되면서 깨워짐(자동으로 transaction latch 는 해제됨)
7	page latch & record lock 획득



버퍼 풀이 가득 차 있는 경우, LRU Policy 에 의거하여 page eviction 이 발생하도록 구현하였다. LRU(Least Recently Used)는 가장 오랫동안 사용하지 않은 데이터는 앞으로도 사용할 확률이 적다는 이론 하에 가장 오랫동안 참조되지 않은 데이터를 제거하는 알고리즘이다. Buffer 는 `prev/next` 로 연결된 double linked list 구조이므로 새로운 프레임이 추가되는 위치는 항상 `head` 가 가리키는 부분이다. 따라서, 가장 오래 된 프레임은 `head` 프레임의 `prev` 프레임이 된다.

여러 트랜잭션이 버퍼에 동시에 접근하는 경우, 한 트랜잭션이 특정 페이지의 내용을 읽거나 쓰는 사이 다른 트랜잭션에서 해당 페이지가 evict 될 위험이 있다. 따라서 버퍼 전체를 보호하는 `buffer_latch` 와, DBMS 의 효율성을 높이기 위해 `page_latch` 를 선언하여 페이지의 순차적 접근을 보장하였다. `page_latch` 를 획득하면 해당 페이지는 사용 중임을 나타내므로 eviction 대상에서 제외된다.

3) Disk Space Management Layer

Index layer 에서 알려 준 record 의 위치를 통해 실제 disk 에 I/O 를 담당하는 부분으로, page(4KB) 단위의 I/O 를 진행하기 위해 페이지 생성 및 해제와 read/write system call 을 호출해주는 역할을 한다. Index layer 와 buffer layer 에서 해당 페이지를 저장하기 위한 read/write page API 를 호출하면, disk space management layer 에서 page 기반 주소를 physical 주소(해당 파일 넘버, offset)으로 변경해서 최종 read/write system call 을 호출한다.

파일은 paginated layout 으로, disk space layer 가 각 파일의 균등화된 offset(4KB)마다 해당 페이지 번호를 맵핑한다. 파일의 가장 앞에는 해당 파일의 metadata 가 담긴 header page 가 존재하여 레코드의 개수, free space 의 위치 등을 저장한다. Free space 가 모두 소진되는 순간 새로 여러 장의 페이지를 생성하는 방식으로 구현하였다.

4) Lock Manager & Transaction Manager

DBMS 의 병행제어(concurrency control)을 담당하고 있는 부분으로, 많은 사용자의 요청을 받아 빠른 시간 내에 올바른 결과를 도출하도록 구현하였다. 데이터베이스는 다수의 사용자들이 한 레코드에 동시에 접근하는 경우가 빈번하게 발생하는데, 이런 상황에서 적절한 처리가 이루어지지 않는다면 데이터베이스의 무결성이 깨져 트랜잭션의 수행에 대해 비정상적인 결과가 나올 수 있다. 따라서 본 DBMS 에서는 transaction manager 에서 제공하는 Concurrency Control 을 통해 DB 를 consistent 한 상태로 유지하여 사용자로 하여금 혼자서만 데이터베이스를 사용하는 것처럼 느낄 수 있게 하였다. Concurrency control 에 대해서는 아래에 자세히 기술하였다.

5) Logging & Recovery Layer

장애가 발생했을 때 안전하게 DB 의 상태를 올바르게 복구하는 crash recovery 와 관련된 API 를 담은 레이어이다. Crash recovery 에 대해서는 아래에 자세히 기술하였다.

Concurrency Control Implementation

트랜잭션이란 하나의 논리적 작업 단위를 구성하는 일련의 연산들의 집합으로, 프로그램 수행의 단위이다. 트랜잭션의 대표적인 예로 계좌 이체가 있다. 계좌 A에서 계좌 B로 5만원을 송금하는 이체 작업은 전체 작업이 정상적으로 완료되거나, 정상적으로 처리될 수 없는 경우에는 아무것도 실행되지 않은 처음의 상태로 돌아가야 한다.

트랜잭션은 문제 없이 정상적으로 수행되어 commit으로 종료되거나, 사용자의 요청에 의해 abort되거나 시스템이 문제를 감지하여 DBMS가 이를 abort하는 경우가 있다. DBMS는 이와 같은 상황에서 트랜잭션을 적절하게 관리하여 사용자로 하여금 정상적인 데이터에 접근할 수 있도록 해 주어야 한다.

트랜잭션이 동시에 실행되는 환경에서 발생 가능한 문제로는 크게 아래의 세 가지가 있다.

1) The Lost Update Problem

P1	Time	P2
<i>r(x)</i>	<i>/* x = 100 */</i>	
<i>x := x+100</i>	1	
<i>w(x)</i>	2	<i>r(x)</i>
	3	<i>x := x+200</i>
	4	
	5	
	<i>/* x = 200 */</i>	
	6	<i>w(x)</i>
	<i>/* x = 300 */</i>	

↑
update "lost"

Observation: problem is the interleaving $r_1(x)$ $r_2(x)$ $w_1(x)$ $w_2(x)$

서로 다른 트랜잭션이 한 레코드에 write를 연속으로 수행하는 경우, 먼저 실행된 트랜잭션 P1은 정상적으로 실행되었으나 P2에 의해 overwrite되어 x의 값이 변경되지 않는 문제가 발생한다.

2) Inconsistent Read Problem

P1	Time	P2
<i>sum := 0</i>	1	<i>r(x)</i>
<i>r(x)</i>	2	<i>x := x - 10</i>
<i>r(y)</i>	3	<i>w(x)</i>
<i>sum := sum + x</i>	4	
<i>sum := sum + y</i>	5	
	6	
	7	
	8	
	9	<i>r(y)</i>
	10	<i>y := y + 10</i>
	11	<i>w(y)</i>

↑
"sees" wrong sum

Observations:

problem is the interleaving $r_2(x)$ $w_2(x)$ $r_1(x)$ $r_1(y)$ $r_2(y)$ $w_2(y)$
no problem with sequential execution

rw-confilct 의 경우, 위의 상황에서는 $x+y$ 의 값이 일정하지 않게 되면서 문제가 발생하게 된다.

3) Dirty Read Problem

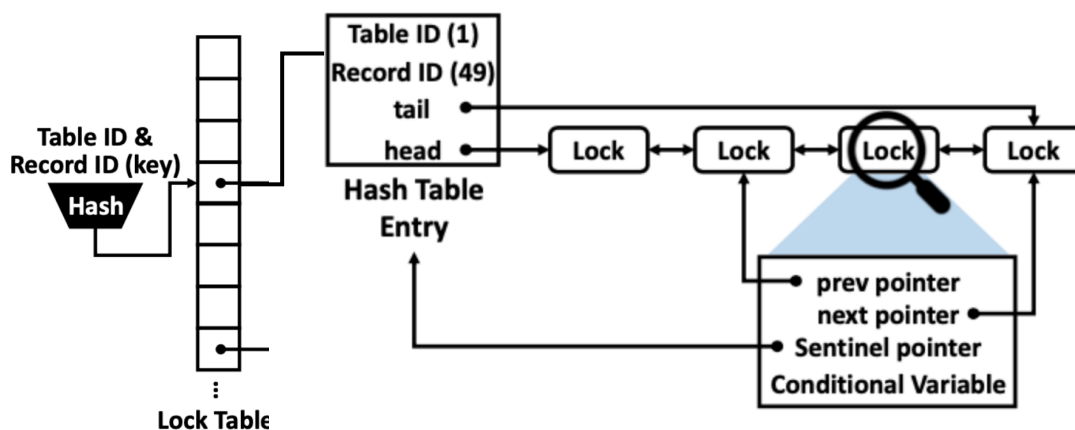
<i>P1</i>	<i>Time</i>	<i>P2</i>
$r(x)$	1	
$x := x + 100$	2	
$w(x)$	3	
	4	$r(x)$
failure & rollback	5	$x := x - 100$
	6	
	7	$w(x)$

↑
cannot rely on validity
of previously read data

Observation: transaction rollbacks could affect concurrent transactions

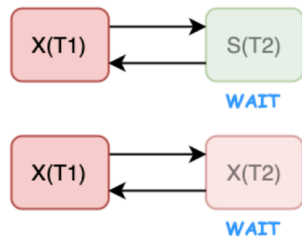
위의 예시도 rw-conflict 의 상황 중 하나로, 한 트랜잭션(P1)이 중간에 abort 되는 경우 동시에 같은 레코드에 접근 중이던 다른 트랜잭션(P2)은 dirty 한 값을 가져와서 write 를 수행하게 된다.

Serializable 이란 어떤 트랜잭션들이 중첩되어 실행되는 것과 순차적으로 실행되었을 때의 결과가 같을 때를 일컫는 것으로, 2PL(Two Phase Locking) Protocol 에 의해 보장된다. 2PL 은 위와 같은 문제를 해결하기 위해 상호 배제 기능을 제공하는 기법이다. 2PL 을 따르는 트랜잭션은 반드시 reading 전에 S-lock 을, writing 전에 X-lock 을 획득하여야만 해당 operation 을 수행할 수 있다. 또한, 획득한 락을 해제하기 전까지는 다른 어떤 락도 추가로 얻을 수 없다.



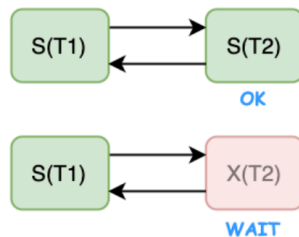
각 튜플(레코드)별로 락의 요청과 획득을 나타내기 위해 선택한 디자인은 다음과 같다. C++ STL Library 인 map 을 활용하여 (table id, record id)가 key 이고 value 로는 hash entry 형 구조체를 갖는 lock table 을 두고, 해당 hash entry 의 head 와 tail, 각 lock 구조체가 가진 prev, next 포인터를 통해 한 레코드에 동시에 접근하는 락의 waiting list 를 표현하였다. Double linked list 로 구현된 Lock list 는 획득 및 해제 요청을 하는 락 오브젝트의 전/후로 충돌 가능성이 있는 다른 락 오브젝트가 존재하는지를 판단한다.

X Lock을 획득한 상태일 때:
T2의 lock의 종류와 상관없이 T1이 commit되기 전까지는 접근 불가



Lock mode 는 크게 Shared Lock(S)과 Exclusive Lock(X)으로 나뉜다. 값의 읽기만 가능한 S-lock 과 달리 X-lock 은 레코드 값을 읽고 쓰는 것이 가능하도록 해 주는 락이다. 따라서 어떤 트랜잭션이 X-lock 을 획득한 상태라면, 이 트랜잭션이 commit 되기 전까지 다른 트랜잭션은 해당 레코드에 접근할 수 없다.

S Lock을 획득한 상태일 때:
T2가 X lock을 요청할 때, T1이 commit되기 전까지는 접근 불가



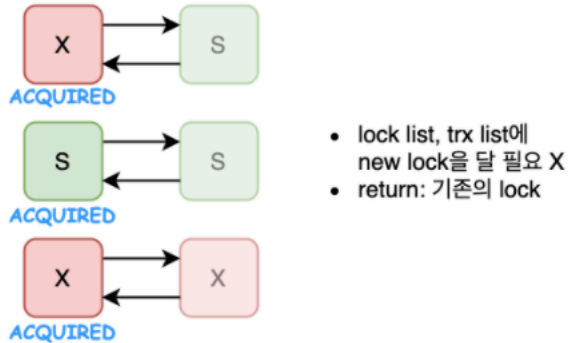
반면 S-lock 을 획득한 상태라면, 다른 트랜잭션의 읽기 접근(S-lock 획득)은 허용된다. 동시에 다수의 트랜잭션이 한 레코드의 값을 읽기만 하는 것은 conflict 가 발생하지 않기 때문이다. 그러나 다른 트랜잭션이 X-lock 을 요청한다면, 해당 요청은 기존의 돌아가고 있는 트랜잭션이 commit 될 때까지 기다리게 된다. 기존 트랜잭션에서 여러 번 db_find()를 호출한다고 할 때, 동시에 다른 트랜잭션에서 db_update()로 값 변경을 해버리면 db_find()의 결과가 다르게 나올 수 있기 때문이다.

따라서 본 프로젝트에서는 아래와 같이 lock_acquire 가 실행되도록 구현하였다. 기존 트랜잭션의 lock mode 가 새로 요청받는 트랜잭션의 lock mode 보다 더 강력하거나 같을 경우, 트랜잭션의 동일함 여부와 상관 없이 lock list 와 transaction list 에 새로 lock 을 추가할 필요 없이 기존의 락 오브젝트를 반환한다. 어차피 '락'은 어떤 트랜잭션이 해당 레코드의 락을 획득하였는지(=현재 시점에 값에 접근할 수 있는지) 알기 위한 개념이므로, 더 강력한 락이 존재한다면 새로 acquire 할 필요가 없다고 판단하여 위와 같이 구현하였다.

기존 트랜잭션의 lock mode 보다 새로 요청받는 트랜잭션의 lock mode 가 더 강력한 경우에는 트랜잭션의 동일함 여부에 따라 case 를 나누어 구현하였다. 같은 트랜잭션에서 더 강력한 lock mode 를 요청하는 경우, lock 을 한 단계 업그레이드 시켜주기 위해 new lock 을 반환한다. 이 때, lock list 와 transaction list 에 new lock 을 추가하고, 기존의 S-lock 은 conversion deadlock 을 위해 제거하지 않고 둔다. 두 트랜잭션이 다른 경우, 앞의 트랜잭션이 trx_commit()을 호출하면서 lock 을 release 할 때까지 lock 의 cond 변수를 이용하여

기다리게 함으로써 기존 트랜잭션이 종료되어야만 뒤의 X-lock 을 획득하면서 새로운 트랜잭션이 마저 실행될 수 있도록 구현하였다.

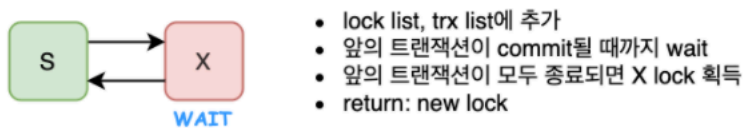
i) lock's mode \geq new_lock's mode인 경우



ii) lock's mode < new_lock's mode인 경우 (같은 트랜잭션)



iii) lock's mode < new_lock's mode인 경우 (다른 트랜잭션)



Deadlock 이란 두 개 이상의 트랜잭션이 서로 lock 이 해제되기를 기다리면서 DBMS 가 무한 대기 상태에 빠지는 현상이다. 본 DBMS 에서는 deadlock 을 해결하기 위해 deadlock detection method 를 차용하였다. 두 개 이상의 트랜잭션이 서로를 기다리는 cycle 현상이 탐지되면 deadlock 이 발생했다고 판단하고 현재까지 진행한 트랜잭션을 abort 하는 작업이 필요하다. 서로를 기다리고 있는지를 확인하기 위해 다음과 같은 디자인으로 구현하였다.

```

int trx_isDeadlock(int trx_id, lock_t *lock) {

    pthread_mutex_lock(&trx_manager_latch);
    // deadlock is detected when trx is waiting itself.
    // so check trx's wait lock list to detect deadlock.
    if (lock->prev == NULL) {
        pthread_mutex_unlock(&trx_manager_latch);
        return false;
    }
    int tmp_trx_id = lock->prev->owner_trx_id;
    lock_t *tmp = trx_manager[tmp_trx_id]->lock_list;
    while (tmp->trx_next_lock != NULL) {
        if (tmp->owner_trx_id == trx_id)
            return true;
        tmp = tmp->trx_next_lock;
    }
    pthread_mutex_unlock(&trx_manager_latch);
    return false;
}

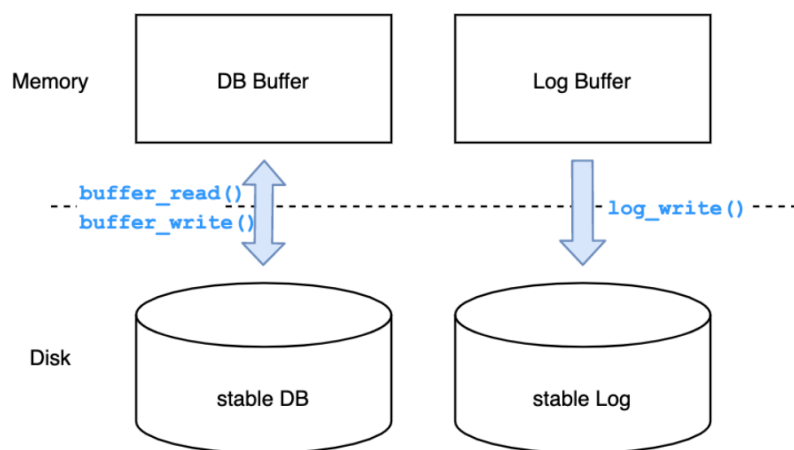
```

우선 lock_acquire()에서 새로 획득한 lock 을 lock list 에 추가하고 나서, 자신보다 선행된 락 오브젝트의 owner_trx_id 속성을 이용하여 선행 락을 가지고 있는 주인 트랜잭션을 찾는다. 트랜잭션 id 를 안다면 트랜잭션 테이블 map 을 통해 해당 트랜잭션의 lock list 를 순회할 수 있다. 만약 trx_id 가 일치하는 락이 존재한다면 deadlock 상태에 교착된다는 뜻이므로, true 를 반환한다. Lock list 를 순회하여도 일치하는 락이 없는 경우, false 를 반환하여 해당 락 오브젝트는 데드락을 발생시키지 않음을 증명한다.

Crash-Recovery Implementation

DBMS에서는 SQL을 실행하는 동안 트랜잭션 장애, 시스템 장애, 미디어 장애 등으로 인한 비정상적인 종료가 발생할 수 있다. DBMS의 Atomicity와 Durability를 보장하기 위해서는 트랜잭션을 수행하는 도중 장애로 인해 손상된 DB를 손상되기 이전의 정상 상태로 복구시키는 작업이 필요하다.

Undo란 로그를 이용하여 오류가 발생한 트랜잭션과 관련된 모든 변경을 취소하여 복구를 수행하는 방법이고, Redo란 로그를 이용하여 오류가 발생한 트랜잭션을 재실행하여 복구를 수행하는 방법이다. 따라서 Undo와 Redo를 수행하기 위해 트랜잭션이 반영한 DB의 변경사항을 별도의 로그 파일에 따로 기록해둔다. 로그(log)란 로그 레코드의 모임으로, 각 로그 레코드는 LSN(Log Sequence Number)이라는 고유 식별자를 가진다. 로그는 항상 뒤에 추가되는 방식으로 기록되므로 LSN은 단조 증가한다.



DB에 buffer의 내용을 write하기 전에 logging 작업을 우선 실행하는 것을 WAL(Write-Ahead Logging)이라고 하는데, 후에 DBMS에 장애가 발생하더라도 로그 파일을 이용해서 DB를 정상적인 상태로 복구시킬 수 있게 해 준다. DB buffer와 마찬가지로 성능을 위해 로그 버퍼에 로그 레코드를 일정 수만큼 모아뒀다가 로그 파일에 write를 진행하도록 구현하였다. 따라서 로그 레코드가 버퍼에서 파일에 써질 때는 다음 상황 중 하나가 될 것이다.

- 1) 어떤 트랜잭션이 commit을 요청한 경우
- 2) WAL을 해야 하는 경우
- 3) 로그 버퍼가 꽉 찬 경우(본 프로젝트에서는 로그 버퍼의 크기를 10MB로 설정하였다.)

로그 레코드는 Redo와 Undo에 관련된 정보를 포함하고 있다. 따라서 로그는 Redo 정보만을 담은 레코드와 Undo 정보만을 담은 레코드, 그리고 둘을 모두 담고 있는 Undo-redo 레코드가 있다. Redo 레코드는 트랜잭션이 실패한 경우 트랜잭션에서 변경한 내용을 재실행하기 위한 정보가 들어있다. Undo 레코드는 트랜잭션이 rollback할 때 변경했던 내용을 되돌리는 데 필요한 정보가 들어있다. 본 DBMS에서는 로그 레코드 구조체 안의 old image와 new image 멤버를 이용하여 데이터의 변경 전, 변경 후 이미지를 복제하여 저장하는 방식으로 구현하였다.

본 DBMS에서는 ARIES method를 기반으로 recovery를 수행하도록 구현하였으나 checkpoint에 대해서는 고려하지 않았으며, torn pages가 발생하는 경우 또한 고려하지 않고 구현하였다. ARIES method는 C.Mohan이 발명한 데이터베이스 복구 알고리즘으로, 수많은 상용 DBMS에서 사용 중인 방법이다. WAL과 LSN을 기반으로 No-force/Steal 정책을 따르며 효율적으로 복구를 수행한다.

ARIES method는 크게 세 단계로 구성되어 있다.

1) Analysis Pass

트랜잭션의 리커버리 시 가장 먼저 실행되는 부분으로, Redo-winners 패러다임의 analysis 알고리즘을 통해 winner와 loser 트랜잭션을 찾는다. stable log의 처음부터 끝까지 scan하며 트랜잭션의 commit 로그가 존재하지 않는다면, Undo의 대상이므로 loser 트랜잭션 리스트에 추가한다. 어디서부터 시스템이 복구를 시작해야 하는지, 어느 트랜잭션들을 복구해야 하는지를 알아낸다.

2) Redo Pass

시스템 장애 시에 데이터베이스를 해당 상태로 복원하는 부분이다. Stable log에 저장된 내용을 이용해 순서대로 다시 실행하여 DB를 원래 상태로 복구한다.

버퍼 풀 내부 페이지의 header에 기록된 page LSN < 로그 레코드의 LSN 일 경우에만 반복해서 Redo를 진행하는데, 이는 WAL에 의해 순차적으로 기록되고 LSN의 sequence가 증가하기 때문에 page LSN이 로그 레코드의 LSN보다 크거나 같다는 것은 현재 stable DB가 이미 최신 상태라는 것을 의미하기 때문이다. Redo를 진행하고 나면 해당 페이지의 pageLSN을 갱신한다.

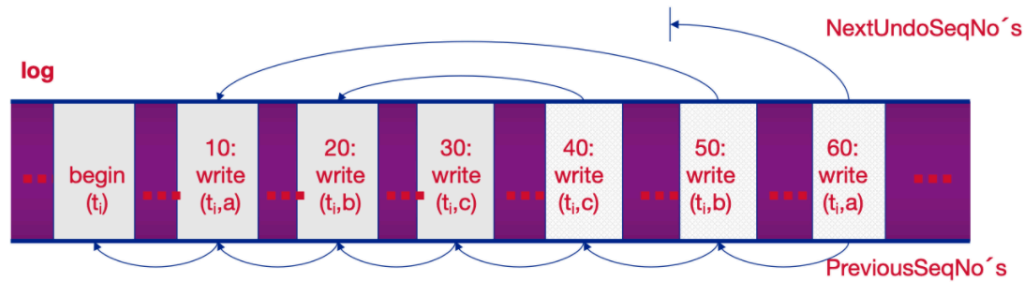
3) Undo Pass

트랜잭션의 deadlock 상태가 감지되어 `trx_abort()`를 호출하면 기존의 값을 저장해 둔 `undo_log`에서 값 변경이 일어난 레코드 부분을 다시 되돌리는 작업이 일어난다. 모든 Loser transaction의 write operations를 Stable Log에 저장된 역순으로 rollback한다. Analysis Pass에서 판별된 Loser 트랜잭션은 uncommitted 상태에서 crash가 발생했다는 의미이므로, DBMS의 Atomicity(All or Nothing)를 보장하기 위해 트랜잭션의 실행 전 상태로 DB를 돌려놓도록 구현한다.

operation 수행 중 수정된 페이지들이 버퍼 관리자의 page eviction에 따라 디스크에 출력될 수 있는데, 이 때 아직 완료되지 않은 트랜잭션이 수정한 페이지가 디스크에 출력될 가능성이 있다. 트랜잭션이 정상적으로 종료되지 않았다면 트랜잭션이 변경했던 모든 페이지를 다시 원상복구시켜야 한다.

따라서 Stable log가 기록된 역순으로 new image→old image의 대체과정을 해당 트랜잭션의 시작 로그까지 반복하면 Undo가 완료된다.

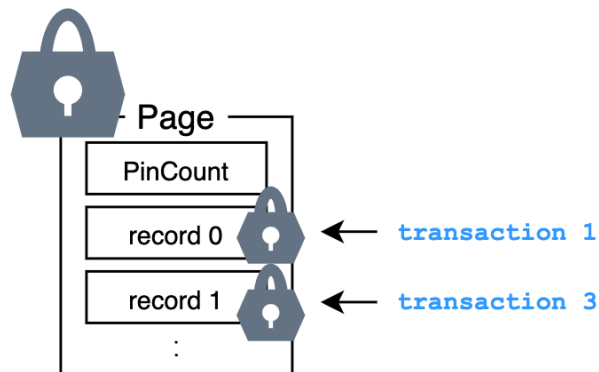
Undo를 수행하고 나면 해당 Undo 작업에 대한 보상 레코드 CLR이 발급되는데, 이는 'Undo 로그의 이전 로그'를 가리키게 해서 한 번 Undo된 로그를 다시 접근해서 re-undo되는(원상복귀) 현상을 방지한다.



또한 compensate log record에만 존재하는 next_undo_LSN을 통해 해당 트랜잭션의 다음 Undo action을 기록해두고, 이미 Undo를 수행한 상태일 경우 이 부분을 통해 part skip이 가능하게 구현한다. 이를 통해 undo recovery를 진행하면 할수록 트랜잭션의 시작 지점에 가깝게 undo sequence number가 설정되고, 결과적으로 undo recovery가 도중에 계속 failure가 나더라도 수행해야 할 Undo의 양은 줄어들게 되어 성능을 향상시킬 수 있다.

In-depth Analysis

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



페이지 구조체에 int 형 변수 pinCount 를 추가하여 mutex 와 pinCount 를 함께 사용한다. pinCount 가 0 이 아니면 다른 트랜잭션이 해당 페이지를 사용 중이라는 의미이나, 만약 락 획득을 요청한 락의 종류가 S lock 이라면 page latch 와 상관 없이 바로 레코드 락을 획득하도록 구현한다. 결과적으로 한 페이지 내에 다른 레코드들에 접근하는 트랜잭션끼리의 병목현상이 일어나지 않게 되고, many concurrent non-conflicting read only transactions 상황에서의 성능 향상을 기대해볼 수 있다.

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

서버를 여러 개 증축하는 distributed database architecture 를 통해 cache 를 여러 개 두고, 사용자마다 특정 ip(entry point)로 서버에 접속하여 write operation 을 수행하도록 한다면 DBMS 의 성능과 신뢰성 모두 높일 수 있다. 또는 host RAM 의 크기를 늘려서 CPU 의 병목현상을 해결하는 방법이 있을 것이다.

