Database System 2020-2

Final Report

ITE2038-11801

2019065687

윤동근

Table of Contents

Overall Layered Architecture

dbms 는 크게 db api, index manager, buffer manager, file manager 의 네 개의 layer 로 구성되어 있다. 각 layer 에 대한 설명은 다음과 같다.

db api 는 db 에 update 와 같은 query 를 받는 layer 이다. 사용자의 입력은 보통 parsing 및 query optimization 을 진행한 후 db api layer 로 넘겨지게 되고, 이 layer 는 index manager 로 query 를 넘겨준 후 해당 query 의 성공 및 실패 여부를 index manager 로부터 받아 다시 상위 layer 로 넘겨주게 된다. 현재 우리의 과제에서는 query optimization 의 과정이 없으므로, 사용자와 직접 맞닿아 있어 사용자의 입력을 받은 후 사용자에게 성공 및 실패 여부를 return 해주는 역할을 한다고 볼 수 있다.

제공하는 api 는 다음과 같다.

1. int init_db(int buf_num, int flag, int log_num, char* log_path, char* logmsg_path)

buf_num 만큼의 buffer 를 가진 db 를 생성한다. 만일 log_path 에 파일이 있다면 log 를 가져와 읽고 recovery 를 진행한다. buffer 와 recovery 에 대한 자세한 설명은 후술한다. 이외 다른 변수는 과제의 테스트를 위하여 추가된 변수이다. return value 는 성공시 0 이며, 실패시 1 이다.

2. int open_table(char* pathname)

pathname 의 파일을 읽는 함수이다. 파일이 존재한다면 기존 파일을 읽어 테이블로 사용하는 것이고, 존재하지 않는다면 새로운 파일을 만들어 테이블로 사용한다. 사용자는 table_id 로 open 된 파일과 상호작용하게 된다. return value 는 성공시 table_id 이며, table_id>0 이고 unique 하다. 실패시 0 이다.

3. int db_insert(int table_id, int64_t key, char* value)

해당 table 에 key 를 key 로하고 value 를 record 로 하는 데이터를 삽입한다. return value 는 성공시 0, 실패시 1 이다.

4. int db_find(int table_id, int64_t key, char* ret_value, int trx_id)

해당 table 에서 key 로 record 를 찾아 ret_value 에 담아 return 한다. 이를 위해 함수를 호출하기 전 ret_value 에 메모리가 미리 할당되어 있어야 한다. return value 는 성공시 0, 실패시 1 이다.

5. int db_delete(int table_id, int64_t key)

해당 table 에서 key 로 record 를 찾아 삭제한다. return value 는 성공시 0, 실패시 1 이다.

6. int db_update(int table_id, int64_t key, char* value, int trx_id)

해당 table 에서 key 로 record 를 찾아 value 를 update 한다. return value 는 성공시 0, 실패시 1 이다.

7. int close_table(int table_id)

table 를 close 하여 buffer 에서 해당 table 의 내용을 flush 한다. return value 는 성공시 0, 실패시 1 이다.

8. int shutdown_db(void)

db 를 shutdwon 하여 buffer 의 모든 내용, log 등을 flush 한다. 할당했던 memory 들도 free 해주므로 다시 db 를 동작하기 위해서는 init_db 를 호출해야 한다. return value 는 성공시 0, 실패시 1 이다.

추가로 transaction 에 관련된 다음 세 api 를 갖는다.

9. int trx_begin(void)

trx 를 시작한다. return value 는 현재 시작하는 trx 의 id 이다.

10. int trx_commit(int trx_id)

trx 의 변경 사항을 db 에 저장한다. return value 는 성공시 trx_id, 실패시 0 이다.

11. int trx_abort(int trx_id)

trx 의 변경 사항을 db 에 저장하지 않고 끝낸다. update 등이 실패했을 시 주로 호출되며, return value 는 성공시 trx_id, 실패하면 0 이다.

여기서 db 에 저장한다는 의미는 실제로 flush 한다는 것이 아닌, transaction 의 성질 ACID 중 Durabilitry, 즉 commit 된 transaction 은 영구적으로 남는다는 성질에 관한 이야기이다.

추가로, 현재 구현에서는 db_insert 등에서 직접 buffer manager 에 접근하는 동작이 있으나, 이는 layer 를 뛰어넘은 것에 해당하므로 index manager 에서 진행되는 것으로 수정되어야 한다.

index manager 는 db_api 로부터 전달받은 table_id 에 해당하는 file 에서 key 의 위치를 찾고, query 를 처리하여 성공/실패 여부를 돌려주는 layer 이다. query 는 table_id 와 key 의 쌍만이 주어지고 해당 key 가 어느 file 의 어느 위치인지는 주어지지 않고, 또 주어질 수도 없기에 index manager 는 해당 간격을 메꿔주는 역할을 한다.

이를 위해 key 의 삽입/삭제/탐색이 빠른 자료구조를 사용하며, 세 operation 모두 $O(lgN)$에 수행하고자 B+ tree 를 사용하게 된다. 이러한 자료구조가 여럿 있지만 B+ tree 를 쓰는 이유는 ping-pong problem 을 delay merge 를 통해 줄일 수 있고, disk 에서 file 을 읽어오는 행위가 매우 느리기 때문에 한번에 많은 양의 data 를 가져와야 하며 B+ tree 가 이에 적합하기 때문이다. AVL tree 같은 경우

rotation 이 자주 발생하여 한번에 변경이 일어나는 page 가 많아진다는 단점이 있고, Red-Black tree 의 경우 기본적으로 binary search tree 이므로 order 를 크게 하기 어렵다는 단점이 있다.

index manager 역시 file 의 해당 위치에 어느 data 가 있는지를 알지는 못하므로, table_id 와 pagenum 을 통해 얻어낸 file 의 위치를 기반으로 file manager 에게 해당 file 의 해당 위치에 있는 page 의 data 를 읽거나 쓰겠다는 요청을 하게 된다. 그러나 매번 file manager 를 통해 data 를 읽고 쓰면 durable 할지언정 속도에서 막심한 피해를 입게 되며, 따라서 실제로는 index manager 과 file manager 사이에 buffer manager 라는 layer 를 추가하여 buffer manager 로 page 를 읽는 요청을 보내게 된다.

buffer manager 는 index manager 이 요청한 page 를 전달해주는 layer 이다. 이를 위해 내부적으로 buffer 를 가지고 있으며, 이 동작과정은 cache 와 유사하다. 즉, 요청 받은 page 가 buffer pool 에 남아 있으면 해당 page 를 그대로 return 해주며, 그렇지 않을 시에만 file manager 를 통하여 page 를 읽어와 buffer 에 가져오는 방식으로 동작한다. 이 과정에서 buffer 가 page 로 가득 차 있다면 file manager 에서 data 를 가져오기 전 사용한지 오래된 page 를 eviction 하는 동작을 진행한다. page 의 변경은 eviction 이 일어나거나 shutdown_db 를 호출했을 시에 file manager 로 쓰기 요청을 보내 disk 에 반영되게 된다. 이러한 동작이 일어나기 전에 db 가 shutdown 된다면 disk 에 수정 내용이 반영되지 않으며, 이는 Crash-Recovery 에서 해결한다.

page 가 buffer pool 에 남아 있는지는 현재 구현에서는 linear search 로 탐색하며, 오래된 page 를 eviction 하기 위해 LRU 정책을 사용한다. 그러나 linear search 는 매우 비효율적이므로, 메모리의 크기가 충분하다면 hash 를 사용하는 자료구조를 사용하여 탐색이 상수 시간에 가깝게 이루어지도록 수정되어야 한다.

file manager 는 상위 계층이 요청한 file 에서 offset 만큼 떨어진 곳에서 len 만큼의 data 를 읽거나 쓰는 역할을 한다. 현재는 file_read(write)_page 로 PAGE_SIZE 만큼만을 읽거나 쓰도록 동작하지만, 후술할 recovery 시 log file 의 read 및 write 까지 수행하기 위해서는 조금 더 general 하게 바뀔 필요가 있다. 또한, 현재 과제의 spec 에서는 buffer manager 에서도 table_id 를 사용하므로 table_id 에 해당하는 file 이 무엇인지를 file manager 에서 바꾸어 주는 역할도 수행한다. 이 layer 는 read()와 같은 system call 을 사용하여 직접 disk 와 상호작용한다

.

Concurrency Control Implementation

db 는 여러 사용자가 요청을 보내도 각 사용자에게 혼자서 db 를 사용하는 것과 다름없는 환경을 제공하는 것이 가장 이상적이다. 또한, 이 과정에서 lost update 와 같은 issue 가 발생해서는 안된다. 이를 위해 다양한 layer 에서 latch 또는 lock 을 사용하고 있다.

먼저 index manager 에서는 record 단위로의 lock 을 요청하게 된다. 이 record lock 은 사용자가 요청한 trx 이 끝날 때까지, 즉 db api 에서 trx_commit 이나 trx_abort 가 발생할 때 까지 유지된다. (현재 구현에서는 project 5 에서 db_update 내에서 abort 를 처리했던 것의 연장으로, trx_abort 가 들어오기 전에 직접 abort 를 처리하고 있다.)

record 를 찾기 위한 과정에서 buffer 에 page 를 요청하게 되는데, 이에 buffer manager 에서 buffer latch 를 걸고 page latch 를 건 후 buffer latch 를 풀며 return 해주게 된다. buffer latch 를 거는 이유는 LRU 등 buffer 내부에서 page 들을 저장한 linked list 에 변형이 일어나기 때문이며, page latch 를 거는 이유는 insert 나 delete 와 같이 구조가 변경되는 operation 이 진행될 경우 탐색하고자 하는 page 가 merge 되어 사라지거나, 탐색하고자 하는 record 가 split 되어 위치가 변경되는 등의 일이 발생할 수 있기 때문이다. 현재 구현에서는 page latch 를 한 번에 하나씩만 걸어주므로 여전히 문제가 생길 수 있으나, buffer manager 에서 page latch 을 풀고 새롭게 page 를 가져오는 동작의 순서를 바꾸어 latch crabbing 을 구현하면 해결할 수 있다.

추가로, layer 에서 설명하진 않았던 trx manager 와 log manager 에서도 lock 을 걸어주어야 한다. 현재 trx manager 는 trx_id 로 trx 를 알아내기 위해 thread unsafe 한 std::unordered_map 을 사용하고 있기에 trx_begin 에서 insert 가 일어나거나 trx_commit 에서 erase 가 일어나는 등의 상황에서 trx manager latch 를 걸어주어야 한다. log manager 또한 자체적으로 log buffer 를 가지고 있으므로 log buffer 에 write 하고 log 를 flush 하는 과정에서 log buffer latch 를

걸어주어야 한다. 이 과정은 Crash-Recovery Implementation 에서 보다 자세히 서술할 예정이다.

record lock 에 대해 보다 자세히 서술하자면, record lock 은 find 연산을 위한 shared_lock, update 연산을 위한 exclusive_lock 의 두 mode 로 이루어져 있으며, 획득 과정은 lock_acquire 와 lock_wait, lock_release 의 세 단계로 나누어져 있다.

먼저 lock_acquire 은 lock manager 에 table_id 와 key 를 통해 record lock 을 요청하는 과정이다. lock manager 는 요청된 record 에 다른 trx 가 lock 을 쓰고있는지, 이미 같은 trx 가 획득한 lock 인지를 검사한다. 이미 같은 trx 가 획득한 적 있는 lock 이었다면 바로 return 해주며, 그렇지 않다면 deadlock 을 탐지하고 해당 trx 의 lock list 와 해당 record 의 lock linked list 에 lock object 를 추가한다. 이 과정에서 trx 에 접근하므로 trx manager latch 를 걸어 trx lock 을 획득하고, lock linked list 에 변경을 하기 위해 lock manager latch 를 걸게 된다. 이후 lock 획득에 성공했는지(0), wait 해야 하는지(1), deadlock 이 발생하였는지(2)를 return 한다. 획득 조건은 shared lock 의 경우 해당 record 의 lock list 에 exclusive lock 이 하나도 없음이며, exclusive lock 의 경우 해당 lock list 가 비어있거나 자신과 같은 trx_id 의 lock 만이 존재함, 즉 자신과 다른 trx 의 lock 이 없음이다. 보다 정확히는 자신 앞에 자신과 다른 trx 의 lock 이 없음이나, lock_acquire 에서는 자신이 tail 이므로 lock_list 전체를 보는 것과 동일하게 된다.

record lock 을 획득했다면(0), page latch 와 record lock 을 갖고 있으므로 db operation 을 수행한 후 page latch 와 trx lock 을 풀고 return 해준다. record lock 은 계속해서 걸려있게 된다.

wait 해야 된다면(1), page latch 를 풀고 lock_wait 를 통해 잠이 든다. 이 때 trx lock 을 풀어주며 잠이 들게 되고 lock release 에서 notify 가 일어날 때 trx lock 을 걸고 lock 획득 조건을 만족했는지 확인하게 된다. 현재 구현에서 획득 조건은 shared lock 은 notify 해주자 마자 wait 가 풀리며 lock 을 획득하게 되고, exclusive lock 은 lock_acquire 에서 사용했던 조건과 같이 lock list 에서 자신 앞의 lock 이

없거나(즉, 자신이 첫번째거나) 자신 앞의 lock 이 자신과 같은 trx_id 의 lock 만이 존재함이다.

deadlock 이 발생했다면(2), return 하고 abort 하게 된다.'

이후 trx_commit 이나 trx_abort 가 발생하면 lock_release 가 호출되게 되는데, lock acquire 에서 trx 별로도 lock_obj 를 저장해 놓았으므로 이 trx 의 lock list 를 순회하면서 lock_release 를 호출해주면 된다. 현재 구현에서 notify 는 lock_obj 단위로 일어나므로, release 되는 lock_obj 로부터 처음으로 나타나는 exclusive lock 까지 notify 를 해주고 있다. 이렇게 구현한 까닭은 project 4 에서 cv 를 전역변수로 공유하는 것과 lock_obj 마다 갖고 있는 것 중 후자의 성능이 더 좋음을 확인했기 때문이며, head – 현재 lock – trx 2 의 shared lock – trx 2 의 exclusive lock 과 같은 경우가 있으므로 shared lock 이 있더라도 exclusive lock 까지 notify 해주는 편이 구현이 더 깔끔했기 때문이다. 현재 lock 이 exclusive lock 일 경우 자신 이후의 shared lock 들을 모두 풀어주어도 무방함을 쉽게 확인할 수 있으며, 현재 lock 이 shared lock 이라면 자신 뒤의 exclusive lock 만을 notify 해주면 되긴 하나 어차피 순회해야 하므로 순회 과정에서 notify 를 해주도록 구현하였다.

사용되는 latch 와 lock 을 각 layer 의 component 별로 정리하자면 다음과 같다.

index manager : page, record

buffer manager : buffer, page

trx manager : trx manager, trx

lock manager : lock manager

log manager : log buffer

Crash-Recovery Implementation

buffer manager 에서 이야기했듯 buffer manager layer 를 추가하면 commit 된 trx 의 변경사항이 buffer 에만 남아 있는 상태로 crash 가 발생할 경우 변경 사항이 disk 에 반영되지 않아 durability 가 깨질 수 있다. 또한, buffer 의 여부와 관계없이 commit 되지 않은 trx 의 변경사항이 disk 에 반영되어 있다면 이를 rollback 해주어야 atomicity 가 지켜진다. Crash 가 났을 때에도 이 두가지를 보장하기 위해 Recovery 를 사용한다.

Recovery 를 위해 db 는 현재 다섯가지의 log 를 사용한다. BEGIN, UPDATE, COMMIT, ROLLBACK, COMPENSATE 이 그것이다. trx_begin 와 trx_commit, trx_abort 는 각각 BEGIN, COMMIT, ROLLBACK log 를 발급하며, db_update 는 UPDATE log 를, abort 하면서 undo 된 log 들은 COMPENSATE log 를 발급한다. BEGIN, COMMIT, ROLLBACK log 들은 image 의 정보가 필요하지 않아 db api 에서 바로 발급해주고 있으나, UPDATE log 의 경우 old image 와 new image 가 필요하여 index manager 에서 record lock 을 acquire 하고 update 하는 도중 발급해주는 방식을 취하고 있다. COMPENSATE log 의 경우 undo 에서 발급해주기로 계획하였으나 이부분을 시간 내에 구현하지 못하였다.

발급된 log 는 log Manager 에 의해 log buffer 에 기록된다. log Manager 는 LSN 과 같이 log 발급에 필요한 정보들과 log buffer 를 갖고 있다. log buffer 는 file manager layer 와 상호작용하여 file 에 flush 해주는 과정이 필요하나, 시간이 없어 log Manager 에서 직접 system call 을 호출하는 방식으로 구현하였다. log buffer 가 flush 되는 상황은 db api 에서 trx_commit 이나 trx_abort 가 일어났을 때, 혹은 buffer manager 에서 eviction 이 일어났음을 알려줄 때이다. 현재는 eviction 이 일어났을 때 flush 되는 과정이 생략되었다. 이를 통해 commit 된 trx 의 log 가 disk 에 반영되지 않고 crash 가 났더라도 redo 될 수 있어 durability 를 지킬 수 있고, commit 되지 않은 trx 의 log 가 eviction 되어 disk 에 반영되었더라도 undo 될 수 있어 atomicity 를 지킬 수 있다.

flush 된 log 들을 통해 init_db 시 Recovery 를 수행한다. 3 pass recovery algorithm 을 채택하였으므로, analysis pass – redo pass – undo pass 의 세 단계를 거치게 된다. 각 pass 에서 하는 일은 다음과 같다.

analysis pass : log 를 순회하면서 winner 와 loser 를 분류한다. 또한 각 log 들을 std::vector 에 넣어 redo pass 와 undo pass 에서 쉽게 순회할 수 있도록 한다.

redo pass : log 를 순회하면서 log 를 redo 한다. 이 때 log 의 LSN 과 page 의 LSN 을 비교하여 page 의 LSN 이 log 의 LSN 이상이라면 consider-redo 한다.

undo pass : log 를 역방향으로 순회하면서 loser 의 log 를 undo 해준다. 이 때 compensate log 는 이미 원래 연산의 undo 를 수행했다는 의미이므로 다시 undo 할 필요가 없다. Next Undo LSN 을 통해 보다 빠르게 undo pass 를 수행할 수 있으나 과제 spec 상 순회가 필요해 보여 구현하지 않았다.

redo 와 undo 는 physical 하게 구현하고 있으므로 직접 buffer manager 에게 table_id(원래는 file 식별자를 사용하나, 과제의 편의상 table_id 를 통해 file 까지 식별한다.)와 pagenum 을 통해 page 를 요청하고 해당 page 의 offset 부터 image 크기만큼 new image 를 memcpy 하거나(redo) old image 를 memcpy 한(undo) 이후 buffer manager 에게 page 를 돌려주며 상호작용하게 된다.

이후 truncate 한다. LSN 은 log manager 이 알고 있으므로 LSN 의 durability 또한 지켜지게 된다. 단 truncate 이후 logging 을 한번도 하지 않고 다시 crash 가 일어나면 durability 가 깨질 수 있으나, header 에 meta data 를 넣어 해결할 수 있다.

update log 의 경우 trx_abort 에서 사용하기 위해 각 trx 별로도 log 를 copy 하여 가지고 있다. 이렇게 구현한 까닭은 trx_abort 에서 사용할 log 가 log buffer 에 남아있지 않고 이미 flush 된 경우 다시 disk 에서 읽어오는 과정이 필요하여 비효율적이게 되기 때문이다. 때문에 abort 가 일어날 확률이 낮더라도 이를 대비하여 약간의 메모리와 시간을 더 사용하여 한번 flush 된 log 를 다시 read 하는 경우가 없도록 구현하였다. 현재 구현에서는 update log 외의 다른 log 들도 trx 별로 갖고 있으나, update log 만 갖고 있도록 구현하여도 충분하다.

In-depth Analysis

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

   많은 read-only transaction 들이 충돌 없이 접근한다면 이론상 하나의 transaction 이 접근하는 것과 같은 환경을 선사해 줄 수 있어야 한다. read-only 의 경우 record 에 여러 thread 들이 lock 없이 동시에 접근하여도 lost update 와 같은 문제가 발생하지 않기 때문이다. 이 때문에 lock_acquire 에서 shared lock 과 exclusive lock 의 두 가지 lock mode 를 분리하는 것이기도 하다.

   그러나 record lock 이 아닌 다른 lock 들에 대해서는 이러한 분리가 제대로 이루어지지 않았고, 이 때문에 read-only transaction 들이 lock 을 acqruie 하기 위해 wait 하는 시간이 길어지게 된다. 이러한 lock 의 대표적인 사례가 page latch 가 있다. index manager 의 find_leaf 함수에서 leaf node 를 탐색하는 과정은 탐색한 route 의 page 들을 수정하는 행위가 아니므로 shared lock 으로 충분하나, 현재 구현은 page latch 를 exclusive lock 으로 처리하고 있어 많은 transaction 들이 접근하는 header page 와 여러 internal page 의 latch 를 얻기 위해 오랫동안 wait 하게 된다. 이 overhead 를 줄이기 위해 다른 lock 에 대해서도 shared lock 과 exclusive lock 을 구분하여 사용하는 design 을 고려할 수 있다. 이 경우 문제 상황에서 wait 하는 데 걸리는 시간이 0 에 가깝게 되어 매우 효율적으로 동작하게 된다.

   또한, shared lock 으로 사용할 수 없으며, 모든 transaction 들이 접근하는 공유 자원에 대해서는 여전히 병목이 생길 수 있다. 대표적인 사례가 lock manager 이다. 모든 transaction 들은 lock acquire 를 통해 record lock 을 획득하고, 이 과정에서 lock manager latch 를 잡고 lock acquire 이 끝날 때 lock manager latch 를 풀어주게 된다.

   다행히 read-only 의 경우 shared lock 의 요청은 바로 lock 을 획득하게 되므로 긴 시간 lock manager latch 를 잡게 되는 경우는 없지만, 그럼에도

latch 를 조금 더 빠르게 풀어주기 위해 약간의 최적화를 추가한 design 을 고려해 볼 수 있다. trx manager latch 를 lock acquire 내내 잡고 있는 대신 trx lock 만을 획득해도 되는 것 처럼, lock manager latch 또한 해당 lock obj 에 해당하는 lock list 의 head 을 구한 후 head 에 lock 을 건 뒤 lock manager latch 를 풀어주어도 된다.

현재 구현에서는 lock list 의 tail 에 lock obj 를 추가하는 동작 정도만 있기에 오히려 문제 상황에서는 lock manager latch 를 풀고 head 의 lock 을 획득하는 과정에서 오버헤드가 생길 수도 있으나, 여기에서 과정이 얼마든지 늘어날 수도 있는 것이며, 전체가 공유하는 자원에 대해 거는 latch 를 최소화한다는 idea 자체는 상황에 따라 충분히 좋은 성능 향상을 얻을 수 있는 idea 이다. 특히 문제 상황은 아니지만 write operator 도 포함되는 경우 dead lock detection 동안 lock manager latch 를 계속 갖고 있는 것은 엄청난 낭비가 될 수 있을 것이다.

마찬가지로 buffer manager latch 에 대해서도 latch 를 갖는 시간을 최소화해야 한다. 그런데 현재 구현에서 buffer manager latch 와 page latch 를 모두 가진 상태로 eviction 을 진행하는데, 이는 disk 에 write 하는 동안 buffer manager latch 를 유지하고 있으므로 비효율적인 동작이 된다. disk 에서 read 하는 동작의 경우 이미 page latch 만을 갖고 상태에서 read 하고 있으나, eviction 의 경우 eviction 이후 LRU 정책에 따라 해당 page 를 head 로 옮기는 과정이 존재해 buffer latch 를 풀지 못하였다.

이에 eviction 의 경우 page 의 LRU 를 먼저 진행한 후, buffer manager latch 를 풀어준 후 disk write 와 read 를 수행하는 design 을 채택하여 buffer manager latch 를 잡으면서 system call 을 수행하는 일이 발생하지 않도록 할 수 있다.

이외에도 lock acquire 에서 lock 을 획득하는 조건과 lock release 에서 notify 하는 조건에 따라 이 과정에서 성능 상의 문제가 생길 수는 있으나, 현재 design 에서 shared lock 의 lock_acquire 조건 체크는 $O(1)$에 일어날

뿐더러 non-conflicting read-only 상황이므로 해당 상황을 고려할 필요는 없다.

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

Crash Recovery 측면에서 많은 transaction 들이 write 를 위해 접근하는 공유자원은 log buffer 이다. 모든 transaction 들이 log buffer latch 를 얻고자 할 것이고, 이 과정에서 성능상 문제가 생길 수 있다. 또한 현재 구현에서 log buffer latch 를 얻은 상태로 flush 를 진행하는데, 이는 4-1 에서 buffer manager latch 를 얻은 상태로 eviction 을 진행하는 것과 유사하게 비효율적인 구현이다.

이를 해결하기 위해 먼저 log buffer 에 write 할 때 log 들이 서로 다른 index 를 write 한다는 점을 관찰하면, log buffer latch 를 잠깐 잡아 해당 메모리를 할당해 주기만 한 후 latch 를 풀어주는 디자인을 떠올릴 수 있다. 즉, log size 에 따라 LSN 을 미리 증가시키고, 증가시키기 전 LSN 부터 log size 만큼 write 하면 모든 transaction 들이 버퍼의 서로 다른 위치에 write 하므로 write 과정이 thread safe 해지게 된다.

그러나 이 경우에도 write 한 log 가 flush 될 때 안정성을 보장할 수 없게 된다는 문제점이 여전히 존재하게 된다. log 를 write 하던 도중 flush 가 일어난다면 write 가 끝나지 않은 log 가 disk 에 적히는 일이 발생할 수 있다. 이를 막기 위해 생각해 볼 수 있는 디자인은 다음과 같다.

1. log 마다 lock 을 걸고 logging 이 끝나면 lock 을 풀어준다. flush 시 log 들을 하나하나 lock 을 획득한 후 write 해 준다.

log 들의 lock 을 하나하나 획득한다고 하였으나, 실제로 write 도 log 하나 단위로 이루어진다면 write 의 system call 의 호출 횟수가 매우 많아지므로 비효율적이다. 따라서 모든 lock 을 획득한 후 한번에 flush 하면 되는데, 이 경우 lock 을 획득하고 풀어주는 것 또한 overhead 이므로 다음과 같은 design 을 생각해 볼 수 있다.

2. write 가 끝나기 전 최대 log 의 개수를 올리고, write 가 끝난 후 끝난 log 의 개수를 올린다. flush 시 두 값이 일치할 때 까지 기다리고 0 으로 초기화한다.

이 경우 flush 되는 log 의 correctness 는 보장할 수 있으나, flush 되는 동안 추가적인 logging 이 불가능하다는 단점은 여전히 존재하게 된다. 추가적인 logging 은 다시 0 번부터 시작하도록 구현해 놓았으므로, flush 가 끝났는지를 flag 변수 등을 통하여 관리하며 대기해주어야 한다.

3. transaction 들이 log 를 갖고 있으므로, transaction 이 commit 혹은 rollback 될 때 한꺼번에 log 를 write 한다.

이 경우 log buffer 를 가질 필요 자체가 아예 사라지며, log manager 에서는 LSN 의 정보만을 가져와 현재 transaction 의 log 들에게 LSN 만큼의 offset 을 추가해주고 flush 해주면 되게 된다. LSN 값은 flush 이전에 update 해 주면 다른 transaction 들 또한 미리 log 들에게 offset 을 추가하고 flush 를 대기할 수 있게 되며, flush 의 순서가 달라지더라도 서로 다른 LSN 에 write 하므로 파일의 일부분이 0 으로 공백이 생길 수 있는 점 외에 다른 문제는 생기지 않게 된다.

그러나 이 design 은 eviction 에 대해 효과적으로 대처할 수 없다는 단점이 존재한다. buffer 크기가 충분히 커 eviction 이 매우 가끔 일어난다는 보장이 있을 경우에만 가능하거나, page 별로 log 를 추가로 관리하는 등의 추가적인 보완을 통해 eviction 이 일어났을 때의 risk 를 줄여야 할 것이다. 혹은 no-steal 정책을 통해 undo 할 필요가 없음을 가정해야 한다. 이러한 가정 하에서는 어차피 commit 되지 않은 log 는 disk 에 반영되지만 않는다면 redo 할 필요조차 없어 logging 할 이유도 없으니 recovery 에서도 효율적으로 동작할 수 있으며, trx 가 begin 부터 commit 까지 이어져서 logging 되므로 추가적인 최적화를 도입할 가능성도 높아질 것이다.

또 다른 단점도 존재하는데, transaction 의 commit 단위로 flush 가 이루어지므로 이를 달리 말하면 여러 transaction 들의 commit 을 모아서 flush 하기가 어렵게 된다. 이를 위해서는 log buffer 를 도입해야 하며, 결국 log buffer 에 write 하는 연산과 log buffer 를 flush 하는 연산 사이의 동기화를 보장해야 한다는 문제점이 여전히 존재하게 된다. 현재 디자인에서는 모든 commit 마다 flush 가 일어나므로 이 경우에는 큰 차이가 없으나, 현재 디자인은 commit 두개, commit 세개 마다 flush 가 일어나도록 수정하여 최적화를 도모하는 방식이(단, 이 경우 durability 가 완전히 보장되지 않는다.) 쉽게 적용될 수 있는 반면, 이 design 의 경우 이러한 최적화를 쉽게 구현할 수 없게 된다.