Database System 2020-2

Final Report

Class Code ITE2038-11800

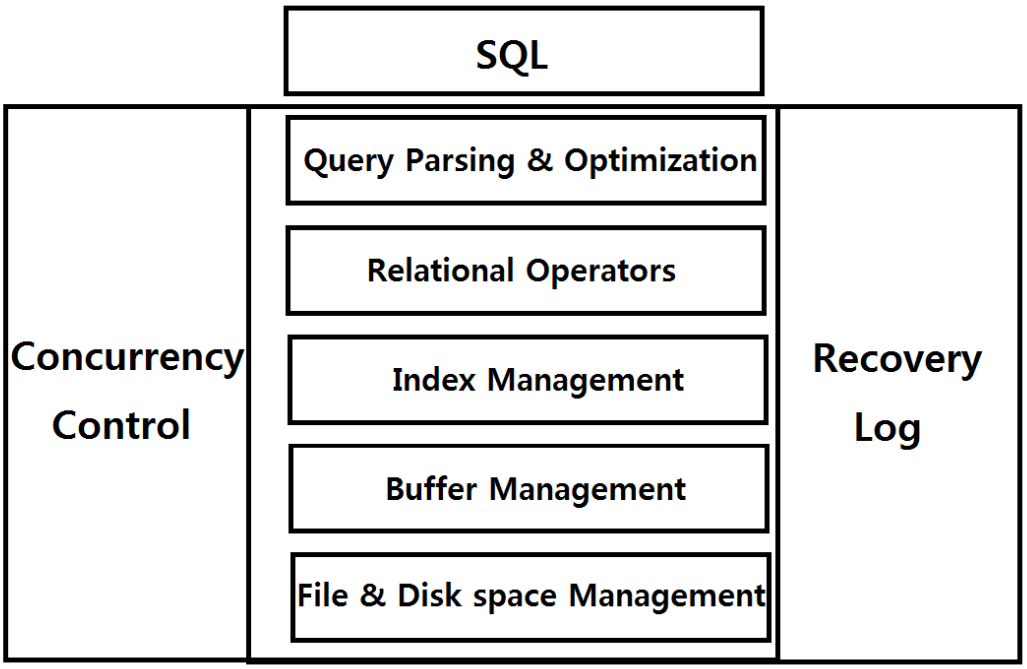
Student Number 2016025650

Name 윤민석

Table of Contents

Overall Layered Architecture

DBMS는 여러 계층들이 존재하고, 각 계층들간의 상호작용을 통해 작동합니다. 자세한 계층들의 설명은 다음과 같습니다.



사용자는 SQL Query를 이용해 요구사항을 DBMS에 전달하고, DBMS는 해당 쿼리를 Query Parsing & Optimization 계층과 Relational Operators 계층을 사용하여 어떠한 레코드에 접근할 것인지를 판단합니다.

접근하고자 하는 레코드의 key값을 Index Manager 계층에 전달하면 Index Manager 계층은 DB의 logical한 자료구조를 활용하여 해당 key의 위치를 파악합니다. 본 프로젝트에서는 B+Tree 구조로 key를 관리합니다.

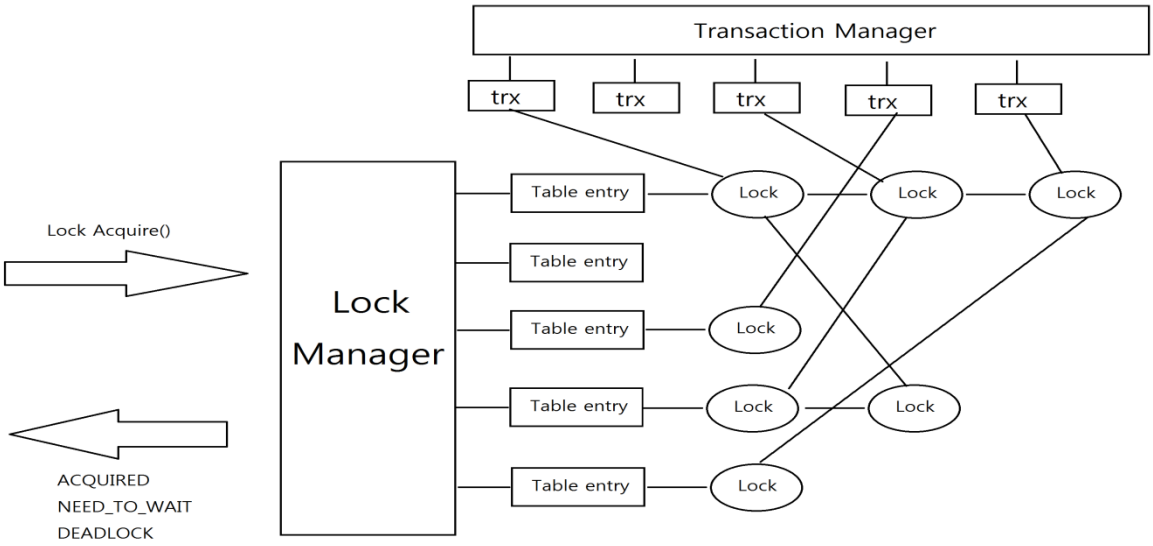
그러나 해당 key가 저장된 disk에 바로 접근하여 읽거나 쓰게 된다면 시간이 많이 걸리는 디스크 IO의 빈도가 높아져 성능이 크게 하락합니다. 이를 보완하기 위해 Buffer Manager 계층을 사용합니다. Buffer Manager 계층은 File Manager로부터 받은 페이지와 Index Manager가 사용한 후의 페이지를 메모리 상에 저장해 놓음으로써 이후 같은 페이지에 대한 접근을 할 때 디스크 IO대신 메모리에 접근하게 되고 이를 통해 디스크 IO의 빈도를 줄여주는 역할을 합니다.

File & Disk space Manager는 레코드의 physical한 부분을 담당합니다. Index Manager로부터 레코드에 대한 요청이 들어오면 File Manager는 해당 레코드가 위치한 실제 디스크에 접근하여 메모리로 레코드가 위치한 페이지를 올려주는 역할을 합니다.

Concurrency Control Implementation

DBMS는 여러 사용자가 동시에 사용할 수 있습니다. 그러나 만약 이러한 상황에서 Concurrency Control에 대한 별다른 기법을 적용하지 않는다면 DBMS가 만족 해야 하는 성질인 ACID를 헤치는 결과가 나올 수 있습니다.

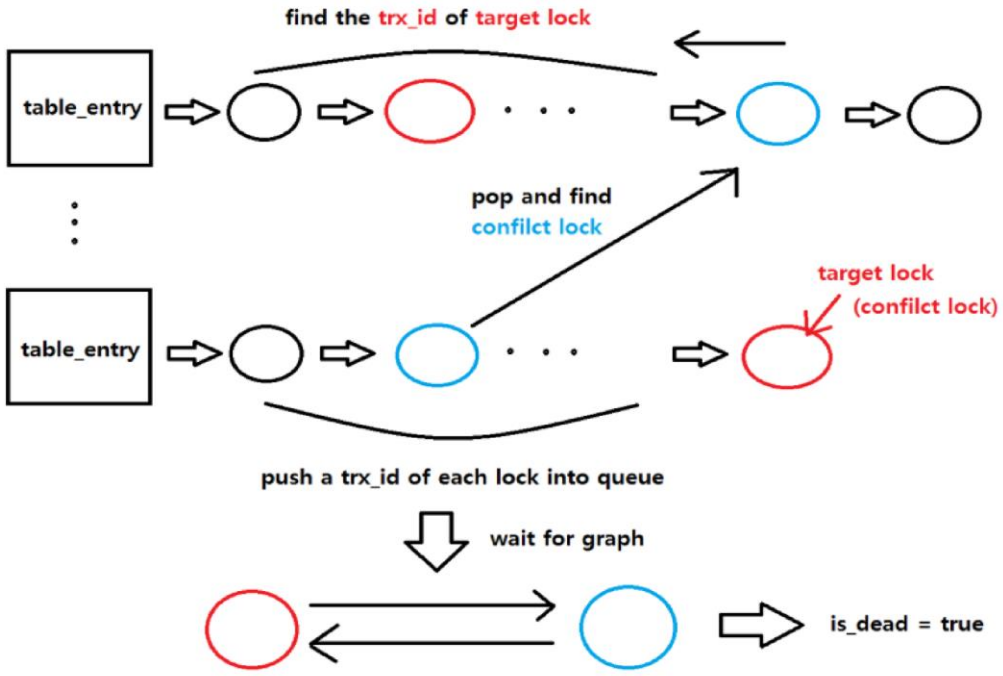
이번 프로젝트에서는 Concurrency Control을 위해서 Strict 2 Phase Locking을 적용하였습니다. 특정 레코드에 접근하기 위해서는 해당 레코드에 대해서 Lock을 획득하여야 합니다. 획득한 Lock은 트랜잭션이 끝날 때까지 보유하다가 트랜잭션이 commit 혹은 abort된다면 그 동안 획득해온 Lock들을 모두 풀어주는 방식입니다. 즉 프로젝트에서 제공하는 API인 db_find와 db_update는 내부적으로 레코드에 대한 Lock을 얻은 후에 레코드에 대한 read, write를 진행할 수 있습니다.



Lock을 얻는 과정을 관리하기 위해서 Lock Manager라는 객체와 Transaction Manager라는 객체를 사용합니다.

Lock Manager는 table entry라는 구성요소로 이뤄져 있습니다. 이 객체들은 (table id, key)로 구별됩니다. 트랜잭션이 레코드에 대해 Lock을 획득하고자 한다면 Lock Manager는 해당 레코드의 table id와 key를 사용하여 그에 대응하는 table entry 객체에 접근하고 이 객체가 가진 멤버 변수들을 기반으로 Lock을 달 수 있는 지를 검사한 후 그 결과를 트랜잭션에 알려주게 됩니다. 이미 레코드에 다른 트랜잭션이 write를 하는 중이라면 기다려야 한다는 신호를 보내고, 만약 Lock을 달았을 때 서로 다른 트랜잭션이 서로를 기다려야 하는 Dead Lock 상황이 발생한다면 Locking을 취소하고 이에 대한 Dead Lock 신호를 보냅니다.

Dead Lock 여부를 판단하는 방법은 다음과 같습니다.



Lock을 달고자 하는 table entry에 대해서 새로운 Lock의 앞에 달려있는 Lock들의 트랜잭션들의 정보를 확인합니다. 이 트랜잭션들이 어떤 Lock에서 대기 중인지를 파악하고 대기 중인 Lock의 앞에서 기다리고 있는 트랜잭션이 새롭게 생긴 Lock의 주인이 되는 트랜잭션과 동일한 것이 있는 지를 확인하고, 만약 그렇

다면 Dead Lock이 발생한 것으로 간주합니다.

이러한 검사를 좀 더 쉽게 하기 위해서 Transaction Manager라는 객체를 사용하고 이 객체는 trxNode라는 구성요소로 이뤄져 있습니다. 이 객체들은 각 트랜잭션에 대응하는 Node들이고 해당 트랜잭션이 보유한 Lock들을 링크드 리스트형태로 보유하며 현재 어떤 Lock에서 해당 트랜잭션이 대기 중인지에 대한 정보도 가지고 있습니다.

Concurrency Control 기법은 Lock Manager와 Transaction Manager에 대한 구현뿐만 아니라 Index와 Buffer layer에도 걸쳐서 구현되었습니다. Index Manager가 key를 탐색하면서 여러 페이지들에 접근 해야 하고, 이 때 페이지들이 서로 다른 트랜잭션에 의해 동시에 접근하게 됩니다. 만약 여러 트랜잭션이 별 다른 제약 없이 같은 페이지에 동시에 write를 하게 방치한다면 둘 중 하나의 트랜잭션의 결과물이 반영되지 않을 수 있기 때문에 page 자체에 대해서 mutex를 부여하여 Concurrency Control을 진행합니다.

또한 Buffer pool 자체에 대해서도 page eviction과 관련하여 mutex가 요구되었습니다. 구체적으로 살펴보면 다음과 같은 상황이 있을 수 있습니다. 어떠한 트랜잭션이 버퍼 상에 존재하는 페이지에 대해서 대기하고 있을 때 만약 버퍼 풀에 대한 mutex가 존재하지 않는다면 다른 트랜잭션의 요청에 의해 해당 버퍼 페이지가 evcition의 대상이 될 수 있고, 이로 인해 원하지 않는 데이터를 읽게 될 수 있습니다. 이를 방지하기 위해 버퍼 풀에 대한 mutex를 적용하여 해당 mutex를 얻은 트랜잭션만 LRU 리스트를 수정할 수 있도록 제한하였습니다. 즉, 페이지를 기다리고 있는 트랜잭션은 버퍼 풀의 mutex를 보유한 채로 기다리고 있기 때문에 원하는 페이지가 eviction될 걱정을 하지 않아도 됩니다.

Crash-Recovery Implementation

시스템이 예상치 못하게 종료되었을 경우 commit 된 트랜잭션의 결과물들은 DB에 반영되어있어야 하고, commit 되지 못한 트랜잭션들의 결과물은 DB에 반영되어있지 않아야 합니다.

이러한 성질을 만족하기 위해서 Recovery가 필요하게 되었습니다. Recovery는 Log Manager와 Recovery Manager에 의해서 구현되었습니다.

Log Manager는 로그의 발행을 관리하는 객체입니다. 로그의 종류로는 begin, update, commit, compensate, rollback과 같은 로그들이 존재합니다. 각각 트랜잭션이 시작할 때, update를 할 때, commit할 때, abort과정에서 update과정을 되돌릴 때, 더 이상 트랜잭션에 되돌릴 update가 없을 때 발행하는 로그입니다.

Log Manager는 로그들을 임시적으로 보관할 로그 버퍼와 트랜잭션 테이블이라는 객체들로 구성되어있습니다. 먼저 로그 버퍼는 디스크 IO를 줄이기 위해서 트랜잭션들이 만든 로그들을 바로 디스크로 보내는 것이 아닌 로그 버퍼에 보관해 놓을 수 있게 해줍니다. 트랜잭션 테이블은 로그를 발행하는 것에 사용되는 트랜잭션 고유의 정보를 가지고 있습니다. 예를 들어 해당 트랜잭션이 마지막으로 발행한 로그의 시퀀스 넘버, 해당 트랜잭션이 발행해온 업데이트 로그들의 시퀀스 넘버 등을 보관하고 있고 이를 기반으로 새로운 로그의 발행을 하게 됩니다.

트랜잭션 테이블 이외에도 로그 발행에 영향을 끼치는 데이터가 존재합니다. 메타 로그라는 이름을 가진 데이터로써 로그 파일의 맨 앞에 위치합니다. 이 데이터는 DBMS가 어떤 로그 시퀀스 넘버부터 로그를 발행해야 하는지, 어떤 트랜잭션 ID부터 트랜잭션을 Begin해야하는 지에 대한 정보를 가지고 있습니다. 로그 매니저 객체가 처음 생성될 때 로그 파일의 맨 앞에 있는 메타 로그를 읽고 이를 기반으로 새로운 로그의 발행이 시작됩니다.

로그 버퍼에 쌓여있는 로그가 디스크로 보내지는 경우는 세 가지로 나뉩니다. 첫 번째로 트랜잭션이 commit된 경우입니다. 트랜잭션이 commit되었다면 crash

이후에도 해당 트랜잭션의 결과물은 DB에 남아있어야 합니다. 이를 위해서는 먼저 로그들이 디스크에 남아있어야 하기 때문에 commit 이전에 로그 버퍼에 남아있는 로그들을 모두 flush 시킵니다. 두 번째로 Buffer Manager에 들어있는 페이지가 디스크로 eviction되는 경우입니다. 만약 페이지가 디스크에 반영이 되었는데 이것에 commit되지 않은 트랜잭션의 결과물이 들어있다면 Recovery 과정에서 이를 바로 잡아야 합니다. 그렇기 때문에 페이지가 디스크에 입력되기 전에 로그 버퍼에 들어있는 로그들을 먼저 디스크에 반영시킵니다. 세 번째로 로그 버퍼에 더 이상 남은 공간이 없을 경우가 있습니다.

Recovery Manager는 저장되어있는 로그 파일을 바탕으로 진행합니다. Recovery 매니저는 로그 매니저의 로그 버퍼와 별개의 로그 버퍼를 가집니다. 로그 매니저의 로그 버퍼는 오직 디스크에 데이터를 보내는 방향만 가능하지만 Recovery 매니저의 로그 버퍼는 오직 디스크로부터 데이터를 불러오는 방향으로만 작동합니다.

Recovery 매니저는 자신의 로그 버퍼로 로그 파일의 처음부터 끝까지 스캔하며 commit 혹은 rollback 로그의 존재 여부로 winner 트랜잭션과 loser 트랜잭션을 구분합니다.

redo 단계 또한 Recovery 매니저의 로그 버퍼를 활용하며 다시 로그 파일의 처음부터 끝까지 redo를 진행합니다. 만나는 update로그와 compensate로그를 확인하며 redo를 진행하고, 해당 로그에 적혀있는 테이블이 open되어있는 지 여부를 Hash 테이블을 이용하여 확인한 후 open되지 않았다면 open을 먼저 하고 redo를 진행합니다.

Undo 단계를 들어가기 이전에 먼저 어떤 loser 트랜잭션부터 undo를 할 것인 것 정해야 합니다. 이를 위해 우선순위 큐를 사용합니다. 각 loser 트랜잭션들이 undo 해야 하는 로그의 LSN값을 담고 있는 Node를 생성하고 이를 가장 큰 값이 pop되는 우선순위 큐에 push합니다. 이후 우선순위 큐에서 pop을 하며 undo할 loser 트랜잭션을 고르고, undo를 진행하며 로그 매니저를 활용하여 compensate로그를 발행합니다. 이 때 로그 매니저의 로그 버퍼와 recovery 매니

저의 로그 버퍼는 별개의 것임을 주의해야 합니다.

이후 undo 단계까지 마무리하면 open 되어있던 테이블들을 close 시킨 후 메타 로그의 정보를 최신화합니다. 최신화된 메타 로그는 이후 발생할 수 있는 새로운 Crash-Recovery의 기준이 될 것입니다.

In-depth Analysis

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

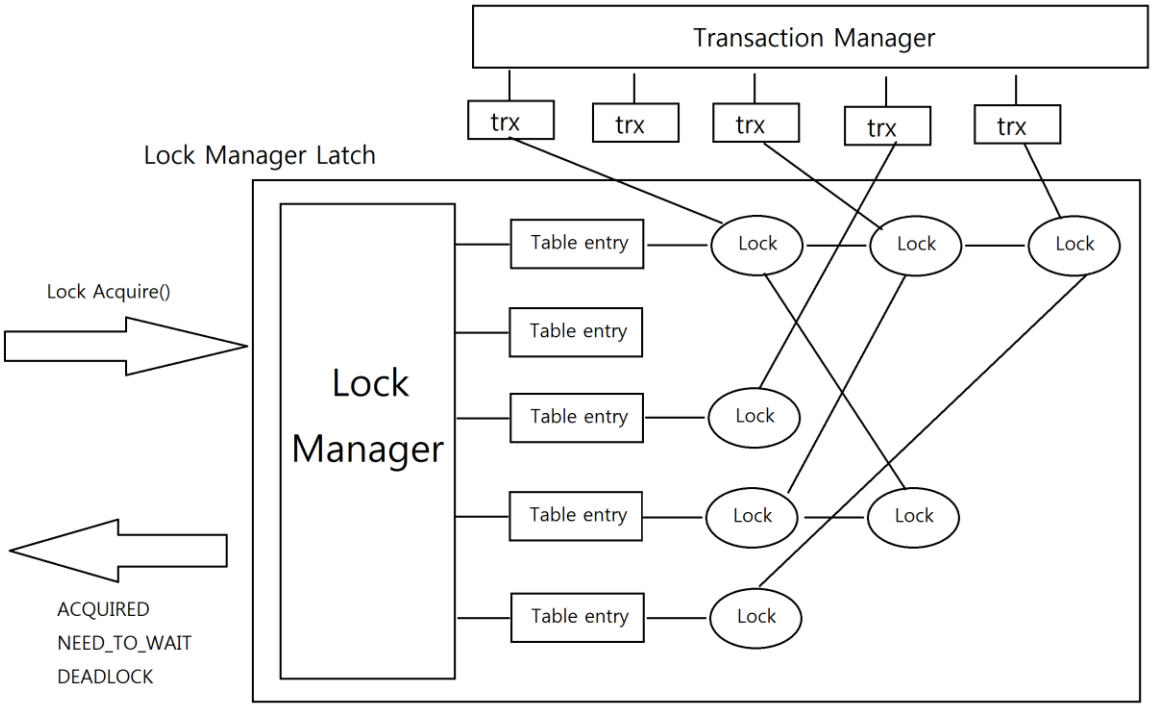
이론적으로 Read만 하는 트랜잭션들이 아무리 많아도 서로 충돌할 일이 없기 때문에 아무리 많은 트랜잭션이 동시에 read를 해도 성능 상에 문제가 없어 보일 수 있습니다.

그러나 Concurrency Control을 위해 Lock Manager 객체를 여러 트랜잭션이 동시에 사용할 수 있고 이를 위해 Lock Manager에 대해서 mutex를 걸어서 사용합니다. 예를 들어 Lock 을 얻고 해제하는데 사용하는 lock_acquire()함수와 lock_release()함수는 함수의 처음 시작 부분에서 pthread_mutex_lock()을 통해 Lock Manager 전체에 대한 mutex를 획득하고 return 이전에 해당 mutex를 풀어주는 방식으로 동작합니다.

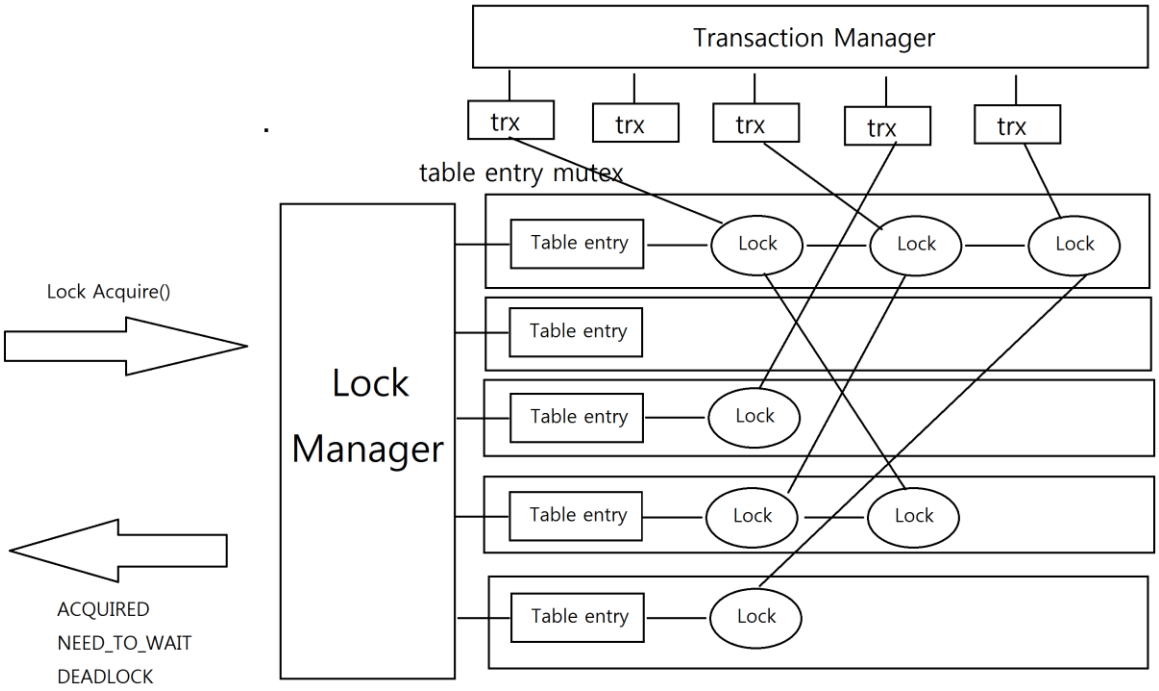
즉, 매우 많은 트랜잭션이 서로 충돌하지 않는 다는 가정하에 read 요청을 동시에 한다면 이론 상으로는 각자 바로 Lock을 획득할 수 있지만 실제 구현에서는 Lock Manager의 mutex를 획득하는 행위 때문에 기다리는 시간이 추가되고, 이는 성능의 하락으로 이어질 수 있습니다.

이를 해결하기 위해서 Lock Manager에 대한 mutex가 아니라 Lock Manager의 구성요소인 Table entry에 대한 mutex를 거는 것을 생각해보았습니다.

(기존의 Lock Manager 사용 시 Lock Manager 전체에 대한 mutex를 획득)



(Lock Manager 전체가 아닌 개별적인 table entry에 대한 mutex 획득으로 변경)



이렇게 table entry 각각에 대한 mutex를 거는 것으로 디자인을 변경한다면 서

로 충돌하지 않는 트랜잭션들이 서로를 기다릴 필요 없이 mutex를 바로 획득하고 작업을 수행할 수 있을 것 같다는 생각을 하였습니다.

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

서로 다른 레코드들에 대해서 매우 많은 write가 발생한다면 그만큼 서로 다른 페이지에 접근하는 빈도가 높아지고 이는 많은 수의 page eviction이 발생하는 것을 의미합니다.

WAL정책에 따르면 Page eviction이 발생하기 전에는 필수적으로 로그 버퍼를 디스크로 flush해주는 작업이 필요합니다. 그 이유는 Crash가 발생하였을 때 commit되지 않은 트랜잭션의 결과물이 DB에 반영되어있더라도 디스크에 존재하는 로그를 바탕으로 이를 되돌려주기 위함입니다.

이번 프로젝트에서 WAL을 구현하기 위해 로그 매니저에 대한 mutex를 사용하였습니다. 그리고 이러한 로그 매니저의 mutex를 로그를 발행할 때와 로그 버퍼의 데이터를 디스크로 보낼 때 잡는 방식으로 구현하였습니다.

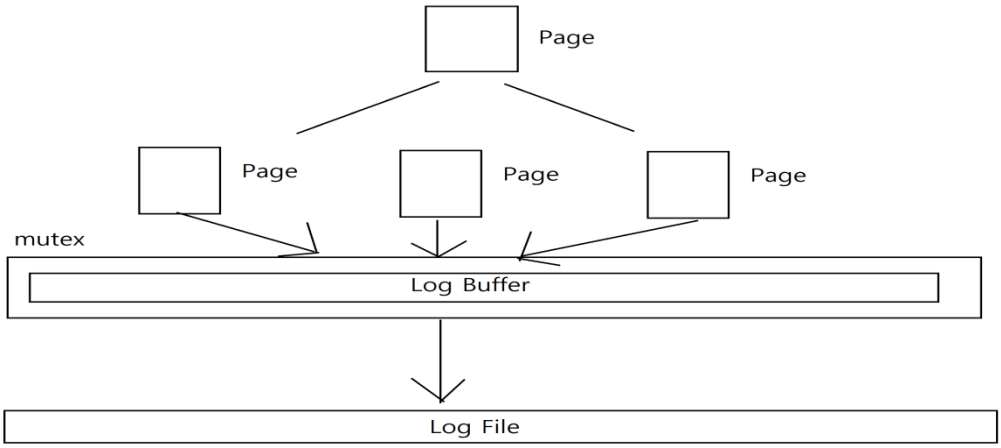
그러나 매우 많은 트랜잭션들이 write 오퍼레이션을 진행한다면 모든 트랜잭션이 로그를 발행할 때마다 로그 매니저의 mutex를 잡아야 하고, 또한 빈번한 page eviction이 발생하여 로그 버퍼의 flush마다 또 mutex를 잡아야 하는 상황이 발생할 수 있습니다. 이러한 상황을 좀 더 빠르게 대처하기 위해서 mutex의 분배를 좀 더 세분화하는 것을 생각해보았습니다.

이 디자인은 로그 버퍼를 좀 더 세분화하여 각 페이지마다 자신만의 로그 버퍼를 가짐으로써 서로 다른 mutex에 의해 로그가 발행되고, 각각의 로그 버퍼를 디스크로 보낼 때는 모든 로그 버퍼가 하나의 통일된 mutex를 사용하는 방식입니다.

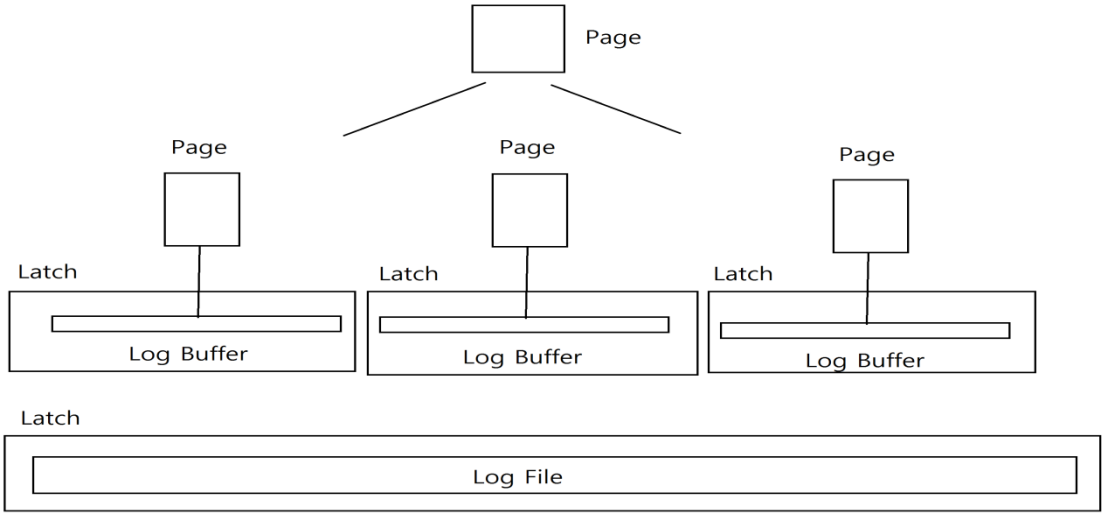
예를 들어 현재 A와 B라는 페이지가 존재하는 상황을 가정해보겠습니다. 여

러 트랜잭션이 A와 B라는 페이지에 write하면서 그에 대응하는 로그들을 생성합니다. 이 때 A와 B는 각자 구별된 로그 버퍼를 가지기 때문에 트랜잭션들이 서로 다른 페이지에 대한 오퍼레이션을 수행했다면 로그를 발행하기 위해서 서로를 기다리는 일이 발생하지 않습니다. 이 때 page eviction이 발생하여 A페이지가 디스크로 이동해야 한다면 로그 버퍼의 디스크 IO를 위한 별도의 mutex를 획득한 후 로그 파일로의 flush를 진행합니다. B페이지를 이용하는 트랜잭션은 B페이지가 eviction의 대상이 아니라면 A페이지와 별개로 로그를 발행하는 작업을 계속할 것입니다.

(기존의 구현)



(로그 버퍼의 세분화)



트랜잭션이 로그를 발행한다면 각 페이지의 로그 버퍼에 대해서만 mutex를 획득하고 로그 버퍼에 데이터를 입력합니다.

Page eviction이 발생하면 해당 페이지의 로그 버퍼에 대한 mutex와 로그 파일에 대한 mutex를 획득한 후 로그 버퍼의 flush를 진행합니다. Flush 과정에서 다른 페이지들의 로그 버퍼에 대한 mutex는 잡지 않았기 때문에 다른 트랜잭션들이 Flush 중인 페이지를 사용하지 않는다면 다른 페이지들에 대한 로그의 발행은 정상적으로 진행됩니다.

물론 이와 같이 페이지 별로 로그 버퍼를 구별한다면 로그 파일에 쌓이는 로그들의 시간 순서는 하나의 방향이 아닐 수 있습니다. 즉 어떤 페이지의 로그가 먼저 생성되었지만 다른 페이지의 로그가 먼저 로그 파일로 flush될 가능성이 존재합니다. 하지만 같은 페이지를 대상으로 한 로그는 시간순서대로 쌓여있기 때문에 페이지에 대한 Recovery에서 문제가 되지 않을 것이라고 생각하였습니다.