Database System 2020-2

Final Report

ITE2038-11800

2017029616

박병현

Table of Contents

Overall Layered Architecture

[db.c] 가장 상위 계층이 구현된 곳입니다. 사용자에게 제공하는 API들이 존재하여 사용자에게 DB에 접근할 수 있는 함수를 제공합니다.

[bpt.c] index layer에 해당하는 곳으로, 메모리에 존재하는 값을 변경하거나 읽어올 수 있는 곳입니다. 또한 Tree를 구조를 관리하는 역할을 수행합니다. Table에 대한 관리도 수행하는데, table을 open 할 때 table에 대한 정보를 저장하고, 파일이 새로 생성된 table일 경우 header page를 설정해주어 해당 테이블을 사용할 수 있게 합니다.

[buffer.c] buffer management layer에 해당하는 곳으로, index layer에서 디스크에 존재하는 파일을 요청하거나 파일에 특정 페이지를 쓰고 싶을 때, 메모리에 미리 내용을 저장하여 그 속도를 빠르게 만듭니다. index layer에서 요청했던 페이지를 버퍼에 임시로 저장해 놓고, 해당 페이지의 재요청을 조금 더 빠르게 응답할 수 있게 합니다. 또한 index layer에서 파일에 write하기로 한 내용을 곧바로 write하지 않고 버퍼에 임시로 저장해 두어, 디스크와 메모리의 이동을 감소시켜 성능을 향상시킵니다.

[file.c] disk management layer에 해당하는 곳입니다. 버퍼에 존재하지 않은 페이지를 요청 시 파일에 직접 접근해서 해당 페이지를 읽어옵니다. 또한 버퍼가 가득 차 eviction이 일어날 때 파일에 직접 내용을 적는 역할을 담당합니다. 추가로 파일에 page가 부족할 때 새로운 페이지를 생성하고, page가 쓸모가 없어지면 free page로 변경해주는 역할을 담당합니다. 주된 역할을 file에 직접 접근해 파일과 관련된 직접적인 작업을 수행합니다.

다음은 수직적인 layer architecture 와 함께 동작하는 layer입니다.

[trx.c] 트랜잭션을 begin하고 commit하는 작업을 수행합니다. 또한 현재 수행중인transaction들에 대한 정보를 관리하는 transaction management작업을 담당합니다.

[locktable.c] 트랜잭션의 관리를 도와 record 단위의 lock을 생성 및 관리하는 역

할을 합니다. 트랜잭션이 deadlock을 유발하지는 않는지 확인합니다. transcation이 record에 share mode인지 Exclusive인지 판단하여, 트랜잭션 간 순서를 적절하게 scheduling합니다.

[log.c] 트랜잭션이 operation을 수행할 때, 어떤 operation이 수행되었는지에 대한 정보를 저장합니다. 또한 트랜잭션의 begin과 commit을 로그 파일에 기록합니다. 만약 Database에 문제가 생겨 예상치 못한 crash가 일어나면, 해당 로그파일을 활용하여 수행했던 operation들을 redo및 undo를 수행하는 역할을 담당합니다.

Concurrency Control Implementation

이번 과제에서 저희는 PCC중 하나인 strict 2PL방식을 이용해 concurrency control을 구현하였습니다. 이 구현의 중요한 목적은 conflict serializable하지 않은 스케줄을 locking을 이용해 conflict serializable하게 진행할 수 있게 하는 것 이였습니다. 그러기 위해서는, conflict한 operation을 수행하는 트랜잭션들 중 몇 개를 대기하게 하여 각각의 operation들이 serializable 하게 수행할 수 있게 하였습니다. 트랜잭션들의 operation의 대기과정은 record단위의 lock을 acquire할 수 있는지의 여부로 판단하였습니다.

lock의 획득과정

저는 이러한 과정을 locktable.c파일에 존재하는 lock_aquire이라는 함수를 통해 구현하였습니다. 어떠한 트랜잭션 operation을 수행하고자 하는 레코드를 먼저 찾으면, 해당 레코드에 접근하기 직전, lock_aquire()을 호출하였습니다. 그 이후의 수행과정은 다음과 같습니다.

1. lock_aquire에서는 해당 레코드에 접근했던 트랜잭션 중 commit이 아직 실행되지 않은 상태의 트랜잭션이 존재하는지 확인합니다. 트랜잭션들이 각 레코드에 처음 접근하는 경우에는, 해당 레코드의 내용을 복사하여 해쉬테이블에 집어넣고, 복사된 레코드가 대기하고 있는 trx들의 lock object를 가리키는 방식으로 트랜잭션의 대기과정을 구현하였습니다. 즉 어떠한 트랜잭션이 특정 레코드에 접근하려 시도하면, 먼저 해쉬테이블에 해당 레코드가 존재하는 지 확인하고, 해당 레코드가 어떤 특정 lock object를 가리키는 과정을 통해 진행됩니다.
2. 만약 해쉬테이블에 operation을 수행하고자 하는 레코드가 존재하지 않는

다면, 어떠한 트랜잭션도 해당 레코드에 접근한 적 없거나, 접근한 적이 있으나 그러한 트랜잭션들이 모두 commit된 상태 이므로, 해당 레코드의 lock을 acquire할 수있습니다. lock을 acquire했다는 것은 더 이상 conflict 관계인 operation이 해당 레코드의 접근이 막힌 것이므로, 이제 실제로 해당 레코드에 접근해 값을 읽거나 쓸 수 있습니다.

3. 만약 다른 트랜잭션이 이미 레코드를 사용했지만 아직 commit되지 않은 상태이고, 해당 트랜잭션이 해당 레코드에 접근하려는 operation과 새로 lock의 획득을 시도하는 트랜잭션의 operation이 서로 conflict관계인 operation이라면, 해당 레코드에 더 늦게 접근한 트랜잭션은 레코드에 operation을 수행하기 전 대기해야 합니다. 이 과정을 통해 serializable한 schedule을 생성할 수있습니다.

추가로, 레코드에 lock을 이어줄 때 생성되는 lock object의 연결은 다음 두가지입니다.

1. 레코드로부터 이어져있는 lock object의 대기 연결
2. 동일한 트랜잭션끼리 생성한 lock의 연결.

1번의 경우는 앞서 설명한 conflict serializable하게 scheduling을 하지 못하여 dirty read or write이 발생하는 일을 피하기 위해 구현된 것입니다. 2번의 경우는 트랜잭션의 commit or abort 진행 시 해당 트랜잭션이 연결한 lock을 전부 해제해야 하기 때문에 연결 되어있습니다. 해당 정보는 transaction manager에 해당하는 trx.c에 구현되어 있습니다. 그러한 정보의 관리는 트랜잭션에 대한 정보를 담는 해쉬테이블에서 트랜잭션들이 생성한 lock을 가르킬 수 있는 정보를 담는 방식을 통해 실행하였습니다. lock_acquire에서는 이렇게 생성되는 여러 트랜잭션간의 관계와 lock object들 간의 관계를 관리하기위한 역할도 수행합니다.

추가로, 트랜잭션이 특정 레코드에 lock_acquire을 시도할 시 또다른 트랜잭션이 동시에 lock acquire을 시도하면 conflict인 operation이 레코드에 동시에 접근할 수 있는 경우가 발생하므로, 해당 작업은 한 트랜잭션만 진행할 수 있게 lock_table_latch라는 mutex를 통해 잠궈 놓는 작업이 필수적입니다.

Transaction의 대기과정

lock_을 acquire하지 못하고 대기상태로 들어가야 한다면, 우선 해당 record를 찾기 위해 쥐고 있던 page lock을 unlock해주어 다른 트랜잭션이 해당 페이지에 접근할 수 있게 합니다. 그 후 lock_wait()이라는 함수를 통해 해당 레코드에 lock object를 대기시켜 두고, 다음 차례가 올 때까지 대기합니다.

대기 상태에서 일어나거나, 처음 레코드에 lock을 이을 때 신경 써야 할 점은, 앞에 대기하고 있는 다른 lock object를 생성한 트랜잭션이 수행하려는 operation 혹은 이어야 할 트랜잭션이 수행하고자 하는 operation들이 어떠한 mode를 가지고 있는 지의 여부입니다. 특히 이어져 있는 lock object들이 연속적으로 shared mode라면, 전부 한번에 lock을 acquire할수있기 때문에 주의가 필요합니다. 따라서 저는 lock을 이을 때 가장 뒤에서 대기하고 있는 lock부터 가장 앞에서 대기하고 있는 lock까지 진행하면서 만약 전부 shared mode로 lock을 이어준 상태이고, 본인의 lock역시 shared mode면 곧바로 lock을 acquire할 수 있게 구현하였습니다. 또한 대기상태를 종료하고 lock을 shared mode로 acquire할 수 있을 때 뒤에 대기하고 있는 lock의 mode를 확인하여 share mode면 추가로 대기상태를 종료할 수 있게 하였습니다.

lock의 해제과정

strict 2PL에 의해 lock의 해제는 commit시에 한번에 일어나게 됩니다. 따라서 lock_table_latch를 잡은 후, 해당 트랜잭션에 연결된 모든 lock을 lock_release()함수를 통해 해제하는 방식으로 진행합니다. 해제 시 해제한 lock 앞뒤로 lock이 존재하는지 유무를 확인하여 적절한 방식으로 남아있는 lock들과 record간의 연결을 재설정하고 lock을 해제해줍니다.

deadlock detection 및 abort과정

deadlock detection: lock object를 레코드에 연결을 시도할 때, 해당 operation이 deadlock을 유발하지는 않는지 확인하는 과정이 필수적입니다. deadlock을 확인하기 위해서는 각 트랜잭션끼리의 대기 관계를 확인해야 합니다. 저는 각 트랜잭션들의 transaction id(trx_id)를 이용해 wait-for graph를 만들었습니다. 각 트랜잭션들의 대기관계를 표현하기 위해 2차원 배열의 adjacency matrix를 생성하여 directed graph의 edge 표현했습니다. 예시로, trx1이 trx2를 기다린다면, arr[1][2]의 값을 1로 변경하여 trx id 1을 가진 트랜잭션이 trx id 2를 가진 트랜잭션을 대기하고있다는 상황을 표현하였습니다.

하지만 이러한 방식은 배열의 index의 개수가 한정적이라는 한계가 존재합니다. 따라서 데이터베이스에 500개 이상의 트랜잭션이 동시에 진행되지 않을 것이라는 가정하에, 500(배열의 크기)의 모듈러 연산을 통해 index를 표현하였습니다. 그 후 DFS방식을 통해 트랜잭션간 cycle을 형성하지는 않는지 확인합니다.

DFS의 search과정에서, 처음 해당 트랜잭션을 의미하는 vertex를 지날 때 time1을, 더 이상 진행할 edge가 없는 vertex에 도착해 되돌아올 때 time2을 기록합니다. 만약 새로 접근한 트랜잭션이 search과정에서 방문한 적이 있어 time1이 기록되었지만 time2가 기록되지 않았으면 기존의 vertex를 기준으로 새로 search를 진행한 vertex가 ancestor이라고 볼 수 있으므로, cycle이 형성되어 deadlock이라고 판단합니다.

그 후 abort를 진행해 해당 트랜잭션이 연결한 lock을 전부 해제합니다. 이 때, write operation을 수행했던 record에 대해서는 value를 되돌려놓는 작업이 필요합니다. 이 작업은 트랜잭션 테이블 상에 변경한 value들을 linked list형식으로 저장하여, abort발생시 해당 리스트에 접근하여 값을 되돌려 놓을 수 있게 하였습니다.

이렇게 저는 lock table을 관리하는 계층(lock_table.c)이 transaction을 관리하는 계층(trx.c)의 도움을 받아 strict 2PL방식으로 record lock을 관리하고, deadlock을 detect 및 abort를 수행할 수 있게 구현하였습니다.

Crash-Recovery Implementation

저희는 No force & Steal 정책에 기반한 crash recovery를 구현하였습니다. 그러기 위해서는 recovery시 analysis, redo 및 undo과정으로 진행되는 three pass algorithm을 기반하여 과제를 수행했습니다.

Log record의 발급 및 저장

Recovery를 수행하기 위해서는 operation마다 로그에 operation에 관한 내용을 쓰는 작업이 필수적입니다. 저는 이러한 작업을 index management layer(bpt.c)에서 update가 수행될 때 마다 log management layer(log.c)에 존재하는 make_update_log를 통해 수행했습니다. 뿐만 아니라 transaction management layer(trx.c)에서 trx_begin(), trx_abort(), trx_commit() 실행시마다 log management layer에 존재하는 함수를 호출하여 log를 생성 후 쓰는 작업을 수행하였습니다. 로그를 생성하고 쓸 때에는 log buffer에 먼저 저장한다음, commit 및 abort가 일어날 때마다 log management layer에 있는 flush_log()를 통해

Log buffer 관리

log buffer의내용을 log file에 쓰는 작업을 수행합니다. log buffer에서 log file에 flush가 일어날 땐 항상 연속적으로 수행된 operation들이 한번에 flush되어야 합니다. 따라서 flushed_index 와 current_log_index를 관리하여 buffer에 연속적인 log record를 flush 할 수 있게 하였습니다.

Recovery-Analysis

DB가 crash가 난 후 재시작시에 recovery가 진행됩니다. Recovery가 진행 될 때엔 winner과 loser을 가려내는 작업이 선행되어야 합니다. 이 과정이 analysis pass에서 진행됩니다. log file에 첫 레코드부터 레코드의 끝까지 이동하며 레코드

들을 조사합니다. 저는 모든 begin 레코드는 각 트랜잭션 마다 한번씩만 발급한다는 특징을 이용하여 해당 레코드를 메모리(losers list)에 저장하였습니다. 하지만 만약 로그파일을 스캔 중 commit 혹은 rollback에 해당하는 레코드가 발견된다면, 해당 레코드와 같은 트랜잭션 id(trx_id)를 갖는 레코드를 begin레코드들이 존재하던 losers list에서 삭제하였습니다. 그 후 발견한 commit혹은 rollback 레코드를 winner list에 추가하였습니다. 결과적으로 losers list에는 begin이 호출되었지만 commit 및 rollback이 호출되지 않아 loser로 판명이 난 트랜잭션들이 발급한 begin 로그 레코드가 존재할 것이고, winner list에는 commit혹은 rollback이 진행된, winner로 판명이 난 트랜잭션들의 commit혹은 rollback레코드가존재할 것입니다. 따라서 해당 레코드들의 id를 통해 winner와 loser를 가려낼 수 있습니다.

Recovery-Redo

그 후 redo pass가 진행됩니다. redo pass시엔 다시 로그파일에 접근하여 처음부터 record를 읽습니다. 그리고 해당 레코드의 type에 따라 적절한 행동을 취합니다. 만약 record type이 begin, commit, rollback이면 해당 레코드를 발견했다는 내용만을 출력하고, update 및 compensate일 경우 old image를 new image로 교체해줍니다. update 혹은 compensate을 진행하기 위해 변경이 일어난 페이지에 접근할 때는, 해당 페이지에 update가 진행될 때 기존에 기록한 LSN값이 페이지에 존재하게 될 것입니다. 만약 해당 LSN값이 redo를 하려는 레코드의 LSN값보다 크다면, 더 최신정보가 디스크에 존재하는 것임으로, consider redo를 진행합니다. 또한 모든 확인한 record를 undo과정에서 활용하기 위해 저장합니다.

Recovery-Undo

undo pass에서는 redo pass시 저장해 놓았던 record list를 활용합니다(all이라는 변수가 가리킵니다). list의 가장 뒤에서부터 레코드에 접근하는데, 해당 레코드가 loser transaction이 발급했는지를 확인합니다. loser transaction이 발급한 update 레코드가 존재한다면 해당 레코드의 에 저장된 new image를 DB상에서 old image로 변경해줍니다. 그 후 해당 레코드에 대한 undo가 끝났다는 의미에서

compensate log를 생성해줍니다. 이때 compensate log는 next Undo LSN을 저장하는데, 해당 트랜잭션이 한단계 전에 update했던 레코드의 LSN을 가리키게 됩니다. 이는 undo pass시 compensate log를 발견했을 때 활용하는데, 이는 과거의 recovery에서 undo pass시 undo를 이미 진행했다는 의미이므로, 해당 레코드가 가리키는 LSN을 만나기 전까지 해당 트랜잭션이 발급한 update record에 대해 더 이상 undo를 진행할 필요가 없다는 의미입니다.

In-depth Analysis

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

Problem

만약 non-conflicting read-only transaction들이 무수히 많이 DB에 접근할 경우, 현재 lock table management 디자인에서는 모든 트랜잭션들이 lock table management layer에서 record lock을 획득하기를 시도할 것입니다. non-conflicting이란 가정에서는 트랜잭션들이 곧바로 lock을 acquire하는데에 성공하겠지만, 모든 트랜잭션들의 불필요하게 lock을 획득하려고 시도하고, 기존에 연결되어 있는 lock이 있는지 확인하고, 뒤에 들어올 수 있는 트랜잭션을 막아버리는 여러 작업들로 인해 [1]불필요한 workload가 증가합니다. 또한 동시에 여러 lock들이 생성되고, lock을 관리하기 위해 많은 메모리가 소모될 것이므로 [2]큰 lock management overhead가 발생할 것입니다. 또한 deadlock이 존재하지 않을 것이므로, [3]deadlock을 detect하는 과정에서 불필요하게 많은 시간을 소요하게 될 것입니다. 또한 현재 제 구현상에는 deadlock detection을 위한 wait-for graph를 adjacency matrix를 이용해 구현하였는데, 이는 [4]index개수 이상의 트랜잭션이 한꺼번에 몰려오면 matrix가 의미 없는 graph가 됩니다. 이러한 한계와, detection과정에서 overhead를 줄이기 위해서는, 현재 구현된 PCC 기반의 strict 2PL locking 방식에서 MVCC기반의 SI(Snapshot Isolation)로 concurrency control을 구현할 수 있을 것입니다. 해당 방식은 각 트랜잭션이 begin한 시점에 존재하던 record를 가져오게 됩니다. 별도의 lock을 획득하는 과정이 존재하지 않고, 그 시점에 맞는 record를 즉각적으로 찾아서 읽어드릴 수 있으므로 매우 빠른 속도로 operation을 수행할 수 있습니다.

Solution1

해당 방식의 구현을 위해, 본 코드에서 lock manager에 해당하는 locktable.c의 내용의 변경이 일어나야 할 것입니다. 만약 value의 update가 일어난다면(그럴 일이 희박하다는 가정이 존재하지만), 원본의 value에 곧바로 접근해서 value를 변

경시키는 것이 아닌, value의 복사본을 만들어 그 복사본을 변경하게 됩니다. 그리고 변경을 완료한 value에 대해선, 어느 시점에 해당 value를 변경했는지에 대한 정보를 기록하기 위해 변경을 한 트랜잭션의 id(trx_id)를 기록합니다. 그리고 연결된 value들을 관리하는 역할을 기존의 lock_table.c에서 수행하게 될 것입니다(파일명이 해당과정과 차이가 있지만, concurrency control을 수행한다는 측면에서 공통점이 존재합니다). 해당 manager는 lock을 관리하는 일을 수행하지 않고, value들을 연결하는 용도로 사용할 것입니다.

만약 어떤 트랜잭션이 특정 레코드에 접근하려 하면, 해당 트랜잭션의 trx_id를 확인하여, 읽어드려야 할 value를 결정합니다. 현재 read only 트랜잭션들이 접근한다는 가정이 있지만, 만약 기존에 여러 트랜잭션들이 해당 value를 변경한 상태라면, read를 하려는 트랜잭션이 begin한 시점보다 과거에 변경된 value중, 가장 최신의 value를 value list에서 찾습니다. value list는 linked list형식으로, 처음 레코드에 접근할 때에는 가장 최신의 value를 가리키고, 각각 그 다음 value를 pointing하는 pointer를 갖고 있습니다. 이러한 방식으로 read하고자하는 트랜잭션이 원하는 value를 찾습니다.

해당 구현은 lock의 관리가 불필요하게 됩니다. 각각의 시점에 맞는 value를 미리 저장해 두기 때문에, 여러 lock들의 연결을 시도하고, 연결을 해제하는 이러한 [1]불필요한 workload가 감소합니다. 또한 매 트랜잭션마다 lock을 생성하는 과정이 사라졌기 때문에, [2]상대적으로 발생하는 overhead가 감소할 것입니다. Snapshot과정으로 동시에 여러 트랜잭션이 하나의 레코드에 접근하여도, 트랜잭션은 begin 시점에 존재하던 record만에 접근할 수 있으므로, [3]deadlock이 발생하지 않아 deadlock detection에 소요되던 시간을 감소시킬 수 있을 것입니다. [4]deadlock detection에 사용되던 한정된 크기의 adjacency matrix의 문제도 해결할 수 있을 것입니다.

Value관리에 관한 세부적인 구현은 다음과 같습니다. 오래된 value는 만약 새로운 value와 오래된 value 사이에 begin을 한 트랜잭션이 존재하지 않는다면, 오래된 value는 더 이상 필요가 없게 되므로 link를 끊어주고 free를 해줄 수 있습니다. 하지만, 위 작업을 진행하기 위해선 어떠한 트랜잭션이 현재 진행중인지 확인할 수 있어야 합니다. 제가 생각한 방법은 transaction manager 역할을 담당하는

trx.c파일에 존재하는 transaction table에 현재 수행중인 trx에 대한 정보를 저장해 두는 방법이 있을 것입니다.

Solution2

처음 주어진 문제에 대한 두번째 해결책은 OCC 방식 중 FOCC 방식을 활용하는 것입니다. FOCC방식은 어떠한 트랜잭션이 validation과정에서 해당 트랜잭션이 변경하려는 레코드에 다른 트랜잭션이 read를 한 상황이라면, 특정 행동으로 해당 상황이 일어나지 않게 변경시켜줍니다. 만약 아주 희귀하게 write이 존재하는 경우, 위와 같은 상황이 일어난다면, write이 존재하는 트랜잭션을 우선적으로 abort 시키는 정책을 활용하게 되면 write operation이 희박하므로 큰 성능 저하없이 concurrency control을 수행할 수 있다고 생각합니다. 해당 과정 역시 lock을 관리하는 과정이 불필요하기 때문에, 기존에 발생했던 문제들을 해결할 수 있다고 생각합니다.

Workload with many concurrent non-confliction write-only transactions.

Problem1

현재 제 디자인에서는 analysis pass에서 로그 페이지를 한번 확인하며 winner와 loser을 정하고, redo pass에서 또다시 log file에 접근하는 방식을 택하고 있습니다. 하지만 이러한 구현은 동일한 내용을 가진 파일에 두번씩 접근하게 되므로, 메모리와 디스크상의 느린 속도차이로 매우 느린 read를 두번씩 수행하게 됩니다. 이러한 방식은 결국 성능의 큰 저하로 이어질 것입니다.

Solution

Analysis pass에서 로그 파일의 스캔을 진행할 시 미리 메모리에 해당 내용들을

저장하는 방법을 사용할 수 있을 것입니다. 해당 내용을 메모리에 한번 저장과 동시에 winner와 loser을 구분할 수 있을 것입니다. 이렇게 analysis pass를 진행하게 되면, Redo pass과정에서 불필요하게 또다시 로그파일을 읽지 않고, 기존에 메모리상에 저장해 두었던 로그 레코드에 대한 정보를 활용하여 redo를 진행할 수 있을 것입니다. Undo과정 역시 해당 메모리에 연속적으로 복사된 log record의 집합을 이용할 수 있습니다. 해당과정은 레코드 리스트의 뒤에서부터 접근하여 undo과정을 진행할 수 있을 것입니다. 이러한 진행과정은 디스크의 불필요한 중복 접근을 줄여 속도를 증가시킬 수 있을 것입니다.

Problem2

Recovery 전체 과정이 init_db의 호출 후 해당 함수내에서 진행됩니다. 현재 제코드에서는 init_db호출 시 오직 하나의 쓰레드로 recovery과정이 진행됩니다. 이렇게 하나의 쓰레드로 레코드를 복구하게 되면, 레코드의 양이 많아지게 되면 계속적으로 파일에 접근하여 읽는 작업이 오래 걸릴 것입니다. 그 이유는 위의 문제와 같이 파일의 접근 속도는 메모리로의 접근속도보다 월등히 느리기 때문입니다.

Solution

하지만 이를 멀티 쓰레드 환경에서 recovery를 시도해 볼 수 있습니다. 처음 analysis과정은 위의 solution과 같이 로그 파일을 읽어서 메모리에 저장합니다. 그리고 redo를 진행할 쓰레드를 생성하는데, 해당 과정에서 쓰레드는 페이지별로 redo를 진행합니다. 즉 하나의 쓰레드는 같은 테이블의 같은 페이지 일어난 update의 redo만을 진행합니다. 따라서 모든 쓰레드는 각각 다른 위치에 존재하는 내용에만 접근할 것이므로, 쓰레드 간의 업무가 병렬적으로 일어날 수 있게 될 것입니다. 위와 같이 병렬적으로 redo를 진행하게 되면, 기존에 레코드의 LSN 순서대로 진행할 경우보다 더욱 빠르게 recovery를 완료할 수 있을 것입니다.

추가로, 이번 과제의 범위가 아니었던 checkpoint를 추가하게 되면, 로그파일을 읽는 양을 현저하게 줄일 수 있을 것입니다. Checkpoint를 활용하게 되면 특정 시

점 이후의 파일만 읽을 수 있으므로, 성능면에서 큰 이점을 얻을 수 있습니다. 만약 이미 undo가 완료된 시점에서 체크포인트를 로그 파일에 저장하면, 만약 다시 crash가 발생했을 경우, 파일의 접근시간이 줄어들 것입니다.