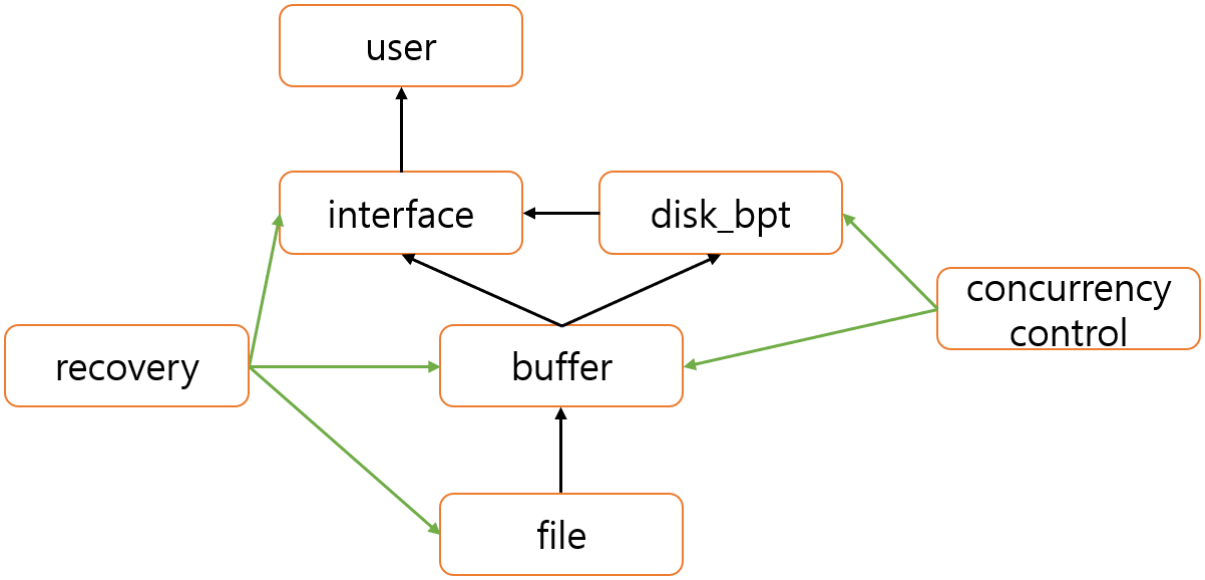
Database System 2020-2
Final Report

Class Code (ITE2038-11800)
2015005187
최철훈

Table of Contents

Overall Layered Architecture



이번학기에 제작한 DB는 위와 같은 Layer로 구성되어 있다. 각각의 Layer는 서로 인접한 Layer끼리만 소통한다.(Transaction Layer 제외)

File/Disk Space Layer : 위 그림에서 file단계가 해당하며 디스크에서 직접 페이지를 읽고 쓰는 작업을 담당한다. 페이지뿐만 아니라 로그 등의 데이터들을 디스크에 직접 읽고 쓴다. 코드에서는 file.c에 구현되어 있다.

Buffer Layer : 위 그림에서 buffer단계가 해당하며 DB가 시작할 때 생성한 버퍼를 이용하여 물리적인 디스크와 메모리의 속도차이를 극복한다. Index Layer에서 페이지를 요청하였을 때, 버퍼에 해당 페이지가 존재한다면 바로 넘겨주고 존재하지 않는다면 Disk Layer에 해당 페이지를 요청하여 버퍼로 읽어온 다음에 넘겨준다. 해당 페이지가 수정되면 Index_Layer에서 is_dirty비트로 체크해둔다. 버퍼가 가득차서 새로운 페이지를 읽어들일 수 없을 때 LRU알고리즘에 의해 Page Eviction이 발생하며, 이 때 is_dirty가 체크되어있다면 디스크에 그대로 반영한다. 데이터 파일을 닫거나 DB를 종료할 때에도 버퍼상에 is_dirty가 체크되어있는 페이지가 있다면 체크되어있는 페이지를 모두 디스크에 반영하고 종료한다. 이렇게 디스크로의 직접적인 입출력의 횟수를 줄임으로써 입출력에서 발생하는 I/O시간을 줄여 DB의 전체적인 성능을 향상시키고

디스크와 메모리의 속도차이를 극복한다. 코드에서는 buffer.c에 구현되어 있다.

Index Layer : 위 그림에서 interface와 disk_bpt단계가 해당한다. 본 DB는 데이터를 저장하고 읽어올 때 B+tree형태의 자료구조를 채택하고 있다. 사용자가 찾기(db_find), 갱신(db_update), 삽입(db_insert), 삭제(db_delete) 등의 오퍼레이션을 사용하면 B+tree의 동작에 따라 Buffer Layer에 페이지를 요청하면서 데이터를 빠르게 찾아와 사용자에게 넘겨준다. disk_bpt단계에서 해당 B+tree의 내부적인 동작을 수행하며 사용자는 오로지 interface단계에서 제공하는 인터페이스만 사용하여 여러 오퍼레이션을 실행할 수 있다. 코드에서는 interface.c, disk_bpt.c에 구현되어 있다.

Transaction Layer : 위 그림에서 concurrency control 과 recovery 단계가 해당한다. DB 는 Atomicity, Consistency, Isolation, Durability(ACID)성질을 가져야 하는데 이 Layer 가 ACID 성질을 만족시켜준다. DB 의 이러한 전체적인 성질을 만족시켜주는 Layer 이므로 Cross Layer 로 이루어져 있다. Concurrency Control 과 Crash-Recovery 로 이루어져 있으며 자세한 내용은 뒤에 이어진다.

Concurrency Control Implementation

Concurrency Control 기법은 Index Layer, Buffer Layer에 걸쳐서 구현되어 있다. 이 기법은 여러 사용자가 동시에 DB에 접근하고자 할 때, 사용자가 의도한 올바른 최신 상태의 데이터를 전달해주면서 자신이 수행하는 오퍼레이션만 동작하는것과 같이 보이도록 하는 데에 목적이 있으며, 트랜잭션과 lock_table, 여러가지 latch들로 구성되어 작동한다.

Transaction : 사용자가 DB의 상태를 변화시키는 단위로써 begin(trx_begin)과 commit(trx_commit)으로 이루어져 있다. 사용자가 begin하면 트랜잭션이 시작하고, 여러 오퍼레이션을 수행하다가 commit하면 종료하며 commit된 트랜잭션이 수행한 오퍼레이션의 결과는 무조건 DB에 반영되어야 한다. begin이후 commit하지 않고 begin이전의 상태로 돌리고 싶다면 abort(trx_abort)를 사용한다. 트랜잭션은 트랜잭션 테이블(trx_table)에 의해서

관리되며 해당 트랜잭션이 실행한 오퍼레이션이 생성한 lock들과 연결되어있다. 트랜잭션은 Index Layer의 interface단계에 구현되어 있다. 그래서 사용자는 trx_begin과 trx_commit, trx_abort로 트랜잭션을 제어할 수 있다.

Lock_table : 실질적으로 Concurrency Control 기법은 Lock_table에 의해서 올바르게 작동한다. lock_table.c에 구현되어 있으며, interface단계에서 db_find, db_update 오퍼레이션을 수행할 때 Lock_table을 사용한다. Lock_table은 해쉬테이블과 링크드 리스트 구조로 설계되어있으며, 해쉬테이블에는 불러온 레코드가 저장된다. 오퍼레이션을 수행하면 해당 오퍼레이션에 맞는 lock을 생성하여 접근하려는 레코드에 순서대로 붙인다. 자신의 차례가 되어서 lock을 잡게되면 해당 lock에 상응하는 오퍼레이션을 수행한다. lock을 잡음으로써 해당 레코드에는 lock을 잡은 오퍼레이션 외에 다른 오퍼레이션은 수행될 수 없게 되며 Concurrency Control이 올바르게 작동할 수 있게 된다. lock을 잡게 되는 경우는 여러 경우가 있다. 기다린다면 해당 lock의 lock_wait = 1로 설정한다.

1. 가장 첫번째 lock이라면 lock을 잡는다.
2. 붙이려는 lock앞에 같은 트랜잭션의 lock들만 존재한다면 무조건 lock을 잡는다.
3. 붙이려는 lock이 lock_mode = 0(read-lock)이고 내 앞에 lock_mode = 0인 lock밖에 없다면 lock을 잡고 lock_mode = 1(write lock)이 1개라도 있다면 사라질 때까지 기다린다.
4. 붙이려는 lock이 lock_mode = 1이라면 앞에 어떤 lock이 있던 가장 첫번째 lock이 될 때까지 기다린다.

lock은 해당 트랜잭션이 commit되거나 abort되면 release되면서 사라진다. release될 시에는 뒤에 기다리고 있는 lock을 이제 차례가 되었다고 알려준다. 이를 깨웠다고 한다. release되는 경우는 여러 경우가 있다. release되면 해당 lock의 lock_wait = 0으로 설정한다.

1. 뒤에 기다리고 있는 lock이 없다면 그냥 release된다.
2. 뒤에 기다리고 있는 lock이 lock_mode = 0이라면 lock_mode = 1인 lock이 나올때까지 모든 lock_mode = 0인 lock들을 깨우고 release된다.
3. 뒤에 기다리고 있는 lock이 lock_mode = 1이라면 바로 뒤 이 1개의 lock만을 깨우고 release된다.

4. Commit의 경우, 중간에 있는 lock이 release된다면 lock_mode = 0인 lock이 해제되는 경우밖에 없어 뒤에 있는 release될 때 앞에 다른 lock이 있으므로 뒤에 있는 lock을 깨우지 않아도 된다. 하지만 abort의 경우에는 중간에 있는 lock_mode = 1인 lock이 release되는 경우가 존재한다. 이 경우에는 앞에 있는 lock이 모두 lock_mode = 0인 경우와 lock_mode = 1인 lock이 1개라도 있는 경우로 나뉜다. 전자의 경우, 뒤에 있는 lock이 lock_mode = 0이면 lock_mode = 1인 lock이 나올 때까지 뒤에 있는 모든 lock을 깨운다. 뒤에 있는 lock이 lock_mode = 1이라면 깨우지 않는다. 후자의 경우, 아무런 lock도 깨우지 않는다.

Concurrency Control에서 모든 문제는 Deadlock상황에서 발생한다. 본 DB는 Pessimistic Concurrency Control을 채택하고 있기 때문에 Deadlock상황을 잡아낼 수 있어야 한다. Deadlock은 아래와 같이 잡아낸다.

1. 붙이려는 lock이 기다려야 되는 상황이라면 앞에 있는 lock들을 하나하나 보면서 해당 lock을 수행하는 트랜잭션의 다른 lock들이 자신을 수행한 트랜잭션의 다른 lock을 기다리고 있는지를 확인한다. 트랜잭션이 commit 혹은 abort가 되어야만 해당 트랜잭션의 lock들이 전부 release되므로 그전에는 계속 기다리고 있어야만 하기 때문이다. 이렇게 되면 두 트랜잭션이 서로를 기다리는 상황이 발생하므로 Deadlock이 된다.
2. 위의 경우를 좀 더 확장하여 일반적인 경우를 생각해본다. 붙이려는 lock앞에 있는 lock을 수행하는 트랜잭션의 다른 lock을 살펴보다가 lock_wait = 1인 lock이 있다면 이 lock이 기다리고 있는 앞의 lock을 하나하나 살펴본다. 살펴보는 lock을 수행하는 트랜잭션의 다른 lock을 또 살펴보다가 lock_wait = 1인 lock이 있다면 이 lock이 기다리고 있는 앞의 lock을 하나하나 살펴본다. 이를 반복하다가 최종적으로 붙이려는 lock을 수행하는 트랜잭션을 기다리고 있는 lock이 나타난다면 이를 wait for graph에서의 순회에 해당한다고 볼 수 있으므로 Deadlock상황이라고 판단하고 abort를 진행하게 된다.

Deadlock상황에서 abort를 진행하는데 또다른 문제가 발생하게 된다. Commit의 경우 commit될 트랜잭션의 모든 lock이 다 실행되고 trx_commit을 호출하기 전에 이 트랜잭션의 아직 release되지 않은 lock을 다른 트랜잭션이 Deadlock을 발견하는 과정에서 접근해도 이미 수행된 오퍼레이션이거나 사라질 lock이기

때문에 아무런 문제가 발생하지 않는다. 하지만 Deadlock상황에서 abort가 진행되는 경우, Deadlock을 감지하고 abort를 진행하기 전이나 진행중에 다른 트랜잭션이 Deadlock을 발견하는 과정에서 이 Deadlock이 감지되어 abort되기를 기다리고 있는 혹은 abort과정에 있는 lock에 접근할 경우 앞선 Deadlock을 발견하는 과정에서 순회에 걸려 무한루프에 빠지고 만다. 이런 상황을 방지하기 위하여 Deadlock발견 후 abort를 진행하게 되면 Deadlock이 발견된 순간부터 abort가 완료되기까지를 lock_table_latch로 잠궈서 이 때는 오로지 abort만을 진행하도록 하였다. abort가 끝나야지만 비로소 다른 lock이 수행될 수 있는 것이다.

Latch : Latch를 이용함으로써 해당 공간을 하나의 트랜잭션만이 사용할 수 있도록 제어한다. 한 공간에 하나의 트랜잭션만 접근해서 수정해야 올바른 결과를 얻어낼 수 있기 때문이다. Buffer Layer에서는 주로 buffer에 대한 보호와 페이지에 대한 보호를 한다. buffer_latch는 버퍼에 대한 보호, 버퍼 내 프레임의 page_latch는 페이지의 eviction에 대한 보호, log_buffer_latch는 log_buffer에 대한 보호, trx_table_latch는 trx_table에 대한 보호, commit_table_latch는 commit_table에 대한 보호를 한다.

Crash-Recovery Implementation

Crash-Recovery 기법은 Index Layer, Buffer Layer, File/Disk Space Layer에 걸쳐서 구현되어 있다. 이 기법은 DB가 구동되고 있는 중에 어떠한 이유로 인해 Crash가 발생하면 이 때까지 commit되었거나 abort된 트랜잭션의 결과는 모두 정상적으로 DB에 반영이 되어야 함을 보장하게 만드는 기법이다. 이를 위해 본 DB에서는 Crash이후의 recovery를 위해 log를 남긴다. Crash가 난 이후 DB를 시작하면 기록된 로그에 대한 recovery가 실행되며 logmsg파일에 recovery로그를 남기게 된다. Recovery가 종료된 이후에 DB는 사용자가 사용할 수 있는 상태가 된다.

Log : Index Layer에서 trx_begin, trx_commit, trx_abort, db_update시에 log를 log_buffer에 기록하며 WAL프로토콜에 의해서 commit되기 직전, 버퍼에서 Page Eviction이 발생하기 직전에 실제 디스크에 기록된다. Buffer Layer에서는 Page Evction이 발생할 때, log_buffer의 내용을 디스크로 쓰는 역할을 한다. File/Disk

Space Layer에서는 log와 commit_table에 대하여 직접적인 디스크 입출력을 수행한다. log_buffer를 디스크에 기록할 때, log_buffer를 모조리 비운다. LSN은 Crash가 발생하고 다시 DB가 시작되어도 0으로 초기화되지 않고 계속 증가한다.

Recovery : recovery는 ARIES알고리즘에 의해서 수행된다. DB를 처음 시작하게 되면 Index Layer에서 init_db에 의해 recovery가 진행되어 Crash가 발생하기 이전에 Commit된 트랜잭션의 결과를 사용자가 받아볼 수 있도록 한다. ARIES알고리즘은 Analysis Phase, Redo Phase, Undo Phase 총 3단계에 걸쳐서 이루어진다.

- Analysis Phase
  redo를 실행할 winner transaction과 undo를 실행할 loser transaction을 구분하는 단계이다. 이를 구분하기 위해 commit_table파일을 만들었으며, commit이 발생할 때마다 commit된 트랜잭션과 여지껏 실행된 총 트랜잭션의 개수가 기록된다. 이렇게 기록된 commit_table파일애서 crash가 발생하기 전에 실행된 트랜잭션의 개수를 불러와 이 개수 까지의 트랜잭션 중 파일에 기록되어 있으면 commit된 트랜잭션으로 판단하여 winner transaction으로 구분한다. 기록되어 있지 않은 트랜잭션은 commit이 완료되지 않은 트랜잭션으로 판단하여 loser transaction으로 구분한다. abort가 완료되어도 winner transaction으로 구분할 것이기 때문에 abort시에도 해당 트랜잭션을 commit_table에 기록하며 abort를 마치면 rollback로그를 기록한다.
- Redo Phase
  Crash가 발생하기 직전의 상태로 돌리기 위해 로그를 하나씩 불러와서 모든 로그에 대해 해당 오퍼레이션을 redo한다. 로그를 하나씩 불러오는 이유는 현재 로그 버퍼의 크기가 99,999로 제한되어 있어 99,999초과의 로그가 존재한다면 한번에 불러올 수 없기 때문이다. begin, commit, rollback로그는 아무런 행위를 하지 않는다. update와 CLR로그에 대해서만 redo한다. 여기서 CLR로그는 abort시에 발생하는 로그로 undo했던 것을 redo하는 로그이다. redo할 때 해당 로그가 loser transaction이 실행한 로그라면 undo하기 위해 CLR을 발급해야하므로 해당 로그의 offset을 기록해둔다. 또한 redo시에 수정하고자 하는 페이지의 LSN이 해당 로그의

LSN보다 크다면 consider-redo상태가 되어 아무런 행위를 하지 않는다. redo를 모든 로그에 대해 진행하고나면, loser transaction이 실행했던 오퍼레이션을 undo하기 위해 CLR을 생성한다.

- Undo Phase
  Crash가 발생했을 때, commit되지 않았던 loser transaction의 오퍼레이션을 앞서 발급한 CLR을 이용하여 undo한다. undo시에도 수정하고자 하는 페이지의 LSN이 해당 로그의 LSN보다 크다면 consider—redo로 처리하여 아무런 행위를 하지 않는다.

현재 DB는 recovery도중에 Crash가 발생한 상황에 대하여 모든 log_buffer와 dirty_page를 디스크에 작성하게끔 구현되어있다. 이로 인해 위와 같이 recovery를 진행할 경우, recovery중에 crash가 발생하여도 해당 로그까지 진행했다고 page에 기록되어있고 CLR의 LSN은 증가하는 방향이기 때문에 recovery를 진행할수록 consider-redo로 처리되는 양이 많아져 recovery의 양을 줄일 수 있다. 하지만 실제 crash가 발생한다면 log_buffer와 dirty_page를 디스크로 작성하지 않을것이므로 이 때는 CLR에 있는 next_undo_LSN을 이용해야 recovery의 양을 줄일 수 있을 것이다.

In-depth Analysis

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

문제점 : 많은 수의 non-conflicting read-only transaction이 동시에 수행될 경우 굉장히 많은 page eviction이 발생할 수 있다. 서로 다른 레코드에 접근하는데 동일한 페이지의 레코드를 접근한다면 page eviction이 잘 발생하지 않겠지만 많은 수의 transaction이므로 다른 페이지를 얼마든지 자주 접근할 수 있기 때문이다. 게다가 현재의 DB는 general lock을 구현하지 않은 상태이기 때문에 더더욱 page eviction이 많이 발생하리라 예상된다. 해당 레코드가 존재하는지를 확인하고 다시 레코드에 접근하려는데 페이지가 evict되어있다면 그 페이지를 다시 불러와야하기 때문이다. page eviction이 많이 자주 발생하게 되면 disk 입출력이 많아져서 I/O에 소요되는 시간이 길어져 읽기만 함에도 불구하고 DB성능의 저하로 이어진다.

해결 : 문제점의 원인이 page eviction이 자주 발생하는 것이었으므로 page

eviction이 자주 발생하지 않도록 해야한다. 그러기 위해서는 첫번째, 사용자가 버퍼를 처음 init_db에서 할당할 때 더 많이 할당해준다. 이는 설계 디자인측면의 해결책은 아니지만 하나의 방법이 될 수 있다. 두번째, 현재 4KB로 설정되어있는 페이지의 크기를 늘린다. 페이지의 크기를 늘리게 되면 한 페이지에 더 많은 수의 레코드가 기록되어 있으므로 다른 레코드를 접근하여도 다른 페이지에 접근할 확률이 줄어들기 때문이다. 세번째, general lock으로 변경한다. general lock으로 변경하게 되면 레코드가 있는지 존재하는지 확인하고 page lock을 풀지 않은 상태에서 바로 lock을 획득하기 때문에 존재유무 확인과 lock 획득 사이에 발생할 수 있는 page eviction을 방지할 수 있기 때문이다.

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

문제점 : 많은 수의 non-conflicting read-only transaction이 동시에 수행될 경우 recovery과정에서 모든 로그에 대해 해당하는 페이지를 불러와야 한다. 그러면 이 또한 1번과 마찬가지로 recovery과정에서 page eviction이 많이 발생하게 되고 recovery시간이 굉장히 길어지게 되어 성능이 안좋아진다.

현재 DB는 recovery도중에 발생하는 Crash상황에서 모든 log_buffer와 dirty_page를 디스크에 기록하게끔 되어있다. 이로 인해 페이지의 LSN을 확인하며 recovery시간을 줄일 수 있었다. 하지만 실제로 recovery도중에 Crash가 발생한 상황에서는 log_buffer와 dirty_page를 디스크에 기록하지 못할 것이다. 이렇게 되면 recovery도중에 Crash발생 후 다시 init_db를 실행하게 되면 page LSN에 전 recovery과정이 제대로 반영되지 않았을 것이므로 CLR을 계속해서 발급하게 되며 consider-redo를 하지 않게 된다. 이렇게 recovery도중에 Crash가 반복해서 발생하게 되면 recovery시간이 점점 길어져서 DB성능의 저하로 이어진다.

해결 : 첫번째 문제점의 원인은 recovery과정에서 page eviction이 자주 발생한다는 것이다. 이를 해결하기 위해서는 page eviction이 자주 발생하지 못하게 해야한다. 그런데 page eviction은 각각의 로그들이 다른 페이지에 접근하기 때문에 발생하게 된다. 그렇다면 로그들이 서로 다른 페이지를 자주 접근하지 않게 만들면 된다. 이렇게 하기 위한 디자인으로는 recovery시에

redo를 LSN 순서대로 진행하지 않고 페이지별로 진행하는 것이다. recovery의 목적은 Crash가 발생했더라도 내가 레코드에 접근하려고 할 때 이전에 commit했던 가장 최신의 상태를 가져올 수 있게 만드는 것이다. 해당 레코드 즉, 페이지만 최신의 상태이면 되기 때문에 모든 로그를 순서대로 실행하지 않고 각 페이지별로 recovery를 진행해도 recovery의 목적을 위배하지 않게 된다. 한 페이지를 가져와서 해당 페이지에 해당하는 모든 로그를 redo와 undo까지 모두 처리한다면 recovery를 마친 이 페이지는 더 이상 접근하지 않아도 되므로 전체적인 page eviction이 현저하게 줄어들게 된다. 그러면 page eviction이 줄었으므로 자연스레 DB의 성능이 향상된다.
두번째 문제점의 원인은 recovery과정에서 Crash가 발생하면 log_buffer와 dirty_page의 내용을 기록하지 않는다는 것이다. 이를 해결하기 위해서는 recovery도중에 일정 주기마다 log_buffer와 dirty_page를 디스크에 기록한다. 그러면 recovery가 해당 페이지에 반영이 되어있어 consider-redo가 실행되고 CLR또한 제대로 발급되므로 recovery시간을 줄일 수 있다. 그러나 이렇게 주기적으로 디스크에 기록해도 주기가 너무 길면 위와 같은 문제가 또 발생할 수 있다. 현재 DB에서는 next_undo_LSN을 사용하지 않지만 이를 사용하게 되면 CLR을 계속 발급하는 문제점을 해결할 수 있게 된다. 또한 주기가 너무 짧게 되면 disk I/O가 많아지게 되어 recovery시간이 길어질 수 있다. 그러므로 주기는 적당하게 타협을 봐서 정해야 한다.