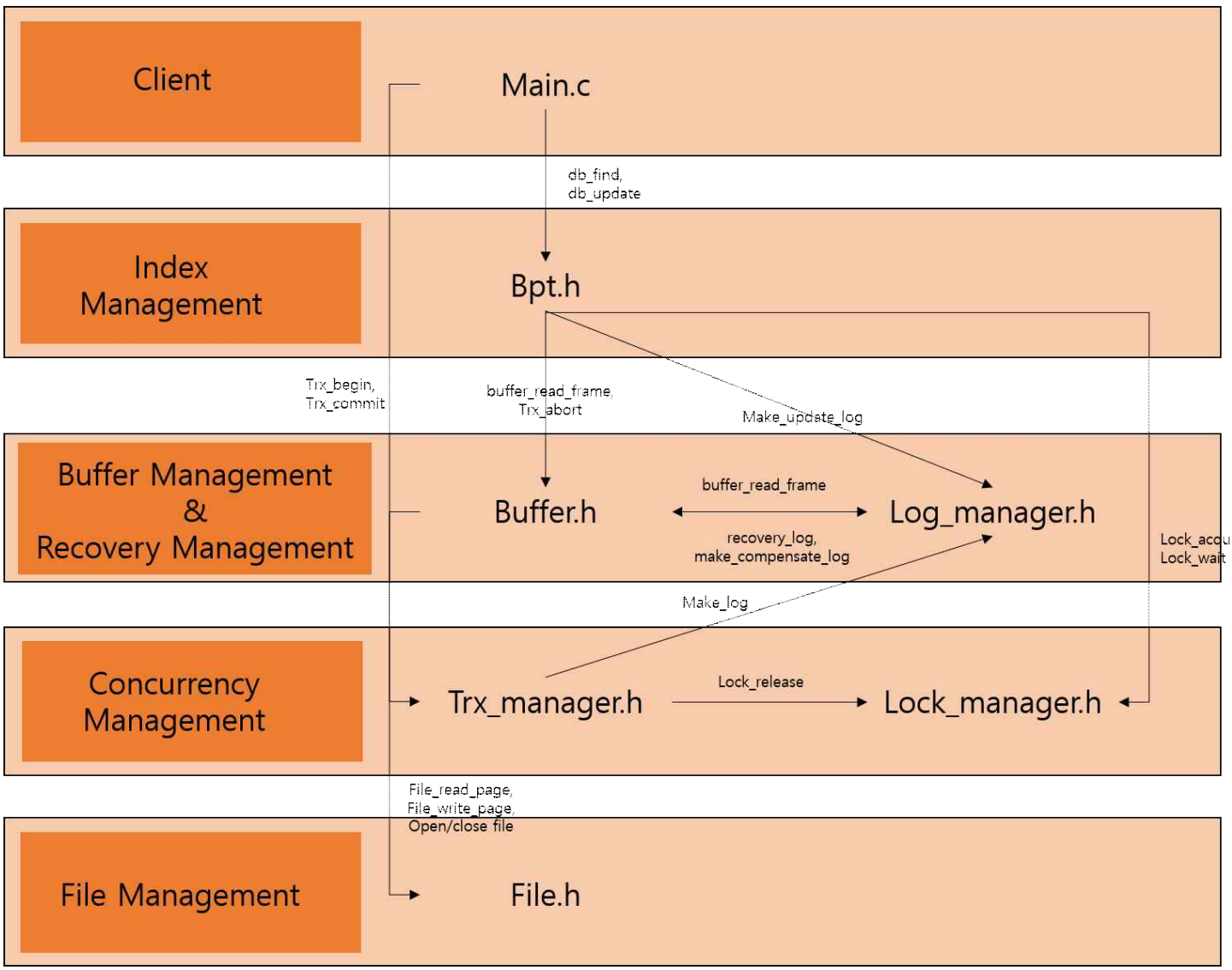
Database System 2020-2
Final Report

ITE2038-11801
2019008504
김동성

Table of Contents

1. Overall Layered Architecture

1.1 주요 함수들을 기반으로 그린 구조도



1.2 Layer 간의 상호 관계

1.2.1 Index Management와 나머지 Layer

가장 기본적인 기능을 하는 Layer인 만큼 file management를 제외 한 모든 계층과 직접적인 연관이 있다. Index Mangement에서 수행하는 db_find 및 db_update는 다음과 같은 단계로 수행이 되고 각각 연관된 layer는 다음과 같다..

1. 필요한 각 페이지를 Buffer에 불러오고 (Buffer Management)
2. 레코드에 대한 lock을 획득시도 하고 (Concurrency Management)
3. 상황에 알맞게 trx_abort 및 lock_wait를 실행한 후
4. find, update에 알맞은 동작을 하고
5. Log를 남긴다. (Log Management)

Index Management는 B+ tree 구조로 이루어져 있으며 다른 layer과 독립적으로 key, value 탐색을 수행하는 layer이다. In-memory에서 동작하면 다른 계층의 동작 여부와 상관없이 기능을 잘 수행할 수 있다. 하지만 on-disk로 동작 시, 속도 향상을 위한 Buffer Management, 동시 동작을 위한 Concurrency management, 복구를 위한 Log Management가 필요하다. 모든 Layer는 Index Management가 실행한 후에 동작하기 때문에 가장 상위 Layer라 할 수 있다.

1.2.2 Buffer Management와 Log Management

Buffer Management는 Data 변화에 File I/O와 Memory에서의 수행을 완충하는 역할을 한다. 그러므로 Index Management와 File Management와 가장 직접적으로 연관이 있다. Buffer Management는 독립적으로 Buffer만을 관리하며, Index Management로부터 필요한 페이지에 대한 호출이 이루어지면 File Management로부터 그 페이지를 불러와 caching을 하여 Index Management에 더 빠른 대응이 가능하도록 한다. 이때, File에 존재하는 내용과 In-memory 상에서의 차이가 발생하는데 crash 발생시 복구할 수 있도록 Log Management가 같이 필요하다.

Log Management는 File I/O를 하기 전 기록을 먼저 남기는 작업을 하는 계층이다. File Management를 제외한 모든 Layer에서 각 계층의 동작 후 Log를 남기는 작업이 실행이 되기 때문에 Layer 상의 위치에 대해서 고민이 많았다. 하지만 핵심 작업이 Index Management에서의 변화가 일어나고 그에 따른 File Management에서의 동작이 일어나기 전에 기록을 남기는 것이기 때문에 Buffer Management와 같은 Layer에 위치하게 했다.

즉 서로의 관계를 정리하면, Buffer Management에서 File I/O를 하기 전 Log Managemet를 통해 기록을 남기고 초기 실행시 Log management가 Buffer Mangement를 통해 복구가 된다.

1.2.3 Concurrency Management와 나머지 Layer

Concurrency Management의 주 업무는 요구되는 operation들에 대한 confilcting 여부를 확인하고 Deadlock이 생성되는지 확인하여 동시 동작을 가능하게 한다. 다른 Layer와 상호작용을 할 때는 다음과 같은 3가지 상황이 있다.

1. Index Management를 통해 record의 존재 여부를 확인 후
   해당 record에 대한 lock을 획득할 때
2. trx_commit 후 Log를 남길 때
3. trx_abort를 위해 update된 record를 다시 되돌릴 때

그래서 Index Management와 Buffer Management 하위에 Layer를 두었고 독립적으로 Concurrency Management를 할 수 있도록 하였다. 단, 원래는 trx_abort 수행을 수행을 위해 Concurrency Management에서 deadlock detection 후 Buffer Management를 불러와 update 초기화를 진행하려 했으나 명세에 따라 설계를 하다 보니 deadlock detection의 결과를 Index Management에서 알 수 있기에 Index Management에서 Buffer Management를 불러와 바로 trx_abort를 하는 것이 더 효율적이라 생각해 trx_abort는 Buffer Management에서 진행하도록 했다. 지금 와서 생각해 보니 Layer의 장점인 독립성을 더 극대화 하기 위해 bpt.cpp에서 trx_manager를 불러오고 trx_manager.cpp에서 buffer.cpp를 불러오는 것이 더 깔끔한 디자인이였을 거 같다.

추가적으로, Trx_manager.cpp와 Lock_manager.cpp는 Concurrency Management의 핵심 작업인 deadlock detection에서 각각의 trx가 획득한 lock에 대한 탐색이기에 같은 Layer에 위치하였다.

2. Concurrency Control Implementation

Concurrency Control에 있어서 핵심 component는 각 레코드에 대한 lock을 획득하는 것과 trx 간의 획득한 lock간의 deadlock의 존재 여부를 탐색하는 것이라고 생각한다. 두 기능은 모두 Concurrency Management에서 구현이 되었고 자세한 설명은 다음과 같다.

2.1 Lock Acquire

DBMS의 중요한 요소인 ACID 중 Concurrency control을 위해서는 Consistency와 Isolation이 지켜져야 하고 이 기능을 담당하는 함수가 lock_acquire이다. 즉, 각각의 (table_id, key)에 대하여 lock table을 생성해 confilcting 되는 trx에 대해서 순서를 지켜서 실행하도록 하는 것이다. 자세한 동작 단계는 다음과 같다.

0. 자기가 그 record의 첫 번째 lock : 바로 반환
1. 이미 lock을 획득했을 때
    1. 획득한 lock의 상태가 X일 때 : 무조건 가능하므로 바로 반환.
    2. 획득한 lock의 상태가 S일 때 :
        1. 추가되는 lock의 상태가 S일 때: 바로 반환 (+우선순위를 위해 head에 추가)
        2. 추가되는 lock의 상태가 X일 때:
            1. 획득한 lock과 추가되는 lock 사이에 다른 trx의 X모드 lock 존재 : abort
            2. 다른 trx의 X 모드 lock 존재하지 않음:
                1. deadlock이 탐지될 경우 : abort
                2. deadlock이 없으면 다른 trx의 S모드 lock이 종료될 때 까지 sleep
2. 획득한 lock이 없을 때
    1. deadlock를 찾은후
        1. 있으면: abort
        2. 없으면:
            1. 추가되는 lock이 X모드:
                앞의 lock이 모두 종료될 때까지 sleep
            2. 추가되는 lock이 S모드:
                앞의 lock이 모두 종료 or 앞의 lock이 S 모드이고 깨어있을 때까지 sleep

이와 같은 단계를 거치면 conflict 되는 trx가 동시에 실행되는 일이 없어 lost update, dirty read 등의 문제가 모두 해결되고 confilct 되지 않는 trx에 대해서는 동시 실행이 일어나도록 한다. 하지만 다른 문제점이 발생하게 되는데, 그 이유는 다른 trx를 기다리게 되는 상황이 일어나 만약 서로를 기다리게 되는 상황이 일어나면 이 DBMS는 더이상 동작하지 않게 되기 때문이다. 이를 보완하기 위해 Concurrency control에서는 추가적인 기능이 필요하고 그 기능은 다음에 설명할 deadlock detection이다.

2.2 Deadlock Detection

앞에서 설명하였듯이, 서로 다른 trx가 서로를 기다리는 상황이 일어날 시 계속 동작할 수 있도록 deadlock을 탐색하고 문제를 일으키는 trx를 abort해 deadlock을 제거하는 별도의 처리가 필요하다. 여기서 중요한 점은 단순히 추가되는 lock의 앞뒤만 보고서는 deadlock을 판별할 수 없고 관련된 모든 lock을 탐색해 봐야 한다는 것이다. 동시에 너무 많을 시간이 걸리지 않도록 적절한 처리를 해야한다. 많은 탐색을 하지 않되 모든 lock을 검사할 수 있는 wait-for-graph를 만드는 방법은 다음과 같다.

먼저 하나의 trx를 표현하는 구조체에 lock를 링크드 리스트 형태로 저장하고, set를 사용하여 기다리는 trx를 모두 저장하였다. 그 후 dfs 방식으로 진행하고 동작 단계는 다음과 같다.

0. trx(N)가 lock을 추가할 때마다 deadlock detection을 수행하고
1. 추가되는 lock 앞에 있는 모든 lock의 trx_id를 N에 모두 기록한다.
2. 그 후에 저장된 정보를 바탕으로 N이 기다리는 모든 trx를 순회하며
3. 그 trx가 기다리는 모든 trx를 또 순회하고
4. 이와 같은 과정을 반복하며 N으로 되돌아 오는지에 대한 확인을 한다.

이때 중복된 trx를 다시 방문할 수도 있기에 순회하기 전 방문 확인을 위한 set을 하나 생성하고 방문할 때마다 그 set에 포함되어 있으면 바로 반환, 없으면 추가하는 방법으로 시간을 더 단축하였다.

3. Crash-Recovery Implementation

Crash-Recovery는 DBMS의 Atomicity와 Durability를 담당하고 있다. 이를 설명하기 위해서는 크게 두 가지 단계로 나눌 수 있다. 먼저 DBMS가 실행되고 있는 도중에 변경 사항들을 기록하는 Make Log, 그 후에 Crash가 일어난 후 Log를 보며 정상 상태로 DB를 변하는 Recovery가 있다.

3.1 Make log

trx의 변화 또는 record의 변화가 있을 때마다 실행이 되는 기능이다. 각각 적절한 정보를 입력하고 memcpy를 통해 in-memory의 Log Buffer에 기록이 되고, flush가 일어날 경우 pwrite를 통해 로그 파일에 저장이 된다. 또한 pre LSN을 관리하기 위해 trx에 마지막 LSN을 저장하는 공간을 따로 만들었다. Log의 종류에는 크게 5가지 있고 그 내용은 다음과 같다.

1. Begin :
   trx가 시작되었을 때 만드는 log이다. 하나의 trx에 대한 첫 LSN을 나타내기 위해 log의 prev LSN에 unsigned long long의 max 값을 저장한다. 그리고 trx의 마지막 LSN을 저장하는 pLSN에 log가 기록된 LSN를 저장한다.

2. Commit :
   trx가 종료될 때 만드는 log이다. 적절한 정보들을 저장하고 commit 종료되기 전에 log_flush를 통해 모든 log를 파일로 저장하여 Durability가 유지되도록 한다.

3. Rollback :
   trx_abort가 일어났고, 모든 record의 rollback이 진행된 이후에 만드는 log이다. commit과 동일하게 log_flush후 rollback이 완료된다.

4. Update :
   db_update가 일어날 때마다 생성되는 log이다. 앞의 log보다 저장되는 정보들이 더 많다(record 관련 정보).

5. Compensate :
   trx_abort 후 Rollback이 진행될 때 record값을 원래대로 되돌릴 때 생성되는 log이다. 한 가지 특이한 점은 속도 향상을 위해 원래대로 되돌릴 때 log에서 기록을 search하지 않고 trx에 update된 value들을 따로 저장해서 되돌린다.

추가적으로 flush가 되는 시점은 page의 eviction이 일어날 때와 db_shutdown이 일어날 때, 그리고 Log Buffer가 가득 찼을 때 flush가 일어나고 이를 통해 Durability를 만족시킬 수 있도록 한다.

3.2 Recovery

Crash로 인해 Log와 DB의 data 차이가 나는 경우, db_init 실행 시 recovery_log에서 3개의 Phase로 Atomicity 및 Durability 만족을 위해 Recovery가 진행이 된다. 특별한 디자인 없이 수업 내용, 명세서 내용대로 굉장히 naive하게 디자인 하였다.

3.2.1 Analysis Phase

이번 과제는 Checkpoint에 대한 디자인이 포함되지 않기 때문에 Log file을 처음부터 linear하게 읽어나간다. 자세한 단계는 다음과 같다.

1. Log_file을 처음 부터 28 단위로 계속 read 하며
2. 읽어온 log_size가 28보다 더 크면 더 read 해 하나의 log를 복원한다.
3. log_file이 끝날 때까지 이를 반복하며
4. begin과 commit이나 rollback을 체크하여 winner와 loser를 분류한다.

4번에 대해 추가 설명을 하자면 winner와 loser를 저장하는 set을 각각 만들고 begin인 경우 loser에 추가하고 commit이나 rollback을 만나면 loser에서 제거하고 winner에 추가한다.
3번 부분이 상당히 속도 저하를 일으키는 부분이라 생각하고 있고 log file을 한번에 메모리에 올려서 작업하는 것이 더 빠를 거 같다. 하지만 좋은 방법이 떠오르지 않아 실제로 구현하지는 못하였다.

3.2.2 Redo Phase

다음과 같은 단계로 이루어져 있다.

1. 복원한 log를 모두 처음부터 다시 선회하며
2. 해당 페이지를 불러 페이지 LSN과 로그 LSN 을 비교하고
3. 페이지 LSN이 더 작으면 redo를 진행하고 그렇지 않으면 consider-redo를 진행한다.

3.2.3 Undo Phase

다음과 같은 단계로 이루어져 있다.

1.맨 뒤에서부터 순회하며
2. loser이고 update이거 compenstate인 LSN을 undo하고
3. compensate log를 추가하며 next undo seq no에 prev_LSN을 저장한다.
4. next undo seq no이 존재할 경우 그 LSN으로 바로 이동하여 undo를 진행한다.

4. In-depth Analysis

4.1 Workload with many concurrent non-confilcting read-only transactions

많은 수의 trx가 서로 다른 record를 접근할 시 예상되는 문제점은 2가지가 있다.

1. 불필요한 hash table 생성, 삭제 반복

현재 디자인에서는 table_id에 대한 hash table을 생성하고 key에 대한 hash table을 따로 생성하는 디자인이다. 즉 새로운 record를 접근할 때마다 hash table이 동적할당 되고 해당 key의 hash table에 lock이 존재하지 않으면 그 table을 삭제한다. 많은 수의 trx가 서로 다른 레코드에 접근하면 당연히 hash table의 생성 삭제가 많을 수 밖에 없고 성능이 떨어지게 된다. 이를 해결하기 위해서는 table_id와 key를 합쳐서 만든 문자열 string을 key 값으로 가지는 하나의 hash table만을 사용함으로써 해결할 수 있다.

2. 같은 테이블 다른 레코드에 많은 접근 시 버퍼에서의 많은 시간 지연 발생

현재 디자인은 record lock에 대해서만 read/write 구분을 하기 때문에 Buffer에서는 두 trx가 동시에 page를 read하는 것이 불가능하다. 즉, 다른 record라도 같은 table에 대해서 많은 read가 발생하면 각각의 trx가 해당 table의 상위 page(root 주변 page)에 대한 read를 끝낼 때까지 다른 trx가 접근을 못하여 성능에 문제가 생긴다. 이를 해결하기 위해서는 page lock에도 read/write를 구분하여 read인 경우 page에 대해서도 여러개의 trx가 동시 접근이 가능하게 해야 한다.

4.2 Workload with many concurrent non-confilcting write-only transactions.

Crash-Recovery와 관련된 성능 측면에서의 문제점은 다음과 같다.

1. Log에는 하나의 trx만이 접근할 수 있다.

현재 디자인은 하나의 trx가 log buffer에 기록할 때 다른 trx는 대기를 해야되는 디자인이다. 동시 접근이 불가능하다면 많은 수의 trx가 시도될 때 성능에서 상당히 문제가 많을 것이다. 따라서 LSN을 생성할 때만 mutex로 관리하고 그 LSN에 기록이 되었는 지를 체크하는 구조체를 따로 만들고, LSN을 기반으로 Buffer의 그 offset에 동시에 기록을 하게 하고 flush하기 전에 모든 기록이 되어있는지 체크 한 후 flush 되게 하면 동시 접근에 대한 해결이 가능할 것 같다.

2. Checkpoint의 부재로 Recovery의 시간이 많이 소요된다.

write만 시행한다면 모든 시행이 기록에 남을 것이다. 하지만 현재 디자인에는 checkpoint가 없기 때문에 Recovery시에 처음부터 모든 기록을 consider-redo 또는 redo를 하기 위해 page I/O가 많이 일어나게 된다. 따라서 Log file 앞부분 8byte에 Flush LSN을 기록해서 바로 그 부분부터 Recovery를 진행하도록 하면 많은 성능 향상이 기대된다.