Database System 2020-2

Final Report

B+tree based DBMS with S2PL & ARIES

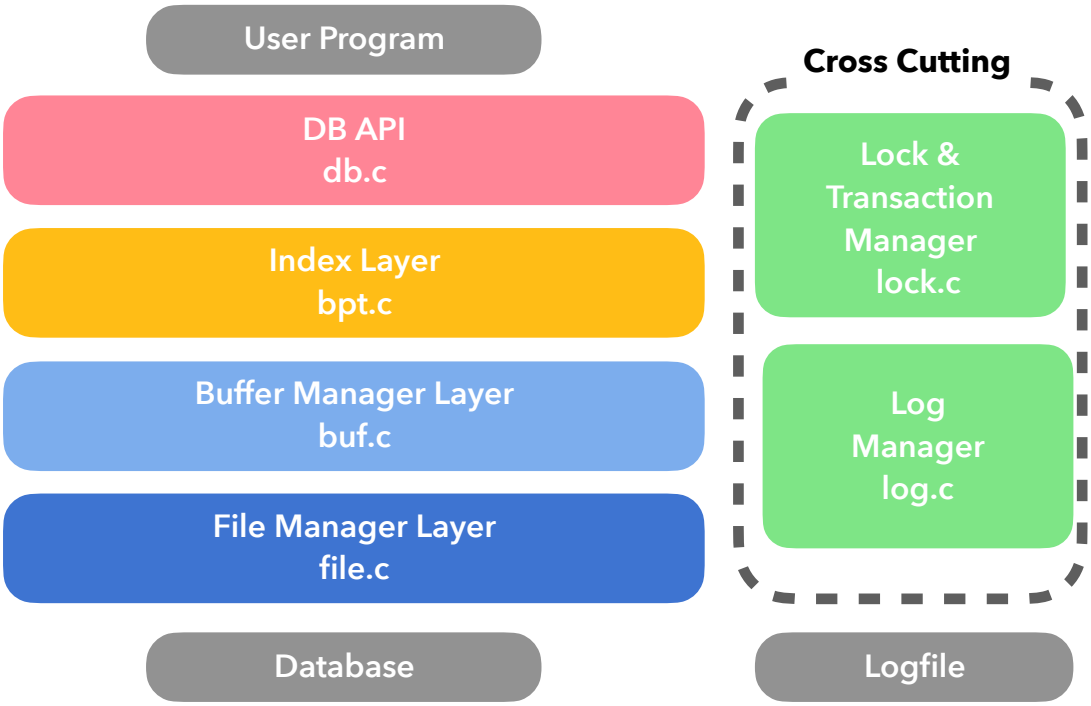
ITE2038-11801

2016024911

이상윤

Table of Contents

Overall Layered Architecture



전체적인 아키텍처는 총 3개의 레이어와 이전체를 아우르는두개의 Lock, Log 매니저로 구성되어있습니다. 레이어의 위에서 부터 설명하면 아래와 같습니다.

DB API에는 사용자가 실제로 자신의 프로그램에서 호출하게되는 Database의 여러 api가 들어있습니다. Insert, delete, find, open_table등 기본적인 database의 작동과 database의 시작과 초기화, 종료등을 다루는 함수를 호출할수 있습니다.

Index Layer에서는 사용자가 호출한 동작에대한 database내부의 자료가 find, delete,insert가 실제로 이루어지는 공간입니다 B+ tree의 구조를 활용하여 탐색, 삭제, 삽입이 이루어지며 모든 단위는 페이지 단위로 이루어지게 됩니다. 이 레이어에서는 실제 Database 파일에 있는 자료에 접근하기위해 Buffer manager Layer의 함수들을 이용해 페이지를 가져오고 저장합니다.

Buffer Manager Layer는 없더라도 database의 correctness에 차이가 없는 유일한 Layer입니다. Index와 File Manager Layer사이에서 페이지의 cashing을 통해 Database의 성능을 향상시키기 위해 존재합니다. Index Layer에서 요구된 동작들이 일어나며 필요에 따라 File manager Layer의 함수들을 이용해 Database file에 있는 페이지들을 불러와 BufferPool에 cashing합니다.

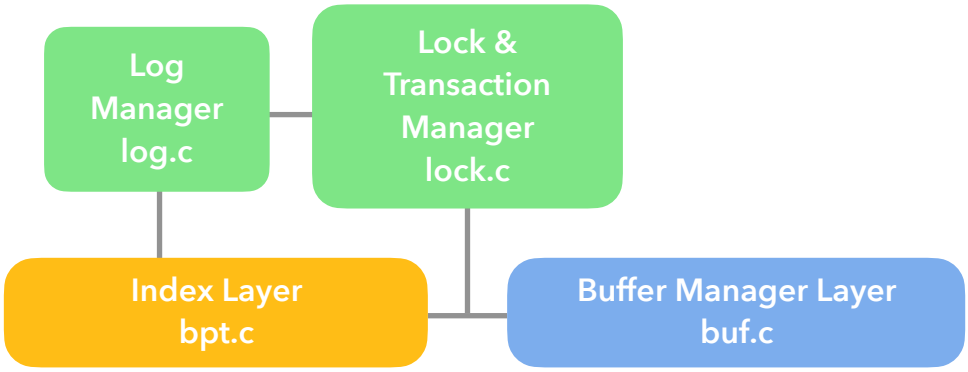
Layer중 마지막인 file manager Layer는 실제로 Database file과의 입출력을 관리하게 되는 Layer입니다. Buffer manager Layer에서 요청한 동작들이 들어올때마다 open과 write시스템콜을 이용해 실제로 페이지를 database file에 쓰고 읽는 동작을 수행합니다. 또한 상위 layer에서 호출된 테이블을 여는 작업이 실제로 이루어지는 곳이기도 합니다.

여러 Layer를 아우르며 작동하는 Cross Cutting으로는 lock manager와 log manager가 존재합니다.

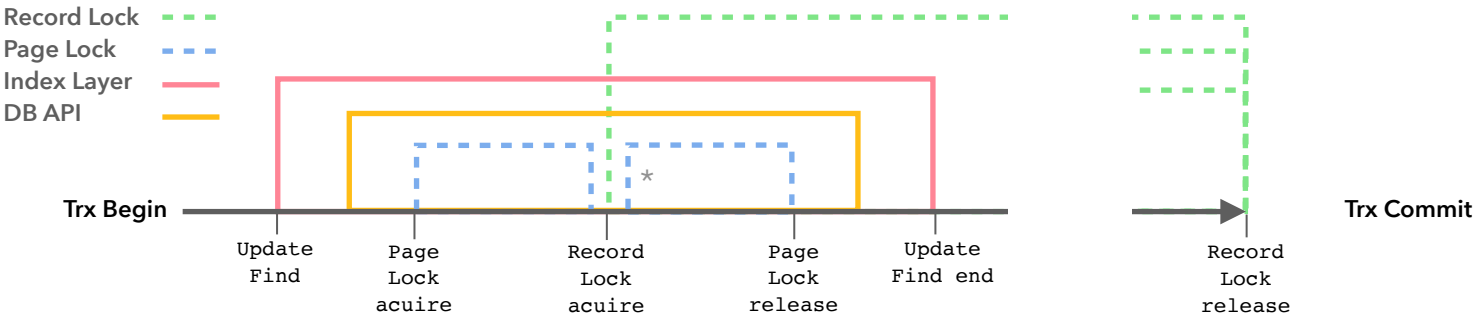
Lock manager는 Concurrency Control을 수행하기 위해 존재하며 레코드락의 구현체와 trx_begin, trx_commit, trx_abort등의 트랜잭션 api가 같이 구현되어 있는 곳으로 여러 트랜잭션이 동시에 수행되더라도 하나의 트랜잭션이 수행되는것과 같은 착시를 일으키는 Councurrency Control을 위해 존재합니다.

Log manager는 Crash recovery를 위해 존재하며 예기치 못한 상황에 DBMS가 종료되더라도 아무런 문제가 없도록 다시말해 “All or Nothing”이라는 키워드를 유지시켜줄수 있도록 존재합니다. Recovery는 사용자가 DBMS를 초기화할때 자동으로 실행되며 log를 기록해 이를 수행합니다. log도 database file과 마찬가지로 cashing이 적용됩니다.

Concurrency Control Implementation



Concurrency Cotrol의 구현은 S2PL(strict two phase locking)을 적용하여 트랜잭션이 시작되면 수행하는 모든동작에 대해 레코드락을 잡다가 트랜잭션이 커밋되는 순간에 락을 모두 놓는 방식을 취합니다. 구현은 lock&transaction manager와 index Layer, Buffer manager Layer에 걸쳐 이루어지게 됩니다. 트랜잭션은 동작의 수행을 위해 buffer manager의 버퍼블록에 구현되어있는 PageLock, lock manager에 구현된 레코드락을 모두 얻어야만 동작을 수행할수 있도록 설계되어있습니다.



위 그림에서 보는 흐름도와 같이 find나 update의 DB API가 호출되면 index layer에서 동작이 수행됩니다. 먼저 동작을 수행할 page를 찾게 되는데 이때 buffuer manager layer에서 해당 페이지에대한 pageLock을 획득하게 됩니다. 또한 page Lock을 획득하는 과정에서 다른 trx들이 buffer pool에 접근하지 못하도록 그동안은 buffer pool latch에 의해 보호 받게 됩니다. Pagelock을 획득하고 나면 bufferpool의 latch를 풀어주게 되며 다시 index layer로 돌아가게 됩니다. 그 후 lock&trx manager에 구현된 lock_aqcuire를 호출해 동작을 수행하고자 하는 레코드에 대한 락을 획득합니다. 락을 획득한 직후에는 페이지락을 다시 획득하고 해당 페이지의 해당 레코드에 대해 동작을 수행합니다.

페이지락은 동작이 완료되면 해제되지만 레코드락은 S2PL디자인에 따라 transaction이 commit될때 한꺼번에 해제됩니다.

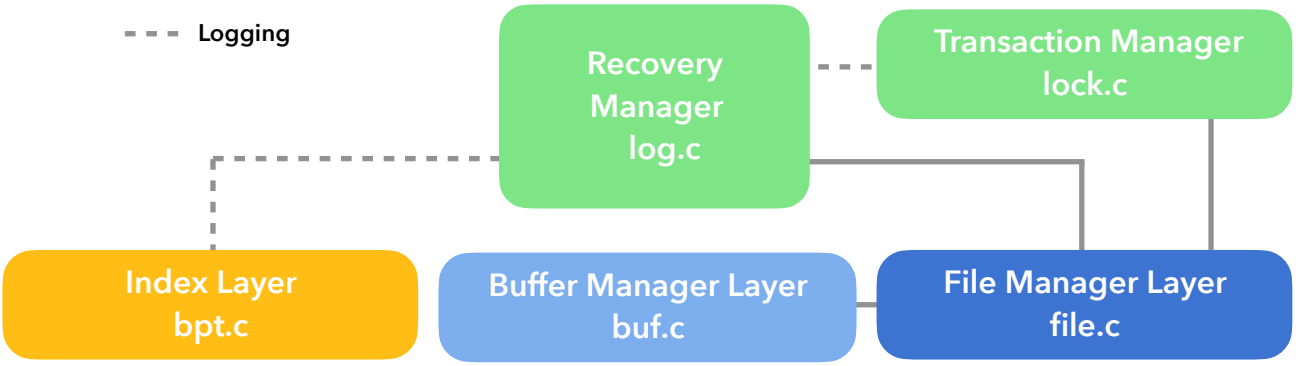
위와같은 정상진행 과정이아닌 레코드락을 얻는 과정에서 다른 transaction이 해당 락을 이미 점유하고 있을 경우 레코드락은 sleep상태에 빠진후 해당 락을 점유한 transaction이 락을 해제 할때 까지 기다립니다. 이과정은 lock manager에서 일어나게 됩니다.

해당락을 점유한 다른 락이있는 경우에 데드락이 감지되면 해당 결과를 index layer에로 보내고 index layer에서는 trx manager의 abort를 호출, 롤백을 진행하게 됩니다. 롤백은 transaction이 이미 수행했던 update 오퍼레이션에 대해 동작 수행시 기록해놨던 로그를 통해 진행됩니다.

Lock manager에서 실제 레코드락을 얻는 과정은 수행하는 transaction table의 trx에 락을 매달고 수행하려는 레코드를 lock table로 부터 찾아 이미 락을 얻은 trx이 있느냐 없느냐에 따라 위에 서술한바와 같이 정상진행, sleep 또는 데드락이 일어나게 됩니다. 물론 lock table과 trx table에 대한 접근은 각각 latch로 보호받게 됩니다.

*실제구현은 페이지락을 획득한채로 레코드락을 획득하는것이아닌 잠깐 페이지락을 해제한체로 레코드락을 취하는 방식으로 구현되었는데 delete나 insert시에 merge 나 split등으로 문제가 발생할수 있지만 이번 프로젝트에서 cocurrency control은 update와 find에 한정되어 구현의 편의상 잠깐 페이지락을 해제했다가 레코드락을 얻은후 락을 얻는 방식을 취하였습니다.

Crash-Recovery Implementation



Crash-Recovery는 WAL(write ahead logging) protocol 의 no force(Redo) steal(Undo) 기법으로 구현됩니다. 로그는 총다섯종류로 Begin, Commit, Rollback, Update, Compensate로 이루어집니다. 이렇게 작성된 로그를 따라 database가 초기화될 때 recovery가 실행되게 되고 이때는 analysis, redo, undo 총 3가지 pass를 따라 리커버리가 진행되며 AREIS기법이 적용되어 빠른 리커버리가 특징입니다.

이런 리커버리의 구현은 총 3개의 레이어와 transaction manager에 걸쳐 구현되었습니다. 먼저 로그가 발급되는 시점에 대해 따라가보면 transaction이 시작되고 종료될때 즉 commit이나 abort가 일어날때 로그가 발급됩니다. 시작할때 begin log를 종료될때 commit이면 commit로그를 abort가 일어나면 Rollback 로그를 발급하며 이런 동작은 모두 transaction manager에서 수행됩니다. 또한 abort가 일어나는경우 Rollback되는 update operation들에 대해 Compensate로그를 발급합니다. Index layer에서는 update가 수행될경우 수행된 update에대해 update log를 발급합니다. 발급은 log manager에서 수행되어 Log를 생성하고 해당 내용을 logbuffer에 쓰게 됩니다. 또한 recovery의 undo pass가 일어날때에도 loser trx의 operation에 대한 undo에대해 abort와 마찬가지로 compensate log를 발급하며 모든 operation을 undo 시키면 rollback log를 발급합니다.

log buffer의 내용이 logfile로 fulsh 되는경우는 WAL protocol에 따라 commit되거나 abort될때, 또 해당page가 evict될때 로그 버퍼의 내용들이 모두 flush 됩니다. 따라서 trx manager에서 commit이나 abort가 일어날때 filemanager의 flush를 호출하게 되고 buffer manager에서 page eviction이 일어날경우에도 file manager의 flush를 호출하게 됩니다.

실제 리커버리 세션은 db초기화 과정이 index layer에서 실행될때 log manager의 recovery를 호출하면서 실행되게 되며 이때 File Manager layer가 호출되어 동작하게 된다 그후 log buffer를 초기화하기 위해 file manager에서 읽은 logfile의 내용을 logbuffer에 쓰게 됩니다. logbuffer의 읽어진 내용을가지고 analysis, redo, undo를 진행해 recovery 를 완료합니다.

In-depth Analysis

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

현재 프로젝트는 Pessimistic Concurrency Control 방식을 취하고 있으므로 find가 발생할때마다 해당 페이지의 전체락을 잡고 레코드락을 획득하기위해 lock_aquire를 수행합니다. Non-conflicting 한 read only trx가 많은 워크로드의 경우에도 페이지 전체에 대한 락을 걸기때문에 concurrent한 read trx가 같은 페이지의 레코드를 읽으려는 경우 어쩔수 없이 concurrency가 떨어지게 됩니다. 또한 매번 레코드락을 획득하기위한 Overhead가 존재하므로 conflict가 일어나지않는 read only trx가 많은 workload에서는 성능 이슈가 있을수 있습니다.

이해대한 해결책으로는 지금처럼 트랜잭션이 작업을 수행하기전에 conflict를 체크하는 방식이 아닌 commit직전에 conflict를 체크하는 OCC(Optimistic Concurrency Control)방식의 BOCC(Backward oriented Councurrency Control),FOCC(Forward Oriented Concurrency Control) 또는 MVCC(Multiversion Concurrency Control)디자인을 차용한다면 락킹이 필요하지 않으므로 추가적인 Concurrency를 얻을수 있습니다

BOCC,FOCC는 read, validation, write 단계로 나뉘며 validation에서 현재 trx을기준으로 이전에 commit된것과 비교하면 BOCC, 나보다 느린 read 단계에 있는 trx들과 비교하면 FOCC입니다. 특히 FOCC는 read only trx에 대해서는 따로 validation단계가 필요없으므로 제시된 workload에서 BOCC보다 더좋은 성능을 보여줄것으로 생각됩니다.

Database의 특정시점의 snapshot을 이용하는 MVCC는 read trx에 대한 blocking이 이루어지지 않으므로 빠르지만 Serializabble Excution을 보장하지 못하는 단점이 있으므로 은행과 같이 high serializability를 요하지 않는 분야에 적합할것 같습니다.

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

현재 프로젝트에 적용된 디자인엔서는 chechpointing기법을 적용하지 않기 때문에 database를 오래사용하면 사용할수록 log가 길어집니다. 특히나 workload가 non-

conflicrting write-only transaction이 많은 상황이라면 crash가 발생했을때 redo pass에서 수행해야한는양이 update log의 길이에 비례해 많을것입니다. 현재 프로젝트에 적용된 consider redo를 적용한다하더라도 계속해서 페이지를 불러와 페이지의 LSN값과 log의 LSN값을 비교해야하므로 update operation에대한 redo는 log가 길어지면 길어질수록 그 부담이 커지고 결국엔 recovery시의 성능이슈로 귀결될것 입니다.

이런 문제를 해결하는 한 방법으로는 checkpointing을 도입하는것입니다. 주기적으로 Database의 상태가 Stable Data Base와 일치하도록 dirty page를 flush 하는 작업(checkpointing)을 추가한다면 이미 new가되버린 내용에대해서는 더이상 log가 필요없기 때문에 log를 truncate시킬수있습니다. 이경우에는 log의 길이가 어느정도의 길이로 유지가 되므로 crash recovery의 redo pass가 줄어들게 될것입니다. 다만 checkpoiting시점에는 lock manager가 점유되므로 다른 update작업은 불가해져 이동안의 foreground trx의 throughput이 떨어집니다. 데이터베이스의 사용률이 낮은시간에 적절한 주기로 checkpointing을 수행한다면 crash recovery시의 성능 이슈를 해결할수 있습니다.