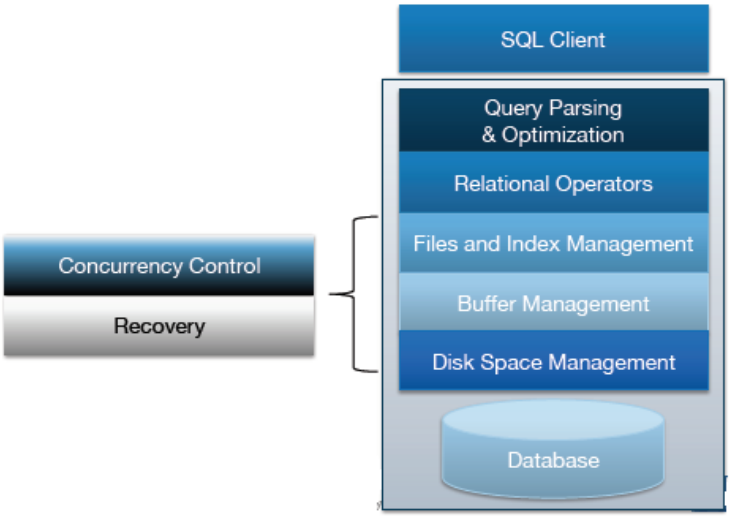
Database System 2020-2
Final Report

Class Code (ITE2038-11801)
2016025678
이강민

Table of Contents

Overall Layered Architecture



(현재 구현한 DBMS의 계층구조를 간단하게 나타내는 사진입니다.)

먼저 가장 하단에 있는 Disk space management는 직접적으로 Database에 접근하여 data를 저장하거나 가져오는 역할을 합니다. Disk space management에서는 data를 page 단위로 다루고 있으며 바로 상위 layer인 buffer management와 data를 주고 받습니다.

Buffer management는 Index management와 Disk space management 사이에 있는 layer로 필수적인 layer는 아니지만 속도를 향상시키기 위해 존재하는 layer입니다. 기본적으로 Disk space management로부터 page를 불러와 buffer에 저장시켜놓고 Index management가 요청하는 buffer를 바로바로 넘겨주는 역할을 하고 있습니다.

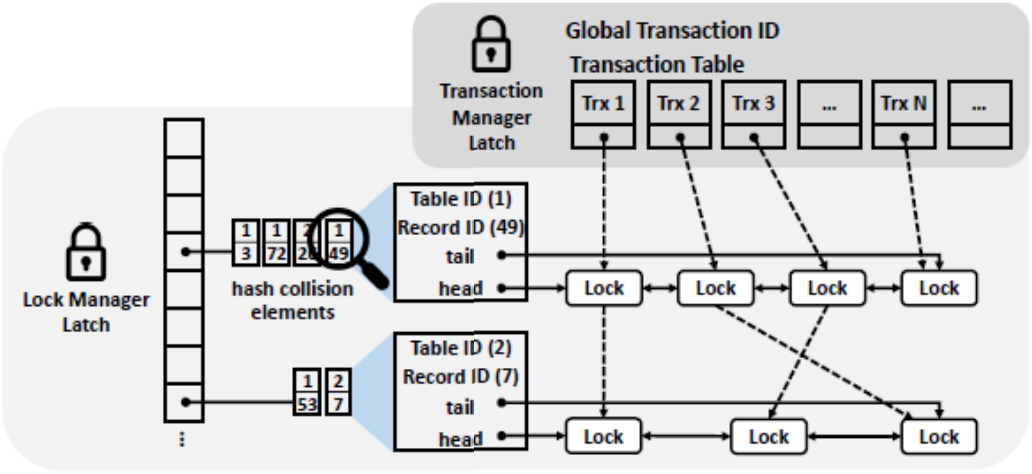
Index management는 상위 layer의 api가 지시하는 명령에 따라 data를 찾거나 삭제하고 추가시켜주는 layer입니다. 제가 구현한 DBMS에서는 DB api의 find, delete, insert를 실행하게 되면 Index management에서 해당 작업을 수행하게 됩니다.

Concurrency Control은 특정 layer에 속하는 것이 아닌 모든 layer를 관통하는 design이며 DBMS의 핵심인 isolation과 consistency를 지켜주고 있습니다. 저의 DBMS는 lock manager를 통해 Concurrency Control을 구현하고 있으며 Strcit-2PL design을 전제로 하고 있습니다.

마지막 Recovery도 Concurrency Control과 마찬가지로 특정 layer에 있는 것이 아닌 모든 layer를 관통하는 design이며 atomicity와 durability를 지켜주고 있습니다. 저의 DBMS는 log manager를 통해 recovery를 구현하고 있으며, No force & Steal 정책을 따르고 있습니다.

Concurrency Control Implementation

제가 구현한 DBMS의 Concurrency control은 Strict-2PL을 기반으로 한 Lock manager로 구현하였습니다.



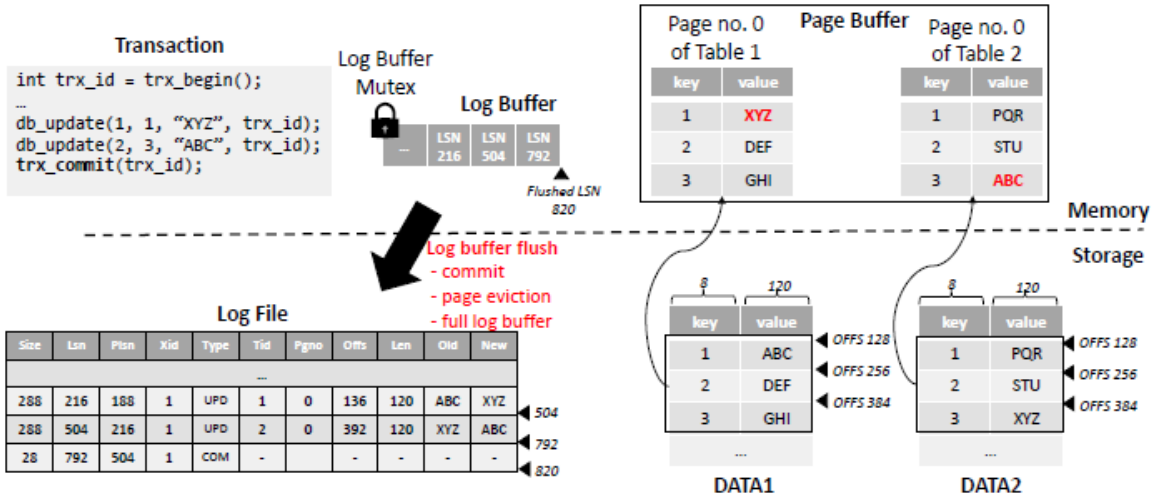
(제가 구현한 Concurrency control를 간단하게 나타내고 있는 사진입니다.)

기본동작을 말하면, transaction이 DB api를 호출하게 되고 호출된 DB api 안에는 해당 data를 접근하기 위한 lock을 얻으려고 시도하게 됩니다. 해당 data에 처음 접근한다면, 해당 lock을 얻으려는 다른 transaction이 없으면 lock을 얻게 되고 있다면 다른 transaction의 lock 획득 여부와 type에 따라 case가 나누어집니다. 앞선 transaction이 lock을 획득한 상태를 먼저 생각해보면, 앞 transaction의 lock type이 Shared일 경우 나의 type도 Shared라면 lock을 획득하게 됩니다. 나의 type이 Exclusive라면 conflict 상황으로 다룹니다. 앞 transaction의 lock type이 Exclusive일 경우 나의 type과 상관없이 conflict 상황으로 처리합니다. 앞 transaction이 lock을 획득하지 못한 상황을 생각해보면, 앞 transaction의 lock type에 상관없이 또한 나의 type도 상관없이 모두 conflict 상황으로 처리합니다. conflict 상황이 되었을 경우 deadlock detection을 이용하여 deadlock을 검사하고 deadlock이 아니라면 wait상태인 lock을 생성해줍니다. 만약 현재 넣을려고 하는 transaction이 해당 data에 lock을 이미 갖고 있는 경우 먼저 만들어 놓은 lock type이 shared 이며 그 사이에 다른 transaction의 lock이 있고 넣으려는 lock의 type이 Exclusive인 경우만 conflict 상황으로 처리합니다.

Lock manager는 index layer와 buffer layer에 걸쳐 구현되어 있습니다. 원래 buffer에는 pin을 이용하여 buffer가 사용중인지 판단하였지만 lock manager를 구현하면서 mutex를 사용하게 되었습니다. buffer pool을 한 번에 하나의 transaction이 사용할 수 있도록 바뀌었고, buffer에서 page를 획득할 때 하나의 transaction만이 접근할 수 있도록 page에도 mutex를 사용하게 되었습니다. Index layer에서 buffer를 사용하는 경우 deadlatch가 일어나지 않도록 buffer get과 put을 사용하도록 바뀌었습니다.

Crash-Recovery Implementation

제가 구현한 DBMS의 Crash-Recovery는 No force & Steal 정책과 Write Ahead logging 프로토콜을 기반으로 한 Log manager로 구현하였습니다.



(제가 구현한 Log manager의 대략적인 동작 사진입니다.)

Log manager의 평소 동작은 transaction이 발생하는 순간부터 transaction이 하는 update나 find같은 db api를 실행하고 여러 상황에 따라 commit이나 abort가 되기까지 일어나는 것을 모두 기록하는 것입니다. 이후 Crash가 발생하고 다시 DBMS를 작동시키면 Recovery가 작동됩니다.

Recovery의 기본동작을 말하면, 3가지 Pass(Analysis pass, Redo pass, Undo pass)로 이루어져 있습니다. 제가 구현한 Recovery의 경우 Analysis pass와 Redo pass를 동시에 구현하였습니다. 먼저 기존에 기록되어 있는 log를 모두 읽어와 처음부터 Crash가 난 지점까지의 log를 모두 실행하는 Redo가 일어납니다. 제가 구현한 Redo의 경우 transaction이 Begin되는 log가 있다면 모두 loser list에 추가시켜줍니다. 이후 commit이나 abort를 완료한 transaction이라면 loser list에서 제외합니다. Redo가 모두 종료가 된 후 Undo를 실행시켜줍니다. Undo는 loser list에 있는 transaction의 log들을 반대로 실행해 없던 것처럼 만들어주는 함수입니다. Crash가 발생한 지점부터 맨 처음까지 작동하며 만약 해당 log가 compensate log일 경우 next undo LSN을 이용하여 좀 더 효율적인 undo를 하게 합니다.

Log manager는 buffer layer, disk space layer에 걸쳐 구현되어 있습니다. Redo를 좀 더 효율적으로 하기 위해 Redo하는 log가 data를 바꾸는 행위일 경우 이미 database에 적용되었는지 확인하는 consider redo를 구현하였습니다. 이를 구현하기 위해 data를 포함하는 page에 page LSN을 추가하였습니다. buffer의 경우 page eviction을 할 때 database에 반영하기 때문에 WAL 프로토콜에 따라 log를 모두 flush 해주어야 합니다. 또한, log 자체를 관리하는 log buffer도 필요하게 됩니다.

In-depth Analysis

1. Workload with many concurrent non-conflicting read-only transactions.
   (많은 수의 non-conflicting read-only transaction 이 동시에 수행될 경우 발생할 수 있는 성능 측면에서의 문제점을 설명하고, 이를 해결할 수 있는 디자인을 제시할 것)

   (본인의 최종 프로젝트 코드 및 디자인을 기반으로 설명할 것)

   (non-conflicting: access different records each other)

   현재 구현한 DBMS로 많은 non-conflicting read-only transaction을 동시에 수행하게 될 경우 한 번에 하나의 transaction이 lock table latch를 잡고 lock 획득을 시도하기 때문에 conflict가 없는 상황임에도 불구하고 성능이 느릴 것으로 예상됩니다.

   제가 제안하는 디자인은 lock table을 관리하는 latch를 하나로 하는 것이 아니라 각 record id마다 혹은 범위를 정해서 각 범위마다 latch를 만들어 관리하는 방법입니다. 이런식으로 구현하였을 경우 서로 다른 transaction이 서로 다른 record의 lock을 획득할 때 기다리지 않고 lock 을 얻을 수 있게 됩니다.

2. Workload with many concurrent non-conflicting write-only transactions.
   (많은 수의 non-conflicting write-only transaction 이 동시에 수행될 경우 발생할 수 있는 **Crash-Recovery 와 관련된** 성능 측면에서의 문제점을 설명하고, 이를 해결할 수 있는 디자인을 제시할 것)

   (본인의 최종 프로젝트 코드 및 디자인을 기반으로 설명할 것)

   현재 구현한 DBMS로 많은 non-conflicting write-only transaction이 동시에 수행하게 될 경우 Undo에서 성능저하가 발생할 것 같습니다. 모든 transaction이 write-only이기 때문에 모든 loser가 write only이므로 compensate log를 만날 때마다 next undo LSN으로 바꾼 후 다시 돌아가 Undo를 수행하기 때문에 성능저하가 발생할 것 같습니다.

   제가 제안하는 디자인은 compensate log를 발급할 때 상쇄되는 update log를 찾아서 표시해 두는 것 입니다. 이를 통해 Undo 실행시 compensate log는 모두 무시할 수 있으며 또한, 표시가 되어있는 update log도 무시할 수 있기 때문에 기존 디자인보다 성능이 좋아질 것 같습니다.