Database System 2020-2
Final Report

Class Code (ITE2038-11800)
Student Number 2019062833
Name 김유진

Table of Contents

1. Overall Layered Architecture

1-1 DBMS의 계층 구조

DBMS(data base management system)는 말 그대로 데이터 베이스를 관리하여 주는 시스템이며 데이터 베이스란 데이터들을 잘 구조화해서 정리한 데이터의 집합을 말한다. 시간이 지남에 따라 정보의 양은 기하급수적으로 증가하고 있고 넘쳐나는 정보들을 제대로 체계화하여 두는 것이 더욱더 중요해지고 있다. 데이터를 '잘' 구조화해서 저장하고 '다수의' 사용자들이 문제없이 정보를 사용하게 하려면 DBMS는 사용자와 데이터 베이스 사이에서 중요한 중계자가 되어야한다.

이를 위해 다양한 임무가 주어지게 되는데 모든 일을 효율적이며 안전하게 처리하기 위해 DBMS는 계층 구조로 이루어지게 된다. 계층 구조로 나뉘어 작업을 수행하게 되면 각 계층은 자신에게 주어진 임무에 대해서만 최선을 다하여 수행하고 이외의 일은 신경 쓰지 않고 해결할 수 있어 복잡한 요구사항이 쉽게 해결되며 성능적인 측면에서도 더 좋은 결과를 낸다.

1-2 각 계층 간의 상호 관계

각 계층은 상호 연관된 필요한 기능을 모아서 만들어 진다. 그렇다면 DBMS가 제공하여야 하는 기능은 어떤 것들이 있는지 알아볼 필요가 있다. 기능은 크게 5가지로 나눌 수 있다.

첫째로 사용자에게서 받아온 요구사항, 즉 query에 대해 parsing하고 최적화하는 기능이다. parsing이란 간단하게 얘기해보자면 사람에게 친숙한 표현을 기계에게 친숙한 표현으로 변경하는 것이다. 이렇게 변경된 내용을 토대로 optimizer가 계획을 최적화하기 시작한다. 하나의 쿼리에 맞는 답을 가져오는 과정은 하나로 정해진 것이 아니라 relation algebra의 다양한 순서와 조합으로 이루어져 있다. 최적화 과정에서는 다양한 조합을 시도하여 최소한의 비용으로 쿼리를 해결하도록 한다. 이를 위해 필요한 또 다른 기능이 있다. 두번째 기능인 Relational operator이다. 이는 실제 relational operation을 데이터에 적용하여 optimizer가 이를 토대로 계획을 수정하고 최적의 plan을 짜도록 보조하는 역할을 한다.

남은 기능들을 설명하기에 앞서 Disk 내부의 구조에 대한 정보를 설명하자면 Disk의 file은 여러 페이지로 이루어져 있고 그 페이지는 레코드들로 이루어져 있다. 단순히 어떤 레코드를 찾고 싶다고 하면 어느 파일에, 어느 페이지에, 몇 번째 레코드인지에 대한 정보를 알아야 한다. 정보를 잘 정리하고 쉽게 찾기 위해서는 이러한 파일, 페이지, 레코드 관리가 매우 중요하다. 디스크의 정보관리를 위해 추가적으로 3가지 기능이 더 필요하게 된다.

세번째는 파일과 인덱스 관리 기능으로 pagination과 레코드 관리를 담당한다. 실제 파일은 우리가 생각하는 것처럼 네모난 페이지들로 이루어져 있지 않고 그저 바이트로 연속되어 있기 때문에 관리를 용이하게 하기 위해 logical한 규격을 잡아서 분리해줄 필요가 있다. 임의로 구획을 나누고 레코드를 담아서 연결시키는 작업을 한다. 위 계층에서 받아온 필요한 레코드의 정보를 가지고 logical한 페이지 위치 정보를 아래 계층으로 전달하여 해당하는 페이지의 실제 disk의 offset에서 정보를 받아온다. 이 과정에서 네번째 계층인 Buffer management와 마지막 계층인 Disk space management가 필요하게 된다.

네번째 계층인 버퍼 매니지먼트는 사실 꼭 필요한 기능은 아니다. 이 계층이 존재하는 이유는 우리가 디스크에서 정보를 읽어올 때 상당한 시간이 소요되기 때문에 성능개선을 위해 캐시역할을 하도록 하여 완충재로써 사용하기 위함이다. 디스크보다 속도가 빠른 메모리를 사용하여 페이지를 이곳에 읽어 두고 정보를 읽고 쓰기를 하는 것이다. 이를 통해 데이터가 마치 메모리에 있는 것처럼 하여 빠른 속도로 정보를 사용할 수 있게 된다. 필요한 페이지가 버퍼에 이미 존재한다면 해당 페이지를 위 계층으로 올려주고 그렇지 않다면 아래 계층에게 해당 페이지를 요청하여 올려 두고 사용한다.

네번째 계층의 요청을 받은 disk manager는 필요로 하는 페이지 영역을 실제 디스크의 offset, 즉 물리적 바이트 형태로 번역하여 정보를 읽어 들여서 전달한다. 이 과정이 끝나게 되면 사용자의 쿼리에 필요한 레코드 요청에 대해 실제 디스크에서 해당하는 정보를 읽어서 전달해주는 모든 작업이 끝나게 된다. 이처럼 각 계층은 자신에게 주어진 할 일을 독립적으로 진행하되 그 이외의 일에 대해서는 다른 계층으로의 인수를 통해 일을 처리하게 된다.

2. Concurrency Control Implementation

2-1 Concurrency Control(동시성 제어) 기법

DBMS는 동시성 제어를 통해 다수의 사용자들이 동시에 DB에 접근이 가능하도록 해준다. 동시성 제어는 DBMS가 반드시 보장해야 하는 ACID(Atomicity, Consistency, Isolation, Durability)에서 C와 I를 보장하게 된다. Consistency란 DB가 consistent한 상태로 시작되었다면 transaction이 끝난 후에도 consistent한 상태로 끝나야 한다는 것이고 Isolation이란 각 실행이 온전히 혼자서 실행되는 것 같이 되는 것이다. 결국 동시성 제어의 최종 목적은 각 transaction의 실행들을 conflict serializable하게 만드는 것이다.

이를 위한 동시성 제어 기법은 크게 두가지로 나눌 수 있다. 하나는 Pessimistic 동시성 제어이고 다른 하나는 Optimistic 동시성 제어이다. Pessimistic은 말그대로 비관적으로 생각하여 Conflict가 발생한다고 전제하여 그 이전에 조치를 취하는 것이고 Optimistic은 낙관적으로 conflict가 나지 않을 것이라고 생각하고 이후에 검사를 진행하고 조치를 취하는 방식이다.

비관적인 동시성 제어는 Locking을 기반으로 작동한다. Transaction은 read를 수행하기 전 해당 레코드에 대한 공유모드의 lock을 잡아야 하며 write를 수행하기 전에는 해당 레코드의 독점모드의 lock을 반드시 획득해야 한다. 한 레코드에 공유모드 lock이 걸려있는 경우 다른 transaction이 공유모드로 lock을 획득하여 사용하는 것이 가능하고 독점모드 lock이 걸려있는 경우 당장은 사용할 수 없고 lock을 잡은 후 sleep 상태로 들어가서 lock을 획득할 수 있을 때까지 기다리게 된다.

한 transaction은 자신의 연산을 수행하면서 lock을 계속해서 획득하고 모든 lock을 다 획득하고 연산을 끝마치면 lock을 해제하는 단계에 진입한다. 이렇게 획득 단계와 해제 단계로 나누어진 locking 방식을 two phase locking(2PL)이라고 일컫는다. 그러나 cascading abort 문제를 해결하기 위해 strict 2PL 방식을 도입하게 되는데 앞의 방법과의 차이점은 단하나로 해제 시점을 commit 시점까지 미뤄두어 한번에 해제를 진행한다는 점이다. 이러한 lock들을 관리해주기 위해 lock manager가 존재하게 되고 각 레코드에

대한 transaction들의 lock과 unlock의 요청을 처리하게 된다.

그러나 Locking 방식을 사용하게 되면 Deadlock이라는 문제가 발견된다. Deadlock이란 transaction들이 서로를 기다리는 상황이 발생하여 영원히 멈춰져서 끝나지 않는 상황을 말한다. 이를 처리하기 위한 방법으로 2가지가 있는데 Deadlock Avoidance와 Deadlock Detection이다. Avoidance는 Deadlock이 발생하기 이전에 미리 막는 방법으로 priority가 높아지는/낮아지는 방향으로만 lock을 잡도록 하고 이외의 상황에서는 abort를 행하는데 선제적으로 abort를 실행하게 된다는 단점이 있다. Detection 방식에서는 주기적으로 사이클이 생기는지 검사하고 사이클을 끊어주는 작업을 한다.

마지막으로 생각해볼 점은 lock을 획득하여 작업을 할 때 어떤 크기의 범위로 lock을 잡아야 하는 가이다. 모든 transaction에 대해 동일한 범위의 lock을 해서는 안된다. Locking Granularity라는 용어는 locking의 범위를 말한다고 볼 수 있다. Fine granularity로 lock을 잡을수록 높은 동시성이 제공되지만 그만큼 locking overhead가 커지게 된다. 상황에 맞게 알맞은 범위로 lock을 잡는 것이 중요하다. 이를 위해 앞서 설명한 공유모드와 독점모드 이외에도 다양한 모드의 lock이 필요하게 된다. 바로 Intention mode로, 이는 fine/coarse 범위에서 작업하는 서로 다른 transaction들이 conflict에 안전하게 수행되도록 하는 것이 목적이다.

Locking은 lock 관리라는 부담이 있을 뿐 아니라 Deadlock 감지와 해소에 대한 부담까지 존재하는 단점이 있다. 그렇기 때문에 conflict가 빈번하지 않는 상황의 경우 낙관적인 관점에서 동시성 제어를 하는 방법을 생각해볼 수 있다. 낙관적인 동시성 제어는 또 세가지로 나눌 수 있는데 하나는 FOCC(forward based optimistic CC)이고 또 다른 하나는 BOCC(backward based optimistic CC). 그리고 Snapshot Isolation 방법이 있다.

FOCC와 BOCC는 둘 다 세 단계로 이루어진다는 점에서는 동일하다. Read phase/Validation phase/Write phase 이 3단계로 이루어져 있는데 두 방식의 차이는 validation 단계에서 드러난다. 가장 먼저 Read 단계에서는 사적인 작업공간으로 데이터를 copy를 해와서 transaction을 수행하고 validation 단계로 진입하여 commit 이전에 체크를 하게 된다. 이 단계를 통과하지 못하게 되면 abort가 일어나게 되고 그렇지 않을 경우 다음 단계로 넘어가게 된다. BOCC에서의 validation 단계에서는 이전의 commit된 transaction의 write set과 현재의 read set을 비교하게 되고 FOCC에서는 아직 valid 단계로 들어오지 않은 동시에 진행중인 transaction의 read set과 자신의 write set을 비교하게 된다. write 단계로 넘어오게 되면 데이터베이스로 자신의 사적 작업공간의 내용을 전달한다.

Snapshot Isolation(IS)는 멀티버전의 동시성 제어라고도 한다. 시간이 지나면서 write가 발생할 때마다 그 시점의 새 데이터베이스 복사본을 찍어 둔다고 생각하면 된다. Read 시에는 해당 transaction의 시작 시점을 기준으로 가장 최근 버전의 데이터베이스의 정보를 읽어오게 된다. Write의 경우 First committer win 규칙을 적용하여 이후에 commit이 된 transaction을 abort하는 방식을 사용한다. 이 방식의 장점은 read 연산이 다른 것에 구애를 받지 않고 빠르게 수행될 수 있다는 점이고 추가적으로 dirty read, lost update, inconsistent read와 같은 문제도 피할 수 있게 된다. 그러나 이 경우 transaction에서 수정된 항목의 이전 상태에 따라 서로 다른 항목을 수정할 때 serializable를 보장하지 않아서 Write skew 문제가

발생할 수 있다는 단점이 존재한다.

2-2 계층적 관점

직접 구현하였던 프로젝트에서는 비관적 동시성 제어인 locking 방식을 기반으로 하여 구현하였기 때문에 이를 바탕으로 설명을 할 것임을 밝힌다.

동시성 제어는 3번째 계층부터 5번째 계층까지 총 3가지의 계층에 더해 추가적으로 lock management 계층에 걸쳐서 구현이 되어있다. 앞서 설명하였던 계층 구조에서는 5가지의 계층만 언급하였지만 다수의 사용자가 존재할 경우 DBMS는 추가적인 계층을 필요로 할 때도 있다. 바로 Transaction management 계층이고 이는 다시 두가지 역할로 나뉘게 되고 동시성 제어를 위한 lock management와 crash recovery를 위한 logging 파트이다.

여러 transaction들이 read/write 연산을 수행하기 시작하였을 때 가장 먼저 3계층인 file, index management 계층에서 해당하는 레코드에 대한 logical page와 그 내부에서의 offset을 파악하고 그 페이지를 아래 계층에서 받아오려고 한다. 동시성 제어를 위해 lock 기능에 더해 mutex기능도 사용하게 되는데 버퍼 계층에서 버퍼풀의 mutex 잠금과 해제, 탐색 중인 페이지에 대한 mutex 잠금과 해제가 일어나게 된다. 만약 현재 버퍼 풀에 필요한 페이지가 올라와 있지 않다면 아래 계층인 디스크 매니저에게 디스크에서 해당 페이지의 부분을 버퍼로 읽어오도록 지시한다. 페이지를 발견한다면 해당 페이지를 위의 index manager로 넘겨주게 되고 이 계층에서 lock/transaction manager의 도움으로 레코드에 대한 lock 획득과 해제가 일어나게 된다.

동시성 제어와 관련된 핵심 계층은 Lock manager 계층이다. 이 방식의 동시성 제어의 핵심은 Lock이고 각 사용자마다, 각 레코드마다의 Lock을 관리하는 과정이 가장 중요한 부분이다.

3. Crash-Recovery Implementation

3-1 Crash Recovery 기법

DBMS는 Crash recovery를 통해 작동 도중 crash가 발생하여도 Atomicity와 Durability를 보장하며 온전하게 복구될 수 있도록 한다. Atomicity(A)란 하나의 transaction이 온전히 실행이 다되거나 아예 아무 일도 일어나지 않거나 둘 중에 하나의 결과만 내는 것이고 Durability(D)란 transaction이 commit이 되었다면 어떤 일이 일어나도 결과가 남아있어야 한다는 것이다.

Recovery를 이해하기 위해서는 먼저 logging에 대한 것을 이해할 필요가 있다. Log란 순서가 있는 기록물이다. 복구를 진행하기 위해서는 crash 이전의 상태에 대한 정보가 필요하다. 이때 log의 정보를 통해 복구가 가능하게 되는데 log의 안전하고 효율적인 저장을 위해 log buffer(RAM)와 log file(Disk)을 따로 두고 사용하게 된다. 우선 Log는 수행하는 일에 따라 commit / update / rollback / begin / compensate 등의 타입으로 나뉘게 된다. log 레코드는 증가하는 고유한 번호가 주어지게 되고 차례로 log buffer에 쌓이게 된다.

주기적으로, 또는 특정 명령이 주어지면 flush를 통해 버퍼의 log를 disk로 내려서 저장하게 되는데 이때 따르는 방식이 WAL(write ahead logging)이다. 해당 페이지가 디스크에 도달하기 전 업데이트를 위한 로그 레코드를 강제로 파일로 내리게 되고 또한 commit 하기 전에 transaction에 대한 모든 로그 레코드를 강제로 log file로 내려서 모든 로그가 파일로 무사히 내려가기 전에는 commit이 되지 않는다.

Recovery 단계에 대한 설명에 들어가기 앞서 현재 대부분의 DB에서 사용하는 Redo-History 방식인 ARIES를 기준으로 기술할 것임을 밝힌다. Crash recovery는 3가지 단계로, 바로 분석 단계, Redo 단계, Undo 단계로 이루어져 있다. Crash 이후 DB가 재가동이 되면 recovery를 시작하면서 로그 파일을 읽어들이고 분석단계가 시작된다. 이 단계에서는 loser와 winner를 구분해내는 작업을 하게 되는데 이때 loser란 begin이 된 transaction 중에서 crash 이전에 commit또는 abort가 온전히 되지 못한 transaction을 말한다.

이후 redo 단계로 진입하게 되고 winner인지 loser인지에 상관없이 오래된 log부터 모든 update 또는 compensate(inverse operation) log를 재실행을 하게 된다. 이때 더 빠르게 회복을 진행하기 위해 consider-redo가 일어나게 된다. 이는 해당 페이지의 최근의 LSN과 해당 log의 LSN을 비교하여 redo를 진행할 필요가 없다고 판단이 되면 실행하지 않고 넘어가는 것을 말한다.

Redo 단계가 crash 없이 정상적으로 끝나게 되면 Undo 단계로 진입한다. Loser로 구분된 transaction들의 update 및 compensate log를 가장 최신의 것부터(redo와 반대 방향) undo하면서 compensate log를 발급한다. 그러나 만약 undo 단계에서 계속해서 crash가 발생할 경우 다시 recovery 시에 compensate의 compensate log를 계속해서 발급하게 되므로 불필요한 undo를 반복하게 되어 너무 많은 시간이 소요될 수가 있다. 이를 해결하기 위해 compensate log는 next undo LSN 값을 보유하고 있으며 undo 단계에서 crash가 발생해도 이 값을 통해 이전의 undo 단계에서 진행했던 불필요한 inverse 연산을 계속해서 반복하지 않고 crash가 난 지점부터 계속해서 undo를 진행할 수 있게 해준다.

3-2 계층적 관점

Crash recovery가 시작되면 log file을 읽어오고 log 레코들의 정보를 바탕으로 데이터베이스를 구현하기 시작한다. 예를 들어 1번 페이지의 3번 레코드가 a에서 b로 변경되었다는 내용의 log를 읽었다고 가정해보자. 그렇다면 1번 페이지의 정보를 읽어와서 last LSN(해당 페이지에 대해 최근에 발급된 로그 번호) 값을 확인하게 되고 만약 현재의 log의 LSN이 더 큰 숫자라면 변경내용이 해당 페이지에 적용이 되지 못했다는 뜻이 되고 log 정보의 적용이 일어나게 된다.

이 과정에서 DB 내부에서의 상황을 살펴보자면 해당 레코드가 들어있는 페이지 정보를 읽어와서 last LSN 값을 확인하는 작업을 하게 된다. 또한 만약 변경사항이 적용되어야 한다고 판단이 되면 해당 레코드의 값을 수정하는 작업이 필요하게 된다. 이때 disk manager, buffer manager, index manager의 도움이 필요하게 되여 3,4,5 계층이 관여하게 된다. 이 3개의 계층 모두 직접적으로 recovery 기능을 구현하는 것은 아니지만 간접적으로 log management를 보조하게 된다.

Crash recovery의 핵심은 다수의 사용자가 존재할 때 필요한 DBMS의 추가적 계층으로 Transaction

manager의 한 부분인 logging manager이다. "Log가 Database이다" 라고 말할 수 있을 정도로 DB의 복구는 온전히 Log를 기반으로 이루어지게 된다.

4. In-depth Analysis

4-1. Workload with many concurrent non-conflicting read-only transactions.

1) 성능 측면의 문제점

이번 프로젝트에서는 동시성제어를 Locking을 기반으로 하는 비관적인 동시성 제어 방법을 사용하였다. 이 방식에서는 read 연산시에는 공유 모드 lock을, write 연산시에는 독점모드 lock을 획득할 수 있도록 lock을 관리하는 비용이 들게 되었는데, 읽기 연산은 같은 레코드에 대해 동시에 가능하기 때문에 write가 들어오지 않고 항상 read연산만이 요청된다면 불필요하게 lock에 대한 연산을 계속하게 된다.

처음 연산이 요청되었을 때 어떠한 연산인지에 따라 다른 모드의 lock을 획득하게 되고 lock을 잡지 못하게 되는 상황인지, 업그레이드가 필요한 상황인지, Deadlock이 발생하는 상황인지 등과 같은 모든 locking 과정이 낭비되어 일어나게 된다. 이와 같은 문제를 해결하기 위해서는 여러 동시성 제어 방식들의 특징을 고려하여 생각해볼 필요가 있다.

2) 해결을 위한 디자인 제시

비관적 동시성 제어는 conflict 상황이 빈번하게 발생할 것이라는 가정하에서 이루어진다. 반면 낙관적 동시성 제어는 conflict 상황이 거의 발생하지 않을 것이라는 관점에서 제어를 한다. 그렇기 때문에 Read only 만이 계속되는 상황이 발생하게 되면 locking overhead를 줄일 수 있는 낙관적 동시성 제어를 사용하는 것이 더 좋은 성능을 낼 수 있게 된다.

4-2. Workload with many concurrent non-conflicting write-only transactions.

1) 성능 측면의 문제점

일반적인 상황과 다르게 non conflicting writer 가 동시에 수행되는 상황이 발생하게 되었을 때 locking을 기반으로 구현된 DB의 상황을 살펴보면 각 transaction들이 다른 레코드에 대해서만 lock을 획득하게 되기 때문에 sleep을 하게 되어 다른 transaction의 commit 지점까지 기다리게 되는 상황이 일어나지 않게 된다. 원래라면 Begin1 Begin2 … …..trx1(a update) …. Commit1 trx2(a update) 와 같이 다른 transaction의 commit을 기다리고 자신의 연산이 수행된다면 transaction의 길이가 계속해서 길어지는 상황이 발생한다. 그렇기 때문에 특정 시점에 crash가 난 경우 begin이 되었는데 commit(또는 abort)가 제대로 끝나지 못한 transaction이 비교적 많아지게 된다.

반면 conflict한 상황이 일어나지 않게 되면 각 transaction들은 다른 transaction의 연산을 기다리거나 하지 않고 곧바로 업데이트를 진행할 수 있기 때문에 길이가 비교적 짧아지게 되는 경향이 있다. 이 경우 특정 시점에서 crash가 발생할 경우 crash발생 이전에 begin이 되었던 transaction들 중 이미 commit 또는 abort 작업이 온전히 끝난 transaction의 비율이 높아지게 된다. 즉, loser의 비율이 줄어들게 되고 앞쪽에서 시작되었던 transaction들은 이미 끝나 있을 확률이 높기 때문에 그 loser들은 전체 시간을 놓고

봤을 때 뒤쪽에 몰려 있을 확률이 높다.

이러한 상황에서 Crash recovery에서 나타날 수 있는 성능적인 문제를 고민해보면 분석단계에서 문제가 발생할 수 있음을 알 수 있다. 이번에 logging 기능을 구현한 프로젝트에서 나의 분석단계에서의 방식은 로그파일의 첫 로그부터 쭉 읽어 들이면서 begin이 된 transaction들에 대한 commit이나 abort log가 이후에 발급이 되었는지를 확인하는 방식이었는데, 앞쪽에서 확인하지 않아도 될 수 있는 transaction에 대한 log를 모두 읽어보아야 한다는 문제가 발생하게 된다.

2) 해결을 위한 디자인 제시

앞에서부터 모든 로그를 차례로 읽어보는 방식이 아닌 뒤쪽으로 몰려 있는 loser를 효율적으로 가려낼 수 있는 새로운 분석 단계 구현을 시도해야 할 것이다.