Final Report

- **DEADLINE: 2020. 12. 20. 23:59**
- You can use any editor program to write the report, however, you must convert the report file into a **pdf file**, named **final_report.pdf**. Then, upload (git push) it into the **final_report** directory of the hconnect project until the deadline.
  (e.g., *your_project_path*/final_report/final_report.pdf)
- You can use either Korean or English, and there is no plus point at all in using English.
- Feel free to use whatever you have written in the Wiki.
- The report should contain the following,

1. Table of Contents

2. Overall Layered Architecture

3. Concurrency Control Implementation

4. Crash-Recovery Implementation

5. (Important) In-depth Analysis
   (This will be the main assessment item, so it must be written very carefully and logically.)

- See the details in the following report templates.

[Template]

Database System 2020-2
Final Report

Class Code (ITE2038-11800)
2016025105
강재훈

Table of Contents

Overall Layered Architecture

(전체 layered architecture 의 구조 및 layer 간의 상호 관계 등에 대하여 자유롭게 기술)

DBMS는 일반적으로 layered architecture로 구현되어 있고 query parsing & optimization, relational operators, files and index management, buffer management, disk space management로 이루어져 있고 추가적으로 lock manager와 logging & recovery가 필요하다고 알고 있습니다. 하지만 현재 제가 구현한 system에선 query parsing & optimization, relational operators 부분은 생략되어 files and index management, buffer management, disk space management, lock manager, logging & recovery 로 구현되어 있습니다. 이제 각 layer에서 하는 일들, layer들 간의 상호 관계, 프로젝트를 진행하며 제가 구현한 방식에 대해 설명해보겠습니다.

query parsing & optimization, relational operators (DBMS의 Front-end)

해당 layer에선 query를 입력받아 query parser가 parsing을 하고 parsing된 query를 query rewriter가 relation algebra로 만들어줍니다. query optimizer는 query rewriter가 만들어준 plan tree를 가지고 같은 result를 갖는 다양한 tree를 plan generator에서 generate합니다. 이후 이 다양한 tree들의 cost를 plan estimator에서 calculate가 아닌 estimation 하게 되고 가장 작은 cost를 갖는다고 생각되는 plan을 가지고 query executor가 query를 실행합니다.

이러한 과정이 가능한 이유는 우리가 사용하는 query는 relational calculus (관계해석, 결과가 어떤 조건을 만족 시켜야 하는지 기술한 것)인데 relational calculus와 relational algebra(관계대수, 주어진 relation으로 부터 해당 작업의 순서를 알려주고 그 순서로 작업을 하면 최종적으로 우리가 원하는 결과를 얻을 수 있다는 것을 보장하는 computational operation description)이 같은 표현력을 갖기 때문입니다.

따라서 query에 다양한 기술&기법들 (ex) selection에 대해선 cascade와 push down을 하는게 좋다 등의 common heuristic, join은 simple nested loops join, sort merge join 등으로 표현 가능 등등) 을 적용해 다양한 형태로 만들고 그 중 cost가 가장 적다고 estimation 되는 plan을 선정 후 실행하는 것이 DBMS의 Front-end에 해당하는 layer들의 역활입니다.

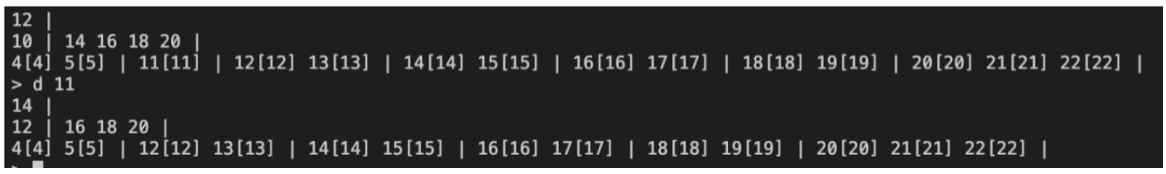
files and index management

해당 layer부터는 DBMS의 Back-end로 볼 수 있습니다. Back-end에선 Front-end에서 계산해준 plan을 실제로 실행하는 일을 합니다. files and index management layer는 memory에서 실행됩니다. 이 layer의 주목적은 DB에 저장된 table을 어떤 형식으로 관리할지 정합니다. table의 내용들은 page에 나뉘어 저장되게 되는데 이 page들을 어떻게 관리할지가 핵심입니다. 예를들어 page들을 별도의 순서 없이 unordered heap file로 관리 할 수도 있고 index를 주어 ISAM이나 B+tree로 관리할 수도 있습니다.

우리의 project에선 B+tree를 채택하였습니다. 각각의 page들은 B+tree에서 node로 사용됩니다. 채택한 B+tree는 leaf node에 key와 record가 들어있고 internal node는 이 leaf node를 빠르게 찾기 위해 routing 하는 역할을 합니다. 따라서 insert와 delete에 꽤나 효율적으로 대응할 수 있고 (split이나 merge가 자주 일어나지 않는다면 효과적, 일어난다고 해도 rebalance하는 cost는 들지만 balance한 tree를 유지한다는 점에서 유익함) 효율적인 searching(index를 이용한 빠른 search, balanced tree를 유지하기 때문에 record들에 대한 search 시간이 거의 일정해짐)이 가능하게 해줍니다.

B+ tree의 성능을 상향 시키기 위해 엄청 많은 data를 insesrtion 하는 경우에 bulk loading을 하는 것과 ping pong problem을 줄이기 위해 delete 시 redistribution을 최소화 하고 merge를 lazy하게 하는 방식이 있습니다. 이에따라 저는 node에 key가 모두 사라져야 merge 되는 명세의 delayed merge를 구현하였고 internal node가 사라져 neighbor와 merge해야 하는 상황에서 neighbor가 가득 차 merge 하지 못하는 경우에만 redistribution을 하도록 구현하였습니다.

ex) 11을 delete 한 후 merge가 일어나야 하지만 neighbor internal이 가득 차 (4[4], 5[5]) leaf가 merge 될 수 없는 상황 -> redistribution



추가적으로 제 구현에선 해당 layer에 table manager와 transaction manager를 두었습니다. table manager는 여러 table이 열리는 상황을 대처하기 위해 table들을 관리합니다. transaction manager는 여러 transaction들을 관리하는데 이는 후술할 concurrency control을 위해 transaction이 획득한 lock들, 획득하고 하는 lock들을 관리해주기 위함과 recovery 시 transaction 별로 log를 관리하기 위해서 입니다.

buffer management

해당 layer는 buffer를 관리해줍니다. 이 layer는 필수적으로 필요한 layer는 아니나 성능향상을 위해 거의 필수적으로 구현되어야 하는 layer 입니다. buffer는 cost가 높은 file I/O operation의 수를 줄여줌으로써 전체적인 성능향상에 도움을 줍니다. 구체적으로 상위 layer에서 필요한 page가 있을 때 당장 disk로 접근해 해당 page를 읽어오거나 disk에 page를 저장할 때 당장 disk에 접근해 저장하는 것이 아닌 buffer를 통해 읽기&쓰기 작업을 진행하게 됩니다. (caching for reading, buffering for writing)

따라서 상위 layer에선 page에 작업을 할 때 우선 buffer에 있는 page인지 확인을 하고 read&write 작업을 진행하게 됩니다. 만약 buffer에 원하는 page가 없을 시 disk에서 buffer에 page를 불러오는 작업을 하게 됩니다. 하지만 buffer 역시 제한된 크기를 가지기 때문에 buffer가 가득 찼을 시 어떤 조치를 취해줘야만 합니다. 간단하게 생각해 buffer가 가득 찼을 시 buffer에 올라와 있던 page 중 dirty page는 disk에 반영해주고 buffer를 빈 상태로 만들어주는 방식을 채택할 수도 있지만 만약 buffer에 올라와 있던 page들 중 이후에 다시 사용할 page (다시 buffer에 올라와야 하는 page)가 있다면 이런 방식은 비효율적일 것입니다. 따라서 buffer 내의 page 중 victim을 선정해 해당 page가 dirty page면 disk에 반영해주고 (dirty page가 아니면 disk에 반영해줄 필요가 없다.) buffer에서 evict 하는 일이 수행됩니다. victim을 선정하는 방식으로는 FIFO (first in first out) 등 여러 방식이 있을테지만 앞서 이야기 했듯 내려갔던 page들 중 다시 buffer로 올라오는 page가 있을 경우 비효율이 발생하니 앞으로 사용하지 않을 page를 buffer로 내리는 방식이 가장 효율적일 것입니다. 하지만 앞으로 사용하지 않는 page가 무엇인지 알아내는 것은 불가능에 가깝기에 workload의 특성에 따라 LRU (least recently used), MRU (most recently used), LFU (least frequently used), MFU (most frequently used) 등과 같은 방식을 page replacement policy로 정해 victim을 선정합니다.

우리의 project에서도 역시 성능향상을 위해 buffer를 구현하였습니다. init_db 함수의 인자로 받은 num_buf의 수 만큼 buffer의 block을 초기화 한 후 사용하게 됩니다. 상위 계층에서 기존의 file에 직접 쓰는 함수를 호출 하는 것이 아닌 buff_read_page, buff_write_page 함수를 통해 buffer에 올라와있는 page에 read&write 작업을 진행합니다. read&write 작업 중 buffer에 해당 page가 없다면 buffer로 page를 불러오는 작업을 해주어야 합니다. 이때 buffer에 빈 공간이 없다면 page를 evict 시키고 load 하게 되는데 page replacement policy로는 LRU를 사용하게 됩니다. LRU 정책은 최근에 사용된 page가 다시 사용될 확률이 높다고 판단하여 최근에 사용되지 않은 page를 victim으로 선정하는 정책입니다. 따라서 LRU list를 만들어 page들을 관리합니다. 제 구현에서는 buffer들의 block을 linked list로 만들어 page를

사용한다면 list에서 pop 해 list의 head에 넣어주는 방식으로 구현되어 있습니다. eviction 되는 page가 diry page인지 확인하기 위해 dirty page bit를 page의 header에 넣어주어 해당 page에 write 작업이 일어나면 bit를 on 시켜 주었습니다. 따라서 page가 eviction 될 때 dirty page bit가 on 되어 있으면 disk에 변경사항을 반영해주는 일을 해주게 됩니다.

disk space management

해당 layer에선 disk에 실제로 접근해 page들을 읽어 오거나 쓰는 작업을 하게 됩니다. 즉 상위 layer에서 table_id, page number 등을 받아 disk의 어느 file에 몇번째 offset부터 얼마만큼 쓰는 작업을 하게 됩니다. 다시 말해 logical한 위치를 physical한 위치로 변경하는 일을 합니다. 이로인해 logical한 위치와 physical한 위치가 분리되어 상위 레이어에서는 logical한 위치만으로 작업할 수 있게 됩니다. 또한 field들의 type을 보고 record를 byte로 어떻게 표현할 것인지 byte representation을 정하고 data를 page에 어떻게 저장할 것인지 layout 룰을 정합니다.

disk space management를 구현할 때 운영체제를 건너뛰고 직접 disk의 track, sector를 지정해서 읽고 쓰는 방법과 운영체제의 file system을 이용해 disk에 읽고 쓰는 방법이 있습니다. 운영체제를 거치게 된다면 안전하게 읽고 쓸 수 있지만 운영체제를 건너뛰고 직접 읽고 쓰는 방법보다는 실행시간 측면에서 아쉬움이 있습니다. 앞서 기술했듯 disk space management에서 logical한 위치를 physical한 위치로 변경해주어야 합니다. 따라서 file 안에 page들을 어떤 방식으로 저장할 것인지에 대해 여러 방법들이 있습니다. 예를 들어 page를 순차적으로 저장하다 일정 시간이나 조건이 되었을 때 file 내의 page들을 compact 시켜주는 일을 하는 방법이 있을 수 있고 file에 meta data를 저장한 header나 footer를 두거나 mapping table을 두어 관리 하는 방법이 있을 수 있습니다. schema의 field들의 type을 보고 record를 어떻게 만들지 결정하는 byte representation을 정하는 것도 중요한 입니다. field들이 고정적인 크기를 갖는 type들이라면 크기를 정해주어 넣어주면 되지만 가변적인 크기를 갖는 type이라면 여러 방법으로 구현 할 수 있습니다. 가변 할 수 있는 최대 크기를 정해주거나 전체 record를 받고 나서 구분자를 이용해 각 field들을 구별하는 방식이 있습니다. byte representation을 정하고 나서 page layout룰을 정하게 되는데 page layout은 record의 길이(record의 길이가 고정적인가 가변적인가)와 page packing (page 안의 hole을 그대로 둘 것인가 아니면 packing 해서 없앨 것인가) 방법에 영향을 받습니다.

우리의 프로젝트에선 record는 key와 value로 이루어져있고 key는 8 byte

integer형, value는 120 byte의 문자열 배열로 이루어져 있으므로 record는 고정적인 크기를 갖습니다. page layout은 page 마다 header를 갖고 internal page인지 leaf page인지에 따라 key와 자식 page를 가르키는 pointer를 가질지 key와 value를 가질지로 나뉩니다. 또한 page 내에 record가 들어갈 땐 B+tree에 따라 정렬된 형태로 빈틈없이 들어가기 때문에 hole이 생기지 않고 packing이 필요하지 않습니다. page들은 file(table) 내에 순차적으로 저장됩니다. 따라서 page number를 알면 file 내에서 page number x PAGE_SIZE로 인해 page의 해당 위치를 찾을 수 있게 됩니다. 사용하던 page가 free page가 된 경우 별도의 compact 작업을 하지 않고 table의 header page가 관리하는 free page list의 맨 앞에 넣어주는 작업만 하게 됩니다. file 자체적으로도 header를 가지고 있는데 이 header는 file 내에 B+tree의 root page number, 다음에 사용할 free page number, 현재 file에 존재하는 page의 수를 저장해 root page와 다음에 사용할 free page를 쉽게 찾고 file에 free page가 없다면 file을 확장 시켜 free page를 만들어주기 위해 사용하고 있습니다. 상위 layer는 file_read_page, file_write_page 함수를 이용해 file에 직접 써주는 일을 할 수 있는데 상위 레이어에서는 table_id와 page number만 주면 이 layer에서 실제 file에 접근하고 offset을 계산해 read&write 작업을 수행하게 됩니다.

lock manager Atomicity, Durability

lock manager는 concurrency control을 해서 DBMS가 지켜줘야 하는 ACID의 성질 중 Consistency 와 Isolation을 지켜주기 위해 구현됩니다. lock manager라는 단어는 lock을 관리하는 manager라는 뜻이고 우리의 project의 구현에서는 실제로 그렇게 사용되었지만 여기서는 concurrency control을 관리하는 manager라는 큰 의미로 보고 설명해보겠습니다. 일단 Transaction (이하 trx)에 대한 이해가 필요한데 trx란 사용자가 여러 query들을 조합해 1개의 연산단위로 만들어 절대 끊어져서는 안되는 일의 단위를 의미합니다. (ex) trx_begin -> db_find -> db_update -> db_insert .... -> trx_commit) Consistency는 무결성 속성으로 trx의 실행이 성공적으로 완료되면 DB는 언제나 일관성 있는 상태를 유지한다는 것입니다. Isolation 독립성 속성으로 실제로는 여러 trx가 실행되더라도 각자의 trx는 혼자 실행되고 있다는 것처럼 만들어주는 즉 trx를 수행 시 다른 trx가 연산 작업에 끼어들지 못하도록 하는 것을 의미합니다. trx들이 동시에 DB에 접근해 operation을 수행 할 경우 Lost update problem, dirty read problem, inconsistent read problem 등이 발생 할 수 있습니다. 따라서 1개의 DB에 이런 여러개의 trx가 들어와도 성능을

최소한으로 낮추고 이 trx들의 연산이 올바른 결과를 얻기 위해서 Concurrency control이 필요합니다.

가장 간단한 concurrency control 방법은 여러개의 trx들을 줄 세워 순차적으로 실행하는 것입니다. 이러면 consistency와 Isolation을 보장해 줄 순 있지만 성능 측면에서 매우 나쁘기 때문에 실제로는 사용하지 않습니다. 가장 좋은 방법은 같은 data에 접근하지 않는 trx들을 모아 같이 실행하는 것이겠지만 모든 trx들의 정보를 알고 있어야 하는데 실질적으로 그럴 방법이 없기 때문에 trx들을 스케쥴링 하는 방법을 택합니다. 마치 trx들이 1번에 1개씩, 순차적으로 실행되는 것처럼 스케쥴링을 해주어야 합니다. 즉 serial execution과 동일하게 실행 시켜주어야 하는 것을 의미합니다. 문제가 되는 상황은 여러개의 trx가 1개의 record에 접근하는데 여러 trx의 다양한 연산 중 적어도 1개 이상이 해당 record에 write 연산을 할 때 입니다. 따라서 여러 trx 중 write operation이 1개라도 수행되기 전에 들어오는 read는 모두 동시에 처리해도 전혀 문제가 없음을 뜻합니다. 그러다 write operation이 들어오면 그전의 read operation들을 기다리고 1개의 write operation을 진행한 이후 여러 trx들의 나머지 연산을 진행해야 합니다. 이러한 방법을 취하더라도 문제는 아직 남아있습니다. 바로 deadlock을 해결해야 하는데 deadlock이란 trx들이 서로가 서로를 기다리고 있는 상황이여서 어느 trx도 연산을 하지 못하고 기다리고 있는 상황입니다. 따라서 concurrency control은 다시 2가지 방법으로 나뉘게 되는데 pessimistic concurrency control(PCC) 과 optimistic concurrency control(OCC) 입니다. PCC는 다수의 trx가 동시에 같은 data에 접근할 경우가 많다고 보는 관점이고 앞서 설명했듯이 lock을 잡는 방식을 취합니다. OCC는 다수의 trx가 동시에 같은 data에 접근할 경우가 적다고 보는 관점입니다. 따라서 PCC의 경우 data에 접근하기 전에 lock을 걸고 접근하고 OCC의 경우에는 일단 data에 접근하고 나서 conflict가 낫는지를 확인합니다. PCC는 보통 획득한 lock의 갯수가 증가하다가 한번 unlock을 하고 나면 lock의 획득 갯수가 계속 줄어드는 방향으로 가는 2PL 정책을 따릅니다. lock을 획득하며 deadlock을 확인하기 때문에 deadlock 상황이면 acquire phase의 최고점까지 도달하지 못할 것입니다. 따라서 conflict serializability는 보장하지만 2PL 정책은 trx이 abort 하기 전에 unlock을 수행하기 때문에 해당 trx가 abort 되면 이후 연관 있는 trx 전부가 abort 되어야 하는 cascade abort가 일어납니다. 따라서 unlock을 commit이나 abort 이후에 하는 strict 2 phase locking (S-2PL)을 이용하는 것이 일반적입니다. PCC에선 lock table과 wait for graph를 이용해서 deadlock을 detection 하고 deadlock 발생시 trx의 priority에 따라 wait-die(priority가 낮은 trx를 abort), wound-die(priority가 높은 trx를 abort) 정책을 사용하여 trx를 abort 시킵니다. 또한 lock을 잡는 대상에 대해서도 고민해볼수 있습니다. locking granularity는 DB의 어느 수준으로

lock을 잡을지에 대해 결정합니다. DB의 작은 단위에 lock을 거는 것을 fine granularity라 부르며 이 방식은 concurrency는 높아지지만 locking overhead 역시 높아진다는 단점이 있습니다. 큰 것에 대해 lock을 잡는 coarse granularity는 concurrecnty는 낮아지지만 locking overhead도 낮아진다는 특징이 있습니다. 따라서 실제 DBMS는 multiple locking granularity를 지원하고 intent 기법 (작은 것에 대해 lock을 잡으려면 그보다 윗단계의 모든 lock을 먼저 잡아야 한다)을 사용하고 있습니다. 하지만 무작정 모든 lock을 잡고 있는 것은 비효율적이니 lock compatibility matrix에 의해 허용되는 lock들을 종류별로 관리하고 있습니다. 마지막으로 phantom problem을 조심해야 합니다. index 전체에 lock을 거는 것은 비효율적이기 때문에 더 작은 단위의 lock을 설정하게 됩니다. 이러한 경우 Scan 연산을 수행하는 SFW존재하고 이 SFW의 where 절이 똑같은 또다른 SFW의 연산이 insert일 경우 Scan 연산을 수행한 SFW가 뒤이어 실행된 SFW의 insertion 결과를 가져오지 못해 올바른 결과를 도출해내지 못하는 상황이 phantom problem입니다. 이런 phantom problem을 해결하기 위해 index 전체에 locking을 하는 방법과 predicate locking을 하는 방법이 있겠지만 전자는 concurrency가 너무 떨어지고 후자는 overhead가 너무 크기 때문에 일정 구간에 lock을 걸어두는 next key locking 을 사용합니다. OCC는 PCC가 갖는 'lock을 management의 overhead가 크다', 'daedlock detection & resolution이 필요하다', '빈번하게 사용되는 object에 대해 lock contention이 발생한다' 등의 문제를 해결하기 위해 등장하였습니다. 따라서 conflict가 잘 일어나지 않는다고 가정한 후 3단계의 phase를 거쳐 실행됩니다. read phase는 DB에 직접 작업하는 것이 아닌 private workspace에서 작업하는 단계입니다. validation phase는 작업한 내용을 commit 해도 되는지 체크하는 단계입니다. 이때 validation을 누구와 하냐에 따라 Backward oriented OCC와 Forward oriented OCC로 나뉘는데 BOCC는 본인보다 이전에 끝난 trx의 write set과 현재 read set을 비교하는 것이고 FOCC는 아직 commit 되지 않은 trx의 read set과 현재 commit 하는 trx의 write set을 비교하는 기법입니다. 이후 작업한 내용을 DB에 반영하는 write phase가 있습니다. PCC는 conflict가 많이 일어나는 환경에서 안정적이고 OCC는 conflict가 많이 일어나지 않는 환경에서 안정적입니다. serializable isolation에서 isolation을 조금 낮추고 performance를 증가시키기 위해 Multiversion concurrency control(MVCC) 기법을 사용하기도 합니다. MVCC기법의 대표적인 예로는 snapshot isolation이 있습니다. snapshot isolation은 trx가 시작한 기점으로 DB가 마치 freeze 된 것처럼 보이게 해주는 기법인데 이를 위해 1개의 record에 대해 multiversion으로 관리하게 됩니다. 이로 인해 read operation은 해당 record가 update가 되건 말건 원하는 버전을 읽어오면 되니까 성능이 매우 향상됩니다. write operation의 경우 first-

committer-wins rule 등으로 관리해 안전하게 보호해줍니다. snapshot isolation은 이런 장점 외에도 inconsistent read나 phantom이 발생하지 않는다는 이점이 있지만 더이상 serializable 하지 않게 되고 write-skew problem (각 trx이 update 하는 대상이 달라 동시에 실행되었는데 serial하게 실행되었을 때와 결과가 다름) 이 발생할 수 있습니다. 따라서 snapshot isolation을 동작 시킬 땐 write-skew problem이 발생하는 것을 막기 위해 serializable이 필요한지 확인해야 합니다.

각 layer와의 상호 관계와 우리 프로젝트에서 concurrency control의 구현은 concurrency control implementation 장에서 자세히 설명하도록 하겠습니다.

logging & recovery

logging & recovery는 DBMS가 지켜줘야 하는 ACID의 성질 중 Atomicity와 Durability를 지켜주기 위해 구현됩니다. Atomicity는 trx이 처음부터 끝까지 완전히 실행되었거나 그렇지 않다면 아예 실행되지 않은 결과를 DB가 반영하고 있어야함을 뜻합니다. Durability는 trx이 끝나면 어떠한 failure가 생기든 trx의 결과가 DB에서 영속적이여야함을 뜻합니다. 이를 위해 trx의 operation들은 log file에 기록되고 (logging) 어떠한 failure가 생겼을 경우 해당 log를 읽어들이며 DB를 복구하게 됩니다. (recovery)

Atomicity와 Durability를 만족 시켜주는 가장 쉬운 방법은 trx이 끝날 때까지 page를 buffer에서 disk로 안내리고 (no steal), trx가 끝날 때마다 page 들을 disk로 내려주면 (force) 됩니다. no steal policy로 인해 결과를 돌려주는 undo logging이 필요가 없어지고 force 정책으로 인해 operation을 다시 실행시켜주는 redo logging이 필요가 없어집니다. 하지만 no steal, force policy를 선택할 경우 엄청난 성능 저하가 일어나기 때문에 일반적으로는 steal, no force policy를 채택하고 undo & redo logging을 하게 됩니다. logging을 하는 방식은 첫째로 log record들은 순차적으로 작성되어야 합니다. 따라서 log record들은 LSN(log sequence number)를 가지고 있습니다. 또한 logging은 WAL (write ahead logging) 정책을 따르는게 일반적인데 WAL 정책은 buffer에서 page가 disk로 내려가 변경사항을 저장하기 이전에 log buffer의 log record를 log file에 먼저 내려 변경사항을 저장하는 것과 page에 변경사항이 생겼을 때 page를 buffer에 내리는 것이 아닌 log에 모든 것을 기록해두는 방식입니다. 전자의 방식의 결과로 Atomicity를 보장해줄수 있고 후자의 방식의 결과로 인해 Durability를 보장해줄 수 있습니다. 결과적으로 우리는 trx이 commit 할때 log buffer를 flush 하고 page가 flush 되기 전에 log buffer를 flush 하는 것으로 구현하게 됩니다. log의 종류로는 physical log와

logical log가 있는데 physical log는 변경사항 전체를 저장하는 log이고 logical log는 변경사항을 유발하는 action이나 operation을 적는 방식입니다. logical log를 사용할 경우 log 자체의 크기는 줄어들 수 있으나 recovery 시 해당 operation의 반대를 수행해주어야 하므로 operation의 overhead가 recovery의 cost에 추가됩니다. log file이 커지는 것을 방지하기 위해 check pointing을 둘 수 있습니다. check pointing 기법은 dirty page가 오래 되었으면 log가 길어질 확률이 높으므로 (해당 dirty page에 변경이 계속 일어났다면) dirty page를 주기적으로 stable db에 반영해줌으로서 기존에 적어둔 log record가 필요 없게 되고 log file의 크기를 줄이는 기법입니다. log file에 대한 설명은 여기까지 하고 본격적으로 recovery를 하는 과정에 대해 설명하겠습니다. recovery는 3-phase로 일어나는데 analysis phase, redo phase, undo phase가 있습니다. analysis phase에선 trx를 winner와 loser로 구분하는데 winner trx는 제대로 종료된 trx를 의미하고 loser trx는 DB crash로 인해 정상적으로 종료하지 못한 trx를 의미합니다. 이후 진행되는 redo, undo phase는 recovery 정책에 따라 다른 양상을 띕니다. redo-winner 정책에선 winner trx에 대해서만 옛날 log부터 최신 log까지 다시 실행하는 redo phase가 진행되고 loser trx에 대해서 최신 log부터 옛날 log까지 실행하는 undo phase가 진행됩니다. 만약 recovery 중 page에 저장된 LSN이 log보다 크다면 log보다 최신의 변경을 담고 있다는 것을 뜻하니 consider redo로 취급하고 다음 redo로 넘어 갈 수 있습니다. 비슷하게 page의 LSN이 log보다 작다면 log의 변경보다 이미 구식의 data를 담고 있다는 뜻이니 consider undo로 취급하고 다음 undo로 넘어갈 수 있습니다. 또한 주의해야 할 점이 abort가 완료 된 trx도 winner로 판단해야 한다는 점입니다. trx이 abort 되며 기존 log로 인해 변경된 결과를 되돌리는 clr (compensation log record)를 발급하고 winner로 처리해야 올바른 recovery 결과를 얻을 수 있습니다. redo-history 정책에선 winner와 loser 상관없이 모두 redo를 하고 loser trx에 대해 undo를 진행하는 것입니다. 따라서 redo phase가 끝나면 해당 DB는 마지막으로 crash가 발생 했을 때의 모습이 됩니다. undo를 진행하면 마찬가지로 clr이 발급되게 되는데 잦은 crash로 인해 clr의 clr의 clr이 발급되는 등 불필요한 log가 늘어나 log file이 커지는 것 뿐만 아니라 redo, undo에 걸리는 시간이 늘어나는 현상이 발생할 수 있습니다. 따라서 next undo sequence number를 이용해 해당 clr의 undo point의 이전을 가르켜 다음 clr이 undo 할 위치를 저장해주는 일이 필요합니다. 이 방식으로 인해 undo 중 crash가 자주 일어나도 undo 해야 할 양은 계속 줄어들게 되어 성능향상이 됩니다.

각 layer와의 상호 관계와 우리 프로젝트에서 logging & recovery의 구현은 Crash-Recovery Implementation장에서 자세히 설명하도록 하겠습니다.

Concurrency Control Implementation

(Concurrency Control 기법이 어떻게, 어느 레이어에 걸쳐져서 구현 되었는지에 대해 세부적으로 기술, 관련 있는 layer 의 핵심 component 에 대한 설 명)

우선 우리의 project5에서 구현한 concurrency control부터 설명하겠습니다. project5에선 insert, delete operation은 발생하지 않고 오직 find와 update operation만 발생한다고 가정하고 있습니다. 따라서 project5에선 buffer의 lock을 잡고 page의 lock을 잡은 뒤 buffer의 lock을 풀고 page의 lock을 풀고 내가 원하는 record를 찾은 뒤 trx manager나 lock manage 등의 lock을 잡고 풀면서 record lock에 접근한 뒤 기다리지 않아도 되는 상황이면 lock을 획득한 후 동작을 하고 기다려야 하는 상황이면 sleep을 한 뒤 다른 trx가 commit되거나 abort되어 내가 lock을 획득 할 수 있으면 비로서 일어나서 동작하는 방식이였습니다. 하지만 이는 일반적인 concurrency control이 아니였기에 project6에서는 새로운 방식으로 lock을 잡게 됩니다. 여전히 find와 update operation만 일어난다는 가정은 동일합니다. project6에선 buffer의 lock을 잡고 page의 lock을 잡은 뒤 lock manager에 lock을 잡습니다. 이후 record lock을 잡을 수 있는지 확인합니다. record lock을 잡을 수 있다면 page와 lock manager의 lock을 풀고 operation을 계속 진행하면 되고 만약 기다려야 하는 상황이라면 trx table의 lock을 획득하고 현재 내 trx의 lock을 획득하고 trx manager의 lock을 해제한 후 page lock과 lock manager를 반납하고 자신의 trx lock을 반납하는 동시에 잠에 듭니다. 그 후 다른 trx가 깨워주면 다시 자신의 trx lock을 획득하고 이 trx lock을 반납한 후 page에 lock을 다시 획득하려고 시도한 뒤 획득하면 operation을 진행하는 형태입니다. 이제 project6에서 최종적으로 구현한 제 concurrency control에 대해 설명하겠습니다.

먼저 최상위 계층에서 lock manager와 trx manager, 그리고 deadlock을 처리하기 위한 wait for graph manager 등을 구현하였습니다. lock manager는 record lock들을 관리합니다. lock manager는 일단 record의 table_id와 key를 받아 hash 값을 계산합니다. 제 구현에서 hash 값은 table_id와 key 값을 concatenate 시키고 기존에 macro로 지정해두었던 HASH_SIZE 으로 나눠서 나온 나머지 값을 취합니다. (ex) talbe_id == 1, key == 0, HASH_SIZE == 500 ==> HASH_KEY == 10) HASH_SIZE가 무한하지 않고 table_id와 key를 concatenate 한 값이 같으면 (ex) table_id == 11, key == 0 과 table_id == 1, key == 10) hash collision이 일어나므로 hash collision에 대한 대처를 해주어야 했습니다. 제 구현에선 collision이 날 경우 같은 hash_key를 갖는 entry를 linked list로 묶어 주었습니다. 이후 이 entry들은 해당 record에 들어온 trx의 lock들을 linked list로 연결해 관리하게 됩니다. trx manager는 trx들이

가지고 있는 lock을 관리하기 위해 만들어주었습니다. trx manager 안에서 trx들은 linked list로 연결되어 있기에 trx manager를 순차적으로 탐색하면 내가 원하는 trx를 찾을 수 있습니다. trx 각각은 자신이 요청한 lock을 순서대로 linked list로 관리하고 있습니다. deadlock을 처리하기 위한 wait for graph는 linked list로 구현한 directed graph 입니다. wait for graph의 node는 어떤 trx가 특정한 record에 대해 lock을 얻으려고 시도할 때 자신이 기다려야 하는 trx가 있다면 이 두 trx에 대해 node를 만들어 줍니다. (이때 graph에 이미 해당 trx의 node가 존재한다면 만들지 않습니다.) 그러고 나서 기다리는쪽에서 나와 기다림을 받는쪽으로 들어가는 단방향 edge를 그어준 후 현재 기다려야하는 trx의 node에서 BFS 알고리즘을 통해 자기 자신으로 돌아오는 cycle이 있나 확인을 합니다. (자기 자신에서 시작해서 완전탐색을 하는데 어떤 node에서 처음 시작한 node로 돌아올 수 있으면 cycle이 있다고 판단) 만약 cycle이 존재한다면 deadlock 상황으로 간주하고 적절한 return 값을 주어 trx를 abort 시킬 수 있도록 하였습니다.

기존 project 5에선 buffer layer의 buffer_read, buffer_write 함수 안에서 buffer의 lock을 걸고 page의 lock을 잡고 buffer의 lock을 풀고 page의 내용을 읽거나 쓴 후 page의 lock을 풀어주는 동작이 일어났지만 general lock에선 page lock을 획득한 채로 record lock을 얻을 수 있는지 확인해야 했기에 기존의 함수들을 바꿔주어야 했습니다. buffer의 lock을 잡아 LRU list 보호하고 찾고자 하는 page가 다른 trx에 의해 evict 되는 현상을 보호해주는 과정까지는 같습니다. 하지만 buff_read (page lock을 잡고 page lock을 안풀고 나오는 함수 , page lock을 안잡고 page lock을 안풀고 나오는 함수), buff_write (page lock을 안잡고 page lock을 풀고 나오는 함수)가 세분화 되었고 해당 page의 lock을 잡고 풀어주는 buff_get_page_latch, buff_release_page_latch 함수가 추가되었습니다. 이는 이후 설명할 lock_acquire 함수 (record lock을 획득하려고 시도하는 함수)를 실행하기 전에 page lock을 가지고 있어야하며 lock_acquire의 과정에서 page lock을 풀어주기도 하고 lock_acquire의 결과로 인해 page lock을 다시 획득해야 하는 경우가 생기기 때문입니다.

이제 lock_acquire 함수에 대해 설명해보겠습니다. lock_acquire 함수는 최상위 계층의 db_find와 db_update에서 사용됩니다. lock_acquire 함수는 인자로 table_id, key 등등을 받고 int형을 return 합니다. return형이 int인 이유는 다양한 상황을 함수를 부른 caller쪽에 알리기 위함입니다. lock_acquire 함수는 lock_table에 해당 entry가 있는지 확인하고 없다면 만들어줍니다. 이후 이 entry에 현재 trx의 lock을 매달고 lock을 획득할 수 있는 상황인지, sleep 해서 기다려야 하는 상황인지, deadlock 상황인지를 체크합니다.

s lock이 들어온 상황에서는 entry의 head가 null이라면 lock을 획득할 수 있습니다. 또한 entry의 head가 현재 trx의 lock이고 x lock이라면 역시 획득할 수 있습니다. 두 경우가 아니라면 현재 매달린 lock의 앞쪽을 순차적으로 entry의 head까지 탐색합니다. 그 과정에서 x lock이 1개라도 있으면 sleep 해서 기다려야 하는 상황인데 sleep 상황을 return 하기 전에 deadlock detection을 합니다. 이후 deadlock이라면 deadlock 상황을 알리는 값을 return 하고 그렇지 않다면 sleep 상황을 return 합니다. 만약 앞에 x lock이 1개도 없다면 shared lock은 여러개가 동시에 record에 접근할 수 있으니 lock을 획득할수 있는 상황이라고 판단하고 해당 상황값을 return 합니다.

x lock이 들어온 상황에서는 역시 entry의 head가 null이라면 lock을 획득할 수 있습니다. 또한 entry의 head가 현재 trx의 lock이고 x lock이라면 마찬가지로 lock을 획득할 수있습니다. 다른 경우로는 entry에 lock이 1개밖에 없고 그 lock이 현재 trx의 s lock이라면 그 s lock을 x lock으로 바꿔준 후 lock을 획득할 수 있습니다. 세가지 경우가 아니라면 x lock은 무조건 기다려야 하므로 deadlock detection을 실행하고 deadlock이면 deadlock 상황을 그렇지 않다면 sleep 상황을 caller에게 알리고 종료합니다.

다시 db_find로 돌아와서 lock_acquire의 결과가 non sleep 이면 page의 record를 읽고 page의 lock을 해제한 후 종료합니다. 결과가 sleep 이면 lock_wait 함수를 호출해 앞서 기술한 general lock 순서로 lock을 반납하고 sleep 합니다. 이후 깨어나게 되면 buffer에 원하는 page가 있는지 확인하고 없다면 disk에서 불러온 후 page lock을 다시 잡고 원하는 operation을 한 뒤 page lock을 풀고 종료합니다. 결과가 deadlock 이면 page의 lock을 풀고 trx_abort를 진행한 뒤 종료합니다.

db_update에서도 비슷한 일이 벌어집니다. lock_acquire의 결과가 non sleep 이면 page에 수정사항을 적고 page lock을 해제한 후 종료합니다. 결과가 sleep 이면 lock_wait 함수를 호출해 앞서 기술한 general lock 순서로 lock을 반납하고 sleep 합니다. 이후 깨어나게 되면 buffer에 원하는 page가 있는지 확인하고 없다면 disk에서 불러온 후 page lock을 다시 잡고 원하는 operation을 한 뒤 page lock을 풀고 종료합니다. 결과가 deadlock 이면 page의 lock을 풀고 trx_abort를 진행한 뒤 종료합니다.

잠든 trx는 해당하는 lock을 잡고 있던 trx가 commit 되거나 abort 될 때 lock_release 함수가 호출되며 깨어납니다. commit 되는 trx는 잡고있던 lock들을 release 해주고 trx manager에서 해당 trx 제거해주면 끝이지만 abort 되는 trx는 현재까지 진행되었던 결과를 roll_back 시켜주는 일이 필요합니다.

따라서 trx가 가지고 있던 마지막 lock부터 거슬러 올라가며 x lock을 잡고 했던 결과들을 전부 roll_back 해주며 lock을 release 해주고 trx manager에서 삭제됩니다. lock_release는 lock manager의 entry에서 해당 lock을 삭제시키며 기다리고 있던 lock을 깨워주는 함수입니다. 만약 release 해주는 lock이 s lock인데 바로 옆에 있는 lock 역시도 s lock이면 자신의 lock 만 제거해주고 끝이 납니다. s lock이 아니라 x lock이 있다면 x lock 1개를 깨워주고 일어납니다. 현재 release 되는 lock이 x lock일때 바로 옆의 lock이 x lock 이면 바로 옆의 lock 만 깨워줍니다. 바로 옆의 lock 이 s lock이라면 다음 x lock이 나오거나 list에 끝에 도달할때까지 존재하는 모든 s lock들을 깨워줍니다.

Crash-Recovery Implementation

(Crash-Recovery 기법이 어떻게, 어느 레이어에 걸쳐져서 구현 되었는지에 대해 세부적으로 기술, 관련 있는 layer 의 핵심 component 에 대한 설명)

우선 우리가 project6의 명세를 살펴보면 3-phase recovery (analysis phase - redo phase - undo phase), consider redo for fast recovery, compensate log record in abort / undo pass, 'undo next sequence number' to make progress despite repeated failure 입니다. 따라서 winner와 loser trx를 나눠주는 analysis phase, winner와 loser trx에 대해 redo 하는 redo phase, loser trx에 대해 undo 해주는 undo phase를 구현해야 하고 page의 LSN이 redo log의 LSN보다 크다면 redo 하지 않고 pass 하는 consider redo를 구현해주어야 하고 abort와 undo pass에서 compensate log record를 적절히 발급해주어야 합니다. 마지막으로 undo next sequence number를 발급해 undo phase에서 crash가 나더라도 undo의 진행상태가 계속 진행되게 해주어야 합니다.

먼저 가장 아래 layer인 disk management layer부터 시작하겠습니다. 일단 log record의 layout을 결정해야 합니다. 하지만 이 부분은 과제의 명세에서 전부 주어졌기에 해당 변수들을 선언하며 layout을 완성했습니다. log record의 LSN은 log record의 시작 offset으로 정하거나 끝 offset으로 정하는 두가지 방법이 있었는데 저는 시작 offset으로 정했습니다. 결과적으로 log record는 BEGIN, COMMIT, ROLLBACK type이라면 28byte, UPDATE type이라면 288byte, COMPENSATE type이라면 296byte를 가지게 됩니다. (C언어의 구조체 패딩에 의해 32, 296, 304byte로 자동 할당 되지만 #pragma pace() 매크로로 28, 288, 296byte로 만들어줄 수 있습니다.) 이후 기존에 우리가 page를 file에 저장할때 필요했던 함수들 중 몇개를 골라 log file에 대해 만들어주었습니다. log_file_open은 log file을 열어주는데 만약 log_file이 이미 존재하면 끝까지 읽고 마지막 log record를 인자로 받은 last_record에 저장시켜 주었습니다. 이 last_record는 buffer layer에서 global_LSN을 저장하기 위해 사용합니다. log를 파일에 쓰기 위해 log_file_write_record 함수를 만들어주었습니다. start_point (log의 LSN), type (log의 type), log record를 받아 start_point와 type을 이용해 offset을 계산한 뒤 log를 파일에 써주었습니다. log file에서 record를 읽어오는 log_read_file_record 함수는 인자로 받은 start_point(읽어오려는 log의 LSN)을 받아 log file에 접근에 해당 log를 읽어와주는 함수입니다. 그리고 기존 page의 layout에 page_LSN을 추가해 해당 page를 마지막으로 접근한 log의 LSN을 적어주었습니다. 이것은 이후에 consider redo에 사용됩니다.

buffer layer에선 log를 위한 log buffer를 구현하였습니다. init_log_buff

함수에서 log buffer를 초기화 해주는데 인자로 받은 num_buff보다 1개 더 많이 초기화 해주어 1개 더 생긴 buffer의 block은 read_only로 사용해주었습니다. 이유는 file에 있는 log가 buffer로 올라와 상위 계층에서 사용되는 일이 존재하는데 buffer가 flush 될 때 해당 log는 file에 적히면 안되기 때문입니다. 또한 buffer를 거치지 않고 상위레이어에서 file의 log를 읽어가는 행위는 layered architecture에 어긋난다고 생각했기 때문입니다. 이후 상위계층에서 log를 발급했을 때 log를 바로 file에 쓰는 것이 아닌 log buffer에 쓰기 위해 log_buff_write 함수를 만들어주었습니다. 또한 log를 읽어와주는 log_buff_search_record 함수도 만들어주었습니다. 여기서 주의해야할 점은 log_buff_search_record는 찾으려는 log를 buffer에서 먼저 찾고 buffer에 없다면 file에서 읽어옵니다. 이렇게 한 이유는 buffer에 남아있는 log는 아직 flush 되기 전이기 때문에 log file에는 해당 log가 존재하지 않기 때문입니다. (read_only block에 있는 log는 file에 존재함, 하지만 한번 쓰여진 log의 내용이 변경되는 일은 없고 log file에서 해당 log를 다시 읽어오는 일은 비효율적이니까 buffer에 있으면 그 log를 사용함) log_buff_flush 함수는 log buffer에 존재하는 모든 log 들을 file에 내려주는 함수입니다. 우리는 WAL 정책을 따르므로 page가 evict 될 때, trx가 commit 될 때, trx가 abort 될 때 log buffer를 flush 해줍니다. 또한 과제의 명세에서 명시했듯이 init_db 함수가 끝나면 log의 buffer가 flush 되어야 합니다. log buffer에는 global_LSN이라는 변수를 가지고 있어 이번에 발급 될 log의 LSN을 저장하고 있습니다.

최상위 계층에선 trx_begin, trx_commit, trx_abort, db_update 함수들이 사용되면서 log들이 발급되고 log buffer에 써지게 됩니다. trx_begin 시 log buffer에 해당 trx의 BEGIN log를 써주고 trx_commit 시 log buffer에 해당 trx의 COMMIT log를 써주고 log buffer를 flush 해줍니다. (앞서 기술 했듯이 log buffer를 flush 하는 경우는 trx가 commit할 때와 page가 evict 될 때, trx이 abort 될 때) trx_abort를 보기 이전에 db_update를 먼저 보면 db_update 수행 중 page의 내용을 변경하기 이전에 UPDATE log를 먼저 발급해주고 page의 내용을 변경해야 합니다. 또한 page에 update가 일어났으므로 page의 page_LSN을 변경하는 작업 역시도 필요합니다. trx_abort를 가장 마지막에 본 이유는 compensate log record를 발급하는 과정이 조금 복잡하기 때문입니다. trx이 abort 됨에 따라 기존에 update 했던 모든 data를 roll back 시켜줘야 합니다. 그리고 이 roll back 과정에서 compensate log record가 발급됩니다. compensate log record는 update log를 거꾸로 돌려주는 log입니다. compensate log record를 발급할 때 과제의 명세를 충족 시키기 위해 compensate log record에 next undo LSN를 적어주어야 합니다. 이 next undo LSN은 현재 compensate log record가 undo 하는 update log의 LSN이 아닌

이번 compensate log record가 undo 하고 다음에 undo 할 update의 LSN을 적어줍니다. 만약 해당 compensate log record가 undo할 마지막 update 였다면 next nudo LSN에 -1 값을 넣어주어 더이상 compensate log record가 발급되지 않게 합니다.

recovery는 최상위 계층에서 진행됩니다. analysis phase, redo phase, undo phase에 필요한 recovery manager를 만들어주었습니다. reocvery manager를 구현하여 log에 존재하는 trx들을 linked list로 엮었습니다. 이 trx들은 analysis phase가 끝나고 나면 winner trx와 loser trx로 나뉘게 됩니다. analysis phase에선 log file을 처음부터 읽습니다. BEGIN log가 나오면 일단 loser trx로 분류하고 이후 COMMIT이나 ROLLBACK type의 log가 나오면 winner로 재분류 해주었습니다. 또한 log를 읽을 때마다 해당하는 trx의 last_LSN을 update 시켜주었는데 이는 나중에 undo phase에서 log file을 전체를 다시 읽어와 undo 할 log를 골라내는 불필요함을 제거하기 위해서 입니다.

그 다음은 redo phase입니다. redo phase에선 winner든 loser든 trx의 모든 log들을 처음부터 순차적으로 redo 해줍니다. redo phase에서 BEGIN, COMMIT, ROLLBACK type의 log들은 그냥 읽고 지나가면 되지만 UPDATE, COMPENSATE type은 page를 읽어오고 data를 update 해주는 과정이 필요합니다. 따라서 우리는 consider redo를 구현해 읽어온 page의 LSN이 log의 LSN보다 크다면 이미 최신 업데이트를 반영하고 있다는 뜻이 되므로 data를 update하지 않고 consider redo로 처리합니다. 이런 redo phase가 모두 끝나면 DB가 crash 되었을 상태로 돌아가게 됩니다.

마지막으로 undo phase입니다. 이전에 analysis phase에서 winner, loser trx를 구별했고 구별과정에서 last_LSN (해당 trx가 발급한 마지막 log의 LSN)을 각 trx마다 저장 했었습니다. 따라서 recovery manager에서 loser trx를 뽑고 각각의 trx의 last_LSN으로 undo_entry를 만들어 줍니다. 이 undo_entry는 항상 정렬된 상태를 유지하는 linked list인데 LSN이 가장 큰 것이 entry의 head에 위치하고 내림차순으로 정렬되어 있습니다. 따라서 우리가 현재 undo 할 것은 이 undo_entry의 head에 위치한 LSN을 갖는 log입니다. (undo 중 compensate log record가 발급된 경우, 이 compensate log record 역시 이 entry에 넣어 실행될 순서를 보장해주었습니다.) head에서 log를 뽑아 진행하며 compensate log record를 발급하고 next undo LSN을 기입해주었습니다. trx의 마지막 log가 update인 경우 compensate log record를 발급하고 next undo LSN을 기입하기 위해 update log의 앞 log들(해당 trx의 LSN인데 이 update log보다 LSN이 작은 log들)을 내림차순적으로 읽어 나갔습니다. 마지막 log가 compensate log record인 경우 compensate log record의 next undo LSN으로 가 compensate log record를

발급하고 next undo LSN을 기입하기 위해 바로 이전의 compensate log record의 next undo LSN에 해당하는 log를 읽어와 해당 log의 앞 log들(해당 trx의 LSN인데 이 update(compensate log record의 undo 대상은 compensate가 아니라 update입니다.)의 log보다 LSN이 작은 log들)을 내림차순적으로 읽어가며 next undo LSN을 발급하였습니다. 두 경우 마찬가지로 발급된 compensate log record가 trx가 실행한 처음 update log였으면 next undo LSN에 -1 값이 들어가 마지막 compensate log record 가 됩니다.

이렇게 구현된 analysis phase, redo phase, undo phase는 init_db 함수 내에서 순차적으로 호출되고 init_db의 flag, log_num 변수에 따라 다양한 상황을 연출할 수 있습니다.

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)

   제 구현에 있어서 operation을 수행하게 되었을때 가장 먼저 잡는 lock은 buffer에 대한 lock입니다. 따라서 buffer에 접근하려는 trx가 많은 경우 뒤에 실행된 trx는 buffer의 lock을 잡기 위해 아무런 동작도 하지 못하고 계속 기다리게 됩니다. 만약 접근하고자 하는 record가 같은 page 내에 있는 trx가 많을 경우 page lock에서도 비슷한 현상이 나타나게 됩니다. 이러한 현상은 뒤의 lock manager 자체에 대한 lock을 잡을때에도 나타나는데 이렇게 병목현상이 겹겹이 겹쳐 성능저하가 심하게 일어날 수 있습니다. 이러한 현상을 해결하기 위해선 intent lock 기법을 사용하고 lock compatibility matrix에 따라 non-conflicting read-only trx은 동시에 실행될 수 있게 해주면 성능저하를 낮출 수 있습니다. 또다른 기법으로 OCC기법이나 MVCC 기법의 snapshot을 이용하는 것입니다. OCC는 애초에 conflict operation이 많이 안발생한다고 가정하기 때문에 non-conflicting read-only trx만 있을 경우 엄청난 성능을 보여줄 것입니다. snapshot의 경우 위에 기술했듯이 write-skew problem이 발생할 수 있으므로 serializable이 필요한 상태인지 확인해야 했기에 overhead가 생길수 있었지만 현재는 non-conflicting read only trx만 존재하는 경우이므로 해당 overhead는 고려하지 않아도 되니 성능 향상을 가져올 수 있습니다.

   두번째론 buffer가 page를 evict 하는 과정에서 큰 성능저하를 불러일으킬 수 있습니다. 제 구현에선 buffer의 page를 evict 시키고 buffer의 hole을 없애기 위해 page를 compact 하는 작업을 진행합니다. 그런데 compact하는 작업을 수행 할 때 buffer의 lock을 계속 잡고 있고 buffer 위의 page들의 lock을 전부 잡고 나서야 compact를 실행하게 구현했기 때문에 page를 eviction 시키는 작업은 overhead가 매우 큽니다. 따라서 non-conflicting read-only trx만 존재하지만 접근하고자 하는 record들이 모두 다른 page들에 들어있다면 page eviction은 자주 일어날 것이고 원래 cost가 큰 file I/O 작업까지 수행하게 되므로 성능저하를 일으키게 됩니다.

해결방법으로는 buffer를 배열로 구현하는 것이 아닌 linked list로 구현하여 page가 evict 되었을 때 compact 작업이 필요하지 않게 만들고 linked list로 구현된 buffer의 크기는 가변적일 수 있다는 점을 이용해 buffer의 크기를 현재 memory 상태를 고려해 최대한 크게 만들어줌으로써 page eviction을 줄이고 file I/O operation이 일어나는 횟수를 낮춰 overhead를 줄이는 것입니다.

세번째론 lock manager가 hash table로 구현되어 있고 hash collision이 일어났을때 entry를 linked list로 계속 엮어주기 때문에 성능저하가 발생할 수 있습니다. 같은 hash key 값(hash key 값은 앞서 설명 했듯이 table_id와 key를 concatenate하고 HASH_SIZE로 나눈 나머지입니다.)을 가지는 record들이 trx들의 operation으로 들어온다면 1개의 hash key에 매달린 entry들이 많아지고 해당 entry를 찾는 탐색시간이 길어지게 됩니다. 제가 생각한 방법은 hash key 마다 일정한 capacity를 두어 해당 hash key의 capacity가 가득 차면 hash key를 구할 때 기존의 계산방식을 쓰지 않고 새로 만든 두번째, 세번째 계산 방식을 이용해 entry들을 hash table에서 분산 시키는 방법입니다. 이렇게 구현하면 전체 hash table에 매달려 있는 entry의 총 갯수는 같지만 key 각각 마다의 entry의 길이는 줄어들어 평균 hash search 시간을 단축 시켜 성능을 향상 시킬 수 있을 것 같습니다.

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)

non-conflicting write-only trx이 동시에 수행될 경우 발생할 수 있는 Crash-Recovery와 관련된 성능 측면에서의 문제점은 log file이 길어져 recovery time이 길어진다는 것이라고 생각합니다. 따라서 log file의 길이를 줄일 수 있는 방법에 대해 서술해보겠습니다.

제 구현에 있어선 read operation은 별도의 log를 쓰지 않는 반면에 write operation은 log를 작성합니다. 따라서 log file을 truncate하는 부분이 없기 때문에 non-conflicting write-only trx의 동작에 의해 update log가 계속 쌓이게 될 것 입니다. 만약 check point를 두고 check point 마다 buffer의 page들을 disk로 flush 시켜주고 필요없는 log record들을 삭제시켜 log file의 길이를 줄이는게 된다면 DB에 Crash가 생겼을 때 file에서 읽어올 log의 양이 적어지고 이는 recovery 하는 양의 감소로 이어지니 crash recovery에서 성능 향상이 있을 것 같습니다.