

Database System 2020-2

Final Report

ITE2038-11800

Table of Contents

Overall Layered Architecture	2.p
1. Overview of layered Architecture	
2. File Manager Layer	
3. Buffer Manager Layer	
4. Index Layer	
Concurrency Control Implementation	12.p
1. Overview of Concurrency Control with layered Architecture	
2. Lock mode	
3. Deadlock detection	
4. Abort and Rollback	
Crash-Recovery Implementation	16.p
1. Overview of Crash-Recovery with layered Architecture	
2. Analysis pass	
3. Redo pass	
4. Undo pass	
In-depth Analysis	20.p
1. Workload with many concurrent non-conflicting read-only transactions	
1) Expected Problems	
2) Solution	
3) Check Problems	
2. Workload with many concurrent non-conflicting write-only transactions	
1) Expected Problems	
2) Solution	

Overall Layered Architecture

1. Overview of Layered Architecture

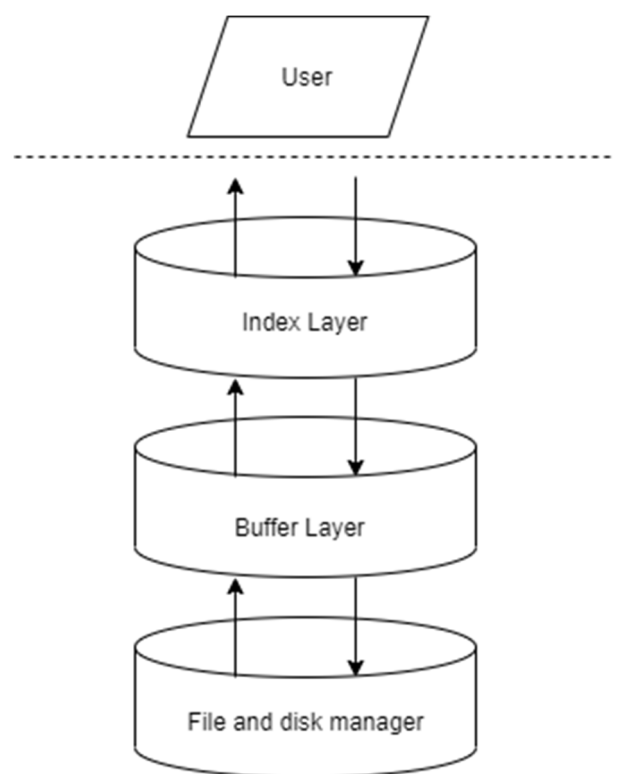
그림 1과 같이 본 데이터 베이스는 3단계로 구성이 되어있습니다.

밑에서부터 살펴보면 File and disk manager는 buffer layer의 요청에 따라서 데이터를 읽고 쓰는 것을 진행합니다.

Buffer layer는 index layer의 요청에 따라서 buffer layer에 있는 정보를 제공하거나 File and disk manager로부터 데이터를 가져와 필요한 정보를 제공합니다.

Index layer는 사용자의 요청에 따라서 빠른 속도로 해당 데이터를 찾으며 이때 사용되는 데이터는 모두 buffer layer로부터 받아서 사용합니다.

다음 페이지부터는 File and disk manager 부터 Index layer의 구체적인 구현에 대해서 알아보겠습니다.



2. File and disk Layer

1). Data Size & Characteristic

본 데이터 베이스는 각 4096 Bytes로 고정된 page 크기를 가지며 record의 크기는 key 8 bytes, value 120 Bytes로 총 128 Bytes의 크기를 가집니다.

또한 record를 찾을 수 있는 정보가 page내부에 저장된 것이 아니라 record 자체가 page내부에 존재하므로 Clustered된 구조를 가지고 있습니다.

2). Type of Referencing Page

1. Header page

Header page는 오른쪽과 같이 현재 데이터베이스의 메타데이터를 가지고 있으며 반드시 데이터 베이스 파일의 가장 앞부분에 존재합니다.

Free Page Number (0~8)
Root Page Number (8~16)
Number of pages (16~24)
(Reserved) (~4096)

2. Free page

Free page는 이전에 할당되었지만 file and disk manger에 의해 해제된 page입니다. 따라서 오직 다음 free page만을 가리킵니다.

Next Free Page Number (0~8)
(Not used) (~4096)

3). File and disk Layer API

```
void file_read_page(int table_id, pagenum_t pagenum, page_t* dest)
```

해당 table_id에 맞는 데이터 파일로부터 해당 pagenum을 가진 페이지를 dest에 저장합니다.

```
void file_write_page(int table_id, pagenum_t pagenum, const page_t* src)
```

해당 table_id에 맞는 데이터 파일로부터 해당 pagenum을 가진 페이지를 찾아서 src로 덮어씹습니다.

`pagenum_t file_alloc_page(int table_id)`

해당 `table_id`로부터 page 하나를 할당하고 그 page의 `pagenum`를 반환합니다

`Void file_free_page(int table_id, pagenum_t pagenum)`

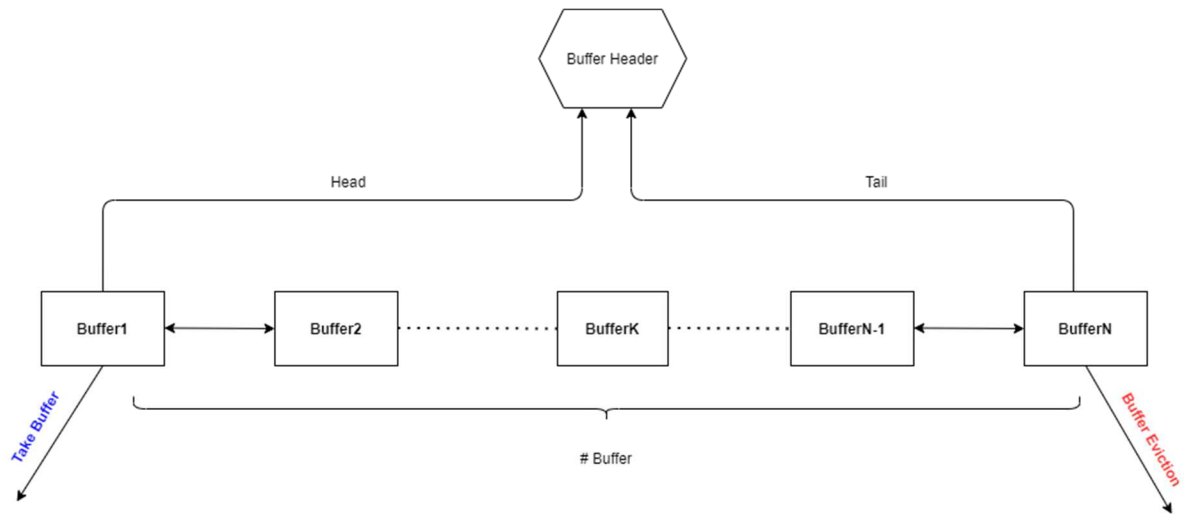
해당 `table_id`에 맞는 데이터 베이스의 `pagenum` page를 free page로 만듭니다.

`void file_initialize(int table_id)`

해당 `table_id`에 맞는 데이터 베이스를 초기화 합니다. (파일의 크기에는 영향을 주지 않습니다.)

3. Buffer Layer

1) Overall Buffer Design



2) Direction of design

본 데이터베이스의 버퍼는 가장 최소의 버퍼 크기에도 작동할 수 있도록 설계된 버퍼입니다. 가장 극한의 경우 table_id당 2개의 buffer만으로 correctness를 보장합니다.

3) Buffer Management Policy

본 데이터베이스의 buffer 관리 정책은 LRU(Least Recently Used)를 따릅니다.

따라서 가장 최근에 사용한 버퍼 즉, 현재 연산 중인 버퍼는 buffer의 제일 앞에 위치하며 안 쓰일수록 뒤로 가서 맨 뒤에 도달하면 eviction의 대상이 됩니다. 이때 eviction을 확인하는 방법은 다음과 같습니다.

1. 맨 뒤 페이지를 eviction 타겟으로 잡는다.
2. 현재 해당 buffer가 사용되고 있다면 이전 페이지를 확인한다.
3. 현재 해당 buffer가 사용되고 있지 않다면 해당 페이지를 eviction 시킨다.

본 페이지의 '사용'의 개념은 병렬제어와 관련 있으므로 Concurrency Control 목차를 참조해주시요.

4) Overall Performance

Notation: N(버퍼 수), W(page 쓰기 I/O), R(page 읽기 I/O)

Page가 buffer 내부에 존재할 때: $O(N)$

Page가 buffer 내부에 존재하지 않을 때: $O(N+2R+W)$

5) Buffer manager API

int buf_read_page(int table_id, pagenum_t pagenum, page_t* dest)

해당 table_id와 pagenum을 key로 하여 해당 page가 존재하는 buffer에서 page를 읽어 dest에 저장한다. (성공:0, 실패: -1)

int buf_write_page(int table_id, pagenum_t pagenum, const page_t* src)

해당 table_id와 pagenum을 key로 하여 해당 page가 존재하는 buffer에서 page를 읽어 src로 덮어쓴다. (성공:0, 실패: -1)

Int64_t buf_read_root_page(int table_id, page_t* dest)

해당 table_id의 root_page를 buffer로부터 읽어서 dest에 저장한다.
(성공: root_page_num, 실패:0)

Int64_t buf_write_root_page(int table_id, pagenum_t pagenum, const page_t* src)

해당 table_id와 pagenum을 key로 하여 해당 페이지를 읽고 src로 덮어쓴다. 그 후 header page의 수정으로 현재 페이지가 root page가 되도록한다.

(성공:0, 실패: -1)

Void buf_set_root_page_num(int table_id, pagenum_t pagenum)

해당 table_id의 root_page를 pagenum으로 수정한다

Pagenum_t buf_set_root_page_num(int table_id)

해당 table_id의 root_page를 반환한다. (성공: root_page_num, 실패: 0)

Void buf_free_page(int table_id, pagenum_t pagenum)

해당 table_id의 pagenum page를 해제한다. 이때 동기화를 위해서 buffer뿐만 아니라 disk도 정보를 즉시반영한다.

Pagenum_t buf_alloc_page(int table_id)

해당 table_id의 free page하나를 할당한다. 이때 동기화를 위해서 buffer뿐만 아니라 disk도 정보를 즉시반영한다 (성공: pagenum, 실패: 0)

Int buf_close_table(int table_id)

해당 table_id의 모든 buffer를 flush하고 buffer를 지운다.

Int buf_shutdown_db()

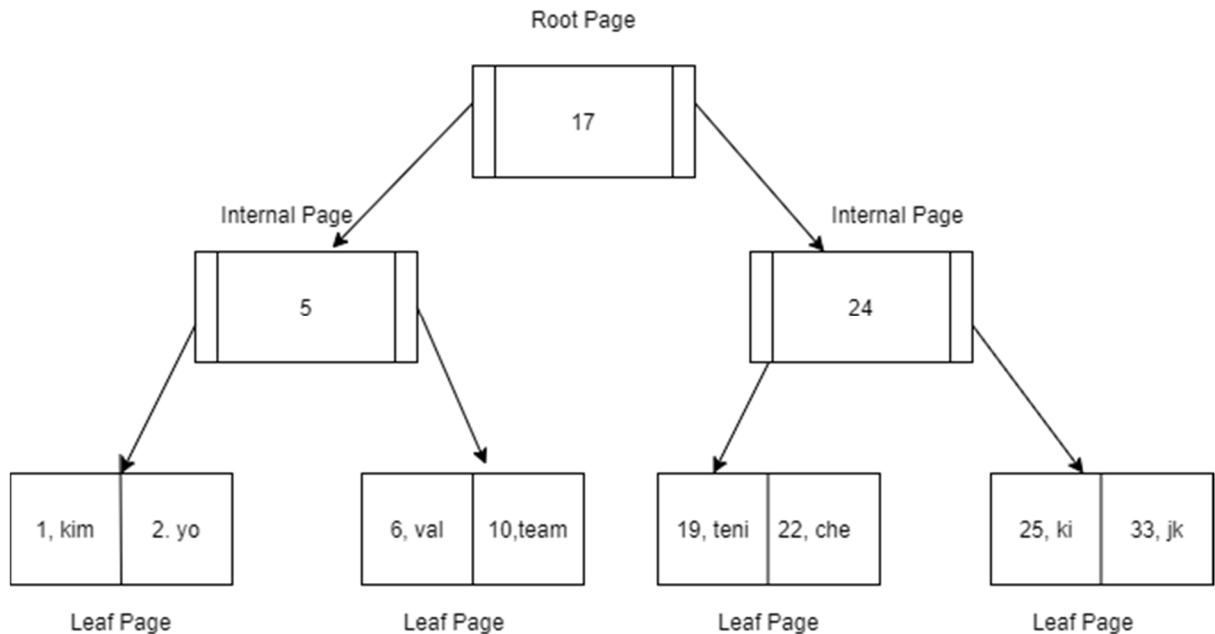
모든 buffer를 개개의 table_id에 flush하고 모든 buffer를 지운다.

Void buf_file_initialize(int table_id)

해당 table_id의 모든 buffer를 제거하고 데이터 파일을 초기화한다.

4.Index Layer

1) Overall Design of Index Layer



2) Design Implementation

본 데이터 베이스의 index layer의 구조는 B+Tree with delayed merge입니다.

모든 데이터를 담고 있는 페이지 즉, leaf page의 height가 모두 동일하며 M-way search tree이므로 모든 page에 대해서 rapid search를 지원합니다.

또한 merge를 진행할 leaf page의 record 개수가 최대 페이지의 개수의 절반 이하 일 때 바로 하는 것이 아니라 leaf page가 모두 비워질 때만 진행하여 B+Tree의 약점인 insert와 delete의 overhead중 delete의 overhead를 일부 해결한 구조를 가집니다.

3)Type of Referencing page

1. Leaf page

해당 page의 0~128byte부분은 page의 header 부분으로 메타데이터를 담고 있습니다.

- Number of keys부분은 페이지의 크기가 4096bytes로 고정이기 때문에 최대 31의 값을 가질 수 있습니다.

Parent Page Number (0~8)
Is Leaf (8~12)
Number of keys (12~16)
(Reserved) (16~120)
Right Sibling Page Number (120~128)
key (8), value (120)
Key (8), value (120)
...
...
Key (8), value(120) (~4096)

2. Internal page

Leaf page와 마찬가지로 0~128byte부분은 page의 header부분으로 메타데이터를 담고 있습니다.

-Number of Keys부분은 페이지의 크기가 4096bytes로 고정이기 때문에 최대 248의 값을 가질 수 있습니다.

-One more page Number는 Internal page의 page들중 가장 왼쪽에 있는 page의 pagenum을 나타냅니다.

Parent Page Number (0~8)
Is Leaf (8~12)
Number of keys (12~16)
(Reserved) (16~120)
One more page Number (120~128)
key (8), Page number (8)
Key (8), Page number (8)
...
...
Key (8), Page number (8) (~4096)

4)Overall Performance(average)

Notation: N(전체 페이지 수), m(internal page의 최대 키 수), m'(leaf page의 최대 키 수)

α (page factor: page에 레코드가 들어있는 비율, 평균적으로 2/3)

Find: $O(\log_M^{N/\alpha} + m')$

Update: $O(\log_M^{N/\alpha} + m')$

Insert: $O(\log_M^{N/\alpha} + m')$

Delete: $O(\log_M^{N/\alpha} + m')$

5) Index Layer API

Int init_db(int buf_num)

Buf_num만큼의 buffer풀을 생성하여 buffer를 활성화합니다. (성공:0, 실패: -1)

Int open_table(char* pathname)

해당 pathname에 있는 데이터베이스를 활성화하고 buffer에 해당 데이터 베이스의 header page를 할당합니다. 모든 init_db를 제외한 모든 함수보다 일찍 실행이 되어야합니다. (성공: pathname에 대한 table id, 실패: -1)

Int close_table(int table_id)

해당 table_id에 해당하는 buffer를 모두 flush합니다.

Int shutdown_db()

모든 buffer를 각각 table id에 해당하는 데이터베이스에 flush하고 데이터 베이스를 닫습니다.

Int db_find(int table_id, int64_t key, char* ret_val, int trx_id)

해당 table_id의 데이터 베이스의 key에 해당하는 데이터를 찾아 ret_val에 저장합니다. (싱글 스레드: trx_id=0, 멀티스레드: trx_id>0), (성공:0, 실패: -1, abort: -2)

Int db_update(int table_id, int64_t key, char* values, int trx_id)

해당 table_id의 데이터 베이스의 key에 해당하는 데이터를 찾아 values로 덮어씁니다. (싱글 스레드: trx_id=0, 멀티스레드: trx_id>0), (성공:0, 실패: -1, abort: -2)

Int db_insert(int table_id, int64_t key, char* value)

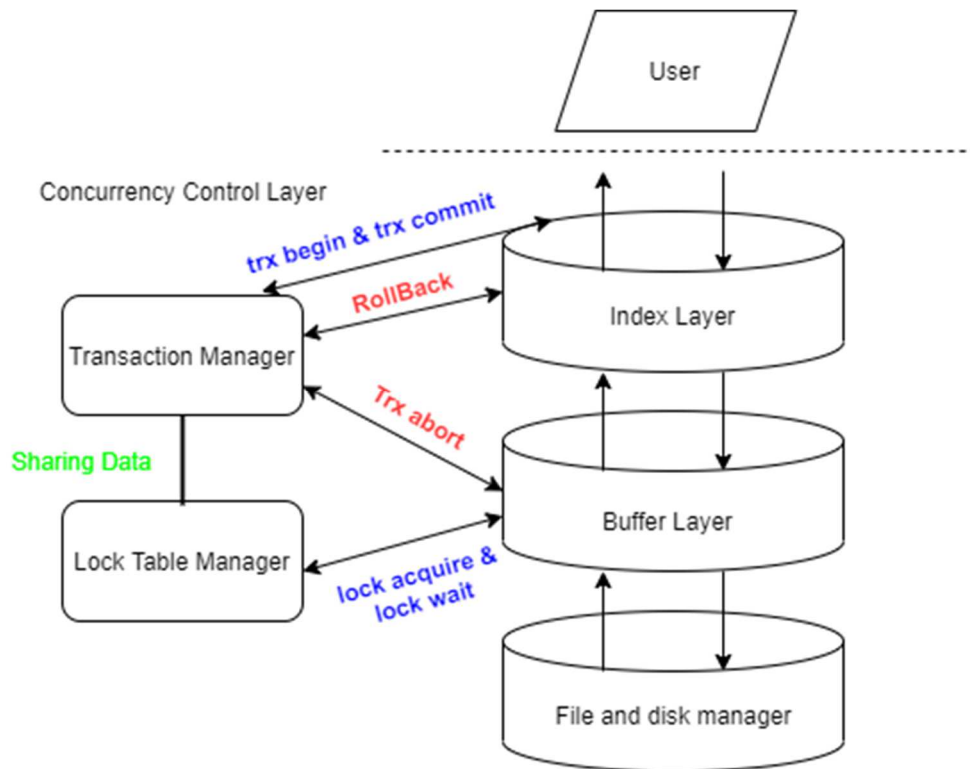
해당 table_id의 데이터 베이스에 key+value를 삽입합니다. 오직 싱글스레드의 경우에만 작동합니다. (성공:0, 실패: -1)

Int db_delete(int table_id, int64_t key)

해당 table_id의 데이터 베이스에 key에 해당하는 record를 지웁니다. 오직 싱글스레드의 경우에만 작동합니다. (성공:0, 실패: -1)

Concurrency Control Implementation

1. Overview of Concurrency Control with Layered Architecture



<그림 2>

본 데이터 베이스의 Concurrency Control Layer는 Transaction Manager와 Lock Table Manager로 나누어져 있습니다.

먼저 Lock Table Manager의 역할을 살펴보면 Buffer로부터 특정한 데이터 (table_id, pagenum, record idx로 결정되는)의 쓰기, 읽기에 대한 lock의 요청을 관리하는 역할을 가집니다. 이때 lock을 가지려고 하는 trx이 기다려야하면 lock wait가 buffer layer 내부에서 수행이 됩니다. Lock을 가지려고 하는 trx가 deadlock 상황이 발생하면 transaction Manager가 이를 처리합니다.

Transaction Manager는 Index Layer로부터 trx begin이 불리면 trx을 만들고 trx commit이 불리지면 해당 trx와 trx에 속한 모든 lock들을 제거합니다. 중간에 deadlock 상황이 발생하여 Buffer Layer로부터 trx abort가 불리면 Index Layer의 roll back을 이용하여 모든 record를 복원한 후 해당 trx와 trx에 속한 모든 lock들을 제거합니다. 그전까지는 절대 lock을 해제하지 않는 strict-2PL 구조를 가지고 있습니다.

2. lock mode

1) kind of lock

Lock Mode	Shared(R)	Exclusive(R)
-----------	-----------	--------------

lock mode는 다음과 같이 2가지 shared, Exclusive lock 2가지가 존재합니다. Shared lock는 record를 읽으려고 할 때 걸리는 lock이며 Exclusive lock는 record를 쓰려고 할 때 걸리는 lock입니다.





이때 lock의 priority는 Exclusive>Shared 이므로 Exclusive lock이 이미 존재하면 shared lock의 획득 없이 데이터를 읽을 수 있습니다.

2) Conflict of lock

이들은 특별한 경우를 제외하면 오른쪽 그림과 같이 shared & shared lock이 걸리는 경우에만 두 trx가 모두 lock을 획득할 수 있습니다.

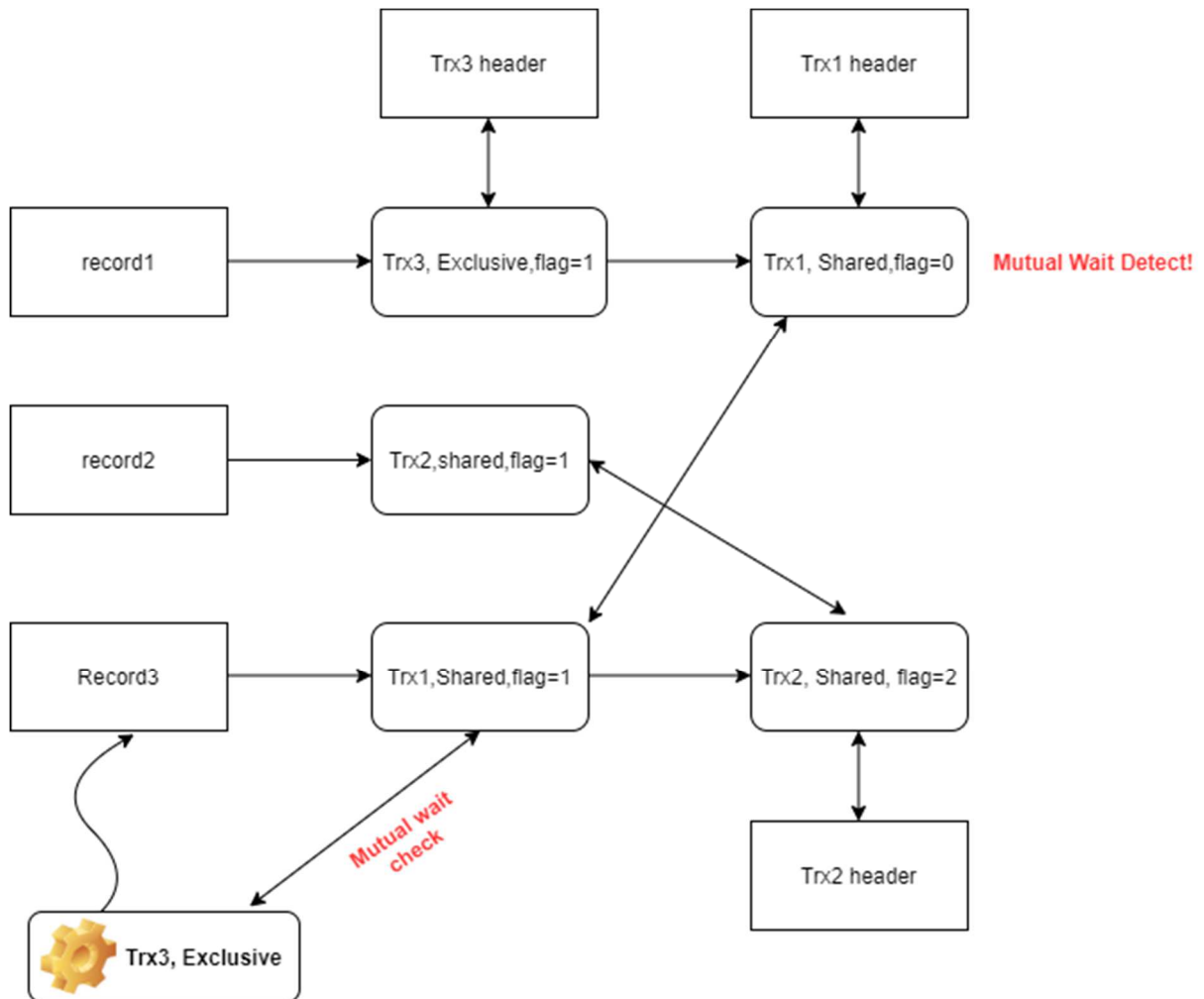
---예외적인 경우

같은 record에 대해서 같은 trx가 lock을 얻으려고 하는 경우 해당 record에 trx가 자신만 존재할 때 shared & Exclusive lock이 공존할 수 있습니다.

Lock Mode	Shared(R)	Exclusive(R)
Shared(R)		
Exclusive(W)		

3. Dead Lock Detection

본 데이터베이스의 Dead Lock Detection 방식은 pessimistic deadlock detection 방식인 wait-for-graph를 이용하는 방식입니다. 이는 서로 다른 record에서 서로 다른 trx가 서로 conflict인 lock를 acquire하려는 시도가 있을 것이라는 판단하에 Trx가 가진 lock을 모두 trx에 모두 연결해 놓고 그러한 시도가 생기는 것을 lock acquire하기 직전에 판단하여 dead lock을 방지하는 방식입니다.



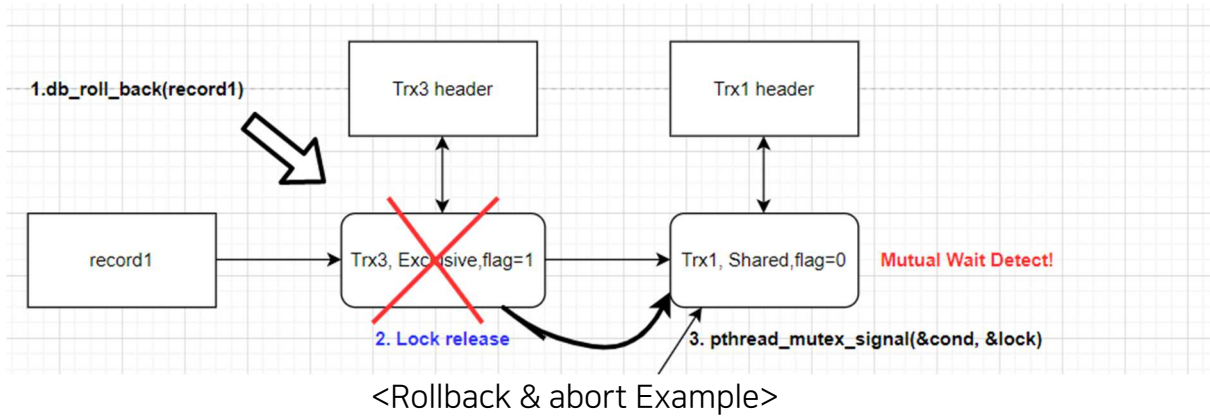
<Dead Lock detection Example>

구체적인 dead lock detection 방식

1. 현재 lock acquire 하려는 lock이 lock_wait를 진행해야하는 지 확인한다.
2. Lock wait가 필요하다면 record의 head lock의 trx부터 하나씩 trx header 확인
3. 찾은 trx의 head lock부터 하나씩 검사해가면서 mutual wait이 존재하는 지 확인한다.
4. Mutual lock이 존재하면 deadlock이다.

4.Abort and Rollback

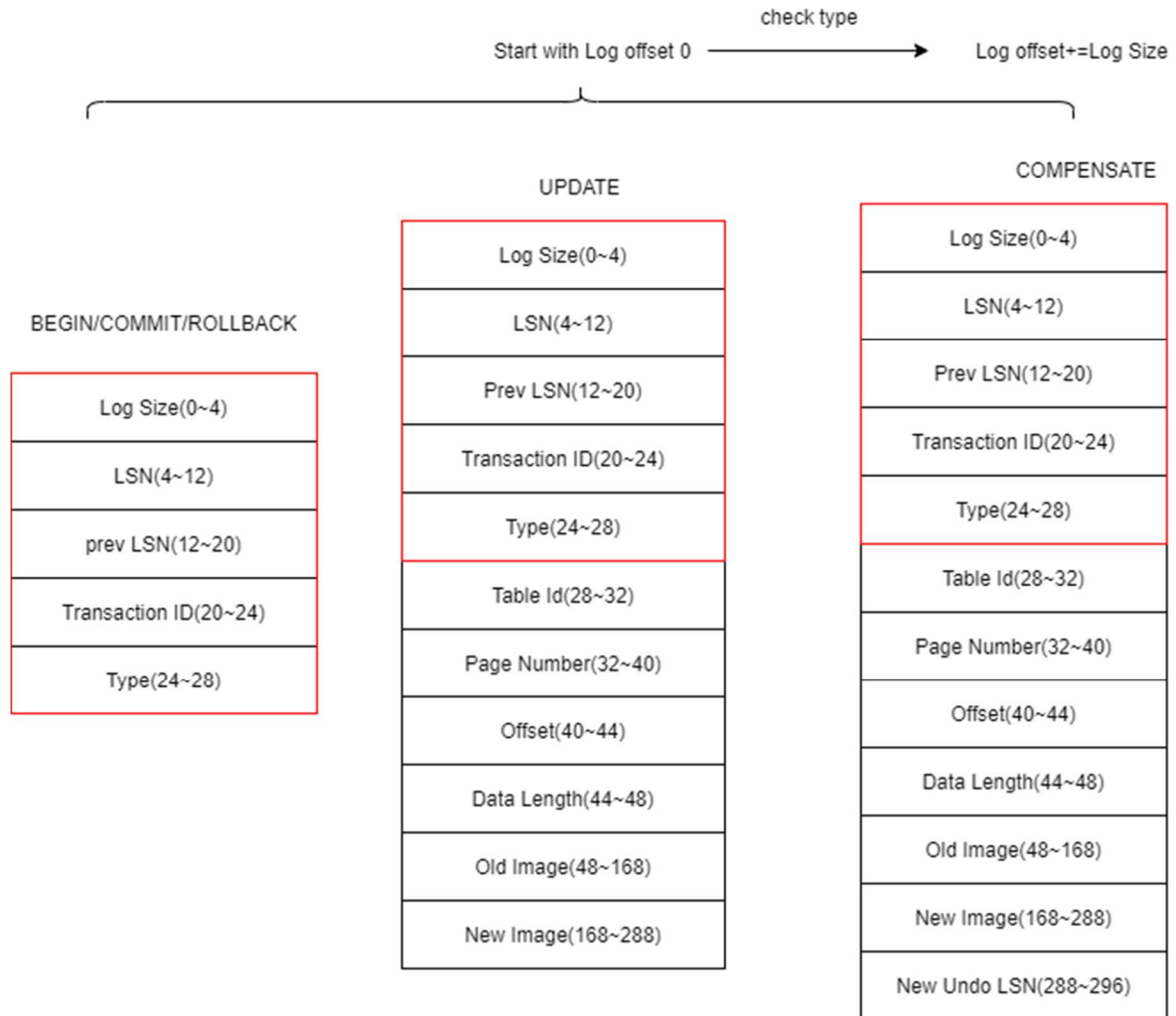
위의 deadlock detection으로부터 abort가 발생하면 trx_abort가 실행되어 roll back & abort가 발생합니다. Rollback & abort의 처리 과정은 다음과 같습니다.



1. Index layer의 db_rollback()을 이용해서 trx에 저장해두었던 백업데이터를 복구합니다.
2. 모든 record의 복구가 끝나면 trx에 속해있는 모든 lock을 해제합니다.
3. Lock을 해제하는 과정에서 is_need_wake_up() 함수를 이용해서 필요시 잠자고 있던 thread를 깨웁니다.

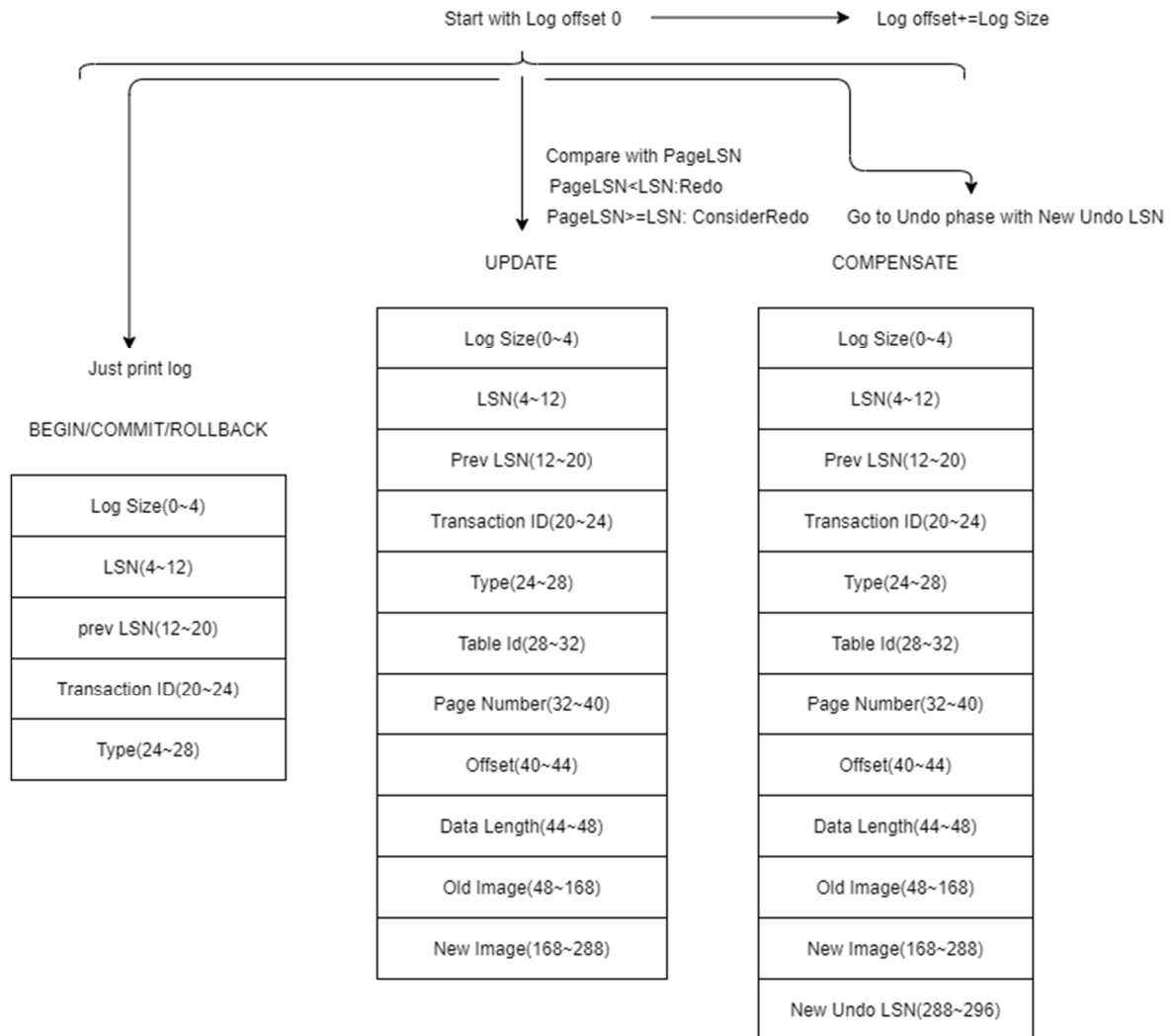
Log Layer는 Log record를 담고 있는 Layer로 trx가 commit이 되거나, Log Buffer Layer가 꽉 차게 되거나, Buffer layer에서 page eviction이 발생하면 즉, Buffer Layer의 record가 disk Layer가 들어가는 상황이 오면 WAL(Write-Ahead Logging)정책에 의해서 먼저 Log Layer에 Log Buffer에 있는 모든 Log가 flush되어 disk layer보다 항상 같거나 높은 최신도를 가지게 만듭니다.

2. Analysis Pass



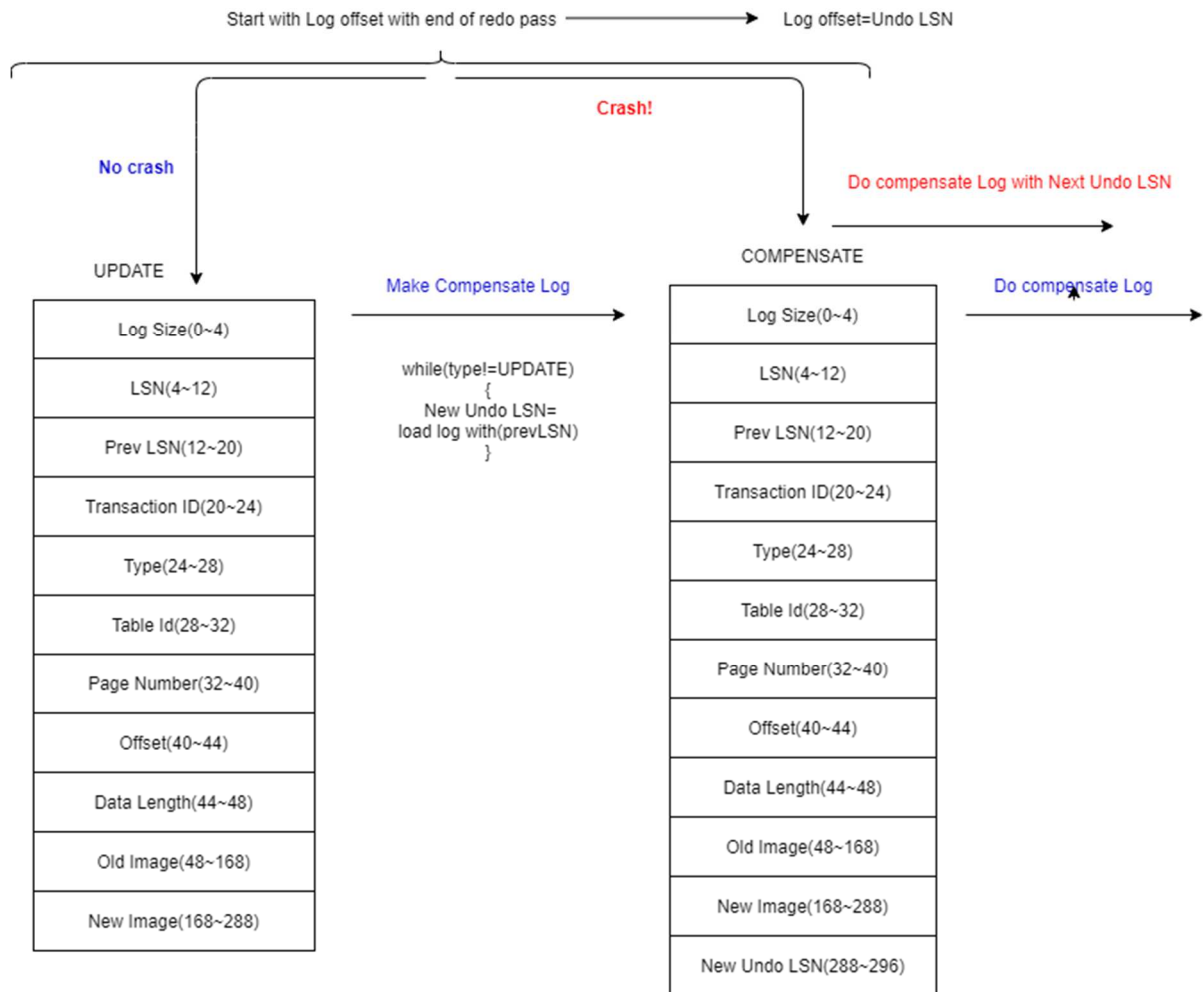
1. Log offset 0부터 시작해서 먼저 28bytes만큼 읽어서 해당 Log size와 Type, transaction ID를 판단한다.
2. 만약 현재 log가 begin/commit/rollback중 하나이면 Log offset+=28, Update면 log offset+=(288), Compensate면 Log offset+=(296)를 수행한다.
3. 1~2를 반복하며 transaction id에 해당하는 trx_begin()이후에 rollback이나 commit이 나왔다면 winner, 아니면 loser로 취급하여 로그 메시지를 출력한다.

3.Redo Pass



본 데이터 베이스는 ARIES 방식의 crash-recovery를 사용하고 있으므로 Redo phase에서는 가능한 모든 Redo를 수행한다. 하지만 이때 Backward chaining을 방지하기 위해서 Compensate Log를 만나면 바로 Undo Phase에 진입한다. Update의 경우 pageLSN과 비교해서 log record의 pageLSN이 더 큰 경우에만 교체한다.

4. Undo Pass



Undo pass의 경우 중간에 crash가 발생한 경우와 그렇지 않은 경우로 나눌 수 있습니다.

Crash가 발생하지 않은 경우

중간에 crash가 나지 않은 경우 UPDATE의 Compensate Log를 발급해가면서 New Undo LSN을 해당 trx의 현재 UPDATE Log의 바로 이전 UPDATE를 가리키도록 합니다. 그리고 log offset을 이전 Update LSN으로 바꾸어가면서 계속 반복합니다. 이후 log offset이 0이 되면 해당 trx id로 db_rollback()을 사용하여 rollback합니다.

Crash가 중간에 발생한 경우

중간에 crash가 발생한 경우 compensate log의 발견으로 인해서 Undo pass에 오게 되는데 이때 compensate log를 수행하고 log offset을 New Undo LSN으로 바꾸어 진행하고 New Undo LSN이 0이 되면 해당 trx id로 db_rollback()을 사용하여 rollback합니다.

In depth Analysis

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

1) Expected Problems

본 데이터 베이스의 Concurrency Control의 구현은 Pessimistic으로 구성되어 있기 때문에 다음과 같은 문제 점들이 발생할 수 있습니다.

- Lock manager overhead

해당 lock을 추가하기 위해서는 lock table에 오직 하나의 trx만이 들어갈 수 있기 때문에 다량의 읽기가 발생할 경우 non-conflict여도 대기시간이 늘어납니다.

- Deadlock detection & solution

Deadlock을 찾을 때 해당 record에 걸려있는 모든 lock을 소유하고 있는 trx를 확인하여 deadlock detection을 해야하기 때문에 non-conflict임에도 불구하고 최악의 경우 모든 trx를 전부 확인해야하는 경우가 발생할 수 있습니다.

Deadlock solution의 경우에도 해당 trx가 잡고 있는 모든 lock을 확인하고 필요시 잠자고 있는 lock을 깨워야 하는데 이 경우에도 lock table manager의 lock을 잡아야하므로 overhead가 발생합니다.

2)Solution

이에 대한 해결책은 Optimistic Concurrency Control을 도입하는 것입니다. Optimistic Concurrency Control는 “lock conflict가 발생할 확률이 낮다”라는 main idea에서 출발하는 Concurrency Control로 이것이 transaction을 처리하는 방법은 다음과 같습니다.

Read phase: trx이 읽을 데이터를 commit이 마지막으로 완료된 db로부터 데이터를 가져옵니다.

Valid Phase: 해당 trx가 다른 trx와 conflict가 발생하는지 확인합니다.

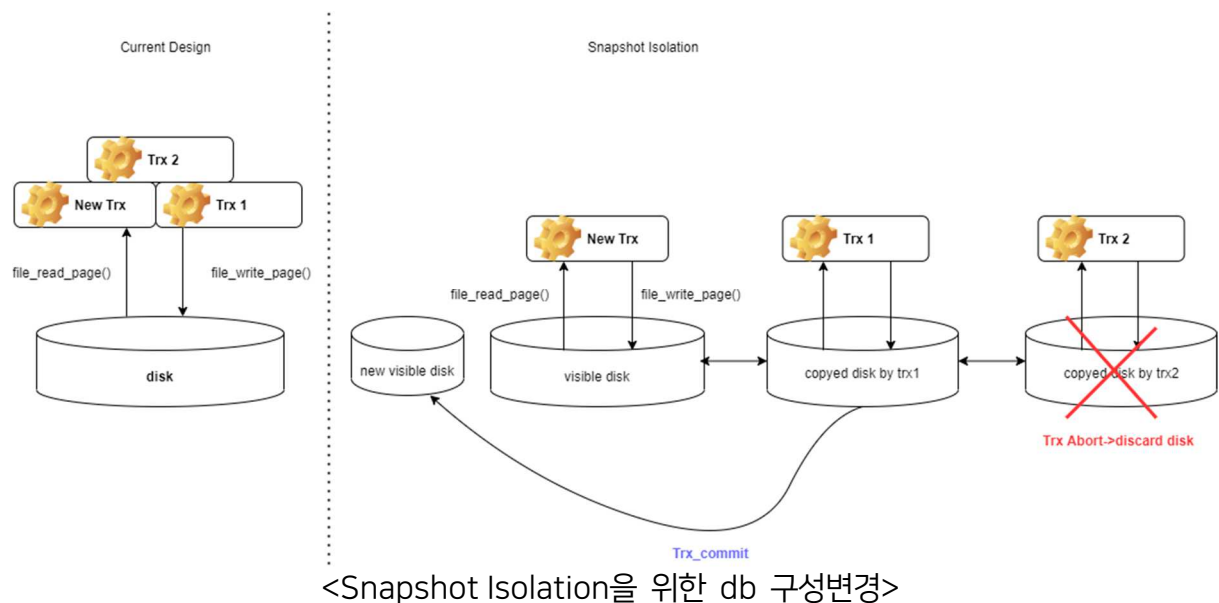
Write Phase: conflict가 없는 trx는 db에 데이터를 삽입하고 아니면 abort시킵니다.

이제 이 구현을 위한 디자인에 대해서 알아보겠습니다.

1. Read phase

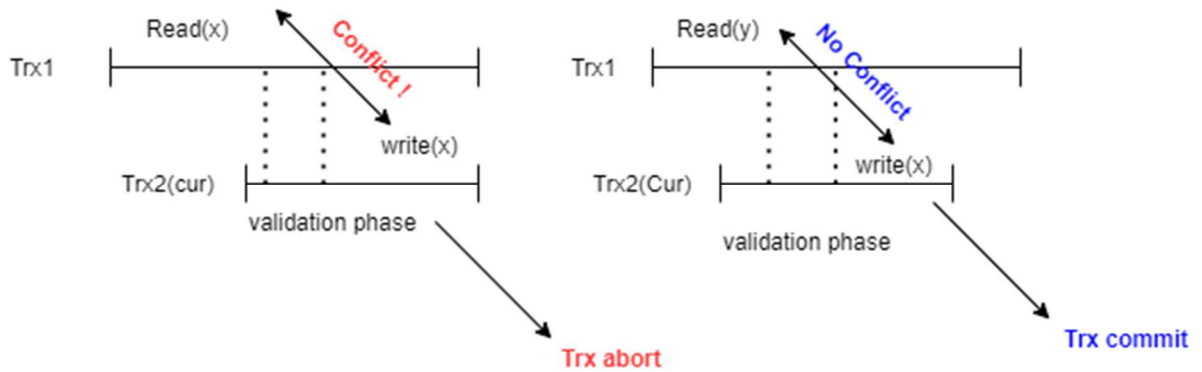
Trx이 데이터를 읽을 때 commit이 마지막으로 완료된 db로부터 데이터를 가져오려면 현재 데이터 베이스가 correctness을 위해서 가지는 격리수준인 serializable보다 한 단계 낮은 격리수준인 snapshot isolation으로 격리수준을 낮추어야 합니다. 이는 마치 데이터 베이스가 멈춘 것처럼 중간에 다른 trx에 의해서 db에 update가 발생하더라도 commit하기 전까지는 trx에서는 해당 update가 반영되지 않고 데이터 베이스 operation들이 작동하는 것을 의미합니다.

이를 이용하면 read의 경우 아예 lock이 없는 것처럼 작동하기 때문에 매우 빠른 속도를 낼 수 있습니다.

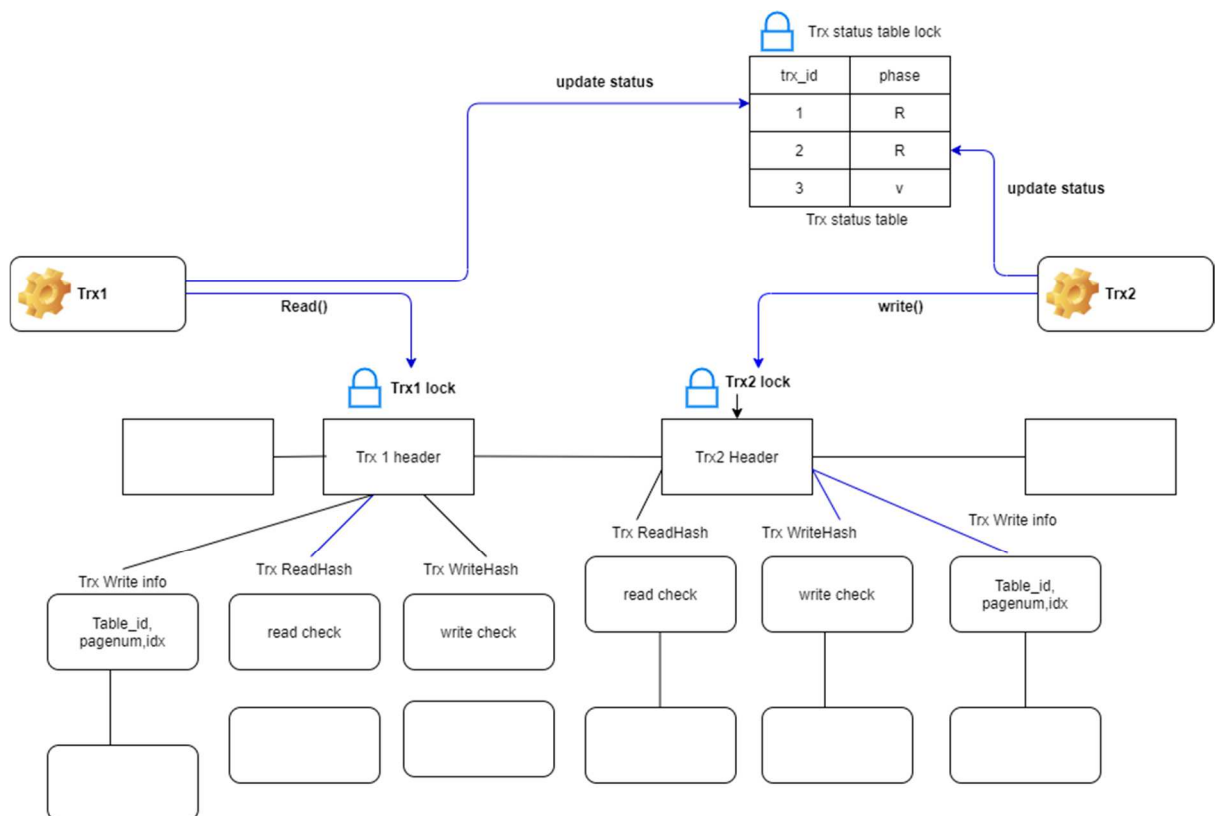


2. Valid Phase

FOCC Validation: validation을 확인하려는 trx의 write set과 현재 작동 중인 trx의 read set의 교집합이 있는지 확인하고 교집합이 없다면 valid, 아니면 invalid



<FOCC Validation Example>



이 FOCC Validation의 구현을 위해서는 lock manager의 변경이 필요합니다. 그 디자인은 아래와 같습니다

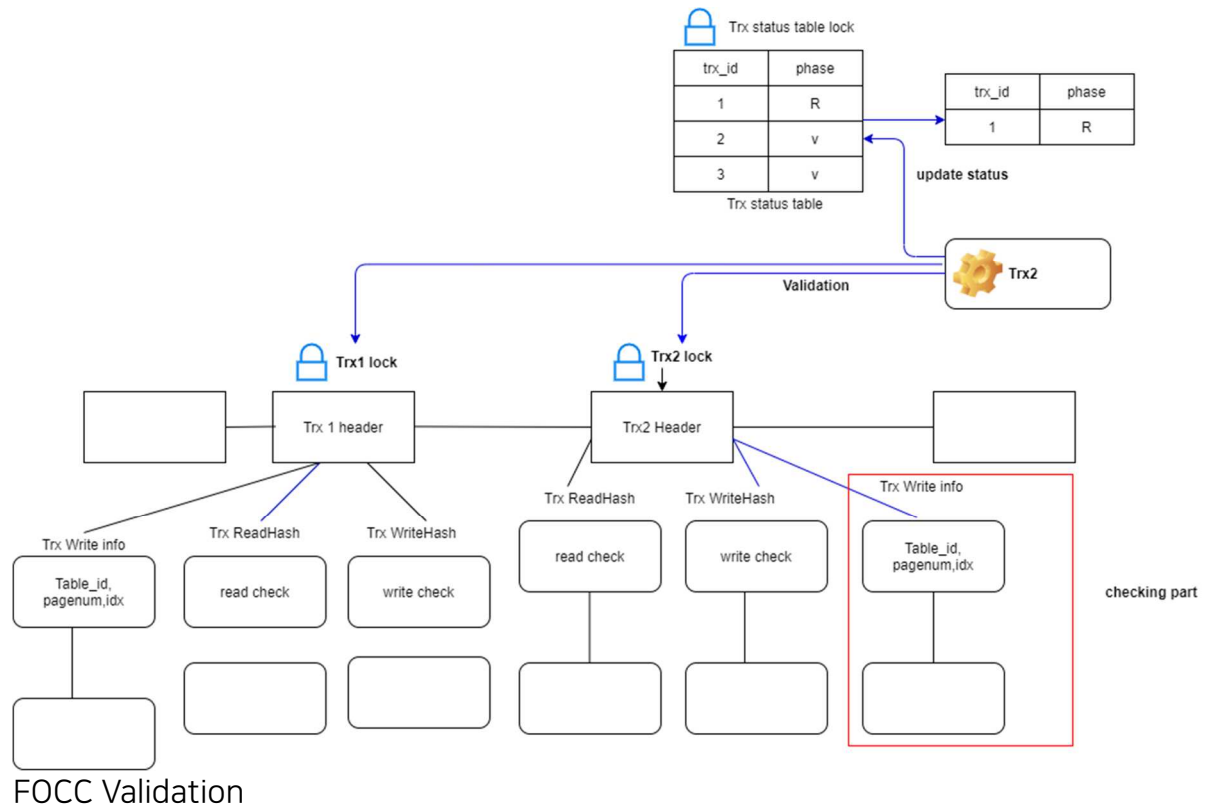
<FOCC Validation을 위한 lock manager 변경>

1. Read

- 1) Trx status table을 update한다.
- 2) Trx lock header를 찾고 Trx ReadHash에 Read check를 삽입한다.

2. Write

- 1) Trx status table을 update한다
- 2) Trx lock header를 찾고 Trx WirteHash에 Write check를 삽입한다.
- 3) Trx write info를 추가한다.



<Validation Phase Example>

1. Trx status를 update하고 Trx status table에서 현재 Read phase인 Trx_id의 정보를 복사해둔다.
2. 자신의 trx_id를 통해서 trx_header를 잡는다.
3. Trx write info가 존재한다면 해당 Table_id, pagenum ,idx로 다른 Read phase의 Trx에도 이 정보가 존재하는지 확인한다.

4. 존재한다면 abort, 아니면 1~3을 반복한다.

5. 이후 trx hash와 trx status table에서 자신의 trx를 제거한다.

3. Write Phase

<Snapshot Isolation을 위한 db 구성변경>과 같이 validation phase가 성공적으로 끝나면 현재 disk를 새로운 visible disk로 만들고 아닐 경우 disk를 버린다.

3)Check problems

1. Lock manager overhead

해당 Trx가 각각의 trx정보를 관리하고 validation phase이외에 다른 상황에서 다른 trx가 trx lock을 가지지 않으므로 lock manager overhead를 줄일 수 있다.

2. Deadlock detection

위 디자인의 deadlock detection은 validation phase라고 할 수 있는데 Read only trx의 경우는 trx write info가 존재하지 않아 validation phase가 바로 끝나기 때문에 매우 빠른 속도로 validation phase을 끝낼 수 있다.

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

1) Expected Problem

- too long unflushed log record

Non-conflict write only-transaction이 많이 수행되면 log buffer 내에서 많은 UPDATE log가 남게 되는데 이때 만약에 해당 trx가 길어서 log buffer가 다 차는 순간까지 log buffer의 flush가 발생하지 않다가 시스템에 crash가 발생하면 이후 recovery할 때 대량의 update redo가 발생하여 시스템의 성능에 큰 영향을 줄 수 있다.

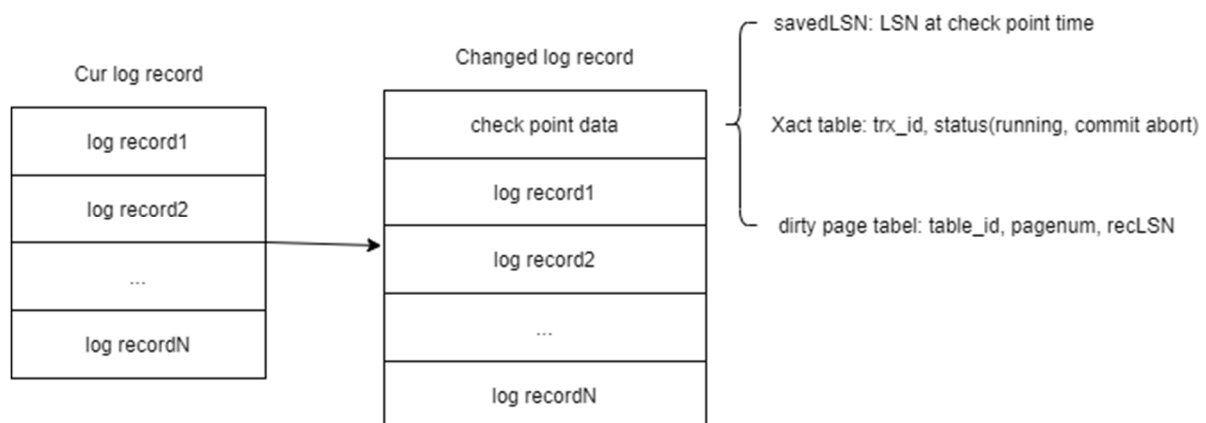
특히 Non-conflict write only-transaction이 같은 record에 대해서 계속 update를 진행할 경우 해당 record는 undo pass에서 수 많은 compensate(update)를 수행하게 되므로 recovery 성능에 큰 영향을 줄 수 있습니다.

2) Solution

Check point

check point란 일정한 주기로 log layer에 log buffer를 flush하는 것을 의미합니다. 이를 구현하게 되면 recovery의 3pass(Analysis, Redo, Undo)가 수행해야할 log record의 수 자체가 줄어들게 되기 때문에 상당한 성능의 향상을 기대할 수 있습니다.

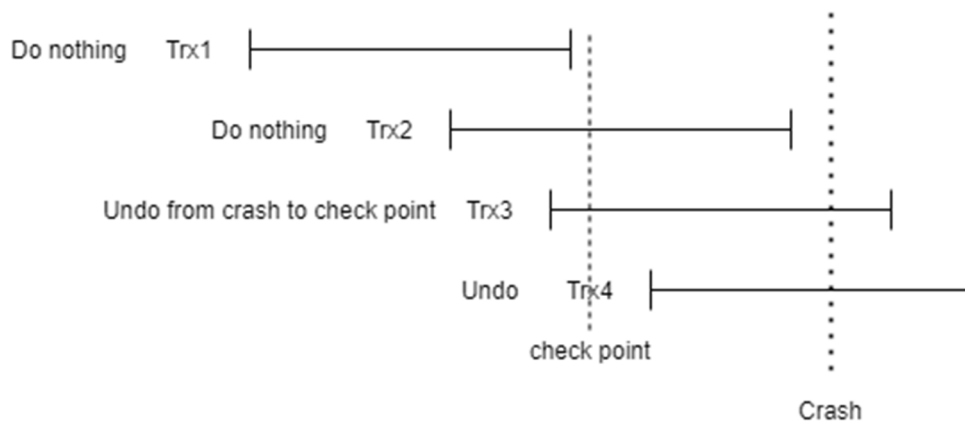
<check point를 위한 log file 변경>



savedLSN: 저장한 순간의 LSN

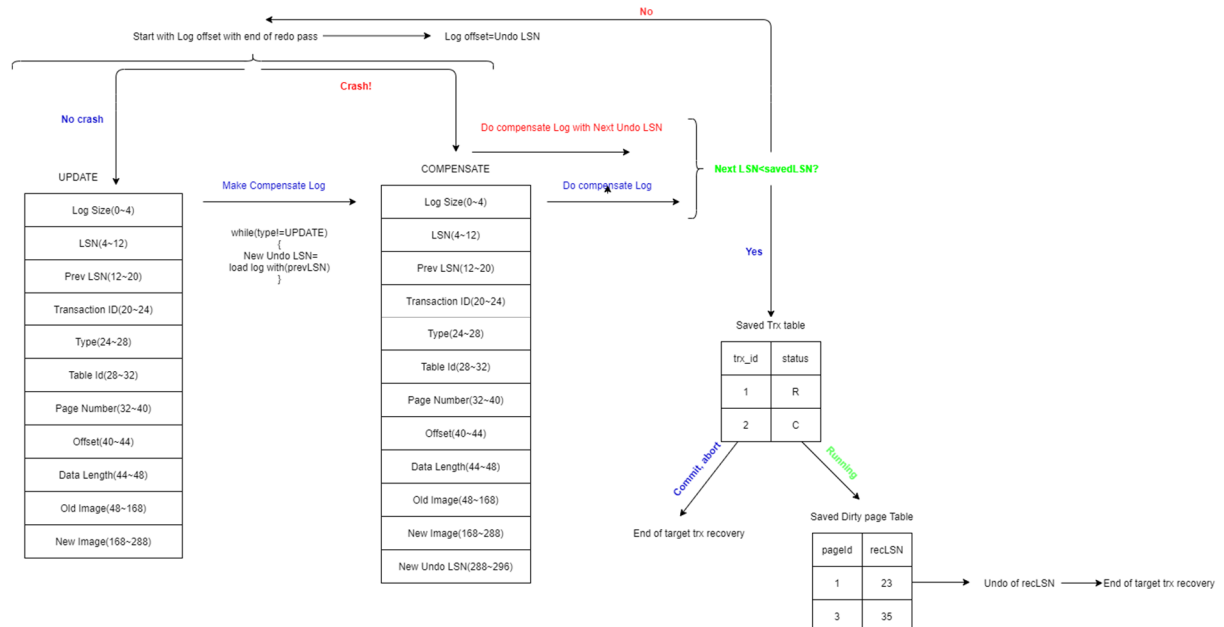
Xact table: 저장한 순간의 trx_id와 그 trx의 상태를 저장한 것

Dirty page table: table_id, pagenum, recLSN(page를 dirty하게 만든 첫번째 LSN)



<check point에 따른 Trx처리>

주요한 Undo pass 분석



기존과 달리 NextLSN이 저장된 savedLSN보다 작고 rollback이 필요하다면 saved dirty page의 Undo 한번으로 해당 page의 Undo를 마칠 수 있다.