# 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.

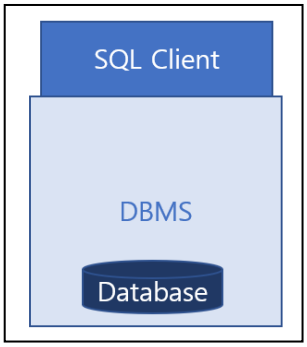
# Database System 2020-2
# Final Report

ITE2038-11800
2019094511
김준표

**Table of Contents**

**Overall Layered Architecture**

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




DBMS 의 layered architecture 에 대해 더 면밀한 이해를 돕기 위해 우선 DBMS 의 역할과 필요성에 대하여 살펴보자.

전체적인 구조를 보면 데이터베이스에 대한 접근은 왼쪽의 구조에 따른다. User 가 SQL Client 를 통하여 일련의 정보를 요청하면, SQL Client 는 DBMS 에 데이터 처리를 요청하고 DBMS 를 통해 Database 에 접근하여 데이터를 불러오게 된다.

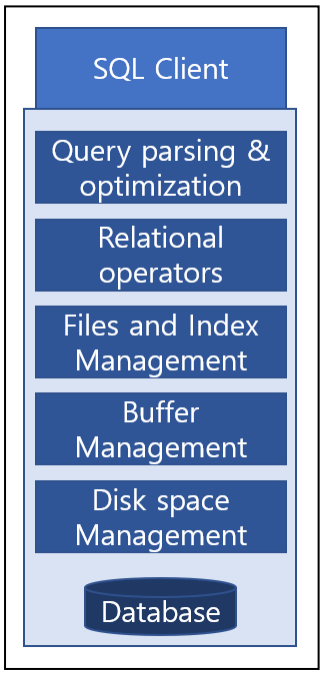
SQL Client:
SQL Client 는 데이터에 대한 structure 를 형성한다. SQL 이란 데이터베이스를 제어하고 관리하는 질의언어로 Data Definition Language(DDL)와 Data Manipulation Language(DML)로 나눠진다.

DDL 은 데이터 베이스 또는 테이블을 생성하거나 수정, 삭제 또는 초기화 작업을 하는 데이터 처리 형식에 관련된 관리를 실시하는 언어이고, DML 은 데이터를 조작하는 언어로 데이터를 update, insert, select, delete 의 동작을 실시한다.

DBMS:
DBMS 에서는 SQL Client 에서 전달받은 SQL query 를 어떻게 처리할 지에 대한 계획과 동작을 실시한다. DBMS 는 여러 layer 로 구성되어 있으며, 각각의 layer 는 독립적인 역할을 갖고 해당 역할에 맞게 상호작용을 하여 데이터를 처리한다.

Database:
Database 는 데이터의 모음이다. 그러나 이는 단순히 데이터를 모아 놓은 것은 아니다. Database 는 데이터를 structed way 로 구성하여 가공한 뒤 모은 데이터의 모음이다. 즉, 데이터를 Schema 로 구조화된 데이터를 모은 데이터 집합인 것이다.

DBMS 의 layer architecture 는 다음 그림과 같이 구성되어 있다.
SQL Client 에서 Database 까지 접근하는 과정은 다음과 같다.




Query Parsing & Optimization:
Query Parsing 단계에서는 SELECT FROM WHERE 로 구성된 SQL 언어를 쪼개 이를 구조화 시킨다. 이후 구조화된 정보를 Relational Operators 단계로 넘겨준다.
Query Optimization 단계에서는 Relational Operators 단계로부터 받은 정보를 토대로 Query plan 을 짜는 작업을 수행한다. 해당 layer 에서는 어떻게 하면 주어진 query 를 짧은 시간안에 끝낼 수 있는지 계산한다.

Relational Operators:
Relational Operators 단계에서는 Query parsing 단계에서 받은 정보를 조합하여 Query Optimization 단계로 전달해주는 역할을 한다. Query Parsing & Optimization layer 와 더불어 레코드에 대한 요청을 처리하는 작업을 한다.

Files and Index Management:
Files and Index Management 단계에서는 레코드에 대한 요청을 페이지에 대한 요청으로 변환하여 페이지에 접근을 실시한다. 이 layer 에서는 데이터를 찾고자 하는 테이블에 접근하며 헤더페이지와 페이지를 구성하여 data 를 구역별로 logical 하게 나누는 pagination 처리를 실시한다. 또한 write page API 와 read page API 를 호출하여 데이터 처리를 요청한다.

해당 내용에 대한 적용은 project2 와 project3 에서 구성했으며, bpt 를 동작시키는 bpt.c 부분에서 이를 구현했다.

Buffer Management:
Buffer Management 단계에서는 메모리와 디스크의 속도차를 극복한다. Database 의 데이터(레코드 또는 페이지의 형태)는 On-disk 상태로 저장되어 있다. 이를 읽고 쓰는 데에는 물리적인 탐색이 필요하여 시간이 오래 걸린다. 이를 Buffer 에서 In-memory 상태로 저장하여 데이터를 읽고 쓰는데 소요되는 시간을 단축시킨다. 해당 layer 는 correctness 가 목적이 아닌 efficiency 가 목적이다.

해당 내용에 대한 적용은 project3 에서 구성했으며 buffer layer management 를 실시하는 buffer.c 를 통해 on-disk bpt 를 좀 더 효율적인 동작이 가능하게 했다. Buffer layer 에서는 buffer 수를 입력 받아 init_db 를 수행하고, 해당 함수에서는 buffer pool 을 초기화하는 작업을 수행한다. 이 때, circular double-linked list 를 이용하여

구현함으로써 buffer pool 에서 buffer 에 대한 접근이 용이하도록 구성했다. 해당 layer 에서 생성하고 관리하는 buffer 에 대하여 File and Index Manager 에서 이를 요청할 때, get_buffer 함수를 통해 buffer pool 로부터 buffer 를 가져와 넘겨주도록 구성했다. 이후 buffer 를 disk 에 적는 과정에서는 flush_buffer 함수를 통해 buffer 의 내용을 disk 에 복사한 후 buffer 를 비워주는 작업을 실시했다. buffer pool 을 circular double-linked list 로 구성하여 각각의 동작에서 버퍼를 찾고, 가져오고, 다시 돌려놓는 세부적인 작업을 위한 함수들도 만들어 buffer manager 를 구성했다.

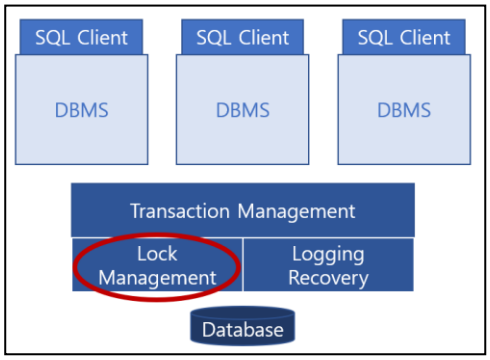
Disk Space Management:
Disk Space Management 단계에서는 페이지의 데이터를 File Offset 에 대한 요청으로 변환하여 디스크에 접근을 실시한다. 해당 layer 에서는 read 와 write 에 대한 system call 을 실시한다.

이에 대한 내용은 project2 에서 구성했으며 file.c 에서 File manage API 들을 구현했다. project2 의 내용은 in-memory bpt 를 on-disk bpt 로 수정하는 것이었다. 이를 위해서 page 를 alloc 하고, free 하는 함수를 통해 페이지 생성과 삭제를 관리했으며 file_read_page 와 file_write_page 를 각각 pread 와 pwrite 함수를 통해 동작하도록 설계했다.

## Concurrency Control Implementation

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

Concurrency Control 이 필요한 이유는, 여러 사용자가 데이터를 요청했을 때, 이를 효율적으로 빠른 시간 내에 올바른 결과를 도출하기 위함이다. 또한 Concurrency Control 이 지향하는 점은 여러 사용자가 동시에 시스템을 사용하더라도 혼자 시스템을 사용할 때와 동일한 결과를 도출해내는 것이다. 이에 대해 구조적으로 살펴보면 오른쪽 그림과 같다.



여러 DBMS 시스템이 하나의 Database 에 접근할 때 필요한 단계가 Transaction Management 이다. Transaction 이란 하나의 온전한 일의 단위이며 이는 Atomicity 를 만족하여야 한다. 즉, 하나의 atomic unit 으로 고려될 수 있는 일의 단위이다. Transaction Management 단계에서는 여러 시스템이 Database 에 접근할 때, 각각의 시스템에서 요청한 일을 Transaction 단위로 실행하는 작업을 실시한다. 이 작업에는 Lock Management 와 Logging Recovery 가 있으며 이중 Lock Management 가 Concurrency Control Implementation 에 해당한다.

여러 Transaction 을 실시할 때, 최우선으로 고려해야하는 사항이 있다. 이는 바로 여러 사용자가 하나의 데이터베이스에 접근하는 경우 발생할 수 있는 데이터 값의 충돌문제이다. 둘 이상의 여러 사용자가 같은 데이터베이스에 접근할 때, 접근하는 사용자 중 임의의 사용자에 의해 write 가 일어나 데이터베이스의 상태(state)가 변하는 경우에 다음의 3 가지 문제가 발생한다.

1. Lost update
   둘 이상의 transaction 이 동일한 값을 읽어 update 하는 상황에서 발생한다. read 와 write 의 갱신 순서에 따라 한쪽 사용자의 update 한 값이 날아간다.
2. Inconsistent read
   동일 데이터에 서로 다른 여러 transaction 이 접근하는 상황에서 write 처리가 일어나는 경우 발생할 수 있다.
3. Dirty read
   동일 데이터에 서로 다른 여러 transaction 이 접근하는 상황에서 read-write conflict 가 일어나 한쪽 transaction 이 rollback 을 실시한 경우 발생할 수 있다.

위의 3 가지 문제를 피하기 위한 핵심은 write 처리를 제어하는 것이다. 그리고 transaction 의 일련의 처리를 하나의 단위로 설정하여 아래와 같이 제어함으로써 이를 충족시킬 수 있다.

1. Begin transaction
2. Do work (read/write)
3. Commit transaction

Transaction 단위의 설정 단계에서 고려해야할 점은 해당 디자인이 ACID 를 만족시키는지의 여부이다.

A: Atomicity, 각각의 transaction 은 하나의 단위로써 그 내용이 모두 실행되거나 모두 부결되어야 한다.

C: Consistency, Database 의 상태는 시작시점부터 종료시점까지 일관되어야 한다.

I: Isolation, 각각의 transaction 은 독립적으로 작동해야 한다.
D: Durability, Transaction 이 종료되면 이의 반영내용은 영구적이어야 한다.

Transaction Management 설계단계에서 고려해야할 사항은 Throughput(처리량)과 Latency(지연시간)이다. 해당 layer 를 디자인할 때, 두가지 변수에 대하여 고려해야 한다. 물론 해당되는 두가지 요인이 모두 최대치인 상태를 이끌어 낼 수 있으면 좋지만, 이 두가지 요인은 서로 충돌하게 된다. 시스템을 제공하는 사업자의 입장에서는 Throughput 이 많아지는 것을 원할 것이고, 시스템을 사용하는 입장에서는 Latency 가 줄어들기를 원할 것이다. 그러나 처리량이 많아지면 지연시간이 길어질 수밖에 없다. 이러한 상황에서 시스템 제공자의 목표와 시스템 사용자의 목표 사이의 합의점을 찾아내는 작업이 필요하다.

하나의 시스템에 여러 transaction 들을 돌리는 가장 단순한 생각은 접근하고자 하는 transaction 을 줄을 세워서 하나씩 접근이 가능하게 하는 것이다. 그러나 이와 같은 방식은 효율성이 매우 떨어지게 되며 우리는 여러 transaction 을 동시에 돌리면서도 안전성을 보장하는 방식을 찾을 수 있다. 이러한 배경에서 Concurrency Control 은 Pessimistic Concurrency Control(PCC)와 Optimistic Concurrency Control(OCC) 두가지 경우로 나누어 생각할 수 있다. PCC 는 conflict 가 발생할 것이라는 가정 하에 transaction manager 를 디자인 하는 방식이고, OCC 는 conflict 가 발생하지 않을 것이라는 가정 하에 transaction manager 를 디자인 하는 방식이다. 두가지 concurrency control 의 목표는 schedule execution 을 serializable 하게 만드는 것이다.

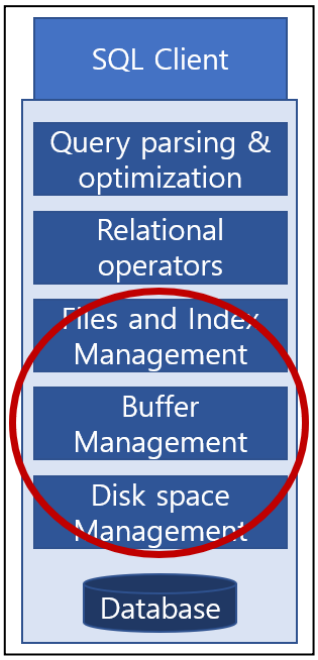
Pessimistic Concurrency Control 은 conflict 가 일어날 것이라고 미리 가정 한 디자인이다. 따라서 이 경우 conflict 에 대한 탐지(detection)작업이 추가된다. 그러나 우리가 DBMS 를 설계할 때 모든 conflict 지점을 알고 설계하는 것은 아니므로 우리는 conflict 가 발생하는 상황을 기준으로 conflict 동작을 실행한다. 이를 위해 필요한 것이 lock 이며 이를 활용한 디자인을 Two Phase Locking(2PL)이라고 한다. 2PL 에서는 shared 와 exclusive 두가지 lock mode 를 사용하여 각각의 lock block 마다의 접근성을 표기한다. 또한 record id 와 transaction id 를 인자로 받아서 해당 transaction 이 해당 record 에 접근 하는 것을 관리한다. 이때, read lock, read unlock, write lock, write unlock 의 조건을 통해서 conflict 가 일어나는 것을 제한한다. 그러나 이 방식에서는 abort 가 일어나는 상황에 대하여 고려하지 않아 abort 가 발생하는 경우 시스템에서 잡은 lock 이 연달아서 해제되면서 dirty read 문제가 발생하게 된다.

이 dirty read 문제를 해결하기 위해 등장한 것이 Strict Two Phase Locking(S2PL)이다. 이 디자인에서는 lock acquisition 과정은 2PL 방식과 동일하지만 abort 가 발생한 경우에는 잡은 transaction 의 끝에서 한꺼번에 release 시켜준다. 이와 같은 방식은 abort 가 발생한 경우에도 database 의 안전성을 보장한다.

Strict Two Phase Locking 에서 abort 가 발생하는 상황 중 하나인 deadlock detect 의 디자인을 잘 설계해야 한다. 이는 waiting graph 에서 cycle 이 발생한 경우 무기한 대기상태로 빠지게 되는 경우이다. 이 역시 데이터를 저장하는 자료구조에 따라 알맞게 구현해야 한다.

Optimistic Concurrency Control 은 PCC 에서 성능을 향상시키는 방향으로 고안된 디자인이다. 이는 conflict 가 일어나지 않을 것이라고 가정을 한 경우이며 이의 디자인을 통해 PCC 에서 발생한 overhead 문제를 해결한다. 그러나 해당 디자인이 PCC 에 비해서 항상 우수한 것은 아니다. 왜냐하면 OCC 의 경우 PCC 와 다르게 conflict 가 발생하면 기다리지 않고 abort 를 수행하기 때문에 write 작업이 많은 경우나 일부 데이터에 대한 접근이 특히 많은 경우(Hot Spot 이 생기는 경우)에 대해서는 abort 가 발생하는 비율이 높아진다.

OCC 는 BOCC 와 FOCC 로 나눠진다. BOCC 는 Backward-oriented Optimistic CC 로 이미 처리가 끝난 operation 을 대상으로 현재 transaction 과 비교하여 conflict 여부를 확인하고 실시한다. FOCC 는 Forward-oriented Optimistic CC 로 다음에 동작할 operation 을 대상으로 현재 transaction 과 비교하여 conflict 여부를 확인하고 실시한다.




Concurrency Control 은 File and Index Management, Buffer Management, Disk Space Management 단계에 걸쳐 구현된다. PCC 와 OCC 의 동작순서에 차이가 있으나 concurrency control 작업의 수행을 위해서는 transaction manager layer 와 더불어 위의 3 layer 에서도 일련의 처리과정을 거쳐야 한다.

PCC 의 경우 lock manager 에서 conflict 여부를 미리 확인한 후 record 에 접근하는 방식으로 동작한다. 이를 단계적으로 살펴보면 다음과 같다.

Files and Index manager 에서는 read API 또는 write API 를 호출한다. PCC 에서는 우선 lock manager 에서 conflict 여부를

확인한다. conflict 가 발생한다면 기다리고 conflict 가 없다면 read API 또는 write API 가 바로 호출된다.

Buffer manager 에서는 buffer 전체의 lock 과 page 단위의 lock 을 관리한다. 이는 In-memory based 구조에서 여러 transaction 이 한꺼번에 버퍼에 접근하는 것을 제한하기 위함이다.

Disk space manager 에서는 lock 이 보장된 후 디스크에 접근하여 읽고 쓰는 동작을 실행하게 된다.

OCC 에서는 우선 record 에 접근 후 commit 단계에서 conflict 를 확인하는 방식으로 동작한다. 이는 FOCC 와 BOCC 두가지로 나누어지는데, 이는 conflict 를 확인하기 위해 비교하는 대상이 이후 validation 인지 이전 validation 인지를 기준으로 나눠진다.

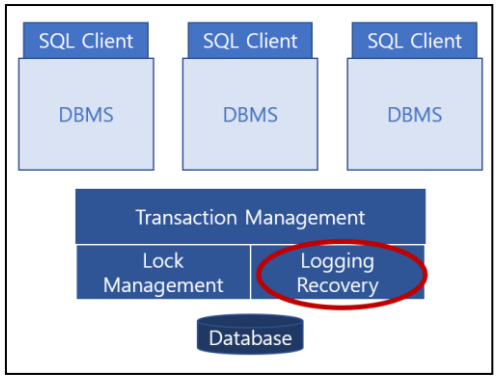
해당 내용에 대해서는 project4 와 project5 에 걸쳐 구성했다. project4 에서는 Concurrency Control 에서 Atomicity 와 Durability 를 만족시키는 디자인을 구현했다. 이를 만족시키기 위해 과제의 구성을 4 단계로 나눠 진행했다.
1 단계: 각각의 Thread 가 lock_acquire 또는 lock_release 를 받아오도록 한다.
2 단계: mutex 를 이용하여 각각의 Lock Table Latch 를 잠근다. 이는 <pthread.h>의 함수들을 사용하여 동작하도록 한다. 이를 통해 각각의 mutex 객체에 접근을 제어한 상태로 작업을 실시한다.
3 단계: 입력 받은 key 와 record 값을 테이블에 삽입하는 동작을 실시한다.
4 단계: lock 의 acquired 와 waiting 을 실시한다.
해당 과정을 통해 여러 Transaction 이 동작할 때에 Atomicity 와 Durability 를 만족시키는 디자인의 기반을 닦았으며 project5 에서 Isolation 과 Consistency 를 추가적으로 만족시키는 디자인을 설계했다.

project5 에서는 Transaction Manager 를 추가하여 여러 Transaction 들에 대해 bpt 를 동작시키는 작업을 실시했다. 해당 과제에서는 Atomicity 를 위해 buffer manager 에 lock 을 추가하고, Consistency 를 위해 Deadlock 을 탐지하고 Abort 를 시키는 것까지 구현하는 것이 목표였다. Transaction 을 따라가며 Deadlock 탐지를 실시하고 이에 대해 Abort 가 일어났음을 확인한 후 이의 리턴 값을 bpt.c 로 넘겨서 Deadlock 이 발생한 Transaction 에 대해 Rollback 시키는 설계를 구성했다.

## Crash-Recovery Implementation

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

Crash-Recovery 는 구조적으로 Transaction Manager 의 Logging Recovery 부분에서 관리되어야 한다. 이의 구조적인 논의를 하기 전에 우선 Crash-Recovery 의 필요성과 어떤 조건을 충족시켜야 하는지에 대해 살펴보자.




Crash-Recovery 는 이전에 논의한 ACID 의 특성 중 Atomicity 와 Durability 를 지키기 위해 동작한다. 먼저 Atomicity 는 Crash 로 인해 Transaction 이 중단되는 경우 해당 Transaction 이 write 와 같은 동작을 수행 중이었다면, 이를 Database 에서 중간결과 상태로 저장하지 않고 실행 이전 상태로 rollback 시키는 Undo 동작으로 만족시킨다.

반면 Durability 는 Crash 로 인해 Transaction 이 중단되는 경우 해당 Transaction 이 Crash 이전에 종료되었다면 이를 Database 에 기록하여 Crash 와 무관하게 Database 에 안전하게 저장하는 동작으로 만족시킨다. 만약 올바르게 Transaction 수행이 끝난 경우 해당 수행 내용이 Database 에 반영이 안되었다면 Redo 작업을 추가적으로 실시해야 한다.

Logging-Recovery 없이 Transaction 에 대한 Atomicity 와 Durability 를 만족시키는 가장 기본적인 방법은 Buffer Layer 를 활용하는 것이다. Buffer Layer 에서 Buffer pool 을 만들어서 해당 Transaction 이 끝날 때까지 Transaction 의 실행을 붙잡고 있는 방식인 No Steal Policy 를 통해 Undo 작업 없이 Atomicity 를 만족시키고, 해당 Transaction 이 종료되면 이를 Database 에 저장하고 commit 을 실시하는 방식인 Force Policy 를 통해 Redo 작업 없이 Durability 를 만족시킬 수 있다. 그러나 이 방식을 사용하면 수행 속도가 매우 오래 걸리며 변동 사항이 많을 때 Buffer 가 모두 차게 되고 overflow 가 생기는 경우가 발생할 수도 있다. 이는 하나의 Transaction 에 많은 일을 부여하지 않는 방식으로 동작할 수 있지만 Concurrency 문제로 인해 성능이 떨어지는 문제가 발생한다.

이러한 성능 문제로 Steal/No-Force Policy 를 시행하며 이 정책을 선택한 경우 Undo 와 Redo 를 통해 Atomicity 와 Durability 를 지킬 수 있다. Undo 작업에서는 Transaction 실행 이전의 Old 값을 기억하여 commit 이전에 Crash 가 발생한 경우에 Database 를 Crash 이전의 상태로 정상 복원을 실시하며, Redo 작업에서는 Transaction 으로 생긴 New 값을 기억하여 commit 이후에 Crash 가 발생한 경우 Database 에 Transaction 의 변동사항을 기록한다. Logging 은 따라서 Old 값과 New 값을 모두 기억해야 한다.

따라서 Logging Recovery 에는 record 에 대해서 Redo 와 Undo 동작을 위한 Log 가 필요하다. 이의 성능향상을 위해서는 In-memory 의 Log Buffer Manager 를 만들어 수행해야한다. Log Buffer Manager 에서는 WAL Policy 를 따라 Buffer Manager 에 저장된 변경 내용들을 Database 에 write 하기 전에 해당 내용을 저장하는 역할을 한다. 이는 Atomicity 를 위해 변경된 data page 가 저장되기 전에 필수적으로 진행되어야 한다. 또한 Durability 를 위해 commit 이전의 내용들을 Log 에 기록해야 한다.

Log Sequence Number 는 Database 의 correctness 를 위해 매우 중요하게 관리되어야 한다. Log record 에는 변경 사항이 발생하는 경우 이에 관련된 값을 저장하기 위해 발급된다. Log record 의 필수적인 구성요소는 다음과 같다.

1. LSN: 현재 Log record 의 Sequence Number
2. prevLSN: 이전 Log record 의 Sequence Number
3. XID: Transaction ID
4. Type: Update, Commit, Abort 와 같은 수행 작업의 종류
5. Page ID, Length, Offset: 되돌리거나 재실행 시 필요한 저장 위치에 대한 기록
6. Old 값인 before-image / New 값인 after-image

이러한 구성으로 이루어진 Log 에는 logical log 와 physical log 가 있다. 먼저 logical log 는 operation 을 기록하는 것이며 평상시에는 오버헤드가 줄어들지만 recovery 시에 오버헤드가 발생한다. 반면 physical log 는 변경된 지점을 저장하는 것으로 저장의 양이 많지만 recovery 가 매우 빠르고 직접적이다.

Checkpointing

Checkpointing 은 Recovery 에 소요되는 시간을 단축시키며, 직관적인 역할은 “Flush dirty pages”이다. Log 는 복구를 위해 필요한 정보를 기록한 것이다. 변경사항이 생기면 이는 페이지 상에서 변경 작업이 일어나고 변경이 일어난 페이지인 dirty page 에 대한 정보를 Log 에 기록하게 되는 것이다. 이때, dirty page 에 대한 정보를 Database 에

write 를 실시해주면 Log Buffer 의 Log 에 있는 dirty page 에 대한 내용은 더 이상 저장하고 있지 않아도 된다. 즉, Database 에서 가장 최신의 내용을 반영하고 있기 때문에 더 이상 Log record 가 필요하지 않은 것이다. 그렇다면, 주기적으로 dirty page 를 Database 에 반영한다면, Log record 가 필요하지 않는 상황이 온다. 이때, Log record 를 truncate 하여 Crush 상황에 대하여 Recovery 에 걸리는 시간을 줄일 수 있다. 그러나 checkpointing 을 자주하면 Database 의 성능이 하락될 수 있으므로, Recovery 에 걸리는 시간 단축은 Checkpointing 에 의한 성능하락과 유기적으로 고려되어야 하는 사항이다.

Crash Recovery 의 단계는 3 단계로 구성되어 있다.

1. Analysis: Winner Transaction 과 Loser Transaction 을 판별한다.
2. Redo: Repeat history
3. Undo: Loser Transaction 에 대하여 실시

Redo-Winners: Commit 이 정상적으로 처리된 Transaction 에 대한 처리
정상 Transaction, Abort Transaction 모두 정상적으로 끝났다면 Winner 로 판별한다.
Winner Transaction 은 Redo 작업을 통해 복구작업을 실시한다. Abort 가 일어나는 경우에는 Transaction 이 다음과 같이 구성된다.

w(a) w(b) (abort point) w-1(b) w-1(a)

이는 commit 이 되면 종료된다.

Undo-Losers: Commit 이 정상적으로 일어나지 않은 Transaction 에 대한 처리
Undo 는 Transaction 이 정상적으로 commit 되지 않는 경우 동작하며 compensation log record 부터 실시한다. 이는 다음과 같이 구성된다.

w(a) w(b) (crush point) w(c)

이때 undo 는 crush point 부터 w(b) w(a) 부분에 대하여 진행되며 이는 Abort 가 발생한 Transaction 에 대해서도 동일하게 작동한다.

해당 Transaction 에 대한 분석을 마쳤다면 ARIES 알고리즘에 따라 Redo-History 를 실시한다. 이는 우선, Cashed Database 를 재구성하고, 여기서 Loser Transaction 에 대해서 Undo 를 실시해주는 방식으로 진행된다. 이때, physical redo 를 실시하여 undo 를 실시할 때 해당하는 키를 바로 찾아갈 수 있으며 abort 에 사용한 기법을 동일하게 사용할 수 있다. Abort 가 발생하여 Undo 를 실시하는 경우 해당

Transaction 에서는 Compensation(CLR)을 발급한다. 이 CLR 에는 Redo 부분과 Undo 부분으로 구성되며 Redo 부분에는 w-1 가 저장되고, Undo 부분에는 (w-1)-1 이 저장된다. 이 구성을 통해 Crash 가 발생했을 때, Rollback 지점을 정하여 Transaction 을 재실행 시킬 수 있게 된다. 이 경우 Undo 가 Fail 이 되는 경우 CLR 이 계속 발급되어 Log 가 길어지는 문제가 생긴다. 이 문제는 NextUndoSeqNo 를 기록하는 방법으로 설계하면 해결할 수 있다. 이러한 설계는 Undo Recovery 시 진행을 하면 할 수록 Undo 를 진행해야 하는 양을 줄일 수 있는 장점이 있다.

**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)*

해당 문제에 대하여 project4 와 project5 의 concurrency control 의 부분과 연관시켜 생각해보았다. 우선 여러 non-conflicting read-only transaction 을 수행하기 위해 필요한 요소를 정리해보자. project4 의 과제에서는 mutex latch 를 통해 각각의 lock 을 고립시켜 읽기 작업이나 쓰기 작업을 실시했다. project5 에서는 이에 transaction 의 개념을 추가하여 transaction 별로 읽기 또는 쓰기 작업을 실시했다. 이는 buffer layer 에서의 추가 조건도 필요했는데, in-memory 상에서 데이터를 저장하는 buffer 에 mutex 를 이용하여 page 에 대한 접근을 고립시킴으로써 isolation 의 조건을 충족시켰다. 그러나 각각의 transaction 에 대해 모두 isolation 을 시키다 보니 transaction 이 길어지면 길어질수록 transaction 을 실시하는 성능 측면의 이득이 없어진다. 따라서 많은 양의 데이터에 대해 수행하는 경우 시간이 매우 오래 걸리게 되었다.

그러나 여기서 발생한 의문점은 과연 non-conflicting read-only transaction 들만 존재하는 상황이라면, 값의 변동이 없는 상황에 제한하여 생각한다면, transaction 의 단위를 매우 잘게 쪼개서 한번에 더 많은 병렬 처리를 하는 것이 성능을 높일 수 있지 않는가? 였다. transaction 의 단위를 쪼개는 방식은 실제로 구현하지 못하였다. 이의 다음으로 든 의문점은 read-only 상황이라면 mutex 를 통한 isolation 을 구현하지 않아도 올바르게 동작하지 않을까? 이었다. project5 를 실시할 때, buffer manager 단계에서 buffer latch 에 대해 온전한 isolation 을 이루지 못하였다. 결국 이를 완성하지는 못했지만, 한가지 확인한 것은, read 만 하는 상황에서는 isolation 을 고집하지 않고, dirty read 를 허용하면 더 빠른 동작이 가능하다는 것이다. 물론 이것이 Database 의 안전성을 해쳐 적합한 디자인이 될 수는 없지만,

변동사항이 전혀 없는 상황을 고려하면, 이를 허용해도 되지 않을까 고민해보았다. 실제로 Database 의 안전성을 만족시키면서 성능을 향상 시키는 방식에 대해 구현이 가능한 디자인을 완성하지 못해 해당 부분에 대한 가능성만 고려해보았다.

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)