Database System 2020-2
Final Report

ITE2038-11800
2019044711
김찬웅

Table of Contents

Overall Layered Architecture ..... 3 p.

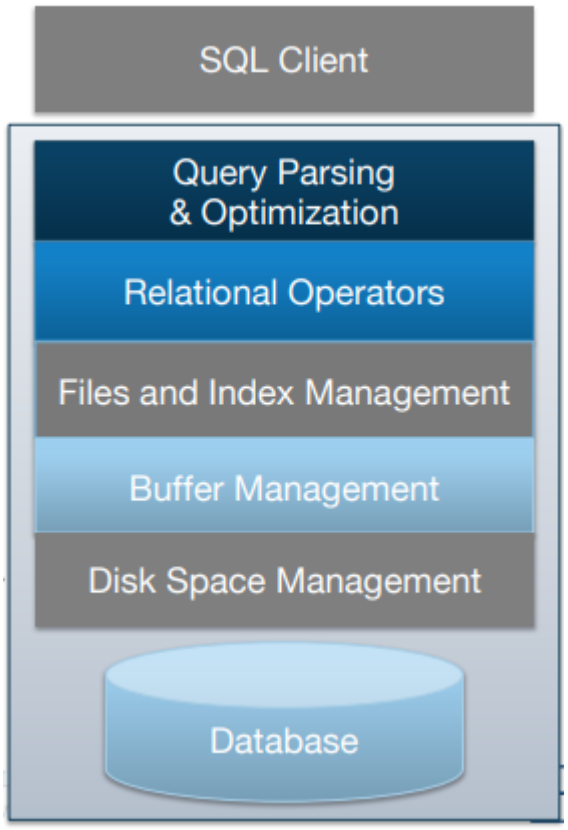
Concurrency Control Implementation ..... 7 p.

Crash-Recovery Implementation ..... 14 p.

In-depth Analysis ..... 18 p.

Overall Layered Architecture

- **DBMS란 DB(Database: collection of structured data)를 관리하는 시스템으로** 사용자와 SQL interface를 통해서 communication을 한다. 또한 이러한 DBMS는 여러 layer로 나눠서 각자의 고유한 일만 하도록 하는 **Layered Architecture**로 이루어져 있다.
- **Layered Architecture**
  - ■ 각자 layer은 자신만의 고유한 일을 한다.
  - ■ Layer들 간에는 api를 이용해서 일을 넘긴다.
  - ■ 장점
    - ◆ **각각의 단계가 할 일만 하기 때문에 자신의 일만 신경 쓰면 된다. (다른 layer들의 결과는 신경을 쓰지 않는다.)**
- **DBMS의 Layered Architecture은 아래와 같이 이루어져 있다.**



- **Query Parsing & Optimization**
- **Relational Operators**
- **Files and Index Management**
- **Buffer Management**
- **Disk Space Management**

- 이제 아래에서 하나씩 설명하도록 하겠다.

- **Query Parsing & Optimization**
  - 어떻게 하면 주어진 query를 빠른 시간 안에 수행할 수 있는지를 맡는 layer이다. SQL interface에서 받아온 SQL code를 machine friendly code로 변환을 하는 역할을 한다.
  - Query optimizer가 하위 layer에게 relation operator로 변경해달라고 요청하고 다시 받아서 optimization을 진행을 한다.
  - 다양한 query plan들을 세우고 각각에 대해서 cost estimation을 진행을 하고 cost가 가장 작은 plan으로 진행을 하게 된다. (다음 단계로 해당 plan을 넘겨준다.)

- **Relational Operators**
  - Optimization을 위해서 parsing된 query를 relational algebra를 이용하여 relational operator로 만들어서 다시 상위 layer로 보낸다.

- **아래의 layer들은 Database의 접근과 관련된 layer들이다.**

- **Files and Index Management**
  - Plan을 기반으로 Database에 접근을 하기 위한 layer
  - Logical한 table을 어떻게 physical하게 구성을 할 것인지 맡고 있는

layer이다. → pagination

- DB의 접근을 위해서 반드시 필요한 layer이며, 나중에 필요한 정보를 찾거나 탐색이 용이하도록 file과 index를 구성을 하는 layer이다.
- 정보를 요약하는 header page, 비어 있는 공간, record가 있는 공간을 잘 구성해야는 그런 part를 맡고 있는 layer이다.

- **Buffer Management**
  - Database paginated architecture에서 Database에 있는 정보들을 buffer에 담아서 빠른 read / write가 가능하도록 하는 layer이다.
  - 사실상 없어도 correctness에는 문제가 없는 layer다. memory (ram)와/과 disk의 속도 차이를 해소할 수 있도록 즉 files and index management와 disk space management의 속도 차이를 해소하기 위한 완충제 역할로, efficiency에 많은 영향을 주게 된다. (거의 필요할 정도)
  - 이 Layer에서는 상위 layer에서 요청한 페이지를 전달을 하게 되는데, 이 때 요청한 page가 현재 buffer에 없을 경우 하위 layer에게서 page를 요청을 하는 역할을 한다.
  - Buffer의 용량에는 한계가 있으므로 요청한 것들을 전부 담아낼 수 없다. 그래서 page replacement policy가 존재한다.
  - Page replacement policy
    - ◆ LRU : Least Recently Used
    - ◆ MRU : Most Recently Used

- **Disk Space Management**
  - Disk Space Management는 Database와 직접 연결이 되어 있는 layer이며, DBMS에서 최하위 layer이고, disk의 space를 관리한다.
  - 주요 기능은 다음과 같다.

- ◆ Mapping pages to locations on disk (page#에서 file#, offset으로)
- ◆ Loading pages from disk to memory
- ◆ Saving pages back to disk & ensuring writes

■ 상위 level에서는 이 layer에게 다음과 같은 기능(api)를 호출한다.

- ◆ Read/write a page
- ◆ Allocate/de-allocate logical pages

Concurrency Control Implementation

- **Concurrency Control**이란 다수의 사용자가 Database에 접근해서 어떤 연산을 수행하더라도 그 Data의 접근을 빠르고 올바르게 결과를 낼 수 있도록 하는 것이 Concurrency Control이다.
- **Concurrency Control**은 실제 연산결과는 순차적 수행된 것과 같이 그리고 Disorderly processing한 것처럼 보이도록 해야 한다.
- **Concurrent Execution의 중요성은 크게 2가지 때문이라고 설명을 할 수가 있다. Throughput argument와 latency argument이다.**
- **The throughput argument :** cpu의 여러 core를 사용해서 multiple transaction을 한다면 throughput이 향상, 즉 성능이 향상되기 때문이다.
- **The latency argument** : 상관이 없는 데이터를 접근하면 다른 실행 중이던 latency에 영향을 받지 않기 때문에 또한 성능이 향상된다.
- Concurrency Control 및 Database에서 중요 개념은 바로 Transaction인데 이는 다음과 같다.
  - Transaction : Collection of operations that form a single logical unit
    - A sequence of many actions considered to be one atomic unit of work
    - 사용자 기반 즉, user defined atomic unit of work이다.
- Transaction 단위로 다양한 operation들을 수행을 할 때 고려해야할 점은 디자인이 ACID를 보장해야 한다는 것이다. ACID는 다음과 같다.
  - ACID Guarantee
    - Atomicity : All actions in the Xact happen, or none happen.
    - Consistency : If the DB starts out consistent, it ends up consistent at the end of Xact

- ◆ Isolation : Execution of each Xact is isolated from that of other Xacts
- ◆ Durability : If a Xact commits, its effects persist.

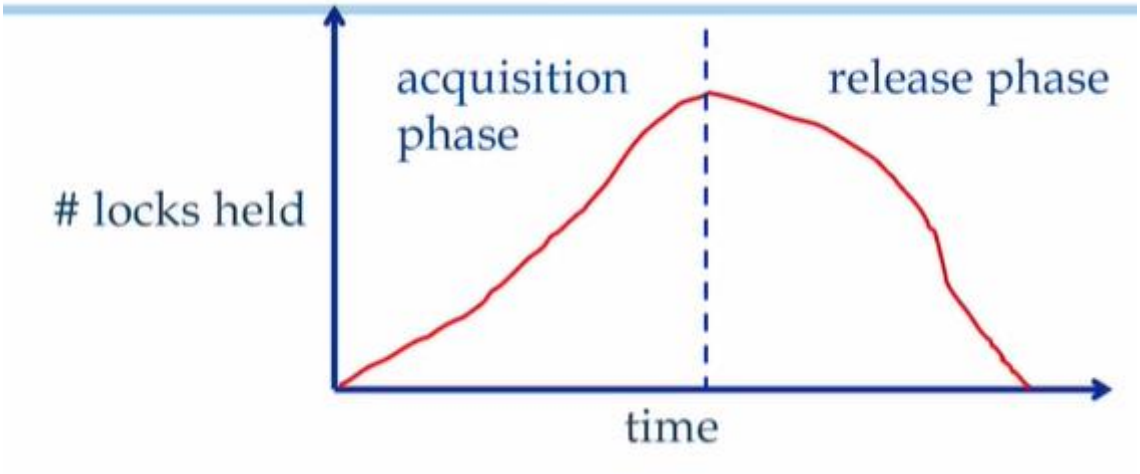
- **** Isolation Property (Concurrency) : 여러 사용자가 Xact를 수행 중이더라도 혼자 돌고 있는 것처럼 해야 한다는 것. (혼자 serial order 수행 중이라고 느낄 수 있도록)**
- Serial Schedules
  - ■ 정의 : transaction set을 펼쳐서 하나씩 수행한 것들
  - ■ Schedules are equivalent if they
    - ◆ 같은 transactions 포함
    - ◆ 각 transaction들의 action들의 순서가 동일
    - ◆ 둘의 schedule의 실행 후 결과의 상태가 동일한 상태
  - ■ Schedule S is serializable if
    - ◆ S가 any serial schedule과 equivalent할 때
- Conflicting Operations
  - ■ 정의
    - ◆ 서로 다른 transaction에서 수행
    - ◆ 같은 object를 대상으로 수행
    - ◆ 둘 중 적어도 하나가 write일 경우
- **CC (Concurrency Control)는** 크게 2가지로 나뉘는 데 Pessimistic CC와 Optimistic CC로 나뉘며 PCC, OCC라고 한다. 이중 PCC의 P는 pessimistic 비관적이라는 뜻이고 OCC의 O는 optimistic 낙천적이라는 뜻인데 conflict에 대해서 pessimistic, optimistic이라는 것이다.
  - ■ 1. PCC (Pessimistic CC)

◆ Conflict가 일어날 것이다.

◆ 미리 조치를 취함 – **Locking (Conflict detecting)**

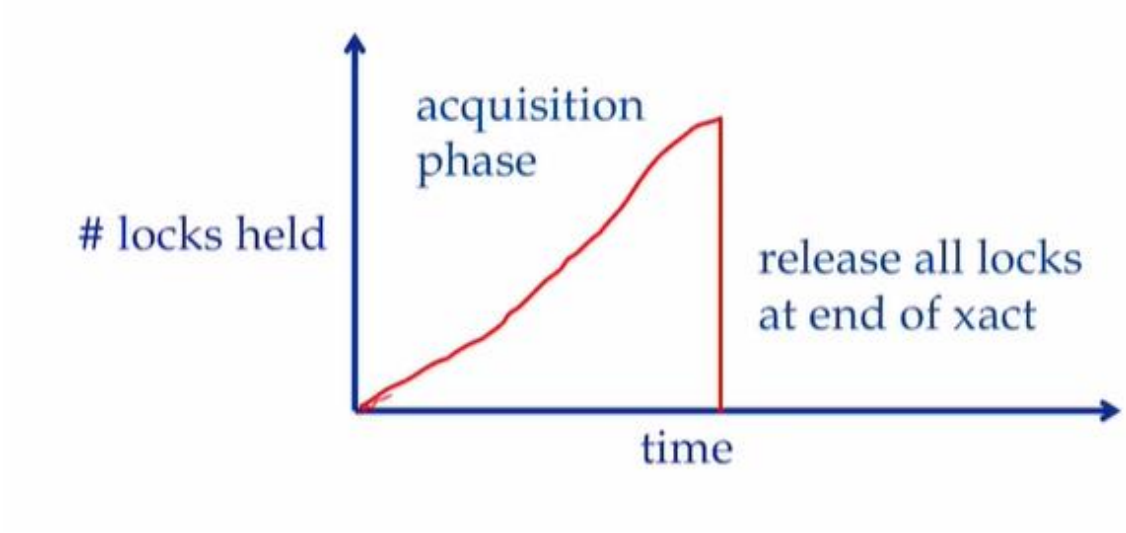
◆ **2PL (Two Phase Locking) – Locking 기법**

- 어떤 output schedule s에 대해 그리고 s의 모든 transaction에 대해서 어떤 ql_i이 첫 ou_i step 뒤에 나오면 안된다는 기법을 말한다. q, o는 (r 또는 w)
- **Lock object contains - <rid, S / X (Lock mode) , TXN_id>**



➔ **Cascading abort 발생 가능성이 있다. (중간에 abort시)**

◆ **Strict 2PL (Cascading abort방지)**



- **모든 transaction이 끝나면 그 때 lock들을 release하는 것 (abort commit 결정 후 unlock하는 것)**

◆ **Lock management**

- Lock을 관리하기 위한 module
- Hash table & linked list를 이용, rid를 key로
- Conflict lock
  - 존재하지 않으면 – 바로 request lock에 넣어주고 proceed
  - 존재하면 – wait queue에 넣음, sleep 시킴

◆ **Deadlocks**

- **transaction들이 서로가 서로를 붙잡고 있는데 cycle이 생길 때**
- **방법**
- **1. Prevention / Avoidance**
  - 누가 붙잡고 있는지, 우선순위가 누가 더 높은 지
  - T_i가 T_j보다 뒤에 들어온 경우
  - **Wait-Die** : 만약 T_i가 우선순위가 높으면 T_j를 기다리고 그렇

지 않으면 T_i abort

- **Wound-Wait** : 만약 T_i가 더 높으면 T_j를 abort 아니면 T_i기다림
- **2가지 방법 모두 우선순위가 높아지는 방향 또는 낮아지는 방향으로만 wait하므로 Deadlock이 발생하지 않음**
- 장점 : 안전한 wait cycle을 만들기 때문에 반드시 수행 가능
- 단점 : 생기지 않는 상황에도 priority를 자꾸 abort를 시키는 상황이 발생

- **2.Detection and Resolution**
  - **Wait-for graph를 생성시킴**
    - **단점 : graph traversal을 자꾸 해야함**
    - **→ 주기적으로 측정해보자 (adaptive하게)**

◆ **Locking Granularity**

- Fine granularity : high concurrency, high locking overhead
- Coarse granularity : low locking overhead, low concurrency
- New Lock Modes
  - Concurrency 향상을 위해서 도입됨.
  - **IS**
  - **IX**
  - **SIX : S & IX (Select for update)**

■ 2. OCC (Optimistic CC)

◆ Conflict가 안 일어날 것이다.

- ◆ 3가지 phase
- ◆ 1. Read phase
  - Record 읽어서 private workspace에 씀 (DB에서 copy)
- ◆ 2. Validation phase
  - **CSR 검사 (conflict serializable)**
  - **RS(t), WS(t)으로 validation검사**
    - **BOCC : t_j를 이전의 모든 t_i와 비교, 처리가 끝난 operation들을 대상으로 현재 transaction과 비교를 해서 conflict를 찾아내는 방법**
    - **FOCC : t_j모든 active transaction set과 비교, 즉 다음에 수행될 operation들을 대상으로 비교하고 conflict를 찾아내는 방법이다.**
- ◆ 3. Write phase
- ◆ Multiversion CC (MVCC)
  - **Snapshot Isolation**

- **Concurrency Control은** File and Index Management, Buffer Management, Disk Space Management 단계에 걸쳐서 구현이 된다.
- Transaction manager에서 schedule을 처리를 해주고 transaction 순서를 정해주지만, 이 3가지 layer에서도 추가적인 처리과정이 필요하다.
- **PCC의 경우**
  - ■ Lock manager에서 conflict여부를 미리 확인을 하고 record에게 접근을 한다.
  - ■ 1. Files and Index Manager에서 read / write API를 호출 (PCC이므로 미

리 conflict여부를 검사하게 되는데 만약 conflict가 존재한다면 기다린다.)

- 2. Buffer manager에서 buffer / page lock들 관리 여러 transaction이 한꺼번에 버퍼에 접근하는 것을 막기 위해서이다.
- 3. Disk space manager에서 lock이 보장된 것들에 대해서 disk(database) read/write 수행

- **OCC의 경우**
  - **Commit 단계에서 conflict 검사** – FOCC / BOCC

- **실제 나의 project code**
  - 해당 내용을 project 4~5에 걸쳐서 구현을 하게 되었다.
  - **우선 PCC**의 방식으로 구현을 하게 되었다.
  - Deadlock 탐지는 dfs알고리즘을 이용해서 cycle을 찾는 방식으로 구현을 하였다.
    - ◆ 하지만 매번 lock이 생성 될 때마다 수행이 되기 때문에 이 점이 성능하락의 원인이 될 수 있다는 생각을 하게 되었다.
  - 수행과정은 다음과 같다.

    1. Transaction이 시작
    2. lock_acquire들이 수행

       A. deadlock중간에서 탐지

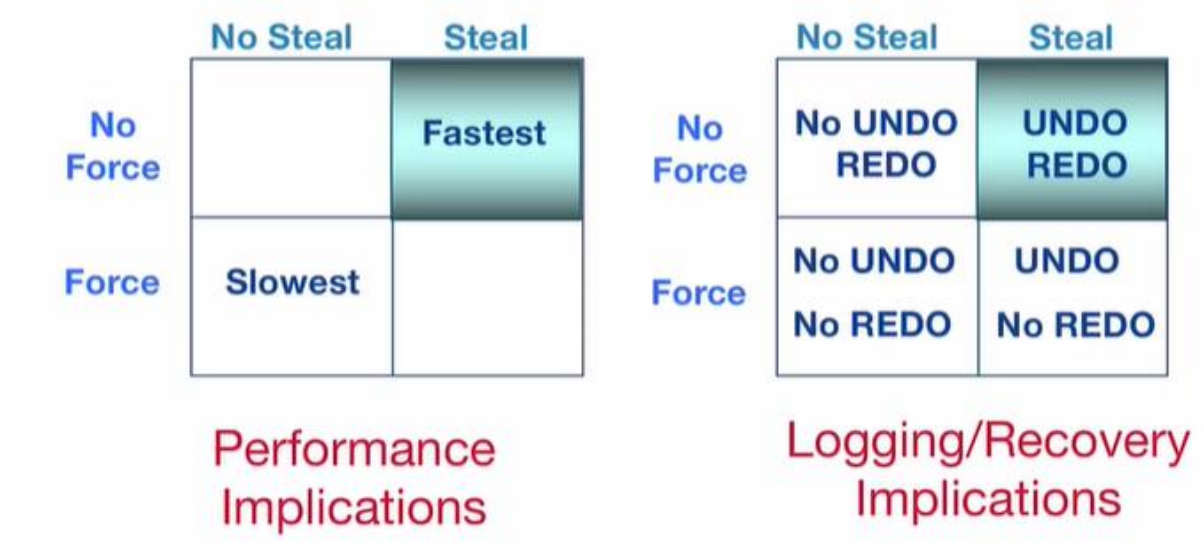
    3. lock이 성공적으로 얻어졌고, ACQUIRED이면 각각의 연산을 수행한다.

Crash-Recovery Implementation

- Recovery : 어떤 종류의 failure가 일어 났던지 간에 결과가 항상 올바르게 있어야 한다는 것
- Recovery Manager : ACID properties 중 Atomicity & Durability를 관여하고 있고 이를 지키기 위해 logging을 하고 있음
  - Atomicity : Transactions may abort ("Rollback")
  - Durability : 중간에 꺼져도 올바른 상태로 남아 있어야함 (올바르게 commit한 결과가 오직 DB에 반영) 진행 중인 것이 종료가 되었으면 rollback을 시켜야 한다.
    - Redo 필요
  - Undo / Redo : transaction이 제대로 끝났는지 끝나기 전에 종료가 되었는지에 따라 나뉨
- 가정 : Strict 2PL, Updates are happening in place
  - Important component : Buffer management가 Buffer pool에 어떤 정책을 가지고 있는지.
    - Force policy : transaction 종료 시 변경 된 내용을 무조건 DB에 반영되는 policy → syncing
      - Durability가 지켜 짐, redo logging이 필요하지 않음
      - Performance에 안 좋은 영향이 있음
    - No STEAL policy : transaction이 완전히 끝날 때 (commit) 까지 buffer에 있는 내용이 절대로 disk에 반영 (write) 이 되지 못하도록 하는 policy
      - Atomicity가 지켜진다, undo logging이 필요하지 않음
      - Performance에 안 좋은 영향이 있음

◆ Steal/No-Force (성능을 위해서)

- 성능이 향상이 되나 위의 policy 보다는 복잡함
- 추가의 recovery action이 필요함
- No Force (Durability와 연관) : Durability를 보장해야 하므로 Redo action이 필요함 (새로 들어온 값을 기억해야 하므로)
- Steal (Atomicity와 연관) : Atomicity를 위해서 Undo가 필요 (old값을 기억해야 하므로)



■ 아래에는 필요한 function & component들이다.

■ Logging

◆ Log : An ordered list of log records to allow REDO / UNDO

- Log record contains
  - <XID, pageID, offset, length, old data (for undo), new data (for redo)> 기본적으로 들어가야할 것들
- 페이지를 변경할 때 마다 log가 발생함
- Write-Ahead Logging Protocol (log를 write하는 규칙)

        - Must force the log record for an update before the corresponding data page gets to the DB disk
            - Atomicity를 위해서 – Undo
        - Must force all log records for a Xact before commit
            - Durability – Redo
    - WAL기법으로 log를 쓴다고 했을 때에는 LSN이 발급이 됨
        - LSN (Log Sequence Number) : pageLSN <= flushedLSN 이것은 잘 지켜줘야 함 아니면 recovery 불가

- Log Records
    - 변경하는 작업에 대해서 발급이 됨
    - Fields
        - LSN : 현재 record의 log sequence number
        - prevLSN : 이전 log, 없으면 NIL (NULL)
        - XID
        - Type
            - Update 등
        - 아래는 필수정보이다.
        - pageID, length, offset : 장소
        - before-image : 이전의 값 (old)
        - after-image : 덮어 쓴 내용 (new)
- trasaction table
    - contains XID, status, lastLSN

- Dirty Page Table
  - Dirty로 만든 첫번째 log record를 담고 있음
  - 너무 오래된 것들은 최우선적으로 반영을 시키기 위해 만들어져 있다.
- Checkpointing
  - Flush dirty pages
  - Log – 복구를 위한 정보, dirty page를 disk에 반영이 되었다면 log를 truncate 시켜도 됨
  - Log를 truncate 시키지 않으면 복구하는데 매우 많은 시간이 걸림, recovery time을 줄일 수 있음 – foreground Xact와 trade off

- 요약
  - Recovery는 크게 2가지 동작으로 볼 수 있다. (Undo / Redo)
  - Log Records, Data pages, Xact Table, Dirty Page Table, Log tail, Buffer pool로 Recovery (Redo, Undo)가 동작하고 있다.

In-depth Analysis

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

- 나의 project 코드를 기반으로 설명을 하자면 현재 어떠한 operation들이 들어오더라도 isolation의 보장을 위해서 buffer의 mutex, page latch를 거쳐야만 어떤 operation (read / write)가 수행이 가능하도록 또 lock의 mutex를 걸어주어서 isolation을 보장을 하는 코드로 만들어 주게 되었다.

  그러다 보니 conflict가 애초에 만들어 지지도 않는 상황인 many concurrent non-conflicting read-only transactions에서도 또한 이러한 mutex들, latch들을 불가피하게 거치는 상황이 만들어지게 되었고 isolation을 보장을 해주기 위한 것들인 mutex가 성능하락의 주원인이 되었다. 즉, 많은 양의 read operation을 수행을 했을 경우에는 현재 내 project code에서의 mutex들 때문에 (page mutex, buffer mutex, lock mutex 등)이 성능하락으로 이어지게 되었다.

- 아직 project5의 구현이 완벽하지 않아서 많은 양의 read-only transaction에 대해서는 생각해본 design에 대해서 적기만 하겠다. 모든 transaction이 read-only임을 보장을 한다면 mutex가 없어도 가능하다고 생각을 하게 되었다. 그래서 read-only임이 보장이 된다면 mutex를 제거해서 빠르게 buffer를 통해서 해당 operation을 수행하는 design을 고려해 보았지만 Database의 안전성과 수행시간의 tradeoff가 어느정도 일어나게 될 것 같다. 하지만, 오직 클라이언트가 수행할 수 있는 operation이 read 뿐이라면 이러한 design도 (mutex를 제거해서 성능을 올리는 방식) 또한 성능측면에서 나쁘지 않다고 판단을 하였다.

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

- 아직 project5구현이 완전하지 않아서 crash-recovery에 대해서 구현을 하지 못했지만 성능 측면에서 문제점과 생각해본 design에 대해서 적어보도록 하겠다.

- 많은 수의 write operation으로 인해서 많은 양의 log들이 생기게 되고 그

로 인해서 나중에 abort가 들어올 경우 많은 양의 log들을 다시 undo를 하는 상황이 발생하는 데 이러한 점이 가장 성능측면에서 악영향을 미칠 수 있을 것으로 생각이 된다. Write-only transaction이 많이 수행이 돼서 많은 양들의 log가 발생하기 때문에 log가 길어지면 공간의 측면에서도 낭비이지만, 나중에 복구 operation이나 abort가 되었을 때 그 많은 log들을 되돌려야 하고 이는 해당 undo되고 있는 record에 대해서 다른 transaction들의 접근이 undo operation이 끝날 때까지 불가능하기 때문에 성능측면에서 굉장한 악영향이 될 것으로 보인다.

- 디자인은 checkpointing을 이용하면 이러한 성능하락을 줄일 수 있다는 생각을 하게 되었다. Checkpointing을 이용해서 dirty page들을 flush시켜서 log의 양을 줄일 수가 있기 때문에 이러한 log로 recovery를 하는 데에서는 성능이 향상될 수 있다. 하지만 너무 checkpointing을 자주 하게 되면 오히려 disk (Database)의 I/O 횟수가 늘어나게 되고 disk I/O에는 평균적으로 봤을 때 buffer I/O에 대해서 시간이 매우 많이 걸리기 때문에 checkpointing을 자주하는 행위는 오히려 성능하락의 주원인이 될 수 있다. Checkpointing을 이용해서 recovery time을 줄이는 design을 생각을 해보았지만 project5구현이 완전하지 않아서 crash/recovery에 대한 구현을 하지 못하였기에 이러한 부분에 대해서 design과 가능성만 생각해보았다.