

Database System 2020-2

Final Report

ITE2038-11800

2019007901

완

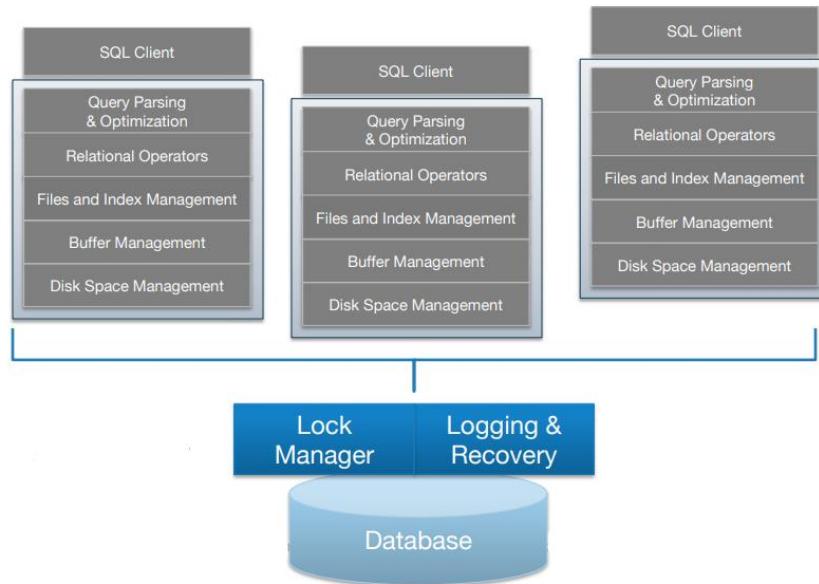
# Table of Contents

Overall Layered Architecture .....	pg.3
Concurrency Control Implementation .....	pg.7
Crash Recovery Implementation .....	pg.11
In-Depth Analysis .....	pg. 14

## I OVERALL LAYERED ARCHITECTURE

## 1.1 Introduction

As we know, database systems are built based on few several layers and each layer handles different workloads. In this whole semester, we learned how a database system works in a very detailed aspects for every layer.



During this whole semester, we had to implement all these layers by ourselves to build our own database system. Each function is divided separately into a few layers and calls upon each other from different layer. Furthermore, call flow will be optimized in situation where a function from another layer will not be called unless needed. My overall layered architecture is as below:



## 1.2 Layers and Its Definitions

- main.cpp

This layer handles all the inputs and commands from the user such as insert, find, delete, update and open table. This layer also initializes the lock table and the buffer so that both of them are initialized during the start-up of the database. In addition, the transaction id is initialized by calling *trx\_begin()* before user inputs any command. After all the initialization are completed, the program is ready to accept any commands from the user. Every time a command is inputted, this layer will call the API functions for various operations from the lower layer, *dbapi.cpp*.

- dbapi.cpp

I would say this layer is the most important layer; the heart of our database system. This is where all the important APIs of the operation on the database are defined such as insert, delete and others. The call flow of the functions varies for some of the APIs in here. For example, insert, delete, find and update operations calls the functions from *bpt.cpp* layer directly while *open\_table*, *close\_table*, *shutdown\_db* are mostly managed by the buffer layer.

- bpt.cpp

Then we arrived at the backbone of our database system. This layer is where all the data files that were requested by the user are handled. Our database is built based on B+ Tree so this layer will handle the insertion, deletion, find, and the modifications of the tree after every operation.

- buffer\_manager.cpp

The buffer layer will handle all the file requests from the *bpt.cpp* layer to the *disk\_manager.cpp* layer. In our database system, we implemented a buffer system in project 3 so that the requested data from upper layer will be cached in *buffer\_pool* defined by this layer. This will greatly reduce the I/O workload between the program and the disk. Plus, in the 3<sup>rd</sup> project is where we had to consider dealing with multiple disks (*open\_table* multiple files with unique id) instead of just a single disk.

- disk\_manager.cpp

All the I/Os between the program and the disk are handled by this layer. All the reads and writes operations depends on this layer. For instance, when the upper layer requesting a page from disk and cache miss occurred in the buffer layer, this layer is responsible to fetch that requested page from disk and pass it into the buffer. Same goes when the buffer had to evict a page from the buffer pool, this layer will write any changes made on the page and write it directly to the disk. In a nutshell, the buffer layer and this

layer communicates a lot to ensure smooth process of I/O between the disk and the program.

- lock\_manager.cpp

In project 5, we had to implement a database with concurrency control features and considers multiple concurrent access to the disk. Hence, we implemented this lock layer to ensure the atomicity of every operations requested from multiple users. We also introduced the transaction system where each operation set are separated by different transactions. To ensure all the transactions holds the ACID property and are isolated from the others, we implemented the mutex lock that we learned in project 4 for 2 operations, find, and update (based on the project specifications). We put this layer under disk manager layer so that every time the disk layer attempts to read/write a page, this layer will supervise the concurrency of the I/Os.

- log\_manager.cpp

Finally, at the final layer, we have a layer which will keep track all the I/O operations that was occurred throughout the whole session and save the logs to several special external files, specifically *logfile.data* and *logmsg.txt*. These 2 files will be used to handle the recovery process and undo all the changes when our database experiencing a crash or system failure. We can think of these 2 files as our back up files and when the system crash or aborted, the program will read the log files and undo all the operations and changes that has been made, revert them to the state before the crash occurs.

In a nutshell, database management system is a single huge system that is managed by separate layers and each layer abstracts the layer below. This will help the developers of DBMS to manage the complexity of operation (even a single insert operation is so complex! With I/Os, logging, concurrency, tree modification etc.) rather of managing the call flow in a single source file. Besides, not only DBMS, this is how most applications are built and this course gave us an opportunity to experience the learning and dissection of each layer one by one.

## II CONCURRENCY CONTROL IMPLEMENTATION

## 2.1 Introduction

In project 5, we had to implement concurrency control onto our database system. Hence, we will have to implement the transaction (TRX) system so that our database system can handle request from multiple users while providing the concurrency between those requests. Our transactions also need to hold the **ACID** properties whereas:

- Atomicity: Either the entire TRX runs successfully at once, or the TRX itself does not run.
- Consistency: Consistency of reading and writing values is guaranteed.
- Isolation: The transaction is isolated with other transactions.
- Durability: The transaction must handle any error operations in any situation.

Isolation has several types, such as serializable, which means it is completely independent, and users can read uncommitted data to the extent that the user feels the system is independent from other users. The meaning of serializable transaction is that when multiple TRXs are executed at the same time and when they are processed in sequence, the same result is obtained.

This can be satisfied by forcing the order of data access through lock. Hence, our implementation of concurrency control **depends heavily on lock\_manager.cpp layer** which will handle the request for locks from particularly 2 operations, find and update (representing read and write respectively from what we learned during CC lectures). First, we define the struct of lock *lock\_t* which holds the information of the table id, key, lock mode (SHARED or EXCLUSIVE) and transaction that is holding this lock. After that, we implement transaction struct *trx\_t* which stores info such as the transaction id and the list of keys (key table) that are held by this transaction.

```
struct lock_t {
    int tid; //table_id of lock
    int key; //key of lock
    int lock_mode; //0=SHARED or 1=EXCLUSIVE
    trx_t* trx; //transaction for this lock
};

struct trx_t {
    int trx_id; //unique transaction id
    std::string state; //state of trx, either WORKING or IDLE
    std::list<lock_t*> lock_list; //lock list of the transaction
};
```

Now we are ready to implement the APIs for this lock layer. Every time the function (either find or update) is called by the upper layer, it will request a lock from the lock layer and that lock will be saved into the lock table of the corresponding transaction.

## 2.2 Components for Lock Layer

The lock layer will handle several operations that are related to lock and transaction managing such as requesting lock, releasing lock, starting and committing a transaction.

- `trx_begin()`

Every time a new transaction is starting, this function will be called first to initialize some few important components of the transaction such as transaction id and its lock table. The lock table is structured by using an unordered map from C++. For transaction id, different transaction has their own unique id and these IDs are incremented every time they are created (which means larger ID = latest transaction). Finally, we save this process of transaction initialization into our log file.

- `trx_commit (int trx_id)`

After all the operations of a transaction is done, we will commit all the changes done by the transaction into disk by calling this function. Remember that since we are required to implement a pessimistic concurrency control, specifically **Strict 2PL**, the transaction will release *all* the locks contained in this transaction's lock table at once. The transaction will be erased from the transaction list and unlocks its transaction mutex. Again, we save this whole commit operation into our log at the end.

- `int trx_abort(int trx_id)`

This function is called when the user cancels the operation before committing the changes to disk. Since the implementation should be similar to commit, hence this function will call the commit function. But before that, the transaction should undergo the rollback process where all changes made are reversed to the point from the previous save point (commit/begin). This is where we access our log file to inspect what changes has been made by this transaction, redo all the operations, and finally commit the transaction.

These are all 3 main function that will handle the management of the transactions in our database system. Now, we will inspect how the locks are managed by this layer.

- lock\_t\* lock\_acquire(int table\_id, int64\_t key, int trx\_id, int lock\_mode)

A lock will be requested by the upper layer (specifically *bpt.cpp* layer) whenever an operation is being requested by the user, either find or update. As mentioned before, since we are implementing Strict 2PL CC, we will have 2 lock modes, SHARED and EXCLUSIVE for both find and update operation respectively. When a lock is requested by a transaction, the lock components such as table id, key and its lock mode are defined by this function. Then the lock will be inserted into lock table of the requesting transaction and its lock mutex is locked.

- int lock\_release(lock\_t\* lock\_obj)

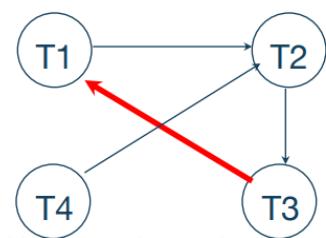
This function is called when we are committing a transaction (recall: all locks released during commit). The process is fairly simple as acquiring lock. Find the lock in the transaction lock table, delete it, and finally unlock its lock mutex.

## 2.3 Deadlock Detection

Deadlock can be occurred when multiple transactions are waiting for each other for the lock. For example, T1 is holding lock of table A and waiting for lock of table B held by T2. But at the same time, T2 is also waiting for lock of table A from T1. Although in our database system, we haven't able to implement this deadlock detection but I do have some few ideas to solve the deadlock problem.

As we learned in lecture before, we can create a “waits-for” graph which represents a directed graph where transaction as vertex and edges as the “wait for lock” from other transaction. For example, if T1 is waiting for lock in T2, then vertex T1 is pointing to T2. Graph can be represented by using list (as we learned in data structure course).

So how is deadlock detection works should be fairly simple. Every time after a transaction calls *lock\_acquire* function, the transaction will be added to the graph and points to the other transaction if the requested lock is held by other transaction. After that, we run a “cycle detection” algorithm (source: <https://www.geeksforgeeks.org/detect-cycle-in-a-graph/>) and if the graph is not acyclic, we simply abort the very latest transaction that we inserted in the graph by calling the *trx\_abort* function and delete that transaction from the graph.



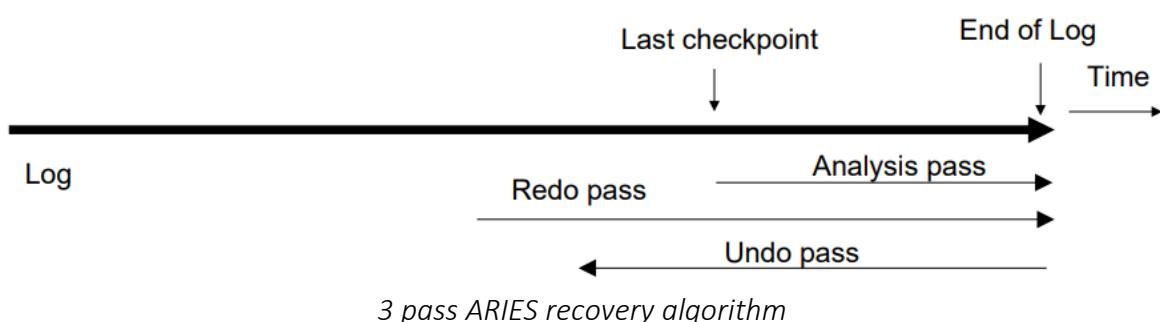
Algorithm detects cycle, the most recent transaction in the cycle (T3) will be aborted.

### **III CRASH RECOVERY IMPLEMENTATION**

### 3.1 Introduction

Whenever a database system crash or encountered an unexpected problem, that database must perform a recovery where it reverts into its original state before the crash. In other words, the database ignores all the changes done after the crash and redo all the transaction that has been done from a specified checkpoint to the point before crash. ARIES recovery uses a system called **log sequence number (LSN)** to keep track all the operations that has been done by the users and to identify what updates that has been applied to the database page. So, each page in database will have a **PageLSN** which indicates LSN of the log records in which the effects are reflected on that page. This PageLSN is also required during redo pass to prevent any unnecessary repeated redo.

In this project, we are required to implement ARIES logging algorithm which involves 3 passes. Analysis pass will determine which transaction operation to redo and which pages are dirty at the time of crash. After that, Redo pass will redo all the transaction operations from the RedoLSN point until the point before the system crashes. Finally, Undo pass will rollback all the incomplete transactions which their abort operation was not completed.



The main layer that handles the logging and recovery features of course would be the **log layer**. This layer is placed right above the disk so that every time an I/O between the program and disk is executed, this layer will record and save every execution for logging and recovery purposes (using write-ahead logging WAL). Hence, we define the struct of log which contains several information about a log like its LSN number, the transaction ID, the size of log, table ID and others as below.

```
typedef struct log_t{
    int64_t LSN; //the LSN number
    int64_t prev_LSN; //previous LSN number
    int trx_id; //Transaction ID corresponding to log
    int log_type; //types of operation which reflects the log activity
    int tid; //Table id of corresponding log
    int pnum; //page number where the log operation was done
    int offset; //Start offset of the modified area within a page.
    int data_len; //length of changed data
    char old_image[120]; //old data before operation
    char new_image[120]; //new data after operation
}log_t;
```

## 3.2 Components for Log Layer

Since this layer handles the logging and recovery of our database system, we will define a few variables and functions such as log creation, log saving and recovery function. Some core variables of course would be the pathnames of the log file and the log message file itself. We also defined a vector struct to hold the log buffer so that whenever a log is created, that log isn't saved into log file first, but rather into this log buffer and finally flush into log file after the transaction is committed.

- `create_log(int trx_id, int log_type, int tid, int pnum, int idx, char* old_image, char* new_image)`

This function is called every time an operation is started by the user. The function will initialize a *log\_t* and all its variables accordingly to the parameter given. *Last\_LSN* variable will be updated to be equal to this log LSN and will keep incremented when the next log is created. Finally, the log will be added into the log buffer *log\_buf*.

- `void flush_log()`

This function is to flush all the logs in the *log\_buf* into the specified "logfile.data" file where all the logs content will be saved. When a dirty page is about to get flushed into the disk, the program will first acquire an exclusive lock onto the page and then flush the page to the disk. After that, this function will be called to indicates that the logging of the changes made onto that page has been successfully saved into the log file. Finally, the lock on that page is released.

- `void print_log()`

While this function is pretty self-explanatory, it will write the content of the log buffer into the log message file for the user to check their saved changes and activity throughout the session.

- `int recover(int flag, int log_num, char* log_path, char* log_msgpath)`

This function is where the three-pass algorithm of ARIES recovery is defined. This function will be called every time the program is starting and during the initialization of the database, the system will undergo this recovery function. This will simulate how the recovery is done when a database system is crashed and restarted. Meanwhile, during the first run of the program, where the log file is not created yet, the recovery process would be skipped during database initialization since the log file is empty.

**IV**

## **IN-DEPTH ANALYSIS**

## 4.1 Workload with many concurrent non-conflicting read-only transactions

- Introduction

In our current implementation of database system throughout the semester, we only considered a small number of operations at a time. Now, let imagine some real DBMS cases, where our database has to deal with a huge number of reads at once. For example, in this analysis, having to execute 100,00 reads to distinct records each, simultaneously. Since all the operations are read-only and each of them accessing different records, concurrency should not be a problem since the records would not have any changes made and we are just reading records as it is.

Additionally, we also can remove the consideration of system crashing during the process of read transactions. Since we are doing read-only, the data of record requested by the users should stay the same whether the record is being read before or after the crash occurred. Our way of indexing records in our implementation (by using B+ Tree) already helped with the performance of searching the record inside the database. Hence, the performance and resource demands such as RAM and CPU workload would be our major concerns in this scenario.

- Performance-related Problems

Now, let's look at how a huge number of read-only transaction will affect the performance of our current implementation of database system. In our current DBMS, whenever a record is requested by user, the upper layer will request the corresponding page containing the requested record from the buffer and not directly from disk. Only when cache miss occurs, the buffer layer will fetch the page from disk and push it to the upper layer. As mentioned before, now let's say we are doing 100,000 reads at once. There are 2 main problems related to buffer that would occur. First, our current buffer pool obviously could not hold all those pages at the same time as it would require a tremendous amount of RAM memory. Second, the amount of cold cache misses will be very huge, hence increasing the number of I/O processes between the buffer and the disk.

Another problem that could lead to performance degradation is the CPU demand. Our current DBMS implementation only handles operation in a serial manner (i.e., using a single thread). But if we are required to handle a tremendous amount of read transaction simultaneously on a single thread, that would cost a lot of CPU performance hence increasing the time to complete the whole operation.

- Solution and Design

Now we understand that our main concerns are the resource demands of RAM and CPU. We will look into details of how these problems should be solved and a descriptive design that follows our planned solution.

For the first problem, where we were concerned about the buffer space size, one of the naïve solutions is that we increase the size of our buffer so that more pages can be cached at once. Increasing cache space (buffer) and additionally implementing prefetch will reduce the number of cold cache miss, hence reducing the I/O workloads between the buffer and disk. But do keep in mind that too much cache space would give us the opposite effect as it will then increase the access time of the pages in the cache. Another suggestion would be implementing a layered cache system which is similar of how caches are implemented in operating systems in general.

The next problem which is related to CPU cost can be solved by creating multiple threads to handle a batch of read transaction each. In other word, we executing the whole operations in a parallel mannerism. In this way, we can divide the workloads to multiple CPU cores and reduce the time of the operation. For starters, we will try implementing 10 threads in our implementation and observe the time. Since we are running the program in parallel, we should also take account of the atomicity of every thread so that they do not clash with each other. The example of where 2 operation are clashing is when T1 and T2 attempting to read record R1 and R2 respectively, if T1 accessed the *find()* function first but T2 also accessing at the same time, T1 might give out the result of reading R2. Hence, we have the lock implementation to avoid this kind of misread.

## 4.2 Workload with many concurrent non-conflicting write-only transactions

- Introduction

Unlike read-only transaction, now we have to consider committing a huge number of record writes at once. For example, in this case, we are attempting to execute 100,000 writes operations at once. This time, we will need to consider the event where our DBMS crashes in the middle of writing a huge number of records. up until our current implementation, a simple logging and recovery would do the job since we are doing write in a small amount at a time. but for a huge amount of write, our recovery process will encounter some performance problems.

- Performance-related Problems

We will examine how our current implementation of recovery process would handle this kind of scenario. The first problem would be low time efficiency. Since our current recovery algorithm works by redo and undo every log one by one, that works fine if we only have to recover a few logs. But what happens if system crashes in the middle of doing 100k writes? Then we would have to undergo the redo and undo process of a lot of logs and that would take a very long time to complete since our recovery is executing one by one.

The second problem is related to how we are going to log those tremendous amounts of workload into our log file. Similar to insufficient RAM memory problem that we discussed in read-only transaction before, our log layer does not write the log directly to the log file but instead keeps them in the log buffer first and flush them only when the transaction is completed. Now, can we ensure that the whole write log can be stored in log buffer (which is in RAM memory) all at once? Furthermore, flushing hundred thousand of logs to the log file at once serially also could be time inefficient. A simple example to rephrase this situation is by imagining creating a back up for the whole of our computer hard drive. We know it does not complete in a matter of seconds.

Finally, we also may be concerned about the log file size. The main idea is, the bigger the log file, the longer the recovery process would take. Then we should ask a few questions for this problem. Is it really necessary to log every single transaction since the beginning of our database initialization? What about the write transactions that are safely committed to disk without encountering any crashes in the middle? To answer this, we will discuss about the solution of the problems in the next discussion.

- Solution and Design

One of the solutions that can solve the slow recovery process is by implementing a rapid recovery, where the recovery process of the DBMS is very important in this kind of situation. There are 3 kinds of rapid recovery which were implemented by Oracle and those are parallel recovery, fast-start recovery and transparent application failover (source:

[https://docs.oracle.com/cd/A87860\\_01/doc/server.817/a76965/c28recov.htm](https://docs.oracle.com/cd/A87860_01/doc/server.817/a76965/c28recov.htm)). For now, we will be focusing on parallel recovery. Though we will not go into detail of how this parallel recovery works, from the term alone, my vague idea was to implement multi-threaded recovery so that a huge number of logs can be divided and handled by each working thread. This way, several processes simultaneously apply changes from redo log files improves the time efficiency of recovery process.

In addition to the parallel recovery implementation, we also can implement partial logging to solve the I/O workload between the log buffer and log file. As mentioned before, imagine logging 100k of write transactions and system crashes in the middle of write and recover the redo undo for those transaction. This will incur a heavy amount of I/O between log buffer and log file, and log file and disk. Hence, partial logging will help improving the performance of recovery process. That means, instead of logging the whole operation from the beginning of the program, we check if a batch of write transactions is successfully committed to disk safely without any crash occurred in the middle of operation. If it is, we can overwrite our log file with the next batch of write transactions. By this method, we can reduce the amount of I/O of accessing log file so that only necessary records that needs to undergo redo undo process are accessed during the recovery.