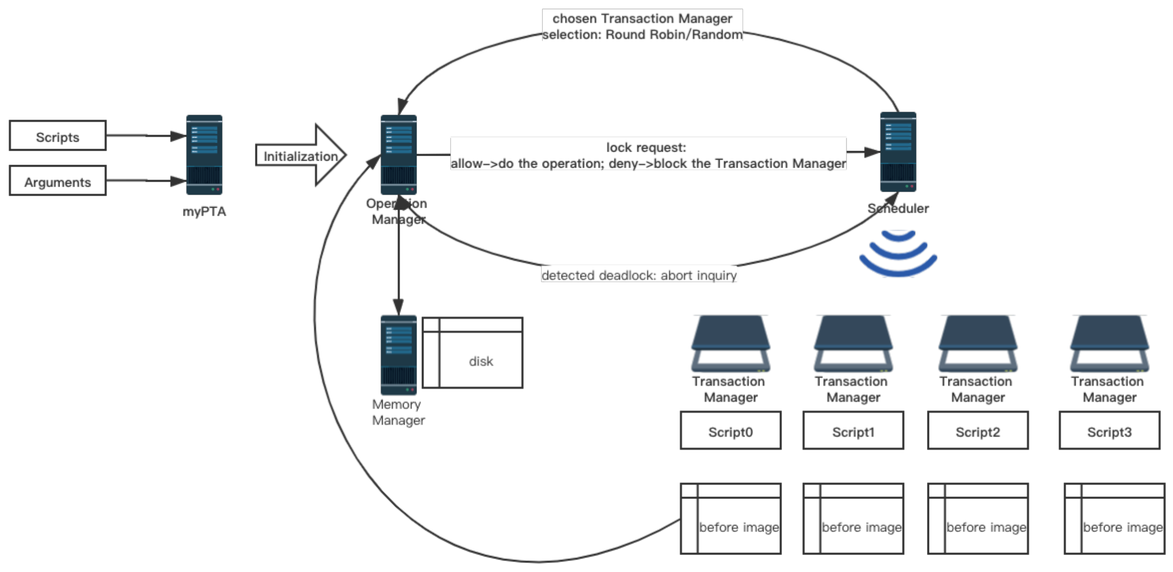
MyPTA Phase 2 Report

Team 1

1. System Flow Chart



MyPTA as the entry of the system is used for arguments check. Each script is managed by an individual transaction manager and the program reads all operations line by line and supports multiple transactions on one script. Scheduler will follow the rule of round robin/random, pass one transaction manager to operation manager each time. After getting a chosen transaction manager, the operation manager will parse the current operation in it and apply to scheduler for lock. If it gets the lock, it will do the corresponding operation, interacting with the memory manager. Otherwise, the chosen transaction manager will be blocked.

Each transaction has a before image which is used for recovery. If the transaction aborts eventually, operation manager will follow the before image to undo operations in this transaction which previously have been done.

2. Scheduler:

- **The concurrency control strategies under different execution modes**

Round Robin:
Under this mode, the operation manager will pick up a transaction manager to process in circular order. In serializable isolation level, there exist some circumstances that one transaction is blocked by other transactions, waiting for them to release lock. At this point, the corresponding transaction manager will stay with the operation which has not been done yet. Otherwise, each time a transaction manager is called, it will generate a new operation by reading next line on script. At read-committed isolation level, since the transaction will release lock once they finish one operation, no transaction

will be blocked. So round robin will simply let each transaction manager execute one operation on script at one round.

Random:
Under this mode, the operation manager will pick up transaction managers in random order. We invited a seed for random number generator to get the same set of numbers each time, in order to test the difference between serializable and read-committed better. The blocking mechanism for serializable is the same.

- **Compatibility Table**
  Lower left triangle is same as upper right
  Com = Compatible
  Con = Conflict
  Operations are assumed on same table

	R	M	W	E	D
R	Com	Com	Con if id same	Con if id same	Con
M		Com	Con	Con	Con
W			Con if id same	Con if id same	Con
E				Con	Con
D					Con

- The locking mechanisms for each operation
  We design a class for Lock where it contains the following attributes: The type of the Lock, TransactionID it belongs to, ID, TableName, AreaCode, WaitforT which is a LinkedList storing all the TransactionIDs that this Lock has to wait for, a boolean value getOrNot (initial false) to state if this Lock is got or still waiting.

  We design a HashMap as *LockTable* to store all the Locks where the Key is TableName(String) and the Value is an ArrayList of Lock(ArrayList<Lock>) storing all the Locks of the corresponding Table.

  We design a method *addTupleLock* to check if the Lock is blocked in which it calls a method *getLatestWaitfor* to get an ArrayList<Integer> containing the TransactionIDs that this Lock has to wait for. If the list is empty, it means this Lock is not blocked and can be got. Set the getOrNot attribute to be true and add this Lock into *LockTable*. Otherwise, it means this Lock is blocked and add this Lock into *LockTable*.

For the method *getLatestWaitfor* we design our Lock compatibility checking with the help of the compatibility table above. It will take a Lock instance L as a parameter and returns an ArrayList<Integer> *ret* containing the TransactionIDs that this Lock has to wait for. It first searches in the LockTable with the tableName of Lock to get the ArrayList<Lock> *locksofTable* which contains all Locks of this Table. Then for each Lock in this ArrayList, check if L is compatible with this Lock. If this Lock is got and they are not compatible, add the TransactionID of this Lock into *ret* which means L has to wait for this Lock(Transaction).

For compatibility detection, we introduce muiti-granularity locking mechanism. DL will lock the whole table (exclusive), ML will lock the selected areaCode (shared), RL, EL and WL will lock the tuple with primary key. While constructing the latestWaitFor, we first check the compatibility at the file level (for D), then check the areaCode level (for M), finally at tuple level (for E/W/R).

For Serializable isolation level(Strict 2PL protocols):
Each operation will add a Lock when executing and release all the Locks in the transaction at Commit/Abort time. The rule used to check compatibility is stated below (Assume the tableName is the same):
WL: Write Lock (with ID)
RL: Read Lock (with ID)
ML: MRead Lock (with areaCode)
EL: Erase Lock (with ID)
DL: Detele Table Lock

- R: Create a RL. Not compatible with EL and WL if their ID is the same.
- W: Create a WL. Not compatible with WL, RL and EL if their ID is the same. Not compatible with ML if the areaCode is the same.
- M: Create a ML. Not compatible with EL. Not compatible with WL and ML if the areaCode is the same.
- E: Create an EL. Not compatible with RL and WL if the ID is the same. Not compatible with ML if the areaCode is the same.
- D: Create a DL. Every Lock has to wait for DL and DL also has to wait for every Lock

For Read Committed isolation level:
Each operation will add a Lock when executing and release the Lock when the execution of operation ends.

- **The deadlock detection and handling mechanisms.**
  We design a counter variable to count the number of operations. Specifically, each time of Loading Next Operation will add 1 to the counter. When the counter is a multiple of 10, the system will begin a Deadlock detect process and return a variable deadLockedTID. If it is -1, it means no Deadlock. Otherwise, there exists a deadlock in the system and it will traverse the TransactionID of each TransactionManager for each script file and abort the transaction whose TransactionID is the same as the deadLockedTID and then start doing Undo with the help of beforeImage log for this aborted TransactionID (recovery mechanisms).

For the deadlock detection, we design a directed *WaitforGraph* where each TransactionID is a node. For each Lock in the *LockTable*, traverse its waitforT (which contains the TransactionIDs it has to wait for) and add an edge from this TransactionID to each of TransactionID in its waitforT. After traversing each Lock in the system and adding edges, then check if there exists a cycle in the *WaitforGraph*. If there is a cycle, it means there is a Deadlock. Otherwise there is not a Deadlock.

- **The recovery mechanisms under transaction abortion.**
  Under two execution modes, we adopt different recovery mechanisms for each transaction.

For Serializable execution, we use a beforeImage to recover from abort. Each operation that changes the state of the database will have a log in beforeImage. Before image contains the operations that need to perform when transactions abort.

For example, write operations will have a corresponding erase operation in log if record does not exist before or a write operation that writes old values. Erase will have a write operation in log. We don't have log for delete table operation, instead we have a Hashset that stores tables that are considered deleted during the transaction. Before we read, we first check if tables are deleted.

When a transaction commit, we just clear beforeimage and the Hashset stores deleted tables. If a transaction aborts, we will run operations logged in before image and state of the database will be restored to state before transaction starts.

In read committed, we store uncommitted operation results in an operation buffer in transactionMangaer. Since read and M operation does not change database state, we don't store their results. TransactionMangaer is private to each transaction and each transaction can only read committed data and its buffer. When a transaction aborts, we clear its buffer. In this way, its operations will not affect committed data. If a transaction commits, we will write data in buffer to committed data.

3. Transaction Manager

- Operation reading mechanisms.

As mentioned above, each script will be processed by one individual transaction manager. Each time a transaction manager is invoked, it will load one line as a command in script or stay with the command which has not been operated last time.

4. Testing

Normal Test

- Concurrency Control

Test the correctness of the concurrency control functionality between all conflicting operation pairs under different execution modes.

Two scripts with all kinds of conflicts and 8 transaction in total, running in round robin execution mode.

B 1	B 1
W X (5, John, 412-111-1111)	W X (6,Felix,412-222-2222)
R X 6	C
C	B 1
B 1	M X 412
W X (5,John,413-111-1111)	R X 5
W X (7, Yuan, 412-333-3333)	C
W X (8, Ben, 412-444-4444)	B 1
C	E X 5
B 1	E X 7
R X 5	C
R X 6	B 1
R X 7	D X
C	C
B 1	
R X 6	
C	

```
Execution logging:
T1: B 1
T2: B 1
T1: W X (5,John,412-111-1111)
MemoryManager: The table does not exist.
Written: (5, John, 412-111-1111)
T2: W X (6,Felix,412-222-2222)
The table X doesn't have record with ID = 6
Written: (6, Felix, 412-222-2222)
T2: C
T1: R X 6      //T1 R X 6 cannot get tuple lock until T2 commit (R/W)
T1: Read: (6, Felix, 412-222-2222)
T3: B 1
T1: C
T3: M X 412
T3: Mread: (5, John, 412-111-1111)
T3: Mread: (6, Felix, 412-222-2222)
T4: B 1
T3: R X 5
```

```
T3: Read: (5, John, 412-111-1111)
T3: C
T4: W X (5,John,413-111-1111)        //T4 W X at id = 5 cannot get lock until T3 commit (M/W)
Written: (5, John, 413-111-1111)
T5: B 1
T4: W X (7,Yuan,412-333-3333)
The table X doesn't have record with ID = 7
Written: (7, Yuan, 412-333-3333)
create   L-0 K X-5 X-7
T4: W X (8,Ben,412-444-4444)
The table X doesn't have record with ID = 8
Written: (8, Ben, 412-444-4444)
T4: C
T5: E X 5       //T5 E X at id = 5 cannot get lock until T4 commit (W/E)
Swap in   L-0 K X-5 X-7
Erased: X 5
T6: B 1
T5: E X 7
create   L-0 K X-5 X-8
Erased: X 7
T5: C
T6: R X 5       //T6 R X at id = 5 cannot get lock until T5 commit (E/R)
Swap in   L-0 K X-5 X-8
The table X doesn't have record with ID = 5
T6: R X 5 Failed!
T7: B 1
T6: R X 6
T6: Read: (6, Felix, 412-222-2222)
T6: R X 7
The table X doesn't have record with ID = 7
T6: R X 7 Failed!
T6: C
T7: D X  //T7 D X cannot get lock until T6 commit (R/D)
T8: B 1
T7: C
Deleted: X
T8: R X 6
MemoryManager: The table does not exist.
T8: R X 6 Failed!
T8: C
```

- DeadLock Detection

Test the correctness of your deadlock detection functionality for all relevant operations. Test scripts are similar except for different operations. For example exam deadlock between RW and run in round robin and serializable mode.

T2	T1
B 1	B 1
W X (2, John2,412-111-2222)	W X (1,John1,412-111-2222)
R X 1	R X 2
C	C

Execution logging:

```
T1: B 1
        T2: B 1
        T1: W X (1,John1,412-111-2222)
        MemoryManager: The table does not exist.
        Written: (1, John1, 412-111-2222)
        T2: W X (2,John2,412-111-2222)
        The table X doesn't have record with ID = 2
        Written: (2, John2, 412-111-2222)
        T1 has been aborted due to a deadlock
        Start undo T1
        Erased: X 1
        End undo T1
        T2: R X 1
        The table X doesn't have record with ID = 1
        T2: R X 1 Failed!
        T2: C
```

Because it runs in round robin, T2 first writes (2, John2,412-111-2222) to X and then writes (1,John1,412-111-2222) to X. Two write operations' ids are not same so they are not conflict with each other and they can both obtain tuple lock. Then T1 and T2 tries to read but they cannot obtain locks. After a set number of iterations, the program will check for deadlock. It will both evaluate T1 and T2. T1 is chosen to be aborted. T2 resume.

We can change R and W to any pair of conflict operations and results are similar.

- Recovery
  Testing the correctness of the recovery functionality for all relevant operations.
  We use the following parameters: LSM_pageSize = 2, bufferSize = 2, seed = 0 (round robin)

For W:
B 1
W X (1,John,412-111-1111)
W X (2,Ben,412-222-2222)
A
B 1
R X 1
R X 2
C

After running the script above, the table X in database has two SSTables in level0 and the first has two records (1,John,412-111-1111) and (2,Ben,412-222-2222) and the second has two records (1, , deleted) and (2, , deleted) as expected. The T2 can't read the record from X as expected too. The content in beforeImage log for T1 is: E X 1 and E X 2 When T1 is aborted, the system will read the beforeImage log for T1 and execute the operations inside: E X 1 and E X 2. When T2 executes R X 1 or R X 2, it will read the SSTable in level0 in a reverse order and will read the record (1, , deleted) and (2, , deleted) first and then return. It means there is no record with ID = 1 or ID = 2 in table X as expected.

For E:
B 1
W X (1,John,412-111-1111)
W X (2,Ben,412-222-2222)
C
B 1
E X 1
E X 2
A
B 1
R X 1
R X 2
C

After running the script above, the table X in database has three SSTables in level0 and the first two is the same as above and the third has two records (1,John,412-111-1111) and (2,Ben,412-222-2222) as expected. The read operations in T3 can get the correct records (1,John,412-111-1111) and (2,Ben,412-222-2222) as expected because it search in the level0 in a reverse order and search the third SSTable first containing these two records as expected. It means when T2 is aborted, the system reads the beforeImage log for T2 and executes the operations inside: W X (1,John,412-111-1111) and W X (2,Ben,412-222-2222) as expected.

In conclusion, we test the correctness of our recovery mechanisms for W and E with the help of beforeImage log.

5. Benchmark test:

We test our system with 100 transactions, each transaction has 1000 random operations. With different isolation levels and file reading methods, the operations stay the same.

SSTable size: 5
LSM Tree buffer size: 10

For read committed isolation level, there will not be any deadlock appears. The operation distribution is as follow:

19.993 % of operations is read
19.991999999999997 % of operations is write
20.251 % of operations is readAreaCode
19.958000000000002 % of operations is Erase
19.806 % of operations is Delete table

1. read committed, round robin: **Average respond time is 8109.59 ms.**

2. read committed, random: **Average respond time is 8320.7 ms.**

3. serializable, round robin:

19.5 % of operations is read
20.0875 % of operations is write
20.3875 % of operations is readAreaCode
19.5875 % of operations is Erase
20.4375 % of operations is Delete table
8 transactions committed

Average respond time is 233.46153846153845 (considering the abort transactions)
Average respond time for committed transactions is 3035.0 ms.

4. serializable, random:
19.8 % of operations is read
20.466666666666665 % of operations is write
19.116666666666667 % of operations is readAreaCode
19.53333333333333 % of operations is Erase
21.083333333333336 % of operations is Delete table
6 transactions committed
Average respond time is 196.66990291262135 (considering the abort transactions)
Average respond time for committed transactions is 3376.1666666666665 ms.

6. Contribution

We use zoom meeting and shared screen for developing and debugging. Guo, Zhu and Yao have equal contribution on OperationManager, Scheduler, TransactionManager.
For testing:
Concurrency Control: Guo
Deadlock: Yao
Abort: Zhu
Benchmark Test: Guo