CS 387 - Project Final Report

Team Sad4Sansa

ACID-D-BASE

https://github.com/harshitgupta412/toydb

BY:	*Roll Number*
Dhruv Arora	190050034
Harshit Gupta	190050048
Pradipta Parag Bora	190050089
Raj Aryan Agrawal	190050097

Introduction

https://github.com/harshitgupta412/toydb

Our project aims to level up toydb to create a **better and robust** interface along with adding ACID properties.
We first focused on abstracting out the low level C functions that were previously defined in toyDB to create a Object Oriented structure. These classes are responsible for handling communications with the dblayer, amlayer and pflayer. The next task was to define various operations such as querying, project, join etc.
Finally, we created a transaction manager class which will manage all the locks and permissions centrally. Further to provide a good interface to the user, we created the QueryObj and Client classes which creates the query evaluation tree and interact with transaction manager respectively.

Pre ACID Requirements and Changes

We first start with changes to the toyDB layers

- PFLayer: This layer was used as it is.
- AMLayer:
  - The AMLayer code was originally very hardcoded for single valued incides. We changed this to allow for indexing over multiple values
  - While DBLayer supports LONG as a datatype, AMLayer did not do so, this support was added to allow indexing on LONG datatypes
  - During an AMSearch, if the search key is not found, the base AMLayer decides to leave the last accessed leaf paged fixed in memory, never to be found because it loses the page number. This leads to the PFLayer freezing since there is no method to obtain the page number of the page fixed
- DBLayer:
  - We assume that there is no page fixed in memory and ensure that this is always the case
  - We added support for searching through the table given a primary key. This is implemented in `tableSearch` function.
  - We also added support for deleting a record in the same file. This is done in `tableDelete`

Our Database Abstraction

Using these layers, we create a heavily abstracted system to allow easy creation, deletion and modification of Tables, Databases and Users.

For this, we create a class `Table` which is written using the low level abstraction provided in the layers to make a higher level abstraction for easy addition of rows, deletions, adding of a schema with support of primary keys, indexing etc. This is done by using the functions of DBLayer, AMLayer and PFLayer in its implementation. This class allows us to actually query the database like a normal database. This class supports most of the functions of relational algebra including Joins, Unions, Intersection, Project and Select. These are some of the major functions supported:

- **Table creation**: Allows creation of a new table with previously given indexes and schema.
- **Get Records**: Get all the records from the table in a void** structure.
- **Add/Delete/Update Row**: Add/delete/update a row from the table. The data (primary key in case of delete) has to be given.
- **Create/Delete index**: We allow creation of new indices. By default, we already create an index on the primary key.
- **Querying**: We support 3 types of queries:
  - Querying using primary key
  - Querying using index
  - Querying using callback functions.

Database and User abstractions are stored as Tables for easy checking and authentication. The Database abstraction stores a table for the list of databases existing, and also a table for the (Database, Table) pairs. The latter also stores the metadata of the table, which is

- Table name
- Schema
  - Number of Columns
  - Column Name
  - Column Type
- Primary Keys
- Indices

This metadata allows easy loading of table object to perform queries on.

The user class allows for creation of users, each with their username and passwords. The passwords are hashed using SHA256 for secure authentication of the user.
We create this class to allow for a client to be authenticated before starting any transaction. This class also allows creation of other users and assigning read/write permissions to them. These permissions are later used by the transaction management to verify query requests.

Transaction Management

After these abstractions, we finally implement ACID Properties to the system created above.

Manager Side

Transaction management in ACID-D-BASE follows 2-phase relation level locking protocol. Processes can request locks from a central "txnmgr" process. Inter process communication is done through sockets for sake of efficiency and to support parallelism. We have added Read-Write locks here which allows for multiple reads but single write for higher parallelism.

The "txnmgr" process runs and maintains connections with multiple clients. It also has a data structure to keep track of processes reading / writing certain tables. A client may ask txnmgr to initiate a transaction. The txnmgr process then assigns a txnID to the client and waits for further requests. A client with a running transaction can request for read / write type locks on relations from the txnmgr. Following the rules of read/write locking protocols and user privileges, these requests may be accepted or denied. If any of the client's requests get declined, they roll back their partial updates and the transaction fails. Otherwise, the transaction's success depends on violation of any constraints. Transaction rollbacks are a part of query execution.

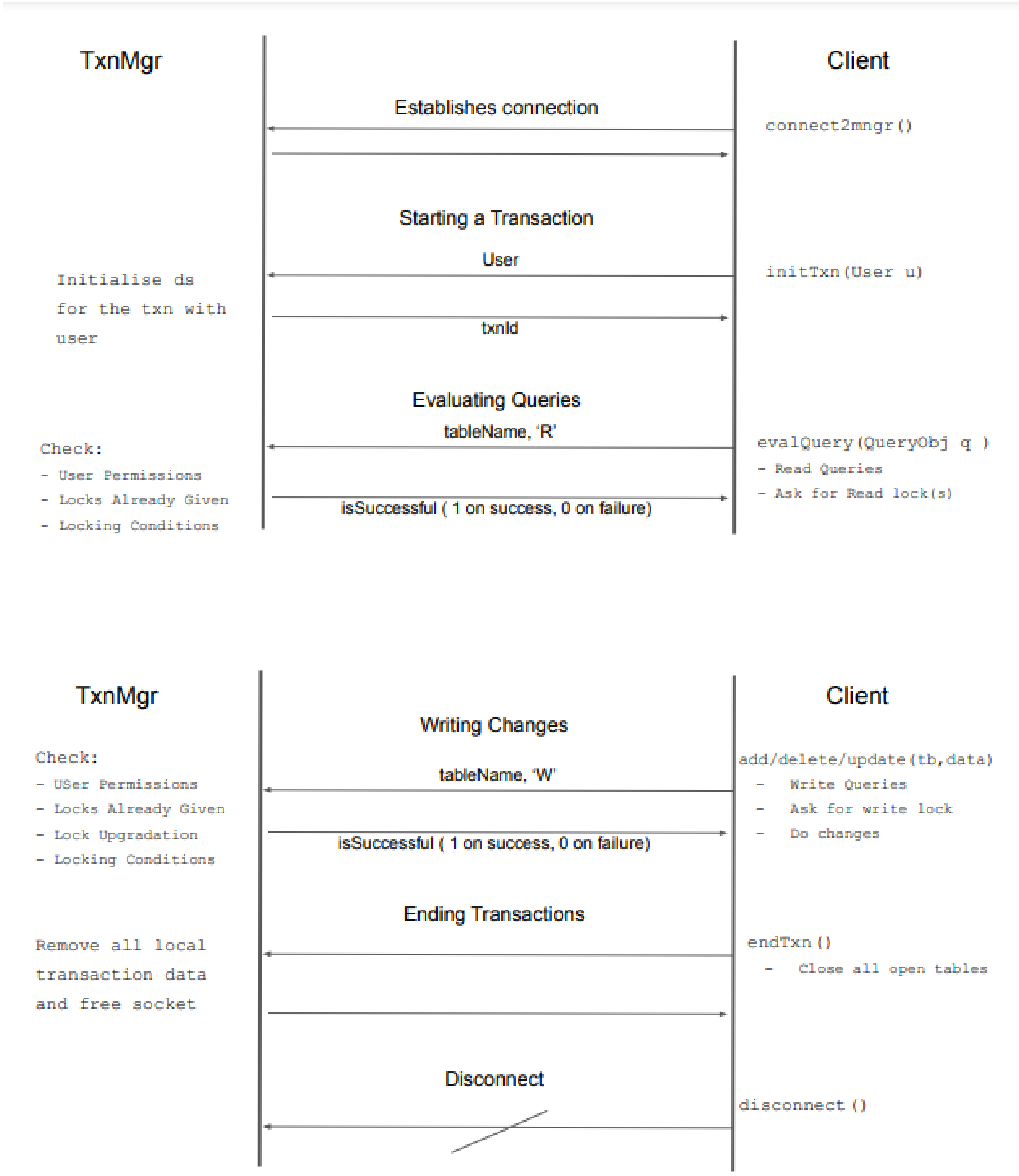
We've separated heavy duty operations like queries, joins, insertions and deletions from locking logic by running a central daemon. This allows multiple independent processes to access and compute on the database concurrently. It also ensures atomicity of updates and isolation among transactions. This method of locking also maintains serializability. Further since the transaction is rolled back as soon as a lock can't be assigned, there will be no deadlocks.

Client Side

Since the above Table class is too powerful to give to the users, we made separate Client Class which the user can use for doing operations. This allows the user to connect to the

txnmgr and request locks. Once locks are granted, the queries are evaluated.
For the read queries, a QueryObj class is provided. This creates the evaluation tree for queries similar to what is done in Spark. Finally we call Client.evalQuery on the object which will give us the result as a (void***).

The transaction protocol



Testing Plan For Basic Functionalities

Table Unions

```
Table 1
Number of Columns: 2
ID1     ID2
Hello   1
Dunno   2
Bye     3
------------------------------------------------------------------------
Table 2
Number of Columns: 2
ID1     ID2
Dunno   2
Bye     3
Hi      4
------------------------------------------------------------------------
Union Result
Number of Columns: 2
ID1     ID2
Hello   1
Dunno   2
Bye     3
Hi      4
------------------------------------------------------------------------
```

Table Intersections

```
Table 1
Number of Columns: 2
ID1     ID2
Hello   1
Dunno   2
Bye     3
------------------------------------------------------------------------
Table 2
Number of Columns: 2
ID1     ID2
Dunno   2
Bye     3
Hi      4
------------------------------------------------------------------------
Intersect Result
Number of Columns: 2
ID1     ID2
Dunno   2
Bye     3
------------------------------------------------------------------------
```

Table Projection

```
Table 1
Number of Columns: 3
ID1     ID2     ID3
Hello   hello   1
Dunno   dunno   2
Bye     bye     3
Hi      hi      4
------------------------------------------------------------
Union Result
Number of Columns: 2
ID1     ID3
Hello   1
Dunno   2
Bye     3
Hi      4
------------------------------------------------------------
```

Table Join

```
Table 1
Number of Columns: 2
ID      VALUE
1       2
2       3
------------------------------------------------------------
Table 2
Number of Columns: 2
ID      VALUE
1       4
2       5
------------------------------------------------------------
Joined Table
Number of Columns: 4
a.ID    a.VALUE b.ID    b.VALUE
1       2       1       4
2       3       2       5
------------------------------------------------------------
```

Table Creation and Deletion

```
Original (Db,Table)
Number of Columns: 3
DBNAME          TABLE                   METADATA
DB              DB_TABLE                DB
DB              DB_CROSS_TABLE          DB
DB_USER_DB      DB_USER_TABLE           DB_USER_DB
DB_USER_DB      DB_USER_PRIV_TABLE      DB_USER_DB
DB_USER_DB      DB_USER_PRIV_DB         DB_USER_DB
TEST_DB         TABLE_2                 TEST_DB
TEST_DB         TABLE_1                 TEST_DB
------------------------------------------------------------
After adding 2 tablesNumber of Columns: 3
DBNAME          TABLE                   METADATA
DB              DB_TABLE                DB
DB              DB_CROSS_TABLE          DB
DB_USER_DB      DB_USER_TABLE           DB_USER_DB
DB_USER_DB      DB_USER_PRIV_TABLE      DB_USER_DB
DB_USER_DB      DB_USER_PRIV_DB         DB_USER_DB
TEST_DB         TABLE_2                 TEST_DB
TEST_DB         TABLE_1                 TEST_DB
TEST_DB         TEST_TABLE              TEST_DB
TEST_DB         TEST_TABLE_2            TEST_DB
------------------------------------------------------------
After deleting TEST_TABLE
Number of Columns: 3
DBNAME          TABLE                   METADATA
DB              DB_TABLE                DB
DB              DB_CROSS_TABLE          DB
DB_USER_DB      DB_USER_TABLE           DB_USER_DB
DB_USER_DB      DB_USER_PRIV_TABLE      DB_USER_DB
DB_USER_DB      DB_USER_PRIV_DB         DB_USER_DB
TEST_DB         TABLE_2                 TEST_DB
TEST_DB         TABLE_1                 TEST_DB
TEST_DB         TEST_TABLE_2            TEST_DB
------------------------------------------------------------
```

Adding, Updating and Deleting rows in table

```
Initial Table
Number of Columns: 2
ID      VALUE
4       2
1       3
------------------------------------------------------------
After deleting row with primary key 4
Number of Columns: 2
ID      VALUE
```

```
1           3
--------------------------------------------------------------------------------
After adding a row and updating row with pk 1
Number of Columns: 2
ID          VALUE
6           4
1           5
--------------------------------------------------------------------------------
Adding row with pk 1 and no update
Number of Columns: 2
ID          VALUE
6           4
1           5
--------------------------------------------------------------------------------
```

Index Creation, Deletion, Index Querying

```
Original Table
Number of Columns: 2
ID          VALUE
4           2
1           3
--------------------------------------------------------------------------------
Creating index on Primary Key
--------------------------------------------------------------------------------
Querying for rows with ID < 3 using index
Number of Columns: 2
ID          VALUE
1           3
--------------------------------------------------------------------------------
Querying for rows with ID > 3 using index
Number of Columns: 2
ID          VALUE
4           2
--------------------------------------------------------------------------------
Deleting index on Primary Key
--------------------------------------------------------------------------------
Negative test for query index
Index not found in query index
```

Custom query using callback

```
Original Table
Number of Columns: 2
ID      VALUE
4       2
1       3
--------------------------------------------------------------------------------
Getting all rows with ID = 1
Number of Columns: 2
ID      VALUE
1       3
--------------------------------------------------------------------------------
Getting all rows
Number of Columns: 2
ID      VALUE
4       2
1       3
--------------------------------------------------------------------------------
```

Testing Plan for Users

```
Adding new user

--------------------------------------------------------------------------------
Assigning permissions to new user

--------------------------------------------------------------------------------
Trying to login with incorrect password
Incorrect username or password

--------------------------------------------------------------------------------
Creating table 0
Creating table 1

--------------------------------------------------------------------------------
Connecting to database with new user

--------------------------------------------------------------------------------
User permission validity
MAIN_DB Allowed to Read? 1
MAIN_DB Allowed to Write? 0
MAIN_TABLE Allowed to Read? 1
MAIN_TABLE Allowed to Write? 1
TEMP_TABLE_1 Allowed to Write? 0

--------------------------------------------------------------------------------
```

Testing Plan for Transaction Management

Querying to Add and Delete Rows

`parallel2.bin` reads the current table and prints it. `api.bin` then inserts a new row which is reflected. Running it again gives a rollback showing the changes are reflected. `api3.bin` deletes this row and running `parallel2` shows this change.

```
./parallel2.bin
John    20    0
Jane    30    1
Joe       40    0
eval done

./api.bin
Table creme.pie1 not open, opening
Table creme.pie1 opened
add done
John        20        0
Jane        30        1
Joe            40        0
hapipola    221001    21220
eval done

./api.bin
Table creme.pie1 not open, opening
Table creme.pie1 opened
Failed to add row
rollback
rollback

./api3.bin
delete done
John    20    0
Jane    30    1
Joe     40    0
eval done

./parallel2.bin
John    20    0
Jane    30    1
Joe       40    0
eval done

./api.bin
Table creme.pie1 not open, opening
Table creme.pie1 opened
add done
John        20        0
Jane        30        1
Joe            40        0
hapipola    221001    21220
```

```
eval done
```

Atomicity by Rollback

Before transaction is complete, `api2.bin` calls a rollback, on reading the table again using `parallel2.bin` we see that the in between changes are not reflected.

```
./parallel2.bin
John    20    0
Jane    30    1
Joe 40    0
hapipola    221001    21220
eval done

./api2.bin
Table creme.pie1 not open, opening
Table creme.pie1 opened
add done
John    20    0
Jane    30    1
Joe 40    0
hapipola    221001    21220
hello8a 10001 23
eval done
rollback

./parallel2.bin
John    20    0
Jane    30    1
Joe 40    0
hapipola    221001    21220
eval done
```

Reader Writer Lock Correctness

1. Parallel Reads was already seen in the above examples
2. A Reader and Writer

```
New connection , socket fd is 17 , ip is : 127.0.0.1 , port : 50018
Starting Decode
Transaction Request
Username: SUPERUSER
Starting Decode
Write Lock Request
DB: creme, Table: pie1
Granted
New connection , socket fd is 18 , ip is : 127.0.0.1 , port : 50019
```

```
Starting Decode
Transaction Request
Username: SUPERUSER
Starting Decode
Read Lock Request
DB: creme, Table: pie1
Table creme.pie1 is currently being written to by another transaction
Starting Decode
End Transaction Request
```

Here writer got the lock request first.

```
New connection , socket fd is 17 , ip is : 127.0.0.1 , port : 50025
Starting Decode
Transaction Request
Username: SUPERUSER
Starting Decode
Read Lock Request
DB: creme, Table: pie1
Granted
New connection , socket fd is 18 , ip is : 127.0.0.1 , port : 50026
Starting Decode
Transaction Request
Username: SUPERUSER
Starting Decode
Write Lock Request
DB: creme, Table: pie1
Table creme.pie1 is currently being read from by another transaction
Starting Decode
End Transaction Request
```

Here the reader gets the lock first and so writer cannot perform the transaction.

3. Writer and Writer

```
New connection , socket fd is 17 , ip is : 127.0.0.1 , port : 50013
Starting Decode
Transaction Request
Username: SUPERUSER
Starting Decode
Write Lock Request
DB: creme, Table: pie1
Granted
New connection , socket fd is 18 , ip is : 127.0.0.1 , port : 50014
Starting Decode
Transaction Request
Username: SUPERUSER
Starting Decode
Write Lock Request
DB: creme, Table: pie1
Table creme.pie1 is currently being written to by another transaction
```

```
Starting Decode
End Transaction Request
```

Analysis and Summary

What we could do

- We implemented a simplistic database engine abstraction which has classes for Database, Tables and Users.
  - For Tables
    - Tables support addition, deletion and updates of rows
    - It also supports primary key integrity checks. These can be a tuple of columns or a single column
    - We also support indexing on a key. AMLayer was also modified to allow indexing on multiple columns for indexing on primary keys and others if necessary
    - Tables allow querying based on standard operators like $>, \geq, <, \leq, =$ using existing indexes. We also allow more complex conditions by allowing to pass a custom callback which can evaluate any row to be considered or not.
    - Table allows queries to project table to a subset of columns, take union or intersection of 2 tables or take join of 2 tables.
  - For Databases
    - This is an abstraction that allows creation and deletion of databases, and also for tables within the database
    - This also allows the database to load a table to give a table object from its name
  - For Users
    - Users are stored in a table with their username and password along with admin status. Only the superuser has admin status, while the other users do not
    - The password of users are stored using a hash function for security
    - Users can also be assigned permissions to read/write a specific database or a table within a database
    - The superuser can create other users and also assign them these permissions
- We have created a daemon process that is constantly running and keeps listening on ports for transaction requests
  - We use sockets to create a server and client side for inter process communications
  - A client can initiate a transaction, perform queries and finally end the transaction, which commits it.

  - The daemon process allows for concurrency of requests with locking at the table level to allow multiple reads and single write

- We also create a simplistic evaluation engine working like
  - It uses relational algebra as a base to form an evaluation tree structure which is eagerly computed
  - Using this evaluation we can find out which locks are required and use it to implement a Two Phase Locking protocol.

What we could not do

- The daemon process
  - currently accepts requests for read and write locks. This can be modified such that it accepts complete evaluation trees and returns query results.
    This would not require major changes however this will end up supporting a database that can be hosted on a network
  - Currently, creation and deletion of tables is not supported during a transaction as this involves major changes like updation of global data structures
- The evaluation engine can be extended to support rudimentary query optimizations using indexes available.
- We also were not able to add integrity constraints for foreign keys because it would require major changes on transaction manager layer