CS 387 - Project Design Document

Team Sad4Sansa

ACID-D-BASE

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

API

This project aims to create C++ library for databases. We provide the following classes and functions:

- `Database:` This class implements the database component of a DBMS system. It stores the permissions of the current user that is accessing the database.
  - `connect(string name, User *current_user)`: connect to the database with the name `name` by the user `current_user`.
  - `createTable(Table* t)`: add the table pointed by `t` to the database
  - `deleteTable(Table* t)`: delete the table pointed by `t`. It is the responsibility of the user to ensure that any other table does not depend on this table.
  - `bool initTransaction()`: This method starts a transaction and returns True if this is successful and another transaction is not in progress.
  - `bool commit()`: This method commits a transaction. Returns false if there was not any active transaction to commit.
  - `bool rollback()`: This method rollbacks a transaction. Returns false if there was not any active transaction to commit.
- `User:` This class implements the users for the database.
  - `User::User(string username, string password)`: Login with username and password.
  - `User::User(FILE* cred)`: Alternate method to login with a credential file.
  - `bool User::isAdmin()`: There is a single preset admin who is allowed to use the following methods :
    - `bool User::addUser(string username, string password)`: Add a new user. Returns true on success.
    - `bool User::assignPerm(User& user, Database& db, int perm)`: Assigns permissions to the user for the database. Returns true on success.
    - `bool User::assignPerm(User& user, Table& tbl, int perm)`: Assigns permissions to the user for the table. Returns true on success.
- `Table:` This will be a modification of the low level Table class which is already present in ToyDB. We will have to add primary key and constraints.
  - `Table(Schema* schema)`: This is the constructor to create a table with a given schema.
  - `const Schema& getSchema()`: Returns a constant reference to the schema of the table.

  - `bool addRow(void* data[], bool update)`: This adds a row to the table. The data is given in the `data` object in which the entries are in the same order as the schema. The `update` parameter denotes whether to update the data if the row with the same primary key exists or give an error. Returns true on success.
  - `bool deleteRow(void* pk)`: Delete the row with primary key `pk`. returns true on successful deletion or if the row does not exist.
  - `Table* query(int colID, Operator operator[], void* value[])`: This function allows efficient queries on tables by giving an operator condition (like $>=, <=, >, <, ==$ etc.) on the columns. Values passed need to be converted to void* and would be automatically de-referenced and type casted using the table's schema. If the columnID is indexed, it uses an index scan, otherwise a sequential scan
  - `Table* query(void* callback)`: This is another function that allows us to query the table. It returns a table pointer pointing to the table which contains the answer to the query. The query is given as a callback function. The callback function takes a single row as a parameter and returns a bool. If the return of callback is true, the row is taken in the answer else not.
  - `void** getRow(void* pk)`: Returns the row with primary key equal to `pk`
  - `void print()`: Self explanatory
  - `int createIndex(int[] cols)`: Creates a B-tree indexing on the columns specified by the cols[] array and stores it along with the table. Returns the ID of the indexing created.
  - `bool eraseIndex(int id)`: Erases the indexing created with id `id`

- `Schema`: This class contains the schema including the primary key, foreign key and other constraints.
  - `Schema(pair<String,type> cols[], int pk[])`: As the name suggests, `cols` contains the names and types of the columns. Notice that in our system, the index of columns is important as all the rows of the table are ordered in the same fashion. `pk` contains the integers which together form the primary key for the schema. If the table has no primary key, then this can be set to empty list.
  - `bool foreignKey(int[] ref_cols, Table* refT)`: This function adds a foreign key constraint to the Schema. `ref_cols` denotes the indices of the columns that references to another table which is referenced to by `refT`. We currently only support foreign key reference to the whole primary key of another table.

- `Table* Join(Table* t1, Table* t2, int[] cols1, int[] cols2)`: This is a separate function used to create join over 2 tables based on the indices provided for each table. The number of indices must be same for both tables, and the order of indices are used when comparing for Join. This only allows for equality checks. The final table will be indexed on the common columns of the join for efficient average case querying. Passing empty lists in the cols will result in a cross product.

For handling transactions, we will use 2 hashmaps to store the records added/updated and the other for the records deleted. Each record (i.e. based on primary key constraints) can occur in at most one of these at a time, reflecting the current state during the transaction.
We also keep a hashmap for the tables created or deleted during the transaction process to keep track of them.
For indexing, we use the B-tree implementation in AMLayer.

Finally, for allowing concurrent accesses, we will be using threads and locking mechanisms to implement a reader writer lock.

Control Flow Diagrams

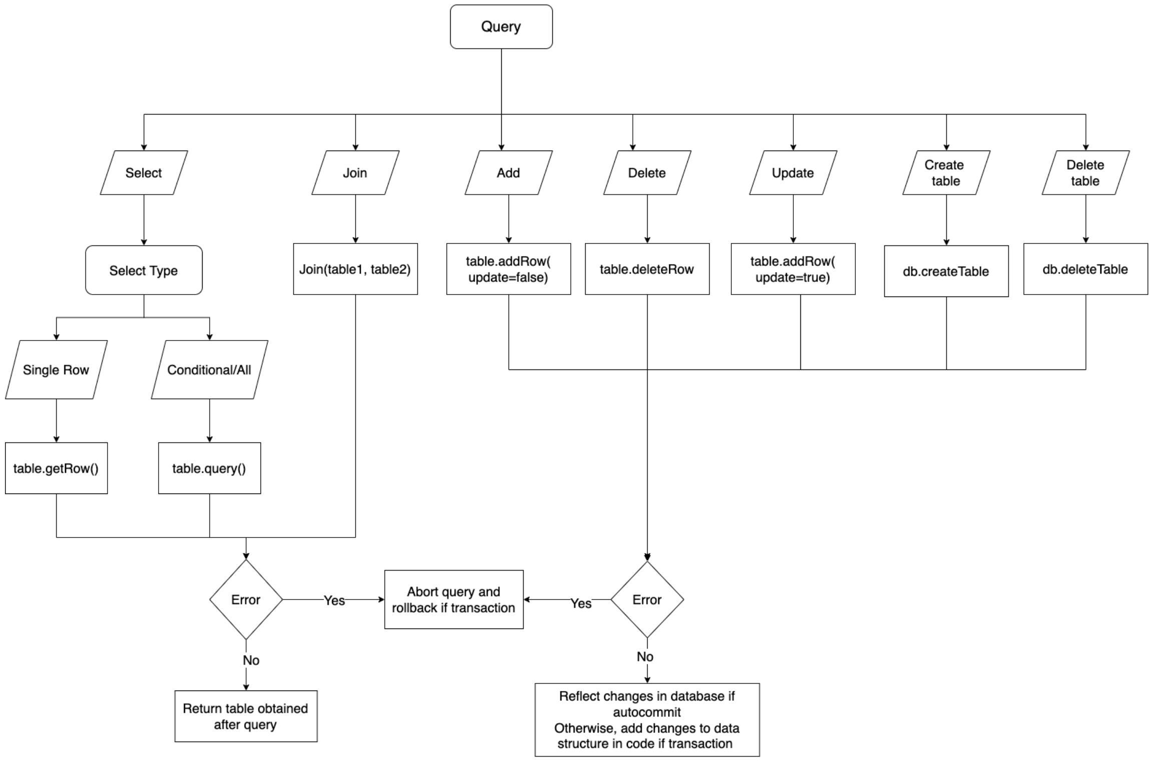
Query
Select
Join
Add
Delete
Update
Create table
Delete table
Select Type
Join(table1, table2)
table.addRow(update=false)
table.deleteRow
table.addRow(update=true)
db.createTable
db.deleteTable
Single Row
Conditional/All
table.getRow()
table.query()
Error
Yes
Abort query and rollback if transaction
Yes
Error
No
No
Return table obtained after query
Reflect changes in database if autocommit Otherwise, add changes to data structure in code if transaction

Figure 1: Control flow diagram for processing a query

A query can be called either within a transaction or without one - in which case it is autocommited. In those cases the changes are directly reflected in the database.

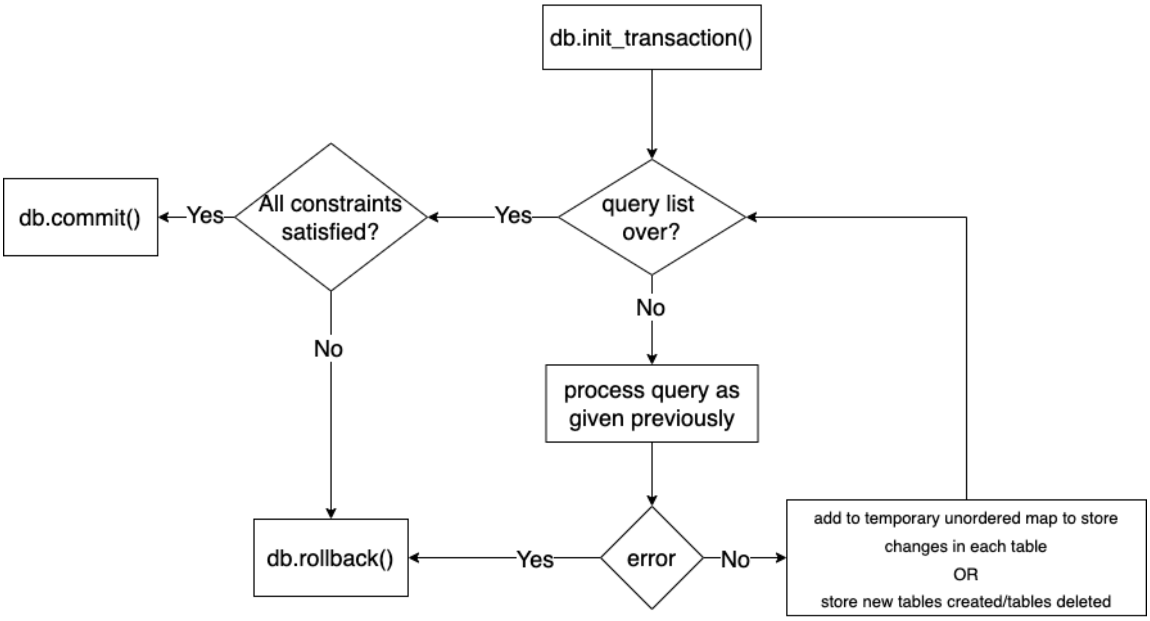
db.init_transaction()
query list over?
Yes
All constraints satisfied?
Yes
db.commit()
No
No
process query as given previously
error
Yes
db.rollback()
No
add to temporary unordered map to store changes in each table
OR
store new tables created/tables deleted

Figure 2: Control flow diagram for a transaction