Database Project Report

2014-17184

Class and Method Implementation

To store the schema of all the created tables, I created a few classes (data structures) that contain necessary informations:

- **Attribute**

Attribute class describes each attribute in a table. It contains the information of its type, name, and whether it has any integrity constraints such as not null, being part of primary key or foreign key.

- **Type**

Type class specifies the type of each attribute. It contains a type name, which is an enum of integer, character, and date, and an integer that indicates the length of *Char* if it is one. It also has some methods that helps print output or check error:

  - **toString**

    toString method is used in *desc* query. It changes an object of type Type, to corresponding appropriate output string.

  - **isValid**

    isValid method is used to check for *CharLengthError.* If the instance's type name is char, and character length is below 1, it returns false, otherwise it returns true.

  - **compareTo**

    compareTo method is overrode to implement interface *Comparable*. It is used to check if the foreign key constraint is preserved.

- **Table**

Table class describes each Table in the database. It consists of its name, primary key, attribute list, and list of tables that reference it with a foreign key. Below are some of the core methods implemented in class Table.

  - **findAttribute**

    findAttribute method returns the attribute in the table given the name of the attribute. If none is found, null is returned.

  - **setPrimaryKey**

    setPrimaryKey method literally sets the primary key of the table. However, during the process it checks if the primary key of the table is already set

(*DuplicatePrimaryKeyDefError),* or if the designated columns actually do not exist (*NonExistingColumnDefError*). On success, the method sets the primary key of the table, and adds corresponding integrity constraint for each attribute. The method returns a Message class containing the error messages if error occurred. Else it returns null.

- **checkDuplicate**

  checkDuplicate method sees if there already is an attribute in the table of the given name. If so, it returns true; else false.

- **addAttribute**

  addAttribute method adds a new attribute to the attribute list of the table. During the process, it calls checkDuplicate and if it returns true, the method returns corresponding error message (*DuplicateColumnDefError*). It returns null on success.

- **addReferred**

  addReferred method adds a table to the "referred list." This is called in setForeignKey method to update the reference information in each table. In project 3, containing a "referring list" might be necessary.

- **setForeignKey**

  setForeignKey method sets a foreign key in the table given the attributes' name list and the referring table. It looks for various errors - *ReferenceTypeError, ReferenceNonPrimaryKeyError, NonExistingColumnDefError, ReferenceColumnExistenceError.*

  On success, it calls addReferred method to add the table to the referred list of the table it is referring to and sets the integrity constraints for each attribute. On error, it returns the error message.

• **DBManager**

DBManager class has a list of all the tables that are currently created and stored in the database. It is a singleton class and the DBManager deals with all the DDL queries required to implement in this step of the project: *create table; show tables; desc; drop table.* Following are some methods implemented to support the DDL queries.

- **findTable**

  findTable method finds table of the given name in the table list and returns it. If none is found, null is returned.

- **addTable**

  addTable method adds the given table to the table list.

- **dropTable**

  dropTable does exactly what *drop table* query does: it removes the table of the given name from the table list. It checks for the existence of the table (*NoSuchTable)*, and if the table is being referenced by some other table *(DropReferencedTableError)*. On success, it removes the table and returns null. Else, it returns a Message object describing the error.

- **descTable**

  descTable method does exactly what *desc* query does: it shows the schema information of the table of the given name. If no table of the given name is found, it returns corresponding error message *(No Such Table)*. Else, it prints the result of *desc* query and returns null.

- **showTables**

  showTables method does exactly what *show tables* query does: it shows all the tables currently in database. If there is no table, the method returns corresponding error message (*ShowTablesNoTable)*. Else, it prints the result of *show tables* query and returns null.

• **Message**

Message class contains all the messages that the DBMS is supposed to print. It also has a string that specifies the argument of the message (e.g. table name, column name … etc). the class in itself has all different types of Message objects. By calling get<MessageName> static method, we can get the corresponding message without having to construct a new message every time.

• **Berkeley**

Berkeley class is a singleton class. It is in charge of updating and retrieving data from the file. For now, it updates the database manager into the file whenever a new table is successfully created or dropped. In further proceeding the project, this class will also have to update records in a table whenever they are created or removed. Below are methods to open, close file and update, retrieve database manager.

- **open**

  open method configures and opens the database (file). Also it retrieves the existing manager and sets it to the DBManager singleton object by calling setDBManager.

- **close**

  close method closes the database (file).

- **updateManager**

  updateManager method updates the database manager object in the file.

- **retrieveManager**

  retrieveManager method retrieves the existing database manager object from the file.

Bird-eye View of the overall Design

• **Error Handling**

One of the core specifications is to print appropriate error messages for various kinds of errors. My program is designed to print all the error messages in printMessage. In addition, printMessage in one location - after each call for query function. In order to pass message to printMessage, every method that has to check for any error before action is designed to return a Message object if an error occurred. This way, the program can easily identify if any action resulted in error, and pass the error message to printMessage method.

• **DBManager as a form of cache**

DBManager class contains schema information for all the tables in the database. In some sense, DBManager works as an index table for other tables. For example, in further proceeding the project, the program will have to access specific attribute of all the records in a specific table. In doing this, it first checks the schema of the table in the DBManager, computes the index for the attribute, and accesses the table afterwards to actually get values from the computed index.

How to Compile and Execute

To run in command line, use the attached Makefile.

• make jj

make jj command generates .java files for the given .jj file

• make compile

make compile command compiles ScratchDBMSParser.java with Berkeley library.

• make run

make run executes the program.

See make file for manual instructions to compile and execute the program.

This project is done on IntelliJ and you can run it by simply opening this project on IntelliJ and clicking on the run button.

Comments

Careful designing was necessary to create good structure for the DBMS.

I hope this design works well when further proceeding the project.