German University in Cairo

Faculty of Media Engineering and Technology

CSEN604: Database II

Mini-Project 1

Instructor: Dr. Wael Abouelsaadat

TAs: Belal Medhat & Ahmed Hesham

Release Date: Feb 15th, 2020

Last Updated: Feb 18th, 12:33PM, 2020

Submissions: March 4th && March 25th

Background

The course projects are worth 20%. Mini-Project 1 (this one) is worth 13%. No best policy.

Team Formation

This assignment should be done in teams of THREE to FIVE. You can make a team cross-tutorial and cross-major. If you cannot find a team, contact your TA **ASAP**, and you will be synched with other students.

Environment

You must use Java to develop this project. You can use any IDE of your choice such as Eclipse, IntelliJ, NetBeans, etc…

Submissions

Submissions will be made electronically via MET website.

Description

In this project, you are going to build a small database engine with support for B+ trees and R trees. The required functionalities are 1) creating tables, 2) inserting tuples, 3) deleting tuples, 4) searching in tables linearly, 5) creating a B+ tree index, 6) searching using B+ tree index, 7) creating an R tree index, and 8) searching using an R tree index.

The description below is numbered for ease of communication between you and course staff.

Tables

1) Each table/relation will be stored as pages on disk.

2) Supported type for a table's column is one of: java.lang.Integer, java.lang.String, java.lang.Double, java.lang.Boolean, java.util.Date and java.awt.Polygon

Note: java.awt.Polygon will be used to store spatial data, while the other types are non-spatial. The user will be able to create an R tree index on any column whose type is java.awt.Polygon while for the remaining 5 types, the appropriate index is a B+ tree.

3) Each table should have an additional column beside those specified by the user. The additional column must be called **TouchDate** and is initialized with the date/time of row insertion and updated with current date/time every time a row is updated.

Pages

4) A page has a predetermined fixed maximum number of rows (N). For example, if a table has 40000 tuples, and N=200, the table will be stored in 200 binary files.

5) You are required to use Java's binary object file (.class) for emulating a page (to avoid having you work with file system pages, which is not the scope of this course). A single page must be stored as a serialized Vector (java.util.Vector). Note that you can save/load any Java object to/from disk by implementing the java.io.Serializable interface. You don't actually need to add any code to your class to save it the hard disk. For more info, check this: https://www.tutorialspoint.com/java/java_serialization.htm

6) A single tuple should be stored in a separate object inside the binary file.

7) You need to postpone the loading of a page until the tuples in that page are actually needed. Note that the purpose of using pages is to avoid loading the entire table's content into memory. Hence, it defeats the purpose to load all pages upon program startup.

8) If all the rows in a page are deleted, then you are required to delete that page. Do not keep around completely empty pages.

9) You might find it useful to create a Table java class to store relevant information about the pages and serialize it just like you serialize a page. Note that to prevent serializing an attribute, you will need to use the transient keyword in your attribute declaration. Read more about that here:
https://en.wikibooks.org/wiki/Java_Programming/Keywords/transient

Meta-Data File

10) Each user table has meta data associated with it; number of columns, data type of columns, which columns have indices built on them.

11) You will need to store the meta-data in a text file. This structure should have the following layout: **Table Name, Column Name, Column Type, Key, Indexed**

For example, if a user creates a table/relation CityShop, specifying several attributes with their types, etc… the file will be:

```
Table Name, Column Name, Column Type, Key, Indexed
CityShop, ID, java.lang.Integer, True, False
CityShop, Name, java.lang.String, False, False
CityShop, X, java.lang.Double, False, True
CityShop, Y, java.util.Double, False, True
CityShop, Z, java.lang.Double, False, True
CityShop, Specialization, java.lang.String, False, True
CityShop, Address, java.lang.String, False, false
```

The above meta data teaches that there are 1 table of 7 tuples (ID, Name, X,Y,Z, Specialization, Address). There are 4 indices created on this table CityShop.

12) You must store the above metadata in a single file called ***metadata.csv***. Do not worry about its size in your solution.

13) You must use the metadata.csv file to learn about the types of the data being passed and verify it is of the correct type. So, do not treat metadata.csv as decoration! ☺

14) You can (but not required to) use reflection to load the data type and also value of a column, for example:

```
strColType  = "java.lang.Integer";
strColValue = "100";
```

```
Class class = Class.forName( strColType );
Constructor constructor = class.getConstructor( ….);
… = constructor.newInstance( );
```

For more info on reflection, check this article:

http://download.oracle.com/javase/tutorial/reflect/

Indices

15) You are required to use B+ trees to support creating primary and secondary *dense* indices. You can use any open source B+ tree index. There are several available online. Before picking an implementation, perform thoughtful testing. You can also implement your own.

16) You are required to use an R tree to support creating primary and secondary *dense* indices. You can use any open source R tree index. There are several available online. Before picking an implementation, perform thoughtful testing. You can also implement your own.

17) Once a table is created, you do not need to create any page until the inseration of the first row/tuple.

18) You should update existing relevant indices when a tuple is inserted/deleted.

19) If a secondary index is created after a table has been populated, you have no option but to scan the whole table.

20) Upon application startup; to avoid having to scan all tables to build existing indices, you should save the index itself to disk and load it when the application starts next time.

21) When a table is created, you do not need to create an index. An index will be created later on when the user requests that through a method call to **`createBTreeIndex`** or **`createRTreeIndex`**

22) Note that once an index exists, it should be used in executing queries. Hence, if an index did not exist on a column that is being queried (select * from x = 20;), then it will be answered using linear scan on x, but if an index is created on x, then x's index should be used to find if there are tuples with x=20. Hence, if a select is executed (by calling **`selectFromTable`** method below), and a column is referenced in the select that has been already indexed, then you should search in the index.

23) Note that indices should be used in answering multi-dimension queries if an index has been created on any of columns used in the query, i.e. if the query is on Table1.column1 and Table1.column2, and an index has been created on Table.column1, then it should be used in answering the query.

Required Methods/Class

24) Your main class should be called **DBApp.java** and should have the following **eight methods** with the signature as specified. The parameters names are written using Hungarian notation -- which you are not required to use but it does enhances the readability of your code, so it is a good idea to try it out! You can add any other helper methods and classes. Note that these 8 methods are the only way to run your Database engine. You are not required to develop code that process SQL statements directly because that requires knowledge beyond this course (the compilers course in 9th and 10th semester!) and it is not a learning outcome from this course!

```
public void init( );     // this does whatever initialization you would like
                         // or leave it empty if there is no code you want to
                         // execute at application startup

public void createTable(String strTableName,
                        String strClusteringKeyColumn,
                        Hashtable<String,String> htblColNameType )
                                                        throws DBAppException

public void createBTreeIndex(String strTableName,
                             String strColName) throws DBAppException

public void createRTreeIndex(String strTableName,
                             String strColName) throws DBAppException

public void insertIntoTable(String strTableName,
                            Hashtable<String,Object>  htblColNameValue)
                                                        throws DBAppException
public void updateTable(String strTableName,
                        String strKey,
                        Hashtable<String,Object> htblColNameValue   )
                                                      throws DBAppException

public void deleteFromTable(String strTableName,
                            Hashtable<String,Object> htblColNameValue)
                                                     throws DBAppException

public Iterator selectFromTable(SQLTerm[] arrSQLTerms,
                                String[]  strarrOperators)
                                                  throws DBAppException
```

Here is an example code that creates a table, creates an index, does few inserts, and a select;

```
String strTableName = "Student";

Hashtable htblColNameType = new Hashtable( );
htblColNameType.put("id", "java.lang.Integer");
htblColNameType.put("name", "java.lang.String");
htblColNameType.put("gpa", "java.lang.double");
createTable( strTableName, "id", htblColNameType );
createBIndex( strTableName, "gpa"  );

Hashtable htblColNameValue = new Hashtable( );
htblColNameValue.put("id", new Integer( 2343432 ));
htblColNameValue.put("name", new String("Ahmed Noor" ) );
htblColNameValue.put("gpa", new Double( 0.95 ) );
insertIntoTable( strTableName , htblColNameValue );

htblColNameValue.clear( );
htblColNameValue.put("id", new Integer( 453455 ));
htblColNameValue.put("name", new String("Ahmed Noor" ) );
htblColNameValue.put("gpa", new Double( 0.95 ) );
insertIntoTable( strTableName , htblColNameValue );

htblColNameValue.clear( );
htblColNameValue.put("id", new Integer( 5674567 ));
htblColNameValue.put("name", new String("Dalia Noor" ) );
htblColNameValue.put("gpa", new Double( 1.25 ) );
insertIntoTable( strTableName , htblColNameValue );

htblColNameValue.clear( );
htblColNameValue.put("id", new Integer( 23498 ));
htblColNameValue.put("name", new String("John Noor" ) );
htblColNameValue.put("gpa", new Double( 1.5 ) );
insertIntoTable( strTableName , htblColNameValue );

htblColNameValue.clear( );
htblColNameValue.put("id", new Integer( 78452 ));
htblColNameValue.put("name", new String("Zaky Noor" ) );
htblColNameValue.put("gpa", new Double( 0.88 ) );
insertIntoTable( strTableName , htblColNameValue );


SQLTerm[] arrSQLTerms;
arrSQLTerms = new SQLTerm[2];
arrSQLTerms[0]._strTableName =  "Student";
arrSQLTerms[0]._strColumnName=  "name";
arrSQLTerms[0]._strOperator  =  "=";
arrSQLTerms[0]._objValue     =  "John Noor";

arrSQLTerms[1]._strTableName =  "Student";
arrSQLTerms[1]._strColumnName=  "gpa";
arrSQLTerms[1]._strOperator  =  "=";
arrSQLTerms[1]._objValue     =  new Double( 1.5 );


String[]strarrOperators = new String[1];
```

```
strarrOperators[0] = "OR";
// select * from Student where name = "John Noor" or gpa = 1.5;
Iterator resultSet = selectFromTable(arrSQLTerms , strarrOperators);
```

25) For the parameters, the name documents what is being passed – for example htblColNameType is a hashtable with *key* as ColName and *value* is the Type.

26) Operator Inside SQLTerm can either be >, >=, <, <=, != or =

27) Operator between SQLTerm (as in strarrOperators above) are **AND, OR, or XOR.**

28) `DBAppException` is a generic exception to avoid breaking the test cases when they run. You can customize the Exception by passing a different message upon creation.

29) `SQLTerm` is a class with 4 attributes: String _strTableName, String _strColumnName, String _strOperator and Object _objValue

30) Iterator is java.util.Iterator It is an interface that enables client code to iterate over the results row by row. Whatever object you return holding the result set, it should implement the Iterator interface.

31) You should check on the passed types and do not just accept any type – otherwise, your code will crash will invalid input.

32) You are not supporting SQL Joins in this mini-project.

Directory Structure

33) Your submission should be a zipped folder containing the following directory structures/files:

```
teamname
        data
                metadata.csv
        docs
        classes
                teamname
                        DBApp.class
                        DBAppTest.class

        config
                DBApp.config
        libs
        src
                teamname
                        DBApp.java
                        DBAppTest.java
        Makefile
```

34) **teamname** is your team name! (choose any name you like!)

35) **data** directory contains the important metadata.csv which holds meta information about the user created tables. Also, it will store binary files storing user table pages, indices, and any other data related files you need to store.

36) **docs** directory contains html files generated by running javadoc on your source code

37) **src** directory is a the parent directory of all your java source files. You should use *teamname* as the parent package of your files. You can add other Java source files you write here.

38) **classes** directory is the parent directory of all your java class files. When you run make all, the java executables should be generated in **classes**.

39) **libs** directory contains any third-party libraries/jar-files/java-classes. Those are the ones you did not write and using in the assignment.

40) **config** directory contains the important configuration *DBApp.properties* which holds a two parameters as key=value pairs

```
MaximumRowsCountinPage = 200
NodeSize = 15
```

Where

```
MaximumRowsCountinPage as the name
      indicates specifies the maximum number
      of rows in a page.
NodeSize specifies the count of keys that
      could be stored in a single Node in a
      B+ tree or in an R tree.
```

41) *DBApp.properties* file could be read using java.util.Properties class

42) You can add other parameters to this file as per need.

43) Makefile is a text file for the make building utility on linux ! If you are going to provide make, have ***make all*** and ***make clean*** targets. Else, you can use maven or ant and submit maven/ant file instead.

Submission Milestones

There are two submissions in this assignment as listed below. You are required to make each submission on time/date. Below is a description of each submission.

Submission #	Date	Submission Focus	Methods to be implemented (or changed) in DBApp.java

1	March 4th, 2020, 11:50PM	create, insert
		update, delete
			`createTable`
			`insertIntoTable`
			`updateTable`
			`deleteFromTable`

2	Mar 25th, 2020, 11:50PM	B+ Tree Index and R Tree Index integration into code	`createBTreeIndex`
			`createRTreeIndex`
			`insertIntoTable`
			`selectFromTable`
			`deleteFromTable`
			`updateTable`

Note that you will resubmit methods such as **`updateTable`** twice. The first submission without an index implementation. In the second submission, you will modify the code to include handling indices.

Marking Scheme

1) +5 Each table/relation will be stored as binary pages on disk and not in a single file

2) +2 Table is sorted on key

3) +4 Each table and page is loaded only upon need and not always kept in memory.
   Page should be loaded into memory and removed from memory once not needed.

4) +2 A page is stored as a vector of objects

5) +3 Meta data file is used to learn about types of columns in a table with every
   select/insert/delete/update

6) +2 Page maximum row count is loaded from metadata file (N value)

7) +3 A column can have any of the 6 types

8) +1 TouchDate supported

9) +6 select without having any index created is working fine

10) +8 select with the existence of an index that could be used to reduce search space

11) +6 insert without having any index created is working fine

12) +8 insert with the existence of an index that could be used to reduce search space

13) +6 delete without having any index created is working fine

14) +8 delete with the existence of an index that could be used to reduce search space

15) +6 update without having any index created is working fine

16) +8 update with the existence of an index that could be used to reduce search space

17) +6 creating an index correctly for a given column whether key column or otherwise.

18) +4 saving and loading index from disk

19) +12 Inserting and deleting from index correctly.

Total: /100

Other Deductions:

Not respecting specified method signatures → -5

Not respecting specified directory structure → -2

Not submitting ant, maven or make file → -1

not performing binary search on table pages → -3

Using ArrayList instead of vector → -2