**EDUDB.NET**

EduDB.Net assignments are made available under the OSI-approved MIT license. Read the license here: http://www.edudb.net/license.html

**Part A - Version 3**

Release Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_

 Due Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Instructions***

• This assignment should be done in teams of \_\_\_\_\_\_\_ . If you cannot find a team, contact your TA: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, who will synch you with other students.

• Assignment to be submitted electronically via \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Note***

• You should be able to start in this assignment after the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ week. In the mean time, learn about the topics listed in the Tips section.

***Description***

***Overview***

In this assignment, you are going to build the first version of a mini database engine in Java. The required functionalities are creating tables, building indices, storing data, deleting data, and querying.

***User Tables***

- Each user table will be stored in one or more pages (depending on the size of the table).

- You are required to use java 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 tuple will be stored in a separate object inside the binary file.

- A file has a predetermined maximum number (N) of rows. For example, if a table has 40000 tuples, and N=200, the table will be stored in 20 binary files.

- Deletion of data will result in fragmentation within a page; do not worry about that in your solution.

***Pages***

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 the pages of all tables upon program startups. You need to postpone the loading of a page until the tuples in that page are actually needed for further processing.

***MetaData File***

- Each user table has meta data associated with it; number of columns, data type of columns, which column is a key, references between table columns and columns in other tables (to enforce constraints),…

- You will need to store the meta data in a file/data-structure. This structure should have the following layout:

 Table Name, Column Name, Column Type, Key, Indexed, References

For example, if a user creates two tables, Employee and Department, specifying several attributes with their types, etc… the file will be:

Table Name, Column Name, Column Type, Key, Indexed, References

 Employee, ID, java.lang.Integer, True, True, null

 Employee, Name, java.lang.String, False, False, null,

 Employee, Dept, java.lang.String, False, False, Department.ID

 Employee, Start\_Date, java.util.Date, False, False, null

 Department, ID, java.lang.Integer, True, True, null

 Department, Name, java.lang.String, False, False, null

 Department, Location, java.lang.String, False, False, null

- For simplicity, you can store this structure in a single file called *metadata.csv.* Do not worry about its size in your solution.

***B+Tree for Range Queries***

- You are required to use B+ trees to support creating primary *dense* indices. Note that a B+ tree stores in its’ leafs pointers (handles in Java terminology) to the data objects.

- You can use any available open source B+ tree.

- Once a table is created, you need to create a primary index on the key of that table.

- You should update existing relevant B+ trees when a tuple is inserted/deleted.

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

- 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.

- 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.

- You should use the B+ tree for range queries only.

***Linear Hashtable for value queries***

- You are required to use a linear hash table to use for value queries (i.e. looking for a specific value).

- Note that the data structure must be paged and you only create/load a page upon need.

- You can use any available open source implementation.

***Main App Class***

Your main class should be calledDBApp and should have the following methods/signature;

public void init( );

public void createTable(String strTableName,

 Hashtable<String,String> htblColNameType,

 Hashtable<String,String>htblColNameRefs,

 String strKeyColName)

throws DBAppException

public void createIndex(String strTableName,

 String strColName)

throws DBAppException

public void insertIntoTable(String strTableName,

 Hashtable<String,String> htblColNameValue)

throws DBAppException

public void deleteFromTable(String strTableName,

Hashtable<String,String> htblColNameValue,

 String strOperator)

throws DBEngineException

public Iterator selectValueFromTable(String strTable,

Hashtable<String,String> htblColNameValue,

String strOperator)

throws DBEngineException

public Iterator selectRangeFromTable(String strTable,

Hashtable<String,String> htblColNameRange,

String strOperator)

throws DBEngineException

public void saveAll( ) throws DBEngineException

// General Notes:

// 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.

//

// strOperator can either be OR or AND (just two)

//

// DBEngineException 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.

//

// Iterator is java.util.Iterator It is an interface that // enables client code to iterate over the results row

// by row. Do not implement the remove method.

//

// The method saveAll saves all data pages and indices to // disk and can be called at any time (not just before

// the program terminates)

***Directory Structure***

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

**teamname-A1**

**data**

 metadata.csv

 **docs**

**classes**

 **teamname**

 DBApp.class

DBAppTest.class

 **config**

DBApp.config

 **libs**

**src**

 **teamname**

 DBApp.java

DBAppTest.java

 Makefile

*Where*

* **teamname** is your team name!
* **data** contains the important metadata.csv which holds meta information about the user created tables. Also, it will store binary files storing user table content, indices, and any other data related files you need to store.
* **docs** contains html files generated by running javadoc on your source code
* **src** 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.
* **classes** is the parent directory of all your java class files. When you run make all, the java executables should be generated in **classes**.
* **libs** contains any third-party libraries/jar-files/java-classes. Those are the ones

 you did not write and using in the assignment.

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

MaximumRowsCountinPage = 200

BPlusTreeN = 20

 *Where*

MaximumRowsCountinPage as the name

indicates specifies the maximum number of rows in a csv/page.

 BPlusTreeN specifies the count of values

that could be stored in a B+ tree node

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

* Makefile is a make file! supporting ***make all*** and ***make clean*** targets.
* DBAppTest.java contains a set of test cases for your code. Each function should be tested separately.

***Tips***

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

• For reading a CSV file, java.io.BufferedReader and java.util.StringTokenizer are useful classes.

•Check the following references for design patterns for loading pages only when needed:

<http://en.wikipedia.org/wiki/Proxy_pattern> <http://en.wikipedia.org/wiki/Lazy_loading>

•Check the following reference for structuring your application as a set of commands:

<http://en.wikipedia.org/wiki/Command_pattern>

The command pattern is quite useful in enabling you to structure your application as a set of commands: CreateTableCmd, InsertRowCmd, etc… which then enables you to divide the work nicely between your team mates.

• You can save any Java object by implementing the java.io.Serializable interface. You don’t actually add any code to your class. For more info, check this article:

<http://www.tutorialspoint.com/java/java_serialization.htm>

• 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/>

• Note that because you are using dense indices, the location of where to actually insert a tuple in a table does not matter as the B+ tree will refer to the location of that tuple. Neither it matters whether the table/page is sorted or not.

• Deletion of a tuple has to be managed carefully. If you delete the object containing the tuple, it will affect the location of objects which follows in that page. Hence, it is better to use a tomb stone to indicate that the tuple has been deleted without actually removing the object. The choice is yours but do not overcomplicate it.

• When you are inserting, it is safer (but more expensive) to always insert into the disk page directly. This way if your program terminates suddenly (e.g power goes off), the loss is minimized. Do not be tempted into inserting into the memory data structure and postponing the saving to disk to later. If you are going to do that – use a timer thread that continuously callbacks part of your code to save the data structures from memory to disk.