German University in Cairo

Faculty of Media Engineering and Technology

CSEN604: Database II

Project 1

Instructor: Dr. Wael Abouelsaadat

TAs: Belal Medhat, Ahmed Hesham, Nouran Sadek, Basem Mikhail

Release Date: April 11th, 2021

Submission: April 30th, and May 10th

Weight & Marking

The course projects are worth 20%. Project 1 (this one) is worth 14%. No best policy. The project marking scheme is included in the last page of this document.

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.

Submission Milestones

There are two submissions in this project 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	April 30th, 2021, 11:50PM	create, insert
		update, delete
			`createTable`
			`insertIntoTable`
			`updateTable`
			`deleteFromTable`

2	May 10th, 2021, 11:50PM	Grid Index +
		Grid Index integration
			`createIndex`
			`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.

Description

In this project, you are going to build a small database engine with support for Grid Index. The required functionalities are 1) creating tables, 2) inserting tuples, 3) deleting tuples, 4) searching in tables linearly, 5) creating a Grid index, and 6) searching using the Grid index.

Note that this is a simplified data base engine – for example, you are neither required to support foreign keys nor referential integrity constraints. Only implement what is mentioned in this doc.

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.util.Date (Note: date acceptable format is "YYYY-MM-DD")

Pages

4) A page has a predetermined fixed maximum number of rows (N). For example, if a table has 40000 tuples, and if 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 because Vectors are thread safe). 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. In the case of insert, if you are trying to insert in a full page, shift one row down to the following page. Do not create a new page unless you are in the last page of the table and that last one was full.

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 Java transient keyword in your attribute declaration. Read more here: https://www.tutorialspoint.com/difference-between-volatile-and-transient-in-java

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, ClusteringKey, Indexed, min, max

ClusteringKey is set true if the column is the primary key. For simplicity, you will always sort the rows in the table according to the primary key. That's why, it is called the clusteringkey. Only 1 clustering key per table.

min and max refer to the minimum and maximum values possible for that column.

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, ClusteringKey, Indexed, min, max
CityShop, ID, java.lang.Integer, True, False,0,10000
CityShop, Name, java.lang.String, False, False, "A", "ZZZZZZZZZZZ"
CityShop, X, java.lang.Double, False, True, 0,1000000
CityShop, Y, java.util.Double, False, True, 0,1000000
CityShop, Z, java.lang.Double, False, True, 0,1000000
CityShop, Specialization, java.lang.String, False, True, "A", "ZZZZZZZZZZZ"
CityShop, Address, java.lang.String, False, false, "A", "ZZZZZZZZZZZ"
```

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. Note that if a multidimension index has been created, you will not be able to know which columns are used with which from this file. You will only know if there is an index.

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 );
```

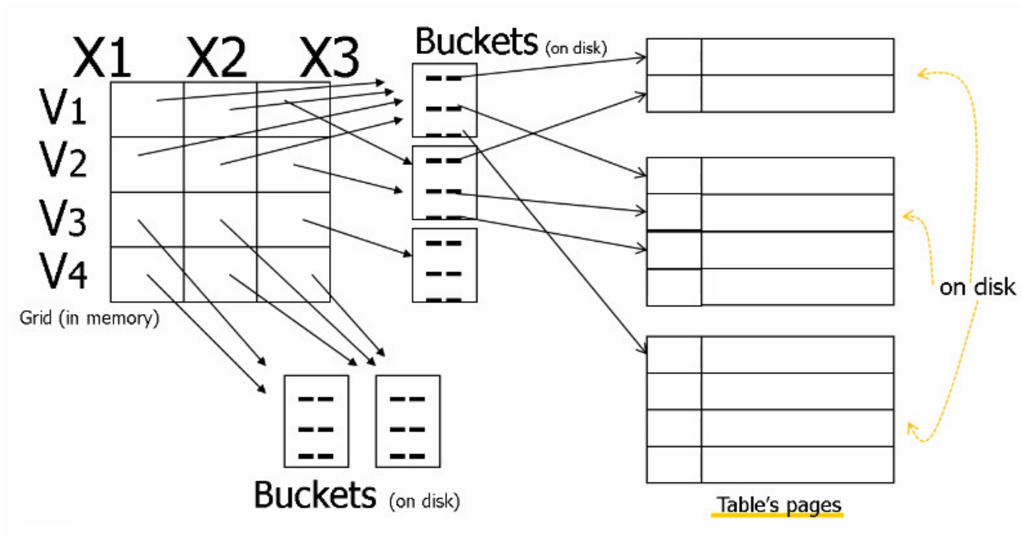
```
Constructor constructor = class.getConstructor( ….);
… = constructor.newInstance( );
```

For more info on reflection, check this article:

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

Indices

15) You are required to use Grid Index to support creating primary and secondary indices. A grid index can be single dimension (in this case a 1D array), or multidimensional one (multidimensional array). Your solution should support any number of dimensions. The Grid index should be implemented as below illustration from lecture 5 slides. Each arrow coming out from a bucket is the address on disk of the table's page which contains the row of interest. Similarly, each arrow coming from a cell in the in-memory grid is the address on disk of an index's bucket/page.



16) Each column has an associated range (minimum, maximum). You should use this range to create the divisions on the Grid Index assigned scale. You should target create close to 10 divisions if possible. For example, in the case of a String with min: "a", max: "z", the divisions can be: a-c, d-f, g-i, j-l,m-o,p-q,r-s, t-u, v-w, x-z

17) Once a table is created, you do not need to create any page until the insertion 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 `createIndex`

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 1) an index has been created on any of columns used in the query and 2) it will be efficient to use it, i.e. if the query is as "Table1.column1=somevalue **and** Table1.column2=somevalue", 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 **seven 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. Note that most parameters are passed as Strings (because that's how they will be entered by user) but you are required to convert them to correct type internally.

```
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
// following method creates one table only
// strClusteringKeyColumn is the name of the column that will be the primary
// key and the clustering column as well. The data type of that column will
// be passed in htblColNameType
// htblColNameValue will have the column name as key and the data
// type as value
// htblColNameMin and htblColNameMax for passing minimum and maximum values
// for data in the column. Key is the name of the column
public void createTable(String strTableName,
                        String strClusteringKeyColumn,
                        Hashtable<String,String> htblColNameType,
                        Hashtable<String,String> htblColNameMin,
                        Hashtable<String,String> htblColNameMax )
                                                        throws DBAppException

// following method creates one index – either multidimensional
// or single dimension depending on the count of column names passed.
public void createIndex(String   strTableName,
                        String[] strarrColName) throws DBAppException


// following method inserts one row only.
// htblColNameValue must include a value for the primary key
public void insertIntoTable(String strTableName,
                            Hashtable<String,Object>  htblColNameValue)
                                                        throws DBAppException

// following method updates one row only
// htblColNameValue holds the key and new value
// htblColNameValue will not include clustering key as column name
// strClusteringKeyValue is the value to look for to find the rows to update.
public void updateTable(String strTableName,
                        String strClusteringKeyValue,
                        Hashtable<String,Object> htblColNameValue   )
                                                      throws DBAppException

// following method could be used to delete one or more rows.
// htblColNameValue holds the key and value. This will be used in search
// to identify which rows/tuples to delete.
// htblColNameValue enteries are ANDED together
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";

DBApp dbApp = new DBApp( );

Hashtable htblColNameType = new Hashtable( );
htblColNameType.put("id", "java.lang.Integer");
htblColNameType.put("name", "java.lang.String");
htblColNameType.put("gpa", "java.lang.double");
dbApp.createTable( strTableName, "id", htblColNameType /* in addition to min
max hashtables” );
dbApp.createIndex( strTableName, new String[] {"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 ) );
dbApp.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 ) );
dbApp.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 ) );
dbApp.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 ) );
dbApp.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 ) );
dbApp.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 = dbApp.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. You should throw the exception whenever you are passed data you that will violate the integrity of your schema.

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 follow the given maven project template (available on cms/MET)

34) In the resources directory, include **a** configuration file *DBApp.config* which holds a parameters as key=value pairs

```
MaximumRowsCountinTablePage   = 200
MaximumKeysCountinIndexBucket = 100
```

Where

```
MaximumRowsCountinTablePage as the name
     indicates specifies the maximum number
     of rows in a page.
While MaximumKeysCountinIndexBucket is the
max keys you can store in a grid index
bucket.
```

Note: you can add any other configuration parameter in DBApp.config

35) *DBApp.config* file could be read using java.util.Properties class

36) If you want to add any external library, you should add it to the pom.xml file in maven as a dependency. Do not include any external 3rd jar files in your submission.

Bonus – Worth 1% of course grade

37) Add support for processing SQL statements. For that, you will need a SQL parser. You can use ANTRL, a parser generator. Here is a page which include a number of ready to use SQL grammars: https://www.antlr3.org/grammar/list.html

38) If you do the bonus, only accept statements that you can run in your mini database engine. For example: create constraint …. Should throw an exception.

39) To pass a SQL string, such as create table …, add a method with the following syntax:

```
// below method returns Iterator with result set if passed
// strbufSQL is a select, otherwise returns null.
public Iterator parseSQL( StringBuffer strbufSQL ) throws
DBAppException
```

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