German University in Cairo

Faculty of Media Engineering and Technology

CSEN604: Database II - MET Project 1

Instructor: Dr. Wael Abouelsaadat

TAs: Ahmed Hassan, Mahinar Reda, Fatma Riad, Abdulrahman Badran, Khaled Tamer Aly

https://piazza.com/guc.edu.eg/spring2024/csen604

***Release Date**:* February 18th, 2024 ***Submission**:* April 15th, 2024

Weight & Marking

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

Team Formation

This assignment should be done in **teams of FOUR to SIX (i.e. 4 <= nTeamSize <= 6).** You can make a team **cross-tutorial** and **cross major**. If you cannot find a team, still submit the teams form (posted on cms) and a TA will synch you with other students.

Team Complaints

If a team member is not doing agreed upon work, you can report that by submitting the 2 forms included in this project folder. Fill the forms, sign them and hand them to your TA, within 2 days after the project submission deadline. After 2 days, we will not accept any team complaints. In that case individual evaluation will be conducted for the whole team to determine the participation of each member.

Environment

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

Submissions

An announcement will be made later on regarding how to submit.

Questions

Direct your questions to project 1 topic on piazza: https://piazza.com/guc.edu.eg/spring2024/csen604

Description

In this project, you are going to build a small database engine with support for B+ tree index. The required functionalities are

1) creating tables
2) inserting and updating tuples
3) deleting tuples
4) searching in tables linearly
5) creating a B+ tree index upon demand
6) using the created indices where appropriate.

Note that this is a very simplified data base engine – for example, you are not required to support foreign keys, referential integrity constraints, nor joins. Only implement what is mentioned in this doc. If any issue arises that is not mentioned here, you can make your assumptions regarding implementation details. Make sure to document your code. In general, you always make some assumptions when writing software. You will never be given a project document that has every detail spelled out. That is why; we are supporting you on piazza to ask questions about alternatives. We are giving you the method names which you need to implement so that we can test your code. So, your starting point is these methods.

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 (each page is a separate binary file).

2) Supported type for a table's column is one of the following Java class types: java.lang.Integer , java.lang.String and java.lang.Double

Pages

3) A page has a predetermined fixed maximum number of rows/tuples (called *N*). For example, if a table has 40,000 tuples, and if *N*=200, the table will be stored in 200 files.

4) 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). Tuples in a page must be stored as a serialized **`Vector`** (java.util.Vector) of objects -- because Vectors are thread safe. A Page class serves you good for that. The page class should override the method **`toString()`**, and returns the tuples in the page in one string, where the records and the tuples are all separated by commas. 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

5) A single tuple should be stored in a separate object inside the Page class. That object must also override the **`toString()`** method to return a comma separated values of its content. For example if tuple consist of name="Ahmed", age=20, address="Zamalek". Calling toString() returns "Ahmed,20,Zamalek".

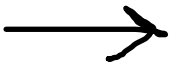
6) You need to postpone the loading of a page until the tuple(s) in that page are actually needed. Note that the purpose of using pages is to avoid loading the entire table's content into memory. Hence, it defeats the purpose to load all pages upon program startup.

7) If all the rows in a page are deleted, then you are required to delete that page. Do not keep around completely empty pages. You are not required to shift tuples between pages due to deletion.

8) 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) Since a table will be split into many pages, you need to track where the pages of a specific table on the hard disk are. You will need to store the names of the page files. For example, if a table is called Student and it has 12000 rows, and *N*=200, then the table will consist of 60 pages/files. You can device a naming scheme such as Student1.class, Student2.class, Student3.class, etc... Now you still need to store those file names somewhere. For that purpose you can create a class called Table which has an array of Strings where you store the file names. Thus, you will 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: to prevent serializing an attribute (i.e. you do not want it to be saved to the hard disk – you only want it when the object is in memory), 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



10) If there is any opportunity to use binary search in-memory, use it. That is applicable with page content as well as with Table pages. As an example if you want to find which page to load and there is a way to use binary search to do that, then use it.

Meta-Data File

11) Each user table has meta data associated with it; number of columns, data type of columns, which columns have indices built on them. For all created tables, you will store that meta data information in a text file. That file should have the following layout:

TableName,ColumnName, ColumnType, ClusteringKey, IndexName, IndexType

- **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 also called the clusteringkey. Only 1 clustering key per table.

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, IndexName,IndexType
CityShop, ID, java.lang.Integer, True, IDIndex, B+tree
CityShop, Name, java.lang.String, False, null, null
CityShop, Number, java.lang.Integer, False, NumberIndex, B+tree
CityShop, Specialization, java.lang.String, False, SpecIndex, B+tree
CityShop, Address, java.lang.String, False, AddrIndex, B+tree

The above meta data text file teaches that there are 1 table of 5 columns (ID, Name, Number, Specialization, Address). There are 4 indices created on this table CityShop; IDIndex, NumberIndex, SpecIndex, and AddrIndex.

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

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 Java reflection API 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( );
```

Read about reflection: http://download.oracle.com/javase/tutorial/reflect/

Indices

15) You are required to support B+ tree indices. You are not required to support other type of indices. You can use any open source B+ tree. Here are few candidates:

- https://github.com/shandysulen/B-Plus-Tree
- https://github.com/davidmoten/bplustree
- A more advanced version of a B+ tree (not covered in course): https://github.com/myui/btree4j

Note:

- You can also implement your own B+ tree but that is not required and you will not get any additional marks on it.

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

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

18) If an index is created after a table has been populated, you have no option but to scan the whole table to get the data read and inserted into the index.

19) 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.

20) 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`**

21) Once an index exists, it should be used in executing queries where possible. Hence, if an index does not exist on columns that are being queried, then do linear search. For example, for **`select * from T where x = 20 and y= 30 and z = 20`**, then the query will be answered using linear scanning in T. However, if an index is created on any one of the three columns (x,y, or z) , then the index should be used and the matching rows should be searched linearly for the other two values.

Required Methods/Class

22) 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 7 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 as SQL) 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
public void createTable(String strTableName,
                        String strClusteringKeyColumn,
                        Hashtable<String,String> htblColNameType)
                                                    throws DBAppException
```

```
// following method creates a conventional index
public void createIndex(String   strTableName,
                        String   strColName,
                        String   strIndexName) 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 row 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
```

Following 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);
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);
```

---

23) 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.

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

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

26) **`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.

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

28) 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.

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

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

Directory Structure

31) In a resources directory inside your project source, include a configuration file *DBApp.config* which holds a parameters as key=value pairs

```
MaximumRowsCountinPage    = 200
```

where

```
MaximumRowsCountinPage as the name indicates
specifies the maximum number of rows in a page.

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

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

33) If you want to add any external library (e.g. B+ tree .jar library), you should add it as a resource to your project. For example, in maven add it to the pom.xml file as a dependency. Do not include any external 3rd jar files in your submission.

Note: Ant, Maven, and Make are all popular automation tools used in software development for automating the build process, managing dependencies, and facilitating project management. In your IDE of choice (Eclipse, NetBeans, IntelliJ, etc… , you will have a choice which one to use).

Bonus – Worth 5% of this project grade

34) Add support for processing SQL statements. For that, you will need a SQL parser. You must 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

35) 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.

36) 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 3 types
8) +7 select without having any index created is working fine
9) +8 select with the existence of an index that could be used to reduce search space
10) +6 insert without having any index created is working fine
11) +8 insert with the existence of an index that could be used to reduce search space
12) +6 delete without having any index created is working fine
13) +8 delete with the existence of an index that could be used to reduce search space
14) +6 update without having any index created is working fine
15) +8 update with the existence of an index that could be used to reduce search space
16) +6 creating an index correctly.
17) +4 saving and loading index from disk
18) +12 Inserting and deleting from index correctly.

Total: _____/100

Other Deductions:

Not respecting specified method signatures → -5
Not submitting ant, maven or make file → -1
not performing binary search where possible → -3
Using ArrayList instead of vector → -2