Data Structures Semester Project

Due Date: Wednesday, May 9, 2018, 11:59 P.M.

> "One of the most important applications for computers is storing and managing information. The manner in which information is organized can have a profound effect on how easy it is to access and manage. Perhaps the simplest but most versatile way to organize information is to store it in tables.
>
> The relational model is centered on this idea: the organization of data into collections of two-dimensional tables…"
> -*Foundations of Computer Science, page 403*

Summary

You are building a database system from scratch. You are not using a database, you're building one! You will gain experience in:

1. building software that isn't a school toy
2. using a software component written by someone else
3. learning a technology (Maven, SQL) without instructions from a teacher
4. implementing data structures from scratch and using them (tables, BTrees)

You may NOT work on the logic of the project with other students in the class. However, you ARE allowed to share test queries on piazza, and you can/should ask any/all clarifying questions on piazza

A Suggestion Regarding How to Go About This Project

Step 1: Learn SQL (see next section)

Step 2: Think through what your Junit tests and DBTest will be and write them.

Step 3: Think through, and write down in English or in pseudo-code, how you would execute each type of query (CREATE, SELECT, etc.). What is the logic for each? What data structures does it access and/or change?

Step 4: Design the Java classes/interfaces to implement your thoughts from step 2 – what classes will you create? What methods and fields will each class have?

Step 5: code, test, debug, submit!

Students in past semesters have taken anywhere from 40 to 120 or more hours for this project. The average seems to be somewhere in the 60 to 100 hour range. It would be very foolish to wait until late in the semester to start working on this.

You already know enough, or can teach yourself with the links provided in this document, to do all aspects of this project other than the BTrees / indexing. You can get started immediately and just leave that part for after we've learned it in class.

What is SQL? How do I parse it?

- If you don't know SQL already, please see Intro to SQL: Querying and managing data and/or SQL Tutorial and/or https://www.codecademy.com/learn/learn-sql, or some other SQL tutorial.
- You must use the SQL parser located in the following git repository: https://github.com/Yeshiva-University-CS/simplesqlparser.git to parse SQL queries
    - DO NOT copy the source code of the parser into your project
    - DO NOT change the parser in any way
- You must use Apache Maven to build your project in order to properly/easily resolve the dependencies of, and use, the SQL parser.

Code Style

This is a big program you are writing. If you use bad code style, it will make it hard for you to debug and complete, and it will make it hard for me to understand your code. Read the following two resources about Java code style, and follow their suggestions:

Writing Good Java Code (IBM DeveloperWorks)
Writing Clear Code (Sedgewick)

Some specific requirements in the area of code style:

1. **No method may be more than 30 lines of code.** If you write a "Monster Method", I reserve the right to delete it and replace its logic with one line (return null;) and mark your project with that replacement.
2. **Do not create any static fields or methods** other than:
    a. public static void main
    b. any utility methods you may write
    c. any factories or builders you may write
3. **Comment your code.** Every method which is not a trivial getter/setter must have a comment explaining what it does.
4. **You MAY NOT have your classes in the default package.** You must decide on some sensible package structure and use it. For example, edu.yu.cs.dataStructures.db….
5. **DO NOT hard code any file paths** anywhere in your Java code on this project or any other project!

Testing

Your code must be tested. An untested project will not be graded. You must provide two types of tests:

1. **JUnit tests** that verify that each individual function/feature of the project is working properly
2. **A class called DBTest:**
    a. …which has a public static void main and demonstrates all the features of the system.
    b. All queries it runs, as well as their results, should be output to System.out. The output should be readable by a human such that they should not have to look at the DBTest class or your other source code and still completely understand what queries were submitted and what their results were
    c. DBTest is NOT interactive and DOES NOT require any user input – it should have all queries hard coded or load them from a file

The Subset of the SQL Language That You Are Required to Support

I. CREATE TABLE

Example:

```
CREATE TABLE YCStudent
(
      BannerID int,
      SSNum int UNIQUE,
      FirstName varchar(255),
      LastName varchar(255) NOT NULL,
      GPA decimal(1,2) DEFAULT 0.00,
      CurrentStudent boolean DEFAULT true,
      PRIMARY KEY (BannerID)
);
```

Comments:

- "`UNIQUE`", "`NOT NULL`", and "`DEFAULT`" can be specified for any column.
- For purposes of this project, `CREATE TABLE` will always specify a `PRIMARY KEY`. Primary key columns, by definition, have unique and not null values, but can NOT have default values. A primary key column is always indexed.
- The data types you must support are: `int`, `varchar`, `decimal`, and `boolean`.
    - In your implementation you should use java.lang wrapper types, not native types, to save data so that a missing value can be saved as null.

II. CREATE INDEX

Example:

```
CREATE INDEX SSNum_Index on YCStudent (SSNum);
```

Creates a new BTree index on the "`SSNum`" column in the table called "`YCStudent`", and the name of the index is "`SSNum_Index`".

III. SELECT – Gets data from the Database

- Get specific, or all, columns from a table:
    - **SELECT** *column1, column2* **FROM** *table1;*
    - **SELECT * FROM** *table1;*
- Get specific, or all, columns from a table for rows that meet a certain condition. The condition can be a complex one. The second example also includes the "distinct" clause, which means only give the unique values of column1 and column2 – no repeats.
    - **SELECT** *column2, column1* **FROM** *table1* **WHERE** *condition;*
    - **SELECT DISTINCT** *column1, column2* **FROM** *table1* **WHERE** *column3='some value'* **AND** (*column4*='some value **OR** *column4*='some other value');
- Get some data and sort (**ORDER BY**) the results in ascending (**ASC**) or descending (**DESC**) order
    - SELECT * FROM YCStudent **ORDER BY** GPA **ASC**, Credits **DESC**;

For any **WHERE** condition above, you must support any arbitrary number of conditions joined by **AND/OR**. For any comparison, you support the following comparison operators:

- =
- <> (this means "not equal")
- >
- <
- >=
- <=

IV. SELECT functions

You must support the following functions in a SELECT statement:

- Get the average of all the values in a column: `SELECT AVG(column_name) FROM table_name;`
- Count the number of values in a column: `SELECT COUNT(column_name) FROM table_name;`
- Count the number of distinct values in a column: `SELECT COUNT(DISTINCT column_name) FROM table_name;`
- Get the greatest of all the values in a column: `SELECT MAX(column_name) FROM table_name;`
- Get the smallest of all the values in a column: `SELECT MIN(column_name) FROM table_name;`
- Get the sum of all the values in a column: `SELECT SUM(column_name) FROM table_name;`

SUM does not apply to VARCHAR or Boolean columns. MIN and MAX do not apply to Boolean columns.

V. INSERT – add a new row to a table

`INSERT INTO TableName (column1, column2) VALUES (value1,value2);`

Example:

- `INSERT INTO YCStudent (FirstName, LastName, GPA, Class, BannerID) VALUES ('Ploni','Almoni',4.0, 'Senior',800012345);`

Comments:

Remember that there can be constraints on what values someone can use in a given column. For example, the primary key must be unique and not null, strings (a.k.a. varchar) can't be too long, and decimal numbers can't have more digits on either side of the decimal point.

VI. UPDATE – change values in existing rows

Examples:

- In the row where BannerID is 800012345, set the GPA to 3.0 and class to super senior: `UPDATE YCStudent SET GPA=3.0,Class='Super Senior' WHERE BannerID=800012345;`
- Set the GPA and class of every row (updates are applied to every row when there is now "where" condition in the query): `UPDATE YCStudent SET GPA=3.0,Class='Super Senior';`

VI. DELETE – remove rows from the database

Example:

- Delete rows that match a condition: `DELETE FROM YCStudent WHERE Class='Super Senior' AND GPA < 3.0;`
- Delete all rows: `DELETE * FROM YCStudent;`

Description of System Behavior

1. **SQL Queries:** Support the SQL subset described in "SQL Subset You Are Required to Support"
2. **Public API:** your API is made up of ONE method: `public ResultSet execute(String SQL)`. The result and return value of a query is:
    1. For a `SELECT`, a ResultSet containing the values that matched the query
    2. For `CREATE INDEX`, `Insert`, `Update`, and `Delete`, a ResultSet that has one column and row, whose value is "true" or "false", indicating if the query was successful. Success for these queries means that the query did not refer to any non-existent columns or tables.
    3. For `CREATE TABLE`, return an empty ResultSet with the columns that were just created.
3. **ResultSet**: your result set class must include the names and data types of your columns, as well as the rows of data that matched the query. Keep in mind that if the query only asked for a subset of the columns in a table, then your result set must include only those columns specified in the query. Your result set should be container for data only – it should not have any logic in it other than getters and setters, and possibly the ability to print itself out.
4. **Data Validation:** For all queries, you must check that all the data being inserted/deleted/updated is the correct data type for the column it is dealing with

Tables and Indices

1. Implement database tables as lists, with each element in the list being a row of data. Think of a table as a "two dimensional list."
2. Implement indices as BTrees
3. Primary Key columns must be automatically indexed without a separate CREATE INDEX query being run
4. An index must be kept consistent with what's in the actual DB table. Therefore, modifications that change any data in an indexed column must result in both the BTree and the DB table being modified
5. When executing a query which references an indexed column, you must use the index to find the data and must not go directly to the table and/or search sequentially in the table
6. A single value can appear multiple times in a column (e.g. 100 students can have GPA = 4.0). Therefore, the value pointed to by an entry in a BTree index must be able to point to many rows in the DB table, not just one

7. BTree entries must reference rows in the database tables directly. It may not store row indices or anything else other than a reference to the actual row objects.
8. The scoring system for this project has changed – ignore what you may see from past semesters. **E.G., not using the BTree to find data in an indexed column will result in the loss of 20 points.**

Additional Miscellaneous Requirements

1. For `SELECT` queries, you don't have to deal with combinations of the syntax other than those I listed earlier. However, the number of clauses in a compound `WHERE` condition, the number of columns selected or ordered by, etc. can be arbitrary.
2. Anywhere in a `SELECT` query that specific columns are selected in the examples, "*" can be used instead to specify ALL columns
3. **You may use a List implementation that is built-in to Java, but you may not use any built-in Java methods or classes to sort or search a List.** Within any built-in implementation of the java.util.List interface, you may use **only** the methods listed below. You may not use any other methods:
    1. **add(E e)**
    2. **add(int index, E element)**
    3. **get(int index)**
    4. **isEmpty()**
    5. **remove(int index)**
    6. **set(int index, E element)**
    7. **size()**
4. Your project must have its dependencies resolved, and must be compiled, using Apache Maven All work is to be done 100% on your own. later courses will give you opportunity for team work, but first you need to get yourself up to speed as an individual.
5. **You must watch piazza for any updates/clarifications about the project and act accordingly.**
6. **I may call for progress reports to be submitted at various points throughout the semester – be prepared to write something short about what you've done so far and what you are doing next**

Additional Materials to Learn More about Databases

Excellent book: A First Course in Database Systems (3rd Edition)

Disambiguating Databases (Communications of ACM) – posted on Piazza under "General Resources"

The Relational Data Model (Foundations of CS book)