Seung Han, John Dileo, Mariano Salem

CSCE315-502

Design Document - DBMS

The purpose of this project is to develop a Database Management System (DBMS) and use it in a real life application. A DBMS is a program that organizes data based on a similar trait. These traits can be a sport’s team, or a company’s list of employees. The DBMS can also manipulate this data: it can search for elements, add new members to a trait, or even create a new trait. A user need only to input a command based on their needs and the DBMS will execute them, provided they are legal commands. These commands are read by a program in the DBMS known as the parser which evaluates the commands and, if the syntax of the commands match the parsers syntax, it will execute the command; otherwise it will display either display an error or nothing, depending on how the parser is programmed.

The project is divided into two sections: Phase I and Phase II. Phase I will focus on the construction of the engine and parser of the DBMS. Phase II goal is to apply the DBMS constructed in phase I into a meaningful application that utilizes databases. Developing of this project will provide understanding of the concepts of organizing and managing data as well as user input.

The project calls for three entities: the parser and engine which are both within the DBMS engine, and the application of the database. The parser receives the users input and translates the commands into meaningful translations for the program to use. The input to the parser follows a DML format with defined grammar such as "CREATE TABLE [table name] VALUES FROM [column names and types]". The parser should be able to identify and handle the numerous possibilities of inputs and have a certain response for each input from the application. The engine will be able to construct and mange a user-defined *Table* class. The Table class contains data and fields specified by the user; it should be able to handle any case of data from cars to shoes depending on the user’s applications. The parser and engine work together to form a template to interpret, manage, and organize the application’s data. The application of the database places the DBMS into a real-life situation. In the case of our project, our application will be centered on the records of a hospital, specifically the patients, doctors, and procedures for a certain hospital.

**Parser**

*Usage:*   
 The parser will be implemented so that the inputs from the application will be tokenized, and the tokenized data will result in a certain response from the DBMS engine. The specifics of the parser will be discussed later but it will implement stringstreams to tokenize the inputs. Currently, our parser will go through if-statements or switch statements to determine the appropriate response for the engine.

*Model:*



*Interaction:*

The parser will take input from the application and produce an output that will be sent to the engine which the engine will respond accordingly. The diagram will be attached at the bottom.

**Engine**

*Usage:*

The engine will consist of a Table class that we have created. It will be applicable to all types of data and will be used to neatly organize and manage the data from the application. The Table class will follow the guidelines addressed below (roughly):

* String name – the name of the table
* Vector<string> keyColumns – the names of the columns the key uses
* Vector<Column> columns – our columns using a user-defined Column struct
* Table(String name, vector<string> keyColumns) (the constructor) – this will be called by the engine under something like createTable()
* Void addColumn(string name, string type) – this adds a column to the table; this will add another key into the table
* Void addRow(vector<Datum> rowData) – this adds a row to the table; this essentially adds another entry into the table and will be called by the engine under something like addEntry()

The Table class will be further updated as we progress further into the project.

In our Table class explanation, we used a vector of a user-defined struct named Column. The Column struct contain three variables: type, name, and data. The Column struct will follow the guidelines addressed below:

* String type – It will contain the name of the type the column deals with. For example, if the column is “Age”, the string *type* will be “int”.
* String name – this contains the name of the column such as “Age” and “Gender”
* Vector<Datum> data – this contains the data of the column

The Column struct uses a string to define its type to simplify and unify the different types we may encounter.

In both the Table class and Column struct, the “data” is of type Datum which is also user-defined. It contains three elements: two variables named number and str, and a function called toString(). The Datum class will follow the guidelines addressed below:

* Double number
* String str
* String toString()

The peculiar thing about the Datum class is that only one of the variables will be used: number or str. Since every representation of any data can be simplified into a number or a string, the parser or engine will decide whether it is a number or a string and initialize accordingly. In order to print the number variable, we will simply call toString() to change the double into a string.

Although we do not have the query functions in place, there will be a PrimaryKey class. The class is used to uniquely identify an entity and will be used in getEntityData(PrimaryKey key). The PrimaryKey class will follow the guidelines addressed below:

* String table – the table the key is being used on
* Vector<string> columnNames – the columns which would have this keys values
* Vector<Datum> values – the values of the key (unique to an entity)

Again, we have not fully incorporated PrimaryKey into the engine but definitely will be in the future.

The engine will be constructed into a class of its own named Engine. The engine will interpret the data from the parser and create and modify tables accordingly. The Engine class will follow the guidelines addressed below:

* Vector<Table> tables – stores tables
* Void createEntityTable (string name, vector<string> keyColumns) – creates a non-relation Table that will be added to the vector of Tables. The constructed table will be empty initially
* Void createRelationTable(string name, string table1, string table2)
  + Name – the name of the relation table
  + Table1 and table2 – the names of the entity tables
  + Uses the tables to add columns into the relation table based on table1.keyColumns and table2.keyColumns
* Void addAttribute(string tableName, string columnName, string type) - finds a certain table and adds a column (calls addColumn())
  + The parser is responsible for calling this once for each column to the entity table being created
  + This will only be used for the entity tables because the relation tables will not need to add columns of any sort since it is a “completed” table upon its birth
* Void addEntity(string tableName, vector<Datum> rowData – adds an entity to the table (calls addRow())
  + Parser assembles the values of rowData from the arguments passed to it via the “INSERT INTO” command

There will be more query functions added in the Engine class to get information from the tables and pass it to the parser.

*Model:*





*Interaction:*

All the classes mentioned in the DBMS engine are related. If you look at the model, you can see the relation between these classes more clearly.

**Application**

*Usage:*

The application of the database will be based on records of a hospital, specifically patients, doctors, and procedures that are available at the hospital. The patients will be defined by their name, ID, gender, and age. The doctors will be defined by their name, ID, department, and specialty. The procedures will be defined by their name, ID, cost, and success rate of a certain procedure. Each of the categories such as name and ID will serve as keys for the columns in the entity and relation tables. The relationships that will exist will be which patients need which procedures and which doctors are assigned to which patients. The relation tables that will be created will simply have two columns that contain the IDs of the patients, doctors, or procedures (according to the certain relation tables).

*Model:*



*Interaction:*

As the diagram above shows, the entity tables (Patients, Doctors, and Procedures) will be used to create relation tables that will help organize data and deliver a more concise and meaningful table other than a table of simply patients and their characteristics.

**Benefits:**

1. Our Table class (as far as our group can tell) accommodates for almost every case possible
2. Our classes are quite general in their functionality. They can be used for any sort of database
3. Our design does not rely on a multitude of if-statements or the like to accommodate for different types of data that the engine might encounter
4. Our application is not too complex or convoluted which will make managing the data easier (obviously)
5. No part of our classes was hardcoded to accommodate for our application

**Risks/Issues:**

1. We do not have an error handler anywhere
2. We have no exactly finished our parser so it will be hard to predict all the possible issues we may face
3. There is a lot of duplicate data the way our classes are right now
4. Code is not optimized

**Assumptions:**

1. The application will only give appropriate input (since we don’t have an error handler)
2. The data is not overly huge so that we do not face huge runtimes