Design Document

==========

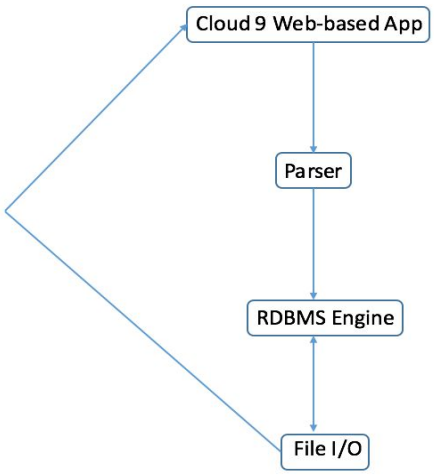
Purpose of the Project:

The purpose of this project is to create and implement a generic relational database management system. This database is designed as a sports management system. It will store information based on the sport, the player, the team, and a record of the team's wins, losses, and ties. The player's given attributes include, but are not limited to: name, age, points scored, and jersey number. The team's attributes include, but are not limited to: team name, wins, and losses. Lastly, the sport's attributes include, but are not limited to: sport name, number of players on the field, and playing surface.

High Level Entities

The picture below represents the Entity-Relationship diagram. This is simply a picture of how the information in our database will all work together.

Below, we have the visual representation of the High Level Entities.



The process starts with the interactive system when the user decides to input data. From there, the parser will accept the pseudo-SQL commands generated by the interactive system. The parser will then parse those commands and call the methods/functions from the engine. ~~The engine will then determine if the java commands are valid and execute accordingly.~~ Lastly, the engine passes all of the data to the json file where we will keep a record.

Engine

Purpose:

The engine is used to complete the bulk of all operations. The engine will receive input from the parser. This input will come in as a string and await execution within the engine. If the input is not valid, the parser will throw an error. ~~Furthermore, the engine will not work directly alongside the interactive system. Meaning that, the engine will work in between the json file and the parser. The engine does all of the operations (insert, delete, select, etc.).~~ The engine will not work directly with the interactive system. It simply works in conjunction with the parser and serial file.

Low Level Design:

~~The low level design will consist of the three objects: Player, Team, and Sport. Each entity will have a corresponding class file to be used within the engine. Inside each class file, the given attributes for each entity will be declared as members. This approach will help to simplify the code as well as increase readability. The attributes for each entity are shown below:~~

- ~~Sport:~~
  - ~~Name~~

    - ~~Number of players on field~~
    - ~~Playing surface (gym, field, etc.)~~
- ~~Teams:~~
    - ~~Team name~~
    - ~~Location~~
    - ~~Wins~~
    - ~~Losses~~
    - ~~Ties~~
- ~~Player:~~
    - ~~Name~~
    - ~~Age~~
    - ~~Jersey number~~
    - ~~Points scored~~

The engine consists of Engine.java and Table.java. The Engine class is composed of a set of static functions listed below:

- Static HashMap<String, ArrayList<Vector<String>>> rdbms_tables_container = new HashMap<String, ArrayList<Vector<String>>>()
    - This is the actual representation of the RDBMS. This is the general structure that holds each table (ArrayList<Vector<String>>), which in itself holds each row of data (Vector<String>)

- Public static void main(String[] args)
    - This function creates a TestList object that is then used to test all of the assorted functions.

- Public static void createTable(String table_name, String[] attributes, String[] p_keys)
    - This function essentially creates an empty table and initializes the first row with the table's ID and the given attributes to be recorded in the structure.

- Public static void dropTable(String table_name)
    - This function takes in the name of table, locates said table, and sets it equal to NULL. This approach was taken because the java garbage collector removes all unreferenced objects.

- Public static void insertRow(String table_name, String[] values)
    - This function creates a Vector<String> based off of the string array of values and then adds the vector into the attribute_table ArrayList of the given table.

- Public static void updateRow(String table_name, String row_id, String[] values)
    - This function checks to see if the table first exists. If so, the non-static updateRow function in the Table class is called in order to change values in the table.

- Public static void deleteRow(String table_name, String row_id)
    - This function checks to see if the given table exists. If so, the non-static deleteRow function in the Table class is called in order to remove a Vector<String> of values from the attribute_table ArrayList.

- Public static void renameTable(String old_table, String new_table)
    - This function takes in two strings that represent the name of the table to be changed, and the name to change the old table to. It does this by creating a temporary table, calling the set function with the new name, removing the old table from the DBMS HashMap, and then adding the new table to said HashMap.

- Public static void show(String table_name)
    - This function takes in a table name of type string and checks to see if it exists. Next, it calls the non-static show function which is used to print the results of the table.

- Public static void selection(String attribute, String operator, String qualificator, String table_name, String new_table_name)
    - This function is designed to return the data that meets one or more conditions from a given table.

- Public static Boolean getOp(String operator, String a, String b)
    - This function takes in an operator and two other values. It then translates it to executable code and returns the outcome of the actual operation.

- Public static void projection(String table_name, String new_table_name, String[] new_attr)
    - This function creates a new table composed of a subset of attributes of a given table. It does this by accepting a list of the wanted attributes as input from the parser. Next, it creates a new table with those given attributes. Lastly, it goes through the old table and pulls out all of the relevant information and inserts them in the newly created table.

- Public static Vector<Integer> getIndices(String table, String[] new_attr)
    - This function accepts a table and a list of attributes in as strings. Next, it creates a Vector<Integer> to hold the indices of each of the corresponding attributes in the array.

- Public static void setUnion(String new_table_name, String table_name1, String table_name2)
    - This function creates a new table with the combined subsets of attributes between the first and second table. From there, it finds the entities that the two

tables have in common, combines their data, and adds them to the new table that was created.

- Public static void setDifference(String new_table_name, String table1, String table2)
    - This function creates a new table with the same set of attributes of the two pre-existing tables being passed in as arguments. Next, it analyzes the elements of the first elements and checks to see if said elements exist in the second table. If the element does not exist in the second table, said element is added to the new table.

- Public static void crossProduct(String table_name1, String table_name2)
    - This function takes in two tables and returns a new table, with every combination between the two sets of data.

- Public static void naturalJoin(String table1, String table2)
    - This function takes two tables in as arguments. It then analyzes each table and finds the entities that they have in common. From there, a new table is created with a combination of the two sets of attributes. Note that only the entities that the tables have in common are added to the new table.

- Public static void writeTable(String table_name)
    - This function writes all of the tables data to a serial file. This is done so that none of the data is lost upon termination of a session.

- Public static void readTable(String table_name)
    - This function is used in order to retrieve the stored data from the previous session. The data is taken from the serial file and used to recreate the database.

The Table class is composed of the following:

- ArrayList<Vector<String>> attribute_table = new ArrayList<Vector<String>>()
    - This is the representation of a relational table. The rows are represented as each element in the ArrayList, and each column is represented as an element in the Vector<String>.

- Vector<Integer> p_key_indices = new Vector<Integer>()
    - This data structure is used to hold the indices of the needed attributes. Those attributes are then used to create a primary key for the sets of data.

- String[] attributes
    - This String array is used to hold a list of all of the attributes used in a table.

- String[] primary_keys
    - This String array holds a list of all of the primary_keys used in a given table.

- String table_name
    - This variable holds the unique table name that is used to identify the table from the rdbms_tables_container.

- Table(String t_name, String[] attribute_list, String[] p_keys)
    - This is the constructor to the Table class. This method accepts the necessary information as arguments and initializes all of the members with said data. Another key feature of the constructor is that it initializes the attributes table with the first row (which consists of the table name and a list of the attributes to be recorded in the table).

- Public void deleteTable()
    - This function essentially just sets the attribute table to NULL. By doing this, the java garbage collector will remove it from memory.

- Public String getPKey(String[] values)
    - This function takes in an array of attributes as input. Next, it refers to the p_key_indices vector to find the values necessary to create a primary key.

- Public void addRow(Vector<String> new_row)
    - This function takes a Vector<String> in as input and uses the built in ArrayList add function to add it to the attribute_table member of the Table object.

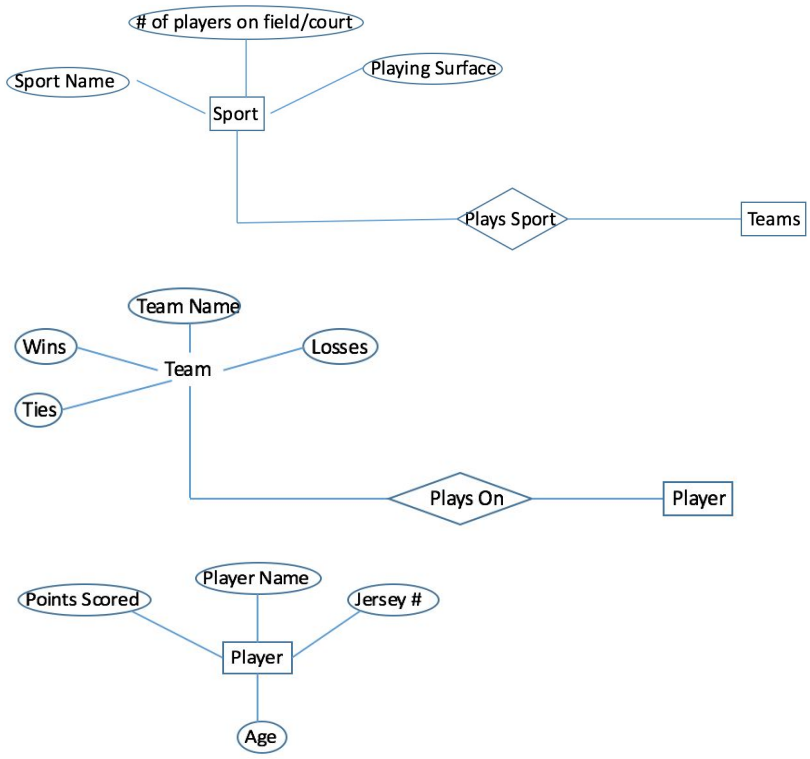
- Public void deleteRow(String row_id)
    - This function first checks to see if the row exists. Next, it uses the ID to locate the row in the attribute_table and uses the built in ArrayList remove functions to delete the row.

- Public void updateRow(String table_name, String row_id, String[] values)
    - This function first checks to see if the data exists. Next, it uses the values parameter to create a new row with the updated values.

- Public Vector<String> getRow(String row_id)
    - The function searches the table for a given row. If found, it is returned. Otherwise, and empty vector is returned.

- Public void show()
    - This function prints out all of the data in a given table by iterating through the ArrayList. Note that this is a temporary function. Upon development of an interactive system, this function will no longer be necessary and will be removed.

The data structure that supports our relational database is as follows: HashMap<String, ArrayList<Vector<String>>>. There are essentially three parts to this structure. The first, and most simple, is the Vector<String>. This vector holds all of the provided data for a given entry, where the first element holds the unique ID for that entry. The possibilities of data are given, but not limited to, in the diagrams below. The decision to use a vector was based on simplicity of the data structure, while having essential functions like: add, contains, remove, etc.



The next step in the makeup of the data structure is: ArrayList<Vector<String>>. This data structure is a member of the Table class (attribute_table). The purpose of this structure and class is to actually create the relational table. The ArrayList (represents the rows of the table) is composed of Vector<String> (represents the columns of the table). This approach was taken for multiple reasons. First, ArrayList also has sufficient add, contains, and remove functions. Next, this is easy to visualize as well as iterate through. Furthermore, it has the simplicity of a two-dimensional array paired with the increased functionality of the Vector class itself. The last (and most outward) step of the database is the Hashmap. By using HashMap<String, ArrayList> we have the ability to quickly find a given table in the database. This is because the insert, delete, and lookup functions in the HashMap class all have a time complexity of O(1).
~~The main data structures in this database are hashmaps. We decided to go with hashmaps because the <key, value> setup allows for easy identification of what values are actually stored within the structure. Furthermore, hashmaps allow for quick insert/delete/lookup function calls at~~

~~just O(1). This is crucial for a database considering that managing data revolves around the ability to insert, delete, and lookup data.~~ The process of giving each entity its own key will be related to combining two static attributes to create a unique ID for each object. Lastly, the data will be written and saved in a .ser file to prevent loss of data in between sessions. The format for reading and writing this data comes from Java's class-implements-serializable support. We chose this over .json because it is compact and easy to use.

~~With all that being said, the three entity requirement has been met. Furthermore, each entity will have its own table. Each of our two relations will also be represented in table, also. This means that we will have met the five table requirement, as well.~~

Benefits/Assumptions/Risks/Issues:

The benefits of this approach rest in the concrete design. Although the design relies heavily on the engine, ~~creating player, team, and sport objects will help to organize the code and make it readable~~ the creation of the Table class helps to organize the code, as well as increase the functionality of each table. Furthermore, as said before, the time complexity of the hashmap functions allow for quick operation. The main assumption that is being made is that the time complexity of the given hashmap functions operate under O(1). Under the worst circumstances, it could take up to O(n). However, the average case states otherwise. Another assumption benefit of this implementation is the quickness of the find and add functions of the vector class. Both functions have the time complexity of O(1). Of course, these are just assumptions based on the best/average cases, however, only in the worst case will the time complexity be O(n). The same is true of an ArrayList. ~~One of the risks of this implementation include the heavy reliance on the engine. As mentioned earlier, the engine is doing all of the dirty work while the parser and interactive system simply work in accordance with it.~~ One of the risks being taken is the use of only two files. By doing this, a lot of code is being placed into each, Table.java and Engine.java. This is a risk in the sense of comprehension to anyone, other than the developers, viewing the source code. If the code was broken down even further, this might have helped simplify the design.

Parser

Purpose:

The parser is used to interpret input (pseudo-SQL commands) from the user and call correlating functions from the engine. In essence, the parser's primary objective is to translate from SQL to Java. The output is then brought back to the engine to await execution.

High Level Modules

The parser will be written in Python, because Python has excellent i/o support. The parser will consist of three primary modules: one to read in input line by line, another to parse the said input, and a third to decide which engine function to use with the parsed input and then pass the data to that function.

Low Level Design:

~~The parser will be implemented through the use of a recursive descent parser. This is because the SQL can be nested. For example, SELECT * FROM (SELECT * FROM <table>) is a valid SQL call. That is why the parser must be recursive.~~

Firstly, the parser will take in a statement from Standard Input. These statements are based on the given pseudo-SQL DML language.

Secondly, the parser will evaluate the statement. We plan to accomplish this using regex to determine if the input is creating a table, selecting a query, and so on. We will only parse a certain part of the input at a time, enabling the parser to know if it is dealing with input that will contain conditions, so it can look for those, or if it is simply creating a table, so it can look for a table name and attribute list. Additionally, there will be a case for recursion. If needed, the parser can call its regex-based parsing function again, and return tables/relations/selections/etc from that. This will be done using a switch case block, and nested if statements. This will allow us to know exactly what engine function to use.

Lastly, once we have the parsed input, we will need to send in the data to the engine. This will be accomplished by sending an array of strings to the engine, always setting the first element to the function name in the engine. All other elements will be the necessary parsed data.

Benefits/Assumptions/Risks/Issues:

Using a recursive descent parser helps to simplify the code. If done right, this will eliminate all of the tedious, time-consuming implementation of parsing each command individually. However, recursive parsers tend to be hard to implement. One of the biggest risks/issues is the lapse in understanding that can sometimes come with recursion.

Interactive System

Purpose:

The purpose of the interactive system is to allow the user to manage and manipulate data without having to deal with SQL. We have chosen to use our database as Sports Management application, so the interactive system will allow the user to create tables for players, sports, games, teams, etc. The user will also be able to create new tables based on relations, such as players on a certain team, or teams that play a certain sport, etc.

~~We have two plans for the interactive system, Plan A and a backup Plan B. Plan A is to go for the extra credit and host the interactive system online, using HTML and JavaScript to render the UI. Plan B is to simply use the terminal and have the user type out data and operations. Despite these differences, the overall idea is relatively the same - a Sports Management application.~~

~~The interactive system will serve as the bridge between the common man and our program. It will present itself as a Sports Management application, and we hope to be able to publish it to a~~

~~free hosting service that can be accessed on any browser. If that proves too difficult, we will revert to simply using a terminal-based application. If we are successful with the website, however, all the user will have to do is input the URL. From then, the user see several tables, with perhaps arrows between them to represent relations, and the user will have the option to either add new data or manipulate current data. As far as the Terminal application goes, it would simply consist of the user typing in data in a directed format (The user will not have to be familiar with any SQL grammar).~~

High Level Modules

The Interactive System will consist of only 2 primary modules: one to speak with the user and retrieve input, and the other to then compile the input into one SQL command (Retrieving the input will be taken in steps for the user's sake).

~~Low Level Design~~:

The interactive system will start by giving the user a list of basic commands. From then, it will take in input from the user, and continue to ask the user for data, until it has enough data to form an SQL command. From here, it will send the SQL command to the parser.

Something else the Interactive System will handle is the display of tables and relations. When the user asks for the system to show table, the table will be returned to the interactive system to be displayed in *that* environment. This will be accomplished by having a format() function that takes in a table to be shown and formats it to fit aesthetically on the Interactive Systems window. This will be done by having the table come in as an array of arrays, which we will then iterate through and space each element appropriately.

~~If time permits, it will be hosted on https://c9.io/, a free website hosting service. We will use HTML to create a simple canvas, and then JavaScript to render the UI that the user will interact with. The UI will primarily consist of several tables, representing the different entities and their data. There will also be a menu with a hierarchy of buttons and drop-down menus with predefined inputs. From these, the user will easily be able to input and manipulate data.~~

~~The week objectives for this portion of the assignment are as follows:~~

- ~~Create the HTML index file rendering a canvas~~
- ~~Use JavaScript to fill the canvas with tables, buttons, and menus that, when interacted with, will send SQL- based statements to be parser via API.~~

Interaction Specification

The Interactive System will be a terminal-based application. The user, after activating the system, will be greeted with a friendly list of possible commands (These will not be the SQL commands, they will be much more user friendly). An example command would be something like

$ Create

From then, the system will ask the user for a table name, a list of attributes to use as keys, and which attributes to use as the primary key. This will be the format for all of our functions, and any time the user tries to input something invalid, he will be asked to try again.

Benefits/Assumptions/Risks/Issues:

The main benefit of using a terminal program is its simplicity. It will allow the user to input raw text as opposed to having, say, some UI-based system that requires a method for each button/menu. Regarding assumptions and risks, we will have to assume that the user knows how to use a CLI. We are not assuming he knows SQL. The only risk we are taking here is the display of tables, as that requires a reformatting of the data structure (from Java Table class to Python Array). It can be done, but we will need to handle it carefully.

~~One of the hardest challenges to tackle with this will be the API. Since we will be using JavaScript to send the SQL-based grammar to our Java parser, we will need to make sure it is done as efficiently as possible. If each SQL operation is represented as a button on the canvas, we will be able to then grab data from text boxes and send that to the parser. However, finding a way to aesthetically do this might be difficult.~~

~~I think in the long run, an easy-to-access website will be extremely beneficial to the end user, and since JavaScript isn't that difficult to handle, we should be fine. We just need to manage our time wisely and make sure the parser and engine function as designed here.~~