Gustavo Pedroso UIN: 423002834

Levi Clark UIN:

Terry Chen UIN: 121007055

Daniel He UIN: 620006827

**PHASE I: Database Management System Engine**

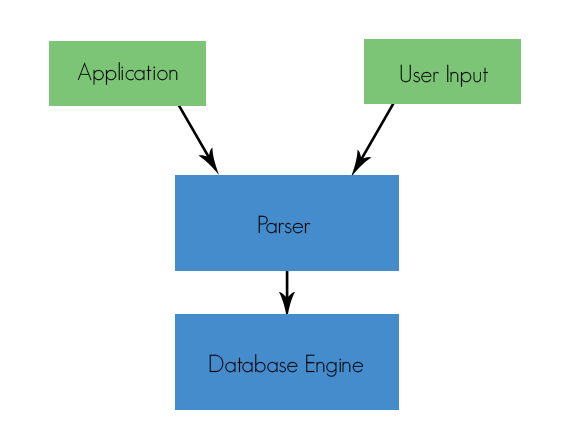
**Purpose**

To create a generic programming tool for creating applications that utilizes basic relational databases. The database management system solves the problem of managing very large amounts of information in a database in an efficient manner. It does so by storing data and organizes it using relational algebraic operations such as selection, projection, renaming, set union, set difference, cross product, and natural join. A relational database management system allows users to enter data and draw relational connections between different elements of data. This allows users to query useful information reliably and easily. Database applications will be able to run on this engine.

**High Level Entities**

There are two main high-level entities to the database management system--the parser and the engine.

Parser: The parser takes in I/O, from file, user input, or an application, and tokenizes it to ensure that the input adheres to the grammar specifications.

Engine: The engine receives tokens from the parser and calls functions accordingly to perform operations, or queries, on the data within the database. 

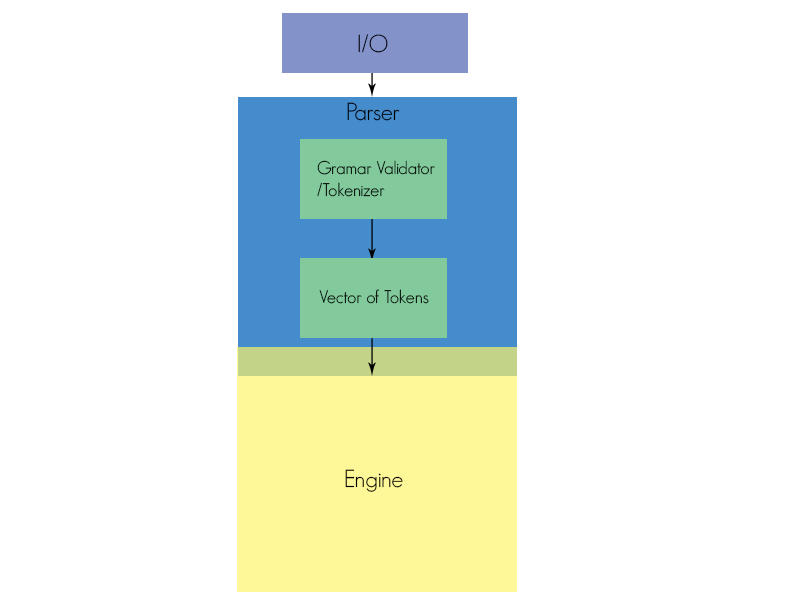
**Low Level Design**

**Engine:**

The engine receives tokens from the parser and calls functions accordingly to perform operations, or queries, on the data within the database. The engine provides functions to perform each command and the expression terms from the grammar.

The engine portion of the DBMS consists of the table class and the engine Class.

The diagram below shows the interaction of the components of our DBMS.



The table class consists of methods that define the table objects and operations that can be done on tables. The table object consists of a vector of tuples that represent columns, a two-dimensional vector of tuples that represents rows, and a string that represents the table’s name. Methods of the table class and their purposes are included in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| Method Name | Input Parameters | Output | Purpose |
| Table() | No input or a string | N/A | Constructor for a table object |
| displayTable() | N/A | N/A | Displays the table that this function is called on |
| writeTable() | N/A | N/A | Writes data into the table that this function is called on |
| openTable() | N/A | N/A | Opens a table to be edited |
| closeTable() | N/A | N/A | Closes a table to editing |
| setPrimaryKey() | String | N/A | Sets the primary key of a table to the input string |
| removePrimaryKey() | String | N/A | Removes primary key status of a key of a table |
| addColumn() | Tuple | N/A | Adds a column to the table |
| addRow | Vector<tuple> | N/A | Adds a row to the table |
| getTableName() | N/A | String | Gets the name of a table |
| getColumnNames() | N/A | Vector<tuple> | Gets the column names of a table |
| getColumnIndex() | String | Vector<tuple> | Gets the column index values of a column in the table |
| getRows() | N/A | Vector<vector <tuple> > | Gets the rows of a table |
| getRow() | int | Vector<tuple> | Gets the row at the index of a table |
| getColumnValues() | int | Vector<string> | Gets the column values at the index of a table |

The Engine class consists of a vector of tables that serves as storage for all of the tables in the DBMS engine. The class also has methods that perform relational algebra functions on the tables in the database. The table below includes all of the methods and their functions.

|  |  |  |  |
| --- | --- | --- | --- |
| Method Name | Input Parameters | Output | Purpose |
| Engine() | N/A | N/A | Constructor for engine object |
| createTable() | String, vector<tuple>, vector<string> | N/A | Given a table name, creates a table with column values from input vector<tuple> and primary keys set to items in vector<string> |
| dropTable() | String | N/A | Given a table name, find a table in the database and remove it |
| compareTables() | String, String | bool | Given two table names, find the tables with the corresponding names and return true if they are the same and false if they are not |
| selection() | Five Strings | N/A | Given a table name, an output table name, operator, column value, and attribute perform the selection operation on table |
| projection() | String, String, vector<string> | N/A | Given a table name, an output table name, and a list of column names, perform the projection operation on table |
| reNaming() | String, String, vector<string> | N/A | Given a table name, an output name, and a list of new names, renames all columns to the names of the input vector |
| setUnion() | Three Strings | N/A | Given a two table names and an output name, finds the union of the two tables |
| setDifference() | Three Strings | N/A | Given a two table names and an output name, finds the difference of the two tables |
| crossProduct() | Three Strings | N/A | Given a two table names and an output name, finds the cross product of the two tables |
| naturalJoin() | String, String | N/A | Given two table names, perform the natural join operation on them |
| Update() | N/A | N/A | Updates value of a table |
| getTableList() | N/A | Vector<Table> | Returns the table list of the engine |

**Parser:**

The parser is used to read input, check for validity, and then call the function or functions that are designated by the input. Input taken in is checked for a number of keywords that represent each function that can be called. If the keyword checks are passed, the input is converted into a list of tokens. The tokens are then scanned again to make sure that no invalid symbols, uneven parenthesis, or other grammatical errors are present. If an invalid artifact is found, the list of tokens will be rejected and an error will be produced. Otherwise, it continues into the engine and the tokens are taken in as parameters for the function(s) that will be called. If conditions are found in the token list, they are taken in and converted into a recursive tree-like structure for ease of use. Each parser has its own engine object that contains all of the methods of the engine class. Major methods of the parser are included in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| Method Name | Input Parameters | Output | Purpose |
| readFromFile() | String | bool | Reads in a file and returns true for success and false otherwise |
| writeToFile() | String | bool | Writes to a file and returns true for success and false otherwise |
| Parse() | String | N/A | Parses the input string |

**Benefits, Assumptions, Risks/Issues**

Benefits:

The benefits of the parser/engine design means that we have a database system that can reliably and efficiently manage information.

Risks/Issues:

If large datasets are used, then main memory will become full.

Assumptions:

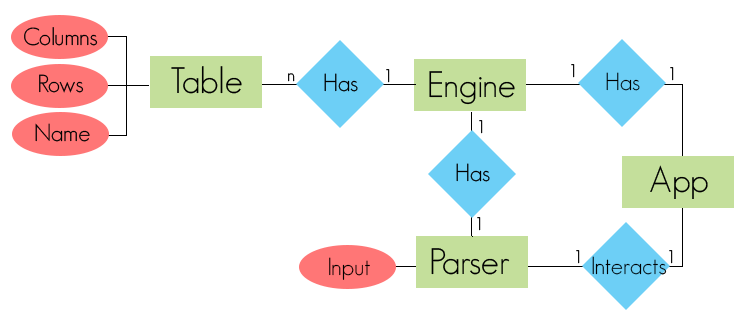
We assume that small datasets will be used because everything will be held in the main memory.

**PHASE II: Database Application**

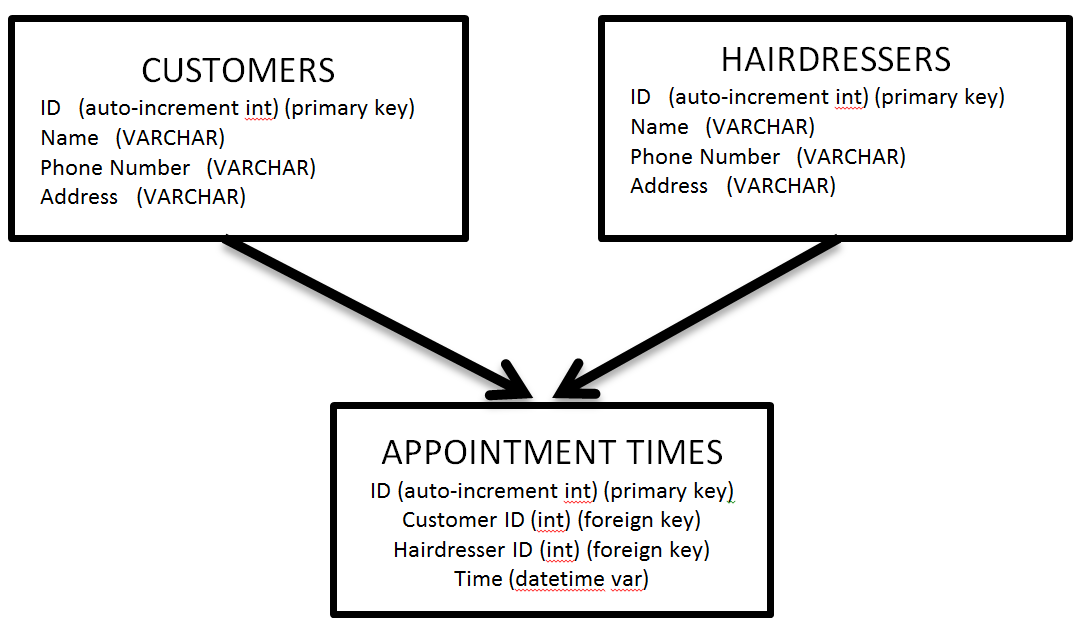
**Purpose**

The purpose of the database application is to provide an application that will run atop the database engine that will be an interface for the user to use the database. This program provides the bulk of the user interface, allowing the user to display menus, enter data and commands, and show results. With the interface, it allows the user to easily perform operations on the database without having to know all the details of database queries and commands. With this application, the database application allows a hairdresser company to manage hairdressers, customers, and appointment times.

**High Level Entities**

User Interface provides an interface for the user to easily interact with the database engine such as performing operations or query data. It allows the customer to enter in his/her information, select a hairdresser, select an appointment time, and determine the cost. The following Entity-Relationship diagram shows the interactions between the individual parts of the DBMS program.

The diagram below shows interaction between the three entities of the application.



**Low-Level Design**

The low level design contains three entities-- customers, hairdressers, and appointment times. It also specifies their relations. The Application has a menu that represents parser commands as options for ease of use. The table below contains major methods in the Application.

|  |  |  |  |
| --- | --- | --- | --- |
| Method Name | Input Parameters | Output | Purpose |
| App() | N/A | N/A | Constructor for App object, creates three tables, Hairdressers, Customers, and Appointments, and adds them to the database |
| displayMenu() | N/A | N/A | Displays the App Menu |
| addHairDresser | Three Strings | N/A | Given a name, phone number, and address, add a hairdresser to the database |
| removeHairDresser() | String | N/A | Given the name of a hairdresser, remove him/her from the database |
| showHairDressers() | N/A | N/A | Shows all the hairdressers in the database |
| showCustomers() | N/A | N/A | Shows all the customers in the database |
| addCustomer() | Three Strings | N/A | Given a name, phone number, and address, add a customer to the database |
| addAppt() | Four Strings | N/A | Given a customer ID, hairdresser ID, time, and date, add an appointment to the database |
| removeAppt() | String | N/A | Given the time of an appointment, remove it from the database |
| showAppts() | Two Strings | N/A | Given a start date and an end date, show all appointments in between |
| showApptsByCustomer() | Three Strings | N/A | Given a customer ID, a start date, and an end date, show all appointments of a given customer in the time period |
| showApptsByHairdresser() | Three Strings | N/A | Given a hairdresser ID, a start date, and an end date, show all appointments to a given hairdresser in the time period |
| showAppointments() | N/A | Vector<Table> | Shows all appointments |

**Benefits, Assumptions, Risks/Issues**

Benefits:

The benefits allow users to easily use the database in a modern application that limits user input error through a UI format. It also allows the customers to interact with the hairdressers and allows a convenient way to schedule appointments rather than using phone calls or paper.

We minimize risk by preventing users from doing terrible things like dropping a table.

Risks/Issues:

If users attempt to set up appointment times simultaneously for a same time slot then there might be a conflict if the hairdressers are the same (race condition). We minimize risks with a graphical interface because user input is limited.

Assumptions:

We assume that hairdressers will not be double booked for identical appointment times.