**Team 9:**
Dmitry Ionan
Lincoln Harris
Shayan Sayahi

# Design Documentation - Database Management System

Version 1

## I. Purpose

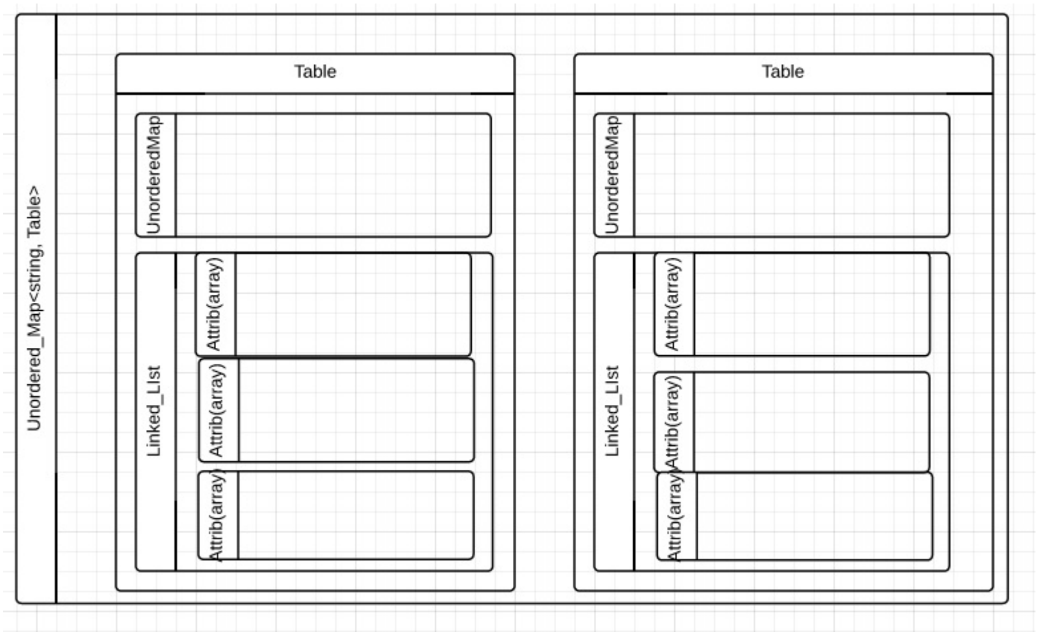
For phase 1 of the project our team is going to create a data manage language (DML) to be used as a database management system (DBMS). The manipulation of data stored in a database is a very common problem since almost every business keeps a database of information with the intent to be used at some point. Since a major constraint on this project is time (four weeks), our DBMS will be based on relational algebra, and its six primitive operations: Selection, Projection, Renaming, Set Union,Set Difference, and Cross Product. The DBMS will also be able to perform queries and commands. A query will act as a view and does not change any of the data or store any of the relations made during the query, whereas a command will update and change the data permanently.

Since we are creating a language, our DML will be able to be implemented as an application for any business that has data base on relations. This particular application provides airports, airlines, and travel agencies with means to store and manipulate passenger, aircraft, and airline entries.

## II. High Level Entities

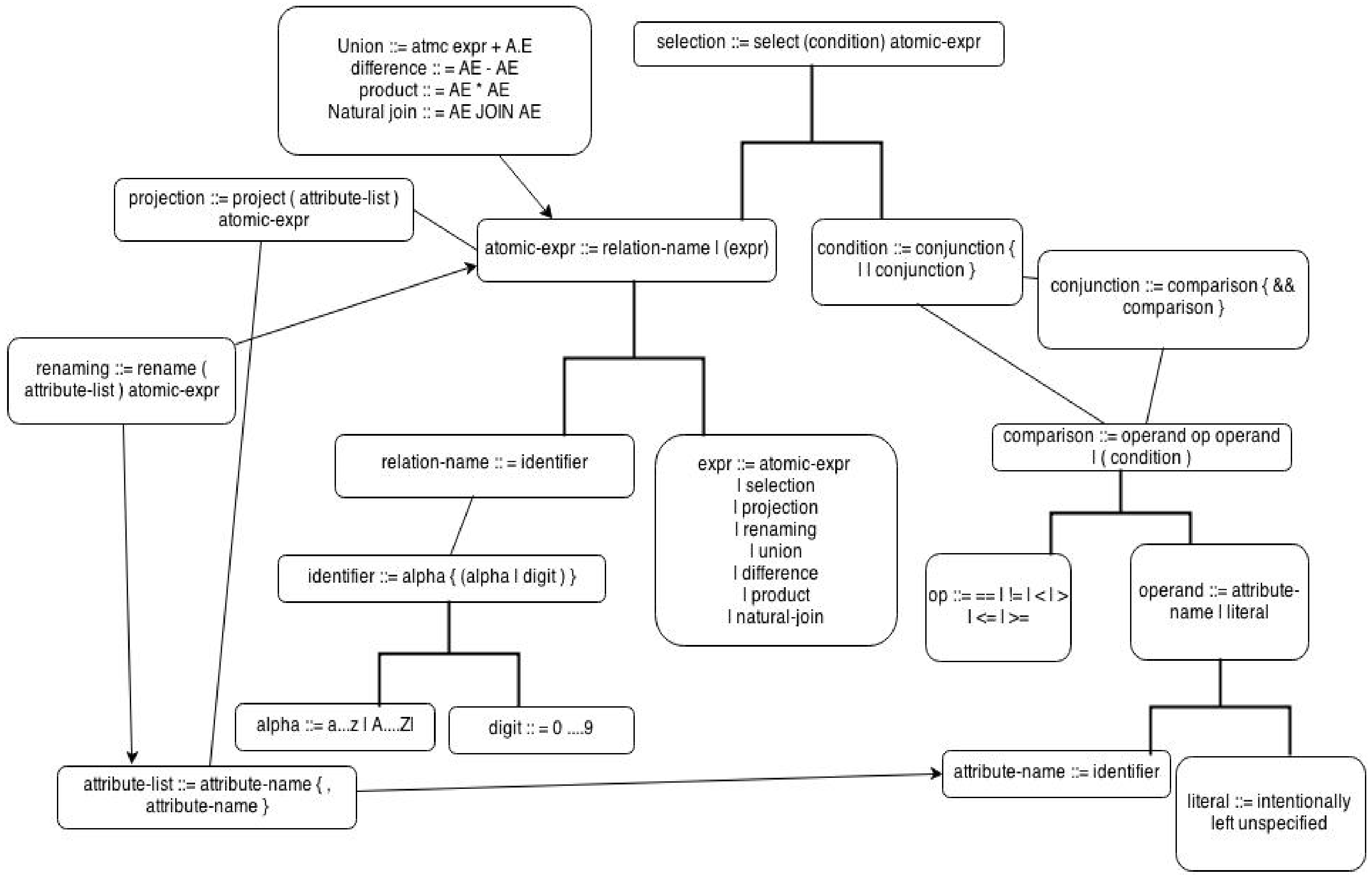
There are three primary sections to the high level design. The Database Management System and the containers that it uses and their relationships is one, the parser which interprets the users input is the second, and the last is the methods that take parsed data and interact with the database containers modifying/adding values.

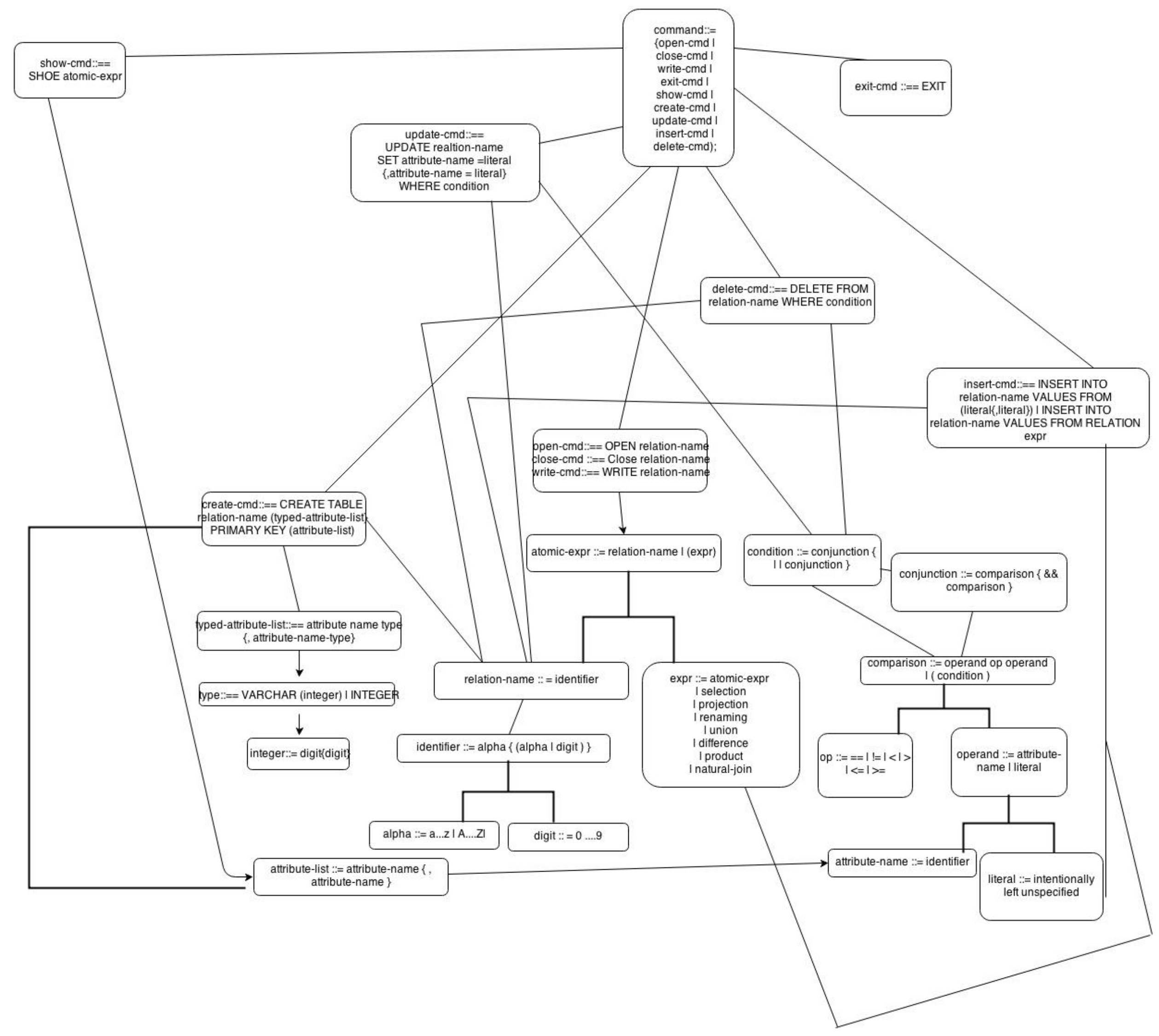
The DBMS containers can be explained in groups. These groups are made up of different C++ containers, recursively holding more containers. The image below helps show the grouping methodology used in the design of the DBMS.



- Unordered_Map holds Entity Sets/Tables.
- A Table/Entity Set has Unordered_Map of pair for attribute name, attribute type.
- A Table/Entity Set has a Linked_List, each index points an Array.
- The Array holds the values for each attribute of that entity.

The parser is the second component to the DBMS system. The parser will be a hand-written recursive-descent parser, which will be fed with a stream of tokenized input. The diagram below shows the relationships of the queries and their components:


Union ::= atmc expr + A.E
difference :: = AE - AE
product :: = AE * AE
Natural join :: = AE JOIN AE
selection ::= select (condition) atomic-expr
projection ::= project ( attribute-list ) atomic-expr
atomic-expr ::= relation-name l (expr)
condition ::= conjunction { l l conjunction }
conjunction ::= comparison { && comparison }
renaming ::= rename ( attribute-list ) atomic-expr
comparison ::= operand op operand l ( condition )
relation-name :: = identifier
expr ::= atomic-expr l selection l projection l renaming l union l difference l product l natural-join
identifier ::= alpha { (alpha l digit ) }
op ::= == l != l < l > l <= l >=
operand ::= attribute-name l literal
alpha ::= a...z l A....Zl
digit :: = 0 ....9
attribute-list ::= attribute-name { , attribute-name }
attribute-name ::= identifier
literal ::= intentionally left unspecified

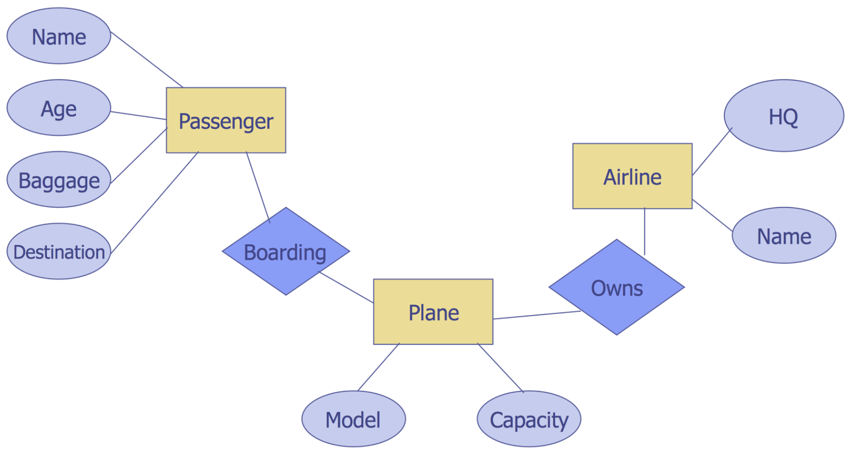

command::= {open-cmd I close-cmd I write-cmd I exit-cmd I show-cmd I create-cmd I update-cmd I insert-cmd I delete-cmd);
show-cmd::== SHOE atomic-expr
exit-cmd ::== EXIT
update-cmd::== UPDATE realtion-name SET attribute-name =literal {,attribute-name = literal} WHERE condition
delete-cmd::== DELETE FROM relation-name WHERE condition
insert-cmd::== INSERT INTO relation-name VALUES FROM (literal{,literal}) I INSERT INTO relation-name VALUES FROM RELATION expr
open-cmd::== OPEN relation-name close-cmd ::== Close relation-name write-cmd::== WRITE relation-name
create-cmd::== CREATE TABLE relation-name (typed-attribute-list) PRIMARY KEY (attribute-list)
atomic-expr ::= relation-name I (expr)
condition ::= conjunction { I I conjunction }
conjunction ::= comparison { && comparison }
typed-attribute-list::== attribute name type {, attribute-name-type}
type::== VARCHAR (integer) I INTEGER
integer::= digit{digit}
relation-name :: = identifier
expr ::= atomic-expr I selection I projection I renaming I union I difference I product I natural-join
comparison ::= operand op operand I ( condition )
op ::= == I != I < I > I <= I >=
operand ::= attribute-name I literal
identifier ::= alpha { (alpha I digit ) }
alpha ::= a...z I A....ZI
digit :: = 0 ....9
attribute-list ::= attribute-name { , attribute-name }
attribute-name ::= identifier
literal ::= intentionally left unspecified

The two components of the parser:

1. The tokenizer / reading input of the user. This portion will simply take in the input and break it into all of its individual components. This can be done through either regular expression or manually written tokenizer.
2. Once all of the components of the line of text are discrete, then the query parser engine can take over. The engine will be created based on a grammar which is sampled in detail (low level design section)

The last component to the project is an application to interface with the DBMS. The application to sample the program will be a collection of Airport Data. The image below describes the relationships of attributes, entities, and relations.

Passenger(Name, Age, Baggage, Destination)
Plane(Model, Capacity)
Airline(HQ, Name)
Boarding(Passenger, Plane)
Owns(Airline, Plane)



This application was chosen because of the variety of components that are available, and the fact that are abundant amounts of relations that can be made between the components. Attributes can also be assigned easily as well as unique keys per entity.

## III. Low Level Design

As described above the project consists of three components: the engine of DBMS, the parser, and the sample application.

**The engine is made up of several containers that are within each other.**
Now to break down the high level components of our containers:

- Unordered_Map holds Entity Sets/Tables.
  The unsorted_map is being used to hold the tables because:
  + It has O(1) access time to any tables requested
  + Can hold a unique key (list of unique attrb) for each table

- A Table/Entity Set has Unordered_Map of pair for attribute name, attribute type.
  The purpose for this is to act as a reference for the searches. This unordered_map shows the index of what type of attribute is store (int, string, float etc), and it also has the

index of that attribute. This will be used to help the Linked_List grab the correct attribute value.
+ O(1) access time for attribute index & it's type

- A Table/Entity Set has a Linked_List, each node points to an Array which represents a tuple.
This is simply a list of all the entities that exist. So each linked list node points to an array of attribute values.
+ Removing an entity is easy as removing 1 node from the linked list O(1)
+ Adding is pushing a new node to the back/front of the linked list O(1)
+ Searching for tuples matching a certain query has the same best case performance of any other container, which is looping over every node O(N)

- The Array holds the values for each attribute of that entity.
This is an array holding the attributes of a specific entity
+ The index and type of attribute is provided from the Unordered_Map we used as reference - rather than having a pair in every single array component
+ An array is the simplest container to hold few values(attributes)

All of the above containers will be used to store all of the Tables, Relations and Attributes for the entire Database. Manipulation and searching/posting will be done using the above containers. The methods to access these containers will be regular functions which will be called based on the grammar of the DBMS.

**The parser** follows the recursive-descent parsing design that is described in detail in [1][1] and [2][2]

**The application** provides the client the option to create lists of *Passengers*, *Aircrafts,* and *Airlines*, along with adding, removing, and updating any member of the list. The application also provides the option to save the database and load it at user's convenience. Lists of the same type could be merged or subtracted (E.g. subtracting a list of old planes from the list of all aircrafts owned by an airline). The client could also access the list of all passenger names (project) and filter out passengers with baggage heavier than a specified value (select). The options for the application will be done using a selection menu - with user input upon specific prompts. A log will be recorded of the entire session in a separate file - all action info will be recorded into this document/text file.

[1] http://en.wikipedia.org/wiki/Recursive_descent_parser
[2] http://blogs.msdn.com/b/ericwhite/archive/2010/07/30/building-a-simple-recursive-descent-parser.aspx

**IV. Benefits, Risks, and Assumptions**
In the design process for the Database Management System we chose to use an Unordered_Map to hold all of our entity sets/ tables

**Benefits**

- For the DBMS, Unordered_Map makes searching for the correct object faster with the use of hash functions O(1)
- For the DBMS, Linked-List will make deleting an entire row of the table faster than a vector O(1)
- The attribute map makes looking up an attribute's index and type O(1)
- Using an array of strings as a tuple makes looking up a specific index O(1)
- Using strings to hold integers makes tuples homogeneous. It also allows storing integers of any size. In addition, the type of each attribute is stored in the attribute map so we eliminate the risk of errors while type casting.
- A hand-written recursive-descent parser is easier to customize and to provide more user-friendly error output.

**Risks/Issues**

- A hand-written recursive-descent parser may not be the most efficient way to parse a query compared to parser generator tools ( especially if the query list of commands is expanded in the future).
- The grammar does not exist in the declarative form in our parser, so it is harder to maintain.
- Storing integers as strings has the added cost of converting from string to int and vice versa each time their value needs to be accessed or manipulated.
- Airplane application does not seem to have an exact cartesian product application

**Assumptions**

- Not going to delete from or add attributes to a table.
- In no parts of the project do we need to provide sorted data.
- The user will use the engine query's with correct syntax