Joshua Kruger
Harrison Kurtz
Joshua Amaku

CSCE 315 Project 1 Design Document Post Project

http://315database.webs.com/

SECTION ONE

This project, “Database Management System”, exists as an exercise in CSCE 315, Progaming Studio course. Its purpose is to learn the inner workings of a SQL like system, while also gaining insight into the technical and design process behind such data bases. This project has two phases; Phase one requires a simple 6 function algebraic relational database management system (DBMS) to be constructed. Phase two of this project is to write a simple DB application that will communicate with the DBMS built in the first phase.

Phase one consists of 2 main components, a parser, and the relational database engine itself. The parser detects and forms the relations based on user input. The user input will be written in a DBML (database manipulation language) which is passed to the parser. The parser then breaks down the language into commands which are carried out by the engine itself. The breakup of language, parser, and engine allow the database to be a expendable tool, conforming to no single templated function. It is a relational database in the sense that 2 entries can be tied together by logic decided by the user, and ultimately used to store dissimilar data in a predictable and searchable form.

Phase two is a simple DB application written in C++ that will interact with the DBMS designed in phase one. The program will take user I/O and implement a custom control flow to carry out user commands and queries. The program will also provide a user interface that will display a menu, take user input, and show the results. The application for this project will be a hospital application that will schedule appointments for patients, search for patients or doctors, and provide various other operations.

SECTION TWO

Phase ONE (Diagram below)

DBMS: The programer interfaces though this part of the software. It contains the language which is used at a high level to manipulate and ultimately uncover data. The language itself will consist of easily understood commands and query options so that the end programer can frictionlessly integrate our database into the end product. Above this level, the DB application will run and call upon these commands.

Parser/Lexer: The Parser/lexer will receive the commands from the DBMS and translate them

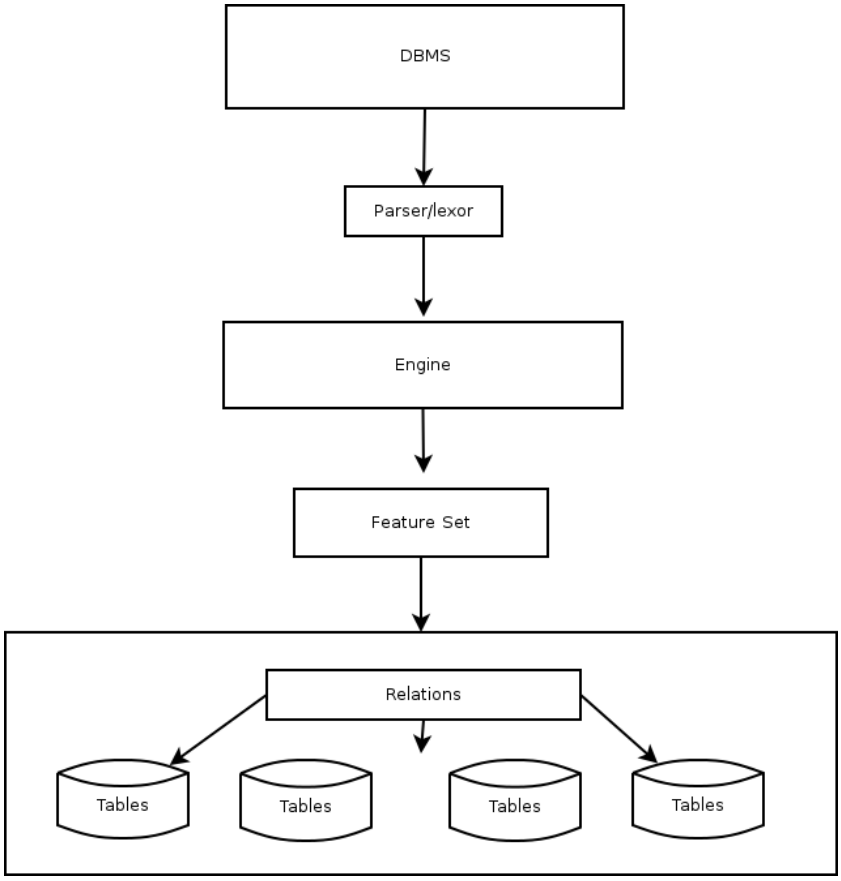
into a subset of commands that will run on our engine. The process will be done via recursive descent parser method.

Engine: Upon receiving the commands from the parser, the engine will break up the commands into calls that can be satisfied by our feature set. The engine is primarily the way in which the database is controlled.

Feature set: The feature set is the functions that are called upon by the engine. Each function is broken up into smaller, more manageable operations. The engine can call upon an number of them in any order to facilitate expendability. Within feature set, the code that controls reading, writing, combining, and other as defined (see low level) functions

Relations and Tables: The Relations will be a class which sole purpose is to keep track of data. It keeps tables of all attributes and tables, along with their relative locations. In this way, a newly created table only has to look through the relations list for links to its contents.

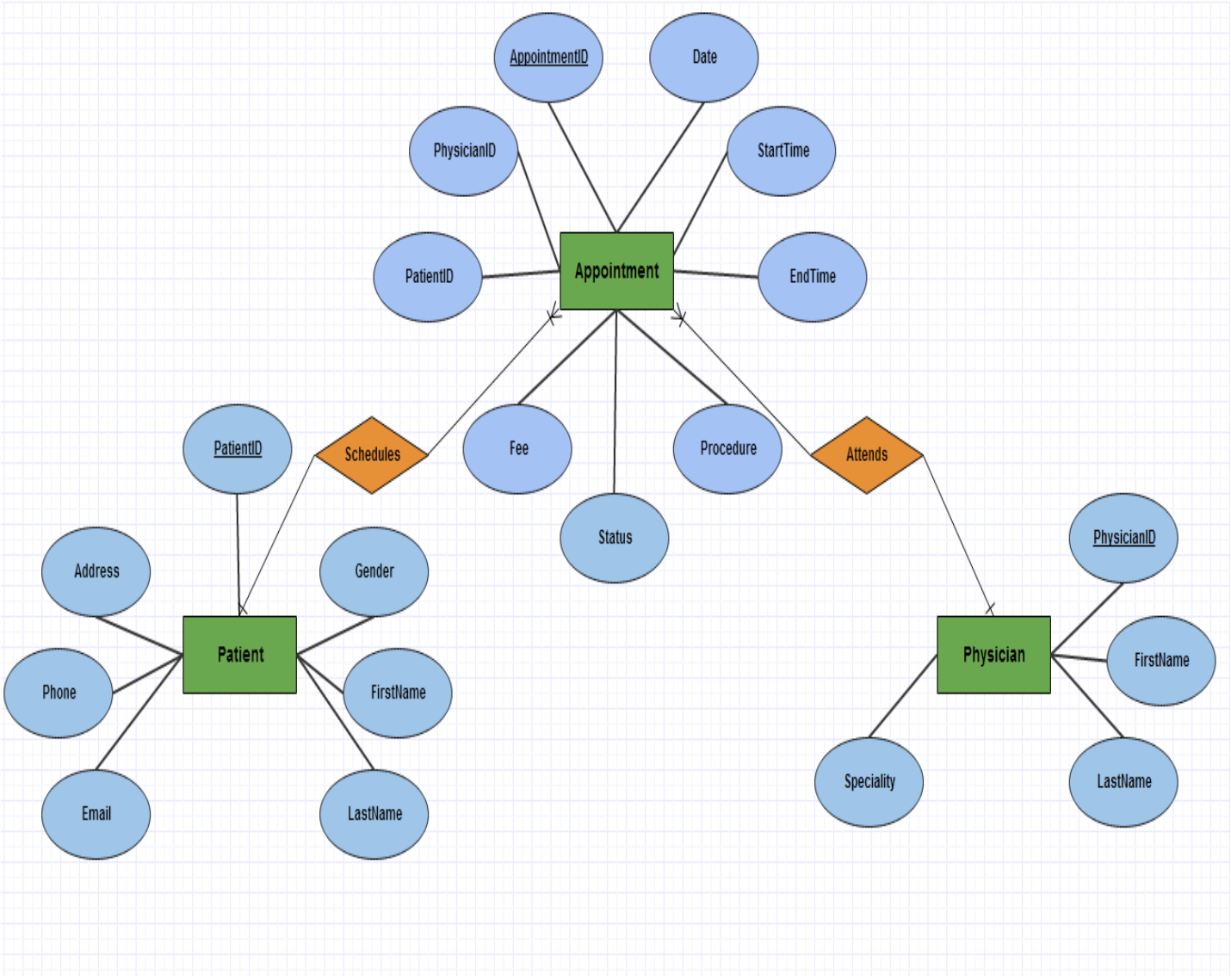
Tables: The tables themselves consist of links to attributes. These are the entities that are created when the user summons a new table. This is what allows the relation part of relational database. See low level explanation for more details.



Joshua Kruger
Harrison Kurtz
Joshua Amaku

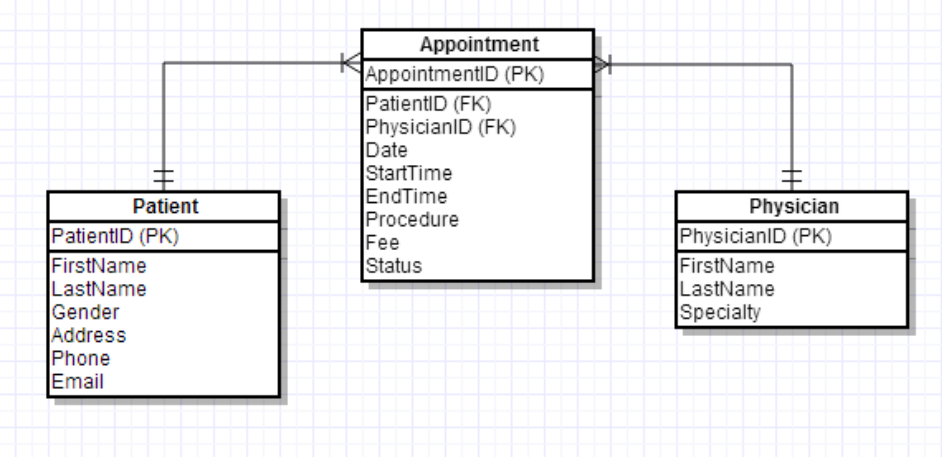
Phase TWO

ER Diagram



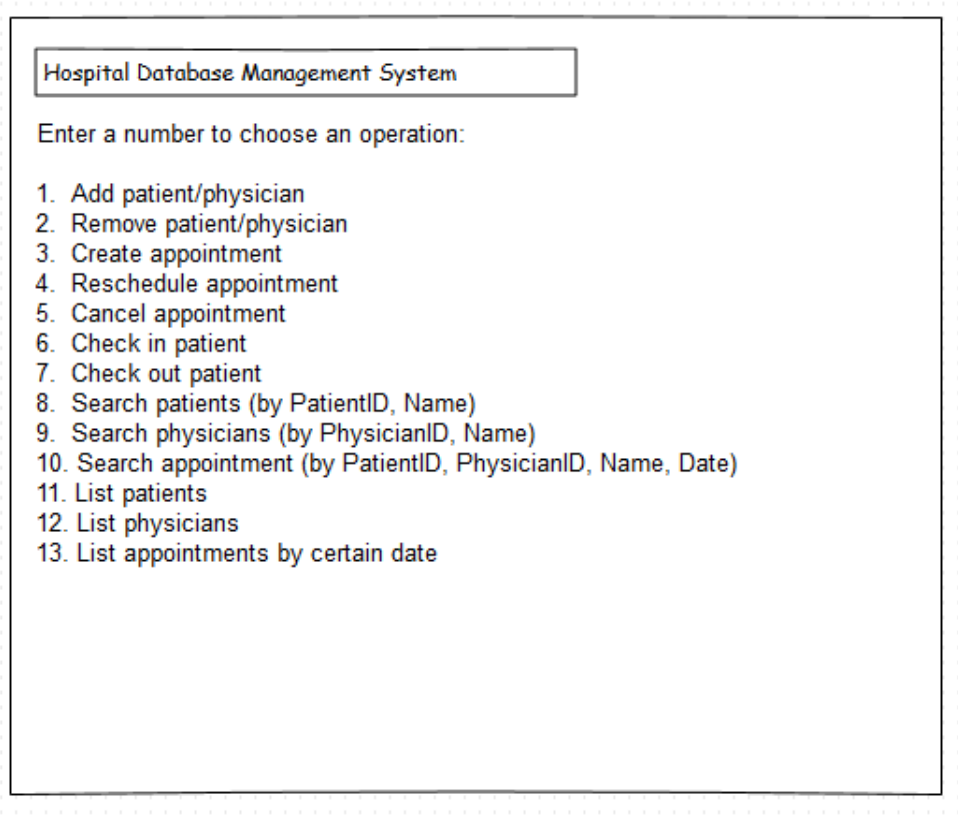
Joshua Kruger
Harrison Kurtz
Joshua Amaku

Relationship Schema



There are 3 entities: patients, appointments, and physicians. The primary keys of each entity are PatientID, AppointmentID, and PhysiciansID, respectively. There are 2 relationships: Patients schedule appointments and physicians attend appointments.

Rough sketch of UI



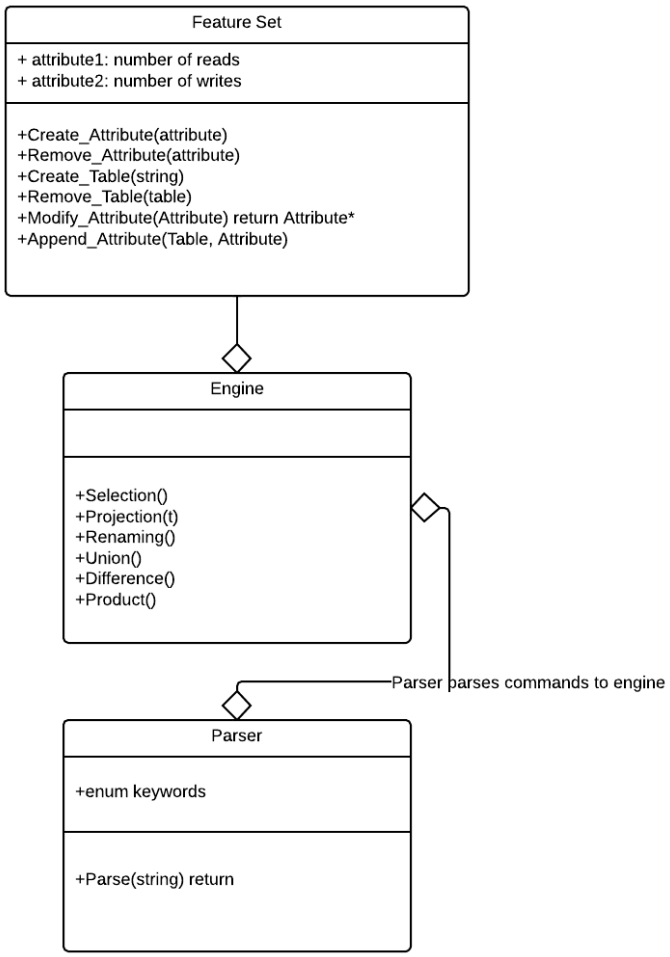
Joshua Kruger
Harrison Kurtz
Joshua Amaku

SECTION 3

Class Attribute
+ name: string - name of attribute + values: string* vector - vector of pointers to unique strings in attribute - domain: string - the domain of the attribute
+ insert_value(typename elem):void - add value to vector + Attibute(string, string) - constructor that set name and domain get_name():string - return name of attribute; set_name(string):void - sets name check_domain(string):bool- takes a potential string for the items vector and checks to see if it is in the domain of said attribute get_items():vector<string*> - returns all items add_item(string):string* - adds item pointer to vector and returns said pointer

Class Relations
+ table_name: string- name of relation + keys: bool vector - holds whether or not corresponding attribute is a key + att_list: attribute pointer vector - holds pointer to attributes + table: vector of vectors of string* - creates table by mapping values from different attributes to certain tuples
+ create_table():void - makes a table from all the vectors that are pointed to - update_table():void - updates the table after changes in the database show_table():int - prints table get_att_list():vector<Attribute*> - returns att_list get_att_list_size():int - returns size of att list get_att(string):Attribute* - returns the pointer of a certain string get_keys():vector<bool> - returns keys get_keys_names():vector<string> - instead of bool vector, returns actual keys names num_of_keys():int - returns how many keys there are get_num_rows():int - returns how many tuples there are insert_tuple(vector<string>):int - inserts a tuple to the table get_tuple(string, Attribute*):vector<string*> - returns a tuple of pointers that has a certain string in its corresponding attribute column in a vector get_tuple_string(int):vector<string> - returns a tuple with strings using the tuple line number get_tuple_string(int):vector<string*> - same thing as above with string*

Joshua Kruger
Harrison Kurtz
Joshua Amaku



SECTION 4

Benefits of approach

- Data stored centrally in list of attributes.
- low memory usage due to tables and relations pointing to objects (no doubles)
- Data concurrent and updated across all tables.
- Quick lookup of data because of soft tables stored in relations class.
- No possibility for inconsistencies.

Risks of approach

- Mass list of pointers becomes unmanageable when building database
- User deletes an attribute which causes conflict with another table that requires it
- Structures with relations of relations could be hard to represent in soft tables
- Passing back data from Function set back to engine could present opportunity for creation of duplicate tables

Joshua Kruger
Harrison Kurtz
Joshua Amaku

Post Development update

Issues:

- Initial idea that storing items inside attributes proved to be difficult to manipulate down the road
- Checking for duplicates proved to be more difficult than originally anticipated

Solutions:

- Workarounds for items storage were built into relations functions to avoid high level errors
- Functions doing small tasks had to be implemented to ensure no structure was lost during actions to the database. Double what was thought originally.

Development Log: (OPEN document https://docs.google.com/document/d/10KESaH81kqyyiXMRSypTSdsgMAi2MBHqDv4bJxq5am0/edit?usp=sharing