**Section 1 – State the purpose of your project/sub-system:**

This product intends to stand alone as a functional low-level database, which is made portable by the definition of a DML and parser. On top of this database, we will construct an application which makes use of the functionalities made available by our work on the database itself. We will then look to further test our database with an application made by a third party, and note any difficulties that arise in that transition, and think about how they could be prevented in the future in hopes of creating a more generalized product.

Many applications rely on a strong database, with high level functionality. This product attempts to stand alone with the ability to take in and store data, and manipulate it when queries to that data are made. From a warehouse to a university administration office, a database such as this could accept and manipulate data with ease, and allow for quick accounting of resources, and provide either a quantitative or qualitative analysis of the data.

The database we are looking to design, as stated, is fairly low level and far from optimized. Rather, it would provide a strong stepping stone towards looking at optimization of databases. Thus, in this case, we use the six core operations of relational algebra (*Selection, Projection, Renaming, Set Union, Set Difference, and Cross Product*), as well as the natural join operation. With this core, the database will have a strong functionality and will be able to create high level applications.

In this document we describe the flow of operation from the user’s perspective. We will have an application which provides direct user interface which uses the specified DML. This application will appear to stand alone, but relies completely on the underlying database which makes up the majority of the problem. The application accesses the database cell, through the parser, which in turn can look at the relations. In reality, the process of creating this project will flow in the opposite direction, from low level to high level, but to present this from a functional perspective we choose this ordering.

**Section 2 – Define the high level entities in your design:**

**Application:**

In the progression of the final product, the functionality begins with an application. The application effectively takes user input and translates that into something that the database language can understand. It is the intermediate step between the user and the database and provides a layer of abstraction which allows only desired access to the database information itself.

**Parser:**

The next layer is the parser. The parser acts as part of the database which converts human-readable language (i.e. INSERT…, JOIN…, etc.) and turns it into the commands and functions which the database takes. It provides the arguments to these functions as well. It is necessary to define this step as it undoes the abstraction created in the application, which is necessary for *portability* of the final product; that is, the parser is the layer that allows any program written in the defined language to be run on our database. Again, note in the Figure 1 how many applications are able to run on the parser. Of course, in this project we only look to create one main application, but we retain the option of general use.

Figure 1: Diagram of final product flow. Applications are portable, and are translated by a parser where information is stored in relations in the database.

**Database:**

The next layer is the database, which is essentially a collection of relations. The database needs to maintain fluid control over all of its relations, and be able to add, remove, and access relations with ease. After the parser turns the language into functions and methods, the database needs to decide which relations it is looking for, determine if those relations are valid, and act upon the command it is passed.

**Relation:**

The final entity is the relation itself. A relation is a table, which contains columns called attributes (i.e. *Name, Age, etc.*) and rows which contain values that make up entries (i.e. *[John Smith, 19], [Bob Joe, 18]*). The relation needs to have quick access to all of its entries and be able to return information about the entries it contains. The relation object itself has little functionality besides storing and returning information.

**Section 3 – For each entity, define the low level design:** This section is where your objects and object relationships are defined. For each object (or set of objects) define the following:

**Application:**

*Usage:* The application is what gives the database system life and purpose. It could vary in complexity, but the way it interfaces with the rest of the program is fairly rigid. It takes user input, and turns that into a call of the database through the DML and parser. The application is deservingly its own entity because it stands alone on the foundation that the rest of the database provides. It simply uses the database as part of its functionality.

*Configuration:---*

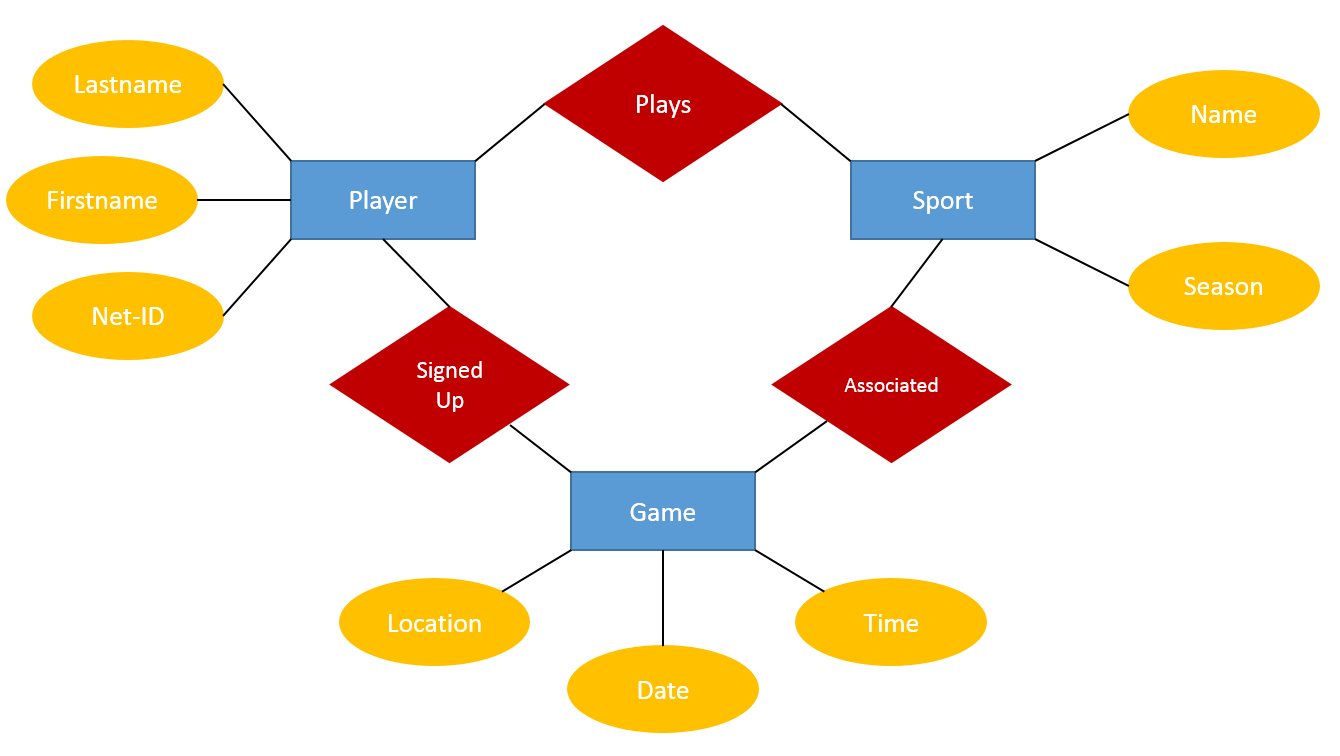
*Model:* The application will mimic one used by an organization in order to keep track of players and the intermural sports that they play. The program will use the database to keep track of player information as well as that of different sports and their corresponding games. The entities in the application will be players, sports, and games. A player will have three fields: Last name, First name, and Net-ID. Each sport will have two fields: Name and Season. A game will have three fields: Location, Date, and Time. (NEW) The application will be able to query the database to find a player's sports, the games for which a player is signed up, and the games associated with each sport. **(Cameron) So in the following diagram, the application has created a relation for players, sports, and games. Using relational algebra, it will query the database using one of the specified commands to return lists such as what games a player is signed up for, which sport someone plays, or when all of the soccer games are.**

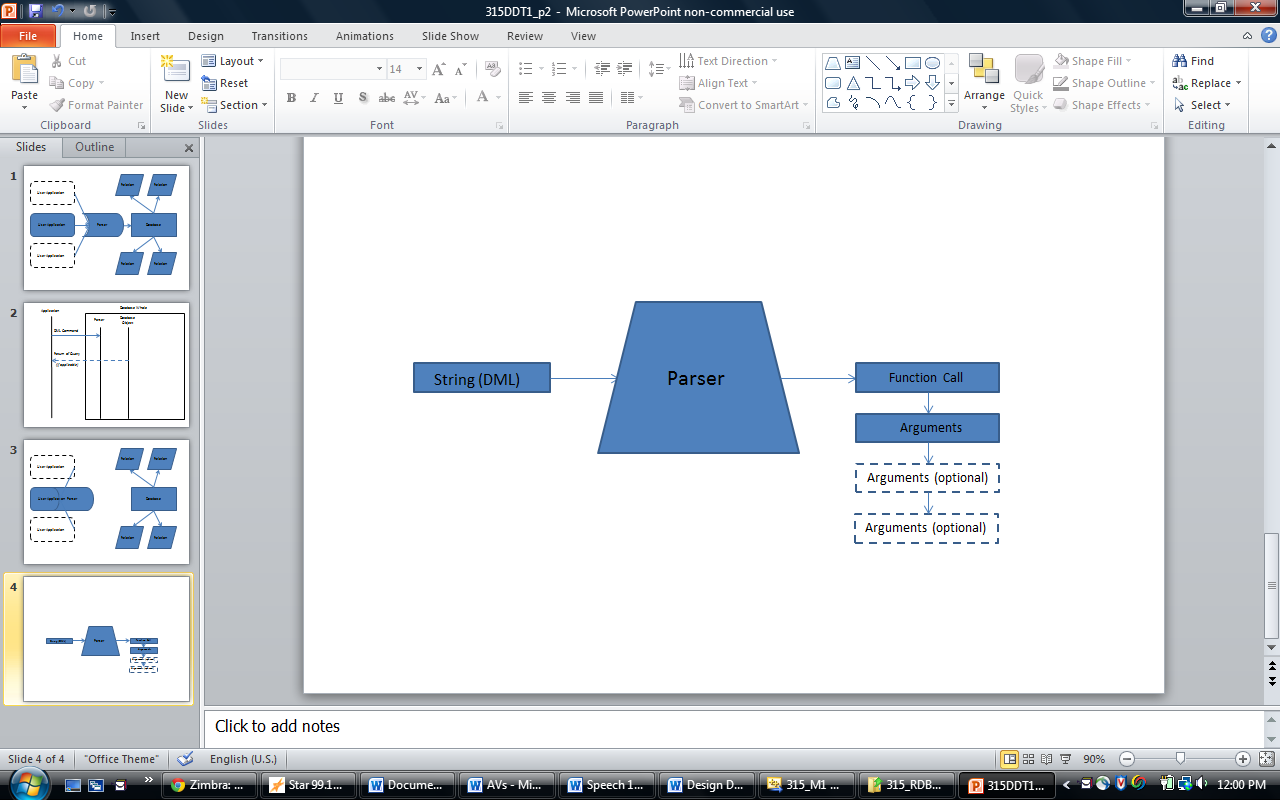
Figure 2: Entities and Relations

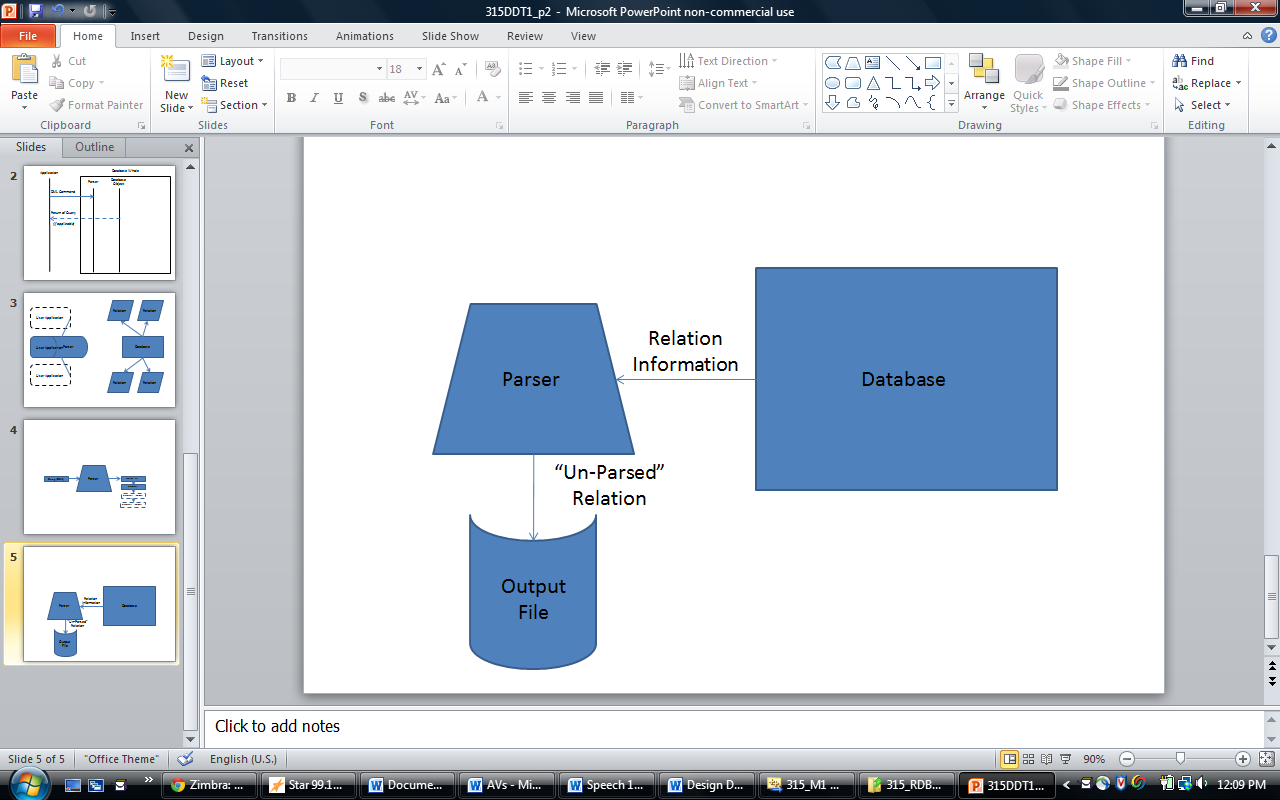
The application will have the following operations: Add player, Add sport, Add game, Remove player, Remove sport, Remove game, List players by (Last name, First name), List sports by Name, List games by (Date, Time), Change game time, Change game location, Display which sports a player plays, Display games for which a player is signed up, Display which players play a specific sport, and Display the games related to a sport.

*Interaction:* The interaction between the application and the database is two-way, yet have different endpoints. In communicating to the database, the application must be routed through the parser, which talks to the actual database object, which returns the value (if any) to the application.

Figure 3: Application Integration

**Parser:**

*Usage:* In the final design, the application uses the DML to act as an interface between the user and the database, however we need something to convert these DML commands into functions for the database to call. This is the parsers job. Notice, the parser is also used whenever the user asks to save or open a file. Whenever a file is opened, the parser reads the commands from the file to reconstruct the relation. It also makes sense to put the function which converts a relation into a series of commands in the parser’s functionality, because that is essentially what the parser already does, except backwards.

*Interaction:* The parser takes in a string command, and from there has to call a function with the proper arguments. This means that the design of the parser and the style of functions in the database need to be strongly related. That said, the only functions the parser has are:

void parse(string dmlCommand); - converts the dml command into a function call for the database, or determines if it is not a proper function call.

void unParse(string fName, vector<String> attributeRow, vector<vector<Entry>> entries, vector<int> keyIndeces); -passes the information of a relation to the parser, which directly outputs it into a file.

The parse function will first determine the sort of function it needs to call, and from there will move on to all of the different arguments it could pass. Because there could be any amount of arguments, all arguments will be in a vector.

**Database:**

*Usage:*

*Configuration:*

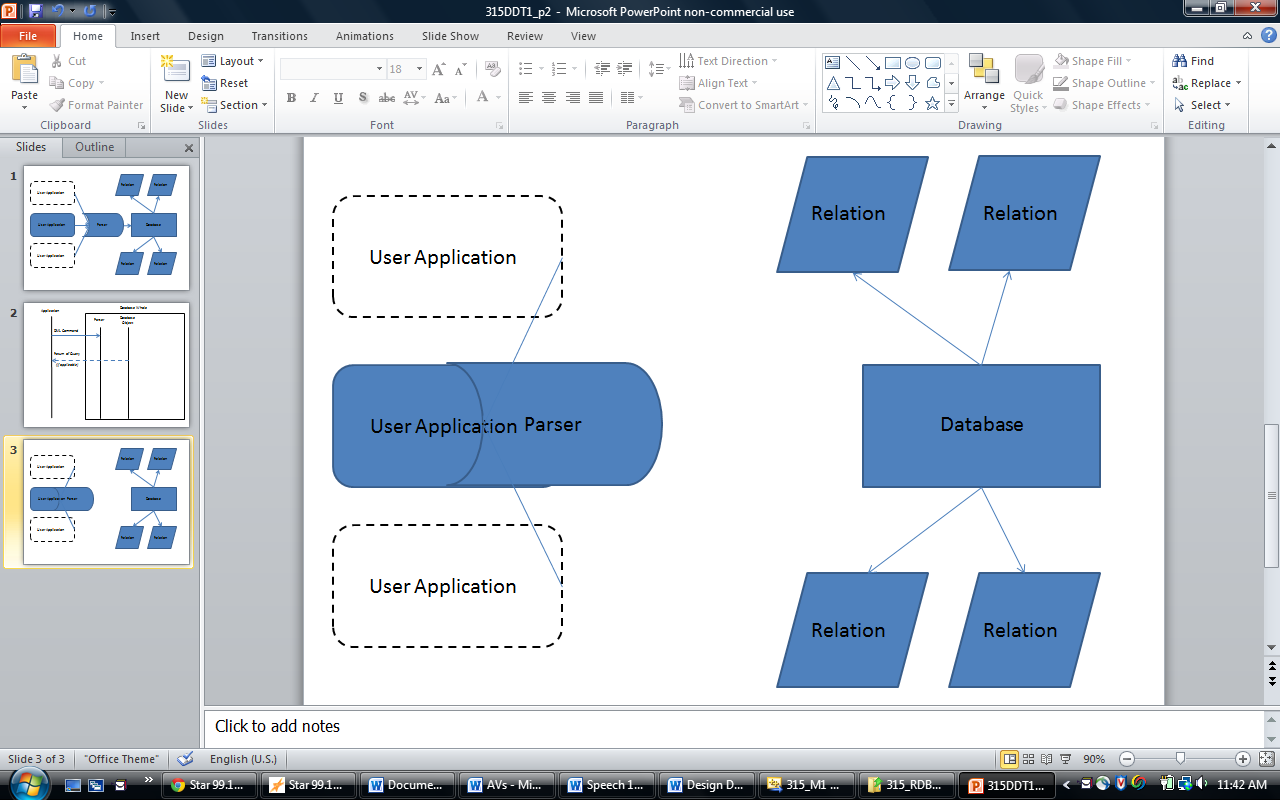
*Model:*

*Interaction:*

**Relation:**

*Usage:* The database that we will construct is relation based. All of the information is stored in objects called relations, which are essentially tables. The purpose is just to have a place to store related information. For example, a relation could store a table of all teams in a league and the number of players, or the subscribers to a bank and their account balances.

*Configuration:* A relation stores a table of values (stored as a vector of vectors), called *entries*. The two types accepted by the database are integers and “varchars,” or variable length strings for our purposes. This is the heart of a relation, however each column in the table, called attributes, also needs a title, as does the relation itself. Finally, when a relation is created it needs to get one or more keys which identify a tuple (a list of entries across a row) as unique. This will be stored as a vector of integers, which contain the indices of the key values.

ie. Relation(vector<String> attributes, vector<int> indices);

*Interaction:* Our relation has a fairly low level design. All a relation does is store the information in an organized manner, and return it when and how the database asks. The planned functions are:

vector<String> getAttributes(); - returns all of the attributes in a vector of strings. The purpose of this is to find the numerical index behind a property. For example, if a user is looking to compare something by age, we want the ‘age’ attribute column, but more so, we want the index of that column in the table.

vector<vector<Entry>> getEntries(); - returns like table itself. There are some operations in the database which require a search through all of the entries in a relation.

vector<Entry> getRow(int index); - returns one row (specified by index)

void addRow(vector<Entry> row); - adds the row to the table

**Usage**

Describe in a paragraph how the object is used and what function it serves. If an object will interface with an external object or system, it is a good idea to show the interface for the object. Most importantly, you must again describe your thought process for defining the object as you did. List the benefits and risks. If an object provides an encapsulation, describe in a sentence why the encapsulation adds value. Use your descriptions to give meaning to the diagrams. They don’t have to be verbose, just enough to get the point across.

**Configuration**

If your object needs any special configuration or initialization, this is a good place to describe it. If not, this section can be left out.

**Model**

Figure 2 shows an example of a to supplement the System Security entity from figure 1. It is not perfect UML, but has some aspects of UML. Most importantly, it describes the design.



Figure 2 (click to see full size)

Don’t worry about perfection in your models, but be sure to describe exactly what is going on in the diagram. Here, two concrete security objects derive from a base security object, and a security factory will create one or the other for a client depending on the security model of the system.

**Interaction**

This is also a good section for interaction diagrams. An interaction diagram shows how a set of objects or entities communicate with each other to perform a complex task. Figure 3 shows an example of an to show how a user might log in. It uses objects from the various entities shown in figure 1.



Figure 3 (click to see full size)

Again, this diagram is not perfect UML, but it explains the communication sequence to accomplish a complex task. Interaction diagrams are most useful when you want to diagram how an object in your system will communicate with an object in another subsystem. This type of diagram will let the other developer verify that the interaction is correct.

**Section 4 – Benefits, assumptions, risks/issues:**

The major benefit from this design using relational algebra is the simplicity it allows. Although efficiency is sacrificed, the implementation becomes much easier.

We also choose to put all of the search functionality in the database itself, as opposed to putting some of the searches in the relation. The argument is that the relation is just a storage device, but this also increases the organization, with all of the computation contained in the database object itself.

The way we choose to store data itself in an Entry object. An entry is either a string or an int, with a field specifying one. We do this because of this limits of the vector storage, where all elements need to be of the same type. This implementation of storage is small, simple, and works for all of our purposes.

One aspect that might be slightly confusing is the difference between the Database object, and the Database proper. The whole database encapsulates the parser, the database object, and the relations. This allows a distinct separation between the application and database, however we centralize the information and processing of the database around a database object. The relation is simply the information, and the parser is the doorway in and out of the database proper.

***Risks:*** We use files as a storing mechanism here, taking advantage of the parser to simply store the commands that create a relation in each file. This assumes that the program will not be used extensively, as in such a case reloading the information would become very inefficient.

In addition, the implementation of the database strives for simplicity rather than efficiency. This compounds the problem just stated, in that there are potentially serious efficiency problems.

In this section, make a list of 5-6 top benefits of the design, a list of **ALL** known risks/issues and a list of ALL assumptions. Some of this may simply be rehashing what you wrote in a previous section of the document. What’s important is getting all of these items into one section so that the reader doesn’t have to read the whole document to understand what the benefits, risks and assumptions are.

**Benefits**

* An efficient means of storing data.
* Can be utilized by any application written with our database API (does not only work for our application).
* Our design uses relational algebra instead of complex, specified algorithms. So this benefits the development time.

**Risks**

* The way we are storing the relations in files is a risk.

**Assumptions**

* We assume that the program will not be overloaded or subject to a serious stress test.
* We assume that blank spaces will not always be inserted between tokens for the parser.