**Design Document for Stand-Alone Database with Application**

**Databases: Design, Application, and Purpose**

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 play that role 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, thus providing 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. 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 to define the operations available. 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 and uses the specified DML. This application will appear to act independently, but instead 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/building 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.

**Overview of User-Level Flow**

***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. Our application is an intramural league manager, with players, teams, and scheduled games. We leave it open to more additions if necessary to prove all of the functionalities of the database.

***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 will undo 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 with a search engine. 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. Inside the database are the stored relations, and all of the functions required for the relational algebra we use. The database is the engine behind the program. It does all of the searching and computing once the parser gives it a function to perform.

***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, called tuples, which contain values that make up entries (i.e. *[John Smith, 19], [Bob Joe, 18]*). In our database, an entry is restricted to an integer, or a variable length string which we call a *varchar*. 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. Tuples within a relation are also unique within a set of attributes called ‘keys’. A set of one or more keys is a subset of attributes that defines a tuple.

**Low Level Design of Product Stages – *Usage, Model, Configuration, Interaction***

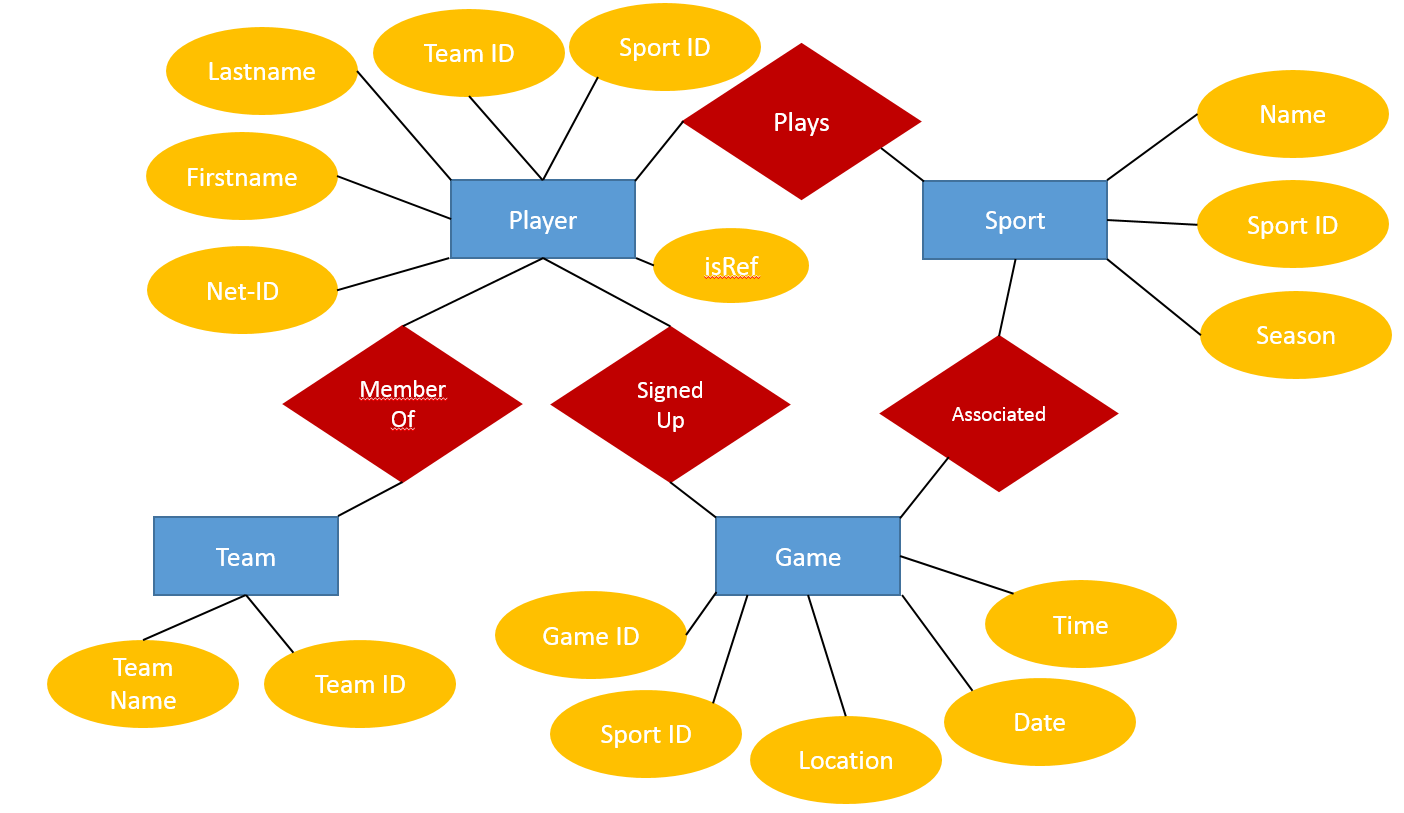
***Application*:**

*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.

*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. It will also be able to maintain records of teams, winning teams, and referees. The entities in the application will be players, teams (& winning teams), referees, sports, and games. A player will have six fields: Last name, First name, Net-ID, Team ID, Sport ID, and isRef. The isRef entry will denote whether a player is also able to perform referee duties in case one of the full-time referees is not available. Each sport will have three fields: Name, Sport ID and Season. A game will have five fields: Location, Date, Time, Game ID, and Sport ID. There are four fields for referees: First name, Last name, Net-ID, and Sport ID. Each team (or winning team) will have two fields: Team name and Team ID. One design feature with teams is that a team is not bound to a single sport. It can have members that play different sports, thus representing the team across the entire league. Also, a winning team has the same entities as a team entry. The significance of the winning team entity is that teams that have a winning season may be stored here also. This makes it easy to determine which teams have winning seasons and which have losing seasons.

The application will be able to query the database to find a player's sports, the games when a team could play, and the games associated with each sport. It will also be able to list all of the referees (both full-time and players who are referees), list players who are not referees, display the losing teams, and show the players that are on a team.So in the following diagram, the application has created a relation for players, teams, sports, and games. Winning teams can be viewed synonymously with regular teams on the diagram. Referees was left off of the diagram in order to save space, but they are associated with a sport in which they “ref.” Using relational algebra, it will query the database using one of the specified commands to return lists such as what games a team could play, which sport someone plays, or when all of the games are for one sport, etc.

Figure 2: Entities and Relations (ER Diagram)



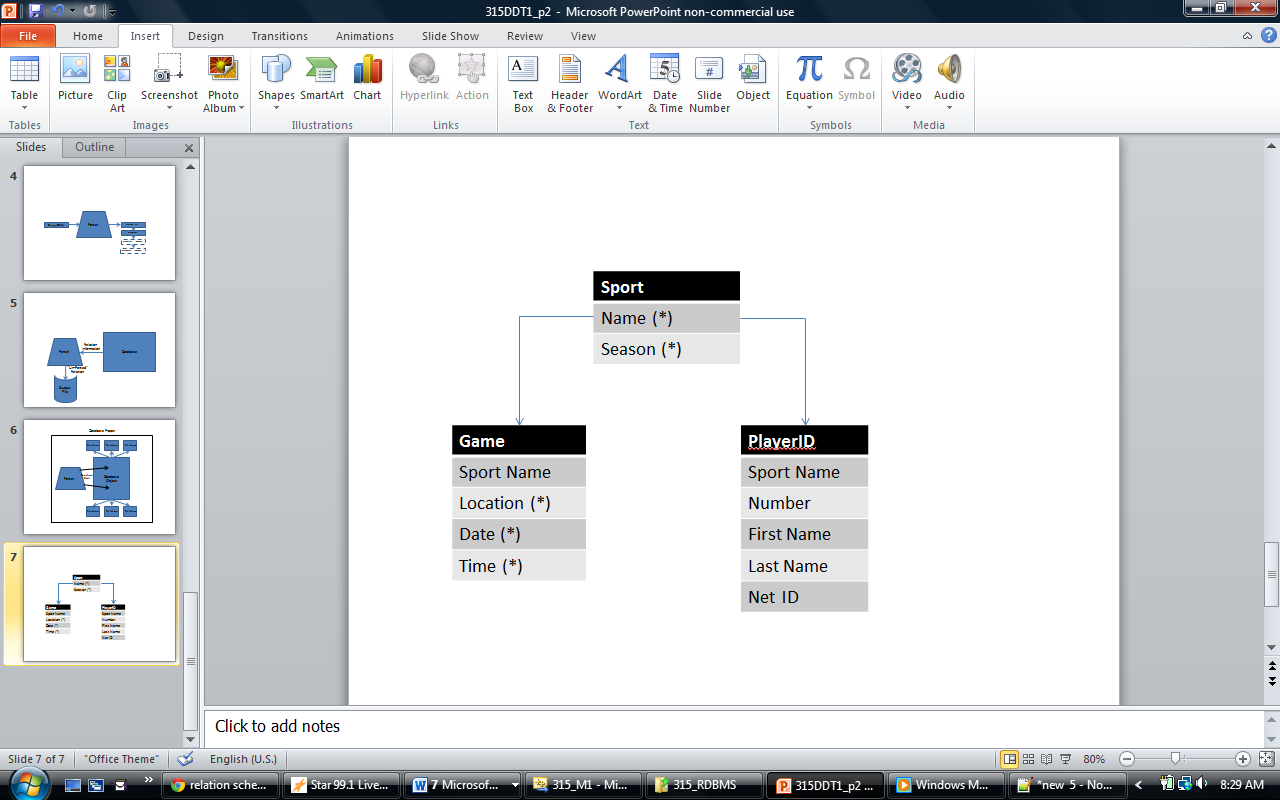


Figure 3: Simplified Relation Schema

*Interaction:* The application will have the following basic operations (functions): Add and remove for each of the six entities, List players, List sports, List games, List Referees, List Teams, and List winning teams. It will also be able to perform: Change game time, Change game location, Change a sport’s season, Display which sports a player plays, Display games associated with a specific sport, Display which players play a specific sport, Get all referees (player and full-time), Get non-referee players, Get losing teams, Show the names of all the sports, List the players on a team, and list players including their team names.



Figure 4: Application Integration

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. The means of communication between the database and the application will be a special relation called stored in the database. Any results will be there, and this relation alone will be the bridge back to the application.

***Parser*:**

Figure 5: Parser-Database Relational Close-Up

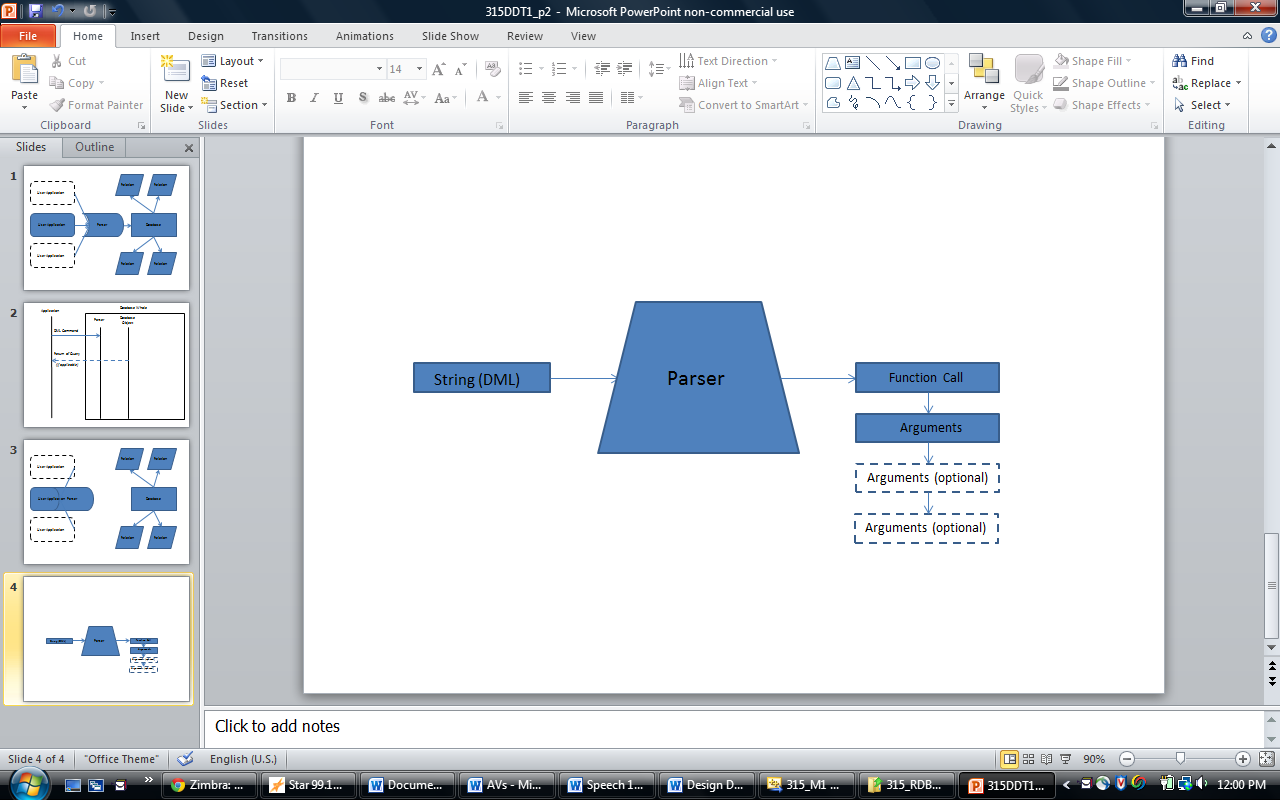
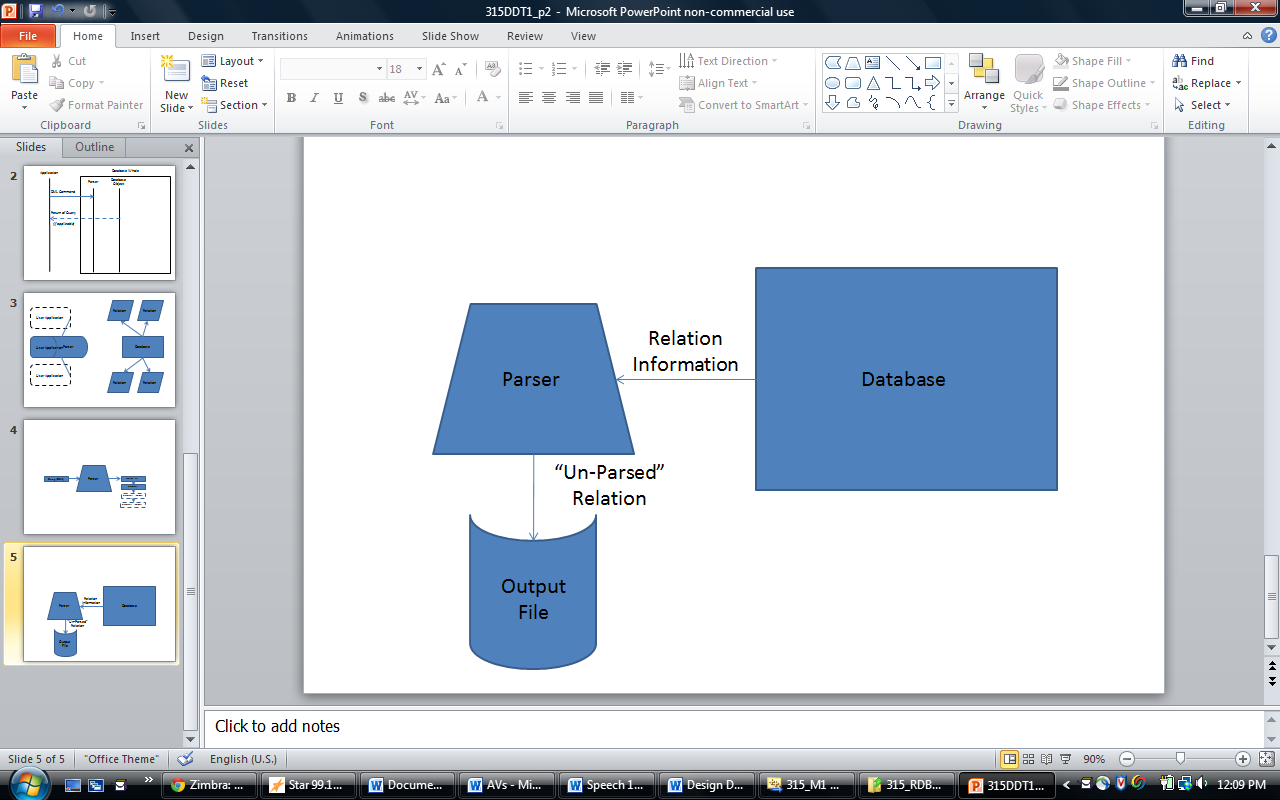
*Usage:* In the final design, the application uses the DML to act as an interface between the user and the database, however we need something to convert these DML commands into functions for the database to call. This is the parsers job. Notice, the parser is also used whenever the user asks to save 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.

Figure 6: “Un-Parse” Functionality

*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.

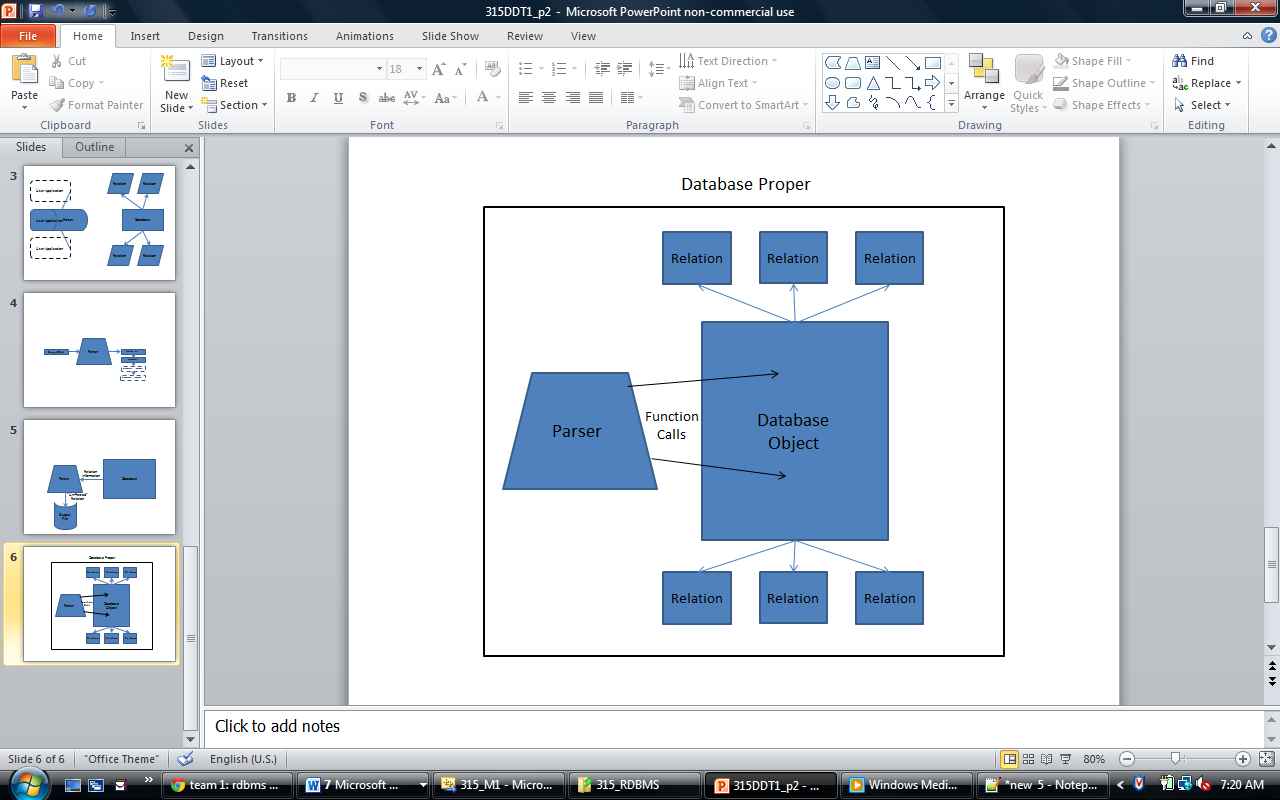
***Database*:**

Figure 7: Relation between Database Object and Database Proper

*Usage:* The database object is the heart of the database cell. It provides all of the search functionality and computation required for the relational algebra that we implement. While the parser provides the doorway in, and the relation is the storage device, the database itself is the foundation of the project.

*Model:* The database is called by the parser through function calls. Although the parser is part of the database cell, it performs the tremendous job of deciphering the incoming commands, and finding the proper function to call and with which arguments.

*Interaction:* The main interaction is between the parser and the database. The majority of the following functions derive directly from the commands that the parser has to be able to execute. On the other hand, the database also talks to its relations to retrieve information. The database has a high number of search focused methods, and subsequent functions to manipulate the information it finds:

void addRelationToDatabase(string name, vector<string> attributes, vector<int> indices); - add a relation/table of information to the database. This makes a simple call to the relation constructor and adds another entry to the databases personal vector of relations.

void addAttributeToRelation( string attributeName, string relatinoName ); - adds an entry as a new column to a relation.

void addTupleToAttribute( vector<Entry> e, Relation& r ); - Adds the entries that form a tuple to the relation

void removeRelation( string relationName ); - removes the specified relation from the database.

void removeAttributeFromRelation( string attributeName, string relationName ); - removes the specified attribute (entry) from an existing relation.

void removeTupleFromAttribute( int index, Relation& r ); - Removes the tuple at ‘index’ from the relation

int findRelationInDatabase( Relation r ); - returns the index of the relation in the database.

Relation& accessRelation( string relationName ); - A helper function which returns a relation reference based on a name

vector<Entry> selection( Condition c, string targetAttribute, string relation ); - prints the tuples that satisfy a condition

vector<Entry> projection( string attributeName, string relationName ); - gets a subset of attributes in a relation.

void renameAttribute( string relationName, string oldName, string newName ); - renames the specified attribute in the specified relation

void unionTwoRelations( Relation& relationA, Relation& relationB ); - union two relations together.

void differenceTwoRelation( Relation& relationA, Relation& relationB ); - compute the difference of two relations.

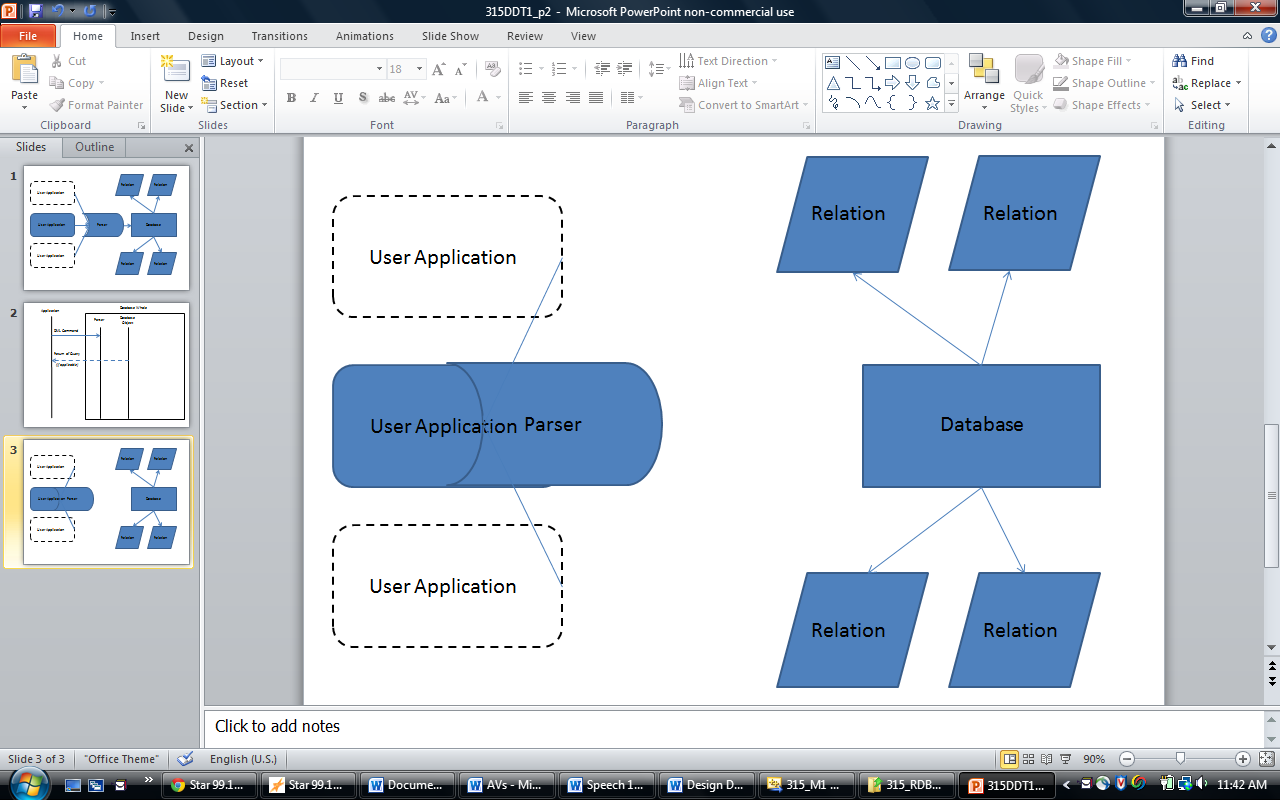
void crossProduct( Relation& relationA, Relation& relationB ); - find the cross product of two relations.

***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.

Figure 8: Up Close Interaction of Database and Relations

i.e. 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

**Benefits, assumptions, risks/issues:**

***Benefits***

* The major benefit from this design using relational algebra is the simplicity it allows. Although efficiency is sacrificed, the implementation becomes much easier.
* Can be utilized by any application written with our database API (does not only work for our application).
* The way we 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 the limits of the vector storage, where all elements need to be of the same type. This implementation of storage is small and simple, and works for all of our purposes.
* We 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.
* 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***

* The way we are storing the relations in files is a risk, as they could be modified, and heavy stress on the system will drastically effect run times.
* The way we store types of entries can be ambiguous at times. One potential solution to keep in mind is to add a type to an attribute. So for example, an attribute would no longer be just a string name, but rather a struct with a string name and an enum type.

***Assumptions***

* We assume that the program will not be overloaded or subject to a serious stress test, which as mentioned could produce unexpected results.
* We assume that blank spaces will not always be inserted between tokens for the parser, ie. no specific style with regards to white space is specified in the DML.