|  |
| --- |
| **Designing and Querying a Relational Database** |
| Querying XML data using SQL style mechanics. |
|  |
| This document contains the final project report for emulating a relational database in querying xml for Advanced Database, Comp 511, PennState Harrisburg. It outlines introduction to the task at hand, relevance toward advance database concepts, features of the program, implementation details, and lastly any references needed to ensure a quality deliverable. |
|  |
| **Allen Brubaker** |
| **11/30/2009** |
|  |

Contents

[Overview 3](#_Toc247385592)

[Detail of the Problem 4](#_Toc247385593)

[Features of the Application 5](#_Toc247385594)

[User Interface 5](#_Toc247385595)

[Querying Mechanics 6](#_Toc247385596)

[General Query Guidelines 6](#_Toc247385597)

[Creating a Table 7](#_Toc247385598)

[Inserting Values 7](#_Toc247385599)

[Querying Tables 7](#_Toc247385600)

[Deleting Rows or Tables 8](#_Toc247385601)

[Importing XML Data 8](#_Toc247385602)

[Implementation Details 9](#_Toc247385603)

[Application Structure 9](#_Toc247385604)

[XQuery.GUI 9](#_Toc247385605)

[XQuery.Classes 9](#_Toc247385606)

[Parsing Queries 10](#_Toc247385607)

[Database 12](#_Toc247385608)

[Memory and File Structure 12](#_Toc247385609)

[Initial Loading of DataSet 12](#_Toc247385610)

[Creating Tables and Inserting Values 13](#_Toc247385611)

[Joining 13](#_Toc247385612)

[Filtering 14](#_Toc247385613)

[Sorting 15](#_Toc247385614)

[Aggregating 15](#_Toc247385615)

[Selecting and Set 16](#_Toc247385616)

[Deleting and Dropping 16](#_Toc247385617)

[Importing arbitrary XML Data 17](#_Toc247385618)

[Test Cases 18](#_Toc247385619)

[Conclusion 19](#_Toc247385620)

[Resources 21](#_Toc247385621)

# Overview

There exists a disconnect between our current knowledge employing databases to not only simply store data but to access data in a manner that is relevant to the end users of the data, and between our true knowledge of the underpinnings of such databases and how, when one enters a properly written query in a predetermined language, data is both queried and returned. Being in an advanced database class, this gives us the perfect opportunity to tackle this inherent disconnect between knowledge of how to use a database software such as SQL Server and then the more subtle yet profound knowledge of what happens underneath.

In order to properly study such concepts, instead of directly studying SQL Server, it was our desire to recreate a relational database such that we would have hands on experience with handling all the underlying data storage of a database, which in this case would be XML. Secondly, we would use this relational database to explore the possibility of converting the problem of arbitrary XML querying to the problem of querying a relational database. In this way, we would more aptly and powerfully query xml than was previously possible by harvesting the xml data and properly fitting them into tables with columns that can then be queried relationally. Finally, it was our goal to create a windows program with an accessible yet friendly user interface to act as the presenter and handler of the relational database. Creating a useful design and concept to not only manage data that should be allowed to be boundless in size, as well as creating a language to efficiently gather and present the data to end users, with a program for end users to test, introduces some issues. These issues consist of how to store data in the relational database intuitively in order to then query the data, the syntax of the language that should be utilized, the means to harvest and convert arbitrary xml data into a relational database, and lastly the programming language that should be used to implement the project.

To face these issues we had to decide upon a useful programming environment to create, manage, and test the entire project, and lastly the style of the querying language. We decided it best to emulate the SQL query language style as closely as possible to give us both concrete goals for the different abilities we wanted our querying language to have (these might include the ability to sort, join, select, filter). Moreover, we chose to emulate a SQL querying style that users may need only minimal time in order to get acquainted with our querying language. Therefore, when they are presented with the final project, they can begin testing the various abilities of our query language rather than spend time having to learn the language. It is of fundamental importance for readers to recognize that it was our goal from the beginning to create a relational database foremost, then to convert the problem of querying an arbitrary xml file into the problem of querying a relational database. This would allow us to fully utilize the power of relational databases in tackling the issues of xml querying elegantly and comprehensively. Details on how this was done can be found in the implementation section of this document.

As for the programming environment chosen, we decided to use a programming environment that we were most familiar with, this being the venerable .Net and more specifically C# with its many invaluable libraries, of which we will be making ample use not only to create the GUI with speed, but to ease the many parsing issues that will present itself when trying to query XML. To some, parsing may be thought of as the major component of our project, but to us we feel that allowing .Net to handle the parsing will allow us to focus on the querying mechanics for which this project was duly designed. This will give us more time to tackle and emulate more of the querying concepts and less of the drudgery of xml parsing. What follows is the breadth of our final project report including a detail of the problem to be solved, features of the application, and implementation details.

# Detail of the Problem

Briefly introduced in the overview section of this proposal, the problem we are tackling with this project of creating an XML Query Language is the large disconnect between our understanding of how a database, such as SQL Server, takes arguments in the form of a language style understood by English speakers (such as SELECT, FROM, WHERE), then efficiently queries an underlying storage unit to return data in a format desirable to the user. The presented data could be sorted, grouped, joined across multiple data units or tables, filtered through various constraints, and although the underlying database is stored in static many times useless forms, when it is queried and joined across multiple tables with filters imposed on the query, the data that is returned is deeply meaningful and thoroughly enlightening for users as they ascertain in far greater ways perhaps subtle trends that their products have followed amongst their customers or even as they maintain employee payroll information with the added nicety of being able to track with accuracy and confidence the history of pay raises.

It is obvious that being able to store data in a database and query the data in a form meaningful to the users creates the entire beauty of the concept of a database itself. Though one can study the many esoteric database concepts of data mining, concurrent transactions, query optimizations and the like; understanding the rudimentary concepts of a database, which consists entirely of storing data and presenting data in as flexible a way as possible, is of utmost precedence. This project desires to study just that, and we intend to accomplish the goals of understanding how a database works by creating and managing our own data storage facility as XML files, designing a language easily understandable to end users, and creating the functions of the language necessary to perform adequate querying across the entire breadth of the data, which will most assuredly consist of multiple tables. We plan to emulate the SQL querying language as closely as possible, while managing ourselves the underlying XML storage facility.

Past attempts at creating an XML Query Language can be found in a project developed by Eric Liskow entitled *A New XML Query Language* (cs.hbg.psu.edu/comp519). Though his project was mainly a success, the majority of his efforts, and indeed many sleepless nights, were spent designing a means to effectively parse the xml data in response to the queries issued by the query language. Because we chose to utilize .Net and C# with its many useful libraries that ease creating and managing XML data sources, less work will be done in handling the less meaningful task of parsing, and more time can be devoted toward creating a truly intuitive querying language employing as many language mechanics as possible in attempts to imitate SQL. These include being able to select, sort, filter data, group data, join data with multiple styles of joins, and the like. What follows is a look at the features of the application including a cursory explanation of the user interface.

# Features of the Application

Users will find this section most valuable when they attempt to delve into fully utilizing the application the way it was intended to be used such that syntax errors in writing queries will be minimized, while the full power of the program can be realized. First, we shall give a brief explanation of the facets inherent with the graphical user interface so users will be able to navigate the application with ease. Subsequently, we will take a look at the features of the querying language and the different functions a user can utilize to query. Lastly, we will explain the syntax of the SQL-like querying language needed to appropriately query the underlying relational database.

## User Interface

Upon double clicking the executable file that comes with the application, the user will be presented with an interface that will be the place where querying the underlying relational database will be accomplished. The user interface is quite simple and intuitive. The main textbox covering the breadth of the application is the location where any number of queries can be entered. One nicety of the textbox is that any recognized keywords will be turned to bold, so that the user can recognize the keyword is supported. Withal, the application is smart enough to parse the queries, partition each query group into a different querying process, and return a table of results as needed. Should the user desire to only query a subset of “query groups”, this can be done with ease. The user simply needs to highlight with the mouse the desired queries and select the *Query* button. If the user fails to highlight the queries, no worries should arise. The application will instead query everything.

The *Query* button has already been introduced, but we now turn our eyes on the other little buttons that appear towards the bottom of the application. There is a *Clear* button that the user can press that will clear any queries present in the textbox and position the cursor inside the textbox so that the user can begin to type more queries as he or she sees fit. The little buttons on the right from left to right are explained as follows: The first button, when pressed, will bring up another window that consists of the current schema or state of the database. The schema is stored as a tree view. The first level of the tree consists of the tables present in the database. If this level is expanded, the items underneath contain the columns that exist in the table. For the user’s sake, next to each table the number of rows present in the database can be found. It is noteworthy to mention that this schema window can be kept open and after any queries are run, the window will automatically update itself without the user’s explicit intervention. Should the *Schema* button be opened multiple times, the application will replace the current window, so that only one window may be opened at a time. In addition, if the user right clicks on any item in the tree, whether it is a table or a column, the name is automatically added to the query textbox to facilitate users in the query design process.

The second button that appears to the right of the *Schema* button is entitled the *Log* button. When pressed a log will appear that will contain sets of timestamps along with their corresponding log message. These log messages can contain messages ranging from “*table: Inserted x rows”*, or “*table: Deleted x rows”,* “*table: Created”,* or even “*table: Dropped*”. These messages are invaluable for delete, drop, create, or import queries, as opposed to select queries, where a result window is not returned and only the state of the underlying database is modified. Similar to the way the *Schema* window automatically updated its information when queries were run on the main window, the *Log* window will also automatically update itself even when open. This way up-to-date feedback on query results can be obtained.

The last window, though not so apparent at first, is entitled the *Results* window. When a user runs any number of select-type queries, the database will query the desired tables and return a *Result* window for each select query found in the set of queries run. Each *Result* window will contain a data grid with columns and rows, which present the results of the query where one might possibly find joined, sorted, or grouped data depending on the query entered.

## Querying Mechanics

What follows in this section is a look at the various features or mechanics of the querying language, what these features allow users to do, and finally the proper syntax required to query effectively. A proper knowledge of SQL querying should be more than needed to completely understand the querying language we present. The functions a user can chose to make use of when querying are as follows: creating tables and columns utilizing a wide range of types; inserting values (tuples or rows) into the tables; selecting columns from tables; selecting aggregates of data from tables (max, min, avg, count, and sum) on columns via grouping mechanisms; selecting from multiple tables by joining on common columns; renaming selected columns; filtering data in disjunctive normal form with *and’s* and *or’s* based on the expansive breadth of operators supported; sorting any number of columns; storing query results into a temporary view or table to query from in subsequent queries; renaming tables temporarily to facilitate the process of creating a query; completely deleting tables and all their rows; deleting only a subset of rows based on filtering criteria; and lastly, importing data from any xml data source into the relational database to then query from.

General Query GuidelinesAny number of spaces or returns can appear between words. All keywords and columns must be separated by at least one space. Commas do not have to be used when querying multiple columns. A semi-colon need not be used to denote the end of a query. Likewise, parenthesis need not be used. Of course if the user desires to include these elements, he/she can feel free to do this, since the application correctly parses these elements out. Any element the user wishes to create such as columns or table names with multiple words in them and strings that contain multiple spaces must be surrounded with brackets [ ]. The entire sum of keywords handled by the program is as follows: create, insert, values, select, from, where, orderby, delete, drop, set, import. These keywords are case-insensitive. Likewise, the entire database is case-insensitive, such that tables and columns and even querying string data present in a table will be selected without regard to exact casing constraints. Columns or tables to be selected can be renamed by appending a colon to the end of the column or table. Types consist but are not limited to short, int, long, float, double, decimal, datetime, timespan, string, char, and bool. Since datetime and timespan values might not be intuitive to users, these types and corresponding values are addressed in the section: Inserting Values. Operators that can be used in the **on** and **where** statements of the application are as follows: <, <=, >, >=, =, !=, and additionally for strings: startswith, endswith, contains, !startswith, !endswith, !contains.

### Creating a Table

|  |
| --- |
| ***Create*** *TableName Type1 Column1 Type2 Column2 ….* |

To create a table type in the **create** keyword and then the table name. Subsequently, type in one or more pairs, where each pair is in the format: ColumnType ColumnName. The list of supported types are listed above.

### Inserting Values

To insert values, type in the keyword **insert**, then the table name. To insert values based only on a subset of columns, type in columns that you desire to insert values for. Any other columns will contain the default value associated with them when inserted (empty string for columns of type string, one space for a char, and 0 for ints or floats). To insert values for all columns, you need not specify any column name. Now, type in any number of rows using the **values** keyword. To insert datetime values use the following syntax: mm/dd/yyyy, or optionally, [mm/dd/yyyy hour:minutes:seconds pm]. To insert timestamp values use the following syntax: days.hours:minutes:seconds.milliseconds.

|  |
| --- |
| ***Insert*** *TableName Column1 Column2 ...* ***Values*** *Value1 Value2 …* ***Values*** *Value1 Value2 … ….* |

### Querying Tables

To query tables you will need to make use of the **select**, **from**, **where**, **on**, **orderby** keywords. Many of these keywords are optional, but you must make sure to have at least one select keyword and one from keyword. You can also select \*, where \* selects all possible columns. Select must appear as the first keyword in the query. These function exactly like their SQL counterparts. The minute differences are as follows: To rename columns in the select statement, append the new name to the end of the column with a colon in between. To aggregate columns prepend one of the following to the column name: max!, min!, count!, sum!, avg!. When aggregating (using the previous aggregating functions) it will group rows based on the columns found in the orderby statement. To emulate the count(\*) functionality present in SQL, you can simply not use any orderby statements, and count a specific column: count!columnName. You cannot select columns that are not in the orderby statement without an aggregating function. Furthermore, be sure that if a column is ambiguous or exists in multiple tables that you are selecting, you specify the full column name: TableName.ColumnName to avoid ambiguity. Since the application is very sensitive to errors, an appropriate error message will prompt the user should invalid logical errors reside within a query. Aggregate columns can also be renamed by appending a colon and the alias onto the end of the column name.

|  |
| --- |
| ***Select*** *Table1.Column1 aggregate!Column2:alias ...* ***From*** *Table1 Table2:alias …* ***On*** *Table1.Column1 <= alias.Column2 and …* ***Where*** *Table1.Column1 > alias.Column2* ***OrderBy*** *Col1 Col2 …* ***Set*** *NewTempTable* |

When using the **from** keyword the following should be adhered to. One limitation is that you cannot join tables to themselves, even if you rename the tables to distinct names. To circumvent this issue, one can use the set keyword as extrapolated upon below. If you want to join multiple tables you can, but you must make use of the **on** keyword. Here you can place the column (or fully qualified column name if ambiguities exist) with an operator as specified in the previous sections. For expressions in the on keyword, you can only separate expressions with “and”. “Or’s” are not supported in the onstatement. Thus the join is similar to a theta join where you can join based on more than just equality between columns. You cannot perform cross products between tables and must specify an expression to link/join all tables together that are present in the from keyword.

Optionally, you can make use of the **where** keyword. This is similar to the on keyword except for the fact that you can separate expressions with any number of or’s and and’s. You cannot use parenthesis to force or’s to have a higher precedence than and’s. Thus the only form that is supported is disjunctive normal form. Still, any expression can be reduced to one that is in disjunctive normal form.

Moreover, the **orderby** keyword can optionally be used to simultaneously group and sort data on any number of columns. This will allow you to sort first by one column, then by another, then by another, where each group that is obtained from the previous sort is what is sorted on in the next sort. You can also optionally include the keyword asc, ascending, desc, or descending after each column to sort by a specific ordering. If no directional keyword is placed directly after a column, the default is assumed to be ascending.

Finally, the **set** keyword can optionally be used to store the resulting table (obtained from the query) temporarily in the database in memory until the program is closed and the table is eliminated. This will allow users to explore for more complex querying because subsequent queries can use this temporary table to join on other tables. Users cannot themselves delete, insert, or drop these temporary tables once they have been created. They can only be used in select queries.

### Deleting Rows or Tables

|  |
| --- |
| ***Delete*** *Table* ***Where*** *expression(s)* ***Drop*** *Table* |

To delete rows from a table the keyword **delete** must be used. If no **where** statement is specified immediately after the delete keyword, all rows in the table are deleted, but the table structure is kept intact. If filter criteria are specified, only rows that meet the criteria will be permanently deleted from the table. To drop tables, or permanently remove them from the database, the use should use the **drop** keyword along with exactly one table. Similarly, the **delete** keyword must contain exactly one corresponding table.

### Importing XML Data

|  |
| --- |
| ***Import*** *FilePath.xml  Header:Table Desc1:Col1 Desc2:Col2 …* |

One can specify an xml data source to obtain data. The application will then harvest the data and create a table-like structure along with inserting rows to store the data. This table can then be used to fully query upon relationally. The user should specify the xml data path where the xml file resides, the header tag that will denote a new row along with an optional “:alias” to rename the table in the database to something other than “header”. Furthermore, the user should specify at least one or more descendent tags from which to obtain the data. These will be the columns in the table. These can also be optionally renamed to a more suitable moniker. Since it would be too cumbersome to search every single value in order to determine the best type for each column, and further so that the user need not take on this responsibility themselves, each column is created of type string.

# Implementation Details

While we have just taken a look at the application from the bird’s eye view and hopefully now have a solid understanding of how to fully utilize the application—though of course testing the application out for oneself would prove invaluable for solidifying the sundry features—it is now imperative that we take a look into the finer workings of the application which include the structure of the underlying code, how the different features were handled, the classes needed, an overview of the different algorithms implemented to handle the different queries, and of course a look at the challenges found along the way. We will first extrapolate on the overall structure of the application.

## Application Structure

The application consists of one .net solution, which in turn consists of two projects. The first project, or set of source files, consist of the user interfaces and windows, and is denoted XQuery.GUI. The second project, entitled XQuery.Classes, consists of the underlying business objects that the former project uses in order to perform the details of its operations.

### XQuery.GUI

Since this project consists of all the interface elements making up the application, we need not spend too much time on the details of this section because they mostly consist of making generous use of the supplied .Net Windows Presentation Foundation (WPF) controls and manipulating them as needed. The controls used were DataGrids to display the results of tables, buttons, textboxes, a rich textbox custom control, and tooltips. In the QueryWindow.xaml codebehind file where the programmatic elements of the control lye, one will find only real noteworthy set of instructions. Navigating there now would aid in the explanation of this section that is to come. Likewise, for this entire section of *Implementation Details*, it would be wise to navigate to the corresponding file as needed in Visual Studio. Here the various buttons are handled opening the other windows (SchemaWindow.xaml and ResultsWindow.xaml). Furthermore, the text that is input in the query textbox is taken from the control, sent into the Query object (to be explained later), parsed for each query group, whether it be select, delete, insert, create, drop, or import queries, and calls the DoQuery() function on each query. This DoQuery function returns the result in the form of a DataTable that is then sent to a new spawning of the *Results* *Window* and attached to the datagrid control that then displays the data. The DataTable is explained in subsequent sections.

### XQuery.Classes

This is where the main implementation of the program resides. The Query.cs class is the business object that is called above. The function of this class is to take a large string that may contain any number of different types of queries, dynamically partition and create one query object for each query that is sent in, error check to see if the query arguments are appropriate to the required syntax as specified above, and call its DoQuery() method, that will reroute the query to the appropriate function depending on the query type. Each function will in turn further break down the arguments and pass them to a corresponding function in the DataHandler object. The functions in the static DataHandler object will either modify or query the in-house database and return nothing or a DataTable with the results (if the query type is Select). Then the DoQuery() method will take that result and return it to the QueryWindow.xaml located in the XQuery.GUI project. It is noteworthy to recognize that the job of the Query.cs class is to parse and check for any syntactical errors, while the DataHandler’s job is to check primarily for any logical errors (such as trying to create tables that already exist). Furthermore, the Global.cs source file contains global enums (such as QueryType), global exception classes (InvalidQueryException). The Log.cs contains the functionality to perform logging within the DataHandler class to record when the database has been modified. The Expression.cs file contains the class that stores expressions, which contains both left and right operands and an operator. These expressions facilitate filtering in the where and on statements. Lastly, the SyntaxProvider.cs is the place that stores all the application-wide keywords, supported types, allowed operators, and tags. This aids the CustomRichTextbox control in searching for recognized keywords and bolding them. It also aids the rest of the application in ensuring valid syntax, such as the expression class when checking for valid operators, and the like.

## Parsing Queries

Now that we have a general understanding of the structure and flow of the application and which class handles a particular element of the functionality of the application, let us now turn our attention more fully to Query.cs to understand in greater detail how parsing was accomplished, some of the pitfalls and challenges that reared its head as programming commenced, and lastly how these pitfalls were dealt with swiftly and exactingly. For the remainder of this section, navigate to Query.cs.

Remember that this class is passed one large unparsed crude string representing the data as entered by the user in the query textbox of the interface. Thus, the Parse(…) static method of this class takes this string, breaks it into a list of strings where each string contains one unique word separated by whitespace in the original large string. It then analyzes each word for keywords that denote new query groups or blocks (either select, insert, delete, drop, or create), and creates a new instance of the Query object for each group, which gets recombined into a string from a list of strings before being passed into the parameter of the constructor. The unique challenge here was trying to figure out a way to take groups of queries and parse them. We found it best to instead of using regular expressions, use the String.Split() method provided by .net, to split the strings by whitespace tokens into a list of strings and analyze each word by word in building the groups. This part was mostly intuitive. It is the next part that presented some challenges, and we still question if this was the best way to approach the problem, but at least it worked effectively enough for our desires.

The PopulateQuery(string query) is called by the constructor. This function is where the breadth of the parsing takes place. Again, it takes the string that is passed, now acknowledging that it is exactly one query and not multiple queries, and breaks up the string for each word. The list of words is analyzed one at a time and when a keyword is found, subsequent arguments are added to the appropriate class-wide fields. The class wide fields are Create, Insert, Values, Select, From, On, etc. etc. For the most part these are simply lists of strings, except for Values, which is a list of list of strings. This makes sense because these can contain multiple sets of data if the user wants to insert multiple rows at a time into a table. The On and Where fields are different in that they are Lists or Lists of lists of expressions each consisting of a left operand, operator, and right operand. After the arguments are placed into their respective containers, we now approach a major parsing issue. How would we adequately parse the expressions in the On and Where lists, while maintaining precedence between or’s and and’s separating each expression? We took the approach of recombining the list of strings in the Where and Or statements into one large string for each, then for the case of the Where field, first splitting the string on the token “or”. This yields a list of strings. Now, each string in this list is split again for any token “and”. This process in its entirety yields a list of list of strings. Each string is sent into the constructor of the expression class, which now splits the string up by spaces and recombines strings that are contained with [ ]. It then populates the Expression.Left, Expression.Right, and Expression.Operand fields present in the class. Thus the On and Where fields are of the data type: list of expression and list of list of expression, respectively. Then, each field or list of strings in most cases are sent through the ReformString() function in order to coalesce any strings that are surrounded with brackets. After this, the SetAliases() method is called that takes any renamed tables in the query, searches anywhere where this alias is used, and replaces the original table name with the alias. Lastly various error checking is performed and the QueryType is set. Though for a while we greatly desired to allow the user to be able to use parenthesis to provide a more fuller set of expressions within the Where and On statements rather than just forcing the user to use disjunctive normal form, the task of creating parse trees was indeed outside the scope of this project. Instead, using disjunctive normal form, though restrictive to the user, provides the same expressive power because any set of expressions can be cast into this form. (Disjunctive normal form being expressions separated by or’s and and’s where there is no way to enforce the or’s of having a higher precedence than the and’s).

The DoQuery() method is called next on each Query. You will notice we take the global enumeration, QueryType, check to see what type the current query is, and call the appropriate function. Within each function, each field is broken up further depending on the parameters accepted by the corresponding function in the DataHandler object. Since the other functions are fairly intuitive, let’s take a look at the DoSelect() function. Before we any operations are performed in this method, we first find it necessary to call the function SetFullyQuailfiedNames(). What this function does it to analyze the all tables and columns to make sure the tables and columns exist within the database, ensure that no ambiguities exists where a column exists in more than one table, and replace any columns with fully qualified names (Table.Column). It does not modify any arguments within the Select field simply because more intricate processing needs to be accomplished later (renaming the resulting table column names, handling aggregate functions, and the like). Next, you can see some preliminary parsing of the OrderBy arguments is performed to make sure we have a list of only columns and then a second list of the direction to sort the corresponding column. Now notice that we create a temporary DataTable (to be explained in the next table), pass the table through a variety of functions to join, filter, sort, group, select, and set the table, and return the result to the QueryWindow.xaml interface. These functions wil be explained shortly.

In this section, hopefully you have understood how this class acts as the intermediate class between both the database and the user interface, where it performs the required parsing and handling of data between the two layers.

## Database

Let us now turn our attention to the actual database and talk about how it is stored within the application, both its in-memory volatile structuring and its non-volatile permanently stored data, as held within xml files. Navigate to the DataHandler.cs class for the remainder of this section.

### Memory and File Structure

As briefly mentioned in the introduction of this paper, the underlying storage location that the database would use to store data permanently (in a non-volatile fashion) would be in xml files structured in generally the same way across files. These xml files are stored in the Data folder found in the application folder. The program manages a schema.xml that contains the breadth of tables found in the database as well as each column found in the table along with its specific type as the attribute to the column tag. For each table, a corresponding table.xml exists with each row tag containing each column of the table, and each column of the table containing the values. In this way rows are stored statically inside these files and are added in a linear fashion by appending the new row (with its columns and values) to the end of the xml table file.

Now that we have talked about the non-volatile file structure of the database, let us turn our attention to the in memory object utilized profusely by the database. This object is called a DataSet, which is basically a hash table, or dictionary, of DataTables. Each DataTable was created by .Net to store columns, types, and any number of rows within a row collection. The major challenge was trying to find the best means to store the values in the xml in memory that would facilitate manipulating and querying the underlying data. We first decided upon using a dictionary of strings, and this could have been done, but the cumbersomeness that this would introduce would have far outweighed any benefit. The DataSet on the other hand allowed us to more easily access specific specific tables and specific columns within these tables, while enforcing strongly typed access to the data elements held within the rows within the table. This is as far as the DataTable structure provides in terms of functionality, only providing the ability to store data in a more concrete and accessible fashion.

The remaining tasks of querying consisted primarily of manipulating the DataSet with its DataTables, while keeping the in-memory DataSet synchronized with the xml non-volatile data source. The general idea of synchronization is dealt with as follows: Whenever any tables are created or dropped, or any rows are inserted or deleted, both the DataSet and the underlying xml data source is updated accordingly. Querying need not be synchronized, and thus it fully utilizes the DataSet only to enhance querying speed, functionality, and efficiency. The problem of synchronization is expounded in greater detail in future sections.

### Initial Loading of DataSet

When the DataHandler class is accessed for the first time, since this class is indeed a static class with a static constructor, the Load() method is called once and only once when the DataHandler class is first accessed. This method reads the structure of the Schema.xml file, creates a DataTable for each entry in the file, structures the DataTable by adding new columns, fashions the columns to store the corresponding data types, and lastly navigates to the Table.xml file and populating all the rows of the DataTable. Each DataTable was then added to the DataSet. Remember that the DataSet is simply hash structure of DataTables, whereby each DataTable can be accessed as follows: DataSet.Tables[“TableName”]. The process of analyzing xml files to read tags and values and attributes of tags was made far easier due to the XmlDocument class provided by .Net. Using this class, you can load xml files, navigate through each node (which are children of the initial document tag, or the main header tag of the file), and even navigate through their children. Furthermore, each node could be analyzed for any text values it contained or any attributes it contained, which can then be collected and stored as rows in the DataTable. The main idea to ascertain from this section is the fact that whenever any function is called, whether it is to create a table, insert values into the table, delete tables or rows, or query the tables, the DataSet contains up-to-date information that fully represents the state of the data in the underlying data source. Thus, the static underlying xml file need not be accessed for anything more than the most fundamental of processes (deleting, inserting, creating). This of course means greater speed of querying because only in-memory objects are manipulated.

### Creating Tables and Inserting Values

All throughout the design process of the application a great emphasis was placed on the importance of proper and comprehensive data handling. This is of course what is anticipated and taken for granted in a regular relational database and query language. Thus you will notice that the first three if statements of the CreateTable(..) function makes sure that the table to be created does not already exist in the database, further that the table has at least one column in it, and finally that each column has a corresponding type associated with it. Remember that the parameters are all passed in as either strings or list of strings as parsed from the Query.cs file. As is common throughout the rest of the DataHandler.cs class, Query.cs handles the job of checking for syntactical errors, while DataHandler.cs checks for logical errors of the sort you have just seen. If any errors are thrown, the errors propagate the entire way up the program stack until it reaches QueryWindow.xaml.cs, where it is then caught and a dialog box with the error message is displayed unobtrusively for the user’s benefit.

What proceeds in terms of code in the function is mostly intuitive. The function simply takes the table and column names with their types, creates the table structure, adds the new DataTable to the DataSet, and finally updates Schema.xml with the table definition while creating a new table.xml to prepare for eventual data population.

In the InsertValues(..) function you’ll notice the same stringency of error checking is performed. It makes sure that the table to be populating rows exists in the DataSet, that columns and values align with what is actually found in the database, and that all value types match their given column type through a series of generic casting (found in the CreateColumn(..) method). Here the schema.xml need not be updated, but the rows are indeed created from the list of strings provided to the InsertValues(..) function, then they are also inserted into the table.xml structure. Again the process was intuitive and apart from the intricacies of dealing with programming and manipulation of the data objects as well as the overall design, the algorithm was simple enough.

### Joining

We come now to our next major function. Remember that all parameters from here on out (except for those passed to DeleteTable(..), DropTable(..), and SelectTable(..)) contain fully qualified names. This will help us to join tables on various, create joined result tables that consist of unique column names. It is also important to remember that in the database (DataSet), fully qualified names are not used for columns within each table. Thus, during the process of joining we must make sure to always use the fully qualified name of the table when analyzing the On criteria.

You’ll notice that first we perform an analysis of the tables given to us as well as the On constraints (the On statement and From statement being the only parameters or variables that are needed in this function). We iterate through each expression in the On variable and look for table names. If they are linked via an expression, then we add it to the “cloud” of table names that are linked together by On constraints. If at the end the cloud does not equal exactly the tables in the From variable, not all tables are fully linked and therefore they will not be fully joined based on the subsequent algorithm we use. Therefore an error is thrown. Remember, we do not allow for cross products between tables but maintain that all tables must be joined by some on some column. If it passes this error validation procedure, it proceeds to the main joining algorithm.

Now, we take the first table in the From argument, find the corresponding DataTable in the DataSet and copy it. This table we will call the result table and it is where the resulting joined tables will be added. Then we make sure that each column contains its fully qualified name. We take this result table and look at the next table in the From variable. The On variable is analyzed to find any expressions that directly play a part in this joining because the left and right operands of the expression refer exactly to the tables we’re joining. This subset of expressions is called “keysOfInterest”. This list of expressions is then iterated through one at a time and each row of the result table is compared to each row in the to-be-joined table. If the values match based on all criteria as found in keysOfInterest the row in the join table is combined with the row in the result table and this new row consisting of the row from result and the row from join is coalesced into one row and appended to the end of the result table. The original row in the result is deleted.

Let us now take a moment to explain why we wished to design our joining in this way. Any astute reader might recognize that we could easily have just crudely joined all the tables together without regard to intermediate constraints and then run through the huge end table all at once and deleted rows as needed. Of course this introduces far more bulk than is needed. Chopping the results down a step at a time introduces more of a pipelining mechanism, which is indeed faster. Of course, doing it this way, we were unable to allow users to cross join and thus we had to introduce the initial data-check of “linked” tables. This was one implementation decision that we could either have taken either way, but we instead opted for the more challenging, efficient, and, albeit, gratifying algorithm.

### Filtering

Once the data is joined, via Query.cs, the resulting table is, passed to DataHandler.FilterTable(..) function. This function takes as its parameters the Where clause, which as mentioned earlier, contains all fully qualified names. These names will match the columns of the DataTable passed in (the result table from above), because it was enforced that this was the case. This will allow for cleaner implementation of filtering. This function was most assuredly gratifying to implement. The algorithm is also somewhat intuitive. All the rows in the passed-in DataTable are iterated through. Recalling that the Where variable consists of a lists of lists of expressions, we first iterate through the outer list, which contains the “or” between the other expressions, then we iterate through the inner list, which contains the “and” conjunction. A Boolean variable is introduced and handled as needed to check if the row fully matches the constraints. If we are using this FilterFunction(..) as a means to figure out which row we want to delete, the deleteMatch will be true. Thus if the Boolean variable fully matches the constraints, keep the row unless deleteMatch is true, else delete the row. If the constraints don’t match, perform the exact opposite, deleting instead of keeping.

The main challenge found in this method, in addition to figuring out how to handle “or’s” and “and’s” and managing them as needed, was the challenge of comparing object values (base container for all types in .Net) because unfortunately each row in the DataTable held values that were stored as objects. Thus we had to analyze the type, parse, and if the type was of short, int, long, float, etc., cast them as a decimal so we could truly compare values based on any of the supported operators. This function Compare(..) that we created also allowed us to uniquely test if the type was a string so we could introduce the “startswith”, “contains”, and “endswith” operators in our query language.

### Sorting

Sorting was one of the most exciting algorithm to create in this entire project, other than aggregating data. The parameters passed to the SortTable(..) function is a DataTable, a list of columns to sort, and a list of corresponding sort directions. This function first creates a list of integers, which contain all the rows indices of the table. When we sort, the actual row themselves are not sorted, but the indices are sorted. The row indices is initialized from 1 to the number of rows in the table. It then passes all the parameters to the recursive SortHelper function. This function analyzes a column at a time in the OrderBy parameter and sorts on the row indices indirectly. The sort utilized is the insertion sort. After the indices are sorted, the indices are used to find each group of data, and each group is then sent to another spawning of the SortHelper. In this way, we can sort on any number of columns efficiently and cleanly. Since indices are the only values changing positions within the list of integers, the original rows of the datatable need not be ordered repeatedly, thus introducing superfluous bloat in the process. Once all indices are sorted, a new table is created and each row is imported from the original table in index order.

The challenges found with sorting were minimal. They consisted of designing the algorithm, being precise in the construction process of the algorithm, and lastly, manipulating the data structures. Similar to the filter method, this method also makes generous use of the Compare(..) method to compare objects using the supported operators.

### Aggregating

Should any columns be selected where an aggregating function is used (max!, min!, avg!, sum!, or count!), the GroupTable(..) function is set into action. While the algorithm somewhat complex to implement, the details and precision needed to manipulate the DataTable was an arduous process. Furthermore, the fact that made sure to not set the fully qualified names on any arguments in the Select field until we got to it in the actual SelectTable(..) function in order to preserve aggregation names, aliases, and column names, it made the process that much more complex. Without introducing any unnecessary verbiage, it is sufficient that we simply exposit on the algorithm. A new table is created with each column found in the GroupBy parameter along with a column for each aggregate function (ie, max!columname, etc). Aliases at this point are ignored and excluded from the column name in the new table. Since the original table is already sorted based on the GroupBy columns, the table is already inherently grouped. Thus we felt introducing a groupby keyword into the application was unnecessary. Using the inherent grouping that already exists, we then collapsed the data down so that each group now contains only one column. To condense columns down that were not in the OrderBy parameter, we made sure that each of these columns had an aggregate function attached to them. If not, an error was thrown. Then, the columns were grouped according to the aggregate function. Two new functions to sum and divide objects were needed, since unfortunately the DataTable failed to provide any sort of functionality for comparing between values in rows. The resulting table consisted of a unique row for each group of the original table, with the column names being either the GroupBy name without an aggregate function, or an aggregate function name: (avg!columnname, for instance).

### Selecting and Set

Selecting was an intuitive process. It consisted of manually analyzing each column name. If one does not select on a column in a table, the column is chopped out of the table. The resulting columns are ordered based on the relative positioning as found in the Select parameter. If any aliases exist, the resulting table’s column name is changed to that alias.

Set also was another intuitive process. One caveat though was to make sure that no columns in the table that one desired to store temporarily to the database contain a “.”. The reason for this is that it would not function well with way parsing was being done within the rest of the application. If it contains a period, a corresponding error message prompts the user to make sure to include an alias beside that column.

These presented little challenges in terms of algorithm implementation, but the intricacies of manipulating the string along with aliases and arbitrary aggregate functions presented other unique challenges that had to be dealt with accordingly.

### Deleting and Dropping

To delete a table the drop command is used. The respective function utilized here is the DeleteTable(..) function. Firstly, the table is removed from the DataSet after it is first made sure that the table exists in the database. Secondly, the table.xml file is removed and the schema.xml is updated to reflect the given changes.

To delete rows one can either specify filter criteria within the Where parameter, or exclude the criteria, whereupon all rows of the table are deleted. Since the filter function had already been created with the ability to delete rows in mind, this was a fairly intuitive procedure. Initially, the function checks to make sure the table exists in the database. Next, if the Where parameter contains any expressions, it is sent through the FilterTable, else all rows are deleted. Finally, instead of having to deal with the drudgery of manually going through each row in the xml to see what was deleted, we found it easiest to completely delete all content (ie, rows) out of the table.xml file and, using the DataTable as a reference, repopulate the xml file with the latest data. This is perhaps an inconveniently slow way of deleting rows from the underlying xml file, and future work on this project might be to implement this way of doing things. However, the process we used yielded acceptable results and was sufficient for the scope of this project.

### Importing arbitrary XML Data

We now approach the xml querying part of the project. This is not to diminish the process of querying xml data, but shows the user that by harvesting xml data and forcing the values in rows in columns in a new table in a relational database, this not only converts the problem of querying xml data into a problem of querying relational data, but allows the full querying power of a relational querying language to be fully realized.

Using the .net XmlDocument class, the xml file is loaded. Subsequently, the recursive HarvestData(..) function is called for the first time and is passed the document header tag (the highest level singleton tag surrounding any xml data). This function is also passed a DataTable and a row where the structure of the DataTable consists of the header and descendents, or children tags. It should be understood that the user specifies a header tag and a set of children tags. Thus the DataTable we create consists of a column for each children tag and the DataTable is given the header as its name. This is just temporarily to easy the process of harvesting and storing the data recursively. The HarvestData(..) function recursively navigates through every single tag, or node, in the arbitrary xml file. When it finds a tag that matches, case insensitively, any of the columns, it takes the value that is inside the tag, and adds it as a string to the current row in the respective cell corresponding to the column. If a header tag is reached, the contents of the previous row are added to the actual DataTable, and the current row is cleared to begin populating again. Once the DataTable is fully populated with data, the ImportTable(..) function dynamically creates a query to create the table and insert values into the new table in the database. The real table name is given the name of the alias as optionally specified by the user.

Limitations of querying xml data in this manner are as follows. Should the user desire to preserve the original xml structure of the data it would be impossible since our implementation only harvests data into columns and rows in our database. Furthermore, if any nested tags exist it is currently impossible to record such higher level data. This can be observed in the flureport.xml file, where there are many states but there is a larger grouping where groups of states are surrounded with a time period tag which includes the number, year, and subtitle, which act as additional information. Currently we can only possibly obtain a state table with columns being abbrev, color, and label, thus losing the timeperiod information. In addition, we cannot currently query attributes that exist within the tag, but currently we can only harvest the text that exists within the enclosing tags, again losing some of the intrinsic value with true xml querying. Of course, with limitations there are a few benefits. One is that relationally users need not learn another query language and one can enjoy the full breadth of querying abilities on the file once the xml file is imported into the database. Another might be that applications can make use of the relational database more readily than an xml querying language since their results are only given as another xml file.

# Test Cases

The following are some test cases that users might chose to follow in testing the application. Of course, they are free to test as thoroughly as they desire. The provided test cases will perhaps guide them in their future testing endeavors.

1. Create/Insert
   1. Start the application and type in the following queries:
      1. create test1 string col1 int col2 float col3 boolean col4
      2. insert test1   
         values a 1 2.3 true   
         values b 1 3.3 false   
         values c 1 5.5 true   
         values b 1 4.4 true  
         values e 3 5.4 false  
         values e 4 5.3 true
   2. Click the log and schema buttons, then finally click the Query button.
   3. Now type in the query:
      1. Select \* from person
   4. Notice that the database correctly creates and inserts values
2. Select/From/Where
   1. Type in the following queries that will utilize the provided data files that came with the application.
      1. Select \* from person where name startswith [allen bru] or name endswith brubaker
      2. Select \* from person where name contains brubaker and height > 5.0
      3. Select \* from person where age > 53 and name contains a and height < 6.6
   2. Press the Query button.
3. Join
   1. Type in the following queries
      1. Select \* from join1 join2 on join1.key j = join2.key
   2. Press the Query button.
4. OrderBy
   1. Type in the following queries.
      1. Select \* from sorttest
      2. Select \* from sortttest orderby col1 col2 col3 col4
      3. Select \* from sorttest orderby col4 col3 col2 col1
      4. Select \* from sorttest orderby col1 col2 desc col3 desc col4 ascending
      5. Select \* from sorttest orderby col1 desc col2 desc col3 desc col4 descending
   2. Press the Query button.
5. Aggregation
   1. Type in the following query
      1. Select \* from sorttest
      2. Select col1 col2 count!col3 max!col3 min!col3 sum!col3 avg!col3 from sorttest orderby col1 col2
   2. Press the Query button.
6. Delete/Drop
   1. Type in the following queries.
      1. Delete test1 where col1 = b
      2. Drop test1
7. Import
   1. Type in the following query
      1. Import data/flureport.xml state:ImportTest abbrev color label
      2. Select \* from ImportTest
   2. Press the Query button.

# Conclusion

Inevitably, as all projects tend to do, the application must approach an ending point. The application in its entirety consists of two parts—both an expressive relational querying language that manages an underlying database storing its data entirely in xml data, and secondly, a means to harvest additional data from an arbitrary xml file in populating a relational database that can then be fully queried. The querying language not only is able to create and delete tables or even simply statically view the tables, but it can query the resulting table in a wide plethora of manners, including giving the user an ability to join across any number of tables, sort the data such that the resulting data is understood in a more logical fashion, filter results based on a complex set of criteria, group the data using any number of aggregate functions, store temporary data to further query upon, and include aliases to allow users to rename columns. Being able to query in such an expressive sense allows users to not only just observe static data but pick apart the data and learn far more than was previously possible. Alas, this is what a true relational database attempts to do, and we feel that our application has in a small way emulated the workings of a real world relational database. Surprisingly, the various limitations inherent in our querying language can be worked around and it can be found that the querying language is more expressive than previously thought. For instance, we only allow the user to, in the where keyword, to use and’s or or’s, but no parenthesis. This enforces a disjunctive normal form yet any set of expressions can be reduced to this form. Similarly, another limitation is that one is not able to join tables to themselves even when a table is renamed. This can be worked around by the user utilizing the set keyword in temporarily storing the exact same table as another name in the database in memory, then joining the original table to this temporary table.

Lastly, we attempted to give users the ability, rather than laboriously inputting row by row into newly created tables, to import data from any arbitrarily structure xml file. It was found that while this process was not as fully expressive and able to query xml data as say a true xml querying language might be, it serves its purpose quite well in query basic xml files and importing large sums of data into our database.

If additional time was allotted in order to enhance the current relational database, various features might be implemented. We would certainly like to be able to implement a more powerfully querying tool in order to parse queries. This might include implementation an expansive and powerful parse tree. Furthermore, we would also like to be able to introduce the concept of null values into our database, enforcing primary keys that must be unique, perhaps giving the a column a seeding ability that would increment by one when a new column is added, and last but not least introduce foreign key constraints, cascading deletes, and the like. And of course, we would desire to implement some sort of query optimization engine that would reconstruct the query in the most optimal sense, store the result in a query tree, and optimally query the database using pipelining techniques. However, we constrain ourselves from digressing to this fantastical oneiric thought.

# Resources

MacDonald, M. (Feb 2008). *Pro WPF in C# 2008: Windows Presentation Foundation with .NET 3.5, Second Edition.* Apress.

Microsoft. (n.d.). *MSDN Library*. Retrieved from Microsoft Developer Network: http://msdn.microsoft.com/en-us/library/default.aspx

Troelsen, A. (Nov 2007). *Pro C# 2008 and the .NET 3.5 Platform, Fourth Edition.* Apress.