**Evolving Dataset Storage Using a General Purpose Schema**

**Computer Science 380 Final Project**

**Section 1: Background and Proposal**

Many form generation websites exist, but a common problem plagues most, if not all of the sites. This problem relates to the storage of the data that the thousands upon thousands of forms generate every day. When a form is used to submit data, three primary methods of storage are currently in use, each with their own benefits and problems.

Storing form data in an XML database is extremely common for websites that require the ability to modify a form after it has begun collecting data. With an XML file, a schema can change on the fly, creating an environment that can start with a certain number of data fields and can end with a completely different number of fields. This may be tempting, but there is no simple solution to securing the data. If a user has the ability to access the XML file, they have access to all of the data within it. In addition to security vulnerabilities, XML also has an inherent storage overhead that can be similar to, or significantly higher than a database depending on the tag length.

Another common method of storing data is one which users of Google Forms will be familiar with. These forms, upon the first piece of submitted data will create a spreadsheet on the fly that is tailored to the form. This approach allows for easy data collection and is very versatile once downloaded. These spreadsheets have a slightly higher storage overhead than a relational database because they contain formatting information, metadata, and other form specific data. In addition, once data collection is started, it is not possible to make last minute modifications to the form. To do this, the data must be reset and the spreadsheet needs to be recreated to allow for the updated schema requirements. Similar to the XML file, the spreadsheet is difficult to grant users access without allowing all of the data to be seen.

Finally, another method of data storage used commonly uses a series of arrays that are posted, or appended to and resubmitted each time data is submitted using the form. This method is extremely space minimizing as the arrays are held in memory or in a temporary cache on the disk. In many cases this can be extremely memory intensive having to recall these arrays before each submission.

The proposed project implements a general purpose schema that allows any number, size, and organization of forms to be submitted to the same database and recalled using basic SQL queries. This schema is organized by input type; because there are a finite number of possible input types in a web form, it is feasible to use one table for each element type to store the data. In addition, because the database can be restricted to specific users, security and access restriction is intuitive and efficient.

**Section 2: Technologies**

This project will implement several programming languages and technologies working seamlessly together to create an intuitive operating environment. The core language used to implement the necessary database communication and data collection functionality will be the Hypertext Preprocessor language. PHP is used to perform many functions from form data processing, database interactions, user interactions and much more. SQL strings can easily be used with PHP to interface with the database in a useful manner. To create the user interface so the data can be submitted in a user friendly and visual manner, the Hypertext Markup Language will be used. HTML is primarily used for visual effects, interactions, and formatting of data. PHP can be seamlessly integrated in line with HTML by using the opening <?php and closing ?> tags. In addition to directly coding in HTML, JavaScript will be used for advanced interactions such as table sorting, data visualization, and web interactions with the users system, such as print dialogue invocation.

To be able to store data, I will be utilizing a preconfigured installation of MySQL on a remote web server. The general purpose schema will be implemented using the query dashboard which allows the execution of SQL code without the need to embed queries in PHP to run them. After setting up a basic schema, all of the database tables were switched to InnoDB to enable support of Foreign Key constraints which are heavily relied upon in the proposed schema.

**Section 3: Difficulties**

Learning three languages on the fly proved to be a difficult but rewarding task. Having some background in HTML I was able to assemble a simple interface that allows the user to interact with the website and data. PHP proved to have many challenges. PHP assignment statements work in any segment of code, be it a condition check, a loop, or even in an incorrect piece of code. Because PHP does not have a native programming environment with a debugger and error messages which are cryptic if they appear at all, debugging code was a real challenge.

In addition to difficulty debugging code, I found that when fetching arrays of results row by row after running an SQL query, the cache of returned data is dropped as you walk through the entries. Once the end of the list is reached, the previously accessed items are purged from memory. The first iteration worked flawlessly, but when accessing the result array a second time, null results were returned, which didn’t become apparent until a much later time. When viewing result arrays, only the object itself can be viewed unless a special dump() method is invoked.

When working with PHP, if an element in a form is left empty, a value is still posted, which could be a null value or an empty string. The only element that does not post every time is a checkbox. If unchecked, the element doesn’t post. This was a problem because a situation where a user left any number of boxes unchecked followed by a checked box, the data would never be submitted due to the nature of the form processing required by the generic schema.

Finally, the last big difficulty I encountered was server load. Originally, every element required multiple sql lookups to locate option IDs and then lookups for the associated element ID. After all of the element information was gathered from the database, it needed to be submitted to the data store with the posted data. Depending on the element and the number of elements on the form, this could possibly lead to hundreds of queries per page submit. After receiving an email from the web host about an unusually high server load coming from our account I decided to reevaluate the process. To circumvent this load problem, a list of element information and option information was queried at the load of the processing file. After this, any data requests were directed to the cached array of information. This reduced the number of queries to n+1 where n is the number of elements in the form. After this point, the server load remained mostly unaffected. To give assurance that I had ample processing power our memory limit was temporarily raised and extra bandwidth was purchased to counteract the continuous uploads and downloads.