Providing public access to a scientific database while maintaining data integrity

Lauren Foutz
David Spooner

February 23, 2004

Abstract

The philosophy behind our geodetic database is that it is to be a place where geoscientists can freely submit their data and browse the data of others. This philosophy boils down to two major design goals that apply to any scientific database open to public access. First that submitting and browsing the database should be as simple as possible; otherwise users will abandon the database out of frustration. Second guarantee the integrity of the scientific information, otherwise users will not trust any of the information they find in the database. The implementation of these two often contradictory goals is presented in this paper.

Contents

1 Introduction

Subduction zones happen when two tectonic plates of the earth's crust collide and one slides under the other.[5] Geoscientists can use geodetic data collected at these zones to determine their seismic hazard.[7] In this context geodetic data is data describing the velocity, deformation, elasticity, and many other properties of the tectonic plates of a subduction zone.[8] Although a lot of geodetic information has been gathered in the last century, there is a lack of simple applications to organize and model this information. The goal of this project is to provide a user friendly application for storing and modeling geodetic information, which is accessible over the internet.

The project is implemented as a series of dynamic web pages that allow the user to manipulate the inputs and outputs of the actual modeling applications. This paper focuses on input manipulation, specifically the selection of geodetic data to model. Java Servlet technology[4] is used to create the dynamic web pages, and the Java Runtime class is used to call and monitor the modeling applications.

A naive implementation of the project interface would have the user submit her data directly to the modeling application. While this method would allow users to model geodetic data, it would not allow them to store the information. If the data is not stored by the project then not only will it be difficult for the user to model multiple datum, but it will be impossible for geoscientists to share their data. The ability for geoscientist to study data gathered at subduction zones they have never observed and to compare data is a major goal of this project. To implement this goal a geodetic database was created to store and organize data contributed by the users of this project.

2 The Geodetic Database

The database is a relational database managed by MySQL 3.23 for the Solaris 9 operating system. It consists of three tables, a geodetic data table that contains individual subduction zone observations, and a geodetic correlation table that contains information shared between entries in the geodetic data table. The third table contains a list of user names and passwords, and is used to allow users of the project to identify themselves and mark information as belonging to them.

The geodetic data table consists of the following fields:

Field	Type	Null	Key	Default	Extra
id	int(11)		PRI	NULL	auto_increment
obs_type	varchar(20)				
longi	float			0	
lat	float			0	
obs	float			0	

azimuth	varchar(7)	YES		n/a	
sigma	float			0	
site_name	varchar(4)	YES		NULL	
contributor	varchar(40)	YES		NULL	
user_name	varchar(20)				
visible	enum('yes','no')	YES		yes	

The fields *id, user_name,* and *visible* are used for administrative purposes and are not visible to the user. *id* is used to identify each row in the table and access it quickly. *user_name* is a foreign key from the user table, it identifies who contributed the row and allows the project to restrict certain operation on that row to people logged in under that username. *visible* has two values, 'yes' or 'no', where 'yes' marks that the row can be viewed by all who access the project, and 'no' marks that the row can only be viewed by people logged in under the name in *user_name*.

The fields *obs_type, longi, lat, obs, azimuth, sigma, site_name,* and *contributor* contain the geodetic information. *obs_type* contains the observation type of the geodetic observation. This database allows data of six different observation types, GPS, strain rate, tilt rate, slip vector, slip rate, and transformation azimuth, each of which measures a different property of a subduction zone. GPS for instance measures the vector of the movement of a plate, while tilt rate measures the rate of vertical movement of the plate that subducts.

longi and *lat* are the estimated longitude and latitude of the area that is observed, while *site_name* is the four character alphanumeric name given to the area by geoscientists. *obs* is the measurement of the observation, what exactly it measures depends on the observation type. *azimuth* is the angle, measured counter clockwise from East in this database, that defines the exact direction of the measurement, it is not applicable to every observation type. *sigma* is the standard deviation, or the estimated error, of the measurement value in *obs*. Finally, *contributor* is the name the user submits as the person or organization that contributed the information to the database.

Four of the observation types, tilt rate, slip vector, slip rate, and transformation azimuth, contain only one measurement, so each observation takes only one row in the table. GPS and strain rate can contain up to three measurements, each of which has its own standard deviation, so each observation can take up to 3 rows in the table. There is also a standard deviation that describes the correlation between the three different measurements of an observation. These correlation values are stored in the geodetic_correlation table, which contains the following fields:

Field	Type	Null	Key	Default	Extra
ref1	int(11)		MUL	0	
ref2	int(11)		MUL	0	

```
| correlation | float   |      |     | 0       |       |
+-------------+---------+------+-----+---------+-------+
```

ref1 and *ref2* are foreign keys related to the *id* field in the first geodetic table. These are the rows that the correlation value stored in *correlation* relates.

Designing the database is simple, but creating an interface that implements the two goals of the project is more interesting.

3 User Login

The concepts of registration and user login violate the first goal of the database interface, that of simplicity. The thought of long registration processes, giving out personal information such as e-mail addresses, and having to remember passwords will prevent some users from using the database. However, in order to implement the second goal, that of data integrity, there must be a way for users to identify themselves to the interface. For instance if a user wants to delete or update her information in the database the interface must be able to verify if she is the one who indeed submitted it. These two conflicting goals where the motivation behind the very simple user login process of the project.

A user can search the database without having to log in, but if a user wishes to submit or manipulate information she must first log in so the interface can verify what information she owns. The user table, which is part of the database, consists of the following fields:

```
+-----------+---------------+------+-----+---------+----------------+
| Field     | Type          | Null | Key | Default | Extra          |
+-----------+---------------+------+-----+---------+----------------+
| rowID     | int(11)       |      | PRI | NULL    | auto_increment |
| user_name | varchar(20)   | YES  | MUL | NULL    |                |
| password  | varchar(20)   | YES  |     | NULL    |                |
| updated   | timestamp(14) | YES  |     | NULL    |                |
+-----------+---------------+------+-----+---------+----------------+
```

Where *rowID* is an auto incremented primary key for the table, *user_name* is the log in name that the user picks for herself, *password* is an optional password picked by the user, and *updated* is a timestamp that is used to identify entries that are old enough to be deleted. When a user accesses any of the Servlets of the project the Servlet checks for the most recent Cookie with its value field set to the value of the *rowID* field of one of the rows of the User table. If no such Cookie is found a new Cookie is created and its value is set to a new, null filled, row of the User table. At anytime the user can log in using either the registration box if she has not registered a user name yet, or the login box if she does have a user name. The login box creates a new Cookie and sets its value to the *rowID* of the row in the User table where the *user_name* matches the one entered by the user, or returns an error if no match is found. The registration box creates a new Cookie and sets the value of the Cookie to the *rowID* of a

new row created with the user name the user entered, or returns an error if that user name already exists in the table.

Users cannot access the submission, update, or delete web pages unless the Cookie points to a row with a non-null value for *user_name*. When a user submits information the *user_name* field of the geodetic data table is automatically set to the value of the user name identified by the Cookie. If the user attempts to delete or update any information in the geodetic data table then the statement *user_name='current_user_name'*, where *current_user_name* is the user_name pointed to by the Cookie, is added to the *where* clause of the SQL statements.

Currently the Cookies are sent between server and client without encryption. Since the Cookies only contain the index of the rows of the User table, they will not be very useful to anyone intercepting the Cookies. However, there is nothing to stop a hacker from spamming the project with Cookies with made up values until she stumbles across a useful one. This will allow a hacker to delete and alter information she does not own, but the interface will still block the hacker from submitting nonsense data, although she could have tried to submit to the database without stealing someone else's user name, or making nonsense alterations to data already submitted. Since geodetic data does not have a wide commercial value the security scheme for this database was designed more to protect against user error than hackers. A method for implementing Cookies that are protected against these threats is described in [6].

4 Searching the database

With the database designed and a user login feature implemented it is now possible to design an interface to allow users to search the database. This project provides two ways for users to search the database, they can select search parameters from an html form, or select a region from a map and have the database return all information contained in that region. In either case the information the user submits is passed to a Java Servlet that uses the parameters to build an SQL select statement.

When users submit information to the database they are given the option to hide the information from anyone not logged in under their current user name. Although this option defeats the main purpose of the database, which is to allow researchers to share their information, this feature was added to allow some researchers to make use of the modeling application without having to share their research before they can publish it. To implement this feature an enumerated field was added to the geodetic data table called *visible*. This field is set to *no* if the user wishes to hide her data, and *yes* otherwise. When the select statement of the two search functions is built, the condition *visible='yes' OR username=current_users_name)* is added. The variable *current_user_name* is set to the value of the user's user name.

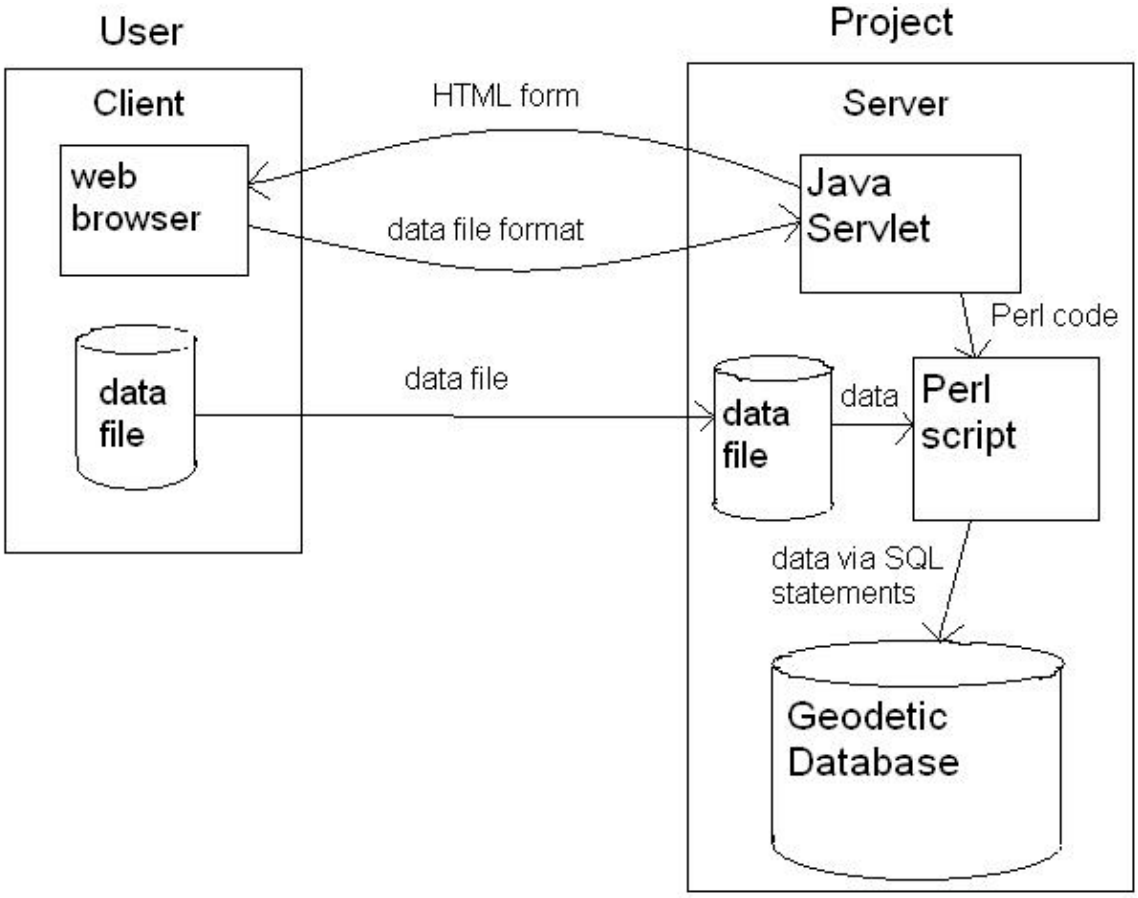
5 Submitting Geodetic Information to the Database

The implementation of the submission process in any public database is the key to its success or failure. If the process is too difficult then few people will be willing to contribute to it. If the information is not properly screened then careless and malicious users can fill up the database with nonsense entries. If people browsing the database see a string of letter and control characters where there should be a decimal value, then they will not trust the accuracy of any of the information they find. The next two subsections discuss how to simplify the submission process, and then how to screen the submitted information. The third subsection shows how the submission process was implemented for this project. After these subsections there is a discussion of the update and delete features of the interface and how they compliment the project goals.

5.1 Designing a Simple Submission Process

One might be tempted to require that users submit their findings as a file with the data arranged in a standard predefined format. While this method may be easy to implement and easy for researchers who usually store their data in that format, it is extremely inconvenient for those who do not. Rearranging a file to meet a certain format is time consuming, especially if it is a large file and the person does not know how to use any of the commands or tools that aid in this task. This can be especially complicated if a commonly accepted format does not exist for the data being collected. A better solution is to require the user to submit information describing the format of her file along with the data file, then using that information to create a tailored program to load the data file into the database.

The diagram below shows the architecture of the submission process:



The process begins with the user filling out an html form that passes the information on to a Java Servlet. The Servlet uses that information to write a download script in Perl and then executes it asynchronously. The Java Servlet could have been used to download the information, but creating a separate script has many benefits that offset the extra memory usage. First the download scripts and their associated data files provide a backup to the database. Second a separate script can be executed asynchronously. That way the user does not have to wait while a possibly large file with many complicated consistency checks is executed. Also, if the server is experiencing heavy traffic then it can put off running the download script until a time of low activity. Plus, the extensive support for regular expressions and string manipulation in Perl make it easy to check data integrity.

The submission form for this project is shown below:

Data Positions

Longitude	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Latitude	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Observation	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Sigma	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Velocity East (mm/yy)	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Velocity North (mm/yy)	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Velocity Up (mm/yy)	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Sigma East	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Sigma North	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Sigma Up	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
North East Correlation	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
East Up Correlation	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
North Up Correlation	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Strain Rate EE (10^-6/yr)	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Strain Rate NN (10^-6/yr)	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Strain Rate NE (10^-6/yr)	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Sigma EE	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Sigma NN	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Sigma NE	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
NN-EE Correlation	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
EE-NE Correlation	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
NN-NE Correlation	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Azimuth (degrees CCW from East)	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Site Name	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th
Name of the Contributor of the Information	○ 1st	○ 2nd	○ 3rd	○ 4th	○ 5th	○ 6th	○ 7th	○ 8th	○ 9th	○ 10th	○ 11th	○ 12th	○ 13th	○ 14th

Select the Observation Type (required)

Enter the name of the person or organization providing the information (if all the information is provided by one person or organization and the name is NOT provided in each line the file)

Enter the character(s) used to separate individual data, leave blank if you just use a space or a tab.

Enter the character(s) (if any) used to indicate comments in the file

Data File

Browse...

e-mail Address (so you can be contacted about any problems loading the file)*

Do not allow others to view this information.

Submit Query Reset

The first part of the submission form is the Data Positions section. Since the data is submitted as a file with columns of information the user must state what data types she is submitting and in which columns they are stored. The user can do this by selecting radio buttons listed in a table that relates data types and positions. For instance if the user stored the coordinates of longitude in the 2nd column, then she would select the radio button labeled 2 that is located in the row labeled longitude. Next the user must select from a pull down menu one of the seven types of geodetic data that the database contains. After that is a text box where the user can write her name so it can be displayed with her data, her name can also be submitted in the data file. After that the user is asked to enter any characters that represent data deliminators (such as commas) or the beginning of comments. Then the user selects the data file from a file box. Next the user enters her e-mail address so she can be informed if the file is not loaded into the database correctly, and why. Finally there is a check box that the user can select if she wishes for her information to be visible only to people logged in under her current login name.

When the user hits submit, the information is passed to the doPost function of the submission Java Servlet. The doPost function is shown and explained in the subsection Implementation of the Submission Process below.

With this submission process the format restrictions on user data files are minimal. For some of the restrictions it would be odd if the user did not already follow them, such as the data must be in a file of text, each entry has to be on a separate line, and each field has to be separated by some character, such as a comma or space. The requirement that the data file only contain information of one type of observation data is a bit restrictive, but must geoscientists naturally follow this format. The most restrictive requirement is that the measurements must be in a certain unit. The geodetic database exists so that users can store information they wish to model using the project's modeling applications. These applications expect certain units. In the case of longitude, unit conversion can be done automatically. The project requires that longitude me measured from -180.00 to 180.00, but many geoscientists measure it from 0 to 360.00. The conversion is done by subtracting 360 from any longitude value greater than 180.00 but less than 360.00. Unfortunately not all conversions are as easy to implement.

5.2 Maintaining Data Integrity

Data integrity is vital to scientific databases. If a search of the database brings up entries that are obviously false then the user will not trust any information in the database. Fortunately the very nature of scientific data often allows for consistency checks that can easily screen out malicious and careless users.

As a simple example, imagine a database called the Triangle Database. As the name implies the Triangle Database contains information on triangles, the degrees of the three angles and the length of the three sides. Users can submit descriptions of any triangle to the database, and can view the submissions of others. The designers of the Triangle Database only want valid triangles to

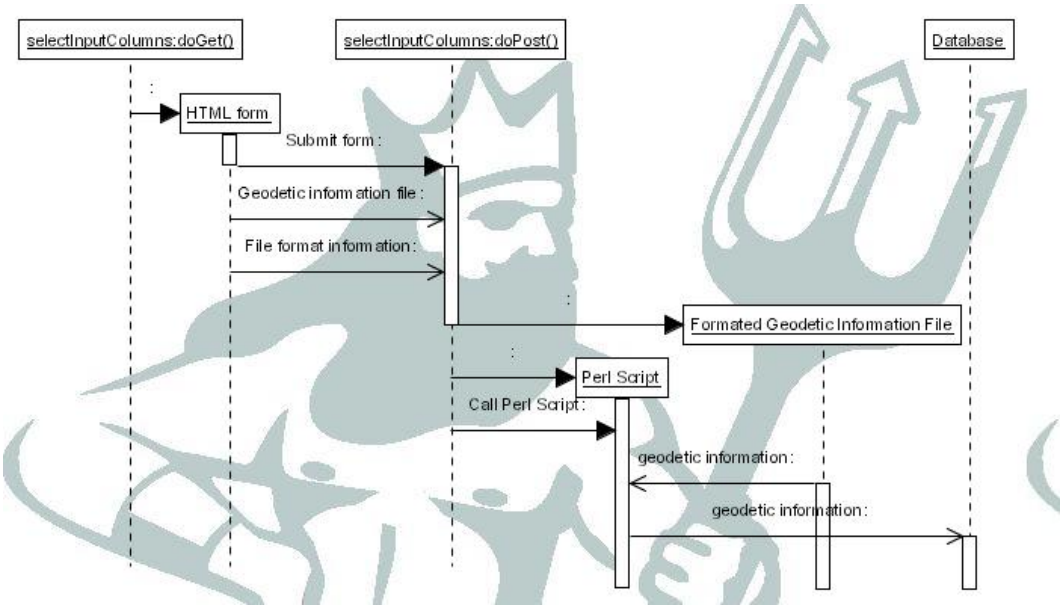
be loaded into the database, so they use consistency checks on the nature of triangles to screen out bad entries. A triangle must have three angles and three sides, so any entry that does not have these six fields is rejected. The angles of a triangle cannot exceed 180 degrees, and the sum of the three angles cannot exceed 180 degrees. So each individual angle is checked against this, and the three angles are summed and checked. The longest side of the triangle must be opposite the largest angle, and the smallest side must be opposite the smallest angle, so the length of the sides can be checked against the size of the angles. If a user submitted a triangle with one angle equal to 190 degrees, then she will be sent an error message stating that that angle has an invalid value. With these checks in place careless users can be caught and malicious users can be frustrated.

This method is even more effective in discouraging malicious users in the geodetic database, since the subtleties of subduction zone observations are not widely known. The submission interface of the geodetic database checks the values submitted by users against regular expressions and boundary conditions. For instance any value submitted for latitude is checked to see if it is a number value between -90.00 and 90.00. All necessary fields are checked to make sure they are present, and fields that are valid for some observation type and invalid for others are also checked. For instance every entry must have a longitude and latitude, and entries that are of the type GPS can have a Velocity East field, but not a Strain EE field. Also any directional observation type must have consistent direction, for example Strain EE must have a Sigma EE, and can have a Correlation EE NN or Correlation EE NE, if the corresponding Strain NN or Strain NE and Sigma NN or Sigma NE are also submitted. Any users whose submissions break these rules are informed of the error. For a more detailed description of how these consistency checks are implemented, see below.

Granted an entry that passes all these checks can still be wrong, but most will be detected. Plus disagreement in observations is part of the scientific process, and variations between observations of the same area can draw researchers' attentions and motivate them to study the area further. If a user sees a discrepancy between her observations and someone else's, and finds her observations to be wrong, then she can use the delete and update features to correct her mistakes.

5.3 How submitted data is passed through the submission components

The flow of the submission process is show below:



The function doGet of the class selectInputColumns, which is an extension of the HttpServlet class, creates the html form in which the user enters the file containing her geodetic information and information on the format of that file. When she submits that information the function doPost processes it. doPost carries out two major functions, saving the geodetic file on the server, and writing the Perl script that loads it into the database. The file is saved in a folder named after the user's database number and the file is named after the user's login name plus a number. The Perl script for that file is then created and named after the file. Most of the Perl code is stored as a string in the Java servlet. When the code is written to a file the only dynamic information that is needed is the value for some Perl variables and the position of the data types in the data file. Once the file for the Perl script is created the function begins to write the lines of code. The Perl script begins with initializing some Perl variables with values retrieved from the html form and connecting to the database. Then the Java servlet writes a section of Perl code for handling each of the 25 different types of information that the user could submit. Since there will never be a case where all 25 types are submitted in the same file, only the code for the types in the file is written. Finally error handling and code to email the submitter of the file is added to the Perl script. When the script is finished the Java servlet runs the Perl interpreter asynchronously and exits after printing a message to the user. The script connects to the database and opens up the geodetic data file. Then it reads each line of the file and loads it into the database using an SQL Insert statement. If the script finishes reading the file without encountering any errors it saves the changes to the database and finishes. If an error was encountered then it erases the changes and finishes.

5.4 Implementation of the submission process

The submission process is implemented by having the user fill out an html form about her data file and passing that information onto a Java Servlet. The Servlet uses that information to write a tailored Perl script to load the data file into the database. An outline for the Servlet doPost function that processes the form information and writes the script is shown below:

"submission.java" 13 ≡

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
       throws ServletException, java.io.IOException
   {
       ⟨Establish a connection to the database and get user Cookie. 14⟩

       ⟨Get the user's username. 15⟩

       ⟨Get the user email and data file. 17⟩

       ⟨Create the Java hash table that contains the data fields. 18⟩

       ⟨Write the Perl script. 19⟩

       ⟨Run the Perl script using the Runtime Java class. 38⟩

       footer.writeFooter(out);
       try{
           Con.close();
       }catch (SQLException e){
       }
   }
```

The first thing the function does is establish a connection to the database and find the user's Cookie. A connection is established using the getConnection function of the DriverManager class. After a connection is established the call to getCookie of the getItrCookie class finds the correct Cookie the user has sent the server. The value of the Cookie is the *rowID* of the user's entry into the User table in the database. This value is stored into the variable *rowID*.

⟨Establish a connection to the database and get user Cookie. 14⟩ ≡

```
            if (req.getParameter("login") != null){
                doGet(req, resp);
            }
            resp.setContentType("text/html");
            PrintWriter out = resp.getWriter();
            header.writeHeader(out);

            Connection Con = null;
            try{
                Class.forName("com.mysql.jdbc.Driver").newInstance();
                Con = DriverManager.getConnection("jdbc:mysql://localhost/
    ITR?user=itr&password=nottherealpassword");
            }catch (Exception e){
                log("Exception " + e);
                out.println("Database error, please contact the server
    administrator.<br>");
                footer.writeFooter(out);
                return;
            }

            Cookie userCookie = getItrCookie.getCookie(req, resp, Con);
            if (userCookie == null){
                out.println("Unable to create a Cookie for your session,
    if you do not accept cookies then you cannot access this system.<br>
    If you do accept cookies and have gotten this error please contact the
    administrator.<br>");
                footer.writeFooter(out);
                log("Error either with the database or with retrieving
    cookies in selectInputColumns.java");
                return;
            }

            String thisUrl = "http://" + req.getServerName() + ":" +
    req.getServerPort() + "/itr2/selectcolumns";
            loginBox.drawLoginBox(out, userCookie.getValue(), Con, thisUrl);
            String rowID = userCookie.getValue();
```

Used in part 13.

The function then uses the value stored in *rowID* to look up the user's username in the User table. The username is used to mark any data the user submits as belonging to her, so only she will be allowed to delete and update that data. If the user is logged in she will have a non-null username, if not then the function will tell the user she must log in and the function will return. The value of username, if it exists, is stored in the variable *username*.

⟨Get the user's username. 15⟩ ≡

```
            String user_name = null;
            try{
                Statement stmt = Con.createStatement();
                ResultSet user = stmt.executeQuery("select user_name from
    User where rowID = '"  + rowID + "'");
                user.first();
                user_name = user.getString("user_name");
                if (user.wasNull()){
                    out.println("You must log in before you can submit
    information to the database.<br>");
                    footer.writeFooter(out);
                    return;
                }
            }catch (SQLException e){
                out.print("Error accessing database: " + e + "<br>");
                footer.writeFooter(out);
                return;
            }
```

Used in part 13.

With the *username* of the user found, the function can now save the data file in the directory, whose path is stored in the *dataDir* variable, under the name $< username > \_ < anumber >$. The number is added to the file name so the user can submit multiple data files under the same username without a name clash existing between files. If the function encounters any problems in writing the Perl script then the data file and unfinished Perl script will be deleted.

Since the Perl script is run asynchronously from the Java Servlet, the user is required to submit an email address so the Perl script can email her with an error message in case it is unable to load her data file. The Perl script sends email using the Mail::Sendmail module.

The MultipartRequest class is part of a package designed to handle form data of type *multi/request*. It is described in the book Java Servlet Programming[3].

⟨Get the user email and data file. 17⟩ ≡

```
        MultipartRequest multi = new MultipartRequest(req, dataDir,
    1024*1024);
        Enumeration file = multi.getFileNames();
        if (!file.hasMoreElements()){
            out.println("No file was loaded.  You must load a datafile.
      Note older browsers do not support loading files so if you did load
    a file and got this message that could be the problem.<br>");
            out.println("<a href=" + thisUrl + ">Previous Page</a>");
            return;
        }
        File datafile = multi.getFile((String)file.nextElement());
        String userName = userCookie.getName();
        String filename = user_name + "_";
        File newDataFile = new File(dataDir + "/" + filename);
        int i = 2;

        String email = multi.getParameter("email");
        if (email.length() == 0){
            out.println("You must submit an email address so you can
    receive information on how to update your entries and be informed if
    there where problems loading your file.<br>");
            out.println("<a href=" + thisUrl + ">Previous Page</a>");
            return;
        }

        String scriptName = null;
        synchronized (newDataFile){
            while (newDataFile.exists()){
                newDataFile = new File(dataDir + "/" + filename + i);
                i++;
            }
            i--;
            if (i != 1){
                filename = filename + i;
            }
            scriptName = filename + ".pl";

            if (!datafile.renameTo(newDataFile)){
                out.println("Could not save file.<br>");
                out.println("<a href=" + thisUrl + ">Previous Page
    </a>");
                return;
            }
        }
```

Used in part 13.

Now the Servlet reads in the values of the radio button groups that describe what columns of the data file hold what data types. This information is used to create a Java hash table, where the number of the radio group, also the position of the field, is the value and the key is the name of the data field. The Perl script reads each line of the data file and splits it into tokens, using the deliminators specified by the user to mark where the line should be split. Using the Java hash table and the Perl array of tokens, the Servlet can write a command in the Perl script to find the value of longitude using the following command:

```
perl_script.write(‘‘$longitude = @tokenArray[‘‘ +
valuePositions{‘‘longitude’’} + ‘‘];’’);
```

⟨Create the Java hash table that contains the data fields. 18⟩ ≡

```
            Map valuePositions = new Hashtable();
            Integer tempX;
            for(int x=1; x < 21; x++){
                tempX = new Integer(x);
                if (multi.getParameter(tempX.toString()) != null){
                    valuePositions.put(multi.getParameter(tempX.toString())
    , tempX);
                }
            }
            if (valuePositions.isEmpty()){
                out.println("No positions of data where entered.  Go back
    and enter them.<br>");
                out.println("<a href=" + thisUrl + ">Previous Page</a>");
                newDataFile.delete();
                return;
            }
```

Used in part 13.

With the preprocessing done the function can now write the Perl script, which is accomplished as follows:

⟨Write the Perl script. 19⟩ ≡

⟨Create the file for the Perl script. 21⟩

⟨Connect to the database and define the prepared SQL statements. 23⟩

⟨Open the data file and read each line. 25⟩

⟨Write a routine for each valid observation. 26⟩

⟨Enter correlation values into the geodetic correlation table. 35⟩

⟨If script executed email the user with success. 37a⟩

⟨If an error is thrown email the user with the error. 37b⟩

Used in part 13.

The Perl script is named after the data file, with ".pl" added as a suffix. The file handle of the Perl script is *perl*. The first thing the Perl script does is declare certain variables, such as *$email*, *$user_name* and *$contributor*, and sets them to the values known by the Java function.

⟨Create the file for the Perl script. 21⟩ ≡

```
        File perlScript = new File(dataDir + "/" + scriptName);
        if(!perlScript.createNewFile()){
            out.println("Script already exists for this data,
unexpected error.<br>");
            newDataFile.delete();
            out.println("<a href=" + thisUrl + ">Previous Page</a>");
            return;
        }
        FileWriter perl = new FileWriter(perlScript);
        perl.write("#!/user/local/bin/perl \n");
        perl.write("use strict; \n");
        perl.write("use DBI; \n");
        perl.write("use DBD::mysql;\n");
        perl.write("use Mail::Sendmail;\n");
        perl.write("my $email = '" + email + "';\n");
        perl.write("my $site_name; \n");
        perl.write("my $contributor; \n");
        perl.write("my $obs_type = \"NULL\"; \n");
        perl.write("my $user_name = \"" + user_name + "\";\n");
        String visible = null;
        if (multi.getParameter("visible") != null &&
(multi.getParameter("visible")).equals("invisible")){
            visible = "no";
        }else{
            visible = "yes";
        }
        String obsType = multi.getParameter("obs_typew");
        if (obsType.length() != 0){
            if(!obsType.equals("GPS") && !obsType.equals("Strain")){
                perl.write("$obs_type = \"" + multi.getParameter(
"obs_typew") + "\";\n");
            }
        }else {
            out.println("You must select a value for the observation
type.  Please go back and reenter that.<br>");
            out.println("<a href=" + thisUrl + ">Previous Page</a>");
            newDataFile.delete();
            perlScript.delete();
            return;
        }
        ⟨Get the value for contributor if submitted. 22⟩
```

Used in part 19.

⟨Get the value for contributor if submitted. 22⟩ ≡

```
        if ((multi.getParameter("contributorw")).length() != 0){
            perl.write("$contributor = \"" + multi.getParameter(
"contributorw") + "\";\n");
            if(valuePositions.containsKey("contributor")){
                out.println("A value for the contributor was entered
and so was a position for it in the datafile.  The contributor can
either be entered or be provided in the datafile, but not both.
Please go back and correct this.<br>");
                newDataFile.delete();
                perlScript.delete();
                out.println("<a href=" + thisUrl + ">Previous Page
</a>");
                return;
            }
        }
```

Used in part 21.

Next the Perl script creates some prepared SQL statements. *$sql* is an insert statement that inserts information into the geodetic table. It is re-declared as a handle called *$sth*. The question marks mark parts of the statement to which values can be bound at a later time. This feature makes it very easy to use the same statement to load all the user data. *$sql_check* is an SQL select and count statement. Before the *$sql* statement is executed, this statement is executed with the same information. If it returns more than 0, meaning this user has already submitted this information, then the *$sth* statement is not executed. The reason for this is because redundant information makes the database harder to browse and analyze. Imagine searching a specific location for subduction zone information and finding 200 entries, most of which are the same. This can happen when an impatient user downloads the same file many times while the first one is still loading, or when a malicious user wants to see if she can run the database out of memory.

If the user is submitting any correlation information then another insert statement is also declared, called *$sql2*, then re-declared as a handle called *$sth2*. This statement inserts information into the geodetic correlation table.

⟨Connect to the database and define the prepared SQL statements. 23⟩ ≡

```
        perl.write("my $dbh = DBI->connect(\"DBI:mysql:database=ITR\",
    \"itr\", \"nottheitrpassword\", {RaiseError =>1, AutoCommit =>0}) ||
    die \"Database connection not made: $DBI::errstr\"; \n");
        perl.write("my $sql = qq{INSERT into ITR.geodetic_data(id,
    obs_type, longi, lat, obs, azimuth, sigma, site_name, contributor,
    user_name, visible) values (NULL, ?, ?, ?, ?, ?, ?, ?, ?, '" +
    user_name + "', '" + visible + "')}; \n");
        perl.write("my $sth = $dbh->prepare($sql); \n");
        perl.write("my $sql_check;\n");
        if (valuePositions.containsKey("corr_ne") ||
    valuePositions.containsKey("corr_nn_ee") ||
    valuePositions.containsKey("corr_ee_ne") ||
    valuePositions.containsKey("corr_eu") ||
    valuePositions.containsKey("corr_nu") ||
    valuePositions.containsKey("corr_nn_ne")){
            perl.write("my $sql2 = qq{INSERT into
    ITR.geodetic_correlation(ref1, ref2, correlation) values (?, ?, ?)};
    \n");
            perl.write("my $sth2 = $dbh->prepare($sql2); \n");
        }
```

Used in part 19.

With the sql statements prepared, the script can begin reading the data file and binding its values to the statements. Each line of the file is read until an EOF is encountered. The line is split into tokens, using the Perl built in function *split* with the deliminator specified by the user, and the tokens are stored in the array *@SqlInput*. The array is then shifted so it starts at index 1 instead of 0, since the user marked the positions in the data file as starting at position 1.

⟨Open the data file and read each line. 25⟩ ≡

```
        perl.write("open (input, \"" + dataDir + "/" + filename + "\")
|| die(\"Could not open file for reading\"); \n ");
        perl.write("my $fileline; \n");
        if (multi.getParameter("delim").length() !=0){
            perl.write("my $delim = '" + multi.getParameter("delim")
+ "'; \n");
            perl.write("if($delim =~ /\\.|\\?|\\*|\\||\\\\|\\+|\\//){
\n");
            perl.write("$delim = \"\\\\$delim\";\n");
            perl.write("}\n");
        }
        if (multi.getParameter("comment").length() != 0){
            perl.write("my $comment = \"" + multi.getParameter(
"comment") + "\"; \n");
        }
        perl.write("my @SqlInput; \n");
        perl.write("my @temp; \n");
        perl.write("my $row_count = 0;\n");
        perl.write("my $current_obstype; \n");
        perl.write("my %current_rowid; \n");
        perl.write("my $long;\n");
        perl.write("my $lat; \n");
        perl.write("my @rows; \n");
        perl.write("eval{ \n");
        perl.write("while($fileline = <input>){ \n");
        perl.write("$fileline =~ s/^\\s\\s*//;\n");
        perl.write("chomp $fileline; \n");
        if (multi.getParameter("comment").length() != 0){
            perl.write("if($fileline =~ /\\d/ && !($fileline =~
/[$comment]/)){ \n");
        }else{
            perl.write("if($fileline =~ /\\d/){ \n");
        }
        if (multi.getParameter("delim").length() !=0){
            perl.write("@SqlInput = split(/\\s*$delim+\\s*/,
$fileline); \n");
        }else{
            perl.write("@SqlInput = split(/\\s+/, $fileline);\n");
        }
        perl.write("unshift @SqlInput, 'Bad';\n");
```

Used in part 19.

Now the tokens can be bound to the *$sth* statement and entered into the database. Since each observation, and its associated sigma value, needs its own entry into the database the binding routine is written for each kind of observation the user submits. In each routine the same values are submitted for *longitude, latitude, azimuth, site name, and contributor*, but a different observation and sigma value is submitted. The design of the program makes sure that the correct sigma value is submitted with each observation, such as obs_n and sig_n, and that the Java Servlet terminates on error if either of these values where not submitted.

⟨Write a routine for each valid observation. 26⟩ ≡

```
                    ⟨Handle each possible observation value. 28⟩

                    ⟨Bind the longitude and latitude. 30⟩

                    ⟨Bind the observation, sigma, and azimuth value. 31⟩

                    ⟨Bind the Site Name and Contributor. 33⟩

                    perl.write("@rows = $dbh->selectrow_array($sql_check);
\n");
                    perl.write("if($rows[0] == 0){\n");
                    perl.write("$sth->execute();\n");
                    perl.write("$row_count++;\n");
                    perl.write("}\n");
                    perl.write("$current_rowid{$current_obstype} =
$sth->{mysql_insertid}; \n");
                } else if ( valuePositions.containsKey(obs[y])
|| valuePositions.containsKey(sigma[y])){
                    out.println("A value was entered for observation but
not for sigma.  .<br>");
                    out.println("<a href=" + thisUrl + ">Previous Page
</a>");
                    return;
                }

            }
```

Used in part 19.

All the observation types require only one row in the geodetic table, except Strain and GPS. Strain and GPS each have 3 subtypes, GPS east, north, and up, and Strain east east, north north, and north east. These subtypes have observation and sigma values that go in the same direction. To capture these relations the four arrays *obs, sigma, GPS,* and *Strain* where created so that they will line up correctly on common indexes. For example $obs[2] = obs\_n, sigma[2] = sig\_n, GPS[2] = GPSn$, which is observation north, sigma north, and GPS north. So on each of the seven iterations of the for loop, inside which values are bound to the *$sth* statement and it is executed, the correct types can be accessed simply by using the control variable, y as an array index.

⟨Handle each possible observation value. 28⟩ ≡

```
        String [] obs = {"obs_", "obs_n", "obs_u", "obs_e", "obs_nn",
"obs_ee", "obs_ne"};
        String [] sigma = {"sig", "sig_n", "sig_u", "sig_e", "sig_nn",
"sig_ee", "sig_ne"};
        String [] GPS = { "BAD", "GPSn", "GPSu", "GPSe"};
        String [] Strain = {"BAD", "BAD", "BAD", "BAD", "StrainNN",
"StrainEE", "StrainNE"};
        for (int y = 0; y < 7; y++){
            if( valuePositions.containsKey(obs[y])
&& valuePositions.containsKey(sigma[y])){
                if (y == 0){
                    perl.write("$sth->bind_param(1, $obs_type,
DBI::SQL_VARCHAR ); \n");
                    perl.write("$sql_check = \"select count(*) from
ITR.geodetic_data where obs_type=\\'$obs_type\\' and \";\n");
                    perl.write("$current_obstype = $obs_type;\n");
                } else {
                    if(obsType.equals("GPS")){
                        perl.write("$sth->bind_param(1, \"" + GPS[y]
+ "\", DBI::SQL_VARCHAR); \n");
                        perl.write("$sql_check = \"select count(*)
from ITR.geodetic_data where obs_type=\\'" + GPS[y] + "\\' and \";\n");
                        perl.write("$current_obstype = \"" + GPS[y]
+ "\";\n");
                    } else if (obsType.equals("Strain")){
                        perl.write("$sth->bind_param(1, \"" + Strain[y]
+ "\", DBI::SQL_VARCHAR); \n");
                        perl.write("$sql_check = \"select count(*)
from ITR.geodetic_data where obs_type=\\'" + Strain[y] + "\\'
and \";\n");
                        perl.write("$current_obstype = " + Strain[y]
+ ";\n");
                    } else {
                        out.println("Internal Error.<br>");
                        newDataFile.delete();
                        perlScript.delete();
                        out.println("<a href=" + thisUrl
+ ">Previous Page</a>");
                        return;
                    }
                }
```

Used in part 26.

The fields, longitude and latitude, are required fields. If they are not provided the Java function informs the user that they are required, and returns. If the values read by the Perl script are not numbers, or do not follow the bounds $-180.00 \leq longitude \leq 360.00$ or $-90.00 \leq latitude \leq 90.00$ respectively, then the script throws an error that is caught and sent to the user via email. Longitude is stored between -180.00 and 180.00 in the database, but many geoscientists request it as a value between 0 and 360.00. Fortunately it is easy to convert these units, if $180.00 < longitude < 360.00$ then 360.00 is subtracted from longitude to convert it to the correct value in the database units.

⟨Bind the longitude and latitude. 30⟩ ≡

```
                if(valuePositions.containsKey("longi")
    && valuePositions.containsKey("lat")){
                    perl.write("$long = $SqlInput["
    + valuePositions.get("longi") + "];\n");
                    perl.write("if ($long =~
    /^(-?\\d+\\.?\\d*|-?\\.\\d+|-?\\d+\\.?\\d+e(-?|\\+?)\\d+)$/
    && $long > -181.0 && $long < 361.0){\n");
                    perl.write("if ($long > 180.0){\n");
                    perl.write("$long = $long - 360.0;\n");
                    perl.write("}\n");
                    perl.write("$sth->bind_param(2, $long,
    DBI::SQL_FLOAT); \n");
                    perl.write("$sql_check .= \"longi > $long - 0.001
    and longi < $long + 0.001 and \";\n");
                    perl.write("}else{\n");
                    perl.write("die \"An Invalid value was found for
    longitude: $SqlInput[" + valuePositions.get("longi") + "]\";\n");
                    perl.write("}\n");
                    perl.write("if ($SqlInput["
    + valuePositions.get("lat") + "] =~
    /^(-?\\d+\\.?\\d*|-?\\.\\d+|-?\\d+\\.?\\d+e(-?|\\+?)\\d+)$/
    && $SqlInput[" + valuePositions.get("lat") + "] > -91.0
    && $SqlInput[" + valuePositions.get("lat") + "] < 91.0){\n");
                    perl.write("$sth->bind_param(3, $SqlInput["
    + valuePositions.get("lat") + "], DBI::SQL_FLOAT); \n");
                    perl.write("$sql_check .= \"lat > $SqlInput["
    + valuePositions.get("lat") + "] - 0.001
    and lat < $SqlInput[" + valuePositions.get("lat") + "] + 0.001
    and \";\n");
                    perl.write("}else{\n");
                    perl.write("die \"An Invalid value was found for
    latitude: $SqlInput[" + valuePositions.get("lat") + "]\";\n");
                    perl.write("}\n");
                } else {
                    out.println("There was no entry for either
    longitude, latitude, or both, you must submit both of these values.
    <br>");
                    out.println("<a href=" + thisUrl + ">Previous Page
    </a>");
                    return;
                }
```

Used in part 26.

Next the script binds the observation, sigma, and azimuth values. If either the observation or sigma values are not numbers, or azimuth has a value and it is not a number, then the script throws an error that is caught and sent to the user via email. If a value for azimuth was not submitted then the value *undef* is bound to the *$sth* statement, which shows up as null in the database.

⟨Bind the observation, sigma, and azimuth value. 31⟩ ≡

```
                perl.write("if ($SqlInput["
+ valuePositions.get(obs[y]) + "] =~
/^(-?\\d+\\.?\\d*|-?\\.\\d+|-?\\d+\\.?\\d+e(-?|\\+?)\\d+)$/){\n");
                perl.write("$sth->bind_param(4, $SqlInput["
+ valuePositions.get(obs[y]) + "], DBI::SQL_FLOAT); \n");
                perl.write("$sql_check .= \"obs < $SqlInput["
+ valuePositions.get(obs[y]) + "] + 0.001 and obs > $SqlInput["
+ valuePositions.get(obs[y]) + "] - 0.001 and \";\n");
                perl.write("}else{\n");
                perl.write("die \"An Invalid value was found for
observation: $SqlInput[" + valuePositions.get(obs[y]) + "]\";\n");
                perl.write("}\n");
                ⟨Bind azimuth 32⟩
                perl.write("if ($SqlInput["
+ valuePositions.get(sigma[y]) + "] =~
/^(-?\\d+\\.?\\d*|-?\\.\\d+|-?\\d+\\.?\\d+e(-?|\\+?)\\d+)$/){\n");
                perl.write("$sth->bind_param(6, $SqlInput["
+ valuePositions.get(sigma[y]) + "], DBI::SQL_FLOAT); \n");
                perl.write("$sql_check .= \"sigma > $SqlInput["
+ valuePositions.get(sigma[y]) + "] - 0.001 and sigma < $SqlInput["
+ valuePositions.get(sigma[y]) + "] + 0.001 and \";\n");
                perl.write("}else{\n");
                perl.write("die \"An Invalid value was found for sigma:
$SqlInput[" + valuePositions.get(sigma[y]) + "]\";\n");
                perl.write("}\n");
```

Used in part 26.

⟨Bind azimuth 32⟩ ≡

```
                if (valuePositions.containsKey("azimuth")){
                    perl.write("if($SqlInput["
    + valuePositions.get("azimuth") + "] =~
    /^(\\d+\\.?\\d*|\\.\\d+|-?\\d+\\.?\\d+e(-?|\\+?)\\d+)$/){\n");
                    perl.write("$sth->bind_param(5, $SqlInput["
    + valuePositions.get("azimuth") + "], DBI::SQL_VARCHAR); \n");
                    perl.write("$sql_check .=
    \"azimuth like \\'$SqlInput[" + valuePositions.get("azimuth")
    + "]\\' and \";\n");
                    perl.write("}else{\n");
                    perl.write("die \"An Invalid value was found for
    Azimuth: $SqlInput[" + valuePositions.get("azimuth") + "]\";\n");
                    perl.write("}\n");
                }else{
                    perl.write("$sth->bind_param(5, undef,
    DBI::SQL_VARCHAR);\n");
                    perl.write("$sql_check .= \"azimuth is null
    and \";\n");
                }
```

Used in part 31.

The site name and the contributor are both optional values. If they are not submitted then a value of *undef* is bound to the *$sth* statement. The value of site name must consist of four letters or numbers. The value of contributor must not contain any quotation marks, since these tend to cause problems with the MySql database and with the Perl database drivers.

⟨Bind the Site Name and Contributor. 33⟩ ≡

```
                if(valuePositions.containsKey("site_name")){
                    perl.write("if($SqlInput["
+ valuePositions.get("site_name") + "] =~ /^(\\w{4})$/){\n");
                    perl.write("$sth->bind_param(7, $SqlInput["
+ valuePositions.get("site_name") + "], DBI::SQL_VARCHAR); \n");
                    perl.write("$sql_check .=
\"site_name=\\'$SqlInput[" + valuePositions.get("site_name") + "]\\'
and \";\n");
                    perl.write("}else{\n");
                    perl.write("die \"An Invalid value was found for
the Site Name: $SqlInput[" + valuePositions.get("site_name") + "]\";
\n");
                    perl.write("}\n");
                } else {
                    perl.write("if(!(defined $site_name)){\n");
                    perl.write("$sth->bind_param(7, $site_name,
DBI::SQL_VARCHAR); \n");
                    perl.write("$sql_check .= \"site_name is null
and \";\n");
                    perl.write("}elsif ($site_name =~ /^(\\w{4})$/){
\n");
                    perl.write("$sth->bind_param(7, $site_name,
DBI::SQL_VARCHAR); \n");
                    perl.write("$sql_check .= \"site_name=
\\'$site_name\\' and \";\n");
                    perl.write("}else{\n");
                    perl.write("die \"An Invalid value was found for
the Site Name: $site_name;\n");
                    perl.write("}\n");
                }
                    ⟨Bind contributor. 34⟩
```

Used in part 26.

⟨Bind contributor. 34⟩ ≡

```
        if(valuePositions.containsKey("contributor")){
                    perl.write("if($SqlInput["
+ valuePositions.get("contributor") + "] =~ /^(.*'+.*|.*\"+.*)$/){\n");
                    perl.write("die \"Quotation marks cannot appear in
the contributor name: $SqlInput[" + valuePositions.get("contributor")
+ "]\";\n");
                    perl.write("}else{\n");
                    perl.write("$sth->bind_param(8, $SqlInput["
+ valuePositions.get("contributor") + "], DBI::SQL_VARCHAR); \n");
                    perl.write("$sql_check .= \"contributor=
\\'$SqlInput[" + valuePositions.get("contributor") + "]\\'\";\n");
                    perl.write("}\n");
                } else {
                    perl.write("if(defined $contributor
&& $contributor =~ /^(.*'+.*|.*\"+.*)$/){\n");
                    perl.write("die \"Quotation marks cannot appear in
the contributor name: $contributor\";\n");
                    perl.write("}\n");
                    perl.write("$sth->bind_param(8, $contributor,
DBI::SQL_VARCHAR ); \n");
                    perl.write("if( !(defined $contributor)){\n");
                    perl.write("$sql_check .= \"contributor is null\";
\n");
                    perl.write("}else{\n");
                    perl.write("$sql_check .= \"contributor=
\\'$contributor\\'\";\n");
                    perl.write("}\n");
     }
```

Used in part 33.

After a line of the data file has been entered into the geodetic table, any correlation values between those lines can be entered into the geodetic correlation table. This is done by taking every correct combination of correlation, observation, sigma, and observation type fields and submitting them to the Java function *enterCorrelation*. If the user submitted any of the combinations then a Perl routine to submit that correlation value is written. If a user submits a correlation value, but not all of its corresponding values, then the Java Servlet reports the error and exits. When the Perl script is submitting the correlation value it checks that the correlation value is a number.

⟨Enter correlation values into the geodetic correlation table. 35⟩ ≡

```
        if (!enterCorrelation("corr_nn_ne", "obs_nn", "obs_ne",
"StrainNN", "StrainNE", thisUrl, valuePositions, perl, out) ||
            !enterCorrelation("corr_ee_ne", "obs_ee", "obs_ne",
"StrainEE", "StrainNE", thisUrl, valuePositions, perl, out) ||
            !enterCorrelation("corr_nn_ee", "obs_nn", "obs_ee",
"StrainNN", "StrainEE", thisUrl, valuePositions, perl, out) ||
            !enterCorrelation("corr_ne", "obs_n", "obs_e", "GPSn",
"GPSe", thisUrl, valuePositions, perl, out) ||
            !enterCorrelation("corr_eu", "obs_u", "obs_e", "GPSu",
"GPSe", thisUrl, valuePositions, perl, out) ||
            !enterCorrelation("corr_nu", "obs_u", "obs_n", "GPSu",
"GPSn", thisUrl, valuePositions, perl, out)){
            return;
        }

        perl.write("undef %current_rowid;\n");
        perl.write("} \n");
        perl.write("} \n");
```

Used in part 19.

"enterCorrelation" 36 ≡

```
        private boolean enterCorrelation(String corr, String obs1, String obs2,
            String obs_type1, String obs_type2, String thisUrl,
            Map valuePositions, FileWriter perl, PrintWriter out){
            if (valuePositions.containsKey(corr)){
                if(valuePositions.containsKey(obs1)
    && valuePositions.containsKey(obs2)){
                    try {
                        perl.write("if ($current_rowid{" + obs_type1 + "}
    && $current_rowid{" + obs_type2 + "}){ \n");
                        perl.write("$sth2->bind_param(1,
    $current_rowid{" + obs_type1 + "});\n");
                        perl.write("$sth2->bind_param(2,
    $current_rowid{" + obs_type2 + "});\n");
                        perl.write("if ($SqlInput[" + valuePositions.get(corr)
    + "] =~ /^(-?\\d+\\.?\\d*|-?\\.\\d*|-?\\d+\\.?\\d+e(-?|\\+?)\\d+)$/){\n");
                        perl.write("$sth2->bind_param(3, $SqlInput["
    + valuePositions.get(corr) + "], DBI::SQL_FLOAT);\n");
                        perl.write("$sth2->execute();\n");
                        perl.write("}else{\n");
                        perl.write("die \"An Invalid value was found for the
    correlation: $SqlInput[" + valuePositions.get(corr) + "]\";\n");
                        perl.write("}\n");
                        perl.write("}\n");
                    } catch (IOException e){
                        out.println("Error writing to file.<br>");
                        out.println("<a href=" + thisUrl + ">Previous Page</a>");
                        return false;
                    }
                } else {
                    out.println("A value was entered for a correlation, but no
    value was entered for either one or both of its corresponding observations.
    .<br>");
                    out.println("<a href=" + thisUrl + ">Previous Page</a>");
                    return false;
                }
            }
            return true;
        }
    }
```

If the data file is downloaded correctly, then the changes are committed and the user is sent an email stating that the file was loaded and how many rows where loaded.

⟨If script executed email the user with success. 37a⟩ ≡

```
        perl.write("$dbh->commit(); \n");
        perl.write("$sth->finish(); \n");
        perl.write("$dbh->disconnect(); \n");
        perl.write("my %mail=(To => $email, From => 'foutzl@rpi.edu',
Subject => 'ITR/AP(GEO) submission confirmation',
Message =>\"Your file was successfully loaded into the database and
$row_count rows where added to the database (if 0 rows where added it
is possible you entered a file with corrupted data or a file that has
already been loaded into the database).  You can now use ITR/AP(GEO)
web pages to access and delete your data by logging in under the same
name under which you submitted this file.  Thank you for contributing
to the geodetic database.\");\n");
        perl.write("sendmail(%mail) or die $Mail::Sendmail::error;\n");
        perl.write("exit 1;\n");
        perl.write("}; \n");
```

Used in part 19.

If the Perl script throws an error during the loading process then this part of the program catches it. This section of the program rolls back all changes and emails the user with the error. All changes to the database are undone, because if some of the user's information is corrupt, then there is a good chance that all of it is corrupt, even if the consistency checks where not able to catch them earlier.

⟨If an error is thrown email the user with the error. 37b⟩ ≡

```
        perl.write("$dbh->rollback(); \n");
        perl.write("$dbh->disconnect();\n");
        perl.write("my %mail=(To => $email, From => 'foutzl@rpi.edu',
Subject => 'ITR/AP(GEO) submission confirmation', Message =>\"An error
has occurred while trying to load your file into the database.  No rows
where added to the database because of this error.  The Error was
reported as: $@.  Please try to submit again, making sure that your
information is formatted as you described it on the submit page and as
specified in the Submit section of the Help pages.  If you continue to
have problems please contact the system administrator at
foutzl\\@rpi.edu\");\n");
        perl.write("sendmail(%mail) or die $Mail::Sendmail::error;\n");

        perl.flush();
        perl.close();
```

Used in part 19.

Now that the Java Servlet has finished creating the Perl script it can run it using the Object Runtime. This runs the Perl interpreter with the Perl script and some flags as arguments. If the Servlet is able to start the interpreter then it prints out a success message to the user and finishes execution.

⟨Run the Perl script using the Runtime Java class. 38⟩ ≡

```
        String [] perlCommand = {"perl", perlFlags,
perlScript.toString()};
        try{
            Runtime executeScript = Runtime.getRuntime();
            Process process = executeScript.exec(perlCommand);
        }catch (Exception e){
            out.println("Error executing script to load your file into
the database: " + e + "<br>");
            out.println("<a href=" + thisUrl + ">Previous Page</a>");
            return;

        }

        out.println("Your file was successfully saved, you will receive
an email within a few minutes stating whether or not it was
successfully loaded into the database.<br>");
```

Used in part 13.

6 Deleting and Updating Information

The ability for users to delete and update their own entries is important to data integrity. It allows users to alter or remove entries from the database that they have learned to be incorrect. This functionality does pose the problems of insuring changes do not affect the data integrity and making sure that users do not alter information they did not submit.

To update or delete an entry the user must first log in. If the user is not logged in she will be denied access to the update or delete page and asked to log in. After that, she is taken to a page that looks identical to the html form search page. Here she can search for the entries she wishes to alter. The search is done in the same way it is done on the search page, except an extra qualifier is added to the where clause that checks to see if the username of a row matches the user's username. Then the user can select the rows she wishes to delete or update from the rows returned from the search. She does this by selecting the check boxes next to the desired rows and hitting a submit button. What happens next depends on whether the user is deleting or updating the selected rows.

If the user is deleting rows then the Java Servlet deletes all rows in the geodetic table with key values that match those selected from the search results. Any row in the geodetic_correlation table in which one of the foreign keys matches a

selected key is also deleted. Each delete statement checks the username of the row it is deleting against the username of the current user.

If the user is updating rows then the Java Servlet retrieves all the rows the user selected and displays them with their values in either text boxes or selection menus, depending on which is appropriate. Some values, like the *id* and *username* remain hidden from the user and cannot be altered. The user can alter the values then hit the submit button. Then an update statement is executed for each row the user submitted, with all the updatable values being updated whether the new value is different from the old one or not. It is just simpler to implement that way. Since the user is submitting new values to the database they have to be checked for consistency before they can be allowed to be updated. All values are checked against the same regular expressions and boundary conditions that where used when they where first submitted to the database. Unfortunately, it is impossible to properly check if alterations to the observation type are correct, because altering it can affect other rows in this table and other tables in the database, so the user is not allowed to alter that field.

7 Further Development

As it is now, the geodetic database is easy to access, but well guarded against misuse. However there are still some improvements that can be done.

One major improvement would be to implement automatic unit conversion for observation, sigma, and azimuth values entered by users. At the moment the database requires one type of unit for each of these values, which is very inconvenient for geoscientists who store their data in different units. Automatic conversion could be done by allowing the user to select from a list of unit types, and then converting the value based on the ratio between the submitted unit and the units used by the database. This solution unfortunately weakens the data integrity of the database interface. Unless hard boundary conditions exist for the field being converted, the user could select the wrong unit type resulting in a converted value that is not necessarily invalid, but any person browsing the database could guess that the value is a mistake. Unfortunately no hard boundary conditions exist for any of these three fields.

The Cookie passed between the user and the project server also needs to be more secure. As currently implemented anyone can easily intercept the Cookie, read its value, and masquerade as the person the Cookie belongs too. Also a hacker could spam the project server with Cookies with made up values until she gets one that corresponds to a user. Using the methods described in [6] would solve both of these problems.

The project is still underdevelopment, but a working version should be available by the Fall of 2004. The meshing and Green's function generation section of the project, based on the prototype described in [2], has been implemented along with the database section. The inversion section is currently being worked on and when it is completed the project will be ready for public use. Additional

functionality, such as 3D image generation of the subduction zone model, will be added to the project in the future.

References

[1] Bishop, Matt. Computer Security. Boston: Addison-Wesley, 2003.

[2] Han, Jing "ITR/AP(GEO) Interactive Web-Based Tools for Modeling Deformation at Subduction Zones." Technical report, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY., 2002.

[3] Hunter, Jason. Java Servlet Programming. Cambridge:O'Reilly, 2001.

[4] Java Servlet Technology. Sun Microsystems. Inc 8 Jan. 2004. <http://java.sun.com/products/servlet/index.jsp>.

[5] McGraw-Hill Dictionary of Scientific and Technical Terms. Lapedes, Daniel N.; McGraw-Hill Book Company: New York, 1978, pp 672-673.

[6] Park, Joon S., Sandhu, Ravi, and Ghanta, SreeLatha "RBAC on the Web by secure cookies." In Proceedings of the IFIP WG11.3 Workshop on Database Security. Chapman & Hall, July 1999.

[7] Williams, Charles and McCaffrey, Robert, "Stress rates in the central Cascadia subduction zone inferred from an elastic plate model, Geophys. Res. Lett., in revision, 2001.

[8] Williams, Charles, Ph.D., Research Associate Computational Geophysics, Rensselaer Polytechnic Institute Department of Earth & Environmental Sciences. Personal Interview. Troy, NY, 1 Apr. 2003.