Lehrstuhl für Systeme und Informationsverwaltung

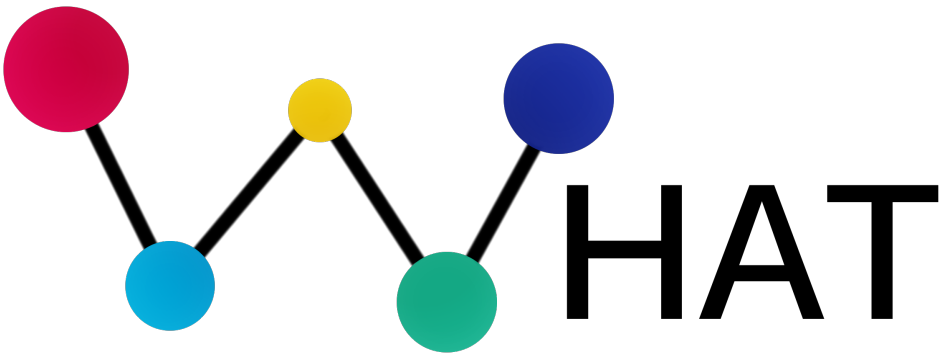
Prof.Dr.Ing. Klemens Böhm
Hoang Vu Nguyen
Marco Neumann

Karlsruher Institut für Technologie

Praxis der Softwareentwicklung (PSE)
WS 2012/2013

Visualizing and Statistically Analyzing Access Behavior to Scientific Databases

Implementation



WareHouse Analyzing Tool

January 25, 2013

Functional Specification **Alexander Noe**
Design **Jonathan Klawitter**
Implementation **Anas Saber**
QA / Testing **Nikolaos Alexandros Kurt Moraitakis**
Final **Lukas Ehnle**

E-M@il: pse10-group14-ws12@ira.uni-karlsruhe.de

Contents

1 Changes in web tier

The WebTier provides the graphical user interface and front end of our application. On the client side it relies heavily on Javascript and the jQuery library. As I haven't done much programming aside of the tasks given at university, which were solely in java and lately some C, programming in Javascript was a very enlightening experience.

1.1 Web page

As for the index page, it has not changed very much on the surface since the prototype layout. There is now an admin login and you can actually change the languages. The play framework made some things, like form validation really easy and allowed us to program our application logic in java. But as I had to learn the difference between scala and java is not as little as one may assume. And because Play is mainly written in Scala it has only a rudimentary java documentation. That may be a problem, if you want to access some advanced functions. Because the specification of the web page was very vague, as we didn't know to which extent the front end would be programmed in java or just plain html/javascript, I can't really state much that has changed since the specification.

1.2 WebpageControllers

This is the class Website: It provides a method for every valid url of the application, which is invoked when a user sends a request. There are many more additonal methods: for login and form validation, chart request handling and so on. the changeLanguage method now changes the language itself instead of calling the Localize class. It still renders the HTML views. 'Render' because play provides the possibility to write scala code directly in this views. These scala statements are evaluated at runtime when the page in question is requested. Play can also serve static files. For that no involvement of a user defined controller is needed.

1.3 Localize

This class is still handling the localization of the webpage. in addition it localizes, other parts of the application, e.g. status and error messages from the parser the get() method was designed for the webpage. If a localized String can't be found, there is an automatic fallback to a predefined standard language, so that still some information is displayed. In the application itself that sometimes was not wanted behaviour, therefore another method has been added getString, which can localize the same key words, but without the fallback mechanism. The changeLanguage method has been removed from this class, as the Webpage controller can handle it itself.

1.4 ChartIndex

This class still scans the charts directory for all available charts at instanciation. It is implemeted with the singleton pattern, to prevent unnecessary cpu and I/O time. It's methods are still the same.

1.5 what

Some helper classes have been put in the what package although they are utilized by the WebpageController. AdminLogin and LogfileUpload are two helper classes which facilitate form validation through usage of functionality provided by Play. AdminAuth helps in determining whether a user is authenticated as admin or not and controlling the behaviour of the secured section. ChartHelper uses private methods to create chart pages dynamically and provides some caching mechanism of these chart pages. The only public method is a static getOptions(String) which handles the singleton pattern for ChartHelper and returns a Html contentType with all options which can be chosen to request the given chart.

2 Changes in and comments to the Application tier

2.1 Working with a config

At the beginning of the implementation phase, we decided to make our application run very dynamically. To ensure this, we wanted to make nearly every work be based on a given configuration file.

This configuration file, written as a JSON file, should contain all the information needed about the database for which the application shall run. For example it should contain the data bases name, like “Skyserver”, all the dimensions of the data warehouse schema for it and all the measures. Also it tells the parser, how to parse the lines in the log files.

Starting the application a *Facade* instance will be constructed and initialized with the configuration file. This instance will have mediators, *ParserMediator* for the parsing taks, *ChartMediator* for chart requests and *DataMediator* for things concerning the data access tier, all with a reference to the configuration. So the methods in the 3 main section all can make their work depending on the configuration.

So how does a configuration look like?

2.2 The configuration package

2.2.1 ConfigWrap

The class *ConfigWrap* is the most important class concerning the config. It is the class which wraps all the information taken from the configuratin file. This contains a orderd array of *DimRow*s and *RowEntry*s.

It provides mostly getters for all the information, like number of dimensions, a dimension or row at a certain index and so on.

2.2.2 DimRow

A DimRow stores the information whether at this point is a dimension or just a normal row (or measure). For dimensions it provides methods to get information about the number of rows, a row at a index, name of the table in the warehouse for this information, name of a row in the warehouse, a tree of strings for the web page to display in the selection boxes.

2.2.3 RowEntry

A RowEntry stores the information of on part in the parsing file. The subclasses of RowEntry tell the parser, whether is a int, a String or something specific like location to parse. Therefore they provide strategy methods.

2.2.4 Addition note

Why is everywhere written row, when it is a column in the warehouse? Yeah in the warehouse it is a column, but in the schema of the warehouse it is a row...

2.3 Chart creation

The visitor patter for the charts got dropped. Because facing some problems with *ResultSet* the decision was made to change the received data nearly to the warehouse into a JSONObject and than add additional information later some way more up.

The design of three diffrent types of charts, with one, two or three dimensions was detected to be stupid. There are no charts with 3 dimensions (on our web page). They either have one dimension and a measure or two dimensions and a measure.
So *DimChart* is now the parent class storing all information every chart needs, like the first dimension and the measure. The child class *TwoDimChart* extends this with a second dimension.

2.3.1 ChartMediator

The *ChartMediator* is the mediator of a chart process. He creates a chart wrap (*DimChart*) with the helper class *ChartHelper* and mainly it delegates the requests against the data tier.

2.4 Parser

2.4.1 ParserMediator

`threadPool` : Added a variable poolsize and a method `setPoolsize()` to set it to another number. If the poolsize is smaller than 1 there will be an error displayed.

`readyQueue` , `addToDatabase()` : Deleted, because the ParsingTask sends the finished dataEntry-Object directly to the DataTier.

`entryBuffer` , `extractLine()` : Deleted, because the ParsingTask requests a new line as soon as it needs it.

`configDB` : Renamed to ConfigWrap.

`parseLogFile(String str)` : Deleted configDB, because ParsingTask knows about the ParserMediator and can request it by itself.

`error(String str), boolean fatalError` : Added a new error() method, which sets fatalError to true, which will make parseLogFile(String str) return false to the Facade.

`increaseFT(), increaseLinedel()` : Added new methods to count finished threads and deleted lines.

`createThreadPool()` : Creates a new threadPool with *poolsize* threads and starts it.

`ParserMediator(ConfigWrap cw)` : Creates the parserMediator.

2.4.2 LogFile

`readLine()` : readLine() had bigger problems than expected, because the SQL-statement may contain several endOfLines. it's now fixed by setting a mark, reading another line, looking if it is actually a new line and if it is, resetting the mark and returning the complete line with the whole statement.

`Logfile(String path, ParserMediator pm)` : Checks if path is actually a valid csv-file and calls the Verification tool to check the correctness of the formatting in the file.

2.4.3 Task

Deleted, because there are only ParsingTasks, no LoadingTasks and so there is no need for a superclass.

2.4.4 LoadingTask

Deleted, because the parsingTask sends the DataEntry directly to the Data Tier making the LoadingTask obsolete.

2.4.5 ParsingTask

`ParsingTask(ParserMediator pm)` : Creates a new ParsingTask.

`splitStr` : Added the splitted string to the attributes of the ParsingTask so that it doesn't have to be splitted by every tool.

2.4.6 DataEntry

The DataEntry was completely remade to improve the general usability and multifunctionality. It now has only an Object-Array of variable size which is filled due to the Config.

2.4.7 VerifyTool

Renamed to VerificationTool.

Completely remade the Verification tool, because the verify-part is already done by the splitting tool. Instead it checks, if the file is actually in the format from the config.

2.4.8 LocationTool

Renamed to GeoIpTool

setUpIpTool(ParserMediator pm) : Sets the GeoIPTool up and checks if the geoLiteCity.dat - file which is needed for this tool is at the correct position.

2.4.9 SplittingTool

Splitted `split(ParsingTask pt)` in 3 parts to create smaller methods.

3 Changes in Data and Data access tier

The data and the data access tier provided a lot of trouble for us.

When the decision was made to go with Oracle one thing was overlooked. The OLAP drivers aren't free to get without a license.They also weren't compatible with the other open source technologies used in our program. So we had to change from Oracle to something else. The first idea to use Mondrian failed, as we weren't able to set it up.

So to get the program running we decided to just run a MySQL database and run a warehouse on it. Facing problems to connect to a local MySQL server, creating the table to fullfill our needs, writting the right queries to load data in the warehouse everything got slowed down.

Still facing problems to write the complex queries for chart requests with filters the first version will just be able to get 1 dimensional charts without filters just grouped by a requested thing.

3.1 Data access classes

There is a *DataMediator* of course which delegates the work and requests.

3.1.1 MySQLAdapter

The class *MySQLAdapter* provides methods for any needed query against the warehouse, whether extracting or loading.

When extracting it uses the help of *DataChanger* to interpret the results received from the warehouse.

3.1.2 WHConnectionManager

This class provides methods to get a connection from a connection pool to the warehouse (mySQL data base).

4 Things that went wrong

A biased and last minute perspective on the problems we faced and the things we learned during the implementation phase

Since everybody else is busy bringing the program to run, I wrote this. The things we learned and the problems we faced can be grouped into technical, and not technical. I will mainly focus on the non technical, because it turns out they were actually more important in the end. This is shocking, and I did not expect this when I chose to study computer science.

4.1 Technical

This section mentions some common technical issues we faced. They are further elaborated in the appropriate sections above.

4.1.1 git

We learned to use git, more or less, although it still ocassionaly annoys us. It is not unusual that someone still complains in the internal skype chat once a day.

4.1.2 libraries

Since we split the areas of responsibility quite early on, everyone more or less used a different library or framework and had to learn different things, among them minor quirks and annoyances. Most are not really worth mentioning.

4.2 Non technical

4.2.1 automation

We lost so much time trying to get software to install. (We still didn't manage to install some of it, hello, mondrian!) Automated dependency management and installation sounds like a dream. The building process, however, and the configuration of gradle took us a significant amount of time. It probably gets better with experience with the tools and more time spent using them, but for us it was hard, as it often was the first time we used them.

4.2.2 Broken builds

Don't you hate it when people commit code to the central repository that doesn't compile? We did that too. I think we treated the central repository as some sort of magical backup device. I don't think we do it anymore.

4.2.3 Splitting the program into parts

This deserves a mention. When your code relies on other people's code to run, their code relies on other people's code to run and so on and so forth for some recursion depth, and their code hasn't been written yet, then, oops, writing and testing your code becomes a lot harder. What do you do? Do you wait for the other people to write their code first? Do you commit it to the repository without, ehm, compiling? (The answer probaby involves mock objects and junit, but we didn't really use that yet. Hopefully the next phase will enlighten us.)

Our program had a tiered architecture, and this especially applied to us. We didn't run it from start to finish until today, 25/01/2013, despite many parts working on their own before.

4.2.4 Licenses

We didn't quite know much about the details and the limitations of the different open source licenses, as well as their compatibility with closed source software like oracle's. It turns out the formal, legal language the software licenses are written in does not make fun bedtime reading. This hurt us, as mentioned avobe.

4.2.5 Dealing with change

Admittedly, we probably were a bit optimistic and started a bit late. However, it turns out planning perfectly is hard (yes, we were also bad at it). Here is an example.

Can you always rely on everything going according to schedule? What happens if unexpected events still happen? We even had a tight specification, no indecisive clients that might change their minds or lose interest (or even many potential clients whose intentions we would have had to guess indirectly through market surveys). Is the waterfall model partially to blame?

In any case, we had to deal with unexpected changes.They are further explained in another section above.

4.2.6 What if only one person understands some code and something happens to them?

Thankfully, this didn't happen to us but was a disaster waiting to happen. We were but a crisis away.

4.2.7 Crises

What happens at unexpected negative events in a group project that is, well, more or less egalitarian and doesn't have a person in charge? What are possible reactions of team members? Do they help where needed, or do they react negatively?

4.2.8 Communication

Lastly, it turns out is really important. Better communication and understanding between us would have meant that we wouldn’t have had to rewrite some things, spend less time arguing, explaining and reexplaining things and more time actually programming.