UNIVERSITY OF TROMSØ UIT

FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

A scalable, interactive widget library for visualizing biological data

Terje André Johansen

INF-3981
Master's Thesis in Computer Science
June, 2011

ABSTRACT

In biology the introduction next generation technology is increasing the amount of data generated rapidly. New sequencing machines are able to produce terabytes of genomic data in days and in later years the cost of storing data has become higher than to produce it. With enormous amounts of data arrives great opportunities, but also new challenges; how should biologists analyze and interpret the results? Going through terabytes of data manually is time consuming, and is in reality not practical. Because of this bioinformatics are working together with computer scientists to create programs that can parse, integrate, analyze and visualize data in ways that can aid the biologist to extract novel biological knowledge from it.

Most data exploration programs that have been created to help biologists are desktop applications. Desktop application has been having the advantage that it can benefit from a high performance and hardware support. However desktop applications require the user install a program and download the data to her local computer. If the program or data is updated, they must often be updated manually.

The web is evolving rapidly too, and we believe that we are close to an era where the distinctions between desktop applications and web applications will be relaxed. Web applications also solve of the problems desktop applications have. They do not require the user to manually download a specific program, they can download fresh data from a common repository and they can be accessed from most places on the planet. Further the applications can be updated without the user even noticing it, and new methods for creating visualizations can be added.

This thesis presents the design, implementation and evaluation of *a scalable, interactive widget library for visualizing biological data*. We investigate if JavaScript enabled web browsers are capable of creating visualizations that are comparable with what we could create using a desktop application.

We have fully implemented a widget able to visualize a heat map. This widget implements two ways for creating a heat map; the first one uses a traditional approach using styled hypertext markup language and the second uses the new HTML5 canvas.

We have conducted experiments with both of these approaches and found that it is possible to create scalable heat map visualizations with good performance. Previous versions of the same application were able to visualize a maximum of 50.000 data cells; the new implementation has been able to visualize 16.800.000 cells at which point we had shown almost every data cell we had available. This is enabled by using a pagination technique in which only partial amount of data is visualized in each page.

We have also evaluated the maximum number of cells we could have in a single page to be approximately 40.000 data cells. This creates a view consisting of 30.000x9.000 pixels, which is 117 times the size of a 1920x1200 pixels display.

The evaluation also explores which of the two approaches works best creating a heat map, and we found that the canvas performs better in almost all the experiments we have conducted.

After the work in this project we believe that it is possible to use web applications for visualizing many types of biological data. If the data can be split into smaller portions, like the heat map, we believe that browsers and JavaScript has become fast enough to handle generation of visualizations using the canvas.

We also believe that it is a good idea to porting applications as early as possible. Even though the HTML5 standard is not expected to be finished before 2020, much of its functionality is already implemented into browsers. Waiting for completion will only delay what could possibly be a fruitful online ecosystem for genomic research.

ACKNOWLEDGEMENTS

I would like to thank my supervisors here at the University of Tromsø, Lars Ailo Bongo and Daniel Stødle. Lars has challenged and motivated me through many interesting discussions. He also provided me with this thesis and has given me some insight into the biological aspects of my application. Daniel has given me several helpful tips on coding. They have both helped me create a much better report than I would have been able to alone.

Thanks to Olga Troyanskaya and Qian Zhu from Princeton University. They have given me kind and helpful comments regarding functionality and the usefulness of the heat map widget. Also thanks to Wei Dong for giving me a user accounts, and helping me set up the web server.

Thanks to my brother, Knut Kristian Johansen, for giving me web application tips and making me realize that this report is a very important aspect of my thesis. I must admit though, I prefer to create code and not books.

Last I would like to thank my fellow students Øyvind Holmstad, Erik Bræk Leer and Arild Nilsen for motivating me. By just being at the university from morning till late night you made me worker harder than I would have done alone.

Contents

Abstract....3
Acknowledgements....5
Contents....7
List of Figures....9
List of Abbreviations....11
1. Introduction....13
2. Background....17
  2.1 Heat maps....17
  2.2 Dendrograms....18
  2.3 Motivations for this project....19
  2.4 How browsers generate a page....21
  2.5 Document object model....22
  2.6 Google Web Toolkit....23
  2.7 Jquery....24
  2.8 Evaluating web applications....25
3. Architecture....27
  3.1 Widget library....27
  3.3 Model-view-presenter pattern....28
  3.2 Heat map widget architecture....29
    3.2.1 Heatmap widget....30
  3.3 Dendrogram widget architecture....34
    3.3.1 Dendrogram widget....35
  3.5 Several views on the same page....37
  3.6 Interaction handeling....38
4. Implementation....41
  4.1. Model....42
  4.2 Presenter....44
    4.2.1 Identifiers....44
    4.2.2 Data....45
    4.2.3 Adding data to the providers....45
    4.2.4 Client data storage....46
  4.3 Heat map pager....48
  4.4 Views....49

4.4.1 Creating the colors 50
4.4.2 Table view 51
4.4.3 Canvas view 53
4.4.4 Headers 55
4.5 Dendrogram 56
4.5.1 Data structures 56
4.5.2 Server 56
4.5.3 Client 57
4.5.4 Making it work 58
5. Widget funtionality 59
5.1 A new search 59
5.2 A new page 60
5.3 Adding columns 61
5.4 Selecting genes and datasets 62
5.5 Detailed information 63
6. Evaluation 65
6.1 Metholodgy 65
6.2 Scaleability 68
6.3 Maximum view size 71
6.4 Data transfer size and time 73
6.5 View size and render time 75
6.6 Adding rows or columns 77
7. Discussion 79
7.1 Table or canvas 79
7.2 Recommended settings 81
8. Conclussion 83
9. Future work 85
Bugs 87
References 89

LIST OF FIGURES

Figure 1 - Heat map from SpellWeb2 ... 17
Figure 2 - Highlighted part is a Dendrogram from Java Treeview ... 18
Figure 3 - Original SpellWeb ... 19
Figure 4 - Webkit main flow ... 21
Figure 5 - A simple DOM illustration ... 22
Figure 6 - Chrome developer tools ... 25
Figure 7 - Events that can be fetched with the navigation timing interface ... 26
Figure 8 - An overview of the MVP-pattern ... 28
Figure 9 - Heat map widget architecture ... 29
Figure 10 - The relation between identifiers, data blocks and view pages ... 31
Figure 11 - DOM for a table consisting of 10 genes and 10 datasets ... 32
Figure 12 - Dendrogram widget architecture ... 34
Figure 13 - A part of a dendrogram with 6000 genes ... 35
Figure 14 - Portion of the dendrogram in figure 13 ... 36
Figure 15 - Multiple heat map widgets on the same page. ... 37
Figure 16 - An illustration of dendrogram and heat map interaction ... 38
Figure 17 - Simplified client/server class diagram ... 43
Figure 18 - The three views that require data in the heat map widget ... 45
Figure 19 - Illustration of the client data storage ... 46
Figure 20 - Heat map widget paging and size handlers ... 48
Figure 21 - 20 genes and 10 datasets rendered with a HTML table ... 51
Figure 22 - 20 genes and 10 datasets rendered with a canvas ... 53
Figure 23 - Double buffering with the canvas ... 54
Figure 24 - Creating a new search with the heat map widget ... 59
Figure 25 - Paging horizontally with the heat map widget ... 60
Figure 26 - Adding new columns with the heat map widget ... 61
Figure 27 - Highlighting datasets and genes in the heat map widget ... 62
Figure 28 - Getting detailed information in the heat map cells ... 63
Figure 29 - Getting detailed information in the horizontal header ... 63
Figure 30 - Evaluation setup ... 66
Figure 31 - Heat map widget test suite ... 67
Figure 32 - Memory usage while inspecting 16.800.000 heat map cells. ... 68
Figure 33 - CPU usage while inspecting 16.800.000 heat map cells. ... 69
Figure 34 - Network usage while inspecting 16.800.000 heat map cells. ... 69
Figure 35 - Memory usage for different views ... 71
Figure 36 - JavaScript heap inspection for a view of 200 columns and 200 rows using a canvas ... 72
Figure 37 - JavaScript heap inspection for a view of 200 columns and 200 rows using a HTML table ... 72
Figure 38 - Transfer size for different data block sizes ... 73
Figure 39 - Transfer time for different data block sizes ... 74
Figure 40 - Canvas and HTML table total rendertime ... 75
Figure 41 - Detailed render time for the HTML table ... 76
Figure 42 - JavaScript timings for adding rows ... 77
Figure 43 - JavaScript timings for adding columns ... 78

Figure 44 - Showing 200 genes and 100 datasets with 1920x1200 pixels using the canvas 80

LIST OF ABBREVIATIONS

API	Application Programming Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
DS	Distributed Spell
GB	Gigabyte (1024^3 bytes)
GWT	Google Web Toolkit
HTML	Hyper Text Markup Language
JSON	JavaScript Object Notation
MB	Megabyte (1024^2 bytes)
MVP	Model View Presenter
ms	millisecond
KB	Kilobyte (1024 bytes)
RPC	Remote Procedure Call
SPELL	Serial Pattern of Expression Levels Locator
TB	Terabyte (1024^4 bytes)
W3C	World Wide Web Consortium

1. INTRODUCTION

The research presented in this thesis is motivated by knowing that it could possibly help scientists gain new knowledge of the human body. If this happens to not be true, we at least gain some insight into how web applications can handle and visualize large amounts of data.

The amount of biological data is increasing rapidly. More than 7% of scientists asked in [K11] has used or created datasets larger than 1TB and about 20% has worked with datasets larger than 100GB. 55% report that they rarely use data from published literature or archival databases in their original research pagers.

Where does all this data come from? The human genome is estimated to consist of about 25.000 human genes. When thousands of researchers are conducting experiments on these genes an enormous amount of data is generated. They started by trying to find out why someone has blue eyes and others brown eyes. Now they are intensively trying to figure out why someone gets cancer and others not. Consider if each person who got cancer had a dataset with tests for their blood work. In Norway around 22.000 people get cancer every year and multiple tests are done [F05]. This alone creates large amounts of data.

"Visualizing biological data on computers is now widely used to help understand and communicate data, to generate ideas and to gain insight into biological processes." [OGG10]

Computer scientists and biologists are working closely together to create applications that display data in ways that enables biology experts to find important patterns which would otherwise be very hard to find. These applications must analyze and interpret the data in ways that remove unnecessary information, but leave the important and interesting parts. Then these important data can be visualized to the user.

The usual approach has been to use desktop applications for visualizing the data. Desktop applications are compiled to machine code and have full access to a computer's hardware. This provides desktop applications with higher performance than applications running inside a browser.

Web applications have traditionally incurred several restrictions which are not present for desktop applications. The applications had no access to the computers file system, ran on a single thread, restricted memory usage and had no graphical hardware[1] support. These limitations have made it hard, if not impossible, to create applications that required heavy computations.

[1] Graphical hardware support refers to the use of graphics cards to help render the web pages

However browsers are evolving and enormous effort has been put into making JavaScript faster. JavaScript as is the way browsers generate dynamic content. Further standards are emerging which to some degree removes all of the previous limitations.

Google Maps and Bing Maps are examples of applications exploring large amount of data. Google has also implemented the Mandelbort Set[2] in JavaScript. Microsoft has created numerous tests[3] that show the performance of JavaScript. Together these applications show that the browsers can work with large amounts of data and compute it.

We therefore believe it is possible to use web applications for visualizing biological data as well. Web applications also provide some advantages for free. Web applications enables users to collaborate over geographical areas, and this is important as researchers often come from different institutions and even continents [BWL06]. Another motivation for using web applications is their portability. Since the application is running inside a browser, it can be used from any operating system supporting modern browsers. Further no installation of third party programs is required for running the application.

Visualizing biological data in the browser is something that we have been working on before. The previous version, SpellWeb2, of the heat map had problems when trying to inspect all the data. The reason for this was quite simple; basically our raw test data are 900MB. That meant that a browser wanting to display all the data would need at least this amount of memory, not including the required memory for the visualization itself. Evaluation [J10] of the old application showed that memory requirements made it impossible to inspect all the values in the datasets.

In this project we seek to properly implement this heat map visualization enabling it to be truly scalable. Further we have started the work on a widget library, containing scalable and useful visualization widgets for presenting biological data. The idea is to provide an easy to use implementation of common visualizations which can be reused at several web pages. Different widgets within a page will be bound together through an event bus, giving the possibility for different views to interact in a way that could highlight results that was not otherwise apparent.

We have implemented a scalable heat map widget for exploring biological data. A heat map is a way to visualize colored values in a grid. In our particular case the values represents the outcome of tests on genes, and these gene tests are parts of different datasets. In practice a heat map can also be used for other applications, and some common examples can be found at [WIKI1].

A web page can add one or more instances of this widget, and these instances can send events to each other through the event bus. Each widget registers on itself on the bus, and events coming through it will be propagated to its listeners. A widget is responsible for handling an event itself, which means that new types of widgets can react differently to events.

[2] http://juliamap.googlelabs.com/
[3] http://ie.microsoft.com/testdrive/

We found the main change required to implement scalability was to split up the data into a number of smaller visualizations. The idea comes from the paging mechanism found in most modern search engines, like Google and Bing. When a user searches only the top 20 results are shown. The user has the ability to increase the number of results returned. If the user does not find what she is looking for in the first page, she can press a link to go to the next page.

Heat maps can follow a similar approach, but with the possibly thousands of rows and millions of columns, a heat map must be able to page in both vertical and horizontal directions.

The evaluation of the widget has shown that we are now able to visualize all the data we have available. This is 24000 genes within 700 datasets, giving a total of 16.800.000 individual cells. The widget should be able to scale to millions of datasets, making it capable of visualizing data for years to come.

We evaluated the maximum size for a single widget to be around 200 rows and columns, containing 40.000 cells of values. The size of this visualization is 30.000x9.000 pixels, which is 117 times the size of a 1920x1200 pixels display.

We have also started the work on another widget for visualizing dendrograms. This widget is unfinished, but we believe it is possible implement it in a scalable way too.

2. BACKGROUND

2.1 HEAT MAPS

Heat maps are typical solutions to visualizing biological data. In figure 1, an example of a heat map is seen. The genes are placed in rows, while different datasets are in the columns. Each cell in the heat map can have several *expression values* which represents the result of a test on that particular gene within that particular dataset. In the figure an expression value is a single color, green or red. The grey cells are datasets which does not include tests on that particular gene.

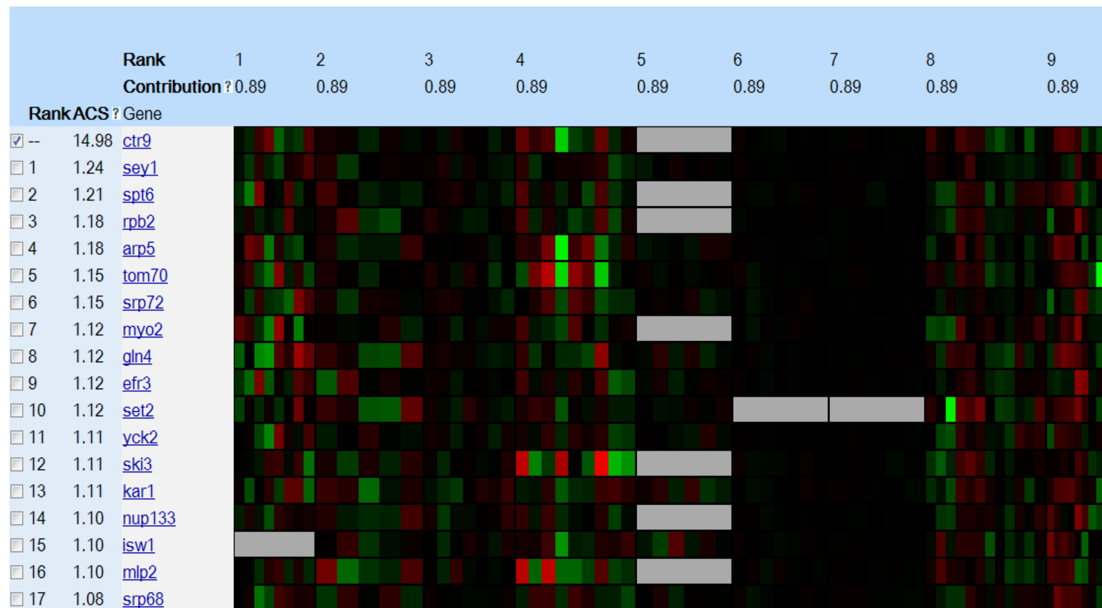


FIGURE 1 - HEAT MAP FROM SPELLWEB2

There are several other types of views that can be beneficial for biology experts. For a list of other biological tools, see [WIKI2], [WS08] or [NCD10]. Different types of biological visualization types can be seen at [WIKI3].

2.2 DENDROGRAMS

A dendrogram is often used to show hierarchical clustering, or relations, between different genes. It is a binary tree where every leaf node is a gene. Every parent of a node is either a connection between two leaf nodes or a connection between two clusters.

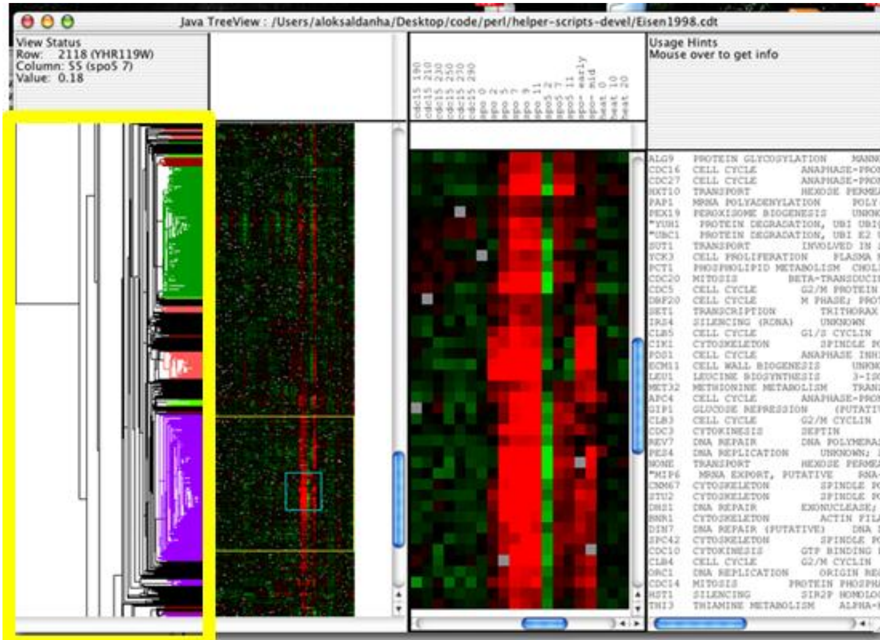


FIGURE 2 - HIGHLIGHTED PART IS A DENDROGRAM FROM JAVA TREEVIEW[4]

Dendrograms have been implemented in several biological applications; this includes Java TreeView [S04] and HIDRA [HWD07]. Both of these are desktop applications.

Figure 2 shows a screenshot from Java TreeView and the dendrogram can be seen on the left hand side that is outlined with a yellow border.

[4] Figure from http://jtreeview.sourceforge.net/examples/index.html

2.3 MOTIVATIONS FOR THIS PROJECT

In this section we explore what has been done prior to this project. We explore why previous versions of SpellWeb have not scaled and what we are focusing on to make it work this time.

SpellWeb[5] was the first web application that used the SPELL algorithm [HHM07].

SPELL (Serial Pattern of Expression Levels Locator) is a query-driven search engine for large gene expression microarray compendia. Given a small set of query genes, SPELL identifies which datasets are most informative for these genes, then within those datasets additional genes are identified with expression profiles most similar to the query set.

SpellWeb was built using the Ruby on Rails framework, with a Java backend to perform the searches. It turned out that this application did not scale for the amounts of data it was provided with. The actual view content is created on the server side.

FIGURE 3 - ORIGINAL SPELLWEB

Other parts of the system were implemented in Java, and in 2006 Google released Google Web Toolkit (GWT) which enabled developer to implement web applications using Java as source code. SpellWeb was therefore ported to GWT by Lars Ailo Bongo. The design document states this reason for the port;

"The three main design requirements are: portability, high performance, and ease of extension. We believe the Google Web Toolkit (GWT) application design satisfies these three requirements. The code is portable, since GWT is already sed by major Google applications such as Wave that runs on most devices with a JavaScript capable browser. GWT has better interactive performance than the Ruby-on-rails (used by the SpellWeb) since it moves the user interaction code from the server to the browser. In addition, the code is easy to understand and extend for developers since everything is programmed in

[5] http://imperio.princeton.edu:3000/yeast/

Java, which is the language used in the other systems (DistributedSpell, Troilkatt, GoTermFinder, and MySpell)."[6]

The idea was that the browser should get the data and create the view itself. It turned out that a straight forward implementation in GWT did not scale. [J10] explored the problem and concluded that the current approach did not scale because of memory requirements. Both of the mentioned web applications used a HTML table approach to paint the heat map.

In this project another approach for showing the values is used. As earlier version has shown it impossible to visualize all the data at once, we use pagination and visualize only a partial amount of the data in each page. To see other values, the user can use buttons to move around in the heat map. Also a new approach for rendering is implemented, using the new HTML5 canvas API.

Other projects have also worked on visualizing heat maps, and one of them is HIDRA presented in [HWD07]. This is a Java application that needs to run on the user's computer, and the datasets must also be provided.

Daniel Stødle has ported HIDRA to work on the distributed Tromsø Display Wall, which uses a similar approach as described in [LCC00].

[6] This document is placed on a server which requires the user to have a username and password. We are therefore not able to give a reference to it.

2.4 HOW BROWSERS GENERATE A PAGE

In this section we give a brief introduction for how a web page is actually created. It is intentionally written quite short, and if the reader is particularly interested we propose to look at [T09] and [W11].

To generate a web page the browser starts by connecting to a URL that the user inputs. This opens a socket and connects to the web server. This will trigger the server into sending back a response, which is normally hypertext markup language (HTML) representing the structure of the page. The browser now parses the HTML and builds the Document Object Model (DOM) required to work with the page.

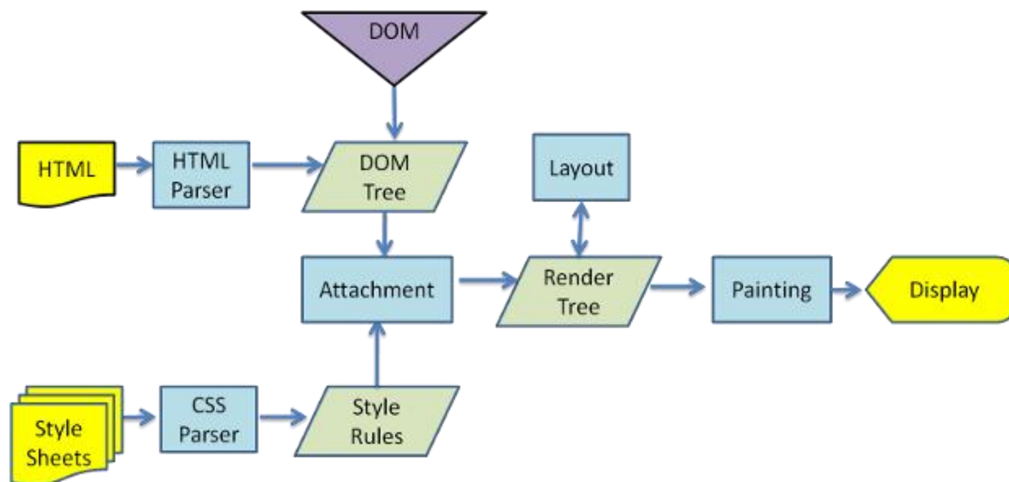


FIGURE 4 - WEBKIT MAIN FLOW[7]

While parsing HTML it can find references to style sheets (CSS), scripts and pictures. Once the parser finds either CSS or scripts it needs to parse these too.

The browser puts together the HTML and CSS into a render tree. The render tree has the syntax for the correct order in which the elements should be displayed on the page and their dimensions. Once this tree has been completed, a layout system will go over it and calculate the actual positioning each element should have. An illustration of these steps can be seen in figure 4.

After completion the web page is ready for painting, and once this is done the page will be displayed to the user.

[7] Figure from http://taligarsiel.com/Projects/webkitflow.png

2.5 DOCUMENT OBJECT MODEL

In this section we give an introduction to the Document Object Model. In its documentation [W3C05] the DOM is described as follows;

"The Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page."

The DOM consists of HTML, CSS and scripts. HTML gives a web page content and structure, CSS is used to describe the look and formatting while scripts enables dynamic changes to the page.

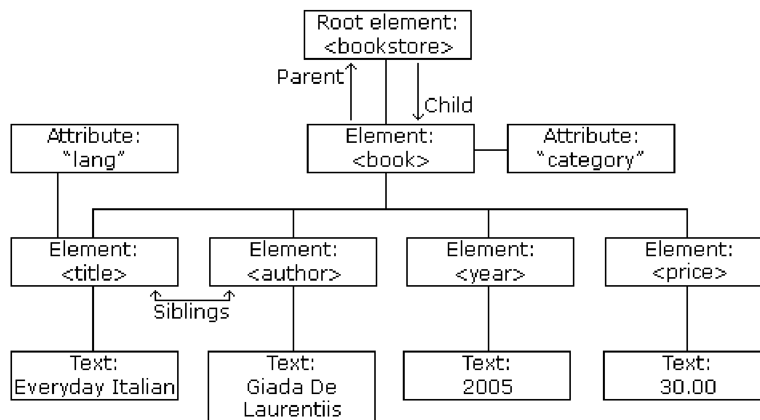


FIGURE 5 - A SIMPLE DOM ILLUSTRATION[8]

The DOM starts with a <html> element and ends with a </html> element. In the <head> section links to style sheets and scripts can be placed. It is however best to place <script> refereences at the bottom of a web page. The reason for this is that the parsing of a page stops once it meets a <script> tag, and it will instead it will execute the script. The content of the page is placed within the <body> tags.

[LDL08] argues that DOM XML is slowest when it is parsing and medicore for adding or removing elements. [W11] propose to never have more than one thousand DOM elements at any given time. The reason is that each DOM element will require time for parsing, styling and layout by the browser, too many elements in a page will degrade the performance of the application.

[8] Figure from [LDL08]

2.6 GOOGLE WEB TOOLKIT

In this section we describe Google Web Toolkit. GWT is the framework which this project is built on, and is therefore of significant importance.

GWT is made to help developers create web applications without having to be experts at HTML, CSS and JavaScript. To do this Google has developed a compiler able to create web applications using Java as source code.

This compiler has several features that help creating fast and usable web applications. It in-lines methods, removes dead code, optimizes strings and more. It obfuscates code which is good for two reasons. It makes it harder to steal code, see [W10], since obfuscated code is harder to figure out. The other good reason is that it decreases the script size, making it faster to download and parse.

Further it compiles several different version of the application, where each of them is created for specific browsers versions. This means that the problem of older browsers not following standards is removed to some extent. A last thing to add is that optimizations to the compiler can create faster applications by just recompiling. This means that if Google releases a new version and it has been optimized this benefits our project as well.

Compilers [GOOGLE09] for optimizing JavaScript also exist which has several similarities with the GWT compiler. The difference is that GWT is able to analyze all code (HTML, CSS and JavaScript) because everything is within the Java source code. This enables the GWT compiler to do optimizations JavaScript compilers cannot.

Earlier versions of SpellWeb used GWT version 1.6. Since then Google has released several new versions, and the current version is GWT2.3. This version includes several new features, and one of them is the Cell API.

"Cell widgets (data presentation widgets) are high-performance, lightweight widgets composed of Cells for displaying data. Examples are lists, tables, trees and browsers. These widgets are designed to handle and display very large sets of data quickly. A cell widget renders its user interface as an HTML string, using innerHTML instead of traditional DOM manipulation."[9]

Another feature that has been released is the UiBinder, which enables developers to write a markup syntax containing calls to GWT code within HTML markup. This makes it much easier to separate the view from logic, making what we find to be a more clear and maintainable code.

It also has support for some of the HTML5 features that new browsers have implemented. This includes canvas, local storage, drag and drop, CSS3 features and more. A presentation of the features can be seen at [L11]. Since the APIs for HTML5 is not completed and not all browsers support them, these classes are marked as *experimental*.

Currently there are also other tools emerging that do the same thing as GWT. Microsoft is developing a tool called script# (script sharp) which compiles .NET to JavaScript. It

[9] http://code.google.com/intl/no-NO/webtoolkit/doc/latest/DevGuideUiCellWidgets.html – Last visited 31th May 2011

was presented in [K11], and the developer has released the source code to the public as of lately.

2.7 JQUERY

GWT has frequent updates, but the framework still has some limitations. One of these are window managing. Third party APIs [J08] exist to help on this, but many of these require the client to download unnecessarily large scripts and contain large APIs that takes time to learn.

As this project is meant to support several views, it therefore needs some window managing. Because of this, jQueryUI[10] has been added. JQuery is a well-known and highly used JavaScript API to help development of web applications. There are two features that we use in this project, drag able and resizable.

To make a HTML element drag able a simple call to <element>.draggable() is needed. To make an element resizable a similar call to <element>.resizable() is required. Because of its simplicity this is our current way to manage windows.

Google is currently implementing a version of jQuery for GWT called gQuery [C09]. The problem is that it was not finished, and did not support the UI operations required by our widgets.

[10] http://jqueryui.com/about/ - Last visited 31th May 2011.

2.8 EVALUATING WEB APPLICATIONS

In this section we describe the problems of evaluating web applications. Unlike desktop applications, there is not a direct access to the operating system. This means that we need to use monitoring tools, debuggers and APIs provided by the browser to find errors and performance bottlenecks.

A common way to evaluate the performance is to have timers set in JavaScript. The problem is that it does not give any actual information when it comes to page load time as it ignores the requests to the server and all the steps that have to be run before JavaScript can even be started itself.

Most browsers have a debugger for web-applications. Internet Explorer 9 has F12 developer tools, Firefox has several plugins in which Firebug is probably the most known, Opera has Dragonfly and Chrome has Chrome Developer Tools.

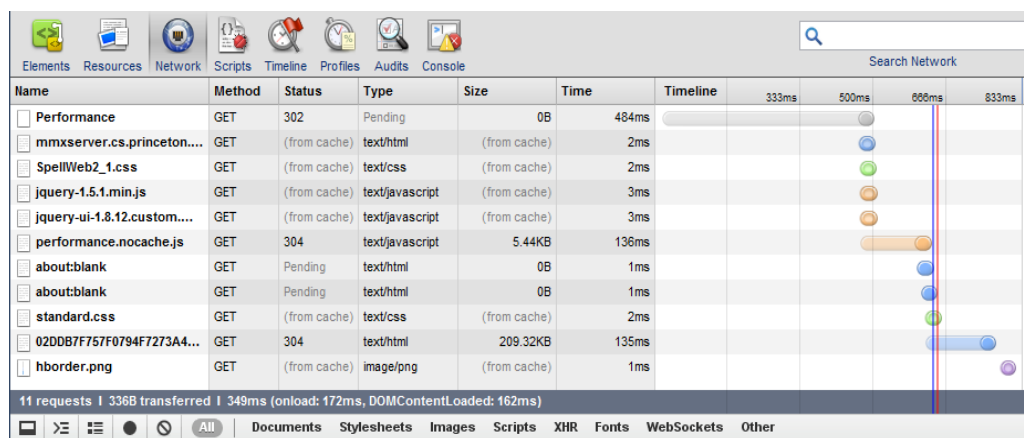


FIGURE 6 - CHROME DEVELOPER TOOLS

Most of these tools have quite similar appearance and functionality, which one to use is often of personal preference. In this project we have mainly used Chrome Developer Tools, and in figure 6 the layout can be seen.

It has the element tab which lets us inspect the DOM and change it. The resources tab has information about which files are used in the application. The network tab lets us inspect network communication. The scripts tab lets us inspect JavaScript, and set break points for debugging. The profiles tab can generate information about JavaScript stack size and which part of the JavaScript is executing a lot on the CPU. The audits tab can be used to inspect the page and look for common optimizations. Last is the console, which lets the developer write JavaScript directly into the current running environment.

In the end this gives a lot of information about how the web application is running inside the browser. However it has no information about how the operating system itself looks upon it. For this we have used the native Performance Monitor for Windows 7. It has a lot of options, in which CPU, memory usage and data transfer can be inspected.

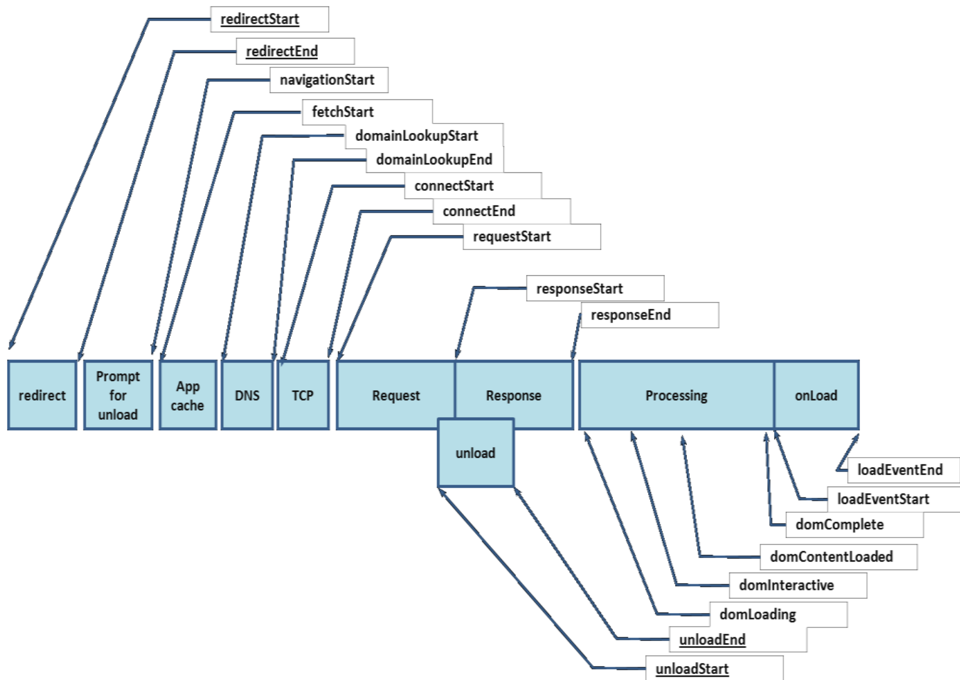


FIGURE 7 - EVENTS THAT CAN BE FETCHED WITH THE NAVIGATION TIMING INTERFACE[11]

"User latency is an important quality benchmark for Web Applications. While JavaScript-based mechanisms can provide comprehensive instrumentation for user latency measurements within an application, in many cases, they are unable to provide a complete end-to-end latency picture."[6]

The new navigation timing interface is currently in release candidate state. This means that it is ready to be implemented in browsers. Currently only Internet Explorer 9 implements this interface. The purpose of this API is to be able to analyze what happens before the HTML parsing is started in the browser. This will hopefully be a very nice feature to enable developers to see what happens before the page is actually read to be rendered.

[11] Figure and text from *http://www.w3.org/TR/2010/WD-navigation-timing-20101026/*

3. ARCHITECTURE

3.1 WIDGET LIBRARY

We have focused on creating a widget library which consists of reusable, scalable components with the ability to interact with each other. Currently we have implemented a heat map widget and we have explored a solution for a dendrogram widget.

Each widget needs to implement a Java Servlet which can be placed in a servlet enabled web server. We use Apache Tomcat for this purpose. A Servlet is a Java class in Java EE that conforms to the Java Servlet API, a protocol by which a Java class may respond to HTTP requests. The servlet will hold data persistently and deliver it once the client side requests it. A developer can also implement long running methods on the server, but we have not done this because we want to utilize the power of clients.

The client side is implemented using GWT. Each widget can implement a handler for common events which are used to interact between the different widgets. The events are shared throughout the widgets by using an event bus, which was introduced in GWT2.2.

Currently we have implemented two events; these are geneclick and datasetclick. We support these events because of their ability to show interaction between separate heat map widgets and also with a dendrogram widget. The number of events can be easily extended by creating new ones, and adding a handler to any widget it makes sense for.

3.3 MODEL-VIEW-PRESENTER PATTERN

A common problem for large applications is that once it grows and becomes more complex, understanding the code and finding bugs becomes harder. In our heat map widget we have therefore used a design pattern, known as the model-view-presenter (MVP) pattern. We use this pattern because [R09] argues that it is best suited for GWT applications. It was first proposed for the first time in [P96].

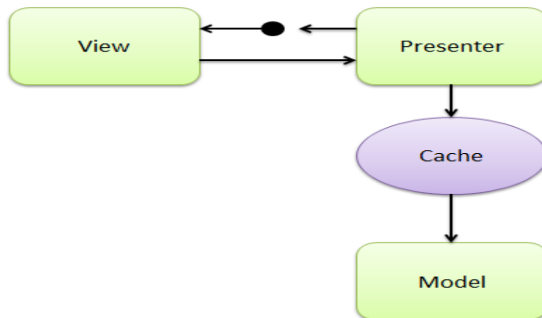


FIGURE 8 - AN OVERVIEW OF THE MVP-PATTERN

The model represents the data layer where data is storage and processed. The presenter is a broker that handles events, fetches data and updates the view as it arrives. The view is responsible for anything a user can see and interact with. The presenter defines an interface, and this interface it what links the view with the rest of the application.

By using this pattern the code becomes easier to maintain and new developers looking at the code will grasp it quicker. By knowing how the widgets are put together, finding out where things happen should be easier.

3.2 HEAT MAP WIDGET ARCHITECTURE

In this section we look at the overall architecture for the heat map widget. We illustrate how the client and server interact to create the visualization.

The widget consists of a typical three layer structure that many web applications use. The three layers are user interface, processing and data layer. The process layer is both on the client and the server. The data layers is initially only on the server, but as the client caches data some of it will be moved over to save bandwidth and improve performance.

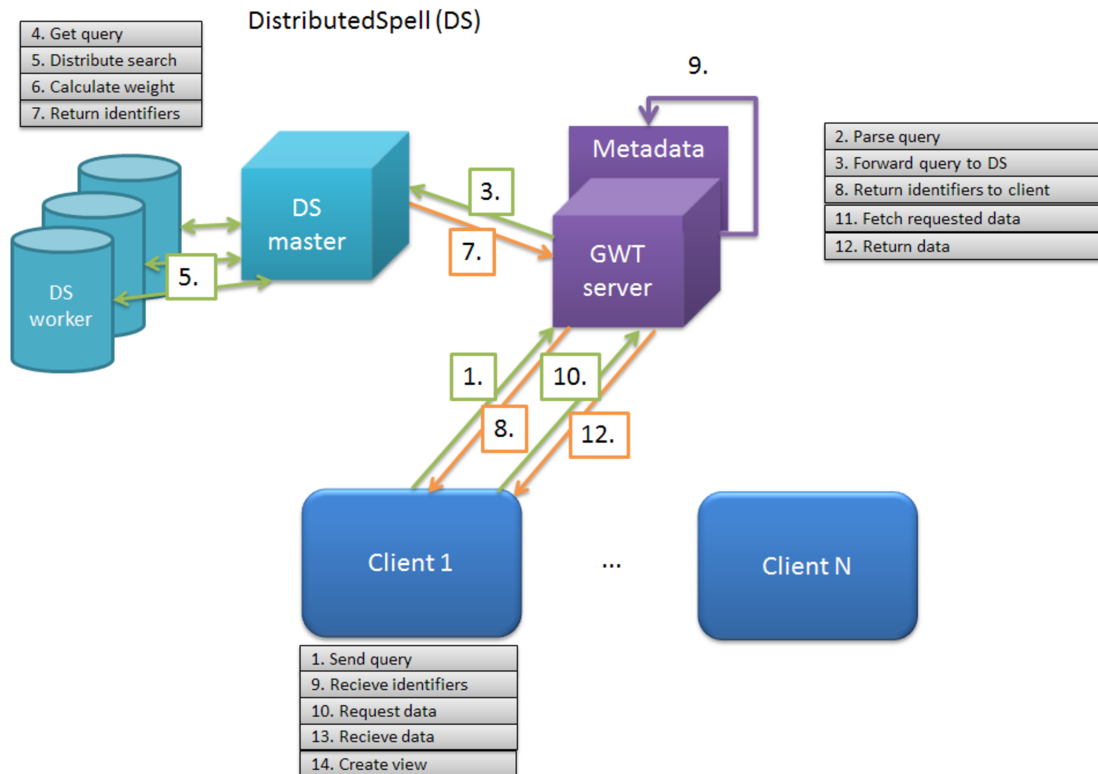


FIGURE 9 - HEAT MAP WIDGET ARCHITECTURE

Figure 9 describes how the system works when it is operative. A client connects to the frontend server by sending it a string which contains a gene name. The frontend server parses the query and looks for errors. If none are found it sends the query to Distributed Spell which will find the identifiers related to this query. The identifiers are used to identify genes and datasets. This result is sent back to the frontend server which then returns it to the client. The client takes a portion of the identifiers and sends it back to the frontend server, which will fetch the data related to these identifiers. This data consists of gene names, expression values (see section 2.1), and any other information it needs about genes. We also return information about datasets, for example names, description and citation. This data is returned to the client, and a view can be created.

It is important to understand that we cannot send all the data in one call, because that could involve sending potentially multiple gigabytes of data.

We have added one optimization which is to send the data for the first page of a view along with the identifiers. This eliminates the added network latency for doing a second call before the view can be created.

3.2.1 HEATMAP WIDGET

In this section we will explain the design of the heat map widget in more detail.

The widget consists of serval components. The user interface is running inside a browser, and consists of HTML, JavaScript and CSS. This client can send queries with genename(s) to a frontend server, which is implemented using Java and hosted in Apache Tomcat. This server is connected to a backend distributed ranking engine, called Distributed Spell (DS), which is also implemented in Java.

When a user uses the application to search, the query is at first sent to the frontend server. This server has a connection open to DS, and it hands the query over to it. DS implements the SPELL algorithm [HHM07] which ranks genes and datasets according to their relevance to the query gene(s). DS only returns identifiers of genes and datasets, meaning the client get an array of gene and dataset identifiers.

The ranking algorithm is the main reason that the heat map widget cannot easily be made up by pre-computed pictures, as in for example Google Maps. Each search is potensially unique, and there are (*number of datasets * number of genes)!* possible ways that the view could be created. For 4 genes and 3 datasets we have 12! giving 479.001.600 possible combinations for a single view.

When the client recieves the identifiers, it splits it up into blocks. For instance consider the test datasets that is used containing approximatly 24000 genes and 700 datasets. Trying to render all of this has been shown to be impossible as the client goes out of memory. This means that the size of the view needs to be smaller.

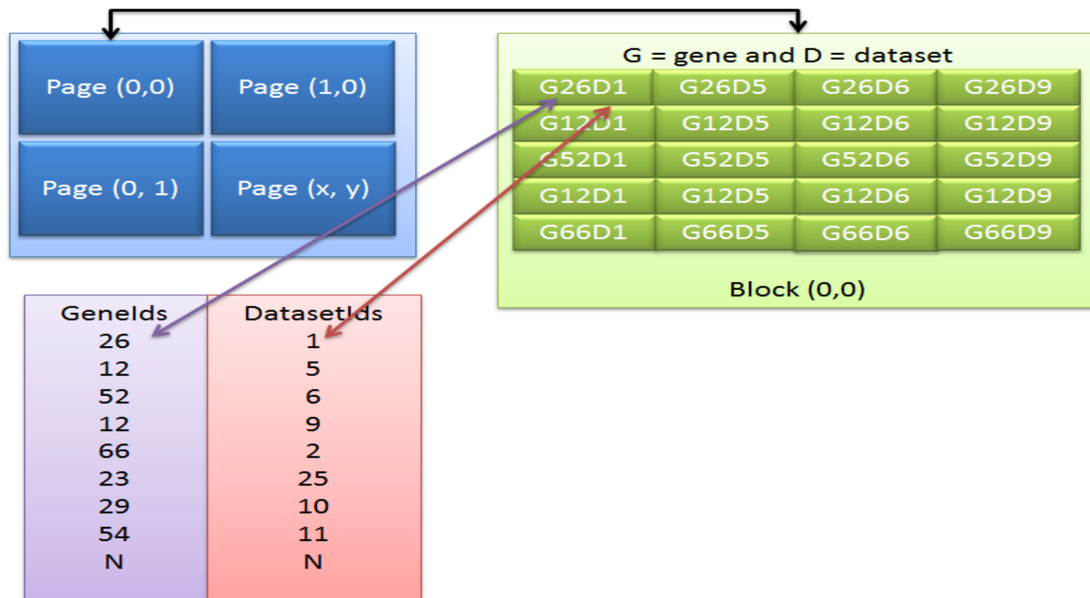


FIGURE 10 - THE RELATION BETWEEN IDENTIFIERS, DATA BLOCKS AND VIEW PAGES

The client splits the identifiers into blocks, the size of one data block is set by the developer. The block size tells the client how much data it should fetch in one exchange between client and server. In figure 10 the relation between identifers, data blocks and view pages can be seen. When the client then sends a request to the server, the server inspects the request and fetches the data related to these identifiers. Data is then returned to the client and it can create a visual representation as a heat map.

The most bandwidth intensive data is the expression values. An expression value is a floating point value that represents the outcome of a test done on a particular gene. As there is a high number of tests that can be performed, the number of expression can vary between zero and hundreds.

In the dataset GDS53 the gene BRCA1 has five expression values like this;

BRCA1 -0.609, -1.226, -0.482, 0.838, 1.480

In GDS266 there are 28 values for the same gene, and certain datasets have several hundred tests on some of the genes. This means that some pages require the server to load more data, transfer more data and the client needs to translate and render more values.

The heat map view is implemented using a pagination pattern. Pages can have varying size, and it does not necciseraily have to be the same size as one data block. If these numbers are uneven, the client will fetch several blocks to generate the view. The interface has event handlers for paging in both horizontal- and vertical-axis. This technique enables the widget to inspect a larger number of genes and datasets then any previous version. The reason is that once a user goes to a new page the view is re-rendered and data for previous pages can be removed from the buffers.

It is however still possible to create a view so large that the browser will kill the web application, and in our evaluation we inpect the maximum size for this widget.

The widget also implements simple caching and pre-fecthing of data for a particular search. The caching works by simply not removing data immidiately as a user changes the current page. The pre-fecthing works by realizing that it can be seen as likely that a user might want to look at pages in proximity. Therefor the client can pre-fetch data for adjacent pages after it has created the view.

Implementing pre-fecthing for a new search is on the other hand quite hard. To be able to do this, data for a large range of searches would be required. By inspecting the widget usage patterns speculative pre-fecthing could be implemented. [MEH10] suggest a way of doing speculative pre-fetching. Each event handler could generate a shadow copy of what would happen when if the handler is invoked. The problem is that there is a very high number of possible search alternatives and there is no way of telling what it would be.

The actual view of the heat map has been implemented in two different ways. The first approach was to use a GWT CellTable[12] and this works quite well for small views with less than 50 genes and 50 datasets. However as the view increases in size, the performance for a HTML table decreases. The reason for this is that as the table increases in size, the number of objects in the DOM increases accordingly.

```
▼<div class="celltable GL11-QJBOH ui-draggable" style="position: absolute; z-index: 1; ">
  ▶<div class="dragghandler GL11-QJBPH">…</div>
  ▼<div class="GL11-QJBDI">
    ▶<div class="GL11-QJBCM">…</div>
    ▼<div style="padding-left: 200px">
      ▶<div class="GL11-QJBEI">…</div>
      ▼<div>
        ▼<table __gwtcellbasedwidgetimpldispatchingfocus="true" __gwtcellbasedwidgetimpldispatchingblur="true" cellspacing="0" class="GL11-QJBHH">
          ▶<colgroup>…</colgroup>
          ▶<thead style="display: none; ">…</thead>
          ▶<tfoot style="display: none; ">…</tfoot>
          ▼<tbody style>
            ▼<tr onclick class="GL11-QJBAG">
              ▼<td class="GL11-QJBPF GL11-QJBBG GL11-QJBCG">
                ▼<div style="outline:none;" tabindex="0">
                    <div style="display:inline-block;background-color:#000c00; width:12%; height: 24px;"></div>
                    <div style="display:inline-block;background-color:#002400; width:12%; height: 24px;"></div>
                    <div style="display:inline-block;background-color:#390000; width:12%; height: 24px;"></div>
                    <div style="display:inline-block;background-color:#6b0000; width:12%; height: 24px;"></div>
                    <div style="display:inline-block;background-color:#006100; width:12%; height: 24px;"></div>
                    <div style="display:inline-block;background-color:#002400; width:12%; height: 24px;"></div>
                    <div style="display:inline-block;background-color:#003600; width:12%; height: 24px;"></div>
                    <div style="display:inline-block;background-color:#480000; width:12%; height: 24px;"></div>
                  </div>
                </td>
              ▶<td class="GL11-QJBPF GL11-QJBBG">…</td>
              ▶<td class="GL11-QJBPF GL11-QJBBG">…</td>
              ▶<td class="GL11-QJBPF GL11-QJBBG">…</td>
              ▶<td class="GL11-QJBPF GL11-QJBBG">…</td>
              ▶<td class="GL11-QJBPF GL11-QJBBG">…</td>
              ▶<td class="GL11-QJBPF GL11-QJBBG">…</td>
              ▶<td class="GL11-QJBPF GL11-QJBBG">…</td>
              ▶<td class="GL11-QJBPF GL11-QJBBG">…</td>
              ▶<td class="GL11-QJBPF GL11-QJBBG GL11-QJBMG">…</td>
              </tr>
            ▶<tr onclick class="GL11-QJBAH">…</tr>
            ▶<tr onclick class="GL11-QJBAG">…</tr>
            ▶<tr onclick class="GL11-QJBAH">…</tr>
            ▶<tr onclick class="GL11-QJBAG">…</tr>
            ▶<tr onclick class="GL11-QJBAH">…</tr>
            ▶<tr onclick class="GL11-QJBAG">…</tr>
            ▶<tr onclick class="GL11-QJBAH">…</tr>
            ▶<tr onclick class="GL11-QJBAG">…</tr>
            ▶<tr onclick class="GL11-QJBAH">…</tr>
            </tbody>
          ▶<tbody style="display: none; ">…</tbody>
          </table>
        </div>
      </div>
    </div>
  ▶<div class="GL11-QJBLH">…</div>
  </div>
</div>
</body>
```

FIGURE 11 - DOM FOR A TABLE CONSISTING OF 10 GENES AND 10 DATASETS

---

[12] This will in practice be compiled into a HTML <table>

In figure 11 the DOM for 10 genes and 10 datasets are shown. Each row has ten column elements. In each column there are *number of expression values* container elements for the expression values. If there is an average of five expression values for in each cell, 4.000 DOM elements are required to show the table. With a view of 50 genes and 50 datasets 1.000.000 DOM elements would be required.

For each element the browser gets a harder job creating the view. The reason for this is that if any changes are made to the page, the browser needs to parse the HTML over again and layout its new positions. With an excessively large number of elements the performance of this operation degrades.

This lead to another implementation of the view. With the emergence of the HTML5 canvas in GWT2.2 there is now a way to natively draw to a bitmap. An implementation of the heat map has been created using this canvas, and it turns out that it increases the performance in most situations. It also decreases the number of DOM elements up to 80 percent.

3.3 DENDROGRAM WIDGET ARCHITECTURE

The current dendrogram widget is unfinished, and during the development we have learned some lessons for how it should not be done. We are however sure that it is possible to create a scalable dendrogram widget that can be integrated into the widget library with more research.

In the following sections we describe what has been done so far, and why it currently is unfinished.

The architecture for the dendrogram widget is quite similar as the for the heat map. It consists of a GWT server and a client that creates a view using JavaScript. The following figure describes the current architecture.

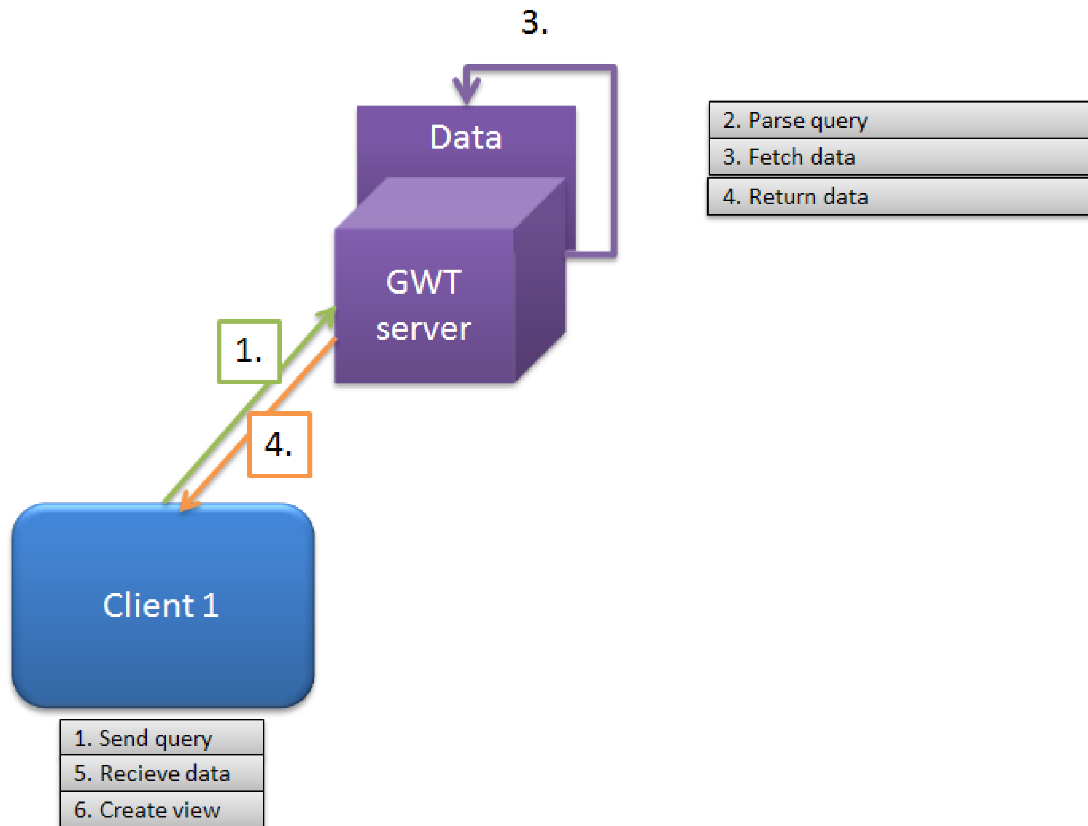


FIGURE 12 - DENDROGRAM WIDGET ARCHITECTURE

The server starts by loading dataset files, which are configured in its constructor. This will parse the files, and create a binary tree structure which is the dendrogram in the server memory. The client can then request this tree using a provided interface. Once all the data is moved over to the client, it can parse this tree and create a visualization of it.

3.3.1 DENDROGRAM WIDGET

The dendrogram implementation utilizes the canvas API which arrived with GWT2.2 at February 11th 2011. We realized that we could port an implementation created for the Tromsø Display Wall, written by Daniel Stødle. His code is written in C++ and uses OpenGL to render the view in its environment. We have ported the code over to Java and GWT, where the server acts as a data service for the client. The client uses remote procedure calls to fetch data, and it uses the canvas to paint it.

The data is structured as a double linked list, where we have a head node with a left and right pointer. We can recursively parse this structure by checking if the left or right node is not null. If the left node is not null, we go left. If both are null, we know that we are at a leaf node. We then return, and go right.

The problem that we had is how we can scale the size of the dendrogram. Our test data consists of *yeast* genes, which are over 6000 genes.

If each leaf node took 20 pixels, and we had a space between them of 10 pixels we would need 6000*20 + 5999*10 which is 179.990 pixels in height. First of all the canvas does not work for this size, it defects when the canvas is around 50.000x500 pixels in size. Second a web page with a height of 180.000 pixels does not seem like a good idea.

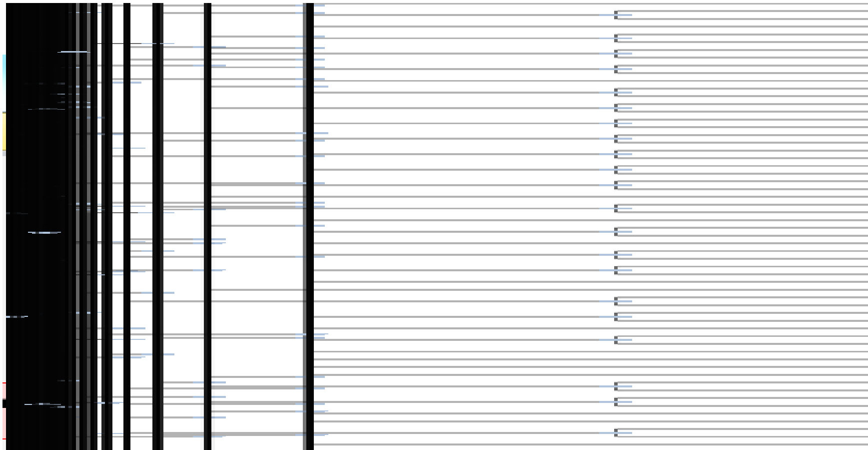


FIGURE 13 - A PART OF A DENDROGRAM WITH 6000 GENES

Further as we can see in the picture above the dendrogram becomes very dense. We therefore tried to split the data into smaller parts, and set new root nodes for the tree. Doing this gave us a dendrogram that can be seen in figure 14.

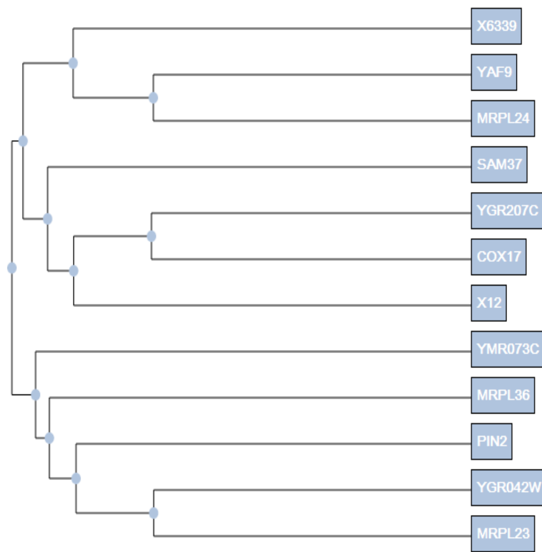


FIGURE 14 - PORTION OF THE DENDROGRAM IN FIGURE 13

In practice it could work to create a dendrogram which consisted by several smaller trees. The problem is that a dendrogram is supposed to show the hierarchical clustering of the genes, and only looking at smaller parts would not enable the user to get the bigger picture.

This means that we intend to show the whole dendrogram. The maximum size for a canvas has yet to be found, all we can verify is that it does crash at sizes bigger than 50000x500 pixels. The documentation does not state any maximum size, the width and height variables are of type int. The theoretical limit should therefore be 2^{32}-1 x 2^{32}-1 pixels.

Because of this we have concluded that we need to do more research before we can actually implement a working dendrogram with several thousands of elements in the browser.

3.5 SEVERAL VIEWS ON THE SAME PAGE

Our project support several widgets on the same page at once. Currently this is done by creating a separate visualization for each search, and binding them together using an event bus. In this project the heat map views does not share the data, and the reason is the way GWT stores values used in a CellTable. A more detailed explanation can be found in section 4.2.4..

FIGURE 15 - MULTIPLE HEAT MAP WIDGETS ON THE SAME PAGE. BACKGROUND IS USING A TABLE IMPLEMENTATION, WHILE THE FOREGROUND USES A CANVAS. THE GENE SEY1 IS SELECTED IN BOTH VIEWS.

The dark blue area at the top has a drag handler created by using jQuery. This enables a user to move the widgets around. On hover a view will increase its z-index to a value which places it in the foreground. The z-index is a CSS property which tells the browser where an element should be placed on the z-axis.

3.6 INTERACTION HANDELING

The views need to implement a way of communicating with each other in order to create integration. The events that are supported are *gene- and dataset-clicks.* We support these events because they are a nice way to show what we could do with interaction.

In the GWT2.2 Google implemented an abstract class called EventBus[13]. This bus supports exactly what this requirement needs, a way of sharing events throughout the whole application.

Any widget that wants to get events from other widgets registers itself on the bus, and implements their own handler for what to do when a certain event arrives. This means that widgets can handle a gene click in any way they want, all they know is that a gene click happened somewhere else in the application.

The number of events can easily be extended, they just need to be registered on the bus and any view that wants to listen to it needs to implement a handler.

Currently clicking genes and datasets headers highlight the selected value in any view that is currently showing them.

The views also support a way of inspecting values in more detail by hovering the cell with the mouse.

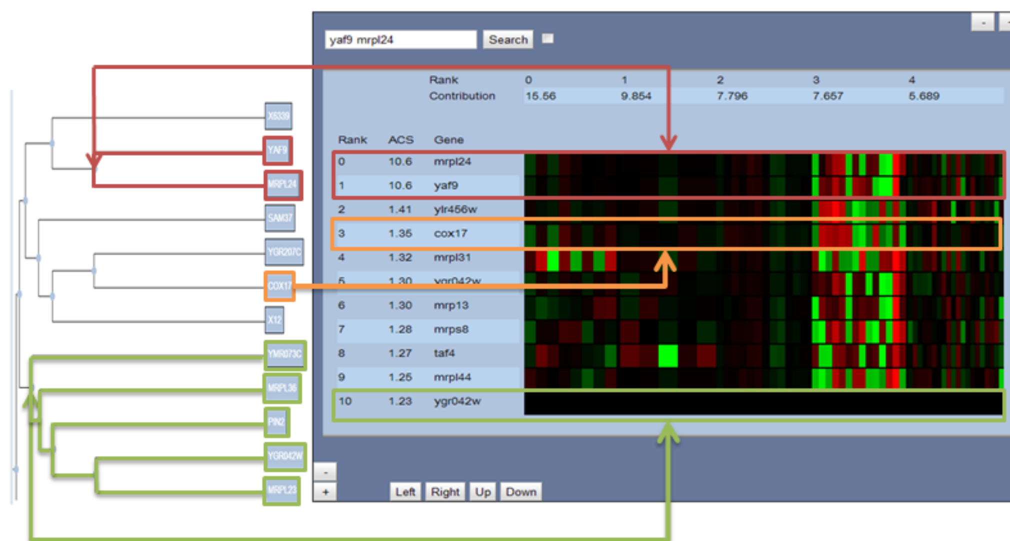


FIGURE 16 - AN ILLUSTRATION OF DENDROGRAM AND HEAT MAP INTERACTION

Figure 16 is an illustration of how we intended the heat map widget and dendrogram widget to interoperate. By either clicking on the dendrogram or heat map gene, a gene select event would be sent out on the event bus. By clicking on cluster intersections in the dendrogram, every gene within that cluster would be highlighted in the heat map.

---

[13] http://google-web-toolkit.googlecode.com/svn/javadoc/2.3/com/google/gwt/event/shared/EventBus.html - Last visited 31th May 2011.

For instance if you clicked at the spot at the topmost left arrow in the dendrogram, the event would send out a “gene click on YAF9, gene click on MRPL24”. As we can see these genes are in the heat map, and a selection would then happen.

4. IMPLEMENTATION

The client side of SpellWeb has been completely re-implemented in this project. Earlier version did not scale and was using an old GWT API. Further the old code was becoming hard to maintain.

The new implementation utilizes the MVP-pattern to help creating cleaner and more modular code. An example of how this has worked out quite well was when the canvas view was added to the heat map widget. It only required the canvas to implement the same interface as the old view did, and the rest of the application required only minor changes.

The project also utilizes the UiBinder[14] which was shipped with GWT2.2. It enables the programmer to create a declarative layout using XML syntax quite like HTML while keeping the logic in Java code. This separation keeps the code much more clear than it was before.

In the next sections we will present a detailed explanation of how the heat map widget works. We first look at data storage, how it is moved from storage to the client, and how we use the data to generate a view. The quick explanation is; model is responsible for keeping persistent data available, the presenter will fetch data from the model and the view will be created with data coming from the presenter.

At last we will present some implementation details for the dendrogram.

[14] http://code.google.com/intl/no-NO/webtoolkit/doc/latest/DevGuideUiBinder.html - Last visited 31th May 2011.

4.1. MODEL

In this section we explain how the model operates in the heat map widget. We look at what is does, and how it provides methods to move data from the server over to the client.

The model is responsible for storing data and fetching it once the client requests it. The server side currently consists of two separate Java modules. Distributed Spell is responsible for ranking genes and datasets. The other module is a Java Servlet running inside Apache Tomcat web server.

The client can communicate with the server through remote procedure calls (RPC). This is done by sharing an interface that the client uses for method calls and the server implements.

Figure 17 shows simplified class diagram for client server data communication. We define the two interfaces SpellWebService and SpellWebServiceAsync. SpellWebService extends RemoteService, which is a GWT specific interface for enabling RPC services on the server. The SpellWebServiceAsync interface is used by the client, and needs to map every method that the server interface provides. The asynchronous interface provides callbacks for the client to use. The client calls a provided method, and once it completes a callback is done with the requested data in its parameters.

As the web server starts it initializes a servlet defined in the application .gwt.xml file. This file provides the servlets constructor parameters. The parameters contain a system path for data files and an URL for finding Distributed Spell.

The servlet itself implements the SpellWebService interface, and we have defined three new methods for this project. This is *getGeneIdentifiers()*, *getGeneIdentifiersAndFirstBlock* and *getRenderData()*.

The getGeneIdentifiers() method returns four arrays; gene identifiers, dataset identifiers, gene scores and dataset weights. GetGeneIdentifiersAndFirstBlock() returns all the previous and also fetches data for the first view in the same call. GetRenderData() returns data associated with the identifiers given in its parameter.

These methods use four classes to contain the data; *SearchIdentifers*, *HeatmapBlock*, *SpellGene* and *SpellDataset*. The search identifiers are the arrays mentioned in the previous paragraph. A heat map block is a structure that contains a set number of SpellGenes and SpellDataset. What a SpellGene and SpellDataset contain can be seen in figure 17.

The server is running Java byte code while the client is running JavaScript. This means that they need a common format to exchange data. This is natively done in GWT by using JavaScript Object Notation (JSON), but it is possible to use other data-interchange formats too.

Currently the servlet loads all data into memory. The plan for a future server implementation is to place the data in Hadoop Distributed File System and Hadoop database. This will enable the server to scale because it does not need to keep all data in memory.

FIGURE 17 - SIMPLIFIED CLIENT/SERVER CLASS DIAGRAM

4.2 PRESENTER

The presenter is the part of the application that connects model and view. It is responsible for fetching data from the server and putting it into the view. It also handles events coming from its view or the event bus.

To create an instance of the presenter it needs three parameters; the view, a data service and an event bus. It then attaches itself to the view using the interface it provides.

The developer can now call the *go(HasWidgets container)* method on the presenter. The container is an element that is provided by developer to contain the heat map view. The presenter will send a request to the server, through the data service, containing a query consisting of one or several gene names. This will make the servlet fetch the identifiers from DS. When the identifiers return to the client, the presenter sends a new request for data required to make the first page of the heat map. If this request is successful the presenter will add event handlers and place the data into the view.

There are a number of methods and classes involved in creating the first page in a view. Let's look at the call stack required.

1. Presenter.getSearchIdentifiers()
2. SpellWebService.getGeneIdentifiers()
3. Presenter.getRequiredData()
4. HeatmapFetcher.executeCallbacks()
5. SpellWebService.getRenderData()
6. For each result – Presenter.addBlockToProviders()
7. On all received – Presenter.allDataRecieved()
8. Presenter.refreshView()

In the following sub sections we will go through each of these calls and explain in more detail what each them does.

4.2.1 IDENTIFIERS

The first call we do will request gene and dataset identifiers from the server. It uses the data interface to send a gene name to the server and the number of genes and datasets that it wants to be compared.

The actual call is asynchronous and has the signature *SpellWebService.getGeneIdentifers(String orgId, String queryStr, int maxScores, int maxWeights, int evGenes, int evDsets, AsyncCallback<GeneSearchIdentifiers> callback)*.

The request can simply fail or succeed, so any AsyncCallback has defined the two methods *onFailure(Throwable caught)* and *onSuccess(T result)* where T is the result type.

When the call returns with a success it sets the gene ids, the dataset ids and creates two hash maps. These maps are defined with gene ids or dataset ids as keys, and return the related scores for the particular key. When this is completed, the presenter puts the data into views for later usage. At this point the presenter will call its getRequiredData() method.

4.2.2 DATA

The getRequiredData() method is a convenience call to get data from the server. It will do the next call to the server and requests data to generate a page. The method signature is *getRequiredData(int startRow, int stopRow, int startColumn, int stopColumn)*. This method uses the parameters to find the correct gene and dataset identifiers.

Once the correct ids are found they are sent over to the HeatmapFetcher. This class is responsible for doing callbacks to the server to get data related to the identifiers.

It defines two types of callbacks, the HeatmapCallback and the HeatmapPrefetchCallback. The standard callback is used to get data which the client needs to render the current page, while the pre-fetch callback is used to get data that might be used on page change or increase of page size.

Once the callbacks are set up, they are sent over to the server which will find the correct data and return it. On successful calls, the HeatmapFetcher will call on the addBlockToProvider() method in the Presenter class. Once all callbacks are completed, it will call the allDataRecieved() method.

4.2.3 ADDING DATA TO THE PROVIDERS

The heat map has three different views which require data. These are the horizontal header which shows dataset information, the vertical header which shows gene information and the heat map container itself. These can be seen in figure 18.

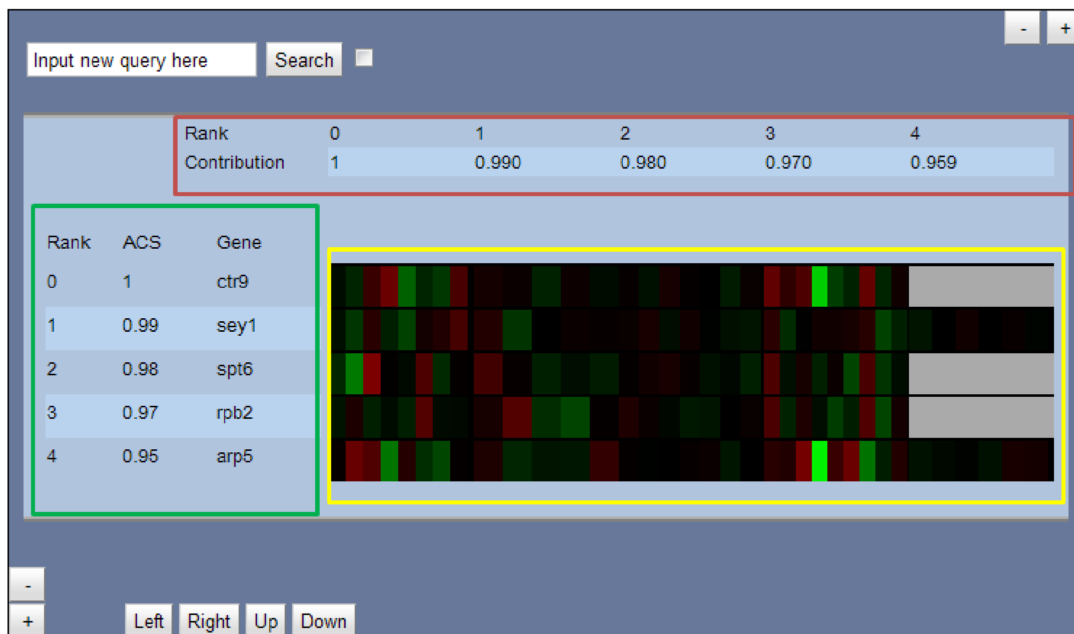


FIGURE 18 - THE THREE VIEWS THAT REQUIRE DATA IN THE HEAT MAP WIDGET. THE RED ZONE IS THE HORIZONTAL HEADER, THE GREEN ZONE IS THE VERTICAL HEADER AND THE YELLOW ZONE IS THE HEAT MAP CONTAINER.

Once the data are arrived, each row will be stored as an entry in a list structure. Each column is an entry in a hash map. Once new data arrives there are two alternatives for what can happen. If the row does not exist a new hash map is created and put into the list. Then the values for each column are put into that hash map. If the row already exists, the existing hash map is fetched, and the new columns are put into it.

4.2.4 CLIENT DATA STORAGE

The data model has in some ways been forced upon the widget because of the way CellTables and their data providers are linked in GWT. Any CellTable must be linked to an AbstractDataProvider<T>.

There are only two implementations of this abstract class which is the ListDataProvider and AsyncDataProvider. Both of the implementations are list based, where each entry in the list represents one row in the view. We believe this implementation is not well suited for a view where each column in the table is of the same type of data.

In the heat map each cell is of the same type. This lead to a design where each entry in the data provider is a LinkedHashMap. Figure 19 illustrates the client data storage. There are several good properties with using a LinkedHashMap. First of all it gives constant-times for get() and put() operations assuming the hash function disperses the elements properly among the buckets. This makes it easy to hold track of elements as the key used is the dataset ids and since there can be no duplicate keys, duplicate data does not go into the map.

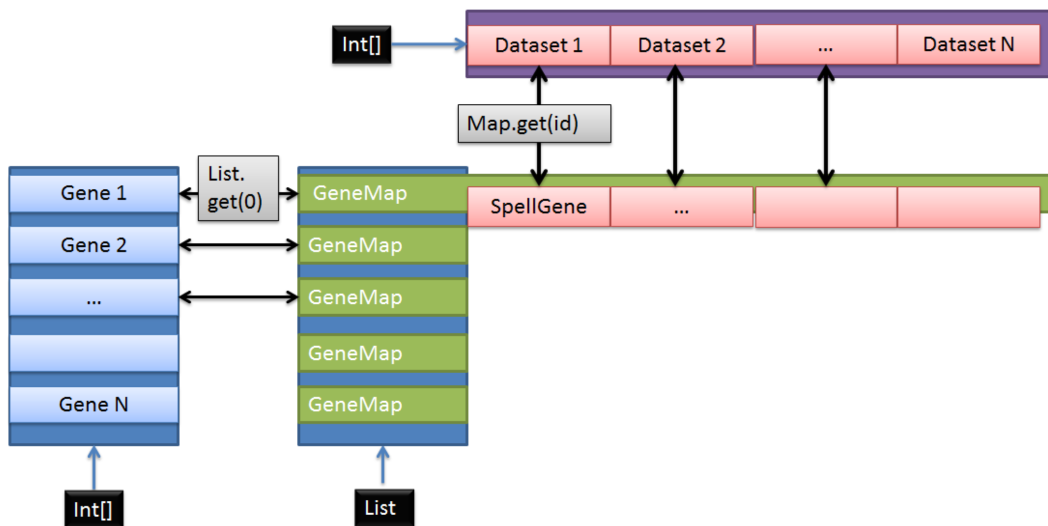


FIGURE 19 - ILLUSTRATION OF THE CLIENT DATA STORAGE

Last is the special property of the *linked* hash map; it can be easily used as a cache. Since there is a linked list holding order of when elements first appeared in the map, it is possible to make the map remove old elements when a certain threshold is reached. In practice this means that old data will be removed automatically as new data arrived, keeping memory usage down.

Since the LinkedHashMap will remove data automatically, it means that we could page along the horizontal axis theoretically an unlimited number of times without going out of memory.

The vertical axis removes data in another way. In the next section we present the pager which is used to know what the current offsets are in the view. Each time we page vertically there is a method that removes data from the hash maps on page x-2 while going down, and x+2 while going up. This enables us to keep memory usage low, but we are not removing the actual hash map itself so there is some overhead.

4.3 HEAT MAP PAGER

The pager is an important class for the widget. It is basically responsible for knowing what the view is currently displaying. It keeps track of the index for the first column and last column, and it the same for rows. It also knows how many rows and columns should be displayed on a page.

This means that any events that make the view change regarding any of these values must use the pager to update its boundaries.

FIGURE 20 - HEAT MAP WIDGET PAGING AND SIZE HANDLERS

In figure 20 we can see three outlined areas. The red zone is used to increase or decrease the number of rows in the view by five, while the yellow zone does the same for columns. Both of these events are handled in the ViewSizeEventHandler class.

The green zone is used to change the current page in the directions the buttons indicate. These events are handled in the PagingEventHandler.

The pager has four methods which the views can use to find the current offset values; *getIndexOfFirstColumn(), getIndexOfLastColumn(), getIndexOfFirstRow()* and *getIndexOfLastRow()*. By using these, they know which data should currently be displayed to the user.

4.4 VIEWS

In the following sections we describe how the view of the heat map widget is created. We detail the different modules it is composed of and relate it to the presenter.

The view for the heat map widget is defined in the two files HeatMapView.java and HeatMapView.ui.xml. This class implements the following interface which the presenter defines:

```
public interface IHeatmapDisplay2_0 {
            IHeatmapTable getHeatmapContainer();
            IHeatmapHorizontalHeader getHorizontalHeaderContainer();
            IHeatmapVerticalHeader getVerticalHeaderContainer();
            IHeatmapPager getPagerContainer();
            HasClickHandlers getAddRowsButton();
            HasClickHandlers getRemoveRowsButton();
            HasClickHandlers getAddColumnsButton();
            HasClickHandlers getRemoveColumnsButton();
            HasClickHandlers getNewSearchButton();
            HasText getNewSearchTextBox();
            void setStartColumn(int columnStart);
            void setStartRow(int startRow);
            void setNumberOfRows(int numberOfRows);
            int getNumberOfRows();
            void setNumberOfColumns(int numberOfColumns);
            int getNumberOfColumns();
      }
```

As we can see the interface requires the class to return four interfaces and a number of UI elements. The UI elements are self-explanatory by looking at the method names.

The three topmost interfaces are highlighted in figure 18, and the last interface is the green zone in figure 20. As we can see the complete view consists of four modules, and the layout is defined in the HeatMapView.ui.xml file. This file contains XML syntax much like HTML and it also defines a number of CSS rules.

The reason we split the view into separate modules was to make it easier to use the widget to display data which did not involve genetics. The idea was that we could provide a widget that would let a developer use the heat map container, but provide other headers. We have later realized that the widget is currently made in such a way that it does need a lot of refactoring before this would be possible.

We provide two implementations of the IHeatmapTable interface. The first one uses a similar approach as the older versions did, using a GWT CellTable. Google presented that the Cell API was faster than older implementation of HTML tables, however the implementation did not perform as well as we hoped for.

Another approach was therefore implemented. It uses the HTML5 canvas API, which is currently in an experimental phase. In practice this means that certain browsers might not support it or that it could change in the future.

In the following sections we first describe the class that translates the expression values to colors, then we present the two different heat map modules, and last we explain the headers.

4.4.1 CREATING THE COLORS

To create the colors in the heat map a simple class has been made. Initially it is constructed with an argument to set the color setting that is wanted. The default version is red-green, but can be extended to support other colors. An important aspect for supporting other colors is color blindness. According to [M08] 7% of American males are color blind in some way. This means that these persons can have a hard time using a particular color scheme.

The translate method takes in an array of expression values, and runs it through a method to generate the appropriate color values.

This class currently also has a bug that needs to be fixed. We have been working with two types of biological data, one is *yeast* and the other is *human*. We started evaluating the widget using the yeast datasets, but once we realized that it scaled for this data we changed to the larger human datasets. The problem was that we did not realize that the human genes also required an additional method which yeast did not use. This method instead returned the color value for grey (#AAAAAA), which is the same value that is used for missing data. This made us falsely believe that we were missing data. This gives an important lesson; create an exception when a method is not actually implemented.

4.4.2 TABLE VIEW

The table view is generated using the CellTable implementation provided by Google. Thereby the main components are a data provider, a cell implementation, CSS resources and the cell table class itself.

The data provider has already been explained in section 4.2.4, so here we focus on the cell, style sheet and how it all works together. The CSS rules are defined just as we would use any other style sheet, and are can be used on the table by adding it in the constructor.

The HeatmapTableView implements a GeneCell, and in its render method the actual visualization of a cell is generated. Each cell has an input object, GeneMap, which is the linked hash map that has all the data for a particular row. By finding the index in the table, and adding that with the pagers start column, the actual index of the cell can be found. Then by using the dataset id at this index, a key into the hash map is fetched and the actual data for a particular cell can be found. For example, with a view size of five, page three has a start column of 15. If we want the key for this cell, it can be obtained from datasetIds[15], and we can then use genemap.get(datasetIds[15]) to obtain the expression values.

The cell table itself has a reference to which row it is currently on. This means that the pager only needs to keep track of columns for this type of view. In the section 4.4.3 we will look at the canvas view which also requires us to keep track of both row and column offset.

FIGURE 21 - 20 GENES AND 10 DATASETS RENDERED WITH A HTML TABLE

A cell table handles adding rows and columns in two different ways, which can be inspected by looking at the source code for the cell table and its related classes.

When the table is adding new rows, a complete redraw is not done according to this comment from the HasDataPresenter class in GWT.

```
// Remember the inserted range because we might return to the
same
// pageStart in this event loop, which means we won't do a
full
// redraw, but still need to replace the inserted nulls in the
view.
```

When we are adding columns however, the table is completely redrawn. This means that adding new columns is a more expensive operation than adding new rows. However as we shall see in the evaluation part both operations are quite slow when the number of rows and columns increase to a certain size.

Another problem with the cell table is that we cannot explicitly control its rendering. This means that any browser painting will not be done before the complete table is ready for visualization.

4.4.3 CANVAS VIEW

When implementing the heat map widget we first did not intend to use the canvas. The main reason for this was the fact that it first appeared in GWT2.2. This version was released February 11th 2011, around one month after this project had started.

Once the table version was implemented we were able to demonstrate that it would scale for small views, but we realized that it did not perform very well when creating views with more than 50 rows and columns. Especially it performed badly when we added new rows or columns on an existing visualization. The main reason for this is the amount of DOM elements the table implementation created.

At this point in time we had also started to look at a dendrogram implementation, and implementing the dendrogram using standard HTML elements did not look promising. In relation to this we had started working with the canvas API, and realized it could also be used for the heat map widget.

The first problem of using the canvas we engaged was the how to fetch events that it we required it to implement. We wanted to be able to show detailed information for cells in the same way we had implemented in the table version.

After discussing this problem with Lars Ailo Bongo and talking with some developers at the IRC channel for GWT on irc.freenode.net, it turned out that handling events was easier than expected. By attaching an event handler on the containing DOM element, and using this elements relativeXPosition() and relativeYPosition() simple arithmetic's could be used to calculate which row and column the mouse was currently at.

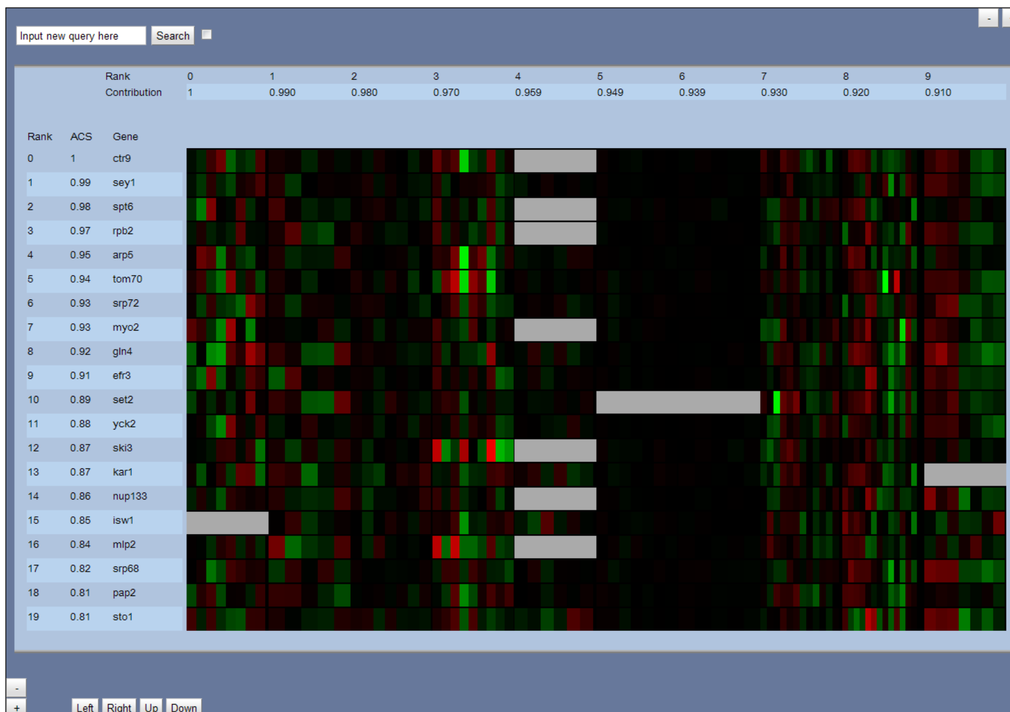


FIGURE 22 - 20 GENES AND 10 DATASETS RENDERED WITH A CANVAS

The canvas is very reliant on the pager to keep track of both row and column offset and uses it in all render operations. The reason for this is that we did not have any API to build on; the implementation is completely made from scratch.

This actually gave us a very nice control of how the rendering is done. Unlike the table, rendering can be done incremental which means that we can use timeouts for rendering only portions of the heat map at once. This leads to a much better user experience as the page does not freeze while the JavaScript is working. It also gives feedback to the user as portions of the view can be seen appearing within each timeout.

Further the total size of the canvas can easily be calculated using *column width * number of columns* and *row width * number of row* in the first portion of the refresh call. Because of this the browser does not need to parse HTML and calculate layout more than once for the whole canvas.

Another feature of the canvas makes it perform faster than the table implementation for every test that involves adding rows or columns. There is however a memory payment for increasing this performance. According to the canvas documentation any changes to width or height will reset it, making old data disappear. This required us to do double buffering, where we buffer the previous view in a temporary canvas, and then copy the old bitmap into the new canvas. While the old canvas has not been garbage collected by the JavaScript engine, it will require the memory of the old view and the new view. The performance is boosted by the fact that it is much quicker to copy a bitmap then to run the render method for the whole view over again.

FIGURE 23 - DOUBLE BUFFERING WITH THE CANVAS. YELLOW PART IS BUFFERED, RED PART IS RENDERED

Figure 23 shows an example of what happens when we add rows to the canvas. Since the yellow part has already been created we save its content in a buffer. Then we copy this over into a new canvas and register the offset. Then we start drawing new values from the offset till we reach the bottom.

4.4.4 HEADERS

Both of the headers currently are implemented as cell tables, and the interfaces for them can be seen in the code listings.

```
public interface IHeatmapHorizontalHeader {
	CellTable<HashMap<Integer, ?>> getTable();
	void setPager(HeatmapPager pager);
	void setDatasetIds(int[] datasetIds);
	void setDatasetContributions(HashMap<Integer, Float>
contributions);
	void setDatasetDetailedInformation(DatasetMap datasetMap);
}
```

The horizontal header currently reacts to two user events; *mouseover*- and *clickevents*.

The mouseover event creates a floating display that renders more information about the current dataset. The click event calls the method in the presenter responsible for handling a dataset click event, containing a dataset identifier. This in turn sends the identifier out on the event bus and any handler listening for dataset click events will do their work.

```
public interface IHeatmapVerticalHeader {
		CellTable<GeneMap> getTable();
		void setPager(HeatmapPager pager);
		void setGeneIds(int[] geneIds);
		void setGeneRanks(HashMap<Integer, Float> ranks);
}
```

The cell currently reacts to a *clickevent*. This event calls the method in the presenter for handling gene click events, containing a gene identifier. This sends the event to the presenter which sends it to the event bus and any listening handler.

In practice a future version should implement these headers using a canvas instead. This would make it easier to implement zoom functionality for the heat map.

4.5 DENDROGRAM

In the following sections we describe our implementation of the dendrogram widget. The dendrogram uses the canvas, and we do believe that it very hard to implement it with any other approach.

The code is currently in a test phase, and does not utilize the MVP-pattern or implement any event handlers. The data to load must also be explicitly set in the server constructor.

As with the heat map widget, the dendrogram code needs to provide two interfaces for data transfer. These are called DendrogramService and DendrogramServiceAsync, and they currently only provide one method;

```
public interface DendrogramService extends RemoteService {
      public DendrogramData getDendrogramData();
}
```

We will now describe the data structures, how the server works and what the client does.

4.5.1 DATA STRUCTURES

As we can see in the code above the data service only provides a single method that returns DendrogramData. This class contains a DendrogramNode root and LinkedList<DendrogramNode> leafNodes.

A DendrogramNode contains an id, a correlation and a name. Each node also has three links, which are DendrogramNode left, DendrogramNode right and DendrogramNode parent. It also has x and y coordinates which indicates where it is placed on the canvas surface.

The whole dendrogram can be traversed by starting at the root and moving to the left and right nodes. A leaf node is found when both of the links are null.

4.5.2 SERVER

In this section we describe how the server parses the data files and creates an in memory dendrogram.

To create the data structure the server needs to parse two files; data.CDT and data.GTR.

The CDT file contains all leaf nodes, with their name and ids. We create a HashMap<String, DendrogramNode>. For each entry we find in the CDT file we put it into this map, and use the id as key. From now on we will refer to this map as the *leaf map*. Once the file is parsed we have created a structure which contains all the leaf nodes, but we don't have any relation between them.

This is found in the GTR file. This file contains all the node connections, and each row consists of a node id, the id of its left link, the id of its right link, and its correlation. The correlation is a value for how strongly two nodes are related to each other. We therefore create a new HashMap<String, DendrogramNode>, in which we put all the nodes into, using the id as key. We will from now on refer this map as the *connection map*. We have two maps, one contains every leaf node and the other contains connections between them.

We now create an iterator for the keys in the *connection map*. Then we iterate through all the keys and fetch nodes from the *connection map*. We know following about these nodes;

- If the current node has a left link in the *connection map* we fetch it. We create a link between them by setting current node->left to fetched node, and fetched node->parent to current node.
- If the current node does not have a left link in the *connection map*, it must[15] have a node in the *leaf map.* We then fetch this leaf node, and set the leaf node->left and node->right to null. We also set the leaf node->parent to the current node.
- If the current node has a right link in the *connection map* we fetch it. We create a link between them by setting current node->right to fetched node, and fetched node->parent to current node.
- If the current node does not have a right link in the *connection map*, it must have a node in the *leaf map.* We then fetch this leaf node, and set the leaf node->left and node->right to null. We also set the leaf node->parent to the current node.

After all this is done, we should have created all the connections between the nodes, and the dendrogram is almost complete. Only one part remains, and that is to find the root node. If the creation of the dendrogram was correct, there should be a single node which does not have a parent. We therefore iterate over the keys again, and find this root.

4.5.3 CLIENT

Once the server has created the dendrogram, it is ready for being transferred over to the client. The client can call on the *getDendrogramData()* which is provided in the data service.

Once all the data is transferred, the client is ready to create a canvas for visualizing the dendrogram. As we wrote DendrogramData contains a LinkedList<DendrogramNode> leafNodes. We know that all of the leaf nodes are to be aligned vertically. We can therefore start by placing these nodes into the canvas, using an arbitrary number of pixels between them. We also put the coordinates for its placement on the canvas into each leaf node.

[15] If this is not true, then there is an error in the files we are parsing and the dendrogram cannot be created.

Once all of the leaf nodes are put into the canvas, we need to create the links between them. The DendrogramData also provides link to the root, and this our entry point into the connections. We recursively parse the dendrogram by following the left and right links. We start by moving left, which will in the end give use the topmost leaf node.

We then know that the y coordinate for the parent of two nodes should be placed at (leftNode->yCoordinate + rightNode->yCoordinate) / 2. The x coordinate is based on the parents correlation compared to the correlation of the two child nodes. The current implementation does a quite poor job when it creates the x coordinates, which leads to a dendrogram looking like the ones in figure 13 and 14.

We continue parsing the dendrogram until we have set the x and y coordinates for each node. Once this is done the last part is to draw the lines between all the nodes.

4.5.4 MAKING IT WORK

The main reason that we did not finish the dendrogram was because we observed that the canvas crashed once the dendrogram reached a size threshold. We have later realized that we could have used the *scale* method the canvas provides. By scaling down each pixel, we could have created a dendrogram that fit into the canvas limits.

We are certain that it is possible to create a working implementation which builds on the current solution, but for now it still stands as challenge.

5. WIDGET FUNTIONALITY

In this section we are looking at what functionality the heat map widget implements. We want to identify what we are actually evaluating in section 6.

The following figures are from a search for the gene CTR9 in the yeast datasets with 50 gene and 50 dataset identifiers. The outcome of different data fetch sizes are explained in text. The view size is set to 5 rows and columns, and adding rows or columns will add 5 rows or columns.

5.1 A NEW SEARCH

1. The user opens the webpage containing the heat map widget.
2. The user specifies the gene(s) she wants to search for.
3. The browser renders the heat map.

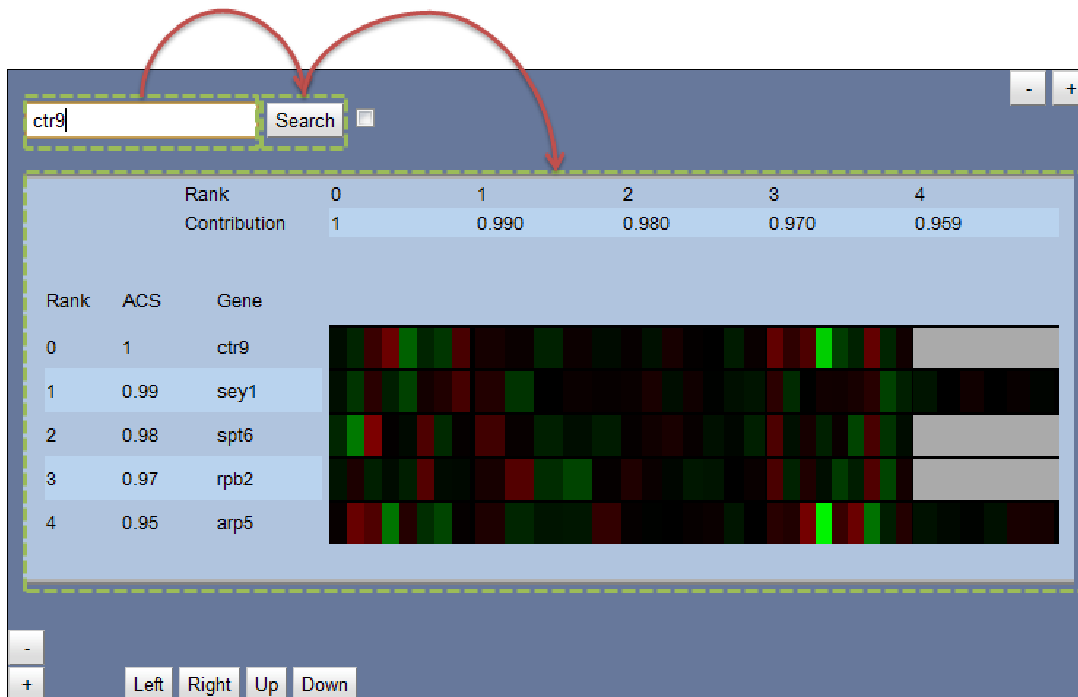


FIGURE 24 - CREATING A NEW SEARCH WITH THE HEAT MAP WIDGET

The client connects to the server and gets 50 genes and 50 datasets identifiers. These identifiers are sent back to the client. The client split these identifiers into blocks of size 5 genes and 5 datasets.

Example: The client requests 50 genes and 50 datasets identifiers. The client gets these and split these identifiers into blocks of 10 genes and 10 datasets. You then have 50 genes * 50 datasets / 10 genes * 10 dataset (block size), which gives us 25 blocks.

The client now inspects how many blocks it needs to generate the view which consists of five rows and five columns. It finds that it only needs to fetch one data block, and

sends the first block of identifiers to the server. The server returns the data, and the client can now generate the view.

5.2 A NEW PAGE

1. The user inspects the data which has arrived and wants to see the next 5 datasets.
2. The user clicks on the "RIGHT" button, which moves the view one page to the right

FIGURE 25 - PAGING HORIZONTALLY WITH THE HEAT MAP WIDGET

The client checks whether or not it has the required data to present the next page. If we follow the previous example where we fetched 10 genes and datasets per call, it would find out that it had the required data and render the data immediately.

If the data was not at the client, it would inspect the identifiers and find the ones for gene 0-5 and dataset 5-10. It would then send a request to the server, and as the data arrived it would generate the view.

5.3 ADDING COLUMNS

1. The user inspects the data which has arrived and wants to see the next 5 datasets.
2. The user clicks on the “+” button in the upper right corner, which adds five columns to the current view.

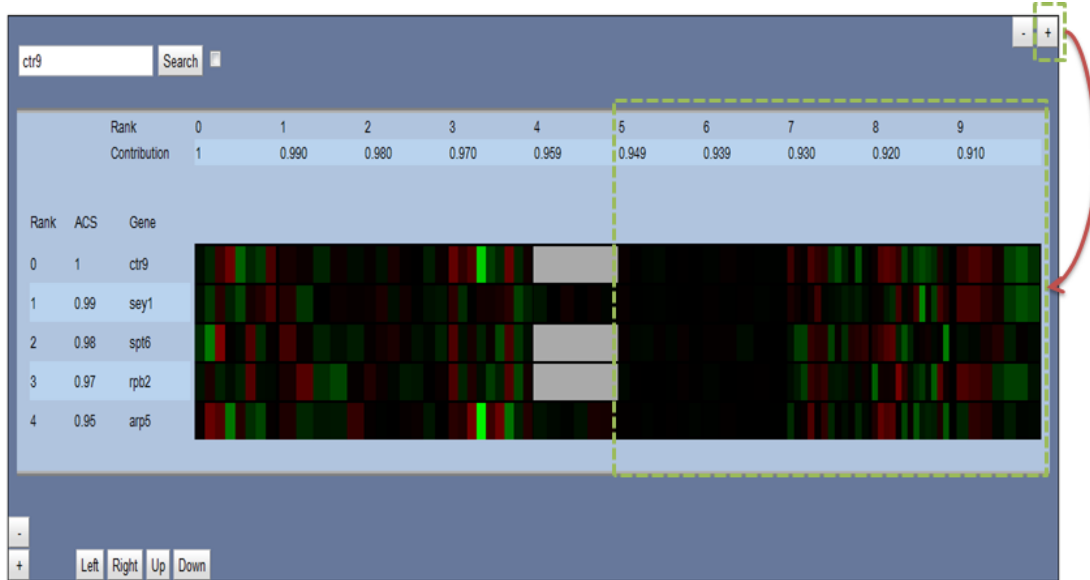


FIGURE 26 - ADDING NEW COLUMNS WITH THE HEAT MAP WIDGET

The client checks whether or not it has the required data to present the next page. If we follow the first example where we fetched 10 genes and datasets per call, it would find out that it had the required data and render the data immediately.

If the data was not at the client, it would inspect the identifiers and find the one for gene 0-5 and dataset 5-10. It would then send a request to the server, and as the data arrived it would add the columns.

5.4 SELECTING GENES AND DATASETS

1. The user wants to highlight the second dataset and second gene because she saw something interesting.
2. The user clicks on the second gene header, and the second dataset header.

FIGURE 27 - HIGHLIGHTING DATASETS AND GENES IN THE HEAT MAP WIDGET

Once the user clicks on the gene header for sey1, the gene click event is invoked. The handler inspects the gene identifier, and sends it out on the event bus. Each view that has registered itself on the event bus with a handler for the gene click event gets the id of this gene. The handler for the heat map is to outline the selected gene, and will do so by setting a white color around the selected gene row.

The dataset click event works in a similar way.

If the user uses paging or adds new rows or columns, the selected genes or datasets are still be highlighted.

1. The user wants detailed information of a cell or dataset.
2. The users hover the cell or dataset

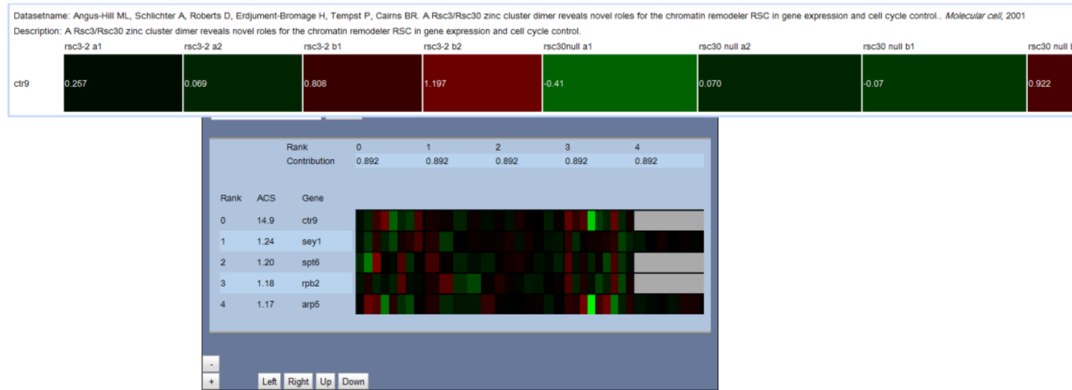


FIGURE 28 - GETTING DETAILED INFORMATION IN THE HEAT MAP CELLS

The widget implements a popup if the user hover a cell in the heat map or a cell in the horizontal header. The outcome can be seen in the figures 28 and 29.

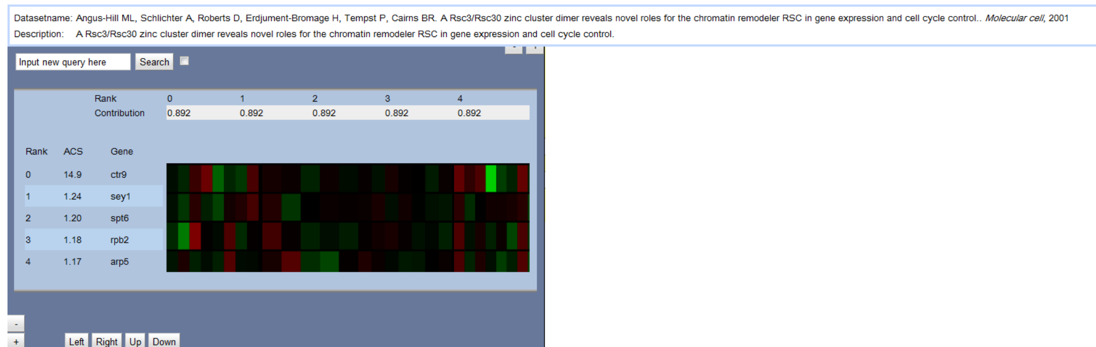


FIGURE 29 - GETTING DETAILED INFORMATION IN THE HORIZONTAL HEADER

6. EVALUATION

In the following section we describe how we have evaluated the heat map widget. We start the evaluation by describing the methodology for the experiments. In the sections after this we look at scalability, maximum view size, data transfer size and time, render time for different view sizes and finally the performance when adding new rows or columns.

6.1 METHOLODGY

The experiments that have been performed have been done with a client computer at the University of Tromsø, while the server has been at Princeton University in USA. The reason is that we get real world latency.

We measured the latency to the server at Princeton to 133ms round trip using the Windows 7 ping utility. This means that any data request has a minimum overhead of approximately 133ms.

The client PC is a relatively new desktop computer with an Intel i5 2500K 3,33GHz quad core processor, 2x4GB DDR3-1337 memory modules and a 1 gigabit network card.

The client is running Windows 7 and tests are performed using Google Chrome. To monitor resource usage the inbuilt *Performance monitor* in Windows 7 has been used. The client is uses a fresh start up of Windows 7, and the only running programs are Chrome and any monitoring tools required.

The server is running an Intel Xeon E5430 2,66GHz quad core processor, 2x16GB DDR2-800 memory modules and an Intel® PRO 1 gigabit network card. The hard drives are 4x Seagate 1,5TB disks.

The server is running CentOS 5.3, and we are running a clean install of Apache Tomcat 6.0.32 as web-server. The server itself is a shared computer meaning that other could be doing tasks at the same time. We did however check if anything was running for every test except the one conducted in section 6.2. The reason for this is that it ran for 2,5 hours.

Evaluating a web application can be a tedious task. With a synchronous application a call to getTime() can be done before and after a method is called, and an exact time can be fetched. In GWT lots of the code is asynchronous, which gives the problem of knowing when an operation is definitively finished. At the same time the browser itself is running two separate engines for JavaScript and rendering, and it is currently not possible to get information from the rendering engine in JavaScript.

To be able to evaluate the application several different tools were used. Timers were placed directly into the JavaScript code to time before and after methods are called. While this does not work at all places, and it does not get hold of rendering information it does give the best timing from a script perspective. Further the Chrome Developer Tools has been used to analyze the latency and size of different block sizes. Last the

inbuilt Performance Monitor in Windows 7 has been used to analyze CPU, memory and bandwidth usage seen for the operating systems perspective.

There is also another limiting factor in heat map widget at the moment, which is the ranking algorithm server. For large comparisons it can use a few seconds. Because of this we have created a real search on the genes BRCA1, BRCA2 and copied the identifiers into a file on the server. This means that DS is not actually involved, and the setup can be seen in figure 30.

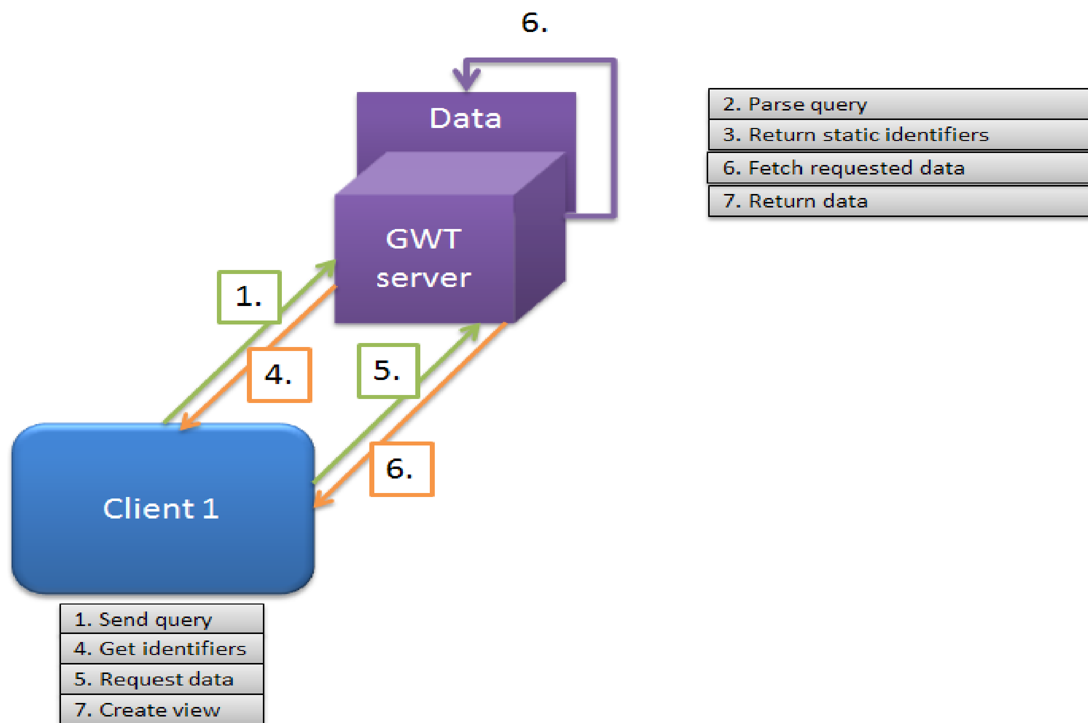


FIGURE 30 - EVALUATION SETUP

Further we created a test suite to test different setups on the view, see figure 31. On this page there are 4 columns, where the green represent data variables, the red represent view variables, the yellow has a number of different automatic tests, and the blue field gets output from JavaScript.

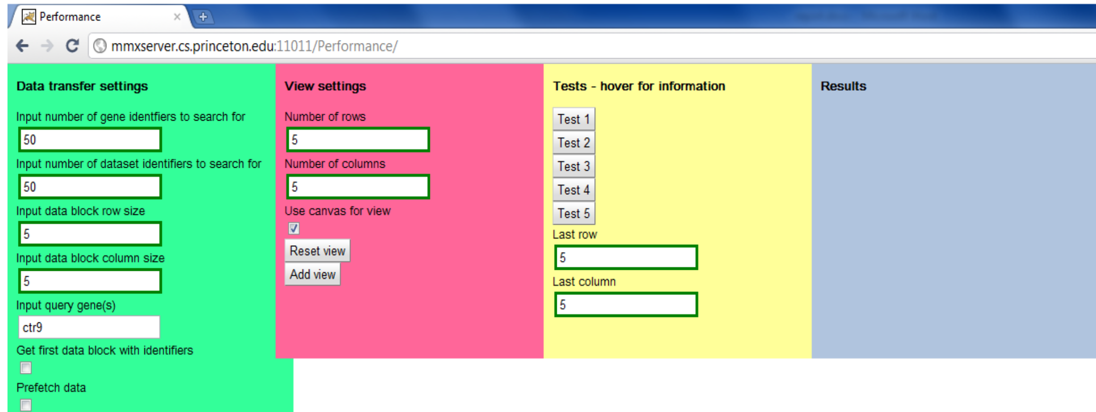


FIGURE 31 - HEAT MAP WIDGET TEST SUITE

The five tests do the following.

1. It fetches the number of identifiers set in the data settings. Then it loads in data it requires to show the first page set in the view size settings. It then pages downwards, while downloading new data, until it reaches the last row. At this point it sets the row offset to 0, and pages right. Then it pages down to the last row again. This continues till it has been through all the identifiers.
2. It adds rows until it reaches the "Last row" parameter set in the test column. If the number of gene identifiers is lower than "Last row" the test does not work properly.
3. It adds columns until it reaches the "Last column" parameter set in the test column. If the number of dataset identifiers is lower than "Last column" the test does not work properly.
4. It pages downwards until it has rendered the row which is set in the "Last row" parameter in the test column. If the number of gene identifiers is lower than "Last row" the test hangs in an infinite loop.
5. Not implemented.

None of the earlier version of SpellWeb has been able to render all values in the yeast datasets, which consists of around 6000 genes and 112 datasets. The new implementation is able to render all the values in these datasets without removing any data from the client's data structures. To give it a challenge we therefore evaluate if it can render all values in the human datasets which consists of around 24.000 genes and 700 datasets. The total amount of data is around 900MBs.

We should mention that we do in fact not show all the values. There are actually 24328 genes and 712 datasets. However there is a potential bug at the last row or column, and to make sure the experiment did not crash because of this we used rounded numbers.

The view size is set to 25 genes and 25 datasets, while fetching blocks of size 50 genes and 25 datasets. The test starts at the top with gene one and dataset one, and moves down to gene 24000. Once it reaches the bottom, it moves up to gene one again, and starts at dataset 25. This same routine runs until it reaches gene 24000 in dataset 700.

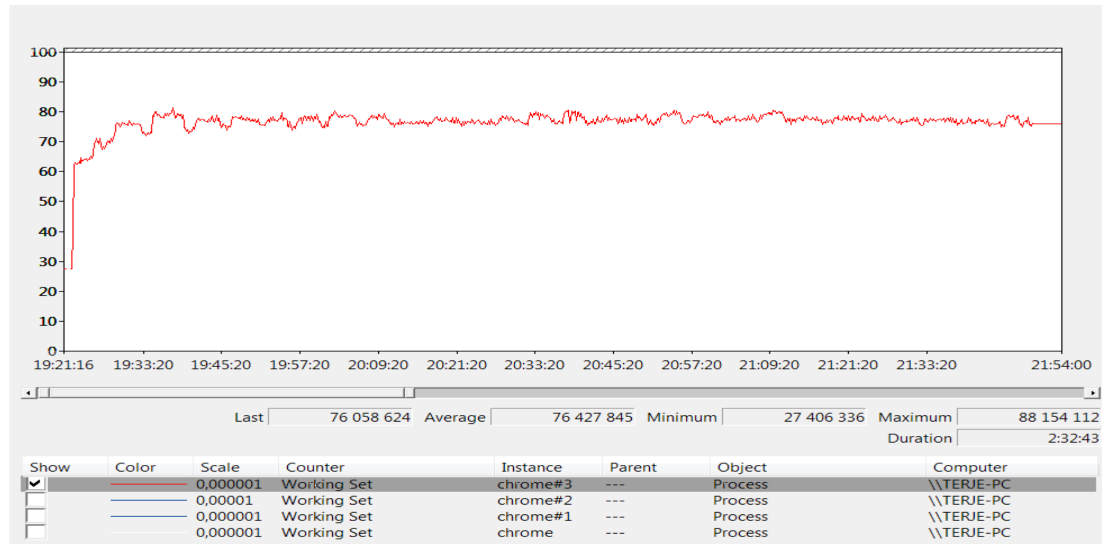


FIGURE 32 - MEMORY USAGE WHILE INSPECTING 16.800.000 HEAT MAP CELLS. THE Y-AXIS IS IN MEGABYTES

Figure 32 shows the average memory usage for Chrome was approximately 76MB of memory, with a maximum of 88MB. In practice this means that we were able to inspect 16.800.000 cells of data.

The reason this works is because we are removing the data for *currentpage-2* each time a new page is rendered. Further the LinkedHashMap is set to have a maximum of 400 entries. In practice this limits the view to a maximum of 400 columns.

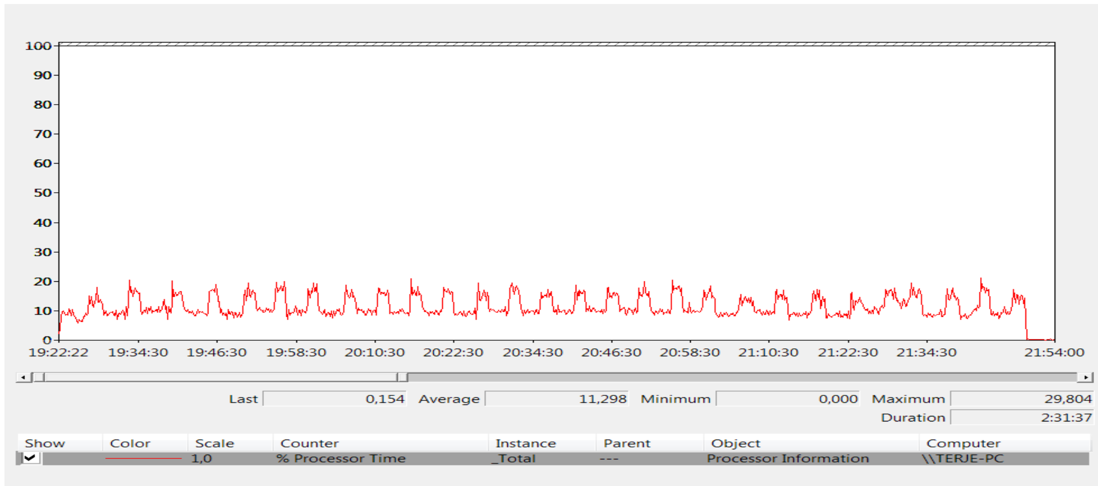


FIGURE 33 - CPU USAGE WHILE INSPECTING 16.800.000 HEAT MAP CELLS. THE Y-AXIS SHOWS PERCENTAGE OF TOTAL CPU USAGE

Figure 33 shows that CPU usage is shaped as a wave which at first sight was puzzling. It turns out that the reason for this is the amount of data each page has. The ranking algorithm places the most important genes at the top, and this is also the area with most data. In the figure 34 the network interface usage can be inspected.

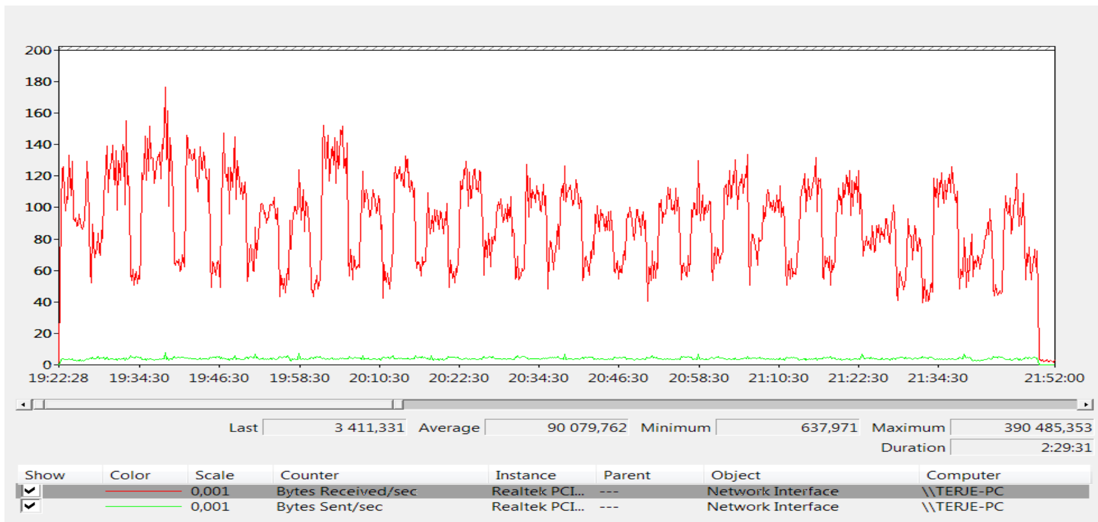


FIGURE 34 - NETWORK USAGE WHILE INSPECTING 16.800.000 HEAT MAP CELLS. THE Y-AXIS SHOWS KILOBYTES PER SECOND

What we can see is that when data transfer is low, the CPU usage is high and vice versa. The reason for this is that when there are low amounts of data, it arrives much faster. Because of this the widget has to draw a new page very often. When the data is high, transfer time is higher. While the widget is waiting for data, the CPU is mostly idle which gives a lower average CPU usage.

There are currently a few things that will stop the application from being able to show an unlimited amount of data.

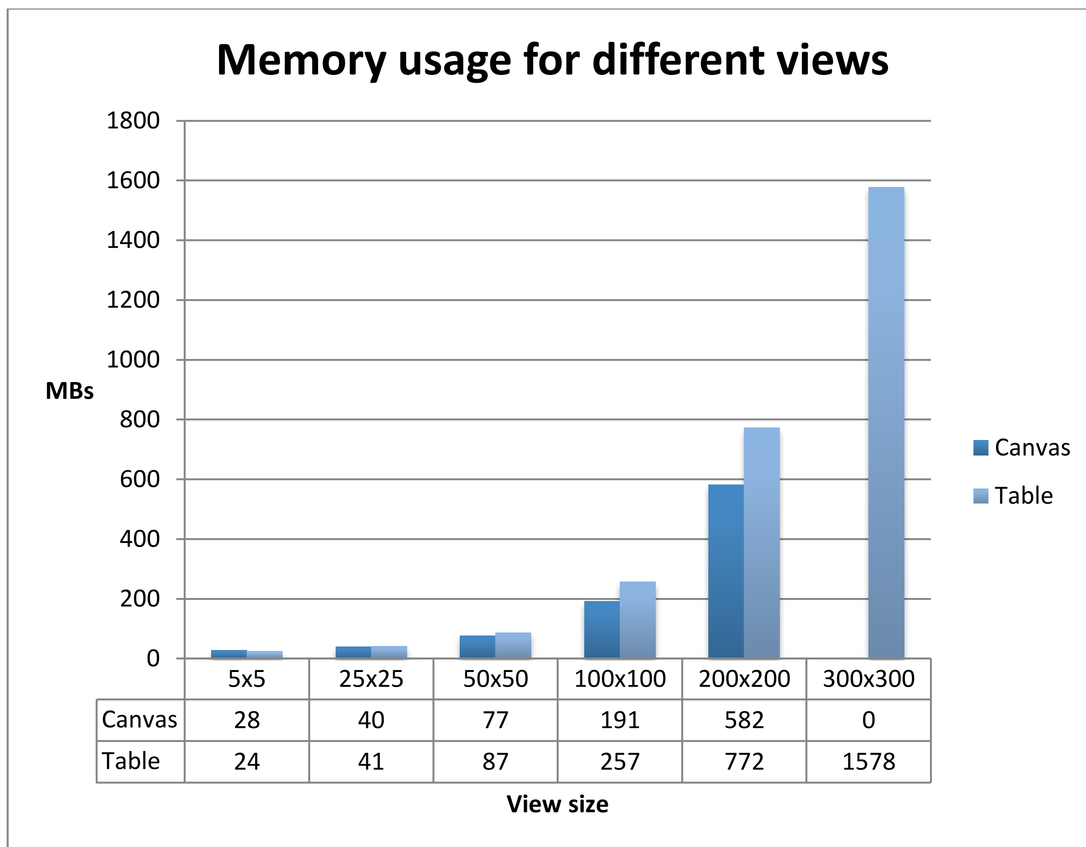
The identifiers that are acquired for each search are never removed. Since each of these normally takes around 4 bytes of data, the view application cannot handle it if these where to become too high. However, having one million gene and dataset identifiers should still only take around 2MB of memory.

Further the application keeps one LinkedHashMap for each row. In the situation where data is removed, the hash map itself is not removed as this would disturb the order of the list maintaining the order of the rows. Assuming each of these empty hash maps required 1KB of memory, the human genome would require 25KB of memory which means this will quite likely not become a large problem in the near future.

6.3 MAXIMUM VIEW SIZE

In this section we evaluate what the maximum view size for the heat map widget is. In theory this should be limited by the amount of memory that is required to show a view.

The settings are quite simplified for this test. We are running NxN identifiers, NxN block size and NxN view size. Because the identifiers set the limit for paging, this means that no more than one page could have been shown with these settings.



	5x5	25x25	50x50	100x100	200x200	300x300
Canvas	28	40	77	191	582	0
Table	24	41	87	257	772	1578

FIGURE 35 - MEMORY USAGE FOR DIFFERENT VIEWS

As the figure shows the canvas uses less memory for every size except the first one. The reason for this is the DOM size. The number of heap objects for 200 rows and 200 columns is approximately 1.1 million objects for the canvas implementation. The table based implementation has approximately 5.5 million objects which are around 80% more.

We can also see that the maximum size for a table is around 300x300, which uses around 1.6GBs of memory. In all test that we have performed Chrome shuts down the application once it uses approximately 2GB of memory.

The canvas stops working at a certain size. Two things can happen: the canvas goes blank, or Chrome crashes. We have not been able to identify the reason for this, but

since the canvas implementation is still under development it will hopefully be fixed in later versions. For a 200 rows and 200 columns view the canvas is 30000x9000 pixels large and we assume that this is not a regular use case.

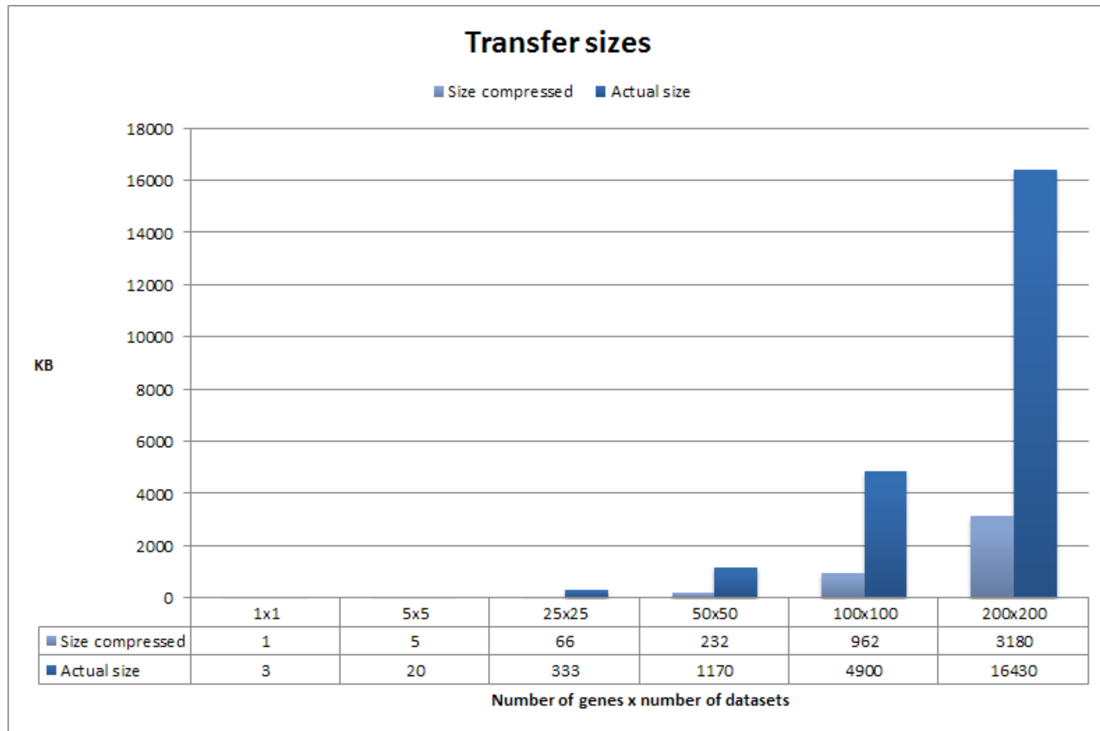
FIGURE 36 - JAVASCRIPT HEAP INSPECTION FOR A VIEW OF 200 COLUMNS AND 200 ROWS USING A CANVAS

FIGURE 37 - JAVASCRIPT HEAP INSPECTION FOR A VIEW OF 200 COLUMNS AND 200 ROWS USING A HTML TABLE

If we consider that the application is running the exact same code except for the view, we can see that it has around 4.4 million objects only for representing the DOM. This would be HTML <tr>, <td> and <div> elements that the canvas version does not need.

6.4 DATA TRANSFER SIZE AND TIME

In this section we identify how much data is transferred for different block sizes, and how long this takes to transfer. This is important to identify so that we can recommend what the settings should be for the widget.



	1x1	5x5	25x25	50x50	100x100	200x200
Size compressed	1	5	66	232	962	3180
Actual size	3	20	333	1170	4900	16430

FIGURE 38 - TRANSFER SIZE FOR DIFFERENT DATA BLOCK SIZES

The first thing we can notice in this figure is the big gap between actual size and compressed size. The ratio goes from 66% compression for the smallest block to around 80% for the largest block. This is a very significant number and shows that the floating point values transferred has a good potential for compression.

Currently no native compression API exist for client usage in GWT, however there exist at least one external module[16] which could be used to compress the data in browser memory. However as the creator writes; "*The code works, but LZMA was not designed to run in a web browser as Javascript. Processing at higher compression levels (which use lots of memory) and/or with huge files may cause your browser to explode.*"

We conclude that we would like browser vendors to implement a standard compression algorithm that could be used.

[16] http://code.google.com/p/gwt-lzma/ - Last visited 31th May 2011.

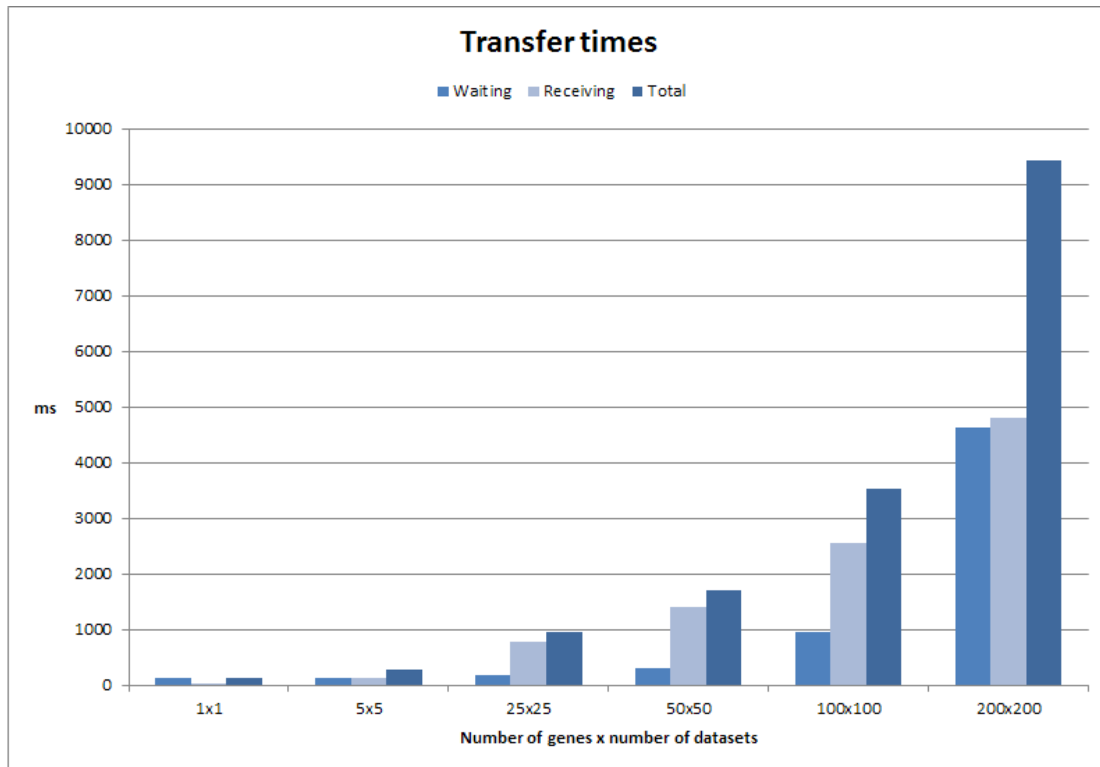


FIGURE 39 - TRANSFER TIME FOR DIFFERENT DATA BLOCK SIZES

The figure above shows the time it takes from sending till the data has arrived at the client. The sending has an average of 0 milliseconds for all data sizes, so it is not shown in the figure.

As we can see the times for sizes 25 genes 25 datasets and lower are all under one second in transfer time. According to usability expert Jacob Nielsen [N93] "*1.0 second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data*"

Further we can notice that as the amount waiting does not grow in the same way as the receiving. Once the data size grows, the time to retrieve the values on the server gets a more significant time than it does for low values, while the receiving time increases non-linearly.

The reason for this is likely to be the way the TCP [WIKI4] is implemented. For small data amounts it is likely that the slow-start will stop the bandwidth to getting to its max speed, while with larger sizes the speed will increase.

6.5 VIEW SIZE AND RENDER TIME

In this section we are looking at how long time it takes to render a view. We use Chrome Developer Tools and look at when the method for generating the view starts. We also look at what the browser is doing after it is done with the JavaScript portion of getting the view on the screen.

In the figure 40 we can see a comparison of how well the canvas and table implementation uses for different screen sizes.

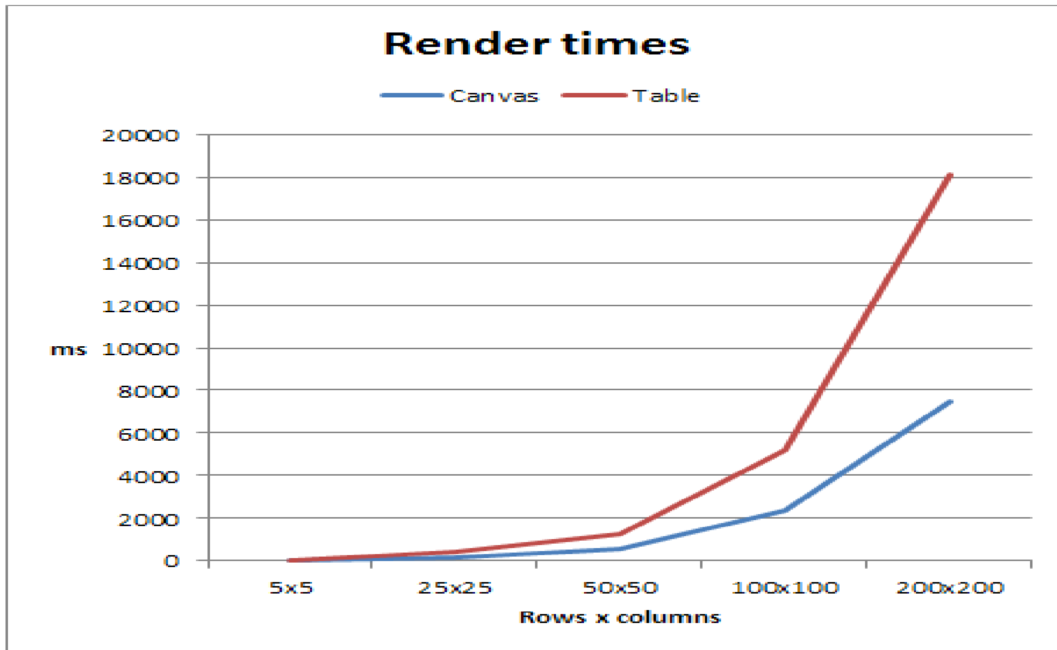
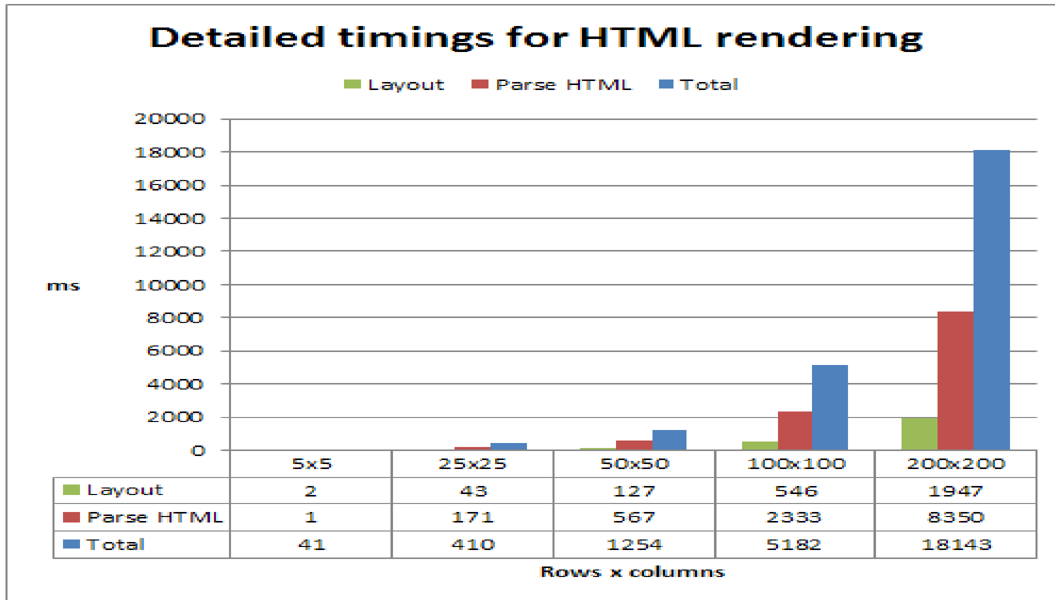


FIGURE 40 - CANVAS AND HTML TABLE TOTAL RENDERTIME

In all cases the canvas is faster and we can that the gap between them increases along with the view size. The reason for this is that the browser has to do more work after the script portion is done with the table implementation. In the next figure we can see what the browser does after it has completed using JavaScript to generate the HTML for the view.



	5x5	25x25	50x50	100x100	200x200
Layout	2	43	127	546	1947
Parse HTML	1	171	567	2333	8350
Total	41	410	1254	5182	18143

FIGURE 41 - DETAILED RENDER TIME FOR THE HTML TABLE

Once the HTML is ready the browser needs to parse it. We can see that this parsing takes almost 8 seconds for a 200 rows and 200 columns view, which is more than the total time for the canvas. Further it also has to do layout after the HTML parsing is done, adding another 2 seconds.

The canvas on the other hand uses less than 10 milliseconds for parsing and approximately 1 millisecond for layout in every test. The reason for this we could see in figure 36 and 37. The table implementation has 4,4 million more elements to parse for a view of this size, and according to [LDL08] parsing the DOM is slow.

6.6 ADDING ROWS OR COLUMNS

In this section we assume that all the data is actually at the client. This is done by setting the data block size to a value that is greater than the final size of the view. We also have set a JavaScript timeout of 100 milliseconds between each call for adding new rows or columns to not freeze the browser in a JavaScript loop.

Figure 42 presents a test where we initially have 5 rows and 25 columns. We then add 5 rows, until we stop at 200 rows and 25 columns.

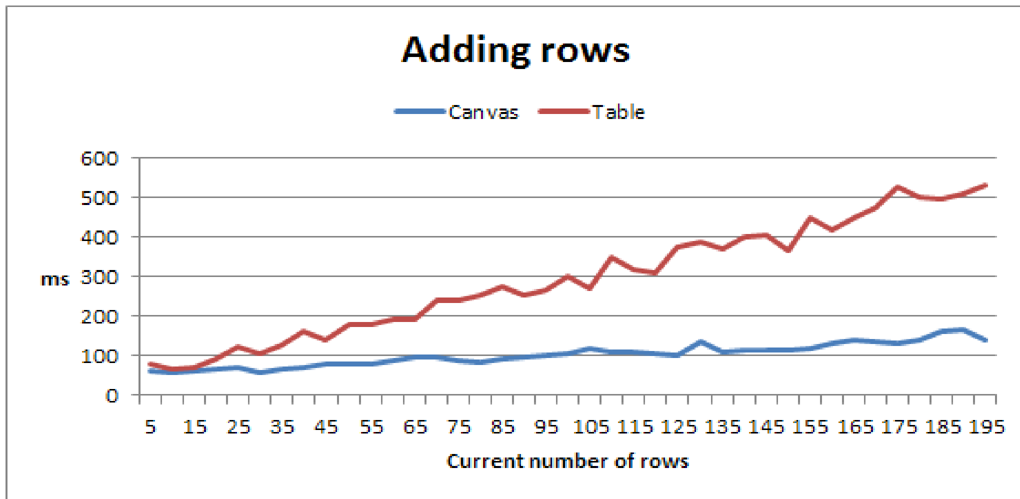


FIGURE 42 - JAVASCRIPT TIMINGS FOR ADDING ROWS

As we can see the JavaScript timings shows that the canvas performs better at every step. The reason is likely to be that we are buffering the previous visualization, see section 4.4.3. However this illustration does not nearly give the complete time required to add rows. If we summarize the total time spent running JavaScript the canvas uses 4,3 seconds and the table uses 11,8 seconds. We have also inspected the time used in Chrome Developer Tools and the actual total time is 8,5 seconds for the canvas and 45,24 seconds for the table. The reason for this is that the table requires the browser to parse HTML and set the coordinates for each element when we add new rows. The canvas does not require this extra job.

Figure 43 is a test where we initially have 5 columns and 25 rows. We then add 5 columns, until we stop at 200 columns and 25 rows.

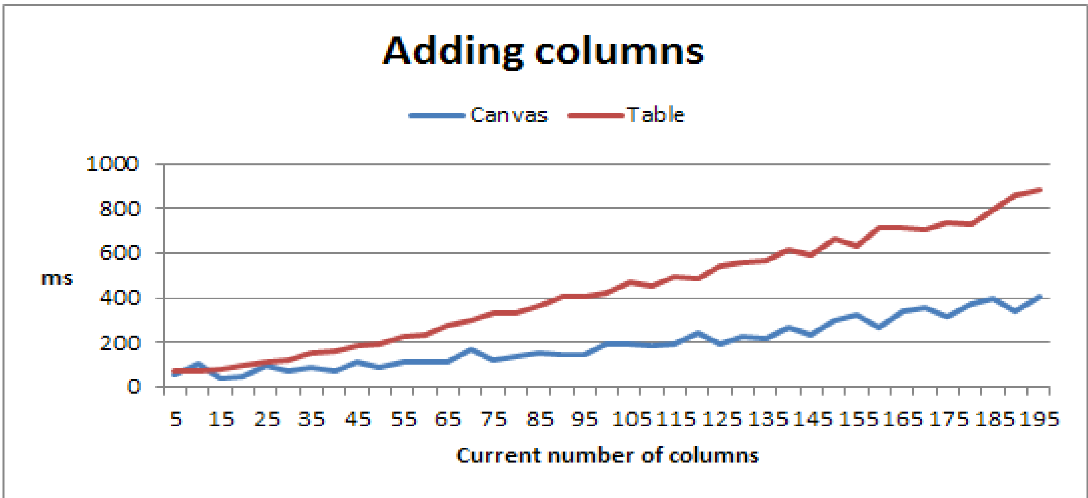


FIGURE 43 - JAVASCRIPT TIMINGS FOR ADDING COLUMNS

As we can see the canvas performs better in this context too. The total time spent in JavaScript is 7,5 seconds for the canvas and 16,7 seconds for the table. If we look at the numbers from Chrome Developer Tools the total time for the canvas is 12,6 seconds, while the table version uses 55,1 seconds.

7. DISCUSSION

7.1 TABLE OR CANVAS

In most of the test that we have performed both the table and canvas has worked fairly well. Still we conclude that the widgets like the heat map should start using the new functionality of the browsers if possible.

The reasons are several and we shall look at some of them here. First of all the canvas implementation of the widget performs better than the table implementation in almost every step in the evaluation. The only place where the canvas does not work better is when we were testing the maximum size of for a view. It uses less memory, but either stops drawing or crash the browser.

Another reason for using the canvas is that the developer has full control of the rendering. For example it is not possible to implement incremental rendering with the CellTable without actually changing its code. This means that nothing will be displayed on the page before the whole table is ready for layout and painting.

We also need to implement a particular data interface for the CellTable, which in this widget does not work as well as we hoped. The CellTable required a ListDataProvider to work. The problem with using a list as a common data source is that we need to keep track of order. If two views where to share the same list, they need an internal mechanism to keep track of where a row is actually placed in its own view.

Another approach that could have been used would be to use a hash map to also keep order of rows. Since the gene identifiers could have been used as a key, they would keep order themselves. In practice this would enable the widgets to utilize the same data source, saving a lot of memory when using several views.

In figure 44 we can see 200 genes and 100 datasets with less than 1920*1080 pixels. This was easily implemented by using the scale method of the canvas. Doing this dynamically using the table based version would likely require quite some amount of work.

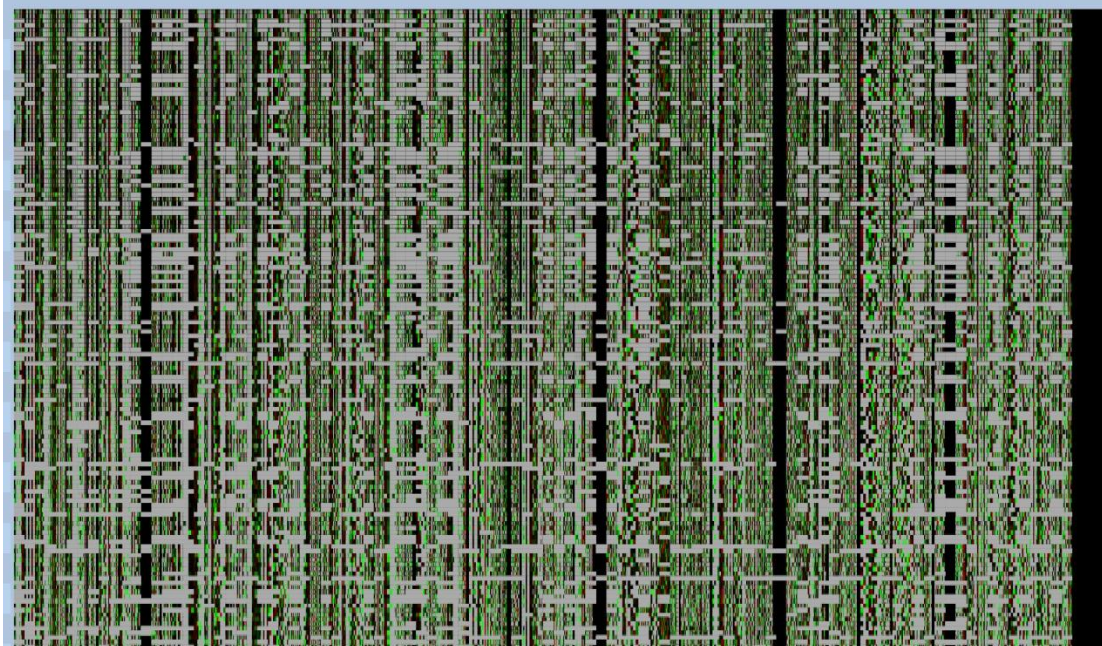


FIGURE 44 - SHOWING 200 GENES AND 100 DATASETS WITH 1920X1200 PIXELS USING THE CANVAS

7.2 RECOMMENDED SETTINGS

Seen from a user perspective it is important that the application is fast and responds quickly to events. It is also important that the JavaScript engine does not get the chance to freeze the render engine for more than a maximum of some hundred milliseconds. In the following text we describe what we would recommend for a user that is watching a single heat map widget.

First of all we recommend always using the canvas if possible. It has better performance in all aspects and it can do incremental rendering. This means that even for large views it will not freeze the browser.

We propose to use a maximum data fetch size of 25 genes and 25 datasets. This has shown to give a transfer time of around one second, which is a fair load time for a web page. We also recommend enabling pre-fetching of data, which will load data for adjacent pages after the current page has been rendered. If we can assume that a user looks at a page for a minimum of two seconds, it is likely that every other page is loaded before the user tries to show them. This is because the four other pages can be requested in parallel, and if the server is not busy they should all arrive within this timeframe. If the user pages before this she will have to wait for the data to arrive before the screen renders itself.

When it comes to the view we purpose that the user should use any size that would give them approximately one screen size for a single page. A view of 25 columns and 25 rows has a size of 2720x1032 pixels and takes around 420 milliseconds to render. Given that the user can control the view size by adding or removing rows or columns, a standard view size of around 10 columns (1220 pixels) and 25 rows (1032 pixels) would be a nice fit between performance and size.

When it comes to a practical maximum view size we recommend it to be around 100 genes and 100 datasets. Our evaluation has shown that the time required for transfer data is around 3,5 seconds per page at this size which is quite a long time to wait. However if we enable pre-fetching of data, and we assume that a user looks at a page for at least 5 seconds we can assume that the data transfer will not be a big problem. The render time for a view of this size is around 2,5 seconds, but as the canvas does incremental rendering the top values can be inspected before the whole page is created.

The reason we call this a practical maximum view size is because the canvas has a tendency to stop working when the size is larger than this. We have not been able to debug Chrome when this happens, so we are not able to identify the reason for this fault. A view of 100 rows and 100 columns has a size of 10220x3282 pixels, or around 14 screens for a 1920x1200 pixels display. Hopefully this is enough for most users at the moment.

8. CONCLUSSION

In this project a highly scalable widget library for biological visualizations has been started. We have fully implemented a heat map widget, and demonstrated that it can currently visualize 24000 genes and 700 datasets with an average memory usage of 70MB in Chrome. The total amount of raw data is approximately 900MB.

Further we have looked at two different implementations for the heat map visualization, one which uses a styled HTML table and the other using a HTML5 canvas. This has shown that the HTML table approach works well for smaller visualizations with less than 50 rows and columns, but as these numbers increase its performance degrades significantly compared to the canvas version. The impact is highest when a user changes the view size. The canvas can buffer its old content and only render the new values. The table has to reparse the HTML tree and do a new layout which is a time consuming process.

We conclude that developers trying to solve similar problems should use a canvas or WebGL based solution. It has better performance and it is much more versatile compared to using a HTML table.

After the work in this project we believe that it is possible to use web applications for visualizing many types of biological data. If the data can be split into smaller portions, like the heat map, we believe that browsers and JavaScript has become fast enough to handle generation of visualizations using the canvas.

We also believe that it is a good idea to porting applications as early as possible. Even though the HTML5 standard is not expected to be finished before 2020, much of its functionality is already implemented into browsers. Waiting for completion will only delay what could possibly be a fruitful online ecosystem for genomic research.

9. FUTURE WORK

In this section we outline directions for future work on our heat map widget and web applications in general.

The first thing any widget needs is a solid server side code. We have been focusing on the client side, but web applications consist of two sides. In our evaluation we disconnected from the ranking server (Distributed Spell). The reason for this was that it gave an overhead of up to a few seconds for getting hold of gene and dataset identifiers. To speed up this it needs to implement caching. If a gene has been searched for once, the identifiers for this should be saved in memory or to a file.

With the new HTML5 file API we could cache identifiers on the client as well. The current quota that can be storage locally for each web page is around 5MBs. If we assume that each identifier can be saved with 4 bytes, we would need 24000*4 + 700*4 = 109000 bytes, or 109KBs to save identifiers for one search. With 5MB we could then store identifiers for around 45 complete searches locally.

We have also been experimenting with a *zoom* for our heat map visualization, and as the canvas implements a *scale* method this was very easy. To truly implement this we need to convert the headers which are still using a table approach.

We also propose that the HTML table based heat map view should be dropped. Even though every browser does not support the canvas, we can display a message for users to get a supported browser. If this was done, we could implement an important change, which is to share data amongst several heat map widgets. See sections 4.2.4 and 7.2 for more information about this problem.

We believe we are starting to see that web applications is going in a direction where it can start implementing functionality that was earlier only possible using desktop applications. For now a browser uses context switches on single thread for page rendering and JavaScript. If we had the ability to use web workers[17], which lets the browser spawn new threads for JavaScript, we could generate several visualizations in parallel or generate portions of a large visualization in parallel. This is done in the JavaScript Mandelbrot implementation created by Google.

New APIs are also coming for local storage which will let a developer save data at the client. Currently this is set to 5MB, but Google suggests letting the browser ask a user if it would like to save more than this. If the user agrees an application could save more data, and enable a truly offline "web application".

Implementations using hardware acceleration is also coming, which will enable the GPU to take over work for the canvas and WebGL .This will enable higher frame rates, and games like AngryBirds[18] are currently working flawlessly in Chrome since this has been implemented.

[17] http://dev.w3.org/html5/workers/
[18] http.//chome.angrybirds.com/

BUGS

We have written several thousand lines of code in this project, and we are sure that there are several bugs there. In this section we list the ones we know about.

If the identifiers search size is set to a number that does not divide to zero with the view size it can lead to a bug when trying to display the page that has the last columns and rows.

Searches on human datasets currently do not always show the correct color values. The reason is that we were not familiar enough with the actual data and thought the missing color values were missing data.

There is a bug when selecting genes and columns after the first page in the heat map using the canvas implementation.

REFERENCES

[K11] "On the Future of Genomics Data, Science", Vol. 331 no. 6018 pp. 728-729. S. D. Kahn. February 2011

[F05] "Kreft i Norge 2002", Frøydis Langmark, http://www.kreftregisteret.no/no/Generelt/Nyheter/Kreft-i-Norge-2002-/, 2005 – Last visited 31th May 2011.

[OGG10] "Visualizing biological data—now and in the future", Seán I O'Donoghue, Anne-Claude Gavin, Nils Gehlenborg, David S Goodsel, Jean-Karim Hériché, Cydney B Nielsen, Chris North, Arthur J Olson, James B Procter, David W Shattuck, Thomas Walter & Bang Wong. March 2010.

[BWL06] "Systems Support for Remote Visualization of Genomics Applications over Wide Area Networks", Lars Ailo Bongo, Grant Wallace, Tore Larsen, Kai Li, Olga Troyanskaya, 2006

[WIKI1] http://en.wikipedia.org/wiki/Heat_map - Last visited 31th May 2011.

[WIKI2] http://en.wikipedia.org/wiki/Biological_data_visualization - Last visited 31th May 2011.

[PWS08] "A survey of visualization tools for biological network analysis", Georgios A Pavlopoulos, Anna-Lynn Wegener and Reinhard Schneider. November 2008.

[NCD10] "Visualizing genomes: techniques and challenges", Cydney B Nielsen, Michael Cantor, Inna Dubchak, David Gordon & Ting Wang, 2010

[WIKI3] http://en.wikipedia.org/wiki/Genome_browser - Last visited 31th May 2011.

[S04] "Java Treeview—extensible visualization of microarray data", Alok J. Saldanha. May 2004. http://jtreeview.sourceforge.net/

[HWD07], Viewing the Larger Context of Genomic Data through Horizontal Integration, Matthew Hibbs, Grant Wallace, Maitreya Dunham, Kai Li, Olga Troyanskaya. July 2007. http://function.princeton.edu/HIDRA/

[J10] "Interactive Data Exploration of Very Large Biological Datasets using Web Applications", Terje Andrè Johansen. December 2010.

[LCC00] "Building and Using A Scalable Display Wall System", Kai Li, Han Chen, Yuqun Chen, Douglas W. Clark, Perry Cook, Stefanos Damianakis, Georg Essl, Adam Finkelstein, Thomas Funkhouser, Timothy Housel, Allison Klein, Zhiyan Liu, Emil Praun, Rudrajit Samanta, Ben Shedd, Jaswinder Pal Singh, George Tzanetakis, and Jiannan Zheng. August 2000.

[T09] "Behind the scenes of modern web browsers", Tali Garsiel. http://taligarsiel.com/Projects/howbrowserswork1.htm - Last visited 31th May 2011.

[W11] "50 Performance Tricks to Make Your HTML5 Web Sites Faster", Jason Weber. April 2011. http://channel9.msdn.com/Events/MIX/MIX11/HTM01 - Last visited 31th May 2011.

[W3C05] "Document Object Model", http://www.w3.org/DOM/ . Last update January 2005 - Last visited 31th May 2011.

[GOOGLE09], "Introducing Closure Tools", November 2009. http://googlecode.blogspot.com/2009/11/introducing-closure-tools.html - Last visited 31th May 2011.

[L11] "GWT + HTML5: A web developers dream!", John Labanca, May 2011. http://www.google.com/events/io/2011/sessions/gwt-html5-a-web-developers-dream.html - Last visited 31th May 2011.

[K11] "Script#: Compiling C# to JavaScript using Visual Studio", Nikhil Kothari. April 2011. http://channel9.msdn.com/events/MIX/MIX11/HTM16 - Last visited 31th May 2011.

[J08] "SmartGWT 1.0 Released!" Sanjiv Jivan. November 2008. http://www.jroller.com/sjivan/entry/smartgwt_1_0_released - Last visited 31th May 2011. Source code at: http://code.google.com/p/smartgwt/ - Last visited 31th May 2011.

[C09] "Progressively Enhance AJAX Applications with Google Web Toolkit and GQuery", Ray Cromwell. May 2009. http://www.google.com/events/io/2009/sessions/ProgressivelyEnhanceAjaxApps.html - Last visisted 31th May 2011. Source at: http://code.google.com/p/gwtquery/wiki/GettingStarted - Last visited 31th May 2011.

[R09] "Google Web Toolkit Architecture: Best Practices For Architecting Your GWT App", Ray Ryan. April 2009. http://www.google.com/events/io/2009/sessions/GoogleWebToolkitBestPractices.html - Last visited 31th May 2011.

[P96] "MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java", Mike Portel. 1996.

[MEH10] "Crom: FasterWeb Browsing Using Speculative Execution", James Mickens, Jeremy Elson, Jon Howell and Jay Lorch. April 2010.

[M08] "Color Blindness: More Prevalent Among Males", Geoffrey Motgomery, 2008. http://www.hhmi.org/senses/b130.html - Last visisted 31th May 2011.

[N93] "Usability Engineering", Jakob Nielsen. 1993. http://www.useit.com/papers/responsetime.html – Last visited 31th May 2011.

[WIKI4] http://en.wikipedia.org/wiki/Transmission_Control_Protocol#Congestion_control – Last visited 31th May 2011.