University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

Selection of Best Data Storage Technology for Report Charting at 360incentives

360incentives

420 Green Street

Whitby, Ontario, Canada

Prepared by

Justin Matthew Palumbo

ID 20424969

userid: jmpalumb

2B Computer Engineering

7 May 2014

7 Dewbourne Place

Whitby, ON

L1R 1X9

7 May 2014

Manoj Sachdev, Chair

Electrical and Computer Engineering

University of Waterloo

Waterloo, ON

N2L 3G1

Dear Sir:

The attached report, titled "Selection of Best Data Storage Technology for Report Charting at 360incentives”, was prepared as my 3A Work Report for the University of Waterloo, in fulfilment of the course WKRPT 301. The purpose of this report is to explain the best method for storing the data used to generate business report charts as determined by my testing and evaluation. Available web storage technologies are evaluated based on their compatibility with common browsers as well as their loading time performances.

360incentives is a software-as-a-service company that provides a platform for the management of manufacturers’ incentive programs. As such, 360incentives develops a website that is used by these clients to create, monitor and manage rebate programs that are offered to their dealerships, sales associates and consumers.

Along with the responsibility of writing and maintaining automated testing scripts to test the website, I was part of a three-person team of co-op students given the task of replacing the charting library previously used by 360incentives with a newer, flexible library. As such, I was involved in designing, programming and configuring a new charting interface. This report was written for the consideration of my team: Tyler Mason, Wenjing Chen and myself, as well as for our supervisor, Adam Goucher, to aid in the decision-making process for improvements regarding the website’s charting functionalities—specifically with respect to how data are stored and loaded.

I would like to acknowledge the assistance of Zoë Waller who aided me with the proofreading and formatting of this report. I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Justin Matthew Palumbo

ID 20424969

## Contributions

At 360incentives, I worked with the software development team, consisting of approximately 15 full-time employees (the team size fluctuated over the course of my work term) and three co-op students. Within this development team, I was a member of a sub-group known as the delivery team, which was responsible for testing and general quality assurance of all the software products developed by 360incentives. This team was also responsible for supporting the software by addressing user complaints and performing minor code fixes.

My role within the delivery team was to write and maintain automated scripts that would test the functionality of the many facets of the 360incentives platform. This included testing of the ‘core’ website (that provided most of the business features and management tools for clients as well as for 360incentives employees), a ‘hub’ website (that was similar to the core website, but customized for specific use by one of the larger clients), and custom claims entry websites that would be used by consumers and sales associates. The testing scripts were written in Python and used the Selenium web driver libraries.

In addition to this testing work, we, the co-op student team, were given our own project to manage and deliver with minimal input from our supervisors. This project was largely given to us for the purpose of developing our management, planning and programming skills, while also producing functional code and improvements to the website that would be used by 360incentives and its clients. The nature of this project was to replace existing libraries that were used to generate business report charts with a newer library that was more flexible and generated more visually appealing graphs.

The replacement of the charting library thus required significant planning to design a new user interface that users would use to generate charts, and substantial consideration as to how the new library would be ‘wired-up’ into the system to input and output data appropriately. As such, I was involved in the planning stages of this project where my fellow Waterloo co-op student, Wenjing Chen and I thoroughly investigated how charts were generated in the older system. We designed around the pre-existing infrastructure to provide charts with the new library. Wenjing and I divided into two separate areas of specialization on this project—he understood and primarily designed the backend C# code while I primarily focused on the user-interface and front-end JavaScript.

This report outlines and analyzes one of the design challenges we faced during this project. Throughout my time at 360incentives, I observed how slowly the website’s pages would load and often became frustrated when attempting to accomplish even basic tasks (such as login into the website). As such, the co-op team placed considerable emphasis on the idea that our charts should attempt to mitigate some of the loading times by optimizing the process by which our charts would store and load data. Therefore, this report is meant to outline the analysis of possible data storage solutions and determine the best solution to implement in order to reduce loading times.

By writing this report and performing the supporting analysis, I have not only taken action to speed up the 360incentives core website, but I have improved my own problem solving and analytical skills. I have gained experience in selecting the best option from viable solutions, benefited from practical experience with designing and performing experiments, and had exposure to performing timing analysis in the work place (which is far different and less ideal or ‘pre-configured’ than timing analyses performed in academic assignments).

My work for 360incentives has helped the company achieve the goal of delivering high quality software to its clients. One of the values embraced by this company is the notion of delivering an ‘unbelievable experience’. Not only have the changes to the chart interface improved user work-flow and enhanced the aesthetics of the page, but the efforts taken by the team of co-op students I worked with to optimize chart loading have reduced loading times for the dashboard page.

As discussed more thoroughly in this report (Section 1.1), optimizing web pages to deliver content quickly is important to preserve user satisfaction and to uphold a positive reputation for the website. If the clients are satisfied with 360incentives’ service then they will be more likely to renew their contracts or refer the service to other companies. It is hoped that the changes of which I have been a part with the implementation of the new charting system will help 360incentives to maintain and continue to build a positive standing with its clients.

## Summary

The purpose of this report is to determine the most appropriate method of storing and loading data that is used to generate charts for the 360incentives website. The scope of this report is limited only to technologies that are currently available and commonly implemented in web browsers (including cookies, browser web storage, indexed databases, web SQL and database servers). This report focuses on using these potential solutions to store data of a predetermined size and structure. Any changes to the data would therefore be out of scope of this report.

Section 1 of this report outlines the reasons for replacing the old charting library and the flexibility provided by the new library. This section also discusses the general design problem and describes a design goal of having web pages load within a two-second to three‑second interval, as well as the negative psychological impacts associated with longer loading times. Section 2 presents the available storage technologies and discusses the design constraints relating to data size, browser compatibility and data persistence. This section concludes by stating that browser local storage and server side databases are the only viable options for storing data. Section 3 outlines the structure of tests that were designed to compare the performance of both of these storage techniques, while Section 4 analyses the data collected from these experiments.

This report concludes several major points. Loading charts from local storage is significantly faster than loading charts from a database and is the only option that could meet a goal of loading within a two-second to three-second duration. Loading charts simultaneously takes more time as more charts are requested, but is more efficient if considered on a time-per-chart basis. Using separate physical machines for a database and web server is slower than using one machine to provide both services. Finally, Google chrome is the fastest browser for loading charts while Firefox is the slowest.

As such, it is recommended that local storage be used where possible to cache data and to prevent unnecessary access to the database. Additionally, the database and web server should be hosted on a single machine if possible, and users should use Chrome to access the website.

## Table of Contents

[Contributions iii](#_Toc387262929)

[Summary v](#_Toc387262930)

[Table of Contents vi](#_Toc387262931)

[List of Figures vii](#_Toc387262932)

[List of Tables viii](#_Toc387262933)

[1. Introduction 1](#_Toc387262934)

[1.1. Highcharts Design Problem 2](#_Toc387262935)

[2. Data Storage Methods 3](#_Toc387262936)

[2.1. Available Solutions 3](#_Toc387262937)

[2.2. Browser Compatibility 5](#_Toc387262938)

[2.3. Requirements and Type/Structure of Data to be Stored 6](#_Toc387262939)

[2.4. Selection of Suitable Methods 9](#_Toc387262940)

[3. Timing Testing of Local Storage and Database Information 10](#_Toc387262941)

[3.1. Structure of Tests 10](#_Toc387262942)

[3.2. Network Configurations 11](#_Toc387262943)

[3.3. Selected Browsers 14](#_Toc387262944)

[3.4. Expected Results 14](#_Toc387262945)

[4. Analysis of Testing Results 15](#_Toc387262946)

[4.1. Effect of Increasing Number of Charts 15](#_Toc387262947)

[4.2. Local Storage versus Database Storage 17](#_Toc387262948)

[4.3. Effect of Browser 19](#_Toc387262949)

[5. Conclusions 21](#_Toc387262950)

[6. Recommendations 22](#_Toc387262951)

[Glossary 24](#_Toc387262952)

[References 25](#_Toc387262953)

[Appendix A: Example of JSON Query String 26](#_Toc387262954)

[Appendix B: Example of JSON Series Data String 27](#_Toc387262955)

[Appendix C: Timing Test Source Code JavaScript 28](#_Toc387262956)

[Appendix D: Chart Generation Timing Values 32](#_Toc387262957)

## List of Figures

[Figure 3‑1. Visualization of Network Configuration 1. Client connects directly to a single server through the local network. 11](#_Toc386568151)

[Figure 3‑2. Client connect to web server through local network. This web server then communicates with a database server before replying to the client. 12](#_Toc386568152)

[Figure 3‑3. Configuration of the network to connect the client to the web server via VPN (Note that the two local networks are the same physical network.) In this configuration, the web server and database server are the same physical machine. 13](#_Toc386568153)

[Figure 3‑4. Configuration of the network to connect the client to the web server via VPN (Note that the two local networks are the same physical network.) In this configuration, the web server connects to the database server before replying to the client. 13](#_Toc386568154)

[Figure 4‑1. Box and whisker plots for all charting configurations and trial sizes. 15](#_Toc386568155)

[Figure 4‑2. Box and whisker plots for all charting configurations. Values are normalized such that times are given as milliseconds per chart 16](#_Toc386568156)

[Figure 4‑3. Box plots for all data storage configurations, showing differences in measured ranges 17](#_Toc386568157)

## List of Tables

[Table 2‑1. Comparison of client-side storage methods 5](#_Toc387262958)

[Table 2‑2. Minimum browser version support for client-side storage methods 6](#_Toc387262959)

[Table 4‑1. Browsers producing maximum and minimum timing values 19](#_Toc387262960)

[Table D‑1. Averages, maximum and minimum values and medians for recorded timing values broken down by storage type/configuration, the number of charts generated per test and by browser 32](#_Toc387262961)

## Introduction

360incentives (360) is a software-as-a-service company that provides its clients with a web platform for creating and managing incentive programs. Such programs include monetary bonuses for retailers and sales associates as well as rebates for consumers. In the past, many companies would process claims for their incentive programs by hand—a task that is both labour intensive and prone to mistakes. The web-based solution provided by 360 attempts to automate and simplify the managing of incentives programs while adding services that would not be available to clients through manual claim processing. Features such as automated fraud detection and sales breakdown reports are made possible through 360’s digitized processing.

To provide these kinds of useful data to the end users, the 360 software uses charting software libraries to generate and display data in a readable fashion. Currently, users have a customized dashboard on their home page that displays all the charts that users have created. At the time of this report’s writing, an overhaul project was in progress to replace the existing charting library, .netCHARTING with the newer, more flexible software package Highcharts JS.

This change was necessary to make 360’s software more flexible as .netCHARTING uses Microsoft’s web parts controls, which require a Windows Server environment to run. Charts are generated by .netCHARTING library on the server and then sent to the user’s browser. While the rest of 360’s core codebase is written in ASP.NET (a Microsoft developed web framework), only the .netCHARTING web parts depend on using a Windows Server. By means of the Highcharts JavaScript library, charts can be generated on the client side and the rest of the code can be compiled and deployed on a Linux server using the Mono platform (which allows some .NET applications to execute on non-windows environments).

Additional factors for the switch to Highcharts include improved aesthetics and greater customizability for charts (support for changing chart properties such as series colours, line formatting, etc.). Highcharts also allows for interactive charts where users can hover over elements and view a data point’s value or, in some cases, click into elements of a graph and view data in a drill-down fashion. Furthermore, Highcharts can easily be configured to support exporting charts to JPEG, SVG, PDF and PNG file formats.

While the conversion from .netCHARTING to Highcharts has some technical challenges, 360 will greatly benefit from conversion to Highcharts by providing more user-friendly software to clients.

### Highcharts Design Problem

One of 360incentives’ fundamental corporate goals is to deliver an “unbelievable user experience” to its clients. In order for this goal to be met a number of technical challenges arise for the development of the dashboard web page on the 360 website. This page shows users relevant, customized data through charts and graphs. To make this dashboard “unbelievable”, many aspects must be considered including user-interface layout and other features including customizability options and organizational tools within the page. One of the most significant attributes to consider is the loading time of the web page. Since this page is the first page the user encounters after logging into the system, loading time should be minimal to show the user that the login process has been successful. As a software-as-a-service company, 360incentives must also attempt to demonstrate that it is capable of providing web solutions that are quick and responsive.

Psychological impact of slow-loading websites have been studied; research indicates that slow-loading websites negatively impact business. A study from content delivery network company Akamai Technologies has found that 47% of consumers who use retail websites expect a page to load within two seconds of a request. Additionally, 40% of users will wait no more than three seconds for a slow page before abandoning it [1]. While the users of 360incentives’ dashboard page are not general public consumers, it is considered that these users could develop similar feelings of impatience while waiting for the dashboard page to load. Such emotions could negatively impact the reputation of 360incentives with existing clients and potentially hinder client retention (or even client base expansion in the most extreme case).

As such, loading time is a crucial consideration for this dashboard project and the scope of this report is centred on that aspect of design. With respect to the conversion of dashboard charts from .netCHARTING to Highcharts, the only factors affecting page loading time that can be modified are those regarding how user chart data are stored and loaded. (The dashboard page contains additional widgets that display other information, separate from the charts. The current modifications to the dashboard are limited only to the conversion of charting from .netCHARTING to Highcharts). This report examines the different methods of storing data, their usefulness in the context of the 360incentives website, and their effects on loading time.

## Data Storage Methods

This section outlines the types of data storage techniques that are available for use in website development and considers the effectiveness of each technique for storing user chart data. Section 2.1 outlines and briefly describes the solutions for modern web data storage at the time of this report’s writing. Section 2.2 outlines the compatibility of these solutions across browsers. Section 2.3 lists the specific requirements for the data being stored and transferred, while Section 2.4 selects suitable methods of storage based on the requirements.

### Available Solutions

Presently, websites are structured such that data can be stored either by a user’s browser (on the client side) or by the web server (on the server side). Typically, server-side data are stored in databases that can be optimized to provide quick access of information from thousands or millions of records and can support many simultaneous user connections or sessions. This report does not consider the design of the database as the 360incentives website already uses a pre-configured MySQL instance and any significant changes to its structure or efficiency are beyond the intended scope of the project to convert user chart generation from .netCHARTING to Highcharts. As such, the practically available method of storing data on the server side is limited to the addition of a table into the existing database. This table can be configured to store user IDs and other relevant information relating to the chart’s configuration, as well as the actual numerical data that the chart displays.

On the client side, several solutions are available for holding data including:

* Cookies,
* Indexed DB API,
* Web Storage, and
* Web SQL API

Each of these options was designed to be used in different contexts and is suitable for different scenarios. Each method has its own advantages and disadvantages.

Cookies and web storage both use simple key/value pairs to store string data in plaintext format. Each set of data has a unique identifier (the key) which is used to access its associated string data. Cookies have a 4 kilobyte data limit [2] whereas the web storage has a limit of 5 megabytes of data per domain origin. (The total data size of all the key/value pairs for a given website is limited to 5 megabytes) [3]. Cookies can be configured to expire at specific dates and times or expire after the user’s session has ended. Web storage has two options, session storage and local storage. These options are used to store data that expire after the termination of a session or data that are indefinitely persistent, respectively [2].

Whereas cookies and web storage are inherently part of worldwide web core specifications, the Web SQL and Indexed DB APIs are separate standards that have been independently implemented to allow for client-side storage of data that is structured more complexly than strings [2]. Both of these APIs utilize a database that is created and stored by the web browser on the user’s computer. Data stored by Web SQL can be queried with SQL syntax; however, this feature is not supported by Indexed DB [4].

The following, Table 2-1 outlines the strengths and weaknesses of web storage, Indexed DB and Web SQL [5].

Table 2‑1. Comparison of client-side storage methods

|  |  |  |
| --- | --- | --- |
| **Storage Type** | **Advantages** | **Disadvantages** |
| Web Storage | * Near universal browser support * Simple to implement and use | * Poor performance when searching, storing and manipulating large or complex data. (Web Storage typically runs synchronously.) * Need to serialize and de-serialize structured data |
| Indexed DB | * Runs asynchronously, providing good performance * Relatively small learning curve | * Complex API * Harder to implement code |
| Web SQL | * Mobile browser support * Runs asynchronously, providing good performance * Rigid data structures ensure data integrity | * Steep learning curve * Deprecated, being phased out of desktop browsers * Rigid data structures diminish programming flexibility |

### Browser Compatibility

360incentives’ current software platform is expected to run on most common desktop browsers. Both 360incentives and many of its clients primarily use Microsoft Windows‑based computers; however, the diversity of computer administration policies that these companies impose creates a challenge for web support. Some companies allow their employees to install and use their choice of browser while other businesses may restrict web access to particular versions of white listed browsers (usually Internet Explorer). Specifically, one of 360Incentive’s largest clients requires the application to work on Internet Explorer 8. Conversely, users of the website from within 360incentives also access the site using current versions of Google Chrome, Mozilla Firefox or Internet Explorer (current version 11). Thus, the browser requirements for the dashboard page are Internet Explorer version 8 and greater, Chrome (current) and Firefox (current). Any storage solution should also be bound to this requirement

Using data from [www.canisuse.com](http://www.canisuse.com) (a reference website for web programmers) [6], Table 2-2 shows the minimum versions of each browser that support each storage method discussed in Section 2.1. Each cell is colour-coded such that green cells indicate that the version meets the browser requirements while red cells indicate they are not acceptable.

Table 2‑2. Minimum browser version support for client-side storage methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Cookies** | **Web Storage** | **Indexed DB** | **Web SQL** |
| Internet Explorer | All versions | Version 8+ | Version 10+ | Not supported |
| Firefox | All versions | Version 2+ | Version 4 + | Not Supported |
| Chrome | All versions | Version4 + | Version 11+ | Version 4 + |

By observation of this table, it is clear that only cookies or web storage meet the current requirements for browser support. If, however, 360incentives’ clients upgrade their policies to allow employees to use more recent versions of Internet Explorer, Indexed DB could be viable for future re-designs of the dashboard page.

Server-side database storage is inherently compatible with all browsers, as the only requirement for it to function is the ability to run JavaScript asynchronously to retrieve data (using asynchronous JavaScript and XML, or AJAX). The World Wide Web Consortium (W3C) has published the underlying mechanisms for accomplishing this (the XMLHttpRequest API) as a standard that web browsers are expected to include in some form [7]. Additionally, other JavaScript libraries can be used to simplify AJAX calls into easy-to-implement methods that can reduce development time. One such library, jQuery, includes an easy mechanism for AJAX calls and is actively supported for current releases of Chrome, Firefox and all versions of Internet Explorer since version 6[8].

### Requirements and Type/Structure of Data to be Stored

In order to deliver the desired flexibility and functionality for the dashboard page, a number of different types of data must be stored. The new Highcharts dashboard will allow users to freely position, resize and stack charts (order by layers). Additionally, the user will be able to see when the information displayed on the charts was last refreshed (i.e., pulled from the database). As such, the following information must be stored for each chart:

* The number of pixels from the top boundary of the interface container to the top left corner of the chart
* The number of pixels from the left boundary of the interface container to the top left corner of the chart
* The width of the chart in number of pixels
* The height of the chart in number of pixels
* The layer index of the chart
* The chart’s identification number

Each of these values is an integer that would be reasonably constrained by the limits of using 2 bytes to represent it. (None of the values needs to be larger than about 32,000.) As such, this information would ideally only require 12 bytes of data per chart. Assuming that the user would require no more than 10 charts simultaneously, this amount of data is both reasonable to store locally or in a server database; however, some of the client-side storage techniques store their values as strings (where each character requires one or two bytes of storage depending on the encoding scheme used by the browser). As such, if each value requires no more than five digits, an upper bound of the amount of data required would be 60 bytes per chart or about 600 bytes for 10 charts. While this is less efficient than storing the data in a more appropriate format, 600 bytes per use is still a reasonable amount to store on either server side or client side.

In addition to this configuration and layout data, the information regarding each chart’s content must be stored in some fashion. There are two different types of data that can be stored in order to save what the chart displays. The first option is to store the numerical series data that is interpreted by the Highcharts library to create the graphs. The second option is to save the user’s inputted parameters, which can then be used to re-query the database to get the actual data to pass to the Highcharts library. (The information displayed by the charts is generated and returned by a number of stored procedures that collect data from several tables that hold statistics regarding business operations for each client.) These two types of data are optimally used depending on the types of storage used.

If client-side storage is used, storing the query strings has no practical value since any timing benefit from loading from client-side storage is lost as a connection to the database to retrieve the numerical chart data is still required. Any implementation that uses client-side storage should be keeping a local copy of series data so that charts can be quickly rendered on the webpage without having to wait for information to be re-downloaded.

Conversely, if each user’s chart information is stored in a new table in the database, saving the numerical data for each chart would result in duplicate blocks of information that do not need to be stored. Storage of unnecessary data is a waste of hard disk space that can also impact performance of the database. Since retrieving information from the database already involves a sacrifice in timing, storing the query string and using those parameters to re-query and download the numerical information is acceptable.

Both of these data types take on a similar structure. Queries and chart data are both stored as strings, formatted in JavaScript Object Notation (JSON). The use of JSON allows the data to be easily serialized (converted to strings for the purpose of storage and transmission) or de-serialized (converted to an object with properties that are easily manipulated). The size of these strings is fairly dynamic, as the number of properties of each JSON object and the data contained by each property are variable. It has been found, however, through experimentation while performing preliminary development of the new dashboard page, that the JSON query strings will be approximately 600 bytes in length. Conversely, the largest series data JSON string encountered was about 15,000 bytes. Examples of both query and series data JSON strings are included in appendices A and B respectively.

If each user is expected to have 10 charts displayed on the dashboard as a reasonable maximum, then about 6,000 bytes (5.9 kilobytes) of space would be require to keep the query strings in a database table or 150,000 bytes (146.5 kilobytes) of space would be needed to store the series data in client-side storage.

### Selection of Suitable Methods

Based on the browser requirements described in Section 2.2, only three storage solutions are viable for use on the 360 dashboard page. Only cookies, web storage and database storage are backward-compatible with enough browsers to provide adequate support for all of 360incentive’s clients.

When the size of data to be stored is considered, however, only two of the aforementioned solutions are practical. As discussed in Section 2.3, the potential size of chart information to be stored on the client side can reach up to 15,000 bytes; however, the maximum size of a cookie is 4,096 bytes, meaning that the implementation of cookies would necessitate a system of splitting data over multiple cookies. This is not a practical solution when web storage can easily store all required data without any extra effort to deal with overflow.

Thus, web storage and sever database storage are the only potential candidates for storing chart information. Ultimately, a database must be used in order to allow users to access their charts when switching between computers and browsers or after a deletion of browser temporary files and history. Nevertheless, using web storage to cache chart data could improve performance by reducing waiting and loading times. Instead of the dashboard page always requesting data from the database, information can be loaded from a locally maintained cache that is updated to mirror information from the server on a regular basis (i.e., when a user creates, modifies or deletes a chart). With a properly designed system, loading from the database would only be necessary in a few situations:

* When a user makes a modification in one browser and switches to another,
* When a user makes a modification in one machine and switches to another, and
* When new chart data is requested (i.e., a new chart is added or a chart is modified).

Sections 3 and 4 of this report investigate the timing difference between loading a chart from the database versus loading a chart from web storage (using local storage) to determine if there is a significant benefit to warrant development of a local caching system for chart data.

## Timing Testing of Local Storage and Database Information

This section of the report describes the testing methods used to investigate the timing differences between loading user chart data from local storage and querying the database to retrieve chart data to pass to the Highcharts library for chart generation. A simple HTML page was created with JavaScript to measure the amount of time needed to generate charts under varying circumstances. Section 3.1 and 3.2 describe the structure of these tests and the network configurations under which they were executed. Section 3.3 briefly mentions the browsers used for testing, while Section 3.4 describes the expected results of the tests.

### Structure of Tests

To measure the amount of time needed to load data and generate charts, a relatively simple testing web page was created. This testing page included 10 standard HTML division tags (in which the HighCharts library would generate charts), a simple interface for saving JSON query or data strings into local storage, a selector for the number of charts to generate per trial and links to start the testing using local storage or the database.

Three types of business reports were randomly selected for testing, each generating data varying in size. The corresponding JSON query strings and data strings for all three reports were saved into the browser’s local storage prior to starting the tests.

To run a test, the user would first select the number of charts to generate per test run from a drop-down selector. This variable was included in the tests to determine whether there was any improved efficiency in chart generation by creating one chart at a time versus multiple charts. The user could select to generate one, five or 10 charts in each test run. The value of 10 was estimated to be reasonable maximum number of charts a user would have saved at any one time.

After selecting the number of charts, the testing page’s JavaScript code would then run 20 trials, generating the selected number of charts in each trial. Start and end times for each trial were recorded using the window.performance.now() method, which returns the number of milliseconds since the web page was first loaded. Subtracting the start value from the end value would return the duration of a trial in milliseconds. Each chart generated was a random selection from the three pre-loaded business reports. The random selection processes was performed before the start time was recorded for each trial as to exclude the time needed to generate random numbers and select charts from the measured chart loading and creation times.

Certain portions of the JavaScript code run asynchronously and would attempt to execute code for a subsequent trial before completion of a current trial, thus custom events were used to control code flow. This way, only after the correct number of charts desired per trial was generated would the next trial be started.

In the case of local storage testing, the timing for a trial was started before the JavaScript would attempt to retrieve chart data of the first chart in that trial from local storage (using the localStorage.getItem method). End times were recorded after passing the retrieved data for the last chart into the HighCharts chart creation method. For database testing, however, all the query strings were pre-loaded into global variables and trial start times were recorded just before the first request to the database for that trial run. This was done to exclude the time to load data from local storage from the timing measurements. Similarly to the local storage testing, the end times for database testing were recorded after the data from the last chart in a trial was passed to the HighCharts library.

The source code for these tests is available in appendix C.

### Network Configurations

To determine the optimal network topology of the database and web server, the database version of the tests described in Section 3.1 was run four different times per browser, with the client computer connected to the web server in one of four different configurations. Specifically, these tests were intended to discover the impact of hosting the database on its own dedicated machine, separate from the web server versus having both the web server application and database running on one computer.

The first of the tested configurations was a simple connection of the client directly to a server hosting both the database and web content through the local network. This configuration is depicted in Figure 3-1.

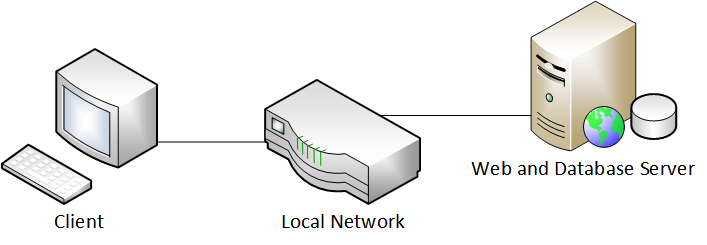


Figure 3‑1. Visualization of Network Configuration 1. Client connects directly to a single server through the local network.

Similarly to configuration one, the second configuration connected the client to the web server through the local network, however the database server was hosted on a separate server (also within the local network). After receiving a request from the client, the web server would then communicate with the database server before sending data back to the client. This layout is depicted in Figure 3-2.

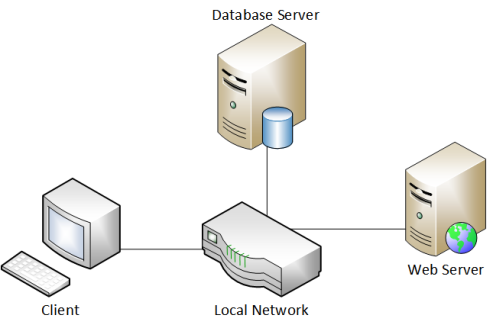


Figure 3‑2. Client connect to web server through local network. This web server then communicates with a database server before replying to the client.

While testing with these first two configurations could show relative differences in using one combined server versus two dedicated servers, the timing results would be inconsistent with realistic results for production since users connect to the 360incentives through the Internet rather than a local network. To simulate an Internet connect, two further network configurations were used. The network was set up so that the web server could be accessed publicly on the Internet and the client computer was configured to send its traffic through a free VPN service called Hotspot Shield (that effectively acted as a proxy service). Use of this VPN connection allowed the client to connect to the web server by first leaving the local network, then passing through a VPN server in the United States and finally returning back to the original local network and to the web server. The United States VPN server was selected in the setup of the Hotspot Shield software on the client computer. This location of the proxy server was chosen to simulate Internet activity coming from North America, as most of 360Incentive’s clients are based in the US and Canada. (The local network that connected the web server, database server and client was located in Whitby, Ontario, Canada where 360incentives is headquartered.)

The two configurations using this VPN service were similar to the first two configurations, only differing by the addition of the VPN service as a middle point between the client and the web server. The first VPN network configuration (third in total) used had the database and web content hosted on the same server. This is depicted in Figure 3-3, while Figure 3-4 shows the second VPN configuration (fourth overall connection) in which the database and web servers were separate machines.

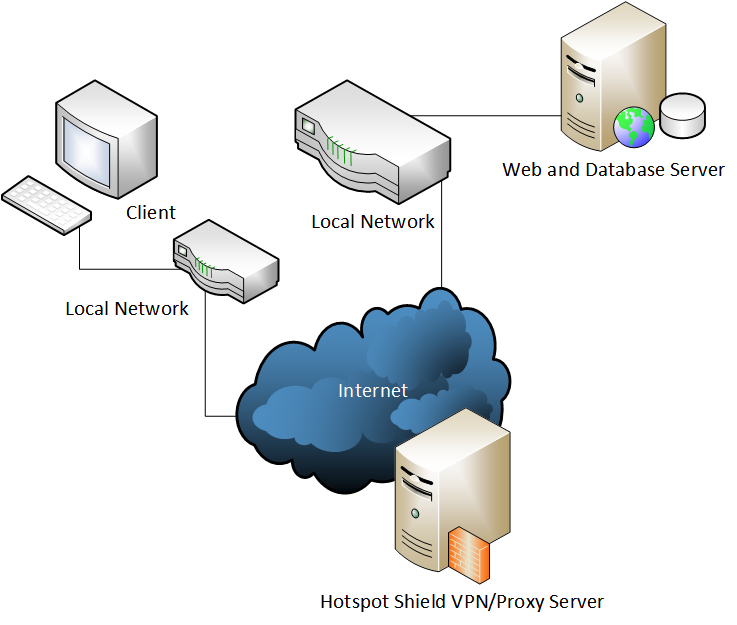


Figure 3‑3. Configuration of the network to connect the client to the web server via VPN (Note that the two local networks are the same physical network.) In this configuration, the web server and database server are the same physical machine.

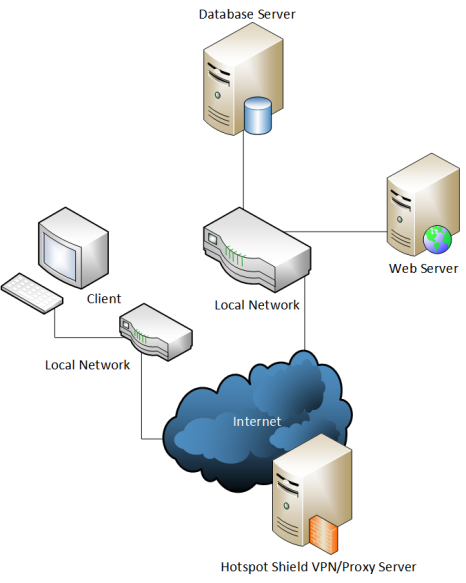


Figure 3‑4. Configuration of the network to connect the client to the web server via VPN (Note that the two local networks are the same physical network.) In this configuration, the web server connects to the database server before replying to the client.

### Selected Browsers

Testing was performed on three different browsers, Internet Explorer, Google Chrome and Mozilla Firefox. These browsers were selected because of the browser compatibility requirements outlined in Section 2.2. (The majority of users of the 360incentives website service will be using one of these browsers). The versions of the browsers tested were all the most up-to-date versions available for public download. While Section 2.2 describes the need to support Internet Explorer as far back as version 8, some users will also be using the current version of the software. For simplicity, only the latest version, Internet Explorer 11 was used for testing. (The computer used as the testing client was pre-installed with Windows 8.1, which comes with this version of Internet Explorer. Downgrading Internet Explorer to version 8 in Windows 8.1 is impossible, and using a virtual machine could potentially skew results by adding unnecessary lag.)

For each browser, tests of generating one, five, and 10 charts were run, each for data coming from local storage and from the database in each of the four network configurations. Thus, a total of 15 tests were executed per browser. Each test also ran 20 trials, meaning that 300 times were recorded for each browser and 900 timing values were recorded in total.

### Expected Results

The tests described in this section were designed to determine the time differences of loading charts from local storage versus loading charts from a remote database. It was expected that when ranked from least amount of time to greatest amount of time needed, the tested configurations would appear in the following order:

1. Loading from local storage
2. Loading from a single combined web server and database server over local network
3. Loading from a web server that connects to a separate database over local network
4. Loading from a single combined web server and database server over simulated Internet (proxy)
5. Loading from a web server that connects to a separate database over simulated Internet (proxy)

Furthermore, the tests were run to determine which of these configurations would fall within the 2 to 3-second limit of acceptable loading time, as discussed in Section 1.1 of this report. Any configurations that take longer than this amount of time would not be considered user-friendly, given that long loading times can lead to frustration and dissatisfaction with the web service.

## Analysis of Testing Results

This section examines the data collected from the timing analysis tests described in Section 3 of the report. Section 4.1 analyses the effects of increasing the number of charts per trial, Section 4.2 examines the effect of local storage versus database storage, Section 4.3 examines the effect of network configurations and Section 4.4 determines any effects caused by using a browser.

### Effect of Increasing Number of Charts

To determine the effect of changing the number of charts per trial, the collected data were grouped by testing configuration for analysis. The ranges of timing values for each test were visualized by creating box and whisker plots, where the colour division in the box represents the median data value, the ends of the box represent the interquartile ranges and the ends of the whiskers represent the maximum and minimum values. In these plots 50% of the data lie within the bounds of the box and most of the data contained by the whiskers can be considered outliers. The box and whisker plots for all tests are displayed side by side in Figure 4-1.

Figure 4‑1. Box and whisker plots for all charting configurations and trial sizes.

From this figure, it is clearly observed that increasing the number of charts strongly correlates with an increased loading time. For all chart storing methods, all the significant data points of median, interquartile medians, maximum and minimum values increase when more charts are loaded. This result is as expected, however, if the timing values are normalized by dividing them by the number of charts being generated per test, the box plots in Figure 4-2 are obtained. In this graph, a noticeable trend is observed that the times to generate a chart get progressively smaller as more charts are added. Notably, the upper bounds of the data (maximum and third interquartile values) tend to decrease with the addition of more charts, while the minimum values do not appear to change consistently. This trend suggests that there may be some form of optimization performed by the browser when loading chart data or rendering chart graphics which actually makes the process more efficient (in terms of milliseconds per chart) to load charts together in batches.

Figure 4‑2. Box and whisker plots for all charting configurations. Values are normalized such that times are given as milliseconds per chart

### Local Storage versus Database Storage

To determine the timing differences between using local storage and using database storage, box plots were created using all the timing data accumulated for each data storage technology. (These box plots included the data for all browsers and all trials, regardless of the number of charts generated in order to visualize the upper and lower bounds of the data storage and access method). Figure 4-3 compares these box plots side by side, illustrating the variance in timing ranges between local storage and the various network configurations for accessing the database.

Figure 4‑3. Box Plots for all data storage configurations, showing differences in measured ranges

Based on the ranges visualized in Figure 4-3, it is clear that local storage is the fastest technique for loading charts. While all the configurations have similar lower bound values, local storage has the shortest median loading time (depicted by the division between the grey and orange boxes) of 2,040 milliseconds as well as the shortest maximum loading time of 9,516 milliseconds. All configurations using the database have a median loading time of at least 7,510 milliseconds and have maximum loading times that surpass 30,000 milliseconds. Based on the fact that users typically get frustrated after waiting for no more than 2 to 3 seconds for web content to load (discussed in Section 1 of this report), it is clear that local storage is the best option to load data. While the upper end of the local storage range does surpass the ideal three-second limit, loading times are within a two-second bound for at least half of the trials. Conversely, for all of the database loading configurations only the loading times within the lowest quarter of the data fall within the desired two to three-second timeframe.

Additionally, changing the configuration of the network for database access appears to have had minimal effect on chart loading time. For both the local network and proxy network setups, using two separate servers to host the web content and database only showed a slight change in performance. The median times for both networks decreased minutely when changing from one server to two servers (from 7,842.5 ms to 7,510 ms on the local network and 9,018.5 ms to 8,654 ms on the proxy network). The third interquartile range only showed a slight increase on separating the servers (from 13,836.5 ms to 14,611 ms on the local network and 17,721.5 ms to 17,790.25 ms on the proxy network) while the maximum values showed a marginally larger jump (from 31,374 ms to 35,439 ms on the local network and 38,517 ms to 3,9608 ms on the proxy network). While these maximum values did increase somewhat significantly, they can be assumed to be edge cases or outliers that would occur infrequently.

Changing from local network to proxy network had a more significant impact, with a jump in the maximum values of almost 4,000 ms, an increase in the medians of about 1,000 ms and increase in the third interquartile value of about 3,000 ms.

### Effect of Browser

An additional analysis was performed on the data to determine if chart loading times were impacted by the use of Internet browser. For each of the test configurations several values were found. Along with the maximum and minimum times, average and median times were computed for each browser. (A breakdown of these values is available in appendix D.) The browser producing the lowest and highest of each of these values was then recorded and entered into a chart for comparison. This chart, Table 4-1, shows these values side by side.

Table 4‑1. Browsers producing maximum and minimum timing values

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Max Average** | | **Min Average** | | **Max Time** | | **Min Time** | | **Max Median** | | **Min Median** | |
| **Local Storage** | | | | | | | | | | | | |
| **1** | Firefox | | Chrome | | Firefox | | Chrome | | IE | | Chrome | |
| **5** | Firefox | | Chrome | | Firefox | | Chrome | | Firefox | | Chrome | |
| **10** | IE | | Chrome | | IE | | Chrome | | IE | | Chrome | |
| **Single Server over Local Network** | | | | | | | | | | | | |
| **1** | Firefox | | Chrome | | IE | | Chrome | | Firefox | | Chrome | |
| **5** | Chrome | | IE | | Firefox | | Chrome | | IE | | Firefox | |
| **10** | Firefox | | Chrome | | IE | | IE | | Firefox | | Chrome | |
| **Two Servers (Dedicated Web and Database) over Local Network** | | | | | | | | | | | | |
| **1** | IE | | Chrome | | Firefox | | Chrome | | IE | | Chrome | |
| **5** | Firefox | | Chrome | | IE | | Chrome | | Firefox | | Chrome | |
| **10** | IE | | Chrome | | IE | | Chrome | | Chrome | | Firefox | |
| **Single Server over Proxy Network** | | | | | | | | | | | | |
| **1** | Chrome | | Firefox | | Chrome | | Chrome | | Chrome | | Firefox | |
| **5** | Firefox | | IE | | Firefox | | IE | | Firefox | | IE | |
| **10** | Firefox | | IE | | Firefox | | IE | | Chrome | | IE | |
| **Two Servers (Dedicated Web and Database) over Local Network** | | | | | | | | | | | | |
| **1** | Chrome | | IE | | Firefox | | Chrome | | Firefox | | Chrome | |
| **5** | Firefox | | Chrome | | Firefox | | Chrome | | Firefox | | IE | |
| **10** | Firefox | | Chrome | | Firefox | | Chrome | | Chrome | | IE | |
| **Count** | Firefox | 9 | Chrome | 10 | Firefox | 9 | Chrome | 12 | Firefox | 7 | Chrome | 8 |
| Chrome | 3 | IE | 4 | IE | 5 | IE | 3 | Chrome | 4 | IE | 4 |
| IE | 3 | Firefox | 1 | Chrome | 1 | Firefox | 0 | IE | 4 | Firefox | 3 |

The count row at the bottom of Table 4-1 shows a tally of the number of times a browser is listed in a given column. The browsers are listed in order of most to least occurrences within each column. By examining these listings, it is easily observed that Firefox holds the top position in all three of the maximum time categories as well as the bottom position for all three of the minimum categories. (This means that Firefox usually had the maximum average time, the maximum median time and/or the maximum overall time for a given test. Additionally, Firefox usually did not produce a minimum loading time nor did it tend to have the lowest average or lowest median loading times.) Thus, Firefox appears to be the slowest loading of the three browsers.

Conversely, Chrome holds the top position for all three of the minimum categories and the bottom position for two of the three maximum categories. This means that Chrome tended to load data the fastest for a given test and is thus the fastest browser tested.

Furthermore, the presence of Internet Explorer in the middle slot for four of the six categories shows that IE was typically neither the fastest nor slowest browser for a given test and is thus neither the fastest nor slowest browser overall (with respect to loading and displaying charts).

## Conclusions

Based on the analysis performed in this report, several key points have been discovered with respect to how charts can be stored and how they load. First, the use of a database server or the use of local web storage are the only feasible means of retaining data. As discussed in Section 2, these techniques are the only methods that support all required browsers and meet the data size requirements for storing chart information.

Second, the analysis from Section 4 shows that increasing the number of charts to load in a batch positively correlates with the overall loading time. (Loading one chart takes less time than loading five charts, which are both faster than loading ten charts, as one would logically expect). However, if the load times are divided by the number of charts being generated, it is clear that the amount of time needed per chart decreases as more charts are added, thus making loading charts together more efficient. This is likely due to some form of behind-the-scenes optimization that is performed either by the jQuery and Highcharts libraries or in the browsers themselves. As the tests for the analysis in this report did not exceed 10 charts, it is unknown whether this trend continues for larger numbers of charts or whether there is a point at which efficiency is lost; however, it is unlikely that most users will have more than 10 charts displayed on their dashboard page at any one time. The dashboard page is intended to show only the most relevant information to the user in an easy-to-read format. Having numerous charts is not practical for the user and defeats the purpose of the dashboard page.

Third, loading charts from local storage is the fastest-loading storage option, outperforming the database connection by several seconds. Local storage is also the only viable option for achieving a desirable two to three second web page loading time, as discussed in Section 1 of this report.

Additionally, using dedicated physical machines for the web server and database server reduces performance by adding extra data transfers over the network; however, this decreased performance is fairly minimal compared to connecting the website via the Internet versus connecting via local network (the former of which is significantly faster).

Finally, the browser that appears to load charts fastest is Chrome while the slowest is Firefox.

## Recommendations

Considering the conclusions discussed in Section 5, this report recommends several ideas to optimize the 360incentives website’s charting capabilities and to improve the overall user experience. Most importantly, the design of the chart data storage mechanisms should attempt to utilize browser local storage wherever possible. As the website must be able to provide data to the user across browsers and computers, the database cannot be entirely removed; however local storage data saving can be added to help improve performance.

A form of data caching on the client side should be used to create a local copy of the chart data so that, after data is initially loaded from the database, subsequent page accessing can load the information faster from web storage. While this solution will not eliminate the potentially lengthy wait when a chart is added or modified, it will reduce loading times upon navigation to the page and, more importantly, upon login to the website. (The dashboard page is the first page displayed after login). Unnecessary network traffic will also be reduced, as loading from a local copy does not require data transmission over the network. Effectively, this can alleviate some strain on the database (by decreasing the number of active connections) and liberate some previously used bandwidth.

Additionally, there is no need to attempt to optimize loading times for multiple charts by loading sequentially, as some mechanism at the browser or library level appears to perform this optimization.

If possible, the database and web servers should also be hosted on the same physical machine. This would provide a further slight performance boost for the cases where data must be loaded from the database. Further optimization could be performed on this back-end infrastructure, especially with respect to the database’s stored procedures. The exact nature of these potential changes is out of scope of this report.

Finally, 360incentives should attempt to persuade its clientele and all of its internal users into using Google Chrome as the web browser of choice for accessing the 360 website. In addition to the findings of this report that show that Chrome loads and displays charts the fastest, there are other potential bonuses for having the user base switch to this browser. Unifying the majority of users to one, modern browser would make software development easier for 360incentives, as minimal effort would be wasted on supporting outdated software (such as IE 8). Additionally, new browsers are typically more secure and more reliable than their predecessors, underscoring the need to keep users up to date.

## Glossary

**API:** Application Programming Interface; a set of functions or methods that provide a programmer the means to accomplish a set of tasks or access a specific software component. Often implemented in the form of a library.

**Client-Side:** Used to describe events, code, data, etc. that occur or exist within the context of a computer or browser controlled by a user.

**Content Delivery Network:** A system of distributed servers designed to provide various content or media to end users with a high degree of reliability, performance and availability

**JSON:** JavaScript Object Notation; a standardized format for defining, writing and storing objects and their properties in human-readable form, similar to XML. JSON strings can easily be converted into a code-manipulable object and vice versa.

**Server-Side:** Used to describe events, code, data, etc. that occur or exist within the context of the server (or servers) that provide some service to a user.

**Software as a Service:** A business model whereby users or clients pay a fee to use a software platform, package or service for a set amount of time.

## References

|  |  |
| --- | --- |
| [1] | Akamai Technologies. “September 14, 2009 - Akamai Reveals 2 Seconds as the New Threshold of Acceptability for eCommerce Web Page Response Times”. Internet: <http://www.akamai.com/html/about/press/releases/2009/press_091409.html>, Sep. 14, 2009 [Apr. 29, 2014] |
| [2] | C. Hudson and T. Leadbetter. *HTML5 Developer’s Cookbook*. Upper Saddle River, NJ: Addison-Wesley, 2012, pp. 259-283. |
| [3] | W3C. “Web Storage”. Internet: <http://www.w3.org/TR/webstorage>, Jul. 30, 2013 [Apr. 29, 2014]. |
| [4] | C. Simms. “HTML5 Storage Wars - localStorage vs. IndexedDB vs. Web SQL”. Internet: <http://www.rfidjournal.com/blogs/experts/entry?10242>, May. 18, 2011 [Apr. 29, 2014]. |
| [5] | M. Mahemoff . “Client-Side Storage”. Internet: <http://www.html5rocks.com/en/tutorials/offline/storage>, Oct. 1, 2010 [Sep. 9, 2013]. |
| [6] | “Can I use…”. Internet: <http://caniuse.com> [Apr. 29, 2014] |
| [7] | W3C. “XMLHttpRequest Level 1”. Internet: <http://www.w3.org/TR/XMLHttpRequest>, Jan. 30, 2014 [Apr. 29, 2014]. |
| [8] | jQuery Foundation. “Browser Support”. Internet: <http://jquery.com/browser-support>, [Apr. 29, 2014]. |

## Appendix A: Example of JSON Query String

The following is an example of the JSON query strings generated by the JavaScript to request chart information when a user adds or modifies a chart. The 13 parameters below refer to various options that are set by the user. This string is passed to the backend C# code which then parses it and retrieves the specified data from the database. Storing these query strings allows charts to be regenerated without duplicating the actual data they display (which are saved elsewhere in the database) for each instance of a chart across one or multiple users.

**Query String**

{

"chartType": "pie",

"reportType": 47,

"bySelect": 0,

"startDate": "04/02/2014",

"endDate": "04/08/2014",

"numDays": null,

"filterSelect": 15,

"programType": 3,

"programSelect": "SomeProgramName",

"brandSelect": "SomeBrandName",

"salesChannelName": "SalesChannelXYZ",

"salesChannelId": 1234,

"reportDetails": "summary"

}

## Appendix B: Example of JSON Series Data String

The following is an example of the series JSON data strings that are returned by the C# code and used by the Highcharts library to create graphs. Note that the values assigned to the properties in the string are gibberish placeholder values that were stored in the 360incentives quality assurance database. The vertical ellipses denotes that many similar data pairs occur between the two shown in the sample below.

{

**Data String**

{

"series": [

{

"data": [

{

"name": "Fcxg Ejcvoqp ",

"y": 4101.95

},

**.**

**.**

**.**

{

"name": "Octkqp Hkvbigtcnf ",

"y": 25

},

],

"type": "pie"

}

],

"subtitle": {

"text": "04/05/2010 to 04/15/2014"

},

"xAxis": {

"title": {

"text": "UserName"

},

"type": "category"

},

"yAxis": {

"title": {

"text": "Amount Claims"

}

},

"plotOptions": {},

"title": {

"text": "Category And Brand Report"

}

}

## Appendix C: Timing Test Source Code JavaScript

var starttime;

var endtime;

var remainingcharts;

var dbtestingnum;

var cookietestingnum;

var lstestingnum;

function getNumCharts() {

return parseInt($("#selNumCharts").val())

}

function getCookie(cname) {

var name = cname + "=";

var ca = document.cookie.split(';');

for (var i = 0; i < ca.length; i++) {

var c = ca[i].trim();

if (c.indexOf(name) == 0) return c.substring(name.length, c.length);

}

return "";

};

function getRandomInt(min, max) {

return Math.floor(Math.random() \* (max - min + 1)) + min;

}

function getRandomNumbers(numberOfInts) {

var numarray = new Array(numberOfInts);

for (var i = 0; i < numarray.length; i++) {

numarray[i] = getRandomInt(1, 3);

}

return numarray;

}

function cookieTestingMethod(testnum) {

var numcharts = getNumCharts();

var cookienums = getRandomNumbers(numcharts);

starttime = window.performance.now();

for (var i = 0; i < numcharts; i++) {

j = (i + 1).toString();

$("#chart" + j).highcharts(jQuery.parseJSON(getCookie("cookie" + cookienums[i])));

}

endtime = window.performance.now();

row = document.createElement("tr");

d1 = document.createElement("td");

d2 = document.createElement("td");

d3 = document.createElement("td");

d1.textContent = testnum + 1;

d2.textContent = Math.round(endtime - starttime);

d3.textContent = endtime - starttime;

row.appendChild(d1);

row.appendChild(d2);

row.appendChild(d3);

table = document.getElementById("resultsTable");

table.appendChild(row);

var myevent = document.createEvent("HTMLEvents");

myevent.initEvent("cookiefinished", true, true);

document.getElementById("lnkCookie").dispatchEvent(myevent);

}

function lsTestingMethod(testnum){

var numcharts = getNumCharts();

var cookienums = getRandomNumbers(numcharts);

starttime = window.performance.now();

for (var i = 0; i < numcharts; i++) {

j = (i + 1).toString();

$("#chart" + j).highcharts(jQuery.parseJSON(localStorage.getItem("s" + cookienums[i])));

}

endtime = window.performance.now();

row = document.createElement("tr");

d1 = document.createElement("td");

d2 = document.createElement("td");

d3 = document.createElement("td");

d1.textContent = testnum + 1;

d2.textContent = Math.round(endtime - starttime);

d3.textContent = endtime - starttime;

row.appendChild(d1);

row.appendChild(d2);

row.appendChild(d3);

table = document.getElementById("resultsTable");

table.appendChild(row)

var myevent = document.createEvent("HTMLEvents");

myevent.initEvent("lsfinished", true, true);

document.getElementById("lnkLocalStorage").dispatchEvent(myevent);

}

function dbTestingMethod(testnum) {

var numcharts = getNumCharts();

var querynums = getRandomNumbers(numcharts);

var queries = new Array();

remainingcharts = getNumCharts();

for (var i = 0; i < numcharts; i++) {

queries[i] = getCookie("q" + querynums[i]);

}

starttime = window.performance.now()

for (var i = 0; i < numcharts; i++) {

val = GetChartData("#chart" + (i + 1).toString(), queries[i], testnum);

}

}

function finishUpDBTest(testnum) {

row = document.createElement("tr");

d1 = document.createElement("td");

d2 = document.createElement("td");

d3 = document.createElement("td");

d1.textContent = testnum + 1;

d2.textContent = Math.round(endtime - starttime);

d3.textContent = endtime - starttime;

row.appendChild(d1);

row.appendChild(d2);

row.appendChild(d3);

table = document.getElementById("resultsTable");

table.appendChild(row)

}

function GetChartData(chartID, querystring, testnum) {

$.ajax({

type: "POST",

contentType: "application/json; charset=utf-8",

url: "Dashboard.aspx/GetChartJSON",

data: "{'filterOptions':'" + querystring + "'}",

dataType: "json",

success: function (result) {

$(chartID).highcharts(jQuery.parseJSON(result.d));

remainingcharts = remainingcharts - 1;

if (remainingcharts == 0) {

endtime = window.performance.now();

finishUpDBTest(testnum);

var myevent = document.createEvent("HTMLEvents");

myevent.initEvent("finished", true, true);

document.getElementById("lnkDB").dispatchEvent(myevent);

}

return true;

},

Error: function () {

alert('Error creating chart');

}

});

};

$(document).ready(function () {

$("#lnkDB").click(function () {

dbtestingnum = 0;

document.getElementById("lnkDB").addEventListener("finished", function (e) {

if (dbtestingnum < 19) {

dbtestingnum = dbtestingnum + 1;

dbTestingMethod(dbtestingnum);

}

}, false);

dbTestingMethod(dbtestingnum);

});

$("#lnkClearTable").click(function () {

table = document.getElementById("resultsTable")

table.innerHTML = "";

row = document.createElement("tr");

d1 = document.createElement("td");

d2 = document.createElement("td");

d1.textContent = "Test Num";

d2.textContent = "Time";

row.appendChild(d1);

row.appendChild(d2);

table.appendChild(row);

});

$("#lnkCookie").click(function () {

cookietestingnum = 0;

document.getElementById("lnkCookie").addEventListener("cookiefinished",

function (e) {

if (cookietestingnum < 19) {

cookietestingnum = cookietestingnum + 1;

cookieTestingMethod(cookietestingnum);

}

}, false);

cookieTestingMethod(cookietestingnum);

});

$("#lnkLocalStorage").click(function () {

lstestingnum = 0;

document.getElementById("lnkLocalStorage").addEventListener("lsfinished", function (e) {

if (lstestingnum < 19) {

lstestingnum = lstestingnum + 1;

lsTestingMethod(lstestingnum);

}

}, false);

lsTestingMethod(lstestingnum);

});

$("#load").click(function () {

cookienum = $("#selCookieNum").val();

if (isNaN(cookienum)) {

if (cookienum.indexOf("q") != -1) {

cookiestring = getCookie(cookienum);

}

else {

cookiestring = localStorage.getItem(cookienum);

}

}

else {

cookiestring = getCookie("cookie" + cookienum);

}

$("#cookieText").val(cookiestring);

});

$("#lnkClearCharts").click(function () {

for (var i = 1; i < 11; i++) {

$("#chart" + i).highcharts({});

}

});

$("#submit").click(function () {

cookienum = $("#selCookieNum").val();

cookieval = $("#cookieText").val();

$("#cookieText").val("");

if (isNaN(cookienum)) {

if (cookienum.indexOf("q") != -1) {

document.cookie = cookienum + "=" + cookieval;

}

else {

localStorage.setItem(cookienum, cookieval);

}

}

else {

document.cookie = "cookie" + cookienum + "=" + cookieval;

}

});

});

## Appendix D: Chart Generation Timing Values

Table D‑1. Averages, maximum and minimum values and medians for recorded timing values broken down by storage type/configuration, the number of charts generated per test and by browser

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Average Time (ms) | Maximum Time (ms) | Minimum Time (ms) | Median Time (ms) |
| Two Servers (Dedicated Web and Database) over Local Network | **11549.76111** | **39608** | **356** | **8654** |
| 10 Charts Per Test | **21760.95** | **39608** | **3869** | **21438** |
| Firefox | 25432.3 | 39608 | 10135 | 25149.5 |
| IE | 22798.85 | 32479 | 8587 | 23998.5 |
| Chrome | 17051.7 | 24679 | 3869 | 17253 |
| 5 Charts Per Test | **10480.7** | **29040** | **2277** | 10966.5 |
| Firefox | 12259 | 29040 | 2686 | 12871 |
| Chrome | 10171.75 | 15422 | 2277 | 11077 |
| IE | 9011.35 | 21815 | 2468 | 8162.5 |
| 1 Chart Per Test | **2407.633333** | **10779** | **356** | **903** |
| Chrome | 2726.3 | 6708 | 356 | 1061.5 |
| Firefox | 2390.7 | 10779 | 546 | 925.5 |
| IE | 2105.9 | 6722 | 402 | 846 |
| Single Server over Proxy Network | **10973.52778** | **38517** | **293** | **9018.5** |
| 10 Charts Per Test | **19867.75** | **38517** | **9101** | **19515.5** |
| Firefox | 20999.8 | 38517 | 10153 | 19833 |
| Chrome | 20045.15 | 34769 | 12741 | 21129 |
| IE | 18558.3 | 27659 | 9101 | 18895.5 |
| 5 Charts Per Test | **10438.43333** | **23029** | **2104** | **8923** |
| Firefox | 11823.45 | 23029 | 2803 | 10853.5 |
| Chrome | 11316.95 | 21862 | 2547 | 9988 |
| IE | 8174.9 | 17690 | 2104 | 7681.5 |
| 1 Chart Per Test | **2614.4** | **12106** | **293** | **896** |
| Chrome | 3055.95 | 12106 | 293 | 964 |
| IE | 2476.6 | 6936 | 538 | 868 |
| Firefox | 2310.65 | 7345 | 517 | 766 |
| Two Servers (Dedicated Web and Database) over Local Network | **9251.927778** | **35439** | **170** | **7510** |
| 10 Charts Per Test | **16749.8** | **35439** | **7250** | **16152** |
| IE | 16981.25 | 35439 | 8244 | 16321 |
| Firefox | 16960.9 | 27340 | 7263 | 16124.5 |
| Chrome | 16307.25 | 28010 | 7250 | 16982.5 |
| 5 Charts Per Test | **8842.566667** | **22280** | **2328** | **8362.5** |
| Firefox | 8973.15 | 15361 | 2501 | 10811.5 |
| IE | 8937.85 | 22280 | 2449 | 8362.5 |
| Chrome | 8616.7 | 14987 | 2328 | 6849.5 |
| 1 Chart Per Test | **2163.416667** | **6238** | **170** | **539.5** |
| IE | 2359.8 | 6021 | 289 | 802.5 |
| Firefox | 2190.7 | 6238 | 391 | 698.5 |
| Chrome | 1939.75 | 5143 | 170 | 470 |
| Single Server over Local Network | **9000.683333** | **31374** | **159** | **7842.5** |
| 10 Charts Per Test | **15422.58333** | **31374** | **2940** | **15070.5** |
| Firefox | 16805.15 | 26675 | 3568 | 16292 |
| IE | 16135 | 31374 | 2940 | 15239 |
| Chrome | 13327.6 | 22984 | 3453 | 11642.5 |
| 5 Charts Per Test | **9357.283333** | **18822** | **2420** | **9981** |
| Chrome | 9793.8 | 17464 | 2420 | 10101 |
| Firefox | 9225.65 | 18822 | 2725 | 7822.5 |
| IE | 9052.4 | 14713 | 2490 | 10360 |
| 1 Chart Per Test | **2222.183333** | **6405** | **159** | **702.5** |
| Firefox | 2942.35 | 5972 | 391 | 1119.5 |
| IE | 2458.85 | 6405 | 321 | 1099 |
| Chrome | 1265.35 | 5194 | 159 | 351.5 |
| Local Storage | **2520.488889** | **9516** | **51** | **2040** |
| 10 Charts Per Test | **4284.05** | **9516** | **936** | **4157.5** |
| IE | 5977.7 | 9516 | 3010 | 5791 |
| Firefox | 4095.8 | 8868 | 1523 | 4259 |
| Chrome | 2778.65 | 4775 | 936 | 2748 |
| 5 Charts Per Test | **2756.95** | **7056** | **444** | **2475** |
| Firefox | 3548.55 | 7056 | 1192 | 2968 |
| IE | 3196.9 | 5457 | 1000 | 2808.5 |
| Chrome | 1525.4 | 2726 | 444 | 1392 |
| 1 Chart Per Test | **520.4666667** | **2833** | **51** | **258.5** |
| Firefox | 681 | 2833 | 135 | 320 |
| IE | 651.7 | 2074 | 112 | 349.5 |
| Chrome | 228.7 | 777 | 51 | 94 |