University of Waterloo

Faculty of Engineering

Department of Electrical and Computer Engineering

Selection of Best Proxy Assignment Algorithm for Daily Data Scraping Tasks at Hubdoc

Hubdoc

1881 Yonge Street

Toronto, Ontario, Canada

Prepared by

Justin Matthew Palumbo

ID 20424969

userid: jmpalumb

3A Computer Engineering

20 December 2014

7 Dewbourne Place

Whitby, ON

L1R 1X9

20 December 2014

Manoj Sachdev, Chair

Electrical and Computer Engineering

University of Waterloo

Waterloo, ON

N2L 3G1

Dear Sir:

The attached report, titled “Selection of Best Proxy Assignment Algorithm for Daily Data Scraping Tasks at Hubdoc”, was prepared as my 3B Work Report for the University of Waterloo, in fulfilment of the course WKRPT 401. The purpose of this report is to explain the best method for assigning daily automated tasks that retrieve financial information from websites to run through proxy configurations as determined by my testing and evaluation. Several algorithms are evaluated based on their overall performance of assigning proxy server connections such that an entire queue of jobs processed quickly.

Hubdoc is a software-as-a-service company that provides its users with software that automatically collects financial data and documents such as receipts, invoices and monthly statements from the websites of billers and banks. Although the end user primarily interacts only with Hubdoc’s website, at the core of the company’s service are the automated scripts that use credentials provided by the user to log into third-party websites and retrieve data.

Along with the responsibility of writing and maintaining the aforementioned scripts to test the website, I, along with another Waterloo co-op student, was given the opportunity to work on developing various new components in Hubdoc’s back-end system. As such, I was involved in designing, programming and configuring a new system of scheduling the data-retrieval scripts to run through proxy servers. This report was written for the consideration of Hubdoc’s current development team: Zachary Yang, Dave McKenna and myself, as well as for my supervisor, Dave Stibrany, to aid in the decision-making process for the implementation of an efficient algorithm.

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

## Hubdoc is a software-as-a-service company that provides its users with a web platform for managing, automatically retrieving, uploading and storing financial documents such as receipts and invoices. Information including bill dates, dollar amounts and account numbers is parsed from the documents and this data can be automatically inputted into accounting software products such as those provided by third-party companies, Xero and Intuit. While the Hubdoc website is accessible to average consumers, the core demographic of Hubdoc users is accountants because of the invaluable service provided by automatic data collection. An accountant can create a Hubdoc user profile and configure sub-profiles for each of his own clients such that he does not need to spend time manually entering data or exert any effort in requesting documents from the client.

## This data collection is made possible through scripts, referred to as ‘robots’, which can automatically log into the websites of billers to retrieve documents in the form of PDFs, CSVs, and common image formats. Currently, robots are available for all major banks in Canada and the United States as well as for many telecom, energy and web-service companies. The login credentials to these websites are provided by the Hubdoc user when they configure a robot to run for that account and are stored securely in Hubdoc’s database. The first time a robot is run for a set of credentials, it will collect all the documents the script can find. To provide users with up-to-date data, all the robot tasks for all available sets of credentials are run daily, but will only retrieve documents that haven’t already been collected.

## These robot scripts are written in JavaScript using the Node.js environment to run as a server-side application. Using various Node.js modules, the scripts are written to make the correct HTTP GET and POST requests for logging into a given biller’s website and downloading financial information. For some of the more complex websites an additional library, PhantomJS, is used to create an instance of a headless browser which is manipulated as necessary to get the desired web content. All of the robot script tasks run on one Linux server known as the ‘robot server’. Although this single server has been effective in processing the daily robot activity, as Hubdoc and its user base grow, optimizations and expansions will need to be implemented in order for the website to meet the demand.

## In the current infrastructure, as significant bottleneck to the system is that tasks for certain billers cannot be processed in parallel. In general, multiple simultaneous robots may run on the robot server, however the scheduler will prevent two instances of some robots, such as the TD Bank robot, from being executing concurrently. This is not due to a limitation on the Hubdoc website, but rather because of limitations imposed by TD Bank as they will only allow one user account to be logged at a time per IP address. To bypass this limitation, a system of using proxy servers has been proposed such that connections to biller websites such as TD Bank’s can be routed through a remote server, so that the IP address detected by the biller website is unique. The use of one or more proxy servers could therefore increase throughput and thereby reduce the overall amount of time required to process daily robot tasks.

### Proxy Assignment Design Problem

## The remainder of this report focuses on finding an effective algorithm for assigning robots to use different proxy servers as required for the purpose of reducing the time needed to run Hubdoc’s daily robot jobs. Currently, robot tasks are ordered for execution by a scheduling algorithm. This scheduler builds a queue of tasks to run daily by accessing information from the table of active robots in Hubdoc’s database but can also run robots that have been manually requested by users. Robots are placed in a list of active jobs when the scheduler determines it is ready to run the task. A process is spawned for the robot and the robot is removed from the list upon process completion. To create a system where individual robots are configured to route their network traffic through proxy servers, a new algorithm needs to be implemented to access the list of tasks the scheduler has selected to run, select appropriate proxy servers for each robot and spawn or block each task as necessary. (Fortunately, configuring a robot to use a proxy is simplified by the libraries used; the address of the proxy server simply needs to be specified as a parameter to the process’s constructor.) The two major criterion for a solution that meets Hubdoc’s needs are:

## The desired solution should take the least amount of time on average to process a queue of robot tasks, and,

## The desired solution should be scalable in the sense that it should continue to function efficiently as the number of billers requiring connections through proxies is increased.

## Several different algorithms for assigning robot tasks to proxies are evaluated to determine their usefulness to Hubdoc. The number of proxies to use is also considered in this report’s analysis. It should be noted that the algorithms examined in this report complete the function of assigning tasks that have been selected to run to an available proxy, and blocking these tasks from running when necessary. The global scheduling and ordering of tasks is outside the scope of this report.

## Proxy Assignment Algorithms

This section outlines the different factors that were considered in the creation of a suitable proxy assignment algorithm and their expected effects on performance. Section 2.1 briefly describes how the algorithm will be interfaced with the existing scheduler. Section 2.2 outlines a proxy assignment paradigm referred to in this report as ‘static’ assignment while Section 2.3 considers the opposing, system herein called ‘dynamic’ assignment. Finally, Section 2.4 discusses the two options considered in this report for handling tasks that cannot currently be executed.

### Integration with Existing System

The existing system used to run robot tasks on the robot server is a simple scheduling algorithm that queries the Hubdoc database to find robots to queue for daily execution. This scheduler also receives requests from user input to run robot tasks on demand. Using these two sources of input (and prioritizing the user generated requests) the scheduler executes the robots and manages a list of running tasks. These tasks are configured to connect to biller websites only using the IP address of the robot server. This system in depicted in Figure 1 below.

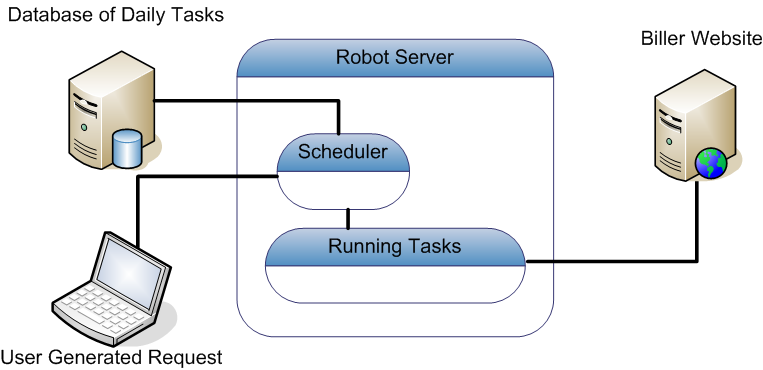


Figure . Depiction of Hubdoc's current robot execution system, using only one IP address to connect to biller websites

To add the functionality of proxy assignment to robots, the list of running robots created by the scheduler will instead be treated as a queue of robots that have been selected to execute in a specific order. The proxy algorithm will be inserted as an extra stage between the scheduler and execution of the robot tasks as depicted in Figure 2.

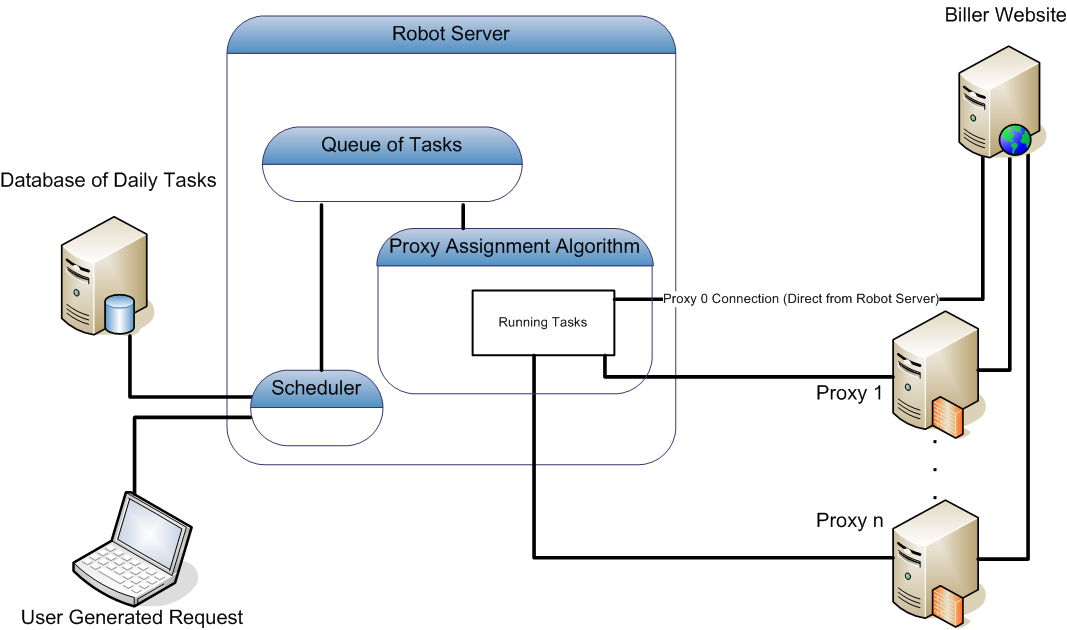


Figure Illustration of the new proxy assignment algorithm’s integration into Hubdoc's robot execution system

The new proxy assignment algorithm will periodically examine the queue to see if the task at the front of the queue can be executed. If so, the task will be removed from the queue, the constructor will be instantiated with the parameter corresponding to the appropriate proxy and the task will be added to a separate list of active jobs for that proxy. It should be note that the code for all robots is always executed on the single robot server and only the traffic for the requests is routed through a proxy. Upon task completion, the robot will be removed from the list of the proxy on which it was executed.

The two factors affecting if a task can be run are whether the server is currently processing the maximum number of concurrent jobs and whether there is an available proxy for that task. For most tasks, there will always be an available proxy because most biller websites do not restrict the number of accounts that can be logged in simultaneously from a given IP address. As such, these tasks can always be run from the same proxy. For simplicity, a task that is running and using a direct connection to a biller website (i.e., using the IP address of the robot server) will be referred to as using ‘proxy 0’. Any task for a biller that does not require a separate IP address per logged in account can be run on proxy 0. (These tasks will, in fact, be assigned to the proxy 0 connection to reduce latency and thereby minimize the amount of time needed to execute the task.) Tasks for a biller that do enforce a logged in restrictions, will have to share the remaining proxies. (Note that if multiple independent billers enforce these restrictions, one task for each biller can be run through each available proxy since an active connection to one website does not interfere with the enforcement rules for another website).

The instance in time where a task is assigned a proxy is determined by whether the static or dynamic paradigm is used. This is discussed further in Sections 2.2 and 2.3.

### Static Proxy Assignment

The concept of static proxy assignment refers to assigning a task to a proxy once and running that task through the assigned proxy when that proxy is available. A proxy is assigned to the task when it is added to the waiting queue, using one of two separate algorithms (described below). When the assignment algorithm tries to run a task at the front of the queue, it will check to see if the assigned proxy is available to run that task. If so, the task will be executed using that proxy’s connection. Otherwise, the task will be processed by one of the two methods described in section 2.4. Table 1 below describes possible advantages and disadvantages for using static proxy assignment.

Table . Advantages and disadvantages for using static proxy assignment

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| Tasks are assigned when robot is added to the queue, meaning that processing time is not used to select a proxy every time a task is at the front of the queue | Possible waste of resources as robots do not update their assignments to use currently available proxies |
| Fair/equal distribution of tasks to proxies | Needs an extra algorithm to decide how to assign proxies statically |
| Faster determination of whether to block or run a task | Needs variables or data structures to keep track of which proxy to assign to (increased memory usage) |

As mentioned in the table above, static proxy assignment needs to use an additional algorithm to assign proxies. For this report, two algorithms were considered: round robin assignment and shortest queue assignment. In both algorithms, when a task enters the queue for a biller that does not enforce IP address restrictions, the task is assigned to proxy 0. Tasks for billers that do enforce IP restrictions however are treated differently in the two algorithm schemes.

In round robin, tasks for billers requiring proxy connections are assigned their proxies in a simple numerical order which cycles back to the beginning after assigning a task to the last proxy. This order is separate for unique billers, meaning that if billers A and B both have IP restricting websites, the round robin assignment will preserve the ordering independently for each biller (for example, two tasks entering the queue for billers A and B can both be assigned to proxy 0).

In shortest queue assignment, tasks are given their proxy assignment based on which proxies have the shortest number of tasks for the same biller that are queue to use it. If there is a tie between proxies, the proxy with the smallest number is selected. In practice, this algorithm should have similar performance to the round robin algorithm at first when daily tasks are loaded into the waiting queue, but this will likely differ after tasks begin to execute and free up proxies at different rates.

### Dynamic Proxy Assignment

Dynamic proxy assignment refers to the concept of assigning a proxy to a task when the task is removed from the waiting queue. When the algorithm tries to run a task at the front of the queue it will run the task on proxy 0 if the task’s biller does not require a proxy connection for concurrent jobs. Otherwise the algorithm will cycle through a list data structure for each proxy and check to see if any of the currently running jobs for each proxy are for the same biller as the task that we want to run. If the algorithm finds that none of the current tasks on a proxy are for the same biller, it will de-queue the new task and assign it the available proxy. If no available proxies are found, the task will be dealt with by one of the methods discussed in section 2.4. The following table, Table 2 shows some possible advantages and disadvantages to using a dynamic proxy assignment.

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| Tasks will only be blocked if there are no available proxies (or max number of concurrent tasks for the robot server has been reached) | More time spent determining whether a task can run (and on which proxy) since the algorithm has to scan through the list of running tasks on each proxy. |
|  | Data structures for the running tasks have to be stored and managed for each proxy |
|  | More overall calculations performed (each blocked task will have the assignment algorithm run every time the task appears at the front of the queue) |

### Blocked Task Algorithms

This report considers two methods of handling tasks at the front of the waiting queue that cannot be executed due to a lack of available proxy connection. (For static assignment, this would mean that the proxy that the waiting task has been assigned cannot currently run this task because another job for the same biller is already using the proxy’s connection. For dynamic assignment, a task would be blocked if all proxies are currently being used for tasks for the same biller).

The first procedure for handling blocked tasks is to send the task to the back of the waiting queue. This algorithm is simple to implement, however may have poor performance as it does not try to optimize by finding a runnable task; it simply moves a blocked task to the back and with the hope that the next task in the queue can then be executed.

The second algorithm for managing blocked tasks is, upon detection that the task at the front of the queue is not currently executable, to traverse backwards along the waiting queue until a task that is runnable is found. This task is then brought to the front of the queue. This algorithm has an advantage in that it will always bring a runnable task to the front of the queue (if a runnable task exists), but it has some processing overhead from having to loop through the queue until it finds a suitable task. This processing overhead may even be inflated if the original blocked task remains blocked for a significant amount of time, forcing the program to repeatedly traverse the queue. Additionally, the algorithm of bringing the first runnable task forward is somewhat more complicated to implement than the code for sending a blocked task to the back of the queue.

## Testing Methodology

This section describes the tests that were used to evaluate the proxy assignment algorithms discussed in Section 2. Section 3.1 outlines the requirements for the tests, while Section 3.2 details how tests were implemented to meet these requirements.

### Requirements for Tests

To create tests that would accurately model how the proxy assignment algorithms would perform in Hubdoc’s environment, the nature of the robot tasks had to be considered. The daily robot tasks Hubdoc are not static in the sense that the list of jobs changes every day. Hubdoc users can add or remove billers in their profiles, new users can sign up, and trial accounts can expire, all causing changes in the queue of work to be done. Additionally, users can asynchronously request a robot to run at any time, forcing the task to be added to the scheduler. This means that the best proxy assignment algorithm needs to perform well on an average case and shouldn’t be optimized for a particular sequence of robot tasks. To test this, simulations have to use randomly generated test patterns.

Additionally, the duration of each task is unknown. Several factors will contribute to the amount of time needed for each task to run: network latency, server loads on the target website, the number of sub-accounts that need to be processed (e.g. a robot for a bank website may need to process several different credit, deposit or savings accounts), and whether this is the first time the robot is run for this user (on the first run, historical documents will be downloaded). As such the randomly generated test patterns should run each task for a random amount of time, between some upper and lower-bounds.

Finally, the test patterns and task durations should remain consistent between tests for each algorithm configuration such that a fair comparison of performance can be made.

### Implementation of Testing Solution

To create tests that met the requirements outlined in Section 3.1, a JavaScript program was created to generate random 10 test sequences. Each of these test sequences included 100 separate tasks where each task was assigned a random biller (lettered from A to E) and a random duration. Using the assumptions that a particularly quick robot would run for 30 seconds and that the robot server system would timeout a slow robot task after 10 minutes, these durations were uniformly distributed between half a second to 10 seconds, each simulation second representing a real-life minute. (This scaling factor was used for the practical reason of allowing the tests to complete faster since 150 configurations were tested, each running the 10 sets of 100 tasks.) The test pattern was saved to a file as a JavaScript array of test sequence objects, each containing an array of task objects. An example of a test pattern with of 5 test sequences, each with 5 tasks is provided in Appendix A.

The proxy assignment algorithms were then tested with another JavaScript program that could be configured via an HTML page. This page provided options for either starting the simulation on a specific configuration of the proxy assignment algorithm or execute a batch test that would run through all the configurations. Additionally, tasks running on each proxy were updated in real-time on this page. The JavaScript used to test the algorithms was implemented in one file with sections of code that would control the flow of execution based on what configuration was being tested (this script is available in Appendix B). The following parameters were specified for each test:

* The number of proxy connections (1 to 5)
* The number of billers requiring a unique IP address for each concurrent task (1 to 5)
* Static versus dynamic proxy assignment
* Sending blocked tasks to the back of the waiting queue versus bringing the next runnable task forward
* Round Robin versus shortest queue proxy assignment (for static proxy assignment only).

Using the appropriate paths of execution for the test configuration, the simulation would clear the necessary memory and iterate through each of the 10 tests. For each test, it would create necessary proxy objects and load the tasks into a global waiting queue and begin polling the start of the queue every 10 milliseconds for tasks that could be run. (A hardcoded limit of 10 concurrent tasks was also imposed. Changing this maximum number of simultaneous tasks was not considered in this report for the purpose of limiting the number of changing variables and thus reducing the number of tests that needed to be executed.) When a task was selected to execute it was popped off the waiting queue, added to the appropriate running lists for the proxy it was assigned and an asynchronous timeout was created using the task’s duration time. Upon the interrupt signal from the timeout, the task was removed from the running list.

In total, 150 configurations were tested on each of the 10 test sequences of 100 tasks. For each of the configurations, timing data was collected for the starting and ending time of the test sequence, and waiting times of each task being loaded into the waiting queue.

## Analysis of Testing Results

This section examines the data collected from the testing described in Section 3. These tests were designed to execute random patterns of simulation tasks in order to determine which algorithm assigned proxy connections most efficiently. The scripts collected information relating to three specific factors: the duration of running 100 jobs (the duration of a test), the waiting time for each task and a ratio of waiting time to task duration for each task. Because the main priority for Hubdoc is to process all the robot jobs quickly rather than treat each task fairly, only the first of these three metrics is considered for determining the optimal algorithm in the remainder of this report. (The latter two parameters, dealing with how long an individual task is waiting on the queue, are mostly irrelevant to Hubdoc’s needs if the overall set of tasks is processed quickly.)

To compare the performance of the algorithms, the median times to execute 100 tasks were plotted to show how wait times changed as the number of billers requiring a new proxy for each concurrent task was increased. Figure 3 shows the performance of each configuration when only one network connection was available.

Figure . Median Test Durations for 1 Proxy Connection

In this plot, it is easily observed the green, orange and gold lines have consistently low median durations regardless of the number of billers requiring a proxy per robot. While the red, blue and grey lines show much higher durations. There is also a significant peak in duration times in the where three billers out of five are these special billers. Similar trends can also be observed in Figure 4, the plot for two network connections.

Figure . Median Test Durations for 2 Proxy Connections

In this plot, the lines showing smaller median durations are again the green, orange and gold lines, and there is also a noticeable peak at the same point of three sole-IP-address billers. Figure 5 shows a similar pattern again for 3 proxy connections, although the peak appears to start sooner, around two sole-IP-billers.

## 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

**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.

Robot:

Configuration A:

Configuration B:

Configuration C:

Configuration D:

Configuration E:

Configuration F:

## 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 Test Pattern Array

The following is an example of a test pattern randomly generated by JavaScript to be loaded into the proxy assignment testing simulation. This test pattern shows 5 test sequences, each with 5 tasks whereas the test pattern that was used for simulation had 10 tests with 100 tasks each.

**Test Pattern**

var t5by5 = [{"id":0,"tasks":[{"duration":4191,"biller\_number":2,"biller":"Biller B"},{"duration":7065,"biller\_number":2,"biller":"Biller B"},{"duration":4691,"biller\_number":3,"biller":"Biller C"},{"duration":8059,"biller\_number":3,"biller":"Biller C"},{"duration":5865,"biller\_number":3,"biller":"Biller C"}]},{"id":1,"tasks":[{"duration":6577,"biller\_number":4,"biller":"Biller D"},{"duration":2872,"biller\_number":3,"biller":"Biller C"},{"duration":2031,"biller\_number":3,"biller":"Biller C"},{"duration":1816,"biller\_number":2,"biller":"Biller B"},{"duration":1957,"biller\_number":2,"biller":"Biller B"}]},{"id":2,"tasks":[{"duration":6071,"biller\_number":1,"biller":"Biller A"},{"duration":8146,"biller\_number":3,"biller":"Biller C"},{"duration":5260,"biller\_number":4,"biller":"Biller D"},{"duration":2502,"biller\_number":3,"biller":"Biller C"},{"duration":4281,"biller\_number":4,"biller":"Biller D"}]},{"id":3,"tasks":[{"duration":8947,"biller\_number":2,"biller":"Biller B"},{"duration":8018,"biller\_number":4,"biller":"Biller D"},{"duration":8904,"biller\_number":3,"biller":"Biller C"},{"duration":9105,"biller\_number":1,"biller":"Biller A"},{"duration":8418,"biller\_number":3,"biller":"Biller C"}]},{"id":4,"tasks":[{"duration":9716,"biller\_number":4,"biller":"Biller D"},{"duration":6263,"biller\_number":3,"biller":"Biller C"},{"duration":3918,"biller\_number":4,"biller":"Biller D"},{"duration":8802,"biller\_number":4,"biller":"Biller D"},{"duration":3811,"biller\_number":1,"biller":"Biller A"}]}]

## Appendix B: JavaScript Code for Testing Proxy Assignment Algorithms

"use strict"

var proxies = [];

var waiting\_queue =[];

var testStats= [];

var currently\_running = [];

var robot\_time\_data;

var test\_array =[t10by100, t100by100, t5by5, t2by100, customtest, test\_grid]; //These are various generated test patterns stored in other JS files

var global\_test\_data\_string = "Sequential Billers,Proxies, Proxy Assignment,Blocked Algorithm,Proxy Algorithm,T\_Max,T\_Q3,T\_Median,T\_Q1,T\_Min,T\_Average,W\_Max,W\_Q3,W\_Median,W\_Q1,W\_Min,W\_Average,R\_Max,R\_Q3,R\_Median,R\_Q1,R\_Min,R\_Average";

var config\_count = "0";

var num\_configs;

$(document).ready(function(){

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

var billers = $("#sequentialBillers").val();

var proxies = $("#proxies").val();

var proxy\_assign = $("#proxy-assignment").val();

var q\_algorithm = $("#q-algorithm").val();

var proxy\_assign\_alg = $("#algorithm").val();

var parameters = {

billers: billers,

proxies: proxies,

proxy\_assign: proxy\_assign,

q\_algorithm: q\_algorithm,

proxy\_assign\_alg: proxy\_assign\_alg

}

startTests(parameters, function() {alert("hello")})

})

$("#button2").click(function() {

var test\_configs = [];

for (var b = 1; b < 6; b++){

for(var p = 1; p< 6; p ++){

for(var p\_a = 1; p\_a < 3; p\_a++){

for (var q = 1; q < 3; q ++){

if (p\_a == 1){

for (var p\_a\_a = 1; p\_a\_a < 3; p\_a\_a ++){

test\_configs.push({

billers: b,

proxies: p,

proxy\_assign: p\_a,

q\_algorithm: q,

proxy\_assign\_alg: p\_a\_a

})

}

} else {

test\_configs.push({

billers: b,

proxies: p,

proxy\_assign: p\_a,

q\_algorithm: q,

proxy\_assign\_alg: 1

})

}

}

}

}

}

num\_configs = test\_configs.length;

async.forEachSeries(test\_configs, function(config, callback){

config\_count++;

startTests(config, callback)

}, function() {

console.log("Done ALL TESTS")

var blob = new Blob([global\_test\_data\_string], {type: "text/plain;charset=utf-8"});

saveAs(blob, "test\_data.csv");

})

})

})

function clearEverything() {

var proxies = [];

var waiting\_queue =[];

var testStats= [];

var currently\_running = [];

var robot\_time\_data = [];

}

function startTests(parameters, cb) {

//intialize

$("#button").prop( "disabled", true );

$("#button2").prop( "disabled", true );

$("body").append("<div id='test\_num'></div>");

testStats = [];

proxies = [];

var test\_sequence = test\_array[0];

var max\_concurrent\_tasks = 10;

$(".proxy\_box").remove();

for(var x = 0; x < parameters.proxies; x++) {

proxies = proxies.concat({

Running: [],

queued: {}

})

if (x == 0) {

$("body").append("<div class='proxy\_box' id='div\_" + x + "'><strong>Proxy " + x + " (Server)</strong><p></p></div>");

} else {

$("body").append("<div class='proxy\_box' id='div\_" + x + "'><strong>Proxy " + x + "</strong><p></p></div>");

}

}

async.forEachSeries(test\_sequence, function(test, callback) {

console.log("Running test " + (test.id + 1) + " of " + test\_sequence.length)

$("#test\_num").text("Running test " + (test.id + 1) + " of " + test\_sequence.length + " for config " + config\_count + " /" + num\_configs)

console.log("--------------------------------------")

var job\_count = 0;

var test\_start = new Date().getTime();

robot\_time\_data = [];

var billers\_job\_count = {}; //used for round robin proxy assignment

//This adds a new task to the waiting queue

var new\_robot\_adder = setInterval(function() {

if (test.tasks[job\_count]) {

console.log("Adding " + test.tasks[job\_count].biller + " id" + job\_count + " to waiting queue")

test.tasks[job\_count].startWaitTime = new Date().getTime();

test.tasks[job\_count].id = job\_count;

test.tasks[job\_count].force\_sequential = test.tasks[job\_count].biller\_number <= parameters.billers ;

//if we are forcing sequential and the proxy assignment is static, then we need to assign a proxy

if (test.tasks[job\_count].force\_sequential){

if (parameters.proxy\_assign == 1){

//Round robin

console.log(JSON.stringify(billers\_job\_count))

if (parameters.proxy\_assign\_alg ==1){

if (!billers\_job\_count[test.tasks[job\_count].biller]) billers\_job\_count[test.tasks[job\_count].biller] = 0;

test.tasks[job\_count].proxy = billers\_job\_count[test.tasks[job\_count].biller] % parameters.proxies;

billers\_job\_count[test.tasks[job\_count].biller]++;

console.log(test.tasks[job\_count].biller + " id" + job\_count + " proxy: " + test.tasks[job\_count].proxy)

} else {

//Shortest Queue Code

var min = null;

var position = null;

var added\_flag = false;

for(var x = 0; x < proxies.length; x ++) {

if (!proxies[x].queued[test.tasks[job\_count].biller]){

proxies[x].queued[test.tasks[job\_count].biller] = 1;

test.tasks[job\_count].proxy = x;

added\_flag = true;

break;

}

if (min == null){

position = 0;

min = proxies[x].queued[test.tasks[job\_count].biller];

}

if (proxies[x].queued[test.tasks[job\_count].biller] < min) {

position = x;

min = proxies[x].queued[test.tasks[job\_count].biller];

}

}

if (!added\_flag) {

proxies[position].queued[test.tasks[job\_count].biller] ++;

test.tasks[job\_count].proxy = position;

}

}

}

}

waiting\_queue.push(test.tasks[job\_count]);

job\_count ++;

if (job\_count > test.length) clearInterval(new\_robot\_adder)

}

}, 10)

var robot\_runner = setInterval(function() {

//update proxies list

for(var x = 0; x < parameters.proxies; x++) {

$("#div\_" + x + " p").text(JSON.stringify(proxies[x].Running))

}

//Nothing running, nothing waiting, nothing needs to be queued End the test.

if (currently\_running.length === 0 && waiting\_queue.length === 0 && test.tasks.length <= job\_count){

testStats.push({

test\_id: test.id,

test\_start: test\_start,

test\_end: new Date().getTime(),

test\_data: robot\_time\_data

})

clearInterval(robot\_runner);

return callback();

}

if (currently\_running.length < max\_concurrent\_tasks) {

//Implement the queuing logic.

if (waiting\_queue.length > 0) {

if (waiting\_queue[0].force\_sequential == false) {

runRobot(0);

} else {

//check if we can run the current robot

var data = canWeRunThisTask(waiting\_queue[0], parameters.proxy\_assign);

//if so, run it

if (data.can\_run){

//Decrement number of waiting tasks on the proxy that was selected if we are using static allocation and the assignment algorithm is shortest queue

if (parameters.proxy\_assign\_alg == 2 && parameters.proxy\_assign == 1) {

proxies[data.open\_proxy].queued[waiting\_queue[0].biller] --;

}

runRobot(data.open\_proxy);

} else {

//else do this:

if (parameters.q\_algorithm == 1) {

//send to back

waiting\_queue.push(waiting\_queue.shift());

} else {

//find the next available robot and run it

for(var k = 0; k < waiting\_queue.length; k++) {

var task = canWeRunThisTask(waiting\_queue[k], parameters.proxy\_assign)

//move the element to the front so that we can process it

if (task.can\_run){

var element = waiting\_queue.splice(k, 1)[0];

waiting\_queue.unshift(element)

//Decrement number of waiting tasks on the proxy that was selected if we are using static allocation and the assignment algorithm is shortest queue

if (parameters.proxy\_assign\_alg == 2 && parameters.proxy\_assign == 1) {

proxies[task.open\_proxy].queued[waiting\_queue[0].biller] --;

}

runRobot(task.open\_proxy);

break;

}

}

}

}

}

}

}

}, 10)

}, function() {

console.log("Done all tests")

$("#test\_num").text("Done all tests")

console.log("--------------------------------------")

$("#button").prop( "disabled", false );

var stats = collect\_data(testStats);

var prox\_Assign;

var blocked\_Alg;

var prox\_Assign\_alg;

if (parameters.proxy\_assign ==1){

prox\_Assign = "Static"

if (parameters.proxy\_assign\_alg ==1){

prox\_Assign\_alg = "Round Robin"

} else {

prox\_Assign\_alg = "Shortest Queue"

}

} else{

prox\_Assign = "Dynamic"

prox\_Assign\_alg = "-"

}

if (parameters.q\_algorithm ==1){

blocked\_Alg = "Send to Back"

} else{

blocked\_Alg = "Bring First Best Forward"

}

global\_test\_data\_string += "\n" + parameters.billers + "," + parameters.proxies + "," + prox\_Assign + "," + blocked\_Alg + "," + prox\_Assign\_alg + "," + stats.testTimes.max + "," + stats.testTimes.q3 + "," + stats.testTimes.median +"," + stats.testTimes.q1 +"," + stats.testTimes.min + "," + stats.testTimes.average + "," + stats.waitTimes.max + "," + stats.waitTimes.q3 + "," + stats.waitTimes.median +"," + stats.waitTimes.q1 +"," + stats.waitTimes.min + "," + stats.waitTimes.average + "," + stats.Ratios.max + "," + stats.Ratios.q3 + "," + stats.Ratios.median +"," + stats.Ratios.q1 +"," + stats.Ratios.min + "," + stats.Ratios.average

return cb();

})

}

function runRobot(selected\_proxy) {

var robot = waiting\_queue.shift();

robot.endWaitTime = new Date().getTime();

currently\_running.push(robot);

console.log("Running " + robot.biller + " id" + robot.id)

robot.proxy = selected\_proxy;

proxies[selected\_proxy].Running.push({

biller: robot.biller,

id: robot.id

})

setTimeout(function() {

//remove from currently running

for(var i = 0; i < currently\_running.length; i ++){

if (currently\_running[i].id === robot.id){

currently\_running.splice(i, 1);

break;

}

}

//remove from proxy's running list

for(var i = 0; i < proxies[robot.proxy].Running.length; i ++) {

if (proxies[robot.proxy].Running[i].id === robot.id){

proxies[robot.proxy].Running.splice(i, 1);

break;

}

}

console.log("Robot " + robot.biller + " id" + robot.id + " finished")

robot\_time\_data.push({

startWaitTime: robot.startWaitTime,

endWaitTime: robot.endWaitTime,

waiting: robot.endWaitTime - robot.startWaitTime,

duration: robot.duration,

ratio: (robot.endWaitTime - robot.startWaitTime)/ robot.duration,

})

}, robot.duration)

currently\_running.push()

}

function canWeRunThisTask(robot, proxy\_assignment) {

var can\_run = true;

var open\_proxy;

if (proxy\_assignment == 1){

//Use assigned proxy

if (robot.force\_sequential == false) {

return {

can\_run:true,

open\_proxy: 0

}

}

for(var i = 0; i < proxies[robot.proxy].Running.length; i ++) {

if (proxies[robot.proxy].Running[i].biller == robot.biller) {

can\_run = false;

break;

}

}

open\_proxy = robot.proxy;

} else {

can\_run = false;

//try to find an available proxy

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

var good\_proxy = true;

for(var j = 0; j < proxies[i].Running.length; j ++) {

if (proxies[i].Running[j].biller ==robot.biller) {

good\_proxy = false;

break;

}

}

if (good\_proxy) {

open\_proxy = i;

can\_run = true;

break;

}

}

}

return {

can\_run: can\_run,

open\_proxy: open\_proxy

}

}

## 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‑. 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 |