Master Degree Project

UNIVERSITY OF SKÖVDE

QUERYING JSON AND XML

Performance evaluation of querying tools for offline-enabled web applications

Master Degree Project in Informatics
One year Level 30 ECTS
Spring term 2012

Adrian Hellström

Supervisor: Henrik Gustavsson
Examiner: Birgitta Lindström

Querying JSON and XML

Submitted by Adrian Hellström to the University of Skövde as a final year project towards the degree of M.Sc. in the School of Humanities and Informatics. The project has been supervised by Henrik Gustavsson.

2012-06-03

I hereby certify that all material in this final year project which is not my own work has been identified and that no work is included for which a degree has already been conferred on me.

Signature: _____

Abstract

This article explores the viability of third-party JSON tools as an alternative to XML when an application requires querying and filtering of data, as well as how the application deviates between browsers. We examine and describe the querying alternatives as well as the technologies we worked with and used in the application. The application is built using HTML 5 features such as local storage and canvas, and is benchmarked in Internet Explorer, Chrome and Firefox. The application built is an animated infographical display that uses querying functions in JSON and XML to filter values from a dataset and then display them through the HTML5 canvas technology. The results were in favor of JSON and suggested that using third-party tools did not impact performance compared to native XML functions. In addition, the usage of JSON enabled easier development and cross-browser compatibility. Further research is proposed to examine document-based data filtering as well as investigating why performance deviated between toolsets.

Keywords: json, xml, querying, javascript, html5, canvas, local storage

Table of Contents

1 Introduction

XML (Extensible Markup Language) and JSON (JavaScript Object Notation) are both formats to store data in, each with their own notation and capabilities. JSON is known for its lightweight status and focus on data storage, XML on the other hand for being more of a markup language with a bigger emphasis on document-based content. XML versus JSON is an oft discussed topic. Research shows JSON being faster than XML (Nurseitov, Paulson, Reynolds & Izurieta, 2009) and displaying less overhead (Lawrence, 2004). XML however may outperform when faced with more document-based semi-structured data (Lawrence, 2004). Querying in XML may present issues because parsing XML can be costly in terms of parsing time (Nicola & John, 2003). JSON on the other hand lacks standardized querying tools completely, but there are strides being made to bring similar tools like those available for XML to JSON in the form of third-party additions. Due to the mentioned overhead in XML, huge amounts of raw data may present a problem when storage space is limited. Local storage for HTML 5 can enable applications to be more secure and provide more benefits than cookies, such as more space and enabling applications to function offline (West & Pulimood, 2012; Hsu & Chen, 2009). Another addition to the HTML 5 (Hypertext Markup Language) suite is the canvas toolset which can be used to display graphics on the web in an animated manner. This could be used to convey a soft real-time appropriate display of raw data found in statistics. Statistical data viewed as an infographic lends itself to the possibility of being a more compelling and easier understood experience (Huang & Tan, 2007). Animating said infographic is an important way to enhance the user's ability to visualize the continued animation beyond the actual raw statistical data present (Lengler & Moere, 2009). Given the inefficiency of XML's data-based content such as scientific data (Lawrence, 2004), raw data found in some XML files, such as statistical spreadsheets, could benefit from a translation to JSON.

Benchmarking has been done on the XML alternatives (Lam, Poon & Ding, 2008) and using JSON for data exchange instead of XML (Erfianto, Mahmood & Rahman, 2007), as well as benchmarks comparing JSON and XML (Nurseitov et al, 2009). However, no studies compare the two alternatives' way of querying data. We ask ourselves what combination of these techniques could be used to successfully create an application capable of rendering real-time graphics at a reasonable speed as well as what combination would be the most optimal one.

We employ a quantitative method analyzing empirical data entirely derived from the various benchmarks, similar to Ratanaworabhan, Livshits and Zorn (2010) or Lawrence (2004). These queries are presented in a graphical manner through the use of HTML 5 Canvas. The dataset used is publicly available and contains the census of the income and various other variables for the Los Angeles and Long Beach areas from 1970 to 1990 (University of California, 2000). This is a similar dataset from the same resource as the one used in Lawrence's (2004) article.

The benchmarks are done on a real-time canvas application created to display a raw data set in an infographic fashion. This application uses various querying methods of aforementioned formats to continuously filter the raw data and use the resulting iteration to power an animation on the canvas. In the end we compare the original and translated file sizes and take a look into the varying browsers' performance for the different querying variants insomuch as they apply to each file size.

2 Background

2.1 XML and JSON

Currently JSON is used in a wide variety of web applications and regards itself as a simpler and more lightweight alternative to XML for use in serializing and transmitting data (JSON.org, 2002). In contrast, XML is a markup language comprised of tags to arrange data (World Wide Web Consortium, 2012), JSON tries to minimize the data transmitted by using simple objects. These objects are composed of name and value pairs, and designed to be very readable for both humans and machines, just like XML, while still catering to high-performance requirements.

Research suggests that JSON is significantly faster than XML for transmitting data as well as utilizing less CPU time on the server (Nurseitov et al, 2009). However, JSON sports fewer features than XML, such as the ability to hold audio and picture data-types through the CDATA feature (JSON.org, 2002).

Compared to storage types such as comma separated files, XML – while more humanly readable (mostly depending on the descriptiveness of the tags) – can be very space inefficient, which could be an issue for scientific datasets and wireless communications. Experiments show that XML exhibits a significant overhead compared to other storage types yet it may outperform when told to handle semi-structured data, or when the data contains more text or redundant data (Lawrence, 2004).

The following is a comparison between an XML object and a JSON object with the same data. In this example JSON takes up more vertical space due to the indentation used, but when comparing the actual character usage the JSON example has a considerable lead.

XML

```
<persons>
  <person>
    <firstname>Arne</firstname>
    <lastname>Smith</lastname>
    <age>81</age>
    <cars>
      <car>
        <model>Volvo</model>
        <year>1980</year>
      </car>
      <car>
        <model>Yugo</model>
        <year>1700</year>
      </car>
    </cars>
 </person>
 <person>
…
 </person>
</persons>
```

JSON

```
{
    "person":
          [
              {
                  "firstname": "Arne",
                  "lastname": "Smith",
                  "age": 81,
                  "cars": [
                      {
                          "model": "Volvo",
                          "year": 1980
                      },
                      {
                          "model": "Yugo",
                          "year": 1700
                      }
                    ]
              },
              {
                …
              }
          ]
}
```

JSON, being very simplistic in nature, lacks any type of validation or schema format. As is the case with querying, there are third-party tools to achieve this, lending to JSON's open environment and extensibility with the caveat of possible third-party issues.

2.2 Navigating JSON versus navigating the XML DOM

2.2.1 Navigating the XML DOM

Navigation through XML to access the information within is done through JavaScript and the DOM (Document Object Model). Parsing the XML file as XML DOM is done through the XMLHttpRequest() object in JavaScript.

```
xmlhttp = new XMLHttpRequest();
xmlhttp.open("GET","xmlnav.xml",false);
xmlhttp.send();
xmlnav=xmlhttp.responseXML
```

After the document has been fetched, the JavaScript method "getElementsByTagName" is used to extract data from specific tags in the DOM of the XML. The method will return an array therefore the user needs to specify which entry of the array to extract, starting from zero.

```
document.getElementById("name").innerHTML = "First Name:
"+xmlnav.getElementsByTagName("firstname")[0].childNodes[0].nodeValu
e;
```

This example would fetch the contents of the very first "firstname" tag's first child node. In the case of the previous XML example, it would return the name "Arne". To print out

everyone's first name in the case of several persons in the XML document, one would need to select the entire "firstname" tag, ignoring child nodes at first.

```
person = xmlnav.getElementsByTagName("firstname");
```

Then iterate through this new array object, selecting the first child node each time, which in this case is the first name of every person.

```
for (i=0;i<person.length;i++) {
    document.getElementById("listofnames").innerHTML +=
    "<br/>"+person[i].childNodes[0].nodeValue;
}
```

In the case of multiple children of children, such as each person owning one or several cars, one would need to traverse upwards and get the parent node of the previous loop. This is to ensure that only that person's car is being listed and not the contents of every car element under every person. Then, using the resulting parent for another loop within the previous one, the model name of every car owned by each person would be displayed.

```
cars = person[i].parentNode.getElementsByTagName("model");
for (y=0;y<cars.length;y++) {
    document.getElementById("listofnames").innerHTML += "<br/>Car:
    "+cars[y].childNodes[0].nodeValue;
}
```

2.2.2 Navigating JSON

The JSON is loaded either through AJAX, as a plain JavaScript file, or attached within Script tags of the HTML document. JSON is navigated easily by handling the JSON as any JavaScript object through the dot notation.

In a similar manner to the examples for navigating XML, navigating to the first name of the first person in the list is done by calling the object and picking which position and path to traverse to.

```
document.getElementById("name").innerHTML = "First Name:
"+jsonNav.person[0].firstname;
```

As with the XML DOM, displaying everyone is done through iterating the length of the object and traversing each position.

```
for (i=0;i<jsonNav.person.length;i++) {
  document.getElementById("listofnames").innerHTML +=
  "<br/>"+jsonNav.person[i].firstname;
}
```

This following example however, displaying everyone's car, is simpler than through the XML DOM. There is no need to traverse back to the parent node of the previous loop only to display each person's corresponding car or cars.

```
for (y=0;y<jsonNav.person[i].cars.length;y++) {
  document.getElementById("listofnames").innerHTML += "<br/>Car:
  "+jsonNav.person[i].cars[y].model;
}
```

Unsurprisingly, navigating through JSON is easier due to the object-based structure and the way JavaScript handles objects compared to traversing the DOM of an XML file.

2.3 Querying

Being able to query aids in the pursuit of finding specific occurrences of data in an unknown space, a helpful tool when specific pieces of data can't be gleaned due to lacking the correct path or needing to find a partial match. Both XML and JSON support querying in varying degrees. There are a number of different tools to aid in attempting to query data in a JSON or XML file. In the case of JSON these are not standardized tools but third-party JavaScript plugins that we will display some examples of.

2.3.1 Querying JSON

JSON lacks a standardized query language akin to XPATH and XQUERY. Nonetheless, several independent alternatives for these tools have surfaced for use with JSON, extending JSON's capabilities for use in more demanding web applications. The three alternatives presented in this chapter offer a different feature set, notation and handling of the data. We will present some code examples showcasing simple differences in both syntax and paradigm.

2.3.2 Querying JSON with jLinq

Querying with jLinq is very similar to querying any kind of database. The user picks a field, applies conditionals and operators and then selects the results. These are given as an array which can later be iterated through to display all the results, very similar to how one would query a MySQL database in PHP and later display results.

```
var results = jlinq.from(jsonQuery.person[0].cars)
.starts("model", "v")
.select();

document.getElementById("cars").innerHTML = results[0].year;
```

The results are selected and the year of the first result of the array returned is displayed. Using the same sample as previously provided, this query would return the array of the "Volvo" model, and the year "1980" would be displayed. Several more operators besides "starts" are available such as "and", "or", "not", "ends" "greater", and combinations thereof such as "andEnds" (Bonacci, 2012).

2.3.3 Querying JSON with jOrder

Unlike the other two query implementations, jOrder works more like a complete database rather than just a database querying language. The user needs to specify a JSON file or structure to be turned into a table as well as the possibility of an index.

```
var table = jOrder( jsonQuery )
```

Specific columns in this table can be indexed. This is not a requirement, but the author of the toolkit states that it is a significant query performance gain at the cost of memory usage, similar to a real database.

```
. index( 'firstname', ['firstname'],    {ordered:true,
type:jOrder.string})
```

Queries can be run on the table structure provided initially. If the queried row is not indexed a warning message will be printed in the browsers' console window. In the following example the "where" clause is used on the table to search for the name "Arne".

```
var results = table.where([{ firstname: 'Arne' }]);
```

The results are returned as an array, which can be iterated through to pick the attribute to display. The jOrder query application supports inserting, updating, sorting and deleting entries (Stocker, 2011). Unfortunately only flat JSON structures are accepted; nested objects or non-ordinal field values are not accepted. Using primary and foreign key pairs can accomplish a similar functionality, at the cost of rearranging the JSON files to accommodate more fields.

2.3.4 Querying JSON with JSONPath

JSONPath was developed to fill the need for an XPath-like tool for JSON. It has the ability to extract data using expressions similar to XPath for XML (Goessner, 2012). We will be displaying how to use JSONPath to retrieve a specific model of a car from the example JSON used previously and then display the first name of the person who owns that car.

A recursive search is done through the entire JSON object to find the car with the model name of "Yugo". Similar to XPath which uses "//" for recursive search, JSONPath instead uses ".." .

```
findCarModel = jsonPath(jsonQuery,
"$..cars[?(@.model.indexOf('Yugo') > - 1)]");
```

To get the path of the car with the model name of "Yugo" a "resultType" argument is added to the end of the function.

```
carPath = jsonPath(jsonQuery, "$..cars[?(@.model.indexOf('Yugo') > -
1)]", {resultType:"PATH"});
```

It is then possible to use that path ("carPath") to get the name of the person who owns the car. Iterate as necessary. In this example only the first result is returned as the example JSON is small enough to show only one result.

```
result = jsonQuery[carPath[0][2]].firstname;
```

The name of the first car model found is returned, "Yugo" in this case, and the name of the person who owns it, "Arne".

```
document.getElementById("cars").innerHTML = "First owner of
"+findCarModel[0].model+" is "+result;
```

2.3.5 Querying XML

Querying XML can be done in various different manners, all of the methods examined here are standardized toolsets available for querying XML files. The tools we will be giving examples of when querying data in XML are XPath, XQuery, and XSLT. XPath is a tool with a notation similar to that of JSONPath discussed earlier. XQuery has more in common with SQL-database querying languages. And finally, XSLT (Extensible Stylesheet Language Transformation), which is a way to transform data in XML files, with the possibility of using XPath to extract specific data to transform and display with certain markup, or as a different language altogether. Although these are standardized tools for querying XML data, XML is known to have pretty poor performance overall compared to transactional databases. This is

due to XML being extremely costly in terms of parsing time, even more so if some type of validation or schema is used (Nicola & John, 2003).

2.3.6 XPath

Through the "evaluate" function an XPath expression can be run against an XML document. This function takes several parameters such as schema and return type. However we will not go into depth on those in the following example. The "evaluate" function tries to find the path to the car model "Yugo" in the following example without a set schema. We are trying to extract the same type of data as in the JSONPath example to show the similarities between the two.

```
path="/persons/person/cars/car[model = 'Yugo']";
```

Unlike the JSONPath example, an absolute path was chosen here as opposed to doing a recursive search for the element "cars". In the JSONPath example the JavaScript method "indexOf" was used to find at least one occurrence of the word "Yugo", whilst in XPath it is applied through the path itself.

```
var iterator = xmlnav.evaluate(path, xmlnav, null,
XPathResult.ANY_TYPE,null);
```

The results are returned as an object which can be iterated through with the "iterateNext()" function, a function part of the XPathResult interface.

```
var nodes = iterator.iterateNext();

while (nodes) {
          document.write(
          nodes.parentNode.parentNode.childNodes[1].textContent );
          nodes = iterator.iterateNext();
}
```

This example prints out the name of the owner of the car model "Yugo". This is done through finding that specific car model, then traversing back to the parent node (cars), the next parentnode (person), then the first element child node of that node (firstname), and lastly retrieving the text content. This could also be done by traversing back and using "getElementsByTagName" to find a specific element. A different progression is used compared to the JSONPath examples. Regardless, both examples find their target through a supplied query, and then use the path of the result to derive what first name to display; either by traversing backwards, or using part of the path.

2.3.7 XQuery

XQuery is a querying language for XML data similar to how a Structured Query Language (SQL) interoperates with data located in a database. XQuery uses XPath expressions and some common programming paradigms to extract data from XML files.

```
for $results in doc("xmlnav.xml")/persons/person
where $results/age>50
order by $results/age
return $results/firstname
```

This example would return the first name of every person with an age greater than 50 and orders the results by age. XQuery requires something to process the queries however, such as

a PHP or Java program with an XQuery implementation (World Wide Web Consortium, 2011a). XQuery can therefore not be run as a pure browser-based application. Strides have been made to enable XQuery support for browsers so they can make use of XQuery supported client-side applications (Fourny, Pilman, Florescu, Kossmann, Kraska & McBeath, 2009). As mentioned by Fourny et al (2009); XQuery, a super-set of XPath – something already supported by web browsers – would be a great addition to the browser-family to help navigate and manipulate the DOM.

Due to XQuery's aforementioned requirement and the need for this application to be entirely JavaScript driven, and due to its inability to be run locally without any need for Java applets, server software or third party browser extensions, XQuery will not be used.

2.3.8 XSLT

XSLT is often used for transforming or translating data from one format to another. This transformation is most commonly done to facilitate the use of some kind of cross-application communication, either through the web or regular operating system specific programs. Groppe and Böttcher (2003) showcase one such use where the hierarchy of an XML document is transformed to another hierarchy, containing only the results of a query in order to alleviate data transformation and transportation costs. XSLT uses XPath to navigate the DOM. We will show some examples of how XSLT could be used to create a new document from existing content.

The XML file needs to link to an XSL file to apply the style sheet, or the transformation, to the XML content. This is done at the top of the XML file.

```
<?xml-stylesheet type="text/xsl" href="persons.xsl"?>
```

The contents of the XSL file generally start with a version declaration as well as what encoding the document will be using, intended for cases of special characters. Next, the style sheet version is declared. This will most likely be 1.0 as 2.0 is not currently supported by any browser and requires another transformation engine. Also included is a link to the official namespace from the World Wide Web Consortium. Anything prefixed with the "xsl:" namespace is a special function and not user customized tags or content that should exist in the transformed document.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

The base of an XSLT transformation is to set a template for initiating the transformation. This template matches a specific XPath and starts its content from within. In this case for example, we start from the "persons" element, and apply this template or transformation to all entries within.

```
<xsl:template match="/persons">
```

Inside this template, code could be written to select several instances or, specific values or a combination thereof, all extracted with XPath. In this example, each "person" element (residing inside the template match, "persons") will create an "employee" tag and a "name" tag.

```
<employees>
    <xsl:for-each select="person">
```

```
    <employee>
    <name>
```

This aforementioned “name” tag will contain the values of “firstname” and “lastname” from the XML document. As all white space is stripped on transformation, the text function from the XSL namespace is used to create a space between first and last name. It is possible to specify specific elements that should either strip or preserve namespace, but by default, “text” is one that always preserves, while all others strip.

```
      <xsl:value-of select="firstname"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="lastname"/>
    </name>
```

The following shows an example of selecting the newest car out of every person’s car(s), and then displays the model name. Afterwards all tags are closed as usual, but the customized makeup, i.e. the non-XSL namespace tags, is entirely optional and user customized.

```
    <newestcar>
      <xsl:value-of select="cars/car/model[not(../car/year >
year)]"/>
    </newestcar>
    </employee>
    </xsl:for-each>
</employees>

</xsl:template>
</xsl:stylesheet>
```

The result of this transformation is shown below. As previously mentioned, all white space is removed so indentation and white space has been added to ease readability. The only remaining white space is between the first and the last name.

```
<employees>
          <employee>
                    <name>Arne Smith</name>
                    <newestcar>Volvo</newestcar>
          </employee>
          <employee>
                    <name>Gudrun Svensson</name>
          <newestcar>Saab</newestcar>
          </employee>
</employees>
```

Most modern browsers have support for XSLT transformations. Simply opening the XML file that has an XSL link to it will trigger the transformation, and show the transformed result instead of the XML file itself.

2.4 HTML 5

2.4.1 Local Storage

The next generation of the HTML standard, HTML 5, tries to meet the demands for demanding web applications by introducing more features. One such feature is the

introduction of a method to store data which may allow web applications to see improvements in performance and security (West & Pulimood, 2012).

According to Lawrence (2004), XML is always less efficient due to the considerable overhead from schema repetition, which is also corroborated by Nicola and John (2003). If the document is highly data-centric, the overhead may be up to four times higher compared to comma separated files. For huge data sets such as scientific or statistical raw data this may become a problem, especially when considering offline use of web applications due to the recommended 5 megabyte limit on local storage in the HTML 5 specification (World Wide Web Consortium, 2011b). Adhering to a smaller size, such as that imposed by local storage, could be beneficial for mobile web applications. Local storage could be used to facilitate offline use of a web application in areas of poor to non-existent wireless reception. Lawrence (2004) also states that due to the larger file size, all operations such as compression, display and most notably querying, will be less efficient. JSON does not forego the ability to be humanly readable to the extent that CSV or fixed-size files do, but saves some space due to the lack of closing tags being replaced by a closing bracket or commas.

The following is an example of how local storage is used to save data. “jsonQuery” is the same example object used previously in the querying examples. Local storage can process a few key methods such as “setItem”, “getItem” , “removeItem”. The “setItem” method takes a key and a value and stores it as a pair in the browser’s local storage, while “getItem” and “removeItem” just process keys and not values. As there is a structure already present in the example object file used, the entire object is converted to a long string before being stored in the local storage.

```
localStorage.setItem('jsonQuery', JSON.stringify(jsonQuery));
```

On subsequent page loads the previous bit of code could be removed and it would not affect the functionality of the next bits of code, unless a new browser is used or the browsers cache is cleared. Next up we retrieve the key “jsonQuery” from the local storage which would be the entire JSON structure as a string.

```
var retrievedObject = localStorage.getItem('jsonQuery');
```

This object is then parsed through with “JSON.parse” to reform the structure it had before being turned into a string.

```
parsedObject = JSON.parse(retrievedObject);
```

After parsing the object, specific key values could be changed and then afterwards turned back into a string to be entered into the local storage under the same or a different key with the new changes present.

```
parsedObject[1].lastname = "Smith";
localStorage.setItem('jsonQuery', JSON.stringify(parsedObject));
```

Before HTML 5 and the storage method introduced therein, the main way of storing data across sessions was cookies. Cookies store a text file on the computer with information about the user that is usually handled on subsequent visits to the site, such as username, passwords or the contents of a shopping cart. However as West and Pulimood (2012) point out, cookies can be used for malicious intent. For example, advertisement can be used to store so-called third-party cookies on the user’s computer. This third-party cookie can track the user across

domains, possibly letting advertisers know what product to target for the user. This data could be used for other purposes as well, all without the behest of the users themselves.

Another advantage of using local storage for web applications is that they can be made with offline capabilities. All necessary code can be stored and executed within the user's browser without accessing a server after the initial download. The application could reconnect after a connection is established again and proceed with transfer of data, but disconnection would not severely hinder the application's ability to function (Hsu & Chen, 2009).

Local storage on the other hand has a few advantages over cookies, one being the storage amount. The size can differ between browsers; the RFC specification states that a minimum of 4096 bytes per cookie is preferred for browsers to accommodate (Kristol & Montulli, 2000). That minimum amount is generally the limit for most browsers.

2.4.2 Offline Applications

Using local storage to access data locally is a good thing both from a security standpoint as discussed earlier, and for reducing transfer amounts (West & Pulimood, 2012). To use this in a web application, the user would still need to either visit the site using the local storage to access the application itself, thus requiring internet access, or save each part of the application manually on the computer for later use. To facilitate proper offline usage without requiring the user to save individual files, HTML 5 specifies an application cache method for saving, or caching, specific data on a webpage that can be accessed offline.

When no Internet connection is detected, the browser agent will automatically revert to the application cache and display the cached content. This is done without requiring an internet connection to navigate to the site in the first place to load the application. The limit for a site's application cache has no defined standard and varies depending on browsers both for desktop and mobiles. There are proprietary ways of installing web applications dependent on user agent, such as Chrome's Web Apps – using Google's own API bundled in a ".crx" file, hosted on their servers – and Firefox Open Web Apps. However HTML 5's application cache API stands as a cross-browser alternative. As of this writing, HTML 5's application cache is supported by all recent versions of major browsers as well as Internet Explorer as of version 10.

To use the application cache, one must create a manifest file, which is a simple text file containing some markers of what to cache, what to fall back on if the online resource is not available, what never to cache, and so on. We will present an example of a manifest file as well as some JavaScript functions for checking the cache.

The manifest file always starts with the first line in the example. A hash symbol is used for comments similar to JavaScript, but in this case it serves another purpose beyond just comments. In a manifest file a version number or date could be written under a comment to force an update on the manifest and all its associated files. If the manifest is unchanged, no files listed within will be fetched from the server, even if a new version of a cached image or file is present. Files to cache are written on separate lines.

```
CACHE MANIFEST
# Version number or date to force update
```

One could add the files to cache right under the previous entry but they can also be specified under the header below. This states that the index as well as the style sheet and JavaScript

files are to be cached, however the file that links to this manifest file (index.html in this case) is automatically cached so it is not explicitly in need of specification.

```
CACHE:
index.html
stylesheet.css
javascript.js
```

The next category specifies for files to always be fetched from the online server, even if the user is offline.

```
NETWORK:
census.php
```

Finally, the fallback category identifies what files are to be replaced with other files on the event that the user is offline. For example if the user is trying to access an online based chat, he would be redirected to a tailored offline page instead. This could work for entire directories as well, for example if a web application does not want to cache an entire photo album for offline use, every picture could be replaced with one single fallback picture.

```
FALLBACK:
chat.html /offline.html
```

The manifest file is linked to through the “html”-tag inside whichever file that is supposed to initiate the caching.

```
<html manifest="manifest.appcache">
```

Some key functions are provided to access the application cache with JavaScript. For example, “window.applicationCache.status” returns the status of the application through 6 constants. Ranging from 0 to 5 they define if the cache is un-cached, idle, currently checking the cache, currently downloading it, if an update is available but is not the newest cache, or if a cache is marked as obsolete.

2.4.3 Real-time graphics and HTML 5

The support and use of JavaScript has grown as more and more web pages use asynchronous JavaScript and XML to provide dynamic web applications (Ratanaworabhan et al, 2010). One way of providing these dynamic web applications is making use of the new canvas method from the HTML 5 specification. Canvas is used to declare an area on the application where JavaScript could draw. This area is a rectangle with a specified width and a height and all drawing functions such as lines, shapes, or even animation, will be employed within.

We will show an example of a simple animation accomplished through the use of canvas. The

example will be an orange colored line expanding across a horizontal plane until it reaches the end. **Fig. 1** shows a screen capture of the simple example application.

Fig. 1 – Screenshot of the example.

The variable values are left out as their function will be explained as they appear. The base for most canvas animations is to set up a timer or interval that will repeat a given function every specific amount of milliseconds. Here we use the function “setInterval”. This function repeats

the “animate” function every few milliseconds, 10 in this case. This gives the illusion of animation at each set interval, the drawing canvas is cleared, and a new drawing is made with new values to accommodate the new positioning.

```
function start() {
interval = setInterval('animate('+endPosX+','+endPosY+')',
intervalMs)
}
```

The “animate” function takes an end position for the X and Y-axis. We have chosen the end position of 400 for the X-axis, which is the width of the box where canvas will draw. The Y-axis stands at 25, making up for half the height of the box. This function verifies the current positions of X and Y with the designated end positions. If the destination is not reached, the length of the shape, or X position, is incremented by one. If it is reached however, the “clearInterval” method is called, stopping the interval started earlier so as to not consume unneeded resources when complete. Regardless of the position, another function is called to draw out the new shape, “drawShape” receives X and Y coordinates of either the finished end position (400 and 25) or the current position.

```
function animate(endPosX, endPosY) {
        if (curPosX == endPosX && curPosY == endPosY){
                clearInterval(interval);
                drawShape(endPosX, endPosY);
        } else {

                if(curPosX != endPosX){
                if(endPosX > curPosX) {
                                curPosX = curPosX + 1;
                        } else {
                                curPosX = curPosX - 1;
                        }
                }
                if(curPosY <= endPosY){
                curPosY = curPosY + 0;
                }
        }
    drawShape(curPosX, curPosY);
}
```

Seen here is the function “drawShape” which contains the bulk of the canvas functions. It sets up the canvas in the tag with the designated ID and sets a context, which can either be “2d” or “webgl”. Before a new shape can be drawn, the previous drawing has to be cleared from the canvas, which is done through “clearRect”.

“beginPath” marks the start of a new path, for which the coordinates are specified with “moveTo”. “lineTo” plots a path to the coordinates given, which is interchangeable with different path plots, such as quadratic curves or Bezier curves. Lastly, “stroke” is used to draw the plotted path. The whole process is analogous to a painter picking a brush size, color, then deciding where to start the first brush stroke on the canvas, how long the stroke will be and what shape it will take before beginning to draw. Text is also drawn out on the canvas to show what position the X-axis of “lineTo” is at presently. Y-axis is neglected in this example due to its fixed size.

```
function drawShape(drawX,drawY){
     var canvas = document.getElementById('canvas');
     var ctx = canvas.getContext('2d');

       ctx.clearRect(0,0,400,50);
       ctx.strokeStyle = "orange";
       ctx.lineWidth = 15;
       ctx.beginPath();
       ctx.moveTo(startX,startY);
       ctx.lineTo(drawX,drawY);
       ctx.stroke();

       ctx.fillStyle    = 'black';
       ctx.font         = 'bold 26px sans-serif';
       ctx.textAlign = 'right';
       ctx.textBaseline = 'bottom';
       ctx.fillText  ("X: "+drawX, 400, 50);
}
```

2.4.4 Infographics

Infographics is an effective way to combine text and visual representations of data in order to display a graphic that can be understood quickly by the user (Huang & Tan, 2007). Canvas could thus be used to render raw data found in storage formats and display it with a combination of text and graphics to create an infographic of the supplied data.

To further expand on this, an animated canvas could be used to convey an improved visualization for the user to imagine the continued rhetorical process after the data set or infographic has finished its display (Lengler & Moere, 2009). Hans Rosling presents this in a lecture presented by the BBC. An animated infographic is shown, showcasing the lifespan and income of two-hundred countries over the timespan of two-hundred years. The visual representation of an animated timespan guides the user into imagining the following years without a need for the actual data. Lengler and Moere (2009) explain that visual inference, such as an animated infographic, is an important component needed to make the data compelling for the user.

Lengler and Moere (2009) also believe that creating a well-conveyed rhetorical figure, i.e. by letting the user fill in the blanks when the animation ends, is not solved by simply programming a data mapping. A designer would be needed to select an apt model to convey a meaning to a specific audience and to select a proper data mapping. Thus we will limit ourselves on the implementation of the animated infographic – it will not necessarily appeal to a specific audience or have the most apt model used for the specific raw data selected.

Simply put, a canvas lends itself to the possibility of conveying a compelling infographic that could be understood easily by the user, while animating said canvas is an important marker for enhancing the users' imagination to visualize the continued animation beyond the actual data present.

2.5 Translating formats

A drawback noted by Lawrence (2004) is that these different, more space efficient formats are not standard XML. Therefore, for application compatibility reasons, it may be advantageous to provide a suitable translation from XML to JSON. Since the journal was

written, several new tools have been developed for JSON which may help the lack of standardized functions, such as querying or filtering, as well as the lack of widespread use compared to XML experienced in 2004.

As mentioned by Wang (2011) translating data could be necessary if the third party application that is proposed to interact with the new application uses a different serialized format. Wang (2011) also proposes that it may be advantageous to provide a generic translation between serialized formats, rather than programming specific translations to extract key data from one format. This enables the translation to be used in more applications and in a wider manner. However, one drawback could be the existence of superfluous code due to the dissimilar makeups of different formats. Thus, to provide a suitable translation without adversely affecting the speed, or simply negating the possibility of a performance impact, the translations will be tailored to suit the raw data provided.

2.5.1 Office Suite formats

Programs or web applications that display datasets or general scientific data in a spreadsheet saved with the OfficeOpen standard (ECMA International, 2011) use XML as the underlying structure. An example of this would be Microsoft Excel with their .XLSX, .XLSM and .XLTX file format (Microsoft, 2012). This might not be the most optimal solution for non-document based content such as scientific data, as explained by Lawrence (2004). A translation from the XML format to JSON could be profitable in terms of query performance and space for very large datasets saved in any program that uses XML.

For example, if saved with Microsoft Office Excel, a container file is created which contains various others, mostly XML, files. They depict the theme and style, which active cell and tab the user had and various other things of no note to the actual data contained. As for the actual data however, contained in “Sheet1.xml” or whatever name assigned in Excel, consists purely of the number values written by the user. All strings, for example if the top cell of a column is named “firstname”, are located in a different XML file containing all shared strings between all sheets.

```
<row r="2" spans="1:3" ht="21" customHeight="1"
x14ac:dyDescent="0.25">
          <c r="A2" t="s">
          <v>3</v>
          </c>
          <c r="B2" t="s">
          <v>4</v>
          </c>
          <c r="C2">
          <v>81</v>
          </c>
</row>
```

From top to bottom, contained within the A2 cell is the value ”3”, this refers to the other XML file containing all the strings, the third entry there is “Arne”, the name of the person whose row is seen above. “4” is the adjacent cell containing the last name. Lastly the value 81 refers to the age. This value, being a number inserted by the user, is not referring to any other file.

We will provide a dataset with a fairly suboptimal markup, but not to the extent of a file saved with the OfficeOpen standard with its original XML formatting. This dataset will be an XML

document which will be translated to an XML file with a smaller and more efficient markup as well as to a JSON file. These three files, the original XML, the translated XML and the translated JSON will be compared and analyzed both in structure and in file sizes. The translated JSON will then be tested against the aforementioned JavaScript querying tools while the translated XML will be tested against XPath. The tests will be to query and filter a statistical dataset and display it through real-time graphics in the three major browsers as of this writing: Chrome, Firefox and Internet Explorer.

3 Problem

Several independently developed tools have surfaced for the purpose of querying data or using path expressions in JSON, as there is currently no standardized or official language for JSON to accomplish this. A few of them are JSONiq, UnQL, JAQL, jLinq, JSONPath and jOrder, each with their own optimization, query capabilities, feature list and notations. Some of these tools have even found other similar fields of use, such as JAQL's ability to generate a query plan of a high-level query specification (Lee, Kuhl, Fowler, Robinson, Haas & Jermaine, 2009). As high-traffic sites have a high performance optimization requirement, the use of some lesser known third-party tools can be a dangerous affair. Most of these different tools use or are compiled in different languages, for example jLinq and jOrder uses JavaScript (Bonacci, 2012; Stocker, 2011), JSONPath uses PHP or JavaScript (Goessner, 2012), UnQL uses C (Hipp & Katz, 2012), JaQL uses Java (Ercegovac & Beyer, 2011) and JSONiq is an in-depth specification of a new version of XQuery that supports JSON (Robie, Brantner, Florescu, Fourny & Westman, 2012).

Plenty of benchmarking has been done on the XML alternatives (Lam et al, 2008) and using JSON for data exchange instead of XML (Erfianto et al, 2007), as well as benchmarks comparing JSON and XML (Nurseitov et al, 2009). None, however, that concern JSON's adaptability for query languages. All research so far point to JSON being known for its speed and efficiency and XML for being a better document-based format with less client-side processing, most likely due to the lack of JavaScript usage.

The subject worth investigating is whether there is a viable combination of technologies and methods to assist in successfully creating a web application capable of rendering real-time graphics at a reasonable speed, given the possible issues with browsers and storage format support. Real-time in this context is a soft real-time system where the application is continually querying new data between each display step. Furthermore, we investigate which of these discussed technologies are not only viable, but optimal. Next we explore whether an application should use JSON or XML as the main serializing language, if canvas will provide a suitable speed in combination, and lastly, if it will benefit from the use of offline and local storage. Then, we examine if an application could be built using purely client-side code to facilitate offline usage without requiring third-party browser extensions, server-based languages or any user intervention in terms of set up. Finally, we contemplate the browser's real world rendering of JavaScript code, as brought up by Ratanaworabhan et al (2010), and whether it affects the application's querying, filtering, drawing or caching.

The aim of this research is to evaluate the performance of the different techniques through a series of benchmarks and to evaluate the efficiency of the different JavaScript components as a complete application running on several browsers. The result of this could indicate whether using XML as a storage format is advisable even for non-document based data when there is a need for querying and filtering said data. This could also lend new credence to the study of Ratanaworabhan et al (2010) having differing JavaScript performance between browsers, especially considering the recent HTML 5 JavaScript techniques developed. However, seeing as XML has been the de-facto standard for data storage and applications on the web for some time, should JSON come out ahead, a transition would take time. This transition may be alleviated with the help of a translation between protocols to enable application communication despite storage formats.

3.1 Objectives

Our objectives are to select a suitable set of storage formats for analysis and presentation, and how they work and can be navigated. We have determined a way to query and filter the data in these formats by use of third-party tools and innate functions. We have selected a set of appropriate browsers for the tests to cover as much ground as possible in terms of browser market shares.

For JSON, we have examined the JavaScript-based tools (jLinq, JSONPATH, jOrder) for use in cross-platform and offline situations as well as being able to gauge the performance impact of different browsers' JavaScript handling when querying locally. For XML we have employed the innate JavaScript functions to handle and traverse the XML format.

We have discussed the performance of these different toolsets and languages. Secondly we looked at the technical complexity, complex queries and features present in each toolset. We compared and contrasted this to the existent supported and standardized XML along with its alternatives. In the end we tried to find out whether using these third-party solutions along with JSON leads to worse performance as a whole than using XML and XPATH for a similar data set. As we were using simple data structures, which are JSON's forte, the results may have been skewed into its favor. We examined whether JSON would still maintain that lead when introducing query languages into the mix. This could possibly speak for XML's tree-based nature, as well as the standardized nature of XML's XPATH and XSLT.

We created an application that translated a dataset from XML to three differently sized files, amongst them the JSON storage type and a slimmed down XML version. This translation was stored using a new local storage method introduced in HTML 5. From this dataset we employ several different predetermined queries, which are used to display certain attributes from the dataset graphically. These queries are analyzed to determine the effect of applying the queries on different data storage formats as well as their querying mechanism. The graphical display is powered by the HTML 5 Canvas method. We then benchmarked this implementation and analyzed the results with respect to the JavaScript performance in different browsers. The performances of the different techniques used are highlighted through the benchmark of the application.

4 Method and approach

The method employed is a purely quantitative one with an attempt to explain and investigate the performance for some widely used query functions in both XML and JSON. A test environment is built that translates as per the previously mentioned specifications. We have queried these translated files for information using the tools and techniques brought up. These query results filters the data to be displayed in a real-time graphical environment suited for the dataset used such as during an animated infographic. This graphical environment is built using Canvas from the HTML 5 specification.

The entire application is functional offline, which removes the support for some languages that could help with querying such as PHP, and functions assisting in space efficiency such as G-zip. Therefor the entire process is handled through JavaScript so it is suitable for a wide array of platforms, ranging from computers to mobile phones. To facilitate offline usage we employ more HTML 5 methods such as local storage for storing translated data as well as the application cache functionality present in the most recent versions of browsers. The application cache caches the entire web application to allow offline usage without the need to visit an online site before start up to run various scripts. The only alternative to successfully store data from a web application are cookies, which only allow an amount close to 4 kilobytes of data dependent on browser, per site. Fig. 2 showcases the process.

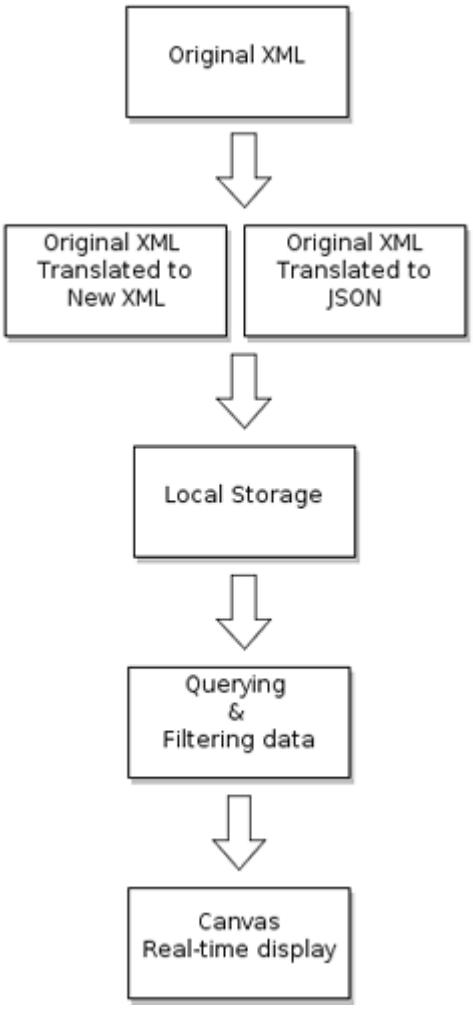


Fig. 2 - Diagram of the used process.

As there is no testing based on qualitative methods we do not see any issues arising from ethical or cultural needs. The benchmarking is entirely client software based and involves no specific type of users, as what is done with the data after parsing is out of scope of this article. We are also not associated with any organization supporting or developing any of the mentioned tools or languages. There are many more storage formats available for use such as comma-separated files and fixed-sized files (Lawrence, 2004). Testing all formats is not feasible so this article will focus on the two most contested formats (Nurseitov et al, 2009), with a balance between human readability and data storage efficiency.

A dataset is used to show various animated graphs displayed through querying certain attributes. One could filter ages into specific age groups, a certain genus or work group and overlay it with income figures. We believe this will really stress the filtering and querying ability of XML and JSON. We believe letting users manipulate the graphical display in real-time shows the performance difference between browsers and XML and JSON as users will anticipate an immediate response to their actions. Regardless of possible user interaction with the application, a predetermined number of queries will be run and measured in a quantitative way for more accurate performance results. It is however key to understand that the reason for speed being a factor is due to the end-users' need to have a compelling experience with the infographic; that is not slowed down by inefficient rendering. As explained by Lengler and Moere (2009), a truly compelling user experience comes from the careful selection of data and model to be used for the infographic. However, this will not be the focus of the article as no users are involved in the tests.

We employ a quantitative method analyzing entirely empirical data derived from the various benchmarks, similar to Ratanaworabhan et al (2010) or Lawrence (2004). These queries are presented in a graphical manner through the use of HTML 5 Canvas. There are alternatives to present a graphical display based on data, but this requires the aid of third-party additions such as Adobe Flash. Due to having focused on providing a complete offline experience without the requirement of extra browser plugins, these alternatives were not used.

Several queries have been written to properly ascertain the performance, usability and features of each aforementioned query language and their implementations. Complex queries have been written in case any optimization has been done for a specific search case, or to test whether support is lacking for a specific query. Benchmarking is also done on the XML's XPATH to establish a baseline for comparisons with the JSON benchmarks.

There is a possibility that certain browsers might have a larger or smaller market share depending on the type of user groups, country or company one might be associated with. The share of Chrome, Firefox and Internet Explorer is so far ahead of the competition that we do not expect this to be an issue (Grossman, 2012). The question whether one of these three browsers is ahead or at a certain spot in the market share list will be irrelevant for the tests as all will be tested in equal terms. This article will also assume the latest versions of the specified browsers, even though the statistics for browser shares account for all possible versions. The testing environment, which includes operating system and various computer specifications, will be the same for all browsers. While the user's gender may play a part in the perception of the graphical display of the dataset at the end, it should not influence the tests or results in any way.

We expect the results to be in JSON's favor as the dataset used is statistical non-document based data. The results may vary to a higher degree due to the nature of unsupported third-

party add-ons used to facilitate the queries and filtering. In addition, there are alternative data that one could place in a storage format, such as XML's favored document-based data. This data is not suited for a real-time graphical display and as such will not be tested.

Different browsers built with different toolkits have varying real world performance with JavaScript. Since most of the application is built using various new JavaScript methods, three different browsers will be tested (Ratanaworabhan et al, 2010). These chosen browsers are Firefox version 10, Internet Explorer version 9 and Chrome version 17.

5 Implementation

5.1 Dataset

The dataset used is publicly available and contains the census of income and various other variables for the Los Angeles and Long Beach areas for the years 1970 to 1990 (University of California, 2000). This dataset, similar to the one used by Lawrence (2004), is derived from the same source they chose to use for their article.

The census file is very large and will need to be truncated to be compatible with the HTML5 specification for local storage. This may cause the overall query and filter time, as well as the possible misrepresentation of data contained within the infographic display, to be considerably lower, but we do not expect this to impair our results.

```
48, Private, 40, 10, Some college but no degree, 1200, Not in universe, Married-civilian spouse present, Entertainment, Professional specialty, Amer Indian Aleut or Eskimo, All other, Female, No, Not in universe, Full-time schedules, 0, 0, 0, Joint both under 65, Not in universe, Not in universe, Spouse of householder, Spouse of householder, 162.61, ?, ?, ?, Not in universe under 1 year old, ?, 1, Not in universe, Philippines, United-States, United-States, Native- Born in the United States, 2, Not in universe, 2, 52, 95, - 50000.
```

Seen above is an excerpt from the census data-file. It contains a large number of categories which have been narrowed down to facilitate a smaller file-size. To derive which categories to keep and which to discard, we looked at the amount of axes the infographic could display. An infographic similar to Hans Rosling's presentation – a type of animated regression plot – could display X and Y axis, time (represented by animation) as well as the size and color of the entities. This would correspond with 5 columns from the data set.

The increment of time would best be represented as an integer, thus age would be a good candidate. X and Y are also best represented as numbers on a scale; due to the lack of continuous number columns contained in the data set a compromise had to be made. As such the education field was converted into a numbered scale corresponding to the order contained in the descriptive file for the data set, in total yielding 17 different ordered educations. A column named "instance weight" was used to show a number that corresponded to how many of the full population that row referenced. As such the size of the entities displayed could be influenced by that column.

Wage per hour was one of the columns filled with more continuous numbers that wasn't a self-reference such as instance weight. Therefore it was chosen for the second axis. A more suitable choice would have been income, however the income in the raw data set provided was truncated to facilitate a binary income description of either over or under 50 000. As the color of the entities need not be a number on a scale but merely a description of itself, the column for class of worker was used. The class of worker column indicated what work sector that row represents, in the above excerpt the class of worker is "Private".

5.2 Translating

The entire CSV file was imported into Excel to easier facilitate the deletion of entire columns without checking the exact delimiter amount between columns in the original comma separated file. All columns beyond the 5 listed above were deleted to clear up space. As the

size was still several times larger than the allowed size for local storage, a few rows had to be truncated too. Because the original file had close to two hundred thousand rows of data, and most of that data contained many zero values for entries such as wage per hour, the entire data set was sorted by wage. This was done to prevent random occasions where the translated structure would contain entries with no column data to handle. Finally, it was saved as an XML file with a mock up XML file as a mapping base.

```
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
          <census>
                    <age>35</age>
                    <classofworker> Private</classofworker>
                    <education> 11</education>
                    <wageperhour>8000</wageperhour>
                    <instanceweight> 2872.93</instanceweight>
          </census>
          <census>
          …
          </census>
</root>
```

This XML file, hereafter referenced as the base file, was used as the base to translate to differently sized XML and JSON files. Translating was done with the help of XSLT and could be done from the source csv-file instead of an intermediary XML file. This would require wrapping the entire csv-file in one root tag to enable access through the DOM. After that one would need to use XPath 2.0 functions to properly split up the delimiter and lines to columns and rows for XML and JSON. As no browser supports XPath 2.0 natively, this was procedure was not done. We will go through the translations and files produced from these translations in the following texts. The exact size differences between the files made will be discussed in the analysis chapter.

There is no perceived bias between the translations beyond the containing tag being shown as “c” in XML and “census” in JSON. This was done to further conform the two storage formats to the same principle of readability, both from a user and a programmatic standpoint.

5.2.1 JSON

Translation to JSON was done through an XML Stylesheet that removed white space (seen on the class of worker field in the above example), indents and conformed to the proper Javascript Object Notation, such as appropriate commas and semicolons where applicable.

```
<xsl:template match="/root">
var census = [
<xsl:for-each select="census">
          <xsl:if test="position() > 1">,</xsl:if> {
          "age":"<xsl:value-of select="normalize-space(age)" />",
          "classofworker":"<xsl:value-of select="normalize-
          space(classofworker)" />",
          "education":"<xsl:value-of select="normalize-
          space(education)" />",
          "wageperhour":"<xsl:value-of select="normalize-
          space(wageperhour)" />",
          "instanceweight":"<xsl:value-of select="normalize-
          space(instanceweight)" />" }
```

```
</xsl:for-each>
];
</xsl:template>
```

This translation would yield the following JSON object to be used in the application.

```
var census = [
 {
"age":"35",
"classofworker":"Private",
"education":"11",
"wageperhour":"8000",
"instanceweight":"2872.93"
}, { … } ];
```

Compared to the example of the base file there is a notable difference in size between the two while retaining the same amount of readability and functionality.

5.2.2 XML

XSLT was also used to translate the base XML file to another XML file to try and make it competitive with the JSON product, both in terms of minimalism and without losing functionality or readability.

```
<xsl:template match="/root">
<r>
<xsl:for-each select="census">
          <c>
          <age><xsl:value-of select="normalize-space(age)" /></age>
          <classofworker><xsl:value-of select="normalize-
          space(classofworker)" /></classofworker>
          <education><xsl:value-of select="normalize-
          space(education)" /></education>
          <wageperhour><xsl:value-of select="normalize-
          space(wageperhour)" /></wageperhour>
          <instanceweight><xsl:value-of select="normalize-
          space(instanceweight)" /></instanceweight>
          </c>
</xsl:for-each>
</r>
</xsl:template>
```

Since XML documents require a root node and for each row of the data set to be wrapped in an element. This requirement was mitigated by abbreviating those tags to the least characters possible. In a document-based file this would be harder to do as one would lose the descriptiveness of the child tags to the root, such as <header> or <body>. In this case however each row of the data set shared the same name and was always reoccurring. As such we did not need the <census> element to wrap every row of the data set and unnecessarily take up space that would not add to readability or functionality. Like with the JSON-file, all white space and indents were stripped to create a competitive file size.

```
<r><c><age>35</age><classofworker>Private</classofworker><education>
11</education><wageperhour>8000</wageperhour><instanceweight>2872.93
</instanceweight></c><c> … </c></r>
```

The above is the product of the translation. While minified it lacks some readability at a document level, however from the point of view of a program making use of the data its functionality is equal to the JSON document. This type of minified structured ended up saving a fair amount of space compared to the base file.

5.3 Local Storage

Both the JSON and XML data were stored in local storage, but not at the same time due to their size and the size restrictions imposed with each browser. As local storage only stores strings, both the data formats were serialized to one long string before storage. Using the data after the fact required a parser to serialize objects from the strings. JSON and XML have different functions to accomplish this.

XML data is downloaded through the XMLHttpRequest method; JSON on the other hand is simply appended as a JavaScript file in the document. The mechanics of this are brought up in chapter 2.2.

```
var objectSerializer = new XMLSerializer();
var stringXML = objectSerializer.serializeToString(xmlDoc);
localStorage.setItem('census', stringXML);
```

XML data is serialized as a string with XMLSerializer and serializeToString, unlike JSON which just used JSON.stringify.

```
var retrievedObject = localStorage.getItem('census');
var objectParser = new DOMParser();
var parsedObject = objectParser.parseFromString(retrievedObject,
"text/xml");
```

To retrieve XML data from local storage we need to un-serialize it and parse the string to bring it back to its previous XML format. While JSON uses JSON.parse, XML uses DOMParser and parseFromString to accomplish the same.

5.4 Canvas

The canvas for the infographic is made out of two parts. The first is the static UI which has no need for refreshes, as it contains no positions to update or animations. The other canvas is home to the actual moving parts. These two canvases are positioned over each other with different z-indexes through CSS to give the effect of one canvas, even though one of them is never updated and is restricted to static UI elements. The static canvas, named “board”, is the housing for the numbering of X and Y-axis, the descriptions for each color and the lines visually connecting each number on the X and Y axis partway through to the drawing field. As canvas X and Y positions originate from the top left instead of bottom left like most mathematical positions, an application that is supposed to start at the bottom left needs to adjust and most likely will need to have most of the Y axis size as the starting position.

```
var linesY = Math.floor((sizeY)/linePad);

for(i=0;i<linesY;i++){
    ctx.moveTo(sizePad-10,linePad);
    ctx.lineTo(sizePad+10,linePad);
    drawText(sizePad-10,linePad+10,Math.round((1-
(linePad/sizeY))*maxYvalue));
    linePad += padding;
```

```
}
```

To ease the user into viewing the positions of the animated entities on the canvas, lines are printed out that connect the numerical position of several positions on the X and Y axis. As these helper lines would start at the bottom left, the starting position of the Y axis would need to be the size of the axis itself as mentioned previously. These lines are then printed out depending on how many helper lines or numerical positions were needed, with appropriate padding between each, as seen in Fig. 3.

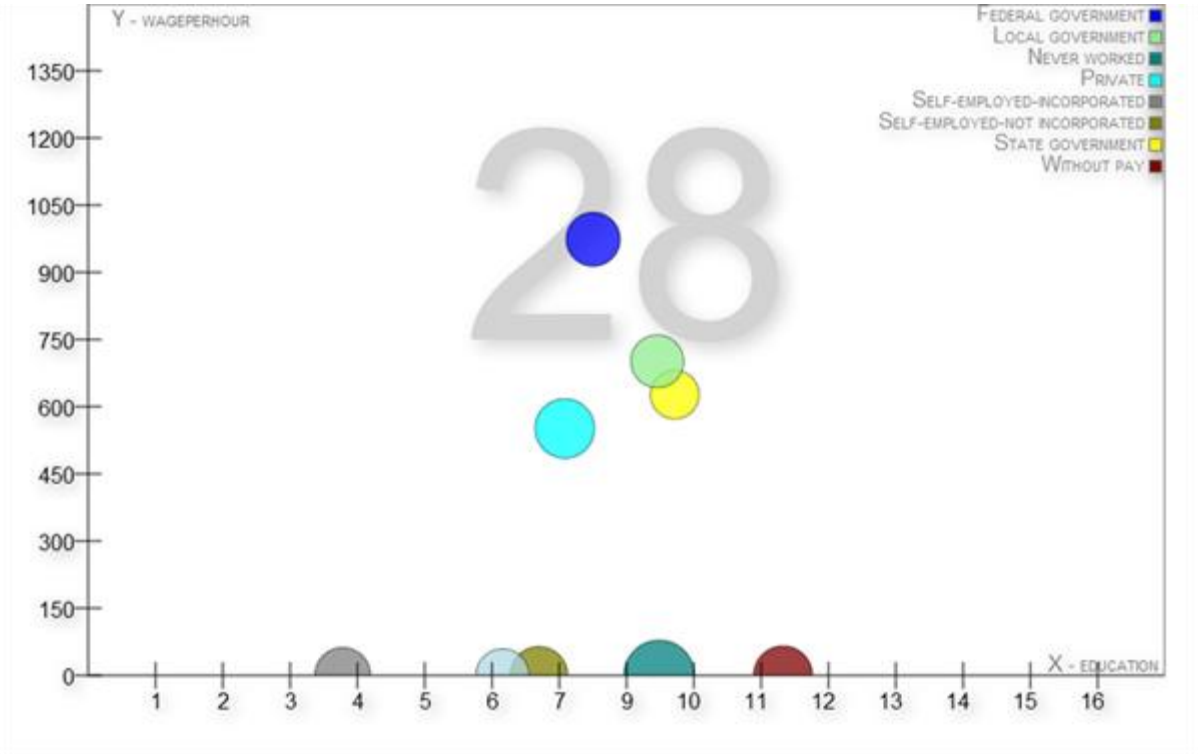


Fig. 3 - Screenshot of the application

The amount of lines needed is calculated from the size of the X and Y axis divided on the padding requested between each line, which could be a user defined value adjusted for that user's personal readability. The same mechanic as the above code is applied to the X axis with the exception of the X axis' positioning being incremented instead of the Y axis. Text of the numerical value is drawn out beside each line, with the value dependent on the current percentage position of that specific line multiplied with the max observed value of that axis.

```
for (var key in parsedObject) {
   if (parsedObject.hasOwnProperty(key)) {
      avgValue += parseInt(parsedObject[key][cSize],10);
    }
}
avgValue = avgValue/parsedObject.length;
```

To establish a baseline size for the circles or entities in the canvas, we walk through the entire data set and collect the average value of the variable specified as the one influencing the sizes, in this case "instance weight" from the data set. This then determines the sizes of the circles so they can be displayed in their correct size relative to each other.

```
function drawText(x,y,text){
```

```
    ctx.fillStyle    = 'black';
    ctx.font         = 'small-caps 16px sans-serif';
    ctx.textAlign = 'right';
    ctx.textBaseline = 'bottom';
    ctx.shadowColor = "grey";
    ctx.shadowOffsetX = 3;
    ctx.shadowOffsetY = 3;
    ctx.shadowBlur = 18;
    ctx.fillText  (text, x, y);
}
```

As an aside, the drawText function used in the example of drawing helper lines is not a standard canvas function. It is a function with a specific drawing context in order to easily enable all the callers of that function to use that same context. For example, they would use the same text alignment, font setups and so on, without needing to save and restore context before each text display. Without this or restoring the previous drawing context, the attributes used could spill over and give unwanted side effects if not cancelled out specifically, such as giving every line a shadow.

```
ctx.strokeRect(sizePad,0,sizeX,sizeY);
```

To properly separate the UI-type objects from the second animated canvas, we draw a rectangle around the field.

5.5 Animation

The animation is done on the second canvas and is refreshed and re-drawn with new positions every 20 milliseconds to emulate animation. Animation, as showcased earlier, is mainly done through the setInterval method, which re-runs the given set of parameters or function; in this case until the interval is cleared through the clearInterval method. To prevent each circle – which moves independently from other circles on the same canvas – from clearing the others, all of the circles had to be drawn out at the same time.

From this fact we work backwards to determine the best way to store the calculated positions of each circle's present and future. This meant that every circle needed to be a position of an array. This array needed to contain another array showing whether the associated values are current or future. All the possible positions for this three dimensional array would be "circle[0-9][0-1][0-5]". 0-9 for the amount of worker classes, 0-1 to specify present or future positions, 0-5 containing the X and Y positions, the color, the radius, and the movement speed of X and Y axis.

5.6 Algorithms and calculations

We determined early on that we needed to have one entity or circle per class of worker or color, and not per result found in the query. We needed to average out all the values found in the query for each specific circle. For example, if a query found two 80 year old people with the same working class attribute, but different education, wages and instance weight associated, an average of these 3 was calculated.

```
positionX = (xAvg / maxXvalue)*sizeX;
positionY = (1-(yAvg / maxYvalue))*sizeY;
scale = scale / avgValue;
```

This average then supported a calculation to determine the positioning and scale of the circles drawn. A percentage position of the max reported value on the X and Y axis was calculated so it could be used with the size of the canvas to pinpoint the exact position in relation to the boards reported values. To counteract the aforementioned caveat with the Y axis starting position we subtracted the decimal-based starting position by 1 before multiplying with the size of the board.

```
angle = Math.abs(Math.atan2(newPositionX-
currentPositionX,newPositionY-currentPositionY)));
xmove = Math.abs(Math.sin(angle));
ymove = Math.abs(Math.cos(angle));
```

The circles on the canvas needed to move at a dynamic speed to ensure that they arrived at their X and Y destination at the same time, to ensure a fluid movement. To accomplish this we used trigonometry to calculate the angle with arc tangent and then used the value of that to derive the X and Y movement vector with sine and cosine respectively.

```
if ((Math.pow((cX-x),2)<=animSpeed) && (Math.pow((cY-
y),2)<=animSpeed) && (Math.pow((cR-r),2)<=animSpeed)){
      finished[i] = 1;
      } else {
        if(!(Math.pow((cX-x),2)<=animSpeed)){
          if(x > cX) {
            cX = cX + xM;
          } else {
            cX = cX - xM;
          }
        }
        if(!(Math.pow((cY-y),2)<=animSpeed)){
          … Y Axis …
        }
        if(!(Math.pow((cR-r),2)<=animSpeed)){
          if(r > cR) {
            cR = cR + 0.1*animSpeed;
          } else {
            cR = cR - 0.1*animSpeed;
          }
        }
       newList[i][0] = [cX,cY,cR,c,xM,yM];
     }
drawCircle(cX,cY,cR,c);
```

The growth rate of the radius was set to a static number multiplied by the speed of the application, regardless of the time it took for the circles to reach their destination. A smoother implementation of this would be to calculate the required growth needed for each circle to reach its maximum size at the same time as it reaches its end position. Besides the angle of movement there had to be mechanisms in place for determining whether the circles had reached their destination, and be able to correct their movement path if they needed to go backwards to arrive at a new position. This algorithm was used for every circle's movement. It checked roughly if the current X or Y position subtracted with the end positions was equivalent to a small integer number. This gave each circle a small threshold where it could finish its movement.

Having a small integer as a threshold for finish and not just arriving at the exact spot prevents the circles from oscillating at their end position. For example, if it would pass by its destination by a single pixel and then try to correct this by going in negative space, it would pass by its goal yet again, and proceed to oscillate. Another way of accomplishing this would be to determine whether the circle is traveling in a negative or a positive movement path in the X and Y axis and then stopping movement once it is past its goal. When the destination is not reached, it will append the current position with the angle calculated previously. Then it sends off the result as the previously known position to enable the calculations to have a new position to work with.

```
for (i = list.length - 1; i >= 0; i--) {
  completed = completed + finished[i];
}
if (completed >= list.length) {
  clearInterval(interval);
  init(queryEngine);
}
```

As all circles needed to have finished their movement and arrive at their end destination before we could move on, we created a gating mechanism. This was to make sure that the application would only proceed once every single circle had reached its end destination. To accomplish this we used an array with a position for each circle. The value of said position would be 1 when finished and 0 when not finished. Once that entire array added up to the amount of circles it would be completed. For example, given 9 circles it would only proceed once the acquired sum of all values of the finished array reached 9. Once finished however, it would clear the interval to prevent any more updates from being applied to the canvas and then run the query function again to get a new age and new positions to reach.

5.7 Querying

Querying was done with the 4 different query toolkits mentioned previously. Focus was put on attempting to retain equality in terms of notation and results gained from each query. To support an animated axis on the infographic we focused on the time aspect of each entry, i.e. the age of each row. We also needed to receive results relevant for each circle, those being of the class of workers. Two variables, which were infused with new values for every iteration, were used in the query of each toolkit. In the following examples the variable “animVal” is the age variable which remains static for that entire query segment until the animation is finished and a new age is needed.

The array “cColorOptions” is simply an array holding all the class of workers and their associated colors, that is, each circle on the canvas. The entire query is looped through for each class of worker, including age, even though age remains static. A better implementation from a performance point of view would be to first query the entire age segment, and then proceed to query the result of that for each class of worker. However in the interest of better gauging the speed of each query toolkit we also chose to query the age each time, increasing the rows processed each age, even though this was unnecessary.

5.7.1 jLinq

jLinq is processed similarly to the examples in 2.3.2 with the exception of requiring an exact match rather than a partial one. As mentioned earlier, a query is done on both the age and

class of worker at the same time, even though the age value will remain static for this iteration.

```
.equals('age', animVal)
.equals('classofworker', cColorOptions[i][0])
.select();
```

The notation is simple and easy to work with and easily expandable to encompass more attributes.

5.7.2 JSONPath

The JSONPath query is a bit different from the example used in 2.3.4 as there is no need to get the path of a result because the structure is a flat table without any sub-elements to traverse to.

```
var results = jsonPath(parsedObject, "$[?(@['age']=='"+animVal+"' &
@['classofworker']=='"+cColorOptions[i][0]+"')]");
```

In this case the notation and expanding upon search terms are a bit more complex. A quick rundown of the example: Search the root object ($) with a specific filter (?), search this level (@) for where age is a certain number, and so on. This example shows that JSONPath is more focused on XML-like JSON documents that aren't just plain flat tables but more multi-leveled with deeper paths.

5.7.3 jOrder

The ability to index the jOrder table was not used in the application. It is best used with data that can make use of unique key values. Aside from that we also felt that an impromptu query would be better suited when pitted against the other query alternatives. It is possible the use of indexing could provide a better result but as no other implementation indexed or cached data other than the browsers' own implementation, we decided it would give fairer results to forego the use of indexing. In addition, indexing should be done once on the table before queries as opposed to in conjunction with the hundreds of queries which the application would yield. This would give jOrder an edge because it would get run time on the data ahead of the other toolkits.

```
var results = table.where([ { "age": ""+animVal+"", "classofworker":
""+cColorOptions[i][0]+"" }], {renumber: true});
```

There was no additional notation needed from the examples in 2.3.3 to query this specific data set. Simply adding another attribute to the existing table search was enough to allow filtering by two different values.

5.7.4 XPath

Querying with XPath brought up a few more complications due to browser incompatibilities and because the application, being JavaScript-based, was fed JavaScript objects.

```
var iterator =
xmlDoc.selectNodes('/r/c[classofworker="'+cColorOptions[i][0]+'" and
age="'+animVal+'"]');
    for (y=0;y<iterator.length;y++) {
          var age = iterator[y].childNodes[0].text;
          var classofworker = iterator[y].childNodes[1].text;
          var education = iterator[y].childNodes[2].text;
```

```
            var wageperhour = iterator[y].childNodes[3].text;
            var instanceweight = iterator[y].childNodes[4].text;
            results.push({"age":age, "education":education,
            "classofworker":classofworker,
            "instanceweight":instanceweight,
            "wageperhour":wageperhour});
```

For Internet Explorer 9 there is no support for the evaluate method which other browsers rely on to parse XPath queries. Instead there is selectNodes which do the same thing with less parameter choices. The result of this is returned as an array which needs to be iterated through to extract all the results.

```
var iterator =
xmlDoc.evaluate('/r/c[classofworker="'+cColorOptions[i][0]+'" and
age="'+animVal+'"]', xmlDoc, null, XPathResult.ANY_TYPE, null );
var thisNode = iterator.iterateNext();
     while (thisNode) {
            var age = thisNode.childNodes[0].textContent;
            var classofworker = thisNode.childNodes[1].textContent;
            var education = thisNode.childNodes[2].textContent;
            var wageperhour = thisNode.childNodes[3].textContent;
            var instanceweight = thisNode.childNodes[4].textContent;
            results.push({"age":age, "education":education,
            "classofworker":classofworker,
            "instanceweight":instanceweight,
            "wageperhour":wageperhour});
            thisNode = iterator.iterateNext();
```

Non-Internet Explorer browsers make use of the evaluate function to parse XPath, which returns an XML-like structure with nodes and elements that can be traversed through with getElementsByTagName. Alternatively simply go through the child nodes manually if the returned result has a known structure, which in this case it has. The evaluate function also allows the use of iterateNext which enables easy looping through the results.

In either case the results are pushed into an array of JavaScript objects similar to the results of the JSON-based query toolkits, to facilitate an easy transition to the JavaScript-based application. This new array is then sent on to the canvas to be displayed.

6 Results and analysis

6.1 Benchmarks

In order to ensure repeatable and safe conditions, a few set procedures were done between each test. Browsers were refreshed between each try for possible cache and memory related issues. No consoles or debuggers were present during the benchmark as they have had documented effects of significantly slowing down parsing and rendering in some browsers due to possible debug processing. In order to benchmark with no console or HTML DOM output, all the benchmark results were stored in an array which was later accessed through each browser's console window, post completion of each test. No add-ons were present in any of the browsers to prevent possible interfering with performance. No third party toolkits such as jQuery were used to ensure that any possible issues and performance evaluations would easily be traceable to the source and the different query engines tested and not to conflicting JavaScript issues.

The benchmarks were run five times for each file-size on each browser. While it may seem like a small enough sample size for queries, in the context of this application five times is multiplied by the ages searched as well as the class of workers. Each file-size in the end received 4050 queries per browser, or 810 queries per test run. The results of these benchmarks would allow a comparison between query engines, and between browsers. However, it would not allow a proper gauge of performance between file sizes as a larger data set may affect the infographical display and cause longer or shorter animations to be needed, thus increasing or decreasing the overall time for a test run.

6.1.1 File size comparisons

There was a notable decrease in storage space required when examining different formats post translation. The original XML file – from which the JSON and XML translations mentioned previously were derived – was split up into three different sizes. The targeted amounts were roughly 5mb, 2,5mb and 1mb. The upper limit was set to approximately what most browsers specify their local storage limit as and then at lower intervals from that point.

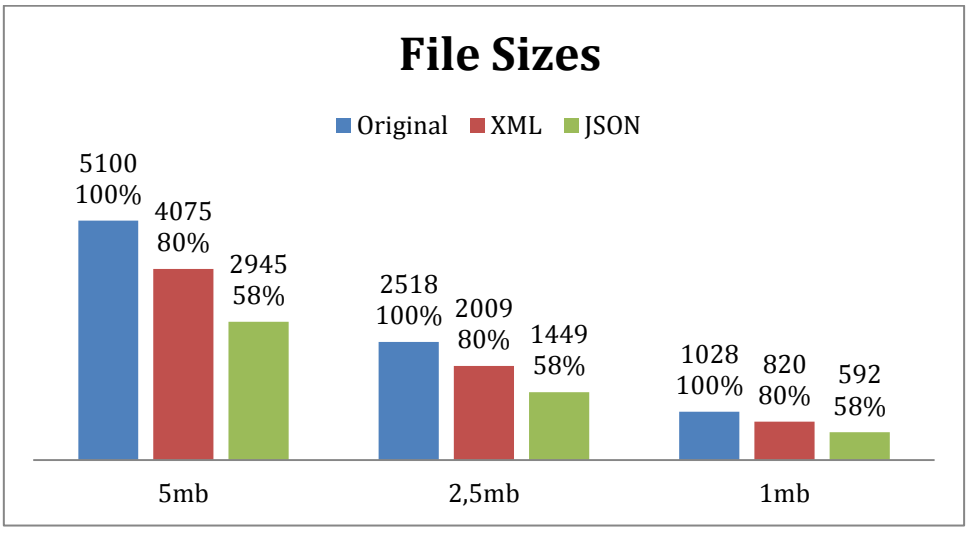


Table 1 - Original and translated data formats.

When translated with the translation and XML makeup listed in chapter 5.2, XML, while the same format, saw a noticeable increase in space efficiency. This depends greatly on the source

file the translations originate from. However, with just stripping whitespace and converting the wrapping tag of each row into a smaller format there was a lot of space gained. As seen in Table 1, the largest translation size went from 5100kb to 4075kb with these simple changes. It is worth reiterating that this depends solely on the source XML file's makeup, in some cases there might not be much to be gained. Conversely, if one would translate completely from an Excel file with the excess markup brought up previously, there could be even more space gained.

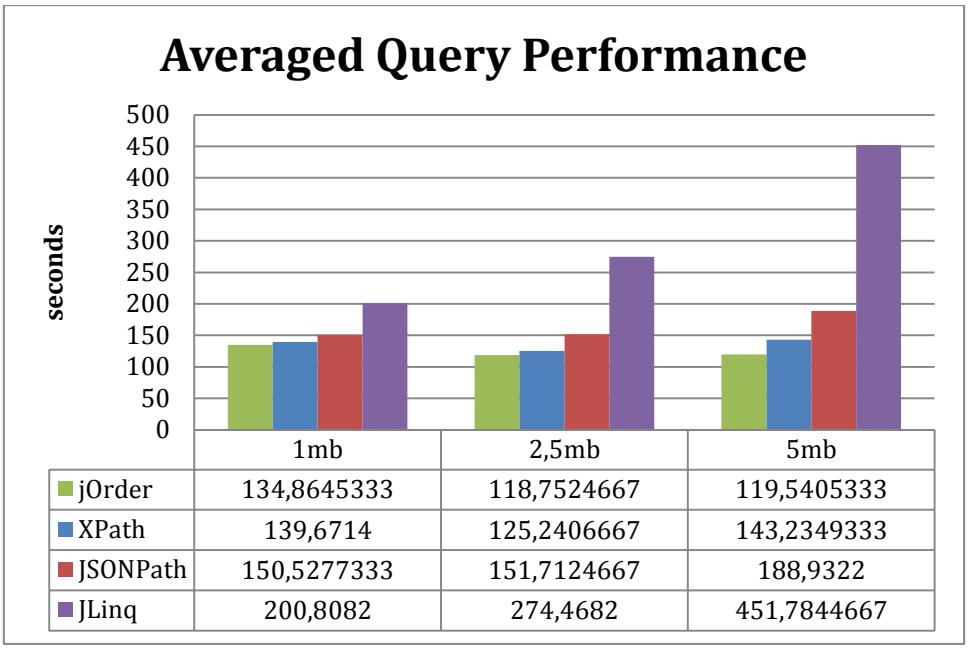
A more comprehensive view would be gained from comparing the translated XML and JSON formats as they share similar structure and readability for the data and give a fairer comparison. In the case of the 5mb source file, the JSON file turned out 1130kb smaller than the XML file. For 2,5mb it is 560kb smaller, and finally for the 1mb category it is 228kb smaller.

This ends up being roughly 27% space saved compared to the translated XML file, regardless of file size. This amount of space saved can matter a great deal when taking into account the small limits of the local storage and application cache functions. It is possible to fit more data and it could be the factor between being able to run offline and not. This corroborates the XML overhead concerns brought up by Lawrence (2004) and Nicola and John (2003). This makes JSON the favored storage format of choice for any case where space limitations are an issue. Another advantage found is the seamless transition from data stored in JSON to applications built using JavaScript. JSON removes the need to traverse a DOM or convert child nodes to adapt to the structure of a JavaScript application, as it most likely shares a similar construct to the application regardless. The results brought forth by Nurseitov et al (2009), where they investigated the CPU, memory utilization and object transfer times, indicated that JSON used fewer resources while being faster than its XML counterpart. This coupled with our results gives a very favorable appearance for JSON.

6.1.2 Query Engine comparisons

We queried jLinq, jOrder, JSONPath and XPath in each of the three browsers separately. All data was stored in local storage before querying. In the case of the 5mb file we found that Chrome, while sporting a 5mb local storage limit, cannot handle any size above 2.49mb, or roughly 2.6 million lines. This is most likely due to the fact that Chrome measures the limit after the data has been stored as UTF-16. This means that the size taken up on the user's hard drive is indeed 5mb, however only half of that is useable for developers.

In this application it meant that Chrome was not able to run the benchmarks of the highest file size. This was circumvented by not storing the data in local storage before querying. This is not feasible for an offline application, but was done in the interest of still being able to test the query capabilities and performance of that browser at that large file size. Judging by the test results, there may not be a performance change from querying the local storage or separate files. Both are most likely loaded into memory or browser caches by the time the application has loaded in the case of online usage.



	1mb	2,5mb	5mb
jOrder	134,8645333	118,7524667	119,5405333
XPath	139,6714	125,2406667	143,2349333
JSONPath	150,5277333	151,7124667	188,9322
JLinq	200,8082	274,4682	451,7844667

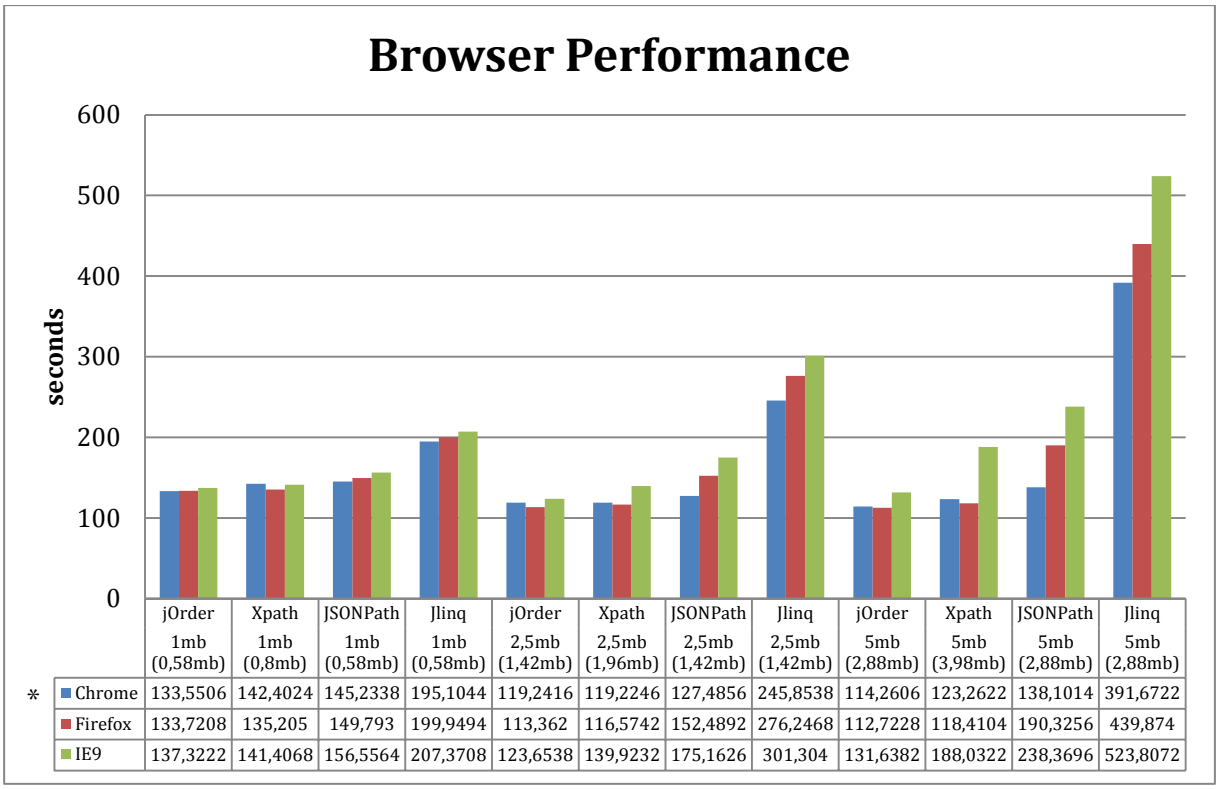
Table 2 - The average query performance of all browsers.

To compile a comparison for the four query mechanisms, we averaged out the five test results done on each toolkit and every browser into one number. This is to discover the overall effectiveness of each query toolkit, regardless of browser. In all three tests on all browsers jOrder came out on top with the fastest query performance. As mentioned earlier, the size of the data set could influence the animation speed and thus change the rendering time, yet a certain speed trend can still be seen in the results. This means that in reality jOrder and XPath do not inversely scale with data. The extra amount of data going from one size category to the next changes the averages in such a way that the animation finishes at different speeds.

As the file sizes grew larger, the applications processing time with jOrder remained roughly the same, showing a capability for scalability. The second fastest was XPath, which showed a surprisingly even performance between the different file sizes and overall speed, given that the data set was not of a document-based nature. JSONPath was not far behind, but even with the similarities between XPath and JSONPath the inbuilt browser query implementation is ahead. jLinq was the slowest of the query implementations measured by a fair margin. The easy notation speaks in its favor, but the lack of scalability for larger data sets puts it far behind the rest. It is worth mentioning though that while jOrder surpassed XPath in this benchmark, XML is able to traverse deep XML structures while jOrder is designed to work on a more table-like structure.

6.1.3 Browser comparisons

As opposed to the query engine comparisons, these results will not be averaged out over every browser and will provide a more in-depth look at how each browser performed on the various queries.



*		jOrder 1mb (0,58mb)	Xpath 1mb (0,8mb)	JSONPath 1mb (0,58mb)	Jlinq 1mb (0,58mb)	jOrder 2,5mb (1,42mb)	Xpath 2,5mb (1,96mb)	JSONPath 2,5mb (1,42mb)	Jlinq 2,5mb (1,42mb)	jOrder 5mb (2,88mb)	Xpath 5mb (3,98mb)	JSONPath 5mb (2,88mb)	Jlinq 5mb (2,88mb)
	Chrome	133,5506	142,4024	145,2338	195,1044	119,2416	119,2246	127,4856	245,8538	114,2606	123,2622	138,1014	391,6722
	Firefox	133,7208	135,205	149,793	199,9494	113,362	116,5742	152,4892	276,2468	112,7228	118,4104	190,3256	439,874
	IE9	137,3222	141,4068	156,5564	207,3708	123,6538	139,9232	175,1626	301,304	131,6382	188,0322	238,3696	523,8072

* 5mb variant benchmarked online.

Table 3 - Browser performance of every query toolkit and file size.

This chart shows a consistent scalability issue with Internet Explorer 9's handling of both XPath and JavaScript. This could be for a number of reasons, such as the evaluate function scaling better than IE9's selectNodes, as well as a poorer JavaScript performance overall. For XPath, IE9 has a lead for the smaller file size but as the file grows larger, the processing time steadily increases in comparison to the other browsers.

An interesting piece of data is the difference in processing times between Firefox and Chrome for different JavaScript based toolkits. For example, Firefox is consistently faster than Chrome for jOrder and XPath but drops to a lower place for jLinq and JSONPath. This could be due to a multitude of different functions used in each toolkit that are more or less supported in one browser than another. Richards, Lebresne, Burg and Vitek (2010) bring up some examples of what functions or methodology could affect browser performance, such as "eval" calls or undeleted properties. More research would need to be done to pinpoint the exact differences but it stands to reason that the claim made by Ratanaworabhan et al (2010) of different real world performance for JavaScript could even apply to toolkits attempting to fill the same function. Overall the browser performance is equal to some extent between Firefox and Chrome while IE9 remains the slowest contender with the exception of small-scale XPath queries.

The variance of data collected in all the benchmarks usually sat around 10-100ms for most tests. The highest discrepancy in all of the test runs was reported at roughly 3000ms, which was for the 5mb XPath benchmark. This means that the queries were very constant between test runs, with a 99.8% precision most of the time and 99% at worst.

7 Conclusions

We selected a suitable set of storage formats and presented how they worked and could be navigated. The results of this examination showed JSON the best storage format when notation, file size and compatibility for web applications were to be a concern. Provided with third party tools and native implementations we navigated these storage formats and filtered their contents based on specific queries. These queries were displayed through a canvas in an infographical fashion. The result of the benchmark on this complete application strongly suggests that JSON is the favored storage format for any type of web development. It also shows canvas as a powerful tool for building infographical displays through animation. Furthermore when JSON is used, the best toolkit to use for any browser out of those tested was jOrder. We chose appropriate browser coverage for these tests to be able to cover the greatest number of users possible. After an analysis on the browsers' usage of the application we came to the conclusion that both Firefox and Chrome have toolkits and implementations that work better with their respective engines. The use of Internet Explorer 9 is not recommended.

The translation proposition brought up by Wang (2011) was shown to be a viable course of action for a fair storage gain in this article. The use of translation could alleviate any compatibility concerns between third-party applications utilizing different storage methods. This article also showed that depending on how each browser handles specific functions, either inefficient ones (Richards et al, 2010) or specific benchmarked ones (Ratanaworabhan et al, 2010), the effect and performance varied between browsers despite the application always performing the same task.

With querying as a major feature of standardized XML now being viable in JSON as well, Lawrence's (2004) concern of standardization might not be a considerable issue anymore. Especially when bearing in mind the results of Nurseitov et al (2009) for JSON being a more efficient alternative on a CPU and memory utilization level. In conjunction with the new found query capabilities, and the interoperability with applications utilizing objects and browser compatibility, JSON is a very competitive alternative to XML.

While not the main focus of this research, it lends credence to Lengler and Moere's (2009) assertion of the lacking viability for a compelling infographic display which was not the result of a study on the data or model.

When there is a need for creating a web application to quickly filter and render raw data through a graphical display, this article maintains that using JSON as a storage format along with jOrder as the filtering mechanism is the best option. XPath and JSONPath are still very viable alternatives, especially if there is a need for traversing a deeper path and speed is less important. Canvas has also shown itself as a viable technology for displaying animated data. It is entirely possible to build an application that has these features while still able to provide a complete offline experience.

7.1 Contributions

In this article we have examined the viability of querying JSON in comparison to XML's native querying implementations for use in offline-enabled web applications. XML and JSON have been studied and examined in other fields but investigating query performance between the two is new ground. Nurseitov et al (2009) have examined JSON's overall speed and investigated possible overhead issues. While Lawrence (2004) have maintained that XML

may outperform JSON when faced with more document-based semi-structured data. This data-type is more suited for XML and its markup notation.

Nicola & John (2003) have investigated query performance in XML with regards to the extra overhead of DTD (Document Type Definition) and schema validation, and how parsing XML may present performance issues compared to transactional database processing. Lam et al (2008) have benchmarked XML's ability to handle stream-based simultaneous XPath engines. Erfianto et al (2007) propose using XSLT to transform XML to JSON for smaller exchange format for use in context-aware applications. Finally, Wang (2011) investigated the viability of using a translation layer to enable communication between XML and JSON implementations. Furthermore, in research concerning offline-enabled web applications, the focus has been on the security implications thereof (West & Pulimood, 2012), as well as enabling automated object persistence to allow applications to function offline (Cannon & Wohlstadter, 2010).

We have shown that JSON is a leading choice for use in web applications even when they have a need for querying data. This reinforces the overall support for JSON as received by many benchmarks and analyses. This work provides a reinforcing standpoint for the use of JSON, alongside the positive results found by Nurseitov et al (2009) of JSON's speed and small overhead. It also lends credence to Erfianto et al (2007) and Wang (2011) regarding their proposition for translations to JSON. These translations would enable faster and smaller exchange format, while still retaining the ability to communicate with XML-based applications. Besides alleviating possible integration issues across platforms, this makes it easier to reduce the usage of XML for data-focused applications while retaining existing capabilities. XML being perceived as the standard was brought up as a merit by Lawrence (2004) but through the result of this article, alongside translations, we maintain that this is not a deterrent for JSON use any longer.

Ratanaworabhan et al (2010) compared JavaScript behavior in tailored benchmark functions against real world applications, ran on different browsers. While they had a different focus on their benchmarks, our result of varying browser performance adds to their article. Similarly, Richards et al (2010) analyzed different pitfalls in JavaScript code which could be applied to the different toolkits we tested.

Finally, we have shown that a HTML 5 Canvas can provide a suitable animated infographic experience. Furthermore, while it was not as a result of our benchmarks, our article does seem to support Lengler and Moere's (2009) suggestion of needing a proper analytical study into determining proper characteristics and models for use in infographics.

8 Discussion

The aim of this research was to evaluate the performance of select technologies through a series of benchmarks on an application utilizing these technologies on different browsers. The result of our article shows the favored techniques and storage formats to use in an application that employs a query filter with a performance requirement. In our application the performance requirement was to display an infographic where fast queries were paramount to display canvas animation without slowdowns.

Despite lack of standardization in the field of querying and filtering data, JSON has shown to have competitive performance to XML's XPath alternative. The results gleaned from the JSON benchmarks differed to a high degree. The third party toolkit jLinq had a high execution time, while the other toolkits found themselves closer to XPath and in the case of jOrder, surpassed it.

The browser results were surprising as there was no clear winner despite the similarities of the toolkit functionalities. Internet Explorer was slowest in most tests, however it did surpass Chrome barely for the 1mb XPath test, it did manage to come close on the remainder of the 1mb tests. It did seem to have a scaling problem however, or any scaling issue prevalent in a toolkit manifested itself in a higher degree for Internet Explorer. In the following chapters we will discuss the results more in depth as they pertain to each of the described problems and objectives.

8.1 XML and JSON for web development

One advantage of JSON is the ease with which it can be integrated into current web applications. Because most web applications are built using JavaScript, using a storage format that has the notation and functionality of objects within JavaScript is a useful advantage. XML on the other hand is a bit harder to use alongside a JavaScript based web application and requires browser sensitive expressions as opposed to JSON which is handled the same in every browser. This also makes it easier to provide compatibility across browsers.

It was apparent in this experiment how the JSON notation benefitted the development of the application. This was mostly due to having the same or very similar object structure, as well as storing and traversing being the same as any JavaScript application. Due to the toolkits for JSON returning the same object structure after a query, any compatibility issues between the application and the filtered results were alleviated. In the end, the application and JSON stored data had a high amount of interoperability.

8.2 Querying mechanisms

The results brought forth by the benchmarks on the query toolkits were surprising. One assumption was that the browser's native implementation of XML querying would put it ahead of third party made toolkits, which it mostly did, with the exception of jOrder. Across all browsers jOrder had a fair lead over the other toolkits and was always somewhat ahead of XPath. All toolkits have different ways of querying data, possibly both in methodological use and in function usage. Some might have been making inefficient calls as brought up by Richards et al (2010). This could very well explain the differences exhibited, as well as how XPath, and the browsers native implementation thereof, can come out ahead of two toolkits. A more interesting comparison could be between XPath and JSONPath, whom both use path expressions and similar ways of querying data, but on different storage formats. XPath had a

fair lead on the larger file sizes when compared to JSONPath, which might be the browser helping out with the client side processing of the XML data.

While jOrder came out ahead in the benchmarks there is still the question of its capabilities, especially in relation to JSONPath and XPath, who both are able to work in document-based structures with non-ordinal fields. This was not tested in the benchmark, or in this application, as data-based structures are more suited for storage of raw statistical data. However, if this is a required feature for the application that one might be building, it leaves out the use of jOrder, at which point XML and XPath becomes the fastest way of processing the data, at the cost of storage.

The usage ease of notation could be a fairly relative question. Nonetheless the notation used in path expressions for JSONPath and XPath were quite a bit more complex than the notation used in jOrder and jLinq. This is most likely due to the fact that JSONPath and XPath are designed to traverse deep paths and not a shallow data-structured list.

There is a visible trend of scaling issues in some toolkits such as jLinq with regards to file size. It is shown clearly when the execution time of jOrder lowered while it increased for the others. This could mean that jOrder has no scaling issues of any kind. After extrapolating the results of the other toolkits and comparing, it is clear that jLinq, JSONPath and XPath have more problems with large datasets than jOrder.

We believe however that benchmarking the entire application has its merits. Treating the entire application as one component gave a more real world applicable result. Similar to how Ratanaworabhan et al (2010) compare standardized benchmarking functions to how some deployed sites perform. This gives more insight into how each part integrates and works together as a whole rather than individual browser capabilities as isolated functions.

8.3 Browser results

When it comes to browsers there are a lot of unknown factors with regards to each browser's implementation of certain functions. Richards et al (2010) bring up some common pitfalls in JavaScript applications that affect performance as mentioned earlier. As browsers handle almost every case differently, some of these application implementations may fare worse in some browsers than others.

For instance if one toolkit has implicitly called "eval" in one of their functions, it could affect performance differently in browsers. Some evidence of this could be seen in the application created thanks to different handling and notation for XPath between Internet Explorer and other browsers. In Internet Explorer we had to use "selectNodes" to perform an XPath query, while in the other browsers "evaluate" was used. This could explain why Internet Explorer is seen as an outlier in the large file-size test while performing better or equal to Chrome in the smaller sizes. There is a different underlying implementation to it that affects performance and changes scaling variables. This may very well apply to even more functions, which makes examining browser functions in detail a hard task. It may be for this reason that entire applications are generally benchmarked, as opposed to snippets between browsers (Ratanaworabhan et al, 2010).

8.4 Dataset

While not the focus of this article, the concern noted by Lengler and Moere (2009) in a previous chapter about the importance of selection of data and model to use for an

infographic held true. It was noted that there were no users involved therefore negating the need for composing a compelling infographic as viewed by others.

The raw data contained a high degree of outliers – for example, thousands of wages at 9999 while up to hundreds of thousands at 0 – and as such the choice of what to truncate affected the display. In addition to this, there were some classes of workers with an inherent quality of no wage, such as "Without Pay", which led to their circles always being displayed at the very bottom edge.

It was intriguing how much the selection of data mattered, especially considering the usage of animation and color, which introduced more specific attribute requirements into the mix. It does seem Lengler and Moere (2009) were accurate in their assessment; using a poorly selected dataset for an inappropriate model was akin to forcing a round peg into a square hole. The lack of a good dataset did not affect or limit the results in any way with the methodological approach used. On the other hand, if the method and tests would have involved users at any level, and their interaction with the application, this pitfall would have had an adverse effect on the tests. This application would be more ideal for historical data rather than the snapshot in time that this dataset displayed. Historical data would show a more interesting development and movement of circles, and some important infographical information could be gleaned from it. However, the lack of this did not affect our results because the manner of display for the dataset makes no difference to the benchmarks and method used. The aim with the dataset, which was consequently reached, was to query a huge amount of raw data to be displayed quickly between animation steps.

8.5 Infographics

The results showed that animating a canvas in real-time with a filtered raw dataset could definitely provide a compelling infographic. Unfortunately, the dataset itself did not provide for a good graphical statistical comparison, nor did it guide the viewer as Lengler and Moere (2009) discussed. Part of this stemmed from the fact that the animation was not based on an absolute point in time but a relative one. As the ages incremented, different people were used as data. This meant that at varying ages, the wages could differ to a high degree within a worker class, and instead of traversing towards one corner, the circles bounced around. Thus the viewer could have had a hard time imagining the continued existence after the animation finished with this seemingly random movement.

The infographic suffered due to a lack of interesting attributes in the dataset. As the dataset needed to be truncated to fit, it lost a lot of statistical value and may have given a wrongful representation of the data.

8.6 Ethical aspects

As we had no human element in these experiments there were no compromises or issues arising from human ethical aspects. In our experiments with the query toolkits we made sure to always use the same testing environment. We also tried to emulate the exact same conditions with regards to possible browser caching, console debugging interference and other elements brought up previously. Furthermore, in the case where one browser lacked support as Chrome did with 5mb local storage, we decided to opt out of using local storage for all browsers on that file size to ensure a fair experiment. As such we do not believe our results can be contested from a research ethical standpoint.

9 Future work

These tests excluded the use of server-based languages but the addition of them could change the results seen here. Server-based languages such as PHP or ASP.NET could have their own useful functions of XML that is not seen in JSON that may well affect functionality and speed when used. We propose further research as to how this affects the results shown here with regards to performance and feature-set.

An issue that arose during benchmarking was that the methodological decision to benchmark the entire application lent itself to varying results between file sizes, to no fault of the browser or toolkits used. Instead, the varying results originated from the animation time of the application. An alternative to this, for more accurate file size comparisons, would be to independently benchmark different components in the application. For example, benchmarking the time it takes between the canvas asking for a new age and receiving it. There would also be a need to benchmark the actual animation, as the browsers could have varying draw speed or interpretation on refresh rates. Benchmarking the calculations done on the results could also be necessary because browsers could differ in the handling of the data. While both options are viable, we suggest that there is more research done on each individual part rather than the whole application. This would show in detail how each browser handles specific functions.

As we moved upwards in file sizes to larger data sets, the averaged attributes for each circle changed. For example, there could have been less outliers or zero-values in the smaller file sizes. This could shift the average in either position as larger files were used. Since the average was shifted, the circles' relative position on the board changed, thus altering the animation time to get to the destination. While this did not affect our conclusion in any way, it would be an interesting continuation on our research to provide a dataset with varying file sizes that always conformed to the same average, so they would always have the same animation time.

Internet Explorer 10 which includes the AppCache functionality was not released as of this writing and Internet Explorer 9 which was used in our tests lacked the functionality. To alleviate this while still being able to test three major browsers, we opted to only use Local Storage as opposed to a combination of both. We do not believe that this would affect the test results however, as running the application from the application cache and after the browser has automatically loaded and cached it yields the same functionality in the end. It would be advantageous to do more research on all browsers with the AppCache functionality in place to showcase any differences between the implementations.

It could also be of value to investigate the performance of JSON and XML query toolkits when processing document-based datasets. This could show how the performance is affected by more document-data and less markup while still retaining the same file sizes. One suitable candidate for such a test could be a document saved in a word-processing program. Similar to Excel, this is also saved with OfficeOpen standards and should yield an XML file that one could query.

Examining the specifics and workings of each query toolkit and how they diverge from one another could prove interesting results as to why they differ in performance. This would be a very extensive study because it would require investigating the source code of each, but could possibly also show why the toolkits' performance differs between browsers.

References

Bonacci, H. (2012) *jLinq*. 2012. Available at: http://www.hugoware.net/Projects/jlinq [Accessed December 11, 2011].

Cannon, B. & Wohlstadter, E. (2010) Automated object persistence for JavaScript. *Proceedings of the 19th international conference on World wide web*. Raleigh, North Carolina, USA, ACM. pp. 191–200.

ECMA International (2011) *Standard ECMA-376*. 2011. Available at: http://www.ecma-international.org/publications/standards/Ecma-376.htm [Accessed February 16, 2012].

Ercegovac, V. & Beyer, K. (2011) *jaql - Query Language for JavaScript(r) Object Notation (JSON) - Google Project Hosting*. 2011. Available at: http://code.google.com/p/jaql/ [Accessed December 11, 2011].

Erfianto, B., Mahmood, A.K. & Rahman, A.S.A. (2007) Modeling Context and Exchange Format for Context-Aware Computing. *Research and Development, 2007. SCOReD 2007. 5th Student Conference on*. pp. 1–5.

Fourny, G., Pilman, M., Florescu, D., Kossmann, D., Kraska, T. & McBeath, D. (2009) XQuery in the browser. *Proceedings of the 18th international conference on World wide web*. Madrid, Spain, ACM. pp. 1011–1020.

Goessner, S. (2012) *JSONPath - XPath for JSON*. 2012. Available at: http://goessner.net/articles/JsonPath/ [Accessed December 11, 2011].

Groppe, S. & Böttcher, S. (2003) XPath query transformation based on XSLT stylesheets. *Proceedings of the 5th ACM international workshop on Web information and data management*. New Orleans, Louisiana, USA, ACM. pp. 106–110.

Grossman, D. (2012) *W3Counter - Global Web Stats*. 2012. Available at: http://www.w3counter.com/globalstats.php [Accessed May 30, 2012].

Hipp, R. & Katz, D. (2012) *Home - UnQL Specification - Confluence*. 2012. Available at: http://www.unqlspec.org/display/UnQL/Home [Accessed December 11, 2011].

Hsu, F. & Chen, H. (2009) Secure file system services for web 2.0 applications. *Proceedings of the 2009 ACM workshop on Cloud computing security*. Chicago, Illinois, USA, ACM. pp. 11–18.

Huang, W. & Tan, C.L. (2007) A system for understanding imaged infographics and its applications. *Proceedings of the 2007 ACM symposium on Document engineering*. Winnipeg, Manitoba, Canada, ACM. pp. 9–18.

JSON.org (2002) *JSON*. 2002. Available at: http://json.org/ [Accessed November 18, 2011].

Kristol, D. & Montulli, L. (2000) *HTTP State Management Mechanism*. 2000. HTTP State Management Mechanism. Available at: http://www.ietf.org/rfc/rfc2965.txt [Accessed February 19, 2012].

Lam, T.C., Poon, S. & Ding, J.J. (2008) Benchmarking Stream-Based XPath Engines Supporting Simultaneous Queries for Service Oriented Networking. *Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE*. pp. 1–6.

Lawrence, R. (2004) The space efficiency of XML. *Information and Software Technology*. 46 (11), pp. 753 - 759.

Lee, L.H., Kuhl, M.E., Fowler, J.W., Robinson, S., Haas, P.J. & Jermaine, C. (2009) Database Meets Simulation: Tools and Techniques. *INFORMS Simulation Society Research Workshop*. pp.

Lengler, R. & Moere, A.V. (2009) Guiding the Viewer's Imagination: How Visual Rhetorical Figures Create Meaning in Animated Infographics. *Information Visualisation, 2009 13th International Conference*. pp. 585–591.

Microsoft (2012) *[MS-OFFDI]: Microsoft Excel Persistence Formats*. 2012. Available at: http://msdn.microsoft.com/en-us/library/dd925326(v=office.12).aspx [Accessed February 16, 2012].

Nicola, M. & John, J. (2003) XML parsing: a threat to database performance. *Proceedings of the twelfth international conference on Information and knowledge management*. New Orleans, LA, USA, ACM. pp. 175–178.

Nurseitov, N., Paulson, M., Reynolds, R. & Izurieta, C. (2009) Comparison of JSON and XML Data Interchange Formats: A Case Study. *CAINE'09*. pp. 157–162.

Ratanaworabhan, P., Livshits, B. & Zorn, B.G. (2010) JSMeter: comparing the behavior of JavaScript benchmarks with real web applications. *Proceedings of the 2010 USENIX conference on Web application development*. Boston, MA, USENIX Association. pp. 3–3.

Richards, G., Lebresne, S., Burg, B. & Vitek, J. (2010) An analysis of the dynamic behavior of JavaScript programs. *SIGPLAN Not.* 45 (6), pp. 1–12.

Robie, J., Brantner, M., Florescu, D., Fourny, G. & Westmann, T. (2012) *JSONiq*. 2012. Available at: http://jsoniq.org/ [Accessed December 11, 2011].

Stocker, D. (2011) *jOrder*. 2011. Available at: http://jorder.net/ [Accessed December 15, 2011].

University of California (2000) *Census-Income Database*. 2000. Available at: http://kdd.ics.uci.edu/databases/census-income/census-income.html [Accessed February 16, 2012].

Wang, G. (2011) Improving Data Transmission in Web Applications via the Translation between XML and JSON. *Communications and Mobile Computing (CMC), 2011 Third International Conference on*. pp. 182–185.

West, W. & Pulimood, S.M. (2012) Analysis of privacy and security in HTML5 web storage. *J. Comput. Sci. Coll.* 27 (3), pp. 80–87.

World Wide Web Consortium (2012) *Extensible Markup Language (XML)*. 2012. Available at: http://www.w3.org/XML/ [Accessed June 13, 2012].

World Wide Web Consortium (2011a) *W3C XML Query (XQuery)*. 2011. Available at: http://www.w3.org/XML/Query/#implementations [Accessed February 23, 2012].

World Wide Web Consortium (2011b) *Web Storage*. 2011. Available at: http://www.w3.org/TR/webstorage/ [Accessed June 13, 2012].