Balisage: The Markup Conference

Fat Markup: Trimming the Fat Markup Myth one calorie at a time

David Lee
Lead Engineer
MarkLogic, Inc.
<dlee@marklogic.com>

Abstract

We all know that XML is "fat" and JSON is the "thinner", "faster", "smaller", "better" markup. We know this to be true because we've been told it over and over. It's "obvious" and " inherently true" because XML has redundant end tags, namespaces, entities and other extra "pounds of fat" that JSON doesn't have. But where is the science supporting this? What are the facts and what is myth? When people make design and architecture decisions it should be supported by facts not speculation. In this paper I show the results of an ongoing series of real world tests of Markup performance in browsers across a wide variety of devices, browsers and operating systems and attempt to quantify markup performance with experimental results and maybe trim the fat myth one calorie at a time.

Table of Contents

The Myth

JSON is lean and XML is fat. We know this to be true. Here is a typical quote stated as an undisputed fact.

> JSON’s lightweight payload allows for reduced bandwidth needs and faster transfers. JSONLIGHT

Douglas Crockford boldly titles his paper

> JSON: The Fat-Free Alternative to XML XMLFAT

[1]

Simply search the web or ask your friends and except for a few evangelist's XML vs JSON they will tell you the same thing. For people who want to promote and use JSON they often use this "fact" to support their cause. For people who want to use and promote XML they usually accept this and point to XML's other features that are more important.

Yet few dispute this fact or attempt a systematic measurement to validate or disprove it. And when they do it's usually a very constrained test with a single corpus AJAX Performance. It's just true.

> The worst misconceptions are those which everyone knows to be true, and yet are completely false. Once a false idea gets into the public consciousness, however, they are very difficult to expunge, and rarely go away completely.

Dr. Steven Novella MISCONCEPTION

What is "Fat Markup"?

When used to describe Markup, "Fat" has many factors and connotations. For the purposes of this paper the following attributes are investigated - size, shape, speed and looks.

Size Matters

The total size of a marked up document matters for some purposes. The larger the size of the document, generally the more memory it consumes when parsed, the more space it takes in storage and the more time it takes to transfer over a network. However, a single measurement of size is misleading.

Size can be measured in bytes, characters, nodes and other metrics - both on storage and in memory. Encoding and compression can affect the byte size given the same character size in storage (and often memory). Choices of particular markup style representing the same document data can affect the number of characters; for example choosing shorter element names can produce a document with less characters. Markup choices can also affect node structure and size; for example using attributes in XML instead of elements for some data produces a different representation of the node tree itself which in turn affects the number of characters and bytes. There are many other considerations such as numeric precision, ignorable and boundary whitespace, defaulted attributes, use of namespaces etc. While some specifics are particular to the markup language the concept is valid in both XML and JSON formats.

Thus given the same document as an abstract object, it is meaningless to attempt to provide a single measurement of its size as expressed in a particular markup format. However, one can measure specific sizes of a particular representation of a document.

But even given a specific metric for the size of a document, does it matter? That would depend on what one cares about. Size is at best an indirect measurement of some other quantity one is interested in such as time to transmit over a network or disk space used. General intuition is that probably smaller is "better". However, to get a more definitive answer then that we need to ask "better for what and whom?" A binary compressed document may be smallest and better at conserving disk space but not better at readability, usability or parsing speed.

Shape Matters

The "Shape" or particular structure of a document matters for some purposes. For example, readability and ease of authoring by humans or machines may be important. The "shape" also affects the design of parsers, data models and programming API's.

A particular shape may be better for some uses then others - an exact mapping to a particular programming language object structure may be useful in that language but cumbersome in another. Even in a single language some models may be better suited for direct access and others for searching.

Shape of a document is hard to quantify, but I suggest that one can use "ease of use" as a proxy. A shape that doesn't fit your use case is hard to use and could be considered "Fat" in terms of the difficulty of using it.

Speed Matters

When something is considered "Fat" it's generally implied that its "Slow". Speed matters. The time it takes to transfer a document from disk to memory, the time it takes to transfer across a network, the time it takes to parse and query a document - all matter. There is also developer time. How long does it take a developer to learn a language and write code to process a document?

Sometimes speed in the small doesn't mean speed in the large. The story of the Tortoise and the Hare can provide good reflection on this. As a paractical example, imagine a document which is very fast to load into a programming language but the data format produced doesnt lend itself for some use cases so extra processing, internal transformations to other models or use of complex and slow libraries may be required to access the document. For example it is common practice in JSON to embeed HTML snippets as strings. If you want to search, combine or reformat them you may then have to construct an HTML document and parse it with an HTML parser to produce an object model which useful. The leanness of the original markup is lost in the overhead of having to perform multiple layers of parsing.

Looks Matter

Fat is generally considered "Ugly". A "Fat Markup" often implies an ugly one. If a document format is so ugly you can't stand looking it, you will try to avoid it.

Fortunately "Beauty is in the eyes of the Beholder". This is true in markup as well as the humanities. As time, exposure, and fashion change so can the subjective beauty of a markup format.

What's it all mean?

Sweeping statements of technologies like "Fat", "Slow", "Bloated" have imprecise meaning. Worse they are often chosen to pull in emotional associations which may not apply to the specific cases. So even if it were considered true, what does it mean by saying "XML Is Fat"? I suggest it means very little, actually of negative usefulness. A markup itself is not used in isolation; it is used in the context of complex work-flows including editing, distribution and processing. So what should one do to characterize a Markup Language in a meaningful and useful way?

I suggest a path lies in defining tests that measure specific use cases and provide reproducible metrics. These metrics can be used to get a better understanding of the performance of markup in a meaningful way.

In short, An Experiment.

The Experiment

I designed an experiment to attempt to quantify some of the attributes attributed to "Fat Markup". In particular I focused on what is generally considered an already answered question in particular because I would like to validate that the "common wisdom" is in fact based on fact, and if so or not, to what degree and in what exact cases. The Experiment focuses on performance of JSON and XML in the use case of being delivered by a standard web Server and being parsed and queried in a Browser. Furthermore, mobile devices are becoming vastly more prevalent then in the recent past and little research addresses Mobile Browsers so I attempt to include them as much as possible in the tested scenerios.

The Experiment I developed attempts to test the following attributes of XML and JSON processing.

Multiple Document Corpuses

Most of the past publications I have found focus on a single corpus of documents and usually attempt to derive a single metric. These are often very artificial, focused on a single domain or structurally similar. For this experiment I take 6 very different documents and a baseline to attempt to cover a range of document uses 'in the wild'.

Subsets of larger corpus and duplication of smaller datasets were chosen so the resultant XML and JSON sizes ranged from 100 KB to 1 MB. These limits were chosen so the size was large enough to take noticeable time to load, parse and query but not so large as to break most reasonable browsers even on mobile devices.

The Document corpus used is as follows

- base
  This is a very basic baseline document. It consists of a root object and a child object with 3 properties. This baseline is used to sanity check the test to make sure proper counting of size and nodes could be easily manually validated.
- books
  This is an expanded sample of the BOOKS sample distributed with Saxon `Saxon`. It contains 600 book items, with 300 categories.
- epa
  A portion of the EPA Geospatial data at http://www.epa.gov/enviro/geo_data.html containing a collection of 100 FacilitySite records.
- ndc
  A subset of the National Drug Code Directory at http://www.fda.gov/drugs/informationondrugs/ucm142438.htm containing a collection of 1000 FORMULATRION records.
- snomed
  A subset of the SNOMED concept database from http://www.nlm.nih.gov/research/umls/Snomed/snomed_main.html containing a collection of 100 SNOMED concept records.
- spl
  A single SPL document from the FDA as provided by DailyMed, a website run by the National Institute of Health http://dailymed.nlm.nih.gov/dailymed/downloadLabels.cfm
- twitter
  A collection of 1000 random tweets from a search of Super Bowl and advertiser terms. Identifying data from the collection was anonymized by randomizing the user names and ID's.

Document Size and shape

Most research I have found makes the incorrect assumption that there is a single representation of a document in a particular markup. I take a different approach. Starting with seven (7) different documents I produce 2 variants in JSON and 3 variants in XML that all represent the same abstract document. These documents span several data styles including simple tabular, highly structured and document format. Some of the corpus is synthetic and some taken from real world data. In addition I test the support for HTTP gzip compression on these documents.

JSON Formats

For JSON documents I produced 2 variants.

One variant similar to the default XML to JSON transformation from json.org (http://www.json.org/java/index.html). This variant, while fully expressive is generally not what a JSON developer would expect. However, it is what a

naive XML to JSON transformation may produce.

The second JSON variant is a custom transformation informed by the specific data structures and is close to what a JSON programmer might expect from a JSON format.

In both cases extraneously white-spaces are removed.

JSON Examples

Twitter 1 Full

The following is an example of a single JSON object in the the twiter-1full.json. This is representative of a nieve conversion from XML using code similar to that at json.org. Whitespace, newlines and indentation was added for presentatin only; it does not exist in the actual document.

Figure 1: Twitter 1 full example


```
{ "status": { "_attributes": { "id": "303239543170138114" }, "_children": [ "\n ", { "created-at": { "_children": [ "2013-02-17T15:28:35" ] } }, "\n ", { "user": { "_attributes": { "created-at":
"2012-05-30T01:40:04", "description": "No regrets im blessed to say the old me dead and gone away", "favorites-count": "1079", "id": "667408927467172427 7", "lang": "en", "name": "AqmtidAkSesSZ",
"screen-name": "ITQMiAqmti" }, "_children": [ "\n ", { "location": { "_children": [ "Hartselle " ] } }, "\n " ] } }, "\n ", { "hash-tags": { "_children": [ "\n ", { "hash-tag": { "_attributes":
{ "start": "65", "end": "78" }, "_children": [ "coolranchdlt" ] } }, "\n " ] } }, "\n ", { "source": { "_children": [ "&lt;a href=\"http://twitter.com/download/iphone\"
rel=\"nofollow\"&gt;Twitter for iPhone&lt;/a&gt;" ] } }, "\n ", { "text": { "_children": [ "Taco Bell has cooler ranch Doritos tacos my life just got better #coolranchdlt" ] } }, "\n ", { "url-
entities": {} }, "\n ", { "user-mention-entities": {} }, "\n " ] }
```


This is an example of the same record represented in a format that a native JSON developer may expect. Whitespace, newlines and indentation was added for presentatin only; it does not exist in the actual document.

Figure 2: Twitter 2 custom example


```
{ "status": { "id": "303239543170138114", "created-at": "2013-02-17T15:28:35", "user": { "created-at": "2012-05-30T01:40:04", "description": "No regrets im blessed to say the old me dead and
gone away", "favorites-count": "1079", "id": "667408927467172427 7", "lang": "en", "name": "AqmtidAkSesSZ", "screen-name": "ITQMiAqmti", "location": "Hartselle " }, "hash-tags": { "hash-tag": [ {
"start": "65", "end": "78", "_value": "coolranchdlt" } ] }, "source": "&lt;a href=\"http://twitter.com/download/iphone\" rel=\"nofollow\"&gt;Twitter for iPhone&lt;/a&gt;", "text": "Taco Bell has
cooler ranch Doritos tacos my life just got better #coolranchdlt", "url-entities": "", "user-mention-entities": "" } },
```


XML Formats

For the first format I used the "native" XML format (if the document originated as XML) or a naive transformation from the object model to XML.

For the second format I took the first format and simply removed boundary and ignorable whitespace

For the third XML format I transformed the XML into an attribute centric format. This was accomplished by taking all XML elements which contained only text content, no attributes, and is not repeating and transformed that element into an attribute.

XML Examples

What follows are 3 examples of the same twitter record as used for the XML documents.

Figure 3: XML 1 example - base xml with whitespace and indentation


```
<status id="303239543170138114"> <created-at>2013-02-17T15:28:35</created-at> <user created-at="2012-05-30T01:40:04" description="No regrets im blessed to say the old me dead and gone away"
favorites-count="1079" id="667408927467172427 7" lang="en" name="AqmtidAkSesSZ" screen-name="ITQMiAqmti"> <location>Hartselle </location> </user> <hash-tags> <hash-tag start="65"
end="78">coolranchdlt</hash-tag> </hash-tags> <source>&lt;a href="http://twitter.com/download/iphone" rel="nofollow"&gt;Twitter for iPhone&lt;/a&gt;</source> <text>Taco Bell has cooler ranch
Doritos tacos my life just got better #coolranchdlt</text> <url-entities/> <user-mention-entities/> </status>
```


Figure 4: XML 2 example - All whitespace and indentation removed


```
<status id="303239543170138114"><created-at>2013-02-17T15:28:35</created-at><user created-at="2012-05-30T01:40:04" description="No regrets im blessed to say the old me dead and gone away"
favorites-count="1079" id="667408927467172427 7" lang="en" name="AqmtidAkSesSZ" screen-name="ITQMiAqmti"><location>Hartselle </location></user><hash-tags><hash-tag start="65"
end="78">coolranchdlt</hash-tag></hash-tags><source>&lt;a href="http://twitter.com/download/iphone" rel="nofollow"&gt;Twitter for iPhone&lt;/a&gt;</source><text>Taco Bell has cooler ranch
Doritos tacos my life just got better #coolranchdlt</text><url-entities/><user-mention-entities/></status>
```


Figure 5: XML 3 example - All non-repeating leaf elements pulled up into attributes


```
<status id="303239543170138114" created-at="2013-02-17T15:28:35" source="&lt;a href=&#34;http://twitter.com/download/iphone&#34; rel=&#34;nofollow&#34;&gt;Twitter for iPhone&lt;/a&gt;"
text="Taco Bell has cooler ranch Doritos tacos my life just got better #coolranchdlt" url-entities="" user-mention-entities=""><user created-at="2012-05-30T01:40:04" description="No regrets im
blessed to say the old me dead and gone away" favorites-count="1079" id="667408927467172427 7" lang="en" name="AqmtidAkSesSZ" screen-name="ITQMiAqmti" location="Hartselle "/><hash-tags><hash-tag
start="65" end="78">coolranchdlt</hash-tag></hash-tags></status></twitter>
```


Sizes of corpus

The following table lists the corpus documents, types, and sizes (raw and compressed). This data is stored as a resource (index) and is available to the client on start-up.

Table I

Corpus documents with sizes

group	type	name	size	compress-size
base	json	base-custom.json	56	83
base	json	base-full.json	93	107
base	xml	base.xml	45	69
books	json	books1-custom.json	204832	1933
books	json	books1-full.json	438133	3758
books	xml	books1.xml	256030	2412
books	xml	books2.xml	226430	2169
books	xml	books3.xml	186230	1819
epa	json	epa1-custom.json	140914	13024
epa	json	epa1-full.json	199025	14862
epa	xml	epa1.xml	195201	14591
epa	xml	epa2.xml	192449	14528
epa	xml	epa3.xml	136813	12729
ndc	json	ndc1-custom.json	95093	8503
ndc	json	ndc1-full.json	235603	9621
ndc	xml	ndc1.xml	176157	9140
ndc	xml	ndc2.xml	153157	8952
ndc	xml	ndc3.xml	96087	8295
snomed	json	snomed1-custom.json	230371	22665
snomed	json	snomed1-full.json	501577	25757
snomed	xml	snomed1.xml	405294	24484
snomed	xml	snomed2.xml	355195	23853
snomed	xml	snomed3.xml	221409	22237
spl	json	spl1-custom.json	55762	10857
spl	json	spl1-full.json	135948	12398
spl	xml	spl1.xml	99571	11974
spl	xml	spl2.xml	97965	11927
spl	xml	spl3.xml	87755	11262
twitter	json	twitter1-custom.json	715005	196939
twitter	xml	twitter1-full.json	1056212	207340
twitter	xml	twitter1.xml	809469	198325
twitter	xml	twitter2.xml	718016	195197
twitter	xml	twitter3.xml	700412	194971

Methodology and Architecture

The experiment contains a Server, Browser, and Analysis components. The components were created using standard open source software and as much as possible designed to focus on the attributes being tested and eliminating introduction of bias from components which are not related to the test goals.

Server

The Server component provides two roles. It serves the source data used by the Client and collects the results submitted by the client.

The Server software used is Apache HTTP serverApache. Client documents (HTML, CSS, JavaScript and corpus data) are served as static resources to minimize server side variations. The server is running on an Amazon EC2 instance in Virginia, US.

Results from the client are sent back to the server via an HTTP POST. The server runs a CGI script which formats the results and queues them to a message queue.[2]

A data collection script periodically polls the queue for new results and when it receives one enriches the results by expanding the UserAgent string into sub-components for easier identification of browser and OS versions.[3] The results are then stored locally as an XML file and also published to a database for future analysis.

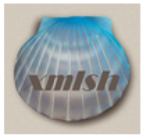
The mechanism used to report results is independent of the tests themselves, do not affect the test data and could be replaced by other analogous methods.

The only Server component that could affect the tests is the serving of static JSON and XML corpus documents. These are exposed as static resources with HTTP GET, with and without gzip compression enabled.

Client

The Client is a Browser based JavaScript application. The GUI components of the client were developed using GWT GWT. The code which performs the measured parts of the tests are hand written JavaScript, with the exception that jQueryjQuery is used for part of the test as indicated.

Figure 6: Client Application



XML - JSON Speed Test

ReadMe

Run Tests And Submit... | Run Tests Only... | Clear Results

Type	Group	Name	Size Raw	Size gzip	Network Raw (ms)	Network gzip (ms)	JS Parse (ms)	JS Query (ms)	JQ Parse (ms)	JQ Query (ms)
json	base	base-custom.json	56	83	69	92	0	1	0	0
json	base	base-full.json	93	107	85	90	0	0	0	0
xml	base	base.xml	45	69	85	90	0	0	0	1
json	books	books1-custom.json	204832	1933	410	107	10	4	5	11
json	books	books1-full.json	438133	3758	720	110	40	14	19	55
xml	books	books1.xml	256030	2412	497	123	20	39	21	161
xml	books	books2.xml	226430	2169	440	99	14	24	14	114
xml	books	books3.xml	186230	1819	397	99	10	8	10	49
json	epa	epa1-custom.json	140914	13024	385	107	4	1	2	1
json	epa	epa1-full.json	199025	14862	469	148	10	3	3	5
xml	epa	epa1.xml	195201	14591	631	149	8	11	7	36
xml	epa	epa2.xml	192449	14528	395	155	6	7	6	52
xml	epa	epa3.xml	136813	12729	330	109	5	2	4	12
json	ndc	ndc1-custom.json	95093	8503	272	95	5	1	2	2
json	ndc	ndc1-full.json	235603	9621	661	163	22	7	6	113
xml	ndc	ndc1.xml	176157	9140	493	114	11	19	11	72
xml	ndc	ndc2.xml	153157	8952	308	99	8	16	8	77
xml	ndc	ndc3.xml	96087	8295	298	103	6	2	5	18
json	snomed	snomed1-custom.json	230371	22665	464	157	9	1	4	1
json	snomed	snomed1-full.json	501577	25757	735	178	33	13	91	19
xml	snomed	snomed1.xml	405294	24484	646	186	26	44	26	152
xml	snomed	snomed2.xml	355195	23853	600	152	17	29	16	142
xml	snomed	snomed3.xml	221409	22237	591	155	13	4	10	27
json	spl	spl1-custom.json	55762	10857	213	99	2	0	1	1
json	spl	spl1-full.json	135948	12398	371	96	21	2	4	5
xml	spl	spl1.xml	99571	11974	269	103	6	7	6	32
xml	spl	spl2.xml	97965	11927	297	124	4	5	4	30
xml	spl	spl3.xml	87755	11262	251	108	4	3	4	26

On start-up the client requests an index document and shows a list of all files in the corpus, their types and sizes. The user can choose to run tests individually or as a whole.

There is one button which when tapped runs all tests then submits the results to the server via an HTTP POST. The time it takes to load, initialize and update the GUI, and to send the results is not included in the results.

Publicly available data available to the client program (browser) is also included in the results. This data is used to identify the browser and OS of the client.

Tests

When the "Run All Tests" button is tapped the following procedure is run. For each of the files in the index the following is performed and individually timed.

- The data file is fetched from the server in raw (not compressed) format and stored as a string. Where possible the actual Content-Length is used to update the data size metric for this file.
- The data file is fetched from the server in compressed (gzip) format and uncompressed and stored as a string. Where possible the actual Content-Length is used to update the compressed size metric for this file.
- The data is parsed into an in memory object. For JSON files this uses the JavaScript eval() method. For XML files standard browser methods are used to parse the XML file.
- The object is "Queried". To simulate a consistent query across the variety of the corpus all nodes of the document are recursively descended and counted. The number of nodes visited is recorded along with the elapsed time.
- The data is again parsed into an in memory object. For both JSON and XML files, jQuery is used to parse the document and produce a jQuery object.
- The object is "Queried". To simulate a consistent query across the variety of the corpus all nodes of the jQuery document are recursively descended and evaluated. The number of nodes visited is recorded along with the elapsed time.

Results

Test Coverage

In addition to a variety of data sources and formats, the experiment attempts to cover a range of devices, operating systems, browsers and networks. This is achieved by "crowd sourcing". The URL to the test was made public and distributed to a range of mailing lists and social media sites including Amazon Mechanical TurkMechanical Turk. Ideally this experiment can continue for a long duration so that trends over time can be measured.

It is expected that performance varies considerably across devices, especially mobile devices, and prior research has largely ignored differences across devices.

Only the browser "User Agent" string is used to distinguish browsers, devices and operating systems. This makes some measurements impossible such as distinguishing between broadband, wifi, 3G, LTE and other networks.

However, even lacking some measurements and precision, seeing the range of performance across platforms is still educational and useful.

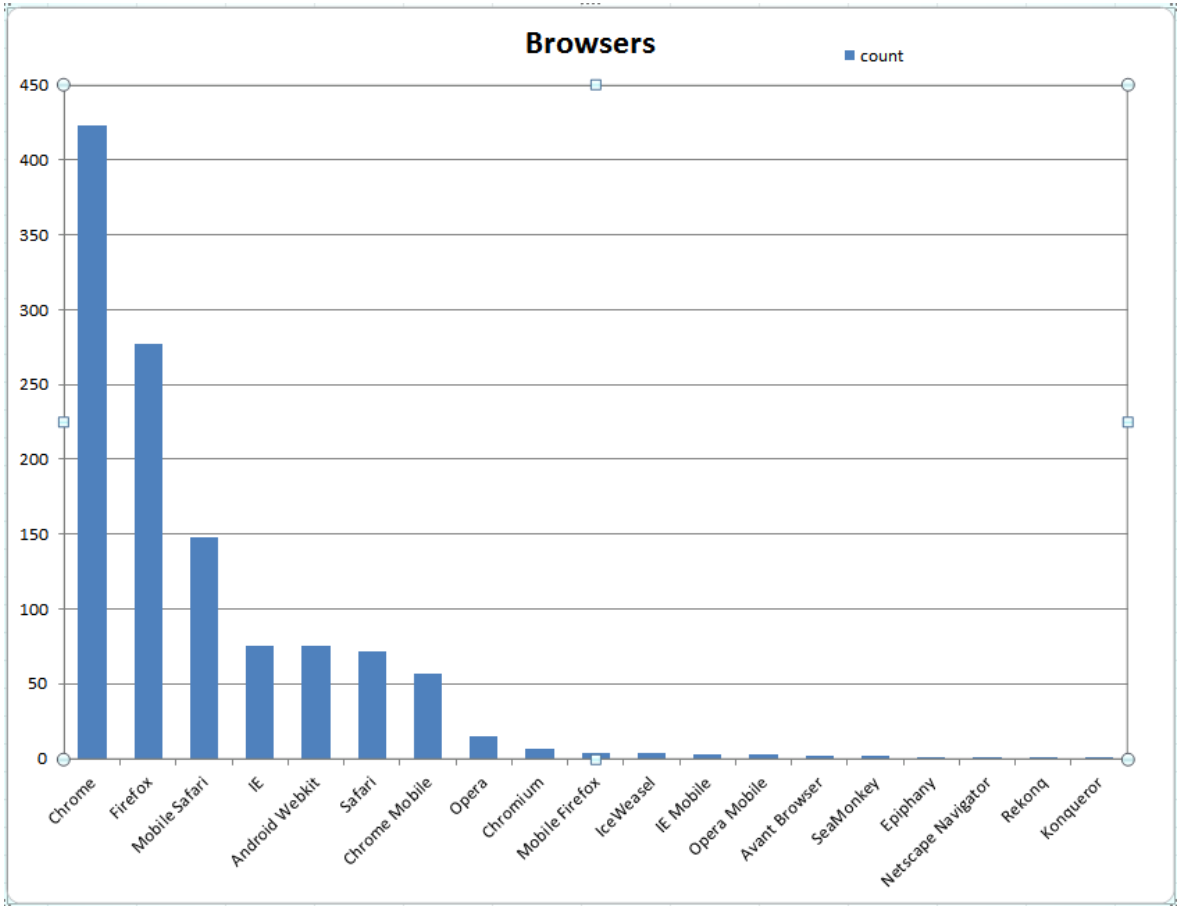
At the time of this paper approximately 1200 distinct successful tests results were collected.

Browser Coverage

Table II

name	count
Chrome	423
Firefox	277
Mobile Safari	148
IE	75
Android Webkit	75
Safari	72
Chrome Mobile	57
Opera	15
Chromium	7
Mobile Firefox	4
IceWeasel	4
IE Mobile	3
Opera Mobile	3
Avant Browser	2
SeaMonkey	2
Epiphany	1
Netscape Navigator	1
Rekonq	1
Konqueror	1

Figure 7: Browsers Covered



Number of tests suites run on specific browsers

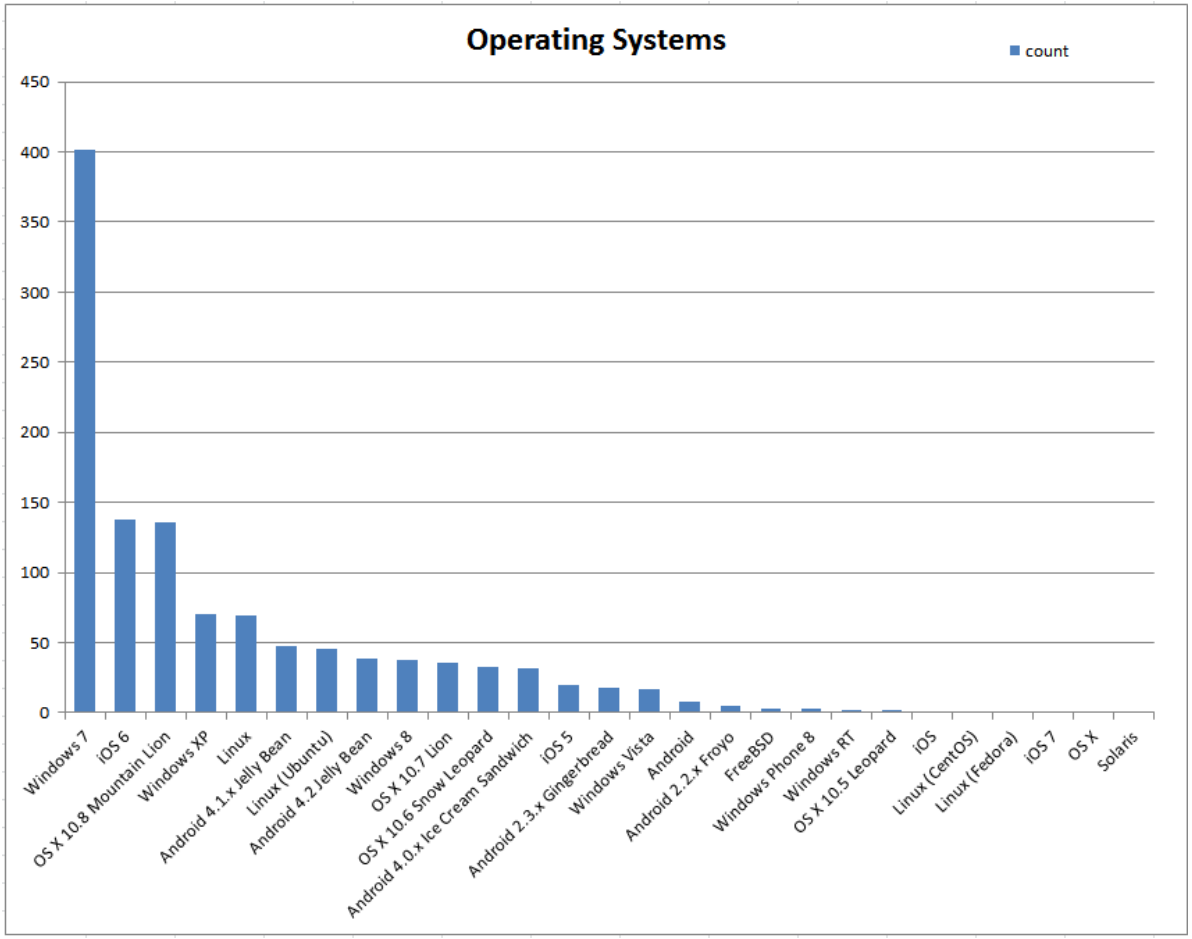
Operating System Coverage

Table III

OS	count
Windows 7	402
iOS 6	138
OS X 10.8 Mountain Lion	136
Windows XP	70
Linux	69
Android 4.1.x Jelly Bean	48
Linux (Ubuntu)	46
Android 4.2 Jelly Bean	39
Windows 8	38
OS X 10.7 Lion	36

OS X 10.6 Snow Leopard	33
Android 4.0.x Ice Cream Sandwich	32
iOS 5	20
Android 2.3.x Gingerbread	18
Windows Vista	17
Android	8
Android 2.2.x Froyo	5
FreeBSD	3
Windows Phone 8	3
Windows RT	2
OS X 10.5 Leopard	2
iOS	1
Linux (CentOS)	1
Linux (Fedora)	1
iOS 7	1
OS X	1
Solaris	1

Figure 8: Operating Systems Covered



Number of tests suites run on specific Operating Systems

Data Sizes

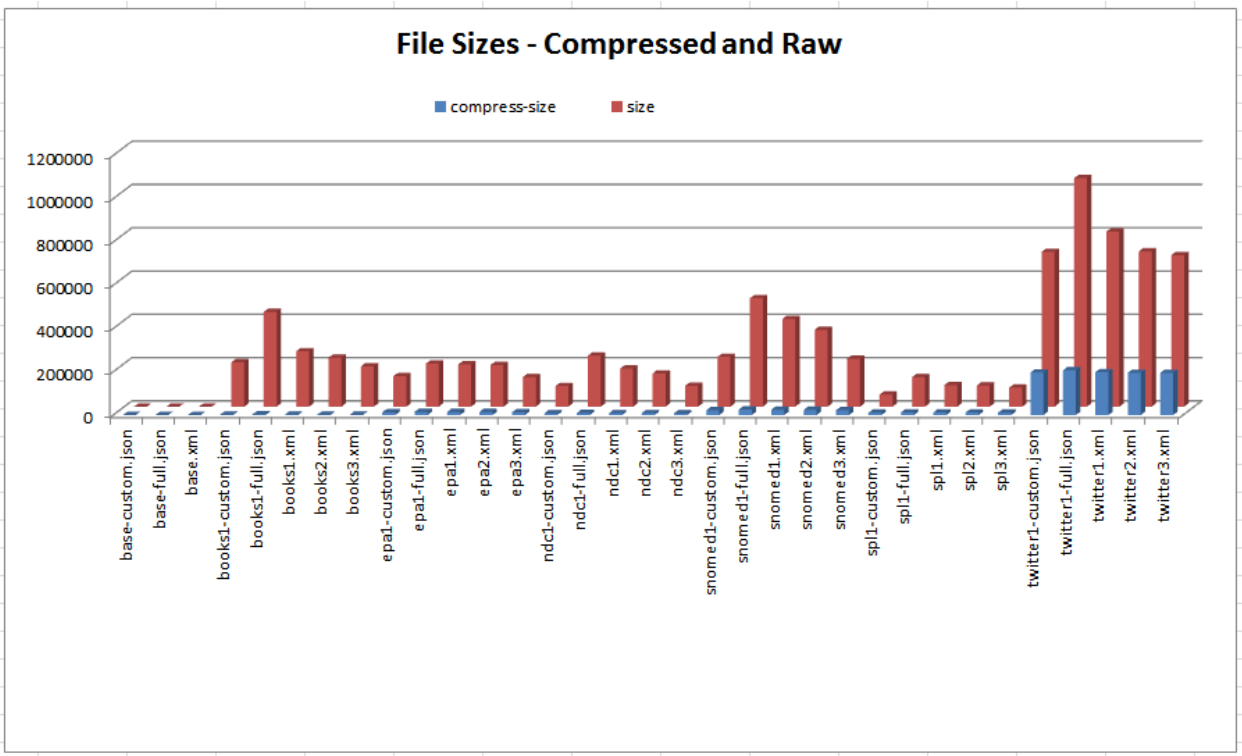
XML is inherently larger then than JSON. This is known as 'obviously true' due to several factors inherent to XML markup.

- The end tags of XML duplicate data in the start tags making the total text longer.
- Namespaces, prefixes, entities, comments, PI's and other XML features add bloat not present in JSON.
- JSON has a direct concise representation for arrays which XML does not.

These are all reasonable presumptions, but how much do these things contribute to data sizes? In real use how much difference does it make? One of the features of modern markup is that we sacrifice some compactness in exchange for readability and usability. If the most important goal was compact size we would be using specialized binary formats. In any case let's look at the actual measurements from the corpus. For the most part size is a static feature of the data files themselves, although this can be affected somewhat by the transmission protocol. Experimentation has shown that overwhelmingly the static data size is very close to the HTTP transmitted size of anything but very tiny documents. HTTP does add a little variable overhead such as chunked encoding and headers, but this is very small. The gzip compression available in HTTP 1.1 has been shown to be equal in size to using the "gzip" command. For the purposes of Data Size we shall examine the static size of the documents on disk. All documents are stored in UTF-8 encoding as plain text files.

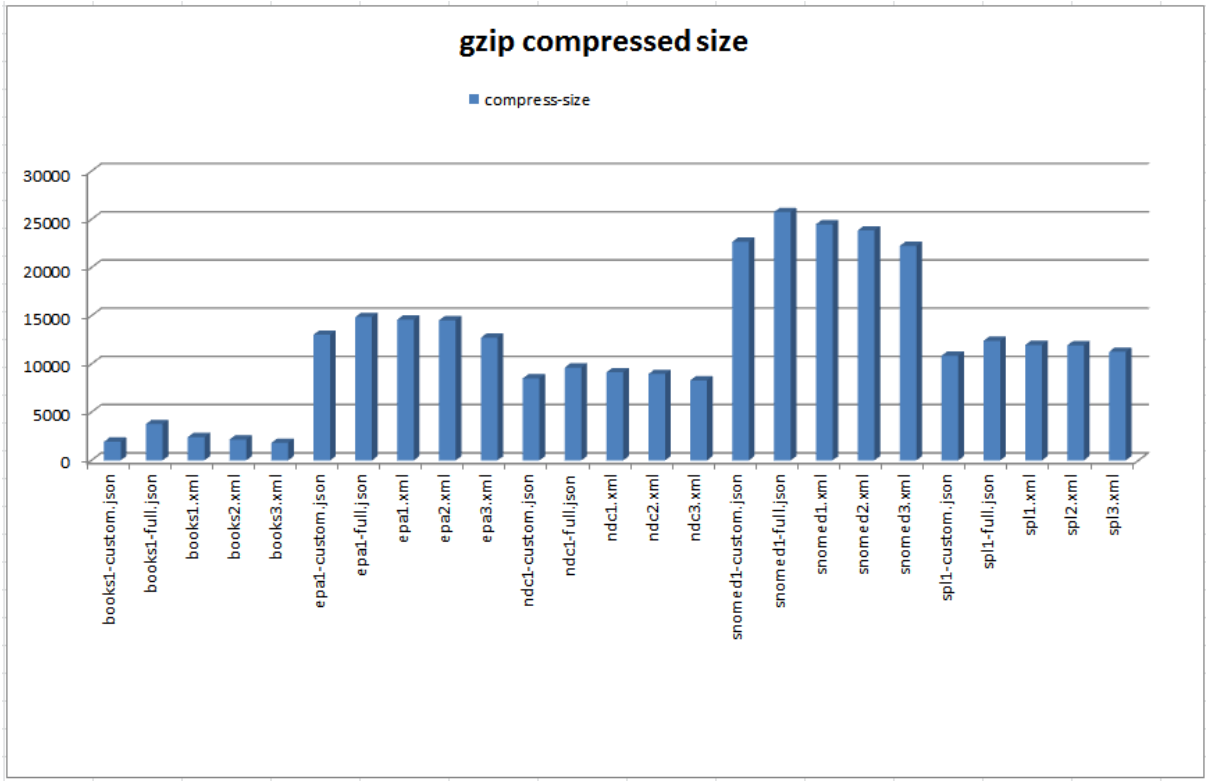
As we can see from Figure 9 and Table I, for each data set the XML and JSON file size can vary significantly depending on the formatting choices used. However, if one carefully chooses a formatting style to minimize size then XML and JSON come very close with XML smaller in some cases and JSON in others. Looking at the compressed sizes of the files in Figure 10 we can see that the file sizes are very close *regardless* of formatting style.

Figure 9: Corpus File Sizes



Compressed and uncompressed file sizes

Figure 10: Compressed File Sizes



Compressed file sizes (excluding twitter data)

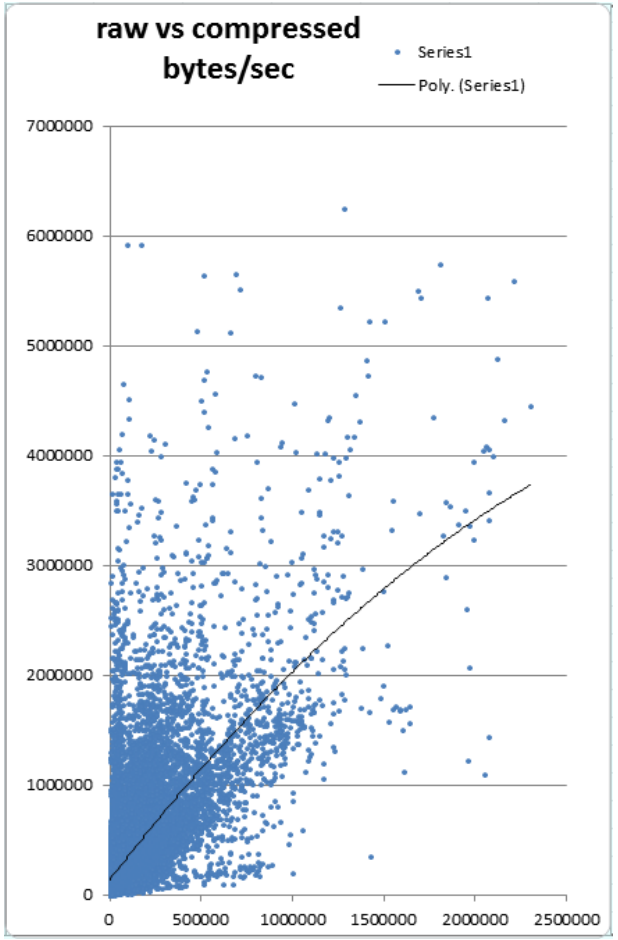
Data Transmission Speed

The predominant factors influencing data transmission speed are the size of the data transmitted and the network speed. In addition, processing speed of the client (browser) and server have some influence as well as packet losses and other internet related issues. Mobile networks generally have significantly higher latency so are more affected by packet loss and retransmission.

This experiment focuses on measuring end to end speeds of HTTP from server to browser using uncompressed and HTTP gzip compression across a range of devices and browsers. It does not attempt to examine the root causes of network bandwidth and traffic.

The range of devices and networks produces a large scatter of transmission speeds. While the trend is that uncompressed and compressed data are fairly linear with respect to bytes transferred there are a lot of outliers. Note that each mark represents the same base document requested uncompressed then compressed via HTTP. If network speed was identical for both then the marks would have x and y values.

Figure 11: Transmission Speed



Transmission time in bytes/sec for Compressed (x) vs Raw (y)

Parsing Speed

So far we have dealt with markup agnostic metrics. It doesn't take any more or less time to transfer the same number of JSON bytes as XML bytes. Parsing is a different matter. This experiment tests two variants of parsing for each JSON and XML. The first variant is to use standard low level JavaScript methods for parsing. The other variant is to use a common library (jQuery). The code for parsing was hand written JavaScript and corresponds to what seems "best practice" in web programming today.

Figure 12: JSON Parsing using native JavaScript

```
eval('(' + responseString + ')');
```

Figure 13: JSON Parsing using jQuery

```
$wnd.jQuery.parseJSON( responseString );
```

Figure 14: XML Parsing using native JavaScript

```
function getIEParser() {
    try { return new ActiveXObject("Msxml2.DOMDocument"); } catch (e) { }
    try { return new ActiveXObject("MSXML.DOMDocument"); } catch (e) { }
    try { return new ActiveXObject("MSXML3.DOMDocument"); } catch (e) { }
    try { return new ActiveXObject("Microsoft.XmlDom"); } catch (e) { }
    try { return new ActiveXObject("Microsoft.DOMDocument"); } catch (e) { }

    throw new Error("XMLParserImplIE6.createDocumentImpl: Could not find appropriate version of DOMDocument.");
};
if ($wnd.DOMParser){
    parser=new DOMParser();
    xmlDoc=parser.parseFromString(responseString,"text/xml");
}
else // Internet Explorer
{
    xmlDoc=  getIEParser();
    xmlDoc.async=false;
    xmlDoc.loadXML(txt);
}
return xmlDoc ;
```

Figure 15: JSON Parsing using jQuery

```
 $wnd.jQuery.parseXML( responseString );
```

Figure 16: Parsing speed using JavaScript in seconds vs file size

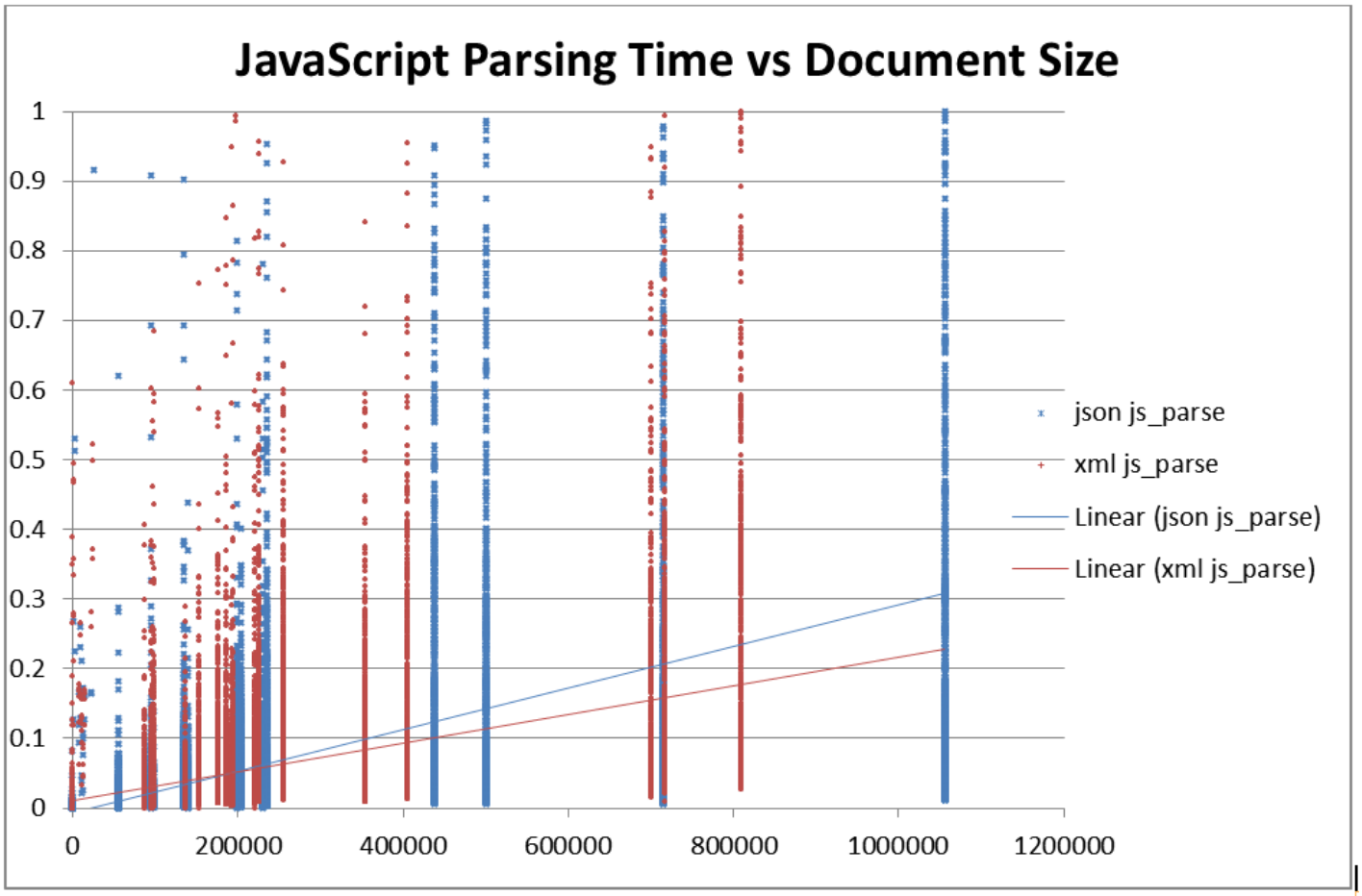
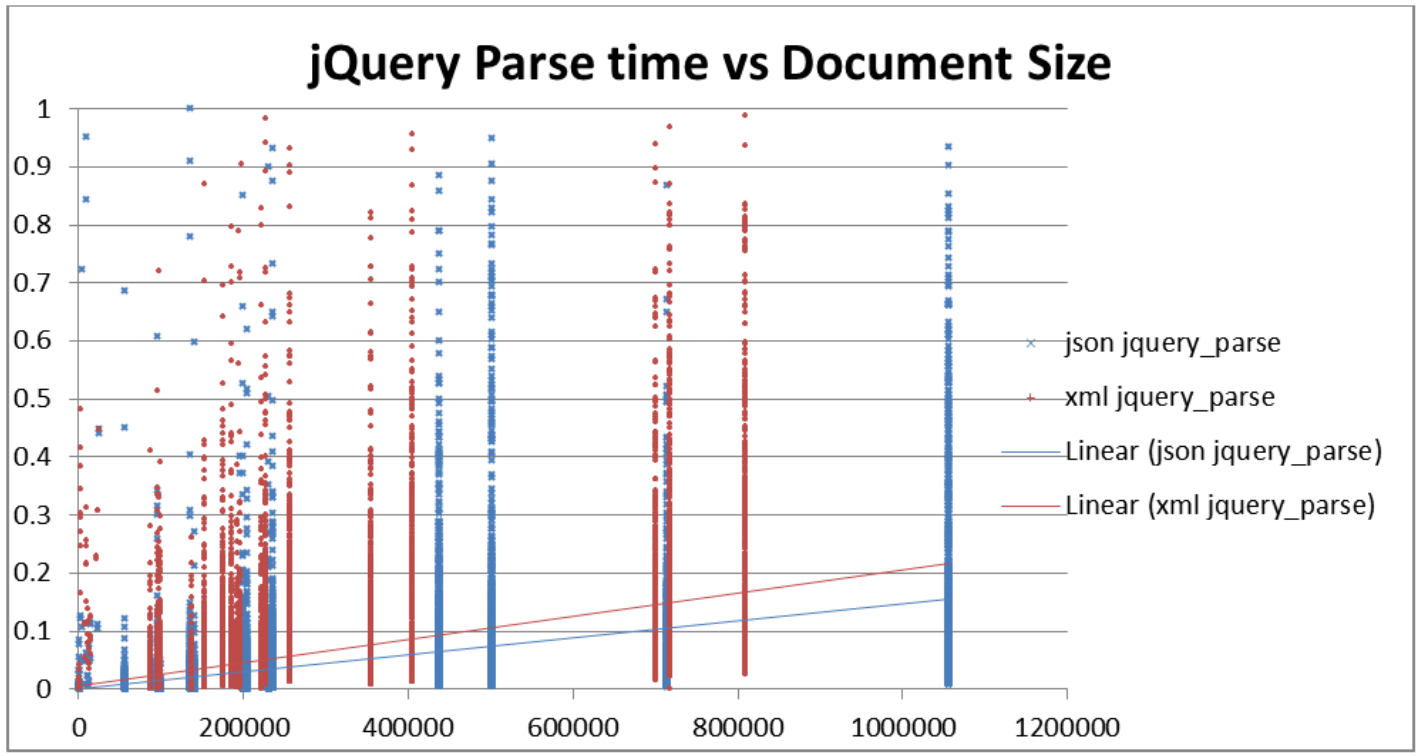


Figure 17: Parsing speed using jQuery in seconds vs file size



Query Speed

Testing query across markup languages and a diverse corpus is tricky to achieve without bias. For the purposes of this experiment I postulate that the purpose of loading the document (JSON or XML) is to use all of it's data in some form.

Therefore to simulate a fair "query" I perform a recursive decent of the parsed object's document model and count every node. There are certainly many other equally good definitions of "query" but I wanted to have a similar test across all the corpus that has a reasonable relevance to typical use cases.

The following listings show the JavaScript code for a recursive decent query of all nodes in the document. It is very interesting to the author that despite popular opinion the code for JSON and XML are extremely similar. XML has attributes which adds a few lines of code but otherwise it's effectively the same amount of programming to recurse and examine a JSON object and an XML object in JavaScript and jQuery. XML adds about 4 lines of JavaScript to handle attributes in both the JavaScript and jQuery case, but other than that the code is nearly identical.

Figure 18: JSON Query using native JavaScript

```
var n = 0;
var walk = function(o){
  if( o == null || typeof(o) == "undefined" )
     return;
  n++;

  for(var prop in o){
    n++;
    if(o.hasOwnProperty(prop)){
      var val = o[prop];

      if(typeof val == 'object'){
          walk(val);
      }
    }
     else
        var val = o ;
  }
};
```

```
walk( this );
return n;
```

Figure 19: JSON Query using jQuery

```
var n = 0;
var walk = function( o )
{
    if( o == null || typeof(o) == "undefined")
        return ;
    n++;
    $wnd.jQuery.each(o, function(key, value) {
      if( value != null && typeof(value) == "object" )
          walk( value )
}
walk(this);
return n;
```

Figure 20: XML Query using native JavaScript

```
var n =0;
var walk = function( o )
{
    if( o == null || typeof(o) == "undefined" )
       return ;
    n++;
    if( o.attributes != null && typeof(  o.attributes) != "undefined" )
        for( var x = 0; x <  o.attributes.length; x++ ) {
            n++;
        }
    if(  o.childNodes != null && typeof(  o.childNodes) != "undefined" )
        for( var x = 0; x <  o.childNodes.length; x++ ) {
            walk(  o.childNodes[x] );
        }
}
walk(this);
return n;
```

Figure 21: XML Query using jQuery

```
var n = 0;
var walk = function( o )
{
    if( o == null || typeof(o) == "undefined" )
       return ;
    n++;
    $wnd.jQuery( o).children().each( function( ) {
        n++ ;
        walk( this );
    })
    var a = o.attributes;
    if( a != null && typeof(a) != "undefined" )
        $wnd.jQuery(a).each( function() {
         n++;
    })
}
walk(this);
return n;
```

Taking a look at the query times however, JSON clearly has an advantage over XML in pure query times. jQuery clearly imposes a significant penalty on XML query but it also imposes a huge penalty on JSON query.

Figure 22: Query speed using JavaScript in seconds vs file size

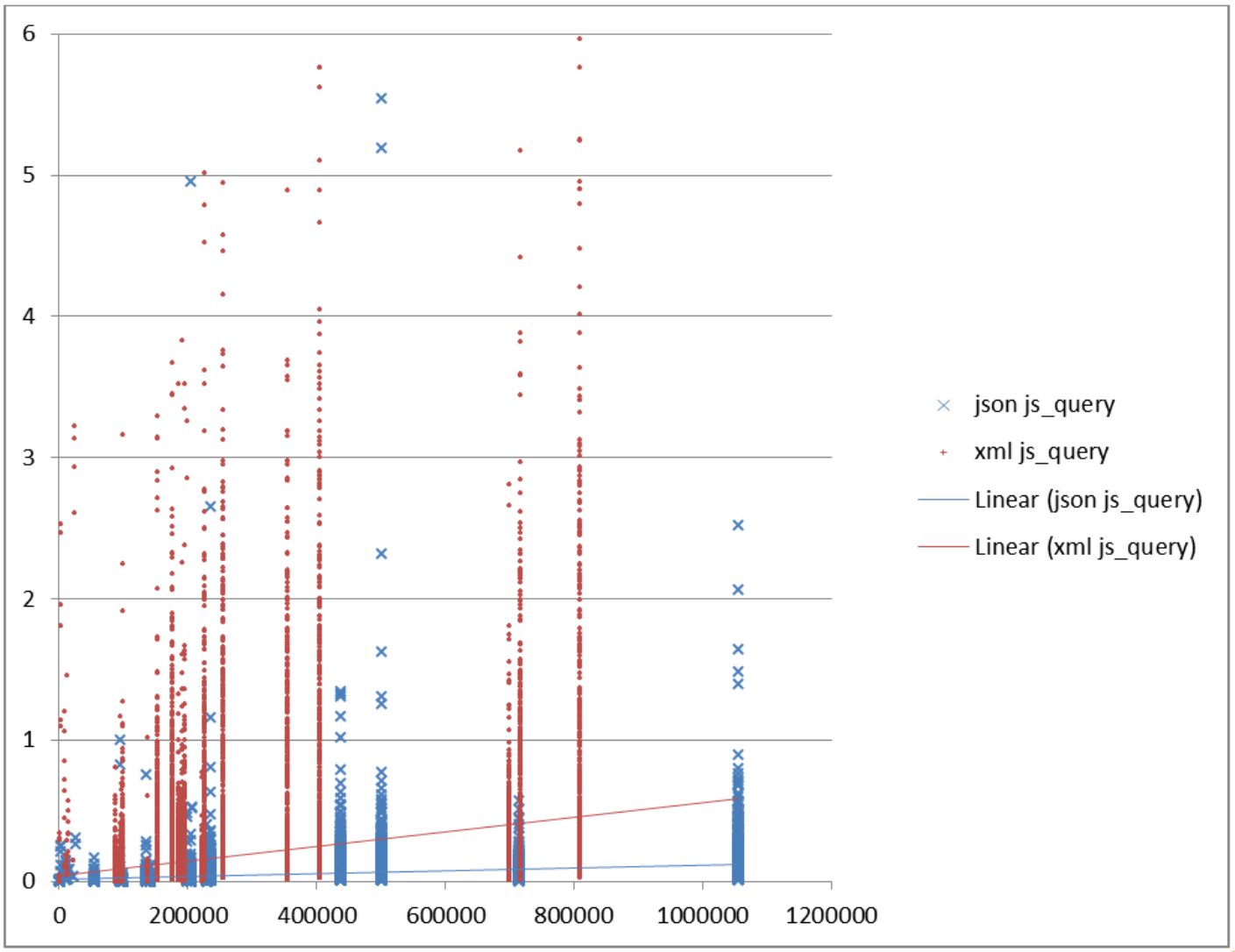
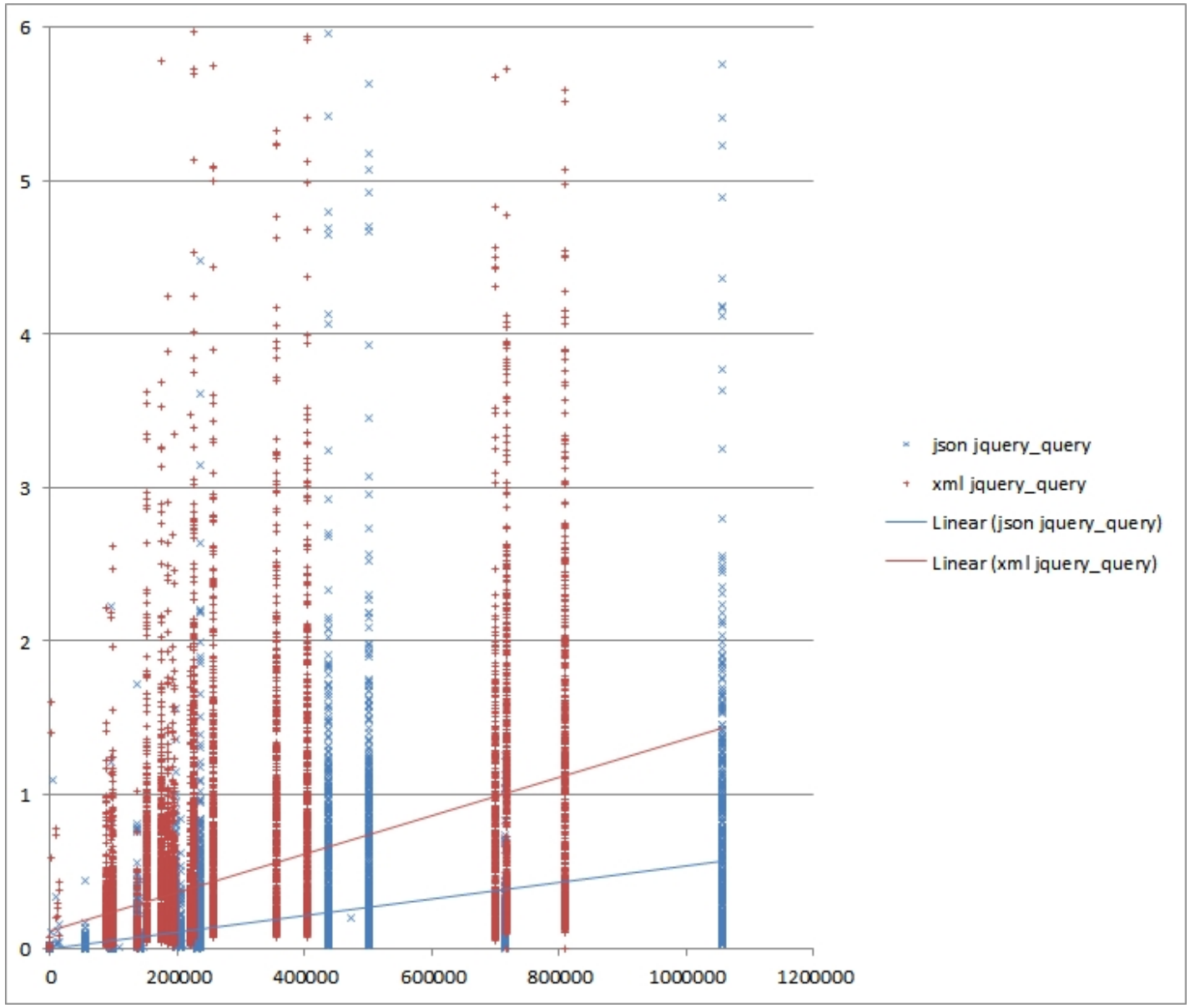


Figure 23: Query speed using jQuery in seconds vs file size



Putting it together

So far we have looked at pieces of the entire work-flow - network speed, parsing and query. Putting it all together what does it look like? Lets assume the developer has chosen the most compact form for each document (JSON and XML), presuming that would also be the most efficient form. What performance can we see across all devices, OS's and browsers tested ? The following figures show the full time for each test using only the most compact form of each document, tested using pure JavaScript and jQuery, both compressed and uncompressed HTTP transfers.

The results are somewhat surprising. it's not a great surprise that jQuery adds a significant performance penalty, but it is a surprise that across all ranges of platforms that the total time for JSON and XML using native JavaScript is effectively identical. Compare Figure 24 and Figure 26. This is still looking at the whole forest and not the trees, but it is surprising to the author that there appears to be no significant performance penalty using XML over JSON in pure JavaScript. However, jQuery does impose a significant performance penalty to both JSON and XML, much more so for XML.

Figure 24: Complete time uncompressed in pure JavaScript in seconds vs file size

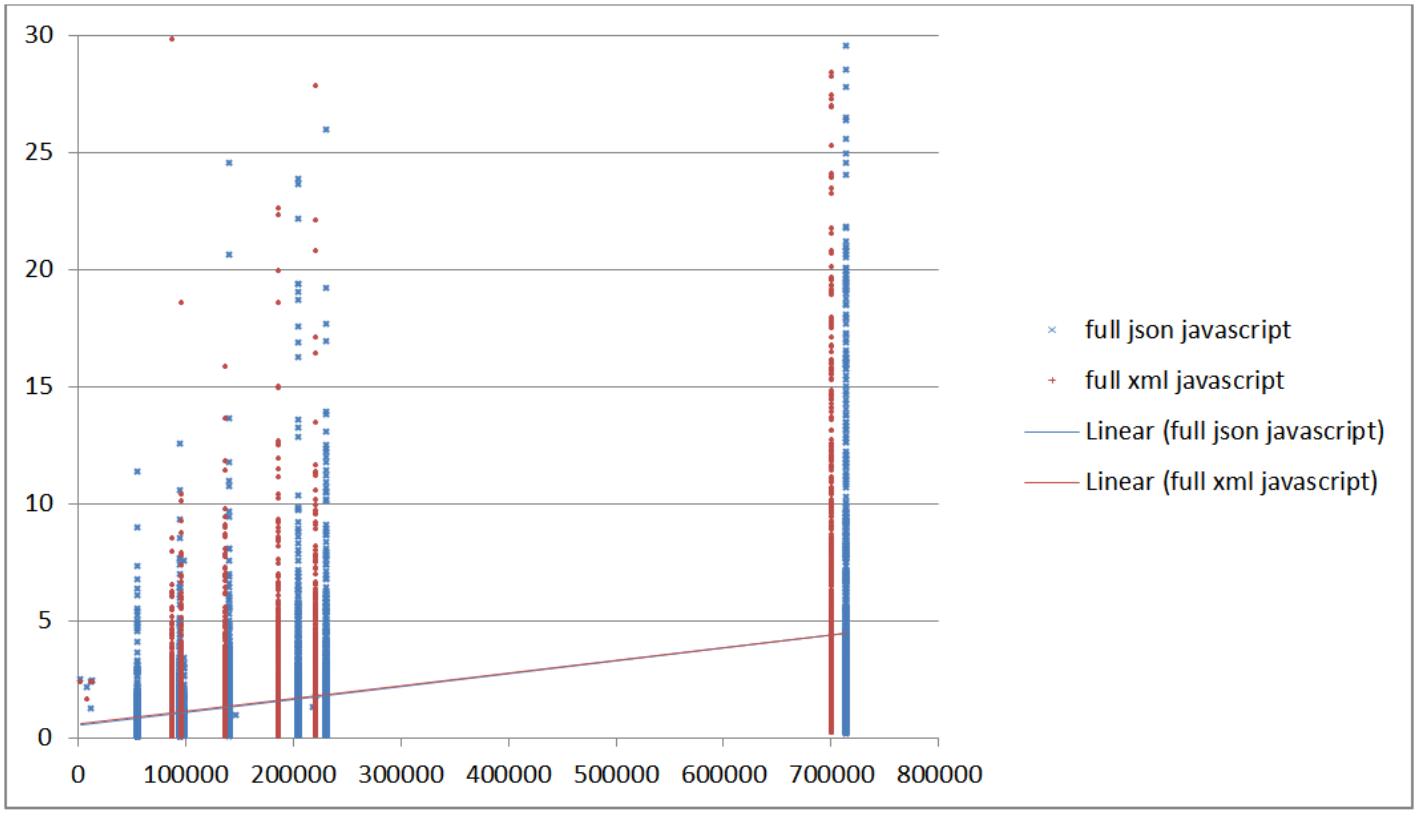


Figure 25: Complete time uncompressed in jQuery in seconds vs file size

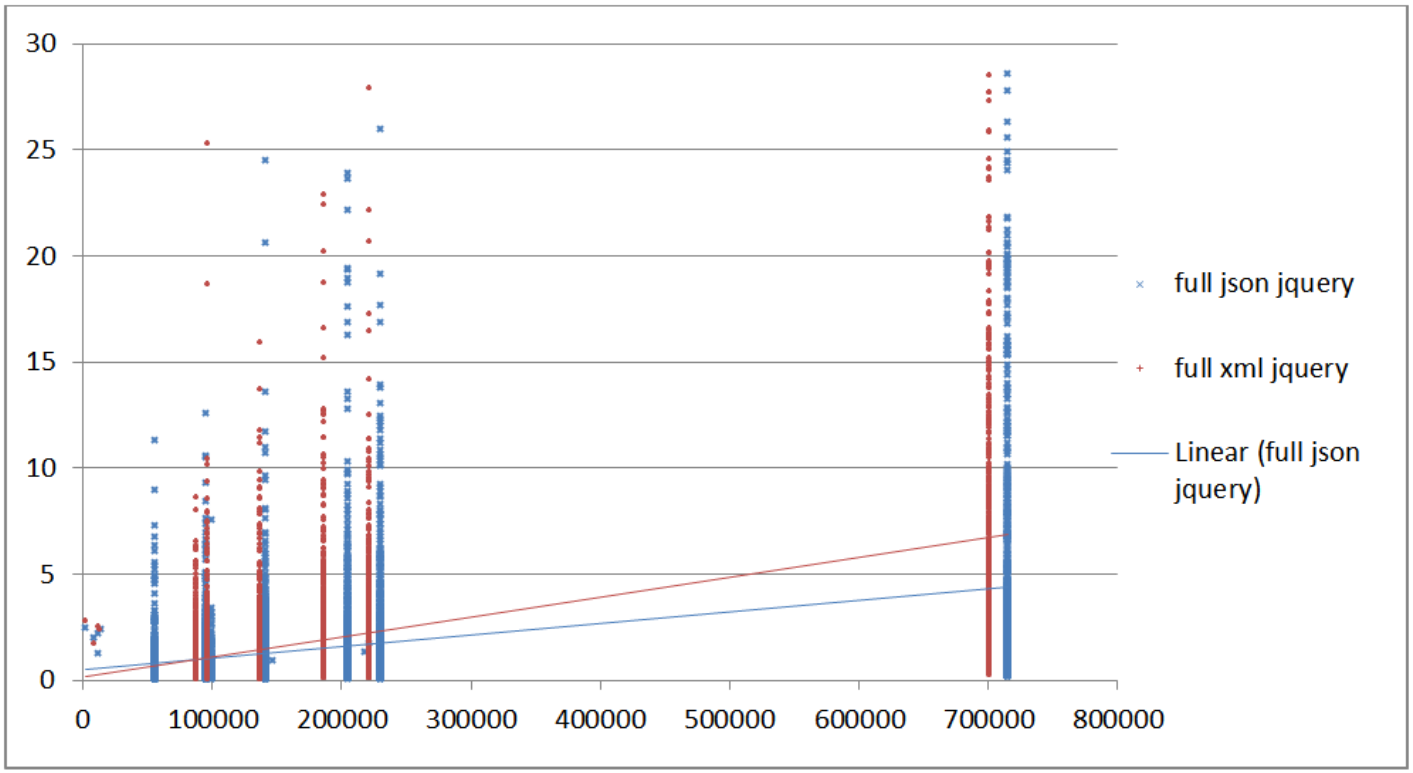


Figure 26: Complete time gzip in JavaScript in seconds vs file size

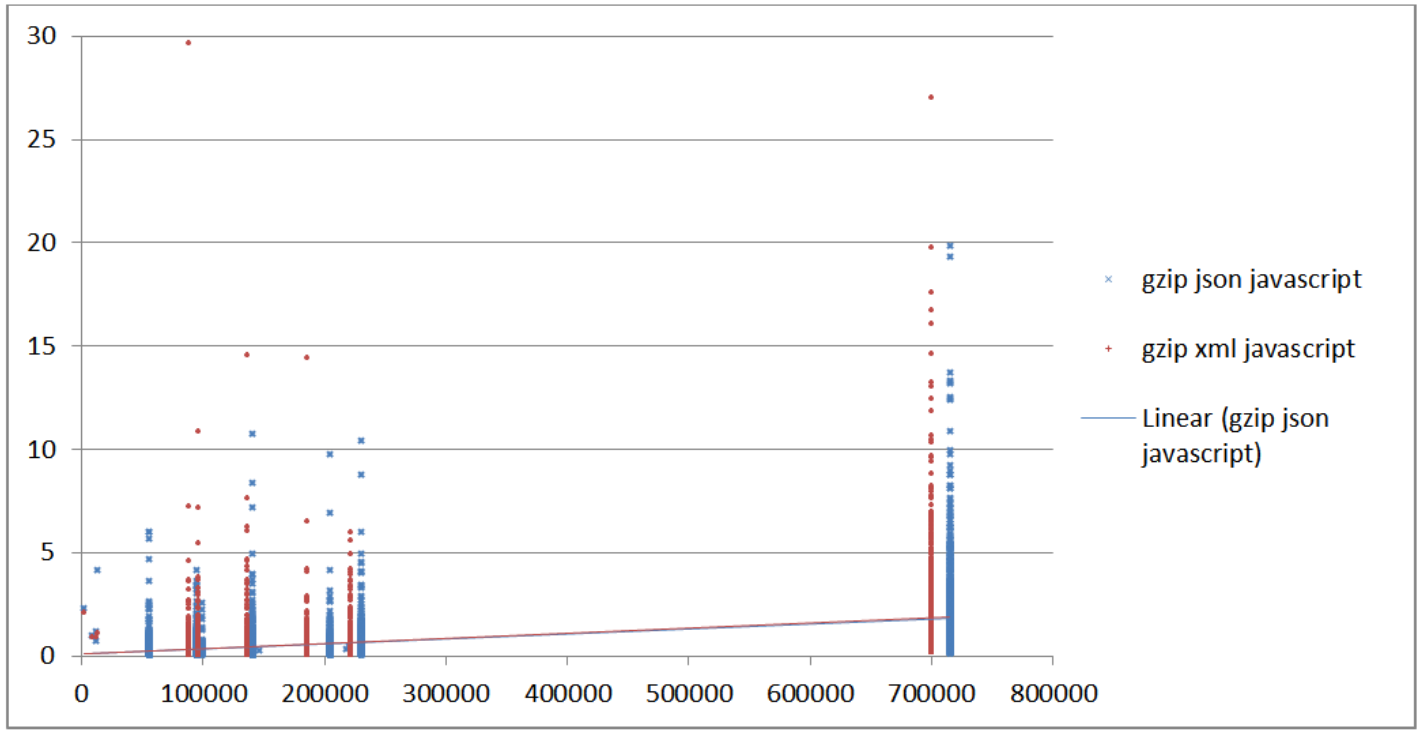
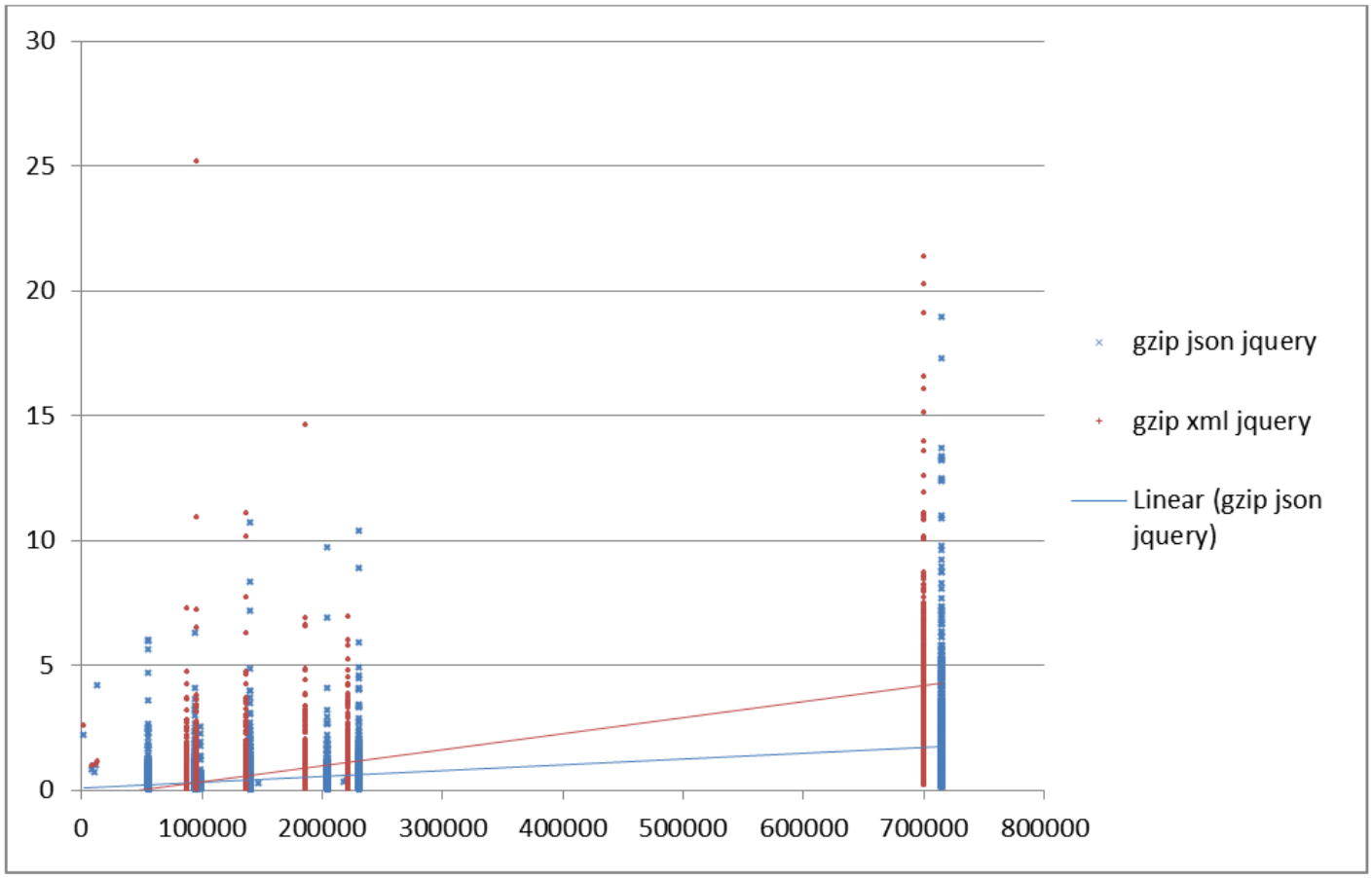


Figure 27: Complete time gzip in jQuery in seconds vs file size



Pulling it apart

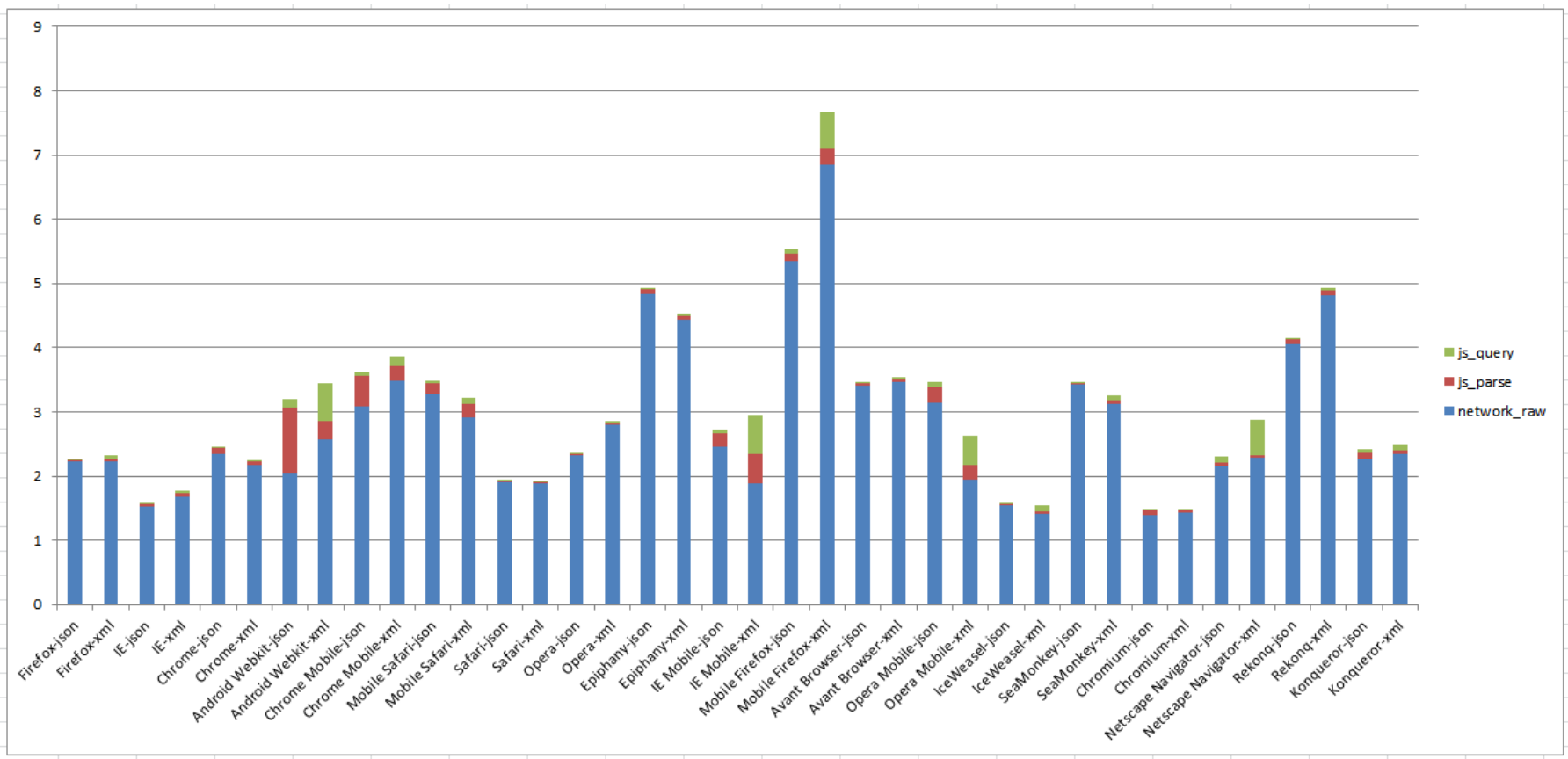
It is interesting to see how everything adds up, but so far we've seen the forest not the trees. What does it look like for a particular user? Where is time spent for a specific document on a particular browser and OS?

Since the experiment collects data from such a wide variety of systems it's difficult to show a meaningful view of this. Averages and even percentiles mean very little when looking at data that spans orders of magnitude. Instead lets look at a couple typical test results which might help make sense of the big picture.

The following are the total of median times (median(data transfer) + median(parse) + median(query)) for the most compact form of the Twitter document in both JSON and XML using both pure JavaScript and jQuery across all browsers. These results span across a variety of devices, some browsers are obvious if they are mobile or desktop and some are not obvious.

In the following charts, the vertical axis is time in seconds (more is slower), and the horizontal axis represents one combination of "User Agent" + either JSON or XML (for every user agent there is two columns of data). Network transfer time is indicated as blue. Parsing time is indicated as red. Query time is indicated as green.

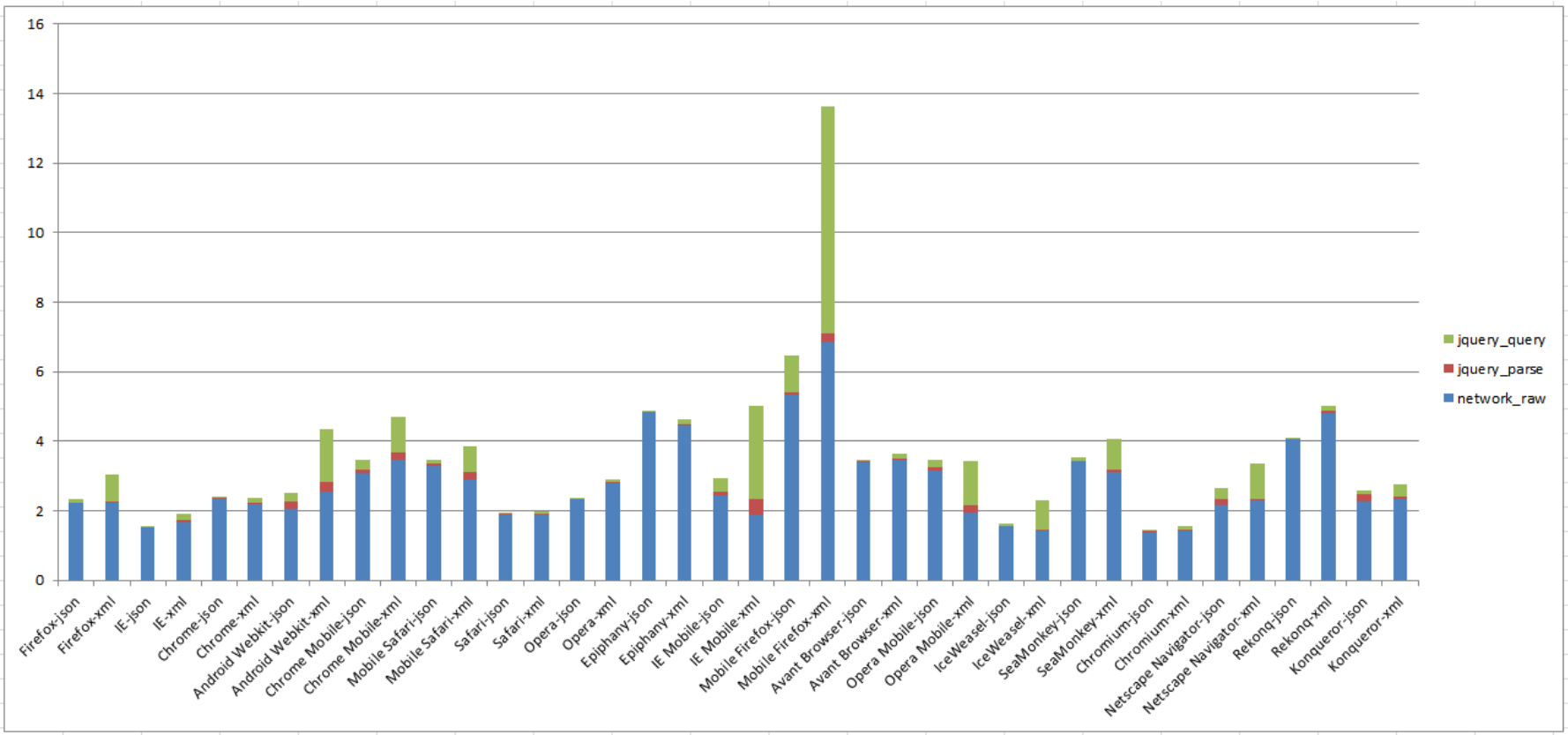
Figure 28: Browser Processing Speed, Uncompressed, JavaScript



Median total times for Twitter document, raw file using JavaScript

In Figure 28 we can see that the total time is dominated by network transmission time. Mobile browsers such as Android Webkit, IE Mobile and Opera Mobile take more time in the parsing and query layer probably due to their slower CPU.

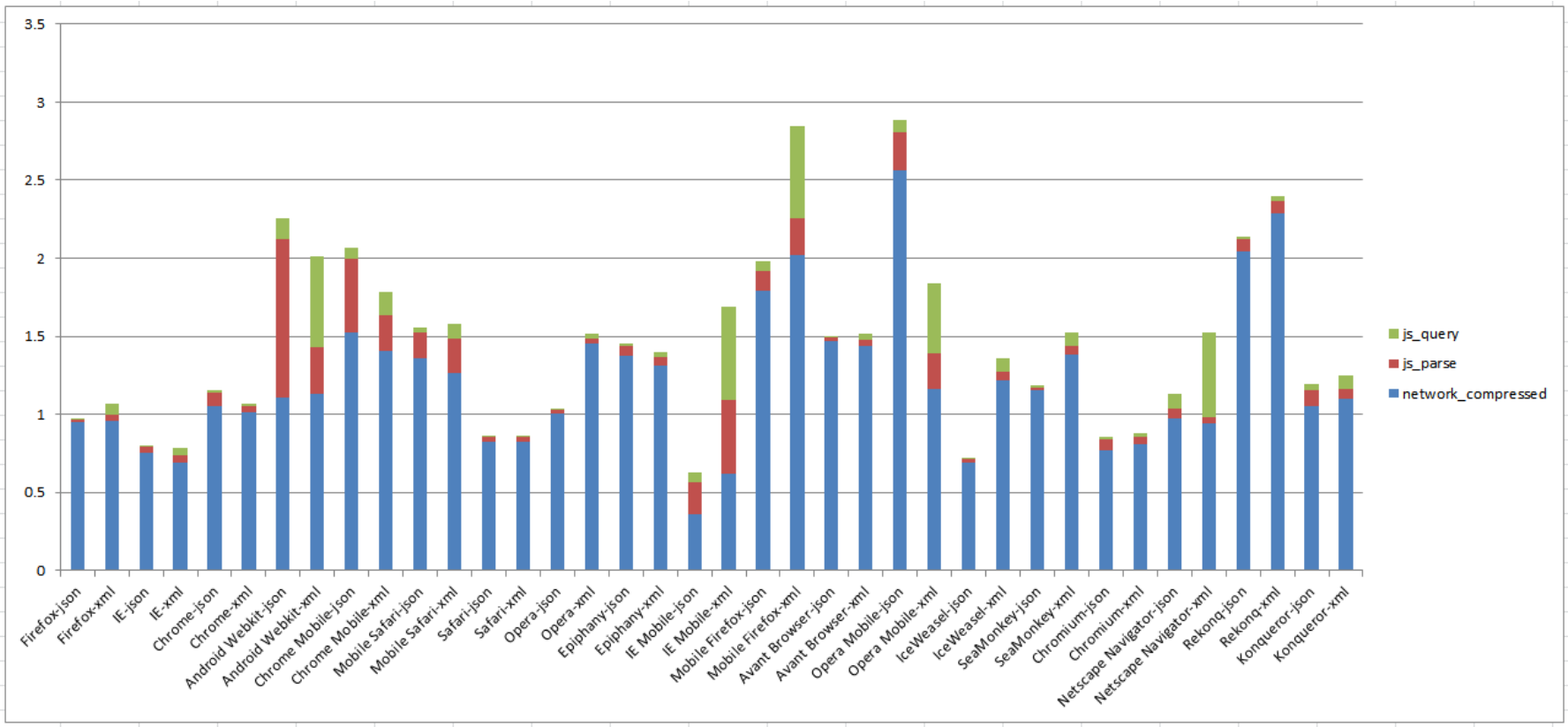
Figure 29: Browser Processing Speed, Uncompressed, jQuery



Median total times for Twitter document, raw file using jQuery

In Figure 29 notice the change in the vertical access to accomidate for large time spent in query and parse on some devices. Network time remains the same but in many more cases are parse and query time a larger contributor.

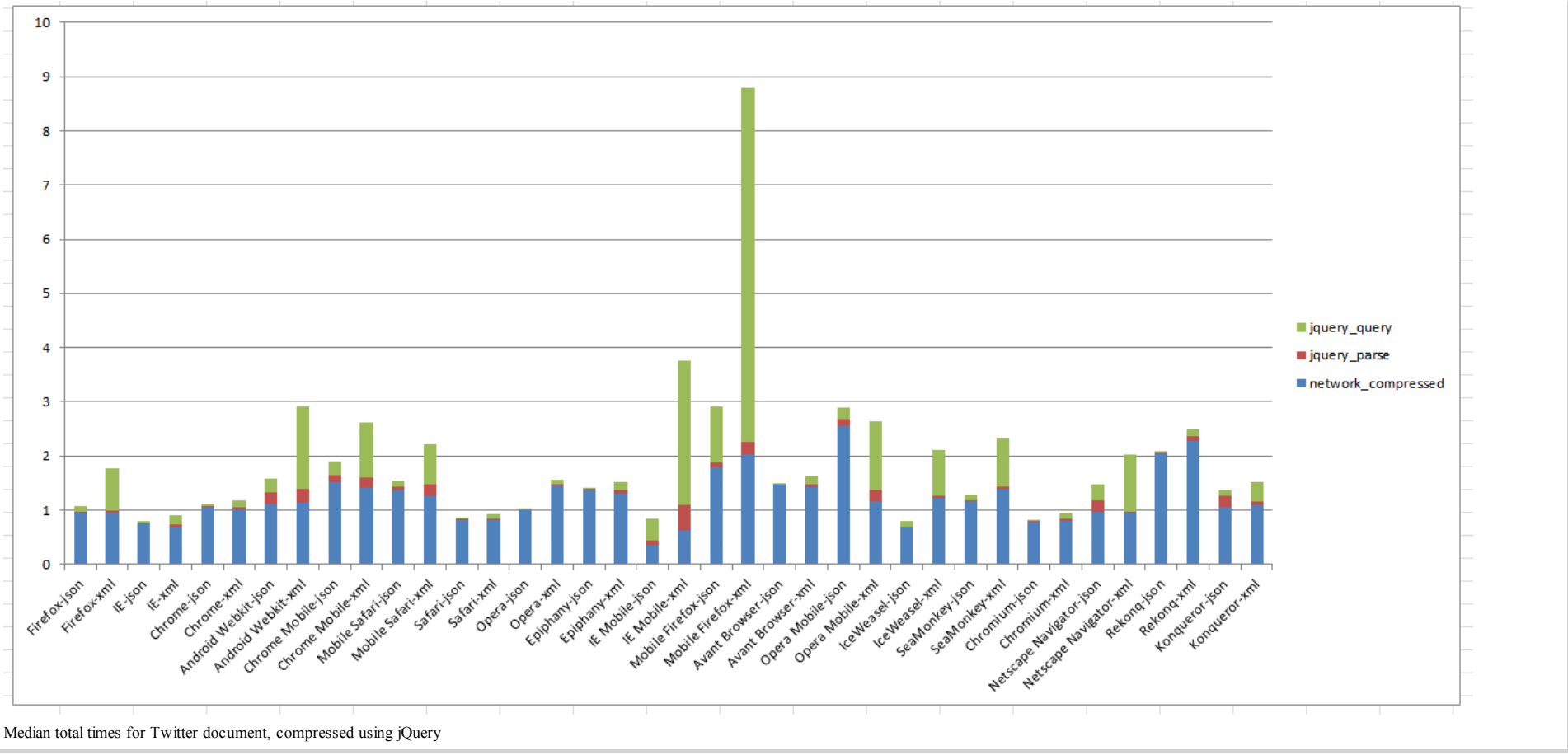
Figure 30: Browser Processing Speed, Compressed, JavaScript



Median total times for Twitter document, compressed using JavaScript

In Figure 30 notice the change in vertical axis. The total time is dramatically smaller revealing the relative times for parse and query being a larger portion of the total time, especially on mobile devices.

Figure 31: Browser Processing Speed, Compressed, jQuery



Median total times for Twitter document, compressed using jQuery

In Figure 31 we can see the impact of using jQuery on mobile devices especially for XML.

Problems and Issues

The topic in general and the experiment in particular are difficult issues to address. The topic itself is "slippery". How does one address a generalized concept where the basis is ill defined and the generalization is hazy? What is trying to be proved? The experiment only addresses a few questions from a huge range of possibilities.

Focusing on the experiment itself there are many specific issues which could be improved and should be addressed in future experiments.

- Browser Focused

  The experiments and this paper focus specifically on the use case of data sent from a "Server" and parsed and queried in a "Browser". There is no testing of server based processing of data. The language in the browser is limited to JavaScript which has very limited choice of technologies so there was no testing of different libraries, languages and technologies.

  Additional experiments involving server to server communications would be useful to collaborate the findings and expand on the range of analysis.

- Browser Errors

  The nature of the client program is such that server errors are not reported. If things go wrong in the client then no record is reported. Only by out of band information have I discovered issues such as individuals who were not able to run the test. For example IE versions 8 and below were particularly problematic and reports indicate it would stop part way through the test. Future experiments should have a means of better reporting of errors.

  In addition approximately 1% of the results showed meaningless data most likely a result of a browser error. An example is tests that report negative times or file sizes. These tests were excluded from the analysis.

- Limited Corpus

  The Corpus was designed to span a wide range of use cases, but ultimately any sample set of data is limited and biased. Future experiments could improve on the variety and focus on if different types of data perform

differently.

- Simplistic parsing tests

  The parsing test does not attempt to do very much with the data besides walk the parsed tree. More complex tests could be performed that attempt to do something with the data such as create new objects or search for specific values. I choose this test as a bare minimum that could be universially applied to the entire corpus but acknowledge that it could be biased in that it does very little with the parsed data so may not reflective of real world use caes.

- Too Much Data

  This experiment produced a lot of data. Compared to many scientific experiments the data size is trivial, but for the purposes of distilling down a few basic principles the amount and variety of test results is daunting especially in its variety. On the other hand, more data is better than less and this experiment improves on many attempts to categorize markup performance in browsers. I am hopeful that the experiment can run for a long duration so that ongoing analysis can be performed. The raw data will be provided for those who wish to analize it themselves.

- Statistical Analysis

  This paper focuses more on showing the range of results with visualizations and trends rather than traditional statistical analysis. However this seems less scientific and exact then ideal. Many more visualizations and analysis would be useful but are limited by forum of publication and time, imagination and skillsof the author. Suggestions on improvement of the analytics and visualizations are greatly welcome.

- Crowd Sourcing

  Due to a lack of an army of volunteers and a vast personal collection of hardware, crowd-sourcing was used to enlist participation. This produced a good number of responses (about 650 as of this writing) but is likely to include self-selected bias. The distribution channel for the solicitation and people who volunteered to run the tests may well not be a statistically good sample set. Future experiments should focus on getting a wider range of people to perform tests.

- Mechanical Turk

  In order to aquire more test samples, Amazon "Mechanical Turk"Mechanical Turk was employed to hire additional testers focusing on mobile devices. This added about 500 additional responses in a 3 day period at an avarage cost of 15 cents (USD) / test.

Conclusions

We have shown that many of the presumptions of "Fat XML", while well imagined, do not hold up to experiment. Given the same document object, one can produce nearly identical sized JSON and XML representations. Network transfer speed is directly related to the document size so is unaffected by the markup given similar size. Compressed documents in all formats even very "Fat" representations of JSON or XML compress to nearly identical size which is an indicator that they contain approximately the same entropy or information content and transferring these documents to a wide variety of devices takes effectively the same time per device. Parsing speed varies on the technique used. Pure JavaScript parsing generally performs better with XML then with JSON but not always, while Query speed generally is faster for JSON, but again, not always. Overall using native JavaScript the use of XML and JSON is essentially identical performance for total user experience (transfer plus parse plus query), however use of the popular JavaScript library jQuery imposes a steep penalty on both JSON and XML, more-so for XML. [4]

From a programming perspective accessing both JSON and XML in a generic fashion, using either pure JavaScript or jQuery is very similar in complexity and difficulty. Not shown is the advantage of accessing JSON objects as JavaScript objects using "dot notation" which provides a programming advantage, however the evolution towards using query languages to access JSON (or XML) such as jQuery largely negates that advantage. Future enhancements in JavaScript libraries and cross compilation technologies (such as CoffeeScriptCoffeScript, GWT and DartDART) may well equalize these discrepancies but that is yet to be seen. The fact that hand-coded JavaScript query of XML can perform as well as query over JSON does suggest that libraries such as jQuery could be optimized for similar performance and especially cross compilers should be able to achieve similar performance. On the other-hand, the wide adoption of JavaScript libraries even in the face of significant performance degradation even for JSON suggests that developers are not as concerned about performance considerations over ease of programming. This may be because the data layer of the application is small compared to the other components of the application such as GUI and business logic.

Suggestions to Architects and Developers

Architects and developers who are seriously interested in maximizing performance should consider the following.

- Use HTTP Compression

  The use of HTTP compression, regardless of the device, operating system, browser or markup language is the biggest factor in total performance, and by inference, user experience. Use of HTTP Compression should be used in all cases where large amounts of data is being transfered from server to client.

- Optimize your markup

  Optimizing markup for transmission and query provides a second order performance enhancement. Sometimes this is at the cost of usability. Compression effectively eliminates the advantages of optimized markup for transmission purposes, but parsing and query times are affected by the particulars of the markup form. Choosing the right balance between ease of programming and transmission should be seriously considered. If your target is specific known devices then optimizing for those devices may be benficial.

- Use optimized libraries or hand coded JavaScript.

  Use of libraries to ease development effort can have a significant performance penalty. This penalty is in general much greater than the penalty of which markup format you are using. If performance is your most important factor then avoiding unoptimized libraries will be your biggest performance gain regardless of the markup format. Look to the future for this issue to be improved. If more developers prioritize performance then library and cross compilation vendors are likely to focus more on performance. In any case your choice of a parsing and query library are by far the biggest performance factor beyond compression, much more so than the markup

format. Look to Cross Compilation technologies such as GWT and Blink to provide machine optimized JavaScript code much like the compilers for C, C++ and Java do for non browser environments.

- Know your users.

  Performance varies *vastly* across browsers, operating systems and devices. If you know your users and their platforms you can make better choices about which technologies and formats to use. But be aware that implementations change frequently and what is efficient today may be slow tomorrow and visa-versa so optimizing too tightly for particular devices may be detrimental in the future. Mobile devices are particularly prone to differences in performance but are often hard to optimize because the biggest hit is network speed which is usually not under the programmers control. However programming for minimum network usage will provide the best advantage for mobile devices. Use of HTML local storage, JavaScript based applications which avoid passing page markup and instead pass data and allow page transitions to be performed without a round trip to the server are likely going to provide good user experience.

- Markup doesn't matter.

  The choice of JSON vs XML is nearly indistinguishable from a performance perspective. There are a few outliers (such as Mobile and Desktop Firefox using jQuery) where there is a notable difference but as time goes on and vendors pay attention to user feedback I expect these differences to be reduced. But for most uses the difference in markup choice will result in little or no user noticeable difference in performance and end user experience. There are significant browser architectural changes coming such as HTML5 and Chrome Blink Blink. We don't know what performance changes these will incur but evidence suggests that performance in the browser is a main goal of new browser technologies. Looking to the future the landscape may change.

- Markup Matters

  Contrary to myth, performance varies very little with different markup formats in current devices and software. However the shape and ease of use of markup formats can matter for developers (both client and server). As can be seen with the use of libraries such as jQuery, performance is often trivially tossed away in exchange for ease of programming - even with JSON. Sometimes that is a reasonable tradeoff. [5]

  Engineering is about balanceing compromises. Make the compromise that maters to you, your product, your business and your customers. If performance matters to you and your applications - do what it takes to achive maximum performance. If it doesnt matter then use whatever technology is easiest for you. Often the "you" is many people in an organization or across organizations. If its harder for the producer of data to change formats then work with them to use their format. If its harder for the consumer to change formats then work with the producers to produce the format the consumer needs.

  In any case make this decision based on facts not myth.

- Is Data really your problem ?

  This paper focuses exclusively on data transmission, parsing and querying. It may well be that in your application that component is a small piece compared to display and business logic. If the data layer is not a significant part of your performance or development problem then it may not be worth the effort to optmize it. As your application evolves your data use may change so always be open to re-evaluating descisions made early in the design processs.

- Don't Trust Anyone

  Don't believe blindly what you are told. Perform experiments, test your own data and code with your own users and devices. What "seems obvious" is not always true.

  As always with engineering; experiment, develop, test. And test again.

  The source code and data corpus is published at https://code.google.com/p/jsonxmlspeed/. The raw results for this experiment will be published along with this paper for peer review. Test it. Dispute it. Come up with better tests and publish the results.

  It is hoped that an ongoing interactive web site will be developed to continue to track, analize, and monitor this research.

References

[AJAX Performance] AJAX - JSON vs. XMLhttp://www.navioo.com/ajax/ajax_json_xml_Benchmarking.php

[XML vs JSON] Edward A. Webb, XML vs JSONhttp://www.edwardawebb.com/tips/xml-json

[JSONLIGHT] janu bajaj, My Open Source Initiativehttp://bajajblog123.blogspot.com/2012/07/json.html

[XMLFAT] Douglas Crockford, 2006; JSON: The Fat-Free Alternative to XMLhttp://www.json.org/fatfree.html

[MISCONCEPTION] Dr. Steven Novella; http://tech.groups.yahoo.com/group/skeptic/message/30893

[Saxon] Saxon XSLT and XQuery Processorhttp://www.saxonica.com

[Blink] Chrome Blinkhttp://www.theverge.com/2013/4/3/4180260/google-forks-webkit-with-new-blink-rendering-engine-for-chrome

[GWT] Google Web Toolkithttps://developers.google.com/web-toolkit/

[jQuery] jQueryhttp://jquery.com/

[CoffeScript] CoffeeScripthttp://coffeescript.org/

[DART] DARThttp://www.dartlang.org/

[Mechanical Turk] Mechanical Turkhttps://www.mturk.com/mturk/

[Apache] Apache HTTPD serverhttp://httpd.apache.org/

[1] Although interestingly he claims that JSON is neither a document format nor a markup language."JSON is not a document format. It is not a markup language."

[2] Amazon SQS service was used for the messaging queue http://aws.amazon.com/sqs/

[3] The service "user agent info" http://user-agent-string.info/ was used for this enrichment.

[4] Experiments comparing different JavaScript libraries would be useful to see if jQuery is unique or is representatvie of JavaScript query libaries in general.

[5] It is interesting to the author that the use of libraries like jQuery entirely eliminate the native advantage in JavaScript of using JSON as the data format ("dot notation"), add significant processing overhead and yet is strongly promoted while XML usage is discouraged under the guise of it not mapping well to native JavaScript data structures and its slow performance. Compare the jQuery code for JSON and XML and you can see the programming difference is nearly indistinguishable.