CONTENT

1 INTRODUCTION

This research studied actual selenium WebDrivers, that available on linux platform. Sometimes it is necessary to write automated tests for website or application to communicate with website (what can't be done via API) etc, which can't be done by simply parsing html, since webpages contains complex websites with javascript logic, that has to be processed in the right way.

Here selenium can do the stuff by communicating with browser via special driver aka WebDriver. But which one is better, faster, uses less system resources and performing well?

2 TESTS

2.1 Setup

Java platform was used to test selenium WebDrivers.

Table 1 displays core system information on which tests were run. All updates were installed for 2015-09-05. Table 2 displays WebDrivers participating in tests.

Table 1 – System information

System model	HP Pavilion 17-e182sr
CPU	Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz
RAM	8+4 GB SODIMM DDR3 Synchronous 1600 MHz (0.6 ns)
Motherboard	Hewlett-Packard PDPWT058J6C1C4
OS	Ubuntu 14.04 LTS 64-bit (3.13.0-62-generic kernel)

Table 2 – WebDrivers participating in tests

Browser name	**Browser version**	**Selenium lib version**	**Other**
Phantomjs.	1.9.8 (latest)	2.41.0 (latest for phantomjs WebDriver)	
Firefox	40.0.3+build1-0ubuntu0.14.04.1 (ubuntu repository).	2.47.1 (latest)	
Chromium	44.0.2403.89 Ubuntu 14.04 (ubuntu repository)	2.47.1 (latest)	ChromeDriver: 2.15 (ubuntu repository)
Chrome	44.0.2403.157 (latest)	2.47.1 (latest)	ChromeDriver: 2.18.343837 (latest)

Each WebDriver was run with default settings on virtual display (preset environment variable “**DISPLAY=:1.0**”), which was started by following command:

```
Xvfb :1 -screen 0 800x600x8
```

Each browser requests were made with “**Cache-Control: no-cache**” header, which tells services to not return cached content. Since there is no API in selenium to add custom headers, browsers were run through proxy (see browser-mob-proxy), which added the header to each request. There were issue with phantomjs running through proxy (50% attempts pages were not loaded), but fortunately there was another way to add custom headers for phantomjs driver, so phantomjs wasn't run through proxy. During initialization each WebDriver visited page that returned headers from request to verify that header was set.

Memory and CPU usage are summed for all subprocesses, that related to WebDriver (if WebDriver instance creates 11 processes then CPU and memory usage were summed).

Memory usage in the tests is the sum of “Private_Dirty” fields from “**/proc/{pid}/smaps**” file, which is equal to value in “memory” column in **gnome-system-monitor** tool.

CPU usage in the tests is the sum from “**ps -p {pid} -o %cpu**” tool (100% means 1 core fully loaded).

Swap file was disabled (there always were free ~4GB of RAM).

JVM was started with preallocated heap size of 2gb to not use garbage collector and heap allocation during tests.

For each WebDriver window resolution were set to 1336x766.

Since some phantomjs version are not loading images by default, such argument was forcibly set to load images “**--load-images=yes**”.

Tests were run at least 3 times to ensure, that results are stable. WebDrivers in the tests were tested sequentially (only one WebDriver were run at the same time).

PC was connected to network through Ethernet interface (RJ-45).

2.2 Create/Destroy test

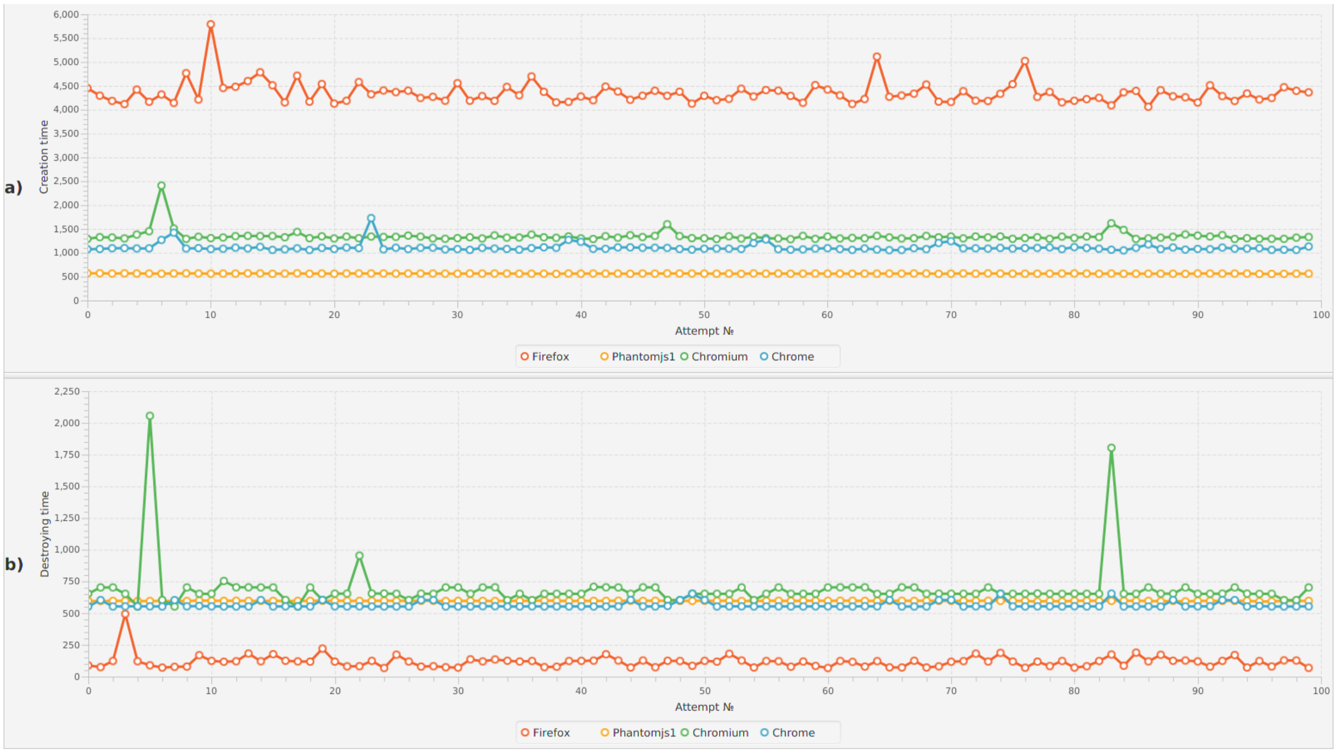
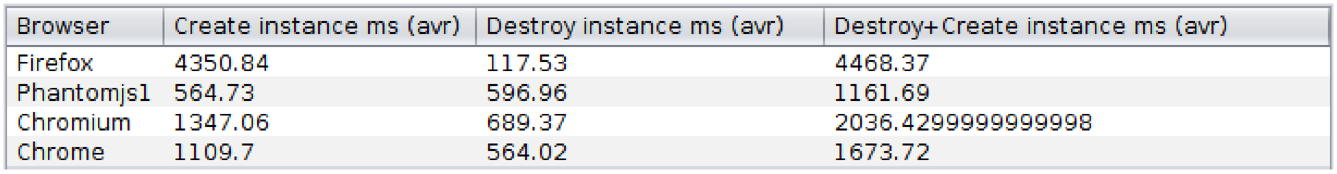


Figure 1 – WebDriver create/destroy instance test. (a) Time to create instance; (b) time to destroy instance

Table 3 – Summary info of create/destroy instance test

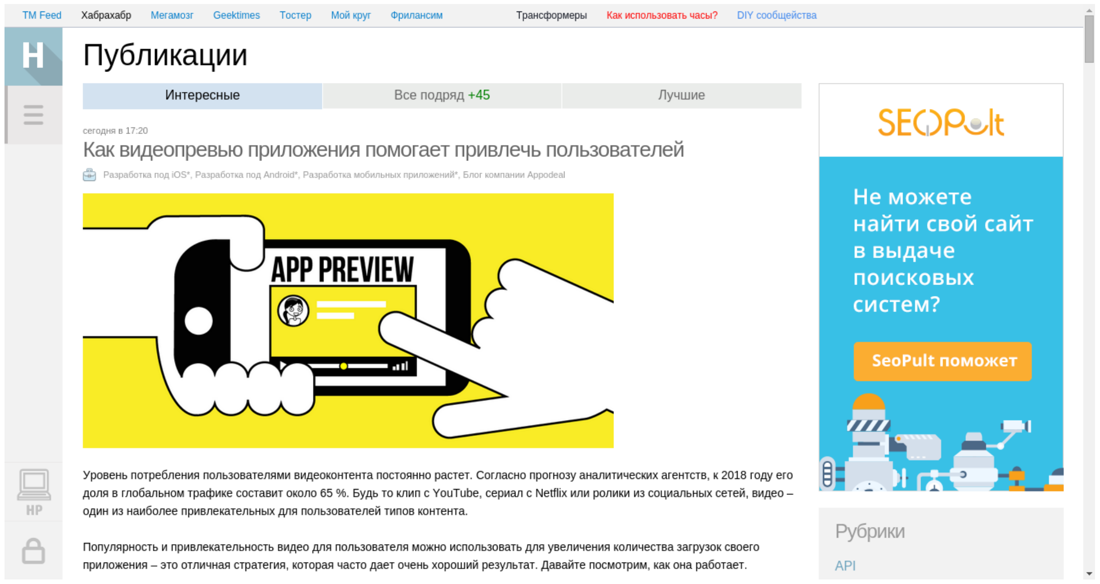


Browser	Create instance ms (avr)	Destroy instance ms (avr)	Destroy+Create instance ms (avr)
Firefox	4350.84	117.53	4468.37
Phantomjs1	564.73	596.96	1161.69
Chromium	1347.06	689.37	2036.4299999999998
Chrome	1109.7	564.02	1673.72

Firefox definitely lost in the test. if application where WebDriver will be used has frequent, but short living jobs then best solution will be phantomjs or chrome/chromium WebDrivers.

2.3 Screenshot capture test

Phantomjs and firefox WebDrivers capture screenshots of whole page without scroll when Chrome/Chromium capture only area which is visible to user together with scroll, can be fixed with workaround by increasing window height to document height before capturing screenshot. See figures 2 – 3.


TM Feed Хабрахабр Мегамозг Geektimes Тостер Мой круг Фрилансим Трансформеры Как использовать часы? DIY сообщества

Публикации

Интересные | Все подряд +45 | Лучшие

сегодня в 17:20

Как видеопревью приложения помогает привлечь пользователей

Разработка под iOS*, Разработка под Android*, Разработка мобильных приложений*, Блог компании Appodeal

Уровень потребления пользователями видеоконтента постоянно растет. Согласно прогнозу аналитических агентств, к 2018 году его доля в глобальном трафике составит около 65 %. Будь то клип с YouTube, сериал с Netflix или ролики из социальных сетей, видео – один из наиболее привлекательных для пользователей типов контента.

Популярность и привлекательность видео для пользователя можно использовать для увеличения количества загрузок своего приложения – это отличная стратегия, которая часто дает очень хороший результат. Давайте посмотрим, как она работает.

SEOPult

Не можете найти свой сайт в выдаче поисковых систем?

SeoPult поможет

Рубрики

API

HP

Figure 2 – Captured screenshot by chrome driver (visible to user area)

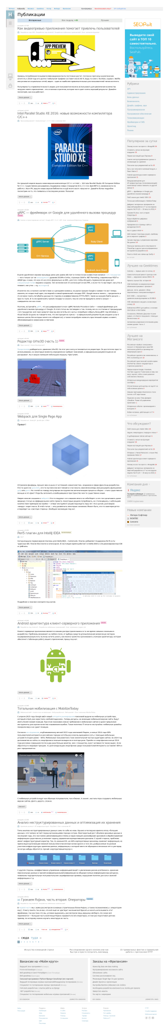


Figure 3 – Captured screenshot by phantomjs and firefox driver (visible to user area)

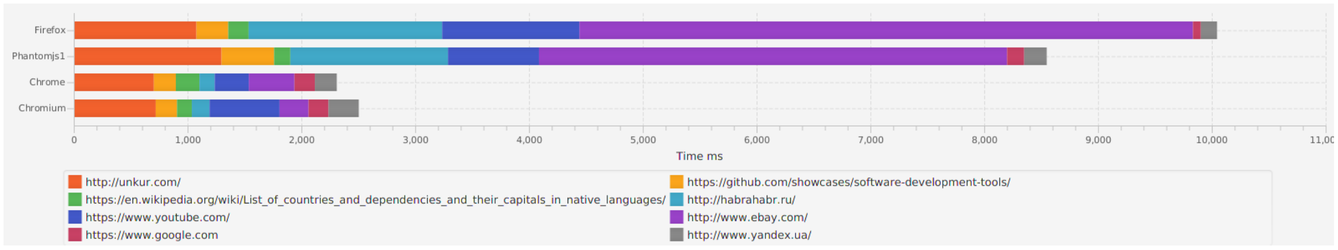


Figure 4 – Time to get screenshot (Chrome/Chromium driver takes less time, but don't take screenshot of full page)

Depending on what application required (screenshot of whole page or screenshot of area, that visible to user) both solutions has place to live. But when it is possible to do a trick (increase window height by calculating document height) with chrome/chromium driver to get screenshot of full page, it is not possible to get screenshot of area, that is visible to user with phantomjs and firefox driver.

Size and color depth of virtual display doesn't matter, since screenshot resolution and color depth is bigger than **800x600x8**.

2.4 Javascript prototype test

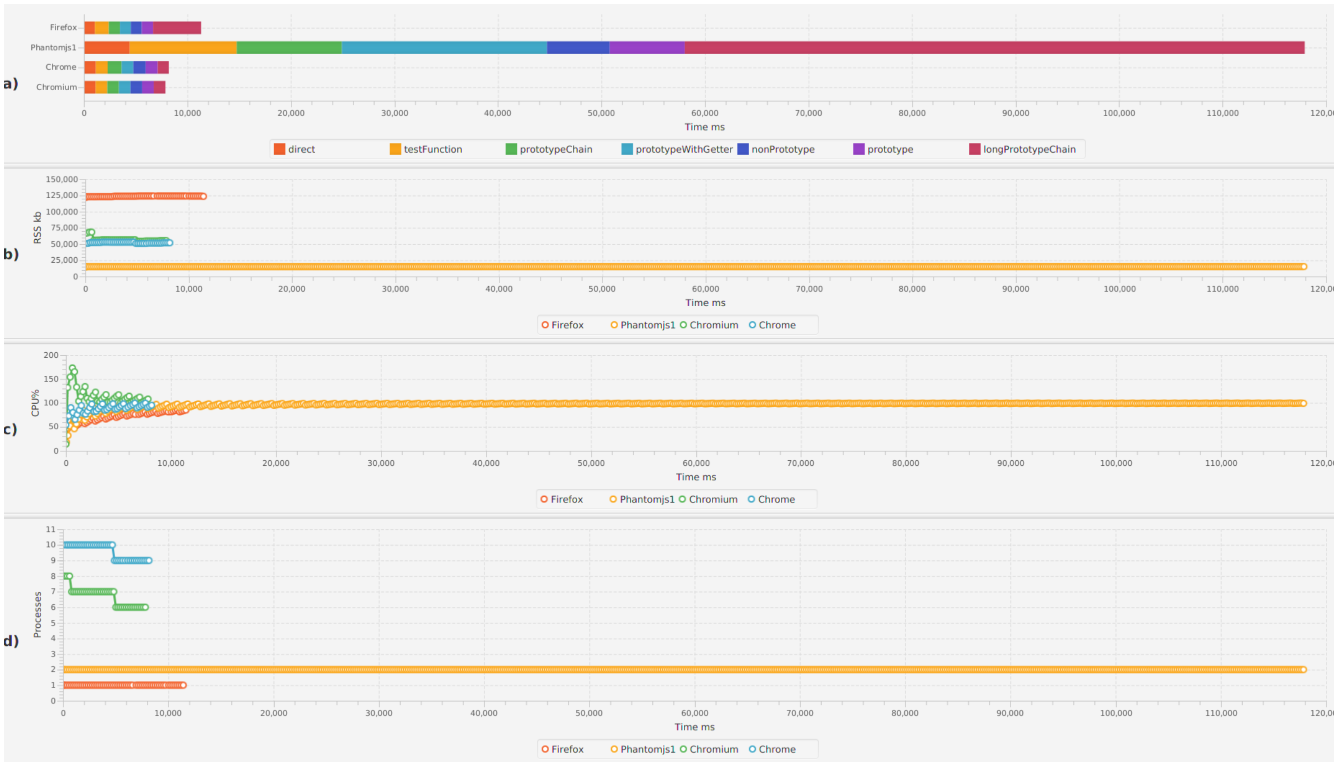


Figure 5 – Javascript prototype test. (a) Time to process test; (b) memory usage; (c) CPU usage; (d) number of processes

See appendix A to see javascript code, that was used for the test.

Phantomjs definitely lost javascript performance test, it took less memory, but time to process javascript logic is more valuable.

Firefox javascript performance is near to chrome, but lost in test with long prototype chain hierarchy to Chrome/Chromium. Also Firefox took much more memory.

Looks like at the moment Chrome/Chromium has fastest javascript engine. Chrome/Chromium will be good solution to process websites with complex javascript logic.

2.5 Web surfing test

For the test local copy of websites were made by WebHTTrack utility to eliminate as much as possible network influence. But utility did not load everything successfully (one site can miss css file, another one js file), but all pages are the same for all WebDrivers (in case when web site returns one page for Firefox and another one for chrome).

Figure 6 display result of test where WebDriver loads pages and finds all links by “**//a[@href]**” XPATH using selenium java API. Figure 7 display result of test where WebDriver loads pages.

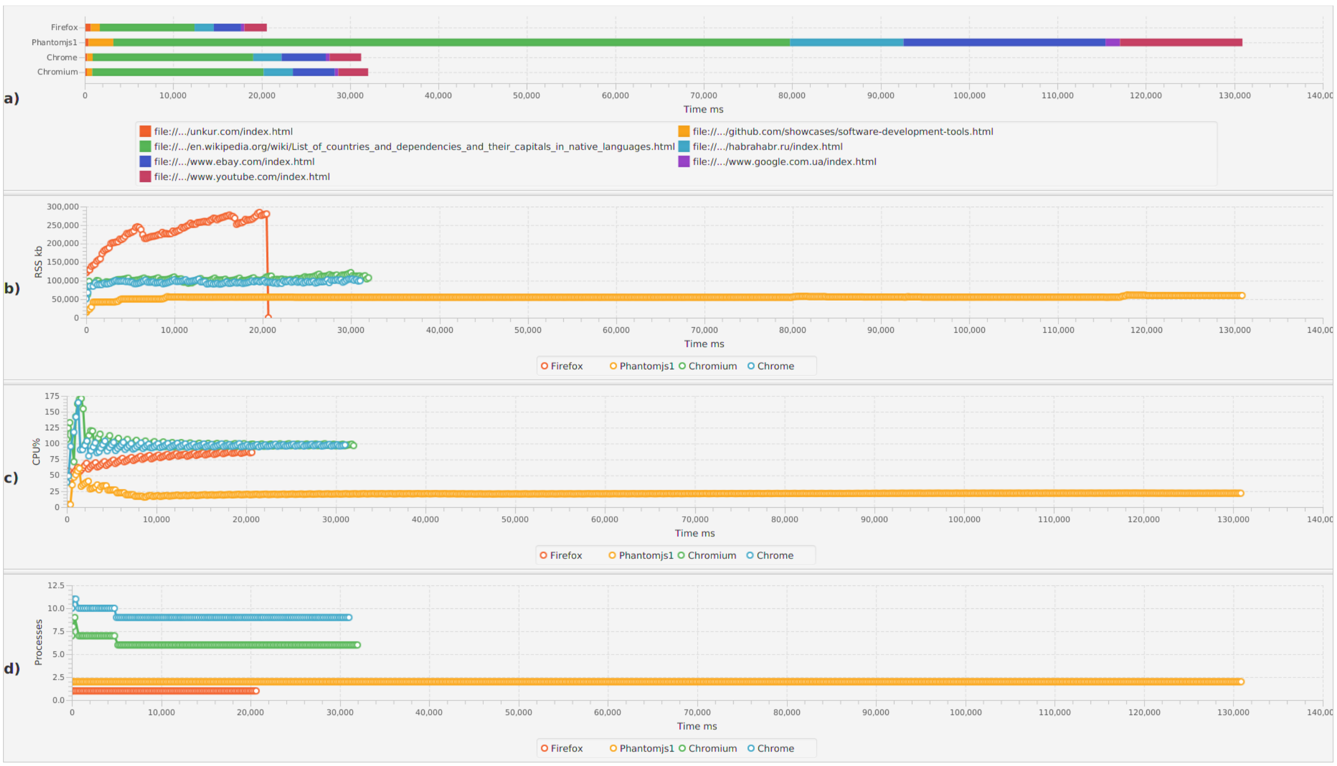


Figure 6 – Test results of loading web pages and finding all links. (a) Time to process test; (b) memory usage; (c) CPU usage; (d) number of processes

By test results from figure 6 phantomjs lost since it took much time to process. Firefox processed test faster than Chrome/Chromium, but used much more memory.

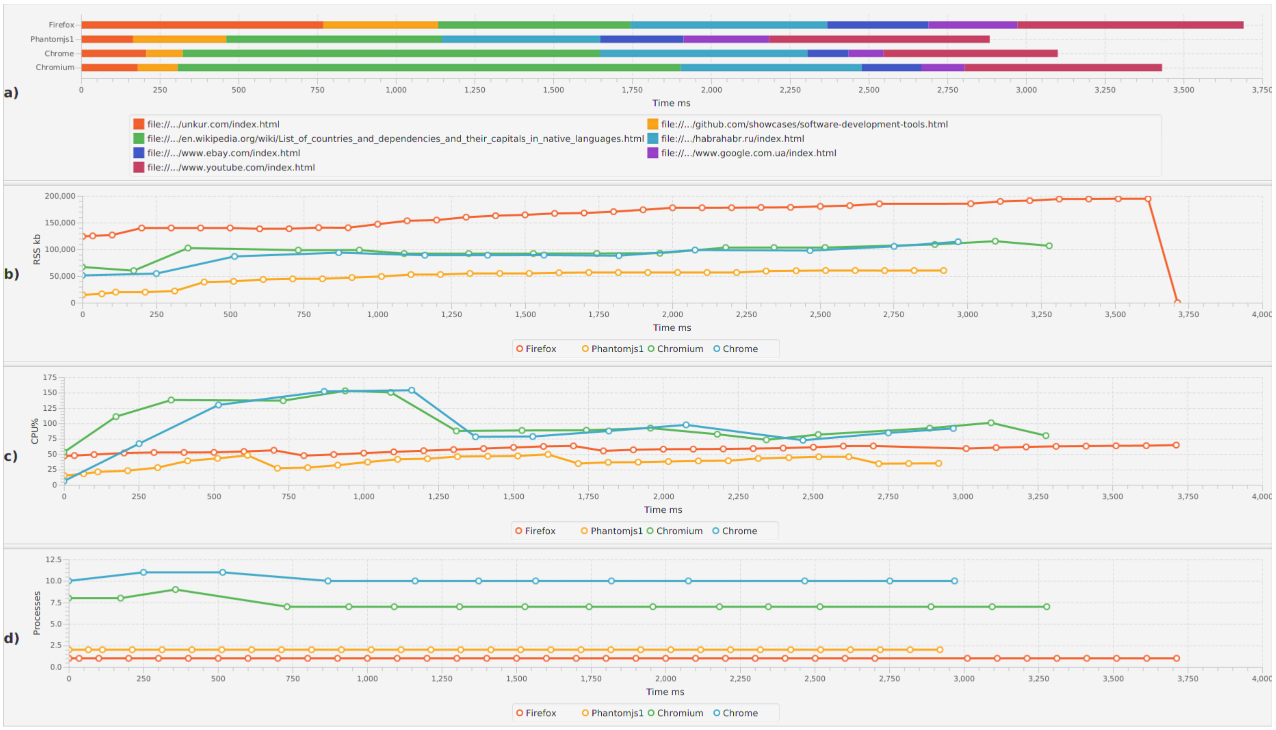


Figure 7 – Test results of loading web pages. (a) Time to process test; (b) memory usage; (c) CPU usage; (d) number of processes

By test results from figure 7 phantomjs wins, it loaded webpages a bit faster and use less memory and CPU resources than Chrome/Chromium and Firefox, maybe due to GUI overhead.

2.6 Group cities by countries test

The most of the time test processing XPATHs using selenium java API. During test WebDriver does following steps:

1. Starting from the page finds all URLs, which contains cities by alphabet (A, B, C etc) using “**//*[@id="mw-content-text"]/ul[2]/li/center/a**” XPATH;
2. Visit each URL from step 1 to parse cities and corresponding country links using XPATH;
3. Find table rows by “**//*[@id="mw-content-text"]/table[@class="wikitable sortable jquery-tablesorter"]/tbody/tr**” XPATH and then from each row parse city name “**.//td[1]/a**” and corresponding country URL “**.//td[2]/a**”;
4. Visit each country URL once from step 3 to resolve country ISO_3166-2 code (by regex “**\/wiki\/ISO_3166-2:(\w+)**” from java application on returned page content).

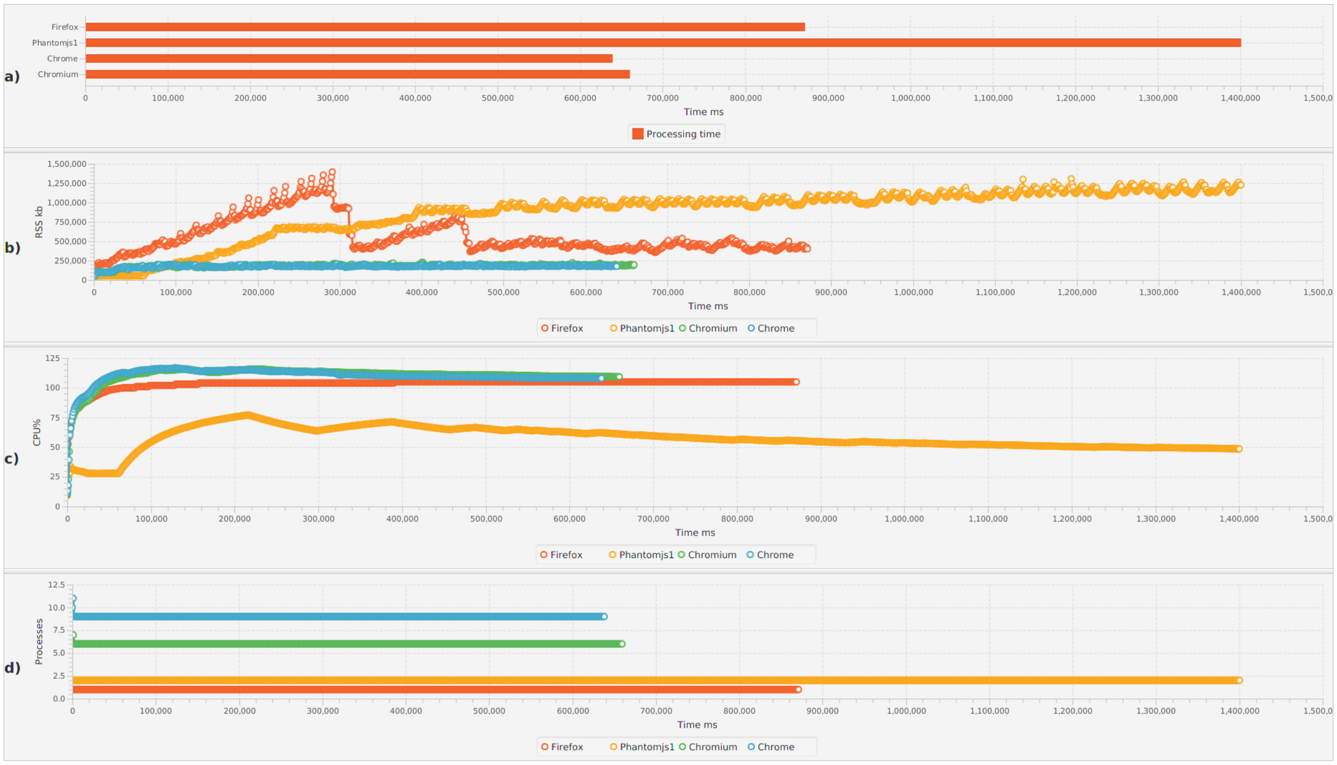


Figure 8 – Test results of parsing wiki to group city names by country ISO codes. (a) Time to process test; (b) memory usage; (c) CPU usage; (d) number of processes

Chrome/Chromium definitely wins the test since it uses much less memory (and memory usage is very stable), than phantomjs and firefox, also it pass test faster.

Memory usage was significantly increased by Firefox during surfing web pages (to resolve country ISO_3166-2 code, at the end when most of country codes were resolved memory usage stabilized), during processing XPATHs Firefox released most of taken memory.

Phantomjs is slowest in the test and consumes more memory the longer it runs, plus phantomjs didn't released memory as it did Firefox. But it used less CPU resources.

Strange that in WebSurfing test Firefox processed XPATHs faster than Chrome/Chromium, but in this one it lost. Maybe it take more time to process XPATHs on found elements (rows), which takes the most time in the test.

CONCLUSIONS

Phantomjs can be good solution in order to quickly start to work. It doesn't required special environment on server and will work out of the box. It use less memory in general, but in long running test with multiple pages it used the more memory the more it runs, when Chrome/Chromium memory usage didn't grow.

In case when faster javascript engine required or phantomjs crashes on specific websites, Chrome/Chromium will be good solution since it use much less memory than Firefox and has faster javascript engine, so will process web pages with complex javascript logic faster than Firefox.

One more situation when Chrome/Chromium and Firefox have to be used is extensions/plugins, which can enhance web-content (adblock, highlighter, content enhancers etc) and of course is not available for phantomjs. Also during debug Chrome/Chromium and Firefox will be much more helpful, since developer can see what is going on in realtime and doesn't have to capture screenshots on each step.

Main disadvantages of firefox that it uses much more memory per WebDriver instance and takes much time to create/prepare instance.

One distinct feature of Chrome/Chromium is number of processes for one instance, which is near to ~10 processes per WebDriver instance, but this should not be an issue. Anyway some kind of cleanup mechanism should exist to destroy uncontrolled Phantomjs/Firefox/Chrome processes, which may not be destroyed successfully (for example deamon process, which destroys such processes by names if their lifetime is more than 10 minutes, which can be determined by standard linux utils).

APPENDIX A – Code of javascript prototype test

```
/**
 * Main function of the test
 */
function testPerformance() {
    return {
        direct: getExecutionTime(testDirect),
        testFunction: getExecutionTime(testFunction),
        nonPrototype: getExecutionTime(testNonPrototype),
        prototypeChain: getExecutionTime(testPrototypeChain),
        prototype: getExecutionTime(testPrototype),
        longPrototypeChain: getExecutionTime(testLongPrototypeChain),
        prototypeWithGetter: getExecutionTime(testPrototypeWithGetter)

    }
}

/**
 * Value that will be used in functions.
 */
var DELTA = 10;

/**
 * Number of iterations^3
 */
var ITERATIONS = 1000;

function getExecutionTime(f) {
    var start = new Date().getTime();
    f();
    var end = new Date().getTime();
    var diff = end - start;
    return diff;
}

/**
 * Prototype chain with 5 children and property on parent object.
 * ChildE->ChildD->ChildC->ChildB->ChildA->Parent->delta
 */
function testPrototypeChain() {

    function Parent() { this.delta = DELTA; };

    function ChildA(){};
    ChildA.prototype = new Parent();
    function ChildB(){}
    ChildB.prototype = new ChildA();
    function ChildC(){}
    ChildC.prototype = new ChildB();
    function ChildD(){};
    ChildD.prototype = new ChildC();
    function ChildE(){};
    ChildE.prototype = new ChildD();

    function nestedFn() {
        var child = new ChildE();
        var counter = 0;
        for(var i = 0; i < ITERATIONS; i++) {
            for(var j = 0; j < ITERATIONS; j++) {
                for(var k = 0; k < ITERATIONS; k++) {
                    counter += child.delta;
                }
            }
        }
        console.log('Final result: ' + counter);
    }
    nestedFn();
}
```

```
/**
 * Prototype chain with 1 children and property on parent object
 * ChildA->Parent->delta
 */
function testPrototype() {
    function Parent() {  };
    Parent.prototype.delta = DELTA;

    function ChildA(){};
    ChildA.prototype = new Parent();

    function nestedFn() {
        var child = new Parent();
        var counter = 0;
        for(var i = 0; i < ITERATIONS; i++) {
            for(var j = 0; j < ITERATIONS; j++) {
                for(var k = 0; k < ITERATIONS; k++) {
                    counter += child.delta;
                }
            }
        }
        console.log('Final result: ' + counter);
    }

    nestedFn();
}

/**
 * Prototype chain with 5 children and method to get property on parent object
 * ChildE->ChildD->ChildC->ChildB->ChildA->Parent->getDelta()
 */
function testPrototypeWithGetter() {
    function Parent() {  };
    Parent.prototype.delta = DELTA;
    Parent.prototype.getDelta = function() {
        return this.delta;
    };

    function ChildA(){};
    ChildA.prototype = new Parent();
    function ChildB(){}
    ChildB.prototype = new ChildA();
    function ChildC(){}
    ChildC.prototype = new ChildB();
    function ChildD(){};
    ChildD.prototype = new ChildC();
    function ChildE(){};
    ChildE.prototype = new ChildD();

    function nestedFn() {
        var child = new ChildE();
        var counter = 0;
        for(var i = 0; i < ITERATIONS; i++) {
            for(var j = 0; j < ITERATIONS; j++) {
                for(var k = 0; k < ITERATIONS; k++) {
                    counter += child.getDelta();
                }
            }
        }
        console.log('Final result: ' + counter);
    }

    nestedFn();
}
```

```
/**
 * Property available on target object without prototype chain
 * Parent->delta
 */
function testNonPrototype() {
    function Parent() { this.delta = DELTA; };
    function nestedFn() {
        var child = new Parent();
        var counter = 0;
        for(var i = 0; i < ITERATIONS; i++) {
            for(var j = 0; j < ITERATIONS; j++) {
                for(var k = 0; k < ITERATIONS; k++) {
                    counter += child.delta;
                }
            }
        }
        console.log('Final result: ' + counter);
    }

    nestedFn();
}

/**
 * Property available on target object, but value is retrieved from helper/util function (non OOP
style)
 * getDelta(object)
 */
function testFunction() {
    function getDelta(object) {
        return object.delta;
    }

    function nestedFn() {
        var child = {delta: DELTA};
        var counter = 0;
        for(var i = 0; i < ITERATIONS; i++) {
            for(var j = 0; j < ITERATIONS; j++) {
                for(var k = 0; k < ITERATIONS; k++) {
                    counter += getDelta(child);
                }
            }
        }
        console.log('Final result: ' + counter);
    }

    nestedFn();
}
```

```
/**
 * More complex prototype chain.
 * Parent object has object with prototype chain of 5 children, which contain target property.
 * Also parent object has 5 children.
 * So to access target object (Parent.sub.delta) following chain pass is required:
 * ChildE->ChildD->ChildC->ChildB->ChildA->Parent->SubChildE->SubChildD->SubChildC->SubChildB-
>SubChildA->SubParent->delta
 */
function testLongPrototypeChain() {
    function Parent() {  };
    Parent.prototype.sub = new SubChildE();
    function ChildA(){};
    ChildA.prototype = new Parent();
    function ChildB(){}
    ChildB.prototype = new ChildA();
    function ChildC(){}
    ChildC.prototype = new ChildB();
    function ChildD(){};
    ChildD.prototype = new ChildC();
    function ChildE(){};
    ChildE.prototype = new ChildD();
    function SubParent() { this.delta = DELTA; };
    function SubChildA(){};
    SubChildA.prototype = new SubParent();
    function SubChildB(){}
    SubChildB.prototype = new SubChildA();
    function SubChildC(){}
    SubChildC.prototype = new SubChildB();
    function SubChildD(){};
    SubChildD.prototype = new SubChildC();
    function SubChildE(){};
    SubChildE.prototype = new SubChildD();


    function nestedFn() {
        var child = new ChildE();
        var counter = 0;
        for(var i = 0; i < ITERATIONS; i++) {
            for(var j = 0; j < ITERATIONS; j++) {
                for(var k = 0; k < ITERATIONS; k++) {
                    counter += child.sub.delta;
                }
            }
        }
        console.log('Final result: ' + counter);
    }

    nestedFn();
}

/**
 * Test target value directly as local variable
 */
function testDirect() {
    function nestedFn() {
        var delta = DELTA;
        var counter = 0;
        for(var i = 0; i < ITERATIONS; i++) {
            for(var j = 0; j < ITERATIONS; j++) {
                for(var k = 0; k < ITERATIONS; k++) {
                    counter += delta;
                }
            }
        }
        console.log('Final result: ' + counter);
    }
    nestedFn();
}
```