Online Meta-Searcher

As described the main goal of this project is to probe pre-configured sources based on a configuration page and return results about a specific domain that involves smartphones. The result of our work includes a PHP Web page that handles all requests, a configuration page, a server management page and a cache page. These requests are handled by a java application and when done they are returned back to the web page. More about them later.

Java Application

The base components of the Java Application include the following:

- A Database Entity Class for our configurator.
- A Web Crawler. This class includes:
    - A product fetcher (as HTML).
    - A generic product parser (parses HTML and outputs a structured product).
    - A schema matcher.
    - A model for our product structure.

The final output of these are structured in a JSON format and returned back to the web page.

Anti-crawling techniques have been also implemented. A random user-agent switcher class hits web pages each time in order to prevent us from being blocked.

About the Generic Product Parser

This parser is based on JavaScript selectors and when parsing an HTML document, it has the ability to extract specific elements. Those elements exist in the configurator file as mentioned earlier.

An example.

If we have an HTML document that has the following structure:

```
<div id="product-listing">
      <ul>
            <li>
                  <span id="product-1">
                  <a href="www.mysite.com/product-1">Product 1</a>
                  </span>
            </li>
            <li>
                  <span id="product-2">
                  <a href="www.mysite.com/product-2">Product 2</a>
                  </span>
            </li>
            <li>
                  <span id="product-3">
                  <a href="www.mysite.com/product-3">Product 3</a>
                  </span>
            </li>
      </ul>
</div>
```

If we want to extract the product URLs we have to select these elements by using the following selector based on JavaScript:

```
$("#product-listing ul li span[id^='product-'] a");
```

Needed selectors are specified in the configuration file. If not familiar with this you have the ability to visit the "Help me" page that describes how selectors actually work. A different selector has been needed in order to extract pagination URLs.

About the Schema Matcher & Product Model

We have agreed that our product needs some attributes in order to be valid when requesting. If these attributes do not exist on the request, they are matched to any value. The product model that we went for has these attributes:

- A URL
- An ImageURL
- A Name
- A Manufacturer
- A Price
- Other Attributes

These other attributes include:

- Screen Size
- Ram Size
- Screen Resolution
- Storage Size
- Camera Resolution
- OS
- Battery Capacity
- Network Capability
- Weight

When no attribute matches any of the above it is considered as an extra attribute and is put in a map of variable size. So our schema considers these base and secondary attributes first and when not being able to recognize any other attribute it puts it in a generic attribute bucket.

When receiving an HTML product the Schema Matcher has to extract and find the attributes mentioned. A Schema Matcher class has been implemented for that job. Regexes are tested on labels and when found then they are returned.

An example regex matching:

```
public static final String screenResolutionValueRegex =
"(\\d+)\\s{0,1}(x|X)\\s{0,1}(\\d+)";

public static final String screenResolutionKeyRegex =
"(Οθόνη|οθονη|ανάλυση|αναλυση)";

private String findScreenResolution(Product product){
      List<String> possibleValues = findPossibleAttributesFirstMatch(product,
screenResolutionKeyPattern, screenResolutionValuePattern);

      if (possibleValues.isEmpty()){
            validations.add("Did not find Screen Resolution");
            return "";
      }
      if (possibleValues.size() > 1)
            validations.add("Found Multiple Screen Resolution");
```

```
        String result = possibleValues.get(0).replace(" ", "");

        result = Arrays
                        .asList(result.split("x"))
                        .stream()
                        .sorted(new ComparatorOfIntegerStrings())
                        .collect(Collectors.joining("x"));
        return result;
}
```

So one regex for label and one regex for value. If our HTML had a label named as “ανάλυση” then that would match our key regex.

IMPORTANT: The scope and domain of these regexes include only Greek products and the strings are fixed. These regexes could be further developed by adding machine learning techniques in the future.

Soft Entity Resolution & Filtering

There are two scenarios implemented in this project. The first one is a softer version when resolving entities. This scenario first downloads all the products, then filters each one before grouping and finally groups similar products based on name and fixed attributes.

About Product Filtering

When receiving and extracting the products the filtering takes place. If for example a request wants the product to have RAM Storage between 3 and 8 Gigabytes then any product not matching the criteria is not passed on for merging.

About Product Grouping

After Schema Matching and Filtering, the products need to be merged. A Master-Detail structure is created and the products are grouped. The rules based on grouping are:

- Name Matching: Based on Jaro Winkler on clean product titles.
- Attribute Matching: Based on the attributes mentioned above.

We preferred Jaro Winkler due to the small size of a clean product title. When “cleaning” the product title we actually remove attributes that exist on the product.

Take for example the following product title:

“Apple iPhone 7s 64GB 16MPixel”

We remove the “Apple”, the “64GB” and the “16MPixel” and assign them to attributes. So we have a structure as the following:

- Name: Apple iPhone 7s 64GB 16MPixel
- Clean Name: iphone 7s
- Manufacturer: Apple
- Storage: 64
- Camera: 16

Now we can match any other product with similar “clean name” and attributes such as:

“iPhone-7s from Apple with 64 Gigabytes”

Hard Entity Resolution & Filtering

This version (the one that we preferred) works out better when resolving products. After downloading all the products then the entity resolution takes place.

All products are grouped in sets by name (the name is cleaned out just like the first scenario). Then we inspect each group's products if they match the criteria of the query. If any product of the group does not fulfil the criteria then the whole group gets dropped out of the result set. Some products might miss some attributes but it is not a problem if any of the products in the set holds that attribute. If for example we have a set of 5 iPhones and only 1 of them has specified its weight then the whole set is considered valid.

Important to keep in mind that when an attribute does not exist at all in a set of products and a query has been placed, then the attribute missing eliminates the whole set. If for example we search for a product with 300 grams weight, and no product in the set includes a weight attribute then the set is dropped out.

Another modification to this grouping is that the "clean" name similarity has to maintain a high value (now it is set to 0.92) in order to not to throw away similar-like products. This results in more groups of products but with higher accuracy output when querying.

Web Application

Home

The web application includes a homepage that creates and passes on requests back to the server. All filters are available to a user before creating a request. This request is created and then passed on a handler PHP page that performs the AJAX call and renders the output result.

The filters include all Schema Attributes as mentioned earlier and are manipulated via sliders, input fields, checkboxes and select boxes. When we need to match "Any" values from a specific attribute we have to disable it via the top bar in order to not be sent as a request parameter.

After finished probing then the result is rendered by the helper PHP page as a Master-Detail output. All products are shown in a familiar way and we are able to view all details and all matches of the same product as well. A number next to the product title displays how many similar products were found. Additionally in the bottom a user can view how long his query took in detail, how many sources he probed and how many products were found.

Configuration

In this page we are able to specify sources and their needed selectors. These selectors are in fact used by the server when probing sources. They include:

- Order
- Name
- Category URL
- Encoding
- Pagination Selectors
- Product Listing Selectors
- Single Product Page Selectors

Order column is used only for source ordering at the moment but we now have the ability to further implement data fusion techniques based on this column. For example in our configuration "tokinito.gr" has a value of 9999 meaning that we do not trust this resource at all. Values are relative from lowest to highest (best to worst).

Name is just for source labeling, category URL is the actual URL we probe, and the rest are used in the rendering phase. We can append or alter a source according to our taste any time.

Cache

A user can check the cache option on homepage when querying. This action creates a row in our database including what he searched for, what the output was (as a link) and its metrics. If a request exists in the database cached then it is immediately returned (the key is the request URL parameter). When needed to perform a clean request a user has to flush the cached results and perform the query again.

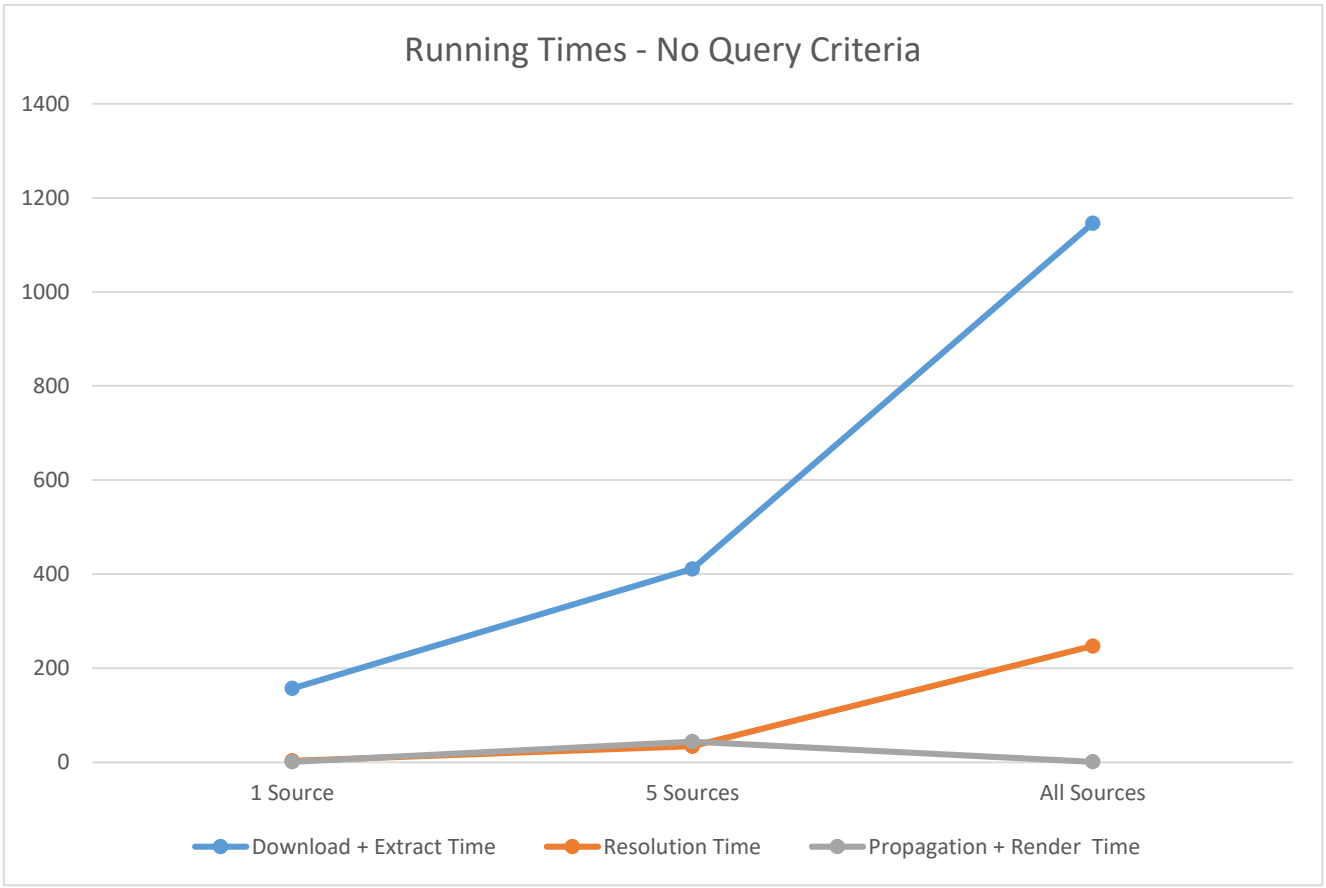
Other Pages

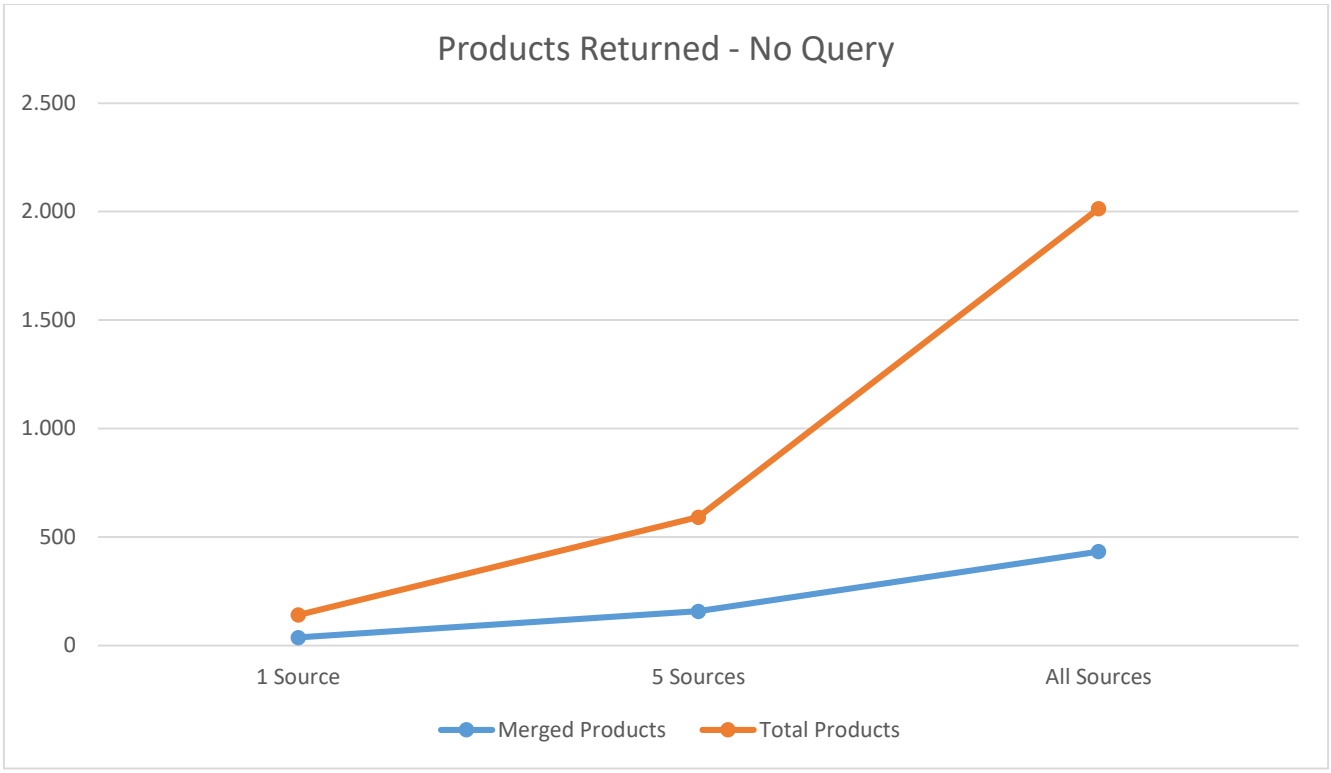
- **Manage Server:** A simple web page that has the ability to stop the server from running.
- **Help Me:** Instructions on how to use the selectors on the configuration file.
- **About:** The page that includes first and final deliverable documentation.
- **GitHub:** The GitHub page that includes the whole project.

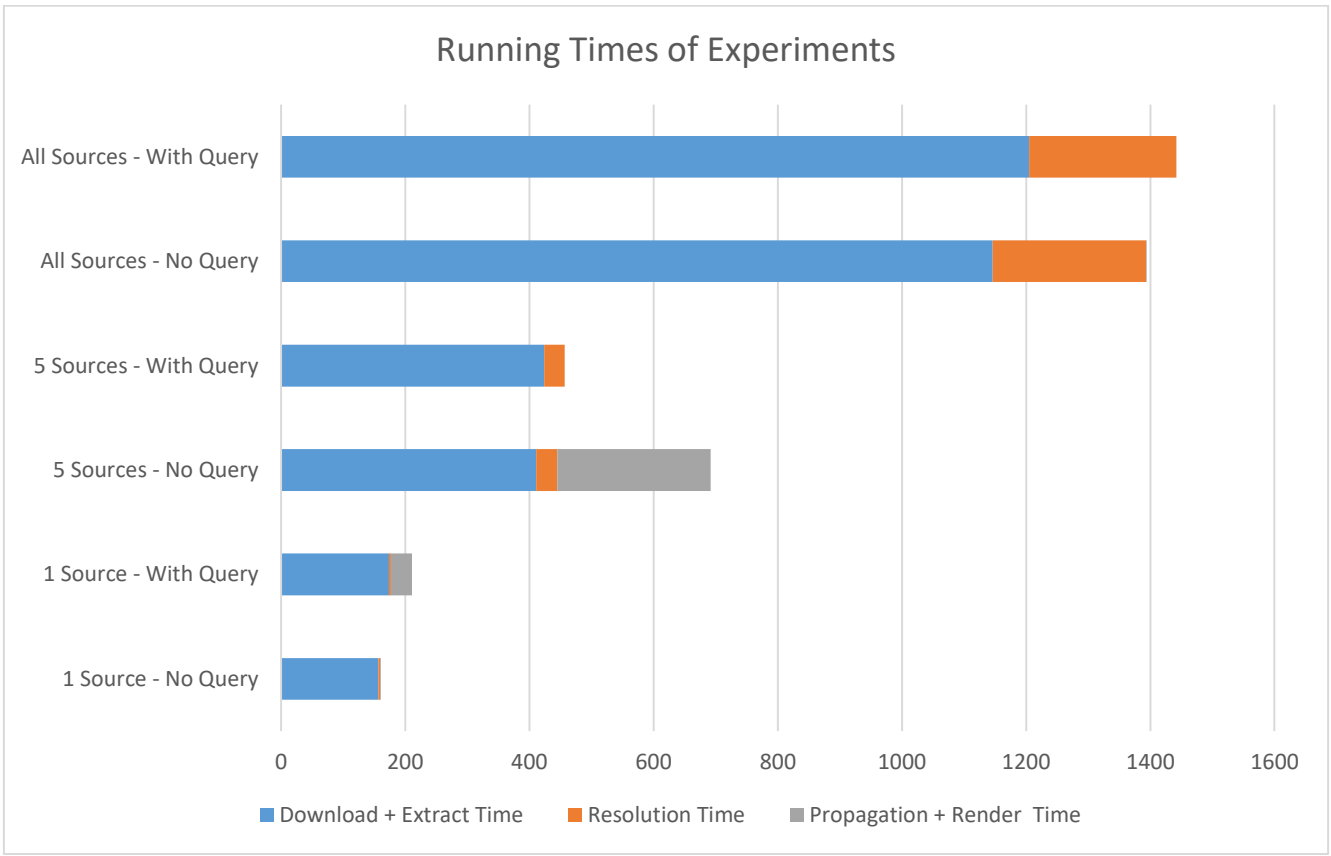
Experimental Results

The experiments following are repeated 5 times each (except the one that used all sources). The first one is probing one random source at a time. The second one probes 5 random sources each time. The last one probes all 15 sources (including bad sources). These sets of experiments are done twice, one with query filters and one without (match any product). Notice that there is a parallelization while downloading HTML documents that can be configured accordingly on the properties file - now it is set to simultaneously download 3 documents per hit. Also the computer downloads on average at 6.5Mbps.

Output




Products Returned - No Query
2.500
2.000
1.500
1.000
500
0
1 Source
5 Sources
All Sources
Merged Products
Total Products


Running Times of Experiments
All Sources - With Query
All Sources - No Query
5 Sources - With Query
5 Sources - No Query
1 Source - With Query
1 Source - No Query
0
200
400
600
800
1000
1200
1400
1600
Download + Extract Time
Resolution Time
Propagation + Render Time

Detailed Results

No filter - 1 source - 5 Experiments

Exp	Total Time (sec)	Server Exec (sec)	Download Time (sec)	Resolution Time (sec)	Probed Sources	Merged Products	Total Products	Avg Per Product (sec)
1	35.184	35.157	35.141	0.016	1 (6)	9	14	**2.513**
2	391.041	390.939	376.909	14.03	1 (1)	95	467	**0.837**
3	170.064	170.024	169.578	0.446	1 (20)	26	86	**1.977**
4	86.253	86.223	85.936	0.287	1 (15)	19	75	**1.150**
5	122.883	122.515	122.098	0.417	1 (4)	34	62	**1.981**
Avg	**161.085**	**160.972**	**157.932**	**3.0392**	**1**	**37**	**141**	**1.691**

No filter - 5 sources - 5 Experiments

Exp	Total Time	Server Exec	Download Time	Resolution Time	Probed Sources	Merged Products	Total Products	Avg Per Product (sec)
1	316.845	316.729	304.098	12.631	5 (5,23,16,19,21)	139	459	**0.690**
2	1246.828	1246.614	1114.303	132.311	5 (1,2,5,19,24)	355	1445	**0.862**
3	328.345	328.261	309.972	18.289	5 (5,15,16,17,21)	150	522	**0.629**
4	220.66	220.605	212.524	8.081	5 (6,7,23,15,17)	81	376	**0.586**
5	117.442	117.385	115.779	1.606	5 (4,11,23,20,21)	64	157	**0.748**
Avg	**401.949**	**445.919**	**411.335**	**34.584**	**5**	**158**	**592**	**0.703**

No filter - All sources - 3 Experiments

Exp	Total Time	Server Exec	Download Time	Resolution Time	Probed Sources	Merged Products	Total Products	Avg Per Product (sec)
1	1361.746	1361.517	1124.448	237.069	15	432	2012	**0.676**
2	1450.506	1449.844	1197.404	252.44	15	433	2016	**0.719**
3	1368.512	1368.287	1116.71	251.577	15	433	2015	**0.679**
Avg	**1393.588**	**1393.216**	**1146.187**	**247.028**	**15**	**433**	**2014**	0.691

Now the experiments with filters. The query used is:

- **Search String:** iphone
- **Screen Size:** 3 - 11 Inches
- **RAM:** 1 - 16 GB
- **Camera:** 2 - 24 MPixels
- **Storage:** 2 - 128 GB
- **Weight:** 50 - 350 Grams
- **Battery Capacity:** 1000 - 6000 mAh
- **Android OS:** Yes
- **Apple OS:** Yes
- **Windows OS:** Yes
- **Other OS:** Yes
- **Price:** 0 - 5000 EUR
- **Company:** Null
- **Network Capability:** Null
- **Screen Resolution:** Null

With filters - 1 source - 5 Experiments

Exp	Total Time	Server Exec	Download Time	Resolution Time	Probed Sources	Merged Products	Total Products	Avg Per Product (sec)
1	145.937	145.887	142.766	3.121	1 (2)	4	76	**1.920**
2	409.409	409.372	395.734	13.638	1 (1)	1	62	**6.603**
3	172.93	172.907	172.892	0.015	1 (6)	0	0	**0 ***
4	112.047	112.021	111.748	0.273	1 (4)	1	12	**9.337**
5	47.949	47.927	47.582	0.345	1 (19)	0	0	**0 ***
Avg	**143.121**	**177.623**	**174.144**	**3.478**	**1**	**1**	**30**	**5.95**

With filters - 5 sources - 5 Experiments

Exp	Total Time	Server Exec	Download Time	Resolution Time	Probed Sources	Merged Products	Total Products	Avg Per Product (sec)
1	412.398	412.331	384.884	27.447	5 (1,23,16,17,19)	1	124	**3.325**
2	1284.288	1284.224	1161.61	122.614	5 (1,2,5,19,24)	1	285	**4.506**
3	326.932	326.884	309.483	17.401	5 (5,15,16,17,21)	1	127	**2.574**
4	158.813	158.768	150.808	7.96	5 (6,7,23,15,17)	0	0	**0 ***
5	117.227	117.171	115.652	1.519	5 (4,11,23,20,21)	2	41	**2.859**
Avg	**459.932**	**459.876**	**424.487**	**33.817**	**5**	**1**	**115**	**3.316**

With filters - All sources - 3 Experiments

Exp	Total Time	Server Exec	Download Time	Resolution Time	Probed Sources	Merged Products	Total Products	Avg Per Product (sec)
1	1531.563	1531.465	1288.138	243.327	15	2	427	**3.586**
2	1328.743	1328.243	1089.35	238.893	15	2	427	**3.111**
3	1470.993	1470.877	1237.692	233.185	15	2	427	**3.444**
Avg	**1443.766**	**1443.528**	**1205.06**	**238.468**	**15**	**2**	**427**	**3.380**

*** These are not considered on the total average per product.**

Notices:

- The total time almost equals the download time.
- Resolution times start from 1.86% of the total execution time and reach up to 17.73% (all sources unfiltered). When resolving each product name and attributes have to be compared with the rest.
- Average product times are around 0.7 sec when no criteria are posed. They can reach up to 4.2 sec per product when criteria exist.
- The more sources the merrier the products and groups - obviously.
- The more sources the more time needed for results. Although average product times are stable when filtered or unfiltered.
- Incomplete sources need to be combined with at least one complete source in order to match any criteria.
- The results were pretty accurate given the fact that the sources are well-structured and most of them include every attribute we needed for our schema. Of course when adding a new source that has a totally different label we will need to add a new regex, but the domain is limited enough and the sources are accurate - obviously if the owners of the e-shop want to sell their products ☺