Think before you Act! Minimising Action Execution in Wrappers*

Tim Furche, Giovanni Grasso, Christian Schallhart, Andrew Sellers
Dep. of Computer Science, Oxford University
Parks Road, Oxford OX1 3QD
firstname.lastname@cs.ox.ac.uk

Antonino Rullo
DEIS Department, Universitá della Calabria
Via P. Bucci, 41/C, 87036 Rende, Italy
nrullo@deis.unical.it

ABSTRACT

Web wrappers access databases hidden in the deep web by first interacting with web sites by, e.g., filling forms or clicking buttons, to extract the relevant data from the thus unearthed result pages. Though the (semi-)automatic induction and maintenance of such wrappers has been extensively studied, the efficient execution and optimization of wrappers has seen far less attention.

We demonstrate that static and adaptive optimisation techniques, as used for query languages, significantly improve the wrapper execution performance. At the same time, we highlight difference between wrapper optimisation and common query optimisation for databases: **(1)** The runtime of wrappers is entirely dominated by page loads, while other operations (such as querying DOMs) have almost no impact, requiring a new cost model to guide the optimisation. **(2)** While adaptive query planning is otherwise often considered inessential, wrappers need to be optimised during runtime, since crucial information on the structure of the visited pages becomes only accessible at runtime. We introduce two basic, but highly effective optimisation techniques, one static, one adaptive, and show that they can easily cut wrapper evaluation time by one order of magnitude. We demonstrate our approach with wrappers specified in OXPath.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: On-line Information Services—*Web-based services*

General Terms

Languages, Experimentation

Keywords

data extraction, query optimization, XPath, OXPath

*The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement DIADEM, no. 246858.

VLDS'12 August 31, 2012. Istanbul, Turkey.

1. INTRODUCTION

Facing ever growing amounts of information, we are surrounded by new, exciting opportunities – which we can only realize by automatizing web data extraction further. But next to reducing the amount of human assistance in wrapper generation, large-scale extraction tasks also require efficient wrapper execution, particularly by avoiding unnecessary page loads. To achieve this goal, we have to statically optimize our wrappers, and even more importantly, we need to apply some crucial adaptive optimizations during runtime to exploit information on the underlying page template: For example, after learning that following the "contact" link in a given page template never leads to pages relevant to the extraction task at hand, we might want to filter the "contact" link.

In response, we propose a framework for static and dynamic wrappers optimization. We base our optimization on a model, where wrappers alternate local selection and extraction steps with action steps, executed on DOM nodes, such as clicking on links. As the performance of a wrapper is overwhelmingly dominated by the number of pages loads, our optimizations aim to **(1)** eliminate irrelevant actions, **(2)** reduce the number of nodes to perform an action on, and to **(3)** delay actions in the hope that they will not be executed due to some predicate failure beforehand. Partly, we achieve these goals with *static optimizations,* e.g., by removing actions which do not lead to any extracted data. Therein, we can only rely on common assumptions on HTML pages, resulting in *HTML-optimal wrappers.* Contrariwise, *dynamic optimizations* are performed during wrapper execution, to exploit the commonalities in the observed action applications, e.g., if an action never yields relevant results, we stop evaluating it. Hence, dynamic optimizations exploit the construed template structure of the observed pages, yielding *template-optimal wrappers.*

We evaluate our approach with OXPath [4], a wrapper language which extends XPath with actions, Kleene stars, extraction facilities, and access to the dynamic DOM of the pages under scrutiny, including all computed styles (briefly introduced in Section 3).

Contributions. **(1)** Our *optimization model* describes wrappers with their execution costs and defines HTML- and template-optimality (Section 2). We start with a number of **(2)** *static optimizations* to eliminate unnecessary actions entirely, or – if unavoidable – evaluate them as seldom as possible (Sections 4). Subsequently, we present our **(3)** *dynamic optimizations* to reduce the number of DOM nodes to perform actions upon and to memoize extraction results (Sections 5). Finally, we **(4)** *apply* and **(5)** *evaluate* these optimizations for OXPath (Section 6). After the related work in Section 7, we conclude the paper.

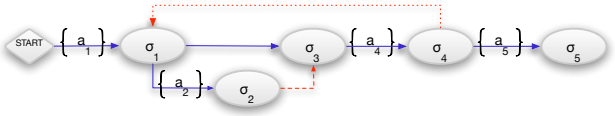


Figure 1: Wrapper plan

2. HTML- & TEMPLATE-OPTIMALITY

Wrappers extract data from many web pages and typically run several minutes or hours rather than seconds as traditional database queries. Almost all existing wrapper languages execute wrappers as a sequence of *selection and extraction operations* partitioned by *action operations*. Selection operations determine which DOM elements extraction operations or actions are performed on. Extraction operations specify how to extract those nodes, and action operations specify browser actions such as submitting a form or clicking a link. Together we refer to all three as *wrapper operations*. We say that action operations *partition* wrapper executions, as all selection and extraction operations between two such actions involve the same DOM (resulting from the preceding action). In case of an, e.g., OXPath wrapper, sequences of XPath steps (selection operations) and extraction markers (extraction operations) alternate with action steps.

DEFINITION 1. *A **wrapper plan** is a (possibly cyclic) graph with selection and extraction operations as node labels and action operations as edge labels, all possibly empty. Aside normal edges, we distinguish two edges types:* Branch edges *indicate where results from one branch are used to filter another branch.* Cycle edges *indicate cycles in the evaluation plan. If we eliminate all cycle edges, a wrapper plan is a directed, acyclic graph with a dedicated start node.*

Figure 1 shows an example plan with actions a_i and selection and extraction operations σ_i. It contains a cycle, from σ_1 to σ_4, which indicates that these expressions are executed as long as they match. Typically such cycles represent repeated actions, such as clicking on "next" links to traverse all paginated results obtained after some query. It also contains a branch at σ_1 to indicate that the wrapper explores the web site in two different ways starting from the page obtained by $\{a_1\}$. The branch specifies an additional restriction on the extracted nodes, indicated by the red, dashed link rejoining the main branch, and thus acts like predicates in XPath or `where` clauses in XQuery. Figure 3 shows branches used to filter another branch.

We call a sequence S of visited pages the *page footprint* of a wrapper plan W (or one of its connected subgraphs), if executing W touches the pages in S in order. A wrapper plan is *sequential* if no node has more than one incoming branch edge. We denote with $\text{SUCC}(o)$ the *successor* to an operation o following only normal edges, disregarding branch or cycle edges.

For the purpose of this paper, we model the *cost of a wrapper* with a rough, static cost model, while our results are easily adjusted to richer cost models. We assume that we can estimate selectivity and cost of each operation o, denoted with $\text{SELECTIVITY}(o)$ and $\text{COST}(o)$. Then the cost of a wrapper W is:

$$\text{COST}(W) = \sum_{o \in W} s(o) \times \text{COST}(o)$$

where

$$s(o) = s(o') \times \text{SELECTIVITY}(o) \times \text{STRUCT}(o) \text{ with o=SUCC(o')}$$

$$\text{STRUCT}(o) = \prod_{\text{incoming edge } e} \begin{cases} \kappa & \text{if } e \text{ is a cycle edge} \\ \pi & \text{if } e \text{ is a branch edge} \\ 1 & \text{otherwise} \end{cases}$$

As the focus of this paper is not a sophisticated cost model, we approximate the impact of cycles and branches roughly through fixed multiplicative factors κ and π (typically, $\kappa = 3$ and $\pi = 0.5$).

DEFINITION 2. *Let W be a wrapper plan. A sequential wrapper plan O is **HTML-optimal** for W, if O produces the same output as W on all HTML pages and has minimal cost among such plans. Let C be a cycle in W and $\mathscr{P}$ a set of page footprints of C. Then O is **template-optimal to** W **under** C **and** $\mathscr{P}$ if W and O produce the same output on all page footprints in $\mathscr{P}$ while O has minimal cost among all such sequences.*

In the following, we focus on a particular aspect of optimizing wrapper plans: As shown in [4] and further substantiated in Section 6, actions dominate the execution of wrappers, as they lead to page loading necessitating network communication and page rendering. Thus, we focus on optimizations that reduce the number of actions to be performed. Corresponding to the two types of optimality introduced above, we introduce static optimizations, based on HTML schema and usage statistics, as well as dynamic optimizations, adapting a wrapper's execution according to observations on already visited pages.

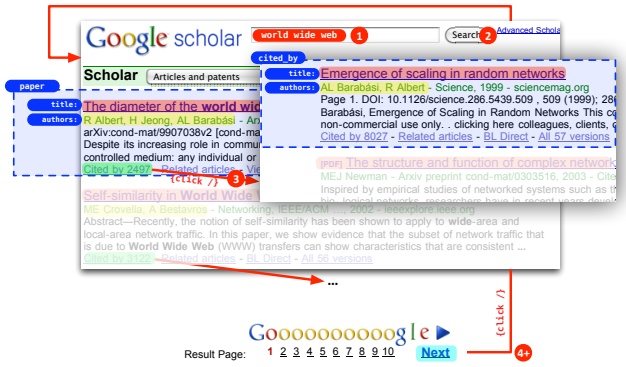
(1) Static schema-based optimization (Section 4): **(a)** REDUCE ACTION CONTEXT. The first optimization focuses on reducing the size of the input set for an action, e.g., by reordering an execution plan to perform branches without actions first. Since such a branch may fail, the far more expensive action branch is executed for a reduced context set. **(b)** ELIMINATE ACTIONS. If an action is followed by operations that never yield results, the action can be dropped ("dead action"). If an action – as part of a filter – is followed by operations that always yield results, it can also be eliminated as the filter has no affect ("tautological action").

(2) Dynamic template-based optimization (Section 5): Many sites contain very similar pages, as those pages are often generated from a commonly shared template. For those pages, we can then **(a)** REPLAN by replacing paths with action-free paths if they are equivalent on the given set of pages. **(b)** RECALL the results of entire sub-plans, if the same context node is reached from different pages (e.g., through links that point to the same fixed page).

3. OXPATH

Suppose you are looking for an apartment in the city center, not larger than 70 m^2, newly refurbished, with balcony and garage, preferably close to a bus stop and a school for your children. All this information is readily available on the web, but even modern aggregators will fail to **(1)** cover the entire market, and to **(2)** interlink the offered properties with background knowledge, such as schools. Although many extraction languages exist, they either fail to support interactions with modern scripted sites, or do not scale well enough to drive large extraction tasks. Hence, in solving such problems, we often have no choice but to manually extract, aggregate, and rank that information – a tedious and often unmanageable task due to the size of the relevant data.

To overcome these drawbacks, we designed OXPath, an extension of XPath, which facilitates interactions with modern web applications for the sake of data extraction: **(1)** *Actions* perform user




```
doc("scholar.google.com")/descendant::field()[1]/{"world..."}❶
    /following::field()[1]/{click/}❷
    /(//a[contains(string(.),'Next')]/{click/})*❹
    //div.gs_r:<paper>[.//h3:<title=string(.)>]
      [.//*.gs_a:<authors=substring-before(.,' - ')>]
      [.//a[.#='Cited by']/{click /}❸
       //div.gs_r:<cited_by>[.//h3:<title=.>]
          [.//*.gs_a:<authors=substring-before(.,' - ')>]]
```

Figure 2: Finding an OXPath through Google Scholar

interactions, such as clicking on a link or hoovering with a mouse over some element, to reach all data supplied by modern scripted sites. Actions are either *absolute* (with an added `/`), where navigation continues at the root of the page retrieved by the action, or *contextual*, in which navigation continues from the element the action is performed upon. **(2)** *Dynamic DOM access* enable us to wrap modern web applications, as their scripts modify the pages at hand continuously. Accessing the dynamic DOM, OXPath wrappers can exploit, e.g., neighborhood relations or colors. **(3)** *Extraction markers* designate the data to be extracted and organize this data, e.g., as XML or RDF document. **(4)** *Kleene star expressions* enable iterations, e.g., following all next links to reach all entries of a paginated result list. Aside a balanced set of language features, we designed OXPath to be easily embedded into host languages such as Java or JavaScript, and to be highly parallelizable. We proved that OXPath satisfies optimal bounds on the number of page buffers, while simultaneously minimizing the number of visited pages.

Figure 2 showcases all OXPath extensions mentioned above, marking all actions **(1)** with red encircled numbers: Lines 1-2 fill and submit the search form, wherein `field()` selects only visible **(2)** input elements to identify the text field and search button. Line 3 realizes the iteration over the set of result pages by repeatedly clicking the "Next" link **(4)**. Lines 4-5 extract a result record and its author and title **(3)**, Lines 6-8 navigate to the cited-by page and extract the papers. The expression yields nested records as in the following.

```
<paper><title>The diameter of the world..</title>
 <authors>R Albert, H Jeong, ...</authors>
 <cited_by>
    <title>Emergence of scaling in ... </title>
    <authors>AL Barabasi...</authors></cited_by>
 <cited_by> .....</paper>
```

For details on OXPath, cf. [4].

Wrapper plans for OXPath. To build a wrapper plan from an OXPath expression E, we segment E into actions and action-free chunks. Then, we form an automaton with actions on edges, and action-free chunks on nodes, to obtain a wrapper plan, meeting Definition 1. For example, consider the query

```
doc('url')//div[a/{click/}//div[a/{click/}//h1~='Person')]
                                  [@class='woman']]
  [@class='person'][h1[1]:<name>]
  [@title='phoneNumber']
```

which returns all `div` elements that **(1)** have at least one anchor element `a` pointing to a page with at least one `div` element, which must **(a)** contain at least one further `a` child pointing a page with an `h1` element containing the string `'Person'`, and **(b)** contain a `class` attribute with `value='woman'`. Each original `div` must also **(2)** feature a `class` attribute with `value='person'`, **(3)** have at least one `h1` child. On the first of these, we apply the extraction marker `<name>`, and finally, the original `div` must **(4)** have a `title` attribute with `value='phoneNumber'`.

In this expression, we have a number of predicates, with two predicates nested inside the first one, as translated directly into the wrapper plan, shown in Figure 3. Note the dashed red branch edges indicating that their results are used to filter the results on the main branch. The order of applying these predicates corresponds to the order in which the branch edges rejoin their main branch. For optimizing OXPath expressions, we have to describe the selectivity $\text{SELECTIVITY}(o)$ and $\text{COST}(o)$ of individual OXPath operations o. For both, we rely on standard estimates for XPath and HTML.

4. STATIC: ACTIONS LAST

In this section, we focus on static optimization striving for HTML-optimality, i.e., optimizations which are valid on all HTML documents (Definition 2). To that end, we **(1)** *reorder* branches (i.e., predicates in the XPath sense) to perform selective predicates without actions before those with actions, and we **(2)** *eliminate actions* in case of dead and tautological actions. For space reasons, we focus on the first case, as dead and tautological action elimination can be addressed with standard XPath containment [10] for OXPath.

Let W be a sequential wrapper plan with a sequence S of nodes each with incoming branch edges, but connected into a path through normal edges. We further assume a function $\text{FIXED}(p)$ that determines if p may be commuted with other branches. In OXPath, $\text{FIXED}(p)$ holds only for branches containing extraction markers or positional functions (i.e., `position()` or `last()`). All other branches can be commuted as necessary. Then, $S = \vec{p}_0 q_1 \vec{p}_1 q_2 \dots q_n \vec{p}_n$ where $q_1, \dots, q_n$ are fixed branches in S and $\vec{p}_i$ a sequence of commutable ones. We sort each $\vec{p}_i$ by cost. In most cases, this means moving predicates without actions first, predicate with actions last as the cost of action execution dominates the overall wrapper cost.

Recall the example from Section 3 with the wrapper plan shown in Figure 3. The following wrapper is obtained by reordering the predicates, both in the outer case and in the inner one.

```
doc('url')//div[@class='person']
              [a/{click/}//div[@class='woman']
                              [a/{click/}//h1~='Person')]]
              [h1[1]:<name>]
              [@title='phoneNumber']
```

It is not possible to move the last two predicates before the one with the actions, as the one starting with `h1` contains an extraction marker and thus partitions the set of predicates.

Our cost model with $\pi = 0.5$ suggests that the additional predicates, applied before the action execution, lead to 50% less action executions each, thus leading to only 25% of the second click actions compared to the naive case. In this example, however, the reduction is actually even more significant, since the reordered predicates preclude almost all matching links from being followed.

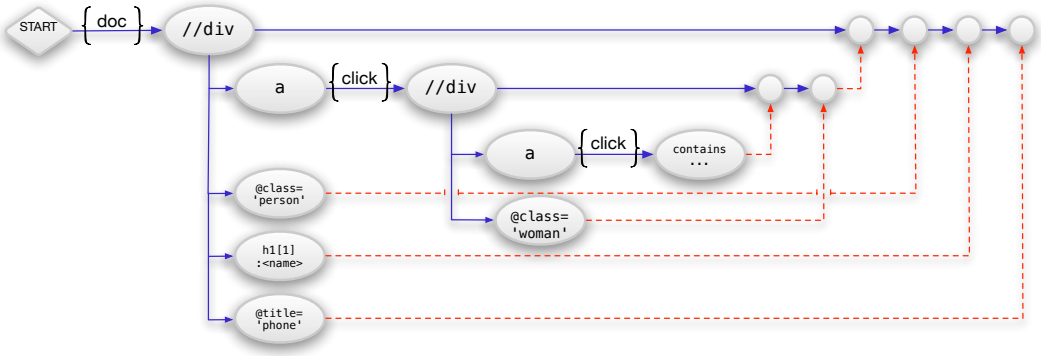


Figure 3: Example wrapper plan

5. ADAPTIVE: RE-PLAN AND RECALL

Wrapping template-based websites usually involves dynamically generated pages, which are similar in structure but different in content. Also, different pages link to a certain number of common pages, e.g., "contact us" info or details pages for products. Therefore, for long running wrappers, e.g., for the exhaustive extraction from a products website, it is possible to **(1)** *collect statistics* at runtime to **replan** successive (sub-)parts of the evaluation that belong to the same template, and **(2) recall** previously stored results when sub-expressions repeatedly access the same pages.

For **(1)** we focus here on replanning expressions in Kleene stars as we are likely to encounter many pages on which to evaluate this expression and those pages are likely to be part of the same template. The aim is to find an expression for the selection in the Kleene-star that involves less actions, yet is equivalent on these template pages. E.g., the wrapper may want to extract all products with an image on their details page. Unfortunately, that requires navigation to the details page. However, on many product web sites this may be already recognisable on the result page, e.g., as all products with a certain class or with a placeholder image (an image of a certain URL). The wrapper author or induction system may nevertheless prefer the prior expression as it expresses the users intent and is likely to be more robust (class and IDs often change in template pages due to redesign or obfuscation). However, at runtime we can verify that the more specific wrapper yields the same results, yet performs significantly less actions.

For **(2)** we focus on tabling page URIs (and parameters) and reuse the tabled entries if the same page is accessed again (in the context of the same expression). This avoids all evaluation starting on that page. We, furthermore, add a number of heuristics that allow us to use the tabled result without even having to go to the tabled page.

Re-plan by Adaptive Rewriting. Let W be a sequential wrapper plan with a cycle C, where $\mathscr{P}$ is a footprint of C during an evaluation of W, such that $\mathscr{P}$ forms a template cluster [7], i.e., a set of pages likely from the same template. We have $C = \sigma_0\{a_1\}a_1 \ldots \{a_n\}\sigma_n$ where each σ_i is an action-free expression (possibly involving branches) and each a_i is an action. Then we build a new sequential wrapper W' as optimisation of W by replacing σ_i with σ_i', such that σ_i' is more selective than σ_i, yet $\sigma_0'\{a_1\}\ldots\{a_n\}\sigma_n'$ is template-equivalent to C under $\mathscr{P}$. Hence, W' is template-equivalent to W under $\mathscr{P}$ and has lower cost (in particular it executes less actions).

How do we obtain σ_i'? Unfortunately, both inferring the regular structure (or DTD) for $\mathscr{P}$ [1] and the optimization of wrappers in presence of such a schema are computationally highly expensive tasks, provably intractable for rich wrapper languages going well beyond navigational XPath [10]. Since we are adapting the wrapper plan during execution, such complete optimization approaches are infeasible, and hence, we choose a set of simple local heuristics, instead. The local heuristics generate alternative expressions for selecting the final node set selected by each σ_i using attributes such as `id` and `class`, the direct children or parents, the content etc. We use these expression generators also to determine if $\mathscr{P}$ is a template cluster (rather than more complete methods such as [7]): If we can find succinct expressions for most σ_i there is a good likelihood that the pages come from the same template.

To illustrate the process, consider an example from the real-estate domain, where we want to extract the URLs to floor plans. Unfortunately, many real estate websites, particularly aggregators, present floor plans mixed with photos of the property. Therefore, a user, tasked with generating a wrapper to extract floor plans, may create the following wrapper: After clicking on all image links, it checks on the resulting page for a floor plan (by considering the alt text not available on the first page).

```
//li//div//a/img/{click/}//img[@alt ~= "floor
    plan"]:<A=string(@src)>
```

It is part of a wrapper that cycles on all properties of the website at hand. Properties on a page are located by `//li`, while their respective picture sections are grouped in a descendant `div` as image links (`a` elements with contained images). These links point to pages with the actual image in full resolution. We identify it using it's `alt` text (i.e., tooltip) that contains the phrase "floor plan".

This is an easy to understand and probably rather robust wrapper, however, it requires one action (and thus page load) for every image. When executing this wrapper, the system finds that in all the pages touched by the cycle, the floor plan images can actually identified already on the initial page: They occur only in specific `li`'s that represent offers and have `data-ajax` attributes. Although together this forms a very specific selection expression it is also a very fragile expression (as, e.g., the IDs are often auto-generated and the `data-ajax` attributes are an artefact of using a specific generation library). Thus neither human wrapper authors nor wrapper induction systems are likely to generate these expressions. At runtime, however, we do not care about robustness across different runs and thus use the more fragile, but less expensive selection:

```
//li[@id ~=
    'listing_']//a/img[@data-ajax]/{click/}//img[@alt~=
    "floor plan"]:<A=string(@src)>
```

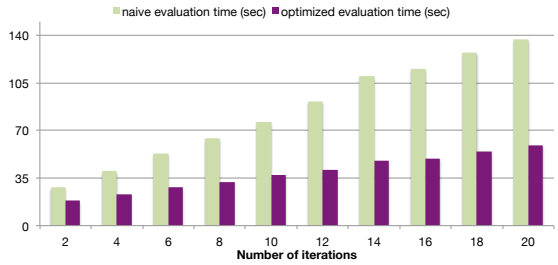


Figure 4: Evaluation of Re-plan Optimization

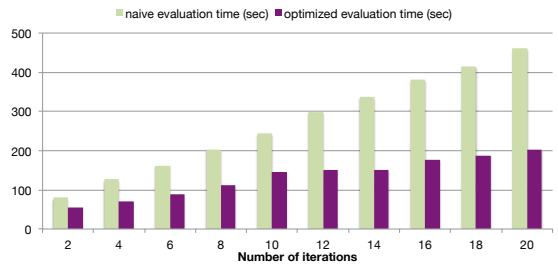


Figure 5: Evaluation of Recall Optimization

With the new wrapper, we only execute one action per floor plan (rather than one per image). The new wrapper is significantly cheaper according to the cost model in Section 2, which estimates that only 2π of the context nodes in the original wrapper will also be context nodes for `{click/}` in the new wrapper, i.e., a reduction by 75%. For the actual web site (`zoopla.co.uk`) the reduction is actually over 90%.

Recall by Action Tabling. For the second optimization, we recall the results of expressions already evaluated on pages touched repeatedly by a wrapper plan. Let $E = \sigma_1 a_1 \rho$ be a connected subgraph of a wrapper plan W where σ_1 is some action-free expression, a_1 an action, and ρ the rest of a branch in the wrapper plan. For recall, we maintain a table that maps action expressions, such as a_1, together with URLs of the resulting pages to the results of evaluating the rest of the wrapper branch (here: ρ) on those pages. An entry is only created if an action yields a new page (rather than modifications to an existing one).

Unfortunately, this still requires that the action is performed each time, though all actions in ρ are skipped. To avoid the initial action, we use a simple heuristic for recognizing nodes on which an action for sure leads to a new page and we can determine that page from the node. For space reasons, we only give a sample of these heuristics: **(1)** For *simple links*, i.e., `a`'s with an `href` attribute, but no event handlers, we table the value of the `href` (properly resolved in case of relative URIs). **(2)** For *simple forms*, i.e., forms with an `action` URL and no event handlers, we also table the value of the `action` attribute.

To illustrate this, consider a wrapper on `rightmove.co.uk` that aims to extract properties together with contact information about the agent. Unfortunately, those contact information are available only from the "Contact us" page, which remains the same for the same agent. Thus, in the following wrapper, we table the `href` attributes of the `.contactUs` links and recall of extracted agent information, if the same `href` is encountered again.

```
//a.contactUs/{click/}//div[1]:<agent>
  [.//li.agency:<name=string(.)>][.//li.addr:<addr=string(.)>]
```

Rather than one action for each of the over 650,000 properties, we thus only perform one for each of the about 10,000 agents.

6. EVALUATION

We show the effectiveness of the presented optimization techniques, using OXPath as wrapping language.

Static. In this experiment on `rightmove.co.uk`, the largest aggregator for the real estate in UK, we extract the addresses of properties offered by agents supplying more than 100 sale offers. To this end, we apply the following expression on each result page:

```
//div[.//div.branchlogo/a/img/{click/}
        //img[@title='Properties for sale']/{click/}
        //div#resultcount[number(.)>100]]❶
  [@class='summarymaincontent']❷
  //span.displayaddress:<addr=string(.)>
```

This expression visits all `div` elements, and – in the assumption that these `div`-s represent indeed offers – attempts to navigate to the agent's logo next to each property, click it, click another image to reach all the agent's offers to finally check the number of these offers. Next, the expression also checks that the `div` has `class 'summarymaincontent'`, before extracting the relevant address.

We apply static optimizations (Section 4) to reorder adjacent predicates to minimize the impact of actions. Since predicate ❶ contains actions while ❷ is action free, our static optimization produces an equivalent expression by reordering them such that ❷ precedes ❶. Hence, the wrapper avoids clicking on logos which are not part of property summaries – avoiding many unnecessary page loads. Also, if the template changes, dropping or changing the `'summarymaincontent'` class, thus spoiling the wrapper, we avoid loading pages without extracting any data at the end.

The evaluation of the original expression takes on a single result page 60 seconds in average, while our optimized expression terminates in 2 seconds on average.

Re-plan. `spareroom.co.uk/flatshare/oxford` lists rooms for rent, that we want to extract as records `R`, each described with a `div#summary` node on the page. To this end, we run several times the following wrapper Q_x, parametrized with x, an upper iteration bound we use to scale the impact of adaptive rewriting.

```
(//a#next/{click/})*{0,x}//div#summary:<R>
[.//div[last()]/a/{click/}//h1.about:<contactUs=string(.)>]
```

Among other attributes (omitted for brevity), we also extract contact information visiting the website's about page by clicking on the corresponding link. This link is located at the bottom of the listed rooms (last `div`), searched for via `//div[last()]/a`. As a result, we select too many links and need to identify the targeted data by matching `h1` elements on the visited pages afterwards. This is a common situation, especially with wrappers generated by some (un-)supervised tool.

As shown in Figure 4, a naive evaluation is very wasteful in time due to the number of unnecessary actions. Indeed, after observing a number of pages generated from the same template, our wrapper is dynamically optimized in considering only those links which lead to successful data extraction. In our case, we re-plan by replacing `//div[last()]/a` with `//div[last()]/a[position()=1 | @name]`, i.e., selecting only links in first position or with attribute `name`.

Recall. For this experiment, we run the following expression Q_x, again on the UK real-estate aggregator `rightmove.co.uk`.

```
//(a[contains(string(.),'next')]/{click/})*{0,x}
//div.clearfix:<R>//div.branchlogo/img/{click/}
[//div#branchdetails/img/@alt:<agent=string(.)>]
[/following-sibling::div.phone[1]:<num=string(.)>]
```

We extract a record `R` for each listed property, including the agent's name and phone number. To reach this data, we first `click` on the agent's logo in each property `//div.branchlogo/img` to open a further page where we extract the relevant data. Throughout evaluation, the same contact data is likely to be extracted multiple times. As described in Section 5, tabling is applicable to sub-expressions preceding actions. In this case, we retain the result of evaluating `//div.branchlogo/img` together with the image's `src` attribute value. Thus, once retained, we reuse the same data instead of re-executing the necessary but now redundant actions. Figure 5 compares the time of naive versus optimized evaluation with an increasing number of loops. While the naive evaluation is basically linear in the number of loops, the optimized evaluation varies irregularly, as it depends on the hit rate of the ongoing tabling. But in any case, it takes significantly less time, with a more pronounced effect at greater iteration numbers, cutting the evaluation time more than half at 20 iterations.

7. RELATED WORK

Though web wrapper induction and maintenance is an extensively studied problem, this is not the case for wrapper optimization. In most cases, optimization is performed as an integral part of wrapper induction [14, 8], however, optimizing more for coverage and precision of the wrapper than for efficiency. In [3], e.g., candidate wrapping rules are filtered based on the expected schema of the data to be extracted, improving the precision of the wrapper.

One might suspect that the wrappers created by wrapper induction systems are close to optimal and thus further optimization is not necessary. However, learning efficient tree patterns or XPath queries is fundamental to many wrapper approaches, while proving to be inherently hard [12], even more so in presence of negative examples. This theoretical concern is underlined by the poor performance of existing wrapper engines demonstrated in [4]. Indeed, the results of this paper illustrate that, considering the vast resources used even by efficient wrapper engines such as OXPath, wrapper optimisation can have significant impact.

Many wrapper induction systems are based partially on standard XML query languages such as XPath or (less frequently) XQuery and can thus profit from existing optimization techniques for these languages. XPath optimization can be distinguished into static, syntactic rewriting techniques, such as [11, 10, 5], and cost-based plan selection as considered in [6, 2]. However, in most wrapping scenarios the cost of action steps dominates all other costs, yet actions are not part of XPath or XQuery and thus existing techniques are only pertinent to action-free parts. Optimizing action-free parts, however, has only little overall impact.

Our static and dynamic optimization methods are both variants of well-known query optimization techniques. The static optimization is an instance of the selection ordering problem [9], which refers to the problem of determining an order for applying a given set of commutative selection predicates (filters) to all tuples of a relation. The dynamic optimization is an instance of adaptive query planning [9] and of tabling techniques used for the evaluation of recursive or fixpoint languages [13]. However, neither has been applied to web wrappers before, where the presence of browser actions significantly affects their application. E.g., for tabling, it is sufficient to store the results for each unique page, as in-page selection and extraction does not significantly affect the overall performance, leading to smaller and more efficient tables.

8. CONCLUSION

We have introduced wrapper optimization as problem in its own right, observing two idiosyncrasies: **(1)** The number of page loads dominates the wrapper performance, such that preexisting techniques from, say, XPath optimization, apply, but only if they are driven by and adapted to a suitable cost model. **(2)** Since many wrappers operate on pages generated from templates, the full potential of wrapper optimization can only be exploited at runtime, after observing relevant facts on the underlying template.

After showing the effectiveness of a fundamental set of optimizations in this paper, in our future work, we will fully implement our approach, possibly with a refined cost model, altogether with further optimizations.

9. REFERENCES

[1] G. J. Bex, F. Neven, T. Schwentick, and S. Vansummeren. Inference of concise regular expressions and DTDs. *TODS*, 35(2):11:1–11:47, 2010.

[2] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery Processor powered by a Relational Engine. In *SIGMOD*, p. 479–490, 2006.

[3] B. Fazzinga, S. Flesca, and A. Tagarelli. Schema-based web wrapping. *Knowledge and Inf. Sys.*, 26:127–173, 2011.

[4] T. Furche, G. Gottlob, G. Grasso, C. Schallhart, and A. J. Sellers. OXPath: A language for scalable, memory-efficient data extraction from web applications. In *PVLDB*, 4(11):1016–1027, 2011.

[5] P. Genevès and J.-Y. Vion-Dury. Logic-based XPath optimization. In *DocEng*, p. 211–219, 2004.

[6] H. Georgiadis, M. Charalambides, and V. Vassalos. Cost based plan selection for XPath. In *SIGMOD*, p. 603–614, 2009.

[7] T. Gottron. Clustering template based web documents. In *ECIR*, p. 40–51, 2008.

[8] Y. Hao and Y. Zhang. A two-phase rule generation and optimization approach for wrapper generation. In *ADC*, p. 39–48, 2006.

[9] Z. G. Ives, A. Deshpande, and V. Raman. Adaptive query processing: Why, how, when, and what next? In *VLDB*, p. 1426–1427, 2007.

[10] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, p. 315–329, 2002.

[11] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *EDBT XMLDM*, LNCS 2490, 2002.

[12] S. Staworko and P. Wieczorek. Learning twig and path queries. In *ICDT*, p. 140–154, 2012.

[13] K. T. Tekle and Y. A. Liu. More efficient datalog queries: subsumptive tabling beats magic sets. In *SIGMOD*, p. 661–672, 2011.

[14] S. Zheng, R. Song, J.-R. Wen, and D. Wu. Joint optimization of wrapper generation and template detection. In *KDD*, p. 894–902, 2007.