

Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities

David F. Huynh, Robert C. Miller, David R. Karger

MIT Computer Science and Artificial Intelligence Laboratory,
The Stata Center, Building 32, 32 Vassar Street, Cambridge, MA 02139, USA
{dfhuynh, rcm, karger}@csail.mit.edu

ABSTRACT

Existing augmentations of web pages are mostly small cosmetic changes (e.g., removing ads) and minor addition of third-party content (e.g., product prices from competing sites). None leverages the structured data presented in web pages. This paper describes Sifter, a web browser extension that can augment a web site with advanced filtering and sorting functionality. These added features work inside the site's own pages, preserving the site's presentational style, as if the site itself has implemented the features. Sifter contains an algorithm that scrapes structured data out of web pages while usually requiring no user intervention. We tested Sifter on real web sites and real users and found that people could use Sifter to perform sophisticated queries and high-level analyses on sizable data collections on the Web. We propose that web sites can be similarly augmented with other sophisticated data-centric functionality, giving users new benefits over the existing Web.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces – Graphical user interfaces (GUI).

General Terms: Algorithms, Design, Human Factors.

Keywords: Web, augment, filter, sort, faceted browsing, dynamic query, tree alignment, HTML, DOM.

INTRODUCTION

Much of the data on the Web resides in structured databases behind web sites. However, as the contemporary web browser understands only rendering instructions (HTML), the typical web site must transform its structured data into HTML before serving up that data. When the data reaches the user's browser, it has lost most if not all of its original structure. Made readable for humans through web browsers, it is no longer conducive to machine processing.

Retaining structure in the data transfer from web site to web browser gives the web browser an opportunity to present

that data in ways that might better meet the user's needs. For example, it can add faceted browsing [25] functionality if none is offered by the original web sites. It can provide more sophisticated, novel visualizations such as starfield displays [7], FOCUS tables [20], and parallel bargrams [23].

Moreover, once capable of processing structured data, the browser can more effectively combine the public data on the Web and the private data on the user's local computer. For example, if the browser could understand that something on a web page is the contact information of a person, it could expose some user interface for adding that information directly to the user's address book. The browser could be told to inject the addresses of the user's close friends into any map of restaurants or movie theatres on any web site.

The web browser could also integrate structured data from multiple web sites. A decision such as buying a house might need support from information on several sites—houses on sale, groceries stores nearby, Vietnamese language schools for the kids, arts supplies stores for the artistic wife, jogging tracks for the athletic husband, churches for the whole family, etc. Yet, no existing web site will satisfy the unique combination of needs of every user. Enabling the browser to integrate data across sites on a per-need, per-user basis points to a possible solution.

This shift from a presentation-focused, textual document-centric Web to a processing-focused, structured data-centric Web has been envisioned but not yet realised by the Semantic Web project [5, 9]. Our previous efforts, Thresher [11] and Piggy Bank [3, 12], have illustrated the appeal of client-side structured data processing capabilities, such as plotting locations extracted from several sites together on a single map.

This paper describes Sifter, a web browser extension that can augment an arbitrary web site with filtering and sorting functionality. The added features work inside the site's own pages, preserving the site's presentational style, as if the site itself has implemented those features. For example, suppose the user visits a library web site and submits a search for books that returns several hundred results divided into 15 pages. Sifter allows the entire set of results to be filtered or sorted (e.g. by author, year, title, or subject), without having to switch away from the web browser or even away from the library site.

Sifter contains an algorithm that extracts structured data out of web pages, usually requiring no examples from the user to learn how to extract the data. For the book search example above, this algorithm would start from the first search result page, iterate through the 14 subsequent pages, and produce a database record for each search result on every page. Having extracted all items in that 15 page collection, Sifter can then compute filtering and sorting choices. As the user invokes filtering and sorting commands, Sifter reconstructs the web page in-place, showing only those items that satisfy the current filters in the current sorting order using data served from the local database, requiring no further help from the original web site.

We tested Sifter on real web sites and real users and found that the test subjects could use Sifter to perform sophisticated queries on realistic data collections on the Web.

RELATED WORK

As the essence of our work is the use of automatic data extraction algorithms within the web browser to augment existing web sites, there are two main areas of related work: automatic web data extraction and web page augmentation.

Although there has recently been a spate of efforts on automatically extracting data from the Web—[14, 18, 22, and 26] to name just a few—these approaches mostly work outside the web browser (e.g., as independent crawlers) and do not make the extracted data usable to end-users. An exception is Thresher [11] (built into Haystack [17]), which embeds a web browser and lets the user extract structured data from a web page by selecting some HTML fragment, invoking a context menu command on that selection, and then labelling

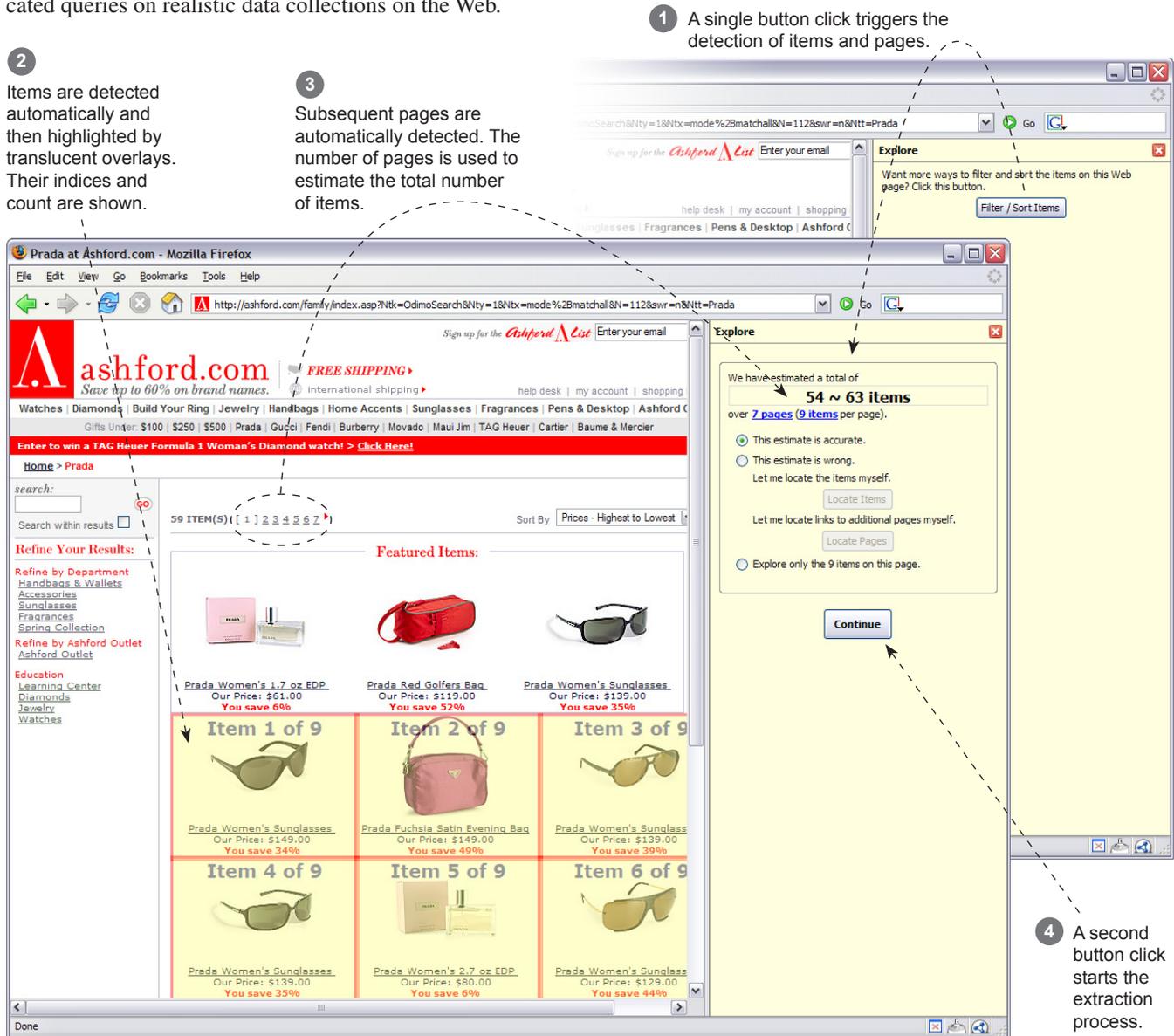


Figure 1. Sifter provides a one-click interface for triggering the detection of items and pages. One more button click starts the extraction process.

fields in that sample item. Having identified the “semantic items” on the web page and extracted their data, Thresher then adds context menu commands to each semantic item (e.g., “Call this person”). The user can also use Haystack’s rich browsing functionality on the items but to do so, she must switch into Haystack’s own browsing view where the original web page’s presentational style is not preserved.

The earliest work on augmenting web pages injected navigation guides into web pages [8, 13]. More recent efforts enable users to script modifications on web pages [2, 10], but so far they lack a rich data model to support augmentations more sophisticated than just cosmetic changes (e.g., removing ads) and simple addition of third-party content to web pages (e.g., injecting prices from competing sites). None can work on multiple pages or extensively re-shuffle the content of a page in response to user interaction.

There has also been related work that augments arbitrary unstructured text (not just web content) with semantic operations, e.g., Data Detectors [15], the Selection Recognition Agent [16], CyberDesk [24], and Microsoft Smart Tags. However, these approaches used hand-coded parsers to discover and extract structured data while Sifter discovers structured data automatically. Furthermore, Sifter provides operations over entire sets of items rather than over individual items.

USER INTERFACE DESIGN

Interaction with Sifter consists of two stages:

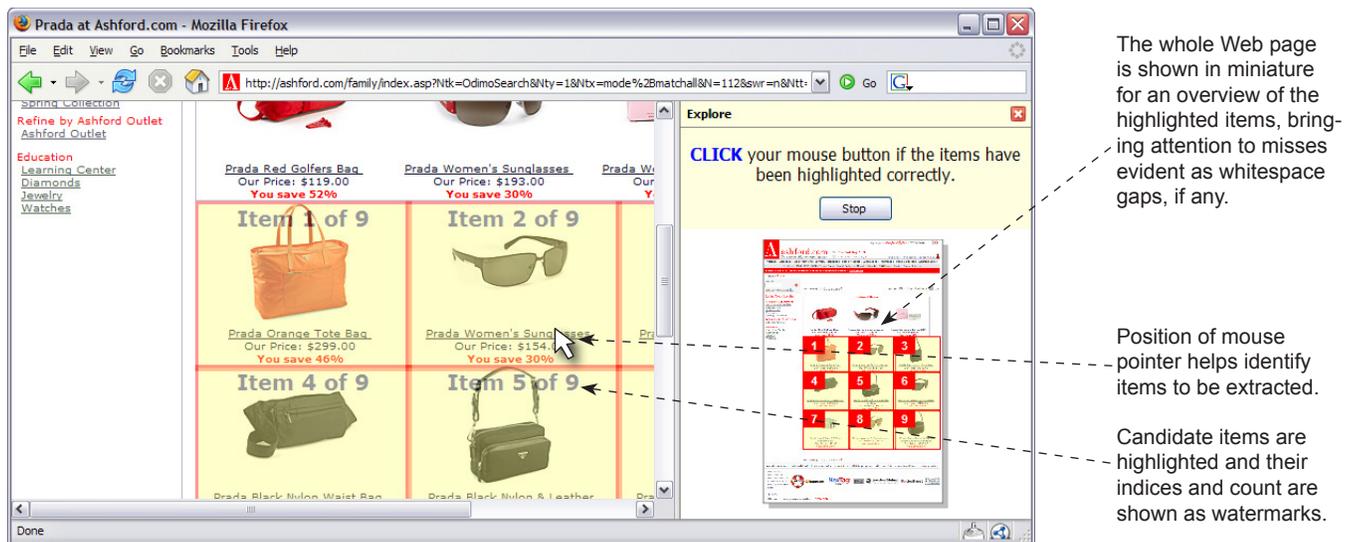
- Extraction: the system ascertains which parts of the web site to extract, and gives the user feedback about this process; and
- Augmentation: the system adds new controls to the web page and the browser that allow the user to filter and sort the extracted items in-place.

Extraction User Interface

Sifter’s user interface resides within a pane docked to the right side of the web browser (Figure 1). When the user first visits a web site, the Sifter pane shows a single button that, when clicked, triggers the detection of items on the current web page as well as links to subsequent pages, if any. (An item is, for example, a product as in Figure 1.) Items are highlighted in-place, and the total number of items spanning the detected series of pages is displayed prominently in the pane. If the system has incorrectly detected the items or the subsequent-page links, the user can correct it by clicking on an example item or a subsequent-page link in the web page. Once the Continue button (Figure 1) is clicked, the extraction process starts and Sifter pops up a dialog box showing the subsequent web pages being downloaded. Over all, getting the data extracted usually takes 2 button clicks.

During the extraction process, Sifter locates all items on all the web pages, extracts field values from each item as well as its HTML code, and stores each item as a record in a local database. For example, the local database accumulated from extracting the 7 pages of 59 items in Figure 1 would contain 59 records, each having one text field (title), two numeric fields (price and percent saving), and an HTML code fragment representing the entire item. This code fragment is used as a rendering of the item when the result set is filtered or sorted.

If the items are detected incorrectly, the user can click on the Locate Items button (Figure 1) and the Sifter pane will change to highlighting mode (Figure 2). In this mode, as the user moves the mouse cursor over the web page, the system inspects the smallest HTML element under the mouse cursor, generalizes it to other similar elements on the page, expands those elements to form whole items, and highlights these candidate items with translucent overlays. When the



The whole Web page is shown in miniature for an overview of the highlighted items, bringing attention to misses evident as whitespace gaps, if any.

Position of mouse pointer helps identify items to be extracted.

Candidate items are highlighted and their indices and count are shown as watermarks.

Figure 2. Sifter’s highlighting mode provides an interactive mechanism for the user to correct the automatic detection of items by hovering the mouse pointer of the web page and clicking once the items have been highlighted correctly.

user is satisfied with the highlighting, she can click the mouse button and the Sifter pane switches out of its highlighting mode.

Through early prototypes, we realized that the extraction UI must be streamlined as much as possible because data extraction is not by itself a user goal, but merely a system precondition for achieving the user's real goal (filtering or sorting). Scraping—transforming data to a more machine processible form—is an unfamiliar concept and perhaps seemingly unnecessary to the user, which the user may initially consider not worth the effort.

We considered eliminating the extraction UI altogether and provided a correction UI after the extraction process has finished. However, as the extraction process may be lengthy and not completely reliable, providing a preview of what the

system is going to do makes the wait more acceptable and gives the user a sense of greater control over the system.

Augmentation User Interface

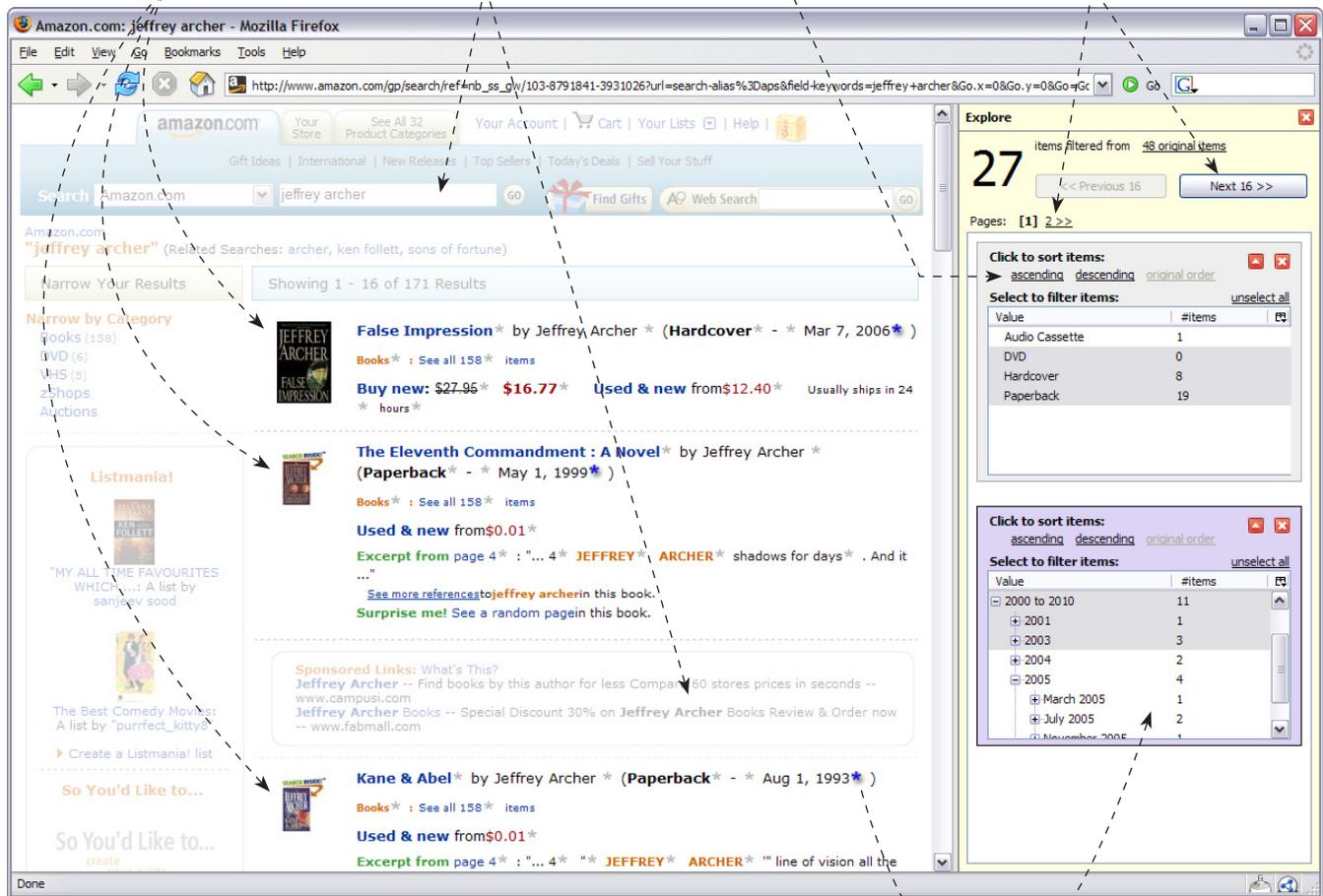
Figure 3 shows Sifter in action as the user makes use of new filtering and sorting functionality. An asterisk is inserted after each field value in the web page. When an asterisk is clicked, a *browsing control box* is displayed in the Sifter pane, containing the filtering and sorting controls for that field. Hovering the mouse pointer over either a field's asterisks or its browsing control box highlights both synchronously, so that the user can distinguish among them. This design addresses a critical problem in automatic data extraction: Sifter cannot automatically derive meaningful field names with which to label the browsing controls, because many field names are completely missing from the web page (e.g., nowhere in the Amazon page in Figure 3 are the book titles explicitly labeled "Title"). So instead of a field name, Sifter relies on the

Items satisfying the current dynamic query are inserted into the original Web page as if the Web site itself has performed the query.

Extraneous content (e.g., sponsor links) is faded away to avoid confusion and distraction.

Sorting controls for a field

Paging controls for the whole collection.



An asterisk is inserted after each field value. Clicking on an asterisk adds a browsing control box to the Sifter pane. Corresponding asterisks and boxes are co-highlighted when hovered.

Figure 3. After extraction is complete, the Sifter pane hosts browsing and sorting controls, which when invoked, re-render the resulting items inside the same web page (without invoking the original web site).

field values themselves, and their context within the items, as a way to attach meanings to the browsing controls.

The filtering controls for different field types (text, numbers, date/time) manage the field values differently. For numbers and date/time fields the values are classified into ranges and sub-ranges hierarchically, while for text fields the values are listed individually. Selecting a value or a range filters the current collection of items down to only those having that value or having values in that range. Multi-selection in a single field adds disjunctive query terms. Filtering on more than one field forms conjunctive queries. Selecting and de-selecting field values or ranges in a browsing control box updates the available values and ranges in other boxes as in any dynamic query interface [19].

As the user invokes filtering and sorting commands, Sifter dynamically rewires the web page to show the set of items satisfying the current filters in the current sorting order, as if the web site itself had performed the filtering and sorting operations. Sifter does so by removing the HTML fragment of each item on the page and then injecting into the same slots (where those removed fragments previously fit) the HTML fragments of the items satisfying the current dynamic query. These HTML fragments are retrieved from the local database, so there is no need to make a request to the original site when the dynamic query changes.

While the user is using Sifter’s filtering and sorting functionality, the rest of the original web page is faded out to indicate that interaction is now focused on the items alone. The original status indicators (e.g., number of items, number of pages) are faded to imply that they no longer apply to the items inside the web page. The original pagination, sorting, and browsing controls are faded to imply that invoking them would switch out of Sifter’s augmentation mode and let the user interact with the web site. We used fading rather than completely removing the rest of the web page so that the user still has a way to invoke the original web site. Furthermore, some parts of the original page may still be vital for the user to understand the items (e.g., column headers).

Since exploring the collection of items may involve clicking a link to view details about an item, Sifter stores the query state and automatically restores it when the user returns to the augmented page.

DATA EXTRACTION

Extraction of structured data from web pages takes 3 steps: locating items to extract; identifying subsequent web pages; and parsing useful field values from each item.

Item Detection

We posit that for many sequences of web pages containing lists of items (e.g., search results, product listings), there exists an XPath [6] that can precisely address the set of items on each page. Thus, our item detection algorithm involves de-

veloping such an XPath. (We will address the cases where each item consists of sibling or cousin nodes [26] in future work.) We also posit that this XPath can be computed just from the sample items on the first page in a sequence of pages. We have found that these assumptions hold on many database-backed web sites that generate HTML from templates.

Our algorithm is based on two observations. First, in most item collections, each item contains a link, often to a detail page about the item. So, links are likely to be useful as starting points for generating hypotheses for the XPath. Second, the item collection is typically the main purpose of the web page, so the items themselves consume a large fraction of the page’s visual real-estate. This gives us a way to choose the most likely hypothesis, namely, the one that uses the largest area of the page.

The item detection algorithm starts by collecting all unique XPaths to <A> elements on the current web page. For each element, its XPath is calculated by stringing together the tag names of all elements from the document root down to that element. CSS class names are also included. The resulting XPath looks something like this: “/HTML/BODY/TABLE/TBODY/TR/TD/DIV[@class=‘product’]/SPAN/A”. Each such XPath is general enough to cover more than just the original <A> element, but restrictive enough to address only those elements similar to it. This is called the generalization phrase, as illustrated by the straight arrows in Figure 4.

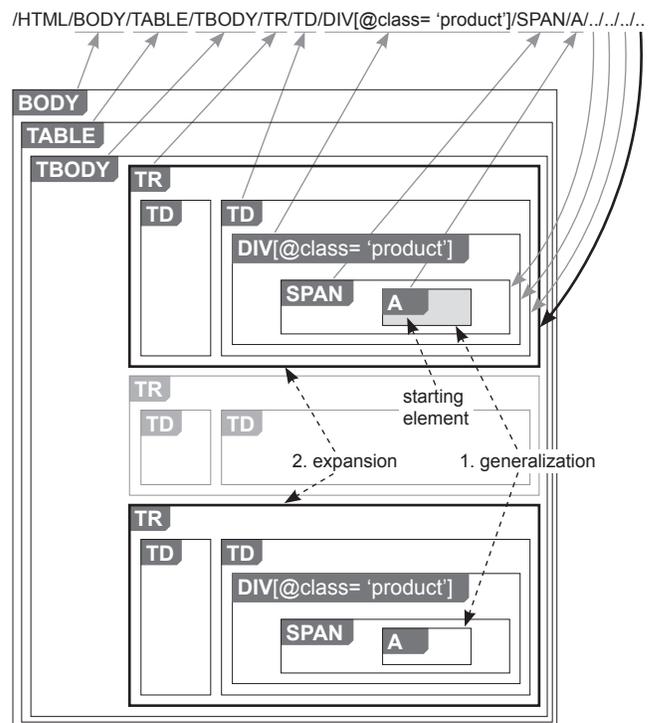


Figure 4. Given a starting HTML element, an item XPath is constructed by generalization to similar elements (straight arrows) and then expansion (curved arrows).

Each of these <A> XPath addresses a collection of <A> elements that could correspond one-to-one with the collection of items to be detected. We wish to find which of these <A> XPaths corresponds to a collection of items that take the largest amount of screen space.

Next, each <A> XPath is expanded to fully encompass the hypothetical items that the <A> elements reside within. To expand an XPath, we repeatedly append “/..” to it (see the curved arrows in Figure 4). As “/..” is appended, the set of HTML elements that the XPath addresses gets closer and closer to the document root. As long as the cardinality of that set remains unchanged, each HTML element in that set still resides spatially inside a hypothetical item. When the cardinality of the set drops, the XPath has been expanded too much such that it now describes the parent node(s) of the hypothetical items. For example, in Figure 4, if we append another “/..”, the resulting XPath would address a single TBODY element rather than two TR elements. We stop appending “/..” just before that happens. The result is a candidate item XPath.

Note that we append “/..” rather than truncate ending segments because truncation loses information. If the XPath in Figure 4 were instead truncated 4 times, the resulting XPath, “/HTML/BODY/TABLE/TBODY/TR”, would have included the middle TR, which does not have a link inside and could be extraneous, intervening content.

For each candidate item XPath, we calculate the total screen space covered by the HTML elements it addresses. The candidate item XPath with the largest screen space wins and is then used to add highlight overlays to the web page.

Subsequent-Page Detection

We use two heuristics to automatically detect subsequent pages. The heuristics are run in the order presented below, and when one succeeds, its results are taken as final.

Link Label Heuristic – Often, a web page that belongs in a sequence of pages contains links to the other pages presented as a sequence of page numbers, e.g.,

Pages: 1 [2] [3] [4] [5] [Next] [Last]

Occasionally, such a sequence shows not the page numbers but the indices of the items starting on those pages, e.g.,

Items: 1–10 [11–20] [21–30]

This heuristic attempts to pick out URLs from such a sequence of linearly increasing numbers. First, the text labels of all <A> links on the current web page are parsed. Only labels that contain numbers are kept. They are then grouped by the XPaths generated from the <A> elements. For each XPath, the numbers parsed from the labels are sorted in ascending order. Only those XPaths with linearly increasing sequences of numbers are kept. These final candidates are then sorted by the lengths of their sequences. The XPath

with the longest sequence is then used to pick out URLs to subsequent pages. If there is a tie, the XPath whose sequence increases at the highest rate wins. This heuristic fails if no XPath has a linearly increasing sequence of numbers.

URL Parameter Heuristic – URLs of pages in a sequence often encode the page numbers or the starting item indices as numeric URL parameters. For instance, Amazon.com encodes page numbers in the “page” parameter and Yahoo.com encodes starting item indices in the “b” parameter (Table 1). This heuristic attempts to detect such parameters so that URLs to subsequent pages can be generated. The URLs pointed to by all the links on the current web page are parsed to extract out URL parameters. For each parameter that has numeric values, its numeric values are collected in an array and sorted in ascending order. Then, only parameters whose values form linearly increasing sequences are kept. These final candidates are sorted by the lengths of their value sequences. The parameter with the longest sequence is then used to generate URLs to subsequent pages. If there is a tie, the parameter whose sequence increases at the highest rate wins. This heuristic fails if no parameter has a linearly increasing sequence of values.

If these heuristics fail then the user can intervene and point at the link to one of the subsequent pages (not necessarily the immediately following page). We compute the XPath of that link, which describes a collection of <A> elements. Given such a collection of <A> elements, we use the following heuristic to pick out the one that points to the next page.

Next Page Heuristic – Table 1 gives some sample URL spaces of contemporary web sites. Pages in a sequence might not differ by only one URL parameter which encodes either the page number or the index of the starting item. In some cases, more parameters are inserted (e.g., “%5Fencoding=UTF8” at Amazon.com) and some existing ones are removed (e.g., “search-alias=aps” at Amazon.com). In other cases, URL parameters are not used at all (e.g., at Dogpile.com). Rather, some segments of the URL are used to specify the current page. Worse yet, the whole URL of the current page is encoded as a single URL parameter to another domain for some tracking purpose (e.g., at Yahoo.com).

The next page heuristic sorts candidate URLs together with the current page’s URL and picks out the URL immediately “larger” than the current page’s URL. Simple string sorting does not work as “page=10” will be “less” than “page=9”. Instead, we break each URL into fragments separated by “/”, “?”, and “&”. We then sort the URLs by comparing corresponding fragments that contain numbers. (This heuristic cannot handle Yahoo’s URL space as shown in Table 1.)

Field Detection

We use a greedy algorithm to build a template out of sample items from sample pages in a sequence of pages. We construct the initial template by copying the first sample item’s

3 cases, items consisted of sibling nodes, which we do not currently handle.

Among the 30 sites, 24 (80%) displayed sequences of page numbers (rather than just “Next Page” links). The Link Label Heuristic detected 13 sequences, and the URL Parameter Heuristic detected 2 (15/24 = 63%). The Next Page Heuristic (requiring users’ intervention) worked on 9 of the remaining 15 cases (9/15 = 60%). Overall, subsequent pages could be identified for 24 out of 30 collections (80%).

There were 22 collections for which item XPaths could be found and subsequent pages could be identified accurately. Out of these 22 collections, 19 were perfectly extracted, yielding precisely the original numbers of items. The overall accuracy of extraction is 19/30 = 63%.

Note that accuracy was measured per collection rather than per item as in related data extraction work. To put this in perspective, the latest work on data extraction [26] processed 72 manually provided pages from 49 sites and achieved 40/49 = 82% collection accuracy. Over the 24 collections for which subsequent pages could be identified, our algorithm processed 176 pages automatically and achieved 19/24 = 79% collection accuracy.

The fields extracted that could be useful for filtering and sorting included: current price, original price, percent saving, author, artist, medium, date, shipping option, brand, number of store reviews, number of bids, container size, city, etc.

EVALUATION OF USER INTERFACE

Augmentation of web sites is a novel concept even to experienced web users, so we conducted a formative evaluation of Sifter to determine whether it was basically usable and useful, assuming the automatic extraction algorithm performed its best.

Design and Procedure

This study consisted of a structured task (during which the subjects took simple steps to familiarize with Sifter) followed by an unstructured task (during which the subjects employed their own knowledge of Sifter for problem solving).

At the beginning of each study session, the subject was told that she would learn how to use something called Sifter herself but was given no particular instructions on how to use it. This was patterned after the study on the Flamenco system in which the subjects were not introduced to the system in order to better mimic real world situations [25].

Task #1 required the subject to:

- follow a sequence of simple steps to use the Sifter pane to sort and filter a collection of 48 items spread over 3 web pages obtained by searching Amazon.com for “jeffrey archer.” The desired final result was the sub-collection of only hardcovers and paperbacks published in 2004 or later, sorted in descending order by their used & new

prices. The sequence of steps consisted of high-level “filter by date” and “sort by price” instructions, not low-level UI “click this button” and “select that list item” actions.

- use the Sifter pane by herself to obtain the list of 3 cheapest (if bought used) paperbacks by John Grisham in 2005 from Amazon.com.
- spend no more than 5 minutes using only the Amazon web site, but not Sifter, to find the 3 cheapest (if bought used) hardcovers by John Grisham in 2004.

Task #2 required the subject to:

- use the Sifter pane to decide whether the sale on Prada products on Ashford.com was good or not.
- use the Sifter pane to list 2 or 3 products among those that the subject considered good deals.
- use only the Ashford.com web site to judge whether the sale on Gucci products on Ashford.com was good or not, using the same criteria that the subject had used in judging the sale on Prada products.

Amazon.com was chosen for Task #1 as its search results were very structured, containing many fields useful for filtering and sorting. Furthermore, Amazon.com is popular and the subjects were more likely to be familiar with it. Ashford.com was chosen for Task #2 as its search results contained only two fields (price and percent saving), making it simpler to perform the high-level task of judging its sales.

At the end of the session, the subject rated her agreement/disagreement with 12 statements (on a 9-point Likert scale) regarding her experience learning and using Sifter.

Participants

Eight subjects (4 male, 4 female) were recruited by sending an e-mail message to a mailing list and posting paper ads around a local college campus. Six were in their 20s, the other two were 30s and 40s. All 8 subjects used the Web almost everyday, and all subjects visited Amazon.com at least a few times a month. None had ever visited Ashford.com.

All subjects had used the Web for more than just shopping. They had searched for some combinations of the following types of information: news; images; contact information of people and organizations; maps and driving directions; hobbies (e.g., recipes, chess); reviews (on restaurants, movies, books, products); tutorials (e.g., languages, logics); and professional research (e.g., publications, scientific data).

Apparatus

Subjects received \$10 each for participating in a 30 – 45 minute study session. All sessions were conducted by one investigator on a single computer (Pentium 4 2.53GHz, 1.00GB) with an 18” LCD flat panel at 1600×1200 resolution in 32-bit color and a Microsoft Wireless IntelliMouse Explorer 2.0 (with mousewheel), running Microsoft Windows XP. UI

strongly disagree 1  9 strongly agree	
	1. Sifter is hard to learn how to use.
	2. Sifter is tedious to use.
	3. The filtering and sorting features in Sifter are slow.
	4. Sifter shows redundant information (easily found on the Web sites).
	5. After clicking “Continue,” I need to wait for a long time before I can use Sifter.
	6. Sifter is simple to use.
	7. Sifter is powerful (providing advanced features).
	8. Sifter displays interesting information.
	9. Sifter displays useful information.
	10. Sifter is enjoyable to use.
	11. Sifter adds value to the Web sites in this user study.
strongly disagree 1  9 strongly agree	12. I believe Sifter will add value to some other Web sites I have used.

Table 2. Results from the exit survey of the formative evaluation show encouraging evidence that the Sifter pane is usable (#1, #2, #6) and useful (#8, #9, #11, #12) even when it is considered to offer advanced functionalities (#7) .

events were recorded in a timestamped log and the investigator observed the subjects and took written notes.

Results

All subjects completed the parts of Task #1 involving Sifter. Only 5 out of 8 completed the parts involving using the Amazon web site *without* Sifter. The other subjects could not learn how to use Amazon to perform sophisticated queries within 5 minutes. Among the 5 subjects who succeeded, only one made use of Amazon’s Advanced Search feature. The other 4, despite their previous experience with the Amazon web site, could only sort the items by one criterion and manually scan the list for items satisfying the other criteria. This indicates that advanced browsing features implemented by the web browser in a unified manner across web sites may be more discoverable, learnable, and usable than those same advanced features officially supported by individual web sites but have been suppressed in favor of more commonly used functionality.

Seven subjects completed Task #2 and one refused to finish Task #2 as he said he had no knowledge of Prada and Gucci products and thus could not judge their sales. For the first part of Task #2, 6 out of the 7 subjects used Sifter to look at the distribution of the percent saving. One subject could not understand how Sifter would help her judge the sale.

Table 2 shows encouraging evidence that the subjects found Sifter powerful yet easy to learn and use. However, the extraction process was thought to be slow. Data extraction speed depends on network performance and web server responsiveness, but on average, each test collection of 50 or so items took 30 seconds to extract.

Although there was no show-stopper problem with the user interface, some users were taken aback by the verification step (when the system announced its estimate of the items

to be extracted and asked for confirmation). As they saw no other choice except clicking “Continue,” they did so just to see what would happen next, in hope but not certain that that route would ultimately allow them to sort and filter. This behavior was not a surprise as web site augmentation was a new experience and the necessity for extracting data before augmentation could take place was poorly understood, if understood at all. To fix this problem, we will have to boost the accuracy of our algorithms, pre-load and process subsequent pages even before the user clicks “Filter/Sort Items,” and let the user make corrections from the augmentation UI.

Sorting operations took very little time and the re-shuffling of items inside the web page was too fast and subtle to shift the user’s attention from the Sifter pane where she just invoked a sorting command to the web page. The user often had to double-check the resulting list of items herself. To fix this, we can slow down or animate the changes in the web page. Filtering operations produced more drastic changes and did not suffer from the same problem.

One subject opened too many browsing control boxes and became confused as to which field each box corresponded to. He was not able to notice the synchronized highlighting of browsing control boxes and field asterisks. To fix this, we can color-code the asterisks and the browsing control boxes as well as use a variety of shapes rather than just asterisks.

Asked to filter for only items published in 2005, some subjects had to manually find one sample item published in 2005 in order to click on the asterisk next to its publishing date. Other subjects simply clicked on the asterisk next to any publishing date. If we are able to derive meaningful field names, this problem will be resolved.

Only 5 of the 8 subjects interacted with the faded areas of the web pages when they needed to use the web sites’ func-

tionality (e.g., performing a search for “john grisham”). In the future, we will need to change Sifter’s UI such that users can feel comfortable making use of the original web sites’ features, knowing the difference between those original features and the added functionality.

One subject—the only one who used Amazon’s Advanced Search feature—asked when she would be able to use Sifter in her own browser. She mentioned that there were a number of sites she used frequently which did not offer the browsing functionality that she needed. Another subject said that although he never had to perform the kinds of task in this study on the Web, he had to perform similar tasks in spreadsheets.

CONCLUSION

We were able to devise heuristics for automatically scraping structured data out of lists of items from sequences of web pages and augment the original web pages with advanced sorting and filtering features right within those pages. Based on the encouraging results from the user study, enabling the web browser with structured data processing capabilities benefits end-users. For better user experience, we propose that web sites also serve structured data in a structure-preserving format along with HTML code so that the inherently slow and inaccurate data extraction process is no longer necessary.

No web site can ever offer an experience over data aggregated from multiple sites chosen for the unique combination of needs of any user in any situation. We will next investigate user interfaces for such aggregated data. The challenges are in preserving important aspects of the original sites (such as the presentation of items) while providing new features over the unified data in a coherent manner.

ACKNOWLEDGEMENTS

This work was supported by [Haystack’s funders], by the National Science Foundation (IIS-0447800), and by the Mellon Foundation. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

We are grateful to all of our user study participants and pilots. We would like to thank members of the Haystack group, the User Interface Design group, and the Simile group at MIT CSAIL for their valuable discussions on our work.

REFERENCES

[1] Evaluation of Sifter’s data extraction algorithm. <http://people.csail.mit.edu/dfhuynh/research/papers/uist2006-augmenting-web-sites-stats.pdf>.

[2] Greasemonkey. <http://greasemonkey.mozdev.org/>.

[3] Piggy Bank. <http://simile.mit.edu/piggy-bank/>.

[4] Resource Description Framework (RDF) / W3C Semantic Web Activity. <http://www.w3.org/RDF/>.

[5] Semantic Web project. <http://www.w3.org/2001/sw/>.

[6] XML Path Language (XPath) Version 1.0. <http://www.w3.org/TR/xpath>.

[7] Ahlberg, C., B. Shneiderman. Visual information seeking: tight coupling of dynamic query filters with starfield displays. *CHI 1994*.

[8] Barrett, R., P. Maglio, and D. Kellem. How to personalize the web. *CHI 1997*.

[9] Berners-Lee, T., J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.

[10] Bolin, M., M. Webber, P. Rha, T. Wilson, and R. Miller. Automation and customization of rendered Web pages. *UIST 2005*.

[11] Hogue, A. and D. Karger. Thresher: automating the unwrapping of semantic content from the World Wide Web. *WWW 2005*.

[12] Huynh, D., S. Mazzocchi, and D. Karger. Piggy Bank: experience the Semantic Web inside your Web browser. *ISWC 2005*.

[13] Joachims, T., D. Freitag, and T. Mitchell. WebWatcher: a tour guide for the World Wide Web. *IJCAI 1997*.

[14] Lerman, K., L. Getoor, S. Minton, and C. Knoblock. Using the structure of Web sites for automatic segmentation of tables. *SIGMOD 2004*.

[15] Nardi, B.A., J.R. Miller, and D.J. Wright. Collaborative, programmable intelligent agents. *Communications of the ACM* 41:33, 96-104, March 1998.

[16] Pandit, M.S., and S. Kalbag. The Selection Recognition Agent: instant access to relevant information and operations. *IUI 1997*.

[17] Quan, D., D. Huynh, and D. Karger. Haystack: a platform for authoring end-user Semantic Web applications. *ISWC 2003*.

[18] Reis, D.C., P.B. Golgher, A.S. Silva, and A.F. Laender. Automatic Web news extraction using tree edit distance. *WWW 2004*.

[19] Shneiderman, B. Dynamic queries for visual information seeking. *IEEE Software*, 11:6, 70-77, 1994.

[20] Spence, M., C. Beilken, and T. Berlage. FOCUS: the interactive table for product comparison and selection. *UIST 1996*.

[21] Tai, K.-C. The tree-to-tree correction problem. *J. Association of Computing Machinery*, 26(3):422-433, July 1979.

[22] Wang, J.-Y., and F. Lochovsky. Data extraction and label assignment for Web databases. *WWW 2003*.

[23] Wittenburg, K., T. Lanning, M. Heinrichs, and M. Stanton. Parallel bargrams for consumer-based information exploration and choice. *UIST 2001*, 51-60.

[24] Wood, A., A. Dey, and G.D. Abowd. CyberDesk: automated integration of desktop and network services. *CHI 1997*.

[25] Yee, K.-P., K. Swearingen, K. Li, and M. Hearst. Faceted metadata for image search and browsing. *CHI 2003*.

[26] Zhai, Y., and B. Liu. Web data extraction based on partial tree alignment. *WWW 2005*.