Skip Blocks: Reusing Execution History to Accelerate Web Scripts

SARAH CHASINS, University of California, Berkeley, USA
RASTISLAV BODIK, University of Washington, USA

With more and more web scripting languages on offer, programmers have access to increasing language support for web scraping tasks. However, in our experiences collaborating with data scientists, we learned that two issues still plague long-running scraping scripts: i) When a network or website goes down mid-scrape, recovery sometimes requires restarting from the beginning, which users find frustratingly slow. ii) Websites do not offer atomic snapshots of their databases; they update their content so frequently that output data is cluttered with slight variations of the same information — *e.g.*, a tweet from profile 1 that is retweeted on profile 2 and scraped from both profiles, once with 52 responses then later with 53 responses.

We introduce the *skip block*, a language construct that addresses both of these disparate problems. Programmers write lightweight annotations to indicate when the current object can be considered equivalent to a previously scraped object and direct the program to skip over the scraping actions in the block. The construct is hierarchical, so programs can skip over long or short script segments, allowing adaptive reuse of prior work. After network and server failures, skip blocks accelerate failure recovery by 7.9x on average. Even scripts that do not encounter failures benefit; because sites display redundant objects, skipping over them accelerates scraping by up to 2.1x. For longitudinal scraping tasks that aim to fetch only new objects, the second run exhibits an average speedup of 5.2x. Our small user study reveals that programmers can quickly produce skip block annotations.

CCS Concepts: • **Software and its engineering → Control structures**;

Additional Key Words and Phrases: Web Scraping, Incremental Scraping, Programming By Demonstration, End-User Programming

ACM Reference Format:
Sarah Chasins and Rastislav Bodik. 2017. Skip Blocks: Reusing Execution History to Accelerate Web Scripts. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 51 (October 2017), 28 pages. https://doi.org/10.1145/3133875

1 INTRODUCTION

Last year we started working with a team of sociologists investigating how rents were changing across Seattle neighborhoods. They aimed to scrape Craigslist apartment listings once a day. We met with the research assistant who would handle data collection and walked him through using the Helena scraping tool. We expected that the main challenges would be scraping script errors, errors like failing to extract data from some webpages. Instead, the biggest obstacle was that the machine he used for scraping – a laptop connected to his home WiFi network – regularly lost its network connection in the middle of the night, partway through long-running scrapes. He would check progress the following mornings, find that the network connection had gone down, then

Corresponding author address: Sarah Chasins, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, schasins@cs.berkeley.edu, 387 Soda Hall MC 1776, Berkeley, CA 94720-1776.
Authors' addresses: Sarah Chasins, University of California, Berkeley, USA; Rastislav Bodik, University of Washington, USA.

2475-1421/2017/10-ART51
https://doi.org/10.1145/3133875

have to restart the script. Restarted executions repeated much of the originals' work, and they were no less vulnerable to network failures.

Worse, the scraping script was repeating work even when the network was well-behaved. Craigslist shows a list of ads that relate to a search query. Each page of results shows a slice of the full list. To select the slice for a new page, Craigslist indexes into the up-to-date search result. The result is updated whenever a new ad is posted, with new ads displayed first — that is, appended to the head of the list. If a new ad is posted between loading page n and loading page $n + 1$, the last ad from page n shifts to page $n + 1$. In a scraped snapshot of the list, the ad appears twice. With new rental listings arriving at a fast pace, a single execution of the scraping script routinely spent 13 hours rescraping previously seen ads.

These are not errors in the scraping script *per se* — if the network were perfect and the website served an atomic snapshot of its data, the scraping script would have collected the data correctly. The importance of network and server failures was an unexpected finding that changed our perspective on how to improve the scraper-authoring experience. This paper presents the result, the design of language support to address extrinsic challenges that hamper long-running web scrapers when they collect large, real datasets.

1.1 Extrinsic Challenges to Long-Running Scraping Tasks

Our experience suggests a programming model for large-scale web scraping must handle these four challenges:

(1) *Web server and network failures.* Server outages and temporary network connection failures can interrupt a scraping script and terminate the script's server session. In general, it is impossible to resume execution at the failure point. While we could checkpoint the client-side script and restart it at the last checkpoint, we do not control the web server and thus cannot restart the server session at the corresponding point. Section 5 discusses recovery strategies, concluding that restarting execution from the beginning appears to be the only general option. However, scraping scripts that run for hours or days are likely to fail again if the mean time to failure of a data scientist's WiFi connection is only a few hours.

(2) *Alternative data access paths.* It is common that some website data can be accessed via different navigation paths. For example, consider scraping the profiles of all Twitter users retweeted by one's friends. If two friends have retweeted user A, we will scrape A's user profile at least twice. Rescraping can produce inconsistent data: one data point indicating that A has 373 followers and one indicating he has 375 followers. Repeated and inconsistent entries could be merged in postprocessing, but duplicates interfere with analytics that we may want to perform as the data is collected. Further, revisiting profiles can substantially slow the execution.

(3) *Non-atomic list snapshots.* A script typically cannot obtain atomic snapshots of website databases because they can be updated during script execution, which changes the web server's output to the browser. This includes cases like the Craigslist ad updates that cause some ads to appear on multiple pages of listings and ultimately to appear multiple times in the scraped snapshot. As with duplicates caused by multiple access paths, this may produce internally inconsistent data. Additionally, rescraping this kind of duplicate wastes up to half of scraping time (see Figure 8a).

(4) *Repeated scraping.* Even when a script finishes without failure, there can be reasons to rerun it. First, the non-atomic snapshot phenomenon may cause data to be skipped during pagination. (For instance, in the Craigslist example, deleting an ad could move other ads up the list.) Rerunning the script increases the likelihood that all data is collected. Second, we may want to scrape a new version of the dataset days or weeks later to study trends over time. In both scenarios, it is common for most data on the website to remain unchanged, so much of the

rescraping is redundant. Empirically, we observe that in our users' datasets about 74% of scraped data was already present one week before.

1.2 The Skip Block

We address all four challenges with a single construct called the *skip block*. The skip block wraps script statements that scrape a logical unit of data — for example, the profile of a single user. When the skip block finishes without a failure, it remembers that the profile has been scraped. When this profile is encountered again, the block is skipped. The rest of this subsection explains some key features of skip blocks.

User-defined memoization. How does the skip block know it is safe to skip the block's statements? After all, the website data may have changed since it was last scraped, in which case the statements would produce new values. In general, it is impossible to ascertain that the block would produce the same values without actually re-executing the entire block. We must therefore relax the requirement that the scraped values are identical. The idea is to involve the user, who defines the "key attributes" for each skip block. For instance, the key could be the triple $(first\ name, last\ name, current\ city)$ for a user profile. When this skip block scrapes a triple that has previously been scraped, it skips the statements that would scrape the remaining attributes of the user, even though they may have changed.

Hierarchical skip blocks. Skip blocks can be nested, which gives us adaptive granularity of skipping. For example, cities have restaurants, and restaurants have reviews. If we want to scrape reviews, we may want to put skip blocks around all three entity types: reviews, restaurants, and cities. These skip blocks will be nested. For this task, as with many large scraping tasks, a single skip block is insufficient. Imagine we use skip blocks only for the restaurant entity, and the network fails during script execution. If the failure occurs after all of Berkeley's restaurants were scraped, the script will still have to iterate through the list of thousands of Berkeley restaurants before it can proceed to the next city. In contrast, if we use a skip block at the city level, the script will simply skip over all of Berkeley's restaurants and their associated reviews, proceeding directly to the next city. However, a single skip block at the city level is also impractical. Scraping a single city's reviews takes hours. If the failure occurs while visiting the 2,000th Berkeley restaurant, we do not want to throw away the hours of work that went into scraping the first 1,999 Berkeley restaurants. By using a skip block at both the city and the restaurant level, we can handle both of these failures gracefully, skipping whole cities where possible but also falling back to skipping over restaurants.

Staleness-controlled skipping. Programmers can override skipping of previously scraped data. This control relies on timestamps stored when the data is scraped. We use two kinds of timestamps: (1) *Physical timestamps.* Say we are scraping the restaurant example once a week and want to continually extend the scraped dataset with fresh website updates. In this scenario, the programmer may specify that a restaurant should be skipped only if it has been (re)scraped in the past month. This means that the data will be refreshed and new reviews retrieved if the most recent scrape of the restaurant is more than a month old. (2) *Logical timestamps.* To recover from a failure — e.g., a network failure — we skip a block if it was collected since the prior *full run*, which we define as an execution that terminated without failure. To handle this scenario, we use logical timestamps. Logical time is incremented after each full run.

Implementation. We have added skip blocks to the Helena web scripting language [Chasins 2017]. Skip blocks could also be introduced to other web scripting languages, such as Selenium [Selenium 2013], Scrapy [Scrapy 2013], and Beautiful Soup [Richardson 2016]; see Section 6.2 for a discussion of alternative implementations of skip blocks.

We selected Helena because it was designed for end users who author Helena programs by demonstration. We believe that access to skip blocks may benefit end users even more than programmers because the latter can overcome the problems that motivate skip blocks by reverse-engineering the website and devising website-specific solutions.

In contrast, the user of a Programming by Demonstration (PBD) scraping tool may lack (i) the programming skills or (ii) the insight into webpage structure necessary to build a site-specific fix. Without explicit language support, many users will have no recourse when network and server failures arise. Skip blocks offer the advantage of being easy for end users to write (see Section 9) and also being applicable regardless of the underlying webpage structure, so that end users can apply them wherever our motivating failures emerge.

1.3 Evaluation

We evaluate skip blocks using a benchmark suite of seven long-running scraping scripts that scrape datasets requested by social scientists. Our evaluation assesses how skip blocks affect failure recovery (Challenge (1)), a single execution (Challenges (2) and (3)), and multiple executions (Challenge (4)). We also evaluate whether skip blocks are usable.

We evaluate the effects of skip blocks on failure recovery and find that they prevent scripts from unnecessarily rescraping large amounts of data, accelerating recovery time by 7.9x on average. The average overhead compared to a hypothetical zero-time recovery strategy is 6.6%. We also evaluate the effects of skip block use on longitudinal scraping tasks. We repeat a set of large-scale scraping tasks one week after a first run and observe that using the skip block construct produces speedups between 1.8x and 799x, with a median of 8.8x.

Finally, we evaluate whether programmers and non-programmers can use the skip block construct with a small user study. We find that users can both learn the task and produce eight out of eight correct skip block annotations in only about 12 minutes.

1.4 Contributions

We make these novel contributions:

- *The skip block, a construct for hierarchical memoization with open transaction semantics.* The programming model (i) allows users to define object equality, specifying which object attributes to treat as volatile and non-identifying and (ii) adapts the memoization granularity during recovery, skipping initially large blocks, then many smaller blocks.
- *A notion of staleness that handles both failure recovery and longitudinal scraping.* With timestamp-based staleness constraints, a script skips a block only if it was previously committed within a specified interval of physical or logical time.
- *A suite of web scraping benchmarks and an evaluation of skip blocks.* We developed a benchmark suite of large scraping tasks. We evaluated how skip blocks affect performance on these benchmarks in three scenarios: failure recovery, a single-execution context, and a multi-execution context. We also conducted a user study that assesses the usability of our construct.

In this paper, Section 2 expands on the rationale for our skip block design. In Section 3, we briefly describe Helena, the web automation language in which we introduce the skip block. Next, we give the semantics of the skip block construct in Section 4. Section 5 discusses the feasability of alternative designs. Section 6 details the implementation. Our evaluation includes details of our benchmark suite (Section 7), then evaluations of both the performance implications of skip blocks (Section 8) and whether they are usable by end users (Section 9). Finally, Section 10 discusses how our approach relates to prior work.

2 OVERVIEW

In this section, we will first categorize the four scraping challenges that motivate our new language construct into two failure classes. Next, we will discuss the key characteristics of the skip block construct, then how these characteristics allow skip blocks to handle the two failure classes.

2.1 Web Scraping Failures

We have been working on a web automation language called Helena, designed for end users and paired with a PBD tool for drafting programs [Chasins 2017]. During the course of developing Helena and testing its expressivity, we wrote scripts for social scientists and directly shared the PBD tool with two groups of social scientists and a handful of computer scientists. Following these experiences and the feedback we received during script development, we put together a list of the failures and programmability issues that we had observed. While this review of failure types was unsystematic, based mostly on users' requests for help, it revealed an interesting pattern. The most serious failures, the four described in Section 1.1, fell into two categories:

- **Access failures**: We use *access failure* to refer to any failure that prevents a server request from producing the expected response in the web browser. This includes cases where the scraper loses the network connection and also cases where the server goes down or otherwise ceases to respond. This also extends to cases in which a session times out or otherwise ends, requiring the user to log back into a site.
- **Data consistency failures**: We use *data consistency failure* to refer to any failure caused by the fact that web server output does not come from a single atomic read of the website's internal state. Websites may update their underlying data stores between page loads within a single execution of a scraping script and also between executions. They may add new objects to lists of objects, reorder objects in a list, or change object attributes. The end result is that objects can fail to appear in a given script execution or they can appear multiple times, sometimes with different attribute values.

Another look at the challenges described in Section 1.1 reveals that Challenge (1) stems from access failures while Challenges (2), (3), and (4) are different manifestations of data consistency failures.

Access and data consistency are classes of important failures that no client-side scraping program can prevent, since they are extrinsic to the scraping process. Even the best custom scraper cannot prevent the server from going down or the site from updating its database between two page loads. Given that these failures cannot be eliminated, it became clear we needed to offer Helena programmers a way to handle them gracefully.

2.2 Running Example

To explore the key characteristics of the skip block construct, we will use a running example. Say we want to scrape information about all papers by the top 10,000 authors in Computer Science, according to Google Scholar. We will iterate through each author. For each author, we will iterate through the author's papers. Finally, for each paper, we will iterate through the histogram of the paper's citations by year.

Figure 1 shows a program that executes this task using the Helena web automation language. Helena offers programmers abstractions for webpages, logical relations (e.g., authors, papers, years), cells of the relations, and DOM nodes. Relations are logical in that their rows may be spread across many pages, which users typically navigate using 'Next' or 'More' buttons. In our example, the `authors` relation is a list of 10,000 `(author_name)` tuples spread over 1,000 webpages, because each page of results shows only 10 authors; there are many `papers` relations, one per author, and a

```
 1  p1 = load("https://scholar.google.com/citations?view_op=search_authors")
 2  type(p1,                   , "label:computer_science") // type query into Google Scholar search bar
 3  p2 = click(p1, 🔍) // click on Google Scholar search button
 4  for author_name in p2.authors{
 5     p3 = click(p2, author_name)
 6     h_index = scrape(p3, 117)
 7     for title, year, citations in p3.papers{
 8        p4 = click(p3, title)
 9        for citation_year, citation_count in p4.citations{
10           addOutputRow([author_name, h_index, title, citations, year, citation_year, citation_count])
11        }
12     }
13  }
```

Fig. 1. The Helena program for scraping papers by authors who are labeled with the 'computer science' tag in Google Scholar. Green-outlined boxes are screenshots of DOM nodes, which the Helena tool takes during a user demonstration phase and displays as user-facing identifiers for DOM nodes. The variables p1 through p4 are page variables, each referring to an individual webpage. The expression `p2.authors` in line 4 evaluates to a reference to the `authors` relation represented in page variable p2. Essentially, `authors` is a function that maps a page to the slice of author data in the page; it uses DOM paths and other node attributes to identify relation cells.

```
 1  p1 = load("https://scholar.google.com/citations?view_op=search_authors")
 2  type(p1,                   , "label:computer_science")
 3  p2 = click(p1, 🔍)
 4  for author_name in p2.authors{
 5     p3 = click(p2, author_name)
 6     h_index = scrape(p3, 117)
 7     skipBlock(Author(author_name.text, author_name.link, h_index.text)){
 8        for title, year, citations in p3.papers{
 9           skipBlock(Paper(title.text, year.text), Author){
10              p4 = click(p3, title)
11              for citation_year, citation_count in p4.citations{
12                 addOutputRow([author_name, h_index, title, citations, year, citation_year, citation_count])
13              }
14           }
15        }
16     }
17  }
```

Fig. 2. The Figure 1 program with skip blocks added. The first skip block (line 7) indicates that an author object should be skipped if a previously scraped author had the same name, link, and h-index. The second skip block (line 9) indicates that a paper object should be skipped if a prior paper with the same author ancestor also had the same title and publication year.

Table 1. A snippet of the dataset output by the Figure 1 and Figure 2 programs. The first two columns are associated with **authors**, the next three columns with **papers**, and the final two columns with **citation years**.

author_name	h_index	title	citations	year	citation_year	citation_count
J. Doe	34	A Paper Title	34	2016	2016	11
J. Doe	34	A Paper Title	34	2016	2017	23
J. Doe	34	Another Paper Title	28	2012	2012	9
...	...	...	...	...	...	...

typical `papers` relation is a list of hundreds of `(title, year, citations)` tuples, spread over 10-50 pages.

In Figure 1, variables p1, ... , p4 refer to the pages on which actions are executed. The expression `p2.authors` extracts rows of the authors relation from page p2. (Note that Helena hides the fact that rows of the relation are in fact spread across many pages.) DOM nodes are represented by names if they have been assigned names (as when they are associated with a cell in a relation) or by screenshots of the nodes. The `addOutputRow` statement adds a row to the output dataset. By default, the Helena PBD tool creates scripts that produce a single relation as output because the target users are familiar with spreadsheets. Our sample program produces an output dataset like the snippet of data in Table 1.

2.3 The Skip Block Construct

We briefly describe the key characteristics of the skip block construct.

Object Equality. Figure 2 shows a version of our running example that includes two skip blocks. The first skip block introduces an Author entity. It indicates that if two different author objects have the same name text, the same name link, and the same h-index text, we should conclude that they represent the same author. It also indicates that all code in the associated block scrapes data related to the author. The attributes for defining object equality can come from multiple different webpages, as they do in this case, with the author name appearing on `p2` but the h-index on `p3`.

Committing. When the body of a skip block successfully finishes, we add a new record to a durable commit log. To register that an object's skip block completed, the program commits the tuple of object attributes used to uniquely identify it — in our example, the name text, name link, and h-index text of the author.

Skipping. When we enter a skip block, we first check the commit log to see if we have already successfully scraped the current object. Recall that we conclude two objects represent the same entity if they share the key attributes identified by the programmer. If we find a commit record for the current object, we skip the body of the skip block. For instance, if we scrape author A in one execution, then come back and scrape the data the next day, we will probably see author A again. This time, as long as the name text and link are the same and the h-index is the same, the script will skip the associated block; even if the author's email or institution have changed, the script will skip the block. However, say the author's h-index is updated; in this case, the script will not skip the block, and it will rescrape author A's papers.

Nested skip blocks. The second skip block in Figure 2 introduces a Paper entity. This is a nested skip block, because it appears in the body of the author block. A nested block is skipped when its enclosing block is skipped. Thus, when an author block is skipped, all the associated paper blocks are also skipped. The paper block in Figure 2 indicates that if two paper objects have the same title text, year text, *and* the same Author ancestor, we should conclude that they represent the same paper. The Author ancestor comes from the enclosing skip block. We could also define Paper without the Author argument, which would change the output dataset. Recall that paper data is DAG-structured. If author A and author B coauthor paper 1, it will appear in the paper lists of both A and B. If we encounter paper 1 first in A's list, then in B's list, we will have to decide whether to scrape it again in B's list. Should we skip it, because we only want to collect a list of all unique papers by the top 10,000 authors? Should we scrape it again because we want to know that both author A and author B have a relationship with paper 1? The Figure 2 definition is the right definition if we want to collect the set of relationships between authors and papers. The alternative definition — the one that does not require an Author match — is the right definition if we just want the set of all papers by the top 10,000 authors.

Atomicity. Note that the paper skip block includes an `addOutputRow` statement. This statement adds a row to the output — that is, the dataset the user is collecting — not to the commit log. However, because it is inside a skip block, this statement now behaves differently. The output rows are no longer added directly to the permanent data store. Rather, they are only added if the entire skip block finishes without encountering a failure. (Note that a failure terminates the whole program.) An `addOutputRow` statement belongs to its immediate ancestor. Thus, the Figure

2 `addOutputRow` adds data when the Paper skip block commits, not when the Author skip block commits.

Open Transactions. Inner skip blocks can commit even if ancestor skip blocks fail. In this sense they are akin to open nested transactions [Ni *et al.* 2007]. This design means that even if the network fails before the script commits author A, if it has scraped 500 of author A's papers, it will not revisit those 500 papers or their citations.

Timestamps. Each commit record is tagged with two timestamps, one physical and one logical. The logical timestamp is incremented each time the end of the scraping script is reached. This means that even if the script must restart to handle an access failure, all commit records produced before a successful script termination will share the same logical timestamp.

Staleness Constraints. The skip blocks in Figure 2 will skip if any run of the program has ever encountered the current object. If we want to change this behavior, we can provide a staleness constraint. By default, we use a staleness constraint of $-\infty$, which allows matches from anywhere in the commit log. However, we can provide an explicit staleness constraint in terms of either physical or logical timestamps. For instance, we could say to skip if we have scraped the current author object in the last year (`skipBlock(Author(author_name.text, author_name.link, h_index.text), now - 365*24*60)`), or we could say to skip if we have scraped the author object in the current scrape or any of our last three scrapes (`skipBlock(Author(author_name.text, author_name.link, h_index.text), currentExecution - 3)`).

2.4 Handling Web Scraping Failures with Skip Blocks

At a high level, skip blocks allow us to handle access failures by restarting the script and skipping over redundant work. In an ideal scenario, the server output after restarting would be the same, and it would be clear which work was redundant and thus how to return to the failure point. Unfortunately, data consistency failures mean the server output may change, making redundancy hard to detect. Thus, the skip block approach handles consistency failures by allowing programmers to define redundancy in a way that works for their target webpages and dataset goals.

To illustrate how skip blocks handle individual access and data consistency failures, we walk through how skip blocks address the four challenges described in Section 1.1.

(1) *Server and network failures.* Upon encountering an access failure, scripts restart at the beginning and use memoization-based fast-forwarding to accelerate failure recovery. With an author skip block in place, the script may assume that when the end of a given author's block has been reached, all his or her papers and all the papers' citations have been collected. Thus, if we encounter this same author during recovery, the script skips over visiting the many pages that display the author's paper list and the one page per paper that displays the paper's citation list. Since page loads are a primary driver of execution time, this has a big impact on performance.
(2) *Alternative data access paths.* Skip blocks allow us to cleanly handle redundant web interfaces, skipping duplicates in DAG-structured data. Say we want to collect all papers by the top 10,000 authors in computer science. With the Google Scholar interface, we can only find this set of papers via the list of authors, which means we may encounter each paper multiple times, once for each coauthor. For our purposes, this web interface produces redundant data. By adding a skip block for the paper entity, we can easily skip over papers that have already been collected from a coauthor's list of papers.
(3) *Non-atomic list snapshots.* Data updates during the scraping process can cause some data to appear more than once. Recall the Craigslist listings that move from the first page of results to the second when new items are added. This crowds the output dataset with unwanted near-duplicates and also slows the scrape. Each time we re-encounter a given listing, we must repeat

the page load for the listing page. For objects with many nested objects (e.g., cities with many restaurants or restaurants with many reviews), the cost of repeats can be even higher. The skip block construct handles this unintentional redundancy. When an old listing is encountered again on a later page, the commit log reveals that it has been scraped already, and the script skips the object.

(4) *Repeated scraping.* Naturally, the reverse of the Craigslist problem can also occur; websites can fail to serve a given relation item. (Imagine the second page of Craigslist results if no new listings were added but a first-page listing was deleted.) Sites generally have good incentives to avoid letting users miss data, but still this particular atomicity failure motivates the need to run multiple scrapes. Rescraping is also desirable for users who are conducting longitudinal collection tasks, or for users who simply want to keep their data snapshot up to date. The rescraping task is complicated by the fact that each type of rescrape should proceed differently. If we are conducting an immediate rescrape to ensure we did not lose Google Scholar authors in a mid-scrape ranking update, we certainly do not want to descend to the level of citations for all the authors already scraped — rather, we want to scrape only any authors we missed in the last execution. If we are rescraping once a week, we are probably most interested in authors who are new to the top 10,000 list, but maybe we also want to refresh data for an author if the last time we scraped him or her was over a year ago. Our skip block staleness constraints allow us to offer both of these options. For the immediate rescrape, we would provide the last scrape's logical timestamp as a staleness constraint. To rescrape all new authors and any author who has not been refreshed in the last year, we would provide the current time minus one year.

3 BACKGROUND: THE HELENA LANGUAGE AND TOOL

We briefly describe Helena [Chasins 2017], the language in which we introduce skip blocks. Helena is not a contribution of this paper, but it enables programming by demonstration (PBD) — as discussed in Section 1.2 — and thus drives the need for a skip block design that end users can embrace. The Helena PBD tool offers a simple programming model:

(1) The user demonstrates how to collect the first row of a relational or multi-relational dataset.
(2) The PBD tool produces a "loopy" program in Helena, a high-level imperative scraping language.

The rest of this section provides a brief overview of these steps and the tool's process for transforming the step one recording to produce the step two program.

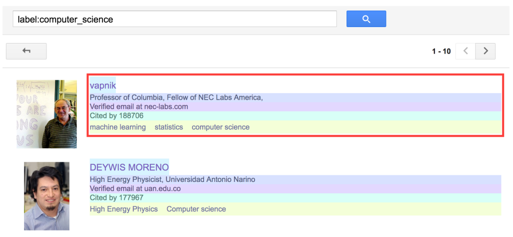
The tool records browser interactions using the Ringer library [Barman *et al.* 2016]. In our running example, the user would first collect information about the first author (circled in red in Figure 3a), then about the first paper of the first author (Figure 3b), then the year of the first citation of the first paper (Figure 3c). From the recording, Ringer produces a straight-line replay script.

With a straight-line Ringer script as input, the Helena PBD tool produces a loopy program with the following steps:

(1) **reverse compilation** from low-level Ringer statements to high-level Helena statements
(2) **identifying relations** within webpages
(3) **inserting loops** to iterate over relations

Reverse Compilation. The PBD tool's first task is to compile in reverse from the low-level Ringer language to Helena's higher-level statements. Ringer programs are verbose, with each statement triggering a single low-level browser event. For example, take the following Ringer statement:

```
waituntil server response matching hostname=='amazon.com' && path=='ajaxv2' && params.id=='bar':
  dispatch action keyup(P) on node matching {type: 'SPAN', ...}
```



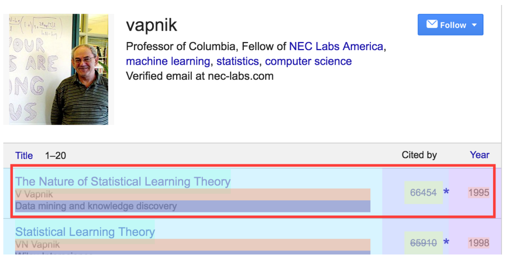
label:computer_science

1 - 10

vapnik
Professor of Columbia, Fellow of NEC Labs America,
Verified email at nec-labs.com
Cited by 188706
machine learning statistics computer science

DEYWIS MORENO
High Energy Physicist, Universidad Antonio Narino
Verified email at uan.edu.co
Cited by 177967
High Energy Physics Computer science

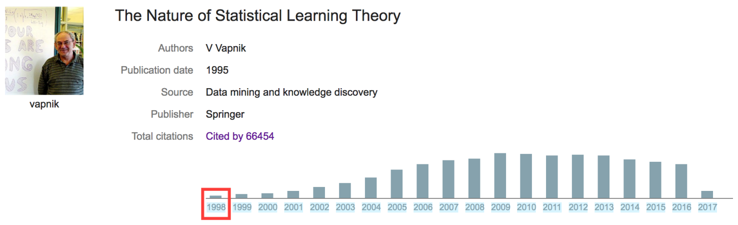
(a) The first page of the **authors** relation.



vapnik
Professor of Columbia, Fellow of NEC Labs America,
machine learning, statistics, computer science
Verified email at nec-labs.com

Title 1–20	Cited by	Year
The Nature of Statistical Learning Theory V Vapnik Data mining and knowledge discovery	66454 *	1995
Statistical Learning Theory VN Vapnik	65910 *	1998

(b) The first page of the first author's **papers** relation.



vapnik

The Nature of Statistical Learning Theory

Authors	V Vapnik
Publication date	1995
Source	Data mining and knowledge discovery
Publisher	Springer
Total citations	Cited by 66454

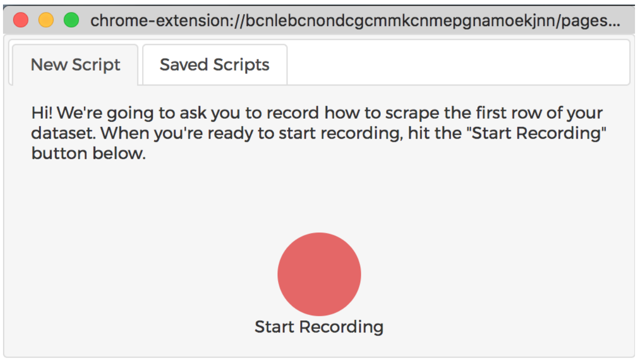
(c) The **citations** relation of the first paper of the first author.

Fig. 3. The webpages on which the user demonstrates how to scrape the first row of output data for our running example. Figure 3a depicts the `authors` relation; Figure 3b depicts a `papers` relation; Figure 3c depicts a `citations` relation. Refer to Figure 4b to see the cells of the first row of output data.

Informally, this statement means "As soon as a server response from amazon.com with the provided path properties has arrived, dispatch the DOM event 'keyup(P)' on the page node that best matches the provided feature map." If a user types the string "PBD", the Ringer replay script would require 12 such statements, and this simple statement already elides much of the detail of the node feature map, which typically has hundreds of entries. This same 12-statement Ringer program is represented in the Helena interface as `type(pageVariable, "PBD")`. To produce this more compact representation, the PBD tool slices the Ringer script according to which sequential statements can be represented with a single summary Helena statement. For this process, the Helena tool analyzes which Ringer statements can be hidden from the user, which adjacent statements interact with the same DOM node, and which perform related actions. The end result is a mapping from slices of Ringer statements to individual Helena statements. The Ringer statements that correspond to each Helena statement are retained as the executable representation of the Helena statement.

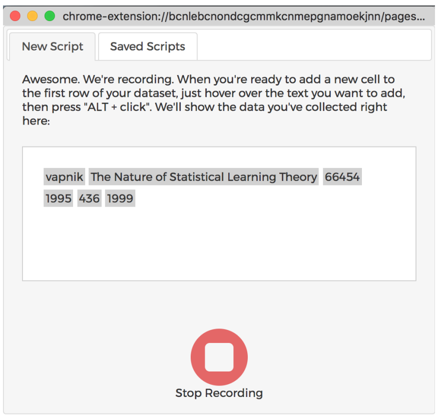
Identifying Relations. Next, the tool analyzes all logically distinct webpages in the recording to discover relations for which an interaction should be repeated. For instance, if the user collected the name and institution of the first Google Scholar author, the tool would assume that those nodes represent the first row of an author relation on the page. The tool feeds those first two cells and the webpage to a custom relation extractor, which produces a small program that extracts the full author relation from the page and other pages generated from the same web template. Repeating this process for each webpage produces the relations highlighted in Figure 3: authors, papers, citation years.

Inserting Loops. With a straight-line Helena script and a set of relations, the PBD tool can generalize to a loopy program. The tool identifies every use of a relation cell in the straight-line script. The use of a relation's cells suggests the user may want to use not just one row, but all


chrome-extension://bcnlebcnondcgcmmkcnmepgnamoekjnn/pages...

Hi! We're going to ask you to record how to scrape the first row of your dataset. When you're ready to start recording, hit the "Start Recording" button below.

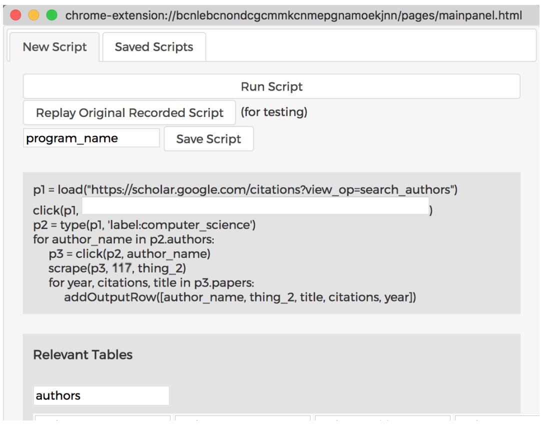
(a) The first page of the user interface instructs the user to record the steps that scrape the first row of the target output dataset.


chrome-extension://bcnlebcnondcgcmmkcnmepgnamoekjnn/pages...

Awesome. We're recording. When you're ready to add a new cell to the first row of your dataset, just hover over the text you want to add, then press "ALT + click". We'll show the data you've collected right here:

vapnik	The Nature of Statistical Learning Theory	66454
1995	436	1999

(b) When recording starts, the UI tells users how to scrape data and shows all cells scraped so far (a preview of the first output row).


chrome-extension://bcnlebcnondcgcmmkcnmepgnamoekjnn/pages/mainpanel.html

```
p1 = load("https://scholar.google.com/citations?view_op=search_authors")
click(p1,                                        )
p2 = type(p1, 'label:computer_science')
for author_name in p2.authors:
    p3 = click(p2, author_name)
    scrape(p3, 117, thing_2)
    for year, citations, title in p3.papers:
        addOutputRow([author_name, thing_2, title, citations, year])
```

(c) After the user completes the recording, the tool identifies relations within the recorded webpages, inserting a loop for each relation. The tool displays the resulting scraping program.

Fig. 4. The UI for the Helena PBD tool, throughout the stages of writing our running example.

rows of the relation. For each relation, the tool inserts a loop into the replay script to iterate over the relation rows. The loop body starts at the first use of the relation's cells and ends after the last use of the relation's cells. Next, the tool replaces each use of a relation cell with a use of the corresponding loop variable, turning a statement such as `click(<image of first author name>)` into `click(author_name)`. Finally, to add scraped data to the output dataset, the tool inserts an output statement at the end of the innermost loop.

The UI in Figure 4 walks the user through the demonstration process and displays the output Helena program at the end. This output program is complete and ready to run. Users may run the output program unchanged, or they may add skip blocks.

$$\frac{V_c(x_1) = \overline{a_1} \quad \dots \quad V_c(x_n) = \overline{a_n}}{\begin{array}{c}\langle skipBlock(x(\overline{a}), x_1 \dots x_n)\{\overline{s}\}, V_c, V_p, D_c, D_p, \overline{x_c}\rangle \rightarrow \\ \langle skipBlockHelper(x(\overline{a}\,\overline{a_1} \dots \overline{a_n}))\{\overline{s}\}, V_c[x := \overline{a}], V_p, D_c, D_p, \overline{x_c}\rangle\end{array}} \ \textsc{CombineVectors}$$

$$\frac{\overline{a} \in V_p[x]}{\langle skipBlockHelper(x(\overline{a}))\{\overline{s}\}, V_c, V_p, D_c, D_p, \overline{x_c}\rangle \rightarrow \langle skip, V_c[x := null], V_p, D_c, D_p, \overline{x_c}\rangle} \ \textsc{AlreadySeenVector}$$

$$\frac{\overline{a} \notin V_p[x]}{\langle skipBlockHelper(x(\overline{a}))\{\overline{s}\}, V_c, V_p, D_c, D_p, \overline{x_c}\rangle \rightarrow \langle \overline{s}; commit(x), V_c, V_p, D_c, D_p, \overline{x_c} + [x]\rangle} \ \textsc{NewVector}$$

$$\frac{V_c(x) = \overline{a} \quad V_p(x) = A \quad D_c(x) = A_d}{\langle commit(x), V_c, V_p, D_c, D_p, \overline{x_c}\rangle \rightarrow \langle skip, V_c[x := null], V_p[x := (A + \{\overline{a}\})], D_c[x := []], D_p A_d, \overline{x_c} - [x]\rangle} \ \textsc{Commit}$$

$$\frac{}{\langle addOutputRow(\overline{a}), V_c, V_p, D_c, D_p, []\rangle \rightarrow \langle skip, V_c, V_p, D_c, D_p[\overline{a}], []\rangle} \ \textsc{OutputNormal}$$

$$\frac{\overline{x_c}.last = x_l \quad D_c(x_l) = A_d}{\langle addOutputRow(\overline{a}), V_c, V_p, D_c, D_p, \overline{x_c}\rangle \rightarrow \langle skip, V_c, V_p, D_c[x_l := A_d[\overline{a}]], D_p, \overline{x_c}\rangle} \ \textsc{OutputInSkipBlock}$$

Fig. 5. The semantics of the skip block construct, and the effect of skip block on the `addOutputRow` statement.

4 SEMANTICS

This section presents the semantics of the skip block construct that Helena uses for detecting duplicate entities. First we describe the semantics without the details of staleness constraints. Next we show how the skip block rules change when we include staleness constraints.

Each skip block annotation takes the form:

$$skipBlock(annotationName(attributes), ancestorAnnotations)\{statements\}$$

A sample annotation might look like this:
```
skipBlock(Paper(title.text, year.text), Author){ ... }
```
Figure 5 contains the semantics of the skip block construct. a ranges over node attributes. x ranges over entity scope variables. s ranges over statements. V_c is the store of currently active skip block vectors (a map $x \longrightarrow \overline{a}$ from variable names to attribute vectors). V_p is the permanent store of completed skip block vectors (a map $x \longrightarrow A$ from variable names to sets of attribute vectors). D_c is the store of output data staged to be added by the current skip block. D_p is the permanent store of the output data.

The CombineVectors rule handles skip blocks that depend on ancestor skip blocks. For instance, our sample skip block indicates that we will consider a paper a duplicate if it shares the title and publication year of another paper, but only if the Author (a skip block variable introduced by a preceding skip block) is also a duplicate. It concatenates the vector of the new skip block ($\overline{a}$) with the vectors of all ancestor skip block arguments ($\overline{a_1}...\overline{a_n}$) and saves the combined vector as the current vector for the skip block (x).

The AlreadySeenVector and NewVector rules handle the case where the skip block has already been converted into a skipBlockHelper (which does not depend on ancestor skip blocks). The AlreadySeenVector rule handles the case where the input vector is already present in $V_p[x]$, the store of previous x input vectors – that is, the case where the skip block has identified a duplicate. In this case, the body statements ($\overline{s}$) are skipped.

The NewVector rule handles the case where the input vector is not present in $V_p[x]$ and thus x represents a new entity. In this case, we execute the body statements, then a commit statement.

$$\frac{V_c(x) = \overline{a} \quad V_p(x) = A \quad D_c(x) = A_d \quad time.now = t}{\langle commit(x), V_c, V_p, D_c, D_p, \overline{x_c}\rangle \rightarrow \langle skip, V_c[x := null], V_p[x := (A + \{\overline{a}t\})], D_c[x := []], D_pA_d, \overline{x_c} - [x]\rangle}\ \textsc{CommitT}$$

$$\frac{\overline{a}t \in V_p[x] \quad t \geq t_l}{\langle skipBlockHelper(x(\overline{a}), t_l)\{\overline{s}\}, V_c, V_p, D_c, D_p, \overline{x_c}\rangle \rightarrow \langle skip, V_c[x := null], V_p, D_c, D_p, \overline{x_c}\rangle}\ \textsc{AlreadySeenVectorT}$$

$$\frac{\nexists\, \overline{a}t \in V_p[x].t \geq t_l}{\langle skipBlockHelper(x(\overline{a}), t_l)\{\overline{s}\}, V_c, V_p, D_c, D_p, \overline{x_c}\rangle \rightarrow \langle \overline{s}; commit(x), V_c, V_p, D_c, D_p, \overline{x_c} + [x]\rangle}\ \textsc{NewVectorT}$$

Fig. 6. The semantics of the skip block construct extended to handle staleness constraints.

The Commit rule saves the current x vector ($\overline{a}$) into the long-term store of x vectors. It also saves all dataset rows produced during the current skip block ($D_c(x_l)$) into the permanent dataset store (D_p).

OutputNormal and OutputInSkipBlock indicate that an output statement outside of any skip blocks (in the situation where the current skip block environment is empty) adds the new output row directly to the permanent dataset store (D_p), while an output statement within a skip block adds the new output row to the dataset buffer associated with the current skip block ($D_c(x_l)$).

Note that descendant `commits` can succeed even if ancestor `commits` fail. That is, we do not wait until all ancestor skip blocks have committed before adding a new skip block vector to V_p or new output data to D_p (see Commit). In general, it is not useful to think of the skip block construct as a transaction construct – it is better to think of it as a form of memoization or a way for the user to communicate which program regions relate to which output data – but the closest analogue from the nested transaction literature is the open nested transaction.

4.1 Staleness Constraints

The Figure 5 semantics elide the details of staleness constraints, making the assumption that users are willing to accept any match in the commit log. Here we describe the simple extensions required to support staleness constraints. Figure 6 shows the rules that must be altered to support staleness constraints, with t ranging over timestamps.

First, we adjust the commit rule to store a timestamp with each commit log record. CommitT stores $\{\overline{a}t\}$, not just $\{\overline{a}\}$, into V_p. AlreadySeenVectorT indicates that to skip a block, we must now find a commit log record $\overline{a}t \in V_p[x]$ such that the associated timestamp t is greater than or equal to the staleness constraint t_l provided to the skip block. NewVectorT indicates that to descend into a block, there must be no commit log record $\overline{a}t \in V_p[x]$ such that the associated timestamp t is greater than or equal to the staleness constraint t_l.

Although these semantics make reference to only a single timestamp, in practice we make the same extension for both physical and logical timestamps. A logical timestamp i associated with a commit log record indicates that the associated block was executed in the ith run of the script.

Programmers can also use a few special timestamps. By default, an empty timestamp is interpreted following the semantics in the previous subsection. That is, it corresponds to using the timestamp version with timestamp $-\infty$, skipping if the block has ever been executed before. Programmers can also use a special argument to indicate that all blocks should be executed, completely ignoring the commit log. This corresponds to using timestamp ∞.

5 ALTERNATIVE DESIGNS FOR FAILURE RECOVERY

We discuss two apparently natural but ultimately fragile alternatives to our duplicate-based recovery strategy.

5.1 URL-Based Recovery

Motivating question: Why not reload the URL that failed and start again there?

Although many sites store all necessary state information in the URL, some sites do not. We have scraped two large datasets using sites that load hundreds of logically distinct pages without ever changing the URL. We wanted a general-purpose approach to restarts, so this strategy was a bad fit for our goals.

Also, data reordering makes this approach fragile. For instance, say we want to scrape information about the top 10,000 authors in a field, according to a Google Scholar author list. The URL for the author list page includes information about the list position; however, rather than retrieving data by asking for the 400th author in the list, it asks for all authors after the current 399th author's id. If a new batch of data comes in, and author 399 sinks from 399 to 520 in the ranking, loading the same URL causes a big gap in our data. Or the author might rise in the rankings and we might be forced to rescrape many authors.

In other cases, as in our Craigslist example, a given URL returns users not to a particular set of data points but to the data points now at the same position in a list. In short, even for tasks where URLs record information about the state, we would still need a duplicate detection approach for ensuring that we have reached the target data. Given that we need a same-data-check even to use a fragile URL checkpoint that can handle only a subset of scraping tasks, we chose not to center our recovery strategy on this approach.

5.2 Pre-Scraping-Based Recovery

Motivating question: Why not visit all pages of the outermost loop first, collecting the links for all items, then come back and load those links for running inner loops later?

Many webpages do store this information in actual link (anchor) tags, or in other tags that store URLs in a discoverable way. For this reason, we are interested in pursuing this technique as an optimization in future, for the cases that do store discoverable links in the DOM tree. Unfortunately, not all webpages do so. Sometimes this approach is infeasible (or slower than expected) because we need to execute a whole form interaction or some other long webpage interaction for each outer loop iteration before the desired link node is even loaded. Sometimes this is infeasible because no true link tag is *ever* present on the page, but rather a click on some other node – a div, a span – is handled by the page's JavaScript, and the JavaScript generates the target URL. As promising as this approach is for optimizing some tasks, we wanted a more general approach.

This strategy also faces the same issue that hobbles the URL-based approach. Sometimes there is no new top-level URL for loop items. In this case, the URLs we extract for loop items would be URLs for AJAX requests. Since there are often many AJAX requests associated with a given page interaction, we would first face the task of identifying which AJAX URL we should store for each loop item; then if the AJAX requests do not produce HTML, we would have to learn to parse the websites' JSON, CSV, or custom proprietary representations, rather than letting the pages' own JavaScript handle this for us.

Further, even this approach cannot completely eliminate data updating issues. Some sites get so much fresh data so often that we would still need a way to handle duplicates even with this approach.

6 IMPLEMENTATION

We discuss the details of our Helena implementation of skip blocks and briefly describe how skip blocks could be implemented in a more traditional language.

6.1 Implementing Skip Blocks in Helena

The Helena language and its associated programming environment are implemented as a Chrome extension. Although scraping is executed on the client side, the scraped data is stored on a centralized server.

To extend Helena with skip blocks, we added an additional statement to the language and an additional component to the troubleshooting section of the Chrome extension's UI. For each loop in the program, this new UI component shows the first row of the relation associated with the loop; the first row is always observed during the demonstration phase, so this information is always available. The user can highlight columns of a relation – the key attributes of the object – then click 'Add Annotation' to add a skip block to the program. The tool identifies the first program point at which all key attributes are available and inserts the new skip block at that point. By default, the skip block body includes all of the remaining statements in the body of the relation's associated loop. This placement is sufficient for our benchmarks, but the user is also permitted to adjust the boundaries of the skip blocks.

Aside from the UI design, the primary decisions that shaped our Helena skip block implementation were the choices of commit log location and format. We chose to store the commit log on a remote server along with the existing Helena data storage. The log is a table in a relational database, which makes it easy to query the log efficiently.

The full extension, including the skip block implementation, is available at https://github.com/schasins/helena.

6.2 Alternative Implementations of Skip Blocks

Although we have focused on using skip blocks to improve the PBD scraping experience, skip blocks can handle the same classes of problems in hand-written scraping scripts. Many modern

```
 1  from selenium import webdriver
 2  driver = webdriver.Chrome("./chromedriver")

 4  # we elide most details of scraping this data; assume here we accumulate a list of author nodes

 6  for i in range(len(author_nodes)):
 7      author_name = author_nodes[i].name
 8      author_url = author_nodes[i].url
 9      def body():
10          driver.get(author_url)
11          paper_rows = None
12          while True:
13              old_paper_rows = paper_rows
14              paper_rows = driver.find_elements_by_xpath('//*[@id="gsc_a_b"]/tr')
15              # stopping criterion; else script keeps clicking a grayed-out next button when paper list ends
16              if old_paper_rows == paper_rows:
17                  break
18              for row in paper_rows:
19                  title = row.find_element_by_class_name("gsc_a_t").find_element_by_tag_name("a").text
20                  year = row.find_element_by_class_name("gsc_a_y").text
21                  print author_name, title, year
22              next_button = driver.find_element_by_xpath('//*[@id="gsc_bpf_next"]/span/span[2]')
23              next_button.click()
24      skipBlock("Author", [author_name, author_url], [], body)
```

Fig. 7. Part of a Python and Selenium script for scraping our running example, the Google Scholar dataset. Line 24 shows how we might use a library implementation of the skip block construct, by passing as arguments a list of key attributes and a function that encapsulates the body of the skip block.

scraping languages are implemented as libraries for general-purpose languages, which makes it easy to extend them with skip blocks. For instance, Scrapy [Scrapy 2013], Beautiful Soup [Richardson 2016], and Selenium [Selenium 2013] can all be used as Python libraries. To introduce skip blocks to these languages, we could write our own Python library. Figure 7 shows part of a Python script that uses Selenium to scrape our Google Scholar running example. We have modified the real script so that rather than scraping an author's papers directly, it wraps some of the scraping code in a `body` function; along with `[author_name, author_url]` (the key attributes of the author object), the program passes the `body` function as an argument to a `skipBlock` function. To implement the `skipBlock` function, we would simply replicate in Python the functionality that Helena implements in JavaScript. Although using a remote database for storing the commit log is still an option for this implementation, the fact that the target user is a Python programmer means that a local database is a reasonable implementation choice.

Naturally, Python is not the only alternative host language for skip blocks. This same library-based approach extends to other languages with higher-order functions. For languages without higher-order functions, a deeper embedding (like our Helena approach) is likely a better fit.

7 BENCHMARKS

Our benchmark suite consists of seven real scraping tasks requested by scientists and social scientists from a range of fields. For each benchmark, the requester described the target data and how to navigate the website to acquire it. In most cases, we wrote and ran the Helena script (using Helena's PBD tool); in two cases, the requester or a member of the requester's team wrote and ran the script. In all cases, the original requesters have verified that the output data is correct and complete. To collect the benchmark suite, we asked a non-author colleague to put out a call for researchers who needed data from the web. The call went out to a mailing list of researchers from many different disciplines who all study some aspect of urban life.

Because the colleague who wrote and sent the call did not know the inner workings of our tool, we believe the benchmarks solicited from this call should not be biased towards tasks that flatter our language. This also means that each benchmark will not neatly exhibit a unique illustrative situation in the problem space. Rather, we observe the messy sets of problems that cooccur in real scraping tasks.

Table 2 describes each scraping task. Although our language and tool do not require that all pages in a given script be from a single website, each requested dataset used pages from only a single site. Table 2 includes this website, a description of the output dataset, and the discipline of the researcher who requested the data.

Table 3 shows a summary of our benchmarks according to whether they exhibit data duplication. We break data duplication down into two forms: unintentional and intentional. Unintentional duplication means that the website backend (which has a perfect snapshot of the data) never includes multiple instances of one data point, but the client-side view of the data does. In contrast, intentional duplication means that even with a perfect snapshot of the underlying data, the client-facing representation of the data would include multiple instances of at least one data point. For intentional duplication, we can be even more specific about who intends the duplication. In some cases, the website is designed to produce duplicates; for instance, the Zimride carpool website is designed so that a given carpool listing is associated with many rides, so the stream of upcoming rides points back to the same listing many times. In some cases, the website does not intentionally duplicate data, but users choose to post duplicate data; for instance, many users post the same apartment listings to Craigslist repeatedly. In the latter case, the website typically lacks awareness

Table 2. For each benchmark, we record the website of the pages that contain the target data, a description of the target data, and the discipline of the researcher who requested the data.

Dataset	Website	Data	Requester Field
Apartment listings	Craigslist	For all Seattle Craigslist apartment listings, the price, size, and other features of the apartment.	Sociology
Carpool listings	Zimride	For all rides in the Zimride University of Washington carpool ride list, the carpool listing features including start and end locations and times.	Transportation Engineering
Charitable foundation tweets	Twitter	For each foundation in a list of the top charitable foundations, the 1,000 most recent Twitter posts.	Public Policy
Community foundation features	Community Foundation Atlas	For all community foundation ons in the Community Foundation Atlas, a large set of foundation features.	Public Policy
Menu items	Yelp	For top Seattle restaurants according to Yelp, all items on the menu.	Economics
Restaurant features	Yelp	For top Seattle restaurants according to Yelp, a set of restaurant features.	Economics
Restaurant reviews	Yelp	For the top Seattle restaurants according to Yelp, all reviews.	Economics

Table 3. We record whether each type of duplication could be observed by each benchmark script, based on knowledge of the site's design. We also record whether each type of duplication was actually observed in script runs reported in our evaluation.

Dataset	Can/Did exhibit unintended duplication	Can/Did exhibit site-intended duplication	Can/Did exhibit user-intended duplication
Apartment listings	Y/Y	N/N	Y/Y
Carpool listings	N/N	Y/Y	Y/Y
Charitable foundation tweets	N/N	Y/Y	Y/Y
Community foundation features	N/N	N/N	N/N
Menu items	Y/Y	Y/Y	Y/Y
Restaurant features	Y/Y	N/N	Y/N
Restaurant reviews	Y/Y	N/N	Y/Y

of the duplication – a different id or url may be associated with each item – but a human observer would likely consider apartment listings or restaurant reviews that share text to be duplicates.

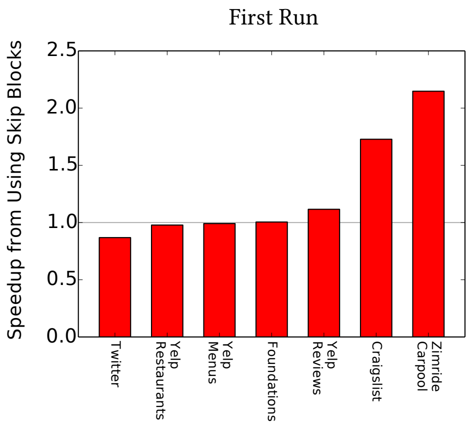
Table 3 reports both whether a form of duplication is possible based on the website design and also whether we observe the duplication in the benchmark runs reported in our evaluation.

8 PERFORMANCE EVALUATION

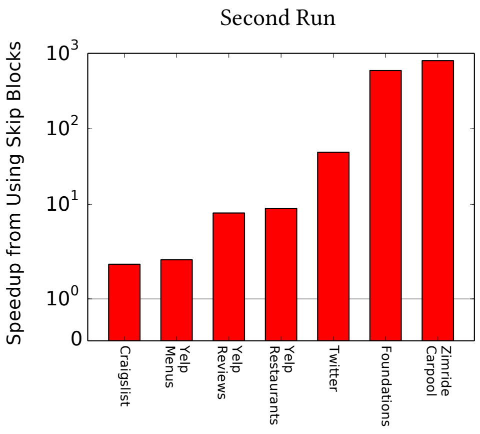
We evaluate the effect of skip blocks on performance in a single-run context, a multi-run context, and a failure recovery context. We find that skip blocks improve performance on unstable datasets in a single-run context, on stable datasets in a multi-run context, and on most datasets in a recovery context.

8.1 Single-Run Performance

Q1: What are the performance costs and benefits of duplicate detection in a single script execution?
H1: For most tasks, duplicate detection will impose a moderate performance overhead. For tasks that involve iterating over live data or over data with intentional redundancy, it will confer a substantial performance benefit.



(a) The speedup associated with using the skip block construct for the first run of each benchmark script.



(b) The speedup associated with using skip block for a second run of each benchmark script one week later. Note the log scale y-axis.

Fig. 8. The speedup associated with using the skip block construct for the first and second runs of benchmark scripts.

In practice, this means that tasks with relatively stable data (or pagination approaches that hide instability) should see little effect, while benchmarks with data that updates quickly should show substantial improvements. Even in the case of stable data, we predict performance improvements if the web interaction is designed to display a particular object multiple times.

8.1.1 Procedure. For each benchmark, we conducted two simultaneous runs of the script, one that used the skip block construct and one that did not.

8.1.2 Results. Figure 8a shows the results of the single-run experiment. Speedups vary from 0.868405x to 2.14739x, with a median of 1.00453x.

We give a sense of the factors at play by describing the best and worst cases for duplicate detection. At the far right, we see the best performer, the carpool listing task. Zimride allows users to post listings requesting and offering carpool rides. Each listing may offer multiple rides. For instance, a listing may repeat each week, every Monday, Wednesday, and Friday. Since many users list commutes, this is common. The Zimride web interface however, rather than offering a stream of listings, offers a stream of upcoming rides. Thus, we may see our hypothetical MWF listing three times in the first week of rides, once for the Monday ride, once for the Wednesday ride, and once for the Friday ride. However, all rides would point back to the same listing and thus be recognized as duplicates. In this case, approximately half of the listed rides were duplicates. Each time we recognize a duplicate, we avoid a network request. Since network delay accounts for most of the scraping time, the cumulative effect is to offer about a 2x speedup.

The Craigslist case is similar, except that most of the duplication comes not from intentionally displaying items at multiple points in the stream, but simply from the high volume of new listings being entered, and the fact that Craigslist's pagination approach allows new listings to push old listings back into view.

On the other side of Figure 8a, we see the Twitter script offering a good example of a worst-case scenario for online duplicate detection. For the Twitter script, detecting a duplicated tweet actually has no performance benefit. By the time we have the information necessary to check whether a tweet is a duplicate, we also have all the information necessary to scrape it. So the first-run version of this benchmark shows a case where we see all the overhead of duplicate detection with absolutely no benefit, since no important work can be skipped even if we do find a duplicate. Recall that the commit log is stored remotely, so communicating with the server to check for a duplicate before proceeding with execution can incur a substantial overhead. Future versions of the tool could certainly optimize this aspect of the interaction.

8.2 Multi-Run Performance

Q2: What are the performance costs and benefits of duplicate detection in a multi-execution context?
H2: For tasks with highly unstable data, duplicate detection will impose a moderate performance overhead. For tasks with fairly stable data, it will confer a substantial performance benefit.

In short, we predict tasks that benefit from duplicate detection in the single-execution case will not gain much additional benefit from tracking entities already scraped in previous executions. However, the highly stable datasets that receive little benefit from duplicate detection during a single run should see substantial performance improvements in later runs.

8.2.1 Procedure. For each benchmark, we conducted a single run using the skip block construct. After a week, we conducted two simultaneous runs of the benchmark, one that did not use the skip block construct, and one that did.

8.2.2 Results. Figure 8b shows the results of the single-run experiment. Speedups vary from 1.82723x to 799.952x, with a median of 8.81365x.

Here we see how the Twitter script, the least performant of the first run, is a star of the multiple-runs scenario with a speedup of 49x. This script was designed for scraping new tweets as they come in over time, so there is no skipBlock for Twitter accounts – we want to scrape every account every time – but the script is configured to break out of the 'for each tweet on page1' loop as soon as it has seen 30 duplicate tweets in a row. The script scrapes up to the first 1,000 new tweets, so since most foundations tweet much less than that in a week, the duplicate-detecting script can stop after scraping only a tiny portion of what the non-duplicate-detecting script must scrape. Further, that tiny portion consists of only the most recent tweets, while the top 1,000 includes many fairly old tweets. Since Twitter loads new tweets in a given stream much, much faster than older tweets, this gives the duplicate detection version an additional advantage.

At the far left, the Craigslist speedup is about the same for the multi-run case as for the first run case. It barely benefits from records committed during the previous scrape but continues to benefit from records committed during a single scrape. This partially supports the first half of our hypothesis, that we will see less improvement for benchmarks with frequently updated data, although we were wrong to predict a slowdown relative to the non-skip block version.

At the far right, we see Zimride with a speedup of almost 800x. This improvement reflects the fact that only one new listing was added to the Zimride site during the course of the week. (Recall that the task was to scrape carpool listings associated with a single school, and there was no semester change during the week, so this is unsurprising.) The same general explanation applies for the Foundations benchmark, which only added data for one new foundation between the two scrapes.

8.3 Failure Recovery Performance

Q3: How does our duplicate-based recovery strategy affect execution time compared to a human-written strategy that can be customized to a given script?
H3: In most cases, our duplicate-based recovery strategy will make the scrape slightly slower than the custom solutions that a programmer might write. In cases where starting from the beginning

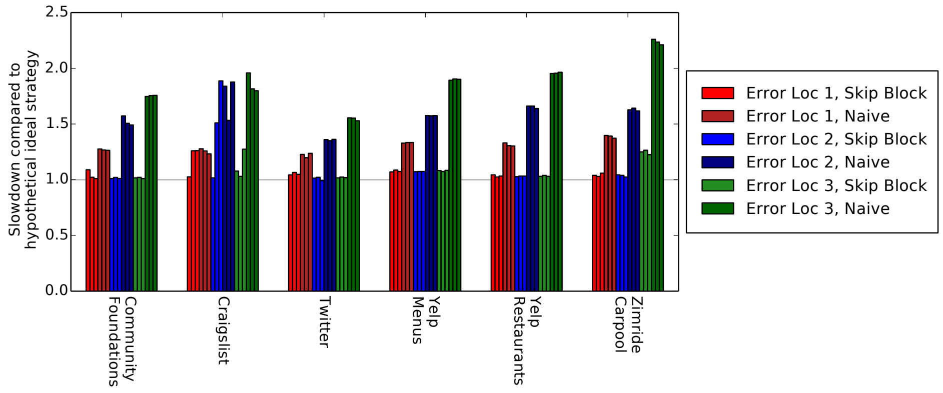


Fig. 9. The results of the recovery strategy experiment. Each bar represents the time to complete one run, normalized by the best run time achieved by the Ideal strategy.

causes the script to encounter new data, duplicate-based recovery will be much slower than custom solutions.

8.3.1 Procedure. To evaluate the efficacy of duplicate-based recovery, we performed a limit study. We first selected three error locations for each benchmark. We selected the points at which the script should have collected one quarter, one half, and three quarters of the final dataset. For a given program point, we simultaneously ran three script variations, then simulated a failure at the target point. The three script variations are:

- A naive script: at the point of failure, this script begins again at the beginning.
- A script using skip block-based recovery: at the point of failure, this script begins at the beginning but uses the store of already-seen entities to return to the correct state.
- An ideal custom script: an imagined custom solution that can return to the correct state in zero time; in practice, we simply allow this script to ignore the simulated failure.

For each benchmark, for each error location point, we repeated this process 3 times.

Naturally, the ideal custom strategy that recovers in zero time is impossible, but the zero-time restriction is not the only restriction that makes this ideal unattainable. For some scripts in our benchmark suite, there are no constant-time recovery strategies available. We make the zero-time assumption simply to make the ideal script into the harshest possible competition. Again, this is a limit study, intended to find an upper bound on how much slower skip block recovery is than a manual, expert-written recovery strategy.

For this experiment, we used benchmark variations that collect smaller datasets than the full benchmarks used for the prior experiments. The variations were designed (based on knowledge of the sites) to avoid encountering intentional or unintentional duplicates during a single execution. This allows us to disentangle the performance effects of single-execution data duplication from the effects of data updates on recovery time. As our results show, this design does *not* shelter the recovery executions from data updates.

8.3.2 Results. Figure 9 shows the results of the recovery experiment. All benchmarks except Craigslist exhibit the pattern predicted by the first part of our hypothesis. The skip-based recovery approach takes slightly more time than our hypothetical ideal. Meanwhile, the Naive approach takes about 1/4 of the ideal's time to recover from a failure at error location 1, about 1/2 of the

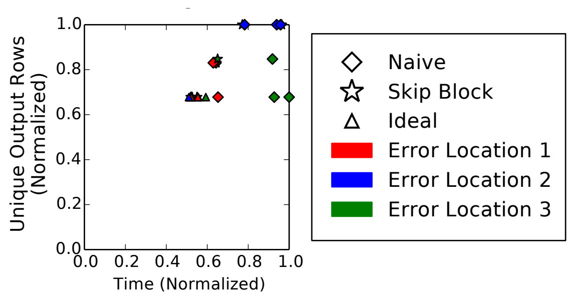


Fig. 10. Amount of unique data collected vs. execution time for each run of the Craigslist benchmark in the recovery experiment, including runs with the hypothetical ideal recovery strategy. This reveals that Skip Block takes substantially longer than Ideal only when it collects more data.

ideal's time to recover from a failure at location 2, and 3/4 of the ideal's time to recover from a failure at location 3.

The results of the Craigslist benchmark demonstrate the second half of our hypothesis, that restarting will be slower in cases where restarting from the beginning causes the script to encounter new data – that is, not only the old data already visited by the pre-error scrape. A close look at Figure 9 reveals that several runs of the duplicate-based strategy produce the same nearly ideal behavior we see for other benchmarks. Others take longer; these represent the runs in which the listing relation was refreshed between the beginning of the initial scrape and beginning of the recovery scrape. If all data from that point forward is new data, duplicate-based recovery performs exactly as the Naive approach performs. If only some of the data is new data, duplicate-based recovery still outperforms Naive.

Figure 10 gives us more context for analyzing the recovery performance for the 'Craigslist' task. It reveals that in the presence of new data, duplicate-based recovery scrapes as much data as the Naive approach, but that in the absence of new data, it performs (almost) as well as Ideal. In short, even when duplicate detection makes for slow recovery, it comes with the compensating feature of additional data.

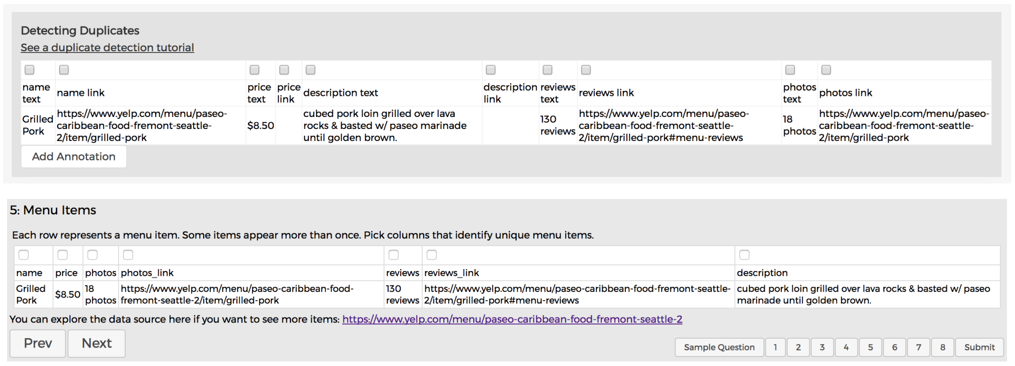
Runs that use duplicate-based recovery take 1.06658x the time of runs that use the hypothetical ideal recovery strategy. This is approximately the overhead we should expect for recovering from an error halfway through execution. Further, duplicate-based fast-forwarding is 7.95112x faster than normal execution.

9 USABILITY EVALUATION

9.1 Writing Skip Blocks Without Tool Support

Q4: Given just the information that a user will have upon first demonstrating a script, can users identify what set of features uniquely identifies an entity?
H4: We expect users will do well overall, but miss some corner cases.

9.1.1 Procedure. For each benchmark in our benchmark suite, we collected the attributes of the first object observed by each skip block construct. For instance, for the Yelp Menu Items benchmark, this is the first restaurant (for the restaurant block) and the first menu item of the first restaurant (for the menu item block). This is the same information that a user is guaranteed to have during script demonstration, and is in fact the same information that we show in the tool's current UI for adding new skip blocks. We constructed a survey in which users were asked to select the set of columns (the object attributes) that can be used to uniquely identify a given object based on this first row of data. For most tasks, they were also given links to the source webpages which they could use to observe more instances of the entity. See Figure 11 for a side-by-side comparison of the real tool UI and the survey question. We made the survey available as an online survey and


Detecting Duplicates

See a duplicate detection tutorial

name text	name link	price text	price link	description text	description link	reviews text	reviews link	photos text	photos link
Grilled Pork	https://www.yelp.com/menu/paseo-caribbean-food-fremont-seattle-2/item/grilled-pork	$8.50		cubed pork loin grilled over lava rocks & basted w/ paseo marinade until golden brown.		130 reviews	https://www.yelp.com/menu/paseo-caribbean-food-fremont-seattle-2/item/grilled-pork#menu-reviews	18 photos	https://www.yelp.com/menu/paseo-caribbean-food-fremont-seattle-2/item/grilled-pork

5: Menu Items

Each row represents a menu item. Some items appear more than once. Pick columns that identify unique menu items.

name	price	photos	photos_link	reviews	reviews_link	description
Grilled Pork	$8.50	18 photos	https://www.yelp.com/menu/paseo-caribbean-food-fremont-seattle-2/item/grilled-pork	130 reviews	https://www.yelp.com/menu/paseo-caribbean-food-fremont-seattle-2/item/grilled-pork#menu-reviews	cubed pork loin grilled over lava rocks & basted w/ paseo marinade until golden brown.

You can explore the data source here if you want to see more items: https://www.yelp.com/menu/paseo-caribbean-food-fremont-seattle-2

Fig. 11. The UI for adding a new skip block in the real tool (top) and the same task in the survey for our user study (bottom).

recruited participants through social networks and mailing lists. Participants were not compensated for their time. We recruited 35 participants (16 non-programmers, 19 programmers).

Before releasing the survey, we selected ground truths. For each entity, we chose *two* ground truths – two sets of columns that could serve as the 'correct' unique identifier. Correctness in this context is somewhat fuzzy in that one user may want to rescrape a tweet each time it has a new number of retweets, while another user may only be interested in the text, which does not change over time. Since giving participants a single sharp, unambiguous notion of entry equality for each task essentially boils down to revealing the target solution, we had to allow them to supply their own definitions of duplicates. Directions could not be more specific than 'pick columns that identify unique <entities >.'

Given the inherent fuzziness, there is no one correct answer, or even two correct answers. Acknowledging this, we elected to use two ground truths that we felt would cover two important interpretations of the duplicate idea. The first set of ground truths, the Site-Based Conservative Standard, attempts to use the websites' own notions of uniqueness to identify duplicates; if a site provided a unique id, or a URL that includes a unique id, this ground truth relies on those attributes, often ignoring similarities that might cause a human observer to classify two rows as duplicates. The second set of ground truths, the Human-Based Aggressive Standard uses the authors' own perceptions of what should qualify as a duplicate, based on having seen the output data. This approach is more aggressive than the Site-Based Standard, collapsing more rows. With the Human-Based Standard, two Craigslist postings with different URLs but the same apartment size, price, and advertisement text are collapsed, and two Yelp reviews with different URLs but the same star rating and review text are collapsed. Importantly, we selected the ground truths before soliciting any survey responses.

For each user-suggested skip block, we identified the rows that the ground truths would remove but the user suggestion leaves, and also the rows that the ground truths would leave but the user suggestion removes.

9.1.2 Results. On average, participants spent 0.93 minutes on each annotation task and 4.0 minutes on the instructions. We are primarily interested in two measures: i) how many rows each user annotation keeps but the Conservative standard discards as duplicates, and ii) how many rows each user annotation discards but the Aggressive standard keeps. We call these keeping and removal errors. For all entities, all participants produced 0 keeping errors relative to the Conservative standard; the average removal error rate ranges from 0.00% (Community Foundations, Twitter Accounts) to 5.8% (Zimride), with a median of 1.3%. See Figure 12 in the next subsection for a visualization of error rates by benchmark.

For a deeper understanding of what these error rates represent, we discuss the most common causes of errors. Since users produce no keeping errors relative to the Conservative standard, we focus on removal errors. Removal errors were most common for Zimride carpool listings and Yelp menu items.

The most serious Zimride error occurred when users collapsed all listings posted by a given username. Essentially, they made the assumption that each user would only post one carpool listing, while in fact a given user may request or offer many different rides – to multiple different places or at multiple different times of day. Surprisingly, even requiring all of [‘carpool_endpoints’, ‘user_name’, ‘carpool_times’, ‘carpool_days’] to match produces some removal errors; this stems from the fact that a few users post the same commute both as a passenger and as a driver. It is worth noting that for the Zimride case, users were at a disadvantage. Because the carpool data came from an institution-specific Zimride site, it was behind a login, and thus this was one of the cases for which we provided no link to the source data. For this reason, participants could not explore additional rows of data within the site.

The most serious Yelp menu errors occurred when participants relied on different menu items having different descriptions, photo links, or review links. On the face of it, this seems reasonable, but in practice many menu items lack all three of these attributes, so all items that lack the attributes are collapsed.

These errors suggest directions for future debugging tools. For instance, one simple debugging tool could show users two objects for which all key attributes match but all other attributes diverge, then ask “did you really mean to treat these objects as duplicates?” This would allow users to quickly identify issues with answers like the Yelp menu photo link mistake. Seeing two rows with different food item names, different prices, and different descriptions collapsed because they lack a photo link would quickly remind users that missing data can subvert their answers. A debugging tool like this is easy to build and easy to use [Mayer *et al.* 2015].

Overall, we are pleased to observe that participants can quickly produce effective skip blocks using the same UI that we offer in the current tool, without additional tool support. We are also encouraged that removal errors are more common than keeping errors, since these are the errors that future tool support can most easily address.

9.2 Programmers vs. Non-Programmers

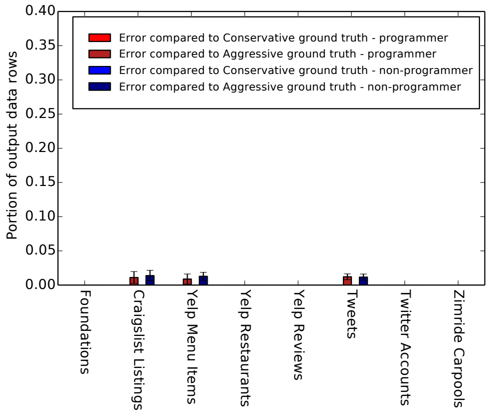
Q5: Do programmers perform better on the attribute selection task than non-programmers?
H5: We expect programmers will perform slightly better than non-programmers on this task.

We expect that programmers will have had more reasons to work with unique identifiers and also that they may have more exposure to web internals and thus be likelier to recognize sites’ own internal unique identifiers in links. With this in mind, we hypothesized that programmers would slightly outperform non-programmers.

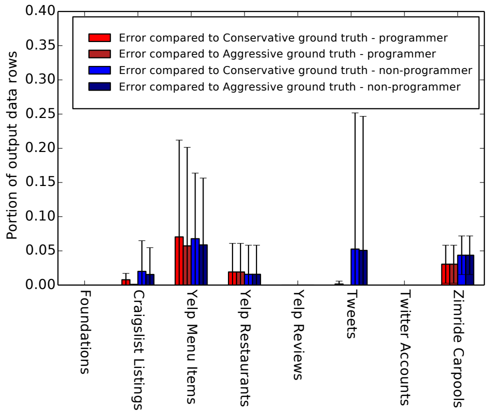
9.2.1 Procedure. Our survey asked participants to indicate whether they are programmers or non-programmers.

9.2.2 Results. Our hypothesis that programmers will perform better than non-programmers is not supported.

Figure 12 shows a comparison of programmer and non-programmer error rates. Overall, we see little difference between programmers and non-programmers, except in the case of the Tweets entity. As it turns out, the high non-programmer bars for the Tweets entity reflect the results of 15 non-programmers with removal error rates under 0.001 and a single non-programmer with an



(a) **Keeping errors** for programmers and non-programmers. When a user answer keeps a row that the ground truth does not keep, this is one keeping error. We see that users do not make keeping errors with respect to the Conservative standard – anything the site recognizes as a duplicate, users also recognize. Relative to the Aggressive standard, users do make errors; this suggests that, as expected, they choose a point between the Conservative and Aggressive standards.



(b) **Removal errors** for programmers and non-programmers. When a user answer removes a row that the ground truth does not remove, this is one removal error. We see that removal errors are more common than keeping errors. Unsurprisingly, users are more aggressive than the Site-based Conservative standard, but it appears they are also slightly more aggressive than our Aggressive Standard.

Fig. 12. A comparison of programmers with non-programmers, based on whether they are more or less aggressive than the Conservative and Aggressive standards.

error rate of 81% – this participant elected to use only the display name and username attributes, apparently interpreting the goal as identifying unique Twitter accounts, rather than unique tweets. (It may be worth noting that this particular user took less than 4 and a half minutes to complete the entire survey, including reading the instructions page and completing all eight annotation tasks.)

The average programmer had a keeping error rate of 0% and a removal error rate of 1.365%. The average non-programmer had a keeping error rate of 0% and a removal error rate of 2.322%. The difference between the two groups is not statistically significant (p=.32 for removal errors).

Programmers and non-programmers also spent similar amounts of time on each annotation task. Programmers spent 0.86 minutes on average, while non-programmers spent 1.02 minutes on average.

Given that non-programmers can produce an average error rate of less than 3% in the context of a 12 minute online survey (and despite lacking familiarity with the data to be scraped), we are satisfied that end users will be able to use our construct for their scraping tasks.

10 RELATED WORK

We have seen few techniques aimed at reducing the effects of data liveness and redundancy on web scraping tasks. The features whose functions most closely resemble the function of our skip block construct are the features intended to improve the performance of longitudinal scraping tasks, so we focus on these. These techniques are often called *incremental scraping* techniques. We classify works according to the skillset demanded of the user.

10.1 Tools for Programming Experts

There are many languages and libraries available for coders who are comfortable writing web automation programs from scratch. Scrapy [Scrapy 2013], BeautifulSoup [Richardson 2016], and Selenium [Selenium 2013] are all excellent tools for various styles of web automation or web scraping, and there are many other less mainstream options [hpricot 2015; Nokogiri 2016]. This category of tools and languages naturally shows the widest variety of incrementalization approaches. Many programmers use custom duplicate detection approaches to implement incremental scraping; many StackOverflow questions ask how to incrementalize individual scraping tasks and many answers suggest approaches that suit a given webpage [StackOverflow 2017]. In general, we are not interested in these custom site-specific solutions, since they offer no guidance for a language like ours that must construct general-purpose incrementalization and recovery approaches with minimal user input.

The most general approach we have seen suggested to coders is the DeltaFetch [scrapy-plugins 2017] plugin for the Scrapy language [Scrapy 2013]. For a subset of data scraping tasks, this offers developers a plug-and-play solution to incrementalizing their scrapers. However, it is not clear that the suitable subset of scraping tasks is large; for reference, only one of our seven benchmarks could be automatically incrementalized with DeltaFetch.

10.2 Tools for DOM Experts

We use 'DOM expert' to refer to users who, although they are not comfortable writing code, understand XPaths, how templates are used to generate webpages, how JavaScript interacts with DOM trees, and other web internals.

There are few tools in this category; it seems tool designers assume that if users can understand the intricacies of the DOM and other web internals, they must be able to or prefer to program. However, we have found one tool that mostly fits into this category, although many of the more advanced features require programming ability. Visual Web Ripper is a tool or visual language targeted at DOM experts [VisualWebRipper 2017]. It offers a "duplicateAction" that can be attached to a page template to indicate that the page template may produce duplicates, and it gives the user the option to remove the duplicate data or to cancel the rest of the scrape. Users also have the option to provide a Java function that can be run to decide whether it is a true duplicate and thus whether to cancel the page template. In addition to the fact that the tool is inaccessible to non-DOM experts, their cancellation options are a much weaker form of control than our skip block; also, the fuller control available with the Java function is only accessible to programmers. This mechanism is certainly related to ours in that it is based on detecting duplicates, but while our skip block approach could be used to implement the options Visual Web Ripper provides, the reverse is not possible.

10.3 Tools for End Users

Prior to our own work, we know of no incrementalization feature targeted at end users. However, this is not primarily because of incrementalization itself, but rather seems to be a function of the web scraping tools that are available to end users. We consider Helena to be a full-featured web automation language, which makes it capable of expressing the kinds of scripts that benefit from incremental scraping. In contrast, most prior end-user scraping tools offer very limited programming models. They typically fall into one of two categories: (i) record and replay tools or (ii) relation extractors.

A web record and replay tool takes as input a recording of a browser interaction and produces as output a script for replaying that same interaction on fresh versions of the pages. Ringer [Barman *et al.* 2016], Selenium Record and Playback [Selenium 2016], CoScripter [Leshed *et al.* 2008], and iMacros [iOpus 2013] all provide this functionality, as do many others [Greasemonkey 2015; Hupp & Miller 2007; Koesnandar *et al.* 2008; Li *et al.* 2010; Mahmud & Lau 2010; Platypus 2013], although some typically require debugging by a programmer or DOM expert and thus might be better treated in the categories above. Since a replayer is only responsible for replaying a single interaction, there is no duplication within a single run, and even incrementalizing could offer only the barest performance benefit, so no current replay tool offers a built-in mechanism for incremental scraping.

A web relation extractor takes as input a webpage and a subset of the webpage's DOM nodes, labeled with their positions in a relation. Its output is a program that extracts a relation of DOM nodes from the webpage. Many tools from the wrapper induction community offer this functionality [Adelberg 1998; Chang *et al.* 2006; Flesca *et al.* 2004; Furche *et al.* 2016; Kushmerick 2000; Kushmerick *et al.* 1997; Muslea *et al.* 1999; Omari *et al.* 2017; Zheng *et al.* 2009]. More recently, FlashExtract [Le & Gulwani 2014], KimonoLabs [KimonoLabs 2016], and import.io [Import.io 2016] have offered highly accessible tools with this functionality. This form limits these tools to scraping all cells in a given row from a single page. Since preventing unnecessary page loads is the primary mechanism by which incrementalization improves performance, there is little incentive for relation extractors to offer incremental scraping.

Although both replay and relation extraction are natural building blocks for end-user-friendly languages, they are not themselves full-fledged web automation languages. None of the tools listed above can scrape any of the datasets in our benchmark suite. So it is natural that they have not had to tackle incremental scraping.

Of all the fully end-user-accessible web programming tools we know, only one brings together both replay and relation extraction. This is a beautiful tool called Vegemite [Lin *et al.* 2009]. Vegemite is essentially a tool for extending an existing relation with additional columns. The authors use the example of having a table of houses and their addresses, then pasting the address into a walk score calculator and adding the walk score as a new column. This is a somewhat restrictive programming model, since all programs must share a single simple structure. Of the seven benchmarks in our suite, only the Community Foundations Atlas script can be expressed with this program structure. So again, this falls short of being a full-scale web programming language, but it certainly brings us closer. The creators of the Vegemite tool never chose to tackle incrementality, presumably because the user was assumed to own the input table and to have done any duplicate detection as a pre-processing stage. Again, in this context, there is little benefit to online duplicate detection.

11 CONCLUSION

This paper introduces a novel construct for hierarchical memoization of web scraping executions with open transaction semantics that is accessible to end users. Our skip blocks improve web script performance by posing users a simple question that, as our user study reveals, end users can answer well and quickly. With recovery speedups of 7.9x, the average user with a poor network connection can run much longer-running scripts than they could feasibly run without skip blocks. Scripts collect more data in less time, and they are less derailed by the inevitable server and network failures. This language feature brings larger datasets in range and makes recovery fast enough that users can easily interrupt long-running scripts when it comes time to close the laptop and head home. Combined with the benefits for handling live data and redundant web interfaces, and given that it offers incremental data scraping by default, we see this as an important step towards developing a highly usable web automation language.

ACKNOWLEDGEMENTS

This work is supported in part by NSF Graduate Research Fellowship DGE–1106400, NSF Grants CCF–1139138, CCF–1337415, and ACI–1535191, a grant from U.S. Department of Energy, Office of Science, Office of Basic Energy Sciences Energy Frontier Research Centers program under award number FOA–0000619, and grants from DARPA FA8750–14–C–0011 and DARPA FA8750–16–2–0032, as well as gifts from Google, Intel, Mozilla, Nokia, and Qualcomm.

REFERENCES

Adelberg, Brad. 1998. NoDoSE - a tool for semi-automatically extracting structured and semistructured data from text documents. *In: Sigmod record.*

Barman, Shaon, Chasins, Sarah, Bodik, Rastislav, & Gulwani, Sumit. 2016. Ringer: Web automation by demonstration. *Pages 748–764 of: Proceedings of the 2016 acm sigplan international conference on object-oriented programming, systems, languages, and applications.* OOPSLA 2016. New York, NY, USA: ACM.

Chang, Chia-Hui, Kayed, Mohammed, Girgis, Moheb Ramzy, & Shaalan, Khaled F. 2006. A survey of web information extraction systems. *Ieee trans. on knowl. and data eng.*, **18**(10), 1411–1428.

Chasins, Sarah. 2017 (July). *schasins/helena: A chrome extension for web automation and web scraping.* https://github.com/schasins/helena.

Flesca, Sergio, Manco, Giuseppe, Masciari, Elio, Rende, Eugenio, & Tagarelli, Andrea. 2004. Web wrapper induction: A brief survey. *Ai commun.*, **17**(2), 57–61.

Furche, Tim, Guo, Jinsong, Maneth, Sebastian, & Schallhart, Christian. 2016. Robust and noise resistant wrapper induction. *Pages 773–784 of: Proceedings of the 2016 international conference on management of data.* SIGMOD '16. New York, NY, USA: ACM.

Greasemonkey. 2015 (Nov.). *Greasemonkey :: Add-ons for firefox.* https://addons.mozilla.org/en-us/firefox/addon/greasemonkey/.

hpricot. 2015 (Aug.). *hpricot/hpricot.* https://github.com/hpricot/hpricot.

Hupp, Darris, & Miller, Robert C. 2007. Smart bookmarks: automatic retroactive macro recording on the web. *Pages 81–90 of: Proceedings of the 20th annual acm symposium on user interface software and technology.* UIST '07. New York, NY, USA: ACM.

Import.io. 2016 (Mar.). *Import.io | web data platform & free web scraping tool.*

iOpus. 2013 (July). *Browser scripting, data extraction and web testing by iMacros.* http://www.iopus.com/imacros/.

KimonoLabs. 2016 (Mar.). *Kimono: Turn websites into structured APIs from your browser in seconds.*

Koesnandar, Andhy, Elbaum, Sebastian, Rothermel, Gregg, Hochstein, Lorin, Scaffidi, Christopher, & Stolee, Kathryn T. 2008. Using assertions to help end-user programmers create dependable web macros. *Pages 124–134 of: Proceedings of the 16th acm sigsoft international symposium on foundations of software engineering.* SIGSOFT '08/FSE-16. New York, NY, USA: ACM.

Kushmerick, Nicholas. 2000. Wrapper induction: Efficiency and expressiveness. *Artificial intelligence*, **118**(1), 15 – 68.

Kushmerick, Nicholas, Weld, Daniel S., & Doorenbos, Robert. 1997. Wrapper induction for information extraction. *In: Proc. ijcai-97.*

Le, Vu, & Gulwani, Sumit. 2014. FlashExtract: A framework for data extraction by examples. *Pages 542–553 of: Proceedings of the 35th acm sigplan conference on programming language design and implementation.* PLDI '14. New York, NY, USA: ACM.

Leshed, Gilly, Haber, Eben M., Matthews, Tara, & Lau, Tessa. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. *Pages 1719–1728 of: Proceedings of the sigchi conference on human factors in computing systems.* CHI '08. New York, NY, USA: ACM.

Li, Ian, Nichols, Jeffrey, Lau, Tessa, Drews, Clemens, & Cypher, Allen. 2010. Here's what i did: Sharing and reusing web activity with actionshot. *Pages 723–732 of: Proceedings of the sigchi conference on human factors in computing systems.* CHI '10. New York, NY, USA: ACM.

Lin, James, Wong, Jeffrey, Nichols, Jeffrey, Cypher, Allen, & Lau, Tessa A. 2009. End-user programming of mashups with Vegemite. *Pages 97–106 of: Proceedings of the 14th international conference on intelligent user interfaces.* IUI '09. New York, NY, USA: ACM.

Mahmud, Jalal, & Lau, Tessa. 2010. Lowering the barriers to website testing with cotester. *Pages 169–178 of: Proceedings of the 15th international conference on intelligent user interfaces.* IUI '10. New York, NY, USA: ACM.

Mayer, Mikaël, Soares, Gustavo, Grechkin, Maxim, Le, Vu, Marron, Mark, Polozov, Oleksandr, Singh, Rishabh, Zorn, Benjamin, & Gulwani, Sumit. 2015. User interaction models for disambiguation in programming by example. *Pages 291–301 of: Proceedings of the 28th annual acm symposium on user interface software & technology.* UIST '15. New

York, NY, USA: ACM.

Muslea, Ion, Minton, Steve, & Knoblock, Craig. 1999. A hierarchical approach to wrapper induction. *Pages 190–197 of: Proceedings of the third annual conference on autonomous agents.* AGENTS '99. New York, NY, USA: ACM.

Ni, Yang, Menon, Vijay S., Adl-Tabatabai, Ali-Reza, Hosking, Antony L., Hudson, Richard L., Moss, J. Eliot B., Saha, Bratin, & Shpeisman, Tatiana. 2007. Open nesting in software transactional memory. *Pages 68–78 of: Proceedings of the 12th acm sigplan symposium on principles and practice of parallel programming.* PPoPP '07. New York, NY, USA: ACM.

Nokogiri. 2016 (Nov.). *Tutorials - nokogiri.* http://www.nokogiri.org/.

Omari, Adi, Shoham, Sharon, & Yahav, Eran. 2017. Synthesis of forgiving data extractors. *Pages 385–394 of: Proceedings of the tenth acm international conference on web search and data mining.* WSDM '17. New York, NY, USA: ACM.

Platypus. 2013 (Nov.). *Platypus.* http://platypus.mozdev.org/.

Richardson, Leonard. 2016 (Mar.). *Beautiful Soup: We called him Tortoise because he taught us.* http://www.crummy.com/software/BeautifulSoup/.

Scrapy. 2013 (July). *Scrapy.* http://scrapy.org/.

scrapy-plugins. 2017. *scrapy-plugins/scrapy-deltafetch: Scrapy spider middleware to ignore requests to pages containing items seen in previous crawls.* https://github.com/scrapy-plugins/scrapy-deltafetch.

Selenium. 2013 (July). *Selenium-web browser automation.* http://seleniumhq.org/.

Selenium. 2016 (Mar.). *Selenium IDE plugins.* http://www.seleniumhq.org/projects/ide/.

StackOverflow. 2017. *Posts containing "incremental scraping" - stack overflow.*

VisualWebRipper. 2017 (Apr.). *Visual web ripper | data extraction software.* http://visualwebripper.com/.

Zheng, Shuyi, Song, Ruihua, Wen, Ji-Rong, & Giles, C. Lee. 2009. Efficient record-level wrapper induction. *Pages 47–56 of: Proceedings of the 18th acm conference on information and knowledge management.* CIKM '09. New York, NY, USA: ACM.