Customizing Software by Direct Manipulation of Tabular Data

Anonymous Author(s)

Abstract

In this paper we show how the behavior of a software application can be extended and adapted by direct manipulation. We introduce *table-driven customization*, a new paradigm where end users directly manipulate a table view of the structured data inside an application, rather than writing imperative scripts (as in most customization tools). This simple model also accommodates a spreadsheet formula language and custom data editing widgets, which provide sufficient expressivity to implement many useful customizations.

We illustrate the approach with Wildcard, a browser extension that implements table-driven customization in the context of existing web applications. Through concrete examples, we show that this paradigm can support useful customizations for real websites. We share reflections from our experiences using the Wildcard system, on both its strengths and limitations relative to other customization approaches. Finally, we explore how this paradigm might lead to new software architectures that encourage this form of customization.

CCS Concepts • **Software and its engineering → Integrated and visual development environments**.

Keywords end-user programming, software customization, web browser extensions

ACM Reference Format:

Anonymous Author(s). 2020. Customizing Software by Direct Manipulation of Tabular Data. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Many applications don't meet the precise needs of their users. End user customization systems can help improve the situation, by empowering non-programmers to modify their software in the way they would like.

Most end user customization systems offer a simplified version of programming. Some scripting languages [6, 9] have a friendly syntax that resembles natural language. Other visual customization tools eliminate text syntax entirely. Macro recorders [3, 8, 9] remove some of the initial programming burden by letting a user start with concrete demonstrations. Despite their many differences, these approaches all share something in common: an imperative programming model, with statement sequencing, mutable variables and loops.

We have known for decades about an alternate approach: *direct manipulation* [18], where "visibility of the object of interest" replaces "complex command language syntax". Direct manipulation is the *de facto* standard in GUIs today, but when it comes to customizing those GUIs, it is rarely to be found. As a result, switching from using software to customizing it poses a learning barrier for users not familiar with programming, and requires an abrupt shift in mental model even for those familiar with scripting.

In this work, we ask: what would it look like to build a direct maniulation interface that lowers the threshold for starting to customize software? We take inspiration from spreadsheets and visual database query interfaces, which have successfully enabled millions of end users to compute with data through direct manipulation.

In this paper we present a technique called *table-driven customization* inspired by those tools. An application's UI is augmented with a table view, where the user can see and manipulate the application's internal data. Changes in the table view result in immediate corresponding changes to the original user interface, enabling the user to customize an application with live feedback.

We have developed a browser extension called Wildcard which uses web scraping techniques to implement table-driven customization for existing Web applications. In Section 2, we illustrate the end user experience of table-driven customization through an example scenario using Wildcard.

In Section 3, we explain the system architecture of table-driven customization. We focus on the *table adapter* abstraction, which allows many different types of underlying data to be bidirectionally mapped to a table. We describe several types of table adapters we've built in Wildcard, and also describe future adapters supported by the general paradigm.

We have successfully used Wildcard to build customizations for 11 different websites. In Section 4, we present reflections from this process, outlining the kinds of customizations we were able to build, limitations we encountered, and the challenges of integrating scraping logic with websites.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

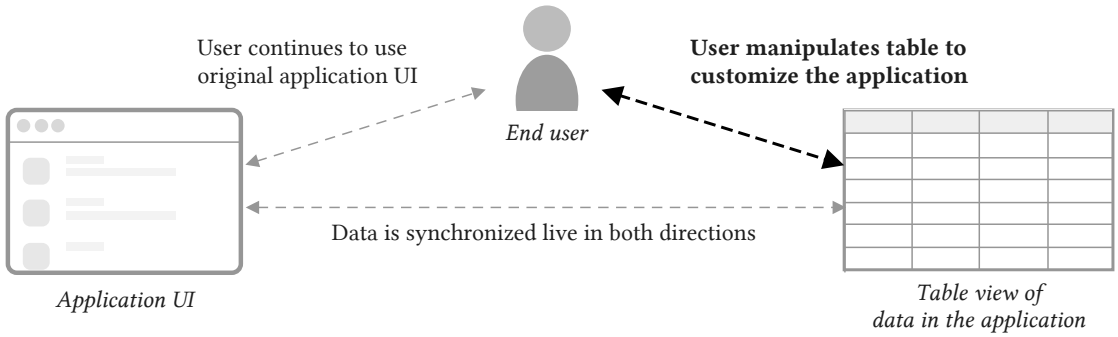

Figure 1. An overview of table-driven customization

In Section 5, we discuss some key themes from our work:

- *Customization by direct manipulation*: We agree with MacLean et al [16] that customization should follow a "gentle slope", where users can gradually invest in more complex techniques to achieve deeper customizations. We believe an important point on this slope is the ability to customize an application by directly seeing and changing its internal structured data, rather than by writing imperative scripts.
- *Wrapping applications for customization*: Typically, tools that don't rely on official extension APIs resort to offering low-level APIs for customization. Instead, we propose a community-maintained library of semantic wrappers around existing applications, enabling end users to work with domain objects rather than low-level representations.

Table-driven customization relates to existing work in many areas. In particular, our goals overlap with many software customization tools, and our methods overlap with direct manipulation interfaces for working with structured data, including visual database query systems and spreadsheets. We explore these connections and more in Section 6.

2 Example Scenario

To illustrate the end user experience of table-driven customization, we consider an example scenario of customizing Hacker News, a tech news aggregator. Figure 2 shows accompanying screenshots.

Opening the table: When the user opens Hacker News in a browser equipped with the Wildcard extension, they see a table at the bottom of the page. It contains a row for each link on the homepage, listing information like the title, URL, submitter username, number of points, and number of comments (Figure 2, Note A). The end user didn't need to do any work to create this table, because a programmer previously created an adapter to extract data from this particular website, and contributed it to a shared library of adapters integrated into Wildcard.

Sorting by points: First, the user decides to change the ranking of links on the homepage. Hacker News itself uses a ranking algorithm in which the position of an article depends not only on its point count (a measure of popularity), but also on how long it has been on the site. If the user hasn't been checking the site frequently, it's easy to miss a popular article that has fallen lower on the list. Sorting the page just by points would achieve a more stable ranking.

To achieve this ordering, the user simply clicks on the "points" column header in the table. This sorts the table view by points, and the website UI also becomes sorted in the same order (Figure 2, Note B)—Wildcard has manipulated the website's DOM to synchronize it with the sort order of the table. This sort order is also persisted in the browser and reapplied automatically the next time the user loads the page, so they can always browse the latest set of links using this sort order.

Adding estimated read times: Next, the user decides to attempt a more substantial customization: to add estimated read times to articles on Hacker News.

The table contains additional columns to the right of the original attributes for each row, where the user can enter spreadsheet-style formulas to compute derived values. We provide generic names for these columns by default as spreadsheets do, so that users don't need to choose column names themselves. The user enters a formula into the `user1` column: (Figure 2, Note C):

```
=ReadTimeInSeconds(link)
```

This calls a built-in function that takes a URL as an argument and uses a public web API to compute an estimated read time for the link's contents. The argument `link` refers to a column name in the table; for each row, the formula is evaluated with the link for that particular row. The result of evaluating this formula is a numerical value in seconds for each article.

The user clicks the `user1` column header to sort the articles on the page in descending order of estimated read time, helping to prioritize reading deeper content. It would be useful to show these read times in the page too, but before

A) Opening the table: The user opens a table view which shows data about each article in the list: its title, link, the number of points and comments, etc.

B) Sorting by points: When the user sorts the table by points, the web page becomes sorted in the same order

C) **Computing estimated read times:** The user enters a formula to fetch estimated read times from an API, and uses another formula to transform the results into a readable label

D) **Showing read times:** The formula results, and manual annotations, are shown in the page next to each article

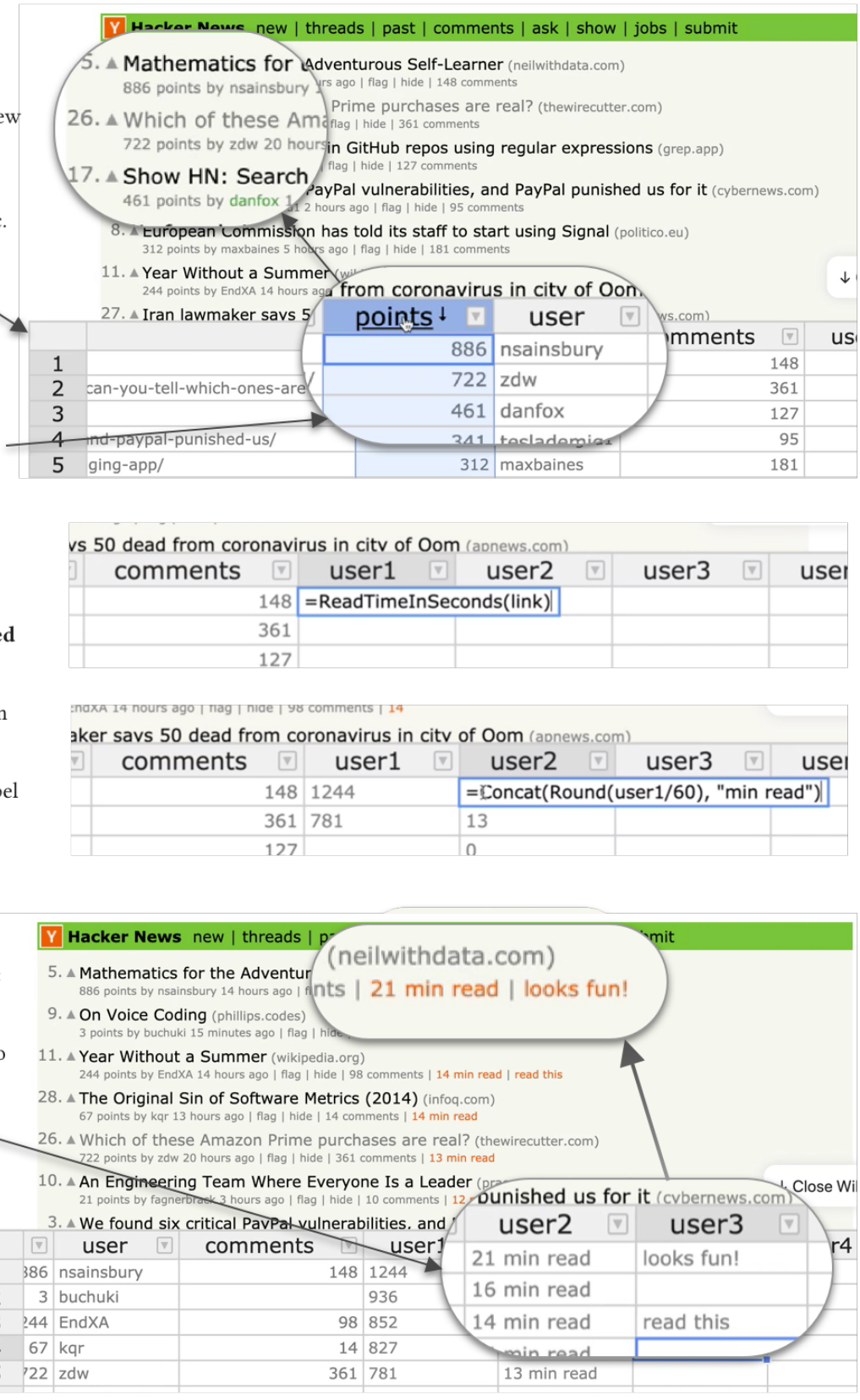

Figure 2. Customizing Hacker News by interacting with a table view

doing that, the user wants to make the time estimates more readable. They enter another formula in the next column, `user2`:

```
=Concat(Round(user1/60), "min read")
```

This formula converts seconds to minutes by dividing by 60 and rounding to the nearest integer, and then concatenates a label to the number, producing strings like "21 min read".

Finally, the user clicks a menu option in the table header to display the contents of this new column in the original page (Figure 2, Note D). Each article on the page now shows an annotation with the estimated read time in minutes. (The format of how annotations appear for each item was determined by the programmer who created the Wildcard adapter for Hacker News.)

Adding notes to links: The user can also manually add notes to the table, by simply entering values into the table without using formulas. In this case, the user jots down a few notes in another column about articles they might want to read, and the notes appear in the page next to the read times (Figure 2, Note D). The annotations are also stored in the browser's local storage so they can be retrieved on future visits.

Filtering out visited links: Another way to use formulas to customize Hacker News is to filter out articles the user has already read. (We omit this example from the figure for brevity.) The user can call a built-in function that returns a boolean depending on whether a URL is in the browser's history:

```
=Visited(link)
```

They can then filter the table to only contain rows where this formula column contains `false`; visited rows are hidden both from the table view and the original page. This is an example of a customization that the original website could not implement, since websites don't have access to the browser history for privacy reasons. But by using Wildcard, the user can implement the customization locally, without needing to expose their entire browser history to Hacker News.

This scenario has shown a few examples of how table-driven customizations can help a user improve their experience of a website. Section 4 explains many other use cases, but first we explain how the system works internally.

3 System architecture

Figure 3 summarizes the overall architecture of table-driven customization, using a simplified illustration of the Hacker News example scenario. In this example, the name and points value for each article is scraped from the web page DOM, and user annotations are loaded from the brower's local storage.

First, the web page and the browser storage are each wrapped by a **table adapter**, which defines a bidirectional mapping between some underlying data source and a table.

In addition to a *read* mapping for how the underlying data should be represented as a table, it also defines a *write* mapping for what effects edits should have on the original data source. In this case, the local storage adapter has a trivial mapping: it loads a table of data stored in the browser, and persists edits to that persisted state. The mapping logic of the DOM scraping adapter is much more involved. It implements web scraping logic to produce a table of data from the web page, and turns edits into DOM manipulations, such as reordering rows of data on the page.

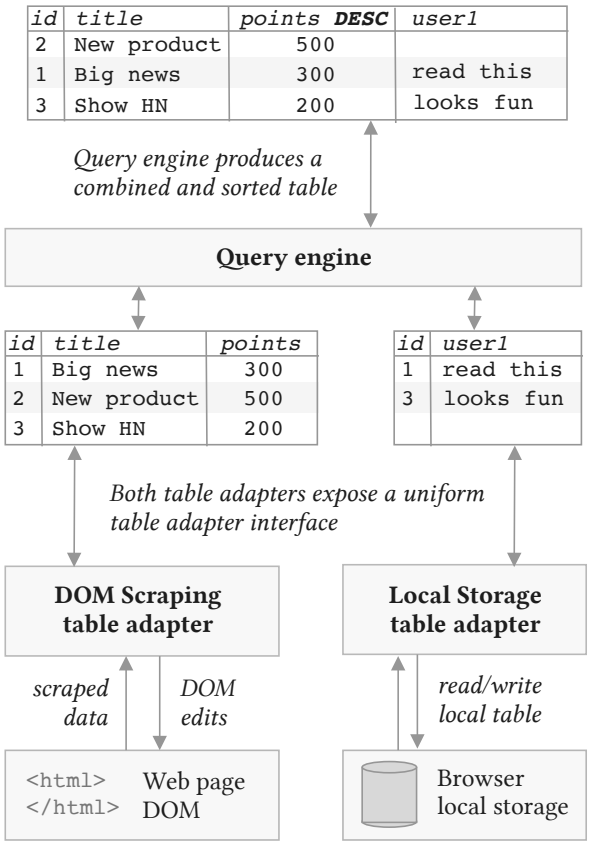

Figure 3. The table adapter architecture

The two tables are then combined into a single table for the end user to view and edit. The **query engine** is responsible for creating this combined view, and routing the user's edits back to the individual table adapters. In this example, the query engine has joined the two tables together by a shared ID column, and sorted the result by the points column.

We now examine each component of the system in more detail.

3.1 Table adapters

A key idea in table-driven customization is that a wide variety of data sources can be mapped to a generic table abstraction. In a relational database, the table matches the underlying storage format, but in table-driven customization, the table

is merely an *interface layer*. The data shown in the table is a projection of some underlying state, and edits to the table can have complex effects on the underlying state.

The first two parts of the table adapter interface resemble a typical database table:

Returning a table: A table adapter exposes a table of data: an ordered list of records. Each record carries a unique identifier and associates named attributes with values. Tables have a typed schema, so the same attributes are shared across all records. The columns also carry some additional metadata in addition to their type, such as whether or not they are read-only or editable.

A table adapter can update the contents of a table at any time in response to changes in the underlying state (e.g., a DOM scraping adapter can update the table when the page body changes). When data changes, the query view is reactively updated in response.

Handling edits: The query engine can issue a request to a table adapter to make an edit to a record. The meaning of making an edit can vary depending on the adapter: in the local storage adapter, an annotation may be persisted into local storage; in the DOM scraping adapter, an edit may represent filling in a form field.

In addition, the query engine also sends additional information about the combined query view to each table adapter:

Sorting/filtering: When the user sorts or filters the query view, an ordered list of visible IDs is sent to each table adapter. The DOM scraping adapter uses this information to change the list of rows shown in the web page.

Data from other tables: The query engine provides each table adapter with the entire combined table shown to the user. The DOM scraping adapter uses this for injecting annotations—values in additional columns from other tables are added to the original web page.

Currently selected record: As the user clicks in different parts of the table view, the query engine communicates the record currently selected by the user. The DOM scraping adapter uses this information to highlight the row in the page that corresponds to the selected row in the table (and scroll to the row if it's not currently showing on the page), which helps clarify the connection between the table and the original UI.

3.1.1 Types of table adapters

We have built three specific types of table adapters so far, to enable web customization using Wildcard.

DOM scraping adapters are the essential component that enables Wildcard to interface with an existing website UI. In addition to the standard web scraping problem of extracting a table of data from the DOM, a scraping adapter must also manipulate the DOM to reorder rows, edit form entries, and inject annotations as the table is edited.

Because each website has unique content, we rely on programmers to create a DOM scraping adapter for each individual website to make it available for customization in Wildcard. To make this approach viable, we have built a generic DOM scraping adapter which a programmer can configure with the minimal site-specific parts.

After specifying basic metadata like which URLs the adapter should apply to, the programmer only needs to implement a single Javascript function that scrapes relevant elements from the page. They are free to use any scraping techniques; in practice simple CSS selectors and DOM APIs usually suffice. The generic adapter then wraps this scraping function to implement the table adapter interface. For example, when the table view is sorted, the generic adapter takes the DOM elements corresponding to the rows in the table, removes all of them from the page, and then reinserts them in the new order.

The programmer can optionally override other default behavior of the generic DOM scraping adapter, including the logic for when to re-run the scraping function in response to page changes, how to style injected annotations, and how to style the currently selected row in the table.

An **AJAX scraping adapter** intercepts AJAX requests made by a web page, and extracts information from those requests to add to the table. When available, this tends to be a helpful technique because the data is already in a structured form so it is easier to scrape, and it often includes valuable information not shown in the UI.

As with DOM scraping adapters, we have made it easy for programmers to create site-specific AJAX scraping adapters. A programmer writes a function that specifies how to extract data from an AJAX request, and the framework handles the details of intercepting requests and calling the programmer-defined function.[1]

In order to join the tables produced by AJAX scraping and DOM scraping, a common set of identifiers is required across records in the two tables. Often there is a server-defined ID present both in the DOM and in AJAX responses; if not, the programmer can use some set of overlapping data (e.g. an item name) as a shared ID.

The **local storage adapter** simply stores a table of data in the browser. This is currently only used to persist annotations: the user makes edits in the table, they are stored privately in the browser, and can be loaded into the table on future pageloads.

3.1.2 Future Adapters

We have designed the table adapter API to be general enough to support other types of useful adapters in the future. Here are two examples:

[1]So far we have only implemented AJAX scraping in the Firefox version of Wildcard, since Firefox has convenient APIs for intercepting requests. It appears possible to implement in Chrome as well, but we have not finished our implementation.

Integrated website adapters: A key benefit of the table adapter abstraction is that Wildcard is not coupled to web scraping as the only means for integrating with existing sites, but can also accommodate first party developers adding support directly into their own websites. An "integrated website adapter" installed by the developer could directly accesses the internal state of the application, providing the same functionality as a DOM scraping adapter but in a more robust way.

With the advent of rich frontend web frameworks, structured application state is now often available in the web client. We suspect it is possible to create plugins for frontend frameworks that expose this state to Wildcard with only minimal effort from the application developers. To test this hypothesis, we have created an early prototype of a Wildcard adapter for the Redux state management library, and used it with the TodoMVC todo list application. To install the adapter, the programmer only needs to make a few small changes to their app:

- *Exposing a table*: In Redux, app developers already represent the state of a user interface as a single object. To expose a table of data, the developer simply writes a function that maps the state object to a table.
- *Editing data values*: The developer defines how edits to records in the table should map to edits in the state object.
- *Sorting/filtering the UI*: The developer inserts an additional data processing step before the state object is rendered in the UI, so that Wildcard can apply sorting and filtering.
- *Annotations*: The developer adds additional React components managed by Wildcard into their UI tree, which render additional data as annotations.

Customizability is sometimes a key selling point for software. An integrated website adapter would provide a way for developers to integrate with an ecosystem of formulas and customization tools without needing to build all that functionality from scratch.

Shared storage adapter: It would be useful to share user annotations among users and across devices—for example, collaboratively taking notes with friends on options for places to stay together on a trip. The existing local storage adapter could be extended to share live synchronized data with other users. This could be achieved through a centralized web server, or through P2P connections that might provide stronger privacy guarantees.

3.2 Query engine

The query engine is responsible for coordinating across multiple table adapters. It joins data across multiple tables and creates a single result table which is shown to the user through the editor. It also handles all user interactions and routes appropriate messages to each table adapter.

Queries are processed in three steps. First, the query invokes a primary DOM scraping adapter that associates table rows with elements in the application's user interface. Next, additional tables (AJAX data, local storage data) are left-joined by ID. Finally, the result table is sorted and filtered according to user-specified predicates.

One way to view this query model is as a tiny subset of the SQL query model. Despite its simplicity, this model has proven sufficient for meeting the needs of customization in practice, and minimizes the complexity of supporting arbitrary queries. But because it fits into the general paradigm of relational queries, it could theoretically be extended to support a wider range of queries.

The query engine is also responsible for executing formulas. We have built a small formula language resembling a spreadsheet formula language, except that formulas automatically apply across an entire column of data, and reference other column names instead of values in specific rows. This is more convenient than needing to copy-paste a formula across an entire column as in spreadsheets, and has worked for all of the customizations we have built.

3.3 Table editor

We provide a basic table editor as the user interface on top of the query engine. Our table editor is built with the Handsontable Javascript library, which provides UI elements for viewing, editing, sorting, and filtering a table.

In addition to the basic table editing operations, we also provide *cell editors*: UI widgets that expose a custom editing UI for a single cell of the table view. A programmer building a cell editor need only integrate it with the table viewer; propagating values into the website UI is handled by the site-specific DOM adapter. In Section 4 we provide some examples of using cell editors.

The table editor only serves as a shallow interface layer over the query engine, relaying user commands to the query engine and rendering the resulting data table. Because of this architectural split, it would be straightforward to develop additional table editor interfaces on top of the Wildcard system. One intruiging future possibility is to apply the idea of cell editors to the entire table, by developing table editors which don't look like a table. For example, we could provide a calendar view for displaying a table containing a date column.

4 Reflections on Usage

To evaluate table-driven customization in practice, we built the Wildcard browser extension, which implements table-driven customization in the context of existing websites. It is implemented in Typescript, and works across three major browsers: Chrome, Firefox, and Edge.

We built Wildcard integrations for 11 websites, including transactional sites like Amazon and Uber Eats, and media

Website	Description	LOC	Example customizations
Airbnb	Travel	73	Add Walkability Scores to listings. Sort listings by price.
Amazon	Online shopping	99	Sort used book sellers by total price, including delivery fees
Blogger	Blogging	36	Use alternate text editor to edit blog posts
Expedia	Travel	41	Use alternate datepicker to enter travel dates
Flux	Data portal	67	Use Wildcard as a faster table editor for editing lab results
Github	Code repository	62	Sort a user's code repositories by stars to find popular work
Hacker News	News	69	Add read times to links. Filter out links that have been read.
Instacart	Grocery delivery	48	Sort groceries by price and category. Take notes on items.
Uber Eats	Food delivery	117	Sort/filter restaurants by estimated delivery ETA and price.
Weather.com	Weather	51	Sort/filter hourly weather to find nice times of day.
Youtube	Videos	80	Sort/filter videos by length, to find short videos to watch.

Table 1. A list of table-driven customizations that we have implemented using Wildcard.

consumption sites like Hacker News and Youtube. Table 1 summarizes these integrations, showing the number of lines of code in the adapter configuration for each site, and some example customizations we were able to perform for each site.

So far, most usage of Wildcard has come from members of the project team. Here we offer our reflections on using the system, focused on two key questions:

- How broad is the range of possible customizations in this paradigm?
- How feasible is it to build DOM scraping adapters for real websites?

4.1 Range of customizations

We have found that table-driven customization can serve a broad range of useful purposes. Here we expand on some archetypal examples that illuminate aspects of using the system in practice.

4.1.1 Sorting/filtering

It might seem that most websites already have adequate sorting and filtering functionality, but we have found it surprisingly helpful to add new sorting/filtering functionality to websites using Wildcard.

Sometimes, websites have opaque ranking algorithms which presumably maximize profit but don't offer much control to the user. For example, Airbnb previously allowed users to sort listings by price, but removed that feature in 2012. We have found other sites where a lack of sorting options seems more like merely an omission; for example, the Instacart grocery delivery service has a spartan UI for viewing a pending order, which doesn't allow for sorting by price or category. In both of these cases, Wildcard enables users to take back some control.

In the current implementation of Wildcard, users can only sort and filter entries that are shown on the current page, which means that users are not entirely liberated from the suggestions of the opaque algorithm. This restriction could be overcome in the future by scraping content across multiple pages, or by using an integrated adapter built in to the application. However, we've also found that sorting/filtering a single page of a paginated list is often an acceptable outcome (and sometimes even a preferable one). It's more useful, for example, to sort 30 recommended Youtube videos than to try to sort all videos on Youtube.

4.1.2 Annotating

Many web annotation systems focus on annotating text or arbitrary webpage content, but Wildcard limits annotations to structured objects extracted by an adapter, resulting in a different set of use cases. Annotating with Wildcard has proven most useful when taking notes on a list of possible options (e.g., evaluating possible Airbnb locations to rent). We have also used it with Instacart's online grocery cart, for jotting down notes like "should we get more milk?"

4.1.3 Formulas

Formulas are the most powerful part of the Wildcard system. So far, our language supports only a small number of predefined functions. Adding more should allow a broad range of useful computations, as shown by the success of spreadsheets.

Formulas are especially useful for fetching data from Web APIs. We've used them to augment Airbnb listings with walkability scores, and to augment Hacker News articles with estimated read times. One challenge of the current language design is that supporting a new web API requires writing Javascript code to add a new function to the language, because web APIs typically return complex JSON data structures that can't be easily displayed in a single table cell. In the future we would like to make it possible to call new APIs without adding a dedicated function, which might require adding functions to the formula language that can manipulate JSON data.

We have also found instances where simple data manipulation is useful, e.g. transforming the results of an API call

with basic arithmetic and string operations, as shown in Section 2.

4.1.4 Cell editors

We developed two cell editors to explore different benefits of adding custom UI for editing the table.

An important benefit of cell editors is that they allow users to incorporate their private information into a web UI without uploading it to a website. To explore this idea, we created a datepicker based on the FullCalendar plugin, which can load data from a Google Calendar. This makes it convenient to enter dates into a website based on the user's personal calendar information.

Another benefit of cell editors is that a user can choose their preferred widget for editing some type of information. We built a text editor based on the CKEditor rich text editor, and demonstrated its use by integrating it with Google's Blogger website and using it to edit blog post content.

4.1.5 Limitations

There are many customizations that are not possible to implement with table-driven customization. Some of the limitations are specific to the current implementation of the Wildcard extension, but many are more fundamental to the general paradigm.

One limitation is that Wildcard can only make customizations that use the available data exposed in the table. If the adapter doesn't expose some piece of data, the user can't use it in their customization. The table data format also rules out customizing certain sites that don't have a way to map to a table. The UI modifications available in Wildcard are also limited in scope; deleting arbitrary buttons isn't possible, for example. There is no facility for running automations when the user isn't actively viewing a page—at one point, we wanted to build an automation to repeatedly load a grocery delivery website to check for open delivery slots, but it didn't seem possible to achieve this in Wildcard.

We consider these limitations to be an acceptable outcome. Our goal is to support as many useful customizations as possible with a low threshold of difficulty, and not to span all possible customizations. If users want to implement more sophisticated customizations, they have the option of graduating to more advanced customization tools like scripting languages.

We have found that one benefit of showing structured data is predictability: once we build an adapter for a website, it is clear what data is available or unavailable for use in customizations. Also, there is sometimes a way to reframe an imperative script in terms of our direct manipulation model. For example, a script that iterates through rows in a page adding some additional information to each row can be reproduced using a single formula in Wildcard.

4.2 Viability of scraping

Our second evaluation area relates less to the conceptual approach of table-driven customization, and more to the specific implementation of customizing existing web applications. In order for third-party customization through Wildcard to succeed, it is important that creating usable adapters for existing websites takes minimal effort.

Nearly all of our DOM scraping adapters were created by members of our team. However, an external developer unaffiliated with the project contributed one adapter, designed to sort the Github page listing a user's repositories, and they described the experience as "very straightforward."

The adapters for our test sites ranged from 36 to 117 lines of code, averaging 68 lines; Table 1 shows the number of lines of code for each adapter. Most of the code in the adapters is simply using DOM APIs and CSS selectors to implement conventional web scraping logic.

Some of the challenges of writing a DOM scraping adapter are the same ones as with writing normal web scraping code, but the more interactive nature of Wildcard introduces additional challenges. One challenge is registering appropriate event handlers to update the table data in response to UI changes that happen after initial page load. Another challenge is persisting updates to the DOM—some websites use virtual DOM frameworks that can occasionally overwrite changes made by Wildcard. So far, in practice we've managed to work around these issues for all of the websites we've tried, but we don't claim that any website can be customized through DOM scraping. As web frontend code gets increasingly complex (and starts to move beyond the DOM to other technologies like Shadow DOM or even WebGL), it may become increasingly difficult to customize websites from the outside without first-party support.

AJAX scraping proved highly useful in several cases. The Uber Eats website was challenging to scrape because it has a complex DOM structure with machine-generated CSS classes, but the site also generates AJAX requests which contain all the relevant data in a structured form that is much easier to extract. We also found examples where relevant information wasn't present in the DOM at all. On the grocery delivery site Instacart, we used AJAX scraping to augment grocery items with their categorization in the store, enabling use cases like viewing all the vegetables in an order together.

5 Key themes

Here we discuss some themes that we've explored with this work, which go beyond our specific approach and address issues in software customization more broadly.

5.1 Customization by direct manipulation

Hutchins, Hollan and Norman [12] define a direct manipulation interface as one that uses a model-world metaphor rather than a conversation metaphor. Instead of presenting

an "assumed" but not directly visible world that the user converses with, "the world is explicitly represented" and the user can "[act] upon the objects of the task domain themselves."

Although most GUIs today employ direct manipulation, software customization tools typically use an imperative programming model, which implements the conversational metaphor rather than direct manipulation. Here, for example, is how a user retrieves a list of of calendar names from the Calendar application in Applescript [9], the scripting language for customizing Mac OS applications:

```
tell application "Calendar"
  name of calendars
end tell
```

Some customization environments like Mac Automator and Zapier forego textual syntax and let the user connect programs and construct automations by dragging and dropping icons representing commands. These environments still do not constitute direct manipulation, though: the objects being manipulated are in the domain of programming, not in the domain of the task at hand.

Imperative programming is a reasonable choice as the model for building customizations. Turing-complete programming provides a high ceiling for possible customizations, and a sequence of commands is a natural fit for automations that simulate a series of steps taken by the user. There is, however, a serious drawback to this approach. MacLean et al [16] describe an ideal for user-tailorable systems: a "gentle slope" from using to customizing, where small incremental increases in skill lead to corresponding increments of customization power. Requiring users wanting to customize their applications to learn programming creates an abrupt "cliff," exacting a significant investment in learning even to implement the simplest customizations. Another goal of MacLean et al is to make it "as easy to change the environment as it is to use it"—at least for some subset of changes. But in scripting languages, the experience of customization does not remotely resemble the experience of use.

With table-driven customization we aim to provide a gentler slope, by using direct manipulation for software customization. The data shown in the table view is the domain data from the original application. The user makes changes to the data by selecting areas of interest in the table, e.g. sorting/filtering by clicking the relevant column header, or adding annotations by clicking on the relevant row. These interactions are common in GUI applications, and Wildcard therefore meets MacLean et al's goal: some one-click customizations are as easy as using the original application. Formulas introduce some additional complexity, but the pure functional semantics and the intermediate visibility of results make the task still easier than imperative programming.

One aspect of directness that we have chosen not to maintain in Wildcard is enabling customization in closer proximity to the original user interface elements, as explored by tools like Scotty [10]. We have found that augmenting the original UI with a distinct, additional representation provides a more consistent experience across all applications, and clearly shows the user what structured data is available to work with.

Ainsworth et al provide a helpful taxonomy of the value of multiple representations [1]. In their terms, Wildcard plays a *complementary role* by supporting a *different set of tasks* from the original application, while displaying *shared information*. Wildcard may also help construct *deeper understanding by subtraction*. By stripping away details and only showing the essential data in an interface, Wildcard helps users think of new ways of using that data, outside the specific constraints of the original application.

As noted in Section 4, there are many customizations that can be achieved in scripting languages that cannot be implemented in Wildcard. We consider this an acceptable tradeoff in exchange for a gentler slope in customization, and we show in Section 2 and Section 4 that our model can still implement many useful customizations.

5.2 Semantic wrappers

Ad hoc customization tools enable customization without using official extension APIs, enabling a broader range of customizations on top of more applications. For example, web browser extensions have demonstrated the utility of customizing websites through manipulating the DOM, without websites needing to provide explicit extension APIs. However, ad hoc customization comes with a cost: these tools typically operate at a low level of abstraction, e.g. manipulating user interface elements, rather than in a meaningful domain model. This makes it harder for end users to write scripts, and makes the resulting scripts more brittle as the specifics of a user interface change.

Anticipated customization tools, in contrast, use explicit extension APIs provided by the application developer. Examples of this include accessing a backend web API, or writing a customization in Applescript for an application that exposes its domain model to the scripting language. The main benefit of this style is that it allows the extension author to work with meaningful concepts in the application domain—"create a new calendar event" rather than "click the button that contains the text 'new event'."—which makes customizations easier to build and more robust. However, the plugin API limits the types of customizations that can be built, and many applications don't have an extension system at all.

With Wildcard, we use a hybrid approach that aims to provide the best of both worlds. Programmers implement site-specific adapters that are internally implemented ad hoc customizations, but externally provide a high-level interface to the application, abstracting away the details of the user

interface. These wrappers are added to a shared repository, available to all users of the system. When an end user is using a site that already has an adapter implemented, they benefit from a semantic customization experience that avoids low-level details.

One way to view this approach is as introducing a new abstraction barrier into third-party extension. Typically, a third party customization script combines two responsibilities: 1) mapping the low-level details of a user interface to semantic constructs (e.g., using CSS selectors to find certain page elements), and 2) handling the actual logic of the specific customization. Even though the mapping logic is often more generic than the specific customization, the intertwining of these two responsibilities in a single script makes it very difficult to share the mapping logic across scripts.

With Wildcard we propose a decoupling of these two layers: a repository of shared wrappers maintained by programmers, and a separate repository of specific customizations built on top of these wrappers. This general architecture has been successfully demonstrated by projects like Gmail.js, an open source project that creates a convenient API for browser extensions to interface with the Gmail web email client.

The success of semantic wrappers depends on a key hypothesis: that a single wrapper created by a programmer can be used for many different purposes by end users. Although we've validated that a single generic adapter can support many customizations, so far the people making the adapters have largely been the same people building customizations on top of them. One helpful future addition would be to allow end users to create wrappers themselves, or at least to modify existing ones. There are many existing projects exploring end-user web scraping (such as Helena [8]) which might prove helpful for this.

The distribution mechanism for semantic wrappers is also important. Currently, the distribution mechanism is simply merging the code for all adapters into the main Wildcard codebase. This is the simplest workable solution, but has its downsides: contributing involves a fairly high barrier of creating a pull request on Github, and using newly contributed wrappers requires installing a new version of the extension. In the future we might explore other mechanisms, like an online repository that the extension pulls from dynamically. Security is also a consideration—DOM scraping adapters can execute arbitrary Javascript code, which opens up the possibility of malicious adapters being contributed. Currently we plan to solve this challenge with centralized code review, but another approach we are considering is using or inventing a more restricted domain-specific language for specifying scraping logic.

6 Related Work

This paper extends work reported in a workshop paper by Litt and Jackson [15] which presented an earlier version of the Wildcard extension. We have substantially extended their work in this paper by creating the table adapter abstarction, reimplementing the internals of Wildcard around that abstraction, evaluating the system on many more websites, and by characterizing the design of the system in much more detail than in their workshop paper.

Table-driven customization relates to two broad areas of related work. Our problem statement is related to software customization tools, and our solution approach is related to spreadsheets and other direct manipulation interfaces.

6.1 Customization tools

Table-driven customization is most closely related to other tools that aim to empower end users to customize software without traditional coding.

This lineage goes back at least to the Buttons system by MacLean et al [16], where Xerox Lisp users could share buttons that performed various "tailoring" actions on the system. The authors proposed the "gentle slope" idea which has greatly influenced our approach to table-driven customization (as discussed in Section 5.1). The authors also point out the importance of a "tailoring culture" where people with different skillsets collaborate to produce useful customizations; in their system, Lisp programmers create buttons that others can use, modify, and rearrange. This division of labor corresponds to our idea of semantic wrappers, where end user customization is supported by programmer-created building blocks.

Some recent web customization tools aim to enable end users to modify web interfaces without programming.

Sifter [13] enables end users to sort and filter lists of data obtained by web scraping, much like Wildcard's sorting features. The main difference between the systems is that table-driven customization has many other use cases besides sorting and filtering. Also, Sifter involves end users in a semi-automated data extraction process, rather than having programmers create wrappers. This provides coverage of more websites, but at the expense of complicating the end user experience. We could integrate end user scraping techniques in Wildcard in the future, but we believe that when possible it's valuable for end users to have a customization experience decoupled from the challenge of web scraping the underlying data. Sifter also implements scraping across multiple pages, a valuable feature for sorting and filtering that isn't present in Wildcard.

Thresher [11] helps end users create wrappers that map website content to Semantic Web schemas like "Movie" or "Director," and augments websites with new functionality by exploiting that schema information. Wildcard shares the general idea of wrappers, but maps to a generic table data type

rather than more specific schemas, increasing the range of supported data and allowing for a simpler mapping process.

There are many software customization tools that offer simplified forms of programming for end users. Chickenfoot [6] and Coscripter [14] offer user friendly syntax for writing web automation scripts; Applescript [9] has a similar goal for desktop customization. There are visual programming environments for customization that don't involve writing any text: Automator for Mac and Shortcuts for iOS are modern options for customizing Apple products, and Zapier enables users to connect different web applications together visually. As mentioned previously, these tools all require writing imperative programs, in contrast to the more declarative and direct approach of table-driven customization.

6.2 Direct manipulation programming interfaces.

Another relevant area involves spreadsheets and visual query systems. We take inspiration from these tools in our work, but apply them in a different domain: customizing existing software applications, rather than interacting with databases or constructing software from scratch.

The most closely related work is in systems that offer spreadsheet-like querying of relational data. SIEUFERD by Bakke and Karger [4] is one such recent system, and their paper presents a survey of many other similar tools. Our work is particularly influenced by the authors' observation that direct manipulation requires that the user manipulate the results of a database query rather than the query itself, and that the user must see intermediate results at every step of constructing the query. SIEUFERD's interface supports a far wider range of queries than Wildcard, but the fundamental principles of the user interface are similar. Airtable is another example of a modern commercial product that offers spreadsheet-like interaction with a relational database.

Our work is also inspired by the many projects that have explored using spreadsheets as a foundation for building software applications, including Object Spreadsheets [17], Quilt [5], Gneiss [7], Marmite [19], Glide. We share the core idea of connecting a spreadsheet view to a GUI, but apply it to software customization, rather than building software from scratch.

Another related system is ScrAPIr, by Alrashed et al. [2], which enables end users to access backend web APIs without programming. ScrAPIr shares our high level goal of end user empowerment, as well as the idea of wrappers, by creating a shared library of wrappers around existing web APIs. Unlike Wildcard, however, ScrAPIr targets explicit APIs exposed by developers. It also focused on backend services and doesn't aim to extend the frontend interfaces of web applications.

7 Conclusion and Future Work

In this paper, we have presented table-driven customization, a new paradigm for customizing software by direct manipulation of the underlying structured data. We have demonstrated the paradigm using a browser extension, and have used it to create useful customizations for a variety of websites.

Our primary goal for future work is to evaluate the system with a broader group of users. What barriers do end users face when using the system? What types of customizations do they choose to create? What formulas prove most helpful?

Another area for future work is exploring whether it is possible to express a broader range of customizations by extending our table-editing paradigm. Are there ways to offer an increase in power and functional complexity, while retaining a programming model that is simpler for end users than conventional coding?

As computing plays an ever greater role in our lives, it is increasingly important that end users have agency over the behavior of their software, rather than having every detail be dictated by companies whose incentives are not always aligned with the user's interests. We hope that table-driven customization can serve as one point, and maybe a guidepost, on our path from normal use to deep modification, in support of a more adaptable experience for all computer users.

References

[1] Shaaron Ainsworth. 1999. The Functions of Multiple Representations. *Computers & Education* 33, 2-3 (Sept. 1999), 131–152. https://doi.org/10.1016/S0360-1315(99)00029-9

[2] Tarfah Alrashed, Jumana Almahmoud, Amy X. Zhang, and David R. Karger. 2020. ScrAPIr: Making Web Data APIs Accessible to End Users. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, Honolulu, HI, USA, 1–12. https://doi.org/10.1145/3313831.3376691

[3] Vinod Anupam, Juliana Freire, Bharat Kumar, and Daniel Lieuwen. 2000. Automating Web Navigation with the WebVCR. *Computer Networks* 33, 1 (June 2000), 503–517. https://doi.org/10.1016/S1389-1286(00)00073-6

[4] Eirik Bakke and David R. Karger. 2016. Expressive Query Construction through Direct Manipulation of Nested Relational Results. In *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*. ACM Press, San Francisco, California, USA, 1377–1392. https://doi.org/10.1145/2882903.2915210

[5] Edward Benson, Amy X. Zhang, and David R. Karger. 2014. Spreadsheet Driven Web Applications. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology - UIST '14*. ACM Press, Honolulu, Hawaii, USA, 97–106. https://doi.org/10.1145/2642918.2647387

[6] Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller. 2005. Automation and Customization of Rendered Web Pages. In *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology - UIST '05*. ACM Press, Seattle, WA, USA, 163. https://doi.org/10.1145/1095034.1095062

[7] Kerry Shih-Ping Chang and Brad A. Myers. 2014. Creating Interactive Web Data Applications with Spreadsheets. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology - UIST '14*. ACM Press, Honolulu, Hawaii, USA, 87–96. https://doi.org/10.1145/2642918.2647371

[8] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *The 31st Annual ACM Symposium on User Interface Software and Technology - UIST '18*. ACM

Press, Berlin, Germany, 963–975. https://doi.org/10.1145/3242587.3242661

[9] William R. Cook. 2007. AppleScript. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages - HOPL III*. ACM Press, San Diego, California, 1–1–1–21. https://doi.org/10.1145/1238844.1238845

[10] James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the Cocoa Nut: User Interface Programming at Runtime. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology - UIST '11*. ACM Press, Santa Barbara, California, USA, 225. https://doi.org/10.1145/2047196.2047226

[11] Andrew Hogue and David Karger. 2005. Thresher: Automating the Unwrapping of Semantic Content from the World Wide Web. In *Proceedings of the 14th International Conference on World Wide Web - WWW '05*. ACM Press, Chiba, Japan, 86. https://doi.org/10.1145/1060745.1060762

[12] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct Manipulation Interfaces. (1985), 28.

[13] David F. Huynh, Robert C. Miller, and David R. Karger. 2006. Enabling Web Browsers to Augment Web Sites' Filtering and Sorting Functionalities. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology - UIST '06*. ACM Press, Montreux, Switzerland, 125. https://doi.org/10.1145/1166253.1166274

[14] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*. ACM, Florence, Italy, 1719–1728. https://doi.org/10.1145/1357054.1357323

[15] Geoffrey Litt and Daniel Jackson. 2020. Wildcard: Spreadsheet-Driven Customization of Web Applications. In *Companion Proceedings of the 4th In- Ternational Conference on the Art, Science, and Engineering of Programming*. Association for Computing Machinery, Porto, Portugal., 10. https://doi.org/10.1145/3397537.3397541

[16] Allan MacLean, Kathleen Carter, Lennart Lövstrand, and Thomas Moran. 1990. User-Tailorable Systems: Pressing the Issues with Buttons. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems Empowering People - CHI '90*. ACM Press, Seattle, Washington, United States, 175–182. https://doi.org/10.1145/97243.97271

[17] Matt McCutchen, Shachar Itzhaky, and Daniel Jackson. 2016. Object Spreadsheets: A New Computational Model for End-User Development of Data-Centric Web Applications. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2016*. ACM Press, Amsterdam, Netherlands, 112–127. https://doi.org/10.1145/2986012.2986018

[18] B. Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (Aug. 1983), 57–69. https://doi.org/10.1109/MC.1983.1654471

[19] Jeffrey Wong and Jason I. Hong. 2007. Making Mashups with Marmite: Towards End-User Programming for the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '07*. ACM Press, San Jose, California, USA, 1435–1444. https://doi.org/10.1145/1240624.1240842