

Building Personal Software with Reactive Databases

by

Geoffrey Litt

B.S., Yale University (2014)

S.M., Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer
Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2023

©2023 Geoffrey Litt. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide,
irrevocable, royalty-free license to exercise any and all rights under
copyright, including to reproduce, preserve, distribute and publicly
display copies of the thesis, or release the thesis under an open-access
license.

Authored by: Geoffrey Litt
Department of Electrical Engineering and Computer Science
August 30, 2023

Certified by: Daniel Jackson
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Building Personal Software with Reactive Databases

by

Geoffrey Litt

Submitted to the Department of Electrical Engineering and Computer Science
on August 30, 2023, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Spreadsheets and relational databases can simplify the creation of a variety of software, particularly for end-users who are less familiar with programming. This thesis extends techniques from those tools in three novel ways. First, we show how existing real-world web applications can be extended without doing traditional programming, using a spreadsheet view. Second, we show how text documents can be gradually enriched into personal software tools using similar techniques. Finally, we demonstrate a new reactive relational data architecture for building complex applications with rich interactions and stringent performance requirements. Together, these projects empower both end users and application developers with simpler tools for developing software.

Acknowledgments

None of the research in this thesis would have been possible without the support of an incredible community of collaborators, friends, and mentors. I'm grateful to:

My advisor Daniel Jackson, who has supported me through my research journey and taught me so much, including how to think crisply, how to write well, how to grow an idea, and how to turn any bureaucratic hurdle into a fun intellectual challenge. Grad school has been a great experience because of him.

My thesis committee members Rob Miller and David Karger, for great discussions and useful feedback.

Josh Pollock, for all the good times and wide-ranging discussions in G-708, and for teaching me so much about diagrams and layout. Nicholas Schiefer, for being my closest grad school collaborator and a great friend. Johannes Schickling, for showing me the value of relentless focus and good API design.

Collaborators at Ink & Switch—Peter van Hardenberg, Paul Sonnentag, Max Schoening, Paul Shen, Alexander Obenauer, James Lindenbaum, Szymon Kaliski, Martin Kleppmann, Marcel Goethals, Orion Henry, Adam Wiggins, Rae McKelvey, and Blaine Cook—for deeply influencing my ideas, values, and process.

Jonathan Edwards, Clemens Klokrose, Shriram Krishnamurthi, Sanjay Sarma, Arvind Satyanarayan, and Alex Warth, for wisdom and mentorship.

Glen Chiacchieri, for inspiring me to do research in end-user programming and helping me get started.

Andrés Cuervo, Josh Horowitz, Steve Krouse, Kevin Lynagh, Slim Lim, Omar Rizwan, Mary Rose Cook, Amelia Wattenberger, Daniel Windham, and all of my computing friends, for helpful advice and energizing discussions online and in person.

MIT friends: Sam Broner, Crystal Lee, Alex Lew, Jonathan Zong, and everyone in MIT HCI, for helping to make grad school fun and engaging.

The wonderful UROPs I had the opportunity to work with: Kathryn Jin, Kapaya Katongo, Gloria Lin, Tyler Millis, and Jessica Quaye.

Santiago Perez de Rosso, for sharing advice early on in my PhD.

The National Science Foundation Graduate Research Fellowships Program (NSF GRFP), the NSF SaTC Program (Award 1801399), Ink & Switch, and RelationalAI for supporting parts of the research in this thesis.

Yoshiki Schmitz, who inspired me and so many others with his fountain of brilliance, and left us too soon.

My parents Misako and David, my brother Henry, and my whole family for supporting my dreams.

And above all, my wife Maggie, who convinced me that I could do this in the first place, and believed in me the whole way.

Contents

1	Introduction	19
1.1	Background	20
1.1.1	Direct manipulation / desktop metaphor	20
1.1.2	Spreadsheets	21
1.1.3	Low-code database platforms	22
1.1.4	Reactive databases	24
1.2	Contributions	28
1.2.1	Wildcard: extending web applications	28
1.2.2	Potluck: enriching text documents	31
1.2.3	Riffle: building reactive relational applications	32
1.2.4	Shared themes	35
2	Design dimensions for reactive databases	41
2.1	Introduction	41
2.2	A simple model of state and views	42
2.2.1	View model	44
2.3	Properties of shared state	45
2.3.1	Reactive	45
2.3.2	Unified	45
2.3.3	Extensible	47
2.3.4	Concurrent	48
2.3.5	Flexible data model	49
2.4	Conclusion	49

3	Wildcard: Customizing Existing Websites	51
3.1	Introduction	51
3.2	Example Scenario	53
3.3	System Architecture	56
3.3.1	Table Adapters	57
3.3.2	Query Engine	61
3.3.3	Table Editor	61
3.4	Vision	62
3.4.1	Decoupling Data from Applications	62
3.4.2	Customization by Direct Manipulation	63
3.4.3	Semantic Wrappers	65
3.5	Related Work	67
3.5.1	Customization Tools	67
3.5.2	Spreadsheets and Visual Query Interfaces	68
3.6	Evaluation: Experience & Limitations	69
3.6.1	Range of Customizations	70
3.6.2	Viability of Scraping	74
3.7	Conclusion and Future Work	75
4	Potluck: Gradually Enriching Text Notes	77
4.1	Introduction	77
4.2	Background	81
4.2.1	The rigidity of apps	81
4.2.2	The flexibility of documents	82
4.2.3	Gradual enrichment	84
4.3	Related Work	84
4.3.1	Text documents as user interfaces	84
4.3.2	Data detectors	85
4.4	Potluck: an environment for dynamic documents	87
4.4.1	Extracting data with searches	89

4.4.2	Running live computations	89
4.4.3	Adding annotations	90
4.4.4	Reusing searches	91
4.4.5	Other features	92
4.5	Evaluation: Experience & Limitations	95
4.5.1	Versatility	95
4.5.2	Tool composition	95
4.5.3	Potluck vs. spreadsheets	96
4.5.4	Challenges of parsing	98
4.5.5	State and UI in text	99
4.5.6	Limitations	99
4.6	Future Work	101
4.7	Conclusion	102
5	Riffle: Reactive Relational State for Local-First Applications	105
5.1	Introduction	105
5.2	Background	107
5.3	Related work	110
5.4	Key Concepts	113
5.4.1	Reactive relational queries	113
5.4.2	Synchronous transactional updates	115
5.5	System Implementation	118
5.5.1	Relational Database Backend	118
5.5.2	View Framework	119
5.5.3	Reactivity Algorithm	119
5.5.4	Query languages	120
5.5.5	Dynamic query generation	120
5.5.6	Query scope	120
5.5.7	Local component state	121
5.5.8	Performance architecture	121

5.5.9	Debugger	122
5.6	A simple example: Todo List App	122
5.7	Evaluation: Experience & Limitations	125
5.7.1	Case study: Music Application	125
5.7.2	Heuristic evaluation	136
5.7.3	Limitations	137
5.8	Future work	139
5.9	Conclusion	140
6	Conclusion	143
6.1	Key Ideas	143
6.2	Future Work	144
6.2.1	Towards data-centric interoperability	144
6.2.2	The role of AI	146
6.3	Conclusion	149

List of Figures

1-1	Command language vs direct manipulation for navigating a filesystem	20
1-2	A spreadsheet provides a unified substrate for both user input and domain data.	22
1-3	Airtable demonstrates common features of reactive database tools. . .	25
1-4	Object Spreadsheets [54] provides a spreadsheet-style interface with a hierarchical view of a relational data model, bound to a UI	27
1-5	SIEUFERD [5] allows the user to specify relational joins, sorting, and spreadsheet-style formulas, all in a direct manipulation editor.	27
1-6	Wildcard enables the user to edit an existing website through a table view.	29
1-7	Customizing Hacker News by interacting with a table view	30
1-8	Left: the Potluck user interface with a text note and reactive tables. Right: use cases including managing a cash register, tracking plant watering schedules, and running a meeting agenda	32
1-9	An overview of the Riffle architecture. The UI visualizes the results of a reactive graph of relational queries on a persistent client-side relational database. The dataflow loop runs synchronously on the UI thread, supporting fast, transactional reactivity. In the background, the local relational database is synchronized with other data sources over the Internet.	33
1-10	Examples of some of the SQL queries and view definitions used to power key features of the Overtone music manager application built in Riffle.	34

1-11	An overview of how Wildcard, Potluck and Riffle make use of shared state in different contexts	36
2-1	Coordinating two views through shared state with unidirectional dataflow	43
2-2	A view v_n can depend on a view model s_n in addition to the shared state: $v_n = f_n(S, s_n)$	44
2-3	Using Wildcard to customize Hacker News entails adding a table view of the articles in the webpage, new state for annotating the articles, and new computed columns for calculating read times.	47
2-4	Three forms of extensibility: adding state, adding derived computed results, and adding a new view	48
3-1	An overview of data-driven customization	52
3-2	Customizing Hacker News by interacting with a table view	54
3-3	The table adapter architecture	56
3-4	Source code for the Hacker News scraper. Some details removed for brevity.	59
3-5	Sorting the used sellers page on Amazon by total price, including fees. The original page doesn't have sorting, and doesn't show the combined price.	71
3-6	Organizing takeout restaurants on Uber Eats by delivery ETA and price	72
3-7	Taking notes on Instacart grocery items, after sorting them by price .	73
3-8	Using a custom text editor widget to edit a blog post on Blogger. The text is synchronized with the Blogger editor through a table cell. . . .	73
4-1	The Potluck interaction model forms a loop: extract data from text, compute with that data, and then display results back in the text. . .	78
4-2	A coffee recipe in Potluck, with a slider for scaling the number of servings	79
4-3	Potluck documents can help with running a cash register, planning a meeting agenda, and tracking a plant watering schedule.	80

4-4	Beyond the core recipe functionality, Paprika’s sidebar has extra features for Groceries, Pantry, Meals, and Menus	82
4-5	Each meal plan entry in Paprika must be assigned to a specific date on the calendar, with no room for ambiguity.	83
4-6	Text documents are a single versatile medium for recording all kinds of information.	83
4-7	Coda supports enriching text documents with interactive computation	85
4-8	A reactive document by Bret Victor explaining a tax policy change. The user can edit values by dragging, and other dependent values in the text automatically update.	86
4-9	A data detector in macOS enables right-clicking on a phone number to add it to contacts or make a phone call.	86
4-10	Creating an interactive quantity scaler for a coffee recipe in Potluck .	88
4-11	Potluck offers several annotation locations: above the text, next to the text, or replacing the text.	90
4-12	It’s more convenient to follow a recipe when the quantities are shown inline in the directions.	93
4-13	A spatial query that finds the quantity of an ingredient in the directions.	94
4-14	The same unit conversion computation, in Potluck on the left and a Notion Table on the right	97
4-15	A pizza dough recipe that computes flour and water amounts based on input parameters	97
4-16	Text notes often contain implicit structure and relationships expressed through a personal micro-syntax.	98
4-17	Showing a calendar view of a workout note in Potluck	101
4-18	Extracting the ingredients from a recipe using GPT-3	102

5-1	An overview of the Riffle architecture. The UI visualizes the results of a reactive graph of relational queries on a persistent client-side relational database. The dataflow loop runs synchronously on the UI thread, supporting fast, transactional reactivity. In the background, the local relational database is synchronized with other data sources over the Internet.	106
5-2	A comparison of different approaches to managing reactive UI state. Riffle simplifies reactive dataflow by offering a single performant reactive loop for managing the entire state of a user interface, including both UI state and domain state.	112
5-3	In a single-page web application, user interactions frequently incur network latency and leave the UI in a temporarily inconsistent state. In contrast, Riffle’s local-first architecture and synchronous transactional updates enable faster responses. The UI can respond to the interaction immediately without showing inconsistent loading states because the data was synchronized to the client before the user explicitly requested it, and database queries are efficiently updated within 16ms.	116
5-4	Implementation architecture: The Riffle library sits between React (for view templating) and SKDB (for all data storage and queries)	118
5-5	The Riffle debugger shows a live view of the data in the underlying database. Other tabs (not shown) include the current reactive queries and an interactive SQL console.	123
5-6	TodoMVC includes a simple example of a dynamic SQL query. The currently active filter setting is queried from a table using a SQL query. A JavaScript query then turns that value into a filter clause in a SQL string, which in turn queries the todos table to produce the final filtered data for the view.	126
5-7	Examples of a SQL query and view definition used in the Overtone music manager.	128
5-8	A debugger that shows recent updates in the Riffle reactive graph . .	131

6-1	Haiku OS stores a list of contacts using structured attributes on files, enabling them to be managed through a generic database view	145
6-2	A spreadsheet table generated by GPT-3 after the user typed “double the quantities”	147

List of Tables

3.1	A list of customizations that we have implemented using Wildcard. . .	69
5.1	Time taken to update materialized view in response to inserting 1 new track (ms)	136

Chapter 1

Introduction

The research in this thesis originated from a simple question: “why can’t making apps be as easy as using a spreadsheet?” Working at an education technology startup, I had seen how non-programmers were able to use spreadsheets to create simple tools specialized to their unique needs. Internally, our operations staff could make metrics dashboards and project plans using spreadsheets; in the school districts we worked with, principals and teachers could create attendance trackers and curriculum plans. And yet, for these people, the idea of making even a small modification to a rich web application seemed completely daunting—and rightfully so, given the complexity of the tools typically used to build such applications today. I observed how this gap created a power dynamic where programmers were able to strongly influence the direction of our software, but other stakeholders were mostly relegated to filing feedback requests. In a world where software increasingly defines how our systems operate, it felt wrong for so many people to be excluded from building and customizing their own tools.

In this thesis, we¹ present three systems which each contribute to allowing everyone to create and edit software with the ease of using a spreadsheet, by drawing on a rich vein of prior work on direct manipulation interfaces, spreadsheets, and databases. In **Wildcard**, we show how users can use a spreadsheet-like reactive table interface to extend and modify the behavior of an existing web application, like reorganizing the prioritization of articles on a news site. In **Potluck**, we show how reactive formulas can be layered over text notes to enrich them with interactive behavior, like scaling ingredients in a recipe. Finally, in **Riffle** we demonstrate how the simple reactivity model from spreadsheets can scale to help developers build complex high-end user experiences, like a music library manager.

Throughout all these tools, the patterns of fast reactive updates, live views of data, and direct manipulation interactions enable simpler approaches to specifying computational behavior than traditional programming tools can offer. These projects contribute towards a future where the creation and tailoring of software is not limited to a programmer elite, but is democratized for all.

¹Each of the systems presented in this thesis was developed together with collaborators; see footnotes at beginning of Chapters 3, 4 and 5 for more details.

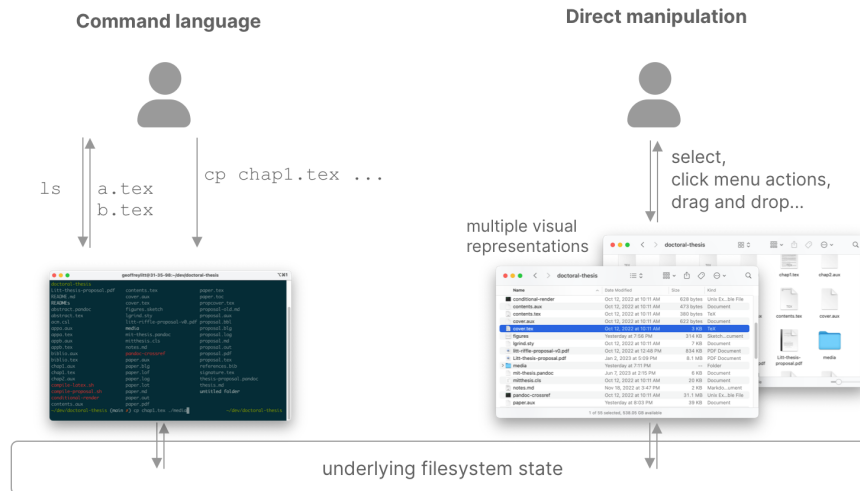


Figure 1-1: Command language vs direct manipulation for navigating a filesystem

1.1 Background

This thesis draws on three areas of work with a deep history in computing—direct manipulation interfaces, spreadsheets, and low-code database toolkits—as well as more recent work that combines influences from these areas. In this section we present a brief overview of this landscape.

1.1.1 Direct manipulation / desktop metaphor

A turning point in the history of interactive computing was the invention of *direct manipulation* [73]. In the early days of interactive computing, users would issue obscure commands in a terminal, which would operate on an opaque system state hidden from the user. In direct manipulation, this “complex command language syntax” is instead replaced by “visibility of the object of interest” — providing a more natural metaphor for viewing and modifying the relevant state of the system.

Take the simple example of manipulating files in a filesystem (Figure 1-1). In a terminal, the state is hidden by default; the user must issue commands to query system state, such as `ls` to list files in a directory or `cd` to change directories. The user must remember the syntax for these commands, as well as relevant arguments like filenames. In contrast, in a visual file navigator such as Finder in Mac OS, the user can always see a visual representation of the system state. The system provides multiple views which are each useful for different tasks: a table for seeing a dense view of metadata across files, a desktop for spatially organizing file icons, and more. The user can take actions like switching directories or moving files by clicking on icons and menus which reveal the possible space of actions and make it easy to visually select objects to act on.

Of course, the views available on a desktop OS go beyond the generic views in the file navigator—*applications* running on the OS provide more domain-specific views

which are specialized to particular tasks. For example, a textual code file can be opened in a text editor like vim, or in an IDE like Visual Studio Code. Each application provides different visualizations and edit actions over the data; for example, vim supports programmable macros for editing text, while Visual Studio Code displays syntactic and semantic hints about the meaning of the code.

One key property of the desktop filesystem is that it *decouples data from applications*. The filesystem makes the same shared data available across different tools, which enables useful workflows such as collaborators editing the same data in their preferred editors (e.g., each programmer on a team using their own preferred code editor), or passing the same file between tools that handle different parts of a workflow. We take this property for granted on filesystems, but it is not universal: modern web and mobile platforms silo data within individual applications by default, creating a much tighter coupling between the underlying data and a particular way of viewing and editing that data. One theme of this thesis is that liberating data from a default application view and allowing alternate representations of that data, as the desktop filesystem does, is a powerful technique for empowering users to tailor their computing experiences to their specific needs.

1.1.2 Spreadsheets

Another canonical example of a direct manipulation interface is the spreadsheet. In a spreadsheet, the main information shown on the screen is the domain data that the user cares about, such as sales figures or course offerings. The user can easily view and navigate by performing visual selections, and can perform edits via “rapid, reversible, incremental actions” [73].

In addition to supporting these basic direct manipulation interactions, spreadsheets are a famously successful example of an environment for *end-user programming*: supporting people in programming to complete a specific personal task, rather than to create software for broader public use [41]. Spreadsheet formula languages provide an easy on-ramp to specifying computations by using a stateless functional programming model. They include built-in functions like `SUM` and `AVERAGE` which map to common tasks in the domains like accounting that users care about. Formulas are always written in terms of a single cell with concrete data, and abstraction is achieved by copy-pasting formulas with relative references between cells. All of these qualities make the spreadsheet more user-friendly than traditional programming environments that typically require writing abstract commands without visibility of concrete data.

Another critical design element of spreadsheets is *fast reactivity*. The user defines formulas with inputs from other cells in the sheet; dependencies between computations are automatically tracked, and downstream values are instantly updated whenever data or formulas change. Reactivity enables a *declarative* interface, since the user can focus on the higher-level structure of their computation without needing to tediously think about propagating change between computations.

There is one more powerful aspect of spreadsheets that is often overlooked: a spreadsheet provides a *unified data substrate* that can store all data relevant to the user’s problem, including both “domain state” related to the domain at hand, as

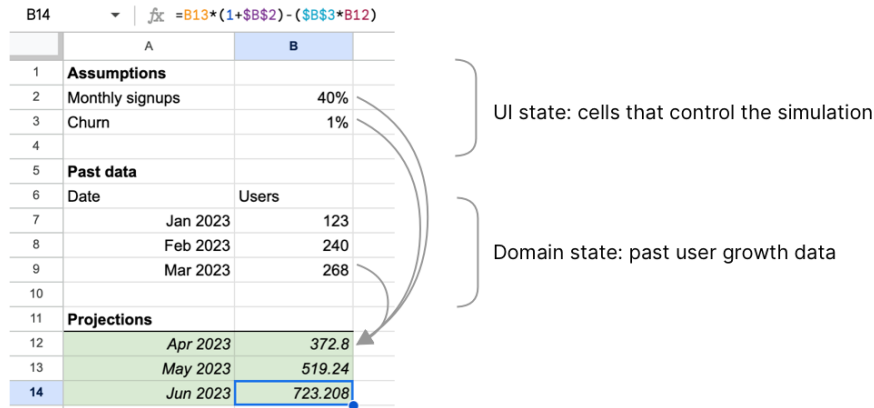


Figure 1-2: A spreadsheet provides a unified substrate for both user input and domain data.

well as “input state” resulting from user interactions. For example: a spreadsheet projecting user growth for a startup might contain past user signup data—stored data which the user is unlikely to edit—as well as input cells like an expected future growth rate, which the user edits to control the simulation (Figure 1-2).

The unified data substrate is easy to take for granted in spreadsheets, but it offers a number of benefits which are surprisingly difficult to achieve in other application development platforms like multi-tier web applications. In a spreadsheet, the user can be assured that all values in the sheet are always up to date, since changes to either the input state or the domain state flow through the same fast reactivity model. Because user input is treated as just another form of data, it is easily persisted alongside domain data. The results of a simulation run can be shared in a spreadsheet file which contains the relevant inputs; alternate scenarios can be saved by creating duplicates of the sheet with new inputs. In the Riffle system, we explore these benefits in more depth and explore how to bring them to full-fledged application development.

Despite the widespread success of spreadsheets, they also have limitations that prevent certain kinds of usage patterns. Spreadsheets encourage flat, denormalized data representations, and have limited tools for representing complex datasets with associations between different kinds of records. Spreadsheets also lack support for building custom interactive interfaces over a dataset, such as a form view for entering a new record, or a report that groups and aggregates data. Next, we turn to another kind of tool which specializes in precisely these kinds of more advanced features: low-code database platforms.

1.1.3 Low-code database platforms

The relational model [21] has several key benefits that have made it a popular way of representing state in computing for many decades. It is flexible and relatively agnostic to how queries are shaped, since normalized data stored in tables can be efficiently queried in many different ways. It provides an abstract data model which

is decoupled from the specifics of how the data is actually stored. It also has a global model where all data in a database can be queried together with a single query language, rather than encapsulating data behind object APIs. And decades of research into relational query languages like SQL has resulted in databases that can efficiently execute declarative relational queries.

Many software applications consist primarily of Create, Read, Update, and Delete (CRUD) operations over a relational database. A CRUD app typically lets users see lists of records (commonly with query options like sorting, filtering, grouping, and joining across tables), see details of individual records, use forms to create and edit records. These basic CRUD operations, when layered over a relational model of the domain at hand, can help manage data in a vast variety of domains. For example, a course catalog app, a customer relationship management app, and a bug tracker might all primarily consist of CRUD operations.

Such applications can be built by professional software teams using frameworks like Ruby on Rails, but there are significant drawbacks in cost and complexity. Building a modern CRUD application requires grappling with the complexity of a vast array of technologies: databases, backend web server programs, designing APIs, building rich frontend applications in JavaScript, and more. This process is cost-prohibitive for building smaller tools; a task like managing personal chores, or handling logistics for an amateur sports team, or assembling playlists for a dance session, often cannot justify the development and maintenance costs of building a fully custom application. And even if such an application can be built from scratch, it will often be missing key functionality or polish.

In response to these challenges, a variety of “low-code” interfaces have aimed to enable users to build these kinds of CRUD applications while avoiding the full complexity of traditional programming. Examples of commercial products in this category include FileMaker, Microsoft Access, and Retool. Enterprise resource planning systems like SAP, which handle enormous databases with complex configuration, also arguably fit qualify as low-code database platforms. All of these products offer visual GUI tools which can manage schemas and data and create forms with custom layout and validations. They enable end-user developers to visually author views of individual records, as well as construct aggregated reports over parts of the database. Together, these features make it more efficient to build many parts of a typical CRUD application than using traditional code.

At the same time, these tools for constructing database-backed CRUD apps are often lacking in several areas relative to spreadsheets. First, fast reactivity is often not present—in many CRUD application architectures, updates propagate more slowly and often require manual refreshes to propagate between views. Also, direct manipulation is less pervasive than in spreadsheets—for example, complex database queries must be authored as SQL code, rather than by directly manipulating the data.

In the next section, we explore more recent work which aims to improve on these limitations by marrying the best parts of database toolkits and spreadsheets.

1.1.4 Reactive databases

As we have described above, spreadsheets and low-code database toolkits have been a part of interactive computing for decades. More recently, a number of systems have explored combining the best aspects of both. This work has used terms such as *spreadsheet-driven applications* [8] and *visual query systems* [5]; we refer to this broad category of ideas as *reactive database* tools, because they combine ideas from database toolkits with the reactive computing model from spreadsheets.²

While it is difficult to define an exact boundary for what constitutes a reactive database system, a useful illustrative example is *Airtable*, a popular commercial end user programming platform used for tasks like project management, CRM, and marketing planning. It allows users to visually define relational schemas, with column types ranging from general ones like text, numbers, and dates to more specific ones like emails and phone numbers. Users can edit the data and the schema in a web browser using a generic table editor UI.

Like a spreadsheet, the table view in *Airtable* puts data in the foreground and provides direct manipulation interactions to edit the data and schema as well as specify computations. A major difference is that the underlying data model is not a freeform 2D grid, but is instead a relational schema with strict column types and explicitly modeled relationships between entities. Using this table view, users can specify *views* (i.e., queries) over tables—including sorting, filtering, grouping, and selecting subsets of columns—all using direct manipulation interactions. They can also add *computed columns*, which contain formulas that compute derived values. Unlike a spreadsheet, the formulas are automatically applied once per record in the table; there is no need to manually copy the formula to every row. Like a spreadsheet, computed values update reactively when underlying data changes. This table view is a sort of hybrid between a spreadsheet and a database. The enforced relational structure prohibits many common uses of spreadsheets, e.g. for constructing arbitrary data layouts in a 2D grid, while the structure simultaneously makes it easier to enforce data integrity and perform rich queries without making mistakes. Figure 1-3 shows some of the key features of the *Airtable* table view.

In addition to the table editor view, *Airtable* comes with several built-in user interfaces for editing and visualizing the data in the relational database. One particularly important kind of view is *forms* for collecting user input that adds rows to the database. Other views include calendars, maps, galleries of cards, kanban boards, timelines, and Gantt charts, which are all fairly generic views which can apply to many domains. All of these views update automatically and reactively when the underlying data changes. In addition, users have tools to visually define their own custom item view layouts and simple interactive UIs on top of the database.

Airtable shows some of the key characteristics of the genre of reactive databases: a structured data model, a table view that supports viewing, editing and querying the

²We avoid using the word *spreadsheet* in our name for this category because the term carries connotations of a freeform 2D grid without a data schema and with only scalar formulas. Many of the tools in this category enforce a stricter schema on the data and support creating formulas that automatically apply to multiple records.

Data can be grouped by a property using direct manipulation interactions

Schema can be edited visually. User can add new properties for storing data, and computed properties using formulas

Relationships between records are supported as a first-class primitive

The screenshot shows the Airtable interface for a 'Sales CRM' database. The table is grouped by 'STATUS' into two sections: 'Qualification' (4 records) and 'Proposal' (7 records). The columns include Opportunity name, Status, Priority, Estimated value, Account, and Primary contact for. The 'Estimated value' column shows aggregate sums for each group: \$41,054 for Qualification and \$145,830 for Proposal. A total sum of \$478,691 is shown at the bottom. Annotations with arrows point to: 1) the 'Grouped by 1 field' dropdown, 2) the 'Status' column header, 3) the 'Primary contact for' column, and 4) the 'Sum \$478,691' aggregate value.

Opportunity name	Status	Priority	Estimated value	Account	Primary contact for
Qualification (Count 4) Sum \$41,054					
1 BPS Pilot	Qualification	Medium	\$10,000	Bear Paw Solutions	Rose Fowler, Bear Paw Solu
2 Acetube inquiry	Qualification	Medium	\$15,133	Acetube	Victoria Porter, Acetube
3 Timbershadow expansion	Qualification	Very high	\$6,154	Timbershadow	Scott Brewer, Timbershadow
4 LKS req	Qualification	Medium	\$9,767	Leonard Krower & Sons	Lori Dixon, Leonard Krower
Proposal (Count 7) Sum \$145,830					
5 BPS second use case	Proposal	Very low	\$24,791	Bear Paw Solutions	Rose Fowler, Bear Paw Solu
6 Galerprises renewal	Proposal	Low	\$23,205	Galerprises	Judith Clark, Galerprises
7 EYS renewal	Proposal	Low	\$23,503	Edge Yard Service	Olivia Guzman, Edge Yard S
8 Payless inbound	Proposal	Low	\$20,999	Payless Cashways	Lauren Chavez, Galerprises
9 Robinetworks renewal	Proposal	Medium	\$24,692	Robinetworks	Judith May, Robinetworks
10 SI expansion	Proposal	Low	\$12,687	Sunlight Intelligence	Michelle Torres, Sunlight Int
28 records Sum \$478,691					

Simple aggregate computations like sums can be configured visually

Figure 1-3: Airtable demonstrates common features of reactive database tools.

data; a scalar formula language for computing derived columns on the table; other rich views over the tabular data; and reactive updates throughout the entire tool. Similar tools have appeared in other popular commercial products like Coda, Notion, and Glide, making this set of features almost a kind of common standard for modern end-user programming interfaces.

Beyond this foundational set of features, a variety of research projects have explored related and adjacent ideas in this space that push the boundaries of the approach.

Two projects that enable the creation of interfaces over structured databases are Exhibit and SIEUFERD. Exhibit [33] contributes a framework for easily publishing interactive pages that let users navigate and query structured datasets made up of objects with simple property-value pairs. It supports both simple table views as well as richer views like maps and timelines, and offers a smooth gradient from declarative HTML views to more custom JavaScript logic. Exhibit demonstrates the power of generic tools and views for querying and visualizing data from any domain. However, it does not aim to cover the full range of use cases for a full-fledged CRUD application, instead prioritizing simpler ways of navigating and querying information. SIEUFERD [5] proposes an interface that can support a very wide range of queries over a relational database—including joins, aggregations, and scalar formulas—all via direct manipulation of the data, and without writing SQL by hand (Figure 1-5). In a sense, it can be seen as an extension of the reactive table concept from Airtable to a much more expressive set of queries. However, SIEUFERD does not support updating the data or the schema.

Several other projects originate closer to the perspective of spreadsheets. Quilt [8] uses spreadsheets most directly: it offers a model for building *spreadsheet-driven web applications* composed of an HTML view bound to a web-based spreadsheet, which can contain both scalar values and tables of data, and which also defines reactive computations. Gneiss [18] takes a similar approach, driving an interactive web application interface from a reactive spreadsheet editor that stores data and defines computations. Object Spreadsheets [54] takes a related approach but goes beyond the standard spreadsheet model, proposing a new computational model and formula language that combines the simplicity of spreadsheets with a more structured tabular data model (Figure 1-4). It offers techniques for handling nested tables, as well as using a spreadsheet-style interface to lay out a user interface. Finally, Mavo [78] weaves spreadsheet-style reactive computations directly into HTML view definitions. A key difference from most of the other systems listed above is that Mavo defines the data schema directly based on the shape of the user interface, foregoing a separate step for data definitions or a separate spreadsheet interface for raw data editing.

These projects differ along many dimensions: the breadth of rich interactive behavior, spanning from simple table views to full-blown applications; the amount of structure imposed upon the data, ranging from strict relational schemas to freeform spreadsheet data; and the degree of separation between the data definitions and the user interface. Yet they all illuminate aspects of a shared path: making it easier for people to author interactive data-centric applications by using techniques inspired by spreadsheets and databases. Key themes that appear in these projects include direct

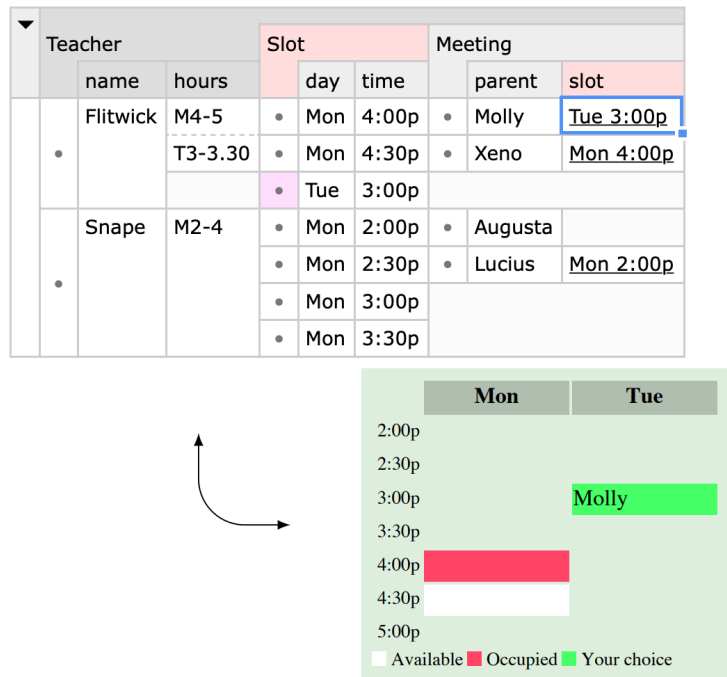


Figure 1-4: Object Spreadsheets [54] provides a spreadsheet-style interface with a hierarchical view of a relational data model, bound to a UI

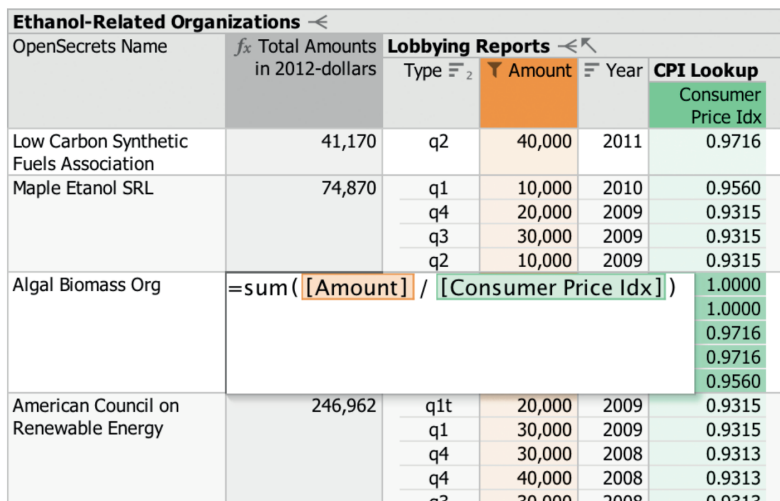


Figure 1-5: SIEUFERD [5] allows the user to specify relational joins, sorting, and spreadsheet-style formulas, all in a direct manipulation editor.

manipulation views of system data, pure functional formulas for scalar computations, automatic propagation of reactive updates, and relational data modeling. All of these ideas make prominent appearances in the systems we propose.

1.2 Contributions

In this thesis, we build on these ideas of reactive databases and propose three systems that empower end users to create and modify personal software in new ways:

- **Wildcard** enables end users to customize existing web applications through a reactive table
- **Potluck** supports end users in turning text notes into personal interactive tools through a reactive table
- **Riffle** supports developers in building sophisticated user interfaces based on a reactive relational data model, with a live table debugger view

These systems illustrate how ideas from spreadsheets and databases can be brought into new contexts, granting new abilities to developers and end users alike. The spreadsheet-like reactive table view serves as a generic user interface with familiar affordances that people can use to view/edit state and specify computations across all of these different contexts. The overall approach is both conceptually simple and practically useful—through many concrete examples, we show that our systems can support the creation of many software customizations and tools in real-world situations. In this section we introduce the main ideas of Wildcard, Potluck, and Riffle, as well as shared themes among them.

1.2.1 Wildcard: extending web applications

When people want to achieve more tailored behavior in their software, it is often more convenient to tweak an existing piece of software than to build a custom application from scratch. Building a new application would require tediously rebuilding all the functionality of the existing tool, whereas extending and modifying the existing application, in theory, might only require a minor change. This gap between rebuilding and customizing becomes especially clear if the desired change to an existing application is relatively small in scope: changing the color of a button, or sorting a list of results in a different order.

Many tools have aimed to simplify this process of end users customizing and extending existing software. Some systems aim to offer friendly natural language syntax for writing scripts, such as Chickenfoot [14] and Coscripter [45] for the web, and Applescript [22] for desktop. There have also been visual programming environments for customization that forego traditional textual syntax, such as Automator for Mac, Shortcuts for iOS, and Zapier for wiring together web APIs. Although these tools do succeed in simplifying some challenging aspects of programming like rigid syntax, they still maintain a relatively traditional view of scripting: specifying sequences of imperative instructions in a command language.

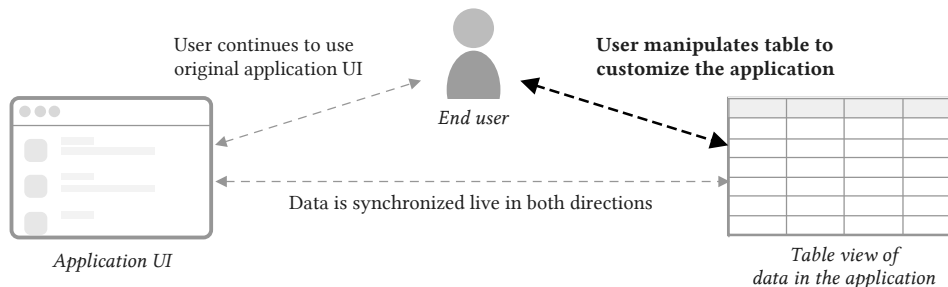


Figure 1-6: Wildcard enables the user to edit an existing website through a table view.

In Wildcard (Chapter 3), we take a different approach. Inspired by principles of direct manipulation and reactive databases, we introduce a reactive table view as an interactive debugger that shows a view of the data underlying a website. Data is scraped out of websites by site-specific adapter code, and bidirectionally synchronized with the reactive table view. The user can then use the table view to manipulate the data in various ways: they can edit values, sort and filter, add annotations in new columns, and write formula computations. All of these interactions are synchronized from the table view to the original website view—sorting and filtering apply to the items in the original UI, and annotations appear in appropriate places within the UI. The reactive table serves as a mediating interface for editing the application (Figure 1-6). The user never writes the kind of imperative code that would be found in a typical browser extension or end-user scripting tools; instead, they achieve the customization through direct manipulation and writing small functional formulas.

As an example of a customization that Wildcard enables, see Figure 1-7. First, the user sorts the list of articles on Hacker News in a different order by clicking on the table header for the desired column. Then, they add expected read times for each article to the web UI, by adding a new column to the table and adding a spreadsheet formula. The new sort order is applied to the page and the read times are displayed next to each article.

The table interactions here are not novel—in fact, they are explicitly intended to be familiar to anyone who has used a spreadsheet, or a reactive table UI like Airtable or SIEUFERD. The innovation lies in tying this reactive database interface to a website DOM so it can be used to customize existing applications, even though the original developer did not in fact expose such an interface.

You may notice a bit of irony in this approach: many websites are in fact built on top of relational databases, and we are essentially reconstructing a fake view that database by scraping data in the user interface. One way to think of Wildcard is as precisely this—a simulation of a desirable future where every web application is built using reactive databases that are exposed to the user for modification. The fact that Wildcard works with existing websites is the key strength and motivating idea of the project, but it is also the source of most of its limitations. For example, often websites contain data in a backend server that is not available in the UI, and therefore cannot be scraped for access in the Wildcard table view. More ambitiously, we might

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.

The screenshot shows the Hacker News interface with a table view. A dropdown menu is open, showing sorting options: 'points ↓' (selected), 'user', 'comments', and 'user'. Below the menu is a table with the following data:

	points ↓	user	comments	user
1	886	nsainsbury	148	user
2	722	zdw	361	
3	461	danfox	127	
4	312	teslademi	95	
5	312	maxbaines	181	

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

The screenshot shows a spreadsheet with formulas for calculating estimated read times. The first formula is `=ReadTimeInSeconds(link)` and the second is `=Concat(Round(user1/60, "min read"), "min read")`.

comments	user1	user2	user3	user4
148	1244			
361	781	13		
127		0		

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

The screenshot shows the Hacker News interface with a table view. The table has columns for 'user', 'comments', 'user1', 'user2', 'user3', and 'user4'. The data is as follows:

	user	comments	user1	user2	user3	user4
1	886 nsainsbury	148	1244	21 min read	looks fun!	
2	3 buchuki	936		16 min read		
3	244 EndXA	98	852	14 min read	read this	
4	67 kqr	14	827	min read		
5	722 zdw	361	781	13 min read		

Figure 1-7: Customizing Hacker News by interacting with a table view

imagine websites *actually* being built from the ground up on top of a unified reactive database that can be exposed to the user; later we explore this direction in depth with Riffle.

1.2.2 Potluck: enriching text documents

The ability to customize an existing application, as in Wildcard, brings a certain amount of malleability to a user’s software experience. And yet, that workflow also starts from the assumption that someone is already using a software application to complete a task.

In practice, many uses of computers fall between the cracks of formal applications. In fact, many times people simply use computers to record data, without needing to perform any computations at all. Joel Spolsky relates a story from his time on the Microsoft Excel team³:

[We] visited dozens of Excel customers, and did not see anyone using Excel to actually perform what you would call “calculations.” Almost all of them were using Excel because it was a convenient way to create a table... Most Excel users never enter a formula. They use Excel when they need a table. The gridlines are the most important feature of Excel, not recal.

The fact that many users do not create formulas should not be seen as a failure. Rather, it is a success that people are able to easily store information in their desired format, and then are optionally able to add structure and computations if needed. We call this workflow *gradual enrichment*. The freeform 2D grid of spreadsheets is critical to enabling gradual enrichment; it also avoids the many pitfalls [71] of forcing people to represent data in formal ways (e.g., specifying strict schemas) too early in their thought process.

In Potluck (Chapter 4), we explore gradual enrichment in the context of a common kind of data which is even more unstructured and flexible than a spreadsheet grid: text notes. Users can start by writing text notes, and then gradually enrich them into interactive applications by extracting structured data from the text into a series of reactive tables. Users can add columns in the tables where they code computations using JavaScript. The results of the computations can be displayed in the original text note as annotation overlays. The system also supports restyling the text based on computational results, as well as inserting basic interactive widgets into the flow of the text. The entire system supports fast reactivity, so updates to the text immediately result in updates to the annotations.

This small set of simple primitives supports building utilities in a variety of personal domains. Potluck documents can scale ingredients in a recipe, create interactive timers to measure baking times or workouts, keep track of deadlines for household chores, and perform basic math to support managing a cash register in a document. For examples of such documents, see Figure 1-8.

³<https://www.joelonsoftware.com/2012/01/06/how-trello-is-different/>

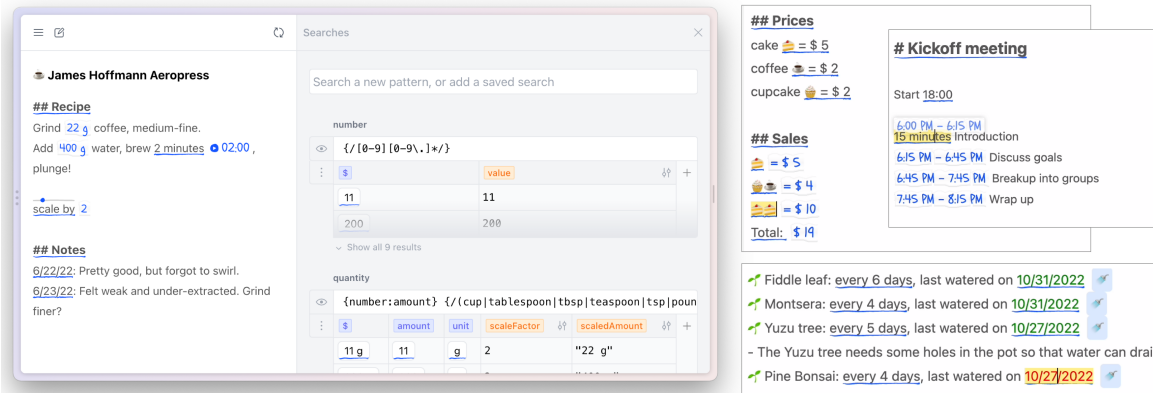


Figure 1-8: Left: the Potluck user interface with a text note and reactive tables. Right: use cases including managing a cash register, tracking plant watering schedules, and running a meeting agenda

The freeform nature of text provides a number of interesting opportunities and challenges. Extracting well-structured data from text requires an ergonomic language for specifying patterns, as well as building in live feedback that encourages users to lightly structure their text in ways that will be amenable to later structuring. We also found that text provides a capable substrate for recording both domain state and UI state, as well as designing primitive user interface layouts within the flow of the text.

1.2.3 Riffle: building reactive relational applications

Many reactive database tools prioritize simplicity over power in order to make it easier for less sophisticated users to successfully build applications within certain constraints. For example, these systems are often limited in the amount of data they can manage, the performance they can achieve, the richness of queries that are available, and the ability to customize visual design. These constraints make sense for ensuring a low barrier to entry, but also make it challenging to build complex applications with more demanding requirements.

On the other hand, the tools used by professional application developers offer a higher ceiling, but in exchange sacrifice much of the simplicity of reactive database tools. In a typical multi-tier web application architecture, developers must manually wrangle copies of data across many layers with different representations and query languages, manually reasoning about maintaining reactive dependencies across process and network boundaries.

Riffle (Chapter 5) aims to bridge the gap between these two ways of building software, bringing the simplicity of “building an application in a spreadsheet” to rich applications with complex design and performance requirements. We apply ideas from reactive databases to provide a novel approach to state management we call the *reactive relational model*, which combines spreadsheet-like reactivity with the ability to easily specify relational queries. All application and UI state is stored in a local

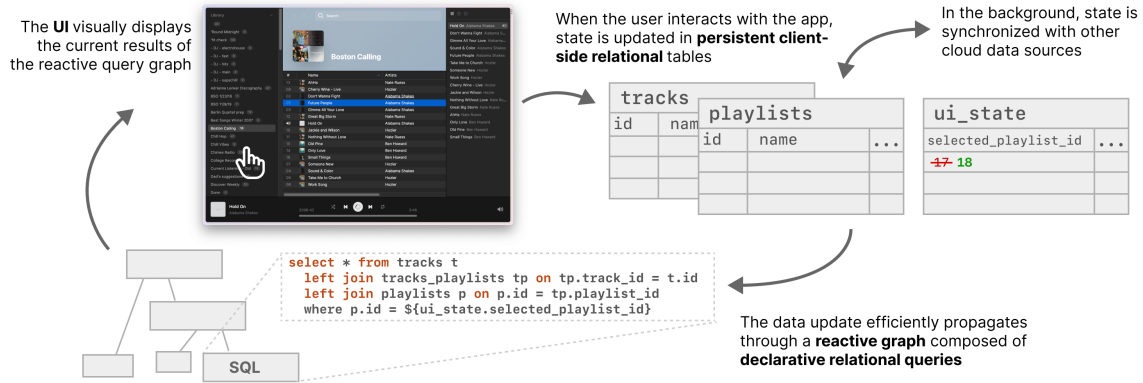


Figure 1-9: An overview of the Riffle architecture. The UI visualizes the results of a reactive graph of relational queries on a persistent client-side relational database. The dataflow loop runs synchronously on the UI thread, supporting fast, transactional reactivity. In the background, the local relational database is synchronized with other data sources over the Internet.

client-side relational datastore, connected to a user interface by fast reactive bindings (Figure 1-9). In the background, the client-side store can be synchronized across devices via a network connection.

An essential benefit of this architecture is that it makes it more convenient to build web applications by removing many of the typical intermediate layers in the stack. It also enables the developer to build their entire application out of declarative relational queries, which are automatically maintained reactively. Because all domain state and UI state is managed in a single relational database which can be queried together, the developer benefits from a simplified mental model, and the end-user benefits from fast reactivity and consistent UI throughout the application, just like in a spreadsheet.

Riffle also has a live table debugger view which visualizes the state of the underlying database as well as the currently running reactive queries and their results, and also enables the user to prototype relational queries live. As with other reactive database systems, the goal is to provide visibility into the system through familiar interactions with a table view. Once an application has been built using the reactive relational model, the table debugger view fits naturally on top.

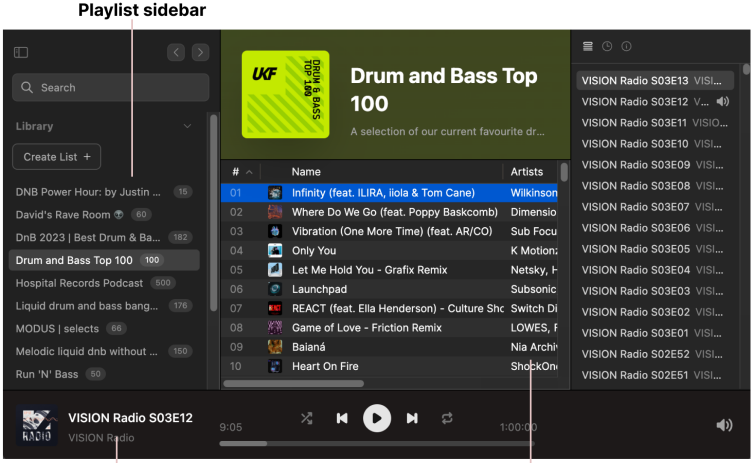
As a case study evaluation, we have used Riffle to build a music manager application called Overtone, which enables users to browse and play music collections synchronized from cloud services such as Spotify. Overtone demonstrates how Riffle can support the construction of an application with a rich data schema and relatively large amounts of data (Figure 1-10). The reactive relational model enables a simple conceptual model for the developer, and fast consistent interactions for the end-user.

The playlist sidebar queries data from an incrementally maintained virtual view which contains a list of playlists with track counts

```
select * from view__playlists_with_track_counts
order by name ASC
```

The definition of the virtual view for playlists with track counts

```
CREATE VIRTUAL VIEW view__playlists_with_track_counts AS
SELECT _p.*, count(_tp.trackId) as trackCount
FROM library_playlists AS _p
LEFT OUTER JOIN view__tracks_playlists
AS _tp ON _p.id = _tp.playlistId
GROUP BY _p.id
```



Playback bar

```
SELECT * FROM playback_state
```

The playback bar runs a simple query to read information like the play/pause state and the offset within the track.

```
select
  addedAtTimestamp,
  trackIndex,
  id,
  name,
  albumName,
  artistName,
  ...
FROM (
  SELECT *
  FROM view__tracks_playlists
  WHERE playlistId = 'playlist-123'
)
ORDER BY trackIndex asc
limit 100 offset 0
```

The Tracklist queries data from an incrementally maintained virtual view which joins together information about tracks, albums, artists, and playlists into a table.

```
create virtual view view__tracks_playlists as
select *
from tracks t
left outer join albums
on t.album_id = albums.id
left outer join tracks_artists
on t.id = tracks_artists.track_id
left outer join artists
on artists.id = tracks_artists.artist_id
left outer join tracks_playlists
on t.id = tracks_playlists.track_id
left outer join playlists
on playlists.id = tracks_playlists.playlist_id
```

dynamic parameters which change based on the current state of the UI

Figure 1-10: Examples of some of the SQL queries and view definitions used to power key features of the Overtone music manager application built in Riffle.

1.2.4 Shared themes

While Wildcard, Potluck and Riffle solve different concrete problems, they all share a similar underlying structure. In each system, state is shared between a relational representation and some other representation like a user interface or a text document. A reactive table view gives users direct access to the relational representation, letting them view and edit the data directly, as well as specify computations over that data. Figure 1-11 summarizes these similarities.

In this section, we explore some shared themes found across all of these projects.

Reactive databases in new places

One important finding is that providing a direct manipulation view of the underlying state in a system helps both developers and users more easily see and modify that state. Benson’s PhD thesis [7] touches on this point:

One way to think about spreadsheet-backed applications... is to think of the spreadsheet as a model with a debugging interface attached. The common Model-View programming pattern does not make any statements about the role of the model, apart from its role as the broker of data. Spreadsheets... [turn] it from [a] programmatic concept into one that is also user-facing for both the developer and the end-user. This means that all data and computation is visible, editable, and debugable in a well known environment, not hidden behind controller code and database APIs.

In all of our systems, we find evidence for the effectiveness of this approach. Making state visible turns many customizations that would have required complex commands over opaque state into a simple matter of direct manipulation. Furthermore, the reactive table provides a particularly useful form of visibility because it is a familiar and generic UI that can be used with any dataset with a relational structure. This versatility rewards investment in making the table UI usable, and rewards investment by users in learning how to use it well. These same qualities can be seen in other generic UIs like text editors which reward deep investment by both their developers and users.

Notably, in all three of our systems, the table view is not new or interesting—in fact, we have intentionally kept it familiar! The novelty lies in bringing that familiar view to new contexts, by creating underlying data models that can be visualized through that view.⁴

For example, in Wildcard, much of the difficulty lies in extracting tabular data representations from web pages and setting up machinery to bidirectionally synchronize that data representation with the original page. Once that underlying model has

⁴Innovation in the table view itself is still useful, but is kept out of scope for our investigations. One example of improving the table view is Bakke’s work [5] on specifying a wider variety of relational queries through direct manipulation of a nested table view.

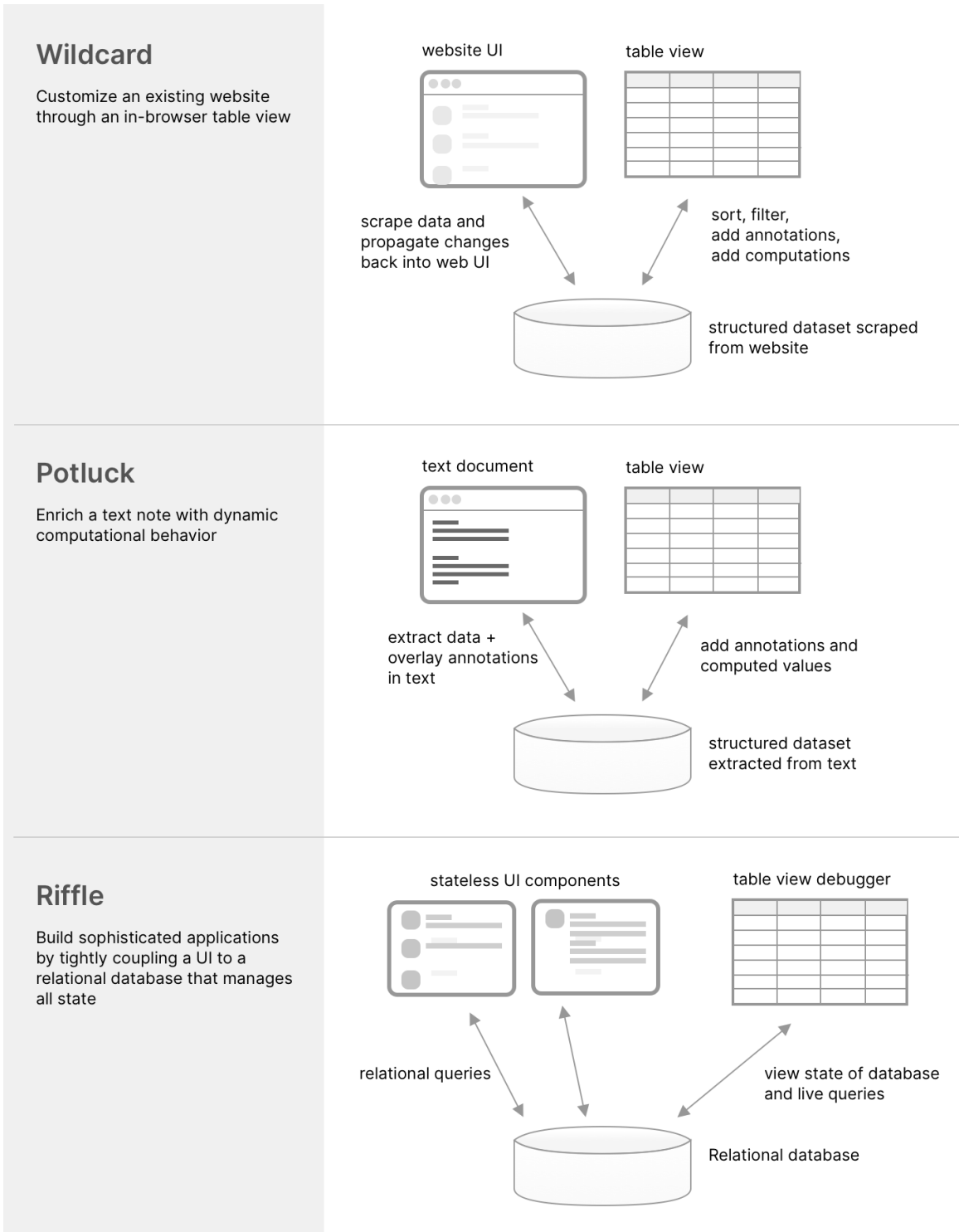


Figure 1-11: An overview of how Wildcard, Potluck and Riffle make use of shared state in different contexts

been established, it is straightforward to add a standard table view UI that visualizes the tabular data, and we can also add a standard spreadsheet formula language to support computations. Similarly, in Riffle, the novel contribution is architecting an entire user interface around reactive relational state; once this model has been established, it is trivial to build a simple debugger.

Our projects have demonstrated that reactive databases can be applied in many different contexts. What makes the pattern so versatile? A large part of the answer is that the relational model is flexible enough to represent a wide variety of data. Also, a flat table is a useful visualization for many kinds of information; It affords very dense overviews and allows for easily comparing the same column across many records.

The power of fast reactivity

Reactivity is a powerful tool for simplifying both users' and developers' mental models. It is one of the best examples of *declarative* programming: specifying the desired end result rather than the details of how to compute it. Users can focus on writing equations that should hold as data changes, rather than reasoning about the imperative details of propagating change. The power of this “value rule” to simplify programming has been known for decades in the context of spreadsheets [36].

One reason it is so valuable for systems to provide reactivity as a foundation is that reactivity is an example of what Ousterhout calls a “narrow but deep” feature [66]. The API contract provided by automatic change propagation is easy to understand, but actually implementing it efficiently can be complicated. As a result, providing reactive synchronization as a built-in platform feature can drastically reduce the amount of work that developers need to think about.

Here are two concrete examples from our projects. First, consider the reactive guarantee provided by Wildcard. A typical browser extension includes imperative code for scraping data from the DOM and making changes to the DOM. The developer must consider issues like how to trigger scraping at the right time in the page lifecycle, or how to handle changes to the DOM that are made by other scripts. In contrast, Wildcard provides a reactive guarantee: the data in the table view is always up to date with the page, and vice versa. The developer does not need to think about how to propagate changes between the table view and the page; they can simply focus on writing the data and computations they want to see.

Second, consider the Riffle project. UI developers have already realized the value of reactivity (see, e.g., early work on UI frameworks like Garnet [60], as well as modern frameworks like React.js), but most approaches offer incomplete reactivity. Traditional application development often involves manually writing logic to fetch data from a server (or local persistence) and synchronize changes back up, which requires tedious work by developers and easily leads to errors. In contrast, in Riffle, developers can simply write queries and the reactive engine ensures the user interface remains up-to-date with the most recent data.

Semantic wrappers

A critical bottleneck for people creating personalized computing tools is extracting structured data from less structured sources. For example, scraping data from websites or authoring regular expressions to parse structure from text are notoriously challenging tasks, especially for users without much programming experience. Our work suggests several strategies for overcoming this bottleneck.

One straightforward solution is for a less skilled user to delegate the task of extracting structured data to a more skilled expert; this is the approach we take in Wildcard, where skilled programmers write site-specific adapter code, and then users can build customizations on top of those adapters. This strategy follows in a long tradition of collaborative end-user programming; for example, Nardi finds that spreadsheet users often cooperate across skill levels, with experts performing a particularly difficult portion of a task [61]. An important assumption made by this strategy is that the extracted data is general enough to support many different kinds of uses and customizations; we find in Wildcard that this assumption often holds true, although sometimes the data to extract depends heavily on the use case.

Another strategy is to promote more reuse and composition in languages for extracting data. For example, Potluck proposes a simple language which allows for composing together smaller named patterns into larger ones; this accommodates system-defined patterns for primitive data types, which can be composed into larger patterns by end users.

One lesson learned from Wildcard and Potluck is that synchronizing data bidirectionally introduces many concerns that aren't present in a one-off one-way data extraction process. For example, bidirectional sync requires making decisions about how edits to a data table should be reflected back in the original webpage or text document.

Looking forward, recent developments in large language models (LLMs) have opened new possibilities for data extraction. Most obviously, LLMs have proven capable at performing a wide variety of structured data extraction tasks from text and other unstructured inputs, which could make it easier to specify scraping logic. LLMs could also assist with authoring or verifying code for performing these tasks. We do not explore these abilities in depth in this thesis, but we briefly discuss future possibilities in Chapter 6.

Towards data-centric interoperability

In today's popular cloud and mobile architectures, data is typically siloed inside of individual applications, and can only be accessed through first-party application clients. This is a stark contrast to the interoperable architecture provided by desktop filesystems, which allow the same file to be edited by multiple tools, and give users more ownership over their data.

There have been a variety of efforts to improve this situation; for example, Tim Berners-Lee has led the SOLID project [52], which aims to give users more control and ownership over their data. However, actually achieving progress towards greater

interoperability is challenging, because it requires convincing developers to build software with a fundamentally new data architecture, while simultaneously convincing users to use software such software. There is a chicken-and-egg cold problem.

The projects in this thesis each contribute a different idea for how to make incremental progress towards a future where data is more decoupled from applications:

- In Wildcard, we avoid the need for developers to change the behavior, by giving users tools to customize existing software from the outside. We simulate a world where data is decoupled from applications and can be visualized in a reactive table view, using browser extension technology and web scraping.
- In Potluck, we start from an existing common data format: text notes. We then leverage that format as a shared substrate on top of which multiple computational tools can coexist. This approach doesn't require convincing anyone to adopt a new data format.
- Riffle helps developers build better apps with a simpler model, even if those developers don't value openness and interoperability. However, at the same time, it lays a foundation for achieving those end goals, by having data locally on-device in a fast reactive database that could be shared by multiple tools.

The systems in this thesis target a variety of goals and users, ranging from end users making small customizations to developers building complex applications from scratch. However, these goals are not separate; they all form part of one continuum. In 1984, Alan Kay envisioned [37] building complex applications like a word processor using familiar, accessible tools like a spreadsheet and paint tool—not because every user would be expected to create their own word processor from scratch, but so that users might encounter a somewhat familiar interface when they decided to “open the hood” of the software they already use to make a small tweak. This idea suggests that making programming easier can benefit both experts building tools from scratch and end-users modifying those tools for their own needs.

Our systems each contribute different ways of helping users open the hood and make changes. Wildcard simulates a simple spreadsheet environment on top of web applications which are in fact often implemented with vastly more complex architectures. In contrast, Riffle offers a new perspective on how to build applications from the ground up with a simpler conceptual model—which helps developers build applications from scratch, but also may provide a stronger foundation for end users to understand and modify the applications they use. Overall, our work shows how declarative approaches to representing application logic, together with direct manipulation views of underlying state, can help both developers building applications as well as end users customizing them.

Chapter 2

Design dimensions for reactive databases

Shared state between views is a key component of the systems presented in this thesis. In Riffle, UI components coordinate over a shared reactive database; in Wildcard, a website UI and a table view coordinate over a shared tabular dataset; in Potluck, a text buffer and a table view are synchronized.

In this chapter, we explore some general considerations involved in designing a shared state abstraction across these varying contexts. First, we present a simple model for thinking about the relationship between multiple views coordinating over shared state. Then, we introduce five properties of a shared state abstraction:

- **Reactive.** To what extent do updates propagate automatically?
- **Unified.** What proportion of state is managed inside a single shared store?
- **Extensible.** How easy is it to extend the shared state with new computations or views?
- **Concurrent.** Can multiple clients simultaneously read/write to the shared data?
- **Flexible data model.** Does the data representation support different shapes of data and access patterns?

Together, these properties define the characteristics of a given shared state model, and the kinds of view composition that are possible over the shared state.

2.1 Introduction

Shared state is everywhere. The idea of coordinating multiple visual interfaces through an underlying shared datastore appears in a huge variety of places throughout computing. It appears in familiar everyday settings, like desktop applications sharing a filesystem, or Airtable components sharing a database. It also appears in more speculative research projects like Webstrates [40], where tools are synchronized through a shared DOM; Dynamicland¹, where programs share a reactive database tied to a

¹<https://dynamicland.org/>

physical space; and Solid [52], where applications share a personal datastore “pod” controlled by the user.

While these examples occur in very different contexts, at different scales and with different particulars of implementation, there is a simple unifying idea underlying all of them: sharing a state object between multiple components is a convenient and powerful pattern for coordination. This is one of the key ideas that makes it useful to build software with reactive databases. At the same time, the implementations of shared state in these systems have important differences which impose meaningful limitations on the kinds of composition that are possible through a given state layer.

We have become so familiar with these limitations that we may not even think twice about them. Consider a small example. Sometimes, when I open an email on my phone, I would like a convenient way to see the same email opened on my desktop email client, so I can easily transition to editing on a bigger screen. Unfortunately, this turns out to be a challenging feature to build, because while the phone and desktop client are both connected to a shared email server, the state representing which email is currently selected is stored only within the client, and not on the server. In this architecture, the choices made about where to store different pieces of state have limited the kinds of composition that are convenient to provide.

This individual example may seem like a minor inconvenience, but it illustrates a broader point: the choices we make about designing and implementing shared state abstractions affect the way we can compose tools. We can also imagine designing new implementations of shared state which make different tradeoffs and open up new ways of building, customizing, and interoperating between software applications.

In this section, we develop a simple, general model of shared state, and describe a set of key design dimensions that help define the kinds of composition that are possible on top of a given state abstraction. Our model is very general, since it is meant to describe a broad variety of systems at different scales. In some cases, the shared state is being accessed within the same device; in others, it might be accessed across a network. In some cases the shared state is an enormous database, in others it may be a tiny in-memory register. Nevertheless, there are common themes across these contexts that are still useful to analyze.

2.2 A simple model of state and views

As a basic foundation, we start with a common paradigm in modern UI engineering called unidirectional dataflow, where we represent our view as a pure function of some underlying state. We can define a state S , a function f mapping the state to a view, and then finally the view itself: $v = f(S)$. For example, S might contain a list of tracks in a music library, and v might be an HTML representation of those tracks displayed in a table.

In unidirectional dataflow, when the user interacts with the view, the view does not directly update—for example, editing the name of a track in an input field does not directly cause a DOM update. Instead, the view expresses an intent to update the underlying state. The UI framework then performs this update to the underlying

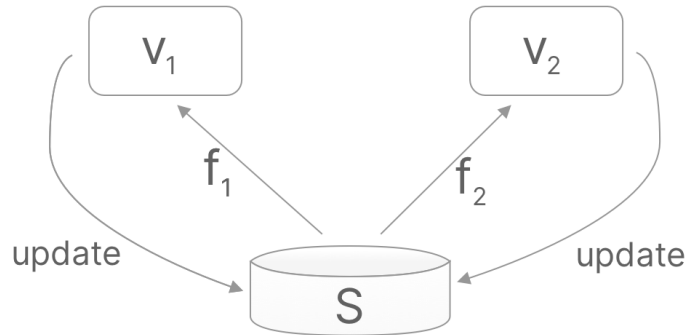


Figure 2-1: Coordinating two views through shared state with unidirectional dataflow

state (perhaps with some scheduling, e.g. batching together multiple updates), and recomputes any views dependent on that state: $v' = f(S')$. To minimize interaction latencies, incremental computation techniques are often used at various stages of this pipeline to efficiently compute the contents of the new view without starting over from scratch.

The reason this model is called unidirectional is that the view did not directly update itself; the update flowed through a loop, first updating the underlying state, which in turn triggered the view to reflect the new state. Structuring the dataflow in this way provides a key advantage: the developer never needs to worry about whether the state and the view have diverged from one another, because the underlying model guarantees that the equation $v_1 = f(S)$ is always maintained. S also serves as a single source of truth for state; so there is never any confusion about reconciling two different copies of state.

This basic pattern of coordinating state between views applies to many concrete architectures. React.js is a well-known example of a view framework that popularized unidirectional dataflow in the web community. Elm is another example of a web framework that uses similar ideas. Even a server-side PHP app can be viewed as a form of unidirectional dataflow at a coarser granularity: an HTML page can request updates by posting HTTP requests to a server; the server then re-renders a view to reflect the new underlying state.

The same unidirectional dataflow pattern easily accommodates multiple views over the same shared state. We can define multiple views each with their own view function: $v_n = f_n(S)$ (Figure 2-1). When the user interacts with either view, an update is triggered on the shared S .

Coordinating multiple views through the same shared state provides the key benefit of consistency: We can always be sure that both views are representing the same state, since any update must flow through the shared S . At the same time, the two views have no dependencies on one another, and there is no need to coordinate di-

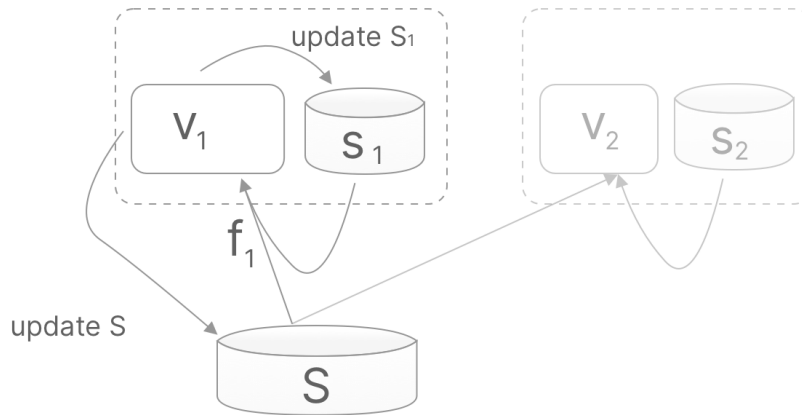


Figure 2-2: A view v_n can depend on a view model s_n in addition to the shared state: $v_n = f_n(S, s_n)$

rectly between the views, e.g. by calling events back and forth; coordination can be achieved through shared state.

This pattern can apply at different architectural scales with different latencies. For example, we might want to show a list of emails in an inbox and a detail view of a single email within an email client—in this case, the shared state is a local database of emails, and the views are UI widgets within the same application. At a larger scale, we might want to use two different email clients to manage the same collection of emails stored on a server—here, the shared state is the server-side collection of emails, and the views are different clients running on different devices.

2.2.1 View model

Many user interfaces have some state which is more closely associated with the view than the underlying domain model. For example, in an email client, this view state might include the currently selected email, the scroll position within an email, or even the hover state of a button.

A common way to handle such state is to extend the pattern above with a *view model*: separate state associated with the view. We can represent this by giving each view v_n its own separate state object s_n . The view function now takes two inputs: $v_n = f_n(S, s_n)$. (Figure 2-2). Some interactions from the view v_n update the shared S , whereas others will update the private s_n .

A view model can be a useful abstraction for indicating the roles of different pieces of state. However, creating strong infrastructure boundaries between the view model state and other domain state can make it harder to coordinate state updates across the two kinds of state. In Riffle (Chapter 5), we demonstrate the benefits of managing view model and domain model state within a single unified model.

2.3 Properties of shared state

We next describe several properties which help characterize any given system where multiple views operate over some shared state, including the systems developed in this thesis.

2.3.1 Reactive

A user interface is *reactive* if updates to the system’s state automatically result in corresponding updates to downstream views. A spreadsheet is one example of a reactive interface. In contrast, a traditional server-rendered app web is not reactive; even if the data in the database changes, the user must actively refresh the page to request an updated view.

Reactivity with live feedback enables more powerful forms of composition through a shared state mechanism. For example, consider editing the same text document in two instances of the Google Docs editor—each keystroke appears reactively across the editors, so users can collaborate in realtime on a shared document across their respective editors. In contrast, if two processes are editing the same text file on a desktop filesystem, often a text editor GUI will not reactively display updates to the file; the user must manually reload the file from disk.

One important component of reactivity is latency: faster reactivity enables more unified experiences. Typically, a tightly coupled set of components within a single application has lower latency and more reactivity than components spanning across multiple different applications (e.g. multiple applications coordinating through a filesystem). However, this correlation is not a law of nature; it is possible for an open-ended, interoperable system to support fast reactivity across more loosely coupled components. One example of this principle in action is Webstrates [40], which allows multiple UI views to coordinate through a shared, synchronized DOM tree in a browser.

Reactivity is an essential component of all of the systems presented in this thesis. In Wildcard and Potluck, reactive synchronization of a text document or a web UI with a table view is essential for providing immediate feedback about how updates are reflected across the two views. In Riffle, we develop a uniquely comprehensive reactivity model where the dataflow loop passes through a persistent database, extending the benefits of unidirectional dataflow beyond their typical limits.

2.3.2 Unified

Earlier we discussed how some architectures incorporate a *view model*: state associated with the view (and usually local to each view), separate from a shared domain model. Having private view models for each view results in a less unified datastore because there is state that is not shared between all views. There are some reasonable justifications for this kind of fragmentation. One reason is to separate concerns between view state and domain state. Network latency may also motivate the split:

if domain state is slow to update across a network, we must manage view model state (e.g., the text in a text box) in some separate way that provides lower latency.

However, a different approach is to combine all state in the system into a single shared store. We refer to this a *unified* approach to state management: managing all state in a single shared place. Instead of each view v_n having its own private state s_n , we combine all state for the system into the shared state S .

Of course, it is still important to have some way to associate view-model state with individual views or parts of views; we can achieve this by keying parts of the state within S on identifiers for the views. The key difference is that these keyed pieces of state are still all managed in a unified system.

At first glance, it may seem as though we have simply removed a useful separation of concerns by unifying state. So what have we gained from this approach?

Managing all state in a single database simplifies the architecture in a number of ways, which are all exemplified in Riffle:

- We can query all the state together using one language; for example, we can write SQL queries that span across UI and domain state.
- We can query it *transactionally*, avoiding mismatches between UI state and domain state.
- Requirements like persistence or sharing can be treated more uniformly across all state—if we want to persist UI state or share it with other users, we can simply flip a configuration switch, instead of needing to migrate state to an entirely different system.
- We can use a single debugger to view and manipulate all state in the system.

There is another more subtle benefit to unified state management: it supports extensibility and makes it easier for other tools to manipulate any state. By construction, all views v_n depend solely on the shared state S . This means that any tool which can edit S can drive any kind of change in any view, solely by editing data, and without needing to directly interact with the view itself. For example, an automation that enters text into a text box, or changes the sort order of a list in a UI, can be built by writing a program that edits S , without needing to click buttons or simulate keystrokes.

This ability to control a view through a shared state representation makes appearances in all three systems described in this thesis. In Riffle, an external process writing to a Riffle database can control UI state like the play/pause state of a track in a music player. In Wildcard, a table debugger view is used to control the sorting and filtering of a webpage UI. (In this case, we *simulate* a unified state model on top of an existing website). In Potluck, most state is unified within a text file, so any process which can manipulate the text—such as an automated script, or even manual text operations like cut/copy/paste—can manipulate any of the state in the system.

One challenge with a unified state model is preserving modularity. If all components can read and write to the same global shared state, this makes it easier to introduce undesirable coupling between those components. Our experience with Riffle has been that this challenge can be managed by coding conventions (e.g. being careful

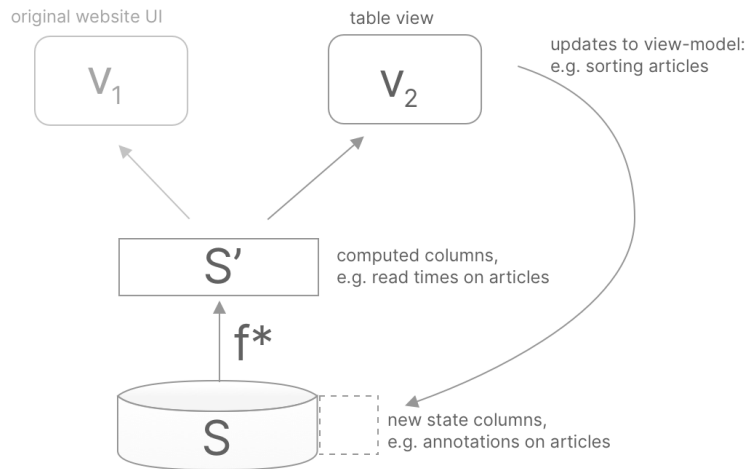


Figure 2-3: Using Wildcard to customize Hacker News entails adding a table view of the articles in the webpage, new state for annotating the articles, and new computed columns for calculating read times.

about editing the “private state” of a view component stored in a global database), and the benefits gained from a unified store outweigh the costs.

2.3.3 Extensible

Extensibility is a key property for supporting open-ended use and customization. As a simple example, a desktop filesystem is an extensible system, since any application running on the operating system can manipulate the files on the computer. If a user wants a different view of the data in some application, there is always a possibility of opening the saved file in another application. (In practice this may be difficult depending on the file format, but at least the data can be technically accessed.)

In contrast, cloud applications tend to be less extensible. By default, the user can only interact with first-party views created by the application’s developers. Sometimes a public API may exist that allows for the creation of third-party views, but often such APIs are limited in scope.

Using other views to access shared data is only one kind of extensibility though! To see why, consider the Wildcard workflow for adding read times and annotations to Hacker News articles, and then sorting the website by read times, shown in Figure 3-2.

Figure 2-3 shows how the Hacker News example fits into our model of shared state. Importantly, this does not actually represent the real underlying implementation of the application at hand, because Wildcard is a browser extension that works with existing websites, which do not necessarily support extensibility over their underlying datastores. Rather, this shows the mental model that we are providing to the user performing the customization.

The shared state in this case is a table of articles on the web page. The customization leverages three distinct forms of extensibility described above:

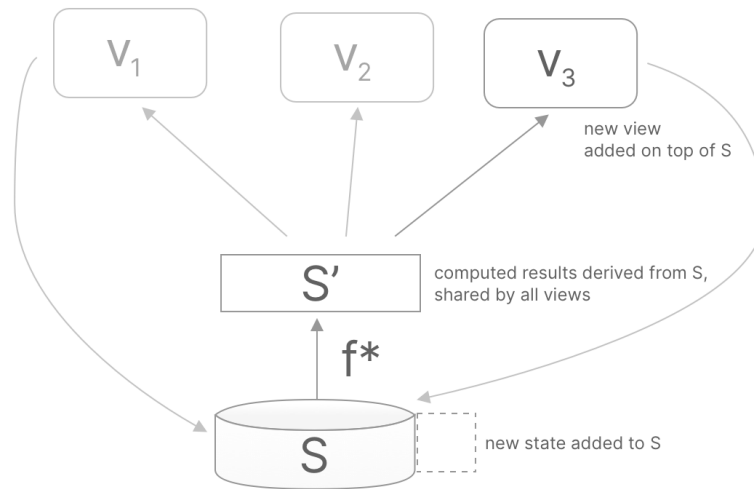


Figure 2-4: Three forms of extensibility: adding state, adding derived computed results, and adding a new view

- **Adding views.** The Wildcard table view is a new view of the list of articles. It is a highly generic one that can visualize any tabular data, but is a view nonetheless. The user can use the table view to perform updates to the underlying shared state—e.g., sorting the list in a different way—and these updates are also reflected in the original website UI.
- **Adding state.** The user can add a new column to the table to store manually written annotations for the articles.
- **Adding derived computations.** The user can add a computed column which calculates read times for each of the articles. The results of the computation are available to all views, not just the table view, so they can be displayed in the underlying webpage.

This example shows that there are several useful classes of extension that we can enable over shared state, shown in Figure 2-4:

- **Adding views.** Adding new views on top of the shared state, which can visualize the state and perform updates on it just like any other view
- **Adding state.** Extending the schema of the state stored in S , e.g. adding new columns or tables to a relational database
- **Adding derived computations.** We can define a function f^* which applies to the raw base state S and computes new derived values which can be used by all views. For example, these might be database queries which can be shared by many views.

2.3.4 Concurrent

One of the main difficulties with building shared state abstractions is correctly handling concurrent edits. For example, multiple text editors operating on the same text

document should be able to insert text simultaneously without errors like interleaving of characters. Without careful and precise implementation, it is easy to create race conditions and corrupted data anytime state is being shared across multiple writers.

Building in concurrency control at the data layer can be a powerful way to relieve developers of some of the burden of reasoning about tricky concurrency issues. Databases have invested substantial effort in addressing this problem and allowing efficient concurrent edits through mechanisms like multi-version concurrency control [11, 43, 83]. Algorithms like Operational Transform (OT) [25, 74] and Conflict-free Replicated Data Types (CRDT) [70, 68] have also developed solutions to concurrent editing of data structures like text sequences and counters.

2.3.5 Flexible data model

When different views are coordinating over the same shared state, the views might each want to display different projections of the state, e.g. combining data across various collections, or grouping and aggregating it in different ways. Supporting this kind of usage requires a data model that is flexible enough to support different kinds of query patterns.

In Riffle we use the relational model [21], which entails storing data in tables, and then executing queries that join together data across tables to fetch the needed results. It encourages storing data in a *normalized* form, where each piece of data has a single canonical representation without redundant copies. This separates the storage format from the projected format needed for any particular view, and allows for many different shapes of queries over the same information.

In contrast to the relational model, some databases use a document-oriented model, which more tightly couples the storage format of the data with the expected access patterns. This model can make it harder to query in ways that span across documents.

An interesting contrast to either of these data models is a traditional filesystem, which allows the contents of files to be arbitrary sequences of bytes. This model is flexible in a sense because any application can store information in any way it likes. However, the lack of any semantic structure imposed by the storage system can make it more difficult for applications to interoperate, since an application must understand how to parse the low-level byte representation of any given file format.

2.4 Conclusion

Shared state with downstream views is a powerful general model for reasoning about coordination between software components. In this section, we have described several key design considerations for designing such an abstraction:

- **Reactive.** Automatic propagation of updates to downstream dependencies makes it easier for users and developers to reason about the effects of changes.
- **Unified.** Managing all system state—including the state of the user interface—in a shared substrate makes it possible to handle all state in a uniform way.

- **Extensible.** Shared state, especially with unified state management, can serve as a powerful foundation for open-ended extensibility.
- **Concurrent.** Support for concurrent editing is a powerful feature to have built-in to a data substrate.
- **Flexible data model.** It is useful for a shared state mechanism to support modeling a wide variety of data. The relational model supports many kinds of data and query patterns.

In the following chapters, we will show how these properties can be integrated into programming systems that use shared state to support users and developers in coordinating behavior across different views. In Wildcard and Potluck, state is reactively synchronized between a base data source and a tabular dataset, and in Riffle, all view components coordinate through a unified reactive relational database.

Chapter 3

Wildcard: Customizing Existing Websites

In this chapter¹, we introduce *data-driven customization*, an approach to enabling software customization through direct manipulation.

We augment existing user interfaces with a table view that shows the structured data inside the application; when users edit the table, their changes are reflected in the original UI. This simple model accommodates a spreadsheet formula language and custom data-editing widgets, providing enough power to implement a variety of useful extensions.

We illustrate the approach with Wildcard, a browser extension that implements this data-driven customization paradigm on the web using web scraping. Through concrete examples, we show that this paradigm can support useful extensions to many real websites, and we share reflections from our experiences using the tool.

Wildcard concretely demonstrates many of the principles laid out in Chapter 1. It provides a familiar reactive table UI as a mediating interface for interacting with the data inside an application. By enabling many customizations through direct interactions with the table and simple spreadsheet programming, it avoids much of the complexity of imperative programming when performing simple customizations.

3.1 Introduction

Many applications don't meet the precise needs of their users, and it is impossible for developers to anticipate everyone's unique requirements. End user customization systems can help close this gap, by empowering non-programmers to modify their software to satisfy their personal goals.

¹The material in this chapter is adapted from the following paper: Litt, Geoffrey, Daniel Jackson, Tyler Millis, and Jessica Quaye. "End-User Software Customization by Direct Manipulation of Tabular Data." In Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 18–33. Virtual USA: ACM, 2020. <https://doi.org/10.1145/3426428.3426914> [46]. I led the project, advised by Daniel. Tyler and Jessica contributed to the evaluation by developing site adapters and example customizations.

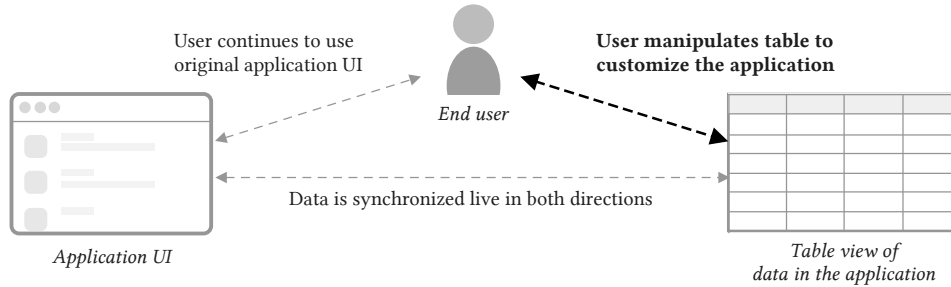


Figure 3-1: An overview of data-driven customization

Many end user customization systems [22, 14, 45, 19] offer a scripting model. They use various strategies to make programming more approachable: friendly syntax, a visual programming environment, or macro recording to bootstrap from concrete demonstrations. But all these techniques build on the same fundamental foundation: an imperative programming model, with statements, mutable variables, and loops.

We have known for decades about an alternative: *direct manipulation* [73], 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. Switching from using an application to customizing it via scripting requires an abrupt shift in interaction model, and poses a steep learning barrier for users not familiar with programming.

We subscribe to MacLean et al.’s vision of a “gentle slope” [51] free of such “cliffs,” where users should only need to make minimal and incremental investments in skill to achieve their desired customizations. We seek to contribute to this gentle slope with a new method for customizing software via direct manipulation, taking inspiration from visual database query interfaces and spreadsheets, which have successfully enabled millions of end users to compute with data through direct manipulation.

In our proposed paradigm, *data-driven customization*, an application’s UI is augmented with a table view where the user can see and manipulate the application’s internal data. These changes don’t just apply to the table; they also result in immediate changes to the application’s original user interface. The user can sort/filter data in the UI, inject annotations, pull in related information from other web services, and more, all using the table as a mediating interface. Interacting with the table view resembles interacting with a familiar spreadsheet, but results in customizing an existing application.

To explore this idea in a real context, we have developed a browser extension called Wildcard that uses web scraping techniques to implement data-driven customization for existing Web applications. We introduce the tool with an example customization of Hacker News in Section 3.2, and then describe the implementation in Section 3.3.

Wildcard is just an initial proof of concept of data-driven customization. In Section 3.4, we discuss our broader vision for how this style of customization could change the relationship between users and creators of software, focusing on three ideas:

- *Decoupling data from applications*: On the modern Web, data is often stored in

proprietary silos, limiting the agency of users to choose their applications and flexibly work with data. We propose data-driven customization as an incremental step towards a more decentralized architecture, where users gain more control over the storage, processing and display of information from web services.

- *Customization by direct manipulation*: We explain how data-driven customization can provide a gentle slope, by allowing a user to customize an application by directly seeing and changing its data, rather than by writing imperative scripts.
- *Semantic wrappers*: 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 data rather than low-level representations.

In Section 3.5 we discuss connections to related work. Our goals overlap with software customization tools, and our methods overlap with direct manipulation interfaces for working with structured data, including visual database query systems and spreadsheets.

Finally, in Section 3.6, we present evidence that Wildcard can produce useful customizations, by sharing reflections from customizing 11 different websites in ways that met our own personal needs.

3.2 Example Scenario

To concretely illustrate the user experience of data-driven customization, we present a scenario of customizing Hacker News, a popular tech news aggregator. Figure 3-2 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 3-2, 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 3-2, Note B). Internally, Wildcard has changed the

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.

	points ↓	user	comments	user
1	886	nsainsbury	148	user
2	722	zdw	361	
3	461	danfox	127	
4	312	teslademi	95	
5	312	maxbaines	181	

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

comments	user1	user2	user3	user4
148				
361				
127				

comments	user1	user2	user3	user4
148	1244			
361	781	13		
127		0		

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

	user	comments	user1	user2	user3	user4
1	886 nsainsbury	148	1244	21 min read	looks fun!	
2	3 buchuki	936		16 min read		
3	244 EndXA	98	852	14 min read	read this	
4	67 kqr	14	827	min read		
5	722 zdw	361	781	13 min read		

Figure 3-2: Customizing Hacker News by interacting with a table view

webpage’s DOM to synchronize it with the sort order of the table. This sort predicate is also persisted in the browser and reapplied automatically the next time the user loads the page, so they can always browse the page sorted by points.

Adding estimated read times. Next, the user decides to attempt a more substantial customization: adding estimated read times to each article, in order to prioritize reading deeper content.

The table contains additional empty columns where the user can enter spreadsheet-like formulas to compute derived values. The user enters a formula into the first column for user-defined formulas or data, which is named `user1` by default (Figure 3-2, Note C): `=ReadTimeInSeconds(link)`.

This formula calls a built-in function `ReadTimeInSeconds` that uses a third-party public web API to compute an estimated read time for the URL’s contents. The `link` argument in the formula refers to a column name in the table; the formula is automatically evaluated across all rows in the table, using the value of `link` for each row.

The user clicks the `user1` column header to sort the articles on the page in descending order of estimated read time. They would also like to display the read times in the page, but a number in seconds isn’t the most legible format, so they enter another formula in the `user2` column: `=Concat(Round(user1/60), "min read")`. This formula converts seconds to minutes by dividing by 60 and rounding, then concatenates the result with a string label, producing results 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 3-2, Note D). Each article on the page now shows an annotation with the estimated read time in minutes. (The formatting of annotations was determined by the programmer who created the adapter for Hacker News.)

Adding manual annotations. The user can manually add notes to the table, by entering data values into the table without 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 3-2, 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. The user can filter out articles they have 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`; links that the user has already visited are hidden both from the table view and the original page. This is an example of a customization that the original website could not have implemented, since websites don’t have access to the browser history for privacy reasons. But by using Wildcard, the user was able to implement the customization locally, without needing to expose their browser history to Hacker News.

This scenario has shown a few examples of how data-driven customizations empower a user to improve their experience of a website. Section 3.6 explains many other use cases and contexts where the technique applies, but first we explain how

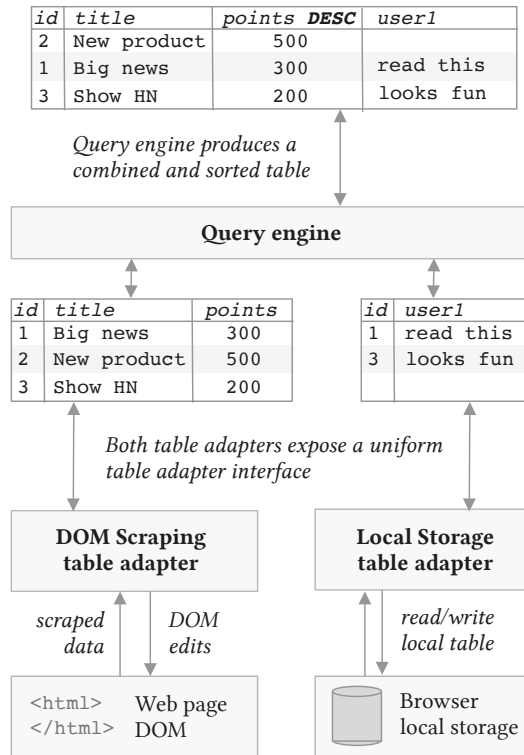


Figure 3-3: The table adapter architecture

the system works internally.

3.3 System Architecture

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

First, the web page and the browser storage are each wrapped by a **table adapter**, which defines a bidirectional mapping between an 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 has a *write mapping* defining the effects that edits have on the original data source.

The local storage adapter has a trivial mapping: it loads a table of data stored in the browser, and persists edits to that 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.

The two tables are then combined into a single table for the 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.

3.3.1 Table Adapters

A key idea in data-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 data-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.

Abstract Interface

We begin by describing the abstract interface fulfilled by a table adapter.

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. We currently support strings, numeric values, booleans, and datetimes as types. The columns also carry some additional metadata, 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, a new value may be persisted into local storage; in the DOM scraping adapter, an edit may result in changing the value of 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 from other tables are added to the original web page.

Currently selected record: As the user clicks around the table view, the query engine broadcasts the record currently selected by the user to each table adapter. The DOM scraping adapter uses this information to highlight the row in the page that corresponds to the selected row in the table, which helps the user understand the mapping between the table and the original UI.

Next, we present the three types of table adapters we have built in Wildcard so far. These do not represent an exhaustive set of all possible table adapters—in Section 3.4

we discuss other types of adapters that would fit well into the general paradigm.

DOM Scraping Adapters

DOM scraping adapters enable Wildcard to interface with an existing website UI. A DOM scraping adapter fulfills the standard web scraping task of extracting a table of data from the DOM, but it also acts in the reverse direction: manipulating the DOM to reflect edits to the table.

In Wildcard, DOM scraping adapters are programmed manually for each website using JavaScript code. It might seem that this prohibits non-programmer users from using the system at all, but we mitigate this problem with a shared repository of adapters. Once an adapter is programmed for a website, it is added to the shared repository, enabling any end user to perform customizations on that website.

In the future, other strategies for producing DOM scraping adapters could reduce this dependence on programmers: an end user could specify the scraping logic via demonstration, or the desired data table could be automatically inferred from the page. While we are interested in these techniques and discuss them in Section 3.4, we believe that a shared repository of manually programmed adapters is a pragmatic starting point; given that many users visit the same popular websites, a critical mass of adapters could serve the needs of many users.

To make it easier to create these adapters, Wildcard provides a framework that makes the process feel more like writing unidirectional scraping code than performing a complex bidirectional synchronization. The key idea is this: programmers return pointers to DOM elements representing table rows and table cells; Wildcard extracts data from these DOM elements, but it also uses the pointers to synchronize table edits back into the page. For example, when the user sorts the table, the DOM elements representing the table rows are moved around in the DOM to reflect the new sorted order.

Figure 3-4 shows an example of the scraper code used for the Hacker News example (with some code eliminated for brevity.) It defines the following main components:

- **enabled**: defines when this adapter should run, usually based on the active URL in the browser.²
- **attributes**: defines a schema for the table, with a name and type for each column
- **scrapePage**: defines a scraping function which returns an array of objects, each containing the data for a single row of the table.

Here are some of the concerns that emerge when building adapters in practice:

Choosing a row ID: When possible, it is best to choose a server-side identifier that remains stable across pageloads. This enables user annotations persisted in local storage to be associated with the same records on subsequent pageloads. We have

²Currently Wildcard can only show a single table at a time, so if multiple adapters are enabled for a single page, we arbitrarily pick one. It would be a straightforward extension to allow the user to switch between multiple possible tables available on the page.

```

const HNAdapter = createDomScrapingAdapter({
  name: "Hacker News",

  // Specify when the adapter should be enabled, based on current URL
  enabled() {
    return (
      urlExact("news.ycombinator.com/") ||
      urlContains("news.ycombinator.com/news") ||
      urlContains("news.ycombinator.com/newest")
    );
  },

  // Define the name and type of each column in the table
  attributes: [
    { name: "id", type: "text", hidden: true },
    { name: "rank", type: "numeric" },
    { name: "title", type: "text" },
    { name: "link", type: "text" },
    // ... other columns omitted for brevity
  ],

  // Iterate over DOM elements, returning information about each row
  scrapePage() {
    return Array.from(document.querySelectorAll("tr.athing")).map((el) => {
      let detailsRow = el.nextElementSibling;
      let spacerRow = detailsRow.nextElementSibling;

      return {
        // Return a unique ID for each row
        id: String(el.getAttribute("id")),

        // Return DOM elements corresponding to this row
        // (this enables moving/hiding the elements for sorting/filtering)
        rowElements: [el, detailsRow, spacerRow],

        // Return data for each column
        dataValues: {
          rank: el.querySelector("span.rank"),
          title: el.querySelector("a.storylink"),
          link: el.querySelector("a.storylink").getAttribute("href"),
          // ... other columns omitted for brevity
        },

        // Specify where annotations should be injected, and what they should look like
        annotationContainer: detailsRow.querySelector("td.subtext") as HTMLElement,
        annotationTemplate: `| <span style="color: #f60;">${annotation}</span>`,
      };
    });
  },
});

```

Figure 3-4: Source code for the Hacker News scraper. Some details removed for brevity.

found that it's usually possible to find such an identifier; for example, each item in a page often contains a link to a page with more details, with a URL that contains a stable ID.

Types of scraped values: For each individual value within a row, there are two options for what type of data can be returned by the programmer-specified scraping function.

The default option is to return a DOM element, in which case the generic adapter extracts the text contents of the DOM element and casts them to the type of the column. The advantage of returning a DOM element is that the value is editable—when the user changes the value in the table, the generic adapter can simply overwrite the inner contents of the DOM element.

Another option is to directly return a value, rather than returning a DOM element. The advantage of this approach is that the adapter author can perform arbitrary computations to derive the returned value—for example, they can use a regular expression to extract a substring. The disadvantage is that the field is no longer writable. The computation used to derive the value isn't reversible, so there's no way to reflect a table edit in the DOM.

Optional overrides: In order to turn a unidirectional scraping function into a bidirectional scraping adapter, there are a number of behaviors that must be specified:

- when should the scraping function be re-run in response to changes on the page?
- how should injected annotations appear in table rows?
- when the user selects a row in the table, how should the corresponding row in the DOM be highlighted?

The scraping framework defines sensible defaults that work well on many sites, but the programmer can optionally override them to provide better site-specific behavior. For example, the Hacker News adapter specifies annotation options that make user annotations appear more naturally in the design of the original webpage.

AJAX Scraping Adapters

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.³

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

³So 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 and Edge as well, but we have not finished our implementation.

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.

Local Storage Adapters

The local storage adapter simply stores a table of data in the browser. This is currently only used to persist annotations.

The table view is initialized with empty columns such as `user1` which serve as the user’s “scratch space,” as shown in Section 3.2. When the user makes edits to these columns, new rows are created in the local storage table. The rows contain the record ID from the DOM scraping adapter, which enables them to be re-associated with the same records on subsequent pageloads.

3.3.2 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 our customization needs, 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. As in visual database query tools like SIEUFERD [5] and Airtable, 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.

3.3.3 Table Editor

We provide a table editor view as the user interface on top of the query engine. Our table editor is built with the Handsontable JavaScript library, which provides built-in 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 3.6 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. For example, we could provide a calendar view for displaying a table containing a date column.

3.4 Vision

We envision data-driven customization as a broad paradigm that could extend well beyond the Wildcard proof-of-concept, and ultimately result in new software architectures that empower end users to mold software to their specific needs. Here we explore some of the deeper ideas underlying our work, and future possibilities beyond the Wildcard tool.

3.4.1 Decoupling Data from Applications

When data is freely available outside the context of a specific application, users have more freedom to choose a suitable application for their needs. For example, an RSS feed can be consumed by many reader applications—a journalist can use a power tool optimized for skimming hundreds of news sources a day, while a casual reader can use a simple app to keep up with a few blogs. Motivated users can even create their own custom workflows for filtering and combining RSS feeds, either via traditional programming or in an end-user programming environment like Yahoo Pipes.

However, on the Web today, data is often siloed and only accessible through a single prescribed application. A Facebook feed can only be viewed through the Facebook application. Podcasts, originally served openly through RSS, are beginning to become exclusive to specific platforms like Spotify. This coupling between data and applications leaves users at the mercy of using a single client optimized for specific purposes (e.g., maximizing engagement) that may not be aligned with users’ individual desires.

Some services provide APIs that mitigate these siloing effects, but APIs fail to provide a full solution to the problem. First, APIs often provide limited access, especially when an open client ecosystem would harm the economic incentives of a web service built on advertising in a first-party client—in 2012, Twitter infamously imposed restrictions on third-party applications that mimicked the “mainstream Twitter consumer client experience.” A second problem is that web APIs have a high barrier to entry—they tend to be designed more for programmers creating entire applications or heavy-duty automations than for end users casually modifying their own experience.

There have been compelling suggestions for more decentralized architectures that would give users more control of their data. Local-first software [39] suggests that productivity applications should run logic and store data locally, while retaining the benefits of realtime collaboration through peer-to-peer synchronization. The SOLID project [10] envisions a decentralized future where users store data on their own servers, and choose to grant limited access to applications. We see data-driven cus-

tomization as aligned with these decentralized visions, but complementary to them in two ways.

Incrementality. Rather than proposing that web services be totally rearchitected, data-driven customization suggests a more incremental path for adding user agency to existing software.

Data-driven customization allows for lightly augmenting a centralized website with decentralized data storage. Section 3.2 demonstrated how a user’s private annotations on a news site could be stored in their browser, without needing to upload the annotation to the website’s server. A hybrid storage model is reasonable here: a centralized storage model makes sense for most of the information on Hacker News that is viewed by all users, but a user’s private annotations can easily just be stored in their browser. This model could even be extended with peer-to-peer sharing—a column in a Wildcard table could be shared directly among friends, providing shared annotation of a common website without needing to use the website itself as a centralized intermediary.

By using third-party data extraction, data-driven customization also works with existing websites that do not expose structured data to the user. Ultimately, in adversarial situations where websites are strongly incentivized to restrict access to their data, scraping is unlikely to be a sustainable solution. But we hypothesize that there are many more situations where websites are *neutral*: not opposed to the idea of end user customization, but also not sufficiently motivated to create and maintain a public API. We see these neutral situations as a context where this kind of incremental approach could succeed. Perhaps some of these websites might be more motivated to provide official extension hooks if they saw the value that users were getting from unofficial community-provided ones.

Focus on end user customization. Having access to the data is a necessary but not sufficient condition for empowering end users to craft their own software experience. Another key ingredient is providing usable tools and interfaces for working with the data.

In data-driven customization, we focus heavily on this part of the solution. By showing raw data in the context of a user interface and allowing small tweaks to the original application’s behavior, we provide a smooth path for people to move from using an application to tweaking it.

In this sense, data-driven customization is a complementary approach to other projects that focus on getting users greater access to their data. In a decentralized future where data is stored locally rather than in cloud silos, interfaces like Wildcard would be one technique for actually making use of this data in service of greater end user flexibility.

3.4.2 Customization by Direct Manipulation

Hutchins, Hollan and Norman [32] 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 calendar names from the Calendar application in Applescript [22], 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. [51] 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 data-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 and typing on the relevant row. At every step, the user receives intermediate feedback, not only in the table view, but also in the original application, so it’s clear whether they are making progress towards their desired result. These types of interactions are common in GUI applications, and Wildcard therefore seems to meet MacLean et al.’s goal: some one-click customizations are as easy as using the original application. Formulas introduce some additional complexity, but spreadsheets have demonstrated that formula programming is still accessible to many users, helped by the pure functional semantics and the visibility of intermediate results.

One aspect of directness that we have chosen not to pursue in Wildcard is enabling customization in closer proximity to the original user interface elements, as explored by other tools like Scotty [24]. While closer proximity might be helpful, we

have found that augmenting the original UI with a distinct, additional representation provides a more consistent experience across all applications, and clearly shows what structured data is available to work with. We also emphasize the mapping between the representations by highlighting content in the original page, similar to the way that browser developer tools highlight the currently selected element in the DOM inspector in the original page.

Ainsworth et al. provide a helpful taxonomy of the value of multiple representations [2]. 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 encourages thinking of an application in terms of its core information, rather than the specific capabilities provided by the current user interface. In our experience, we’ve often found that looking at a site’s data in table format tends to spur new ideas for customizations which weren’t evident from looking at the original UI.

3.4.3 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 any plugin API.

With Wildcard, we use a hybrid approach that aims to provide the best of both worlds. Third-party programmers implement site-specific adapters that are internally implemented as 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, 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 con-

structs (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 wraps the Gmail web client in a convenient API for browser extensions to build on.

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, so more work is needed to fully test this hypothesis.

The distribution mechanism for semantic wrappers is also important for encouraging an ecosystem of shared wrappers. Currently, the distribution mechanism is simply merging the code for all adapters into the main Wildcard codebase. This is a simple solution, but makes it fairly difficult to contribute new wrappers and requires installing a new version of the extension to gain access to new wrappers. 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 means a malicious adapter could exfiltrate sensitive information from a page. Approaches to security could include centralized code review, using a restricted scraping DSL, or creating a sandboxed context for scrapers without access to networking APIs.

Alternate Mechanisms for Wrapper Creation

Requiring programming to create wrappers has an obvious limitation. If an end user wants to customize a site and no programmer has contributed a wrapper for that site, then they have no means of customizing it. Although a sufficiently vibrant community of programmers could produce wrappers for many popular sites, it's unrealistic to imagine covering all websites that end users might want to customize. We envision two strategies for dealing with this problem:

End user wrapper creation. If end users could create wrappers without programming, they could customize any website, as long as it worked with the wrapper creation process. We could try integrating techniques from projects like Helena [19] which enable end users to scrape websites by demonstration. Another intriguing possibility we plan to explore in the future is blurring the boundaries between scraping and customizing by using spreadsheet formulas as a means of guiding the scraping process.

First-party wrapper creation. An integrated adapter installed by the developer of an application could directly access internal state, 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. This kind of plugin would allow developers to integrate with an ecosystem of formulas and customization tools without needing to build that functionality from scratch.

3.5 Related Work

Data-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.

3.5.1 Customization Tools

Data-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. [51], 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 data-driven customization (as discussed in Section 3.4.2). 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.

More recently, Tchernavskij introduced the notion of *malleable software*, which aims to allow users to tailor their software by pulling apart and recombining individual user interface elements [76]. This work builds on prior work on customizing graphical interfaces, including Beaudouin-Lafon’s idea of instrumental interaction [6] and Klokmose et al.’s customizable Webstrates environment [40]. Data-driven customization shares the overall goals of this line of work, and proposes new interaction techniques for achieving them. In particular, by showing users a structured data representation and allowing them to perform lightweight programming through formulas, our approach supports customizations of intermediate complexity: more sophisticated than those that can be achieved merely by manipulating existing interface elements, but simple enough to not require full-blown programming.

Other web customization tools have also aimed to enable end users to modify web interfaces without programming. Sifter [34] 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 data-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 might integrate end user scraping techniques in Wildcard in the future, but we believe that, when possible, it is valuable for end users to have a customization experience decoupled from the challenge of 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 [30] 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 [14] and Coscripter [45] offer user friendly syntax for writing web automation scripts; Applescript [22] 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 data-driven customization.

3.5.2 Spreadsheets and Visual Query Interfaces

Another relevant area involves spreadsheets and visual query interfaces. 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 [5] 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 a user should be able to modify queries by interacting with the results of the query rather than some representation of the query itself. SIEUFERD's interface supports a far more general range of queries than Wildcard, but the core principles of the user interface are the same. 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 [54], Quilt [8], Gneiss [18], Marmite [80], and Glide. We share the main idea of connecting a spreadsheet view to a GUI, but we apply it to software customization, rather than building software from scratch.

Another related system is ScrAPIr, by Alrashed et al. [3], 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

Table 3.1: A list of customizations that we have implemented using Wildcard.

Website	Description	LOC	Example customizations
Airbnb	Travel	73	Add Walkability Scores to listings. Sort listings by price.
Amazon	Online shopping	99	Sort third party sellers by total price, including 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.

of wrappers around existing web APIs. Unlike Wildcard, however, ScrAPIr targets explicit APIs exposed by developers. It also focuses on backend services and doesn’t aim to extend the frontend interfaces of web applications.

3.6 Evaluation: Experience & Limitations

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

We developed site-specific adapters for 11 websites that we personally use frequently, and then built customizations for those websites using the Wildcard table view. Table 1 summarizes these results, showing the number of lines of code in the adapter for each site, and some example customizations we created. Here we offer our reflections from these experiences 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?

3.6.1 Range of Customizations

We have found that data-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.

Sorting and 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 restrict user agency. For example, Airbnb previously allowed users to sort listings by price, but removed that feature in 2012. In other cases, a lack of sorting options seems more like an innocent omission; for example, the Instacart grocery delivery service has a spartan UI for viewing an order, which doesn't allow for sorting items 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 site's original ranking. This restriction could be overcome in the future by scraping content across multiple pages, or by using an integrated adapter and avoiding scraping altogether. However, we've also realized that sorting/filtering a single page of a paginated list is sometimes an acceptable outcome (and 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.

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 as we review an order and consider modifications (shown in Figure 3-7).

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 as shown in Section 3.2. 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 to product information | Every purchase on Amazon.com is protected by an [A-to-z guarantee](#). | Feedback on this page? [Tell us what you think](#)



The Humane Interface: New Directions for Designing Interactive Systems (Paperback)
by Jef Raskin (Author)
★★★★☆ 73 customer ratings
Access codes and supplements are not guaranteed with used items.

Refine by [Clear all](#)

Shipping

Prime

Free shipping

Condition

New

Rental

Used

Like New

Very Good

Good

Acceptable

\$6.52 **Used - Acceptable**

+ \$3.99 shipping + \$0.41 estimated tax

Used showing normal wear and tear. May have some stickers, residu... » [Read more](#)

- Arrives between June 2-9.
- Want it delivered Wednesday, June 3? Choose **Expedited Shipping** at checkout.
- [Shipping rates and return policy.](#)

Goodwill of North Georgia ★★★★★ 98% positive over the past 12 months. (2,579 total ratings) [Add to cart](#)

\$6.52 **Used - Acceptable**

+ \$3.99 shipping + \$0.41 estimated tax

Fairly worn, but readable and... If applicable: Dust... » [Read more](#)

- Arrives between June 1-4.
- Want it delivered Tuesday, June 2? Choose **Expedited Shipping** at checkout.
- [Shipping rates and return policy.](#)

GoodwillBook ★★★★★ 97% positive over the past 12 months (332,246 total ratings) [Add to cart](#)

	total_price ↑		delivery_detail	rating
Wildcard v	10.92	Used - Acceptable	Arrives between June 2-9.	5
1	10.92	Used - Acceptable	Arrives between June 1-4.	5
2	11.16	Used - Acceptable	Arrives between June 2-9.	4.5
3	11.31	Used - Acceptable	Arrives between June 2-9.	5
4		Used - Acceptable	Arrives between June 1-4.	5
5		Used - Good	Arrives between June 2-9.	4.5
6	13.77	Used - Good	Arrives between June 2-9.	4.5
7	14	Used - Very Good	Arrives between June 2-9.	5

Figure 3-5: Sorting the used sellers page on Amazon by total price, including fees. The original page doesn't have sorting, and doesn't show the combined price.

	name	eta	categories	price_bu
	The Scoop N Scoop	30-45 Min	Ice Cream and Frozen Yo	\$
	bbq-chicken - Boston, M	45-60 Min	Korean,Bar Food	\$\$
	63 Cambridge Rd)	45-60 Min	American,Fast Food,wing	\$
	ong Kong Restaura	65-80 Min	Chinese,Asian,Asian Fusio	\$
	Chipotle Mexican Grill (59	30-45 Min	Healthy,Mexican	\$
1	the-scoop-n-scooter	30-45 Min	Ice Cream and Frozen Yo	4.6
2	bbq-chicken-boston-ma	45-60 Min	Korean,Bar Food	4.4
3	kfc-163-cambridge-rd	45-60 Min	American,Fast Food,wing	4.6
4	new-hong-kong-restaura	65-80 Min	Chinese,Asian,Asian Fusio	0
5	chipotle-mexican-grill-59	30-45 Min	Healthy,Mexican	4.4
6	wings-over-somerville	40-55 Min	Traditional American,Wing	0.99
7	all-star-pizza-bar	20-30 Min	Burgers,American,Sandw	2
8	harvard-house-of-pizza	40-60 Min	Pizza,American,Italian,Wil	4.4
9	smashburger-495-riversi	45-55 Min	American,Burgers	2.99

Figure 3-6: Organizing takeout restaurants on Uber Eats by delivery ETA and price

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 3.2.

Cell Editors

We developed two *cell editors*: custom UI widgets for editing values in the table.

One benefit that cell editors provide is enabling users to incorporate their private information into a web UI. We created a datepicker widget (based on the FullCalendar plugin), which can load calendar data from a Google Calendar. This makes it convenient to enter dates into a website based on the user's personal calendar information, without needing to upload a user's calendar to the website itself.

Another benefit is that a user can choose a single preferred widget for editing a certain type of information across different sites. For example, a user could use their favorite rich text editor to edit text in various websites like blogging platforms and task trackers. To demonstrate this capability, we built a text editor based on the CKEditor rich text editor. We used the editor with Google's Blogger website, by building a site adapter that represented the contents of a blog post as a single table cell containing an HTML string (shown in Figure 3-8).

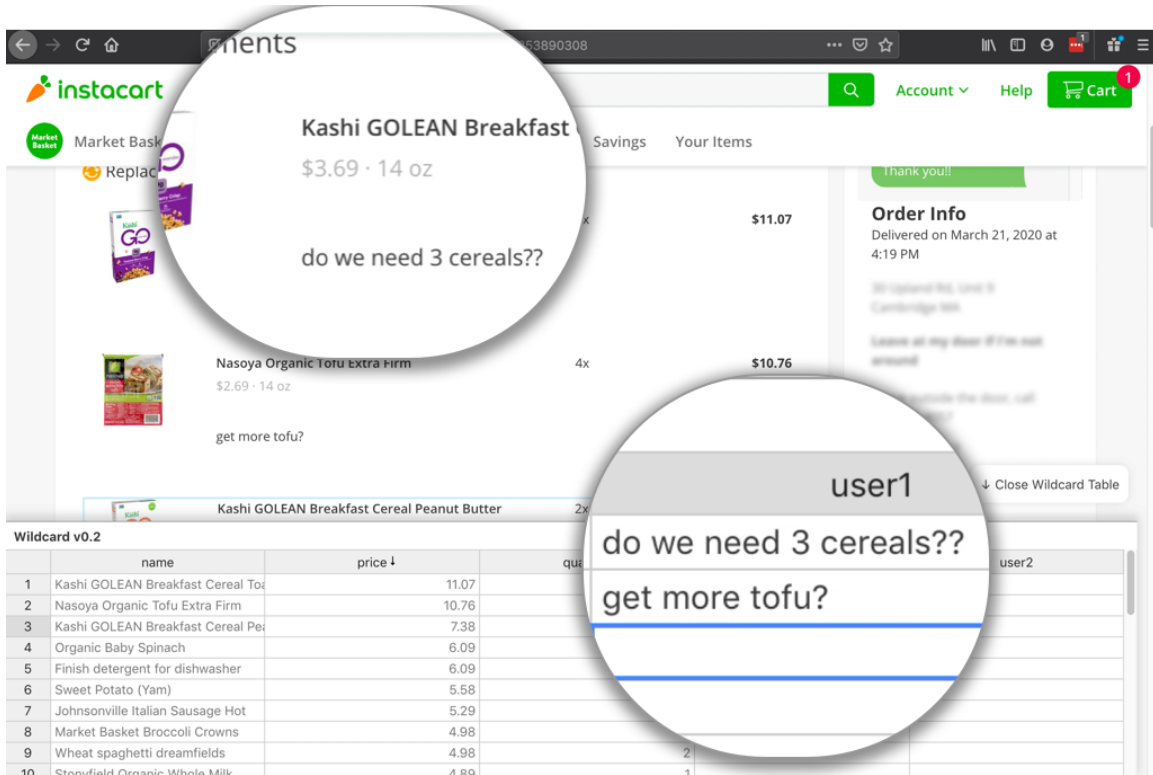


Figure 3-7: Taking notes on Instacart grocery items, after sorting them by price

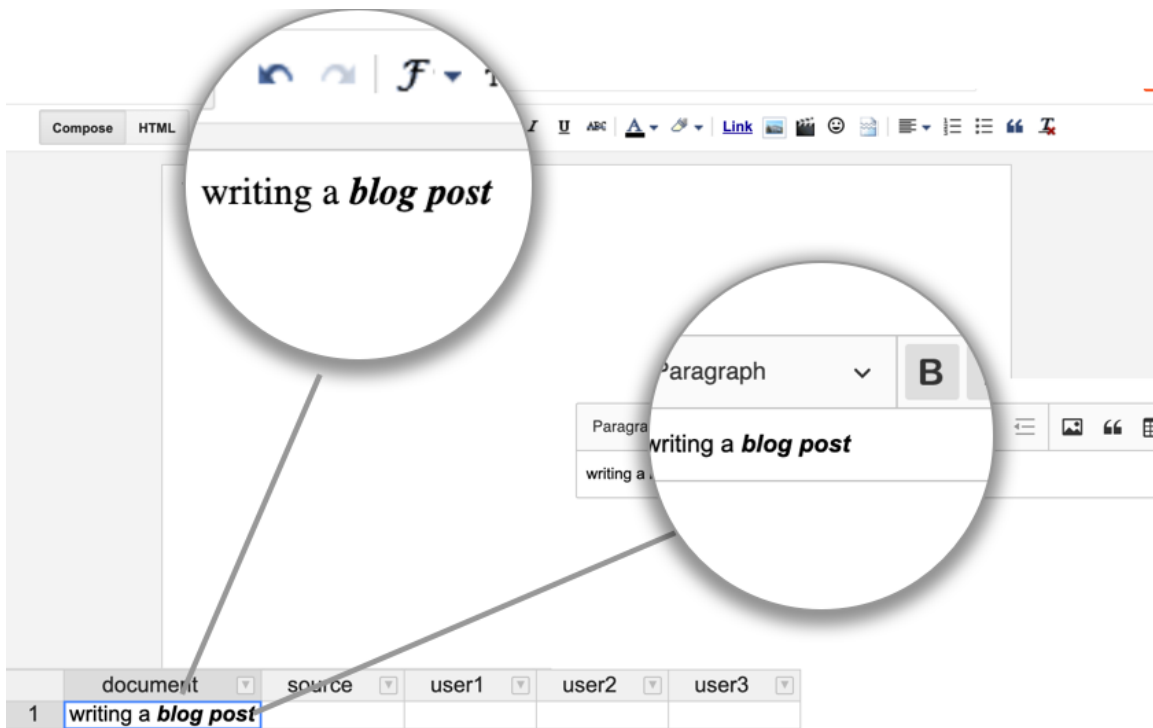


Figure 3-8: Using a custom text editor widget to edit a blog post on Blogger. The text is synchronized with the Blogger editor through a table cell.

Limitations

There are many customizations that are not possible to implement with data-driven customization. Some of the limitations are specific to the current implementation of the Wildcard extension, but others 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 acceptable, since our goal is to support as many useful customizations as possible with a low threshold of difficulty, and not to span all possible customizations.

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.

3.6.2 Viability of Scraping

Separately from the range of customizations, we also evaluated the feasibility of building DOM scraping adapters in practice. In order for third-party customization through Wildcard to succeed, it is important that creating 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. Sometimes, addressing the desired set of elements can be difficult, and when sites change, scrapers can break; we observed several instances where sites changed their CSS classes and caused Wildcard adapters to no longer work. One benefit of a library of shared wrappers is that if many customizations depend on some piece of scraping logic, rather than having the scraping logic embedded in a single browser extension, it should be more likely to be fixed quickly.

The interactive nature of Wildcard also introduces additional challenges beyond

normal web scraping. 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 very 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 uses 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 found that AJAX requests contained useful information not shown in the UI, like the category and barcode ID of an item.

3.7 Conclusion and Future Work

One direction for future work could be to characterize the limits of the 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? For example, we could enable users to set up triggers to perform actions like sending notifications when certain conditions are met in the table view.

Also, more broadly, data-driven customization suggests new possibilities for how multiple applications might be integrated in new ways, by synchronizing their underlying data representations in a shared format. So far, we have mostly explored the implications for customization within a single application, but it would be interesting to explore how end users could use these techniques to synchronize data across applications to avoid manual coordination work.

In summary, in this chapter, we have shown how ideas from reactive database editors and spreadsheets can be applied to customize existing web applications in useful ways. Wildcard demonstrates that it is possible to take an incremental approach where direct manipulation table views are used to interact with an application state, even if the application itself is actually built using traditional techniques.

Chapter 4

Potluck: Gradually Enriching Text Notes

In this chapter, we introduce our second approach to creating personal software¹. It uses a strategy for bridging the gap between text documents and apps called *gradual enrichment*: allowing users to record information in natural, messy ways, and then slowly adding formal structure and computational behavior only as needed.

4.1 Introduction

One inspiration for gradual enrichment is spreadsheets. In a spreadsheet, a user can start writing down data in a freeform grid, without committing to any particular structure. They can then write formulas that run computations on information, if and when it's useful to do so. Eventually, after lots of iterations, they might arrive at a highly complex software application, but one grown organically from their data and unique needs. At every point along the way, the artifact remained useful and grounded in real needs.

Here we explore how to apply this kind of gradual enrichment to text documents. People often jot down freeform text notes—recipes, schedules, workouts, chores, and more—using apps like Apple Notes and Notion. These notes contain meaningful information, like quantities in a recipe or weights in a workout log, that can serve as the basis for useful computations. **How might we enable people to gradually turn these text documents into custom pieces of software?**

Potluck is a research prototype that demonstrates a workflow for gradually turning text documents into interactive software. It has three parts:

- **Extensible searches.** Users can define *searches*: custom patterns that detect data within the text of a note. Searches are defined in a compositional pattern

¹The material in this chapter is adapted from the following paper: Litt, Geoffrey, Max Schoening, Paul Shen, Paul Sonnentag. “Potluck: Dynamic Documents as Personal Software.” 2022 LIVE Workshop at SPLASH. <https://www.inkandswitch.com/potluck/> [48]. I led the project, and all authors made substantial contributions to the main ideas. The work was supported by the Ink & Switch research lab.

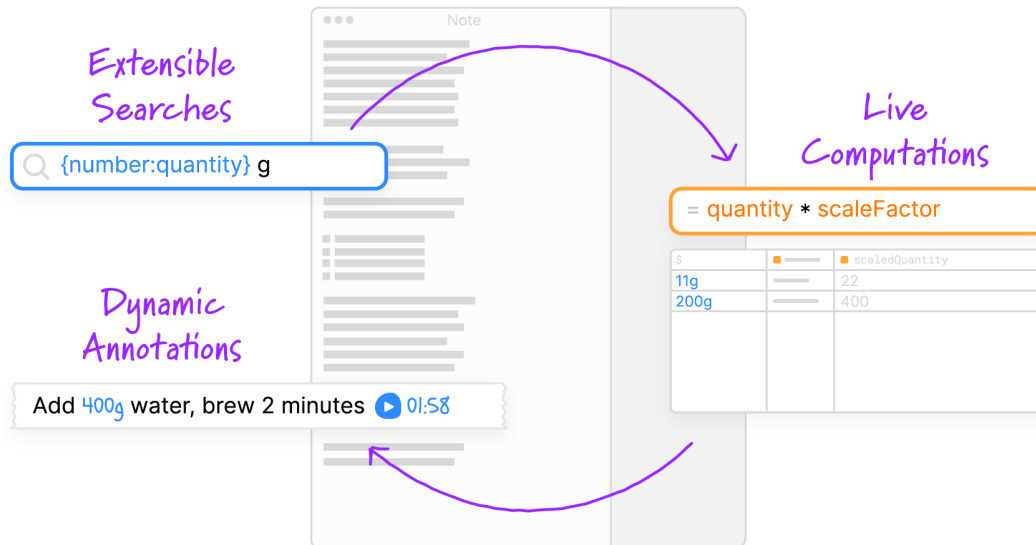


Figure 4-1: The Potluck interaction model forms a loop: extract data from text, compute with that data, and then display results back in the text.

language which allows reusing patterns that others have written.

- **Live computations.** Once data is extracted from the text document, users can write formulas that compute new values based on the extracted data. Formulas are written using JavaScript, in a live programming environment that resembles a spreadsheet.
- **Dynamic annotations.** Computed values can be displayed in the original text document as *annotations*. Potluck provides a few annotation types that can insert new text, cover up or restyle the original text, or even inject interactive widgets.

Together, these ideas form a loop. By treating text as both a source of information and a substrate for hosting a user interface, we can turn a text document into an interactive application (Figure 4-1).

Figure 4-2 shows an example of a simple Potluck document: a recipe for making coffee, which includes an interactive slider that can scale up the number of servings.

The interactive and computational behavior of this document was constructed within Potluck itself. Later we will explain the details of how it was made; for now it's sufficient to know that we've 1) constructed a *search* that finds the quantities of coffee and water, 2) run a *computation* that multiplies those quantities, and then 3) set up some *annotations* to overlay the scaled quantities over the original ones.

We have found that many different kinds of software can be built in Potluck. We've used it to build tools for tracking household chores, managing a cash register, organizing a meeting agenda, tracking workouts, splitting a bill, and planning a trip. Figure 4-3 shows several examples.

Starting with freeform documents promotes a wide variety of use cases. Existing

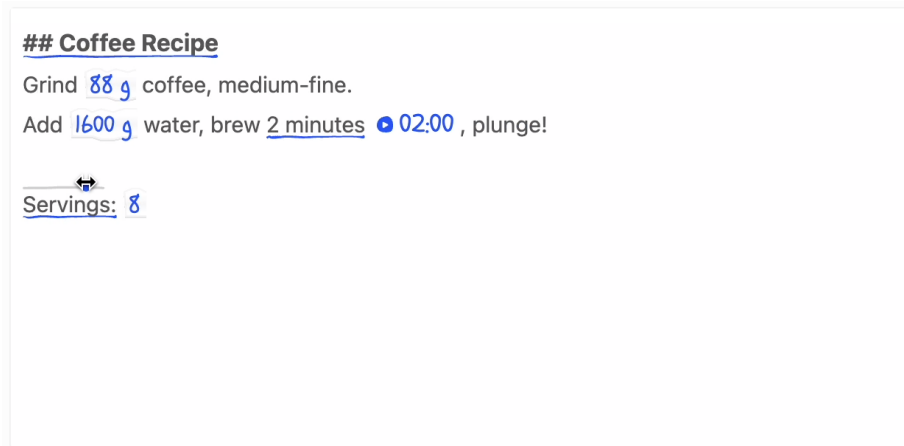


Figure 4-2: A coffee recipe in Potluck, with a slider for scaling the number of servings

notes can serve as inspiration, rather than trying to build applications from a blank slate. We've also found that barebones text-based UIs are sufficient for many personal use cases, without the need for a full layout design.

The idea of gradually enhancing a document into a software application is not new. It is related to document-based productivity tools like Coda, Notion, Roam, and Logseq, as well as research systems including Documents as User Interfaces [12], Webstrates [40], and Smalltalk [35]. What, then, are the contributions of this work? Potluck extends in two directions less explored by prior work:

- **Using freeform text data as a source of information for computations.** Many tools only allow users to compute with data that's been put into a specific structured format. In Potluck, we encourage people to write data in freeform text, and define searches to parse structure from the text.
- **Using text annotations to power an interactive interface.** Even in tools that combine documents and computation, there's often some separation between editable text and computational results. In Potluck, we deeply entangle interactive elements with the user's text, by providing dynamic annotations that can overlay or restyle the original document. The effect is to treat the text itself as a place to host UI.

There are still some important limitations and open questions that we haven't yet resolved. While we care about enabling non-programmers, and have made some design decisions with them in mind, our current prototype does expect the user to have basic knowledge of JavaScript, and our test users have mostly been skilled programmers. We are also not yet sure exactly where the limits of this model are—what kinds of apps are possible and desirable to build in this style?

Prices

cake 🍰 = \$ 5

coffee ☕ = \$ 2

cupcake 🧁 = \$ 2

Sales

🍰 = \$ 5

🧁☕ = \$ 4

Total: \$ 9

Organizing a cash register for a bake sale

Associate each item with an emoji symbol and a price. Enter emojis for each order and a total price is computed.

Kickoff meeting

Start 18:00

6:00 PM - 6:30 PM
30 minutes Introduction

6:30 PM - 7:00 PM Discuss goals

7:00 PM - 8:00 PM Breakup into groups

8:00 PM - 8:30 PM Wrap up

Planning a meeting agenda

Write a duration for each segment, and see the start and end time for that segment

🌱 Fiddle leaf: every 6 days, last watered on 6/21/2023 🌱

🌱 Montsera: every 4 days, last watered on 6/24/2023 🌱

🌱 Yuzu tree: every 5 days, last watered on 6/24/2023 🌱

- The Yuzu tree needs some holes in the pot so that water can drain.

🌱 Pine Bonsai: every 4 days, last watered on 6/24/2023 🌱

Tracking a plant watering schedule

Write how often each plant needs watering and when it was last watered; the date turns red if it's time to water the plant.

Figure 4-3: Potluck documents can help with running a cash register, planning a meeting agenda, and tracking a plant watering schedule.

4.2 Background

Applications are of the most familiar metaphors in using a computer today, so it might seem strange to question their value. But in this section we'll argue that, when examined closely enough, the application model imposes serious rigidity on users. In contrast, we'll show that *documents* have some intriguing benefits due to their more freeform nature, but have their own drawbacks.

4.2.1 The rigidity of apps

Let's examine some of the kinds of rigidity that applications impose on users. To ground our analysis, we'll use some concrete examples from Paprika, a popular recipe management app that provides useful features like scaling ingredients and setting timers.²

One problem with apps is that they **have predefined feature sets** that the creators deemed appropriate for the goal. It's impossible to make a small tweak, or even remove an unwanted feature, unless the creator of the app has explicitly allowed for the change. Each app has a (often narrow) domain that it considers in scope, requiring us to learn to use many independent apps that don't compose together well. In short, apps enact rigid boundaries between tasks, and define rigid solutions within those boundaries.

As an example, consider the sidebar in the Paprika recipes app, which includes many extra features beyond recipes: grocery shopping, pantry management, meal planning, and a feature for assembling "menus" out of recipes (Figure 4-4). On the one hand, this feature set is very broad: the extra sidebar items may be unnecessary for many users, but there's no way to remove them. On the other hand, the application still has a narrow focus, since it siloes away cooking as a separate domain from the rest of life. In the analog world, it's natural to combine a grocery shopping list with other chores for the day, but there's no way to easily integrate a Paprika shopping list with other notes in another app.

There's another important dimension of inflexibility: apps create **rigid data schemas** that define the kinds of information we can record within them. We can fill out the available form fields, but we can't add new fields or scribble in the margins. Structured data inputs struggle with ambiguity—when faced with a list of radio buttons, there's no way we can select two options, like we might have done on a paper form.

Here's an example of this schema rigidity in Paprika: every recipe must list the ingredients in a separate section from the directions. This means that people can't write recipes by simply mentioning the ingredient quantities directly within the directions. Also, when the recipe is scaled up, the multiplier only applies within the officially designated ingredients section, and doesn't affect any quantities shown within the directions.

²Our critique isn't meant to single out Paprika. In fact, we chose Paprika because it's a very popular app, so its flaws are more likely to show problems with apps in general rather than one particular app.

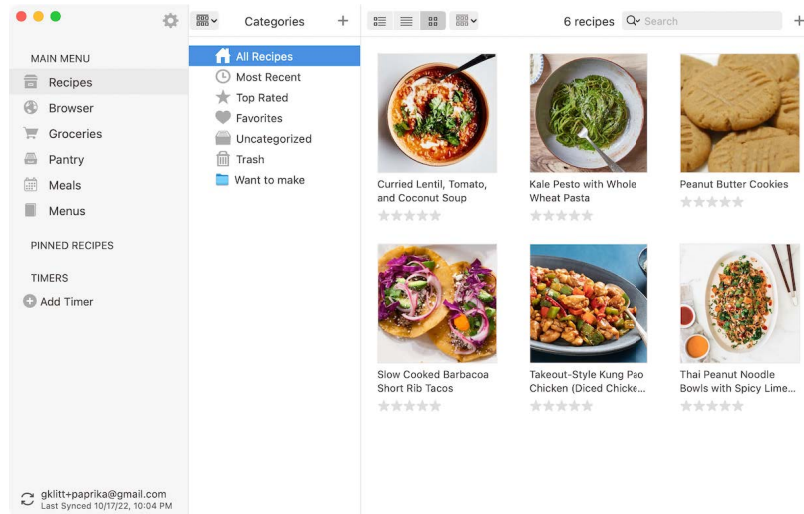


Figure 4-4: Beyond the core recipe functionality, Paprika’s sidebar has extra features for Groceries, Pantry, Meals, and Menus

As another example of schema rigidity, when constructing a meal plan, you can only add a recipe in a specific slot for some date. There’s no way to ambiguously assign a recipe to *either* Tuesday or Wednesday, which would be natural to do in a paper notebook (Figure 4-5).

The overall effect is one of limited agency—once a user has picked the best available app, their choices end there. People aren’t encouraged to think about little further changes they might want to make, or naturally make tweaks as they go; instead, they just adapt their behavior to whatever the app encourages.

4.2.2 The flexibility of documents

It’s striking to contrast the rigidity of a recipe app with the natural ways that people use analog tools to manage recipes. They maintain boxes or binders of favorite recipes and hand them down as treasured family heirlooms. They write in physical cookbooks to leave their own annotations. On one sheet of scrap paper, someone can write a fuzzy sketch of a meal plan for the week, a grocery list, and some notes about other ongoing todos. Because paper is an inherently permissive medium, there’s no need to adhere to preset rules.

In the digital world, we find a close analogue to this kind of flexibility in text and multimedia *documents*, where we can write whatever we want (Figure 4-6). Documents have a couple key advantages relative to applications.

First, **documents are useful for all kinds of tasks**. A note can capture any kind of information, without needing to worry about what bucket it fits in. This generality makes for more versatile tools—there’s a common set of conventions that people are familiar with for editing text, and they’re implemented to a high degree of quality in many editors and libraries because text is such a versatile data format.

Also, **documents don’t enforce a schema**. Text doesn’t enforce rigid schemas

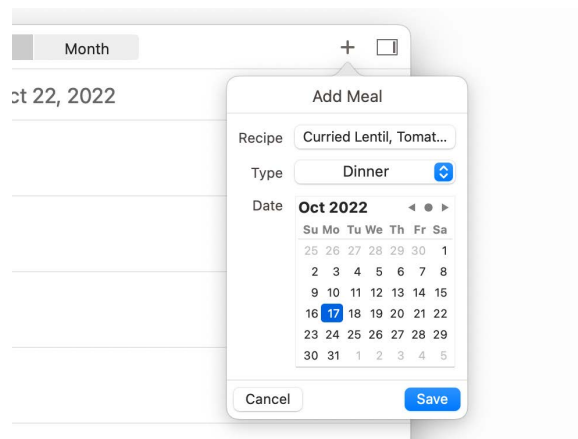


Figure 4-5: Each meal plan entry in Paprika must be assigned to a specific date on the calendar, with no room for ambiguity.

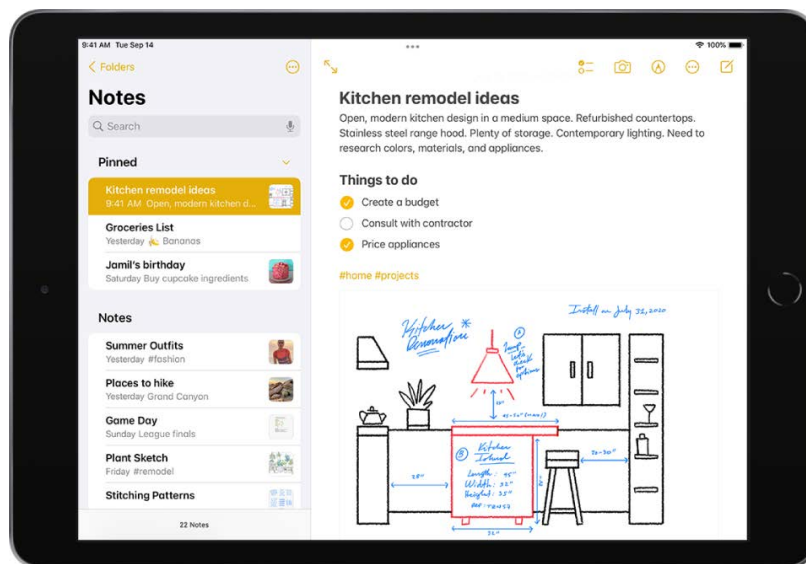


Figure 4-6: Text documents are a single versatile medium for recording all kinds of information.

on what we can write, so we can work in a medium closer to how we think instead of conforming to an external system. It feels entirely natural to come up with a personal way of writing down todos or workout logs in a text note, because the medium is so flexible.

However, there is a tradeoff: **documents are *static***. However, documents lose the convenient computational features of an application, like automatically scaling ingredients, setting timers, or tracking nutrition statistics. This is not just a lack of engineering effort—there’s a fundamental challenge, which is that the data hasn’t been entered in a consistent format that the computer can reliably understand.

4.2.3 Gradual enrichment

We’ve seen that applications and documents each have their own advantages. How might we get the best of both worlds?

We think an appealing approach is to enable people to *gradually enrich* text documents. A user can start with a text document, and as they work with it, they can add bits of structure and computational behavior as needed. This process should be incremental, so that the document remains useful at every step along the way, and there’s never any unnecessary work up front.

Notably, this process can’t avoid the need to teach the computer how to interpret meaning from freeform data. The point is to *defer* this process until it’s absolutely needed. It’s okay to end up with structured schemas when we need them to support computational features, but when they’re not necessary, text is a perfectly adequate representation for humans to interact with.

4.3 Related Work

Many others have explored tools for mixing documents and interactive functionality. Here are a few projects we took inspiration from.

4.3.1 Text documents as user interfaces

A key part of the Potluck interaction model is using a text document itself as an interactive interface, by showing computed values and interactive widgets in the text.

This idea has been explored at least as far back as 1991, in Eric Bier and Ken Pier’s work on Documents as User Interfaces at Xerox PARC [12]. They demonstrated that if we treat specific segments of text as clickable buttons, then we can arrange them in a UI by simply moving the text to the appropriate place in the flow of the document.

Modern commercial tools like Coda³ and Notion⁴ have also explored intertwining text and computation. Coda’s goal is particularly close to ours: allowing users to gradually enrich documents into apps, including by embedding interactive widgets and

³<http://coda.io>

⁴<https://notion.so>

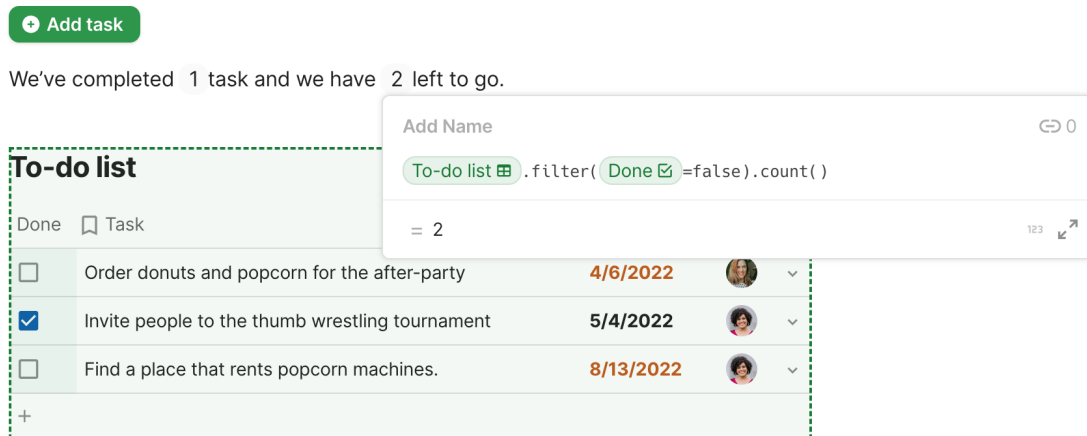


Figure 4-7: Coda supports enriching text documents with interactive computation

computational results in documents (Figure 4-7). However, there’s a key difference in the model: in Coda, computations run on data that stored in structured tables with defined schemas, whereas in Potluck, computations run on data interpreted from freeform text. Later we’ll address some of the tradeoffs between these approaches.

Plaintext knowledge management tools like Emacs Org Mode, Obsidian, Logseq, and TaskTXT allow users to write plaintext files in a specific syntax that supports dynamic functionality, such as extracting todos and surfacing relationships between entities. These tools share Potluck’s goal of building on top of the familiar and portable format of plain text, and many of them also have robust ecosystems of user-authored plugins which can extend the tools to more domains and tasks. However, these tools also tend to come with some degree of built-in, opinionated syntax and features, and writing plugins often requires leaving the tool and applying expert programming skills. Potluck places a greater emphasis on allowing end-users to define their own syntax and features, and aims to apply to a broader set of domains.

Another inspiration is Bret Victor’s reactive documents⁵ (Figure 4-8), which integrate a spreadsheet-like model into text. This allows a reader to interactively change the assumptions in a written explanation and see the consequences of those changes in realtime. While Potluck has a slightly different goal—building personal tools, rather than writing explanations for other people—we share the principle of combining the explanatory power of natural language and the dynamism of computation in a single medium.

4.3.2 Data detectors

In their 1998 paper Collaborative, Programmable Intelligent Agents [62], Nardi, Miller, and Wright describe *data detectors*: intelligent pattern recognizers built into the operating system which can detect structured data like phone numbers and street addresses contained within everyday unstructured documents, and then allow the

⁵<http://worrydream.com/ExplorableExplanations/#reactiveDocument>

Analysis:

Suppose that an extra \$27 was charged to 85% of California taxpayers. Park admission would be free for those who paid the charge.

This would collect an extra \$242 million (\$313 million from the tax, minus \$72 million lost revenue from admission) for a total state park budget of \$642 million. This is sufficient to maintain the parks in their current state, but not fund a program to bring safety and cleanliness up to acceptable standards.

Park attendance would rise by 35%, to 101 million visits each year.

Figure 4-8: A reactive document by Bret Victor explaining a tax policy change. The user can edit values by dragging, and other dependent values in the text automatically update.

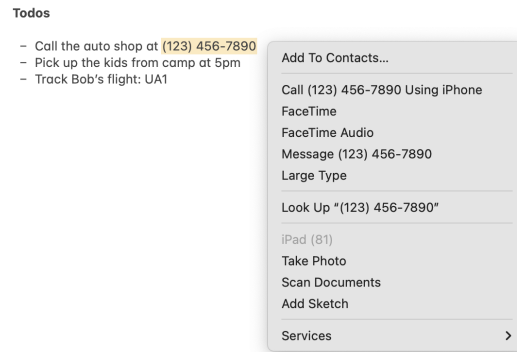


Figure 4-9: A data detector in macOS enables right-clicking on a phone number to add it to contacts or make a phone call.

user to take actions on that structured data. This idea was productized and lives on to this day in MacOS and iOS, although without the user extensibility envisioned by the original paper (Figure 4-9).

We find data detectors promising because they allow people to represent information on their own terms. However, today data detectors are more of a minor convenience than a core metaphor in the OS. We think the reason is that the *interaction model* around data detectors is limited: all a user can do is manually click on a detected value and take a single action. There's no ability to perform more sophisticated computations with the detected data, or to automatically show annotations on detected data.

Feedback loop

In order for a data detector to feel good to use in an interactive setting, it's critical that the system can provide immediate feedback as a user types and help them develop consistent expectations for how the system will interpret their text.

One inspiration was Fantastical, which parses a natural language description into a structured calendar event. It shows instant visual feedback, and is also fairly predictable—even though the input feels freeform, the parser acts consistently enough that it's possible to learn how to reliably write well-parsed inputs. Fantastical even

publishes recommended guidelines for how to type events to help out the parser. Another example of a fast predictable parser is Souolver, a “notepad calculator” app that instantly performs math computations over a notepad with flexible text input.

In our experience, these kinds of tools feel nice to use because they strike a **good balance between natural language and formal syntax**. They allow the user to type naturally, but they don’t aim to accurately interpret everything someone could possibly write. There’s a feedback loop where the user adjusts their input in response to the machine.

Extensibility

People should have the ability to encode their own knowledge and personal micro-syntax into their tools. However, defining abstract patterns over plain text can be difficult even for skilled programmers; regular expressions are notoriously hard to use. We need ergonomic tools for defining patterns.

One category of approaches is to develop better languages. For example, LAPIS [57] by Miller and Myers developed a set of tools for specifying and extracting patterns from text by composing together existing patterns, including a friendly syntax that resembles natural language. Another route is *programming by example* (PBE): letting users provide concrete examples to specify a more general pattern. This technique has been explored by many systems, including the Flash Fill [27] system deployed in Microsoft Excel, related systems like FlashExtract [44], and LAPIS. There are also some interesting hybrid interaction models between PBE and code editing—Mayer et al. use PBE to generate candidate programs, but also let users directly edit the resulting programs [53].

In the next section, we explore how our Potluck tool embodies these ideas of fast, predictable, and extensible data detectors.

4.4 Potluck: an environment for dynamic documents

Potluck is a research prototype we built to explore the idea of gradually enriching text documents into interactive application. Our goal was to design a concrete model for gradual enrichment, and then to evaluate that model by using it to build personal tools.

To understand the core ideas in our prototype, let’s see how someone could use it to build the coffee recipe shown at the beginning of this chapter. As a reminder, the final note contains a slider that scales the recipe quantities, and an interactive timer to keep track of the brew time. These aren’t particularly interesting features in and of themselves; they’re common features found in recipe apps. The important point is that in Potluck, a user can build them from scratch on top of a text note without leaving the tool, and then keep customizing and extending the tool for their needs.

Figure 4-10 shows the steps of the process.

Step A:
Write a *search* that extracts quantities from the document into a table

The screenshot shows a document titled "James Hoffmann Aeropress" with a recipe and notes. The recipe includes "Grind 11 g coffee, medium-fine." and "Add 200 g water, brew 2 minutes, plunge!". The notes mention "6/22/22: Pretty good, but forgot to swirl." and "6/23/22: Felt weak and under-extracted. Grind finer?". The "Searches" panel on the right shows a search pattern `{number:amount} g` with a table of results:

amount	value
11 g	11
200 g	200

Step B:
Write a *computation* that doubles the quantity and replaces the original text

The screenshot shows the same document as Step A. The "Searches" panel now shows a search pattern `{number * 2} g` with a table of results:

amount	scaledQuantity
11 g	"22 g"
200 g	"400 g"

Step C:
Use a formula to insert an interactive slider into the document

The screenshot shows the document with a new search pattern `Slider($)` applied to the text "number cups". The "Searches" panel shows a table with a slider:

value
1

Step D:
Connect the slider to the quantity scale factor to change the scale interactively

The screenshot shows the document with the text "number cups" circled and a value of 3 next to it. The "Searches" panel shows a search pattern `{number:amount} g` with a table of results:

amount	scale	col4
11 g	3	"33 g"
200 g	3	"600 g"

Figure 4-10: Creating an interactive quantity scaler for a coffee recipe in Potluck

4.4.1 Extracting data with searches

Potluck starts out looking like a familiar note-taking application; there is a list of notes to the left, an editable text area with the note's content in the center, and a search panel to the right. The first step towards scaling our ingredient quantities is to help the system recognize those values as structured information embedded within the recipe text. It's important that users can define flexible patterns that can accommodate messy real world data, but also that users have control over the patterns they define, and can develop consistent expectations about how they behave.

Potluck allows users to define patterns that are recognized within a text document. Patterns are created with a *search* interaction. Users are already familiar with searching for content in a word processor or web browser, so it's a natural on-ramp to creating live data detectors.

As an easy start, the user can search for a string literal: `11 g`, the quantity of coffee in our recipe. The search result appears in a table in the search panel, and is also underlined in the text note itself. Searches run continuously against the note's content, so the results update as we update the text note:

Of course, a string literal is usually too specific to find all the data the user wants—in this recipe, the search has detected the quantity of coffee, but not yet the quantity of water. The search needs to be generalized to find any number followed by the `g` symbol. In Potluck, the user can do this by rewriting their search from `11 g` to `{number} g` (Figure 4-10 Step A). This works because Potluck searches can reference other existing searches, by putting the name of an existing search inside of curly braces. Potluck comes with many simple patterns built in, including numbers, dates, and phone numbers.

Later on, the user will want to do arithmetic using only the number and not the unit, so they can also extract the number using a *named capture group*. If they edit the search to say `{number:amount} g`, the table of results will automatically show a column that just contains the amount.

For more advanced patterns, the language also supports arbitrary regular expressions. The user could have generalized their search to support other units besides grams, by changing the unit part of the search to use a regular expression: `{/g|kg|ml/:unit}`. However, regular expressions can be confusing to read and write, so we generally encourage encapsulating them inside of reusable named searches. For example, the built-in `number` search is internally implemented with a regular expression, but the user doesn't need to see that implementation detail.

4.4.2 Running live computations

Now that the quantities have been found in the text, the user needs to multiply them by some scaling factor. They can do this by creating *computed properties* that contain small JavaScript expressions.

In this case, the user adds a computed property called `scaledQuantity` that is the result of multiplying the quantity by a scale factor. For now, they want to make 2 cups of coffee, so they just write a simple formula: `amount * 2`. Later they'll make

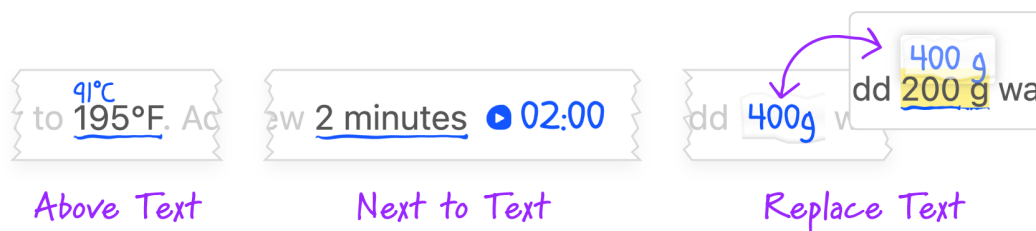


Figure 4-11: Potluck offers several annotation locations: above the text, next to the text, or replacing the text.

the scale factor adjustable. (Figure 4-10 Step B)

Our choice of JavaScript was motivated by convenience—it was easy to implement and easy to teach to people who already know JavaScript. However, we’ve applied some well-known design ideas from live programming and spreadsheets to try to make the environment friendly to program in. Each column computes a small pure expression that automatically re-evaluates reactively, and the current output is shown live in the table. We also provide built-in higher level functions for common high-level tasks that are cumbersome in JavaScript, like summing up a list of numbers. The result is that simple computations resemble small spreadsheet formulas, while the full power of JavaScript is still available to expert users.

4.4.3 Adding annotations

The user has computed a scaled value, but it’s not yet visible in the text document. They’d like to complete the scaler feature by covering up the original quantity with a scaled quantity. They can do this by setting up an *annotation*, which allows any column from a computation to show up in the text document.

Annotations don’t edit the original text content; they exist in a separate overlay layer. This preserves a clear separation between text and annotations, keeps the original text freely editable, and avoids circular feedback loops that could happen if annotations were themselves searchable. The visual design also differentiates between user input, displayed as normal text, and computational annotations, displayed with blue ink.

Annotations are placed in the document near the corresponding search result. The user can choose to place an annotation in one of several locations, each useful in different situations: Above Text, Next to Text, or Replace Text (Figure 4-11). When the “Replace Text” option is chosen, the annotation moves out of the way when the text editing cursor moves into the annotation, so that the underlying text can still be seen and edited.

In this case, the user tries out a few options for where to place the annotation, and decides to cover up the original text with the scaled value (Figure 4-10 Step B).

Interactive widgets

Now the scaled values are available in the text, but there's not a nice interface for *setting* the scale factor within the text. Perhaps after making coffee a few times, our user gets tired of editing the formula each time and would like to add a slider interface to set how many cups they plan to make.

In Potluck, the user can use dynamic annotations to add interactive widgets which are built into the environment and exposed via special formulas. In the search panel, they can call a `Slider()` formula which returns a slider widget as a value, and then use the annotation mechanism to display it in the document (Figure 4-10 Step C).

Now there is a slider in the document, but its value does not affect the rest of the document in any way. The next step is to wire up the slider to the quantities. Previously the user hardcoded a scale factor of 2; they need to replace this with a computed expression that retrieves the value from the slider.

Potluck contains a number of helper functions that allow a computed expression to pull data from other searches. They can return to the quantities table where they previously hardcoded the scale factor, and instead retrieve the scaling factor from the other search that contains the slider. They get the slider value with the formula `Find("scale").data.value`, and put that in a column called `scale`. They can then replace the previously hardcoded `amount * 2` with a new formula, `amount * scale`. Finally, they can add a unit label with JavaScript string interpolation: `${amount * scale} g`.

This completes the working recipe scaler (Figure 4-10 Step D).

In summary, we've seen how an interactive recipe scaler can be built in Potluck using a search for quantities, a computation that multiplies the quantities, and a slider widget annotation, all configured live within the tool.

4.4.4 Reusing searches

Next, our user wants to add another feature: a timer to help them track the coffee brew time.

They could build this up from scratch just as they did with the scaler, but because adding timers for durations in a document is a very general use case, they can instead **reuse an existing search** that someone else already made. The user opens the search panel to add a pre-existing search called `duration`, which recognizes duration strings like "2 minutes". This search also comes with a predefined computed property that shows a countdown timer, which is another kind of built-in interactive widget.

Reusing searches is a key concept in Potluck, because many searches for things like numbers, durations, and even Markdown syntax are useful across domains. For these common use cases, people shouldn't need to build everything from scratch. Reuse still allows for further remixing—the duration search's pattern and computed property are defined in userspace, so the user can still edit searches and computations at any time.

4.4.5 Other features

Changing text styling

Sometimes, instead of using a computational result to add annotations to the text, it's more helpful to change the styling used to display the text. Potluck supports this via *dynamic formatting*.

Consider a document with a watering schedule for house plants, including how often the plant needs to be watered, and when it was last watered. It's hard for a user to tell at a glance which plants should be watered today. One way they could solve this is by coloring dates in red if they're too far in the past and it's time to water that plant. The user can do this by adding a computed property which outputs “red” or “green”, and assigning that to control the color of the underlying search result. The resulting plant tracker is shown in Figure 4-3. Any CSS property can be addressed by dynamic formatting, including color, font size, and font weight.

Dynamic formatting is reminiscent of syntax highlighting in code—in fact, the Markdown preview syntax in Potluck (e.g., bolding headings) is powered by this mechanism. The difference from syntax highlighting is that both the parser grammar and the formatting rules themselves are meant to be easily editable on the fly by end-users. For example, it's straightforward to create a rule that a given word should always be highlighted when it appears in a document.

Editing the text programmatically

Both annotations and dynamic formatting share an important characteristic: they are overlays applied to the text at view time, and they don't durably edit the state of the underlying text in any way. We designed the system this way because it preserves a clear separation between the editable text content and downstream derived data, eliminating the potential for confusing feedback loops between the two layers.

However, we found that there were use cases where we wanted exceptions to this model. For example, in the plant tracker demo from above, every time we water a plant we have to remember today's date in order to record it in the log. It'd be much easier to have a programmatic way of inserting today's date into the text.

To support this, Potluck includes *buttons* that can insert or replace text in the document. Buttons are similar to other interactive widgets like sliders, but they have an additional capability for editing the surrounding text. The user specifies what should happen when the button is clicked: what text should be inserted, and where. Once the user has created a button, they can mark a plant as watered with a single tap. (These buttons are also shown in Figure 4-3.)

This button's update behaves just like any other text edit—we get features like undo and redo for free. Furthermore, changing the text causes the parsers and formatting to rerun, updating the plant color indicators. Also, because the text edit only happens upon an explicit user interaction, it's impossible to create runaway infinite loops where the computations and text update each other.

We've seen how users can extract data with searches, perform computation on the structured data, and inject annotations back in the text. Clicking a template button

1.) Cook spaghetti 100 g according to package directions.
2.) Meanwhile, in a bowl, whisk together lemon juice 25 ml, garlic 1 cloves, salt 0.5 tsp, agave nectar 0.5 tsp, mustard 0.5 tsp, & olive oil 1.5 tbsp. Set aside.
3.) When spaghetti 100 g is almost done, remove a bit of pasta water & reserve.

Figure 4-12: It's more convenient to follow a recipe when the quantities are shown inline in the directions.

changes the text and starts the cycle again. Potluck's interaction model turns static documents into an interactive and stateful computational medium.

Extracting spatial relationships

Now that we've seen a full loop of the interaction model, let's briefly return to the first stage of data extraction. So far, we've seen the user extract relatively simple patterns, but in some cases, the document contains richer structure that requires the ability to establish looser *spatial relationships* between different parts of the text.

Consider the example shown in Figure 4-12. A common problem with following recipes is looking up the quantity of a given ingredient while following the directions. It's useful if we can see the quantities directly within the directions.

To support this use case, we need to start from an ingredient in the directions (e.g., "spaghetti"), and find the corresponding quantity in the ingredients section of the recipe. This relationship can't be easily expressed in a pattern or a regular expression; we need a *spatial query* that can look across longer distances in the document. In this case, we can write a spatial query that runs the following logic: "Start from the ingredient name in the directions, get all previous ingredient names and take the quantity of the first ingredient with a matching name":

Or, in code that we could enter into a Potluck formula:

```
AllPrevOfType(ingredient, "ingredient")  
  .find(other => (  
    other.data.quantity !== undefined &&  
    other.isEqualTo(ingredient)  
  ))
```

We found spatial queries to be an essential tool for capturing relationships between parts of the text. For example, spatial queries can help associate a heading with the content underneath the heading, or help associate an ingredient with its nearby quantity.

🍋 Lemon Garlic Pasta

scale by 1

makes 2 servings

- 100 g spaghetti
- 25 ml lemon juice (freshly squeezed)
- 1 cloves garlic (minced)
- 0.5 tsp sea salt
- 0.5 tsp agave nectar
- 0.5 tsp Dijon mustard
- 1.5 tbsp olive oil
- 10 g toasted pine nuts
- 10 g fresh parsley (chopped)

- 1.) Cook spaghetti according to package directions.
- 2.) Meanwhile, in a bowl, whisk together lemon juice, garlic, salt, agave nectar, mustard, & olive oil. Set aside.
- 3.) When spaghetti is almost done, remove a bit of pasta water & reserve.

Figure 4-13: A spatial query that finds the quantity of an ingredient in the directions.

4.5 Evaluation: Experience & Limitations

In addition to using Potluck extensively ourselves to build various tools (many of which are shown above), we also conducted informal testing sessions with about a dozen people. Most of the test users were programmers familiar with JavaScript, but several of our testers were designers with only some limited programming experience. We gave users a brief tutorial on how to use Potluck and then observed as people tried to build their own tools in the environment. Here are some findings on the benefits and limitations of Potluck we learned from that process.

4.5.1 Versatility

A key question about this paradigm of gradual enrichment is how versatile it is. How far can you stretch these primitives, and when are they most useful? To explore this, we used Potluck to build documents for various use cases: tracking trip expenses, stock portfolios, planning trips, scheduling workshop agendas, managing a cash register, and more. (See the interactive demos in the introduction for some examples.)

In general, we found that the freeform nature of text made it possible to adapt Potluck to a wide range of use cases. Any text document, whether it contains prose or a structured personal micro-syntax, can serve as the starting point for an interactive tool. The medium is inherently permissive and flexible.

Meanwhile, some properties of Potluck—at least in its current form—limit the kinds of applications that can be built. Text must serve as both the input mechanism for the data and the substrate for designing the user interface, which obviously rules out applications with non-textual data or rich user interfaces and visualizations.

Working with fuzzy text data also increases the chances of errors—when reusing searches developed by someone else, it’s possible to write invalid data that isn’t recognized by the search. This is less of a problem in traditional software that performs stricter input validation.

Most of our example applications are also relatively small, with fewer than a dozen searches, each containing only a few computational columns. Even at this scale, we started to find it time-consuming to understand and modify tools built by others within our team. This is an unsurprising challenge that also emerges in traditional software and spreadsheets, but it suggests a need for better techniques to manage complexity.

4.5.2 Tool composition

We found that dynamic documents in Potluck naturally promote two kinds of composition.

The first kind of composition is reusing the same tool in different contexts. A timer can be used for remembering to take a pie out of the oven, or for holding a plank in a workout. A quantity scaler is useful in both recipes and event planning. Because all tools are made up of searches, tools, and annotations on a shared text substrate, it’s possible to reuse tools across documents.

Sometimes, simple tools like a timer can be trivially reused across documents; in other cases, we found it necessary to adapt or extend an existing tool. For example, often a search needs to be edited to support a different syntax. A quantity scaler might start out only looking for quantities in grams, and later need to be generalized to support more kinds of unit.

One unresolved design challenge is how to propagate changes across reused searches. In our prototype, changes to a search automatically affect all the documents where it's used. This makes it convenient to improve searches globally, but also makes it very easy to break other documents when editing a search. We suspect a better approach would be to decouple the usage of a search in different documents.

A second kind of composition is combining multiple tools and domains in a single document. One document can contain information that might have been tracked in separate apps, like combining meal planning with an exercise log, or combining a trip agenda and group expense tracker in the same document. These kinds of combinations feel very natural in freeform documents, but are often difficult to achieve in traditional software.

In larger documents, especially those dealing with multiple domains, we found that it was sometimes useful to restrict a particular search to only certain regions of a document. For example, when a recipe document has notes at the bottom, it might be useful to have recipe-related searches apply to the recipe part of the document, but not the notes. To support this, we built a mechanism where searches can be configured to only operate in part of a document.

4.5.3 Potluck vs. spreadsheets

We and our test users sometimes found ourselves preferring Potluck instead of a spreadsheet for performing simple computations.

One reason is that it's common to start out writing information in a text document, and a text-based computational tool avoids the need to move the information to a spreadsheet. We also noticed that it often felt easier to edit data in text than in a spreadsheet. This might be because we're more familiar with the affordances of text editors. Also, notes apps are more common than spreadsheets on mobile devices (and more ergonomic to use, since text wraps on narrow screens) which makes them convenient for editing on the go. Figure 4-14 shows an example of the same computation expressed in a text document and a computational table view.

However, in some cases Potluck felt more tedious than using a spreadsheet. Some documents contain many individual values which are hard to address using textual patterns; for example, the document in Figure 4-15 which lets the user enter 7 different variables as inputs to a calculator for a pizza dough.

In a spreadsheet it would be easy to just enter each of these values into a cell and reference the cell by name, but in Potluck, the user has to construct searches that can find these values in the text document based on some surrounding pattern. It's possible we could improve upon this by extending Potluck with some mechanism for identifying named values within the text itself.

Sun hoodies		sun hoodies ...	
		# g	Σ oz
red ridge merino	173g <u>6.1 oz</u>	173	6.102396089951
mint green Patagonia	144g <u>5.1 oz</u>	144	5.079451080653
blue nw alpine	136g <u>4.8 oz</u>	136	4.79725935395
teal rab	101g <u>3.6 oz</u>	101	3.562670549625

Figure 4-14: The same unit conversion computation, in Potluck on the left and a Notion Table on the right

number of dough balls: 10
ball weight: 40
water %: 50%
salt: 3%
oil: 1%
proof hours: 3
room temp: 75 F

Flour: 2250 grams
 Water: 2250 grams

Figure 4-15: A pizza dough recipe that computes flour and water amounts based on input parameters

Workout

rest 1 minute ● 01:00 in between sets

Gym 7/20/22

Bench 30 Squat 40, Maintain next time.

Gym 7/18/22

Bench 30kg 10x3
Squat 35kg 10x3 (easy, could increase weights next)

Figure 4-16: Text notes often contain implicit structure and relationships expressed through a personal micro-syntax.

4.5.4 Challenges of parsing

A key challenge in Potluck is accurately parsing structured information from a text document. We noticed that the difficulty of parsing depends a lot on how the text content arrives in the document: it’s much easier to parse information from a personal micro-syntax being typed into Potluck than it is to parse preexisting content being pasted in.

Accurately parsing structured information from preexisting text data—like a recipe from the internet—is extremely challenging. For example, a recipe might call an ingredient “pork”, but later refer to it as “the meat”—how do you recognize that these are the same entity? These are difficult problems in the field of natural language processing.

The situation is quite different for personal notes typed into Potluck. When the user controls the shape of the text and is receiving feedback as they type, it’s much easier to create text that conforms to the parsers active in a given document. For example, if fractions aren’t recognized as numbers but decimals are, you can either edit the number recognizer to understand fractions, or just rewrite the text to decimals. Editing the search seems like the more principled choice, but in practice we often found ourselves editing the text because it was easier to do in the moment.

We also found it very natural to represent information using lightweight text syntaxes. In personal notes, people implicitly develop syntaxes to write down information like times, durations, or domain-specific information, and these conventions can be encoded in Potluck as patterns. For example, the plant watering document from the demo section above showed a simple syntax for recording watering dates. The figure below shows another example, where workout activities are grouped underneath dates. We found that Potluck’s combination of pattern searches and spatial queries was generally flexible enough to capture the underlying structure in many different kinds of informal syntax (Figure 4-16).

4.5.5 State and UI in text

In Potluck, application state lives in the text. For example, you might note the last watered date for a plant using the text syntax `08/31/2022`, or use `[x]` to indicate a completed task. There is no hidden metadata; searches are just a function of the text.

Text editors are generic and refined tools that have many built-in features like copy/paste and undo/redo. Having state directly in the text gives us these features for free. For example, you can copy a document to a different text editor to edit and then paste it back into Potluck, and it retains all of its behavior. By using text as the source of truth, Potluck inherits the affordances and powers of text.⁶

In some cases, our demos violate this general principle by storing ephemeral state which isn't stored in the text. For example, our default timer widget doesn't store the remaining time in the text, so a running timer won't survive a copy-paste. This wasn't a particularly principled decision though; in theory, any state that can be encoded as text can be stored in the document itself.⁷

In Potluck, text also serves as the *output* medium for displaying computed data. We enjoyed designing tools in Potluck because the text medium forces a simple one-dimensional layout and avoids many of the choices found in conventional UI layout. Of course, there are limits to this approach—many kinds of apps and views don't make sense to build using text as the substrate. But we didn't notice these limits too acutely when building apps in Potluck, perhaps because we were already in the mindset of “writing a text document” and not “designing an interface”. Still, to extend the range of scenarios where Potluck is useful, one direction for future work could be to allow users to extend text documents with richer visual components.

4.5.6 Limitations

In our testing sessions, we discovered several key limitations of Potluck.

Recursive searches

A key design decision we made in Potluck is that searches are not recursive: annotations in the document cannot feed into further searches. On each edit of the document, the system performs a single loop of searching and annotating. We believe avoiding recursive searches keeps the mental model and system implementation much simpler—for example, since the order in which searches are executed is irrelevant, and infinite loops are not a concern—but it also limits certain kinds of composition.

For example: imagine a recipe document where a user wants to both convert units on quantities and scale up quantities. If the system supported recursive searches, these

⁶Originally we tried allowing users to manually highlight entities in the text. We abandoned this approach mainly because manual highlighting was tedious, but also because it created hidden state outside the text that was hard to reason about.

⁷The text-based todo list app TaskTXT has a good solution to storing timer state in the document. When a timer is started, it records the start time into the text document in a human-friendly format. The result is that even a running timer can survive a cut-paste.

could be written as two separate spreadsheets; the unit conversion sheet could produce annotations which would then appear as search results in the quantity scaling sheet. But in the absence of recursive searches, we must express these two computations in a single table.

Live searches vs. manual tagging

Some systems allow users to manually tag an arbitrary span of text with an annotation based on the position of the span in the text. For example, a Google Docs comment behaves in this way. In Potluck, we do not allow this—all searches must be expressed as a pattern match over the contents of the text. This design avoids hidden state and enables the user to always reason about the behavior of searches for example, copy-pasting text works in a predictable way.

However, this design decision can also make it harder to express certain kinds of searches. A common example is when displaying the result of a computation at a specified place in the document. The user cannot specify a location in the document that will serve as an anchor for the annotation display; instead, they must write a label in text, and author a search that will only match that label.

Formula language

In general, users with strong JavaScript proficiency generally found it much easier to program the system, but users with limited JavaScript experience often required substantial hints to successfully create a Potluck document.

One particular point of confusion was the way that search results are represented. In the Potluck formula language, search results are not represented as primitive JavaScript datatypes like strings or numbers; they are stored as references to locations in the text; preserving this metadata enables formulas that traverse spatially from a given search result to find related data. As a result, search results must be explicitly converted to primitive datatypes like numbers before they can be used in regular JavaScript computations. Users often got confused about whether a given cell in a table contained a search result or a regular JavaScript primitive. This could likely be improved through better language design and UI affordances.

Some participants also struggled with referencing data across tables. Potluck allows formulas in a table to reference data from another table, or even another document; this functionality is used, for example, in the coffee recipe to bring the slider value into the scaling computation. We observed that users made more mistakes when trying to reference data across tables than when computing within a single table; one reason was that the editor environment does not offer support for correctly referencing the name of another table. Writing computations across tables also sometimes required effectively writing relational joins (i.e., looping over the contents of one table once for each row of another table, and performing some kind of aggregation); currently this must be done manually in JavaScript, and first-class language support would make these kinds of computations easier to express.

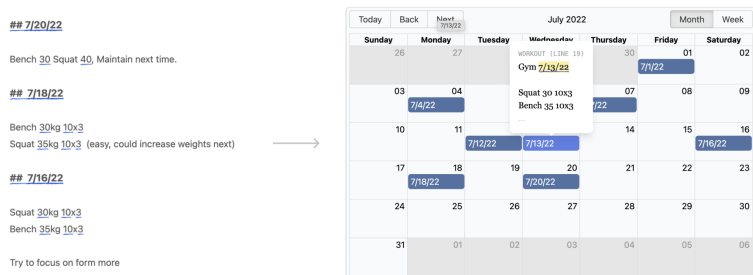


Figure 4-17: Showing a calendar view of a workout note in Potluck

4.6 Future Work

Structured data views

Annotations in the text can't create more complex layouts or graphical visualizations. To address these shortcomings, we did some small experiments with structured views that are defined outside of the text. For example, Figure 4-17 shows a text note recording recent workouts, with a calendar view that shows the dates of the workout in a more convenient format.

One thing we found surprisingly useful in this experiment was to show *context* around each search result within the structured view. You can hover over any date and see the text from the original document surrounding that date, which makes the structured view a useful window into the original text.

We think it would be interesting to explore structured views in more detail. What is the relationship between text and structured data views? Do the views live inside of the document or are they separate from the text? As a starting point, people could connect their text notes to existing structured views, including generic views like calendars. As a step beyond that, we envision people creating their own views using some mechanism besides traditional programming.

Integrating machine learning

There are limits to how well the deterministic pattern descriptions we used in Potluck can interpret structured data out of text. For example, consider the difficulty of finding all the ingredients in a typical recipe—how do you decide on the list of known foods? How do you match up mentions of the same ingredient that use different words?

We think AI and machine learning could help solve these kinds of problems and make Potluck both more approachable and powerful. Recent advances in large language models like GPT-3 have shown the power of ML for interpreting structured information out of text. Some Potluck searches might be made both more accurate and more easily specified by using a language model (Figure 4-18).

Another role for ML could be in helping end-users write code for search queries

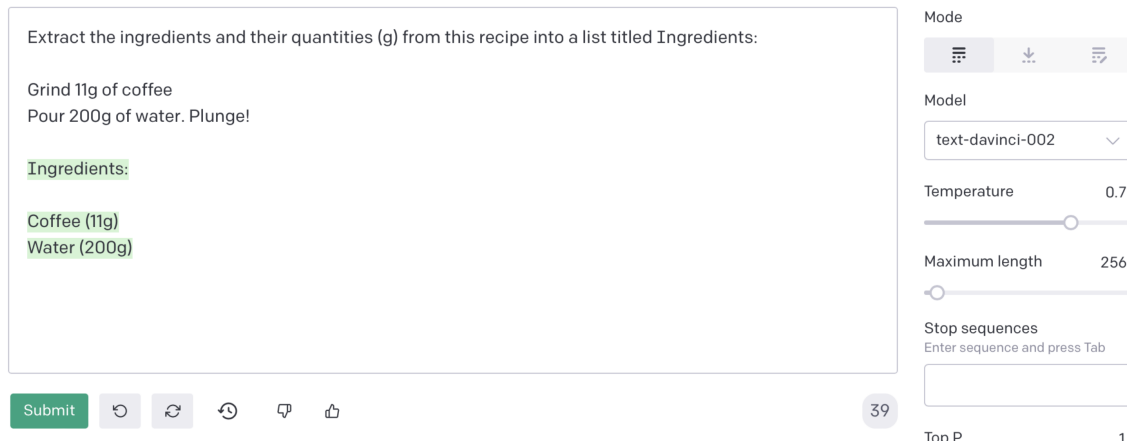


Figure 4-18: Extracting the ingredients from a recipe using GPT-3

and computations, using an approach like GitHub Copilot. This would still result in fast, predictable searches, but could make it easier to author them. In Chapter 6, we demonstrate an experimental prototype of adding this approach to Potluck.

There is a delicate tension between ML-based automation and maintaining user control. We’d want to avoid an approach that delegates too much work to an AI model; every step of the process should be repeatable, inspectable, and understandable.

Machine learning could also help extend the ideas of Potluck to other types of source material, like handwritten ink, photos, or video. Many of the same core ideas would apply: extracting meaningful symbols from freeform data.

4.7 Conclusion

In this chapter, we’ve shown how searches, computations, and annotations can come together to enable users to gradually enrich text documents with structure and meaning.

While we’ve shown these ideas in the context of a specific research prototype, we don’t necessarily think that a new “notes app” is the right way to implement them. Perhaps these concepts could be more powerfully applied at deeper levels of the stack.

Imagine an operating system with the principles of Potluck deeply woven in. People could start by just organically recording information however they want. As they come across places where the computer could help them, they would gradually add *structure* to their data, but only as much structure as is needed for the task at hand. They would then add bits of computational behavior, borrowed from others or created from scratch, to complete the task. The resulting tools might resemble “apps”, but in fact would be precisely tailored to one’s own needs. The tool fits the workflow, rather than the workflow fitting the tool.

In summary, Potluck demonstrates how reactive table interfaces can help people build small personal tools on top of existing unstructured text data. We have shown

that it is possible to use a table UI to perform computations over structured tabular data, while simultaneously allowing for flexible data entry that avoids premature formalism.

Chapter 5

Riffle: Reactive Relational State for Local-First Applications

In the previous chapters we have demonstrated how ideas from reactive databases can help end-users develop simple personal tools. In this chapter¹, we apply similar techniques to a different problem: supporting skilled programmers who are building complex applications.

5.1 Introduction

A key part of application development is *managing state*: displaying a view of the application’s data and keeping that view updated over time. For example, in a music app, the application state includes the structure of a user’s music library—the tracks, playlists, favorites, etc.—as well as UI state, such as the currently selected album, or the sort order applied to a list of tracks. The application’s user interface (UI) can be thought of as a live visualization of queries over these underlying sources of state: for example, fetching and sorting the tracks within the currently selected playlist.

One powerful abstraction for defining such queries is *reactive data transformation*, as seen in early UI frameworks like Garnet [60] and in modern frameworks such as React.js. In a reactive system, a developer declaratively specifies data transformations and dependencies, freeing them from manually propagating updates. The concept of reactivity is perhaps best known in spreadsheets, arguably the most successful paradigm for allowing programmers of all levels of skill to describe complex computations. In fact, prior work [18, 9] has shown that spreadsheets themselves can even be used as a substrate for specifying reactive data transformations that power simple GUI applications.

¹The material in this chapter is adapted from the following paper: Litt, Geoffrey, Nicholas Schiefer, Johannes Schickling, and Daniel Jackson. 2023. Riffle: Reactive Relational State for Local-First Applications. In The 36th Annual ACM Symposium on User Interface Software and Technology (UIST ’23), October 29–November 1, 2023, San Francisco, CA, USA. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3586183.3606801> [47]. The main ideas were developed together with Nicholas, advised by Daniel. Johannes led the development of Overtone and contributed improvements to Riffle.

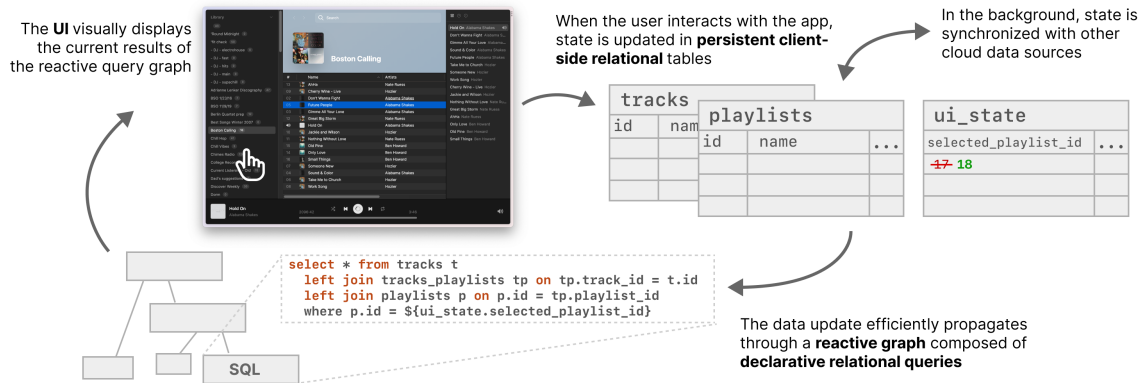


Figure 5-1: An overview of the Riffle architecture. The UI visualizes the results of a reactive graph of relational queries on a persistent client-side relational database. The dataflow loop runs synchronously on the UI thread, supporting fast, transactional reactivity. In the background, the local relational database is synchronized with other data sources over the Internet.

In practice, however, real-world applications tend to require more complex mechanisms than a spreadsheet to define reactive dataflow, for two main reasons. First, most spreadsheet languages lack the expressive power needed to code a complex application. Second, many real-world applications handle large amounts of data and have strict performance requirements. As a result, state tends to be spread across many layers—a backend database, a server-side ORM, a client-side cache, in-memory UI state, etc.—breaking the conceptual simplicity of a reactive system and forcing the developer to reason across many layers of abstraction. Reactivity is often present in part of the stack (e.g., at the view layer), but not all the way through. The core simplicity of “application as spreadsheet” has been lost.

In this chapter, we propose Riffle, a novel architecture for application state management which both provides a simple conceptual model and scales up to meet the needs of complex real-world applications (Figure 1). We use a *local-first* [39] architecture where all of an application’s state is stored in a persistent client-side relational database. All updates, whether to domain state or UI state, flow synchronously through this database. In the background, data can be synchronized over a network. Figure 5-1 visualizes this architecture.

Our architecture has two key concepts:

- **Reactive relational queries.** Data transformations are represented by a directed acyclic graph of relational queries. Reactivity ensures that updates propagate automatically through the graph without intervention from the developer. Relational queries let the developer provide high-level declarative specifications of transformations while benefiting from performant query execution.
- **Synchronous transactional updates.** Whenever a write occurs to the database, downstream dependencies are synchronously updated within a transaction. Since UI state and domain state are both handled in the same app system, we can guaran-

tee that the UI and all state visible to the programmer is always in an internally consistent state, without inconsistencies across parts of the view.

A naive implementation of this architecture would have unacceptably slow interaction latencies, since any interaction might require re-executing expensive relational queries before updating the UI. To make this architecture fast enough to support a responsive UI, we build on top of SKDB², a relational database that supports efficient incremental updates and network synchronization. A primary contribution of Riffle is applying performance advancements in incremental databases to enable simpler abstractions for application programming. End-users also benefit, since applications built in Riffle have fast interaction latencies.

We have implemented these ideas in a TypeScript library that provides APIs for application developers to specify data transformations using reactive relational queries. The library uses React³ as a view templating framework, which turns the reactive query results from Riffle into DOM elements in the browser.

We demonstrate the benefits of Riffle in two case studies. First we build the TodoMVC reference application, showing how Riffle provides a simple model for reactive state in a small-scale application (Section 5.6). Next, we describe a formative case study where Riffle has been used over the course of a year to develop Overtone, a professional music management application. The resulting application demonstrates how Riffle scales up to support large volumes of data and stringent performance requirements, while preserving simplicity for the developer and speed and reliability for the end-user (Section 5.7.1). To further analyze the benefits and tradeoffs of Riffle, we present a heuristic analysis following Olsen’s criteria for evaluating systems research [65] (Section 5.7.2).

Riffle suggests a new way of thinking about application development. By tightly coupling the UI to a fast, reactive client-side database that supports ergonomic queries, Riffle offers a kind of *stack compression* that simplifies the data architecture, leading to applications that are simpler for developers to create and maintain, easier for systems engineers to optimize, and faster and more reliable for end-users. Looking ahead, this approach also lays the foundation for future benefits like end-user customization and data-centric interoperability across applications.

5.2 Background

The complexity of state management

Today, application developers bear much of the complexity of managing state in their applications. Most web applications have a multi-layer architecture involving at least three programs running on different computers: a client-side program running in JavaScript, a server-side program written in a language like Python or Java, and a database queried with a language like SQL. Developers must wrangle data across these

²<https://skdb.io/>

³<https://react.dev/>

layers, manipulating copies of the same information, often in different languages and representations. They must also manually manage the network boundary, including serializing data and appropriately handling latency or failures. State is often persisted in various caches along the way to improve performance. Reactivity may exist within the view layer, but often does not span across the entire stack.

Experienced industry developers report that they are forced to spend their effort thinking like distributed systems developers, instead of focusing on the application itself. One developer, Tristan Hume, writes [31]:

When I’ve worked on any kind of distributed system, including systems as simple as a web app with frontend and backend code, probably upwards of 80% of my time is spent on things I wouldn’t need to do if it weren’t distributed. . .

In addition to requiring tedious effort, application-level state management code is also a common source of bugs [4]. Writing correct code for distributed state management is a famously hard problem, even for distributed systems engineers [83].

Riffle simplifies application development by managing all state on behalf of the programmer, in a single client-side system that subsumes the responsibilities of various parts of the traditional state management stack. In doing so, it supports better correctness, expressiveness, and performance. Riffle achieves this goal by building on recent advances in three related research areas, which we describe next.

Applications as spreadsheets

Anyone who has used a spreadsheet is familiar with the value of one-way reactive constraints. Spreadsheets are easy to use because of Alan Kay’s “value rule” [36]: every cell has its value defined by a rule which is reactively maintained, making it easy to understand and debug the provenance of computations. More broadly, reactive constraints have a long, successful history in user interface development, ranging from early systems like Sketchpad [75], Thinglab [15], and Garnet [60] to modern frameworks like React and SwiftUI.

In fact, reactive constraints provide such a simple model of application state management that they can even support end-user programming. This has been demonstrated by spreadsheet-driven application development environments—including research projects like Gneiss [18] and Quilt [9], as well as commercial products like *Airtable*⁴ and *Glide*⁵, which have successfully enabled end-users to define application dataflow logic in a spreadsheet UI.

However, while these environments successfully achieve a “low floor,” so that the simplest applications are easy to build, they also have a “low ceiling,” ruling out more ambitious applications due to limitations in expressiveness and performance. For example, *Airtable* has a limit of 100,000 rows in a collection, and does not support full SQL (omitting even relational join), and *Quilt* [9] incurs latency by routing UI

⁴<https://www.airtable.com>

⁵<https://www.glideapps.com>

updates through a cloud-hosted spreadsheet. As a result, these tools are often useful for building small-scale applications, but do not extend to full-featured, commercial-grade applications.

Riffle takes inspiration from these systems and gives the developer a simple mental model by preserving the spreadsheet model for state management. At the same time, it also achieves the performance required to drive large-scale, complex applications, as we demonstrate in Section 5.7.1.

Incremental view maintenance

Performance is a key challenge for reactive systems, especially those that process considerable amounts of data, like large music and photo libraries. To achieve performant reactivity, Riffle applies advances from the database research community in *incremental view maintenance* (IVM) [13]. In many databases, the user can specify a *view* over some tables in the database as a SQL query. With IVM, the database can efficiently keep that view up to date as small changes occur to the underlying data, without recomputing the view from scratch upon each update. In effect, IVM is a structured form of reactivity for relational query languages like SQL, rather than imperative or functional programming languages. We summarize related work on IVM in more detail in Section 5.3.

Local-first software

Riffle employs a variant of the *local-first* architecture [39] in which all application data is replicated to the client device, and the client is treated as a source of truth. The UI can access and edit the local data at any time, regardless of whether a network connection is present. Local and remote updates both update the client-side datastore, which in turn updates the UI view.

Local-first software is not local-only. Cross-device collaboration is still possible because a background process takes care of synchronizing data when a network is present. Cloud persistence is also possible, by synchronizing with a peer hosted in the cloud, but the conventional role of the cloud server is diminished.

The key benefit for developers is *stack compression*. Rather than needing to manipulate and transport data in many different layers, the developer is instead freed to think in more synchronous, local terms. They can simply read from and write to a local datastore, and let a general data management abstraction handle the process of synchronizing data across the network. As an architect for one local-first application explains⁶:

To create a new feature as an engineer, you essentially render and modify local in-memory data structures to build new functionality. All the complexity that comes with requests, conflicts, network errors and retries are handled by sync for free.

⁶<https://twitter.com/artman/status/1558081815113617409>

Local-first software offers also offers benefits for end-users. Interaction latencies can be lower since more data is available on-device without needing to access the network. Offline mode becomes easier to implement because data is eagerly loaded onto the client and can be edited directly there. The end-user gains more privacy and ownership over their data, since the data primarily lives on their client rather than on a server.

The local-first model is a good fit for applications that help manage personal information, or collaborative work with a small team. For example, the local-first architecture has been applied to a note-taking app, a budgeting app⁷, an issue tracker⁸, an email client⁹, and an RSS reader¹⁰. Our music manager case study in Section 5.7.1 also fits this profile. In these applications, the benefits of fast, reliable, offline-capable UI are particularly salient—they manage important information for serious use, and are accessed repeatedly throughout a day. These are also applications where all the state relevant to a user is small enough to be replicated fully to the client. In contrast, a social network or an e-commerce site might be a more challenging context in which to apply a local-first architecture; we expand more on the limits of the local-first pattern in Section 5.7.3.

5.3 Related work

There are many systems that handle various aspects of managing application state, including data storage, query models, cross-device synchronization, and reactive UI. In this section, we summarize three categories of work which we draw on in Riffle to create a novel combination.

Applications as spreadsheets

First, we discuss work focused on reactive dataflow. In the space of web applications, we can roughly differentiate between client-side and full-stack approaches.

Client-side reactivity: A vast number of frameworks have been developed for maintaining reactive dataflow within the UI client. The idea goes back at least to early UI frameworks like Garnet [60, 59], as well as functional reactive programming libraries like Flapjax [56] and Vega-Lite [69]. Modern web view templating layers like React.js¹¹ and Vue.js¹² provide automatic reactive maintenance of UI trees, and also provide a basic approach to application state. Special-purpose state management libraries such as MobX¹³, Recoil¹⁴, Jotai¹⁵, and Datascript¹⁶ provide additional utilities

⁷<https://actualbudget.com/>

⁸<https://linear.app/>

⁹<https://superhuman.com/>

¹⁰<https://readwise.io/read>

¹¹<https://react.dev/>

¹²<https://vuejs.org/>

¹³<https://mobx.js.org/>

¹⁴<https://recoiljs.org/>

¹⁵<https://jotai.org/>

¹⁶<https://github.com/tonsky/datascript>

for managing state and reactive dependencies outside the tree of UI components, as Riffle does.

Riffle shares the basic idea of these tools, but has two important differences from them.

First, client-side reactive UI libraries typically do not offer relational queries as a first-class citizen, forcing the developer to manually compute relational queries in a general-purpose language like JavaScript. As one example, the Redux¹⁷ state management library for React recommends¹⁸ representing state in a normalized form, but then suggests that the developer query this data using generic JavaScript. Manual joins across tables written by application developers are less declarative and require more work to optimize. (One exception to this generalization is Datalog, which provides a Datalog query engine and was one of our inspirations for seeing the utility of relational data management in client UIs.)

A second difference is that in web-based systems, client-side reactive frameworks often are assumed to only be dealing with a small partial subset of all the data, with most data held on the server. This assumption allows for a simple reactivity model within the scope of the client, but fragments the reactivity of the system as a whole into multiple parts which the developer must manage manually. In contrast, Riffle applies reactive relational queries to a much larger collection of data (e.g., the user’s entire music collection), which reduces fragmentation.

Full-stack reactivity: These systems provide developers with simpler abstractions for managing the entire stack of a client-server system, in particular providing automatic reactivity across the network. Quilt [9] and Object Spreadsheets [54] provide end-users with tools to define web applications that can persist data to a server backend, while defining data transformations in a spreadsheet interface. Links [23] and Ur/Web [20] implement a “tierless” web development pattern, where a single statically typed functional program is compiled into programs that run on the database, the server, and the client. Commercial systems like Meteor¹⁹ and Firebase²⁰ have offered tools for managing reactivity across the client-server boundary for years. Newer commercial efforts in the space include Convex²¹, which offers a custom reactive transactional database, and Electric Clojure²², which is a Clojure/Script DSL that automates reactivity across the network boundary.

Riffle shares the general goal of these systems: making it easier to reason about reactive dataflow throughout an entire application. However, it differs in its approach to the network. In full-stack web frameworks, a slow and unreliable network is often kept as a core part of each user interaction, incurring UI latency. To avoid this, some frameworks like Meteor perform *optimistic updates* on the client before updates are sent to the server, which makes them more similar to local-first frameworks which we discuss next.

¹⁷<https://redux.js.org/>

¹⁸<https://redux.js.org/usage/structuring-reducers/normalizing-state-shape>

¹⁹<https://www.meteor.com/>

²⁰<https://firebase.google.com/>

²¹<https://www.convex.dev/>

²²<https://github.com/hyperfiddle/electric>

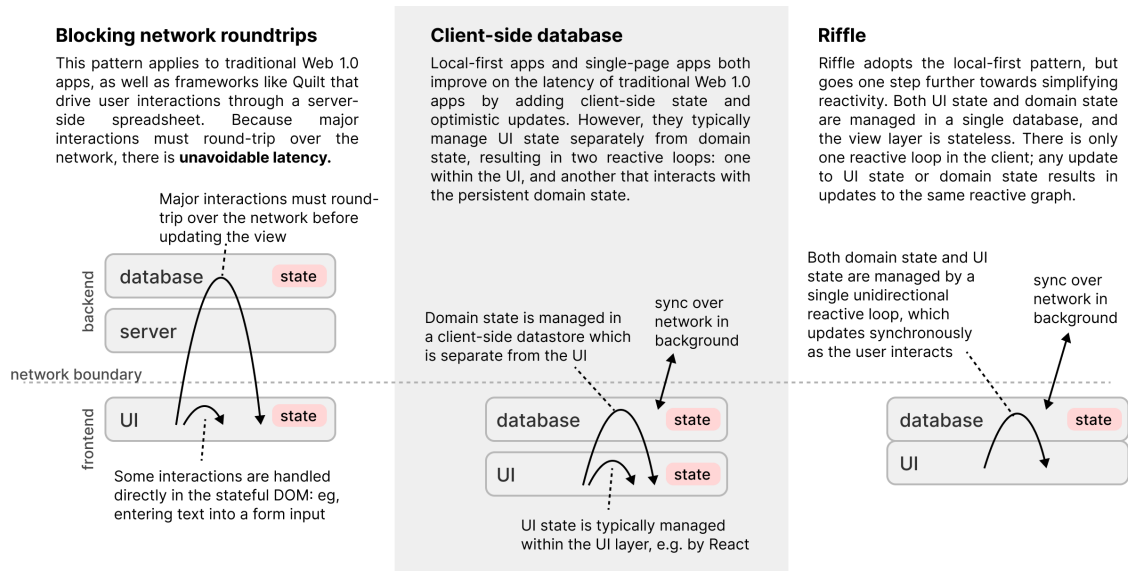


Figure 5-2: A comparison of different approaches to managing reactive UI state. Riffle simplifies reactive dataflow by offering a single performant reactive loop for managing the entire state of a user interface, including both UI state and domain state.

Local-first frameworks

Kleppmann et al [39] proposed local-first software as a solution to the problems of depending on the network during user interactions. The key idea is to eagerly synchronize application state to the client and treat the local client-side data storage as a source of truth that can be queried or written to at any time, synchronizing across the network whenever it is available.

Our approach proposes a tighter coupling between the user interface and the data storage layer than most existing local-first systems. Existing frameworks typically preserve a distinction between UI state and domain state—this resembles the split between client and server state in the traditional web architecture, but with both parts contained within the client. In contrast, Riffle takes advantage of the presence of local data and unifies all state into a single system, enabling benefits such as synchronous transactional updates explained below.

Another difference between Riffle and existing tools is the support for a relational data model. Many prominent libraries supporting local-first software, such as Automerge, Yjs, PouchDB, and Replicache, use a simple document data model which lacks support for executing relational queries. Some systems like TinyBase, VLCN, and Electric SQL²³ do provide a relational model, but do not tightly couple the UI and database to the extent that Riffle does.

²³<https://electric-sql.com/>

Incremental view maintenance

Riffle builds on decades of work on incremental view maintenance (IVM) in database systems. Summarizing the extensive literature on IVM is beyond the scope of this thesis; we refer the reader to [72] for a survey of classic IVM techniques. Some form of IVM can be found in most mature, commercial relational database management systems (RDBMSs) such as Oracle, Microsoft SQL Server, and DB2, although they haven't become widespread in popular open source RDBMSs such as PostgreSQL²⁴. Typically, IVM is implemented only for a subset of the SQL query language²⁵: for example, it is not common to support incremental view maintenance for correlated subqueries or recursive computed table expressions, although techniques for these are known [1, 28]. Other systems, such as Noria [26], add some form of IVM on top of existing RDBMSs.

In the past decade, several new databases built specifically for efficient, low-latency, and universal IVM have been developed. Materialize is a data warehouse designed specifically to support low-latency IVM of complex SQL queries²⁶, building on the timely dataflow [58] and differential dataflow [55] incremental computation frameworks. SKDB, the embedded database we use in Riffle, achieves fast IVM through Skip²⁷, a systems programming language that offers native support for dependency tracking and incremental cache invalidation.

5.4 Key Concepts

The two main concepts in the Riffle architecture are *reactive relational queries* and *synchronous transactional updates*. In this section we motivate those concepts and explain how they benefit both developers and end-users.

5.4.1 Reactive relational queries

A significant fraction of application code is spent transforming state to display in the user interface—for example, joining together and grouping data about tracks, albums and artists, to show in a playlist view in a music player.

Riffle's abstraction for these kinds of data transformations needs to balance two seemingly conflicting goals. First, in order for UI state updates (like hover, selection, and clicks) to feel responsive, Riffle must offer very low latencies, similar to those that might be achieved by explicitly coding all interactions imperatively in a low-level language. At the same time, Riffle must also provide high-level abstractions that are friendly for application developers, without requiring a deep understanding of systems programming and performance optimization.

²⁴https://wiki.postgresql.org/wiki/Incremental_View_Maintenance

²⁵<https://docs.oracle.com/database/121/DWHSG/refresh.htm#DWHSG8372>,
<https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-refresh-sql-command.html>

²⁶<https://materialize.com/blog/olvm/>

²⁷<http://skiplang.com/>

To resolve this tension, Riffle takes inspiration from two sources, *reactive programming* and *relational queries*, which each provide declarative abstractions to programmers along different dimensions. Riffle combines these complementary paradigms and expresses its computational work as a graph of *reactive relational queries*. In doing so, it gives users simple, flexible, and high-level abstractions that can be used to specify performant data transformations.

Reactive programming. Reactive programming provides a declarative approach to propagating updates. Downstream dependencies are kept up-to-date by the system rather than manually managed by the user.

In a reactive system, the burden of performance for individual transformations is strict, because a slow transformation could be called at any time when one of its inputs changes. One technique that can be used to improve performance in a reactive system is to make incremental updates occur at a finer granularity—for example, only recomputing part of an individual query.

Relational queries. The relational model [21] entails storing data in tables, and then executing queries that join together data across tables to fetch the needed results. It encourages storing data in a *normalized* form, where each piece of data has a single canonical representation without redundant copies.

The relational paradigm provides a declarative approach to specifying data transformations. In a relational language like SQL, a programmer describes their program as a series of high-level operations on relations; it is the job of the database to determine how to execute those logical operations efficiently. Crucially, the relational paradigm decouples the *semantics* of queries from the efficient *read and write access patterns* that implement them. For example, a user can join on any column of a table efficiently (as long as appropriate indices are defined), while in a document or nested object model, lookups must be restricted to the keys of a document or fields of an object, since other lookups are much slower.

While relational databases have been widely used for decades, including in web server backends and for data storage in desktop applications, they are rarely used in web application clients. Perhaps as a result, many existing local-first systems, which take inspiration from frontend tools, rely instead on key-value or document data models [38], which couple efficient access patterns with the application schema and do not accommodate sophisticated queries.

Combining reactive and relational. Reactive programming and relational queries each provide their own type of declarative programming:

- Reactive programming enables the programmer to declaratively specify a function over state, and then implements efficient *updates* of that function when the state changes.
- The relational model makes individual data transformations declarative, and provides a data model that decouples read, write, and storage patterns.

In Riffle, we combine these two kinds of declarative programming in *reactive relational queries*. A Riffle application is defined by a graph of relational queries, maintained through reactive updates. The application developer describes a directed

acyclic graph of data transformations, primarily as relational queries in a SQL-like query language, including business logic that would normally be written in a non-relational language.²⁸

The results of these queries are automatically kept up-to-date through a reactive programming system. In order to update fast enough, the individual queries in the graph must also be run on a reactivity-aware database that offers efficient, fine-grained reactivity *within* a query. We discuss further in Section 5.5 how we achieve this by building on a database that supports this functionality.

When combining reactivity and relational queries, the two forms of declarative programming described above reinforce each other—it is easier to implement incremental algorithms for relational queries than it is for imperative code with mutable state, and relational queries also present a simple programmer-facing abstraction over the complex internal logic driving incremental updates. This allows us to take advantage of the decades of database research on performantly indexing and querying relational tables without requiring application developers to understand the details of systems programming.

Despite this natural connection, combining reactivity and relational queries has seen only minimal adoption in application development frameworks, even as reactive programming models have becoming widely adopted for building applications. The core observation behind Riffle is that reactive relational queries are a natural and powerful abstraction for specifying data transformations in applications.

5.4.2 Synchronous transactional updates

Synchronous transactional updates ensure that, in the typical flow of using an application, the UI is always both *consistent* and *responsive*.

Consider the scenario depicted in Figure 5-3. There is an application whose UI includes a sidebar that selects the content shown in the main pane. At first, Item A is shown in the main pane; then, the user clicks on Item B in the sidebar. How does the UI react?

Typical web app. In a single-page web application (SPA), data is often only loaded after the user interacts (Figure 5-3, left side). Before showing Item B in the main pane, the UI must wait for roundtrip network traffic, as well as time spent on the backend server and database to query and serve the relevant data. During this time, the user is forced to wait for the UI to update. The UI is also in an *inconsistent* state, since Item B is selected in the sidebar, but Item A is still shown in the main pane, which can be confusing for the user if it lasts a long time.²⁹

²⁸Recognizing the limitations of existing relational query languages like SQL, Riffle also allows developers to write queries in GraphQL or as pure TypeScript functions. See Section 5.5.

²⁹There are other design options available that make different tradeoffs between responsiveness and consistency, but there is no way to avoid the latency of data loading. For example, in a traditional server-rendered “multi-page” web app (MPA), this user interaction would trigger a new page load, and the browser would show a blank loading screen before showing the new consistent page. This architecture prioritizes consistency more than the SPA, since the sidebar and main pane always match each other, but it sacrifices responsiveness, since nothing is shown while the data loads.

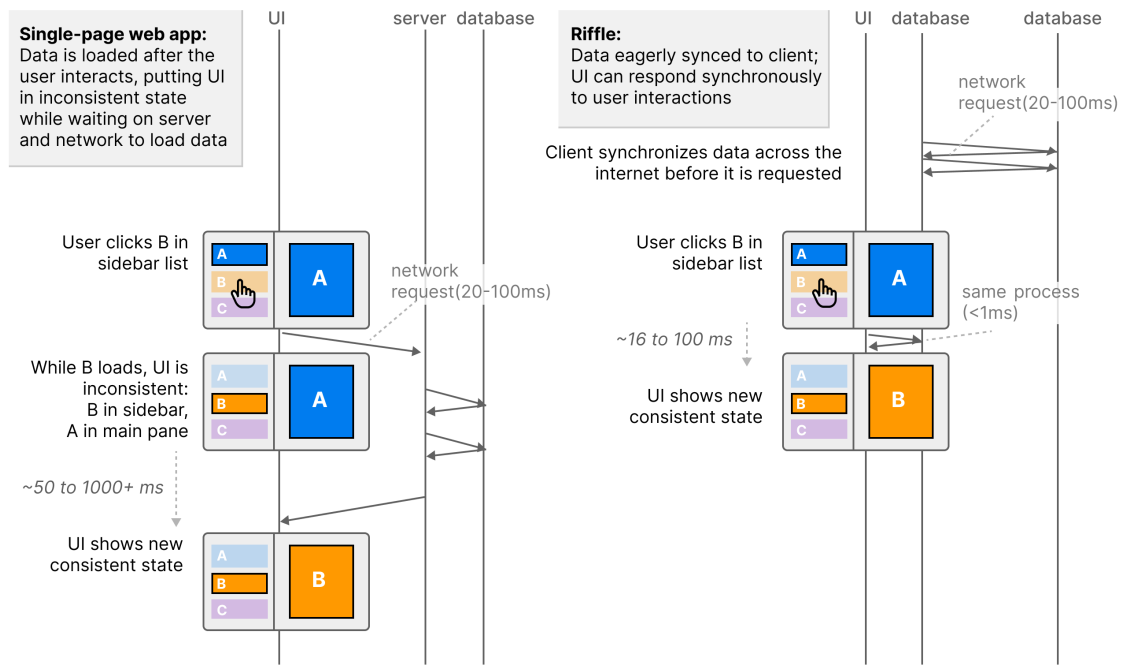


Figure 5-3: In a single-page web application, user interactions frequently incur network latency and leave the UI in a temporarily inconsistent state. In contrast, Riffle’s local-first architecture and synchronous transactional updates enable faster responses. The UI can respond to the interaction immediately without showing inconsistent loading states because the data was synchronized to the client before the user explicitly requested it, and database queries are efficiently updated within 16ms.

Riffle’s approach. In contrast, using Riffle, all the data needed by the user is synchronized ahead of time to a client-side database (Figure 5-3, right side). Because the data for pane B is already preloaded into a database running within the UI process, the application is able to quickly render a final state that reflects the user’s interaction. Throughout the execution, the UI is always in a consistent state; in fact, it is impossible for this UI to show a state where the sidebar selection does not match the main pane, because the two are transactionally updated together.

Coordinating UI state and domain state. Riffle’s approach to managing both UI state and domain state together helps support synchronous transactional updates. In many application frameworks, UI state is managed within the view layer, whereas domain state is managed in a separate persistent layer. This separation of concerns makes it difficult to keep the UI internally consistent. For example, in the scenario above, a traditional framework would manage the selection state of the sidebar within the view layer, and the data shown in the main pane in a separate persistent layer. The sidebar updates immediately after the interaction (because the view framework drives that process), while the main pane loads separately and asynchronously. In contrast, in Riffle, managing both UI state and domain state inside a single reactive database makes it possible to achieve transactional consistency.

Response times. What is a reasonable time budget for responding to user interactions? Nielsen defines 100ms as the rough response time limit for an action to feel instantaneous to a user [63]; some applications like Superhuman aim for under 50ms³⁰. In general, Riffle aims to fall well within these limits by usually responding to user actions within 16ms (representing a single frame on common 60fps displays), and treating 100ms as an upper limit. Notably, removing network latency from an interaction is necessary but not sufficient for responsiveness; recomputing queries must also be fast, which we support through reactive relational queries.

The limits of synchrony. While it is possible to avoid many of the sources of asynchrony seen in typical web applications by using a local-first architecture, we cannot entirely avoid asynchrony in all cases. There may be some cases where we need to make a request to an external API or run a heavy computation.

Riffle models these kinds of processes as asynchronous side effects outside of the synchronous update cycle. When an asynchronous effect completes (e.g., a networked API responds with data), that may trigger an update on the synchronous state, just as a user interaction would. Segregating asynchrony in this way preserves the simplicity of the core synchronous update loop.

The need for occasional asynchrony does not invalidate our general design principle. We still eliminate unnecessary asynchrony as much as possible, by eagerly syncing data to the client, loading it into memory, and managing UI state and domain state together in a transactionally consistent datastore. As we explain in Section 5.7.1, we have found that this approach is sufficient to eliminate many sources of asynchrony that are commonly found in streaming music applications.

³⁰<https://blog.superhuman.com/superhuman-is-built-for-speed/>

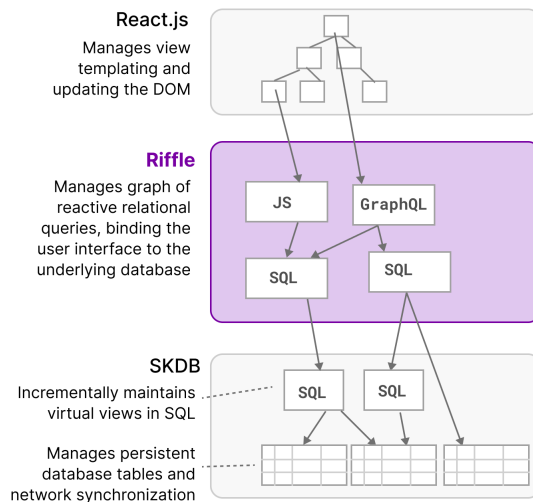


Figure 5-4: Implementation architecture: The Riffle library sits between React (for view templating) and SKDB (for all data storage and queries)

5.5 System Implementation

In this section we describe details of the implementation of Riffle as a TypeScript library. Figure 5-4 shows a high-level overview of the architecture.

5.5.1 Relational Database Backend

As a backend for persistence and querying, Riffle uses a relational database called SKDB, which has two particularly useful properties for building the kinds of UIs that Riffle aims to support.

First, SKDB supports **performant incremental updates**: when a small change is made to a table, the results of queries over that table can be updated much more efficiently than recomputing from scratch.³¹ The result of an incrementally maintained query is materialized into memory, and is known as a *virtual view*. Virtual views can have indexes defined on them that are also incrementally maintained.

Second, SKDB supports **data synchronization** over the network. Changes made to the database can be synchronized live between clients through a server using WebSockets. Synchronization can be enabled at a per-table level, which is useful for making some changes local to each device or user.

A full discussion of the details of SKDB’s incrementality and synchronization is out of scope for this thesis; we treat these features as black boxes. These are highly general primitives that we depend on to enable the Riffle architecture.

³¹Incremental updates enable the Riffle architecture. In a previous iteration of Riffle, we implemented it using the popular SQLite database as the backend. This worked conceptually, but was too slow in practice—some expensive joins took hundreds of milliseconds to recompute, which would block the UI thread and cause noticeable lag.

Riffle’s contribution is to provide abstractions and architectural patterns for using this underlying relational database to ergonomically construct a user interface. The remainder of this section shows many examples of such patterns, including mechanisms for locally binding queries and state to UI components, and dynamically generating queries as a UI evolves.

5.5.2 View Framework

Riffle integrates with React.js as a view templating layer. In principle, the ideas could apply to many view libraries; we chose React because of its popularity.

In React, developers use *hooks* to incorporate state, effects, or library logic into a view component. Riffle defines a hook called `useRiffleComponent` that developers can use to specify the data dependencies of a component. In its simplest form, the developer can simply pass in a single relational query as an argument, indicating that the component should subscribe to that query. In more complex forms, the developer can subscribe to multiple queries at once, establish dependencies between them, and/or specify a schema for local component state. We will see examples of all these forms in the case studies below.

5.5.3 Reactivity Algorithm

In an ideal world, the reactive graph could be fully *dynamic*: any query could be initialized at any time and maintained reactively from then on. In practice, however, many queries are too expensive to initialize dynamically in response to a user interaction, for the same reason that a non-reactive database is too slow to power Riffle in general. For this reason, Riffle supports two layers of reactivity: a *static* layer and an *on-demand* layer.

The static layer is implemented using SKDB’s virtual views, which are defined statically in the code, globally scoped over all the data, and initialized when the application first boots. They are then incrementally maintained eagerly by SKDB. These virtual views typically compute expensive joins over the application’s state.

The on-demand layer represents a smaller set of currently active queries being used by the UI, and is maintained within Riffle itself. The on-demand layer invalidates and reruns queries with table-level reactivity: the programmer annotates which queries depend on which tables (or virtual views) in the database, and also which writes affect which tables, and then the reactive graph automatically reruns queries when data changes.³² Updates propagate through the graph in a topologically sorted order; we compare new values to old values at every node and perform early cutoff if a query returns the same results as it previously returned.

In some cases, it is desirable to defer reactivity and avoid immediately updating the UI for performance reasons. For example, a batch process might be performing many writes in the background, and there is no reason to perform expensive UI updates

³²Table annotations are optional; if omitted from a query, then the query will rerun upon every write to any table. Also, we envision removing manual annotations in a future version of Riffle, by taking deeper advantage of SKDB’s built-in reactivity.

after every individual write. In a database with support for concurrent writes we could handle this with a separate write process executing a long-running transaction, but this is not possible running the database in the UI thread. As a result, we also offer support for application code to indicate certain updates which should not trigger a “push” update to the UI. Stale query results are marked, and the next time a normal update occurs (e.g. in response to user interaction), all queries are brought up to date to reflect a consistent state in the UI.

5.5.4 Query languages

Virtual views in SKDB are defined in SQL. For dynamic queries, Riffle lets the developer choose between one of three languages:

- **SQL** offers a powerful declarative model for joining, filtering, and aggregating over relational data. Many web developers already have some familiarity with SQL.
- **GraphQL**³³ provides an additional layer on top of SQL with several advantages. It has a concise syntax for specifying simple traversals of object graphs, and has the ability to directly produce nested tree-shaped result sets which are often needed to construct a UI tree.
- **JavaScript** can also be used to write any computation as a query, as long as it is pure and side-effect free. This may be more verbose than SQL or GraphQL, but is also more flexible for expressing arbitrary logic.³⁴

5.5.5 Dynamic query generation

Many user interfaces require queries where the logic of the query itself depends on the results of other queries. A common case is that a query contains a parameter which must be bound to a value. Sometimes, the level of dynamism needed exceeds the simple parameter binding available within a SQL query—for example, we might want to entirely omit part of the query if some runtime condition holds. Riffle supports highly dynamic queries by allowing the developer to specify SQL query fragments as strings using JavaScript; we show some examples of this in Section 5.6.

5.5.6 Query scope

Queries can exist in two scopes: they can either be *global*, or *local* to a component. Global queries are initialized outside of the UI tree, and are typically maintained as long as the application is running. A local query has a lifetime and scope tied to a

³³<https://graphql.org/>

³⁴There are currently some restrictions on the ordering of the languages in the graph, because SQL and GraphQL queries can only run directly on database tables. The results of a SQL query can flow into a JavaScript query that applies further transformations, but the results of a JavaScript query cannot be queried using SQL. This is an incidental limitation of our implementation, and not a principled choice.

specific component in the UI tree: it is initialized together with the component, maintained while the component exists in the tree, and torn down when the component is removed.

Local queries accomplish two goals. First, they are necessary for efficiency, since they provide a way to subscribe only to data that is currently being used in the UI. They also provide a convenient scoping abstraction, since they are able to incorporate local component state into the query, as we describe next.

5.5.7 Local component state

UIs are commonly constructed out of a tree of components that each maintain local state. Riffle provides an abstraction for easily managing such state within component code, while storing it in a persistent relational database.

For a given component type, the developer defines a *state schema*, a set of columns representing the state associated with each instance of that component. For example, for a `TrackList` component in a music app, the developer might specify `scrollPosition` and `selectedTrackId` in the state schema.

Given this schema, Riffle automatically creates a database table with the given columns, which will store one row per instance of this component type. Inside a component instance, the developer may read and write the local state values; Riffle maps these to queries and writes over the appropriate row in the component state table.

Each component instance is associated with a *component key* which uniquely identifies that instance. By default, Riffle automatically generates a unique key for each component instance that is created in the UI, meaning that the state of the instance will never be loaded from persistent storage. If the developer would prefer that the local state of the component is saved, they can define a stable key; for example, the natural key for a `TrackList` component would be the ID of the playlist being shown by that list.³⁵

5.5.8 Performance architecture

We make two implementation choices that are crucial for performance.

First, we run SKDB synchronously in the main UI thread, in order to avoid messaging overheads associated with sending large data blobs to a separate process such as a Web Worker. This brings the risk that slow queries will block the browser from updating the UI, so ensuring fast queries through incremental maintenance is essential.

Second, we batch state updates to React. Whenever there is a state change, we update all queries in the reactive graph before sending them all to React in a single batch. This choice has semantic importance, since it means we will never render a

³⁵So far, we have not found it necessary to implement a system to garbage collect stale local component state, but such a system could be straightforwardly implemented by saving a creation timestamp with each component state record, and periodically removing old records as space limitations are hit.

UI that only contains some of the downstream updates implied by a state update. It also brings a performance benefit, since React does not need to repeatedly update in response to the same state change.

As mentioned above, we also give the application code tools to defer reactivity even further, for batch write processes which do not need to immediately reflect their results in the UI.

5.5.9 Debugger

We have implemented a simple debugger (shown in Figure 5-5) which exposes the underlying structure of a Riffle application, showing:

- A live view of the tables in the underlying database, including their data and schemas
- A live view of the queries currently executing over the database, and their results
- An interactive console where the user can execute arbitrary queries over the current state of the database

The debugger brings some degree of live programming facilities to Riffle, because the developer can understand the data and queries backing the application, and try out new queries. Although it currently cannot actually edit the underlying application code stored on the filesystem, the developer can still prototype a query in the debugger and then copy-paste it into their code editor.

5.6 A simple example: Todo List App

In this section, we demonstrate how the Riffle concepts apply in practice, by implementing TodoMVC³⁶, a small reference application commonly used to compare UI and state management tools. We will walk through developing a relational data schema, specifying a reactive query graph, and binding the queries to the user interface.

Relational schema. Designing a relational schema for a Riffle application is similar to designing a schema for the backend database of any web application. One minor difference from traditional backend data modeling is that eventually our *UI state* will also be modeled in the relational schema. For now, we begin by just modeling our *domain state*, in this case a simple table `todos` with the following schema:

- `id: string`
- `text: string`
- `completed: boolean`

A simple reactive query. To display the list of todos in our database, we can define a reactive SQL query that loads all the todos. We write it inside a React component using a hook provided by Riffle called `useRiffleComponent`.

³⁶<https://todomvc.com>

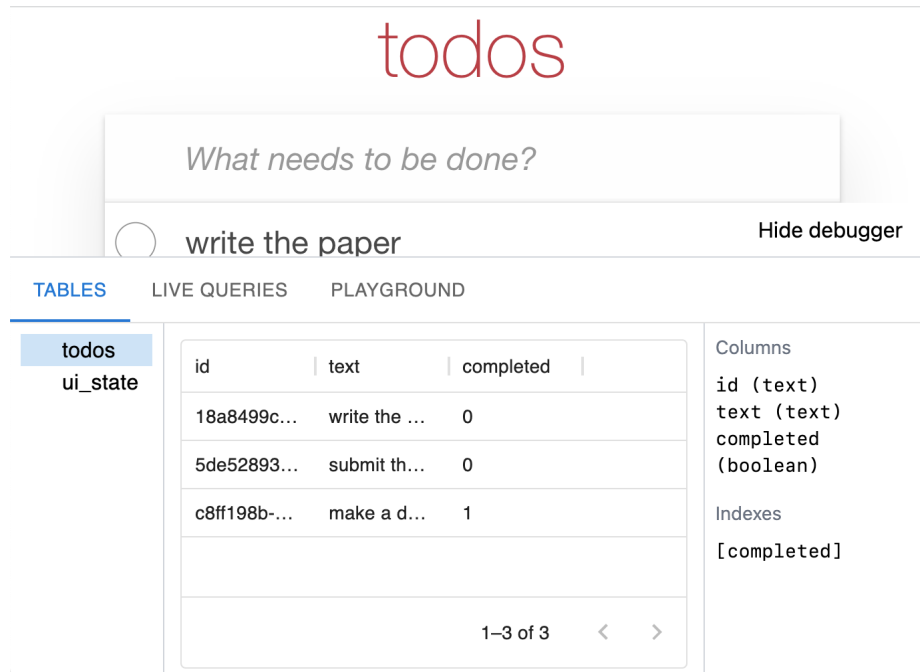


Figure 5-5: The Riffle debugger shows a live view of the data in the underlying database. Other tabs (not shown) include the current reactive queries and an interactive SQL console.

```
export const MainSection = () => {
  const { todos } = useRiffleComponent({
    queries: ({ rxSQL }) => {
      todos: rxSQL(sql`select * from todos`)
    }
  })

  return <ul className="todo-list">
    {todos.map((todo: Todo) =>
      (<li key={todo.id}>{todo.text}</li>))}
  </ul>
}
```

This hook establishes a subscription to the results of this reactive query, and returns a list that we can use in the view template. Every time the contents of the `todos` table change, this component is re-rendered with the new list.

Because the table of `todos` is managed through Riffle, it is not just available in memory. It is also automatically persisted locally and synchronized over the network.

Storing UI state. Next, we need a text box where the user can type in text for a new todo. Typically, in React, the state of the input box would be treated as local in-memory state, but in Riffle, we instead model this UI state in the database.

To store the text box state, we can create a new table `ui_state`, with a single text column named `newTodoText`. Because there is only ever one text box in `TodoMVC`,

we will only ever have one row in this table, so there's no need for a further key. We can also configure this table to not be synchronized over the network, since it's typically useful to keep UI state local per device.

To read the value, we can define a reactive query which subscribes to the value. (We use a Riffle helper called `asScalar` which extracts a value from a single-row, single-column table.)

To update this value, we can define a *write event* in the database schema named `updateNewTodoText` which performs a SQL update. (Write events provide a thin abstraction over SQL statements because the same updates are sometimes used across multiple parts of the UI.) Now, in the UI component, we can bind the text input to the value of the todo text in the database:

```
export const Header = () => {
  const { newTodoText } = useRiffleComponent({
    queries: {
      newTodoText:
        rxSQL(sql`select newTodoText from ui_state;`)
        .asScalar()
    }
  })

  return <input
    value={newTodoText}
    onChange={(e) => store.applyEvent(
      'updateNewTodoText',
      { text: e.target.value })}
  />
}
```

This provides the same “unidirectional dataflow” that is typically used in React for managing state. Instead of letting the DOM input element manage its own state, the input is treated as a pure view of an underlying state. In this case, though, we have routed that unidirectional dataflow through a persistent database, instead of just the view framework.

Transactional updates. When the user hits the enter key in the text box, we want to simultaneously (1) create a new todo with the text in the box and (2) clear the contents of the text box. Using Riffle, we can apply events for those two updates within a *transaction*. This means that the UI will never be in a state where the text is still shown in the box but the list of todos is not updated; the state of the entire app will tick in a single transactional step.

Chaining reactive queries. So far, we have seen *static* query definitions. While the results of the query may change at runtime, the query definitions themselves have been fixed. However, in some cases, the definition of the query depends *dynamically* on the results of another query. In `TodoMVC`, we will need this functionality to let the user filter the list to incomplete or completed todos.

First, we must store the value of the filter toggle; we can extend our UI state in the database by adding a new column, `filter`, to the existing `ui_state` table. Next, we need to use the filter state to perform the actual filtering. When the filter contains the value `all`, we can use the same query as above. But if it is set to `completed`, then we'd like to append a clause to our query: `select * from todos where completed = true`.

Figure 5-6 shows how this chaining is established in Riffle. We first write a reactive SQL query which loads the value of the filter from the database. Then we initialize a JavaScript computation which computes a SQL clause applying the appropriate filter in a query. (It refers to the previous SQL in the chain using a name `filter$` which has been assigned.) Finally, we compute a SQL query which interpolates that clause and loads filtered todos. By writing these computations, a reactive chain has automatically been established. Whenever the contents of the `todos` table or the value of the `filter` in `ui_state` change, the visible todos will be updated.

A more complex query. So far, we have not gained too much from the relational model, since we only have a single `todos` table storing domain state. However, the benefits of the relational model grow as our application grows in complexity.

For example, imagine we wanted to add labels to the TodoMVC app, where a todo can have multiple labels, and the user can filter by label. In a document data model, we might start by embedding label names directly inside the documents for todos, but this denormalized data model makes it hard to do things like rename a label or efficiently find all the todos with a given label.

A relational model is a natural fit for this kind of data modeling. We could simply create a `labels` table with a `todos_labels` join table, and then query across the tables with joins. It would be very easy to view the todos in one or more labels, or to filter together by label and completed status—all with reactive updates.

5.7 Evaluation: Experience & Limitations

5.7.1 Case study: Music Application

The benefits of Riffle become more evident in a *data-intensive* application that manages a large amount of data in a complex schema and has stringent performance requirements. We have used Riffle to build exactly such an application: a music management application called Overtone. In this section we first describe the features of the application, and then share reflections from the development process.

Goals

Overtone is a web and desktop application that allows a user to access their music in two streaming music services, Spotify and SoundCloud. It also supports subscribing to music podcasts via RSS. To enable the reactive relational paradigm, the user's entire metadata library, as well as the UI state of the application, is stored locally in a Riffle

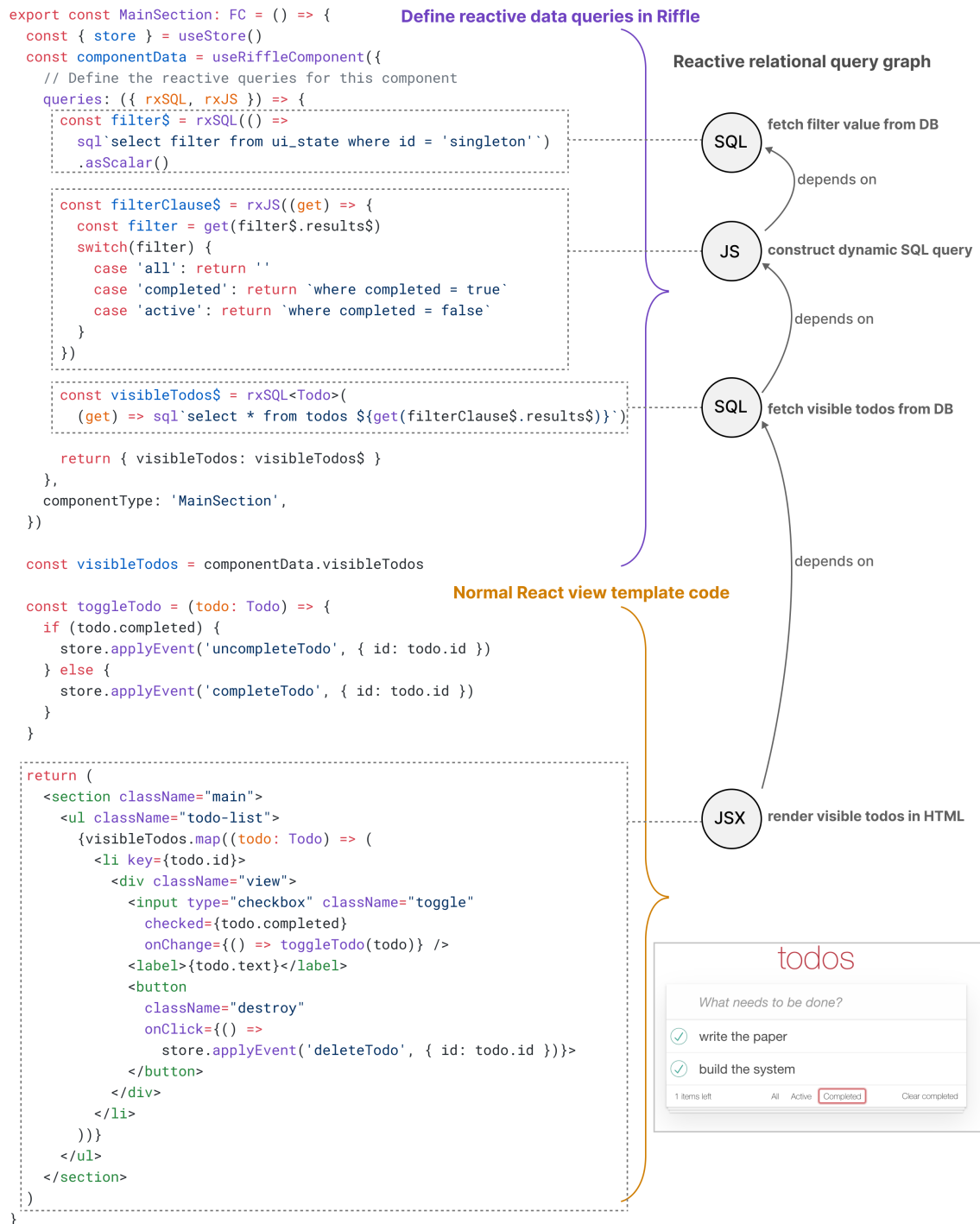


Figure 5-6: TodoMVC includes a simple example of a dynamic SQL query. The currently active filter setting is queried from a table using a SQL query. A JavaScript query then turns that value into a filter clause in a SQL string, which in turn queries the todos table to produce the final filtered data for the view.

database after being downloaded from the streaming provider.³⁷ Music playback still happens via streaming network API calls, because Spotify and Soundcloud don't allow users to download music files. Overtone is currently in alpha with limited usage, but intended eventually for commercial release.

Overtone aims to improve upon the experience of existing streaming music clients in both *performance* and *flexibility*.

Performance. Overtone's performance goal is to respond to all user interactions within 16ms to render smoothly at 60fps; as a minimum threshold, we adopt Nielsen's limit of 100ms [63] for interactions feeling "instant". This target is far beyond the performance seen in music clients for many popular streaming services. For example, the Spotify desktop client can take thousands of milliseconds to switch between views for different playlists owned by a user, even if the playlists are short and have already been recently loaded; we suspect the main culprit is network access. It also exhibits flickering effects like black screens while data is loading.

Flexibility. It is often useful to see a music collection through a variety of different views: browsing by playlist, album, or artist; sorting by various fields; or filtering and doing full text search. Overtone has a general goal of offering flexible views of music metadata. Storing the music collection in a relational database that can be queried with SQL makes it easy to support a variety of rich views over the data. In contrast, some of these capabilities are surprisingly absent in music streaming service clients; for example, Spotify offers no way for users to view all tracks by a given artist.

Data schema

The relational schema for Overtone currently includes 15 tables. These tables store domain state like tracks, albums, artists, playlists, podcasts, as well as relationships between those entities. In the future, this schema is likely to grow to support more complex entities such as genres and tags. The schema also includes tables for storing UI state such as navigation state, the current play queue and playback state, and the user's authentication credentials.

We also define 8 virtual views over the base tables to efficiently join together data, e.g. joining data about tracks and artists into a single result table. One example of such a virtual view is shown in Figure 5-7. We also define a GraphQL schema on top of the base relational schema; we describe further below the problems that motivated the GraphQL layer.

Application features

In this section, we describe several important features of the Overtone application, and how they use Riffle to achieve the goals of performance and flexibility.

Metadata synchronization. Overtone currently supports adding tracks from two streaming services (Spotify and SoundCloud) or a podcast RSS feed. Overtone imports changes in a user's Spotify and SoundCloud libraries into their local library by

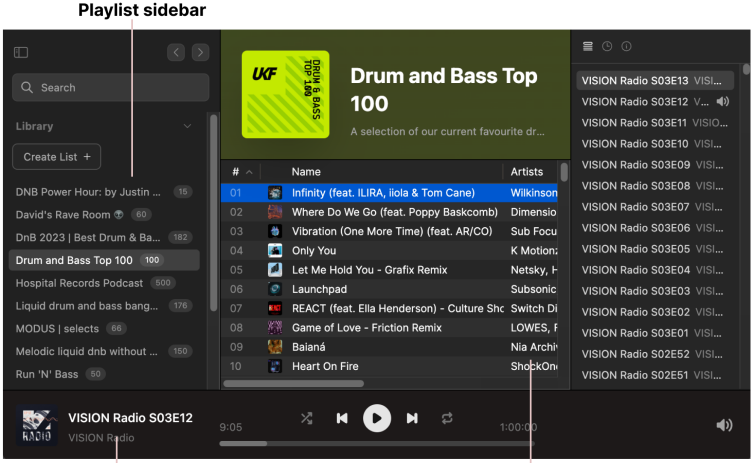
³⁷Overtone only eagerly synchronizes the data which a user has saved to their collection; it would be impractical to synchronize all the music in Spotify's global catalog to a client device.

The playlist sidebar queries data from an incrementally maintained virtual view which contains a list of playlists with track counts

```
select * from view__playlists_with_track_counts
order by name ASC
```

The definition of the virtual view for playlists with track counts

```
CREATE VIRTUAL VIEW view__playlists_with_track_counts AS
SELECT _p.*, count(_tp.trackId) as trackCount
FROM library_playlists AS _p
LEFT OUTER JOIN view__tracks_playlists
AS _tp ON _p.id = _tp.playlistId
GROUP BY _p.id
```



Playback bar

```
SELECT * FROM playback_state
```

The playback bar runs a simple query to read information like the play/pause state and the offset within the track.

The Tracklist queries data from an incrementally maintained virtual view which joins together information about tracks, albums, artists, and playlists into a table.

```
create virtual view view__tracks_playlists as
select *
from tracks t
left outer join albums
on t.album_id = albums.id
left outer join tracks_artists
on t.id = tracks_artists.track_id
left outer join artists
on artists.id = tracks_artists.artist_id
left outer join tracks_playlists
on t.id = tracks_playlists.track_id
left outer join playlists
on playlists.id = tracks_playlists.playlist_id
```

```
select
  addedAtTimestamp,
  trackIndex,
  id,
  name,
  albumName,
  artistName,
  ...
FROM (
  SELECT *
  FROM view__tracks_playlists
  WHERE playlistId = 'playlist-123'
)
ORDER BY trackIndex asc
limit 100 offset 0
```

dynamic parameters which change based on the current state of the UI

Figure 5-7: Examples of a SQL query and view definition used in the Overtone music manager.

polling over the network.³⁸ The writes for these imports are scheduled and throttled so that they do not interfere with smooth operation of the UI while the import is happening.³⁹ Virtual views are incrementally updated as imports happen, amortizing the cost of computing whole-table joins across many incremental updates.

The track list. A central feature in Overtone is the *track list*, (shown in Figure 5-7) which displays a table of music tracks drawn from a playlist, album, or collection of works by an artist, filtered and sorted by properties selected by the user. For each track, the UI shows metadata about the track, and about linked entities. To efficiently load this data, we query a virtual view which joins together this information.

The track list also supports sorting and full text search. These features can be supported by *chained queries* in the reactive graph. Just as we applied filtering to a list in the TodoMVC example, we can apply a **where** clause to the query based on a text search input, or apply an ordering based on the column selected for sorting. Chained queries also offer a convenient abstraction for implementing *virtualized list rendering*. Many virtualized list implementations require complex layers of caching, but in Riffle there is a simple solution: we track the user’s scroll position as UI state, use that scroll position to determine a window of tracks that should be visible, and then pass that information to the query that loads the tracks.

The track list uses the local component state mechanism described in Section 5.5 to store the current selection, sort property and direction, and scroll position. This supports convenient *persistent UI state*; for example, the sort order and scroll position for each playlist is saved by default.

The playlist sidebar. The left sidebar in Overtone shows a list of playlists, with a count of tracks within each playlist. The data for this component is once again defined by an incrementally maintained virtual view (shown in Figure 5-7). The currently selected playlist is stored in Riffle; we have found that this is a useful piece of UI state to persist across reloads of the application.

When the user clicks on a playlist in the sidebar, we trigger an update on the UI state for the currently selected playlist. In turn, this propagates a change through the reactive graph, which stabilizes in a transactional way.

Other features. Riffle supports a variety of other features in Overtone, including a text search box that filters items in the music collection, using a relational query with a text filter; playback state (e.g., time and overall duration) for the currently playing track; and a queue upcoming tracks to play.

Performance. Some Overtone users have libraries containing many tens of thousands of tracks. This scale of data introduces performance challenges for the developer building the track list. We must efficiently find the tracks within a given collection, join in associated metadata for albums and artists, and apply any relevant sorting and filtering to the collection. The view might need to change if the metadata library

³⁸Because Overtone has so far been designed to show a view of music collections which are already stored in existing cloud services, we have not yet used SKDB’s synchronization features to share the state of relational tables across devices for Overtone. However, product development in the near future will likely involve synchronizing the state of relational tables between devices.

³⁹Currently the throttling happens in the application layer; we plan to incorporate support for throttled background writes into Riffle itself in the future.

changes, but also every time the user performs an interaction, like selecting a new track.

In early experiments, we found that joining together this data at interactive latencies was challenging. Even a mature relational database like SQLite, with appropriate indexes, would sometimes take over 300ms to join together the data for the track list with a large music library. This might be acceptable latency for a traditional app architecture where state is spread across many layers with different performance guarantees, but it is too slow for running directly within a UI.

With SKDB, we achieve more predictable performance. Individual reads and writes usually complete within 10ms, since expensive joins are incrementally maintained—and often much faster, on the order of 1ms. In exchange for some memory overhead and some small time overhead on writes, reads are made efficient.

Although Riffle generally offers good performance by default, during the development of Overtone, adding new features has sometimes resulted in performance regressions where interactions exceed the 100ms threshold. So far, we have been able to solve these kinds of problems by restructuring reactivity (e.g., moving work from read queries into eagerly maintained virtual views), or performing standard UI optimization techniques like reducing the size of result sets using virtualized rendering. We have also found the need to optimize other parts of the UI stack beyond data transformations.⁴⁰

Debugging experience

We have implemented several debug views inside of Overtone which make use of Riffle’s dependency tracking metadata to help understand the state of the system.

One main debug view, shown in Figure 5-8, shows a history of recent updates that have occurred in the Riffle reactive graph, including writes that have triggered queries to re-run, as well as new queries that have been registered by new UI components being mounted. This view provides visibility into the behavior of the system: what actions are occurring, and which queries are updating in response. Because Riffle explicitly tracks dependencies in a reactive graph, it is easy to access the causal information about which queries were triggered by which upstream actions.

The debug view also helps understand the performance of the system, by showing the total time taken to run queries in response to an update, as well as the time taken by each individual query.

We have found having a historical timeline of recent events in the application to be a useful complement to a current live view of the system state. The historical view enables a workflow where we can perform an interaction and then inspect afterwards the events that occurred during that interaction.

⁴⁰We found that UI responsiveness is often impaired not just by delays in loading data, but also by delays at the rendering layer. After optimizing the data loading in Riffle, we found that updating the browser DOM through React.js had become the new bottleneck, so we re-implemented the table view using a library⁴¹ that draws to the more efficient Canvas API.

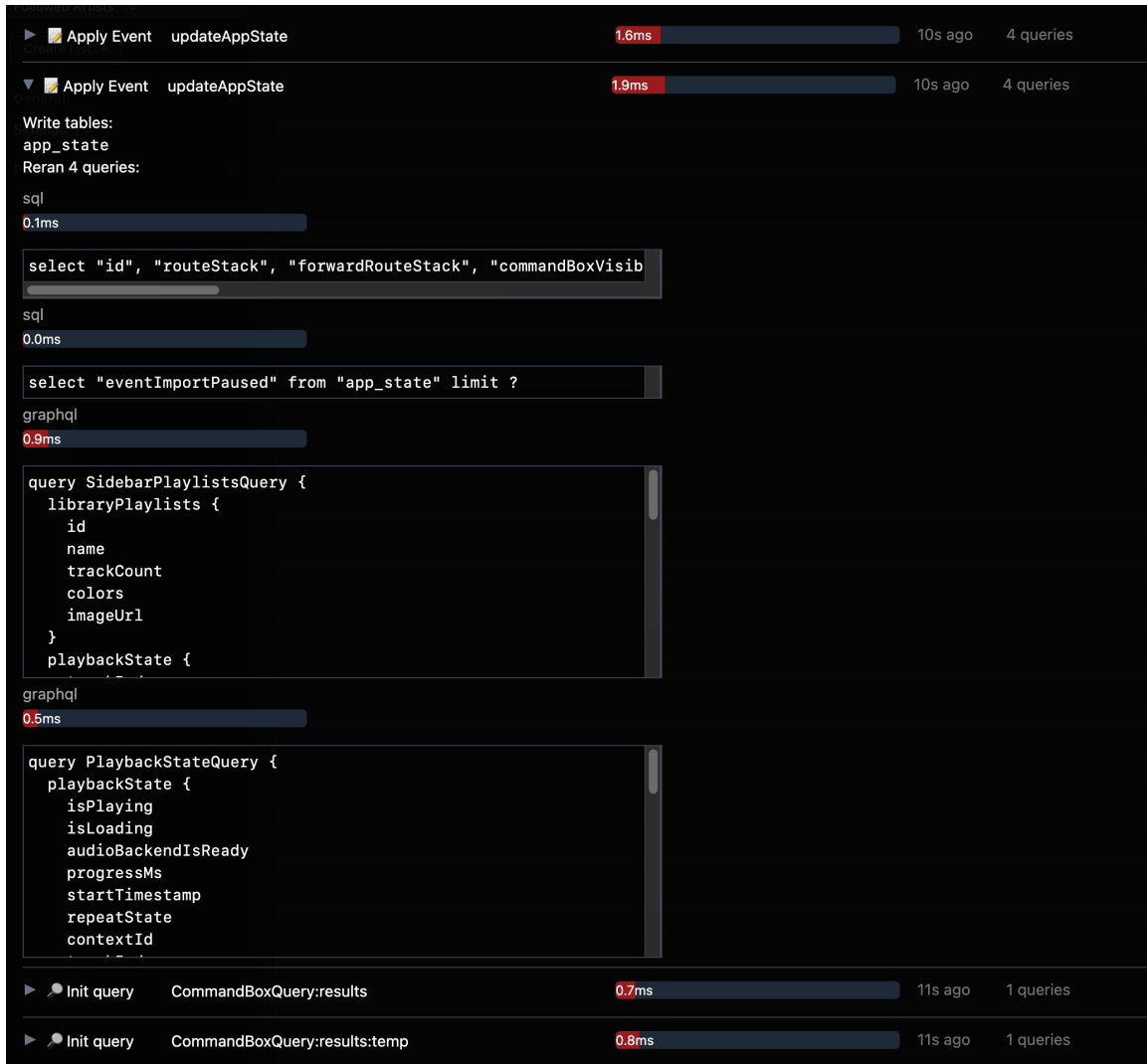


Figure 5-8: A debugger that shows recent updates in the Riffle reactive graph

Reflections on Development Experience

The design of Riffle has co-evolved with this application over the course of about a year and a half. The lead developer of the application, Johannes Schickling, started out as an external partner; he ended up making substantial contributions to Riffle itself and became a contributor to the core project. Most development was done by Johannes, with some part-time help from the other Riffle collaborators.

The goal of the case study was to test the ideas of Riffle in the context of a real application with substantial complexity. Many of the problems we encountered in the creation of Overtone resulted in changes to Riffle; as a result, this case study should be seen as a formative process that guided the design of Riffle, not a one-time evaluation of a pre-existing system. In this section we present some of the main lessons we have learned from building Overtone using Riffle. Many of these lessons have already fed back into the design of Riffle; others suggest unresolved limitations for future work.

Synchronous architecture. In our original prototype of Riffle we used a common architecture for running databases in the browser: our database ran in a Web Worker thread, communicating asynchronously with the UI thread. However, when building Overtone using this prototype we discovered two main problems.

First, performance was inadequate in some cases. In a complex app like Overtone, Riffle needs to re-run over 10 SQL queries in response to a single user interaction. In some browsers each query was incurring up to ~ 2 ms of overhead due to inter-process communication, often dwarfing the time needed to execute the query itself, and adding up to substantial delays across multiple queries. We saw very noticeable lag in interactions like text entry and button hover states, which pass through the database in Riffle, and require low latency to feel fluid.

A second problem was that asynchronous data fetching complicated our mental model of the application. As one example: when a new component would appear in the UI, it would not have data available on its first render (because React requires render functions to be synchronous); we would need to explicitly handle this case, e.g. with an empty state or loading spinner.

In response to these challenges we switched to the synchronous architecture described in this chapter. Eliminating the overhead of frequent inter-process communication in the browser solved many of the user-facing latency issues. It also made our UI code simpler by eliminating the need for intermediate loading states. (These observations led to our principle of synchronous transactional updates described in Section 5.4.2.)

The synchronous architecture introduces its own new challenges. Scheduling reactive query updates on the UI thread incurs the risk of blocking the UI and causing lag; we avoid these problems in Overtone by making sure that the queries in the application are fast enough over typical data sizes. Also, we now load all data into memory, which makes memory a limiting factor in data sizes. We elaborate more on these problems in Section 5.7.3.

Incremental view maintenance. Our initial prototype used SQLite as an underlying persistent database. SQLite is a mature, optimized database, and we found that its performance met our needs for most of our queries. However, queries over

large playlists that required many joins would sometimes take over 10 milliseconds, resulting in frame drops and noticeable lag. We attempted to solve these problems by introducing a materialized view that would precompute the joins, but this did not resolve the problem because the materialized view itself needed to be refreshed with an expensive query when the underlying data changed.

These challenges motivated us to switch to SKDB as a backing database, since it supports incremental view maintenance which can maintain a materialized view without recomputing from scratch. We have found that efficient view maintenance is important for building a database-backed application where queries with many joins over large data must react with low latency to updates on the underlying data.

Another approach we have tried is to *manually* incrementalize updates on expensive views by specifying explicit logic for how the view should update in response to writes. This approach has several disadvantages: it loses the advantage of declaratively specifying the view in SQL, and requires careful testing to ensure the update logic is correct. However, it does provide a pragmatic technique to improve performance even in a database like SQLite which lacks built-in incremental view maintenance.

Limits of SQL. Initially we were enthusiastic about SQL as a relational query language, but while writing queries to support Overtone we quickly ran into several limitations. SQL cannot produce tree-shaped results, has a verbose syntax for traversing associations (e.g., a two-step join across a many-to-many association), and lacks good support for composing fragments of queries into a larger query. We also found that a lack of automatic type inference for the resulting types of SQL queries made it difficult to use SQL queries in a TypeScript environment.

In response to these challenges, we added GraphQL⁴² as a supported reactive query type in Riffle. We use a standard GraphQL setup: we define a *schema* which defines core data types in the application, and implement a *resolver* which interprets GraphQL queries at runtime by executing SQL queries. Most components in Overtone do not use raw SQL queries; instead they specify a GraphQL query for their data requirements.

We have found that GraphQL is a convenient existing language (and tooling ecosystem) for papering over some of SQL's limitations. It can produce tree-shaped results, it has a concise syntax for traversing associations and selecting fields, and it has an existing tooling ecosystem for inferring TypeScript types from queries. On the other hand, it adds significant complexity: there are now multiple data schemas and query languages in the application. In the future, replacing SQL with a new relational query language better designed for UI programming might be able to replace these two layers with a single language.

UI state. Persisting UI state by default turned out to be even more beneficial than we'd imagined. Users were happy that their selected playlist and track were preserved, for example. But we also encountered some bad cases. We initially included the playing state of a track in the persisted UI state, but this meant that opening the app could cause a track to start playing spontaneously, which might be annoying, and

⁴²<https://graphql.org/>

even dangerous, if the user hasn't adjusted their volume setting. We have handled these kinds of cases by simply resetting the state to an initial value when the app boots.

In general, we found that we could simply take UI state that would have been managed in React, and put it into Riffle instead. However, there are some parts of a UI's state which are typically managed by the browser DOM and not React, such as the scroll position within a component, and these cases required extra work. In order to store playlist scroll position in Overtone, we needed to implement additional code to bidirectionally synchronize scroll position with the database, by listening to scroll events as well as updating the scroll position in the browser to match the database.

Managing UI state in the database also exposed some of the limitations of the relational model. For example, the Overtone routing navigation stack is represented as a list of values, with each value being some branch of a tagged union representing a type of page and type-specific parameters for that page. This structure proved clumsy to represent in a relational database (because of the limited support for tagged unions) so we have currently resorted to storing the routing stack as a string-serialized value within a single cell in the relational database.

Debug views. We have found it convenient during Overtone's development to have the entire state accessible in a single database. We have been able to open SQL databases, share them with each other as files, and have even built features into the app to hydrate state from a saved previous state. We found that it was often easy to reproduce bugs because the entire state of the system could be shared as a file, and the UI depends entirely on the state of the database. We also built a debugger view showing recent refreshes (what caused them, and which queries refreshed) which has helped us fix numerous bugs at both the application level and within Riffle's implementation; having explicit dependency tracking made it easy to provide these debug views because the system always knows the provenance for the causes of any update.

Separate reactivity for data and view. Riffle has its own reactive graph with dependency tracking at the data layer, and also integrates with React.js which has its own reactivity model for the view layer. We found that having a separate reactivity model for the data layer helped with efficiency because we could propagate data updates independently of the UI tree. This particularly helps when two child components far away in a UI tree need to subscribe to the same data. For example, in Overtone, changing the selection in a list of tracks updates the contents of a sidebar which shows details for the selected track. In normal React usage, the state for the selected track would be "hoisted" to a parent component containing both the list view and the sidebar, and the entire UI tree below that parent would need to re-render in response to changes to selection state. In contrast, in Riffle, the selection can be stored in the database, and the list view and sidebar can each independently subscribe to shared state from the database without needing to pass that state through the UI tree.

Occasionally, we have gotten confused by having multiple layers of reactivity, since there is caching going on at multiple levels of the system, including within the Riffle reactive graph and within React. As future work, subsuming DOM output into Riffle's

reactivity model might be able to simplify the architecture by removing React as a separate layer.

Loading data on startup. At first, we designed Overtone to import all metadata for a user’s music collection upfront, when the app was started. But this wasn’t ideal: the import process could take several minutes, during which the user could interact only with the tracks loaded so far. If you had a particular song in mind, you wouldn’t be able to navigate to it and play it until the import had finished. Because the import latency was constrained by the streaming service’s rate limiting, and by fundamental performance limitations of the database (the throughput of inserts), there was no easy way to shorten this process. This problem reflects a common limitation of local-first software that employs an eager synchronization workflow.

Our solution was to prioritize imports based on user input. First, the application shows a list of playlists without loading all their tracks; if the user clicks on a playlist, the tracks for that playlist are immediately prioritized for import. The resulting experience is a kind of hybrid between a traditional web application and standard local-first application. Once all data has been synchronized, interactions are synchronous since data is available locally. But while the data is loading, the user can still navigate to pages that asynchronously load data, which resembles the experience of a standard web app.

Performance analysis

In this section we present a brief performance analysis that shows how view maintenance scales in Overtone. Overtone stores tracks, albums, and artists in separate normalized tables. Tracks belong to a single album and store a foreign key directly; there is also a join table `tracks_artists` supporting a many-to-many relationship between tracks and artists. In the user interface, the information for all of these tables must be joined together. We compute a materialized view which stores the results of this join, so that further downstream queries do not need to compute the joins in response to user interactions; this view is one of the primary queries in our application.

Here is a simplified version of the SQL for the view:

```
select * from tracks, albums,
  tracks_artists, artists
where tracks.albumId = albums.id
and tracks_artists.trackId = tracks.id
and tracks_artists.artistId = artists.id;
```

Whenever the track, artist, or album metadata changes (e.g., while importing a metadata change from Spotify), the materialized view must also be updated. In SQLite, this requires re-running the entire join query and re-inserting the contents of the view from scratch. In SKDB, the view can be maintained incrementally, only updating the changed data.

Table 1 shows the time in milliseconds taken for each of these two systems to update the materialized view in response to inserting a single new track (along with

Size of table	100	1000	10000	50000
SKDB	1.1	1.6	1.6	2.4
SQLite	1.7	14	130	778

Table 5.1: Time taken to update materialized view in response to inserting 1 new track (ms)

a corresponding album and artist) into the base tables. Each column represents a different number of pre-existing tracks in the database before the insert. Both databases were run using WASM running in Google Chrome on a 2021 MacBook Pro M1, on the same synthetic dataset.

Using SQLite, the time taken to recompute the view scales linearly with the number of tracks in the view. While the recomputation times are reasonable for small collections, with 50,000 tracks (not an uncommonly large music collection), the recomputation takes over 700 milliseconds, an unacceptably long time to block the UI in response to a small change. In contrast, in SKDB, the time taken stays relatively constant: the view can be updated in under 3 milliseconds even for a large music collection. This benchmark demonstrates the value of using a database that supports incremental view maintenance when building responsive applications that store large amounts of data.

5.7.2 Heuristic evaluation

Olsen [65] introduces a set of criteria for evaluating a complex UI system. Several of the criteria especially pertain to Riffle.

Importance and generality. Olsen notes that the importance and generality of the problem being solved is a major factor in assessing the value of the solution. The problem being solved by Riffle—managing reactive state in applications—is important and general, as demonstrated by the large number of research and commercial systems aiming to solve it. Our solution is general enough to apply to any application which can be architected in a local-first way and have its state managed relationally; this is a broad class of applications which is not limited to any particular domain. We discuss the limits of the appropriate applications for Riffle further in Section 5.7.3.

Scale. It would be much easier to build a version of Riffle that only works for small toy applications; most of our effort has gone into making these simple abstractions scale up to a real context. The music application case study demonstrates that Riffle can scale up to meet the performance and expressiveness needs of a real-world application. Many aspects of Riffle’s design, including the performance architecture, the addition of GraphQL as a query language, and the design of our APIs, were specifically informed by the needs of this large application. We believe the observations from Overtone should generalize to any complex application with relational data and high performance requirements, such as an email client or budgeting app.

Expressive leverage. Olsen defines expressive leverage as “where a designer can accomplish more by expressing less.” Riffle achieves expressive leverage by enabling the developer to declaratively specify reactive relational queries.

Using typical web technologies, the UI developer for an application like Overtone would need to write API calls to query information from a backend (e.g., polling for new changes), low-level JavaScript code for data transformations like relational joins, and would need to explicitly write code for persisting UI state. In Riffle, the developer only needs to declaratively specify relational state and queries in order to meet all of these needs. Query results are automatically updated when the database changes, relational joins are efficiently executed within the database, and UI state is automatically persisted by the framework.

Synchronous transactional updates are an example of where Riffle can help developers achieve a better user experience with less code. Usually, web UI developers must write code to handle asynchronous data loading and intermediate states such as loading spinners. In a Riffle app like Overtone, this code is eliminated, and the UI is more responsive to user interactions.

These benefits also relate to Olsen’s notion of *expressive match*: “an estimate of how close the means for expressing design choices are to the problem being solved.” Reactive relational queries offer a high-level mental model that allows developers to think in terms of questions like “what data depends on what other data?” and “what should the shape of these query results be?”, rather than concerning themselves with lower-level implementation details of correctly propagating updates and efficiently implementing joins.

On the other hand, Riffle does require developers to specify more explicit information than some competing approaches. The most prominent example is the need to specify a relational schema, which requires more up-front work than a schemaless document data model. We believe this is a worthwhile tradeoff for complex applications, since the schema makes it easier to enforce data constraints and model normalized data.

5.7.3 Limitations

In this section we list several key limitations of our general architecture and our current implementation.

Local-first architecture. Riffle relies on a local-first architecture, which imposes some restrictions on the kinds of applications that are a good fit for the design.

First, we synchronize more data to the local device than is a typical web application, meaning the client device must have enough storage space for the synchronized data. The first-time experience may also require waiting for more data to load.⁴³ As a result, the local-first architecture is a better fit for applications with repeated frequent use (e.g., a music application or a productivity tool), as opposed to applications which are intended for less frequent use and may not justify the initial data load time (e.g., an e-commerce website).

Allowing users to concurrently make edits offline and without a central server makes it harder to preserve certain data invariants. Distributed data structures like

⁴³In Overtone we have somewhat mitigated this load time limitation by adopting a synchronization strategy which prioritizes events to scrape from cloud music providers based on user interactions; this is a kind of hybrid between naive eager synchronization and lazy data fetching.

CRDTs [70, 68, 82] offer techniques for preserving low-level invariants like the order of elements in a sequence, but some constraints like uniqueness or foreign key constraints are difficult to preserve while allowing offline editing. For example, a room booking system might not be a good fit for Riffle because users might concurrently book the same room. (These kinds of stronger data invariants could possibly be added to the model by requiring certain operations within an application to be validated by a central server.)

Local-first applications also limit the kinds of access control that are easily achievable. With a single user, all of the user’s data may be synchronized, but if data is shared between multiple users, a more sophisticated approach is needed. If the data can be easily segmented into large coarse-grained units (e.g., “projects” or “libraries”), these units may be used to determine what is synchronized to a given user’s device; however, an application like a social media network may not offer such convenient boundaries between discrete datasets. Access control in local-first software is an active area of research [79].

Schema evolution is a challenge in any local-first system where state is spread across multiple devices—for example, it is difficult to rename a column if the rename cannot be performed atomically across all clients. In Riffle, the problem is more prevalent than usual because we store UI state in a persistent schema in addition to domain state. We have not yet developed a principled solution to this problem; in general we have reset UI state when changing the schema for that data. Some approaches to managing schema divergence in decentralized systems have been proposed in the context of a document-based data model [49] but it is unclear how this approach would extend to a relational model.

Relational model. Riffle relies heavily on the relational model, which has several limitations in the context of building user interfaces. Representing sequences and nested hierarchies is less straightforward in the relational model than using the data structures like lists and objects available in programming languages, or in a document-based database. This limitation appears more acutely in Riffle than in many uses of relational databases because we encourage moving UI state (which often includes structures like ordered lists) into the database.

Another limitation is that most relational query languages such as SQL produce tabular relational results, but user interfaces frequently show tree-shaped results with nesting. This is technically a limitation of existing relational query languages and not a fundamental limitation of the relational model since it is possible to query relational data and produce nested trees at the final projection step; Partiql⁴⁴ is one example of a relational query language that can produce nested output. In Riffle we use GraphQL and JavaScript to transform relational query results into tree-shaped data for the UI.

Synchronous execution model. A major part of Riffle’s simplicity comes from turning interactions that would typically require asynchronous data fetching into synchronous operations operating on local data. However, sometimes asynchrony is unavoidable. One example is making network requests to search a large dataset that

⁴⁴<https://partiql.org/>

cannot be synchronized locally; another example is handling particularly slow SQL queries that can't be made fast enough to execute synchronously within the UI.

In Riffle, application developers must handle these cases manually by writing imperative code performs asynchronous operations which write to the Riffle database. For example, a search over a cloud music service might trigger asynchronous network requests and write the results from the network responses back into the database. This is a practical solution but it loses the simplicity of Riffle's declarative model. A possible direction for future work is suggested by DIEL [81], which offers a declarative relational model that spans across the network boundary and incorporates asynchronous requests.

Performance limits. As shown in the Overtone case study, the Riffle architecture is generally capable of supporting responsive interactions in a real application. However, the currently implemented system does have performance limits.

For the current feature set of Overtone, most interactions in the application are responsive within the 100ms “instant” threshold for music collections in the tens of thousands of tracks, but larger collections can cause some interactions to exceed that threshold. We believe that continued performance optimization, both within SKDB and at the interaction point between Riffle and SKDB, is likely to continue to provide further speedups. Crucially, incremental maintenance provides extra options for addressing bottlenecks in a way that is impossible with a traditional non-reactive database, since the low-level database can be optimized further to efficiently handle small changes.

So far, we have done our testing on fast modern devices (e.g., a MacBook Pro with an M1 processor). More testing on slower client devices may reveal further performance limitations, since the Riffle architecture depends on the performance of the client. Another limitation is that we do not currently support datasets that are too large to fit in memory in the browser's WASM heap (limited to 1GB), and we have no automated mechanism for asynchronously executing slow queries off the main thread.

In general, we see the current implementation of Riffle as an *existence proof* that a reactive relational database can be made performant enough to support synchronous transactional updates and UI state in the database, but improving performance further is important to make the architecture viable in more situations.

5.8 Future work

Data substrate for interoperability. Prior projects have explored *shared data substrates* that enable interoperability between tools. For example, Webstrates [40] stores information in the browser DOM and synchronizes it over the Web, SOLID [52] stores information in personal data pods controlled by the user and accessed via Web APIs, Plan 9 [67] uses the desktop filesystem to share data among tools, and Dynamicland⁴⁵ offers a global reactive database tied to a physical space

⁴⁵<https://dynamicland.org/>

In a similar spirit, we envision using Riffle to build such a data substrate where multiple applications and tools could all act on a user’s personal data. For example, a user’s Overtone music collection stored in a Riffle database could also be accessed by other special-purpose tools, e.g. a tool that analyzes tracks and adds additional specialized metadata. Riffle has several characteristics that could make it an attractive choice for this kind of usage:

- **Flexible access patterns.** The relational data model is access pattern agnostic, and so could support different tools querying data in ways that are different from the original application. At the same time, relational data constraints could preserve useful integrity properties.
- **Fast reactivity.** Fast reactive updates could enable live state synchronization across different tools in an operating system. A single application could in fact be composed of multiple smaller tools exclusively coordinating through the shared Riffle data layer in realtime.
- **UI state in the substrate.** Storing all UI state in the database provides a unique opportunity to provide scripting and automation capabilities through the shared data substrate. Because all actions in the UI must flow through database writes, any tool connected to the shared reactive database can programmatically trigger the same actions as original GUI itself. For example, an external script could play/pause tracks or change the selected playlist in Overtone.

As one example of this potential, we have run some small experiments connecting a generic SQL editor GUI to a running application, and editing the state (both domain state and UI state) in the generic editor. Future work would involve creating multiple rich applications and having them interoperate through Riffle as a conduit.

Live programming. The debugger view we have built (Section 5.5) is a small example of the kinds of live programming interfaces that could visualize the structured dataflow graph created by Riffle. Future work could explore making this debugger more powerful. Some directions could include allowing for actually dynamically editing running queries on the fly within the debugger, as well as richer views of the dependency structure, such as a dependency graph with edges between queries that depend on one another. We have found it useful to draw these kinds of diagrams manually ourselves, and Riffle provides exactly the underlying structured dataflow needed to support such visualizations.

5.9 Conclusion

In this chapter, we have presented Riffle, a new architecture for user interfaces that couples the UI with a fast, reactive client-side relational database. *Reactive relational queries* provide an ergonomic declarative model for developers to define data transformation logic that can be efficiently executed. *Synchronous transactional updates* enable the UI to efficiently step forward in consistent steps.

In combination, these properties enable both a simpler way for developers to write applications, and higher quality experiences for end-users. Riffle shows how ideas from

reactive databases and spreadsheets can be applied to simplify the creation of complex software by skilled developers.

Chapter 6

Conclusion

In this thesis, we have introduced three novel techniques for applying the power of spreadsheets and reactive databases to support developers and end users in building and customizing personal software.

- **Wildcard** enables end-users to customize existing web applications through a reactive table interface.
- **Potluck** supports users in turning text notes into personal interactive tools through a reactive table view.
- **Riffle** supports developers in building sophisticated user interfaces based on a reactive relational data model, with a live table debugger view.

6.1 Key Ideas

While these techniques apply to a range of scenarios—from making a small customization to an existing website to building a complex UI application from scratch—they all demonstrate the value of a few key ideas.

Making state visible in reactive table views. We have shown that software development can be simplified by providing a view of underlying state in a reactive table—making visible information that would typically be opaque, and allowing the user to directly interact with it. As demonstrated by Wildcard and Potluck, this approach can even be applied when the information is being extracted from a source that is not in a relational format, like a rendered UI or a text note.

Fast reactivity. More broadly, we have shown a few ways that declarative abstractions can simplify software development by allowing users to interact with higher-level models of a system. One key abstraction is fast reactivity, which maintains invariants across different state representations and avoids the need for users to manually propagate change through a system. Wildcard and Potluck demonstrate how existing data representations can be reactively synchronized with a tabular representation—users can think in terms of higher-level invariants (such as patterns to extract from a text document) rather than the lower-level logic of scheduling these updates appropriately.

Semantic wrappers. Another useful abstraction is semantic wrappers, which extract structured data from unstructured sources in a reactive loop. These wrappers provide a modular boundary between extracting data and working with that data. This modularity can enable users of different technical ability to tackle the two parts of the task, or can allow the same user to focus separately on data extraction and working with the data. A well-designed semantic wrapper makes it feel seamless to work in a structured way with less structured data.

Reactive relational model. Finally, Riffle demonstrates the power of combining reactive programming with relational queries, which are themselves a powerful abstraction that enables users to think in terms of high-level query logic rather than lower-level execution details. Defining a user interface as a graph of relational queries with reactive dependencies not only gives the developer a high-level model for reasoning about the behavior of the UI, but also allows the underlying reactive and relational execution engines to optimize performance.

Overall, these systems build on the long history of work on spreadsheets and reactive database systems, and demonstrate new ways of applying these ideas to the creation and customization of many different kinds of software.

6.2 Future Work

6.2.1 Towards data-centric interoperability

The metaphors we use in our computing environments define the kinds of composition that are possible to achieve. For example, modern web-based applications tend to tightly couple a specific application interface together with data storage and management. Each application stores its own private data, making it difficult to collaborate together on that data using disparate tools, or to collectively manage data together across multiple tools. This architecture forces teams to make compromises over their tooling choices, and can force individuals to learn new tools to work with a specific team [64].

We can find other points in the design space of possibilities by looking at historical precedent. As discussed in the introduction to this thesis, the desktop filesystem is an example of a data storage system that provides a foundation of shared state and coordination across multiple applications. The same file can be edited by collaborators in their respective preferred editors, or passed between tools that each handle different parts of a pipelined workflow. Files can be organized together into folders regardless of their formats or the applications used to open them. Furthermore, filesystem-based version control systems like git make it possible to collaborate over this data; collaborators using git enjoy the privilege of editing the same files using different editors, such as different code editors.

Various systems have explored building on these desirable properties of the desktop filesystem in new ways. Webstrates [40] and Pushpin [77] have proposed shared repositories of document-structured data that can be reactively edited live by multiple tools. Another direction is to introduce relational structure into the shared data—

File	Window	Attributes				
	Size	Contact name	Work phone	E-mail	Company	City
👤	0 B	Clara Oswald	08772-776380	clara@suffle.fr		Whelton
👤	0 B	Edward Richman	07761-872083	edward.richman@crispc.org	Crisp Consulting	Fnordshire
👤	0 B	John Nox	07761-872021	john.nox@crispc.org	Crisp Consulting	Fnordshire
👤	0 B	Darren Garbley	0825-625518	garbled@dromiak.be	Rarslang Inc.	Bramsdén
👤	0 B	Wilbur Barlington	0825-625501	wilbington@dromiak.be	Rarslang Inc.	Bramsdén

Figure 6-1: Haiku OS stores a list of contacts using structured attributes on files, enabling them to be managed through a generic database view

as demonstrated by the filesystem in Haiku OS, which supports storing arbitrary structured attributes on files and then viewing and editing those attributes in a generic database editor. In Haiku, a contacts editor can be represented simply as a list of empty files with attributes such as names, emails, etc. (Figure 6-1). Microsoft also made a failed attempt called WinFS to similarly reorient the Windows filesystem around a relational database which would have more understanding of the schema of the data inside¹.

The work in this thesis contributes some ideas for enabling data-centric interoperability.

First, our work suggests interaction techniques for exposing a relational dataset to the user in the context of a running GUI. Wildcard shows how a reactive database debugger view can be shown alongside a rich GUI application as a means of understanding and modifying its behavior, e.g. by sorting data differently or by adding new computational properties. These interactions would layer naturally on top of a relational filesystem backing the state of a GUI app.

Second, Potluck demonstrates techniques for integrating less structured data into an interoperable datastore. Every system needs to deal with some combination of highly structured and less structured data; the naive approach would be to treat these data in isolation, forgoing any opportunity to leverage the semantic meaning contained within less structured data like a text note. Potluck suggests a different approach: letting users define methods for extracting meaningful structure from text data, and then using that structured data to support computations which can be integrated back into the text. If the data from a Potluck note were extracted into a shared OS-level filesystem, other applications and interactive views could be built on top of that structured data.

Finally, Riffle proposes a new UI architecture that would fit very naturally on top of a relational database filesystem. If the user’s system had a reactive relational “filesystem” responsible for managing all data, application developers could delegate state management (query planning, reactivity, persistence, synchronization) to this database, and would only need to be responsible for implementing UI components that query the database and visualize its contents. Furthermore, storing all data including UI state in this shared datastore would open up interesting opportunities

¹<https://en.wikipedia.org/wiki/WinFS>

for scripting across applications, since any tool could drive any part of the UI of another tool (even things like selection state or pressing buttons) through the shared data substrate.

There are some key challenges to achieving this kind of shared structured relational datastore across multiple applications. One challenge is managing diverging data schemas. Often applications that perform similar tasks in the same domain (drawing applications, todo lists, etc.) use different data representations that are specific to the needs of that application. Common standards are difficult to agree upon and often only cover the lowest common denominator of needed behavior; a better approach would be to somehow share data as much as possible between applications, while also allowing each application to manage its own unique format.

Cambria [49] suggests one direction for handling this problem in the context of document-formatted data: declaratively define *bidirectional edit lenses* that can translate edits on one data format into edits on the other, and vice versa. Similar techniques might be applicable to relational data, although the fact that relations can reference one another in flexible ways might make the problem more challenging in the relational setting.

Another challenge is managing boundaries between data for sharing and privacy. In a traditional filesystem or a document-based data model, there is a natural hierarchy to the system which helps establish boundaries for access control and sharing. In contrast, if all data is stored in a relational database, there is no obvious canonical hierarchy, and different approaches such as row-level access controls are needed.

6.2.2 The role of AI

As of this writing in summer 2023, there has been a whirlwind of recent progress in large language models (LLMs). One of the most intriguing advances is the ability of models to write code: the LLM-based GitHub Copilot autocomplete product has been used by one million people², and GPT-4 has proven capable of generating relatively sophisticated programs such as a 3D game [17]. What might this progress entail for tools that aim to simplify the creation and editing of software, especially by less technically sophisticated users? To take a pessimistic stance: it possible that decades of research into better tools has been superseded by AI techniques and chatbots?

While it is too early to tell, I believe that LLM-based code generation will, at least in the short term, prove *complementary* to the techniques introduced in this thesis. Here we briefly describe two promising directions for integrating LLMs with end-user programming tools.

Writing computations

One useful role for AI could be to help users fill in small bits of code within spreadsheet-style programming environments, such as formula computations in Wildcard or Potluck. In this setup, users would preserve live visibility into data; being able to see live intermediate results from a computation could help people understand the behavior

²<https://github.blog/2023-02-14-github-copilot-for-business-is-now-available/>

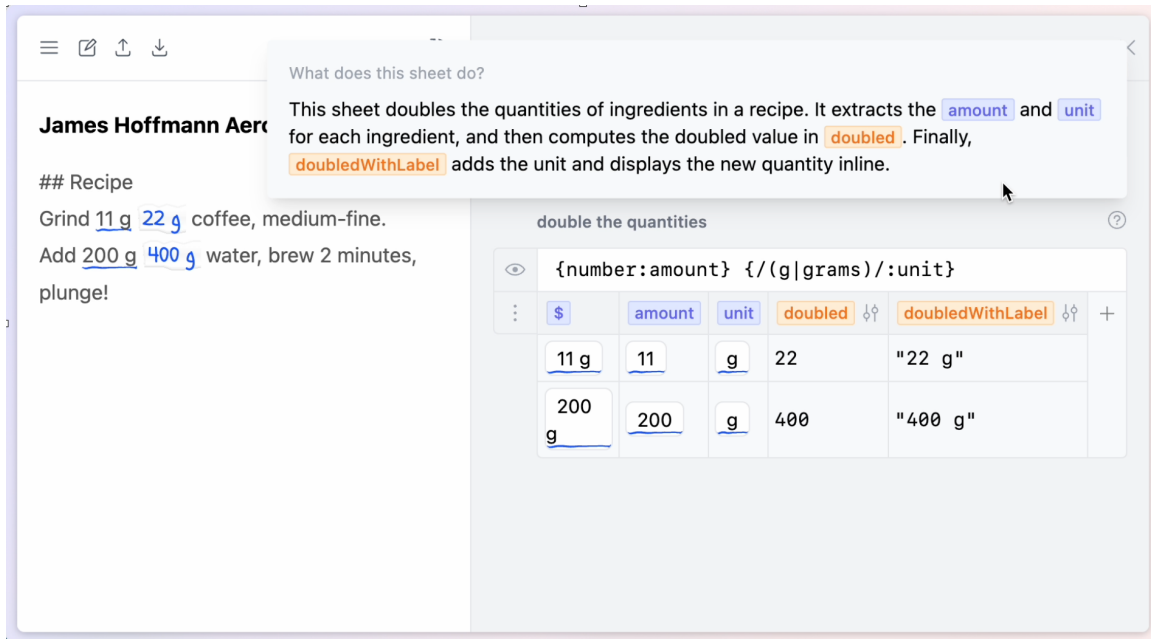


Figure 6-2: A spreadsheet table generated by GPT-3 after the user typed “double the quantities”

of automatically generated computations. Direct manipulation interactions would remain possible: editing data values or sorting by certain columns is more easily achieved by direct manipulation than through a natural language chat interface.

To illustrate what we mean, here is one specific extension to Potluck that we have prototyped using GPT-3 [16], shown in Figure 6-2. Instead of writing a search pattern such as `{number} g` into the search box, the user can input a natural language goal like “double the quantities”. An LLM interprets this goal and automatically outputs a table which contains 1) search patterns expressed in the Potluck search language which extract relevant data from the text, and 2) computed columns which calculate derived results over the extracted data to achieve the stated goal.

With this interaction model, the user retains visibility and control. The search patterns and computations generated by the AI are inserted into a live programming environment where the user can see how the computation works, and modify the generated code. They can iterate directly on the code for simple changes (e.g.: changing a 2 to a 3 to scale up the recipe further), or can ask the LLM to make further adjustments to the code on their behalf. The environment also allows the user to click a button and see an LLM-generated natural language description of how the computation works.

In this arrangement, the spreadsheet programming environment broadly, as well as the formula language specifically, serve as a *shared representation* which the user and an AI bot can fluidly collaborate on together. Heer has described [29] how such shared representations can help balance the need for user agency with useful automation of specific tasks.

Expressing logic in a declarative style can also help support AI code generation. In

Potluck, reactivity is deeply built in to the foundation of the system; the AI needs only generate search patterns and JavaScript computations, and the system automatically takes care of executing these reactively as the user edits a text document. The relational queries in Riffle could serve a similar role: the AI could generate SQL queries expressing the logic of a transformation, which can then be efficiently executed by a powerful query optimizer. In all of these cases, a powerful runtime helps either an AI or a human user express the core logic of a task without worrying about low-level implementation details.

Writing semantic wrappers

Another powerful role for LLMs could be in authoring supporting the extraction of structured data from unstructured sources. There are at least two distinct ways LLMs could be used to support data extraction, each with different tradeoffs: 1) *LLM code generation*, using LLMs to author extraction code, and 2) *LLM scraping*, using LLMs to directly perform extraction.

LLM code generation would use LLMs to author data scraping and extraction code, such as Wildcard site adapters and Potluck searches; we showed one example of this above when an LLM authored a Potluck search. Once the code is authored, it executes deterministically. Having the LLM author semantic wrappers could be a useful way to balance automation and agency: an LLM could author code to extract a clean structured dataset from an unstructured source, and then the user could specify their own behaviors and computations over that cleaned data. This split of responsibilities resembles the split we originally envisioned for Wildcard, where skilled developers would play the role of creating site adapters for popular websites. Instead of relying on skilled human developers, one could imagine LLMs automatically authoring site adapters, or perhaps automatically verifying and maintaining adapters which were initially authored by humans.

A second approach is *LLM scraping*, where the LLM itself performs data extraction. For example, in Wildcard, upon loading a page, an LLM could be prompted with a phrases like “Produce a JSON table of Airbnb listings based on this HTML”. Early experiments have suggested that LLMs are quite capable at these kinds of tasks [42]. This approach would build on earlier efforts in end-user programming research to enable “sloppy programming” with fuzzy natural language syntax for web automations [50].

The two approaches have interesting tradeoffs. Code generation produces a process which runs deterministically and predictably—given the same input it will always produce the same output. As a result, the user can learn the rules of the system and create data which follows the rules, e.g. developing a personal micro-syntax in Potluck which will always trigger a given search in the right way. The user also has some chance (given the right tools and languages) of understanding what the scraping code does. Finally, scraping code can be executed efficiently; this supports interactions like re-scraping data reactively upon every keystroke.

On the other hand, some kinds of scraping are very difficult to achieve with deterministic code. Tasks like extracting food names from a text recipe, or extracting

names of people from a meeting notes document, stretch the limits of traditional “scraping”; these tasks are more accurately described as natural language processing (NLP) tasks for extracting semantic meaning from unstructured inputs. LLMs provide a powerful new toolkit for performing these kinds of open-ended flexible scraping tasks.

In summary, LLM-powered code generation and scraping seems likely to contribute significant advances to the state of developer tooling and end-user programming interfaces. But ideas such as declarative representations of logic and semantic wrappers will likely remain relevant as a tool for mediating collaboration between LLM agents and human users.

6.3 Conclusion

Software ought to be the ultimate medium for free expression. We are not bound by the laws of physics; nearly any computational tool is possible to create, at least in theory.

In practice, the structures we have today for creating software too often get in the way. For skilled developers, they introduce mountains of incidental complexity, making it far harder than it should to build great user experiences. And for end users without much programming expertise, using a computer usually boils down to using prefabricated experiences created by developers, without much hope for modification.

In this thesis we have shown how ideas from spreadsheet interfaces and relational databases can help simplify the creation and modification of software, both by developers and end users. Spreadsheet-style table interfaces allow for visibility into underlying data provide direct manipulation affordances for editing data and specifying computations. Reactivity simplifies the mental model of dataflow. Relational databases offer powerful abstractions for modeling and querying data. Taken together, these techniques offer new ways of thinking about building and editing software.

Bibliography

- [1] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. arXiv:1207.0137 [cs] <http://arxiv.org/abs/1207.0137>
- [2] 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](https://doi.org/10.1016/S0360-1315(99)00029-9)
- [3] 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>
- [4] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, Melbourne Victoria Australia, 1327–1342. <https://doi.org/10.1145/2723372.2737784>
- [5] 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>
- [6] Michel Beaudouin-Lafon. 2000. Instrumental Interaction: An Interaction Model for Designing Post-WIMP User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '00)*. ACM, New York, NY, USA, 446–453. <https://doi.org/10.1145/332040.332473>
- [7] Edward Benson. 2014. *Reducing Authoring Complexity on the Web with a Relational Layer for Web Content*. Thesis. Massachusetts Institute of Technology. <https://dspace.mit.edu/handle/1721.1/93056>
- [8] 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>

- [9] 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*. ACM, Honolulu Hawaii USA, 97–106. <https://doi.org/10.1145/2642918.2647387>
- [10] Tim Berners-Lee. 2019. One Small Step for the Web. . . . https://medium.com/@timberners_lee/one-small-step-for-the-web-87f92217d085
- [11] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Pub. Co, Reading, Mass.
- [12] Eric A. Bier and Ken Pier. 1991. Documents as User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '91)*. Association for Computing Machinery, New York, NY, USA, 443–444. <https://doi.org/10.1145/108844.108994>
- [13] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently Updating Materialized Views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data (SIGMOD '86)*. Association for Computing Machinery, New York, NY, USA, 61–71. <https://doi.org/10.1145/16894.16861>
- [14] 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>
- [15] Alan Borning. 1981. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems* 3, 4 (Oct. 1981), 353–387. <https://doi.org/10.1145/357146.357147>
- [16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, 1877–1901.

- [17] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early Experiments with GPT-4. arXiv:2303.12712 [cs] <http://arxiv.org/abs/2303.12712>
- [18] 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>
- [19] 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>
- [20] Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. Association for Computing Machinery, New York, NY, USA, 153–165. <https://doi.org/10.1145/2676726.2677004>
- [21] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387. <https://doi.org/10.1145/362384.362685>
- [22] 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, 1–1–21. <https://doi.org/10.1145/1238844.1238845>
- [23] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Vol. 4709. Springer Berlin Heidelberg, Berlin, Heidelberg, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12
- [24] 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>

- [25] C. A. Ellis and S. J. Gibbs. 1989. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data - SIGMOD '89*. ACM Press, Portland, Oregon, United States, 399–407. <https://doi.org/10.1145/67544.66963>
- [26] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: Dynamic, Partially-Stateful Data-Flow for High-Performance Web Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 213–231. <https://www.usenix.org/conference/osdi18/presentation/gjengset>
- [27] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Austin Texas USA, 317–330. <https://doi.org/10.1145/1926385.1926423>
- [28] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. *ACM SIGMOD Record* 22, 2 (June 1993), 157–166. <https://doi.org/10.1145/170036.170066>
- [29] Jeffrey Heer. 2019. Agency plus Automation: Designing Artificial Intelligence into Interactive Systems. *Proceedings of the National Academy of Sciences* 116, 6 (Feb. 2019), 1844–1850. <https://doi.org/10.1073/pnas.1807184115>
- [30] 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>
- [31] Tristan Hume. 2020. Fragile Narrow Laggy Asynchronous Mismatched Pipes Kill Productivity. <https://thume.ca/2020/05/17/pipes-kill-productivity/>
- [32] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1985. Direct Manipulation Interfaces. (1985), 28.
- [33] David F. Huynh, David R. Karger, and Robert C. Miller. 2007. Exhibit: Lightweight Structured Data Publishing. In *Proceedings of the 16th International Conference on World Wide Web - WWW '07*. ACM Press, Banff, Alberta, Canada, 737. <https://doi.org/10.1145/1242572.1242672>
- [34] 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>

- [35] Daniel Ingalls. 2020. The Evolution of Smalltalk: From Smalltalk-72 through Squeak. *Proceedings of the ACM on Programming Languages* 4, HOPL (June 2020), 1–101. <https://doi.org/10.1145/3386335>
- [36] Alan Kay. 1984. Computer Software. *Scientific American* 251, 3 (1984), 52. https://www.academia.edu/1533030/Computer_Software
- [37] Alan Kay. 1984. Opening the Hood of a Word Processor. (1984). <http://worrydream.com/refs/Kay%20-%20Opening%20the%20Hood%20of%20a%20Word%20Processor.pdf>
- [38] Martin Kleppmann and Alastair Beresford. 2018. Automerge: Real-time Data Sync between Edge Devices. <https://mobiuk.org/abstract/S4-P5-Kleppmann-Automerge.pdf>
- [39] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2019*. ACM Press, Athens, Greece, 154–178. <https://doi.org/10.1145/3359591.3359737>
- [40] Clemens N. Klokmoose, James R. Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology - UIST '15*. ACM Press, Daegu, Kyungpook, Republic of Korea, 280–290. <https://doi.org/10.1145/2807442.2807446>
- [41] Andrew J. Ko, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, Susan Wiedenbeck, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, and Henry Lieberman. 2011. The State of the Art in End-User Software Engineering. *Comput. Surveys* 43, 3 (April 2011), 1–44. <https://doi.org/10.1145/1922649.1922658>
- [42] Rebecca Krosnick and Steve Oney. 2023. Promises and Pitfalls of Using LLMs for Scraping Web UIs. *Computational UI Workshop at CHI 2023* (2023). http://www-personal.umich.edu/~rkros/papers/LLMs_webscraping_CHI2023_workshop.pdf
- [43] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6, 2 (June 1981), 213–226. <https://doi.org/10.1145/319566.319567>
- [44] Vu Le and Sumit Gulwani. 2014. FlashExtract: A Framework for Data Extraction by Examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Edinburgh United Kingdom, 542–553. <https://doi.org/10.1145/2594291.2594333>

- [45] 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, New York, NY, USA, 1719–1728. <https://doi.org/10.1145/1357054.1357323>
- [46] Geoffrey Litt, Daniel Jackson, Tyler Millis, and Jessica Quaye. 2020. End-User Software Customization by Direct Manipulation of Tabular Data. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, Virtual USA, 18–33. <https://doi.org/10.1145/3426428.3426914>
- [47] Geoffrey Litt, Nicholas Schiefer, Johannes Schickling, and Daniel Jackson. 2023. Riffle: Reactive Relational State for Local-First Applications. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. San Francisco, CA. <https://doi.org/10.1145/3586183.3606801>
- [48] Geoffrey Litt, Max Schoening, Paul Shen, and Paul Sonnentag. 2022. Potluck: Dynamic Documents as Personal Software. In *LIVE Workshop at SPLASH*. <https://www.inkandswitch.com/potluck/>
- [49] Geoffrey Litt, Peter van Hardenberg, and Orion Henry. 2021. Cambria: Schema Evolution in Distributed Systems with Edit Lenses. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data (Pa-PoC '21)*. Association for Computing Machinery, New York, NY, USA, 1–9. <https://doi.org/10.1145/3447865.3457963>
- [50] Greg Little, Robert C. Miller, Victoria H. Chou, Michael Bernstein, Tessa Lau, and Allen Cypher. 2010. Sloppy Programming. In *No Code Required*. Elsevier, 289–307. <https://doi.org/10.1016/B978-0-12-381541-5.00015-8>
- [51] Allan MacLean, Kathleen Carter, Lennart Löfstrand, 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>
- [52] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Abounaga, and Tim Berners-Lee. 2016. A Demonstration of the Solid Platform for Social Web Applications. In *Proceedings of the 25th International Conference Companion on World Wide Web - WWW '16 Companion*. ACM Press, Montréal, Québec, Canada, 223–226. <https://doi.org/10.1145/2872518.2890529>
- [53] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User

- Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology - UIST '15*. ACM Press, Daegu, Kyungpook, Republic of Korea, 291–301. <https://doi.org/10.1145/2807442.2807459>
- [54] 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>
- [55] Frank McSherry, D. Murray, R. Isaacs, and M. Isard. 2013. Differential Dataflow. In *Conference on Innovative Data Systems Research*. <https://www.semanticscholar.org/paper/Differential-Dataflow-McSherry-Murray/f5df61effe8047eb9ea1702cfcc268dbba678567>
- [56] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 1–20. <https://doi.org/10.1145/1640089.1640091>
- [57] Robert C. Miller and Brad A. Myers. 2002. LAPIS: Smart Editing with Text Structure. In *CHI '02 Extended Abstracts on Human Factors in Computing Systems*. ACM, Minneapolis Minnesota USA, 496–497. <https://doi.org/10.1145/506443.506447>
- [58] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, Farmington Pennsylvania, 439–455. <https://doi.org/10.1145/2517349.2522738>
- [59] Brad A. Myers. 1996. The Amulet User Interface Development Environment. In *Conference Companion on Human Factors in Computing Systems (CHI '96)*. Association for Computing Machinery, New York, NY, USA, 327. <https://doi.org/10.1145/257089.257351>
- [60] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. 1995. GARNET Comprehensive Support for Graphical, Highly Interactive User Interfaces. In *Readings in Human-Computer Interaction*. Elsevier, 357–371.
- [61] Bonnie A. Nardi and James R. Miller. 1990. An Ethnographic Study of Distributed Problem Solving in Spreadsheet Development. ACM Press, 197–208.

- [62] Bonnie A. Nardi, James R. Miller, and David J. Wright. 1998. Collaborative, Programmable Intelligent Agents. *Commun. ACM* 41, 3 (March 1998), 96–104. <https://doi.org/10.1145/272287.272331>
- [63] Jakob Nielsen. 1993. Response Times: The 3 Important Limits. <https://www.ngroup.com/articles/response-times-3-important-limits/>
- [64] Midas Nouwens and Clemens Nylandsted Klokmose. 2018. The Application and Its Consequences for Non-Standard Knowledge Work. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*. ACM Press, Montreal QC, Canada, 1–12. <https://doi.org/10.1145/3173574.3173973>
- [65] Dan R. Olsen. 2007. Evaluating User Interface Systems Research. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology - UIST '07*. ACM Press, Newport, Rhode Island, USA, 251. <https://doi.org/10.1145/1294211.1294256>
- [66] John K. Ousterhout. 2018. *A Philosophy of Software Design* (first edition ed.). Yaknyam Press, Palo Alto, CA.
- [67] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. 1995. Plan 9 from Bell Labs. *Computing systems* 8, 3 (1995), 221–254.
- [68] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *J. Parallel and Distrib. Comput.* 71, 3 (March 2011), 354–368. <https://doi.org/10.1016/j.jpdc.2010.12.006>
- [69] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 341–350. <https://doi.org/10.1109/TVCG.2016.2599030>
- [70] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A Comprehensive Study of Convergent and Commutative Replicated Data Types*. Report. Inria – Centre Paris-Rocquencourt ; INRIA. <https://hal.inria.fr/inria-00555588>
- [71] Frank M. Shipman and Catherine C. Marshall. 1999. Formality Considered Harmful: Experiences, Emerging Themes, and Directions on the Use of Formal Representations in Interactive Systems. *Computer Supported Cooperative Work (CSCW)* 8, 4 (Dec. 1999), 333–352. <https://doi.org/10.1023/A:1008716330212>
- [72] Rada Shirkova. 2011. Materialized Views. *Foundations and Trends® in Databases* 4, 4 (2011), 295–405. <https://doi.org/10.1561/19000000020>

- [73] 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>
- [74] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. 1998. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction* 5, 1 (March 1998), 63–108. <https://doi.org/10.1145/274444.274447>
- [75] Ivan E. Sutherland. 1963. Sketchpad: A Man-Machine Graphical Communication System. In *Proceedings of the May 21-23, 1963, Spring Joint Computer Conference (AFIPS '63 (Spring))*. Association for Computing Machinery, New York, NY, USA, 329–346. <https://doi.org/10.1145/1461551.1461591>
- [76] Philip Tchernavskij. 2019. *Designing and Programming Malleable Software*. Ph.D. Dissertation. Université Paris-Saclay, École doctorale n°580 Sciences et Technologies de l'Information et de la Communication (STIC).
- [77] Peter van Hardenberg and Martin Kleppmann. 2020. PushPin: Towards Production-Quality Peer-to-Peer Collaboration. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, Heraklion Greece, 1–10. <https://doi.org/10.1145/3380787.3393683>
- [78] Lea Verou, Amy X. Zhang, and David R. Karger. 2016. Mavo: Creating Interactive Data-Driven Web Applications by Authoring HTML. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. Association for Computing Machinery, New York, NY, USA, 483–496. <https://doi.org/10.1145/2984511.2984551>
- [79] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. 2021. Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Virtual Event Republic of Korea, 2024–2045. <https://doi.org/10.1145/3460120.3484542>
- [80] 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>
- [81] Yifan Wu, Remco Chang, Joseph Hellerstein, Arvind Satyanarayan, and Eugene Wu. 2021. DIEL: Interactive Visualization Beyond the Here and Now. <https://doi.org/10.48550/arXiv.1907.00062> arXiv:1907.00062 [cs]
- [82] Weihai Yu and Claudia-Lavinia Ignat. 2020. Conflict-Free Replicated Relations for Multi-Synchronous Database Management at Edge. In *2020 IEEE International Conference on Smart Data Services (SMDS)*. IEEE, Beijing, China, 113–121. <https://doi.org/10.1109/SMDS49396.2020.00021>

- [83] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, Virtual Event China, 2653–2666. <https://doi.org/10.1145/3448016.3457559>