Riffle: Reactive Relational State for Local-First Applications

Geoffrey Litt
Massachusetts Institute of Technology
Cambridge, MA, USA
glitt@mit.edu

Nicholas Schiefer
Anthropic
San Francisco, CA, USA
schiefer@mit.edu

Johannes Schickling
N/A
Berlin, Germany
schickling.j@gmail.com

Daniel Jackson
Massachusetts Institute of Technology
Cambridge, MA, USA
dnj@mit.edu

ABSTRACT

The reactive paradigm for developing user interfaces promises both simplicity and scalability, but existing frameworks usually compromise one for the other. We present Riffle, a reactive state management system that achieves both simplicity and scalability by managing the entire state of a web application in a client-side persistent relational database. Data transformations over the application state are defined in a graph of reactive relational queries, providing developers with a simple spreadsheet-like reactivity model. Domain state and UI state are unified within the same system, and efficient incremental query maintenance ensures the UI remains responsive. We present a formative case study of using Riffle to build a music management application with complex data and stringent performance requirements.

CCS CONCEPTS

• **Human-centered computing** → **User interface programming**; • **Information systems** → **Data management systems**.

KEYWORDS

UI State Management, Reactive Programming, Relational Databases

ACM Reference Format:
Geoffrey Litt, 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 01, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3586183.3606801

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. One powerful abstraction for defining such views is *reactive data transformation*, as seen in early UI frameworks like Garnet [19] 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.

UIST '23, October 29–November 01, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0132-0/23/10.
https://doi.org/10.1145/3586183.3606801

The concept of reactivity is well known in spreadsheets, arguably the most successful paradigm for allowing users of all levels of skill to build software. In fact, prior work [3, 4] has shown that spreadsheets themselves can even be used as a substrate for specifying reactive data transformations that power simple GUI applications.

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 database, an ORM, a client-side cache, in-memory state, etc.—losing the 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 throughout. The core simplicity of "application as spreadsheet" has been lost.

This paper describes a novel architecture for application state management which both provides a simple declarative programming model and scales up to meet the needs of complex real-world applications (Figure 1). Our work makes the following contributions:

1) Reactive Relational Architecture. We describe an architecture with two key concepts (Section 3):

- **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 system, we can guarantee that the UI and all state visible to the programmer is always in a consistent state, without inconsistencies across parts of the view.

We use a *local-first* [10] 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.

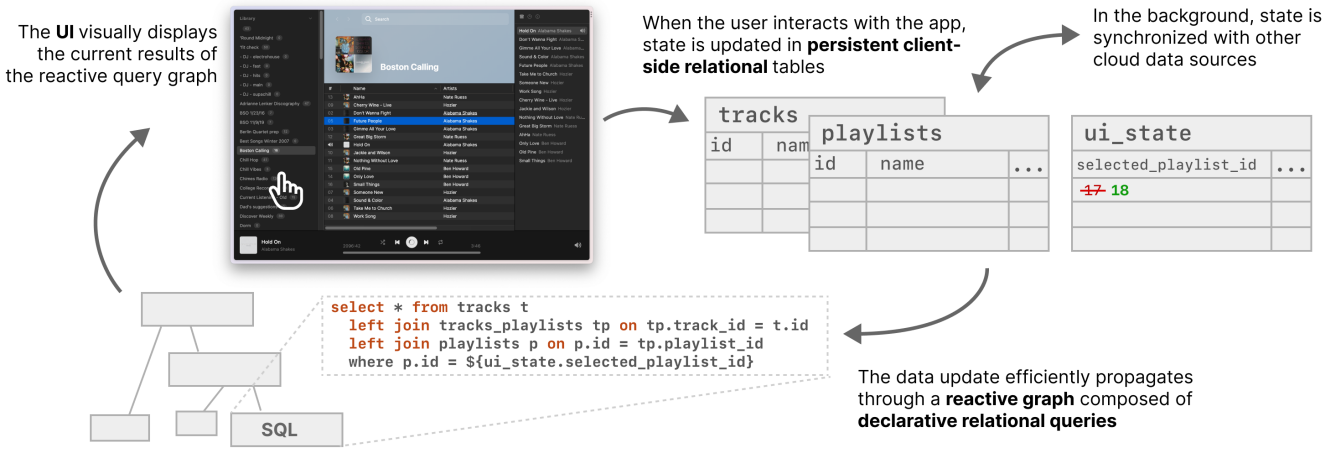


Figure 1: An overview of the Riffle architecture. The UI visualizes the results of a reactive graph of relational queries on a persistent client-side relational database. The dataflow loop runs synchronously on the UI thread, supporting fast, transactional reactivity. In the background, the local relational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

2) Implementation. We demonstrate a working implementation of these ideas in a TypeScript library that provides APIs for application developers to specify data transformations using reactive relational queries. To ensure responsiveness we build on top of SKDB[1], a relational database that supports efficient incremental updates, as well as network synchronization of data.

3) Case study. 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 6).

4) Heuristic Analysis. To further analyze the benefits and tradeoffs of Riffle, we present a heuristic analysis following Olsen's criteria for evaluating systems research [21] (Section 7).

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. Looking ahead, this approach also lays the foundation for future benefits like end-user customization and data-centric interoperability across applications.

2 BACKGROUND AND RELATED WORK

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: e.g., a client written in JavaScript, a server written in Python, and a database queried with SQL. Developers must wrangle data across these layers, manipulating copies of the same information in different representations. They must manually manage the network boundary, including 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. In addition to requiring tedious effort, application-level state management code is a common source of bugs [2].

Riffle proposes a simpler architecture by building on advances in three related research areas.

Reactive UI. In the space of web applications, we can roughly differentiate between *client-side* and *full-stack* approaches to reactive dataflow.

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 [18, 19], as well as functional reactive programming libraries like Flapjax [16] and Vega-Lite [24]. Modern web view templating layers like React.js provide automatic reactive maintenance of UI trees; state management libraries such as MobX, Recoil, Jotai, and Datascript provide additional utilities for managing state and reactive dependencies outside the UI tree.

Riffle shares the basic idea of these tools but has two important differences. First, Riffle applies reactive relational queries to the user's entire dataset (which has been synchronized to the client device) rather than only offering reactivity over a small partial subset of the user's data like most tools for reactive UI on the web. Second, Riffle allows users to write relational queries, which are more declarative and easier to automatically optimize than transformations written in general-purpose languages.

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 [3] and Object Spreadsheets [14] give end-users tools to define web applications that can persist data to a server backend, while defining data transformations in a spreadsheet interface. Links [6] and Ur/Web [5] 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

[1] https://skdb.io/

client. Commercial systems like Meteor, Firebase, Convex, and Electric Clojure offer a variety of tools for managing automatic reactivity across the client-server boundary.

Local-first software. Riffle shares the goal of the systems above of reducing the burden of manual network management, but with a major difference in its approach. In most full-stack web frameworks, the network is kept as a core part of each user interaction; in contrast, Riffle uses an architecture known as *local-first* [10] in which local client-side data is treated as a source of truth that can be queried or written to at any time, and is synchronized across the network in the background when a connection is available. This architecture can support fast interaction latencies and good offline support because interactions primarily proceed on the local client.

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 app[2], an issue tracker[3], an email client[4], and an RSS reader[5]. Our music manager case study in Section 6 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 8.1.

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 SQL[6] do provide a relational model, but do not tightly couple the UI and database to the extent that Riffle does.

Incremental view maintenance. Riffle builds on decades of work on materialized views [9] and incremental view maintenance (IVM) in database systems. Summarizing the extensive literature on IVM is beyond the scope of this paper; we refer the reader to [26] 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. Typically, IVM is implemented only for a subset of the SQL query language[7]: for example, it is not common to support incremental view maintenance for correlated subqueries or recursive computed table expressions, though techniques for these are known [1, 8]. Other systems, such as Noria [7], add IVM on top of existing databases.

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 queries, building on the timely dataflow [17] and differential dataflow [15] incremental computation frameworks. SKDB, the embedded database we use in Riffle, achieves fast IVM through Skip[8], a programming language that offers native support for dependency tracking and cache invalidation.

3 RIFFLE 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.

3.1 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.

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:

- 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, since programmers can specify a logical query and let a database execute it efficiently

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

[2] https://actualbudget.com/
[3] https://linear.app/
[4] https://superhuman.com/
[5] https://readwise.io/read
[6] https://electric-sql.com/

[7] https://docs.oracle.com/database/121/DWHSG/refresh.htm#DWHSG8372, https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-refresh-sql-command.html
[8] http://skiplang.com/

language, including business logic that would normally be written in a non-relational language.[9]

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.

When combining reactivity and relational queries, the two forms of declarativity 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.

3.2 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 2. 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 2, 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. The obvious downside is latency for the user, but there is a more subtle problem: the UI is in an inconsistent intermediate state, since Item B is selected in the sidebar, but Item A is still shown in the main pane. This adds new states to the system that the developer and the user must reason about.[10]

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 2, 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. The architecture enforces consistency throughout the execution; the sidebar and the main pane *must* always reflect the same state.

Coordinating UI state and domain state. Synchronous transactional updates are enabled by Riffle's approach of storing UI and domain state together. In many application frameworks, UI state and domain state are managed in separate layers separated by an asynchronous boundary, making transactional coordination or impossible. 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. Nielsen defines 100ms as the rough response time limit for an action to feel instantaneous to a user [20]; some applications like Superhuman aim for under 50ms[11]. 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 a local-first architecture avoids the frequent sources of asynchrony seen in typical web applications, we cannot always avoid asynchronous calls—e.g. requests to an external API or running a slow computation.

We model these 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.

4 SYSTEM IMPLEMENTATION

Riffle provides data management abstractions that sit between existing systems at the view layer and the database layer. We use React.js as a view framework to handle outputting UI to the browser DOM, and SKDB as a relational database which handles queries (including incremental view maintenance), persistence, and network synchronization. The Riffle library provides APIs that the application developer uses to bind data and queries to UI components, and contains an implementation of a reactive update graph that tracks dependencies between queries to efficiently schedule updates. We elaborate further on the details of the implementation in Appendix A.

5 RIFFLE BY EXAMPLE: TODO LIST

In this section, we demonstrate how the Riffle concepts apply in practice, by implementing TodoMVC, a small reference application commonly used to compare UI and state management tools. We 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 difference from traditional data modeling is that our UI state will eventually be modeled in the relational schema, but for now, we begin by modeling our *domain*

[9] Recognizing 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 A.

[10] There 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.

[11] https://blog.superhuman.com/superhuman-is-built-for-speed/

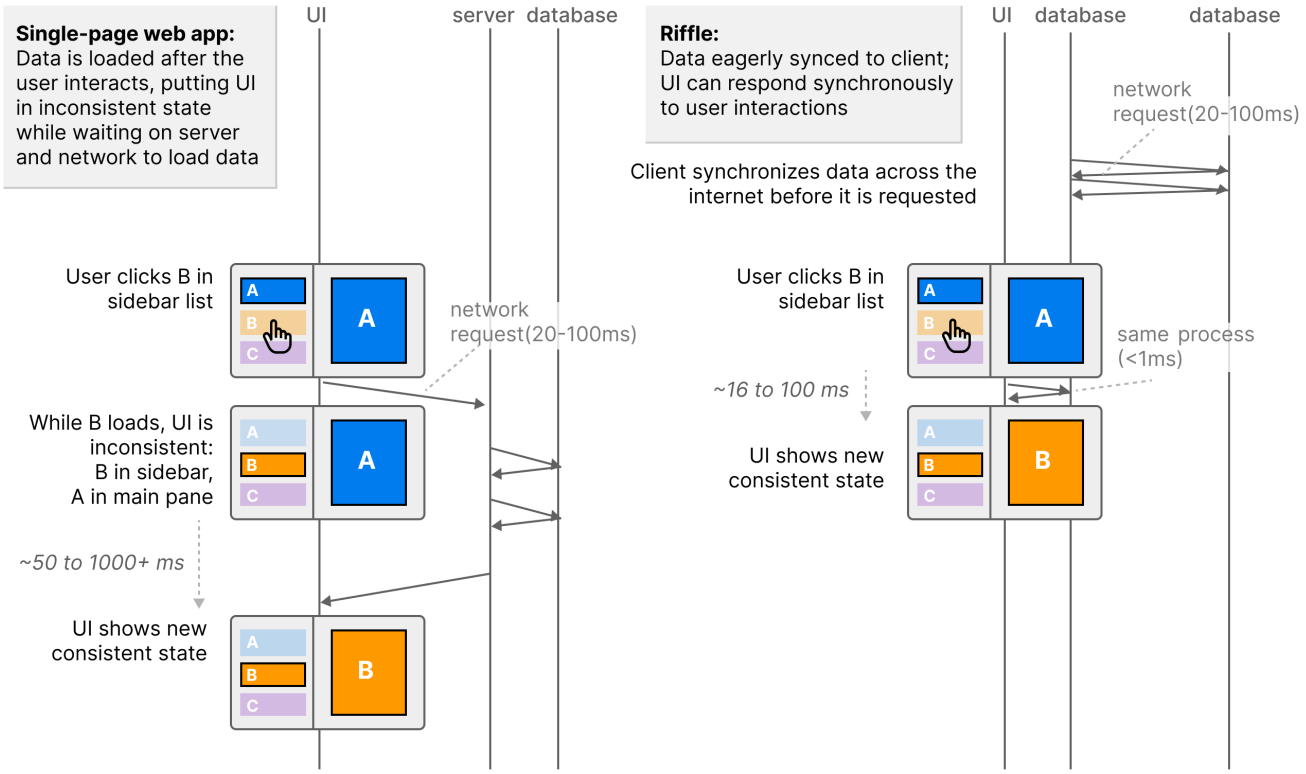


Figure 2: In a single-page web application, user interactions frequently incur network latency and leave the UI in a temporarily inconsistent state. In contrast, Riffle's local-first architecture and synchronous transactional updates enable faster responses. The UI can respond to the interaction immediately 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.

state: a `todos` table with 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, `select * from todos;`, that loads all the todos. We write it inside a React component using a hook[12] provided by Riffle called `useRiffleComponent`:

```
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>
}
```

[12]Hooks are a standard mechanism in React for mixing in state and behavior to a component.

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. React.js handles reactivity at the view layer: it computes a virtual DOM representing the new UI structures, diffs this against the previous virtual DOM for the component, and then efficiently applies minimal (expensive) changes to the browser DOM.

Because the `todos` table is managed by an underlying instance of the SKDB database, it is also automatically 1) persisted locally and 2) 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.

We create a new table `ui_state`, with a single text column named `newTodoText`. To read the value, we 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 })}
    />
}
```

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 store the value of the filter toggle by extending the existing `ui_state` table with a `filter` column. Now we'd like to use this state to drive queries: eg, if the filter is set to `completed`, then we want to append a clause to our query: `select * from todos where completed = true`.

Figure 3 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 write 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.

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 easy to view the todos in one or more labels, or to filter together by label and completed status.

6 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.

6.1 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 database after being downloaded from the streaming provider.[13] 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*.

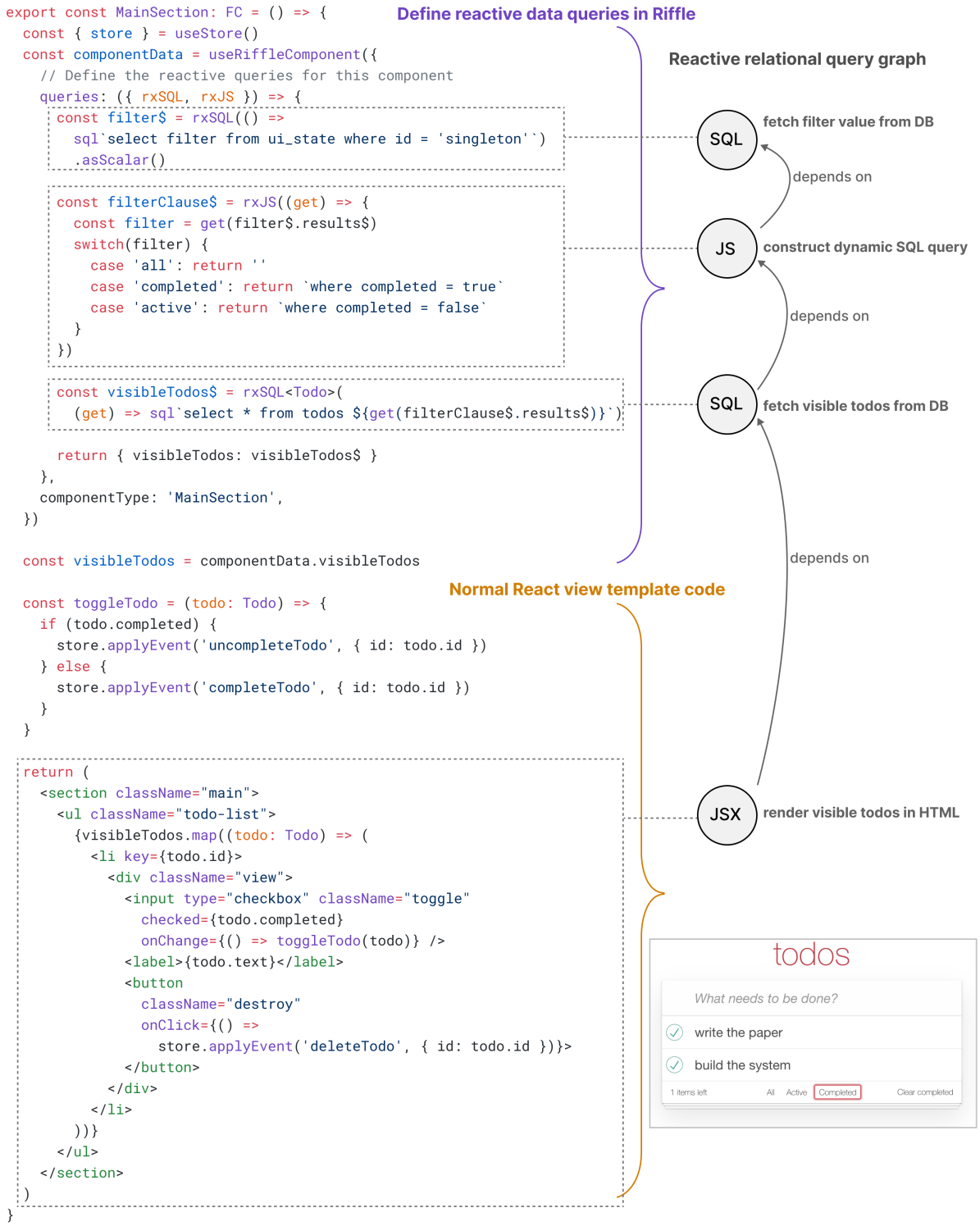
Performance. Overtone's performance goal is to respond to all user interactions within 16ms to render smoothly at 60fps; as a minimum threshold, we adopt Nielsen's limit of 100ms [20] for interactions feeling "instant." This target is far beyond the performance seen in music clients for many popular 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.

6.2 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

[13] Overtone 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.




```
export const MainSection: FC = () => {
  const { store } = useStore()
  const componentData = useRiffleComponent({
    // Define the reactive queries for this component
    queries: ({ rxSQL, rxJS }) => {
      const filter$ = rxSQL(() =>
        sql`select filter from ui_state where id = 'singleton'`)
        .asScalar()

      const filterClause$ = rxJS((get) => {
        const filter = get(filter$.results$)
        switch(filter) {
          case 'all': return ''
          case 'completed': return `where completed = true`
          case 'active': return `where completed = false`
        }
      })

      const visibleTodos$ = rxSQL<Todo>(
        (get) => sql`select * from todos ${get(filterClause$.results$)}`)

      return { visibleTodos: visibleTodos$ }
    },
    componentType: 'MainSection',
  })

  const visibleTodos = componentData.visibleTodos

  const toggleTodo = (todo: Todo) => {
    if (todo.completed) {
      store.applyEvent('uncompleteTodo', { id: todo.id })
    } else {
      store.applyEvent('completeTodo', { id: todo.id })
    }
  }

  return (
    <section className="main">
      <ul className="todo-list">
        {visibleTodos.map((todo: Todo) => (
          <li key={todo.id}>
            <div className="view">
              <input type="checkbox" className="toggle"
                checked={todo.completed}
                onChange={() => toggleTodo(todo)} />
              <label>{todo.text}</label>
              <button
                className="destroy"
                onClick={() =>
                  store.applyEvent('deleteTodo', { id: todo.id })}>
              </button>
            </div>
          </li>
        ))}
      </ul>
    </section>
  )
}
```

Figure 3: 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.

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 incrementally maintained 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 4. We also define a GraphQL schema on top of the base relational schema; we describe further below the problems that motivated the GraphQL layer.

6.3 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 polling over the network.[14] 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.[15] 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 4) 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 Appendix A 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 4). 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 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.[16]

6.4 Reflections

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 started out as an external partner; he ended up making substantial contributions to Riffle itself, and is now a co-author on this paper. Most development was done by the solo lead developer, with some part-time help from the other authors of this paper as well as several other developers.

[14] Because 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.

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

[16] We 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 library[17] that draws to the more efficient Canvas API.

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
```

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
```

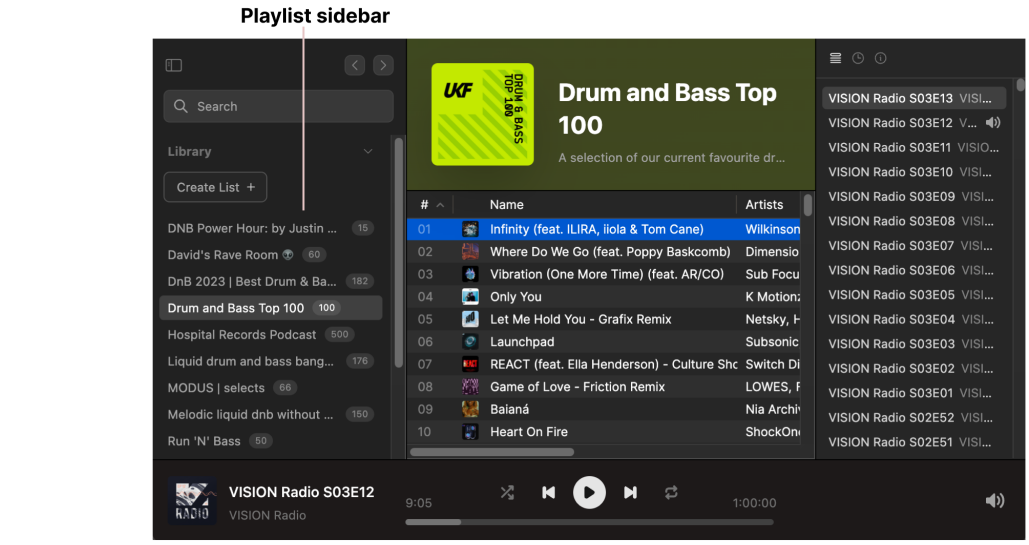


Figure 4: Examples of a SQL query and view definition used in the Overtone music manager.

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 ~2ms 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 noticeabe 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 paper. 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 3.2.)

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 8.1.

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.[18]

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 GraphQL[19] 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 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 until the import had reached that

[18] 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.

[19] https://graphql.org/

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

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

song. Because the import latency was constrained by the streaming service's rate limiting, and by database insert throughput, 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.

6.5 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 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.

7 HEURISTIC EVALUATION

Olsen [21] 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 8.1.

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.

8 DISCUSSION

8.1 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.[20] 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 CRDTs [23, 25, 29] 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 [27].

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 [12] 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; Partiql[21] is one example of a 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 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 [28], 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

[20] As mentioned in Section 6.4 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.

[21] https://partiql.org/

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.

8.2 Future work

Data substrate for interoperability. Prior projects have explored *shared data substrates* that enable interoperability between tools. For example, Webstrates [11] stores information in the browser DOM and synchronizes it over the Web, SOLID [13] stores information in personal data pods controlled by the user and accessed via Web APIs, Plan 9 [22] uses the desktop filesystem to share data among tools, and Dynamicland[22] offers a global reactive database tied to a physical space. 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. 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.

Live programming. The debugger view we have built (Section A) 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 dynamically editing running queries within the debugger, and showing richer views of the dependency structure such as a visual graph.

9 CONCLUSION

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 a responsive and consistent UI. We have demonstrated that this architecture is capable of supporting the ergonomic development of a real-world application with a powerful user experience.

[22]https://dynamicland.org/

ACKNOWLEDGMENTS

Thank you to Skip Labs for collaboration on integrating SKDB, to RelationalAI for their support of this project, to Matt Wonlaw for advice on using SQLite, and to David Karger for feedback on this paper. Geoffrey Litt was supported by an NSF GRFP Fellowship and the NSF SaTC Program (Award 1801399). Nicholas Schiefer was supported by a Simons Investigator Award.

REFERENCES

[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] 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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] Alon Y. Halevy. 2001. Answering Queries Using Views: A Survey. *The VLDB Journal* 10, 4 (Dec. 2001), 270–294. https://doi.org/10.1007/s007780100054

[10] 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

[11] Clemens N. Klokmose, 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

[12] 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 (PaPoC '21)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3447865.3457963

[13] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, 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

[14] 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

[15] 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

[16] 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

[17] 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, Farminton Pennsylvania, 439–455. https://doi.org/10.1145/2517349.2522738

[18] 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

[19] 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.

[20] Jakob Nielsen. 1993. Response Times: The 3 Important Limits. https://www.nngroup.com/articles/response-times-3-important-limits/

[21] 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

[22] 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.

[23] 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

[24] 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

[25] 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

[26] Rada Shirkova. 2011. Materialized Views. *Foundations and Trends® in Databases* 4, 4 (2011), 295–405. https://doi.org/10.1561/1900000020

[27] 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

[28] 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]

[29] 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

APPENDIX

A IMPLEMENTATION DETAILS

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

Relational database: As a backend for persistence and querying, Riffle uses SKDB, an open-source relational database[23]. SKDB has two particularly useful properties for building the kinds of UIs that Riffle aims to support.

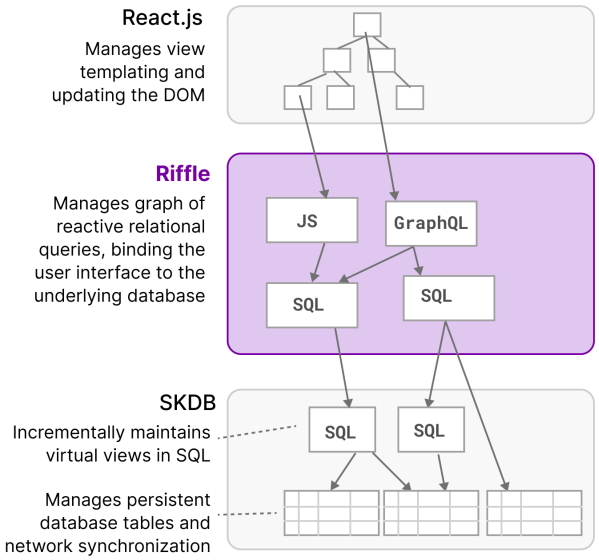


Figure 5: Implementation architecture: The Riffle library sits between React (for view templating) and SKDB (for all data storage and queries)

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.[24] 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 paper; we treat these features as black boxes. These are highly general primitives that we depend on to enable the Riffle architecture.

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.

[23]Note: the source code for SKDB has not been released as of this writing, but it will be open source by the time this paper is published. A WASM binary is publicly available here: https://github.com/SkipLabs/skdb. For more information, see: https://skdb.io/

[24]Incremental 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.

Figure 6: The Riffle debugger shows the underlying state and queries for the TodoMVC application.

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.

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.[25] 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.

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**[26] 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.[27]

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.

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 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.

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

[25]Table 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.

[26]https://graphql.org/

[27]There 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.

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.[28]

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 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.

Debugger: We have implemented a simple debugger (shown in Figure 6) 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.

[28] So 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.