DIEL: Interactive Visualization Beyond the Here and Now

Yifan Wu, Remco Chang, Joseph M. Hellerstein, Arvind Satyanarayan, and Eugene Wu

Abstract— Interactive visualization design and research have primarily focused on local data and synchronous events. However, for more complex use cases—e.g., remote database access and streaming data sources—developers must grapple with distributed data and asynchronous events. Currently, constructing these use cases is difficult and time-consuming; developers are forced to operationally program low-level details like asynchronous database querying and reactive event handling. This approach is in stark contrast to modern methods for browser-based interactive visualization, which feature high-level declarative specifications. In response, we present DIEL, a declarative framework that supports asynchronous events over distributed data. As in many declarative languages, DIEL developers specify only what data they want, rather than procedural steps for how to assemble it. Uniquely, DIEL models asynchronous events (e.g., user interactions, server responses) as streams of data that are captured in event logs. To specify the state of a visualization at any time, developers write declarative queries over the data and event logs; DIEL compiles and optimizes a corresponding dataflow graph, and automatically generates necessary low-level distributed systems details. We demonstrate DIEL's performance and expressivity through example interactive visualizations that make diverse use of remote data and asynchronous events. We further evaluate DIEL's usability using the Cognitive Dimensions of Notations framework, revealing wins such as ease of change, and compromises such as premature commitments.

Index Terms—Interactive Visualization Toolkit/Library, Scalability, Asynchrony

1 Introduction

Recent advances have made authoring browser-based interactive visualizations quite simple, via novel abstractions for specifying encodings [6, 56], layout, data transformations, and interactions [44, 46]. Critically, these abstractions enable *declarative* specification: developers can compose these visualization elements and defer execution concerns to the toolkit runtime. Declarative abstractions thus enable developers to build and iterate on designs quickly and expressively, while reducing the errors that would arise from imperative implementation [18, 46]. However, these abstractions do not gracefully support use cases where data may be distributed across multiple locations, or where events cannot be synchronously handled to completion. To illustrate, we discuss two interactive visualization settings—one quite traditional and another more complex—and reflect on the difficulty of addressing these issues with current practices.

Consider the *client-server architecture*, a classical design for scalable interactive visualizations that remains common even in recent research [30, 31]. A concrete example is shown in Fig. 1①, where the brush selection over the scatterplot is computed by a remote database. To handle the brush selection, a developer can no longer rely on existing declarative abstractions (e.g., Vega-Lite's selections [44]) alone. Instead, they must now translate interaction logic from a visualization specification to the query language of the database, network with the remote database, and manually bookkeep the provenance of interactions and database responses to ensure that the interface is displaying the correct results, in the correct order, in the face of non-deterministic processing latency.

A more specialized example is *streaming data*, increasingly investigated in research [25, 53, 62] and industry [64]. An example is shown in Fig. 1②, where the points can be streamed in the scatterplot and selected by a brush interaction. The two processes — the user's brushing interaction, and the tweet stream — introduce ambiguity about what should happen when both attempt to update the visualization concurrently. For instance, consider the case when a new point streams in, falling within a brushed region. Should this new point be part of the existing selection, or should it be isolated in a new separate selection? Should the new point be blocked until the user removes the brush selection, or should the point be added and the brush selection removed? These decisions can be further complicated if the brush selection is processed by a remote server, which introduces another asynchronous event that multiplies the complexity.

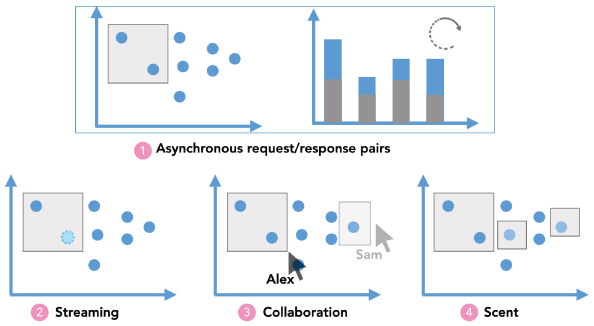


Fig. 1: Example interactive visualization use cases where (1) data may not fit into the browser and thus lives in a remote database, and/or (2) events, including user interactions and data updates from servers, need to be evaluated asynchronously. Concretely, ① shows interactions taking time to process, creating asynchrony between interactions and result updates (see Fig. 4 for details). ② shows data updating in real time as the user is interacting with a visualization. ③ shows an example collaboration experience [33]; ④ shows a visualization of past interactions to curate "information scent" (see Fig. 9).

For a designer to explore decision space fluidly [17], we want to to minimize the friction of experimenting with these choices. But distributed and streaming visualizations still have a high programming barrier, despite progress on declarative libraries for browser-based data visualizations. The effect is that designers get "cornered" into whatever is easiest: for example opting for a high-latency blocking design, or disallowing users from interacting with streaming data, purely for ease of implementation rather than superior usability. To improve design fluidity in these use cases, we identify two key abstractions that are missing from existing interactive visualization frameworks:

Distributed Data. Data is increasingly distributed at different compute nodes Traditionally, this was across multiple database servers hosting a large dataset, however user-facing applications increasingly

- *Yifan Wu and Joseph M. Hellerstein are with the University of California at Berkeley. E-mail: yifanwu@berkeley.edu.*
- *Remco Chang is with Tufts University. E-mail: remco@cs.tufts.edu.*
- *Arvind Satyanarayan is with MIT CSAIL. E-mail: arvindsatya@mit.edu.*
- *Eugene Wu is with Columbia University. E-mail: ewu@cs.columbia.edu.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

distribute data across browser threads, and across browser state and remote databases as well. As such, developers currently need to write code to access the data, which they often need to further optimize. Our goal is to develop abstractions that (1) free developers from sending data manually between local and remote processes, (2) allow developers to easily change their data storage and compute sources (e.g., between the main browser thread, a WebWorker [32], a local database, or cloud databases) without rewriting their application code, and (3) unobtrusively perform low-level optimization (e.g., caching).

Asynchronous Events. When data is local on one client, user interactions are the only events that a developer has to work with. Thus, computation caused by interactions, such as filtering data based on a selection, can be handled synchronously. However, when data is remote, asynchronous events are the norm: responses from a server, streaming data from real-world events, or events generated by other users. These asynchronous events are outside of the application's direct control and, thus, have to be handled as they arrive. Our goal is to develop abstractions and a library that allow developers to (1) create consistent user experiences in the face of concurrency and out-of-order execution, and (2) easily experiment with different designs for handling asynchronous events.

To effectively work in this new paradigm of distributed data and asynchronous events, we adapt two patterns from the field of distributed systems programming:

Logical Constraints. Rather than imperatively manipulating data, developers express logical constraints (i.e., queries) over data, and the system determines the methods used to execute the queries [9, 40]. By decoupling the *how* from the *what*, logical constraints allow the system to plan and optimize the execution of the query (possibly spanning multiple databases) [40], relieving the developer of this burden. This approach also has practical implications. In particular, SQL and dataframe libraries express logical constraints over data, and have been widely adopted by developers and database engines [3, 35, 49] alike.

Immutable Events. Past events, along with their time steps, are stored as an immutable log by the system. Developers declare the current state of the application as a logical constraint over the immutable log rather than transiently handling events and mutating state via side effects [2, 19]. As a result, developers can coordinate events declaratively without worrying about how to update the application state through callbacks over transient events.

In this work, we bring these ideas from distributed systems programming to the context of interactive visualization. DIEL[1] models interactions and asynchronous data computation events as timestamped records in event tables, and the state of the interface as a query over the event tables and data tables. The DIEL runtime maintains an event loop to reevaluate queries as event tables change, ensuring that the interface is responsive to interactions. With DIEL, we are able to achieve the requirements outlined earlier — developers no longer need to implement low-level networking with remote databases or perform optimizations, and have a way to easily coordinate asynchronous events that maintain a consistent interface.

To evaluate DIEL's expressiveness, we construct a diverse set of interactive visualizations over distributed data and asynchronous time. Examples include ways to handle request-response asynchrony [59, 61], interactions on streaming data, composing two related interactions, and interaction scents [14, 57]. These examples show that DIEL's abstractions allow developers to rapidly and concisely explore different interface designs. We also present a heuristic analysis of the usability of these abstractions using the *Cognitive Dimensions of Notation* framework [5], highlighting both gains to fluidity, and compromises to premature commitment. Finally, we verify the viability of DIEL through performance measurements of a prototype, and show that DIEL adds low overhead and scales as data increases in size.

2 Related Work

DIEL builds on prior work in databases, visualization systems, and distributed systems programming.

[1] **D**eclarative **I**nteraction **E**vent **L**og

Database and Visualization Systems. Research at the intersection of visualization and database systems has traditionally focused on enhancing performance — for instance, by offloading data aggregation and filtering to a remote database [30, 51] or by embedding a high-performance database-like query engine into the browser [44]. In contrast, our work is *not* motivated primarily by performance but instead by the programming experience of visualization developers. Although DIEL embeds a SQL query engine in the browser, it does so for its benefits as a *programming construct*, and its ability to interoperate with a remote database. Critically, DIEL is agnostic to the specific database system, which could range from a SQLite instance in the browser's main thread [27], to commonly used databases on a server, such as SQLite, Postgres [35], or new research systems [26, 39].

FORWARD is a general web application programming framework that directly binds database records and query results to DOM elements [15]. DIEL is similar in that it manages how elements in the visualization are updated based on changes in the underlying database that represents user interactions. However, DIEL uniquely addresses the challenges of asynchronous events, which are shown to be especially challenging for interactive visualizations [61]. DIEL does so through a novel declarative programming API over event histories.

Visual Programming over Tables. There is a close connection between visual querying and textual queries — visual querying systems combine textual query specification with direct manipulation and data visualization. For instance, VIQING maps visual selections, joins, and reordering to SQL operators [34]. Polaris endowed the pivot operator with powerful interactive capabilities [51]. Wrangler brings together interactions and a query based DSL for data transformations [23]. Many commercial tools, such as Trifacta, Tableau, and Microsoft Power BI/Query, provide a combination of these capabilities.

Beyond the connection between queries and interactive visualization, Psallidas and Wu demonstrated the application of query *lineage* in interactive visualizations [38]. Also leveraging lineage, B2 instruments interactive cross-filtering visualizations in computational notebooks [60]. DIEL builds on these close connections using relational queries to define the data transformation logic of interactions. However, DIEL differs from these systems in that it is a programming abstraction for a middleware layer designed to support general purpose interactive visualization programming.

Interactive Visualization Libraries. While DIEL makes use of interactive visualization libraries, such as D3 [6] and Vega [44], it is not one itself. DIEL relies on the front-end visualization libraries to map data to visual encodings, perform visualization-specific transformations (such as `voronï`, `treemap`, or `wordcloud` [37]), create interactors, and capture selection values. Existing abstractions in these libraries support interactions in a *synchronous* setting where the computation is expected to finish immediately. As a result, the events are also transient by default and assumed to be unnecessary for subsequent events. In contrast, DIEL supplements existing frontend visualization libraries by filling in the gap of working with distributed data and asynchronous events. Of particular note is how DIEL differs from Vega, which also models interaction events as streaming data. Like other libraries, events in Vega are transient by default and, although a rich set of data transformations are offered, they only operate over client-side data. DIEL, on the other hand, records all events into a persistent log and offers a simple set of relational abstractions over distributed data. Future research could consider how to further extend Vega's dataflow abstractions [45, 46] to better integrate with, or adopt DIEL's abstractions.

Frameworks for Provenance and Asynchrony. With current programming paradigms, developers need additional instrumentation to support features like logging or undo/redo. For example, *Trrack* is a purpose-built library that augments existing interactive visualizations with a provenance graph of state history [12]. DIEL, in contrast, offers a more general-purpose set of abstractions which provide provenance tracking via first-class event histories.

Similarly, to be able to share and synchronize multiple users' changes over the network, programmers of collaborative groupware applications rely on special-purpose frameworks. Toolkits such as *Janus* help developers resolve concurrent and possibly conflicting events to ensure that

each participant views a consistent artifact [47]. DIEL takes inspiration from these systems, e.g., the time-stamped events in Janus [47], but also adds specialization for interactive visualizations, such as providing support for working with distributed data and tracking request-response provenance. DIEL was directly inspired by distributed programming language research, notably the Bloom [2] language and CRDT [36] data structures. Like Bloom, DIEL focuses on the semantics of atomic timesteps immediately after the user interacts, helping developers reason about out-of-order events. DIEL differs from the prior work in its focus on reifying a log of history as a core aspect of its data model.

Reactive Programming. *Reactive Extensions* (Rx) is a widely-used example of a Functional Reactive Programming (FRP) library, which coordinates event-based and asynchronous computations such as mouse moves and high-latency calls to Web services [29]. Vega [46] and Bloom [2] also follow the model of reactive programming. DIEL takes inspiration from these libraries, implementing reactive programming semantics. Meijer's article about Rx, *Your Mouse is a Database* [29], informed our formulation of user events as record insertions into tables.

However, compared to Rx, DIEL has a much more restrictive model targeted at visual analytic applications. Unlike Rx, DIEL does not support general purpose applications but provides domain specific functionalities for the use cases in interactive visualization. One could implement DIEL-like logic as a specialized design pattern in an FRP language like Rx but doing so would involve implementing the distributed query execution logic that DIEL provides, which is not a casual task for a visualization developer.

3 Programming Interactions Across Space & Time

To address the challenges of asynchronous events and distributed data, we look to distributed systems programming for inspiration and adapt the ideas to interactive visualization programming.

3.1 Asynchronous Interactions: Immutable Events

Asynchronous events, unlike their synchronous counterparts, need to be coordinated by developers to ensure a good user experience [61]. This coordination can be challenging as developers need to maintain relevant information of past events and selectively trigger event handlers. And, since different parts of the event handling and bookkeeping logic are scattered across functions and variables, these imperative bookkeeping and callbacks can become tedious to maintain and prone to errors. This situation is exacerbated when new types of events need to be supported (e.g., a new interaction). To address these challenges, we look to key methods in distributed systems programming.

Events as Data. An important technique in distributed systems programming is making events immutable and first class [19]. Events are stored as a log, and the state after each event is defined as a function (or query) of the log [1]. This design abstracts away the details of maintaining state (no more mutations with callbacks) and makes it easier to reason about the consistency of the application [2]. We can apply this framing directly to interactive visualizations: user interactions are events, and queries over these events (and other tables) can specify the state of the interface.

Atomic Timesteps. Compared to synchronous events, asynchronous events can, by definition, arrive out of order. This behavior can lead to a variety of "inconsistent" visualization states, as documented by Wu et al. [61]. One simple example is naively rendering a response when it is received, even if it is for an "old" interaction made prior to the most recent one. To deal with unpredictable sequences of events, developers need to reason about what prior events occurred, and when they did so.

Given this challenge of inconsistency, one obvious solution is to synchronize the events. However, this solution violates a core HCI principle: *responsiveness* [20, 21]. Making events synchronous means that the user is *blocked* from performing new interactions until their previous interaction result has been executed completely (including being sent to the server, computed on the server database, received by the client, and rendered to the UI). As this process can take a significant amount of time to complete, making the interface unresponsive yields a frustrating user experience.

To maintain a smooth user experience, it is important to work *with* asynchronous events. Here, one idea from distributed programming is particularly helpful: *atomic timesteps* [2], whereby the system guarantees an atomic unit of evaluation by computing the state of the application in full before admitting another input event. With atomic timesteps, developers reason independently, "frame by frame", about what computation needs to be computed synchronously within a given timestep. Moreover, developers can reference the explicit timesteps stamped on events, with the guarantee that an event with a smaller timestep precedes an event with a larger timestep. Finally, developers can be sure that the interface satisfies the constraints specified at every timestep, which could support robust UX in the face of asynchrony.

3.2 Distributed Data: Logical Constraints

To be able to scale an interactive visualization application, a developer must be able to flexibly change where the data is stored and computed upon. They may initially build a prototype over small datasets that are stored and computed in the main thread of the browser. To use a larger dataset, they may utilize WebWorkers [32], which allow the data-rich tasks to run asynchronously in the background and not block the interface. Then, as the developer deploys the application to real-world datasets, they may move the computation to a database on their local machine or a cloud database.

To achieve this flexibility, the system should abstract data access details from developers. The field of databases has tackled this problem using a concept that is not unfamiliar to interactive visualization developers: *relations* (also called *tables*). Relations are sets of data tuples with a fixed schema, and computation over relations is governed by an algebra of select, project, join, and group by operators [9]. Relational query languages bring with them two important properties for our purposes:

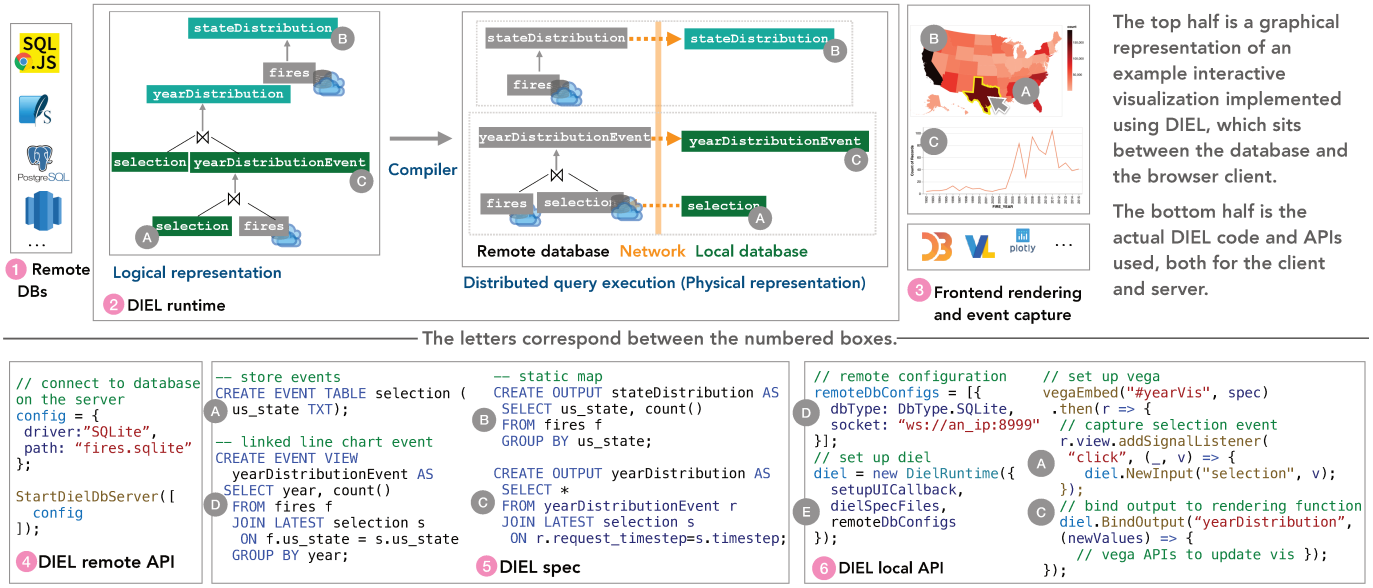
Physical Data Independence. Currently, to change from storing and computing data in the main browser thread to any other options requires custom code: developers may need to map interaction logic from JavaScript to SQL, handle the database connection, and perform networking. This work is needed for two reasons. First, developers often specify *how* to access the data. If the data location changes, the program changes. Second, the computation abstractions on the client and other locations may be different. Relational languages can address both of the factors. They allow developers to specify *what* data to access, making the program independent from the physical details of the data. Furthermore, if a standard relational language is used everywhere, be it the client, a WebWorker thread, or a remote database, the developer can work with one abstraction and not have to translate between specifications.

Rich Optimization Currently, developers may implement optimizations manually, such as caching. This is not ideal for two reasons: one is a higher programming barrier, and another is that the developer may not have enough time for more involved optimizations beyond a simple cache. Having relational abstractions over both the client and the server can relieve the optimization burden from the developer and allow the system to compile the logical specification into a physical execution plan, using a wealth of optimization techniques [40].

4 The DIEL Model

To address the challenges in Section 1, DIEL manages user interactions, remote databases, and the communication between local and remote. On the browser frontend, the developer specifies the data that the user can interact with and select (e.g., using Vega-Lite [44]) and translates them into events that are sent to DIEL via its JavaScript API. DIEL stores events in the local in-browser local database and manages query processing on the remote databases, which are connected to DIEL through a set of remote-side APIs. Through standard database connection libraries such as `node-postgres` and `sqlite3`, DIEL allows the developer to connect to remote databases ranging from SQLite, to PostgreSQL, to cloud databases like Amazon Redshift.

Note that remote databases and their asynchronous complexities can arise in surprising settings. In addition to databases on remote servers (such as Redshift), remote databases also arise when data is




```
// connect to database
on the server
config = {
 driver:"SQLite",
 path: "fires.sqlite"
};

StartDielDbServer([
  config
]);
```

```
-- store events
CREATE EVENT TABLE selection (
  us_state TXT);

-- linked line chart event
CREATE EVENT VIEW
  yearDistributionEvent AS
  SELECT year, count()
  FROM fires f
  JOIN LATEST selection s
    ON f.us_state = s.us_state
  GROUP BY year;
```

```
-- static map
CREATE OUTPUT stateDistribution AS
  SELECT us_state, count()
  FROM fires f
  GROUP BY us_state;

CREATE OUTPUT yearDistribution AS
  SELECT *
  FROM yearDistributionEvent r
  JOIN LATEST selection s
    ON r.request_timestep=s.timestep;
```

```
// remote configuration
remoteDbConfigs = [{
  dbType: DbType.SQLite,
  socket: "ws://an_ip:8999"
}];
// set up diel
diel = new DielRuntime({
  setupUICallback,
  dielSpecFiles,
  remoteDbConfigs
});
```

```
// set up vega
vegaEmbed("#yearVis", spec)
  .then(r => {
  // capture selection event
  r.view.addSignalListener(
   "click", (_, v) => {
     diel.NewInput("selection", v);
  });
// bind output to rendering function
diel.BindOutput("yearDistribution",
  (newValues) => {
    // vega APIs to update vis });
});
```

Fig. 2: Example code using DIEL to power the interactive visualization of wild fires in the US. The user can click on a US state to filter the distribution of fires over the years 3. To create this visualization, developers use DIEL in conjunction with data visualization libraries and remote databases 1. Developers query over timestamped event logs (`selection`) and base dataset (`fire`) to specify the state of the application (a logical spec 5), and DIEL orchestrates the computation between the local and remote databases (an execution plan 2).

managed by a different process on the user's computer. For instance, browser WebSockets communicate with the web page's main thread via asynchronous messaging. For this reason, DIEL is designed to work for any distributed database setting, irrespective of whether the databases are on the cloud.

The developer provides a specification that DIEL uses to coordinate query processing between the local and remote databases. The specification defines the application state that DIEL will manage, and is written in a SQL-like language. We chose SQL as the basis because it is the most widely used data processing languages today [50], and is familiar to developers working with large datasets. However, DIEL is not tied to SQL, and alternative relational languages. It is straightforward to layer a dataframe-like API atop DIEL's abstractions. Given the specification and runtime API calls, DIEL coordinates the local and remote database to exchange data and query results. The query results then update visualization state back to the application via a JavaScript API so that it can render results and update the visualizations.

We next discuss how developers specify different types of application state, and how it is managed during run time. Fig. 2 depicts the running example, and Tables 1 and 2 summarize the syntax.

4.1 Data Model

Static Visualizations Static visualization is a well-accepted domain, where tabular data is fed to APIs that implement a grammar of graphics [44, 56]. For static visualization, DIEL is responsible for preparing the data to be rendered, and invoking the visualization API when the data is ready. The tables used to create visualizations could either be stored tables or the results of queries. For instance, in Fig. 2, the map visualization maps the fields (`us_state, count`) from the query result in (5B) to polygons and fill color in the visualization. When the application is first loaded, DIEL evaluates the initial queries to render the static visualizations. Using the `BindOutput` API (6C), the developer registers a callback *rendering function* that is called when the query results are updated. The developer annotates the queries that the rendering function will access using the `OUTPUT` keyword. The system uses the information to inform the dataflow as well as instrument the `BindOutput` API.

Events DIEL stores events as timestamped tables. When the developer defines visualization interactions, such as selections, the data records that are selected are ingested into DIEL through the `NewEvent` API and stored in tables. For instance, the selection on the map (3A) is mapped via a Vega listener (6A) and stored in the `selection` table (5A), with a single column, `us_state`, representing the state selected, e.g., "CA". An event may be one or more rows. At runtime, each new event is augmented with a *logical timestep*, which we discuss in Section 4.2 , and a *physical timestamp*, recording the wall-clock time of the event. Other external events, such as those generated from an automated process in the browser on behalf of the user (e.g., timers), follow the same model.

Developers mark event tables with the `EVENT` keyword (5A). DIEL generates the corresponding JavaScript handler for the API `NewEvent`. Developers are free to use their favorite frontend libraries (e.g., Vega-Lite, D3) to generate events; they simply invoke the DIEL API to store new events in the event table.

Interactive Visualizations User interactions add events to DIEL event

keyword	syntax CREATE
`event`	`EVENT TABLE <table_name>(<column>)...` `EVENT VIEW <view_name> AS SELECT...`
`output`	`OUTPUT <output_name> AS SELECT...`

Table 1: DIEL syntax for creating tables. An `EVENT TABLE` stores the history of an event. Developers can access the table with two additional columns maintained by DIEL: *timestep* (logic time of the event) and *timestamp* (wall-clock time of the event). An `EVENT VIEW` is a named SQL query (view) that spans the local and server databases. The system appends new view evaluation results to the `EVENT VIEW`, annotated with the timestep of the corresponding `EVENT TABLE` entry tracked by DIEL, *request_timestep*, as well as the *timestep* caused by the new event. Developers access these views similar to how they access event tables. An `OUTPUT` is a view whose results, after each timestep, are evaluated and passed to the rendering function bound via the `BindOutput` API. These three key constructs, `OUTPUT`, `EVENT` and *timestep*s augment SQL with reactive semantics for non-blocking interactive visualizations.

API	syntax
input	`diel.NewEvent('<event>', <object>)`
output	`diel.BindOutput('<output>', <rendering_func>)`

Table 2: DIEL JavaScript APIs to interface with the frontend. `NewEvent` takes in events from the developer, e.g., user's selection. `BindOutput` binds the outputs to a rendering function specified by the developer that takes a table and renders a visualization.

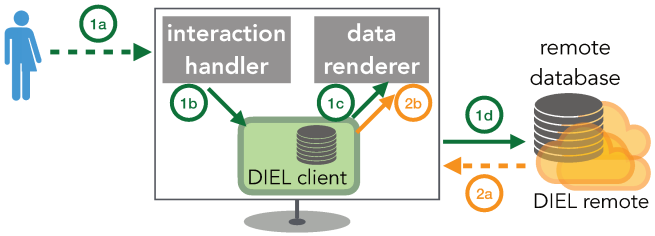


Fig. 3: Illustration of the DIEL runtime. The dashed arrows indicate asynchronous events that advance the system logical timestep forward. There are two in this diagram, one from user interaction (green) one from a database response (orange). The solid arrows indicate synchronous evaluation after each new event.

tables that ultimately update the visualizations. This happens naturally because the developer can query the EVENT tables in the same way as any normal data table. Any such queries marked with OUTPUT will automatically generate the corresponding BindOutput API to register a rendering function callback (used in e.g., 5C).

When a developer is working with a remote database, they need to query *across* both the local and the remote databases. For instance, the fires table is on a remote database, whereas selection is in the local database. DIEL automatically manages the asynchronous communication between the two databases. Due to this asynchrony, we treat such queries as events as well, and ask developers to explicitly mark these queries with EVENT VIEW so they are aware of the table being an event log (despite the fact that DIEL can automatically detect them). For each EVENT VIEW, DIEL maintains the corresponding timesteps of the event that caused its reevaluation, and stores the timestep values as a column named request_timestep in the EVENT VIEW. Developers then use the data in EVENT VIEWs to specify OUTPUTs, with the help of the timesteps to specify how asynchronous events should be handled in the face of out-of-order events.

For example, the line chart in Fig. 2 (3C) visualizes the distribution of fires by year. The query in (5D) fetches the distribution of fires based on the us_state selected: it combines the fires and selection event tables using the JOIN operator, and then filters the rows in fires by applying the filter on the selected us_state selected using the JOIN...ON operator. (5C) is a query that selects the year distribution results of the most recent interaction based on the timestep data. This ensures that the interface displays the correct data despite out of order responses.

4.2 DIEL Execution

DIEL orchestrates the local database and one or more remote databases in response to new events. Each event is evaluated *synchronously* in the local thread, corresponding to one *logical timestep*. As a result, the application-level effects of events with more recent logical timestamps are guaranteed to occur after events with earlier logical timestamps. From the perspective of the local database, user interactions and responses from remote databases are both events, however they are handeled slightly differently, as described in Fig. 3.

Interaction Event: The application creates new events from user interactions (1a), and inserts them into the corresponding EVENT TABLE using the NewEvent API call (1b). DIEL re-computes the local OUTPUT tables, and triggers the corresponding render function callbacks (1c). Finally, DIEL handles any EVENT VIEWs that depend on the new event. If it accesses remote data, DIEL first sends the appropriate query to the remote database and handles the response as described below (1d). Otherwise, DIEL executes the local EVENT VIEW query.

Data Response Event: Once the local DIEL runtime receives a response from a remote database (2a), which is tagged with the originating event's timestep, it calls NewEvent to insert it into the EVENT VIEW. Finally, the OUTPUTs are reevaluated, and the results are sent to the application rendering functions.

5 A Prototype Implementation of DIEL

To verify the feasibility of the DIEL model, we implemented a prototype. It ingests a DIEL specification and produces a distributed dataflow that is executed in conjunction with the frontend JavaScript libraries and backend databases. There are two main challenges in this process. First, queries over distributed data cannot be executed directly since the computation requires data to be co-located. Second, the DIEL model poses performance challenges.

An implementation should both create a dataflow such that necessary data is exchanged between the local and remote databases, and overcome these performance hurdles to keep the interface responsive. As such, DIEL builds the distributed dataflow in four phases. The first three phases address the challenge of distribution, and the last phase address the challenge of performance.

Two libraries manage these steps: (1) a local (browser-based) library updates the JavaScript state, and (2) a remote library configures access to the remote database. The developer provides these respective libraries with the connection configurations (e.g., Fig 2 4 and 6).

In the first step, DIEL parses and compiles the specification into an abstract syntax tree that captures the relational operations between tables. The tree diagrams in Fig. 2 2 (left) illustrates that of the running example. With the tree of operators, DIEL can now reason about the data exchange needed to co-locate tables for evaluation.

To reason about the exchange, DIEL next identifies the tables and their schema in the remote databases, i.e. a table catalog. Then, with the catalog, DIEL identifies queries that involve tables on both the local and remote databases. DIEL determines, for such queries, what data need to be exchanged between these databases. This process transforms the high-level logical specification into a distributed dataflow that can now be executed.

An example of such a transformation done by the current prototype is shown in Fig. 2 2 (right). DIEL decides to perform the evaluation of yearDistributionEvent on the remote database, thus requiring the selection table to be sent over the network to the remote database from the local database, and the result yearDistributionEvent to be sent over the network from the remote to the local database. The output yearDistribution is then evaluated within the local database upon each step. Note that this is not the only distributed dataflow possible. Alternative implementations of DIEL could instrument more advanced algorithms that can further optimize for performance by changing what data needs to be shared over the network. One example is to partition the execution of queries into more granular subqueries, and another is to leverage WebWorkers on the local client to parallelize compute.

In the final phase, DIEL optimizes the physical execution plan from the previous step. Four mechanisms are used.

1. Selective output invocation. Interactive dashboards often contain multiple visualizations, and sometimes ones with many visual elements that are expensive to render. A naive plan will reevaluate all the queries and re-render all the visual marks, which could block the main browser thread from responding to user events. To address this issue, the DIEL prototype analyzes the execution plan and builds a dependency graph of the DIEL queries. Using the dependency graph, DIEL selectively evaluates and invokes rendering functions for only the output views dependent on the current event.

This issue can be addressed via static analysis of DIEL queries at compile time. From the queries, we can extract dependencies across queries, views and tables syntactically. For example, in Fig. 2, there are two outputs—stateDistribution and yearDistribution. Looking closely at 2 2, note that selection "flows into" yearDistribution, but does *not* flow into stateDistribution. This reflects the syntax of those outputs in 2 5, where selection appears only in the definition of yearDistribution. Using this information, DIEL knows to selectively update yearDistribution when there is a new user interaction (selection).

2. Materialized intermediate views. One interaction can require updating multiple visualizations. Data transformations are sometimes shared when computing these updates. For instance, the filtered view in Fig. 5 is shared by two outputs. Since views are just named queries in SQL, this can lead to redundant work: whenever the database evaluates any query that references a view, the view is reevaluated as part of the query. This wasteful reevaluation can be prevented if the view results are materialized into a table for use by downstream queries [8]. For materialized views to work correctly, they must be updated appropriately when the

data they query changes. We tackle this problem pragmatically in our prototype: we implement update mechanisms for views in the our local database code; for remote databases, we count on their native support for materialized view maintenance.

3. Automatic caching. Developers often build a client-side cache by hand to amortize the compute and network time to a server. For any given specification, DIEL can automatically instrument this functionality, thus saving the developers' time. The server response data are already stored in the event log. To make use of the past values, DIEL analyzes the execution plan at this stage and instruments a layer of indirection to the evaluation of the `EVENT VIEW`s when a new event arrives: DIEL hashes the parameters of the relevant rows of the event tables and looks up the hash in an *event cache table* instrumented by DIEL. If the hash is present, DIEL returns that value, and if not, DIEL dispatches the dataflow. DIEL's automatic caching process saves storage by not having multiple copies of the responses, which is especially limited on the client. In addition to helping save the developers' time, DIEL's automatic caching process also alleviates the additional memory usage caused by DIEL's log of events.

4. Automatic Index Selection. Good database performance is typically tied to building indexes that suit your query workload. For a given DIEL program, this workload is known in advance as part of the spec, hence DIEL analyzes the query structure to statically determine indexes that will result in good performance.

Besides optimizing the dataflow, we also optimized the execution speed of the local database. Unlike the remote databases, the implementation of the local database is controlled by DIEL. The local database runs in the browser's main-thread and any delay would directly block user interactions and HTML updates. Therefore, it is critical that the local database executes quickly. We make use of a recent advance in browser technology, WebAssembly [54]. It provides fast execution of SQLite in the browser by compiling C code to WebAssembly.

The above highlights the most novel aspects of the prototype implementation. Other details, such as the full DIEL syntax specification, along with additional language features discussed in Section 8, are detailed in supplementary materials.

6 Evaluating DIEL's Expressivity

To assess the benefits of DIEL, we show examples of interactive visualizations that deal with the variety of challenges that arise when managing asynchrony and distribution, including coordinating responses from remote servers, streaming data, and composing interactions. Our discussion focuses on the aspects addressed by DIEL and omits implementation details of the frontend. We also show how DIEL specifications compose in a modular manner, by building each new interaction on the running example of Fig. 2. Since the challenges are independent of the particular choice of visual encodings, we do not focus on varying the visual designs.

Coordinating Requests and Responses. Latency from remote databases' responses can cause inconsistencies in the interface, if not handled properly [61]. For instance, Fig. 4 shows a timeline of user interactions (`selection`) and server responses (`yearDistributionEvent`). The user clicks `TX` and then `CA`, but the remote database responds with the result for `CA` first. Rendering the most recent result (`TX`) would surprise a user who is expecting `CA`. To avoid inconsistent interfaces, developers can use DIEL to specify a range of designs in a few lines of code. We walk through three possible designs shown in Fig. 4.

Option (A) always displays the result of the most recent interaction. The DIEL spec first identifies the interaction by selecting the rows with the highest timestep (`LATEST selection`), then selects the rows in `yearDistributionEvent` with matching request-response timesteps (`d.request_timestep = e.timestep`). If no match is available yet, nothing is returned. The front-end visualization logic could indicate as such, e.g., using a spinner as shown.

Option (B) always displays the most recent response and its corresponding selection, as well as pending selections. The DIEL spec first selects the most recent response (`LATEST yearDistributionEvent`), then joins it with the `selection` table to retrieve the corresponding value of the selection (`e.us_state`) by matching their timesteps. The second output `pending` represents pending selections and is computed by finding the request(s) that do not have a corresponding response.

Option (C) displays "snapshots" of all interactions [59], where past interactions and their results are scaled down into a scrolling pane of small multiples at the bottom. The DIEL spec simply selects all the interactions from the event table `selection`, joined with the corresponding responses by their timesteps. The snapshots allow a user to interact with the visualization and navigate to prior states, concurrent with the loading of new responses.

Without DIEL, implementing these designs would require the developer to manually keep track of events — store the points of interest selected, their respective responses, and the global ordering of all the events — and coordinate multiple event handlers. While each step is simple in isolation, put together the complexity of this low-level data-recording and event handling compounds substantially. Developers may have trouble reasoning about the *overall* design.

DIEL, on the other hand, encourages a consistent experience by asking the developer to specify which of the events should be in the output at any given time. There is no accidental design resulting from interrupt-driven event handling, such as immediately rendering whatever response arrives. Furthermore, DIEL takes care of recording event history and provenance. The developer can query these data directly — for example identifying the responses from remote databases with timestep data that DIEL automatically maintains. As a result, the developer can iterate on alternate designs without instrumentation overhead.

Streaming Data. Given the real-time nature of fires, the developer may want to incorporate streaming data into their visualization. Fig. 6 overlays the choropleth in Fig. 2 with a symbol map of active current fires. The event `incident` contains the location of the fire and whether it is new or contained. When a new `incident` event arrives, fires are either added to or removed from the symbol map overlay. To implement this design with DIEL, the event `incident` is captured as an event table and the data for the overlay is captured by the output table `fireMap`. Each tuple from the latter is used to query the `incident` table to identify fires that have not yet been controlled, via the `NOT IN` subquery. Note how the developer can rely on a few lines of DIEL code, instead of programming custom JavaScript functions to store and manipulate streaming events[2].

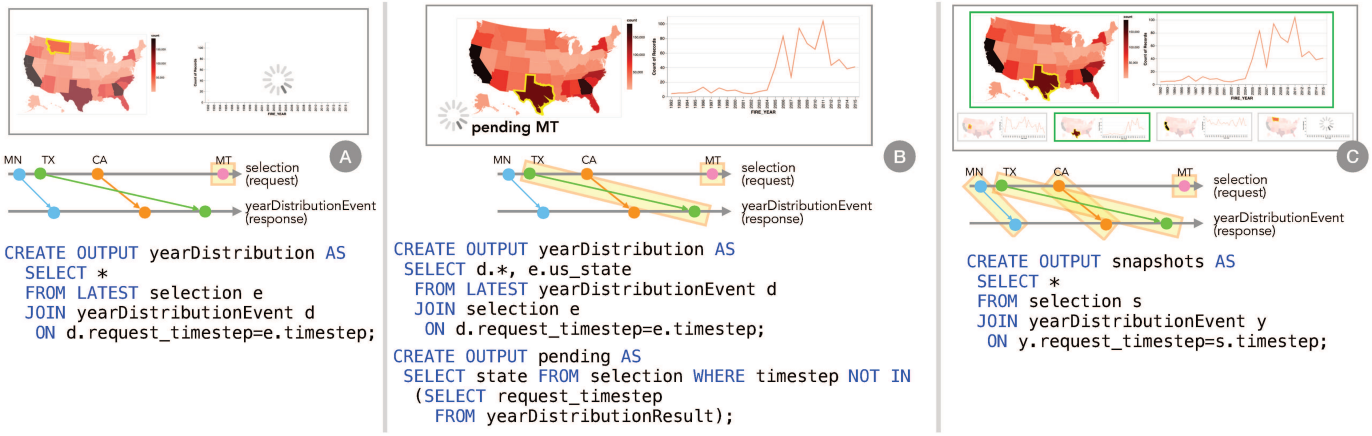
Composing Events: Interaction and Streaming. Cross-linking is a common interaction technique [48]. Fig. 7 shows a new brush selection added to the map visualization, linked to the bar chart (the backing query is not shown for brevity). `contained` checks if the `lat, lon` values in `fireMap` are within the `min` and `max` bounds of the brush. It is a utility function defined by DIEL, using the *user-defined function* (UDF) construct in SQL [40]. UDFs are supported by the frontend database library we use, `sql.js` [28], and developers can define UDFs through the DIEL runtime API, `AddUDF`.

The new interaction composes with the streaming `firemap` view from before — if there is a new event received that falls into the brushed area, the incidents selected in `brushedIncidents` will also be updated, and any dependent output views will be updated as well. This subtle instrumentation, automatically performed by DIEL, may be difficult for a developer to catch in a traditional implementation where the logic may be dispersed into different event handlers.

Composing Events: Brushing and Panning. Different interactions serve different purposes and more than one interaction could be employed for the same visualization, which the developer may need to coordinate. Following the running example, suppose now the developer wishes to introduce a pan-zoom interaction, shown in Fig. 8. The `brush` table is defined in geographic coordinates, so the user can pan the map to an area where the brush is no longer visible, and the value of the selection is in question. We present two possible designs to address this ambiguity. Both derives a new brush, `effectiveBrush` for use in place of the raw `brush`.

Option (A) invalidates the brush when the user initiates a new panning interaction. The DIEL spec selects the most recent brush (`LATEST`

[2] As with materialized views in Section 5, we take a pragmatic approach in our prototype: we handle streams in the client database, but do not support streaming in the remote databases.



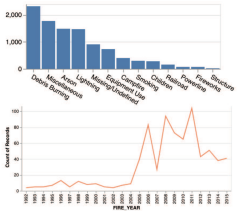

```
CREATE OUTPUT yearDistribution AS
  SELECT *
  FROM LATEST selection e
  JOIN yearDistributionEvent d
   ON d.request_timestep=e.timestep;
```

```
CREATE OUTPUT yearDistribution AS
 SELECT d.*, e.us_state
  FROM LATEST yearDistributionEvent d
  JOIN selection e
   ON d.request_timestep=e.timestep;
CREATE OUTPUT pending AS
 SELECT state FROM selection WHERE timestep NOT IN
  (SELECT request_timestep
    FROM yearDistributionResult);
```

```
CREATE OUTPUT snapshots AS
 SELECT *
 FROM selection s
 JOIN yearDistributionEvent y
  ON y.request_timestep=s.timestep;
```

Fig. 4: Designs to coordinate asynchronous requests and responses when querying over distributed data: (A) renders the most recent interaction requested; (B) renders the most recent response received as well as any *pending interactions*; (C) renders snapshots of all interactions and their corresponding results [59].

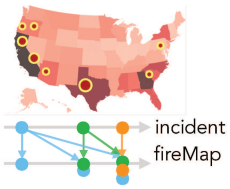
```
CREATE VIEW filtered AS
 FROM fires f
  JOIN LATEST selection s ON f.state = s.us_state;
```



```
CREATE EVENT VIEW
 causeDistributionEvent AS
 SELECT cause, count()
   FROM filtered GROUP BY cause;
```

```
CREATE EVENT VIEW
 yearDistributionEvent AS
 SELECT year, count()
   FROM filtered GROUP BY year;
```

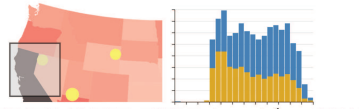
Fig. 5: An example of reuse : the filtering logic based on the selection in Fig. 2, `filtered`, is shared by two visualizations. DIEL analyzes the query plan and materializes `filtered` to avoid evaluating it twice.




```
CREATE EVENT
TABLE incident(
 id TXT,
 type TXT,
 lat FLOAT,
 lon FLOAT,
 time INT);
```

```
CREATE OUTPUT fireMap AS
SELECT f.lat, f.lon
FROM incident f
WHERE type='new'
 AND f.id NOT IN (
 SELECT f.id
 FROM incident
 WHERE type='contained');
```

Fig. 6: Example DIEL spec for the symbol overlay of active fires, determined by selecting `incidents` that do not yet have a row with the column `type` of `contained`.




```
CREATE EVENT TABLE brush(
  min_lat FLOAT,
  max_lat FLOAT,
  min_lon FLOAT,
  max_lon FLOAT);
```

```
CREATE VIEW brushedIncidents
AS SELECT m.id
   FROM fireMap m
   JOIN LATEST brush b
    ON contained(m.*, b.*)
```

Fig. 7: Example DIEL spec for brushing interaction: `brushedIncidents` selects fires in the symbol map `fireMap` that fall into the brushed region `pan` to update the bar chart (query omitted). The user-defined function `contained` helps make the code more concise.

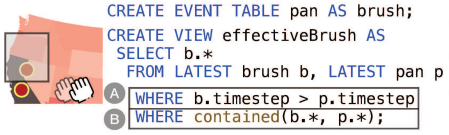


Fig. 8: Example DIEL specification of composing two interactions: panning and brushing (from Fig. 7). There are two ways to coordinate: (A) removes the brush selection whenever there is a more recent panning event, and (B) removes the brush only when it is panned out of view.

brush) only if it occurred *after* the most recent pan (`LATEST pan`). If we replace option (A) with option (B), we instead invalidate the brush only when a new panning event moves the brush *out of view*. In either case, developers do not manually modify the callbacks to the panning interaction handler to check and remove the previous brush selection. The "removal" is specified declaratively and enforced implicitly by DIEL as the logical timesteps progress.

Interaction Scent. Research has shown the benefit of visualizing interaction history to provide a visible trail ("scent") of prior interactions. *HindSight* visualizes the user's prior interactions in the visualization [14]; *Scented Widgets* visualize interactions by other users [57]; *Interaction Snapshots* visualize historical interactions and their results [59]. We have shown an example of *Interaction Snapshots* in Fig. 4; we now discuss the other two designs in Fig. 9.

(A) instantiates a *HindSight* design, where all prior brushes are shown [14]. The DIEL spec for `brushScent` selects unique brushes through the `UNIQUE` operator. (B) shows an example design of *scented widgets*, where a histogram of all prior `us_state` selections are aggregated and counted [57]. The `selection_all` is a table in the database that stores all the event tables from each session, which is saved by the developer from event tables to the remote databases (and not handled automatically by the current DIEL prototype), e.g., `INSERT INTO selection_all SELECT * FROM brush`.

With DIEL, the data backing these visualizations is already in event tables ready to be queried, and the visualization scents are automatically updated. Without DIEL, the developer would have to manually store the events and re-render the visualization scents after each interaction, or use libraries like *Trrack* [12].

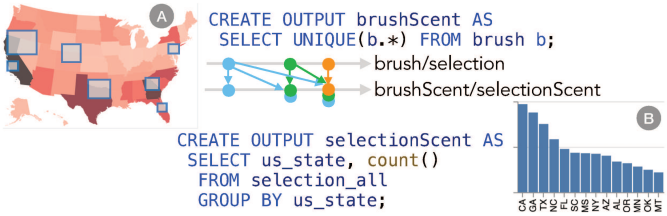


Fig. 9: Example DIEL spec for (A) Hindsight: select all unique brushes in the `brush` event table, and (B) Scented Widget: a bar chart that shows the frequency of selections across users.

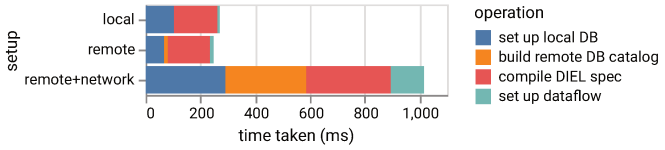


Fig. 10: The x-axis represents the respective time taken for DIEL's initialization steps, in milliseconds, and the y-axis represents the type of setup being measured.

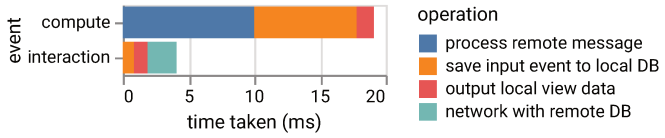


Fig. 11: The x-axis represents the time taken for the event handling steps, and the y-axis the type of event being measured. These results are evaluated against both the *remote+network* and the *remote* only setup. The numbers follow the same distribution because the tick performance is decoupled from the database query processing performance. The steps are all *local* evaluations after receiving the asynchronous events . The actual latencies for each individual response are depicted in Fig. 12.

7 Evaluating DIEL's Performance

We evaluate the feasibility of the DIEL model using our prototype implementation. We focus on the overhead that the DIEL middleware introduces during initial setup and user interactions, as well as its ability to scale to large datasets as compared to leading visualization libraries.

We used Kaggle's wildfire dataset [22], creating the visualization shown in Fig. 2, and conducted evaluations on a MacBook Pro with 2.7 GHz Quad-Core Intel Core i7 and 16 GB of memory. In the experiments, the "remote" database server is a SQLite process running on a web application deployed on Heroku ("Free Dynos" tier with 512 MB of RAM). The network bandwidth was 18.5 Mbps. Changing the dataset size or database will not affect the overhead that the DIEL middle-ware incurs. Thus we chose to focus on the above evaluation configurations.

Initialization. Fig. 10 shows the time taken to setup DIEL. The *local* setup only involves a local database in the main browser thread. DIEL (1) sets up the synchronous database in the main thread; (2) compiles the DIEL spec into logical representations, including optimization steps such as caching; and, (3) sets up the views in the local database. The *remote* setup accesses the remote database server. DIEL shares step (1) and (2) of the local setup, then it builds a *catalog* of tables in remote databases and sets up a distributed dataflow based on the catalog. The *remote+network* results are the same as *remote*, but with the addition of nontrivial network communication overheads. These introduce fixed costs—such as the 200ms overhead to fetch the SQL.js library in order to set up the local database—as well as variable costs to send results between the client and server. These costs depend on the network bandwidth. For instance, decreasing the network bandwidth to 1.4 Mbps caused the set up local database to be 8687ms.

In all three cases, we see that the initialization time does not pose a usability challenge, especially given that the initiation is a one time cost at the beginning of loading the web page containing the visualizations.

Event Handling. Fig. 11 shows the time taken to handle events. One type of event is user interaction, which took less than 5 ms to handle. During that time, DIEL (1) saves the input event to the local database (serializing from JavaScript to SQLite); (2) outputs the view data (deserializing from SQLite to JavaScript); and, (3) networks with the remote database. Another type of event is a remote database response, which takes about 20 ms to handle. During that time, DIEL (1) processes the remote message; (2) saves the input event to the local database; and, (3) outputs the view data. Processing the data from the remote database and serializing that data into the local database takes the bulk of the time. We see that the overhead in either case is well under the limit of 100 msec prescribed by Card et al. to sustain perceptual causality [7].

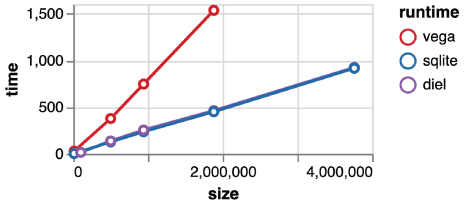


Fig. 12: A visualization of how long the query processing takes as the data grows in size. One program is implemented with Vega, another with DIEL used in conjunction with a *remote* SQLite database, whose raw query time is also plotted. DIEL is able to make use of the backend database and process data beyond the capacity of the browser, with less than 1s of additional overhead for (de)serialization when working with the remote database.

Different Data Sizes. To demonstrate a DIEL program's ability to scale, we created samples from 10 thousand rows to 4 million rows and benchmarked two implementations of the visualization shown in Fig. 2: one with DIEL (the SQLite server and static Vega visualizations) and one with Reactive Vega running in the browser. We measured the time taken between the handler receiving the interaction and outputting the computed result. This excludes the rendering logic, which is common across the experimental runs. The SQLite and DIEL results also exclude the network latency since it is shared in both cases.

Fig. 12 shows that DIEL is able to handle the increasingly large data by leveraging resources beyond the client, whereas a client-only Reactive Vega application freezes the browser at 4 million rows of data. We also measured the query evaluation time directly against the remote SQLite database — DIEL's processing time follows closely with negligible serialization/serialization overhead. Additionally, the DIEL implementation benefits from a *non-blocking* design because it does not take up costly resources on the main thread.

While this study does not exhaustively analyze DIEL's performance, we believe it addresses DIEL's core motivations of usability, expressivity and scalability via remote data servers.

8 A Discussion of DIEL's Usability

We evaluate the programming experience of DIEL using the *Cognitive Dimensions of Notation* [5], a set of considerations to evaluate the effectiveness of notational systems. Cognitive dimensions have been used by prior visualization systems to evaluate their usability [42, 43, 46, 63]. We pick a relevant subset and contrast the effectiveness of DIEL against that of current common practices.

Viscosity *(resistance to change).* DIEL follows relational database abstractions and benefits from its ease of change. First, its declarative query specification abstracts *what* to compute from *how* to compute (known as "physical data independence" in database literature). As a result, switching from a client-only application to a client-server one (or to one using WebWorkers) requires only a few lines of change to configuration details. Fig. 2(4E) provides an example of the configuration, `remoteDbConfigs`, a JavaScript object containing high-level specification such as the type of the database used and what socket to connect to the DIEL database wrapper. Query changes are moreover isolated from table changes due to the "logical data independence" property of the relational model [40]. For instance, if the `incident` schema changes because the data provider now also includes the reporting station, none of the downstream queries would need to change. Finally, views also provide a way to reuse logic within an application. Fig. 5 shows an example where a change to the internal specification of the query `filtered` would be abstracted away from the dependent views as long as its `SELECT` clause is unchanged. These properties combine to make more complex interactive visualizations easier to author, since multiple components could be reused and changed easily. Consider the running example of visualizations over the wildfire data: they could be combined easily into one visualization dashboard.

Closeness of Mapping *(closeness of representation to domain).* Since SQL is widely used in modern databases and for general data manipulation tasks [50], DIEL closely represents the data processing domain.

However, DIEL does not fully represent the complex domain of data visualizations. For example, the Vega authors identify that visualizations involving small multiples often require hierarchical structures with second-order quantification [45].

We agree that the expressiveness of SQL, and hence DIEL, is more limited than interactive visualization frameworks, which are implemented in turing complete languages. Relational languages like DIEL express only first-order logic. In particular, DIEL can define and quantify relationships between entities, but cannot quantify over a data-dependent set of table names or column names. Yet this limitation is key for distributed execution and optimization. Neither physical data independence nor the rich optimization methods discussed in Section 3 would be easy to implement without the relational abstraction. In this sense, any alternative approach towards distributed execution at scale will end up using the same principles and hence facing similar limitations. Moreover, developers could transform the data into other forms by manipulating the output tables provided by DIEL in any JavaScript functions. For instance, they can use Vega to transform the tabular data into nested groups to render small multiples. These data transformations happen at the end of DIEL's dataflow and does not diminish the effectiveness of DIEL's ability to orchestrate query evaluations across the client and remote databases.

Consistency *(similar semantics are expressed in similar syntactic forms).* In DIEL, the only data structure is a table. One of the key contributions of DIEL's model is to unify both "live" events and "stored" data in a single frame of reference—tables—which store both data and history. Once events are reified as data in an event table, DIEL presents a unified data-centric language.

Premature Commitment *(constraints on the order of doing things).* For DIEL to be effective, it imposes a premature commitment. Developers must represent the state of visualizations using tables (rather than arbitrary data structures) upfront. This premature commitment can hamper a rapid prototyping process, but we believe the advantages of the table format outweigh these concerns. As discussed in Section 3, the table format facilitates working with distributed data, and makes explicit possibly concurrent processes.

Role-expressiveness *(the purpose of an entity is readily inferred).* DIEL reuses an existing, well-established data model of relational tables and queries, and introduces only two additional constructs: `EVENT`s and `OUTPUT`s. This approach has proven sufficiently expressive, as demonstrated by the examples in Section 6. DIEL operates as a middleware layer between the frontend and backends, and lifts the logic of data exchange between client and remotes. The relatively small surface area of DIEL's abstractions stands in contrast to the existing imperative code written to support such use cases, which is often dispersed across custom functions with a commensurate burden of role-expressiveness.

Hidden Dependencies *(important links between entities are not visible).* DIEL makes dependencies quite explicit. The only type of dependency DIEL introduces is between tables, which are syntactically evident in queries: the table a query creates is dependent on the tables it references. In contrast, current imperative practice distributes dependencies across different functions, each with custom logic and bookkeeping formats that require additional effort to navigate and make sense of.

Hard Mental Operations *(high demand on cognitive resources).* There are two potentially challenging programming tasks in DIEL. One challenge is debugging in SQL [16]. Consider the case where the developer is debugging a view *O* which involves both *V1* and *V2* views; they need to inspect *both* of the views to locate the error. To address this challenge, we built *view level constraints*, similar to SQL table constraints [40], so that developers could make assertions on intermediate queries. For instance, if *O* is unexpectedly empty, the developer could assert `V1 NOT EMPTY` and `V2 NOT EMPTY` respectively to pin-point the error as they arise. Another challenge is not being able to mutate state. It could be challenging to define the state of the visualization with only raw events, especially when the logic is more complex (e.g., *undo-redo*). To help, we took a page from the construct of *state programs* in *relational transducers* [1], which allow developers to maintain derived state by inserting values into tables after events.

Diffuseness *(verbosity of language).* The current DIEL syntax hews close to SQL, and as such does not have syntactic conveniences one might like for visualization (e.g., *binning* [24]). Some aspects of DIEL's current verbosity can be alleviated by introducing syntactic sugar for common operations. Through our own experience working with DIEL and analyzing code snippets, we identified the most common programming patterns and implemented a handful of syntactic sugars. For instance, `LATEST` selects the most recent event (i.e. row(s) with the highest timestep). Similarly, the *default* asynchrony policy for output views over distributed data creates an event table for the developer and selects the response for the most recent interaction. We provide additional details in the supplement.

9 Conclusion and Future Work

By adapting two key ideas from distributed systems programming — immutable events and logical constraints — DIEL contributes a substantive step towards declarative programming over distributed data and asynchronous events for interactive visualization. Through examples, we demonstrate that developers can use DIEL to declaratively specify a variety of emerging interactive visualization use cases. And, to assess the challenge that DIEL's use of a relational language poses to developers, we conducted a heuristic evaluating using the Cognitive Dimensions of Notation [5]. We find that although DIEL introduces premature commitment and possible hard mental operations, these disadvantages are outweighed by a decrease in viscosity when working with distributed data and asynchronous events. Moreover, as our performance benchmarks suggest, this declarative model allows DIEL to reason about the specification and optimize the execution plan.

As data grows in size and computation grows in complexity, optimizing the performance of interactive visualization application is a hot topic. DIEL's unique middleware architecture that spans the local and remotes allows for a number of research opportunities. To start, operator-level materialized-view maintenance techniques [8] can make the frontend database even faster. *Federated* databases that optimize globally across multiple databases [13] can help us optimize data exchange between the local database and remote database. Another possibility is to automatically parallelize query evaluation [40] across multiple threads of computation, e.g. multiple WebWorkers in a browser. Finally, we can enhance the performance of each timestep with "garbage collection" by removing rows that are no longer in use from logs. This pattern is common in many areas, such as in replicated database systems [41], multi-version concurrency control [4] and distributed systems programming [10].

In terms of usability, having SQL as the host language has both advantages and disadvantages (discussed in Section 8). To reduce the disadvantages of expressibility, future iterations of DIEL may benefit from using a syntax closer to dataframe libraries like `pandas` that are better integrated with JavaScript [58].

In terms of functionality, DIEL does not yet support an important distributed use case, *collaborative interactive visualizations* (Fig 13). Coordinating communication between multiple users is a classic challenge in distributed systems and CSCW [52]. A global order of events across multiple editors cannot be guaranteed without explicit coordination that decreases the interface's responsiveness. Instead, various coordination-free proposals have emerged that use more involved metadata than simple local timesteps to provide distributed consistency guarantees [11, 55]. Supporting change from multiple locations would also unlock the support for streaming, since a change from the remote database could then drive change on the client. It would be interesting to extend DIEL with ideas from this work.

DIEL is an open source system available at `https://github.com/yifanwu/diel`.

Acknowledgments

We thank Ryan Purpura and Lucie Choi for their contributions as undergraduate researchers at UC Berkeley; Michael Whittaker for his feedback on methods to perform the distributed execution. The work is supported by NSF 1564049, 1564351, 1942659, 1845638, 2106197, 1452977, 1939945, 1940175, and CCF-1730628.

REFERENCES

[1] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Journal of Computer and System Sciences*. Elsevier, 2000.

[2] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in Bloom: a Calm and collected approach. In *CIDR*, 2011.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *SIMOD*, 2015.

[4] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. In *ACM Computing Surveys*. ACM New York, NY, USA, 1981.

[5] A. F. Blackwell, C. Britton, A. Cox, T. R. Green, C. Gurr, G. Kadoda, M. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, et al. Cognitive dimensions of notations: Design tools for cognitive technology. In *Cognitive technology: instruments of mind*. Springer, 2001.

[6] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. In *TVCG*. IEEE, 2011.

[7] S. Card, T. Moran, and A. Newell. The model human processor- an engineering model of human performance. In *Handbook of perception and human performance.*, 1986.

[8] R. Chirkova, J. Yang, et al. Materialized views. In *Foundations and Trends in Databases*. Now Publishers, Inc., 2012.

[9] E. F. Codd. A relational model of data for large shared data banks. In *Communications of the ACM*. ACM, 1970.

[10] N. Conway, P. Alvaro, E. Andrews, and J. M. Hellerstein. Edelweiss: Automatic storage reclamation for distributed programming. In *pVLDB*. VLDB Endowment, 2014.

[11] N. R. G. Conway. *Language Support for Loosely Consistent Distributed Programming*. PhD thesis, 2014.

[12] Z. T. Cutler, K. Gadhave, and A. Lex. Trrack: A library for provenance tracking in web-based visualizations. In *OSF Preprints*, 2020.

[13] A. Deshpande and J. M. Hellerstein. Decoupled query optimization for federated database systems. In *ICDE*, 2002.

[14] M. Feng, C. Deng, E. M. Peck, and L. Harrison. Hindsight: Encouraging exploration through direct encoding of personal interaction history. In *TVCG*. IEEE, 2017.

[15] Y. Fu, K. W. Ong, Y. Papakonstantinou, and E. Zamora. Forward: Data-centric uis using declarative templates that efficiently wrap third-party javascript components. In *pVLDB*. VLDB Endowment, 2014.

[16] S. Gathani, P. Lim, and L. Battle. Debugging database queries: A survey of tools, techniques, and users. In *CHI*, 2020.

[17] T. R. Green. Instructions and descriptions: Some cognitive aspects of programming and similar activities. In *AVI*, 2000.

[18] J. Heer and M. Bostock. Declarative language design for interactive visualization. In *TVCG*. IEEE, 2010.

[19] P. Helland. Immutability changes everything. In *Queue*. ACM, 2015.

[20] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. In *Human Computer Interaction*. Taylor & Francis, 1985.

[21] J. Johnson. *Gui Bloopers 2.0: Common User Interface Design Don'Ts and Dos*. Morgan Kaufmann, 2007.

[22] Kaggle. 188 million us wildfires.

[23] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, 2011.

[24] T. Kraska. Northstar: An interactive data science system. In *pVLDB*. VLDB Endowment, 2018.

[25] M. Krstajic and D. A. Keim. Visualization of streaming data: Observing change and context in information visualization techniques. In *IEEE Big Data*, 2013.

[26] Z. Liu, B. Jiang, and J. Heer. Immens: Real-time visual querying of big data. In *Computer Graphics Forum*, 2013.

[27] O. Lojkine. Sqlite compiled to javascript through emscripten. https://github.com/kripken/sql.js/, 2017.

[28] O. Lojkine. sql.js api (including how to add user defined functions). https://github.com/sql-js/sql.js/blob/master/src/api.js, 2021.

[29] E. Meijer. Your mouse is a database. In *Communications of the ACM*. ACM, 2012.

[30] D. Moritz, D. Fisher, B. Ding, and C. Wang. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *CHI*, 2017.

[31] D. Moritz, B. Howe, and J. Heer. Falcon: Balancing interactive latency and resolution sensitivity for scalable linked visualizations. In *OSF Preprints*, 2019.

[32] Mozilla. Using web workers, 2018.

[33] R. Neogy, J. Zong, and A. Satyanarayan. Representing real-time multi-user collaboration in visualizations. In *arXiv*, 2020.

[34] C. Olston, M. Stonebraker, A. Aiken, and J. M. Hellerstein. Viqing: Visual interactive querying. In *Visual Languages*, 1998.

[35] PosgreSQL. Postgresql: The world's most advanced open source relational database. https://www.postgresql.org/, 2019.

[36] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *ICDCS*, 2009.

[37] V. Project. Vega transforms. `https://vega.github.io/vega/docs/transforms/`.

[38] F. Psallidas and E. Wu. Provenance for interactive visualizations. *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, 2018.

[39] F. Psallidas and E. Wu. Smoke: Fine-grained lineage at interactive speed. In *pVLDB*. VLDB Endowment, 2018.

[40] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw Hill, 2000.

[41] S. K. Sarin and N. A. Lynch. Discarding obsolete information in a replicated database system. In *TSE*. IEEE, 1987.

[42] A. Satyanarayan and J. Heer. Lyra: An interactive visualization design environment. In *Computer Graphics Forum*, 2014.

[43] A. Satyanarayan, B. Lee, D. Ren, J. Heer, J. Stasko, J. Thompson, M. Brehmer, and Z. Liu. Critical reflections on visualization authoring systems. In *TVCG*. IEEE, 2019.

[44] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. In *TVCG*. IEEE, 2017.

[45] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. In *TVCG*. IEEE, 2016.

[46] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative interaction design for data visualization. In *UIST*, 2014.

[47] C. Savery and T. Graham. It's about time: Confronting latency in the development of groupware systems. In *CSCW*, 2011.

[48] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *The Craft of Information Visualization*. Elsevier, 2003.

[49] SQLite. Small, fast, self-contained, high-reliability, full-featured, sql database engine. https://www.sqlite.org, 2017.

[50] StackOverflow. Developer survey results, most popular programming languages. https://insights.stackoverflow.com/survey/2019, 2020.

[51] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. In *TVCG*. IEEE, 2002.

[52] C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *CSCW*, 1998.

[53] F. Wanner, A. Stoffel, D. Jäckle, B. C. Kwon, A. Weiler, D. A. Keim, K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, et al. State-of-the-art report of visual analysis for event detection in text data streams. In *Computer Graphics Forum*, 2014.

[54] WebAssembly. Webassembly, a binary instruction format for a stack-based virtual machine. `https://webassembly.org/`.

[55] S. Weiss, P. Urso, and P. Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *ICDCS*, 2009.

[56] L. Wilkinson. *The Grammar Of Graphics*. Springer Science & Business Media, 2006.

[57] W. Willett, J. Heer, and M. Agrawala. Scented widgets: Improving navigation cues with embedded visualizations. In *TVCG*. IEEE, 2007.

[58] Y. Wu. Is a dataframe just a table? In *PLATEAU*, 2020.

[59] Y. Wu, R. Chang, J. Hellerstein, and E. Wu. Facilitating exploration with interaction snapshots under high latency. In *VIS*, 2020.

[60] Y. Wu, J. M. Hellerstein, and A. Satyanarayan. B2: Bridging code and interactive visualization in computational notebooks. In *UIST*, 2020.

[61] Y. Wu, J. M. Hellerstein, and E. Wu. A devil-ish approach to inconsistency in interactive visualizations. In *HILDA Workshop at SIGMOD*, 2016.

[62] Z. Xie. Towards exploratory visualization of multivariate streaming data. In *VIS Doctoral Colloquium*, 2007.

[63] J. Zong, D. Barnwal, R. Neogy, and A. Satyanarayan. Lyra 2: Designing interactive visualizations by demonstration. In *TVCG*. IEEE, 2020.

[64] Zoomdata. Analyze the freshest data. `https://www.zoomdata.com/resource/streaming-data-why-it-makes-sense-and-how-work-it`, 2018.