Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization

Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer

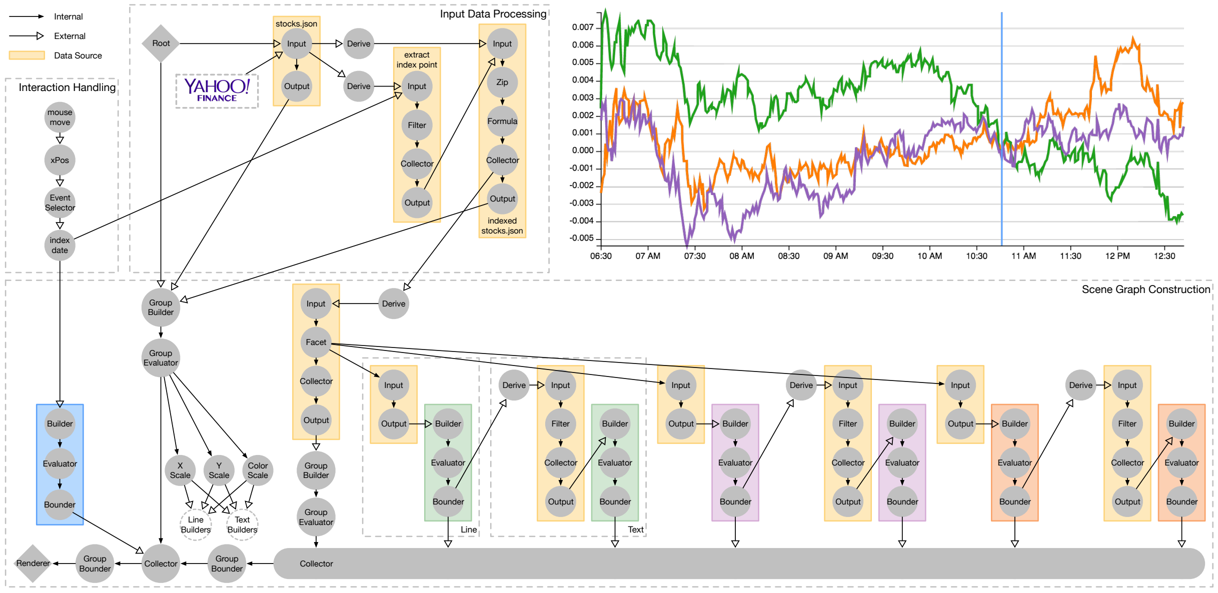


Fig. 1. The Reactive Vega dataflow graph created from a declarative specification for a interactive index chart of streaming financial data. As streaming data arrives from the Yahoo! Finance API, or as a user moves their mouse pointer across the chart, an update cycle propagates through the graph and triggers an efficient update and re-render of the visualization.

Abstract—We present Reactive Vega, a system architecture that provides the first robust and comprehensive treatment of declarative visual and interaction design for data visualization. Starting from a single declarative specification, Reactive Vega constructs a dataflow graph in which input data, scene graph elements, and interaction events are all treated as first-class streaming data sources. To support expressive interactive visualizations that may involve time-varying scalar, relational, or hierarchical data, Reactive Vega's dataflow graph can dynamically re-write itself at runtime by extending or pruning branches in a data-driven fashion. We discuss both compile- and run-time optimizations applied within Reactive Vega, and share the results of benchmark studies that indicate superior interactive performance to both D3 and the original, non-reactive Vega system.

Index Terms—Information visualization, systems, toolkits, declarative specification, optimization, interaction, streaming data

1 Introduction

Declarative languages such as D3 [10], ggplot2 [38] and Vega [35] have become popular tools for authoring visualizations. By deferring control flow and execution concerns to the runtime, they free designers to focus on visual encoding decisions. The separation of specification and execution can also facilitate retargeting across platforms [20] and enable programmatic generation of visualizations in graphical design tools [31], statistical packages [18] and computational notebooks [21].

Although interaction is a critical aspect of effective data visualization [26, 30], existing languages lack support for declarative interaction design. Our recent work [32] closes this gap through composable interaction primitives that model input events as first-class streaming data. As a result, user input can be processed through the full range of data transformation operators and participate in visual encoding rules. The primitives are grounded in Event-Driven Functional Reactive Programming (E-FRP) [36] semantics to shift responsibility for coordinating event-driven state changes from the designer to the language runtime. However, we developed only a proof-of-concept system sufficient for demonstrating the viability of declarative interaction design.

Here, we contribute Reactive Vega, the first system architecture to provide robust and comprehensive support for declarative, interactive visualization design. Our design is motivated by four primary goals.

A Unified Data Model. Existing reactive visualization toolkits [22, 32] feature fragmented architectures where only interaction events are modeled as time-varying. Other input datasets remain static and batch-processed. This artificial disconnect restricts expressivity and can result in wasteful computation. For example, interaction events that manipulate only a subset of input tuples may trigger recomputation over the entire dataset. In contrast, Reactive Vega features a unified data model in which input data, scene graph elements, and interaction events are all treated as first-class streaming data sources.

Streaming Relational Data. Modeling input relational data with E-FRP semantics alone does not supply sufficient granularity for targeted recomputation. As E-FRP semantics consider only time-varying

• *Arvind Satyanarayan is with Stanford University. E-mail: arvindsatya@cs.stanford.edu.*
• *Ryan Russell, Jane Hoffwell, and Jeffrey Heer are with the University of Washington. E-mails: {ryan16, jhoffs, jheer}@uw.edu.*

Manuscript received 31 Mar. 2015; accepted 1 Aug. 2015; date of publication xx Aug 2015; date of current version 25 Oct. 2015. For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

scalar values, operators would observe an entire relation as having changed and so would need to reprocess all tuples. Instead, Reactive Vega integrates techniques from streaming databases [1, 2, 4, 5, 12] alongside E-FRP, including tracking state at the tuple-level and only propagating modified tuples through the dataflow graph.

Streaming Nested Data. Interactive visualizations, particularly those involving small multiples, often require hierarchical structures. Processing such data poses an additional challenge not faced by prior reactive or streaming database systems. To support streaming nested data, Reactive Vega's dataflow graph rewrites itself in a data-driven fashion at runtime: new branches are extended, or existing branches pruned, in response to observed hierarchies. Each dataflow branch models its corresponding part of the hierarchy as a standard relation, enabling operators to remain agnostic to higher-level structure.

Interactive Performance. Reactive Vega performs both compile- and run-time optimizations to increase throughput and reduce memory footprint, including tracking metadata to prune unnecessary computation, and optimizing scheduling by inlining linear chains of operators.

Reactive Vega offers composable primitives for both visual encoding and interaction, and enables portability of rendering and interaction modalities across devices. We demonstrate these advantages through a variety of example applications. In addition, we conduct benchmark evaluations of streaming and interactive visualizations and find that Reactive Vega meets or exceeds the performance of both D3 and the original, unreactive Vega system.

2 Related Work

Reactive Vega draws on prior work in functional reactive programming, data stream management, and visualization systems. We defer discussion of declarative visualization tools to the subsequent section.

2.1 Functional Reactive Programming

Functional Reactive Programming (FRP) models mutable values as continuous, time-varying data streams [6]. We focus on a discrete variant called Event-Driven FRP (E-FRP) [36]. To capture value changes as they occur, E-FRP provides *streams*, which are infinite time-ordered sequences of discrete events. Streams can be composed into *signals* to build expressions that react to events. The E-FRP runtime constructs the necessary dataflow graph such that, when a new event fires, it propagates to corresponding streams. Dependent signals are evaluated in a two-phase update: signals reevaluated in the first phase use prior computed values of their dependents, which are subsequently updated in the second phase. E-FRP has been shown to be viable for authoring interactive web applications [16, 28] and visualizations [14, 22, 32].

However, naive applications of E-FRP to visualization tasks can result in wasteful recomputation. Traditional E-FRP primitives support only *scalar* values, whereas visualization pipelines must also process relational and hierarchical data. Modeling these latter data types as scalar values provides insufficient granularity to perform targeted recomputation. Reactive Vega's declarative interaction primitives remain grounded in E-FRP semantics, and they preserve the two-phase update: interdependent signals are updated in the order in which they are defined in the specification. However, to efficiently support relational data, Reactive Vega integrates methods from the streaming database literature. To support streaming hierarchical data, Reactive Vega's dataflow graph dynamically rewrites itself at runtime, instantiating new branches to process nested relations.

2.2 Data Stream Management

The problem of managing streaming data has been well studied in the database community. Researchers have developed an arsenal of techniques through the development of systems such as Aurora [2], Eddies [5], STREAM [4], and TelegraphCQ [12]. As tuples are observed by these systems, they are flagged as either new or removed. Tuples, rather than full relations, are passed between operators in a query plan (realized as a dataflow graph). As a result, operators can inspect just the updated tuples to perform efficient computation. However, for some operations a set of changed tuples is insufficient. For example, a join of two relations requires access to all tuples within a specified window. In such cases, caches (sometimes referred to as *views* [2] or *synopses* [4]) are used to materialize a relation, and shared among dependent operators.

Borealis [1] extends this work in two ways. To support streaming modifications to tuples, the system introduces a revision processing scheme. An operator can be *replayed* with revised tuples in place of the original data; the operator will then only emit corresponding revisions. Similarly, to enable dynamic operator parameters, Borealis introduces *time travel*. When an operator parameter changes, an `undo` is issued to the nearest cache. The cache emits tuple deletions, effectively "rewinding" the system to a previous time. A subsequent `replay` then performs recomputation with the new parameter value.

However, existing streaming data systems concern flat relations. Reactive Vega instantiates these techniques, alongside E-FRP, within a visualization pipeline and extends them to support streaming nested data. To do so, Reactive Vega's dataflow graph dynamically rewrites itself at runtime with new branches. These branches unpack nested relations, enabling downstream operators to remain agnostic to higher-level structure while supporting arbitrary levels of nesting.

2.3 Imperative and Dataflow Visualization Systems

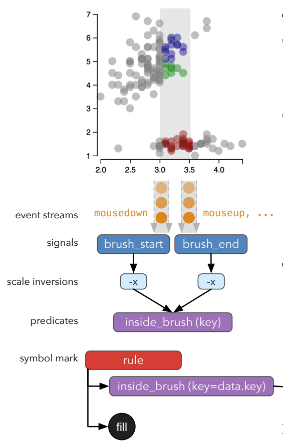
Dataflow architectures are common in scientific visualization systems, such as IBM Data Explorer [3] and VTK [33]. Developers must manually specify and connect each required operator into a network, which can support updates in a demand-driven fashion (e.g., as data is modified) or an event-driven fashion (e.g., in response to user input). These systems expose fine-grained control over the construction of the dataflow graph. For example, VTK developers can choose to favor memory efficiency over processing speed, which causes dataflow operators to delete their output after computation. While Reactive Vega shares some dataflow strategies with these systems — for example, using pass-by-reference for unchanged tuples to reduce memory consumption — it abstracts such execution concerns away from the user. The dataflow graph is automatically assembled based on definitions found in a declarative Vega specification, and optimizations are transparently performed such that output data is only stored when needed by downstream operators and shared wherever possible.

Within the domain of information visualization, the Stencil language [14] is also grounded in FRP and uses a dataflow model. Like Reactive Vega, it provides a unified data model where both input data and interaction events are modeled as first-class streaming data sources. However, Reactive Vega is more expressive than Stencil in two important ways. Building on prior work [32], Reactive Vega offers interaction primitives which enable fine-grained manipulation that event streams alone lack. Moreover, graphical primitives can be arbitrarily nested with Reactive Vega, drawing from either hierarchical or distinct data sources. This ability is critical to concisely specifying small multiples displays, and requires Reactive Vega's dataflow graph to dynamically rewrite itself at runtime. To the best of our knowledge, Stencil's architecture does not support self-instantiating dataflows.

Improvise [37] features active variables called "live properties," which may be bound to control widgets and parameterize a visualization. Using an expression language, live properties are assembled into a coordination graph to dynamically evaluate visual encodings and generate views of data. While Improvise and Reactive Vega share some conceptual underpinnings, Improvise places a higher burden on users to correctly construct the necessary graph. As Reactive Vega takes a declarative approach to visualization design, users need only compose the necessary primitives into a specification. Reactive Vega parses this specification to build the corresponding dataflow graph.

3 Background: Declarative Visualization Design

Reactive Vega builds on a long-running thread of research on declarative visualization design, popularized by the Grammar of Graphics [39] and Polaris [34] (now Tableau). Here, we aim to provide readers with sufficient background to understand the remainder of the paper. In particular, we focus on concepts used by Vega [32, 35], and its predecessors Protovis [9, 20] and D3 [10].



```
"scales": [{"name": "x"}],

"signals": [
  {"name": "brush_start", "stream": "mousedown"},
  {"name": "brush_end",
     "stream": "mouseup, [mousedown, mouseup] > mousemove"}
],

"predicates": [{
    "name": "inside_brush",
    "type": "in",
    "set": {
      "range": [{"signal": "brush_start.event.x"},
                {"signal": "brush_end.event.x"}],
      "scale": "-x"
    },
    "item": {"arg": "key"}
  }
],

"marks": [{
    "type": "symbol", "properties": { "update": {
      "rule": [
        {
          "predicate": "inside_brush",
          "input": {"key": {"field": "data.key"}},

          "fill": {"scale": "color", "field": "data.key"}
        },
        {
          "fill": {"value": "grey"}
        }
      ]}
}}]
```

Fig. 2. A declarative specification for a brushing interaction [32].

Visual encodings are defined by composing graphical primitives called *marks* [9], which include *arcs*, *areas*, *bars*, *lines*, plotting *symbols* and *text*. Marks are associated with datasets, and their specifications describe how tuple values map to visual properties such as position and color. Scales and guides (i.e., axes and legends) are provided as first-class primitives for mapping a domain of data values to a range of visual properties. Special *group* marks serve as containers to express nested or small multiple displays. Child marks and scales can inherit a group mark's data, or draw from independent datasets.

Mark specifications are processed by a multi-stage pipeline [20] to produce a visualization. Of note are the *build* and *evaluate* phases, which generate one mark instance per data tuple through a *data join* [10] and set appropriate properties based on a status flag. The status of a mark instance is determined by whether its backing data tuple has been added (*enter*), retained (*update*), or removed (*exit*).

Although interaction is a crucial component of effective visualization [26, 30], existing declarative visualization models, including widely used tools such as D3 [10] and ggplot2 [38], do not offer composable primitives for interaction design. Instead, if they support interaction, they do so through either a palette of standard techniques [9, 10] or imperative event handling callbacks. While the former restricts expressivity, the later undoes many of the benefits of declarative design. In particular, users are forced to contend with interaction execution details, such as interleaved events and coordinating external state, which can be complex and error-prone [13, 17, 29].

To address this gap, our recent work [32] introduces a model for declarative interaction design. Our approach draws on Event-Driven Functional Reactive Programming (E-FRP) [36] to abstract input events as time-varying streaming data. An event selector syntax facilitates composing and sequencing events together, for example `[mousedown, mouseup] > mousemove` is a single stream of `mousemove` events that occur between a `mousedown` and `mouseup` (i.e., "drag" events). Event streams are modeled as first-class data sources and can thus drive visual encoding primitives, or be run through the full gamut of data transformations.

For added expressivity, event streams can be composed into reactive expressions called *signals*. Signals can be used directly to specify visual primitive properties. For example, a signal can dynamically determine a mark's fill color or a scale's domain. Signals can also parameterize interactive selection rules for visual elements called *predicates*. Predicates define membership within the selection (e.g., by specifying the conditions that must hold true) and can be used within sequences of *production rules* to drive conditional visual encodings.

Figure 2 shows how to use these interaction primitives to specify a brushing interaction. Using this model, declaratively-specified interaction techniques can be encapsulated and parameterized into a standalone "interactor" definition. An interactor can then be reused and repurposed with any number of visualizations, functioning like a mixin.

4 The Reactive Vega Architecture

The Reactive Vega system architecture integrates streaming database techniques with Event-Driven Functional Reactive Programming (E-FRP), and extends both to support expressive visualization design. It comprises the necessary set of dataflow operators and methods to model both raw data and interactions events as streaming input in a uniform fashion. Dataflow operators are instantiated and connected by the Reactive Vega *parser*, which traverses a declarative specification containing definitions for input datasets, visual encoding rules, and interaction primitives as described in § 3. When data tuples are observed, or when interaction events occur, they are propagated (or "*pulsed*") through the graph with each operator being evaluated in turn. Propagation ends at the graph's sole sink: the renderer.

The Reactive Vega architecture and parser are implemented in the JavaScript programming language, and are intended to run either in a web browser or server-side using node.js. By default, Reactive Vega renders to an HTML5 Canvas element; however, it also supports Scalable Vector Graphics (SVG) and server-side image rendering.

4.1 Data, Interaction, and Scene Graph Operators

Reactive Vega dataflow operators fall into one of three categories: input data processing, interaction handling, or scene graph construction.

4.1.1 Processing Input Data

Reactive Vega parses each dataset definition and constructs a corresponding branch in the dataflow graph, as shown in Figure 3. These branches comprise input and output nodes connected by a pipeline of data transformation operators. Input nodes receive raw tuples as a linear stream (tree and graph structures are supported via parent-child or neighbor pointers, respectively). Upon data source updates, tuples are flagged as either *added*, *modified*, or *removed*, and each tuple is given a unique identifier. Data transformation operators use this metadata to perform targeted computation and, in the process, may derive new tuples from existing ones. Derived tuples retain access to their "parent" via prototypal inheritance. This relieves operators of the burden of propagating unrelated upstream changes.

Some operators require additional inspection of tuple state. Consider an aggregate operator that calculates running statistics over a dataset (e.g., mean and variance). When the operator observes added or removed tuples, the statistics can be updated based on the current tuple values. With modified tuples, the previous value must be subtracted from the calculation and the new value added. Correspondingly, tuples include a `previous` property. Writes to a tuple attribute are done through a setter function that copies current values to the `previous` object.

4.1.2 Handling Interaction

Reactive Vega instantiates an event listener node in the dataflow graph for each low-level event type required by the visualization (e.g., `mousedown` or `touchstart`). These nodes are directly connected to dependent signals as specified by event selectors [32]. In the case of ordered selectors (e.g., a "drag" event specified by `[mousedown, mouseup] > mousemove`), each constituent event is connected to

```
"data": [
   { "name": "stocks", "url": "data/stocks.csv"}},

   { "name": "index point",
     "source": "stocks",
     "transform": [{
       "type": "filter",
       "test": "month(d.date) == month(indexDate) &&
                year(d.date) == year(indexDate)"
     }]
   },

   { "name": "indexed_stocks",
     "source": "stocks",
     "transform": [
       { "type": "zip",
         "with": "index",
         "as": "index_term",
         "key": "symbol",
         "withKey": "symbol",
         "default": {"price": 0}
       },
       { "type": "formula",
         "field": "indexed_price",
         "expr": "d.index_term.price > 0 ?
                 (d.price - d.index_term.price) / d.index_term.price : 0"
     }]
   }
]
```

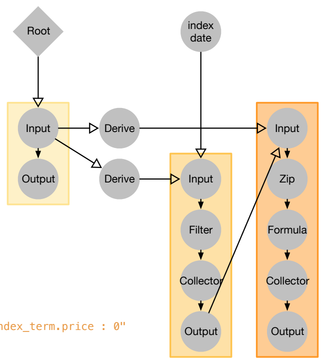
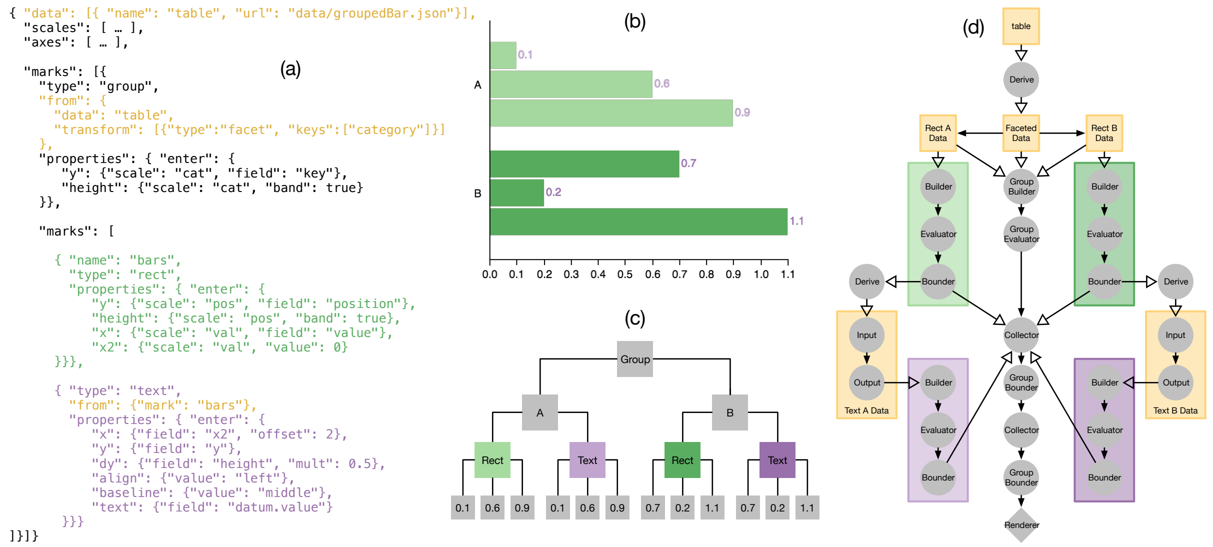


Fig. 3. The declarative specification (left) and resultant dataflow graph (right) for processing the input data of the streaming index chart in Fig. 1.



```
{ "data": [{ "name": "table", "url": "data/groupedBar.json"}],
  "scales": [ … ],
  "axes": [ … ],

  "marks": [{
    "type": "group",
    "from": {
      "data": "table",
      "transform": [{"type":"facet", "keys":["category"]}]
    },
    "properties": { "enter": {
       "y": {"scale": "cat", "field": "key"},
       "height": {"scale": "cat", "band": true}
    }},

    "marks": [

      { "name": "bars",
        "type": "rect",
        "properties": { "enter": {
           "y": {"scale": "pos", "field": "position"},
           "height": {"scale": "pos", "band": true},
           "x": {"scale": "val", "field": "value"},
           "x2": {"scale": "val", "value": 0}
      }}},

      { "type": "text",
        "from": {"mark": "bars"},
        "properties": { "enter": {
           "x": {"field": "x2", "offset": 2},
           "y": {"field": "y"},
           "dy": {"field": "height", "mult": 0.5},
           "align": {"value": "left"},
           "baseline": {"value": "middle"},
           "text": {"field": "datum.value"}
       }}}
]}]}
```

Fig. 4. (a) The specification for (b) a grouped bar chart, with (c) the underlying scene graph, and (d) corresponding portion of the dataflow graph.

an automatically created anonymous signal; an additional anonymous signal connects them to serve as a gatekeeper, and only propagates the final signal value when appropriate. Individual signals can be dependent on multiple event nodes and/or other signals, and value propagation follows E-FRP's two-phase update [36] as described in § 4.3.

4.1.3 Constructing the Scene Graph

To construct a scene graph, Reactive Vega follows a process akin to the Protovis bind-build-evaluate pipeline [20]. When a declarative specification is parsed, Reactive Vega traverses the mark hierarchy to *bind* property definitions: property sets are compiled into encoding functions and stored with the specification. At run-time, *build* and *evaluate* operators are created for each bound mark. The build operator performs a data join [10] to generate one scene graph element (or "mark") per tuple in the backing dataset, and the evaluate operator runs the appropriate encoding functions. A downstream *bounds* operator calculates the bounding boxes of generated marks. For a nested scene graph to be rendered correctly, the order of operations is critical: parent marks must be built and encoded before their children, but the bounds of the children must be calculated before their parents. The resultant scene graph exhibits an alternating structure, with individual mark elements grouped under a sentinel node that holds the mark specification. Figure 4 illustrates this process for a simple grouped bar chart example.

Generated scene graph elements are modeled as data tuples and can serve as the input data for downstream visual encoding primitives. This establishes a *reactive geometry* that accelerates common layout tasks, such as label positioning, and expands the expressiveness of the specification language (prior versions of Vega do not support reactive geometry). As generated marks can be run through subsequent data transformations, higher-level layout algorithms (e.g., those that require a pre-computed initial layout [15]) are now supported in a fully declarative fashion.

4.2 Changesets and Materialization

All data does not flow through the system at all times. Instead, operators receive and transmit *changesets*. A changeset consists of tuples that have been observed, new signal values, and updates to other dependencies that have transpired since the last render event. The propagation of a changeset begins in response to streaming tuples or user interaction. The corresponding input node creates a fresh changeset, and populates it with the detected update. As the changeset flows through the graph, operators use it to perform targeted recomputation, and may augment it in a variety of ways. For example, a `Filter` operator might remove tuples from a changeset if they do not meet the filter predicate, or may mark modified tuples as `added` if they previously had been filtered. A Cartesian product operator, on the other hand, would replace all the tuples in the incoming changeset with the results of a cross-product with another data stream.

While changesets only include updated data, some operators require a complete dataset. For example, a windowed-join requires access to all tuples in the current windows of the joined data sources. For such scenarios, special *collector* operators (akin to *views* [2] or *synopses* [4] in streaming databases) exist to materialize the data currently in a branch. In order to mitigate the associated time and memory expenses, Reactive Vega automatically shares collectors between dependent operators. Upon instatiation, such operators must be annotated as requiring a collector; at run-time they can then request a complete dataset from the dataflow graph scheduler.

Finally, if animated transitions are specified, a changeset contains an interpolation queue to which mark evaluators add generated mark instances; the interpolators are then run when the changeset is evaluated by the renderer.

4.3 Coordinating Changeset Propagation

A centralized dataflow graph scheduler is responsible for dispatching changesets to appropriate operators. The scheduler ensures that changeset propagation occurs in topological order so that an operator is only evaluated after all of its dependencies are up-to-date. This schedule prevents wasteful intermediary computation or momentary inconsistencies, known as *glitches* [13]. Centralizing this responsibility, rather than delegating it to operators, enables more aggressive pruning of unnecessary computation. As the scheduler has access to the full graph structure, it has more insight into the state of individual operators and the progress of the propagation. We describe scheduling optimizations in § 5.2.

When an interaction event occurs, however, an initial non-topological update of signals is performed. Dependent signals are reevaluated according to the order of their definitions within the declarative specification. As a result, signals may use prior computed values of their dependencies, which will subsequently be updated. This process mimics E-FRP's two-phase update [36], and is necessary to enable expressive signal composition. Once all necessary signals have been reevaluated, a changeset with the new signal values is sent to the scheduler for propagation to the rest of the dataflow graph.

4.4 Pushing Internal and Pulling External Changesets

Two types of edges connect operators in the dataflow graph. The first connects pairs of operators that work with the same data; for example a pipeline of data transformation operators for the same data source, or a mark's build and evaluate operators. Changesets are pushed along these edges, and operators directly use, augment, and propagate them.

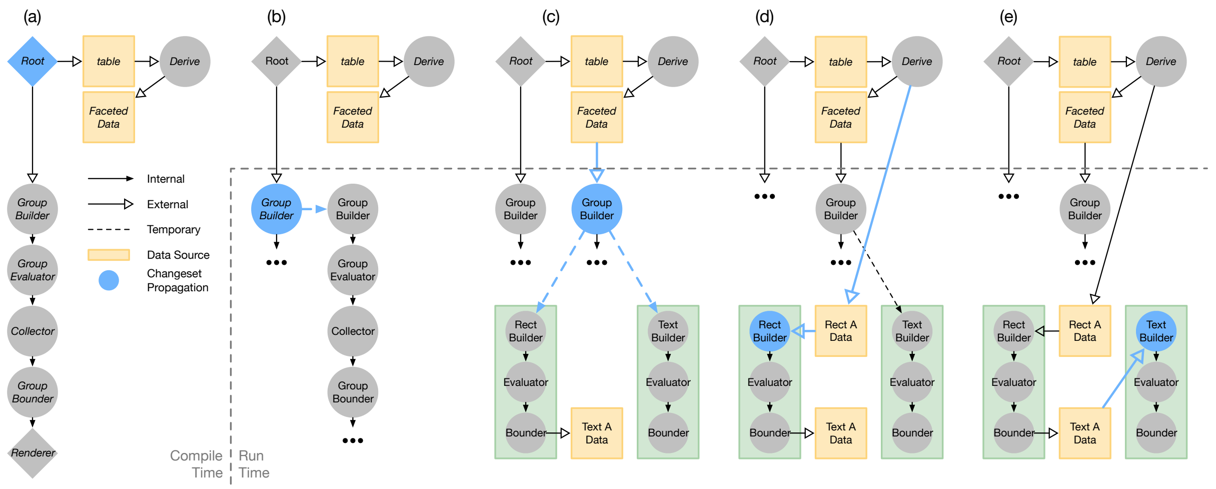


Fig. 5. Dataflow operators responsible for scene graph construction are dynamically instantiated at run-time, a process that results in the graph seen in Fig. 4. (a) At compile-time, a branch corresponding to the root scene graph node is instantiated. (b-c) As the changeset (in blue) propagates through nodes, group-mark builders instantiate builders for their children. Parent and child builders are temporarily connected (dotted lines) to ensure children are built in the same timecycle. (d-e) When the changeset propagates to the children, the temporary connection is replaced with a connection to the mark's backing data source (also in blue).

The second type of edge connects operators with external dependencies such as other data sources, signals, and scale functions. As these edges connect disparate data spaces, they cannot directly connect operators with their dependencies. To do otherwise would result in operators performing computation over mismatched data types. Instead, external dependencies are connected to their dependents' nearest upstream `Collector` node, and changesets that flow along these edges are flagged as *reflow changesets*. When a `Collector` receives a reflow changeset, it propagates its tuples forward, flagging them as modified. The dependents now receive correct input data and request the latest values of their dependencies from the scheduler.

The only exception to this pattern is when signals rely on other signals. Reflow changesets still flow along these edges but, as they operate in scalar data space, they are not mediated by `Collectors`.

This hybrid push/pull system enables a complex web of interdependent operators while reducing the implementation complexity of individual elements. For example, regardless of whether a signal parameterizes data transforms or visual encoding primitives, it simply needs to output a reflow changeset. Without such a system in place, the signal would instead have to construct a different changeset for each dependency edge it was a part of, and determine the correct dataset to supply. Figures 1, 3, 4, 5, and 11 use filled and unfilled arrows for internal and external connections, respectively.

4.5 Dynamically Restructuring the Graph

To support streaming nested data structures, operators can dynamically restructure the graph at runtime by extending new branches, or pruning existing ones, based on observed data. These dataflow branches model their corresponding hierarchies as standard relations, thereby enabling subsequent operators to remain agnostic to higher-level structure. For example, a `Facet` operator partitions tuples by key fields; each partition then propagates down a unique, dynamically-constructed dataflow branch, which can include other operators such as `Filter` or `Sort`.

In order to maintain interactive performance, new branches are queued for evaluation as part of the same propagation in which they were created. To ensure changeset propagation continues to occur in topological order, operators are given a *rank* upon instantiation to uniquely identify their place in the ordering. When new edges are added to the dataflow graph, the ranks are updated such that an operator's rank is always greater than those of its dependencies. When the scheduler queues operators for propagation, it also stores the ranks it observes. Before propagating a changeset to an operator, the scheduler compares the operator's current rank to the stored rank. If the ranks match, the operator is evaluated; if the ranks do not match, the graph was restructured and the scheduler requeues the operator.

The most common source of restructuring operations are scene graph operators, as building a nested scene graph is entirely data-driven. Dataflow branches for child marks (consisting of build-evaluate-bound chains) cannot be instantiated until the parent mark instances have been generated. As a result, only a single branch, corresponding to the root node of the scene graph, is constructed at compile-time. As data streams through the graph, or as interaction events occur, additional branches are created to build and encode corresponding nested marks. To ensure their marks are rendered in the same propagation cycle, new branches are temporarily connected to their parents. These connections are subsequently removed so that children marks will only be rebuilt and re-encoded when their backing data source updates. Figure 5 provides a step-by-step illustration of how scene graph operators are constructed during a propagation cycle for the grouped bar chart in Figure 4.

5 Architecture Performance Optimizations

Declarative language runtimes can transparently perform a number of performance optimizations [20]. In this section, we describe optimization strategies Reactive Vega uses to increase throughput and reduce memory usage. We evaluate the effect of each strategy through benchmark studies. Each benchmark was run with datasets sized at N = 100, 1,000, 10,000, and 100,000 tuples. For ecological validity, benchmarks were run with Google Chrome 42 (64-bit) and, to prevent confounds with browser-based just-in-time (JIT) optimizations, each iteration was run in a fresh instance. All tests were conducted on a MacBook Pro system running Mac OS X 10.10.2, with a quad-core 2.5GHz Intel Core i7 processor and 16GB of 1600 MHz DDR3 RAM.

5.1 On-Demand Tuple Revision Tracking

Some operators (e.g., statistical aggregates) require both a tuple's current and previous values. Tracking prior values can affect both running time and memory consumption. One strategy to minimize this cost is to track tuple revisions only when necessary. Operators must declare their need for prior values. Then, when tuples are ingested, their previous values are only tracked if the scheduler determines that they will flow through an operator that requires revision tracking.

We ran a benchmark comparing three conditions: always track revisions, never track revisions, and on-demand tracking. Although the "never" condition produces incorrect results, it provides a lower-bound for performance. We measured the system's throughput as well as memory allocated when initializing a scatterplot specification, and after modifying either 1% or 100% of input tuples. The scatterplot features two symbol marks fed by two distinct dataflows, `A` and `B`. Both

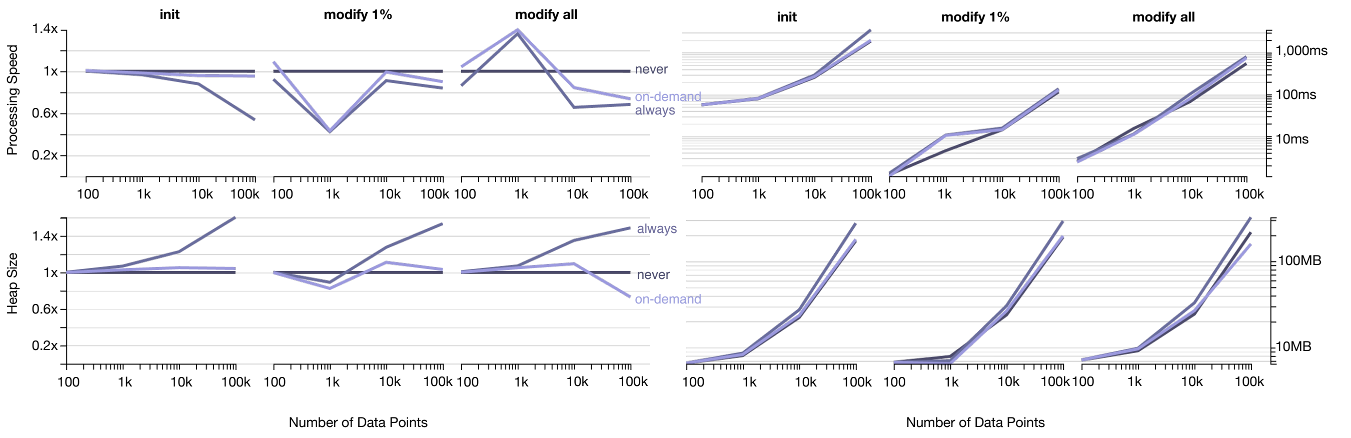


Fig. 6. Effects of tuple revision optimizations on average processing speed (top) and memory footprint (bottom). Left-hand figures show relative changes using no-tracking as a baseline (closer to 1.0 are better), and right-hand figures show the absolute values on a $\log_{10}$ scale (lower is better).

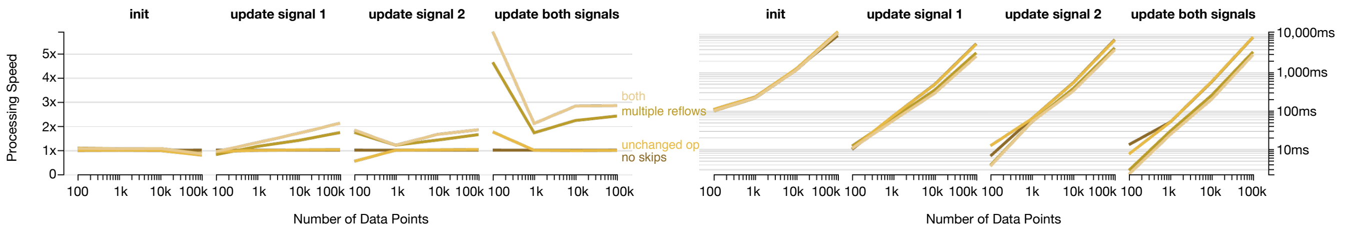


Fig. 7. The effects of pruning unnecessary computation on average processing speed. (a) A relative difference between conditions (higher is better). (b) Absolute values for time taken, plotted on a $\log_{10}$ scale (lower is better).

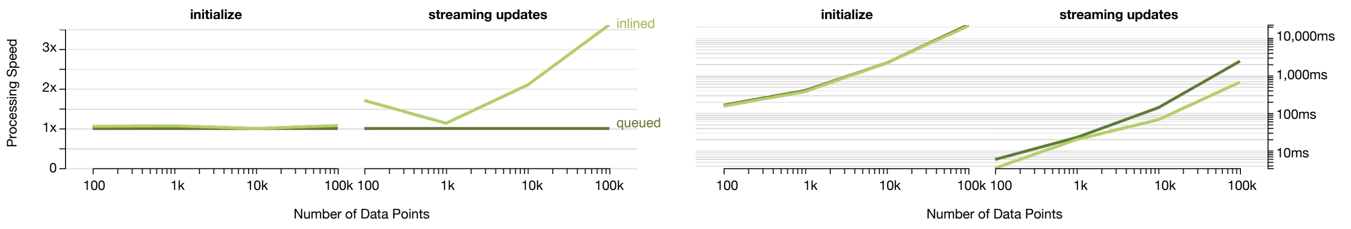


Fig. 8. The effects of inlining sequential operators on average processing speed. (a) A relative difference between conditions (higher is better). (b) Absolute values for time taken, plotted on a $\log_{10}$ scale (lower is better).

branches ingest the same set of tuples, and include operators that derive new attributes. However, `B` includes additional aggregation operators that require revision tracking.

The results are shown in Figure 6, with the effects of revision tracking most salient at larger dataset sizes. Always tracking revisions can require 20-40% more memory, and can take up to 50% longer to initialize a visualization due to object instantiation overhead for storing previous values. Our on-demand strategy effectively reduces these costs, requiring only 5-10% more memory and taking 5% longer to initialize than the “never” condition.

5.2 Pruning Unnecessary Recomputation

By centralizing responsibility for operator scheduling and changeset dispatch, we can aggressively prune unnecessary recomputation. The dataflow graph scheduler knows the current state of the propagation, and dependency requirements for each queued operator, allowing us to perform two types of optimizations.

Pruning multiple reflows of the same branch: As the scheduler ensures a topological propagation ordering, a branch can be safely pruned for the current propagation if it has already been reflowed.

Skipping unchanged operators: Operators identify their dependencies — including signals, data fields, and scale functions — and changesets maintain a tally of updated dependencies as they flow through the graph. The scheduler skips evaluation of an individual operator if it is not responsible for deriving new tuples, or if a changeset contains only modified tuples and no dependencies have been updated. Downstream operators are still queued for propagation.

To measure the impact of these optimizations, we created a grouped bar chart with five data transformation operators: `Derive` → `Fold` → `Derive` → `Filter` → `Facet`. The first `Derive` is parameterized by a signal, and the latter `Derive` and `Filter` operators are parameterized by a second, distinct signal. We then benchmarked the effect of four conditions (processing all recomputations, pruning multiple reflows only, skipping unchanged operators only, and applying both optimizations) across four tasks (initializing the visualization, updating each signal in turn, and updating both signals simultaneously).

Results are shown in Figure 7. Preventing multiple reflows is the most effective strategy, increasing throughput 1.4 times on average. Skipping unchanged operators sees little benefit by itself as, in our benchmark setup, only the two operators following a fold are skipped when changing `signal1`, and only the first derivation operator is skipped when changing `signal2`. When the two strategies are combined, however, we see a 1.6x increase in performance. This result was consistent across multiple benchmark trials. After careful hand-verification to ensure no additional nodes were erroneously skipped, we hypothesize that the JavaScript runtime is able to perform just-in-time optimizations that it is unable to apply to the other conditions.

5.3 Inlining Sequential Operators

To propagate changesets through the dataflow graph, the scheduler adds operators to a priority queue, backed by a binary heap sorted in topological order. This incurs an O(log N) cost for enqueueing and

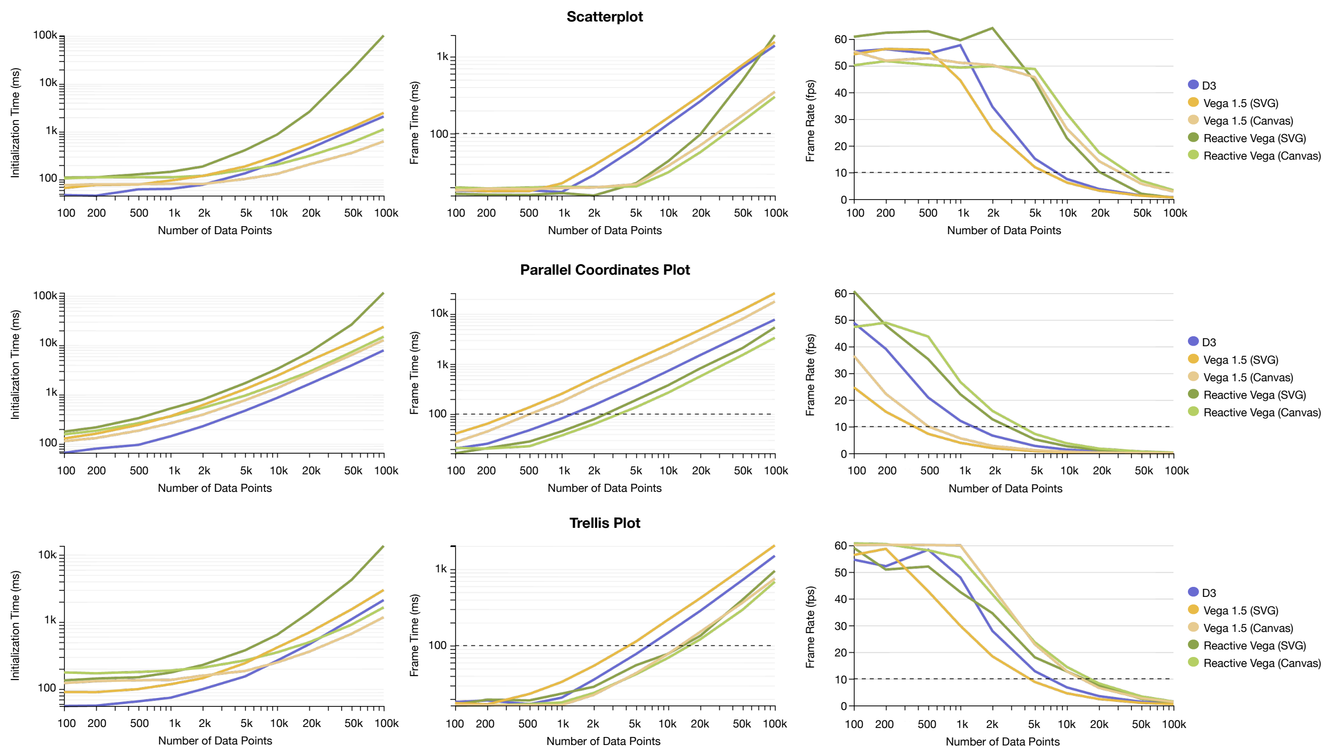


Fig. 9. Average performance of rendering (non-interactive) streaming visualizations: (top-bottom) scatterplot, parallel coordinates, and trellis plot; (left-right) initialization time, average frame time, and average frame rate. Dashed lines indicate the threshold of interactive updates [11].

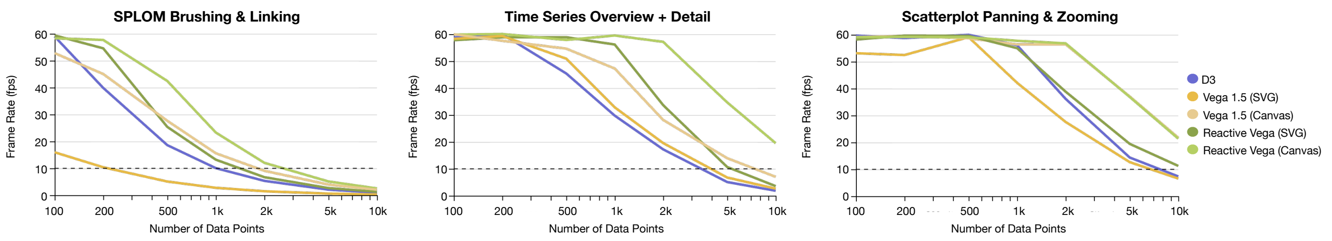


Fig. 10. Average frame rates for three interactive visualizations: (left-right) brushing and linking on a scatterplot matrix; brushing and linking on an overview+detail visualization; panning and zooming on a scatterplot. Dashed lines indicate the threshold of interactive updates [11].

dequeueing operators, which can be assessed multiple times per operator if the graph is dynamically restructured. However, branching only occurs as a result of operators registering dependencies, and dependencies are only connected to `Collector` nodes. As a result, much of the dataflow graph comprises linear paths. This is particularly true for scene graph operators, which are grouped into hundreds (or even thousands) of independent mark build-evaluate-bound branches.

We explore the effect of inline evaluation of linear branches, whereby operators indicate that their neighbors can be called directly rather than queued for evaluation. The scheduler remains responsible for propagating the changeset, and thus can continue to apply the optimizations previously discussed. Although inline evaluation can be applied in a general fashion by coalescing linear branches into "super nodes," for simplicity we only evaluate inlining of scene graph operators here. Mark builders directly call evaluators and bounders, and group mark builders directly call new child mark builders rather than forming a temporary connection.

Figure 8 shows the results of this optimization applied to a parallel coordinates plot (PCP). The plot uses a nested scene graph in which each line segment is built by a dedicated build-evaluate-bound branch. As we can see, inlining does not have much impact on the initialization time. This is unsurprising, as the largest initialization cost is due to unavoidable graph restructuring. However, inlining improves streaming operations by a 1.9x factor on average. As streaming updates only propagate down specific branches of the dataflow graph, inline evaluation results in at least 4 fewer queuing operations by the scheduler.

6 Comparative Performance Benchmarks

We now evaluate the performance of Reactive Vega against D3 [10] and the original, non-reactive Vega system (v1.5.0) [35]. These performance evaluations use the setup previously described.

6.1 Streaming Visualizations

Figure 9 shows the average performance of (non-interactive) streaming scatter plots, parallel coordinates plots, and trellis plots. We first measured the average time to initially parse and render the visualizations. To gauge streaming performance, we next measured the average time taken to update and re-render upon adding, modifying, or removing 1% of tuples. We tested with datasets sized between 100 and 100,000 tuples, and ran 10 trials per size.

Reactive Vega has the greatest effect with the parallel coordinates plot, displaying 2x and 4x performance increases over D3 and Vega 1.5, respectively. This effect is the result of each line in the plot being built and encoded by its own branch of the dataflow graph. Across the other two examples, and averaging between the Canvas and SVG renderers, we find that although Reactive Vega takes 1.7x longer to initialize the visualizations, subsequent streaming operations are 1.9x faster than D3. Against Vega 1.5, Reactive Vega is again 1.7x slower at initializing visualizations; streaming updates perform roughly op-par with the Canvas renderer, but are 2x faster with the SVG renderer.

Slower initialization times for Reactive Vega are to be expected. D3 does not have to parse and compile a JSON specification, and a streaming dataflow graph is a more complex execution model, with higher overheads, than batch processing. However, with streaming visualizations this cost amortizes and performance in response to data changes becomes more important. In this case, Reactive Vega makes up the difference in a single update cycle.

6.2 Interactive Visualizations

We evaluated the performance of interactive visualizations (measured in terms of interactive frame rate) using three common examples: brushing & linking a scatterplot matrix, a time-series overview+detail visualization, and panning & zooming a scatterplot. We chose these examples as they all leverage interactive behaviors supported by D3, with canonical implementations available for each[1,2,3]. For Reactive Vega, we expressed these visualizations with a single declarative specification. For D3 and Vega 1.5, we use custom event handling callbacks. The Vega 1.5 callbacks mimic the behavior of the fragmented reactive approach used in prior work [32]. We tested these visualizations with datasets sized between 100 and 10,000 tuples.

Figure 10 shows the results — on average, and across both Canvas and SVG renderers, Reactive Vega offers superior interactive performance to custom D3 and Vega event handling callbacks. This effect primarily stems from Reactive Vega's unified data model, and is most noticeable with brushing & linking a scatterplot matrix and the time-series overview+detail visualization. In both examples, interactions manipulate only a subset of all data tuples. With Reactive Vega, only these tuples are processed, and their corresponding scene graph elements re-encoded and re- rendered. By comparison, with Vega 1.5's fragmented reactive approach, the entire scene graph must be reconstructed and rendered in response to changes in input data.

7 Example Streaming and Interactive Visualizations

Prior work [32] demonstrates the expressivity of declarative interaction design with example visualizations that cover a taxonomy of interaction techniques [41]. Figure 12 illustrates several of these interactive visualizations using Reactive Vega. We now describe additional examples that illustrate new use cases that Reactive Vega enables, highlighting advantages of declarative interactive visualization design.

7.1 Streaming Financial Index Chart

Prior work [32] used declarative interaction primitives to specify a financial index chart. A static snapshot of time-series stock price data for several companies is visualized as a line chart and interactively normalized by an index point.

With Reactive Vega, we can extend this example to use real-time stock prices, rather than static historical data, by leveraging the Yahoo! Finance API. We initialize the visualization by requesting the stock prices of companies over the past 24 hours, at a minute-level resolution. Then, every minute, we poll the API endpoints again to request the most recent prices. A predicate checks the `timestamp` property of incoming tuples to ensure that only new data is added to the visualization. This guards against adding duplicate data when our update cycle does not coincide with Yahoo's, or when the markets are closed. API calls are synchronized and a changeset is only fired through the graph once all requests have received responses. As a result, all lines update together every minute. The resulting real-time index chart and corresponding dataflow graph are shown in Figure 1.

7.2 DimpVis: Touch Navigation with Time-Series Data

DimpVis [23] is a recently introduced interaction technique that allows direct manipulation navigation of time-series data. Starting with a scatterplot depicting data at a particular time slice, users can touch plotted points to reveal a "hint path": a line graph that displays the trajectory of the selected element over time. Dragging the selected point along this path triggers temporal navigation, with the rest of the points updating to reflect the new time. In evaluation studies, users reported feeling more engaged when exploring their data using DimpVis [23].

[1]Brushing & Linking: http://bl.ocks.org/mbostock/4063663
[2]Overview + Detail: http://bl.ocks.org/mbostock/1667367
[3]Pan & Zoom: http://bl.ocks.org/mbostock/3892919

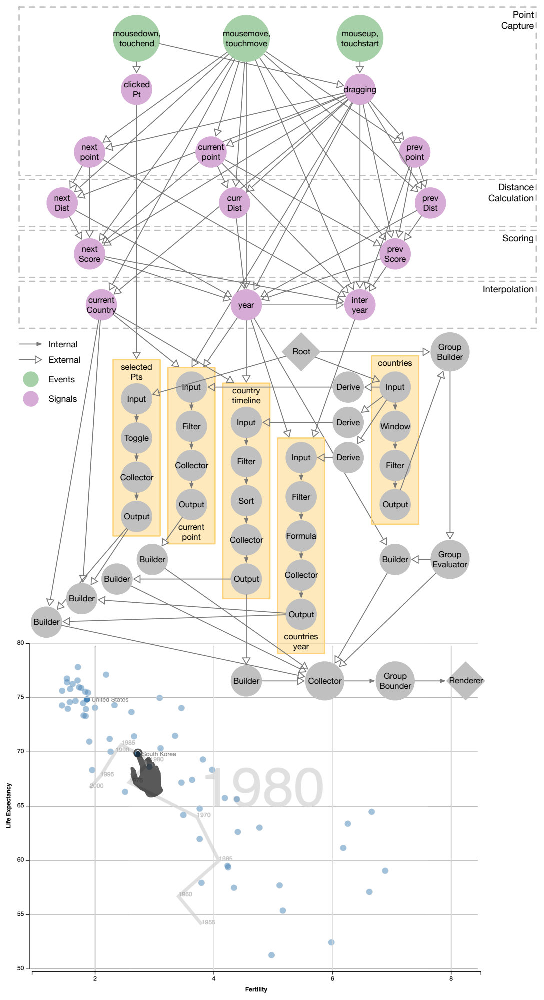


Fig. 11. The dataflow graph produced by parsing a declarative specification for the DimpVis interaction technique [23]. Users tap and drag points to navigate the data through time. Signals calculate distances and scoring functions to smoothly interpolate the points in response.

We can recreate this technique with Reactive Vega's declarative interaction primitives and the GapMinder country-fertility-life-expectancy dataset used by the original. Input data is passed through a `Window` transform, such that every tuple contains references to the tuples that come before and after it in time, and filtered to remove triplets that span multiple countries. Signals constructed over mouse and touch events capture the selected point, and downstream signals calculate distances between the user's current position and the previous and next points. A scalar projection over these distances gives us scoring functions that determine whether the user is moving forwards or backwards in time. Scores feed a signal that is used in a derived data source to calculate new interpolated properties for the remaining points in the dataset. These interpolated properties determine the position of plotted points, thereby producing smooth transitions as the user drags back-and-forth. To draw the hint map, an additional derived data source filters data tuples for the currently country across all years. Figure 11 shows the dataflow graph produced by this specification.

7.3 Reusable Touch Interaction Abstractions

With the proliferation of touch-enabled devices, particularly smartphones and tablets, supporting touch-based interaction has become an increasingly important part of interactive visualization design. However, HTML5 provides a low-level API for touch events, with only three event types broadly supported — `touchstart`, `touchmove`, and `touchend`. On multitouch devices these events contain an array of touch points. The application developer is responsible for the book-

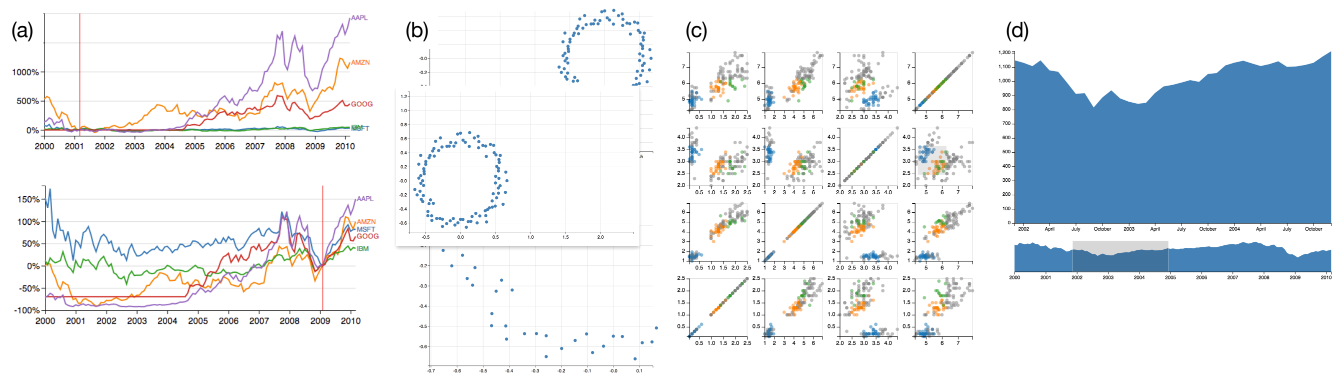


Fig. 12. Example declarative interactive visualizations [32] that cover an existing taxonomy of interaction techniques [41]: (a) *Reconfigure* with an Index Chart; (b) *Explore* by panning & zooming; (c) *Select & Connect* with brushing & linking; (d) *Abstract/Elaborate* with overview+detail;

keeping involved with tracking multiple points across interactions, a cumbersome and difficult process.

Declarative interaction design enables us to abstract low-level details away, and instead expose higher-level, semantic events in a reusable fashion. Definitions for signals and other interaction primitives can be encapsulated and parameterized as standalone "interactors." When an interactor is added to a visualization, Reactive Vega merges the two specifications together, name-spacing components to prevent conflicts. In this way, we can build an interactor comprising signals that perform the necessary logic for common multitouch gestures. When included in a host visualization, the visualization designer can safely ignore lower-level events, and instead build interactions driven by signals provided by the interactor. For example, after including a touch interactor as part of a scatterplot specification, the designer can use `twotouchmove` and `pinchDelta` signals to drive panning and zooming interactions.

8 Discussion and Future Work

Declarative languages are a popular means of authoring visualizations [9, 10, 20], but have lacked first-class support for interaction design. Though recent work [32] addresses this gap with composable interaction primitives, it provides only a proof-of-concept system. In response, we contribute Reactive Vega, the first system architecture to support declarative visualization and interaction design in a comprehensive and performant fashion.

It is important to note that although Reactive Vega provides an complete end-to-end system — whereby users invoke the parser to traverse an input declarative specification and instantiate the necessary architecture components to render a visualization — this process can be decoupled. Alternate parsers can be supplied, and higher-level tools can opt to manually construct and connect required operators. Regardless of the specification process, the Reactive Vega system architecture provides the dataflow operators and management necessary to support expressive visualization design, with streaming raw data and interaction events modeled uniformly.

Authoring a declarative specification, however, can present a hurdle to users. Although separating specification and execution frees designers to focus on visual encoding decisions, it also hides the underlying execution model. Language-level optimizations and delayed property evaluations make debugging particularly difficult [10], with internal structures exposed only when errors arise. To better understand these tradeoffs, we evaluated Reactive Vega's declarative model in our prior work [32]. Using the Cognitive Dimensions of Notation [8], we determined that although declarative specification introduces hidden dependencies and decreases visibility, these are outweighed by an increase in the specification consistency of visual encoding and interaction, and a decrease in viscosity of abstraction primitives. Moveover, Vega's declarative JSON syntax simplifies programmatic generation of visualizations, enabling the creation of programs that generate and reason about visualizations at a higher level.

These findings are mirrored in the real-world adoption of Vega. For example, Wikipedia, a security-concious environment where it would be difficult to allow users to write imperative visualization code, has recently integrated Vega [27] to enable visualization of data embedded in articles. Similarly, Vega's declarative format is well-suited for generation by higher-level tools. For example, Lyra [31] allows designers to create Vega visualizations through direct-manipulation; and Voyager [40] eschews the specification process for data exploration, by providing faceted search over a gallery of Vega visualizations.

Still, improved support for authoring and debugging Vega specifications remains a promising avenue for future work. Visualizations of program behavior have been shown to improve learnability [19], and Reactive Vega's dataflow graph offers an execution model that can be readily visualized. Linked selection among specification text, data sources, a dataflow graph diagram, and the output visualization could aid understanding and debugging. New debugging environments for Reactive Vega could instrument the dataflow graph to enable inspection, for example stepping through changeset propagation.

Reactive Vega's architecture also offers opportunities to study scalable visualization design. Interactive visualization of large-scale datasets often requires offloading computation to server-side architectures. For example, Nanocubes [24] and imMens [25] assemble multidimensional data cubes that can be decomposed into smaller data tiles and pushed to the client. Such components could be integrated into a dataflow graph with execution distributed across server and client. For example, as the dataflow graph scheduler is responsible for propagation, it might anticipate possible user interactions and prefetch data tiles in order to reduce latency [7].

Finally, as previously mentioned, an ecosystem of higher-level systems is developing around Vega. Statistical packages (ggvis [18]), data exploration tools (Voyager [40]), computational notebooks (iPython [21]), and graphical design tools (Lyra [31]) use earlier versions of Vega to automatically construct visualizations as part of the data analysis process, or to facilitate visualization prototyping and publishing. With Reactive Vega, these systems can extend the types of visualizations they support to include interactive and streaming examples. This expressivity, in turn, may spur development of alternate forms of specifying interactions, for example through higher-level languages or via direct manipulation and demonstration.

Reactive Vega is an open source system, and we have merged it with the existing Vega project. It is available at `http://vega.github.io/vega/`, along with a "live editor" and a number of example interactive visualizations.

Acknowledgments

This work was supported by an SAP Stanford Graduate Fellowship, the Intel Big Data ISTC, the Moore Foundation, and DARPA XDATA.

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The

design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB JournalThe International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[3] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *Proceedings of the 6th conference on Visualization'95*, page 263. IEEE Computer Society, 1995.

[4] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab, 2004.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. *ACM SIGMOD Record*, 29(2):261–272, 2000.

[6] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):52, 2013.

[7] L. Battle, R. Chang, and M. Stonebraker. Dynamic generation and prefetching of data chunks for exploratory visualization. In *IEEE InfoVis Posters Track*, 2014.

[8] 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*, pages 325–341. Springer, 2001.

[9] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Visualization & Comp. Graphics*, 15(6):1121–1128, 2009.

[10] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics*, 17(12):2301–2309, 2011.

[11] S. K. Card, T. P. Moran, and A. Newell. An engineering model of human performance. *Ergonomics: Psychological mechanisms and models in ergonomics*, 3:382, 2005.

[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

[13] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Programming Languages and Systems*, pages 294–308. Springer, 2006.

[14] J. Cottam and A. Lumsdaine. Stencil: a conceptual model for representation and interaction. In *Information Visualisation*, pages 51–56. IEEE, 2008.

[15] css-layout. `https://github.com/facebook/css-layout`, March 2015.

[16] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. In *Proc. ACM SIGPLAN*, pages 411–422. ACM, 2013.

[17] J. Edwards. Coherent reaction. In *Proc. ACM SIGPLAN*, pages 925–932. ACM, 2009.

[18] ggvis: Interactive grammar of graphics for R. `http://ggvis.rstudio.com/`, June 2015.

[19] P. J. Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584. ACM, 2013.

[20] J. Heer and M. Bostock. Declarative language design for interactive visualization. *IEEE Trans. Visualization & Comp. Graphics*, 16(6):1149–1156, 2010.

[21] The IPython Notebook. `http://ipython.org/notebook.html`, June 2015.

[22] C. Kelleher and H. Levkowitz. Reactive data visualizations. In *IS&T/SPIE Electronic Imaging*, pages 93970N–93970N. International Society for Optics and Photonics, 2015.

[23] B. Kondo and C. Collins. Dimpvis: Exploring time-varying information visualizations by direct manipulation. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2003–2012, 2014.

[24] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2456–2465, 2013.

[25] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data. *Computer Graphics Forum (Proc. EuroVis)*, 32, 2013.

[26] Z. Liu and J. T. Stasko. Mental models, visual reasoning and interaction in information visualization: A top-down perspective. *IEEE Trans. Visualization & Comp. Graphics*, 16(6):999–1008, 2010.

[27] MediaWiki Extension:Graph. `https://www.mediawiki.org/wiki/Extension:Graph`, June 2015.

[28] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. *ACM SIGPLAN Notices*, 44(10):1–20, 2009.

[29] B. A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proc. ACM UIST*, pages 211–220. ACM, 1991.

[30] W. A. Pike, J. Stasko, R. Chang, and T. A. O'Connell. The science of interaction. *Information Visualization*, 8(4):263–274, 2009.

[31] A. Satyanarayan and J. Heer. Lyra: An interactive visualization design environment. *Computer Graphics Forum (Proc. EuroVis)*, 2014.

[32] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative interaction design for data visualization. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 669–678. ACM, 2014.

[33] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proceedings of the 7th conference on Visualization'96*, pages 93–ff. IEEE Computer Society Press, 1996.

[34] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Visualization & Comp. Graphics*, 8(1):52–65, 2002.

[35] Vega: A Visualization Grammar. `http://trifacta.github.io/vega`, March 2015.

[36] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Practical Aspects of Declarative Languages*, pages 155–172. Springer, 2002.

[37] C. Weaver. Building highly-coordinated visualizations in Improvise. In *Proc. IEEE Information Visualization*, pages 159–166, 2004.

[38] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009.

[39] L. Wilkinson. *The Grammar of Graphics*. Springer, 2005.

[40] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Trans. Visualization & Comp. Graphics*, 2015.

[41] J. S. Yi, Y. ah Kang, J. T. Stasko, and J. A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, 2007.