Mosaic: An Architecture for Scalable & Interoperable Data Views

Jeffrey Heer and Dominik Moritz

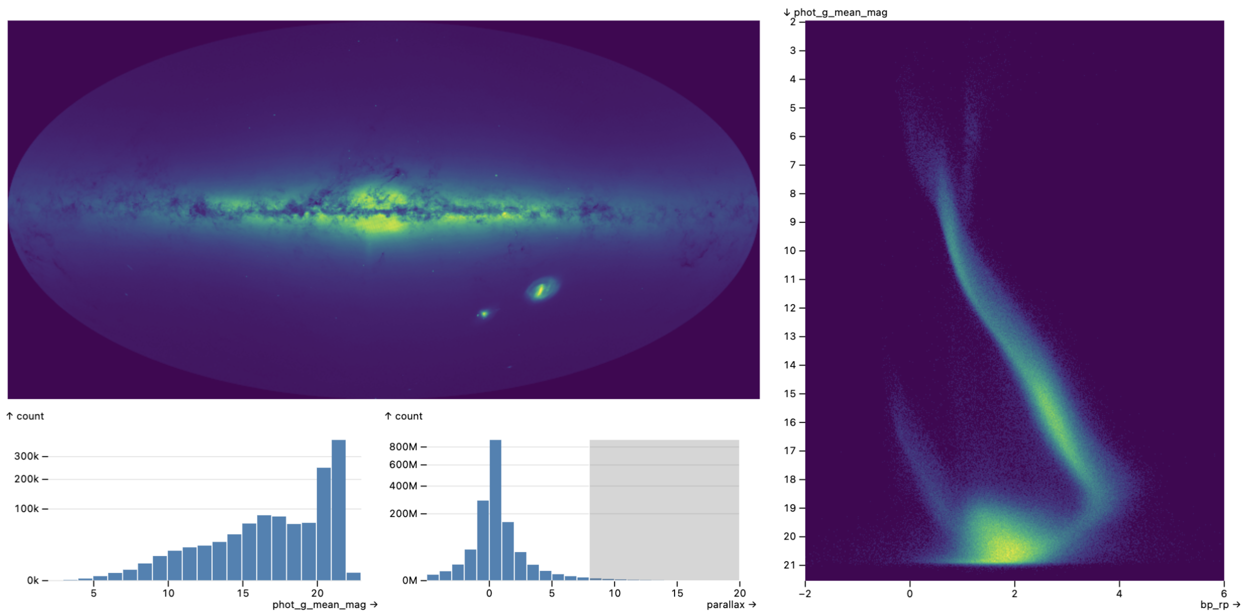


Fig. 1: A Mosaic-based interface for interactive visual exploration of all 1.8 billion stars in the Gaia star catalog. A high-resolution density map of the sky reveals our Milky Way and satellite galaxies. Stars with higher parallax values are interactively selected, forming a Hertzsprung-Russell diagram of color versus stellar magnitude on the right. Mosaic offloads density and histogram computation to a backing scalable database, and automatically builds optimized data cube indexes to support interactive linked views.

Abstract—Mosaic is an architecture for greater scalability, extensibility, and interoperability of interactive data views. Mosaic decouples data processing from specification logic: clients publish their data needs as declarative queries that are then managed and automatically optimized by a coordinator that proxies access to a scalable data store. Mosaic generalizes Vega-Lite's selection abstraction to enable rich integration and linking across visualizations and components such as menus, text search, and tables. We demonstrate Mosaic's expressiveness, extensibility, and interoperability through examples that compose diverse visualization, interaction, and optimization techniques—many constructed using *vgplot*, a grammar of interactive graphics in which graphical marks act as Mosaic clients. To evaluate scalability, we present benchmark studies with order-of-magnitude performance improvements over existing web-based visualization systems—enabling flexible, real-time visual exploration of billion+ record datasets. We conclude by discussing Mosaic's potential as an open platform that bridges visualization languages, scalable visualization, and interactive data systems more broadly.

Index Terms—Visualization, Interaction, Scalability, Grammar of Graphics, Software Architecture, Databases

1 Introduction

Though many expressive visualization tools exist, scalability to large datasets and interoperability across tools remain challenging [7]. The visualization community lacks open, standardized tools for integrating visualization specifications with scalable analytic databases. While libraries like D3 [8] embrace Web standards for cross-tool interoperability, higher-level frameworks often make closed-world assumptions, complicating integration with other tools and environments.

As a concrete example, consider the Vega [36] and Vega-Lite [35] ecosystem. By default, data transformations are performed within a JavaScript runtime, limiting scalability due to both data movement and a lack of parallel computing. Meanwhile, other architectural decisions impede extensibility and interoperability. Vega-Lite's *selection* abstraction provides a powerful, concise model for interaction, yet is realized in terms of opaque internal Vega constructs that complicate coordination with external components and environments such as notebooks. As a result, online requests[1] for table views, data-driven input widgets, more interoperable selections, and new mark types (e.g., for raster heatmaps and contour plots in Vega-Lite) have gone unaddressed for years. Other tools face similar limitations [4, 43].

We propose a standardized "middle-tier" architecture that mediates data-driven components and backing data sources. A common layer between databases and components can coordinate linked selections and parameters among views, while providing automatic query optimizations for greater scalability. We focus on the Web browser as the primary site of rendering and interaction, and seek to coordinate diverse components via standard protocols for communicating data needs, dynamic parameters, and linked selection criteria.

We contribute *Mosaic*, an architecture for interoperable, data-driven components—including visualizations, tables, and input widgets—backed by scalable data stores. A key idea of Mosaic is to decouple data processing from specification. Mosaic *Clients* communicate their data needs as declarative queries. A central *Coordinator* manages these queries, applies automatic optimizations, and pushes processing to a

• *Jeffrey Heer is with University of Washington. E-mail: jheer@uw.edu.*
• *Dominik Moritz is with Carnegie Mellon University. E-mail: domoritz@cmu.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

[1] See github.com/vega/vega/issues/ and github.com/vega/vega-lite/issues/

backing *Data Source* (by default DuckDB [34]). Dynamic *Params* and *Selections* enable coordinated updates to both clients and queries to support linked interaction. A variety of components and toolkits can interoperate via Mosaic's data management and selection facilities.

A main contribution of Mosaic is to unify the abstractions of popular visualization toolkits with scalable visualization techniques. Individual clients may perform local optimization in their generated queries. Meanwhile, the Mosaic Coordinator optimizes over multiple views and interaction cycles by caching, consolidating similar queries, and building data cube indexes for linked selections over aggregated data. We present a novel indexing approach that extends prior work on imMens [25] and Falcon [31] to support *automatic* index generation over a larger set of query types and aggregation functions. Mosaic also enables flexible deployment: using DuckDB and varied data connectors, Mosaic can process data directly in the browser via WebAssembly, within a Jupyter Notebook kernel, or on local or remote servers.

We demonstrate Mosaic's extensibility and interoperability by developing both data-driven input widgets and *vgplot*, a grammar of interactive graphics in which graphical marks are Mosaic clients. Marks in vgplot push filtering, binning, aggregation, and regression transformations to a backing database. Interactors for pan/zoom, point, and interval selections produce dynamic queries mediated by Mosaic *Params* and *Selections*. Akin to Vega-Lite, Mosaic users can write portable, declarative specifications that can be generated by various languages and integrated in computational notebooks.

To assess scalability, we present benchmark results for both static and interactive visualizations. Mosaic outperforms Vega, VegaFusion, and Observable Plot, typically by one or more orders of magnitude. For the static cases, DuckDB performance and client-level optimization account for the bulk of Mosaic's benefits. In the interactive cases, Mosaic's automatic data cube indexing enables real-time interaction with billion+ record datasets. We conclude by discussing limitations and Mosaic's use as a platform for research and development.

2 Related Work

Mosaic seeks to unify prior work on scalable visualization methods with expressive languages and tools for interactive visualization.

2.1 Scalable Visualization

Methods for scaling visualizations to larger datasets include sampling, fitting parametric models, and binned aggregation (potentially with smoothing). While valuable, both sampling and modeling are lossy, and so may fail to preserve structures and outliers of interest. Carr et al. [10] describe scalable scatter plot methods using hexagonal binning. Later works, including imMens [25] and Wickham's bin-summarise-smooth framework [44], further describe how to scale up a variety of common plots via binned aggregation. Density, violin, and contour plots can be constructed by smoothing binned counts [17, 41]. To scale traditional line or area charts, M4 [19, 20] uses a pixel-aware binning scheme that is visually identical to normal line rasterization. While many techniques focus on binning individual data points, Curve Density Estimates [23] and DenseLines [28] instead convey rendered densities for many series drawn as lines or curves. Mosaic provides an expressive and extensible system to develop and flexibly deploy such techniques.

Other methods focus on *interactive* visualization, supporting efficient updates for filtering and (re-)aggregation. Nanocubes [24] are specialized indexes for spatio-temporal queries, but can take considerable time to build. imMens [25] and Falcon [31] use *multivariate data tiles*: pre-aggregated data cubes [16] that can be rapidly queried to compute filtered aggregates, making interactive performance dependent on a chosen binning resolution rather than the number of backing records. Falcon also performs *prefetching* by requesting tiles when the mouse cursor enters a plot, prior to any selections. ForeCache [5] prefetches data tiles for multi-scale pan/zoom operations based on a model of user navigation behavior. Khameleon [27] streams approximate data tiles with a scheduler that trades off result quality and available bandwidth.

The Mosaic architecture supports indexing, prefetching, and other optimizations. Most notably, Mosaic's *Coordinator* analyzes client queries and linked selections to determine if pre-aggregated indexes are applicable; if so, it prefetches data cubes *automatically*. In contrast to prior work, our implementation supports aggregations beyond `count` (e.g., `sum`, `avg`, `min`, `max`), uses sparse indexes that scale to larger datasets, and caches data cubes in a backing database for reuse.

Kyrix [39] provides an API for scalable zoomable user interfaces, including precomputation of spatial positions and indexes within a backing database to support low-latency interaction. Kyrix-S [38] extends Kyrix with additional operators and a declarative specification syntax for scatter plot visualizations. Mosaic differs in providing a more general and extensible architecture (e.g., one could implement Kyrix-like systems using Mosaic) and in its use of reactive parameters and selections to coordinate interactive updates. Meanwhile, DIEL [48] orchestrates computation between local and remote databases. In contrast to Mosaic's reusable higher-level abstractions, DIEL requires writing application-specific queries and library integrations.

2.2 Visualization Languages and Tools

Commercial systems, including Tableau (previously Polaris [37]) and business intelligence tools, support visualizations backed by databases. However, the techniques used by these systems are proprietary, rarely published, and unavailable for researchers to freely use and extend.

Multiple research systems focus on linked interactions across views. Snap-Together visualization [33] and DEVise [26] support coordinated views based on a relational data model. Users of Glue [2] provide schema mappings between datasets to enable linked plotting and filtering. Nebula [11] represents coordination behaviors using structured natural language templates. Improvise [42] supports coordination via dynamic parameters (*live properties*) and *coordinated queries* defined over these parameters. Mosaic supports dynamic parameters (*Params*) as well as linked *Selections* modeled as declarative query predicates.

Meanwhile, open-source tools inspired by Wilkinson's Grammar of Graphics [45]—including ggplot2 [43], Vega [36], Vega-Lite [35], and Observable Plot [4]—support an expressive range of visualizations, often with a concise, combinatorial syntax. Vega and Vega-Lite further support declarative specification of interaction methods. Vega-Lite's *selection* abstraction combines input events and scale inversions to form query predicates over selected intervals or point values. These selections are realized as non-standardized internal Vega constructs, complicating interoperation with non-Vega tools.

Moreover, these languages were not designed to handle millions of data points. For greater scale, VegaFusion [22] and VegaPlus [49] analyze Vega dataflows and push transformations to a database. Still, many Vega transformations are not well supported. As these systems modify Vega internally, limitations around extensibility and interoperability (Section 1) also remain. Mosaic clients instead publish queries directly, sidestepping the complexities of translating Vega dataflows to other computation models. Our benchmark results (Section 8) find that Mosaic provides greater scalability across a larger set of visualizations.

Mosaic is an open, middle-tier architecture that higher-level languages such as ggplot2, Vega-Lite, or Observable Plot could target. Using a shared architecture, a visualization grammar could readily interoperate with other libraries, including input components and other visualization tools. We demonstrate this through the design of both input widgets and *vgplot*, a Mosaic-based grammar of interactive graphics that combines concepts from existing visualization tools.

Mosaic offers scalability by proxying queries to a backing database, and supports interaction by standardizing and generalizing Vega-Lite-style *selections*. Compared to Vega-Lite, Mosaic selections are decoupled from input event handling and support more complex resolution strategies. A single Mosaic selection may combine predicates provided by a variety of diverse views and input techniques. Mosaic selections can also synthesize different predicates for different views (clients), enabling complex coordination behaviors such as cross-filtering.

3 The Mosaic Architecture

A Mosaic application consists of data-consuming *Clients* registered with a central *Coordinator*. Clients publish their data needs as declarative queries. A Coordinator manages these queries, performs potential optimizations, submits queries to a backing *Data Source*, and returns

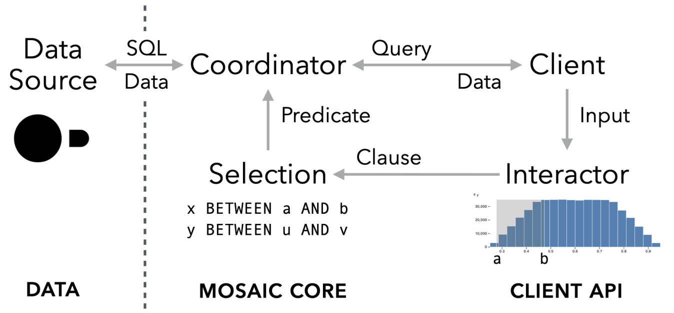


Fig. 2: Mosaic architecture overview. A *Coordinator* proxies queries to a backing *Data Source* for one or more data-consuming *Clients*. *Params* and *Selections* broadcast reactive updates for scalar values or query predicates, respectively. Interactions that update Params and Selections may be handled directly by a client, or via *Interactor* components.

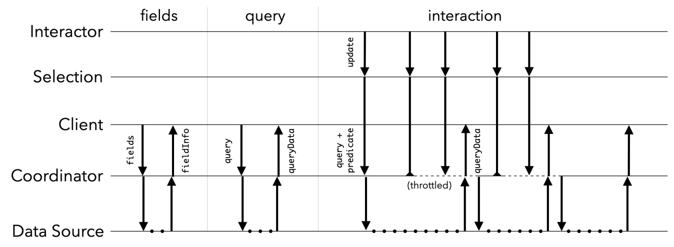


Fig. 3: Example Mosaic event timeline (not to scale). A client can provide a list of fields for which the Coordinator returns metadata. Next, the Coordinator requests a query from the Client, submits it to a Data Source for execution (dotted lines), and returns the result, providing the data the client needs to render. Interactions such as interval selections or pan/zoom update the state of linked Params or Selections, triggering additional rounds of query and update. While a query is being executed, corresponding selection updates are throttled (dashed lines): intermediate updates are dropped and only the most recent update is serviced.

results or errors back to clients. Interactions among components are mediated by *Params* and *Selections*, reactive variables for scalar values and query predicates, respectively. Figure 2 illustrates this architecture. For clarity the figure depicts a single client; Mosaic applications typically include multiple clients with shared Params or Selections.

Though various query languages might be used, given the ubiquity of the relational data model and the availability of scalable databases, we focus on SQL (Structured Query Language). Our reference implementation uses DuckDB [34] as the backing data source. DuckDB is a high-performance open-source analytic database that can run both server-side and in the browser via WebAssembly (WASM) [21].

3.1 Clients

Mosaic *Clients* are responsible for publishing their data needs and performing data processing tasks—such as rendering a visualization—once data is provided by the Coordinator. Clients typically take the form of Web (HTML/SVG) elements, but are not required to.

Figure 3 depicts a Mosaic lifecycle. Upon registration, the Coordinator calls the client `fields()` method to request an optional list of fields, consisting of table and column names as well as statistics such as the row count or min/max values. The Coordinator queries the Data Source for requested metadata (e.g., column type) and summary statistics as needed, and returns them via the client `fieldInfo()` method.

Next, the Coordinator calls the client `query()` method. The return value may be a SQL query string or a structured object that produces a query upon string coercion. Mosaic includes a query builder API that simplifies the construction of complex queries while enabling query analysis without need of a parser. The `query` method takes a single argument: an optional `filter` predicate (akin to a SQL `WHERE` clause) indicating a data subset. The client is responsible for incorporating the filter criteria into the returned query. Before the Coordinator submits a query for execution, it calls `queryPending()` to inform the client. Once query execution completes, the Coordinator returns data via the client `queryResult()` method or reports an error via `queryError()`.

Clients can also request queries in response to internal events. The client `requestQuery(query)` method passes a specific query to the Coordinator with a guarantee that it will be evaluated. The client `requestUpdate()` method instead makes throttled requests for a standard `query()`; multiple calls to `requestUpdate()` may result in only one query (the most recent) being serviced. Finally, clients may expose a `filterBy` Selection property. The predicates provided by `filterBy` are passed as an argument to the client `query()` method.

3.2 Coordinator

The *Coordinator* is responsible for managing client data needs. Clients are registered via the Coordinator `connect(client)` method, and similarly removed using `disconnect()`. Upon registration, the event lifecycle begins. In addition to the `fields` and `query` calls described above, the Coordinator checks if a client exposes a `filterBy` property, and if so, adds the client to a *filter group*: a set of clients that share the same `filterBy` Selection. Upon changes to this selection (e.g., due to interactions such as brushing or zooming), the Coordinator collects updated queries for all corresponding clients, queries the Data Source, and updates clients in turn. This process is depicted in Figure 3.

As input events (and thus Selection updates) may arrive at a faster rate than the system can service queries, the Coordinator also throttles updates for a filter group. If new updates arrive while a prior update is being serviced, intermediate updates are dropped in favor of the most recent update. The Coordinator additionally performs optimizations including caching and data cube indexing, detailed later in Section 6.

3.3 Data Source

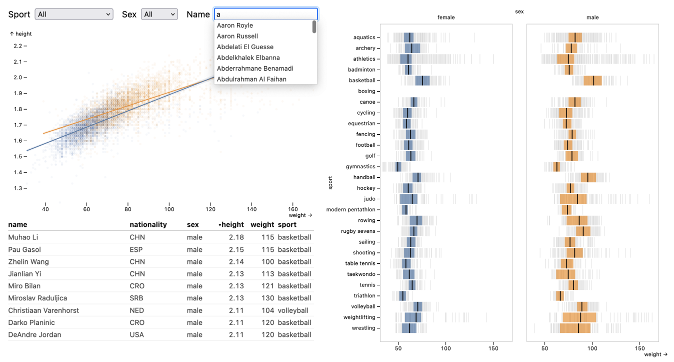
The Coordinator submits queries to a *Data Source* for evaluation, using an extensible set of database connectors. By default Mosaic uses DuckDB as its backing database and provides connectors for communicating with a DuckDB server via Web Sockets or HTTP calls, with DuckDB-WASM in the browser, or through Jupyter widgets to DuckDB in Python. For data transfer, we default to the binary Apache Arrow format [1], which enables efficient serialization of query results with no subsequent parsing overhead. While the socket and HTTP connectors also support JSON, this is more costly to serialize, results in larger payloads, and must be parsed on the client side.

3.4 Params and Selections

Params and *Selections* support cross-component coordination. Akin to Vega's *signals* [36] and Improvise's *live properties* [42], Params are reactive variables that hold scalar values (accessible via the `value` property) and broadcast updates upon changes. Params can parameterize Mosaic clients and may be updated by input widgets. The Mosaic architecture is agnostic as to where Param and Selection updates come from. As we will illustrate later, updates may be initiated by clients themselves or by dedicated interactor components.

A Selection is a specialized Param that manages one or more predicates (Boolean-valued query expressions), generalizing Vega-Lite's selection abstraction [35]. Interaction components update selections by providing a *clause*, an object consisting of the *source* component providing the clause, a set of *clients* associated with the clause, a query *predicate* (e.g, the range predicate `column BETWEEN 0 AND 1`), a corresponding *value* (e.g., the range array `[0,1]`), and an optional *schema* providing clause metadata (used for optimization, see Section 6). Upon update, any prior clause with the same *source* is removed and the new, most recent clause (called the *active* clause) is added. Selections override the Param `value` property to return the active clause *value*, making Selections compatible where standard Params are expected.

Selections expose a `predicate(client)` function that takes a client as input and returns a correponding predicate for filtering the client's data. Selections apply a *resolution* strategy to merge clauses into client-specific predicates. The *single* strategy simply includes only the most recent clause. The *union* strategy creates a disjunctive predicate, combining all clause *predicates* via Boolean `OR`. Similarly, the *intersect* strategy performs conjunction via Boolean `AND`. Any of these



name	nationality	sex	▾height	weight	sport
Muhao Li	CHN	male	2.18	115	basketball
Pau Gasol	ESP	male	2.15	115	basketball
Zhelin Wang	CHN	male	2.14	100	basketball
Jianlian Yi	CHN	male	2.13	113	basketball
Miro Bilan	CRO	male	2.13	121	basketball
Miroslav Raduljica	SRB	male	2.13	130	basketball
Christiaan Varenhorst	NED	male	2.11	104	volleyball
Darko Planinic	CRO	male	2.11	120	basketball
DeAndre Jordan	USA	male	2.11	120	basketball

Fig. 4: Interactive dashboard of Olympic athletes. *Left:* Input widgets with data-driven content drive filtering by sport, sex, and name. A scatter plot with regression lines conveys the relationship between weight and height. A sortable table component shows record-level details. *Right:* A faceted plot shows weight distributions by sport and sex. Interquartile ranges overlay the raw values. Mosaic manages all data access, interactive filtering, regression, and IQR calculations.

```
const sel = Selection.intersect()
hconcat(
 menu({label: 'Sport', from: 'athletes', column: 'sport', as: sel}),
 menu({label: 'Sex', from: 'athletes', column: 'sex', as: sel}),
 search({label: 'Name', from: 'athletes', column: 'name', as: sel})
)
```

Fig. 5: JavaScript specification of the inputs along the top of Figure 4.

strategies may also *crossfilter* by omitting clauses where the *clients* set includes the input argument to the `predicate()` function. This strategy enables filtering interactions that affect views other than the one currently being interacted with.

Both Params and Selections support `value` event listeners, corresponding to value changes. Selections additionally support `activate` events, which provide a clause indicative of likely future updates. For example, a brush interactor may trigger an activation event when the mouse cursor enters a brushable region, providing an example clause prior to any actual updates. As discussed in Section 6, activation events can be used to implement optimizations such as prefetching indexes.

3.5 Extensibility and Interoperability

As a middle-tier architecture, Mosaic is designed to be extensible for both practical purposes and research experimentation. The Client API was carefully designed to offload all query management responsibility to the Coordinator. As a result, Coordinator-specific optimizations (Section 6) and even the entire Coordinator implementation itself can be replaced without affecting client implementations.

While our implementation primarily targets SQL, DuckDB, and Apache Arrow, any of these pieces might be replaced. We strive to use standard SQL constructs, enabling other relational databases. Clients can use alternative query languages if an augmented Coordinator and Data Source support them. Similarly, other data transfer formats could be used, so long as the result either conforms to standard JavaScript iterables or the clients involved can handle the specialized format.

Mosaic is intended to support integration across a variety of diverse clients. In the next sections, we demonstrate how both standard input widgets and a full grammar of interactive graphics can be implemented and flexibly interoperate using the Mosaic API. In the future these could be further augmented with clients for custom visualization tasks (e.g., graph drawing) and rendering methods (e.g., 3D WebGL views).

4 Mosaic Input Components

Inspired by Observable's inputs package [3], we developed a set of common input widgets and a table viewer. Each is implemented as a Mosaic client and use Params and Selections for linked interactions. Figure 4 shows a dashboard that incorporates `menu`, `search`, and `table` components, while Figure 8 incorporates a `slider` component.

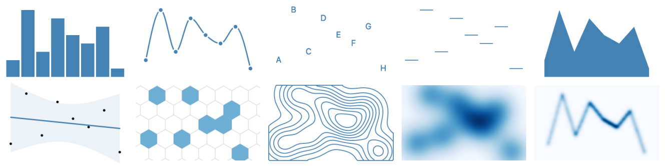


Fig. 6: A subset of mark types supported by *vgplot*: `bar`, `line`, `text`, `tick`, `area`; `regression`, `hexbin`, `contour`, `raster`, and `denseLine`.

The `slider`, `menu`, and `search` inputs support dual modes of operation: they can be manually configured or they can be backed by a database table. If a backing table and column are specified, the `slider`'s `query()` method gets the minimum and maximum column values to parameterize the slider. The `menu` and `search` components instead query for distinct column values, and use those to populate the menu or autocomplete options (Figure 4), respectively.

All input widgets can write updates to a provided Param or Selection. Param values are updated to match the input value. Selections are provided a predicate clause (Section 3.4). This linking can be bidirectional: an input component will also subscribe to a Param and track its value updates. Two-way linking is also supported for Selections using *single* resolution, where there is no ambiguity regarding the value.

The `table` component provides a sortable, scrollable table grid view. If a set of backing columns is provided, the table `fields()` method requests metadata for those columns. If a backing table is provided without explicit columns, `fields()` instead requests *all* table columns. The returned metadata is used to populate the table header and guide formatting and alignment by column type.

To limit data transfer, the table `query()` method requests rows in batches using SQL `LIMIT` and `OFFSET` clauses. As a user scrolls the table, the `requestQuery()` method is used to request a new batch with the proper offset. To reduce latency, the component further requests that the Coordinator *prefetch* the subsequent data batch.

Table components are sortable (Figure 4): clicking a column header toggles ascending and descending order. When sort criteria change, the current data is dropped and `requestQuery()` is called to fetch a sorted data batch. As a user scrolls, these sort criteria persist. If provided, the `filterBy` Selection is used to filter table content.

5 vgplot: An Interactive Visualization Grammar

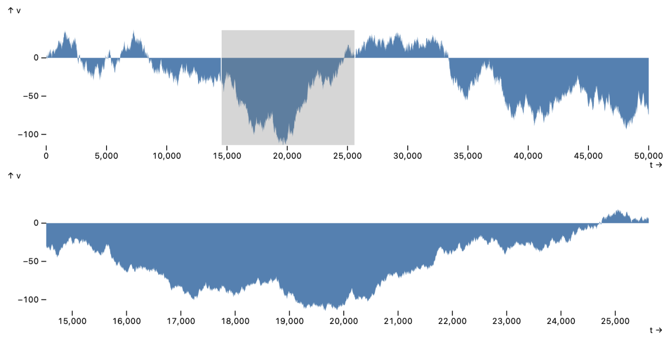
To demonstrate extensibility and interoperability in Mosaic, we developed *vgplot*, a grammar of graphics [45] in which graphical marks are Mosaic clients. As its name suggests, vgplot combines concepts from multiple grammars. vgplot adopts Observable Plot's chart semantics [4], which incorporate ideas from both ggplot2 [43] and Vega-Lite [35]. Like Vega-Lite, vgplot supports interaction and declarative specification either using an API or standalone JSON/YAML specs. However, as vgplot is based on Mosaic, it can readily interoperate with any other Mosaic clients, such as the input components of Section 4.

5.1 Plot Elements

A **plot** is a visualization in the form of a Web element. As in other grammars, a plot consists of *marks*—graphical primitives such as bars, areas, and lines—that serve as chart layers. Each plot includes a set of named *scale* mappings such as `x`, `y`, `color`, `opacity`, etc. Plots can facet the `x` and `y` dimensions, producing associated `fx` and `fy` scales. Plots are rendered to SVG output by marshalling a specification and passing it to Observable Plot. A plot is defined as a list of directives defining plot *attributes*, *marks*, *interactors*, or *legends*.

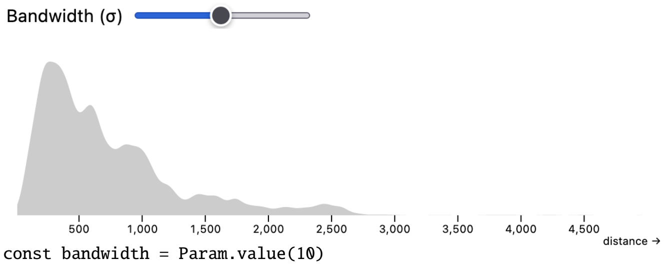
Attributes configure a plot (`width`, `height`) and its scales (e.g., `xDomain`, `colorRange`, `yTickFormat`). Attributes may be Param-valued, in which case a plot updates upon Param changes. vgplot also introduces a `Fixed` scale domain setting (e.g., `xDomain(Fixed)`), which instructs a plot to first calculate a scale domain in a data-driven manner, but keep that domain fixed across subsequent updates. Fixed domains prevent disorienting scale domain "jumps" that hamper comparison across filter interactions (a limitation of Vega-Lite).

Marks are graphical primitives, often with accompanying data transforms, that serve as chart layers. Each vgplot mark is a Mosaic client



```
const brush = Selection.intersect()
const channels = { x: 't', y: 'v', fill: 'steelblue' }
vconcat(
 plot(
  areaY(from('walk'), channels),
  intervalX({ as: brush })
 ),
 plot(
  areaY(from('walk', { filterBy: brush }), channels),
  yDomain(Fixed)
 )
)
```

Fig. 7: An overview+detail visualization of a 50,000 point time-series and JavaScript API specification. The `areaY` mark uses M4 optimization [19] to reduce the number of drawn points by over an order of magnitude.



```
const bandwidth = Param.value(10)
vconcat(
 slider({ as: bandwidth, min: 0.1, max: 100, step: 0.1 }),
 plot(
  densityY(from('flights'), { x: 'distance', bandwidth }),
  yAxis(null)
 )
)
```

Fig. 8: 1D kernel density estimate (KDE) of airline miles flown. Linear binning is performed in database, subsequent smoothing is performed in browser. *Param* updates to the kernel bandwidth from the `slider` are calculated immediately, without having to re-query the database.

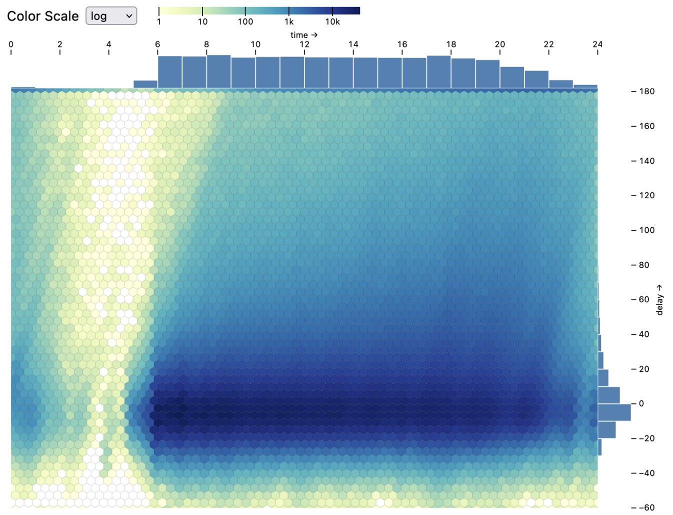
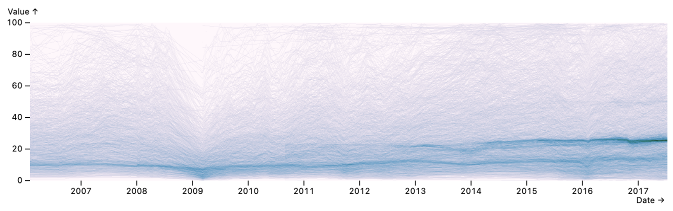


Fig. 9: Hexagonal bins of airline delay by scheduled departure time, alongside marginal histograms. Hex binning and aggregation are performed in database. Interactive changes to the color scale (e.g., linear, log, or square root scale) are processed immediately in browser.



```
plot(
 denseLine(from('stocks_after_2006'), {
  x: 'Date', y: 'Close', z: 'Symbol', fill: 'density'
 }),
 colorScheme('pubugn')
)
```

Fig. 10: An arc-length normalized density line chart [28] for 240k monthly stock price values. Note high points across the top, the 2008 crash, and distinct bands of $25 and $15 stocks.

that produces queries for needed data. Figure 6 shows some supported mark types. Marks accept a data source definition and a set of supported options, including encoding *channels* (such as `x`, `y`, `fill`, and `stroke`) that can encode data *fields*. A data field may be a column reference or query expression, including dynamic *Param* values. Common expressions include aggregates (`count`, `sum`, `avg`, `median`, etc.), window functions (e.g., moving averages), date functions, and a `bin` transform. Field expressions are specified using Mosaic's SQL builder methods.

Basic marks, such as `dot`, `bar`, `rect`, `cell`, and `text` mirror their namesakes in Observable Plot [4]. Variants such as `barX` and `rectY` indicate spatial orientation and data type assumptions. For example, `barY` indicates vertical bars—continuous `y` over an ordinal `x` domain—whereas `rectY` indicates a continuous `x` domain. Basic marks follow a straightforward `query()` construction process: Iterate over all encoding channels. If no aggregates are found, `SELECT` all fields directly. If aggregates are present, include non-aggregate fields as `GROUP BY` criteria. If provided, map the `filter` argument to a `WHERE` clause. For more details on query generation, see Appendix A.

The `area` and `line` marks connect consecutive sample points. Figure 7 presents an overview+detail area chart. The queries for spatially oriented marks (`areaY`, `lineX`) apply M4 optimization [19, 20]. The `query()` method determines the pixel resolution along the major axis and performs perceptually faithful, pixel-aware binning of the series, limiting the number of drawn points. This optimization offers dramatic data reductions, potentially spanning multiple orders of magnitude.

The `regression` mark (Figure 4) visualizes linear regression fits. Statistical calculations are performed in a single aggregate `query()`. The mark then draws regression lines and confidence intervals.

The `densityX/Y` marks perform 1D kernel density estimation. Figure 8 shows a `densityY` mark, with a `slider`-bound `bandwidth` Param. The generated `query()` performs *linear binning* [18, 41], which proportionally assigns point weights between adjacent bins to provide greater estimation accuracy [17]. Subqueries for "left" and "right" bins are aggregated into a 1D grid, then smoothed in-browser using Deriche's accurate linear-time approximation [14, 17].

The `density2D`, `contour`, and `raster` marks compute densities over a 2D domain. Binning and aggregation is performed in database, while dynamic changes of bandwidth, contour thresholds, and color scales are handled immediately in the browser. The `hexbin` mark pushes hexagonal binning and aggregation to the database (Figure 9); color and size channels may map to `count` or other aggregate functions.

Rather than point densities, the `denseLine` mark (Figure 10) plots densities of line segments [28]. The `query()` method pushes line rasterization and aggregation to the database with a multi-part process described in Appendix A.5. We added the `denseLine` mark late in our development process to test extensibility, overriding the `raster` mark with a new `query()` method. As a result, the `denseLine` mark inherits `raster`'s smoothing capability to create curve density estimates [23].

Interactors imbue plots with interactive behavior. Most interactors listen to input events to update bound Selections. The `toggle` interactor selects individual points (e.g., by click or shift-click) and generates a selection clause over specified fields of those points. Other interactors include: `nearestX/Y` to select the nearest value along the `x` and/or `y` channel, `intervalX/Y` to create 1D or 2D interval brushes (Figure 7), and `panZoom` interactors that produce interval selections over corresponding `x` or `y` scale domains (c.f., Vega-Lite [35]). `interval` interactors accept a `pixelSize` parameter that sets the brush resolution: values may snap to a grid whose bins are larger than screen pixels, which can be leveraged to optimize query latency (Section 6). The `highlight` interactor updates the rendered state of a visualization in response to a Selection, querying for a selection bit vector and then modifying unselected items to a translucent, neutral gray color.

Legends can be added to a plot or as a standalone element. The `name` directive gives a plot a unique name, which a standalone legend can reference (`legendColor({for:'name'})`) to avoid respecifying scale domains and ranges. Legends also act as interactors, taking a bound Selection as a parameter. For example, discrete legends use the logic of the `toggle` interactor to enable point selections. Two-way binding is supported for Selections using *single* resolution, enabling legends and other interactors to share state, as in Figure 11.

Finally, the **layout** functions `vconcat` (vertical concatenation) and `hconcat` (horizontal concatenation) enable multi-view layout. Layout helpers can be used with plots, inputs, and arbitrary Web content such as images and videos. To ensure spacing, the `vspace` and `hspace` helpers add padding between elements in a layout.

5.2 Declarative Specification

In addition to a JavaScript API, all vgplot constructs, Mosaic inputs, and layout helpers can be authored using a JSON specification format. YAML, a more human-readable format that maps to JSON, can also be used. As a result, Mosaic applications can be conveniently generated from other programming languages. Figure 11 shows integration of Mosaic with Jupyter notebooks. The Mosaic release includes a notebook widget which, given a YAML or JSON spec, creates a DuckDB connection in the Jupyter Kernel that communicates via Jupyter Comms with a notebook output cell running Mosaic JavaScript code. The Python DuckDB API can directly access Pandas data frames, supporting easy and efficient integration into Jupyter workflows. In addition, Param and Selection values are shared with the kernel, accessible as Python variables. We implemented the Jupyter connector in about a hundred Mosaic-specific lines of code, showing the ease of adapting Mosaic to new environments. In the future, a Python API akin to Altair [40] could programmatically generate Mosaic/vgplot JSON specifications.

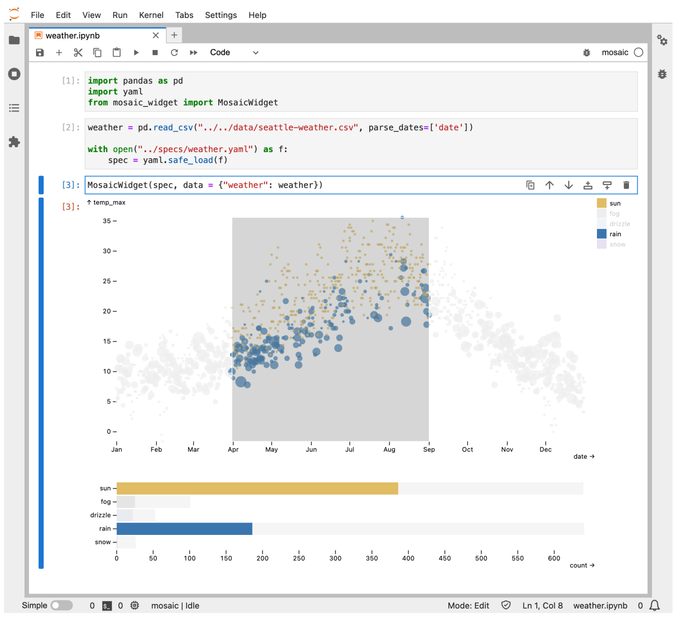
6 Coordinator Optimizations

Mosaic supports query optimization at multiple levels. For optimizations that are specific to a given visualization type (such as M4 for line and area marks), clients can perform local optimizations as part of query generation. The Coordinator, in turn, provides support for optimizations that extend over multiple views and interaction cycles.

6.1 Query Caching, Consolidation, and Prefetching

Query caching optimizes for repeated queries. The Coordinator uses a standard LRU cache, storing query results keyed by SQL query text. Future research could explore alternative cache eviction policies that account for latency, dataset size, or other factors. Already, we notice greater benefits for caching than first anticipated: we observe that interactive states are often revisited (even during brushing), and `highlight` bit vector queries are often shared across plots.

The Coordinator also applies **query consolidation**, merging overlapping queries into a single query to reduce processing and network time. As multiple queries tend to be issued in tandem (for example upon initialization), the Coordinator waits one animation frame, collects incoming queries, and merges those that query the same backing table and `GROUP BY` dimensions. Upon query completion, the Coordinator parcels out appropriate projections to clients. Query consolidation is valuable for optimizing multiple views that show data at the same level of aggregation. For example, data for a 4x4 scatter plot matrix requires only 1 consolidated query rather than 16 separate queries.



```
[1]: import pandas as pd
     import yaml
     from mosaic_widget import MosaicWidget

[2]: weather = pd.read_csv("../../data/seattle-weather.csv", parse_dates=['date'])

     with open("../specs/weather.yaml") as f:
         spec = yaml.safe_load(f)

[3]: MosaicWidget(spec, data = {"weather": weather})
```

```
params:
  click: { select: single }
  domain: [sun, fog, drizzle, rain, snow]
  colors: ['#e7ba52', '#a7a7a7', '#aec7e8', '#1f77b4', '#9467bd']
vconcat:
- plot:
  - mark: dot
    data: { from: weather, filterBy: $click }
    x: { dateMonthDay: date }
    y: temp_max
    fill: weather
    r: precipitation
  - { select: intervalX, as: $range }
  - { select: highlight, by: $range, fill: '#eee' }
  - { legend: color, as: $click, columns: 1 }
  xyDomain: Fixed
  colorDomain: $domain
  colorRange: $colors
- plot:
  - mark: barX
    data: { from: weather, filterBy: $range }
    x: { count: }
    y: weather
    fill: weather
  - { select: toggleY, as: $click }
  - { select: highlight, by: $click }
  xDomain: Fixed
  colorDomain: $domain
  colorRange: $colors
```

Fig. 11: Interactive exploration of Seattle weather with YAML specification making use of data transforms, filtering, highlighting, and linked legends. Param and Selection references use a $ prefix. If a Selection is not explicitly defined, one with the *intersect* resolution strategy is created.

Prefetching reduces latency by querying data before it is needed. Clients can issue queries using the Coordinator's `prefetch` method. These requests are enqueued with a lower priority than standard queries. Prefetched results are then stored in the Coordinator's cache, available for subsequent requests. Clients may `cancel` prefetch requests in response to interactive updates. As described next, the Coordinator also performs automatic prefetching when building data cube indexes.

6.2 Data Cube Indexes

To optimize filtering of aggregated data, the Coordinator automatically builds **indexes** in the form of small *data cubes* [16] that can rapidly service interactive queries. Upon updates to a *filter group* Selection, the Coordinator analyzes the active selection clause and client `query()` output. If generated queries involve group-by aggregation using supported aggregate functions (currently `count`, `sum`, `avg`, `min`, and `max`), the Coordinator will rewrite queries to create data cube index tables.

Akin to Falcon [31], Mosaic builds indexes that pre-aggregate data between an active view and another filtered view. However, Mosaic extends Falcon in multiple ways. Rather than assume a prespecified set of visualizations, Mosaic *automatically* applies data cube indexing based on observed queries and selections. Next, index tables are built and stored in the database, rather than shipped to the browser. Queries to an index thus use the more scalable database for processing and indexed tables are cached in-database for reuse. In contrast to Falcon, we do not create summed area tables [13], which use a dense representation that can overwhelm available memory (Section 8.2). Foregoing summed area tables also permits indexing of aggregate functions other than `count` and `sum`, including `avg`, `min`, and `max`, in a straightforward way.

For example, given an interval selection over the column `$u`, the Coordinator uses the number of `$pixels` and the minimum and maximum possible brush values in the data domain `[$bmin, $bmax]` to produce pixel-level bins for all possible brush positions. The following query creates an index table for brush interactions between a linear, one-dimensional selection clause and a single histogram over column `$v` (with initial value `$v0` and bin size `$step`):

```
CREATE TEMP TABLE IF NOT EXISTS cube_index_a097caa4 AS
SELECT
  $v0 + $step * FLOOR(($v - $v0) / $step) AS x1,
  $v0 + $step * (FLOOR(($v - $v0) / $step) + 1) AS x2,
  COUNT(*) AS y,
  FLOOR($pixels * ($u - $bmin) / ($bmax - $bmin)) AS activeX
FROM $table
GROUP BY x1, x2, activeX
```

The table name includes a hash of the `SELECT` statement that creates the table, enabling easy reuse. If the table already exists it is not re-created. Upon selection updates, the Coordinator issues queries to the index table. For data-space brush endpoints `$b0` and `$b1`, the index query is:

```
SELECT x1, x2, SUM(y) FROM cube_index_a097caa4
WHERE activeX BETWEEN
 FLOOR($pixels * ($b0 - $bmin) / ($bmax - $bmin)) AND
 FLOOR($pixels * ($b1 - $bmin) / ($bmax - $bmin))
GROUP BY x1, x2
```

The size of the index table is bound by the number of bins (binned `$v` steps and `$pixels`), not the size of the input data. Two-dimensional brushes are handled similarly, resulting in both `activeX` and `activeY` index table columns. To index client queries with subqueries or common table expressions, the indexer performs *subquery pushdown* of interactive dimensions (e.g., brush pixel bins) so that these values pass from subqueries to the top-level aggregation (see Appendix A.4).

Index tables include bins for each interactive dimension of an active selection clause. To determine these dimensions, the indexer uses metadata provided by a Selection clause's *schema* property. The *schema* indicates the abstract predicate type: one of *interval* or *point*. The schema for *interval* types also includes spatial `x`/`y` scale definitions and the interactive pixel size. Larger interactive pixels lower the interactive resolution, requiring fewer index bins [31]. The indexer determines interactive extents from the scale ranges and interactive pixel size, and bins non-linear scales such as log, sqrt, and symlog based on the scale type. In sum, Selection clauses provide not only query predicates, but also the encoding information necessary for automatic optimization.

Data cube indexes can dramatically reduce interactive latency [25, 31]. While common cases include cross-filtered histograms (Figure 13), automatic application also optimizes cases we did not initially expect, including `hexbin` and `denseLine` filtering over fixed domains (e.g., brushing histograms in Figure 9) and the weather example in Figure 11 (index-optimized filtering of the lower plot).

Index construction can sometimes be costly (Section 8.2). To **prefetch** indexes, each *filter group* monitors Selection `activate` events, which interactors trigger when a pointer enters a view. Upon activation, the Coordinator submits queries to build index tables *before* an interval or point selection is initiated. For longer construction times, the Coordinator and Mosaic DuckDB server can record interactions to **precompute** bundles of queries and index tables for future use.

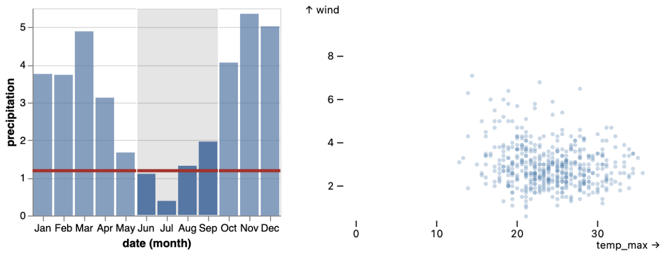
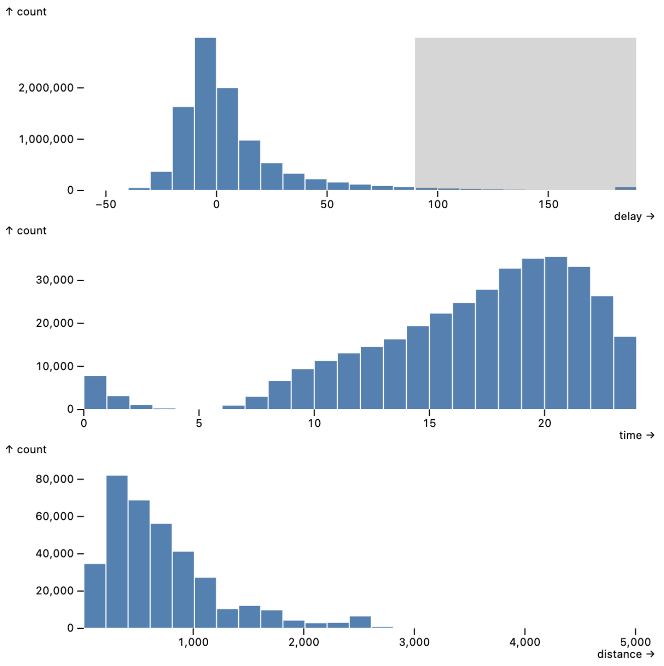


Fig. 12: Linked weather plots. A Vega-Lite plot (left) populates a Mosaic selection to drive an average line (red) and filter a vgplot chart (right).



```
const brush = Selection.crossfilter()
vconcat(['delay', 'time', 'distance'].map(column =>
 plot(
  rectY(from('flights', { filterBy: brush }), {
   x: bin(column), y: count(), fill: 'steelblue', inset: 0.5
  }),
  intervalX({ as: brush }), xDomain(Fixed)
 )
))
```

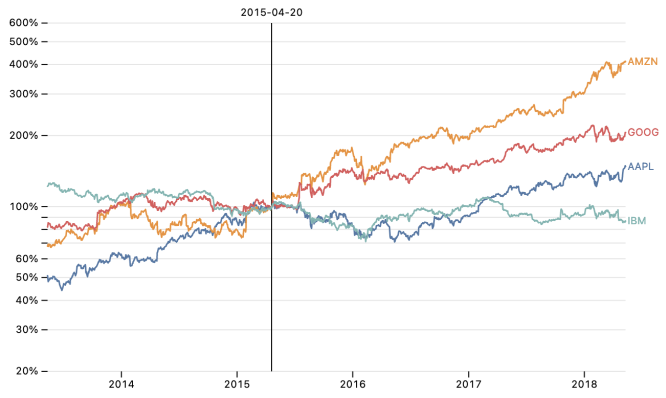
Fig. 13: Histograms showing arrival delay, departure time, and distance flown for 10M flight records. An interval selection reveals that flights departing later in the day are more likely to be delayed.

7 Examples

We demonstrate Mosaic's expressivity and concision through examples. Earlier instances include scalable area charts (Figure 7) and density plots (Figures 8–10). Here we provide more complex examples. For additional examples, see Appendix B in the supplemental material.

Seattle Weather. Figure 11 shows interactive exploration of Seattle weather, adapted from a Vega-Lite example. The top plot uses a `dot` mark and `intervalX` and `highlight` interactors. The `x` channel field uses a `dateMonthDay` transform to map multi-year data to a single year. The bottom plot uses a `barX` mark to count days per weather type, with a `filterBy` selection driven by a selected interval. Both the bars and legend serve as `toggle` interactors that drive a `highlight` for the bars and a `filterBy` selection for the `dot` above. The legend and `barX` marks share a selection and update in tandem.

Vega-Lite Integration. Figure 12 demonstrates tool interoperability: a Vega-Lite plot of precipitation filters a vgplot scatter plot of temperature and wind. Given the Vega-Lite specification, this example extracts transforms to generate Mosaic clients, and maps internal Vega-Lite selections to Mosaic selections. As a result, Mosaic mediates both data access and linked interactions for the Vega-Lite plot.



```
const point = Param.value(new Date('2015-04-20'))
plot(
 ruleX({ x: point }),
 textX({ x: point, text: point, frameAnchor: 'top', ... }),
 lineY(from('stocks'), { x: 'Date', stroke: 'Symbol',
  y: sql`Close / (SELECT Close FROM stocks
         WHERE Symbol = source.Symbol AND Date = ${point})` }),
 nearestX({ as: point }),
 yScale('log'), yGrid(true), yTickFormat('%')
)
```

Fig. 14: Stock returns normalized to a hypothetical purchase date. On mouse move the prices are renormalized for the nearest date by a parameterized expression created with Mosaic's `sql` template literal.

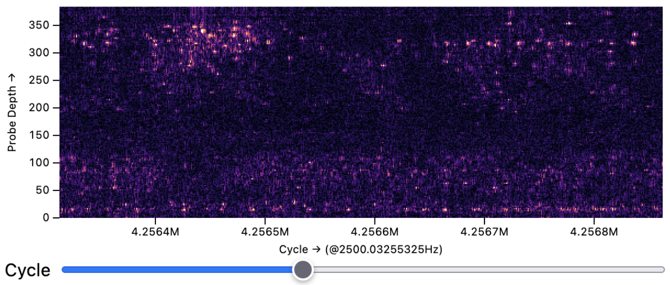


Fig. 15: Electrical recordings from a mouse brain. Prefetching enables smooth panning of a `raster` over 10.7M time samples (4.1B rows).

Olympic Athlete Dashboard. Figure 4 shows an interactive dashboard of Olympic athlete data, showing interoperation of input widgets, tables, regression models, and a multi-layer faceted plot. All components share a single *intersect* selection for filtering across views.

Flight Data. Figure 13 shows linked histograms for 10M flights [9]. Each consists of a `rectY` mark with a `bin` transform, `count` aggregate, and a shared `filterBy` *crossfilter* selection driven by `intervalX` brush interactors. Interactive updates are served by data cube indexes.

Normalized Stock Prices. Figure 14 shows stock prices on a log scale. A `nearestX` interactor selects the nearest market day; this value parameterizes a field expression that normalizes prices to show returns if one had invested on that day. Normalization is performed in database by a one-line expression with a scalar subquery. A similar Vega-Lite example requires an extra transform pipeline with a lookup join and derived calculation; the Mosaic version is simpler and more efficient.

Neuron Spike Data. Figure 15 shows a `raster` of electrical activity measured by our neuroscience collaborators using a probe in a mouse brain. Measurements of 384 sensor channels across 10.7M time points (4.1B rows) are drawn from an 18GB Parquet file. To smoothly pan the display, a tiled variant of the `raster` mark queries adjacent batches of data using the Coordinator `prefetch` method.

Gaia Star Catalog. The Gaia star catalog is a sky survey of over 1.8B stars [15], from which we extracted selected columns into a 34GB Parquet file. Figure 1 visualizes all records with a `raster` sky map, `rectY` histograms of magnitude and parallax, and a `raster` Hertzsprung-Russell diagram of stellar color vs. magnitude. The map uses the equal-area Hammer projection, performed in the database. The views are linked by a *crossfilter* selection driven by `interval` interactors, automatically optimized by data cube indexes.

8 Performance Benchmarks

To further assess Mosaic, we conducted performance benchmarks examining both initial rendering and interactive update times. Unless otherwise noted, we ran all benchmarks on a single MacBook Pro laptop (16-inch, 2021) with an Apple M1 Pro processor and 16GB RAM, running MacOS 12.6 and Google Chrome 109.0.5414.119. Server-side DuckDB (v0.6.1) instances used a standard configuration of all processor cores and a maximum of 80% RAM, with network communication over a socket connection and Apache Arrow data transport.

8.1 Initial Chart Rendering

We first compare initial rendering times for Mosaic using either WASM or a local server against Vega [35,36], VegaFusion [22], and Observable Plot [4]. All tests use a synthetic dataset with four columns: an index counter (`i`), a categorical variable of cardinality 20 (`w`), a uniform random variable (`u`), and a random walk value (`v`). We vary the dataset size from 10 thousand to 10 million rows.

We measure render times for six visualizations: *bars* of average `v` grouped by `w`, a linear *regression* plot of `v` on `i`, a *2D histogram* binned on `u` and `v`, an *area chart* of `v` over `i`, and finally *density contours* and *hexbins* over the domain of `u` and `v`. We chose these conditions to cover both common visualization needs and a range of scalable visualization types. Plots in the Vega condition are created using Vega-Lite [35], except for density contours. Vega-Lite does not support contour plots so we use Vega directly. As neither Vega nor Vega-Lite support hexagonal binning, the Vega conditions omit the *hexbins* visualization.

Benchmark results are plotted in Figure 16. Vega and Observable Plot scale similarly, but with a slight advantage for Plot. Plot renders to SVG directly and does not have Vega's overhead of constructing a reactive dataflow graph and intermediate scenegraph. VegaFusion performs server-side optimization for *bars* and *2D histograms* only, otherwise providing results identical to Vega.

Meanwhile, Mosaic roundly outperforms these tools, often by one or more orders of magnitude. Mosaic WASM fares well at lower data volumes, but at larger sizes is limited by WebAssembly's lack of parallel processing. DuckDB aggregate query performance drives Mosaic's improvements for the *bars*, *regression*, *2D histogram*, and *density contours* charts. The *hexbins* example benefits from the `hexbin` mark expressing hexagonal binning calculations within a SQL query.

All non-Mosaic tools fail to render *area charts* of larger datasets, as Chrome will not draw an SVG path with a million or more points. Here the Mosaic `area` mark client's use of M4 enables greater scale, as the number of drawn points is a function of available screen pixels.

8.2 Interactive Updates

Next we assess interactive performance. We drop the Vega and Observable Plot conditions, as Vega can not handle the dataset sizes tested and Plot does not support interaction. We compare Mosaic using WASM, a local server, and a remote server with 2 12-core 2.6 GHz Intel Xeon

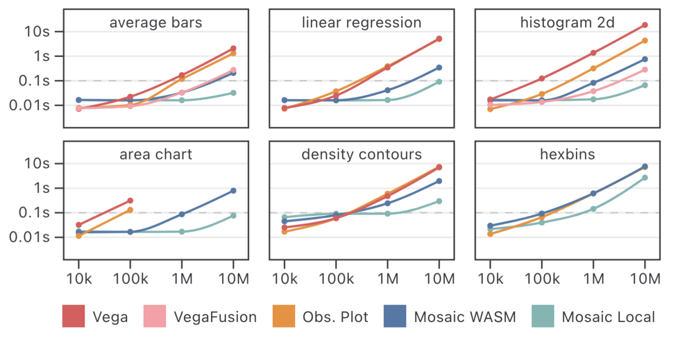


Fig. 16: Initial rendering performance. Median times are shown, interquartile ranges are smaller than the plotted dots. Mosaic provides order-of-magnitude performance improvements over Vega or Observable Plot for a range of visualizations. VegaFusion only optimizes the *average bars* and *2D histogram* conditions. Mosaic WASM does not scale as well as a local server, in part due to the lack of parallel processing.

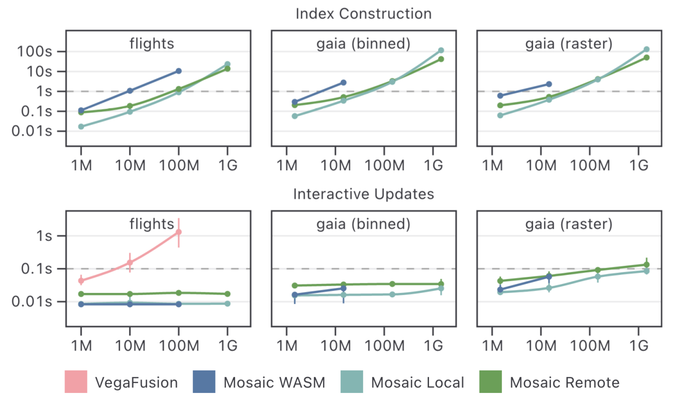


Fig. 17: Index construction (top) and interactive update performance (bottom). Median times and interquartile ranges are shown. Mosaic server configurations maintain interactive update rates at high data volumes, though start to degrade as indexes for raster data become denser.

processors and 512GB RAM running Rocky Linux 9.1, accessed from 2 miles away over a WiFi router and fiber optic Internet connection.

We measure both data cube index creation and subsequent interactive update times across three real-world applications: the *flights* dataset (Figure 13), the *Gaia* star catalog (Figure 1), and a simplified Gaia example (*gaia-bins*) that replaces high-resolution rasters with simpler 2D histograms (as in Falcon's evaluation [31]). We vary dataset sizes from millions to billions of rows. As our *flights* dataset has 10M rows, we duplicate data to construct larger 100M and 1B row tables. For Gaia, we use 0.1%, 1%, 10%, and 100% samples of the 1.8B row dataset.

Figure 17 plots the results. Server-based Mosaic instances provide performant index construction (< 5 seconds for up to 200M records) and interactive updates (100ms or faster). Network latency causes the remote server to underperform until larger (100M+) dataset sizes, at which point the additional memory and parallelism provide benefits. Mosaic WASM runs out of memory for 1B flights and for Gaia samples of 10% and up. We tested VegaFusion on the *flights* data only (VegaFusion does not optimize density rasters). VegaFusion does not perform indexing, leading to over 1 second latency at 100M. We attempted to test 1B points, but canceled after 10 minutes passed with no updates.

Across Mosaic conditions, index creation time increases with dataset size. These indexes optimize subsequent updates. In the Gaia raster condition, update times begin to slow as the indexes for cross-filtering between raster displays become large. Here the maximum possible data cube size is 3.6B rows (300x200 interactive pixels times 300x200 raster cells to render). In practice many fewer rows are needed due to sparsity; nevertheless, index sparsity decreases as the dataset size increases. Techniques that materialize a dense index, including imMens [25] and Falcon [31], will run out of memory and fail at this scale. Mosaic remains capable of providing an interactive experience.

9 Discussion

Mosaic is a middle-tier architecture that coordinates interactive data-driven components and scalable data stores. Mosaic Clients publish data needs as declarative queries while interactions are coordinated via dynamic Params and predicate-based Selections. The Mosaic Coordinator mediates between client implementations and data management, while performing automatic query optimization. We demonstrate interactive exploration of large-scale data using Mosaic-based input widgets and *vgplot*, a grammar of interactive graphics. A range of examples highlight Mosaic's extensibility and interoperability, while performance benchmarks show significant scalability and interactive performance improvements over existing web-based visualization tools.

Whereas prior work on scalable visualization has largely contributed individual techniques, Mosaic provides a framework that integrates databases, query optimization, and an expressive set of visualization abstractions within a unified system. The Mosaic Coordinator provides *automatic* optimizations such as caching, query consolidation, and data cube indexes, while individual clients can perform *local* optimizations (such as M4 or Coordinator-assisted prefetching) based on known visualization or interaction semantics. The vgplot library also illustrates how many visualization transforms (including a variety of density displays) can be implemented as database queries.

One limitation is the time required to build indexes for larger (500M+ row) datasets. Rather than assume a "cold start", Mosaic also supports index precomputation; at most a few minutes are needed for the full Gaia catalog. Slower interactive updates are partially compensated by Mosaic's event throttling: one can interact in real-time (e.g., adjust brushes) though the data may update after a short but noticeable delay. As noted earlier, Mosaic also supports reducing interactive resolution [31] to produce smaller data cubes that are faster to query.

Moreover, we have found it valuable to adjust the sample level during exploration, navigating under low latency with a smaller sample, then switching to a large sample to gain resolution and detail. In this way, Mosaic leverages both sampling and binned aggregation akin to Moritz et al.'s optimistic visualization [29]. Future work might add more performant indexing or prefetching schemes (c.f., [27]).

While Observable Plot has proven a convenient renderer for vgplot, it does not yet support incremental rendering, slowing updates involving many unchanged graphical elements. For fast drawing of 100k+ data points, future Mosaic clients could use hardware accelerated rendering. In addition, operations such as graph layout and cartographic transformations can be difficult or inefficient to implement in terms of database queries. While they can instead be performed in-browser, ultimately we would like to provide scalable support. As DuckDB is an extensible, open source engine, future extensions might better support GIS or specialized visualization workloads. Future Mosaic Coordinator implementations could also federate query processing across standard SQL databases and alternative engines.

A fundamental question here concerns how data transformation and visual encoding are best partitioned among a database and clients [30, 47]. In vgplot, most preparatory transformations are pushed to the database while visual encoding is performed in-browser. This approach supports immediate changes of color encodings and even kernel smoothing without querying the database. However, the distinction between transformation and encoding is not always clear cut. To perform hexagonal binning, the `hexbin` mark query performs a screen space mapping and then maps back to data space for consistency and integration with Observable Plot. Cartographic projection is particularly challenging, as not all projections are invertible, preventing scale inversion from screen space to data space. For projected maps, linked selection queries are better supported using post-projection coordinates, as in the Gaia example of Figure 1. Future efforts might more flexibly partition "encoding" transforms between the database and browser.

Going forward, we hope that Mosaic can serve as an open platform to develop and deploy scalable, interactive data exploration methods. We carefully designed the Mosaic Coordinator to decouple component implementations from data management, with the goal of making it easier for database specialists and visualization/UI specialists to contribute to separate parts of the system. Mosaic can be extended with new client components (or entire component libraries), while vgplot can be extended with new marks or interactors, as is appropriate.

One area of future work is to further integrate Mosaic with other systems. Figure 12 shows a proof-of-concept integration with Vega-Lite; more effort is needed to develop an alternative Vega-Lite parser that provides automatic Mosaic integration. Meanwhile, visualization generation systems such as PI2 [12] or coordination specifications such as Nebula [11] might be adapted to leverage Mosaic.

As previously noted, Mosaic can serve as a testbed for improved query caching, indexing, and other optimization methods. By logging queries and selections, Mosaic can also assist empirical research into data exploration workloads and user modeling [5, 6]. Given its use of declarative specification, Mosaic could serve as a target for future visualization reasoning and recommender systems (such as Draco [32] and Voyager [46]), including new automated reasoning rules for high-volume data. Mosaic, vgplot, and all corresponding components are available as open source software at uwdata.github.io/mosaic.

Acknowledgments

We thank the UW Interactive Data Lab for their suggestions and feedback. This work was supported by the Moore Foundation.

References

[1] Apache Arrow. https://arrow.apache.org/. 3

[2] Glue: multi-dimensional linked-data exploration. https://glueviz.org/. 2

[3] Observable Inputs. https://github.com/observablehq/inputs. 4

[4] Observable Plot. https://github.com/observablehq/plot. 1, 2, 4, 5, 8

[5] L. Battle, R. Chang, and M. Stonebraker. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016. doi: 10.1145/2882903.2882919 2, 9

[6] L. Battle, P. Eichmann, M. Angelini, T. Catarci, G. Santucci, Y. Zheng, C. Binnig, J.-D. Fekete, and D. Moritz. Database Benchmarking for Supporting Real-Time Interactive Querying of Large Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 2020. doi: 10.1145/3318464.3389732 9

[7] L. Battle and C. Scheidegger. A Structured Review of Data Management Technology for Interactive Visualization and Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1128–1138, 2021. doi: 10.1109/tvcg.2020.3028891 1

[8] M. Bostock, V. Ogievetsky, and J. Heer. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011. doi: 10.1109/tvcg.2011.185 1

[9] Bureau of Transportation Statistics. On-Time Performance. https://www.bts.gov/. 8

[10] D. B. Carr, R. J. Littlefield, W. L. Nicholson, and J. S. Littlefield. Scatterplot Matrix Techniques for Large*N*. *Journal of the American Statistical Association*, 82(398):424–436, 1987. doi: 10.1080/01621459.1987.10478445 2

[11] R. Chen, X. Shu, J. Chen, D. Weng, J. Tang, S. Fu, and Y. Wu. Nebula: A Coordinating Grammar of Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 28(12):4127–4140, 2022. doi: 10.1109/tvcg.2021.3076222 2, 9

[12] Y. Chen and E. Wu. Pi2: End-to-end Interactive Visualization Interface Generation from Queries. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, 2022. doi: 10.1145/3514221.3526166 9

[13] F. C. Crow. Summed-area tables for texture mapping. *ACM SIGGRAPH Computer Graphics*, 18(3):207–212, 1984. doi: 10.1145/964965.808600 7

[14] R. Deriche. Recursively implementing the Gaussian and its derivatives. Tech Report, INRIA, 1993. 5

[15] Gaia Collaboration, A. Vallenari, A. G. A. Brown, and 453 others. Gaia Data Release 3: Summary of the content and survey properties. 2022. doi: 10.48550/arXiv.2208.00211 8

[16] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1:29–53, 1997. 2, 6

[17] J. Heer. Fast & Accurate Gaussian Kernel Density Estimation. In *2021 IEEE Visualization Conference (VIS)*. IEEE, 2021. doi: 10.1109/vis49827.2021.9623323 2, 5

[18] M. C. Jones and H. W. Lotwick. On the errors involved in computing the empirical characteristic function. *Journal of Statistical Computation and Simulation*, 17(2):133–149, 1983. doi: 10.1080/00949658308810650 5

[19] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4. *Proceedings of the VLDB Endowment*, 7(10):797–808, 2014. doi: 10.14778/2732951.2732953 2, 5

[20] A. Kohn, D. Moritz, and T. Neumann. Dashql – Complete Analysis Workflows with SQL. 2023. doi: 10.48550/ARXIV.2306.03714 2, 5

[21] A. Kohn, D. Moritz, M. Raasveldt, H. Mühleisen, and T. Neumann. Duckdb-wasm. *Proceedings of the VLDB Endowment*, 15(12):3574–3577, 2022. doi: 10.14778/3554821.3554847 3

[22] N. Kruchten, J. Mease, and D. Moritz. Vegafusion: Automatic Server-Side Scaling for Interactive Vega Visualizations. In *2022 IEEE Visualization and Visual Analytics (VIS)*. IEEE, 2022. doi: 10.1109/vis54862.2022.00011 2, 8

[23] O. D. Lampe and H. Hauser. Curve Density Estimates. *Computer Graphics Forum*, 30(3):633–642, 2011. doi: 10.1111/j.1467-8659.2011.01912.x 2, 5

[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. doi: 10.1109/tvcg.2013.179 2

[25] Z. Liu, B. Jiang, and J. Heer. *imMens*: Real-time Visual Querying of Big Data. *Computer Graphics Forum*, 32(3pt4):421–430, 2013. doi: 10.1111/cgf.12129 2, 7, 9

[26] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. Devise. *ACM SIGMOD Record*, 26(2):301–312, 1997. doi: 10.1145/253262.253335 2

[27] H. Mohammed, Z. Wei, E. Wu, and R. Netravali. Continuous prefetch for interactive data applications. *Proceedings of the VLDB Endowment*, 13(12):2297–2311, 2020. doi: 10.14778/3407790.3407826 2, 9

[28] D. Moritz and D. Fisher. Visualizing a Million Time Series with the Density Line Chart. 2018. doi: 10.48550/ARXIV.1808.06019 2, 5

[29] D. Moritz, D. Fisher, B. Ding, and C. Wang. Trust, but Verify: Optimistic Visualizations of Approximate Queries for Exploring Big Data. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 2017. doi: 10.1145/3025453.3025456 9

[30] D. Moritz, J. Heer, and B. Howe. Dynamic Client-Server Optimization for Scalable Interactive Visualization on the Web. In *IEEE VIS Data Systems for Interactive Analysis (DSIA) Workshop*. Chicago, IL, 2015. 9

[31] D. Moritz, B. Howe, and J. Heer. Falcon. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, 2019. doi: 10.1145/3290605.3300924 2, 7, 9

[32] D. Moritz, C. Wang, G. L. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):438–448, 2019. doi: 10.1109/tvcg.2018.2865240 9

[33] C. North and B. Shneiderman. Snap-together visualization. In *Proceedings of the working conference on Advanced visual interfaces*. ACM, 2000. doi: 10.1145/345513.345282 2

[34] M. Raasveldt and H. Mühleisen. Duckdb. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 2019. doi: 10.1145/3299869.3320212 2, 3

[35] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2017. doi: 10.1109/tvcg.2016.2599030 1, 2, 3, 4, 6, 8

[36] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, 2016. doi: 10.1109/tvcg.2015.2467091 1, 2, 3, 8

[37] C. Stolte, D. Tang, and P. Hanrahan. Polaris: a system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002. doi: 10.1109/2945.981851 2

[38] W. Tao, X. Hou, A. Sah, L. Battle, R. Chang, and M. Stonebraker. Kyrix-S: Authoring Scalable Scatterplot Visualizations of Big Data. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):401–411, 2021. doi: 10.1109/tvcg.2020.3030372 2

[39] W. Tao, X. Liu, Y. Wang, L. Battle, Ç. Demiralp, R. Chang, and M. Stonebraker. Kyrix: Interactive Pan/Zoom Visualizations at Scale. *Computer Graphics Forum*, 38(3):529–540, 2019. doi: 10.1111/cgf.13708 2

[40] J. VanderPlas, B. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert. Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software*, 3(32):1057, 2018. doi: 10.21105/joss.01057 6

[41] M. P. Wand. Fast Computation of Multivariate Kernel Estimators. *Journal of Computational and Graphical Statistics*, 3(4):433, 1994. doi: 10.2307/1390904 2, 5

[42] C. Weaver. Building Highly-Coordinated Visualizations in Improvise. In *IEEE Symposium on Information Visualization*. IEEE. doi: 10.1109/infvis.2004.12 2, 3

[43] H. Wickham. A Layered Grammar of Graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010. doi: 10.1198/jcgs.2009.07098 1, 2, 4

[44] H. Wickham. Bin-summarise-smooth: A framework for visualising large data. Tech Report, 2013. 2

[45] L. Wilkinson. *The Grammar of Graphics*, pp. 375–414. Springer Berlin Heidelberg, 2011. doi: 10.1007/978-3-642-21551-3_13 2, 4

[46] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and

J. Heer. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):649–658, 2016. doi: 10.1109/tvcg.2015.2467191 9

[47] E. Wu, L. Battle, and S. R. Madden. The case for data visualization management systems. *Proceedings of the VLDB Endowment*, 7(10):903–906, 2014. doi: 10.14778/2732951.2732964 9

[48] Y. Wu, R. Chang, J. M. Hellerstein, A. Satyanarayan, and E. Wu. Diel: Interactive Visualization Beyond the Here and Now. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):737–746, 2022. doi: 10.1109/tvcg.2021.3114796 2

[49] J. Yang, H. K. Joo, S. S. Yerramreddy, S. Li, D. Moritz, and L. Battle. Demonstration of VegaPlus: Optimizing Declarative Visualization Languages. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, 2022. doi: 10.1145/3514221.3520168 2

A Query Generation & Date Cube Indexes

Here we further detail how vgplot mark clients generate queries and how the Coordinator performs data cube indexing.

A.1 Basic Marks

Each vgplot *mark* is as a Mosaic client that generates queries. The Coordinator invokes the client `query()` method and manages the returned query. Basic mark types use a straightforward query generation procedure. For example, consider a standard scatter plot (`dot` mark) with `x`, `y`, and `r` (radius size) encodings. For a backing data table `$table` and corresponding table columns denoted as `$u`, `$v`, and `$w`, the mark `query()` method returns a query that selects the data directly:

```
SELECT $u AS x, $v AS y, $w AS r
FROM $table
```

Subsequent visual encoding—such as mapping the data through scale transforms—is performed in the browser. Mark clients may also have a `filterBy` Selection property, which if set, is used to generate a predicate that is passed as the `filter` argument to the mark `query()` method. For basic marks, the `filter` predicate is appended as a SQL `WHERE` clause. Given an interval selection for column `$b` over the domain `[$b0, $b1]`, the following query is produced:

```
SELECT $u AS x, $v AS y, $w AS r
FROM $table
WHERE $b BETWEEN $b0 AND $b1
```

If a mark encoding involves an aggregate operation, the non-aggregated fields are included as SQL `GROUP BY` criteria. Consider a bar chart (`barY` mark) that, for an ordinal column `$a` on the x-axis, shows the average values of columnn `$b` on the y-axis:

```
SELECT $a AS x, AVG($b) AS y
FROM $table
GROUP BY $a
```

A.2 Connected Marks and M4 Optimization

Connected marks such as `lineX/Y` and `areaX/Y` can be further optimized. Consider the area charts in Figure 7. For a table `$table` with column `$t` visualized along the x-axis and column `$v` along the y-axis, the basic query generation method above would select all data points:

```
SELECT $t AS x, $v AS y
FROM $table
ORDER BY x
```

We can do better by applying shape-preserving, pixel-aware binning using the M4 method [19]. For a given chart pixel width `$w` and minimum and maximum plotted `$t` values `$t0` and `$t1`, the optimized query used by Mosaic takes the form:

```
SELECT MIN($t) AS x, ARG_MIN($v, $t) AS y
 FROM $table
 GROUP BY FLOOR($w * (x - $t0) / ($t1 - $0))
UNION
 SELECT MAX($t) AS x, ARG_MAX($v, $t) AS y
 FROM $table
 GROUP BY FLOOR($w * (x - $t0) / ($t1 - $0))
UNION
 SELECT ARG_MIN($t, $v) AS x, MIN($v) AS y
 FROM $table
 GROUP BY FLOOR($w * (x - $t0) / ($t1 - $0))
UNION
 SELECT ARG_MAX($t, $v) AS x, MAX($v) AS y
 FROM $table
 GROUP BY FLOOR($w * (x - $t0) / ($t1 - $0))
ORDER BY x
```

For each pixel, M4 selects the extremal `$t` and `$v` values—two minimums and two maximums, hence "M4". The resulting output data has at most four points per pixel. We use the "AM4" variant of M4 [20], which uses `ARG_MIN` and `ARG_MAX` aggregates to select a matching co-ordinate for each extremum.

A.3 Linear Binning

The `densityX` and `densityY` marks visualize kernel density estimates. To produce these estimates in a scalable manner, we use an approximation that first bins the data points into a grid. We perform *linear binning* [18, 41] in the database to make the approximation more accurate [17]. Linear binning proportionally distributes the weight of a point between adjacent bins. If a data point x_i lies between bins with midpoints b_0 and b_1, linear binning assigns weight proportional to $(b_1 - x_i)/(b_1 - b_0)$ to bin b_0 and $(x_i - b_0)/(b_1 - b_0)$ to bin b_1.

To bin column `$v` linearly into `$n` bins over the domain `[$v0, $v1]`, we use a query with two subqueries—one for the "left" bin and one for the "right" bin—and aggregate the results of their union:

```
SELECT index, SUM(weight) AS weight
FROM (
 SELECT
   ($n - 1) * ($v - $v0) / ($v1 - $v0) AS p,
   FLOOR(p) AS index,
   index + 1 - p AS weight
  FROM $table
 UNION ALL
  SELECT
   ($n - 1) * ($v - $v0) / ($v1 - $v0) AS p,
   FLOOR(p) + 1 AS index,
   p - index - 1 AS weight
  FROM $table
)
GROUP BY index
HAVING weight > 0
```

The return value is a one-dimensional grid of binned values, with an integer `index` and corresponding `weight`. To generate two-dimensional densities, we perform linear binning in 2D using an analogous procedure involving four subqueries. Subsequent smoothing is performed in the browser using Deriche's linear time approximation [14, 17], allowing rapid updates upon change of kernel bandwidth.

A.4 Binned Aggregation and Data Cube Indexes

Figure 13 shows three histograms with cross-filtering interactions. For each histogram (`rectY` mark), we use a `bin` transform on the `x` encoding channel and a `count` aggregate for the `y` channel. The `bin` transform provides an expression generator function that is called by the basic mark query generation procedure. The query for a single histogram of column `$v` over the domain `[$v0, $v1]` is:

```
SELECT
  $v0 + $step * FLOOR(($v - $v0) / $step) AS x1,
  $v0 + $step * (FLOOR(($v - $v0) / $step) + 1) AS x2,
  COUNT(*) AS y
FROM $table
GROUP BY x1, x2
```

To cross-filter, each mark has a `filterBy` Selection that produces predicates driven by selection brushes (`intervalX` interactors). As above, the default query generation procedure adds those predicates to a SQL `WHERE` clause. For an interval brush selection `[$b0, $b1]` over the column `$u`, the resulting query for a cross-filtered histogram is:

```
SELECT
  $v0 + $step * FLOOR(($v - $v0) / $step) AS x1,
  $v0 + $step * (FLOOR(($v - $v0) / $step) + 1) AS x2,
  COUNT(*) AS y
FROM $table
WHERE $u BETWEEN $b0 AND $b1
GROUP BY x1, x2
```

If data cube indexing is enabled, these queries are automatically optimized by the Coordinator, in a fashion completely decoupled from the mark itself. If generated queries involve group-by aggregation using supported aggregate functions (currently `count`, `sum`, `avg`, `min`, and `max`), the Coordinator will rewrite the query to create a *multivariate data tile* [25]. The result is stored in the database as a new table. The table name includes a hash of the SQL query string that creates the table, enabling easy reuse. If the table already exists it is not re-created.

For an interval selection over the column `$u`, the Coordinator uses the number of `$pixels` and the minimum and maximum possible brush values in the data domain `[$bmin, $bmax]` to produce pixel-level bins for all possible brush positions. The following query creates a data cube for brush interactions between a one-dimensional active selection clause and a single histogram:

```
CREATE TEMP TABLE IF NOT EXISTS cube_index_a097caa4 AS
SELECT
  $v0 + $step * FLOOR(($v - $v0) / $step) AS x1,
  $v0 + $step * (FLOOR(($v - $v0) / $step) + 1) AS x2,
  COUNT(*) AS y,
  FLOOR($pixels * ($u - $bmin) / ($bmax - $bmin)) AS activeX
FROM $table
GROUP BY x1, x2, activeX
```

Upon selection updates, the Coordinator issues queries to the data cube index rather than use the basic filtered query described previously. Given data-space brush endpoints `$b0` and `$b1`, the index query is:

```
SELECT x1, x2, SUM(y)
FROM cube_index_a097caa4
WHERE activeX BETWEEN
 FLOOR($pixels * ($b0 - $bmin) / ($bmax - $bmin)) AND
 FLOOR($pixels * ($b1 - $bmin) / ($bmax - $bmin))
GROUP BY x1, x2
```

The size of the data cube is bound by the number of bins (binned `$v` steps and the number of `$pixels`), not the size of the input data. As a result, for large datasets the data cube index queries can be computed substantially faster [25, 31]. Two-dimensional brushes are handled similarly, resulting in both `activeX` and `activeY` index columns.

Data cube indexes can be created for complex queries involving subqueries or common table expressions (CTEs). In such cases, the Coordinator walks the query tree and performs *pushdown* of active selection columns. In the `densityY` query below, the column `$u` is pushed down into the subqueries and then pixel-binned by the outer query. Subsequent index queries thus amortize the cost of both interactive updates and the original, complex aggregation.

```
CREATE TEMP TABLE IF NOT EXISTS cube_index_b0d4fe30 AS
SELECT
  index, SUM(weight) AS weight,
  FLOOR($pixels * ($u - $bmin) / ($bmax - $bmin)) AS activeX
FROM (
 SELECT
   ($n - 1) * ($v - $v0) / ($v1 - $v0) AS p,
   FLOOR(p) AS index,
   index + 1 - p AS weight,
   $u
  FROM $table
 UNION ALL
  SELECT
   ($n - 1) * ($v - $v0) / ($v1 - $v0) AS p,
   FLOOR(p) + 1 AS index,
   p - index - 1 AS weight,
   $u
  FROM $table
)
GROUP BY index, activeX
HAVING weight > 0
```

A.5 Line Density

Density mark calculations can use either the linear binning method described above or standard binning, in which the "mass" of point is allocated to a single bin only. However, these methods apply to point data only. Density line charts [28] and curve density estimates [23] instead show densities for series, not just individual data points. The `denseLine` mark subclasses vgplot's `raster` mark, generating an alternative query that performs line rasterization and normalization in the database to produce line densities.

As shown below, the generated query is complex and consists of multiple processing steps specified as common table expressions (CTEs). The key steps are: (1) bin data points to raster grid coordinates (`source` subquery), (2) identify line segments as start points and delta offsets (`pairs` subquery), (3) compute integer indices up to the maximum line segment run or rise (in raster bins, `indices` subquery), (4) join the line segments and indices to perform line rasterization (`raster` subquery), (5) normalize column weights for each series to approximate arc-length normalization [28] (`points` subquery), and (6) aggregate all density values into an output grid (outer query).

```
WITH
 source AS ( -- 1. source data, bin x and y
  SELECT
    FLOOR(($x - $x0) * ($nx - 1) / ($x1 - $x0)) AS x
    FLOOR(($y - $y0) * ($ny - 1) / ($y1 - $y0)) AS y,
    $z AS z,
  FROM $table
 ),
 pairs AS ( -- 2. form line segments: start point + offsets
  SELECT
    z, x AS x0, y AS y0,
    LEAD(x) OVER w - x AS dx,
    LEAD(y) OVER w - y AS dy
  FROM source
  WINDOW w AS (PARTITION BY z ORDER BY x ASC)
 ),
 indices AS ( -- 3. integer indices up to the max rise or run
  SELECT UNNEST(
    range(
      SELECT GREATEST(MAX(ABS(dx)), MAX(ABS(dy)))
      FROM pairs
    )
  ) AS i
 ),
 raster AS ( -- 4. perform line rasterization
  SELECT -- case where run is greater than rise
    z,
    x0 + i AS x,
    y0 + ROUND(i * dy / dx) AS y
   FROM pairs, indices
   WHERE ABS(dy) <= ABS(dx) AND i < ABS(dx)
  UNION ALL
   SELECT -- case where rise is greater than run
    z,
    x0 + ROUND(SIGN(dy) * i * dx / dy) AS x,
    y0 + SIGN(dy) * i AS y
   FROM pairs, indices
   WHERE ABS(dy) > ABS(dx) AND i < ABS(dy)
  UNION ALL
   SELECT -- case of final line segment end point (no offsets)
    z, x0 AS x, y0 AS y
   FROM "pairs" WHERE dx IS NULL
 ),
 points AS ( -- 5. perform per-column, per-series normalization
  SELECT
    x, y
    1.0 / COUNT(*) OVER (PARTITION BY x, z) AS w
  FROM raster
  WHERE (x BETWEEN $x0 AND $x1) AND (y BETWEEN $y0 AND $y1)
 )
SELECT -- 6. sum normalized weights from all series
  x + y * $width AS index,
  SUM(w) AS weight
FROM points
GROUP BY index
```

In the query above, `$x` and `$y` correspond to input columns mapped to spatial dimensions (over the domains `[$x0, $x1]` and `[$y0, $y1]`), while `$z` is a categorical variable indicating different line series and `$width` is the width of the output grid in raster cells. The resulting grid of line densities can optionally be smoothed in the browser, using the inherited functionality of the `raster` mark.

B Additional Examples

Here we share additional examples of Mosaic applications, supplementing the examples presented in §7.

B.1 Neuron Spike Measurements

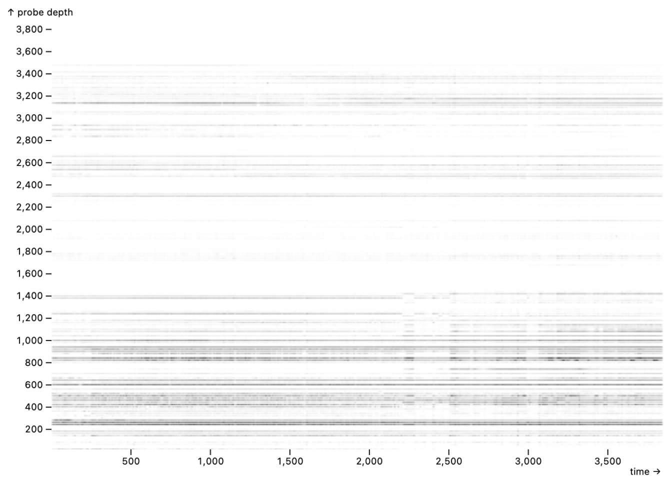


Fig. 18: Time-series heatmap of neuron spike activity along a probe.

We apply Mosaic to data from neuroscience collaborators. Figure 15 shows raw electrical data, which is important to visualize in order to assess potential errors of subsequent analysis algorithms. Here, Figure 18 is a `raster` density map of algorithmically extracted neuron spikes from a single experimental run, consisting of 8.3M rows in a 95MB Parquet file. A `panZoom` interactor enables real-time panning and zooming.

Our collaborators were excited that we were able to write Mosaic code within just a few minutes to visualize and interact with their data. Their current workflow involves long-running batch processes to generate static images for each experimental session. With Mosaic, we are able to provide real-time interactive visualizations on-demand. We are now working jointly on richer dashboards for examining the results across many experimental runs.

B.2 X-Ray Scatter Images

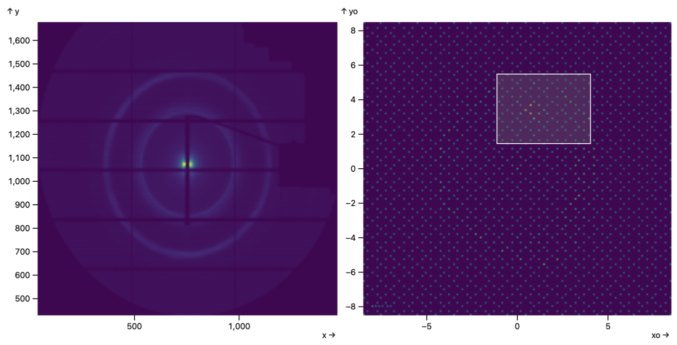


Fig. 19: An X-ray beam pointed at a sample (right) creates a scattering pattern (left). Selecting a region of the sample shows pronounced rings in the scattering image.

Here Mosaic visualizes data from beamline X-ray scattering experiments that help understand physical materials' properties. Figure 19 shows a scatter image (left) and the position of the X-ray beam on the sample (right). Scientists can see the aggregated scattering image for a region by selecting regions of the sample. The data collected at Brookhaven National Lab shows scatter images of a coffee stain with 1,776 scatter images in this dataset resulting in 1.9B data points. While updates are real-time, similar to the 100% sample Gaia raster case in our benchmarks (§8.2), index creation is not ideally interactive. Precomputation of indexes amortizes this cost, enabling immediate interaction in subsequent sessions.

B.3 Zoomable Gaia Sky Map

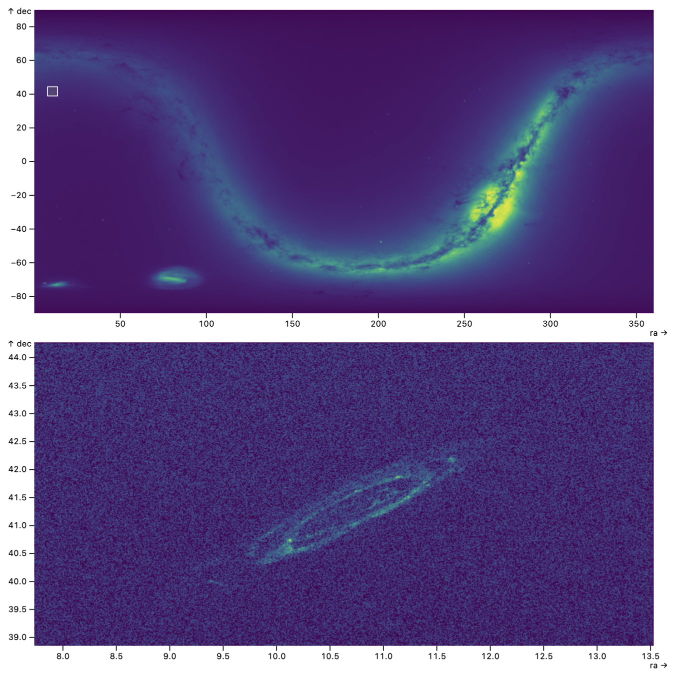


Fig. 20: An overview+detail configuration of the Gaia sky map. Selecting a small region of the sky (top) resolves the Andromeda galaxy (bottom).

In addition to the Gaia dashboard of Figure 1, we built the alternative overview+detail sky map shown in Figure 20. Rather than apply a cartographic projection, here the original right ascension (`ra`) and declination (`dec`) coordinates are plotted directly. Brushing in the overview region produces a zoomed-in view in the detail panel. A patch of sky is selected in Figure 20 (top), revealing the Andromeda galaxy (bottom).