Falcon: Balancing Interactive Latency and Resolution Sensitivity for Scalable Linked Visualizations

Dominik Moritz
University of Washington
domoritz@cs.washington.edu

Bill Howe
University of Washington
billhowe@uw.edu

Jeffrey Heer
University of Washington
jheer@uw.edu

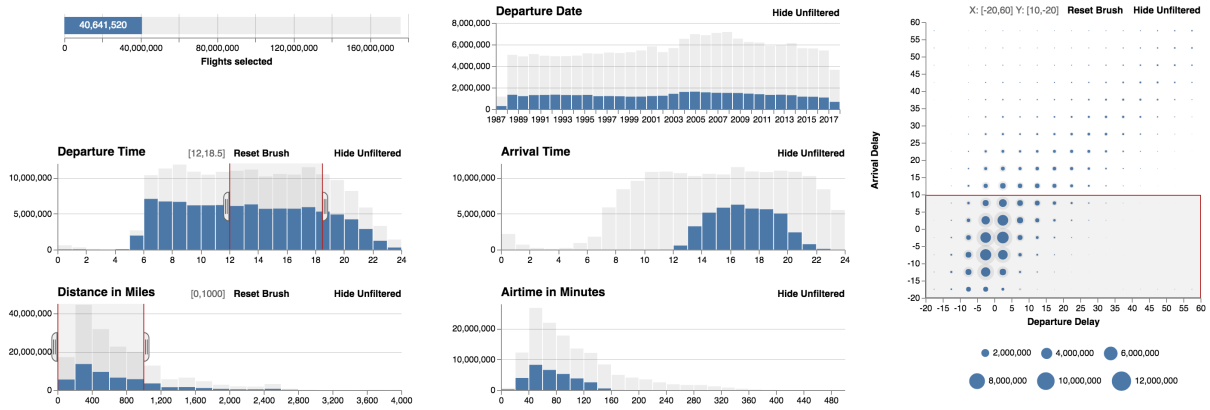

Figure 1: Falcon visualizing binned aggregates for 180 million flights [31] in a web browser. The brushes select short afternoon flights with no more than a 10 minute arrival delay. The views update instantly when the user draws, moves, or resizes a brush.

ABSTRACT

We contribute user-centered prefetching and indexing methods that provide low-latency interactions across linked visualizations, enabling cold-start exploration of billion-record datasets. We implement our methods in Falcon, a web-based system that makes principled trade-offs between latency and resolution to optimize brushing and view switching times. To optimize latency-sensitive brushing actions, Falcon reindexes data upon changes to the active view a user is brushing in. To limit view switching times, Falcon initially loads reduced interactive resolutions, then progressively improves them. Benchmarks show that Falcon sustains real-time interactivity of 50fps for pixel-level brushing and linking across multiple visualizations with no costly precomputation. We show constant brushing performance regardless of data size on datasets ranging from millions of records in the browser to billions when connected to a backing database system.

CHI 2019, May 4–9, 2019, Glasgow, Scotland UK

CCS CONCEPTS

• **Human-centered computing** → **Visualization systems and tools**; *Interactive systems and tools*; • **Information systems** → *Online analytical processing engines.*

KEYWORDS

data visualization; scalability; latency; brushing and linking

ACM Reference Format:
Dominik Moritz, Bill Howe, and Jeffrey Heer. 2019. Falcon: Balancing Interactive Latency and Resolution Sensitivity for Scalable Linked Visualizations. In *CHI Conference on Human Factors in Computing Systems Proceedings (CHI 2019), May 4–9, 2019, Glasgow, Scotland UK*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3290605.3300924

1 INTRODUCTION

To support effective exploration, interactive visualization systems must provide rapid response times for latency-sensitive operations. Further, delays between user actions and corresponding updates may break the perceived correspondence between action and response, reducing the user's engagement with the system and leading to fewer observations made [17, 26, 41]. As the scale and heterogeneity of available data continue to increase, greater emphasis is being placed on efficient exploration. However, large datasets incur delays that negatively affect user's exploration. Poor support for interactive exploration can skew analyst attention toward "convenient" and familiar datasets, causing selection

biases. This work seeks to reduce the friction of using new data by enabling *cold-start analytics*: exploration without time-consuming precomputation.

Traditionally, the different stages of the data processing pipeline—query processing, data transfer, and rendering—have been optimized as independent modules. For example, many efforts to reduce latency have centered on query processing, paying scant attention to the corresponding interface design. Recent GPU databases [35] can achieve query times of seconds or hundreds of milliseconds over billions of records; however, interactivity is still difficult to achieve due to factors outside the scope of database optimizations, including network latency and sub-optimal client-side application design. Even short query times accumulate when a UI generates thousands of queries. And network-induced delays remain unpredictable, especially in low-connectivity or mobile networks.

In Falcon, we take a holistic approach to system design by optimizing the interface and query systems together. We prioritize the allocation of compute resources to interactions for which users are more latency-sensitive, in particular *brushing and linking*. For example, in Figure 1 a user can resize the brush in the arrival time view, which immediately updates all other views.

To eliminate latency for brushing interactions, we contribute prefetching and indexing techniques. Rather than treating every query as an independent request, we model and optimize a user's session with client-side state. In a session, we leverage the fact that a user typically interacts with only a single view at a time—the *active view*. When the user moves the brush at pixel resolution, the aggregated data for all other views—the *passive views*—are computed in constant time using Falcon's indexes. Brushing interactions are decoupled from computations over the raw data; the interactive resolution of the brushes is decoupled from the bin resolution. As a result, both index size and interactive latency depend only on bin size and available pixel resolution and are independent of the raw data.

When the active view changes, Falcon reindexes the data to support interactions with the new active view. For datasets of up to millions of records, the client can perform the necessary aggregations. For larger datasets, aggregation can be offloaded to a backing database system. Switching the active view in Falcon incurs processing delays. Such switches usually occur with a shift in a user's attentional focus, a less latency-sensitive action [7]. To limit view switching times, Falcon initially lowers the resolution of the index data so brush boundaries "snap-to" units larger than individual pixels. Analogous to progressive rendering or query processing (e.g., *online aggregation* [10, 21]), this reduced interaction resolution can then progressively improve.

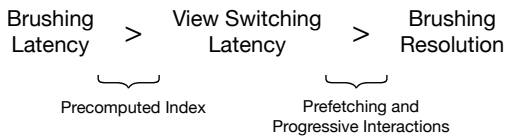

Figure 2: The Falcon system uses indexes to optimize brushing latencies and progressively improves interactive resolution to reduce switching times.

In summary, Falcon prioritizes brushing latency over view switching delay, and it prioritizes view switching delay over initial interactive resolution (Figure 2). Our prototype implements methods to support coordinated brushing and filtering for cross-filter and aggregation applications (i.e., ensembles of visualizations of 1D and 2D bin counts). The system avoids expensive precomputation by prefetching only the data necessary for interactions with a single active view, enabling *cold-start* analytics. In Falcon, charts update in response to brush changes at 50 fps. We demonstrate that this performance is invariant on data sets ranging from thousands to billions of records. Because the system progressively improves interactive resolution, we use interpolation to approximate higher-resolution interactions. We make Falcon available as open source software with supporting documentation and demos at www.github.com/uwdata/falcon.

2 BACKGROUND AND RELATED WORK

Falcon is a visualization system for interactive brushing and linking across coordinated views of *binned aggregates*. Binned aggregates summarize data by dividing the domain of variables into discrete units (bins), and then by aggregating the data records in each bin [27, 40]. For example, *histograms* are visualizations of *bin counts*. Each graphical mark in a visualization of binned aggregates summarizes a large subset of records in the original dataset. Falcon uses binned aggregates because they convey both global patterns (e.g., densities) and local features (e.g., outliers) and enable multiple levels of resolution via the choice of bin size. To focus on relevant subsets of the data, analysts select ranges of data in one view using an interactive brush, which then updates all linked views. Commercial data visual analysis systems such as Tableau [37], PowerBI [28], and Immerse [32] use coordinated views with visual querying.

Falcon extends prior work on scalable interactive analysis systems, incorporating findings from experimental studies of the effect of latency on exploratory visual analysis.

Latency in Interactive Analysis

Informed by prior accounts of latency in psychology and HCI [1, 6, 7, 22], Liu and Heer [26] conducted controlled experiments (later replicated by Zgraggen et al. [41]) to understand how latency affects user behavior in exploratory

visual analysis. Comparing different operations under two latency conditions, they found that additional delays of 500ms over the low-latency condition (20ms for both brush and link and select, 100ms for pan, and 1s for zoom) negatively impact user behavior. They also found that initial exposure to delays negatively affects subsequent performance even when the delays were removed in later sessions.

In recent years, system designers have reduced latency by optimizing the different stages of the visualization pipeline: data management, scenegraph construction, and rendering [26]. Their efforts have largely addressed each stage separately. Many system designers have adopted 500ms as a uniform latency threshold goal (e.g., [3, 13, 16, 34]); however, experimental results indicate that some operations (e.g., zooming) are less sensitive to delays than others. Liu and Heer [26] found that panning, brushing, and range selection are the most latency-sensitive of the studied operations.

Scalable Visual Analysis Systems

Interactive analysis systems with coordinated views run in a client application. The data being analyzed can either be loaded into this client or offloaded to a dedicated server. Table 1 compares the characteristics of different visual analysis systems for coordinated brushing and linking. For a dataset small enough to be loaded client-side, visual analytics tools support real-time interactivity using local indexes. Square's Crossfilter [39] uses bitmap indexes to support brushing and linking of hundreds of thousands of records entirely in the browser. Liu et al.'s imMens [27] uses precomputed summaries to enable real-time interactions in the browser, but their interactions are limited to the binning resolution and a single interactive brush.

For greater scalability, many systems adopt a client-server architecture. In this approach, all components of the information visualization reference model [6] are not necessarily on the same machine. Instead, the server stores the data, processes incoming queries, and sends reduced aggregates to the client. In the client-server model, changes to the client-side UI state require a request to the server; this incurs a delay between the user action and the corresponding update, a delay dominated by the network round-trip and query execution times. Since network bandwidth and latency are often beyond the control of the tool designer, optimizations aim to increase query performance. We discuss these techniques in the next subsection.

Falcon supports both client-only and client-server setups. For data sizes up to tens of millions of records, Falcon can load the full dataset in the browser and index it there. For larger datasets, costly computations can be offloaded to a scalable server-side database system.

Scalable Data Processing for Visualization

Three main approaches can speed up query evaluation: *parallel evaluation*, *indexing*, and *approximation*.

Parallel evaluation. To reduce query latency in large-scale online analytical processing (OLAP) systems [9], we can distribute work across multiple machines. However, the additional communication overhead typically exceeds the query times necessary for interactive data exploration.

Indexing. Indexes and data cubes [20] significantly speed up query evaluation by precomputing aggregates along some dimensions. Specialized hierarchical data structures for visualization, like Nanocubes [24], are compact indexes of spatiotemporal data that can fit in the main memory of a single machine. The Nanocube structure leverages sparsity to more efficiently build a specialized tree index that consists of quadtrees (for spatial dimensions) or flat trees (for categorical attributes). The trees organize and aggregate data records for each dimension, which are then combined into a larger index. Hashedcubes [33] further improves this design with a more compact index. Profiler [23] also builds in-memory data cubes for query processing. The size of the data cube depends on the resolution and number of dimensions, not on the data size. Thus, data cube approaches enable scalable data processing on a single machine and support low latency responses to aggregation queries over billions of records. However, they can impose lengthy index building times, e.g., Nanocubes takes up to 6 hours to build an index for a dataset with 210M objects [24].

Liu et al.'s imMens [27] uses a dense data cube structure with precomputed aggregations. The size of the full data cube is $\prod_i b_i$, where b_i is the bin count for dimension i; it is polynomial in the bin count and exponential in the number of dimensions. To overcome the exponential growth with more dimensions, Liu et al. decompose the full cube into a set of overlapping 3- and 4-dimensional projections, or "data tiles." This approach enables real-time brushing and linking but limits interactions to a single brush. Moreover, since the resolution of the imMens data cube is the resolution of the visible bins, brushes snap to these bins. For large datasets, the tiles must be precomputed since they are costly to calculate.

Falcon makes a critically different trade-off: it decomposes a data cube so that it supports interactions with a single active view only. The size of its full index is linear in the number of views, which avoids a combinatorial explosion. An index is loaded when the user interacts with a particular view. Falcon supports multiple brushes by conditioning it on the brushes in the passive views. Further, each view can be filtered by all brushes except the brush in the view itself instead of only the union of all filters. Querying and transferring the smaller index for a single view is less costly

Feature \ System	Square Crossfilter	imMens	Nanocubes	OmniSci Immerse	Falcon
Approach	Client-side Index	Dense Data Tiles	Sparse Cube	Live Queries	View-Specific Tiles
Architecture	Client	Client (Server)	Client-Server	Client-Server	Client (Server)
Demonstrated data size	10^5	10^{12}	10^{12}	10^{12}	10^{12}
Cold-start	Yes	No	No	Yes	Yes
Interactive resolution	Pixels	Bins	Pixels	Pixels	Pixels
Multiple brushes	Yes	No	Yes	No	Yes
2D binning	No	Yes	Yes	Yes	Yes
Zooming	No	Yes*	Yes	Yes	Yes
View switching cost	No	No	No	No	Yes

Table 1: Comparison of different approaches to scalable linked views. Not shown: PowerBI [28] and Tableau Public [38] use a similar approach to Immerse; Hashedcubes [33] builds on Nanocubes [24] and shares similar properties.
*** supported for predefined zoom levels**

than doing so with a full data cube (e.g., imMens) that supports interactions with all views. The smaller index can be computed and loaded on demand. We can also increase its resolution to support brushing at pixel resolution rather than snapping brushes to visible bins. Falcon supports cold-start exploration of new datasets and is more flexible about zoom levels (as both imMens and Falcon require a new index when the user zooms).

Approximation. Approximate query processing systems [10] estimate result values and their uncertainty using a data sample. SampleAction [17], Vizdom [13], and Pangloss [29] demonstrate that progressively refined approximate results are often sufficient for exploratory analysis. However, these systems do not support interactive brushing and linking. Although the current version of Falcon does not use progressively growing samples to approximate aggregates, we apply an analog of these techniques in the interaction domain: we initially load an index that supports interactions at a granularity larger than single pixels, then we progressively refine the granularity to one pixel.

Prefetching

To mitigate query and network latency, a system can try to predict queries that the UI will likely issue, then precompute and cache results [15]. Chan et al. [8] show this approach for time series data. Battle et al. developed a series of systems [2, 3] that prefetch data tiles for a panning and zooming interface. Falcon combines ideas, such as projected data cubes in imMens [27], with prefetching methods. It prefetches an index that supports all interactions with the current view, and is conditioned on the brushes in all other views.

3 THE FALCON INTERFACE DESIGN

We now describe the Falcon interface, how users can interact with its charts, and how it prefetches data to rapidly update charts. Here, we highlight how Falcon works. The following section provides a more detailed discussion of its implementation.

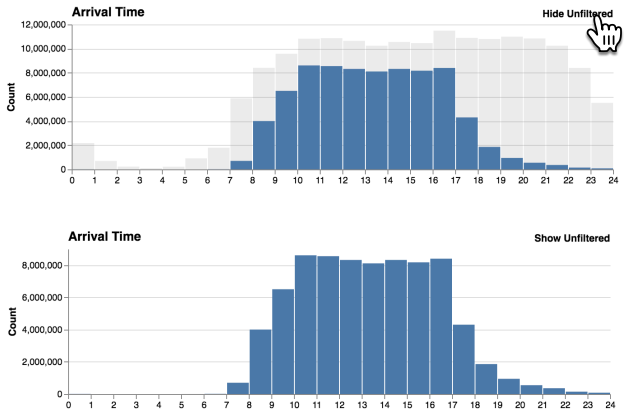

Figure 3: The top histogram plots filtered counts (blue bars) relative to the unfiltered data (gray bars). A user can toggle to show the filtered distribution only (bottom).

Falcon provides a dashboard of views that visualize the number of records, grouped by zero, one, or two binned dimensions. Figure 1 shows Falcon loaded with a U.S. flight delays dataset [31]. Using application constructor parameters, developers can configure the dataset, configure the chart layout, and customize the chart design.

Charts for Zero-, One-, and Two-Dimensional Data

Falcon's views show aggregates grouped by zero, one, or two binned dimensions. We implement all visualizations in Vega [36]. For zero-dimensional data, a view simply shows the record count. Developers can choose among a vertical or horizontal bar chart (e.g., Figure 1, top left) or a text view. Blue bars show the number of records that match all filters, while gray bars show unfiltered counts for comparison.

For views that are grouped by a single binned dimension, Falcon uses bar charts (e.g., Figure 1, top center). Again, Falcon shows the total unfiltered counts as gray bars. Unfiltered counts provide context and keep the domain of the Y-scale constant as a user filters data. To see the filtered distribution only, users can toggle the gray bars (Figure 3).

For bivariate views, we must use additional visual channels, such as size (e.g., Figure 1, right) or color. Size (e.g., circular area) is known to be more perceptually effective for numerical comparison [12], though it requires significantly more pixels compared to color encoding. Most importantly, by encoding counts as the size of circles, Falcon can show unfiltered counts as gray circles behind blue circles, establishing a consistent visual language across all three chart types. Nonetheless, developers can switch to color encoding [27, 29], but they can no longer see unfiltered counts.

Applying consistent binning schemes over 1D and 2D views ensures compatibility of linked selections between plots. Falcon uses Vega's [36] binning algorithm to compute bin width and offset from the scale range. The algorithm may extend the scale range to find "nice" bin thresholds (e.g., using only multiples of 5 and 10).

Brushing in the Active View

Upon initialization, Falcon shows unfiltered counts (Figure 4, a). A user can then filter the data—for example, to show only flights that arrive in the afternoon—by drawing a brush in any view (Figure 4, b). We call the view with which the users interacts the *active view*. Falcon aims to show the counts of the selected subset and update the data for all other views—*the passive views*—as the user changes the brush. The active view does not change.

Because re-aggregating counts from the raw data for every brush movement can be too costly, Falcon uses an index, which is a compact summary that contains the details needed to update passive views for any possible brush in the active view. Falcon decouples rapid brush updates (using any of the actions in Figure 5) from costly computations over the full dataset. To limit its size, the index contains binned aggregates for passive views at their bin resolution and supports brushes in the active view at pixel resolution. Falcon achieves a much smaller index than one that supports interactions with all views [24, 27] by focusing on a single active view.

Switching Active Views

For a user to draw a brush in a different view, Falcon must switch the active view, requiring it to make a new index. For example, to change from brushing over arrival time to distance (Figure 4, c), the index for the arrival view must be replaced with an index that supports brushing in the distance view. The new index must be conditioned on any existing brushes (here, for arrival time), which means the counts in the new index must be filtered according to the selected ranges. The only exception is for the arrival time view itself, since the bin counts should not be filtered by its own brush. This rule generalizes to any number of brushes: the index for each passive view must always be conditioned on the brushes in all other passive views.

Zooming the Active View

When a user zooms in a binned chart, the visualized range changes. Since scale changes require no new data, Falcon can immediately give visual feedback and zoom the chart. When the zoom interaction ends, Falcon computes a new bin width and offset. If the computed parameters differ from the current ones, Falcon computes updated bin counts for the active view as well as a new index. The latter is needed to support interactions at the new resolution. Recomputing the index may impose delay. However, as discussed earlier, research has shown that zooming is less latency-sensitive than brushing [26].

Prefetching

Instead of waiting for the first interaction with a new active view to create a new index, Falcon can prefetch indexes before a user starts brushing. Figure 6 shows example timings for the brushing interactions shown in Figure 4. After modifying the arrival time brush, an analyst might move the cursor to hover over the distance view when preparing to draw a new brush. In this case, Falcon would not yet perform a view switch (i.e., change the index), but it would prefetch the index. When the analyst starts brushing (around second 20), Falcon switches to the prefetched index. Hovering over a chart is only one signal that we could use to predict what chart the user will interact with next. Techniques from previous work on prefetching [2] could also be used, but we found that mouse hover is a strong indicator of user attention [11]. In addition to prefetching on mouse hover, Falcon can use long idle times between interactions to precompute additional indexes.

4 FALCON SYSTEM IMPLEMENTATION

We now discuss how Falcon implements the interactions just described in section 3.

An Index of Data Tiles

A Falcon index contains data needed to render passive views for every possible brush in the active view. The data for binned aggregate views with zero, one, or two grouping dimensions can be stored in a zero-, one-, or two-dimensional array; this projection of the data cube [20] is called a *cube slice*. For example, the histogram in Figure 5 needs an array with 24 entries for the flights for each hour of the day.

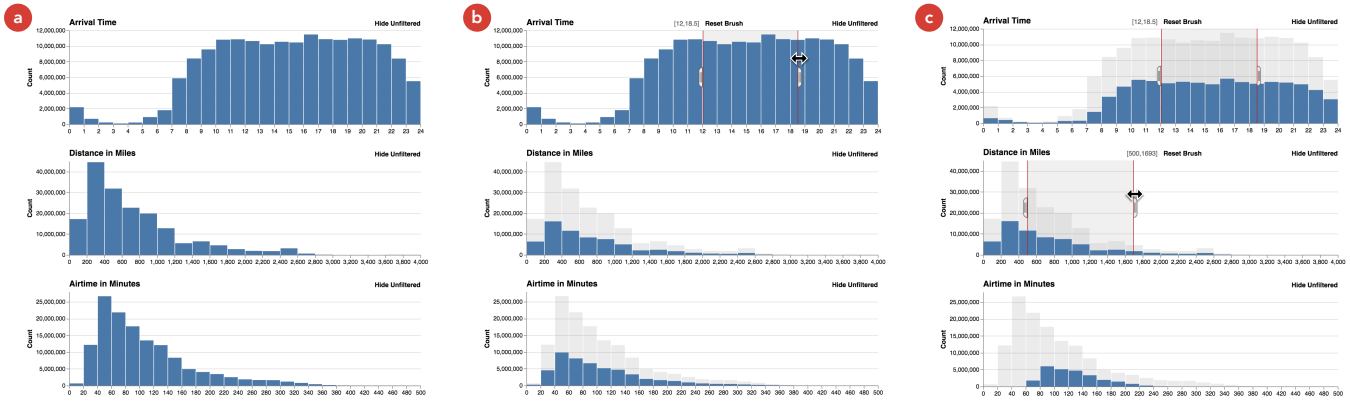

Figure 4: View switching in Falcon. (a) When the user initially loads Falcon, it shows unfiltered histograms. (b) The user can draw a brush in the histogram of the arrival time (active view), and all other passive views will be updated. (c) After a view switch the distance histogram is active, and the user can draw a brush there.

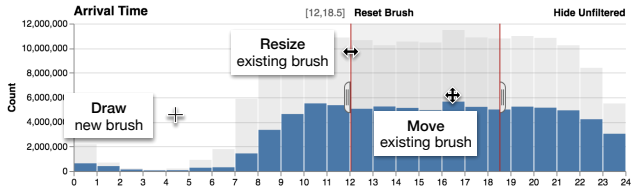

Figure 5: Brush interactions in Falcon. Users can draw a new brush or move and resize an existing one.

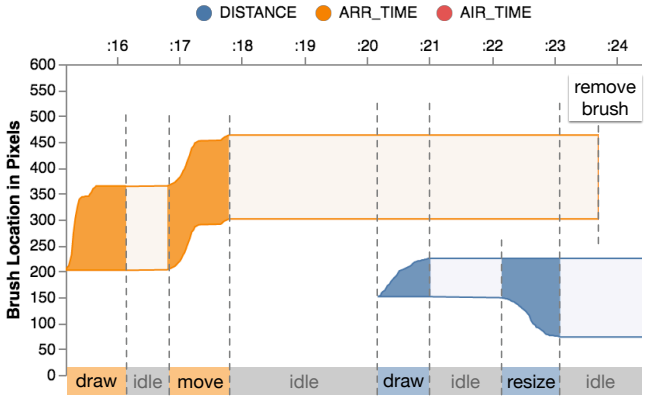

Figure 6: Visualization of the timing for the brushing interactions in Figure 4. The user first draws and then moves a brush in the arrival time histogram before drawing and resizing a brush in the distance view. Finally, the user deletes the arrival time brush. Between interactions, the app is idle, waiting user inputs.

The user can draw brushes in one-dimensional histograms or two-dimensional visualization of bin counts. For a 1D histogram, there are in theory an infinite number of possible brush configurations. However, in a pixel display a histogram that is p pixels wide has only p^2 distinct brushes, with a brush start and end at two pixel locations. Storing p^2 cube slices

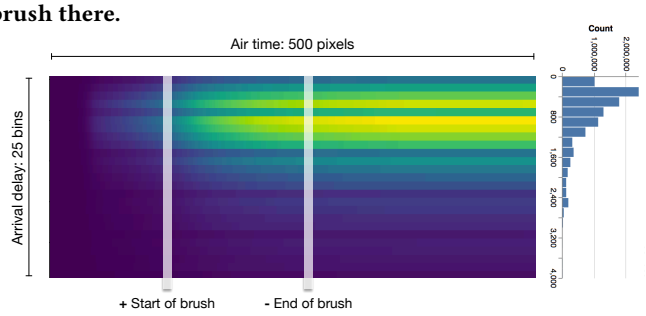

Figure 7: A visualization of a data tile with departure time as the active view and distance as the passive view. A lighter color indicates larger cumulative counts. A histogram for the passive view conditioned on a brush can be computed as the difference between the cumulative bin counts at the end of the brush and the start of the brush.

for each passive view remains prohibitively large. Falcon therefore encodes these p^2 slices as p *cumulative* slices and stores these cumulative counts in a single multidimensional array, which (following imMens) we call a *data tile*.

Figure 7 shows a data tile with airtime as the active view and arrival delay as the passive view. Since each column stores the sum of all counts from the start, a specific cube slice is the difference between the cumulative slices for the start and end of the brush. For a fixed number of bins, this difference is computed in constant time ($O(1)$).

Computing a sum (e.g., of counts) as the difference of cumulative sums is often used in computer graphics and is known as *summed area tables* [14] or *integral images*. Summed area tables generalize to many dimensions; in Falcon, we use the same approach for brushing in 2D views. Here, a cube slice for a passive view is computed from the four corners of the brush in the active view. A data tile stores the cumulative sums along the dimensions of the active view.

In addition to a data tile for each passive view, the index must also store the cube slice for the case where there is no

brush. This is necessary because a brush that spans the full extent of the active view does not contain all records in the raw data when a user has zoomed in on a view.

In the general case, a data tile is an array whose dimensionality is the sum of the dimensionalities of its corresponding active and passive views. In Figure 7, the active and passive dimensions are each grouped by a single dimension, so the data tile has two dimensions. This concept generalizes: to support brushing in a 2D active view and filter a 2D passive view, we need a four-dimensional array.

The size of a data tile is the product of the bin counts for each dimension of both the active and passive views. The number of data tile bins corresponding to the active view depends on the active view's pixel screen size. The ratio of active view pixels to corresponding data tile bins determines the interactive resolution, which is at most 1 pixel per bin.

Computing Data Tiles

We implemented two query systems to compute data tiles for Falcon. The first is a query engine that runs in the user's browser alongside the Falcon UI. This engine supports queries over tens of millions of records (as Apache Arrow files [18]), above which latency becomes unacceptably large. The second system generates SQL queries for a database server. The scalability of this approach depends on the database system.

Both engines use a similar approach to compute data tiles. For each passive view, they aggregate the records that are not filtered out by any brush in other passive views. From these counts, the engines build the cumulative data tile.

In-Browser Engine. The engine in the browser computes a data tile by first creating an empty multidimensional array of the binned dimensions. In a single pass over the filtered data, it then counts how many records fall into each cell of the array. In a final step, it computes the cumulative sums along the dimensions of the active view. While the engine iterates over the records, it counts how many records match the filters in the other passive views but fall outside the extent of the active view; it uses these values to determine unfiltered counts. Since the size of the data tile is independent of the size of the data, the running time of the last step is bounded only by the number of bins in the dimensions of the active and passive views. We implemented the engine in JavaScript; thus, it is single-threaded and blocking.

Engine for Database Server. The database engine issues queries that perform binning and aggregation in a scalable database system, such as OmniSci Core [35] (formerly known as MapD). Falcon generates aggregate queries that filter by the brushes in the other passive views and group by the bins in the dimensions of the active and passive views (Figure 8). The same query counts the records that fall outside the extent of the active view. Because the queries for each passive view use different filters and group-by clauses, they cannot be naturally expressed as a single query. Query results are received client-side and written into a multidimensional array. Falcon computes the cumulative counts in the client, since some databases (including OmniSci Core) do not support window aggregates, which are necessary to compute cumulative counts efficiently. Queries execute asynchronously without blocking the UI.

```
SELECT
    CASE
        WHEN airtime BETWEEN 0 AND 500
        THEN floor((airtime - 0) / 1)
        ELSE -1 END AS binned_airtime
    , count(*) AS cnt
    , floor((arrdelay - -20) / 5) AS binned_delay
FROM flights
WHERE arrdelay BETWEEN -20 AND 60
GROUP BY binned_airtime, binned_delay
```

Figure 8: The SQL query to compute the counts for Figure 7 and a special bin (-1) for the unfiltered counts. The cumulative counts are computed on the aggregated data.

Progressive Interaction

Switching the active dimension and zooming are not as latency-sensitive as brushing. Nonetheless, delays may be frustrating to users. To address this issue, we propose *progressive interaction*, an analog of progressive refinement of approximate aggregate queries [21] or progressive loading of images [4] and data [19], but in the interaction space. It works as follows: initially, Falcon loads small data tiles, where the bin count of the dimensions of the active view is lower than the pixel count. The user can interact with the active view, but the brush will snap to the closest available data tile bin boundaries. Falcon then loads data tiles for interactions at the pixel resolution in the background. Our current prototype implements progressive interactions in two steps: (1) Falcon loads data at the resolution of the visible histogram bins, and (2) Falcon loads the full pixel resolution.

Interpolation

When brushes snap to the closest bins, users cannot set brushes at the pixel resolution. Continuous brushes have two advantages. First, users can set brushes between bin boundaries. Second, histograms in the passive views change smoothly as users brush. When Falcon starts with low-resolution data in progressive interaction, it approximates brushing at pixel resolution using interpolation. We interpolate between bin counts for the passive view at the bin boundaries closest to the current brush ends. Though interpolation errors are usually small and resolved as soon as high-resolution data arrives, interpolation remains an approximation with unknown error bounds. To make users aware of this, we decrease the opacity of passive views while users interact with

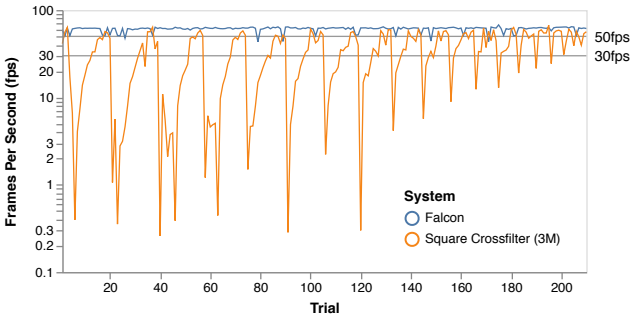

Figure 9: Latency between brush interactions with one chart and updates to 5 passive views, averaged across 5 trials. We compare Falcon to Square's Crossfilter with 3 million records. Falcon's performance is constant, close to the browser's maximum frame rate of 60fps regardless of the full dataset size. Crossfilter reacts slowly when many records are added to or removed from the brush.

an active view with low-resolution data tiles. Interpolated bin counts for smooth brushing at pixel resolution let users place brushes at a precise location; this helps users place multiple brushes without waiting for a full resolution version to load. Falcon uses linear interpolation for 1D brushes and bilinear interpolation for 2D brushes.

5 BENCHMARK EVALUATIONS

We now present a benchmark evaluation of Falcon's brushing performance and the cost of indexing datasets of different sizes. Falcon reduces latency by progressively computing indexes with increasing resolution. We measure the time to compute an initial low-resolution index and the errors of interpolating pixel-level interactions from this data. We then discuss our results and their implications.

Brushing Performance

Figure 9 compares Falcon's brushing performance to Square's Crossfilter. For this benchmark, we programmatically update the brush by iteratively changing its start and end. We run experiments on a 13" 2014 MacBook Pro using the Chrome 70 browser, with Falcon performing all indexing on the client. Using a chart configuration of 6 histograms for flight delays [31], Falcon consistently updates the 5 passive views at more than 50 frames per second. Due to its incremental processing of original data records, Crossfilter's query update times spike when the brush moves over parts of the view where many records are either added to or removed from the filter. In addition, Crossfilter needs ~10s to parse the CSV file and ~30s to initialize internal data structures for 3M records. Falcon works on binary data [18]—there is no parsing or initialization cost—and for up to 10M records requires less than one second to switch views.

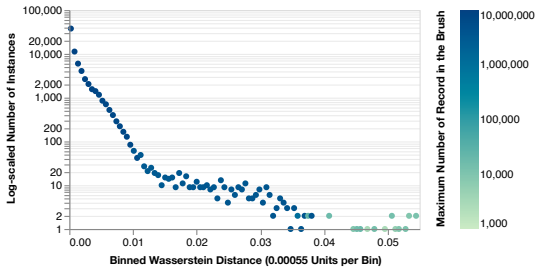

Figure 10: Wasserstein distance between the true and interpolated bin counts for various brushes in the flight dataset. Most instances have small distances. Some instances with high selectivity (few tuples remain) have distances over 0.04.

View Indexing Cost

Before a user switches views, Falcon prefetches data tiles for all passive views. Table 2 shows the mean, median, and 95th percentile of the time it takes to prefetch data for all passive views for different configurations and data sizes. We measure indexing times for high resolution (500 pixels for 1D and 200 × 200 pixels for 2D) and bin resolution (25 and 25 × 25 bins). We use three datasets: *flights*, *weather*, and *GAIA*. The *flights* dataset [31] contains information about flight time, length, distance, and delays for all 180M commercial flights in the U.S. since 1987. The *weather* dataset [30] contains NOAA weather statistics for different locations in the U.S. *GAIA* [5] is a sky survey from the European Space Agency with records for more than a billion stars.

We find that indexing time unsurprisingly increases with data size. In the browser, view switching times stay below 5s even for datasets with 10 million records. Computing a low-resolution index does not reduce indexing time in the browser but can reduce average time by up to 6× with a backing database server (here, OmniSci's Core). The time to load the first data tile in an index from Core is up to 24× faster than the time to finish loading all data tiles in an index.

Approximation Error of Interpolated Brushes

To support brushing at pixel resolution even with low-resolution data tiles, users can enable interpolation. In this experiment, we measure interpolation error using the Wasserstein metric (i.e., earth mover's distance) between interpolated and true bin counts. The metric is 1 when the complete mass of a distribution must be moved from one end to the other. We compare interpolated bin counts (from data tiles with 25 bins in the active dimension) in the flight dataset to true bin counts for a various systematically enumerated brushes. As Figure 10 shows, the majority of the cases show small errors ($<< 0.01$). The error is largest (> 0.04) for highly selective filters ($< 0.2\%$) since bin counts with few records are more susceptible to noise, and the Wasserstein metric compares two distributions.

Dataset	Engine	Size	Views	Indexing at Pixel Resolution						Indexing at Bin Resolution					
				Mean		Median		P_{95}		Mean		Median		P_{95}	
Weather	Browser	1M	$1 \times 0D$, $6 \times 1D$	0.34		0.33		0.38		0.33		0.33		0.38	
Weather	Browser	3M	$1 \times 0D$, $6 \times 1D$	1.0		1.0		1.1		1.0		1.0		1.1	
Weather	Browser	10M	$1 \times 0D$, $6 \times 1D$	1.2		1.2		1.3		1.2		1.2		1.2	
Flights	Browser	1M	$1 \times 0D$, $4 \times 1D$, $1 \times 2D$	0.29		0.31		0.52		0.30		0.28		0.38	
Flights	Browser	3M	$1 \times 0D$, $4 \times 1D$, $1 \times 2D$	0.92		0.88		1.2		0.95		0.90		1.1	
Flights	Browser	10M	$1 \times 0D$, $4 \times 1D$, $1 \times 2D$	3.0		2.8		3.9		2.8		2.6		3.4	
Flights	Core	7M	$1 \times 0D$, $4 \times 1D$, $1 \times 2D$	0.15	0.15	0.13	0.11	0.30	0.13	0.13	0.11	0.13	0.11	0.15	0.13
Flights	Core	180M	$1 \times 0D$, $4 \times 1D$, $1 \times 2D$	2.1	0.34	1.7	0.33	3.9	0.47	1.2	0.36	1.3	0.34	1.5	0.46
GAIA	Core	1.2B	$1 \times 0D$, $3 \times 1D$, $2 \times 2D$	33.8	1.4	6.5	1.3	94	2.6	5.8	1.0	5.3	0.93	9.5	1.5

Table 2: Mean, median, and 95th percentile time in seconds to compute data tiles for all views for different dataset sizes across 5 runs and for pixel resolutions (500 for 1D and 200 × 200 for 2D) and bin resolutions (25 and 25 × 25 bins). Times for OmniSci's Core include network roundtrip on a university network when accessing a cloud-based instance. Gray colors show the time until the data tile for the first view is computed (applies only to non-blocking requests).

Discussion

Our benchmarks show that Falcon delivers constant brushing performance regardless of how many records match the filter defined by a brush. Its update rates are comparable to those of imMens and outperform those for Square's Crossfilter. Unlike imMens, Falcon supports higher interactive resolutions and multiple brushes. This demonstrates that even though our prefetching methods were designed for client-server applications with massive datasets, precomputing a fixed-size index also benefits client-only applications.

In addition to low latency, constant and predictable performance are important. When a system behaves inconsistently, users may adapt by only performing those interactions that are fast [26]. For the Square Crossfilter application, users might begin to explore only histograms showing few changes. Falcon decouples brushing actions from the full dataset, and all computations have constant complexity with respect to data and brush sizes. Future systems might consider not just the average or worst performance but also the degree to which performance varies for different interactions.

In-Browser Engine Performance. Indexing times for small datasets (≤ 1M records) in Table 2 are at most a few hundred milliseconds, about the time needed to hover over a view and begin to brush, so users may not even notice a delay. For datasets up to 10M records, indexing times in our browser engine are at most a few seconds, as shown in Table 2. After these few seconds, users can brush without delays, a trade-off not available in previous systems [24, 27, 32].

Since computing a low-resolution index in the browser is not significantly faster than computing a high-resolution one, we do not use progressive interaction here. Although a high-resolution index has more bins, the vast majority of time is spent iterating over the full dataset. The index has the same size regardless of the full dataset's size.

Database Engine Performance. For datasets that are too large to be loaded and processed in the browser, Falcon can issue queries to a database system. Our prototype uses OmniSci Core, one of the fastest analytics database systems available [25]. Our benchmark evaluation shows that the time to compute an index for the dataset of all commercial flights in the U.S. (180M records) never exceeds a few seconds. However, for the GAIA dataset, computing a high-resolution index can take more than a minute.

A significant amount of time is spent receiving and deserializing database query results. Some of this overhead results from limitations in the OmniSci API. For instance, aggregated counts must be sent as a relational table instead of as a dense multidimensional array. Nevertheless, while the index is loading, users can already be drawing a brush in the active view.

To further improve the user experience, Falcon leverages two observations from the benchmark. First, loading the initial data tiles takes about a second. Thus, Falcon's UI can update individual passive views as soon as their data tiles have been loaded and provide visual feedback. Second, low-resolution indexes load much faster using the database-backed engine. Our progressive interaction method leverages these faster query times to update passive views faster—improving average initial load times by 5×, from ~30s to ~6s—and then progressively improves them as high-resolution indexes are loaded. Our experience shows that view switching times are reasonable, but more careful assessment of how these delays affect people's behavior remains as future work.

Falcon's interpolation of high-resolution interactions from low-resolution data tiles enables progressive interactions without snapping brushes to the low resolution. Our measurements in Figure 10 show that the interpolation error is negligible in most cases. The largest errors occur when brush filters are highly selective. Since Falcon visualized bin

counts by default on the scale of unfiltered data (Figure 3), the filtered bin counts and any visual differences in the chart are small. Moreover, when filters are highly selective and visualizations of aggregates are based on few records, the visual gestalt of the chart is susceptible to noise in the data. In general, analysts should make judgments about distributions based only on large samples.

6 LIMITATIONS AND FUTURE WORK

Falcon's index is significantly smaller than that in previous approaches (e.g., [27]), allowing it to be computed on the fly from a single scan by the backing database under appropriate latency assumptions. To support datasets that are too large to be scanned within a given latency threshold, Falcon uses existing databases to compute the necessary aggregates; it can take advantage of approximation and sampling techniques from the database literature. However, our prototype does not yet apply these approximation techniques.

For binned aggregate plots, the data necessary to render visualizations depends only on pixel resolution and not data size. This common assumption enables visualizations whose visual complexity is invariant to the size of the full dataset. Thus, Falcon does not support non-aggregated views where each record is rendered as a separate mark. Future work might involve separate marks for outliers. Our prototype system implements aggregate visualizations of the counts with zero-, one-, and two-dimensional grouping by bins. We have not yet implemented grouping by categorical dimensions, but we plan to add them.

Falcon assumes that a user interacts with a single view at a time. On a desktop computer with a mouse, this assumption is trivially met. However, on touch-enabled devices, a user could use both hands to modify multiple brushes simultaneously. We believe that this scenario is rare. We conducted informal user observations with an iPad, and none of the participants attempted to use simultaneous brushes. Nonetheless, a future version of Falcon could support simultaneous brushes by combining the dimensions of multiple active views, though at the cost of much larger data tiles.

While Falcon prioritizes brushing and linking as the most latency-sensitive interactions [26], future systems should use techniques presented here to prioritize zooming or other interactions. By prefetching data at different zoom levels, a system could support continuous zooming and re-binning. Battle et al. [3] demonstrate prefetching techniques for panning interactions and show that prediction models can help prioritize which data to prefetch. This was not necessary in Falcon since the computation of data for one brush or for all brushes both require one pass over the data. However, future systems could use prediction models and prefetch in multiple interaction vectors: e.g., linked brushing, linked selection, zooming, and panning.

The Falcon system does not take advantage of concurrent queries: the in-browser engine is written in JavaScript and thus single threaded, while OmniSci Core executes queries sequentially. Future system iterations could speculatively precompute indexes for interactions with a non-active view. The cost of such aggressive prefetching could be offset by caching results in a middleware layer, possibly even for other users. The middleware could also leverage structure in the data tiles to compress them. Neighboring cells in a data tile often have similar values (see Figure 7, section 5), which is similar to images or videos. The large body of work on perception-aware image and video compression could be applied to compressing data tiles between the server and client. Compression could significantly reduce the time needed to transfer a Falcon index from server to client.

To support constant latency for brushing interactions, we limit Falcon to summable aggregate functions (e.g., sum and count). Our prototype only implements count. Some aggregate functions are algebraic, meaning they can be constructed as a combination of summable functions; this includes the mean (sum, count) and variance (sum, count, sum of squares). Distributive functions (e.g., min and max) can be computed by iterating over matching bins, which results in a linear (or, with extra data structures, logarithmic) lookup time with respect to brush size. Future iterations of Falcon could implement these aggregate functions.

7 CONCLUSION

In this paper we contribute the idea of prioritizing brushing latency over view switching latency, as suggested by prior work on the impact of latency on analysts' behavior. We also show that it is possible to lower the initial resolution of interactions to improve view switching times. We implement these methods in Falcon, our prototype system. Falcon supports brushing and linking across views over datasets of tens of millions of records in the browser and billions of records when connected to a backing database system, without the need for costly precomputations or significant limitations to supported interactions.

ACKNOWLEDGMENTS

Matthew Conlen helped to develop an earlier prototype system. We thank Leilani Battle for technical feedback and Sandy Kaplan for writing feedback. Daniela Huppenkothen helped us understand the GAIA data. Jennifer Rogers supported us with her knowledge about distance metrics. Brian Hulette and the Arrow team helped implement the in-browser engine. The OmniSci team, and in particular Venkat Krishnamurthy, Todd Mostak, and Randy Zwitch, helped us with the Core database. This work was supported by a Moore Foundation Data-Driven Discovery Investigator Award, NSF Award 1740996, and by Microsoft.

REFERENCES

[1] Dana H Ballard, Mary M Hayhoe, Polly K Pook, and Rajesh PN Rao. 1997. Deictic codes for the embodiment of cognition. *Behavioral and Brain Sciences* (1997).

[2] Leilani Battle, Remco Chang, and Michael Stonebraker. 2016. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *International Conference on Management of Data (SIGMOD '16)*. ACM. https://doi.org/10/gd7hg9

[3] Leilani Battle, Michael Stonebraker, and Remco Chang. 2013. Dynamic reduction of query result sets for interactive visualizaton. In *Conference on Big Data*. IEEE. https://doi.org/10/gd7hgw

[4] Thomas Boutell. 1997. *PNG (portable network graphics) specification version 1.0.* Technical Report.

[5] A G A Brown, A Vallenari, T Prusti, J H J de Bruijne, C Babusiaux, C A L Bailer-Jones, Gaia Collaboration, and Others. 2018. Gaia Data Release 2. Summary of the contents and survey properties. *arXiv preprint arXiv:1804.09365* (2018).

[6] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman (Eds.). 1999. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann Publishers Inc.

[7] Stuart K. Card, Allen Newell, and Thomas P. Moran. 1983. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc.

[8] Sye-Min Chan, Ling Xiao, J. Gerth, and P. Hanrahan. 2008. Maintaining interactivity while exploring massive time series. In *IEEE Symposium on Visual Analytics Science and Technology*. https://doi.org/10/fbqvj9

[9] Surajit Chaudhuri and Umeshwar Dayal. 1997. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record* (1997). https://doi.org/10/bst468

[10] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *International Conference on Management of Data (SIGMOD '17)*. https://doi.org/10/ct5p

[11] Mon-Chu Chen, John R. Anderson, and Myeong-Ho Sohn. 2001. What can a mouse cursor tell us more?: correlation of eye/mouse movements on web browsing. In *CHI Extended Abstracts*. https://doi.org/10/d7rvs2

[12] William S. Cleveland and Robert McGill. 1984. Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods. *J. Amer. Statist. Assoc.* (1984). https://doi.org/10/gdvmwd

[13] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2015. Vizdom: Interactive Analytics Through Pen and Touch. *VLDB Endowment* (2015). https://doi.org/10/gd7hg3

[14] Franklin C. Crow. 1984. Summed-area Tables for Texture Mapping. In *Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*. https://doi.org/10/bbrtvf

[15] Punit R. Doshi, Elke A. Rundensteiner, and Matthew O. Ward. 2003. Prefetching for Visual Data Exploration. In *Conference on Database Systems for Advanced Applications (DASFAA '03)*. IEEE.

[16] Philipp Eichmann, Carsten Binnig, Tim Kraska, and Emanuel Zgraggen. 2018. IDEBench: A Benchmark for Interactive Data Exploration. *CoRR*.

[17] Danyel Fisher, Igor Popov, Steven Drucker, and m.c. schraefel. 2012. Trust Me, I'm Partially Right: Incremental Visualization Lets Analysts Explore Large Datasets Faster. In *Conference on Human Factors in Computing Systems (CHI '12)*. https://doi.org/10/f3tvr5

[18] Apache Software Foundation. 2017. Arrow. https://arrow.apache.org/.

[19] Michael Glueck, Azam Khan, and Daniel J. Wigdor. 2014. Dive in!: Enabling Progressive Loading for Real-time Navigation of Data Visualizations. In *Conference on Human Factors in Computing Systems (CHI '14)*. ACM. https://doi.org/10/cx3f

[20] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. 1996. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *Conference on Data Engineering*. https://doi.org/10/c6dskg

[21] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. In *International Conference on Management of Data (SIGMOD '97)*. ACM. https://doi.org/10/dqxgnm

[22] Ricardo Jota, Albert Ng, Paul Dietz, and Daniel Wigdor. 2013. How Fast is Fast Enough?: A Study of the Effects of Latency in Direct-touch Pointing Tasks. In *Conference on Human Factors in Computing Systems (CHI '13)*. ACM. https://doi.org/10/gd7hgz

[23] Sean Kandel, Ravi Parikh, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2012. Profiler: Integrated Statistical Analysis and Visualization for Data Quality Assessment. In *International Working Conference on Advanced Visual Interfaces (AVI '12)*. ACM. https://doi.org/10/gd7hg2

[24] Lauro Lins, James T. Klosowski, and Carlos Scheidegger. 2013. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics* 12 (2013). https://doi.org/10/bwrc

[25] Mark Litwintschik. 2017. Summary of the 1.1 Billion Taxi Rides Benchmarks. http://tech.marksblogg.com/benchmarks.html

[26] Zhicheng Liu and Jeffrey Heer. 2014. The effects of interactive latency on exploratory visual analysis. *IEEE Transactions on Visualization and Computer Graphics* 12 (2014). https://doi.org/10/f3tvrw

[27] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. 2013. ImMens: Real-time visual querying of big data. *Computer Graphics Forum* (2013). https://doi.org/10/f3tvr4

[28] Microsoft. [n. d.]. PowerBI. https://powerbi.microsoft.com/.

[29] Dominik Moritz, Danyel Fisher, Bolin Ding, and Chi Wang. 2017. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *Conference on Human Factors in Computing Systems (CHI '17)*. ACM. https://doi.org/10/b7jt

[30] NOAA. [n. d.]. Climate Data Online. https://ncdc.noaa.gov/cdo-web/datasets. Accessed: 2018-09-12.

[31] Bureau of Transportation Statistics. [n. d.]. On-Time Performance. https://www.bts.gov/. Accessed: 2018-09-12.

[32] OmniSci. [n. d.]. Immerse, Interactive Visual Analytics for Big Data. https://www.omnisci.com/platform/immerse/.

[33] Cicero A. L. Pahins, Sean A. Stephens, Carlos Scheidegger, and Joao L. D. Comba. 2017. Hashedcubes: Simple, Low Memory, Real-Time Visual Exploration of Big Data. *IEEE Transactions on Visualization and Computer Graphics* (2017). https://doi.org/10/gdrbsx

[34] Y. Park, M. Cafarella, and B. Mozafari. 2016. Visualization-aware sampling for very large databases. In *Conference on Data Engineering*. https://doi.org/10/gd7hg4

[35] Christopher Root and Todd Mostak. 2016. MapD: A GPU-powered Big Data Analytics and Visualization Platform *(SIGGRAPH '16)*. ACM. https://doi.org/10/gd7hg8

[36] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2015. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Trans. Visualization & Comp. Graphics (InfoVis)* (2015). https://doi.org/10/gdqd7v

[37] Tableau Software. [n. d.]. Tableau Desktop. https://tableau.com/products/desktop.

[38] Tableau Software. [n. d.]. Tableau Public. https://public.tableau.com/.

[39] Square. [n. d.]. Crossfilter: Fast Multidimensional Filtering for Coordinated Views. https://square.github.io/crossfilter/.

[40] Hadley Wickham. 2013. Bin-summarise-smooth : A framework for visualising large data. *InfoVis* (2013). http://vita.had.co.nz/papers/bigvis.html

[41] E. Zgraggen, A. Galakatos, A. Crotty, J. Fekete, and T. Kraska. 2017. How Progressive Visualizations Affect Exploratory Analysis. *IEEE Transactions on Visualization and Computer Graphics* (2017). https://doi.org/10/gbns23