Eurographics Conference on Visualization (EuroVis) 2019
M. Gleicher, H. Leitte, and I. Viola
(Guest Editors)

Volume 38 (2019), Number 3

Kyrix: Interactive Pan/Zoom Visualizations at Scale

Wenbo Tao[1], Xiaoyu Liu[1], Yedi Wang[2], Leilani Battle[3], Çağatay Demiralp[4], Remco Chang[2] and Michael Stonebraker[1]

[1] Massachusetts Institute of Technology,[2] Tufts University,
[3] University of Maryland, College Park, [4] Megagon Labs

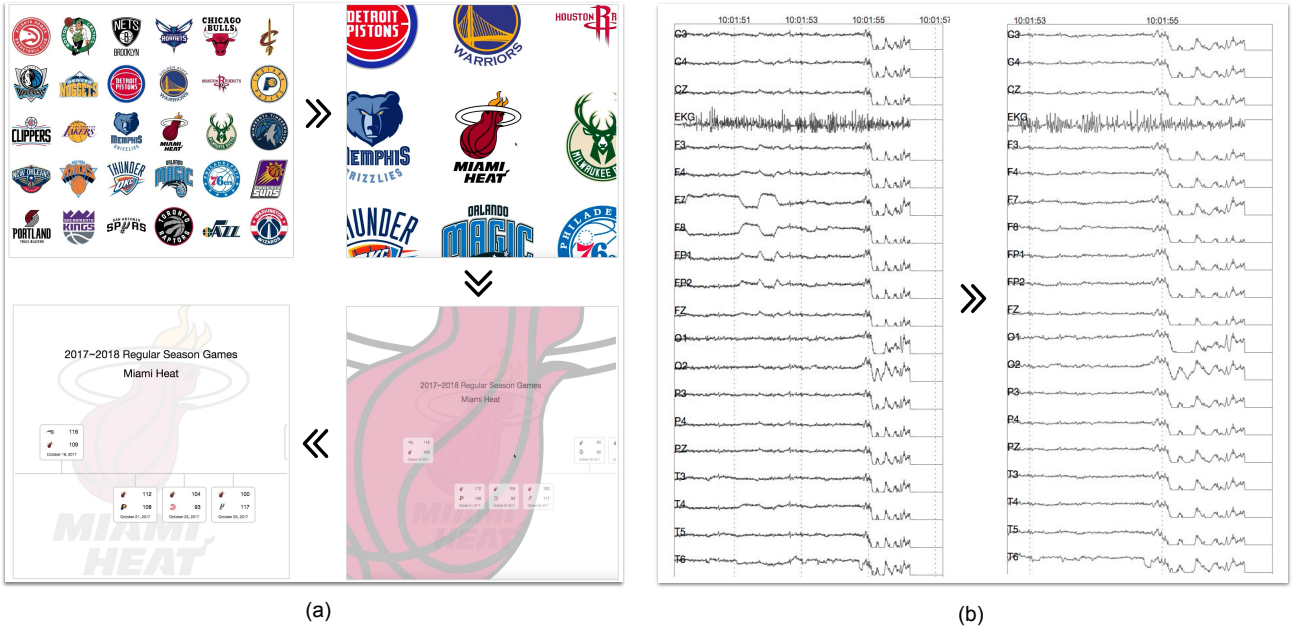

Figure 1: *Two example visualizations created using Kyrix: (a) a visualization of the 2017-2018 regular season of the NBA, where the user can zoom from one view showing NBA team logos to another view showing a timeline of NBA games and (b) a pannable and zoomable EEG time series consisting of 100 million data points.*

Abstract

Pan and zoom are basic yet powerful interaction techniques for exploring large datasets. However, existing zoomable UI toolkits such as Pad++ and ZVTM do not provide the backend database support and data-driven primitives that are necessary for creating large-scale visualizations. This limitation in existing general-purpose toolkits has led to many purpose-built solutions (e.g. Google Maps and ForeCache) that address the issue of scalability but cannot be easily extended to support visualizations beyond their intended data types and usage scenarios. In this paper, we introduce Kyrix to ease the process of creating general and large-scale web-based pan/zoom visualizations. Kyrix is an integrated system that provides the developer with a concise and expressive declarative language along with a backend support for performance optimization of large-scale data. To evaluate the scalability of Kyrix, we conducted a set of benchmarked experiments and show that Kyrix can support high interactivity (with an average latency of 100 ms or below) on pan/zoom visualizations of 100 million data points. We further demonstrate the accessibility of Kyrix through an observational study with 8 developers. Results indicate that developers can quickly learn Kyrix's underlying declarative model to create scalable pan/zoom visualizations. Finally, we provide a gallery of visualizations and show that Kyrix is expressive and flexible in that it can support the developer in creating a wide range of customized visualizations across different application domains and data types.

1. Introduction

Interactive visual data exploration for massive datasets is becoming increasingly important with the rapid generation of data across domains, from healthcare to sciences. Data analysts often have to deal with datasets of sizes in the order of terabytes or petabytes. When exploring data of this size, it is not unusual for them to be burdened by information overload [Wur01], leading to error-prone and prolonged analysis processes.

Pan/zoom interfaces [Bed01, DFW08, DCW12, Goo] have been shown to be effective in facilitating the navigation in large dataspaces. By presenting information in multiple levels of details and enabling the user to smoothly traverse between and within levels, these interfaces reduce the user's cognitive load and help preserve their sense of position and context [Shn96]. Figure 1 shows two example pan/zoom visualizations created using the system we describe in this paper. Figure 1b shows an EEG diagram of one patient in a large US hospital we collaborate with. To detect abnormal patterns in large EEG data, the doctor can *pan* to conveniently scroll through the long time series, or *zoom* in to see larger, detailed views of the visualization. In Figure 1a, a basketball fan can click on[†] the logo of his favorite NBA team in the first view, then zoom into a timeline view showing the team's basketball games.

The usefulness of pan/zoom interfaces has led to the development of a number of zoomable UI (ZUI) toolkits, e.g., Pad++ [BH94] and ZVTM [Pie05]. However, while these toolkits support the developer in designing pan/zoom visualizations, they do not provide the backend database support but instead assume that data can fit in memory. Nowadays, datasets are often too large to fit in memory, containing millions or billions of records that require storage in disk-based database systems [LJH13, CXGH08, LKS13]. Therefore, as data gets large, pan/zoom interfaces developed using existing ZUI toolkits can fail to bound interaction response times within 500ms, which is required for sustaining an interactive user experience [LH14]. In addition, they do not provide data-driven primitives for specifying data-visual mappings. Low-level graphics primitives make it tedious and fault-prone to author large data visualizations [BOH11, SH14]. As a result of these inadequacies of existing ZUI tools, many purpose-built pan/zoom systems (e.g. Google Maps [Goo] and ForeCache [BCS16]) have emerged, using highly-customized solutions to support the exploration of large amounts of data. Nevertheless, these systems are often hardcoded for certain data types and applications and thus cannot be easily extended to support general scenarios.

To ease the creation of general and scalable pan/zoom visualizations, we need tools that can help the developer handle large datasets and use effective optimizations to ensure interactivity. This warrants an integrative approach to data-driven visual specification, where performance optimizations and data are pushed to the server side computation and data management systems.

In this paper, we present Kyrix[‡], an integrated system for developing scalable visualizations driven by pan and zoom interactions.

[†] Point-click serves as a convenience mechanism for quick navigation between levels [PF93, BH94, Pie05].

[‡] Code is available at https://github.com/tracyhenry/kyrix.

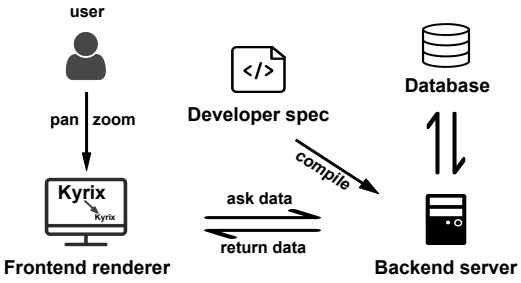

Figure 2: *Kyrix system architecture.*

Our goal is to achieve *generality* (support for general data types and visualizations), *ease of development* and *scalability*. Figure 2 shows the system architecture. On the developer side, we offer a concise yet expressive declarative model for easy specification of pan/zoom visualizations. Declarative designs hide execution details (e.g. backend optimization and frontend rendering) from the developer, so that they can focus on visual specification [SWH14]. On the execution side, the *compiler* parses the developer's specification and performs basic constraint checking. Based on the specification, the *backend server* then precomputes necessary database indexes for performance optimizations. The *frontend renderer* is responsible for listening to user activities, communicating with the backend server to fetch data and rendering the visualizations.

As a unified system, Kyrix contributes the following:

- An integrated visual specification and data management pipeline to ease the creation of large-scale pan/zoom visualizations.
- To our best knowledge, the first declarative model for authoring general pan/zoom visualizations of large, disk-based data (Section 4).
- A suite of performance optimizations that integrate with the underlying data management system to guarantee interactivity on large datasets (Section 5).

We evaluate the expressivity of Kyrix's model through building several example visualizations (Section 6). To assess Kyrix's accessibility, we conduct a developer study with 8 visualization developers recording task performance time and accuracy along with qualitative feedback (Section 7). Results show that developers can quickly learn Kyrix's programming model and create nontrivial visualizations by completing partial specifications. Also, feedback from developers suggests that Kyrix can be valuable in accelerating the development of interactive visualizations at scale, addressing an important need in practice. Lastly, we report results from performance experiments to demonstrate the scalability of Kyrix (Section 8). We find that Kyrix can support interactive exploration over 100 million data points with an average latency of 100 ms or below.

2. Related Work

Kyrix is related to prior research in ZUI design tools, scalable visualization systems and declarative visual encoding.

2.1. ZUI Toolkits and Systems

Perlin and Fox introduces the Pad system [PF93], which redesigns the computer desktop as a fully zoomable user interface. This seminal work has sparked multiple efforts in

designing toolkits to support the creation of ZUIs, including Pad++ [BH94], Jazz [BMG03] and ZVTM [Pie05]. These tools provide application programmers with low-level graphics primitives and have enabled the creation of numerous ZUIs in various domains [Bed01, DCW12, DFW08, Goo, SZG+96, SGKC03, RB05].

Nevertheless, the aforementioned tools cannot scale to large datasets due to two common limitations. First and foremost, they assume data can fit in main memory – an assumption that does not hold for large datasets that require disk-based data storage [LJH13]. Second, they lack data-driven primitives for easy mapping from data to visual properties. For instance, to create visual objects that match a dataset, the developer is required to individually create each visual object and attach pan/zoom event listeners. Similar to how native Javascript hinders large-scale visualization authoring [BOH11], this cumbersome process prevents the developer from reasoning on the data level, and therefore is ill-suited for creating large-scale pan/zoom data visualizations.

In contrast, Kyrix offers an integrated workflow for declarative visual authoring and large-scale data management, providing programmers with high-level data-driven abstractions while freeing them from writing complex execution code to optimize performance and render visualizations.

2.2. Performance Optimization in Visualization Systems

The inability of general ZUI toolkits to handle large data has led to many custom-made pan/zoom systems optimized for specific data types and applications.

Image tile browsers such as Deepzoom [Mic08], Google Maps [Goo] and Zoomify [Zoo99] generally assume or create a pyramid of image tiles with varied resolution, and only render tiles that fall within the viewport. While convenient for viewing a high-resolution image at multiple scales, this rigid paradigm does not work well with general web-based visualizations (unless a tedious conversion from a web-based visualization to multi-resolution images is done first). We will later use an example application (Figure 9b) to show that Kyrix can also be used as an image tile browser.

ATLAS [CXGH08] adopts predicative caching to enable interactive pan and zoom on time series data. In a similar vein, ForeCache [BCS16] prefetches data tiles to efficiently render dense array-based data such as satellite imagery data. Aperture Tiles [CSK+13] precomputes and fetches image tiles from distributed storage systems with a focus on geospatial applications. HiGlass [KAL+18] is a recent system for visualizing genomic data which precomputes image tiles. Different from these purpose-built systems, Kyrix is agnostic to data and visualization types. Kyrix also uses novel databse spatial indexing and extends some of the optimization techniques in these systems (e.g. prefetching and caching) to optimize general pan/zoom interactions.

Prior works have studied in-memory techniques to fetch only needed data in response to user actions. The Splash framework [GKW14] offers the developer an interface for writing a data fetching procedure that returns data items falling in the current viewport. Despite its flexibility, writing this procedure can be nontrivial. Kyrix offers a more lightweight mechanism by allowing the developer to specify a data-driven function that assigns bounding boxes to data items, and then automates the data-fetching process using database spatial queries in a disk-based setting. This idea draws inspiration from Pad++ [BH94] and ZVTM [Pie05] which provide shape-level bounding box specifications.

A long line of research also studies how to reduce visual clutter [ED07] on large data visualizations using techniques such as sampling [DCHW03, DE02, Raf05] and binned aggregation [EF10, GR94]. This type of data manipulation is often performed before data visualization [GR94, GKW14], so we assume this is an orthogonal process that is done either outside Kyrix or through a custom preprocessing procedure (Section 4.4).

Multidimensional data tiles/cubes [LJH13, LKS13, PSSC17, Bre16] have been widely adopted to support interactive aggregation queries. However, due to huge amounts of memory used, the index structures proposed cannot support complex pan/zoom interactions where frequent querying of visual objects falling in a rectangular viewport is needed. In contrast, our method is based on database spatial indexes to perform spatial queries on disk-resident data.

2.3. Declarative Visualization Specification

Kyrix's declarative model is related to earlier research on declarative visual analysis grammars. Wilkinson introduces a grammar of graphics [Wil99] and its implementation (VizML), forming the basis of the subsequent research on visualization specification. Drawing from Wilkinson's grammar of graphics, Polaris [STH02] (commercialized as Tableau) uses a table algebra, which has later evolved to VizQL [Han06], the underlying representation of Tableau visualizations. Wickham introduces ggplot2 [Wic10], a widely-adopted package in the R statistical language, based on Wilkinson's grammar. Similarly, Protovis [BH09], D3 [BOH11], Vega [SRHH16], Brunel [Wil17], and Vega-Lite [SMWH17] all provide grammars to declaratively specify visualizations.

Some of these declarative languages (e.g. D3 [BOH11] and Vega [SRHH16]) are capable of expressing pan/zoom interactions on small data. However, because of their general-purpose nature, the specification is often verbose, involving tens or hundreds lines of imperative event handling [BOH11] or event-driven functional reactive programming code [SRHH16]. Vega-lite [SMWH17] offers much simpler primitives to specify pan and zoom, but at the cost of low customizability. Kyrix enables declarative specification of pan/zoom interactions in a few lines of code, and is flexible enough to support general visualizations.

Part of Kyrix's declarative abstractions shares conceptual similarities with existing grammars. However, our abstractions are designed to delineate pan/zoom visualizations with multiple levels of details, and are conducive to integration with a server-side data management pipeline. For example, while the layer abstraction is common in prior grammars [Wic10, SMWH17], in Kyrix a layer is also associated with a bounding box function to enable fast data fetching on the server side.

3. Design Requirements

We first present a set of requirements we identify before and during the development of our system, inspired by limitations of prior art, established design principles and our multi-year experiences working with visualization users and developers. These requirements inform the choices we make and guide us to refine our design through multiple design iterations.

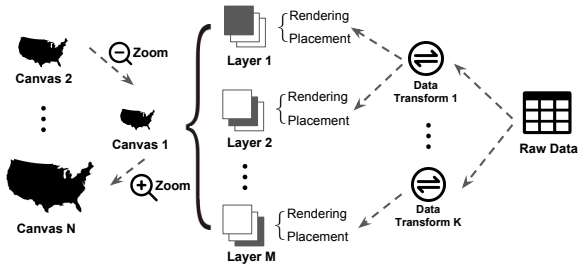

Figure 3: *Kyrix's declarative model. Canvases are "zoom levels" connected by zooms. A canvas has multiple layers. A data transform prepares the data source for rendering a layer. A rendering function maps data to visual objects. A placement function provides spatial information of objects to enable fast data fetching.*

R1. Generality. In terms of design space, our system should support general data types and visual encodings, i.e., not limited to certain data types such as time series data.

R2. Ease of development. From the developer's standpoint, our system should allow simple visual authoring of large visualizations. More specifically, we collect the following sub-requirements:

- **R2-a. Data-driven primitives.** The system should provide data-driven primitives rather than shape-level function calls [BH94, Pie05], which are laborious and error-prone especially for large data. Data-driven abstractions make it more accessible to author data-dependent visual properties [BOH11].
- **R2-b. Easy creation of interactions.** Specifying interactions should be declarative and should avoid complex imperative event handling code.
- **R2-c. Automatic performance optimizations.** To decouple specification from execution details, performance optimizations should be hidden from the developer and performed behind the scenes. This also requires that the specification model provides the backend with enough information to perform optimizations.

R3. Scalability. As established in [LH14], we should bound response times to user operations within 500ms, a crucial threshold for enabling fluid interactions.

4. Declarative Model

Our declarative model contributes an easy mechanism to specify general and scalable pan/zoom visualizations. In this section, we first give an overview of the concepts in our model and then describe them in more detail. We use the basketball data visualization (Figure 1a) as a running example and show in Figures 4–7 relevant specification snippets. In our current implementation, the developer specifies visualizations using Javascript.

4.1. Overview

Figure 3 is an illustration of Kyrix concepts and their relationships. Considering the fact that a pan/zoom data visualization typically comprises multiple levels of details [EF10, GKW14], we naturally use a *canvas* to model one level of details, and use a *zoom* to model that one can zoom in/out from one canvas to/from another. Canvases and zooms form a connected directed graph if we consider canvases as nodes and zooms as edges.

```
1  var viewportWidth = 1000, viewportHeight = 1000;
2  var p = new Project("nba", viewportWidth, viewportHeight);

3  // ================== logo canvas ==================
4  var width = 1000;
5  var height = 1000;

6  // construct a canvas object
7  var logoCanvas = new Canvas("logo", width, height);
8  p.addCanvas(logoCanvas);

   // construct a logo layer (static)
9  var logoLayer = new Layer(transforms.logoTransform, true);
10 logoLayer.addRenderingFunc(renderers.logoRendering);
11 logoCanvas.addLayer(logoLayer);

   // ================== timeline canvas ==================
12 var width = 1000 * 16;
13 var height = 1000;

   // construct a canvas object
14 var timelineCanvas = new Canvas("timeline", width, height);
15 p.addCanvas(timelineCanvas);

   // timeline layer (dynamic)
16 var timelineLayer = new Layer(transforms.timelineTransform, false);
17 timelineLayer.addPlacement(placements.timelinePlacement);
18 timelineLayer.addRenderingFunc(renderers.timelineRendering);
19 timelineCanvas.addLayer(timelineLayer);

   // background layer (static)
20 var timelineBkgLayer = new Layer(transforms.bkgTransform, true);
21 timelineBkgLayer.addRenderingFunc(renderers.bkgRendering);
22 timelineCanvas.addLayer(timelineBkgLayer);
```

Figure 4: *Specifications of canvases and layers for the NBA example in Figure 1a.*

A canvas is composed of one or more overlaid *layer*s. To render a layer, the developer needs to specify a *data transform* as its data source, a *rendering function* mapping data to visual objects and a *placement function* that informs the backend of the locations of the visual objects on the canvas for fast data fetching.

The four views in Figure 1a show the progression of zooming from an initial canvas (showing NBA team logos) to the second canvas (showing a timeline). Note that the second canvas has two layers: a static layer showing a logo background and a title text, and a dynamic layer where the user can pan across the timeline.

In the following, we describe the Kyrix model in more detail. In addition to providing specification details, we present relevant design rationales by connecting the design choices made with requirements established in Section 3.

4.2. Canvas and Layer

A canvas sets up a shared Cartesian coordinate system for its layers. This coordinate system is a rectangular painting area with developer-specified width and height (in number of pixels). If the size of a canvas is larger than the size of the viewport, the frontend renderer automatically enables panning (**R2-b**).

The layer concept in our model is conceptually analogous to the layer operator proposed in existing visual specification grammars [SMWH17, Wic10]. The primary goal is to enable multiple different visual encodings on a single view [Wic10]. Nonetheless, our layer concept has its own unique definition in a large-scale pan/zoom visualization setting.

First, we want to enable a mixed visual representation by allowing a layer to be either *dynamic* or *static*. Dynamic layers move and

```
    // ======== teamlogo -> teamtimeline ========
1   var selector = function (row) {
2       return true;
3   };

4   var viewport = function (row) {
5       return [0, 0];
6   };

7   var predicate = function (row) {
8       return {
9           "layer 0" : "home_team=" + row.team_id + " and "
10              + "away_team=" + row.team_id,
11          "layer 1" : "id=" + row.team_id
12      };
13  };

14  p.addZoom(new Zoom(logoCanvas,
15      timelineCanvas,
16      selector,
17      viewport,
18      predicate));
```

Figure 5: *Specification of the zoom from the logo canvas to the timeline canvas for the NBA example in Figure 1a.*

trigger dynamic data fetching as the user pans on the canvas. Static layers, on the other hand, are for creating static visual objects such as background images, titles and legends.

Second, each dynamic layer is associated with a placement function that is used by the backend to perform fast data fetching in response to pan/zoom interactions (**R2-c**). We describe this concept in more detail in Section 4.6.

4.3. Zoom

A zoom can simply be constructed by specifying a *source* and a *destination* canvas (**R2-b**). The user will then be able to perform continuous geometric zoom on *source* before the magnification reaches the zoom factor (determined by the sizes of two canvases), at which point the scenegraph is updated to show the *destination* canvas. Point-click based shortcut is achieved using smooth zoom transitions.

To enable an expressive design space, we propose several lightweight data-driven abstractions that allow customization of various aspects of a zoom (**R1, R2-a, R2-b**).

Selector enables custom selection of visual objects that can trigger a zoom. In the NBA example, every logo can trigger a zoom into the timeline view (line 1, Figure 5). A more interesting scenario would be that only playoff teams can trigger a zoom into a "playoff" view. A selector is specified using a function that takes a data item as input and returns whether visual objects that this data item is bound to can trigger a zoom.

Viewport customizes the viewport location after the zoom. This is a function that takes the data item bound to the zoomed-in object (the visual object the user clicks on or hovers over when the zoom happens), and returns the coordinates of the new viewport. In the NBA example, this function returns a constant viewport location (line 4, Figure 5) indicating that the user will see the start (leftmost part) of the timeline after the zoom.

Predicate is a data-driven function used to select a subset of data

```
1   var logoTransform = new Transform("logoTransform",
2       "select * from teams;",
3       "nba",
4       function (row){
5           var id = parseInt(row[0]);
6           var y = Math.floor(id / 6);
7           var x = id - y * 6;
8           var ret = [];
9           ret.push((x * 2 + 1) * 80);         // x coordinate of logos
10          ret.push((y * 2 + 1) * 80 + 100);   // y coordinate of logos
11          for (var i = 1; i <= 4; i ++)
12              ret.push(row[i]);               // raw data attributes
13          return ret;
14      },
15      ["x", "y", "team_id", "city", "name", "abbr"]);
```

Figure 6: *Specification of the data transform for a layer in the NBA example in Figure 1a.*

```
1   var timelinePlacement = {
2       centroid_x : "column:x",
3       centroid_y : "column:y",
4       width      : "constant:160",
5       height     : "constant:130"
6   };
```

Figure 7: *Specification of the placement function for a dynamic layer in the NBA example in Figure 1a.*

to render on the destination canvas. For example, in line 7 in Figure 5, the predicate function enables displaying only games of the zoomed-in team. In a sense, this enables "faceting" the destination canvas, i.e., one can create a series of views sharing a common data schema without creating a canvas for each view.

4.4. Data Transform

A data transform serves as the data source for rendering a layer. To support general large disk-based data, our model allows this data source to be specified as a generic query to the underlying database (**R1**). For simplicity, we assume raw data is stored in a relational database for the rest of the paper.[§] Therefore, this query should be a SQL query. Optionally, a *preprocess function* can "cook" raw data into desired form before further passed into the rendering/placement functions. Examples include adding canvas coordinates, scaling and sorting data.

Figure 6 shows the data transform used by the only layer in the logo canvas, which essentially queries team information via a SQL query (line 2) and then calculates canvas coordinates of each logo in a preprocess function (lines 9 and 10).

4.5. Rendering Function

A rendering function is associated with each layer to map data transform results to visual objects. Our model can work with arbitrary renderers that bind data to visual objects (**R1**). The purpose of data binding is to enable data-driven specifications of placement functions and zooms. In the current implementation, we allow Javascript-based renderers (e.g. D3 [BOH11]).

[§] Kyrix currently supports three popular databases: PostgreSQL, MySQL and Vertica. In general, it is straightforward to put Kyrix on top of any database with spatial indexes.

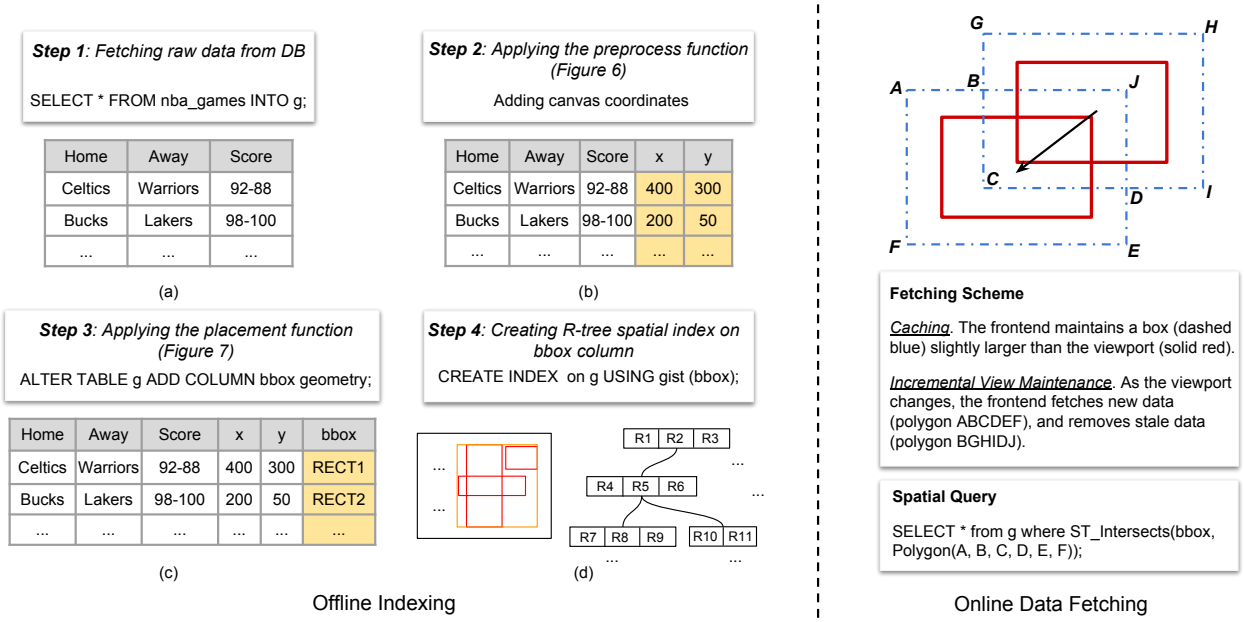

Figure 8: *An illustration of performance optimizations in Kyrix.*

4.6. Placement Function

The key to high performance is to fetch only needed data when the viewport changes. Prior pan/zoom systems [BH94, GKW14] use bounding boxes of shapes to render only shapes whose bounding boxes intersect the viewport. The developer needs to specify a bounding box for each shape, which is tedious and error-prone.

In our model, we extend this idea but instead associate each dynamic layer with a more lightweight data-driven placement function (**R2-a, R2-c**). This function calculates a bounding box for each row in the data transform result representing where this row appears on the canvas. To simplify the specification, we allow the centroid, width and height of a bounding box to be either a constant or a column from the data transform result. An example is in Figure 7.

Compared to the use of bounding boxes in earlier systems, another differentiating factor is that we perform data fetching in a much larger, disk-based setting. We will describe how Kyrix uses the bounding boxes to perform optimizations in Section 5.

4.7. Implementation

We implement Kyrix's declarative language as a *Node.js* library. After the developer specifies an application, the compiler checks whether basic constraints (e.g. a dynamic layer requires a placement function, a canvas must have at least one layer, etc.) are satisfied, and gives error messages if the checking fails. If the specification passes all constraint checks, it is passed to the backend server and saved in the database.

Upon receiving a new specification, the backend precomputes necessary indexes for performance optimizations (details are in the next section). At runtime, the frontend communicates with the backend to dynamically fetch data. The frontend renders visualizations using SVG and uses D3's zoom library [BOH11] to implement interaction listeners and zoom animations.

5. Performance Optimizations

Kyrix uses a suite of performance optimizations to enable fluid interactions at scale (**R3**). All these are done in the backend or the frontend and are transparent to the developer (**R2-c**).

The key optimization problem is how to only fetch visual objects falling into the viewport as the viewport is frequently changed by pan/zoom interactions. A natural idea we adopt is to build spatial indexes (e.g. R-trees [Gut84]) for visual objects and only fetch those whose bounding boxes (specified by placement functions) intersect with the viewport. The idea of using spatial indexes is also adopted in prior systems [BH94, Pie05]. However, they assume the spatial indexes can fit in memory while typical spatial indexes consume linear space in the data size. Therefore, these systems cannot scale to large data.

To support frequent spatial queries at scale, instead of maintaining R-trees in memory, we keep the R-trees on disk by utilizing R-tree indexing offered by modern databases. We describe in Section 5.1 how to build and search disk-based R-tree indexes based on the developer specification.

While disk-based spatial indexing allows for scalability and removes the in-memory requirement of existing ZUI toolkits, there are two challenges when used in highly interactive visualization systems. First, the cost of a lookup (e.g. triggered by a user's pan interaction) is more expensive because each lookup requires issuing a query from the frontend to the backend and further to the database. Rapid user interactions will lead to frequent network and database trips that consume both bandwidth and CPU resources on the backend. Second, when using a disk-based indexing scheme, the backend needs to be aware of the frontend in order to fetch the data items that correspond to the user's interaction and fall within the viewport. To cope with these challenges, we devise caching and view maintenance techniques to reduce the communication be-

tween the frontend and the backend while ensuring interactivity. We describe these techniques in Section 5.2.

5.1. Building and Searching Database Spatial Indexes

Our approach to a disk-based spatial index makes use of an auxiliary table in the database for each dynamic layer specified by the developer. This table is precomputed offline and stores two pieces of derived information for each data item in the layer: (1) the data representation after *data transform* and (2) the bounding box of the visual object (derived from the *placement function*). The R-tree indexes are then built on the bounding boxes. Specifically, the four steps for computing this table are:

- **Step 1**: run the SQL query of the *data transform* to fetch raw data. The backend then processes raw data records one by one. For instance, game records are fetched for the timeline layer of the NBA example (Figure 8a).
- **Step 2**: for each record in the query result, apply the preprocess function defined in the *data transform*. In Figure 8b, canvas coordinates are added to raw data records.
- **Step 3**: for each preprocessed record, apply the *placement function* associated with the layer. This step adds a column typed `geometry` representing the bounding boxes of records (Figure 8c). Modern databases generally have built-in geometry types for representing spatial objects.
- **Step 4**: create an R-tree spatial index [Gut84] on the bounding box column. Modern databases (or their spatial database extensions) generally provide R-tree indexes to efficiently process spatial queries that consider relationships between geometries (e.g. intersection and containment).

To fetch data inside a given viewport, the backend can issue a spatial query that returns all records whose bounding boxes intersect with the viewport. As shown in the bottom right-hand corner in Figure 8, this spatial query has a predicate involving a built-in function `ST_Intersects` applied on the bounding box column. The underlying database will use an R-tree index scan (with logarithmic time complexity) to execute this query.

5.2. Caching and Incremental View Maintenance

Fetching data in exactly the viewport is problematic because every time the user pans or zooms, the frontend needs to send a request to the backend asking for new data, which incurs one network and one database trip. Frequent requests are detrimental and will drain valuable CPU resources on the server side especially in a multi-user setting.

To reduce the number of network and database trips, the Kyrix frontend implements a simple caching strategy that fetches data in a box slightly larger than the viewport (e.g. 50% larger in width/height). This eliminates communication with the backend while the user is exploring inside this box. The right part of Figure 8 illustrates this fetching scheme. The frontend sends a request to the backend to fetch a new box only when the viewport moves close to the boundary of the box (e.g. the distance from the viewport to the box is within one third of the box size).

It is not efficient to fetch an entire new box for each request, since consecutive boxes fetched often have much overlap. Therefore, the backend executes an incremental view maintenance approach by caching the last box fetched and fetching the intersection between the new box and the last one. The intersection is represented as a polygon and fed to the `ST_Intersects` function (see Figure 8). Upon receiving new data, the frontend first renders new data (polygon ABCDEF) and then removes stale data (polygon BGHIDJ).

5.3. Comparison with Existing Optimization Frameworks.

Many purpose-built systems (e.g. ForeCache [BCS16], Aperture Tiles [CSK+13] and HiGlass [KAL+18]) use an "image tiling" framework where a canvas is partitioned into equal-sized tiles that are precomputed offline and fetched online. However, this approach has the following drawbacks. First, when rendering a canvas as images, the frontend loses track of spatial information of objects, making point-click based shortcut more difficult. Second, it is often hard to decide a tile size because small tile sizes lead to excessive network/database trips (one for each tile) while large tile sizes often cause extra data being fetched. In contrast, our use of spatial indexing is novel in that it enables point-click based shortcut by preserving the placement of objects and strikes a balance between database accesses and the amount of data fetched. Note that, however, our spatial index can still be used to fetch data in tiles (without precomputing all tile images). We leave an in-depth performance study on these two data fetching granularities as future work.

6. Example Visualizations

We demonstrate the expressivity of Kyrix's declarative model through a gallery of example pan/zoom visualizations (Figures 1 and 9). In the following, we first describe details of these example applications (Section 6.1). We then use these examples to describe an expressive design space enabled by our model (Section 6.2).

6.1. Using Kyrix's Declarative Model to Create Example Visualizations

NBA. Figure 1a shows two canvases of a basketball data visualization. Descriptions of this example can be found in Section 4.1.

EEG. The visualization in Figure 1b shows an EEG time series of a patient in a large US hospital where doctors apply Kyrix to visualize their data. There are two canvases connected by a zoom (zoom factor is 2). One can zoom from the top canvas into the bottom canvas and see more detailed time series. Both canvases are horizontally pannable. The whole EEG is 7-hour long, consisting of 100 million data points in total.

Cluster. Figure 9a shows a zoomable multi-class scatterplot of 17 million 2-second EEG segments from over 2,000 patients. This data comes from the same US hospital we mention in **EEG**. Doctors use a t-SNE projection [MH08] to map 2-second EEG segments into a 2D space to identify potential clusters and outliers. Different colors represent different EEG patterns (e.g. Seizure). There are 7 canvases (zoom levels) in this example arranged in a multi-scale layout. Random sampling is performed on canvases 1–6 to reduce visual clutter. The bottom-most canvas has all 17 million data points.

Forest. Figure 9b is a map of animals in the Amazon rain forests. There are two canvases (zoom levels), each with two layers. One shows background images. The other layer shows the animals. In the top canvas, animals are previewed as white dots. In the bottom canvas, images of the animals are shown. The background images

Figure 9: *Four more example applications created using Kyrix: (a) a scatterplot visualizing 17 million 2-second EEG segments; (b) a map of animals in the Amazon rainforest; (c) a zoomable crime rate map of the US; (d) a zoomable circle packing layout of the class hierarchy in Flare, an ActionScript library for visualization [UC 08].*

in the bottom canvas are higher-resolution versions of those in the top canvas.

USMap. The visualization in Figure 9c shows a crime rate map of the US. There are two canvases. The top canvas is a state-level map of crime rates per 100,000 population. Darker colors indicate higher crime rates. The user can click on a state to zoom into a second canvas showing a pannable county-level map initially centered at the selected state. Each canvas has two layers: a pannable map layer and a static legend layer.

Flare. Figure 9d visualizes a tree hierarchy, where the classes in the Flare visualization library [UC 08] are arranged in a circle packing layout. The user can click on a class to zoom into another view showing its direct child classes. This visualization is composed of only one canvas, so the zoom object is a self-loop of this canvas.

6.2. Expressivity of Kyrix's Declarative Model

In the following, we demonstrate an expressive design space enabled by our declarative model.

General Data Types and Visualizations. In our model, the data source of a layer is specified using a generic database query. The rendering function for a layer can also be arbitrary renderers with minimal constraints (Section 4.5). Therefore, our model naturally supports generic data types and visual representations (**R1**).

The six example visualizations cover a variety of data types: 2D spatial data (*USMap*, *Forest* and *Cluster*), temporal data (*EEG*), hierarchical data (*Flare*) and general relational data (*NBA*).

Highly Customizable Zooms. The zoom concept in Kyrix's declarative model provides lightweight data-driven abstractions for customizing zooms between canvases (**R2-a, R2-b**).

Zoom selector. The selector function decides which visual objects on the canvas can trigger a zoom. Some example visualizations (*NBA*, *USMap*, *Flare*) utilize this function. For instance, in *USMap*, we use the selector function to ensure that only visual objects on dynamic layers can be zoomed in.

New viewport location. Recall that the viewport function is used to specify the viewport location after a zoom. Besides constant coordinates, we allow this function to return a viewport location using the data item bound to the zoomed-in object. This data-driven viewport location adds more expressivity to our model. For example, in *USMap*, after the user clicks on a state, the zoomed-in view is centered at the clicked state. This is achieved by letting the viewport function return the centroid location of the clicked state scaled by a zooming factor.

Predicate. The predicate function enables custom selections of data on the destination canvas. For example, in *NBA*, the predicate function is used to render games of the zoomed-in team. Similarly in *Flare*, the predicate function is used to select child classes of the zoomed-in class.

7. Developer Study

We conducted an observational study with developers to evaluate the accessibility of Kyrix and its declarative language. We recruited 8 developers with different backgrounds (7 males, 1 female; ages range from 23 to 44) by posting recruitment ads.[¶] All participants reported prior experience using Javascript and SQL. Four of the participants (P1-P4) reported long-term experience in using visualization tools such as D3.js and Tableau. The remaining four (P5-P8) had little or no experience with visualization programming.

7.1. Protocol

Participants were given a tutorial on how to program in Kyrix after filling out a consent form. They were then asked to perform a warm-up exercise, which involved completing the specification of an example visualization used in the tutorial. After the warm-up exercise, participants were asked to complete two programming tasks (with access to the code from the warm-up exercise). Each task involved completing the specification of a Kyrix application, which

[¶] Demographics information were collected in a sign-up form. We excluded one participant due to English communication barriers.

we describe in detail below. Before the start of each task, the experimenter verbally described the task. A completed visualization was also shown to the participants. After the completion of the tasks, the participants were asked to provide feedback by completing a questionnaire and a semi-structured post-study interview.

All tasks were completed on a laptop with a resolution $2,880 \times 1,980$. During the tasks, one experimenter sat next to the participant to observe their coding behavior and answer questions if necessary. We used a think-aloud protocol throughout the study and a second interviewer transcribed notes during each session. We also audio recorded the interviews. Participants were compensated $30 for a 2-hour session.

Task 1. Task 1 required each participant to complete the specification of a scatterplot visualization with one million points (Figure 10) using Kyrix. The scatterplot had two zoom levels (canvases) with two layers on each canvas (one scatterplot layer and one static layer showing a title text). In this task, participants were given completed data transforms along with rendering and placement functions, but were required to complete the definitions of canvases, layers and a zoom. Specifically, Task 1 involved the following specifications:

a) A top-level canvas with two layers.
b) A bottom-level canvas with two layers.
c) A zoom from the top-level canvas to the bottom-level canvas.

Task 2. Task 2 required participants to complete a partial specification of the NBA example in Figure 1a, which has two canvases (logo and timeline) and a zoom between the two. Similar to Task 1, participants were provided with data transforms and rendering functions, and then were asked to complete the following specifications:

a) Two layers on the timeline canvas.
b) The placement function for the timeline layer.
c) The zoom between the two canvases.

7.2. Results and Discussion

Task completion. All participants completed Task 1, Tasks 2a and 2b under minimal or no guidance. Three participants (P1, P2, P5) completed Task 2c under minimal guidance, and the remaining five completed Task 2c with more hints. Finishing times are: $\mu = 17.25$ min, $\sigma = 3.69$ min for Task 1, $\mu = 26.25$ min, $\sigma = 7.07$ min for Task 2.

Ease of learning. In the post-study questionnaire, participants rated the ease of understanding concepts in Kyrix's declarative model on a 5-point Likert scale (1−very difficult, 5−very easy). The results indicate that our model is easy to learn: canvas ($\mu = 4.50, \sigma = 0.76, M = 5, IQR = 1$), layer ($\mu = 4.50, \sigma = 0.76, M = 5, IQR = 1$), data transform ($\mu = 4, \sigma = 0.76, M = 4, IQR = 0.5$) and zoom ($\mu = 4, \sigma = 0.76, M = 4, IQR = 0.5$).

Participants gave many positive comments about canvases and layers in the interview. They thought they were *"intuitive* (P1)," *"really nice* (P2)," *"user-friendly and understandable* (P8)." Some drew connections with concepts in other software packages: *"layering seems familiar to Illustrator* (P1)," *"(layer) It's like Photoshop layers, you don't have to think about it any more* (P6)."

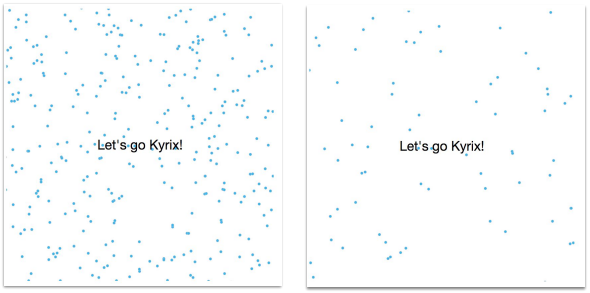

Figure 10: *The scatterplot visualization used both in Task 1 of the developer study and in the performance evaluation: (a) the top-level canvas; (b) the bottom-level canvas. Two canvases are connected by a zoom. The zoom factor is 2.*

Data transform and zoom were not as easy to learn for the participants, as indicated by the relatively lower ratings and longer completion times of Task 2. A recurring pain point we observed was that participants often could not recall what the input to the functions (the data item bound to the zoomed-in object) meant when completing Task 2c, and often confused it with visual objects on the destination canvas. As noted by P7, *"I was a bit confused about the data flow, how data was moving from one view to the other, specifically when defining the predicates."* Our imperfect implementation also contributed to the confusion, which we discuss later in this section.

Fortunately, participants praised the shallow learning curve of our system: *"I don't think it's complicated at all once you get the hang of it* (P1)," *"...once you know what the pieces you need to do, which is probably similar across different projects, you can go a lot faster* (P2)," *"If you get in the mindset of how it works, you can go faster* (P4)."

Ease of coding using Kyrix. Participants rated that it was easy to code Kyrix applications overall ($\mu = 4.50, \sigma = 0.53, M = 4, IQR = 1$, 1−strongly disagree, 5−strongly agree). They also reported that it was straightforward to create a new Kyrix application by just imitating existing ones. One comment from P6: *"It was enjoyable to use it. Once you know the concepts, the declarative part of it is quite clear."*

Scalability and expressivity. Participants liked Kyrix's ability to scale to very large datasets: *"The fact that it can handle a ton of data is really cool* (P2)," *"To plot a huge amount of data, I don't know if there is any tool that can do that in such an easy way* (P4)."

Participants were also impressed by the example visualizations: *"I like the general look and feel of the whole visualization. The way you can jump from one canvas to another, from a visual point of view, it's nice, I like it a lot* (P4).

Suggestions and improvements. When asked about improvements that can be made to Kyrix, most participants pointed out that our current Javascript API was not very polished. For example, we asked developers to write SQL predicates for the predicate function, instead of writing Javascript-style objects which Kyrix could turn into SQL predicates by itself. This actually caused much con-

fusion when participants were completing Task 2c. We plan to address this type of issues by revising our API to be cleaner and more understandable.

P1 and P6 also commented that the time required to precompute indexes (3 minutes for Task 1) hindered the development flow. As visualization developers, they tend to seek more rapid feedback. As P1 noted, *"In lots of tools I used, I try something, compile it, run it and see what happens. In Kyrix the time it actually takes to run it seems slow due to the precomputation."* In the future, we plan to reduce this turnaround time by applying more sophisticated performance optimizations and sampling techniques. More discussion about debugging Kyrix applications is in Section 9.

Note that controlled studies carried in the lab provide useful but partial assessment of accessibility by design. We make the source code of Kyrix available at https://github.com/tracyhenry/kyrix with several real world examples, enabling a broader evaluation of Kyrix in the future that would account for diverse developer backgrounds and workflows.

8. Performance Evaluation

In this section, we report results from performance experiments on two real large datasets (*EEG* and *Cluster*) and synthetic benchmarks. Specifically, we evaluate Kyrix's ability to scale as the data size grows, performance on real applications and effects of the caching and incremental view maintenance strategies. For each dataset, we run a synthetic pan/zoom trace and report the number of data fetching request triggered, the average response time per data fetching request (i.e. time elapsed from the backend receiving the request to the backend getting the data from the database), and the average network transmission time (i.e. the time taken to send the data back to the frontend). We design the synthetic traces to be both challenging (e.g. going through dense areas) and comprehensive (e.g. with alternating pans and zooms). All experiments were run on an AWS EC2 m4.2xlarge instance with 8 cores and 32GB RAM. PostgreSQL 9.3 was used as the backend database. All numbers reported were averaged over three runs.

8.1. Scalability

To test the scalability of Kyrix, we used the scatterplot in Figure 10 as a benchmark visualization and varied the data size from 1 million to 100 million points. We generated the data such that there were always approximately $5,000$ points in the viewport for the top level canvas. We used a user trace where the user first panned 2,000 pixels to the right on the top canvas, then zoomed into the bottom canvas and then panned 2,000 pixels to the right again. The average response and network times are shown in Figure 11.

As can be seen, the latency remained stably under 50 ms as the data size grew. This scalability came from using database spatial index to efficiently fetch data as the user's viewport changes, as well as caching and incremental view maintenance strategies. Note that this does not mean Kyrix can scale to infinite data size. Kyrix's scalability is limited by the underlying database's scalability.

8.2. Performance on Real Applications

In this experiment, we evaluated Kyrix's performance on *EEG* and *Cluster*, two large-scale real applications. The index build time for *Cluster* and *EEG* were respectively 40 minutes and 6 hours.

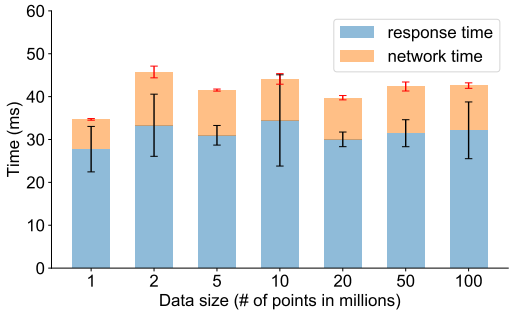

Figure 11: *Kyrix's ability to scale with increasing data size. The scatterplot in Figure 10 is used as the benchmark visualization. Average response time and network latency are shown.*

On *EEG* with 100 million data points, we used a trace where the user first panned 2,000 pixels to the right on the top canvas, then zoomed into the bottom canvas and then panned 2,000 pixels to the right. There were always 40,000 points in the viewport. The average response time per data request was 70.6 ms, while the network transmission time was 29.7 ms on average.

On *Cluster* with 17 million data points arranged in 7 zoom levels, we used a trace where the user first zoomed into the second level, panned 1,000 pixels to the right, then panned 1,000 pixels downwards, and then zoomed all the way into the bottom zoom level. The visual density varies due to skewed data distribution, so the user trace is made to traverse through the densest green area shown in Figure 9a, where around 5,300 points are visible at the same time. The average response time per data request was 15.7 ms. The average network transmission time was 6.4 ms.

These results indicate that not only does Kyrix support the maximum response latency for interactive visualizations of 500ms [LH14], it has the potential to support real-time visualizations. *Cluster*'s total response time of 22.1ms equates to rendering at 45 frames-per-second (fps), surpassing the 30 fps typically required for commercial 3D games [CCD06, CC06]. *EEG*'s response time is 100.3ms, or 10 fps, which is close to the rate where humans perceive animation (instead of individual frames) [BKT86].

8.3. Effects of Caching and Incremental View Maintenance

In this experiment, we evaluated the effects of caching and incremental view maintenance techniques described in Section 5.2.

Caching. We disabled caching (i.e. fetching exactly data in the viewport rather than in a 50% larger box) and tested the performance on *EEG* and *Cluster* using the same user traces. The average response times were respectively 20.6 ms (*EEG*) and 3.9 ms (*Cluster*). Although this was a speedup compared to when caching was used (because it fetched less data), it came at a cost of significantly more data requests (83.3 requests on average vs. 27 on *EEG* and 73.7 vs. 22 on *Cluster*, the fraction was due to subtle differences of network speed across three runs). This result showed that caching could be useful to reduce communication between the frontend and the backend to save bandwidth and CPU resource on the server side, while maintaining desirable interactivity.

Incremental View Maintenance. We disabled incremental view

maintenance and instead fetched the entire box for each request. On *EEG*, the average response time increased by 4.5× to 390.2 ms (was 70.7 ms), whereas the average network latency increased to 265.1 ms (was 29.7 ms), making the overall latency exceed 500 ms. On *Cluster*, the average response time increased to 42.1 ms (was 15.7 ms) and the average network latency increased to 30 ms (was 6.4 ms). This result showed that our incremental strategy greatly reduced both response and network latency by avoiding fetching already fetched data.

9. Limitations and Future Work

While Kyrix eases the creation of scalable pan/zoom visualizations, there is still room for improvement. We identify four areas of future research moving forward.

Performance Hygiene. Currently, the developer needs to carefully design the application so that visual density is not too high (e.g what canvases exist and how data is distributed on the canvases). High visual density can slow down both the frontend and the backend. One future research direction is to detect overly high visual density before runtime, and use sampling or aggregation schemes [BSC13, EF10] or server-side rendering techniques to automatically manage visual density.

Dynamic data. Maintaining the spatial indexes upon data updates is automatically handled by the underlying database (e.g. PostgreSQL automatically updates index when a table is modified [psq19]). However, recall that in Kyrix, spatial indexes are built on auxiliary tables, which are computed from raw data (Section 5). To handle dynamic data, we aim to use database triggers to automatically update the auxiliary tables upon changes to the raw data. This will in turn result in updates to the spatial indexes.

Debugging. As noted in Section 7, the long precomputation time for large data can be detrimental to the iterative debugging workflow of visualization developers. We plan to investigate algorithmic ways to reduce the precomputation time. Another avenue of future research is to augment Kyrix's debugging capabilities with visualizations of canvas and layer states, zooms between canvases, etc.

Higher-level abstractions. Despite that many specifications (e.g. data transforms and rendering functions) can be shared across zoom levels, canvases and rendering functions can be tedious to write. We plan to offer higher-level abstractions that enable the developer to specify, in some cases, just a few parameters (e.g. mappings from data columns to 2D dimensions) and generate the definitions of canvases, layers and rendering functions automatically.

10. Conclusion

To accelerate the development pace of interactive visualization systems at scale, tools are needed to help the developer easily author large-scale visualizations and use effective performance optimizations for sustaining interactive rates. In this paper, we present the design of Kyrix, a novel integrated system for the developer to build interactive pan/zoom visualizations at scale. Kyrix provides a declarative language for easy specification of visualizations, while utilizing Kyrix's suite of optimizations and data management model. Our evaluation of Kyrix has demonstrated that Kyrix meets the design requirements we identify, namely generality, ease of development and scalability.

11. Acknowledgement

We thank the anonymous reviewers for their thoughtful feedback. This work was in part supported by NSF IIS-1452977, DARPA FA8750-17-2-010 and the Data Systems and AI Lab (DSAIL) initiative under Grant 3882825.

References

[BCS16] Leilani Battle, Remco Chang, and Michael Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1363–1375, New York, NY, USA, 2016. ACM. 2, 3, 7

[Bed01] Benjamin Bederson. Photomesa: A zoomable image browser using quantum treemaps and bubblemaps. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, UIST '01, pages 71–80, New York, NY, USA, 2001. ACM. 2, 3

[BH94] Benjamin Bederson and James Hollan. Pad++: a zooming graphical interface for exploring alternate interface physics. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 17–26. ACM, 1994. 2, 3, 4, 6

[BH09] Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2009. 3

[BKT86] Kenneth Boff, Lloyd Kaufman, and James Thomas. Handbook of perception and human performance. 1986. 10

[BMG03] Benjamin Bederson, Jon Meyer, and Lance Good. Jazz: an extensible zoomable user interface graphics toolkit in java. In *The Craft of Information Visualization*, pages 95–104. Elsevier, 2003. 3

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011. 2, 3, 4, 5, 6

[Bre16] Maarten A Breddels. Interactive (statistical) visualisation and exploration of a billion objects with vaex. *Proceedings of the International Astronomical Union*, 12(S325):299–304, 2016. 3

[BSC13] Leilani Battle, Michael Stonebraker, and Remco Chang. Dynamic reduction of query result sets for interactive visualizaton. In *Big Data, 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013. 11

[CC06] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, 2006. 10

[CCD06] Mark Claypool, Kajal Claypool, and Feissal Damaa. The effects of frame rate and resolution on users playing first person shooter games. In *Multimedia Computing and Networking 2006*, volume 6071, page 607101. International Society for Optics and Photonics, 2006. 10

[CSK+13] Daniel Cheng, Peter Schretlen, Nathan Kronenfeld, Neil Bozowsky, and William Wright. Tile based visual analytics for twitter big data exploratory analysis. In *Big Data, 2013 IEEE International Conference on*, pages 2–4. IEEE, 2013. 3, 7

[CXGH08] Sye-Min Chan, Ling Xiao, J. Gerth, and P. Hanrahan. Maintaining interactivity while exploring massive time series. In *IEEE Symposium on Visual Analytics Science and Technology*, pages 59–66, 2008. 2, 3

[DCHW03] Mark Derthick, Michael Christel, Alexander G Hauptmann, and Howard D Wactlar. Constant density displays using diversity sampling. In *Information Visualization, 2003. INFOVIS 2003. IEEE Symposium on*, pages 137–144. IEEE, 2003. 3

[DCW12] Marian Dörk, Sheelagh Carpendale, and Carey Williamson. Fluid views: A zoomable search environment. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, AVI '12, pages 233–240, New York, NY, USA, 2012. ACM. 2, 3

[DE02] Alan Dix and Geoff Ellis. by chance enhancing interaction with large data sets through statistical sampling. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 167–176. ACM, 2002. 3

[DFW08] Raimund Dachselt, Mathias Frisch, and Markus Weiland. Facetzoom: A continuous multi-scale widget for navigating hierarchical metadata. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1353–1356, New York, NY, USA, 2008. ACM. 2, 3

[ED07] Geoffrey Ellis and Alan Dix. A taxonomy of clutter reduction for information visualisation. *IEEE transactions on visualization and computer graphics*, 13(6):1216–1223, 2007. 3

[EF10] Niklas Elmqvist and Jean-Daniel Fekete. Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):439–454, 2010. 3, 4, 11

[GKW14] Michael Glueck, Azam Khan, and Daniel Wigdor. Dive in!: Enabling progressive loading for real-time navigation of data visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 561–570. ACM, 2014. 3, 4, 6

[Goo] Google, Inc. Google maps. https://www.google.com/maps. 2, 3

[GR94] Jade Goldstein and Steven Roth. Using aggregation and dynamic queries for exploring large data sets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 23–29. ACM, 1994. 3

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM. 6, 7

[Han06] Pat Hanrahan. Vizql: a language for query, analysis and visualization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 721–721. ACM, 2006. 3

[KAL+18] Peter Kerpedjiev, Nezar Abdennur, Fritz Lekschas, Chuck McCallum, Kasper Dinkla, Hendrik Strobelt, Jacob M Luber, Scott Ouellette, Alaleh Azhir, Nikhil Kumar, et al. Higlass: web-based visual exploration and analysis of genome interaction maps. *Genome biology*, 19(1):125, 2018. 3, 7

[LH14] Zhicheng Liu and Jeffrey Heer. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics*, 20(12):2122–2131, 2014. 2, 4, 10

[LJH13] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. imMens: Real-time visual querying of big data. *Comput. Graphics Forum*, 32:421–430, 2013. 2, 3

[LKS13] Lauro Lins, James T. Klosowski, and Carlos Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE TVCG*, 19(12):2456–2465, 2013. 2, 3

[MH08] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008. 7

[Mic08] Microsoft Corporation. Deepzoom. https://www.microsoft.com/silverlight/deep-zoom/, 2008. Accessed: 2018-09-19. 3

[PF93] Ken Perlin and David Fox. Pad: an alternative approach to the computer interface. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 57–64. ACM, 1993. 2

[Pie05] Emmanuel Pietriga. A toolkit for addressing hci issues in visual language environments. In *null*, pages 145–152. IEEE, 2005. 2, 3, 4, 6

[psq19] Introduction to postgresql indexing. https://www.postgresql.org/docs/current/indexes-intro.html, 2019. 11

[PSSC17] Cicero Pahins, Sean Stephens, Carlos Scheidegger, and Joao Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *IEEE Transactions on Visualization and Computer Graphics*, pages 671–680, 2017. 3

[Raf05] Davood Rafiei. Effectively visualizing large networks through sampling. In *Visualization, 2005. VIS 05. IEEE*, pages 375–382. IEEE, 2005. 3

[RB05] Gonzalo Ramos and Ravin Balakrishnan. Zliding: fluid zooming and sliding for high precision parameter manipulation. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 143–152. ACM, 2005. 3

[SGKC03] Kenneth Summers, Timothy Goldsmith, Steve Kubica, and Thomas Caudell. An experimental evaluation of continuous semantic zooming in program visualization. In *Information Visualization, 2003. INFOVIS 2003. IEEE Symposium on*, pages 155–162. IEEE, 2003. 3

[SH14] Arvind Satyanarayan and Jeffrey Heer. Lyra: An interactive visualization design environment. *Computer Graphics Forum*, 33(3):351–360, jun 2014. 2

[Shn96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, 1996. 2

[SMWH17] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2017. 3, 4

[SRHH16] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics*, 22(1):659–668, 2016. 3

[STH02] Chris Stolte, Diane Tang, and Pat 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. 3

[SWH14] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. Declarative interaction design for data visualization. In *ACM User Interface Software & Technology (UIST)*, 2014. 2

[SZG+96] Doug Schaffer, Zhengping Zuo, Saul Greenberg, Lyn Bartram, John Dill, Shelli Dubs, and Mark Roseman. Navigating hierarchically clustered networks through fisheye and full-zoom methods. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(2):162–188, 1996. 3

[UC 08] UC Berkeley Visualization Lab. Flare data visualization tool. http://flare.prefuse.org/, 2008. Accessed: 2018-09-19. 8

[Wic10] Hadley Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010. 3, 4

[Wil99] Leland Wilkinson. *The Grammar of Graphics*. Springer, 1st edition, 1999. 3

[Wil17] Graham Wills. Brunel v2.5. https://github.com/Brunel-Visualization/Brunel, 2017. Accessed: 2018-04-04. 3

[Wur01] Richard Saul Wurman. *Information anxiety.* Number 302.234 WUR. CIMMYT. 2001. 2

[Zoo99] Zoomify, Inc. Zoomify. http://www.zoomify.com/, 1999. Accessed: 2018-09-19. 3