Kyrix: Interactive Visual Data Exploration at Scale

Wenbo Tao
MIT CSAIL
wenbo@mit.edu

Xiaoyu Liu
Purdue University
liu1962@purdue.edu

Çağatay Demiralp
MIT CSAIL
cagatay@csail.mit.edu

Remco Chang
Tufts University
remco@cs.tufts.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

ABSTRACT

Scalable interactive visual data exploration is crucial in many domains due to increasingly large datasets generated at rapid rates. Details-on-demand provides a useful interaction paradigm for exploring large datasets, where the user starts at an overview, finds regions of interest, zooms in to see detailed views, zooms out and then repeats. This paradigm is the primary user interaction mode of widely-used systems such as Google Maps, Aperture Tiles and ForeCache. These earlier systems, however, are highly customized with hardcoded visual representations and optimizations. A more general framework is needed to facilitate the development of visual data exploration systems at scale. In this paper, we present Kyrix, an end-to-end system for developing scalable details-on-demand data exploration applications. Kyrix provides the developer with a declarative model for easy specification of general visualizations. Behind the scenes, Kyrix utilizes a suite of performance optimization techniques to achieve a response time within 500 ms for various user interactions. We also report results from a performance study which shows that a novel dynamic fetching scheme adopted by Kyrix outperforms tile-based fetching used in traditional systems.

Keywords

Scalable visual analysis, interactivity with large data, query optimization.

1. INTRODUCTION

Interactive visual data exploration over massive datasets is becoming increasingly important. With the rapid generation of data across domains, it is not unusual for analysts in application domains to deal with datasets of sizes in the order of terabytes or petabytes. Since fluid interactions help allocate human attention efficiently over data [13], interactivity should not be compromised when exploring big datasets, which can easily overwhelm analysts.

9th Biennial Conference on Innovative Data Systems Research (CIDR' 19) January 13-16, 2019, Asilomar, California, USA.

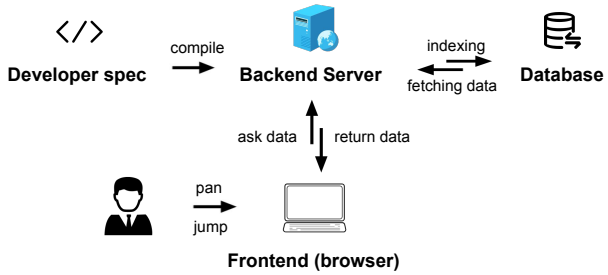


Figure 1: Architecture of Kyrix, an end-to-end system for developing scalable details-on-demand visualizations.

Details-on-demand [21] is a common interaction pattern that arises from exploratory data analysis practices and can be particularly effective in exploring complex datasets, reducing the user's information load. In this paradigm, the user starts with an overview of a dataset and then zooms into a smaller subset of interest within the dataset to examine this data patch [16], while querying details on items within the focused region as needed. The user repeats the same process after zooming further into or zooming out of the current region. However, most visual exploration systems cannot handle very large datasets, let alone enable details-on-demand interactions. Large datasets make it challenging to bound the interaction response times within 500 ms, which is required for sustaining an interactive user experience [13].

Several earlier details-on-demand systems address interactivity challenges at scale with highly-customized implementations. Google Maps and Aperture Tiles [8] precompute image tiles of the entire world map at multiple levels of details. Similarly, imMens [14] supports interactive brushing & linking in binned plots by precomputing tiled data cubes. ATLAS [7] uses predictive prefetching and level-of-detail management to improve the panning and zooming performance on large time-series datasets. ForeCache [2] also adopts predictive prefetching and data tiling to sustain interactive exploration of large amounts of satellite images. Although these earlier systems use similar approaches to scale to large datasets, they are one-off tools developed from scratch for specific datasets. The optimization techniques used in these systems are often inaccessible to visualization developers at large, who are not necessarily experts in performance optimizations. Furthermore, current general-purpose data visualization tools [5, 18, 9] provide limited support for the

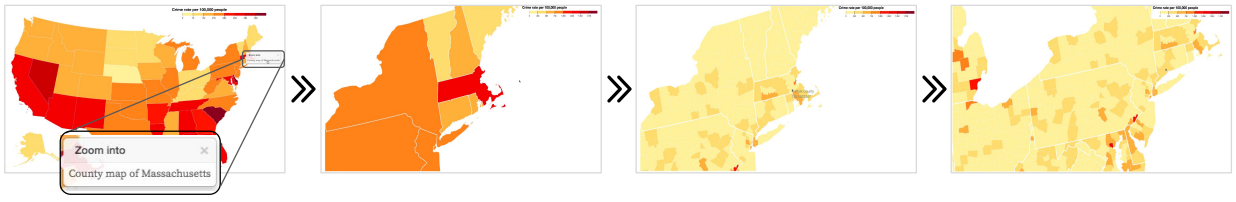


Figure 2: Interactive map of crime rates in the US: (a) a state-level crime rate map where the user can click on a state and zoom into a county-level crime rate map centered at the selected state; (b) Kyrix frontend starts a smooth zoom transition; (c) county-level crime rate map centered at Massachusetts; (d) the user pans on the county-level map.

developer to create visual exploration applications at scale.

To accelerate the development of scalable visual data exploration systems, we need general-purpose tools that can help the developer handle large datasets by using effective optimization techniques (e.g. indexing, caching and prefetching). This warrants an integrative, end-to-end approach to visualization specification, where performance optimizations and data are pushed to the backend server and databases.

In this paper, we present the design of Kyrix, a novel system for the developer to build large-scale details-on-demand visualizations. Our goal is to achieve both generality and scalability. Figure 1 shows the architecture of Kyrix. On the developer side, we offer a concise yet expressive declarative language for specifying visualizations. Declarative designs enable the developer to focus on visual specification without being concerned with execution details (e.g. backend optimization and frontend rendering) [20]. On the execution side, there are three main components: the compiler, the backend server, and the frontend renderer. The compiler parses the developer's specification and performs basic constraint checkings. Based on the developer's specifications, the backend server then builds indexes and performs necessary precomputation. The frontend renderer is responsible for listening to user activities, communicating with the backend server to fetch data and rendering the visualizations.

In the following, we first discuss a simple map visualization created using Kyrix, briefly demonstrating the use of its declarative language. We then introduce optimizations used by Kyrix that facilitate fluid details-on-demand interactions. Next we discuss useful extensions to the Kyrix system along with avenues of future research. We then put Kyrix in the context of earlier scalable visualization systems and visual specification grammars. We conclude by summarizing our contributions and reiterating our vision on accelerating the development of interactive visualizations for massive datasets.

2. DEVELOPING INTERACTIVE VISUALIZATIONS WITH KYRIX

The goal of Kyrix is to provide an end-to-end solution for the developer to create details-on-demand visualizations. To this end, Kyrix offers a declarative language for easy visualization specification.

2.1 Kyrix Declarative Language

Kyrix's declarative model has two basic abstractions: *canvas* and *jump*. A canvas is an arbitrary size worksheet with one or more overlaid *layers*, forming a single view showing a static visualization. A jump is a customized transition from one canvas to another. This model allows easy specifications of common details-on-demand interactions such as panning, geometric and semantic zooming[1].

The Kyrix declarative language is data type agnostic and supports a myriad of specific visualizations. To render a layer, the developer specifies the following:

(1) The data needed for the layer. This is specified using a SQL query to a DBMS along with a transform function postprocessing the query result. The developer can use existing visualization libraries (e.g., D3 and Vega) to specify a desired transform function (e.g., layout transforms, scaling, etc).

(2) The location of each returned data item on the canvas. This is specified using a placement function.

(3) A rendering function that converts a data object to shapes on the screen. A rendering function can be written using lower-level visualization specification libraries such as D3 [5].

A jump transition can be established simply by specifying a *source* canvas, a *destination* canvas and a transition type (currently it can be geometric zoom, semantic zoom or both). It can also be customized in many ways. For example, the developer can specify a subset of objects on the *source* canvas that can trigger this jump. For more details on the language, interested readers can refer to our developer manual.[2]

2.2 Example: Map of US Crime Rates

We now describe an interactive application created using Kyrix. The example visualizes the US crime rates per state and county (Figure 2). There are two canvases in this application. The initial canvas in Figure 2a shows a map of the state-level crime rates. The user can click on a state and zoom into a second, pannable canvas that shows the crime rates at the county level (Figure 2c). In the current implementation, the developer is expected to write specifications in Javascript. Figure 3 shows a snippet for this example

[1]Geometric zooming refers to scaling the visualization to show different levels of details. Data type and visual encoding are unchanged. Semantic zooming, in contrast, connects different views showing related data using smooth zoom-like transitions. Data type and visual encoding can both be changed.

[2]https://github.com/tracyhenry/Kyrix/wiki/API-Reference

```
1  // construct an application object
2  var app = new App("usmap", "config.txt");
3
4  // ========== state map canvas ==========
5  var stateMapCanvas = new Canvas("statemap");
6  app.addCanvas(stateMapCanvas);
7
8  // add data transforms
9  stateMapCanvas.addTransform(transforms.emptyTransform);
10 stateMapCanvas.addTransform(transforms.stateMapTransform)
       ;
11
12 // static legend layer
13 var stateMapLegendLayer = new Layer("empty", true);
14 stateMapCanvas.addLayer(stateMapLegendLayer);
15 stateMapLegendLayer.addRenderingFunc(renderers.
       stateMapLegendRendering);
16
17 // state border layer
18 var stateBorderLayer = new Layer("stateMapTrans", false);
19 stateMapCanvas.addLayer(stateBorderLayer);
20 stateBorderLayer.addPlacement(placements.
       stateMapPlacement);
21 stateBorderLayer.addRenderingFunc(renderers.
       stateMapRendering);
22
23 // ========== county map canvas ==========
24 ...
25
26 // ========== state -> county ==========
27 var selector = function (row, layerId) {
28     return (layerId == 1);
29 };
30 var newViewport = function (row) {
31     return [0, row[1] * 5 - 1000, row[2] * 5 - 500];
32 };
33 var jumpName = function (row) {
34     return "County map of " + row[3];
35 };
36 app.addJump(new Jump("statemap", "countymap", "
       geometric_semantic_zoom", selector, newViewport,
       jumpName));
37
38 // set initial canvas
39 app.initialCanvas("statemap", 0, 0);
```

Figure 3: A Javascript snippet of the US crime rate map example.

application. An application object (Line 2) is constructed by specifying the application name and a configuration file containing information such as the underlying DBMS. The state map canvas is specified in Lines 5–21. This canvas contains two overlaid layers: a static legend layer (lines 13–15) and a pannable state border layer (lines 18–21). Each layer is specified using an identifier of a data transform (lines 9 and 10) and a boolean value indicating whether this layer is static (Lines 13 and 18). Static layers do not need to be re-rendered as the user pans. So as the user browses within a canvas, the legend stays unchanged in the upper right-hand corner, overlaid on the state border layer. The county map canvas is similarly specified. In Figure 3 we leave out the specification of the county map canvas along with the transform, rendering and placement functions due to limited space.

A jump transition from the state canvas to the county canvas is defined in line 36. In the constructed jump object, the first two arguments are respectively the state and county canvases. The third argument specifies the jump type. The rest of the arguments are used to customize the jump transition. To complete specifying the application, the developer also specifies an initial canvas and a viewport location (line 39).

3. INTERACTIVITY IN KYRIX

In general, the interactivity problem in Kyrix is to achieve a 500 ms response time to the following user interactions: (1) A pan to a different location on the same canvas and (2) a jump to a different canvas.

In Section 3.1 we discuss how Kyrix fetches data in response to user interactions. Then in Section 3.2, we give some general guidelines that assist with achieving our goal. Lastly, Section 3.3 gives some end-to-end performance numbers. We discuss in Section 4 other performance options.

3.1 Data Fetching

As the user performs one of the operations (pan or jump), Kyrix's frontend communicates with the backend to retrieve the data needed to render the viewport. Like previous systems (e.g., ForeCache [2]), Kyrix employs both a frontend cache and a backend cache. If there is a cache miss in both, Kyrix backend will talk to the backing DBMS to fetch data. In this data fetching process, we identify two important factors that can affect Kyrix's performance:(1)*fetching granularity* and (2)*database design and indexing*. In the following, we describe these two factors in detail.

Fetching Granularity. The standard wisdom, as applied in Google Maps, ForeCache[2] and Aperture Tiles[8], is to decompose a canvas into fixed-size *static tiles* (Figure 4a). The frontend then requests the tiles that intersect with the given viewport. Every tile is individually fetched and rendered. Kyrix currently supports static tiling. Kyrix also contributes a novel fetching granularity, *dynamic boxes*, which amounts to requesting a box that encompasses the given viewport (Figure 4b). We call this enclosing box a dynamic box because its size and location changes dynamically. Whenever the viewport moves outside the current box, the frontend sends the current viewport location to the backend and requests a new box. There are numerous ways to calculate a box, e.g., a box centered at the viewport center having a width (height) 50% larger than the viewport width (height). We expect dynamic boxes to outperform static tiles for the following reasons:

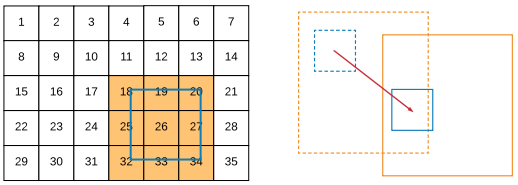
(1) compared to large tiles, dynamic boxes fetch less data;

(2) compared to small tiles, dynamic boxes require fewer frontend-backend requests in general;

(3) in cases where data is not uniformly distributed, dynamic boxes can adjust their sizes and locations based on data sparsity, incurring much fewer network and database trips than static tiles.

In Section 3.3, we use two simple box calculation algorithms to experimentally show that dynamic boxes are a more performant option than static tiles. We leave an in-depth performance study as future work.

Database Design and Indexing. We now describe two database designs along with two indexing schemes that we use to support static tiles and dynamic boxes. Our first database design maps tuples to static tiles and has two tables. The first table is a record table containing all raw data attributes in addition to an auto-increment tuple_id attribute. The second table contains two columns tuple_id and tile_id. Each record in this table indicates that a raw data tuple overlaps a tile. Kyrix backend uses placement functions specified by the developer to precompute the second



(a) Static tiles (b) Dynamic boxes

Figure 4: An illustration of two fetching granularities. (a) A canvas is partitioned into 35 tiles. The blue rectangle is the viewport. Tiles in orange are fetched. (b) Blue rectangles are viewports. Orange rectangles are what are actually fetched. Dashed-line rectangles are before the user pans. Solid-line rectangles are after the user pans.

table. We then build Btree/hash indexes on the tuple_id column of the first table and the tile_id column of the second table. At runtime, tile queries are answered by joining these two tables on the tuple_id column.

Our second database design is based on spatial indexes in PostgreSQL. In addition to raw data attributes, we store a bbox attribute representing the bounding box of a tuple on a canvas.[3] We then build a spatial index on the bbox column. Using this design, queries that request tuples whose bounding boxes intersect with a given rectangle should run fast. Therefore, this design can be used by both static tiles and dynamic boxes.

3.2 Performance Hygiene

Parallelism. We can apply parallelism to improve the data management in Kyrix. All data and metadata (canvas definitions, etc.) are stored in and retrieved from the DBMS. Although the performance experiments in the next section use PostgreSQL, it would be prudent to replace the DBMS with a parallel one if performance requirements warrant a switch. Currently, rendering is performed by a separate process on a separate CPU in the frontend. This operation can also be easily parallelized. Lastly, each concurrent Kyrix application is run in a separate process, since there is no interaction between them, except through the DBMS. Right now, Kyrix applications function like a read-only browser. Future releases will extend Kyrix to allow editing updates, which can be supported by DBMS concurrency control.

Application Design. Managing visual density on the screen, which can overwhelm users as well as the client (e.g., the browser) resources, is an important concern in visualization of large datasets. Application design must deal with what canvases exist and how to put data onto these canvases so that visual density is not too high.

Separability. Recall in Section 3.1, we describe how Kyrix precomputes database tables and indexes to ensure data fetching speed. However, when data is huge or the SQL query corresponding to a canvas layer is complex, this precomputation process can take a long time. We identify a common case where this precomputation process can be avoided: the (x, y) placement of objects are directly raw data attributes, or some simple scaling of raw data attributes.

[3]We assume records are generally rendered bigger than a single pixel. This bounding box information is derived from the placement functions specified by the developer.

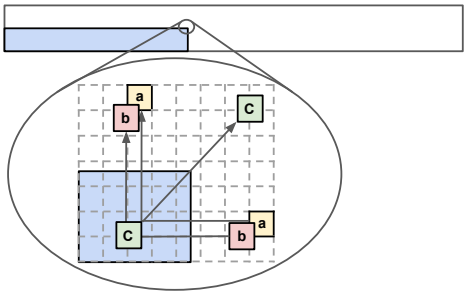


Figure 5: Viewport movement traces used in our experiments. Blue shaded area is the dense area in the dataset SKEWED. Dotted lines are the boundaries of tiles with size 1,024.

In these *separable* cases, if we assume DBAs have built spatial indexes on relevant raw data attributes when data is first loaded into the DBMS, we do not have to precompute the tables described in Section 3.1. For separable cases, we provide the developer with the option to specify the relevant attributes so that precomputation can be skipped by Kyrix. There are cases where this requirement cannot be met, i.e., the placement of an object depends on multiple data attributes or the placements of other objects. We call these cases non-separable. Pie chart is an example.

3.3 Initial Performance Results

We conducted performance experiments on two synthetic datasets using three viewport movement traces. The goal of these experiments is to study the characteristics of the two fetching granularities when combined with different database designs. All experiments are done on an AWS EC2 m4.2xlarge instance with 8 cores and 32GB RAM. PostgreSQL 9.3 is the backing DBMS.

Datasets. We used two synthetic datasets, UNIFORM and SKEWED. In UNIFORM, there are 100M random dots evenly distributed on a 1M×0.1M canvas. In SKEWED, 80M dots lie in 20% of the canvas area (a 0.4M×0.05M rectangle) and 20M dots lie in the rest of the canvas. SKEWED corresponds to the likely scenario when objects are distributed unevenly on a canvas.

Viewport Movement Traces. In our experiments we use three viewport movement traces illustrated in Figure 5.

(a) The viewport is always aligned with tile boundaries. It horizontally moves leftwards six steps (the length of a tile) then vertically up six steps.

(b) The viewport is never aligned with tiles. It also horizontally moves leftwards six steps (the length of a tile) then vertically upwards six steps.

(c) The viewport moves diagonally from bottom left to top right. There are six steps in total.

Fetching schemes. We evaluated the following fetching schemes.

Dbox: Dynamic boxes with spatial index. The box fetched is exactly the viewport in each step.

Dbox 50%: Dynamic boxes with spatial index. The box fetched is 50% larger than the viewport.

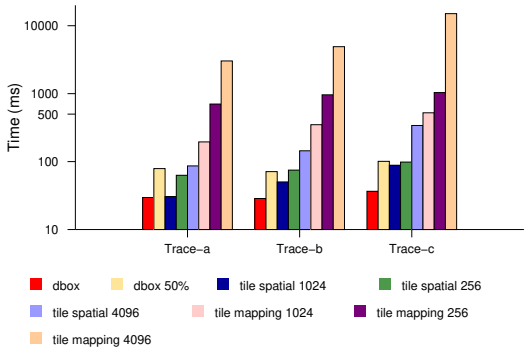


Figure 6: The average response times of dynamic box and static tiling on uniformly distributed data.

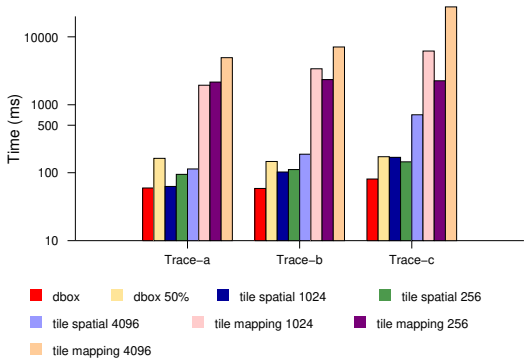


Figure 7: The average response times of dynamic box and static tiling on skewed data.

Tile spatial: Static tiles with spatial index (three tile sizes tested: 256, 1,024 and 4,096).

Tile tuple-tile mapping: Static tiles with tuple-tile mapping (tile size 1,024 tested). Btree index is used on tuple_ID and tile_ID columns.

Results. We measured the average response time (per step) of all fetching schemes on three traces. Average results over three runs are shown in Figures 6 and 7. We have the following observations:

(1) *Dbox* has the best overall performance on both UNIFORM and SKEWED. The reasons are twofold. First, it fetches the least amount of data needed to render the viewport. Second, compared to small tiles, it issues much fewer queries.

(2) *Tile 1,024 spatial* has competitive performance on trace-a, and is even better than *Dbox 50%*. This is because the viewport completely aligns with tile boundaries in trace-a.

(3) *Tile 4,096* and *256 spatial* have the worst performances. This is expected since the tile size 4,096 fetches more data than other fetching schemes and the tile size 256 issues more queries than other fetching schemes.

4. DISCUSSION AND FUTURE WORK

Previous work [2] has studied prefetching data ahead of the user's interaction. Specifically, both momentum-based and semantic-based prefetching were considered in a tiling context. To determine what to prefetch, semantic-based prefetching uses the similarity to recently viewed data in data characteristics (e.g., distribution). Whereas, momentum-based prefetching takes the user's recent movements (e.g., pan and zoom) into account to that end. We plan to evaluate the effectiveness of momentum-based prefetching in the context of dynamic boxes. Our future work will also study caching options for Kyrix. Caching and prefetching are challenging given the jump operation, and will be more challenging by the extension of Kyrix to support coordinated views.

Currently, we are collaborating with a neurology group at Massachusetts General Hospital (MGH), which we anticipate motivating various future extensions of Kyrix. Our collaborators want to be able to interactively explore 50 terabytes of electroencephalogram (EEG) data collected from sleeping subjects. They want three different views of the data, a temporal view, a spectral view and a composite clustering view, to be coordinated. For instance, movement in the temporal view should cause an appropriate change in the spectral view. Hence, Kyrix must be extended to support multiple canvases on the screen simultaneously and to have pan/zoom operations in one canvas cause desired actions in other canvases. In addition, MGH wants an update model for Kyrix so they can edit and tag relevant data. Fifty terabytes will require a parallel multi-node DBMS to achieve our performance goals.

Lastly, we envision Kyrix as an integrated environment for developing scalable visualization applications. To this end, e.g, we plan to work on an "application by example" interface, whereby a user can drag and drop screen objects, and Kyrix can learn to automatically generate the location function (and perhaps other parts of the application).

5. RELATED WORK

Kyrix is related to prior efforts in scalable visualization systems and declarative visualization specification.

5.1 Scalable Visualization Systems

Earlier research has proposed methods for scalable interactive data analysis that fall into one of the two categories in general: precomputation and sampling [10]. Precomputation, which traditionally referred to processing data into formats such as prespecified tiles or cubes, has been the prevalent approach to interactively answer queries via zooming, panning, brushing and linking. Google Maps precompute image tiles for multiple zoom layers to support scalable panning and zooming. Extending the tiling idea to structured data, imMens [14] computes multivariate data tiles in advance along with projections corresponding to materialized database and performs fast "roll ups"and rendering on the GPU. Nanocubes [12] stores and queries multi-dimensional aggregated data at multiple levels of resolution in memory for visualization. Hashedcubes [15] improves on the memory footprint and implementation complexity of Nanocubes with an incurred cost of longer query times. ForeCache [2] uses data tiling together with predictive prefetching and in-memory caching to enable scalable panning and zooming for visualizations of array-based datasets. When precomputation is not possible (e.g., queries are not known in advance),

sampling, often combined with precomputation, and online aggregation [11, 1, 3] are used to improve user experience.

Kyrix precomputes database indexes and uses novel data fetching mechanisms to efficiently respond to pan and zoom interactions. Kyrix's new dynamic-box fetching together with spatial index outperforms tile-based fetching used in earlier systems. To ensure the 500ms response time, Kyrix also adopts predictive prefetching and caching techniques [7, 2].

5.2 Declarative Visualization Specification

Earlier research proposes declarative grammars over data as well as visual encoding and design variables to specify visualizations. In a seminal work, Wilkinson introduces a grammar of graphics [24] and its implementation (VizML), forming the basis of the subsequent research on visualization specification. Drawing from Wilkinson's grammar of graphics, Polaris [22] (commercialized as Tableau) uses a table algebra, which later evolved to VizQL [9], the underlying representation of Tableau visualizations. Wickham introduces ggplot2 [23], a widely-popular package in the R statistical language, based on Wilkinson's grammar. Similarly, Protovis [4], D3 [6], Vega [19], Brunel [25], and Vega-Lite [17] all provide grammars to declaratively specify visualizations.

Kyrix's declarative grammar differs from these earlier efforts by providing constructs for specification of scalable interactive visualizations and integrating visualization specification with a server-side processing and scalable data management for performance optimization.

6. CONCLUSION

The current practice of purpose-built scalable visualization tools is itself not scalable under the fast growth of large datasets across domains. To accelerate the development pace of interactive visualization systems at scale, we need to make it easier for developers to access scalable data management models as well as performance optimizations needed for sustaining interactive rates. In this paper, we present the design of Kyrix, a novel end-to-end system for developers to build interactive, details-on-demand visualizations at scale. Kyrix enables developers to declaratively specify visualizations, while utilizing Kyrix's suite of optimizations and data management model. Kyrix also contributes a novel dynamic fetching scheme that outperforms tile-based fetching common to existing systems.

7. REFERENCES

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[2] L. Battle, R. Chang, and M. Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In *ACM SIGMOD*, pages 1363–1375, 2016.

[3] L. Battle, M. Stonebraker, and R. Chang. Dynamic reduction of query result sets for interactive visualizaton. In *Proc. IEEE Conference on Big Data*, 2013.

[4] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2009.

[5] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011.

[6] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[7] S.-M. Chan, L. 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.

[8] D. Cheng, P. Schretlen, N. Kronenfeld, N. Bozowsky, and W. Wright. Tile based visual analytics for twitter big data exploratory analysis. In *Big Data, 2013 IEEE International Conference on*, pages 2–4. IEEE, 2013.

[9] P. 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.

[10] J. M. Hellerstein. Interactive analytics. In *Readings in Database Systems*. MIT Press, 5th edition, 2015.

[11] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *ACM SIGMOD Record*, volume 26, pages 171–182. ACM, 1997.

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

[13] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics*, 20(12):2122–2131, 2014.

[14] Z. Liu, B. Jiang, and J. Heer. imMens: Real-time visual querying of big data. *Comput. Graphics Forum*, 32:421–430, 2013.

[15] C. A. L. Pahins, S. A. Stephens, C. Scheidegger, and J. L. D. Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *IEEE Transactions on Visualization and Computer Graphics*, pages 671–680, 2017.

[16] P. Pirolli and S. Card. Information foraging. *Psychological review*, 106(4):643, 1999.

[17] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2017.

[18] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics*, 22(1):659–668, 2016.

[19] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2016.

[20] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative interaction design for data visualization. In *ACM User Interface Software & Technology (UIST)*, 2014.

[21] B. 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.

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

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

[24] L. Wilkinson. *The Grammar of Graphics*. Springer, 1st edition, 1999.

[25] G. Wills. Brunel v2.5. https://github.com/Brunel-Visualization/Brunel, 2017. Accessed: 2018-04-04.