Democratizing Details-on-demand Data Visualizations at Scale

by

Wenbo Tao

B.Eng., Tsinghua University (2016)
S.M., Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author . . . .
Department of Electrical Engineering and Computer Science
May 14, 2021

Certified by . . . .
Michael Stonebraker
Adjunct Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Democratizing Details-on-demand Data Visualizations at Scale

by

Wenbo Tao

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Details-on-demand is a powerful interaction paradigm which features the use of simple mouse interactions such as pan and zoom to help the viewer navigate through a large data space. In the past years, we have witnessed an increasing amount of data visualization applications that embrace this paradigm to facilitate data exploration and analysis. Web maps are a clear example. However, due to the highly specialized nature of these applications as well as the lack of general scalable toolkits, building new details-on-demand data visualizations remains hard especially for large datasets. This thesis proposes new tools and systems to "democratize" details-on-demand-based data visualizations, i.e., to make it much easier to build such applications at scale. The main approach is to offer declarative data visualization languages for developers to author applications in small amounts of code, and work with a database backend to transparently handle the rendering and performance optimizations needed to enable fluid interactions on large datasets.

Thesis Supervisor: Michael Stonebraker
Title: Adjunct Professor of Electrical Engineering and Computer Science

Acknowledgments

This section will be a little bit long and narrative. Bear with me - I simply have too many people to thank, and too many thankful words to type. Hopefully the stories will be interesting to you.

To my advisor Mike Stonebraker

"*I'm a big fan of details on demand*". This is what Mike said to me when pitching the idea of Kyrix. We then started the incredible journey of building our data visualization systems. Mike actually repeated this sentence many times when introducing our systems to other people. Of course, he is not saying that we should be a fan of details on demand because he is. There is an underlying reason to his fandom of the details-on-demand paradigm. Quoting Mike again: "*The user does not need a manual to start using a details-on-demand interface. It's very simple.*"

This actually exemplifies one big thing that I'm grateful for being around Mike. He is essentially an idea filter that highly discourages ideas that are too complex to be expressed in one or two sentences. This had pushed me to organize my ideas in a way that can be succinctly summarized. If I could not, it is probably too complex to make the idea work. Of course, the real world is powered by many complex ideas that probably would not pass Mike's filter. Yet his strong inclination towards simple ideas taught me to think about my ideas critically, to try to express ideas in a way that a layman can understand as much as possible. "*Simple is good, complex I don't trust.*" This is another quote from Mike that firmly stuck in my head.

Looking back, I could not thank Mike enough for teaching me the importance of keeping users in the loop when building a system. When we first had a working prototype of Kyrix, we were introduced a collaborator from MGH who had a use case for our system. I was reluctant to collaborate because their use case seemed to deviate a lot from what we initially intended to support. Yet Mike was very decisive and suggested that we do all we could to support their use case. As a result, we added three important features that had benefited lots of later applications. Meeting with users to polish our systems was a norm in building the systems. I learned a great deal by just observing how Mike talked to the users to learn about their requirements deep down. It was Mike's strong focus of practicality that constantly reminded me of the importance of engaging with users in all stages of system building.

Mike had also helped me in many other ways which I am very grateful for. I remember that Mike invited me to his home to help revise my first paper in my first year. It was only a couple days after his lower body surgery! Mike was well known for disliking revising papers for students and often commented that "every graduate student in the building is a bad writer." So you could easily understand my ecstasy when Mike commented on my last paper that "It reads pretty well. Did you suddenly become a good writer?"

No, I did not. Without Mike's help, I would not be who I am today. Thank you Mike.

To my advisor Remco Chang

As my co-advisor, Remco had provided me with immense amount of help and support. Countless times when I was stuck, Remco would invite me to his lab at Tufts to talk it through. Every time I went to Tufts, we would talk for hours about not only technical stuff, but also life, sports and how Kyrix should not be named after Kyrie Irving. Those conversations at Tufts were one of my most cherished moments during my PhD. They got me unstuck. They showed me things from different perspectives. They taught me professional and people skills. Whenever I am back in Boston in the future, Remco's office will for sure be on the top of my list of places to visit (even above TD garden).

In retrospect, what Remco brought to our projects was indeed invaluable. It was easy for us database people to get too obsessed with database optimizations. Whenever that happened, we would hear the sane voice from Remco which reminded us that we were building a visualization system. Remco would offer useful insights from the visualization standpoint but also with insights into the underlying optimizations because of the cross-disciplinary expertise he had. Remco was very humble and open-

minded, which made our discussions easy and productive. I want to also thank Remco for his coming to MIT for our weekly meeting. To a busy professor that was a huge sacrifice.

Remco is a unique professor. The first thing on his homepage is not the list of projects, the list of publications nor a short bio. It is a list of his students who have become professors. He sees mentoring of students as his first priority. That is probably why I have learned so much from him. Although I may not be added to that list, I will make sure to become a useful person to the society whom Remco can be proud of. Thank you Remco.

To the genius hacker Adam Sah

Adam was a former student of Mike who joined our project halfway and since made our systems orders of magnitudes better. Given that he was a pretty senior type of person (a successful serial entrepreneur), it was a huge surprise to me when he actually started contributing code to Kyrix. As he said, he is an old Berkeley hacker who loves to code. In Mike's terms, Adam is a "super programmer".

The series of improvements he made to Kyrix had life-changing influences on me. I had zero industrial experiences back then. Adam's code showed me what I never thought was possible. To scale Kyrix beyond single-node PostgreSQL, he orchestrated six systems together (PostgreSQL, Citus, PLV8, Kubernetes, Docker and GCP), two of which he had to learn from scratch. I was not only amazed by what his code can do, but also his "hacker" mentality, i.e., to be super enthusiastic about making things work, usually through lots of trial and error. Adam seemed to have awaken that programming nerd inside me who had since become fascinated by systems engineering. I had determined to start "hacking" like him, no matter what I do in the future.

I am also grateful for just having the opportunity to witness how Adam communicated his work to other people. The way he made proposals and reported experiment results was eye opening. As he often joked about, he was a 20+th year PhD student of Mike. So while I was the single PhD student in the Kyrix project, Adam served as that senior "student" who I could learn a lot from. Thank you Adam.

To everyone on the Kyrix team

The Kyrix team has been a big one. In addition to Mike and Remco, Leilani Battle was another professor involved. She contributed a lot of useful ideas and feedback on my work. Thank you Leilani. Near my graduation, Leilani introduced Ameya Patil, who became the second PhD student (in addition to me) in the Kyrix family. So I will pass the torch to Ameya, who plans to augment Kyrix in exciting ways, such as supporting graph networks.

The people I spent the most time with are two visiting students Xiaoyu Liu and Xinli Hou. Between us, tons of meaningful conversations and pair programming happened that led Kyrix to a better place. Thank you Xiaoyu and Xinli. You were the best company on my PhD journey.

I want to thank especially Peter Griggs (a master student) for his work on Kyrix. Peter's story was a little twisted because he spent quite some time investigating different indexing methods which did not end up in his thesis. But frankly, those early work was important and informative. I especially respected Peter's ambitious attempt in writing a PostgreSQL extension to support faster Kyrix indexing. Although it did not end up being fruitful, it showed Peter's hacker mentality since writing a PostgreSQL extension is non-trivial.

Four other master students also did amazing work that I learned a lot from. Jim Peraino produced an amazing application which showcased how Kyrix can be extended to 3D. Abhishek Bassan designed an elegant caching algorithm for the Kyrix backend. Erica Zhou studied how Kyrix can be applied to trace the transmission paths of infectious diseases. Amy Zhang built a neat front-end authoring tool which made the authoring of Kyrix easier for non-experts.

Lastly, I want to acknowledge all other people who had contributed to Kyrix through either PDF or code: Yedi Wang, Maxime Schoemans, Giovanni Simonini, Elkindi Rezig, Lei Cao, and Ishan Sen. The names listed here are by no means complete. Thank you everyone who was a member of the Kyrix family.

To my other committee member Arvind Satya

I am very grateful to Arvind for agreeing to be on my committee and providing valuable feedback on my thesis and defense talk. We only had a few meetings but every meeting was informative. The design of the systems in this thesis was also inspired by many Arvind's works on visualization authoring. I look forward to following more works of his in the future. Thank you Arvind.

To my friends

I enjoyed being with folks in the data systems group (formerly the database group). They are super smart people who can also be very good friends with me. I enjoyed the lunch chatting, trips, board games, billiard and ping pong. Spending time with them chatting about life and tech was a good break from my research. Thank you guys!

There are fellow students at MIT whom I often hung out with: Favyen Bastani, Jie Xu, Dong Deng, Songtao He, Shichao Yue, Oscar Moll, Yu Xia, Joana M. F. da Trindade, Yi Lu, Zeyuan Shang, Lei Cao, Yuanming Hu, Hao He, Kapil Vaidya, Hongzi Mao, Matt Perron, Xiangyao Yu and many others. I enjoyed every meal we had, every movie we watched, every time we played basketball and every Celtics game we went to together. Thanks for the company and best of luck!

To my family

Lastly, I want to thank my family especially my parents and my twin brother. While I had been studying abroad, they have provided unconditional support. COVID has separated us for so long. Hopefully we can reunite soon once the world reopens.

There is an interesting story about the names of mine and my brother's. The first name of my brother means the master degree in Chinese. And my first name means the doctoral degree. My brother got his master degree a while ago and went into industry. Now I will get my doctoral degree. We should thank our grandfather, the giver of our names, who accurately predicted (or destined?) our academic paths.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Details-on-demand (DOD) is a powerful interaction paradigm which enables the user to locate data of interests in a large data space using intuitive interactions such as pan and zoom. In the past three decades, we have seen increasingly wide application of this paradigm in a variety of domains. This popularity dates back to the advent of Zoomable User Interface (ZUI) toolkits [103, 22] in the 90s, which has prompted the use of DOD in a diverse range of user-facing applications [21, 43, 49, 124]. Shneiderman in 1996 proposed the information seeking mantra "Overview first, zoom and filter, then details on demand" in his seminal work [120], which has inspired a series of subsequent research efforts on building systems and applications that embody the spirit of DoD. As smartphones and tablets become ubiquitous in the mid 2000s, the popularity of web maps (e.g. Google Maps), an exemplary class of applications that embrace the DOD paradigm, has skyrocketed. This widespread adoption is also due to the intuitive nature of the interactions: without reading a user manual, the user can start using the application via pan and zoom actions to get dynamically updated map content. Web maps have also become a platform where sophisticated data exploration and analysis is performed. Data objects are usually arranged at different resolutions [62, 45, 32, 90], allowing the user to both capture overviews and inspect details. Furthermore, the DOD paradigm is also frequently adopted to visualize graph networks [69], analyze genomic data [78], support workspace awareness [64] and so on.

As datasets become larger and more complex, DOD data visualizations will continue being an indispensable technique for people to make sense of their data. In this thesis, we study how to make creating a DoD data visualization easy at scale.

1.2 Research Challenges

Despite the usefulness of the DOD paradigm, developing a DOD-based data visualization application still remains surprisingly hard. While many tools exist to support the developer in designing DOD data visualizations, there are a number of challenges that these systems do not address well, and thus prevent the developer from efficiently creating an application at scale.

Scaling to large datasets. First and foremost, a DoD system should enable fluid interactions on large datasets, which requires the system's response times to user interactions to be bound within 500 ms, an empirical upper bound used by popular websites [11]. However, existing tools for developing DOD-based data visualizations often lack the backend support to handle large datasets. More often than not, tools and systems assume that data fits in the memory of one computer [22, 23, 106, 95, 62, 90, 32, 45], and fetch everything into the memory before computations can begin. This assumption does not hold any more because the datasets nowadays typically have large data tables with millions to billions of data items that are too large to fit in memory [87, 31, 84]. As a result, these systems cannot respond to user interactions within 500 ms on large datasets that need disk storage and thus fail to sustain an interactive user experience.

Handling multiple tables. The requirements for "being scalable" entail more than handling a large data table. Nowadays, data are often stored in a database management system in the form of many interconnected data tables. The relationships between tables are often very complex and difficult to understand especially for new users [138]. DoD is especially helpful in helping the user navigate between related data tables. For example, in a relational database, there are lots of data columns from multiple tables that represent the same real-world entity, which are typically referred to as primary keys and foreign keys [143]. Consider one DEPARTMENT table with `department_id` being the primary key and one STUDENT table with `department_id` being a foreign key. We can connect a scatterplot visualization of departments with a bar chart showing the average GPA of students by allowing the user to click on one department in the scatterplot, and "jump" to the bar chart to see the average GPA of students in that particular department. In other words, these relationships between tables/visualizations can be utilized to see different facets of data visualizations [50, 145] and perform drill down analysis [52, 18].

While several tools have been developed to capture such relationships between visualizations [122, 145, 50, 52, 18] using DoD, they all assume that data is a single data table and focuses on "pivoting" between visualizations of that table. Unfortunately this single-table assumption does not hold in practice. Further, it is hard to extend those systems to a multi-table setting because they often lack the multi-table semantics needed to support effective data explorations among multiple tables.

Supporting rapid authoring on general data. To make it more accessible to create DoD data visualizations, we need systems and tools that offer easy-to-use developer interfaces. These interfaces should provide primitives that are 1) simple, which enables quick prototyping and 2) expressive, which means being able to support a variety of different types of DoD visualizations and being agnostic to application domains and data types.

Status quo. To our best knowledge, while point solutions exist that address some of the challenges, we are not aware of a prior system that addresses all three challenges. For example, as a result of the limited scalability of existing tools, many DOD data visualization applications have been purpose-built for specific domains and data types (e.g. geographical analysis [62], satellite imagery [19], genomics data [78]) to support large-scale data. These applications are typically hardcoded for the underlying domain and cannot be easily extended to support more general usage scenarios.

1.3 Solution

The goal of this thesis is to design and build systems to address the aforementioned challenges and limitations of existing DOD data visualization systems, and ultimately take one concrete step towards democratizing DOD data visualizations at scale. Specifically, this thesis contributes three end-to-end systems Kyrix, Kyrix-S and Kyrix-J which respectively focus on large-scale DOD data visualizations of various kinds. These three systems are built on top of one another and all adopt the same high-level approach: offering declarative visualization languages for rapid authoring and transparently handling the underlying rendering and performance optimizations needed for scalability.

Kyrix is a low-level system which serves as an expressive authoring engine for general DOD data visualizations and also the foundation on which both Kyrix-S and Kyrix-J are built. Kyrix contributes a declarative language that works with arbitrary data types (by assuming that the data comes out of a generic database) and expresses a wide range of DOD visualizations using simple concepts such as canvases (zoom levels) and jumps (transitions between zoom levels). Making use of database spatial indexes, Kyrix dynamically fetches the data in the user's viewport on demand to achieve interactive response times on large disk-based datasets.

Kyrix-S focuses on a very common type of DOD data visualizations which we call *scalable scatterplot visualizations* (*SSVs*). An *SSV* expands a static scatterplot onto multiple zoom levels. With more screen resolution available, the overdraw issue of static scatterplots can be effectively mitigated [90]. Data objects can be represented with a dot, polygon or aggregation-based marks such as pie charts and bar charts. Kyrix-S offers a declarative language for *SSVs*, which allows the developer to describe a complex *SSV* in 10s of lines of code. Behind the scenes, Kyrix-S works with a distributed database to calculate the layout of objects across all zoom levels in a highly paralleled fashion, and uses parallel database spatial indexing to achieve interactive browsing of *SSVs* with billions of objects.

Kyrix-J uses DOD to enable visual data exploration of a relational database that has large amounts of relationships between data tables and visualizations. Given pairs of columns that represent the same information (e.g. primary-key and foreign-key relationships), Kyrix-J automatically adds DOD-based interactions between visualizations to facilitate faceted browsing of data tables and drill down analyses. To help the user stay oriented during their exploration of these relationships between visualizations, Kyrix-J implements a series of visual aids using coordinated multiple views, animated transitions, text search, bookmarking and so on.

Extensive experiments are conducted to study both the usability and performance of these three systems. All three systems are open-sourced[1] which will enable evaluation of the systems in a broader setting.

[1]Kyrix and Kyrix-S are open-sourced at `https://github.com/tracyhenry/kyrix`. Kyrix-J is open-sourced at `https://github.com/tracyhenry/kyrix-j`

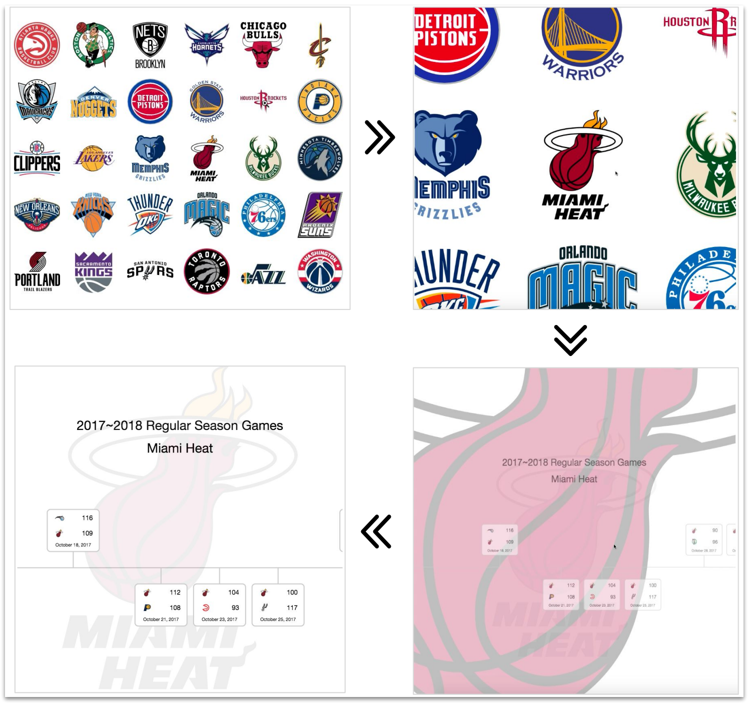
Chapter 2

Kyrix: General Details-on-demand Data Visualizations at Scale

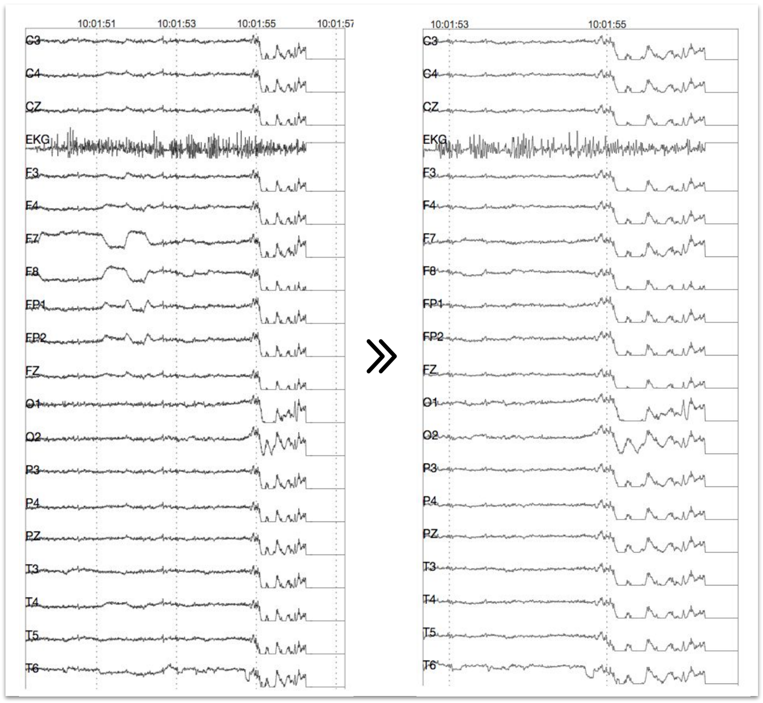
2.1 Introduction

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 [140], leading to error-prone and prolonged analysis processes.

As discussed in Chapter 1, DoD data visualizations [21, 43, 49, 61] 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 [121]. Figure 2-1 shows two example DoD data visualizations created using the system we introduce in this chapter. Figure 2-1b shows an EEG diagram of one patient in a large US hospital. 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 2-1a, a basketball fan can click on the logo of his favorite NBA team in the first view,



(a)



(b)

Figure 2-1: 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.

then *jump* to a timeline showing the team's basketball games.

The usefulness of DoD data visualizations has led to the development of a number of zoomable UI (ZUI) toolkits, e.g., Pad++ [22] and ZVTM [106], to support zooming-based DoD applications. However, while these toolkits support the developer in designing DoD data 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 [87, 31, 84]. Therefore, as data gets large, DoD visualizations developed using existing ZUI toolkits can fail to bound interaction response times within 500 ms, which is required for sustaining an interactive user experience [11]. 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 [28, 111]. As a result of these inadequacies of existing ZUI tools, many purpose-built systems (e.g. Google Maps [61] and ForeCache [19]) 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 DoD 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 chapter, we present Kyrix[1], an integrated system for developing scalable visualizations driven DoD interactions such as pan and zoom. Our goal is to achieve *generality* (support for general data types and visualizations), *ease of development* and *scalability*. Figure 2-2 shows the system architecture. On the developer side, we offer a concise yet expressive declarative model for easy specification of general DoD visualizations. Declarative designs hide execution details (e.g. backend optimization

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

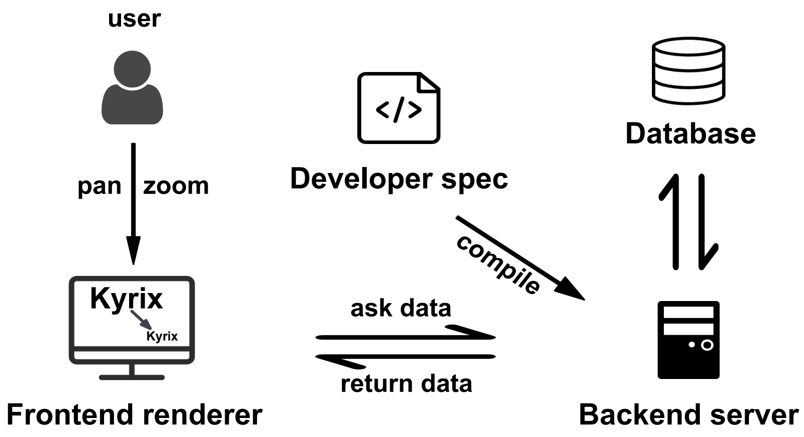


Figure 2-2: Kyrix system architecture.

and frontend rendering) from the developer, so that they can focus on visual specification [115]. 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 general DoD visualizations at scale.
- To our best knowledge, the first declarative model for authoring general DoD visualizations of large, disk-based data (Section 2.4).
- A suite of performance optimizations that integrate with the underlying data management system to guarantee interactivity on large datasets (Section 2.5).

We evaluate the expressivity of Kyrix's model through building several example visualizations (Section 2.6). 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 2.7). 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 2.8). We find that Kyrix can support interactive exploration over 100 million data points with an average latency of 100 ms or below.

2.2 Related Works

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

2.2.1 ZUI Toolkits and Systems

Perlin and Fox introduces the Pad system [103], 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, a prominent example class of DoD interfaces. Examples include Pad++ [22], Jazz [23] and ZVTM [106]. These tools provide application programmers with low-level graphics primitives and have enabled the creation of numerous ZUIs in various domains [21, 49, 43, 61, 116, 127, 108].

Nevertheless, the aforementioned tools cannot scale to large datasets due to two common limitations. First and foremost, they assume that data can fit in main memory – an assumption that does not hold for large datasets that require disk-based data storage [87]. 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 [28], this cumbersome process prevents the developer from reasoning on the data level, and is ill-suited for creating large-scale 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.

2.2.2 Performance Optimization in Visualization Systems

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

Image tile browsers such as Deepzoom [91], Google Maps [61] and Zoomify [147] 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 2-9b) to show that Kyrix can also be used as an image tile browser.

ATLAS [31] adopts predicative caching to enable interactive pan and zoom on time series data. In a similar vein, ForeCache [19] prefetches data tiles to efficiently render dense array-based data such as satellite imagery data. Aperture Tiles [37] precomputes and fetches image tiles from distributed storage systems with a focus on geospatial applications. HiGlass [78] 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 database spatial indexing and extends some of the optimization techniques in these systems (e.g. prefetching and caching) to optimize general DoD interactions.

Prior works have studied in-memory techniques to fetch only needed data in response to user actions. The Splash framework [59] 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++ [22] and ZVTM [106] which provide shape-level bounding box specifications.

A long line of research also studies how to reduce visual clutter [53] on large data visualizations using techniques such as sampling [47, 48, 107] and binned aggregation [55, 60]. This type of data manipulation is often performed before data visualization [60, 59], so we assume this is an orthogonal process that is done either outside Kyrix or through a custom preprocessing procedure (Section 2.4.4).

Multidimensional data tiles/cubes [87, 84, 99, 30] have been widely adopted to support interactive aggregation queries. However, due to huge amounts of memory used, the index structures proposed cannot support complex DoD 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.

2.2.3 Declarative Visualization Specification

Kyrix's declarative model is related to earlier research on declarative visual analysis grammars. Wilkinson introduces a grammar of graphics [136] and its implementation (VizML), forming the basis of the subsequent research on visualization specification. Drawing from Wilkinson's grammar of graphics, Polaris [125] (commercialized as Tableau) uses a table algebra, which has later evolved to VizQL [68], the underlying representation of Tableau visualizations. Wickham introduces ggplot2 [135], a widely-adopted package in the R statistical language, based on Wilkinson's grammar. Similarly, Protovis [27], D3 [28], Vega [114], Brunel [137], and Vega-Lite [113] all provide grammars to declaratively specify visualizations. Some of these declarative languages (e.g. D3 [28] and Vega [114]) are capable of expressing DoD 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 [28] or event-driven functional reactive programming code [114]. Vega-lite [113] offers much simpler primitives to specify DoD interactions pan and zoom, but it is a high-level language not designed for customizability. Kyrix enables declarative specification of DoD 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 DoD 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 [135, 113], in Kyrix a layer is also associated with a bounding box function to enable fast data fetching on the server side.

2.3 Design Requirements

We first present a set of requirements we identify before and during the development of Kyrix, 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.

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

R2. Ease of development. From the developer's standpoint, Kyrix 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 [22, 106], which are laborious and error-prone especially for large data. Data-driven abstractions make it more accessible to author data-dependent visual properties [28].
- **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 empirically suggested by popular websites [11], we should bound response times to user operations within 500 ms, a threshold for enabling fluid interactions.

2.4 Declarative Model

Our declarative model contributes an easy mechanism to specify general and scalable DoD data 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 2-1a) as a running example and show in Figures 2-4$\sim$2-7 relevant specification snippets. In our current implementation, the developer specifies visualizations using Javascript.

2.4.1 Overview

Figure 2-3 is an illustration of the declarative concepts and their relationships. Considering the fact that a DoD data visualization typically comprises multiple levels of details [55, 59], we naturally use a *canvas* to model one level of details. We model the relationships between two canvases using two types of connections *zoom* and *jump*. Canvases and their connections form a connected directed graph if we consider canvases as nodes and connections as edges.

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* which informs the backend of the locations of the visual objects on the canvas for fast data fetching.

The four views in Figure 2-1a show the progression of a *jump* from an initial canvas (showing NBA team logos) to the second canvas (showing a timeline). Note

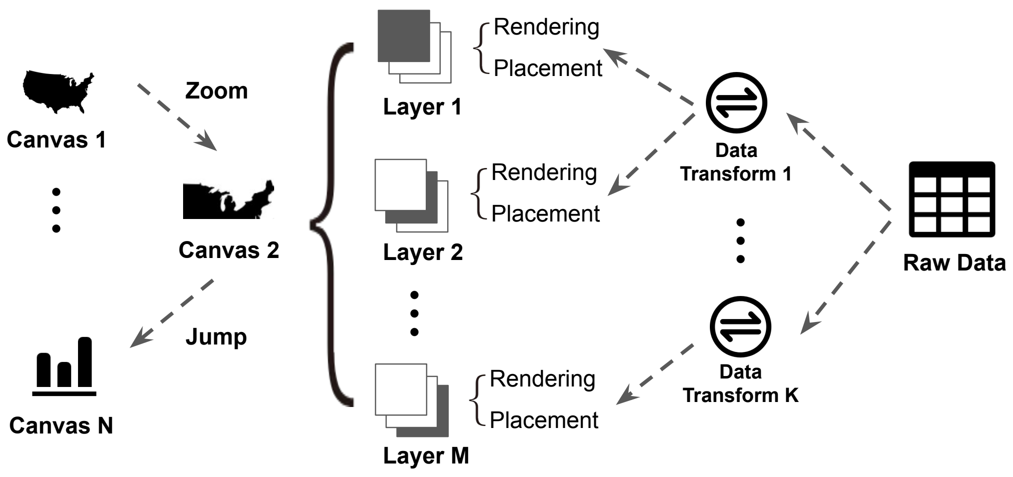


Figure 2-3: Kyrix's declarative model. *Canvas*es are "levels of details" connected by *zoom*s or *jump*s. A canvas has multiple *layer*s. A *data transform* prepares the data source for rendering a layer. A *rendering function* maps data to visual objects. A *placement function* provides spatial information of objects to enable fast data fetching.

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. Figure 2-1b shows a *zoom* from coarser-grained time series to finer-grained time series. We detail the difference between a zoom and a *jump* later in Section 2.4.3.

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 2.3.

2.4.2 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 [113, 135]. The primary goal is to enable multiple different visual encodings on a single view [135]. Nonetheless, our layer concept has its own unique definition in a large-scale DoD visualization setting.

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

// ================== logo canvas ==================
var width = 1000;
var height = 1000;

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

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

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

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

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

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

Figure 2-4: Specifications of *canvas*es and *layer*s for the NBA example in Figure 2-1a.

First, we want to enable a mixed visual representation by allowing a layer to be either *dynamic* or *static*. Dynamic layers move and 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 DoD interactions such as pan and zoom (**R2-c**). We describe this concept in more detail in Section 2.4.6.

2.4.3 Zoom and Jump

The major distinction between a *zoom* connection and a *jump* connection lies in whether the two canvases share the same coordinate system. Two canvases connected by a zoom typically share the same type of spatial dimensions, and sometimes the same type of visual representations. For example, in the zoom represented by Figure 2-1b, two corresponding canvases both have time and amplitudes as the two axes and both render the data points as EEG time series. As another example, canvases in Google Maps all have longitude and latitude as the two spatial dimensions. But their visual representations can vary: top levels can show continents and oceans while bottom levels visualize cities and rivers. On the other hand, two canvases connected by a jump can have disparate spatial dimensions and visual representations. The jump often serves as a smooth transition from one visualization to another vastly different one. Figure 2-1a is a clear example.

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.

A jump connection is usually triggered by selecting an object, and can be customized with several lightweight data-driven abstractions (**R1, R2-a, R2-b**).

Selector enables custom selection of visual objects that can trigger a jump. In the NBA example, every logo can trigger a jump to the timeline view (line 1-3, Figure 2-5). A more interesting scenario would be that only playoff teams can trigger a jump 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 jump.

Viewport customizes the viewport location after the jump. This is a function that takes the data item bound to the clicked object (the visual object that trigger the jump), and returns the coordinates of the new viewport. In the NBA example, this function returns a constant viewport location (line 4, Figure 2-5) indicating that the

```
    // ======== 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.addJump(logoCanvas,
15            timelineCanvas,
16            selector,
17            viewport,
18            predicate);
```

Figure 2-5: Specification of the *jump* from the logo canvas to the timeline canvas for the NBA example in Figure 2-1a.

user will see the start (the leftmost part) of the timeline after the jump.

Predicate is a data-driven function used to select a subset of data to render on the destination canvas. For example, in lines 7-13 in Figure 2-5, the predicate function specifies that only games of the clicked team are displayed. In a sense, this enables “faceting” the destination canvas, i.e., to create a series of views sharing a common data schema, without creating a canvas for each view.

2.4.4 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 Chapter 2.[2] 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

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

```
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 2-6: Specification of the *data transform* for a layer in the NBA example in Figure 2-1a.

```
var timelinePlacement = {
    centroid_x : "column:x",
    centroid_y : "column:y",
    width      : "constant:160",
    height     : "constant:130"
};
```

Figure 2-7: Specification of the *placement function* for a dynamic layer in the NBA example in Figure 2-1a.

adding canvas coordinates, scaling and sorting data.

Figure 2-6 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).

2.4.5 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 jumps. In the current implementation, we allow Javascript-based renderers (e.g. D3 [28]).

2.4.6 Placement Function

The key to high performance is to fetch only needed data when the viewport changes. Prior systems [22, 59] 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 2-7.

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 2.5.

2.4.7 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 [28] to implement interaction listeners and zoom/jump animations.

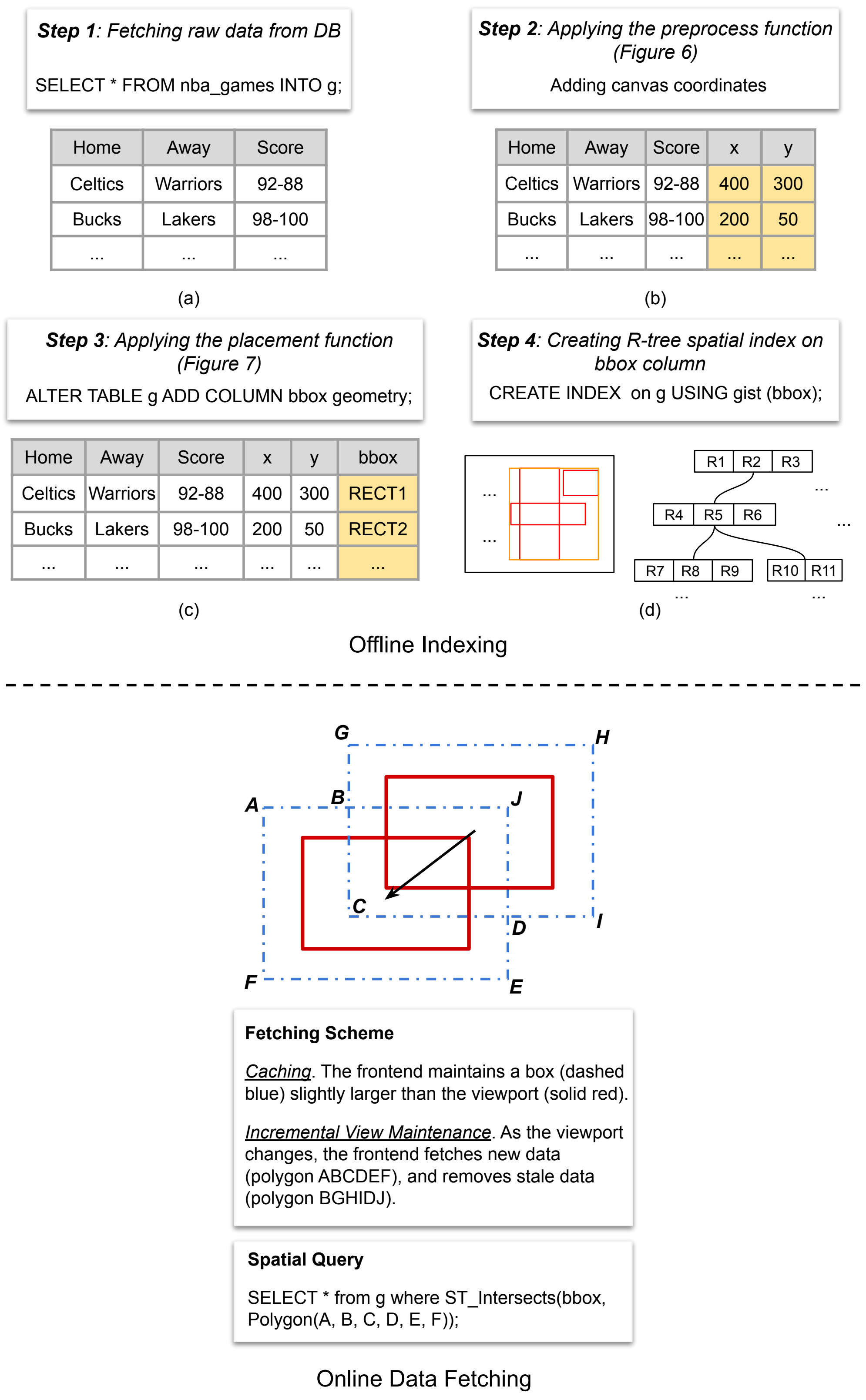
2.5 Performance Optimizations

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 DoD interactions. A natural idea we adopt is to build spatial indexes (e.g. R-trees [63]) for visual objects and only fetch those whose bounding boxes (specified by placement functions) intersect the viewport. The idea of using spatial indexes has also been adopted in prior systems [22, 106]. However, those systems assume that the spatial indexes can fit in memory while spatial indexes typically consume space that is linear in the data size. Therefore, they 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 2.5.1 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 between the frontend and the backend while ensuring interactivity. We describe these techniques in Section 2.5.2.


***Step 1**: Fetching raw data from DB*

```
SELECT * FROM nba_games INTO g;
```

Home	Away	Score
Celtics	Warriors	92-88
Bucks	Lakers	98-100
...	...	...

(a)

***Step 2**: Applying the preprocess function (Figure 6)*

Adding canvas coordinates

Home	Away	Score	x	y
Celtics	Warriors	92-88	400	300
Bucks	Lakers	98-100	200	50
...	...	...	...	...

(b)

***Step 3**: Applying the placement function (Figure 7)*

```
ALTER TABLE g ADD COLUMN bbox geometry;
```

Home	Away	Score	x	y	bbox
Celtics	Warriors	92-88	400	300	RECT1
Bucks	Lakers	98-100	200	50	RECT2
...	...	...	...	...	...

(c)

***Step 4**: Creating R-tree spatial index on bbox column*

```
CREATE INDEX  on g USING gist (bbox);
```

(d)

Offline Indexing

Fetching Scheme

Caching. The frontend maintains a box (dashed blue) slightly larger than the viewport (solid red).

Incremental View Maintenance. As the viewport changes, the frontend fetches new data (polygon ABCDEF), and removes stale data (polygon BGHIDJ).

Spatial Query

```
SELECT * from g where ST_Intersects(bbox,
Polygon(A, B, C, D, E, F));
```

Online Data Fetching

Figure 2-8: An illustration of performance optimizations in Kyrix.

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 attributes produced by the *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 2-8a).

- **Step 2**: for each record in the query result, apply the preprocess function defined in the *data transform.* In Figure 2-8b, 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 2-8c). Modern databases generally have built-in geometry types for representing spatial objects.

- **Step 4**: create an R-tree spatial index [63] 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 issues a spatial query that returns all records whose bounding boxes intersect the viewport. As shown in the bottom in Figure 2-8, 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.

2.5.2 Caching and Incremental View Maintenance

Fetching data in exactly the viewport is problematic because every time the user pans, zooms or jumps, 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 bottom part of Figure 2-8 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 2-8). Upon receiving new data, the frontend first renders new data (polygon ABCDEF) and then removes stale data (polygon BGHIDJ).

2.5.3 Comparison with Existing Optimization Frameworks.

Many purpose-built systems (e.g. ForeCache [19], Aperture Tiles [37] and HiGlass [78]) 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 interactions with objects (e.g. clicking on an object to start a jump) 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 interactions with objects 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.

2.6 Example Visualizations

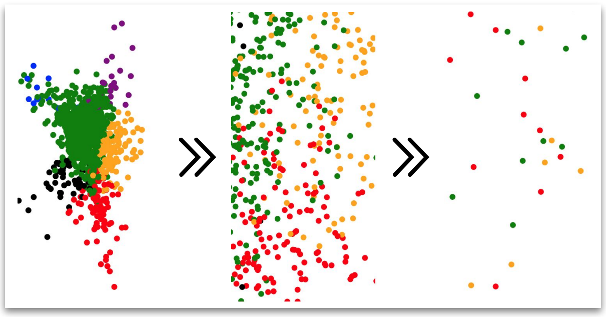
We demonstrate the expressivity of Kyrix's declarative model through a gallery of example DoD data visualizations (Figures 2-1 and 2-9). In the following, we first describe details of these example applications (Section 2.6.1). We then use these examples to describe an expressive design space enabled by our model (Section 2.6.2).

2.6.1 Using Kyrix's Declarative Model to Create Example Visualizations

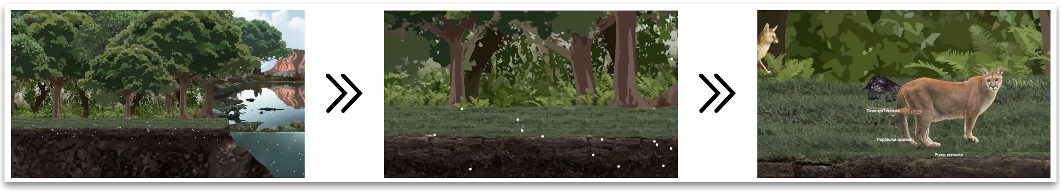
NBA. Figure 2-1a shows two canvases of a basketball data visualization. Descriptions of this example can be found in Section 2.4.1.

EEG. The visualization in Figure 2-1b 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.

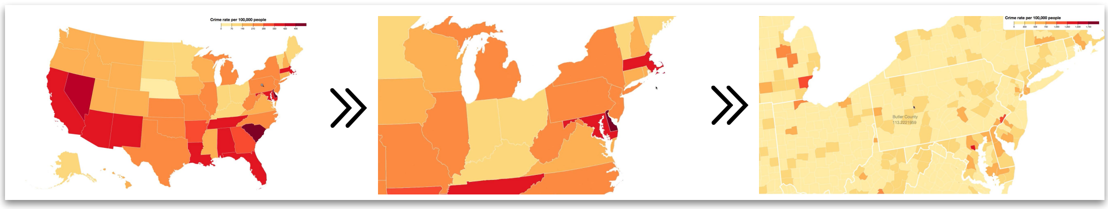
Cluster. Figure 2-9a shows a zoomable multi-class scatterplot of 17 million 2-second EEG segments from over 2,000 patients. This data comes from the same US hospital we mention in **EEG**. Doctors use a t-SNE projection [88] to map 2-second EEG segments into a 2D space to identify potential clusters and outliers. Different colors represent different EEG patterns (e.g. Seizure). There are 7 canvases (zoom levels)



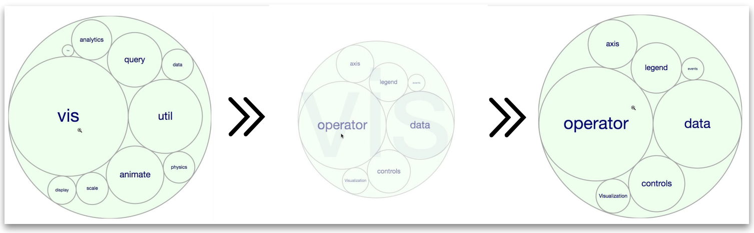
(a)



(b)



(c)



(d)

Figure 2-9: 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 [130].

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 2-9b 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 in the bottom canvas are higher-resolution versions of those in the top canvas.

USMap. The visualization in Figure 2-9c 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[3] 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 2-9d visualizes a tree hierarchy, where the classes in the Flare visualization library [130] are arranged in a circle packing layout. The user can click on a class to jump to another view showing its direct child classes. This visualization is composed of only one canvas, so the jump is a self-loop of this canvas.

2.6.2 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 2.4.5). 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

[3]Kyrix allows zooming via clicking on an object, in addition to via spinning a mouse wheel and using zoom buttons.

general relational data (*NBA*).

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

Jump selector. The selector function decides which visual objects on the canvas can trigger a jump. Some example visualizations (*NBA* and *Flare*) utilize this function. For instance, in *Flare*, we use the selector function to ensure that only visual objects representing child classes can be clicked on for a jump.

New viewport location. Recall that the viewport function is used to specify the viewport location after a jump. Besides constant coordinates, we allow this function to return a viewport location using the data item bound to the clicked object. This data-driven viewport location adds more expressivity to our model. For example, one can click on a state in a scatterplot of states in the US, and then jump to the bottom canvas in *USMap* with the viewport centered at the clicked state. This can be 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 clicked team. Similarly in *Flare*, the predicate function is used to select child classes of the clicked class.

2.7 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.[4] 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

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

D3.js and Tableau. The remaining four (P5-P8) had little or no experience with visualization programming.

2.7.1 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 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 2-10) 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.

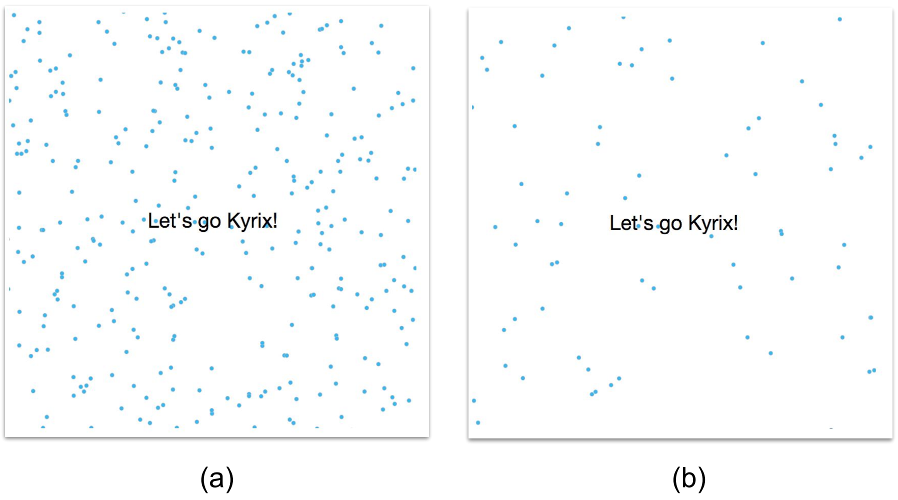


Figure 2-10: The scatterplot visualization used both in Task 1 of the developer study and in the performance evaluation: (a) the top-level canvas; (b) the bottom-level canvas. Two canvases are connected by a zoom. The zoom factor is 2.

Task 2. Task 2 required participants to complete a partial specification of the NBA example in Figure 2-1a, which has two canvases (logo and timeline) and a jump 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 jump between the two canvases.

2.7.2 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)*."*

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 clicked 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 Javascript APIs used in the study were 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 confusion when participants were completing Task 2c. We have addressed 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 2.9.

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.

2.8 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 user 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.

2.8.1 Scalability

To test the scalability of Kyrix, we used the scatterplot in Figure 2-10 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 2-11.

As can be seen, the latency remained stably under 50 ms as the data size grew. This scalability came from using database spatial indexes 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.

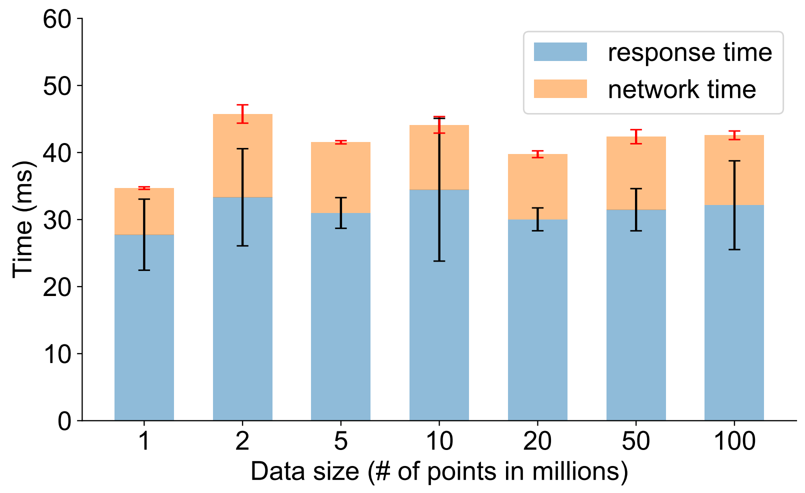


Figure 2-11: Kyrix's ability to scale with increasing data size. The scatterplot in Figure 2-10 is used as the benchmark visualization. Average response time and network latency are shown.

2.8.2 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.

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 2-9a, 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 500 ms [11], it has the potential to support real-time visualizations. *Cluster*'s total response time of 22.1 ms equates to rendering at 45 frames-per-second (fps), surpassing the 30 fps typically required for commercial 3D games [39, 38]. *EEG*'s response time is 100.3 ms, or 10 fps, which is close to the rate where humans perceive animation (instead of individual frames) [26].

2.8.3 Effects of Caching and Incremental View Maintenance

In this experiment, we evaluated the effects of caching and incremental view maintenance techniques described in Section 2.5.2.

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.

2.9 Discussions

2.9.1 Limitations and Future Work

While Kyrix eases the creation of scalable DoD 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 [20, 55] 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 [16]). However, recall that in Kyrix, spatial indexes are built on auxiliary tables, which are computed from raw data (Section 2.5). 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 2.7, 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.

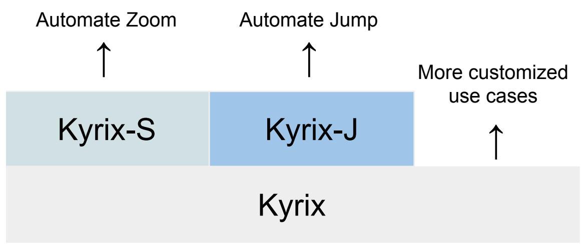


Figure 2-12: Kyrix is a foundational DoD data visualization system on which more higher-level authoring systems can be built.

2.9.2 Using Kyrix as a Foundation System

The general purpose nature of Kyrix makes it a perfect low-level "foundational" platform on top of which more extensions can be built to expose higher-level authoring primitives and further ease the creation of certain types of DoD visualizations. The two extension systems Kyrix-S and Kyrix-J are exactly built based on this idea, and further automate a large class of zoom-based and jump-based applications respectively (Figure 2-12).

As also shown in Figure 2-12, higher level extensions are focused on applications with specific traits and do not attempt to replace Kyrix completely. For use cases that require lots of customizations, Kyrix will be a choice that offers flexibility.

2.10 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 chapter, we present the design of Kyrix, a novel integrated system for the developer to build interactive DoD 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. Thanks to its general purpose nature, Kyrix can serve as the foundation for high-level systems which we will introduce in the following chapters.

Chapter 3

Kyrix-S: Authoring Scalable Scatterplot Visualizations of Big Data

3.1 Introduction

Scatterplots are an important type of visualization used extensively in data science and visual analytic systems. Objects in a dataset are visualized on a 2D Cartesian plane, with the dimensions being two quantitative attributes from the objects. Each object can be represented as a point, polygon or other mark. Aggregation-based marks (e.g. pie chart, heatmap) can also be used to represent groups of objects. The user of a scatterplot can perform a variety of tasks to provide insights into the underlying data, such as discovering global trends, inspecting individual objects or characterizing distributions [110].

Despite the usefulness of static scatterplots, they suffer from significant overdraw problem on big skewed datasets [104, 90]. Here, we focus on scatterplots with millions to billions of objects, where significant overlap of marks is unavoidable, leading to visual clutter that makes the visualization ineffective. To address this issue in scatterplots, there has been substantial research [71, 73, 87, 84] on devising aggregation-based scatterplots using visual aggregates such as contours or hexagon bins. While avoiding visual clutter, the resulting visualization lacks the functionality to inspect individual objects, which is a fundamental scatterplot task [110]. Prior works also used trans-

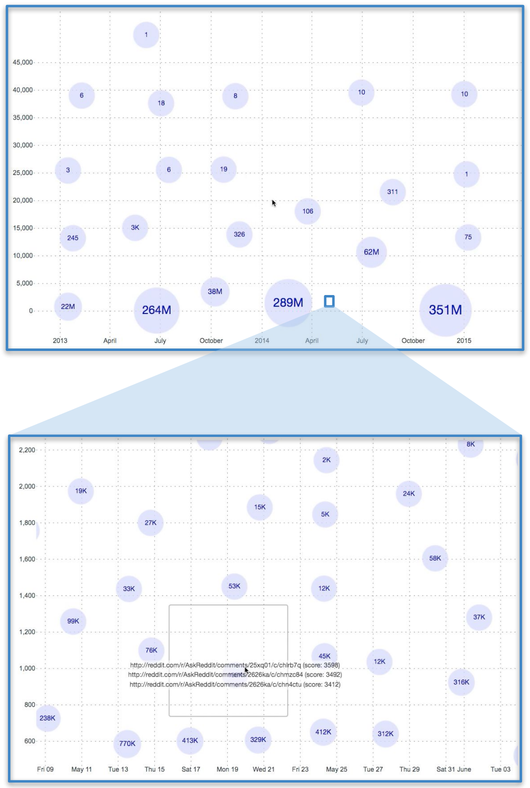
Table 3.1: Comparison of existing systems for authoring SSVs.

	High Scalability (scale well to billions of points)	Concise Authoring (10s of lines of code)	Diverse SSV Designs (arbitrary marks, bounded density, etc)
Kyrix-S	✓	✓	✓
General DoD systems (e.g. Kyrix, Pad, Jazz, ZVTM)			✓
Specialized SSV systems (e.g. [62], [32], [83], [2])		✓	

parency [57, 80], animation [33] and displacements of objects [132, 77, 128] to ease the overdraw problem. However, due to limited screen resolution, these methods have scalability limits.

On the other hand, the use of zooming in scatterplots has the potential to effectively mitigate visual clutter. By expanding the 2D Cartesian plane into a series of zoom levels with different scales, more screen resolution becomes available, allowing objects to be placed in a way that possibly avoids occlusion and excessive density. Interacting with large amounts of individual objects thus becomes feasible. Aggregation-based marks such as circles or heatmaps can still be used to visualize groups of objects. Figure 3-1 shows such a visualization created by Kyrix-S, the system we introduce in this chapter, which shows one billion comments made by users on `Reddit.com` from January 2013 to February 2015, where X is the posting time and Y is the number of characters in the comments. Additional examples are in Figure 3-2. For simplicity, we term such visualizations *scalable scatterplot visualizations*, or SSV.

There has been significant work on building systems/toolkits to aid the creation of SSVs (e.g. [22, 62, 45]). 3.1 shows a comparison of existing systems for authoring SSVs. Specifically, prior systems can be classified into two categories: *general DoD systems* (next to last row) and *specialized SSV systems* (last row). Kyrix is a typical example of a general DoD system. These systems are typically expressive, supporting



```
http://reddit.com/r/AskReddit/comments/25xq01/c/chlrb7q (score: 3598)
http://reddit.com/r/AskReddit/comments/2626ka/c/chmzc84 (score: 3492)
http://reddit.com/r/AskReddit/comments/2626ka/c/chn4ctu (score: 3412)
```

```
{
  data: {query: "SELECT * FROM comments;"},
  layout: {
    x: {field: "created_utc", extent: [1356998400, 1425167999]},
    y: {field: "body_len",  extent: [0, 10000]},
    z: {field: "score", order: "desc"}
  },
  marks: {
     cluster: {
        mode: "circle",
        config: {circleMinSize: 50, circleMaxSize: 80}
     },
     hover: {
        rankList: {
           mode: "custom",
           custom: redditCommentRenderer,
           topk: 3
        },
        boundary: "bbox"
     }
  },
  config: {axis: true}
};
```

Figure 3-1: A scalable scatterplot visualization created by Kyrix-S and its Kyrix-S specifications. One billion comments made by users on `Reddit.com` from Jan 2013 to Feb 2015 are visualized on 15 zoom levels. On every level, X and Y axes are respectively the posting time and length of the comments. Each circle represents a cluster of comments. The number inside each circle is the size of the cluster and also encodes the radius of the circle. As can be seen, the distribution of comments is roughly uniform over time but rather skewed in length. Through a pan or zoom interaction, the user can navigate this multi-scale data space to get either an overview (left) or inspect an area of interest (middle). One can hover over a circle to see three highest-scored comments in the cluster, as well as a bounding box showing the boundary of the cluster.

not only SSVs, but also DoD visualizations of other types of data (e.g. hierarchical and temporal data) or that connect multiple 2D semantic spaces[1]. Specialized SSV systems (e.g. [62, 32]), on the other hand, generally have a narrow focus on SSVs.

While these systems have been shown to be effective, they can suffer from some drawbacks that limit their ability to support general SSV authoring at scale. In particular, **limited scalability** is a common drawback of both types of systems. As often as not, implementations assume all objects reside in the main memory of a computer [32, 90, 83, 62, 84, 93, 22, 106].

General DoD systems, while being flexible, generally incur **too much developer work** due to their low-level nature. When authoring an SSV, the developer needs to manually generate the layout of visual marks on zoom levels. In very large datasets, there will be many levels (e.g. Google Maps has 20). Individually specifying the layout of a set of levels is tedious and error-prone. In particular, big skewed data can make it challenging for the developer to specify a layout that avoids occlusion and excessive density in the visualization.

Another drawback of specialized SSV systems is **low flexibility**. Oftentimes systems are hardcoded for specific scenarios (e.g., supporting specific types of visual marks such as heatmaps [104, 84] or points [45, 32], enforcing a density budget but not removing overlap, etc.) and are not extensible to general use cases. The developer cannot make free design choices when using these systems, and is forced to constantly switch tools for different application requirements.

In this chapter, we describe Kyrix-S[2], a system for SSV authoring at scale which addresses all issues of existing systems.

To enable rapid authoring, we present a declarative language for SSVs, which

[1]A 2D semantic space consists of zoom levels sharing the same coordinate system and visualizing the same type of objects. An SSV has only one semantic space. General DoD systems typically allow "semantic jumping" from one semantic space to another [106] (e.g. from a space of Reddit comments to a space of Reddit forums). More examples can be found in Chapter 2.

[2]The birth of Kyrix-S is driven by the limitations we see when we use Kyrix, the general DoD system described in the previous chapter, to build real-world SSV-based applications. The name Kyrix-S here suggests that we implement Kyrix-S as an extension of Kyrix for SSVs, rather than a replacement. S may suggest scale, scatterplots, skew or spatial partitioning. More detailed discussion on the relationship between the two systems can be found in Sections 3.2 and 3.7.

Kyrix-S implements. To enable rapid authoring, we abstract away low-level details such as rendering of visual marks. The developer can author a complex SSV in a few tens of lines of JSON. We show that compared to Kyrix, this is 4X–9X reduction in specification on several examples. In addition, we build a gallery of SSVs to show that our language is expressive and that the developer can easily extend it to add his/her own visual marks.

This language for SSVs is supported by an algorithm that automatically chooses the layout of visual marks on all zoom levels, thereby freeing the developer from writing custom code. We store objects in a multi-node parallel database using multi-node spatial indexing. As we show in Section 3.8, this allows us to respond to any pan/zoom action in under 500 ms on datasets with billions of objects.

To summarize, we make the following contributions:

- An integrated system called Kyrix-S for declarative authoring and rendering of SSVs at scale.
- A concise and expressive declarative language for describing SSVs (Section 3.4).
- A framework for offline database indexing and online serving that enables interactive browsing of large SSVs (Sections 3.5 and 3.6).

3.2 Related Works

3.2.1 General DoD Visualization Systems

A number of systems have been developed to aid the creation of general DoD visualizations(e.g. [22, 23, 106]). These systems are expressive and capable of producing not only SSVs, but also DoD visualizations of other types of data (e.g. hierarchical, temporal, relational, etc) or with multiple semantic spaces connected by jumps [106]. However, as mentioned in the introduction, these systems fall short in supporting SSVs due to **limited scalability** and **too much developer work**.

The Kyrix system introduced in Chapter 2 is a general DoD visualization system we have developed. Here, we summarize the novel aspects of Kyrix-S compared to Kyrix:

- Kyrix-S provides a high-level language for SSVs, which enables much shorter specification than what Kyrix's language requires for the same SSV (see Section 3.8.2 for an empirical comparison);
- Kyrix-S implements a layout generator which frees the developer from deciding the layout of objects on zoom levels. Kyrix does not assist the developer in choosing an object layout, which makes authoring SSVs using Kyrix fairly challenging;
- Kyrix-S is integrated with a distributed database which scales to billions of objects. In contrast, Kyrix only works with a single-node database which cannot scale to billions of objects.

Note that Kyrix-S has a narrow focus on SSVs and is not intended to completely replace general DoD visualization systems. As we will discuss more in Section 3.7, we build Kyrix-S on top of Kyrix as an extension.

3.2.2 Specialized SSV Systems

There has been considerable effort made to develop specialized SSV systems, which mainly suffer from two limitations: **low flexibility** and **limited scalability**.

Many systems focus on a small subset of the SSV design space, and are not designed/coded to be easily extensible. For example, many focus on specific visual marks such as small-sized dots (e.g. [45, 32, 76]), heatmaps (e.g. [104, 84, 100, 93, 85]), text [105], aggregation-based glyphs [83, 24] and contours [90]. Some works maintain a visual density budget [45, 104, 62], while some focus on overlap removal [24, 32, 47].

In contrast to these systems, Kyrix-S aims at a much larger design space. We provide a diverse library of visualization templates that are suitable for a variety of scatterplot tasks. For high extensibility, Kyrix-S's declarative language is designed with extensible components for authoring custom visual marks.

In addition to the limited focus, most specialized SSV systems cannot scale to large datasets with billions of objects due to an in-memory assumption [2, 46, 62, 90, 32, 81, 51, 97]. We are only aware of the work by Perrot et al. [104] which renders large heatmap visualizations using a distributed computing framework. However, that work only focuses on heatmaps.

Specialized SSV systems generally come with a layout generation module which computes the layout of visual marks on each zoom level. The design of Kyrix-S's layout generation is inspired by many of them and bears similarities in some aspects. For example, favoring placements of important objects on top zoom levels is adopted by many works [62, 105, 45]. The idea of enforcing a minimum distance between visual marks comes from blue-noise sampling strategies [104, 32, 62].

However, the key differentiating factor of Kyrix-S comes from its more stringent requirements on scalability and the design space. Theses requirements (see Section 3.3) pose new algorithmic challenges. For instance, Sarma et al. [45] uses integer programming to generate the layout without considering overlaps of objects. To enable overlap removal, one needs to add $O(n^2)$ pairwise non-overlap constraints into the integer program, making it hard to solve in reasonable time. As another example, Guo et al. [62] and Chen et al. [32] do not support visual marks that show a group of objects with useful aggregated information. This requires a bottom-up aggregation process which breaks their top-down algorithmic flow. In order to scale to billions of objects, Kyrix-S cannot rely on existing algorithms and instead needs to compute visual mark layouts in parallel using a distributed algorithm as described in Section 3.6.

3.2.3 Static Scatterplot Designs

Alleviating the overdraw problem of static scatterplot visualizations has been a popular research topic for a long time. Many methods have been proposed, including binned aggregation [94, 87, 73], appearance optimization [57, 80, 33], data jittering [132, 77, 128] and sampling [48, 36]. We refer interested readers to existing surveys on scatterplot tasks and designs [110], binned aggregation [71] and visual clutter reduction [53, 54]. Kyrix-S's design follows many guidelines in these works, which we elaborate in Section 3.3.

3.2.4 Declarative Visualization Languages

Numerous declarative languages have been proposed for authoring visualizations at different levels of abstractions. The first of these is Wilkinson's grammar of graphics (GoG) [135], which forms the basis of subsequent works. For example, ggplot2 [134] is the direct implementation of GoG in R and is widely used. D3 [28] and Protovis [27] are low-level libraries that provide useful primitives for authoring basic visualizations. Vega is the first language that concerns specifications of interactions. Built on top of Vega, Vega-lite [113] offers a more succinct language for authoring interactive graphics. Recently, more specialized languages have emerged for density maps [73], unit visualizations [101], and DoD visualizations (Kyrix).

Despite the diversity of this literature, not many languages support SSVs well. Some low-level languages such as D3 [28], Vega [114] and Kyrix's language can express SSVs, but the specification is often verbose due to their low-level and general-purpose nature. Also, they do not help the developer manage the layout of visual marks. Kyrix-S, on the contrary, uses a high-level language that abstracts away unimportant low-level details and is designed with several components that help the developer control the layout, density and occlusion.

3.3 Design Goals

Limitations of prior art, existing guidelines and our experience with SSV users drive the design of Kyrix-S. Here, we present a few goals we set out to achieve.

G1. Rapid authoring. Our declarative language should enable specification of SSVs in a few tens of lines of code. This goal is inspired by the design rationale of several high-level declarative languages (e.g. Vega-lite [113] and Atom [101]), and driven by the limitations we see in using Kyrix author SSVs.

G2. Visual expressivity. Kyrix-S should enable exploration of a broad SSV design space and not limit itself to specific visual representations. Moreover, it is crucial to allow inspection of individual objects in addition to showing aggregation information. As outlined by Sarikaya et al. [110], there are four common object-centric scatterplot tasks: *identify object*, *locate object*, *verify object* and *object comparison*. A recent study [81] also highlights the importance of browsing objects in multi-scale visualizations.

G3. Usable SSVs. The SSVs authored with Kyrix-S should be usable, e.g. free of visual clutter, using simple visual aggregates, etc. We identify usability guidance from a range of surveys and SSV systems (e.g. [54, 62, 45]), which we formally describe in Section 3.6.

G4. Scalability. Kyrix-S should be able to handle large datasets with billions of objects and potentially skewed spatial distribution. This goal has the following two subgoals:

- **G4-a. Scalable offline indexing**. Offline indexing should finish in reasonable time on big skewed data, and scale well as the data size grows.
- **G4-b. Interactive online serving**. The end-to-end response time to any user interaction (pan or zoom) should be under 500 ms, an empirical upper bound that ensures fluid interactions [11].

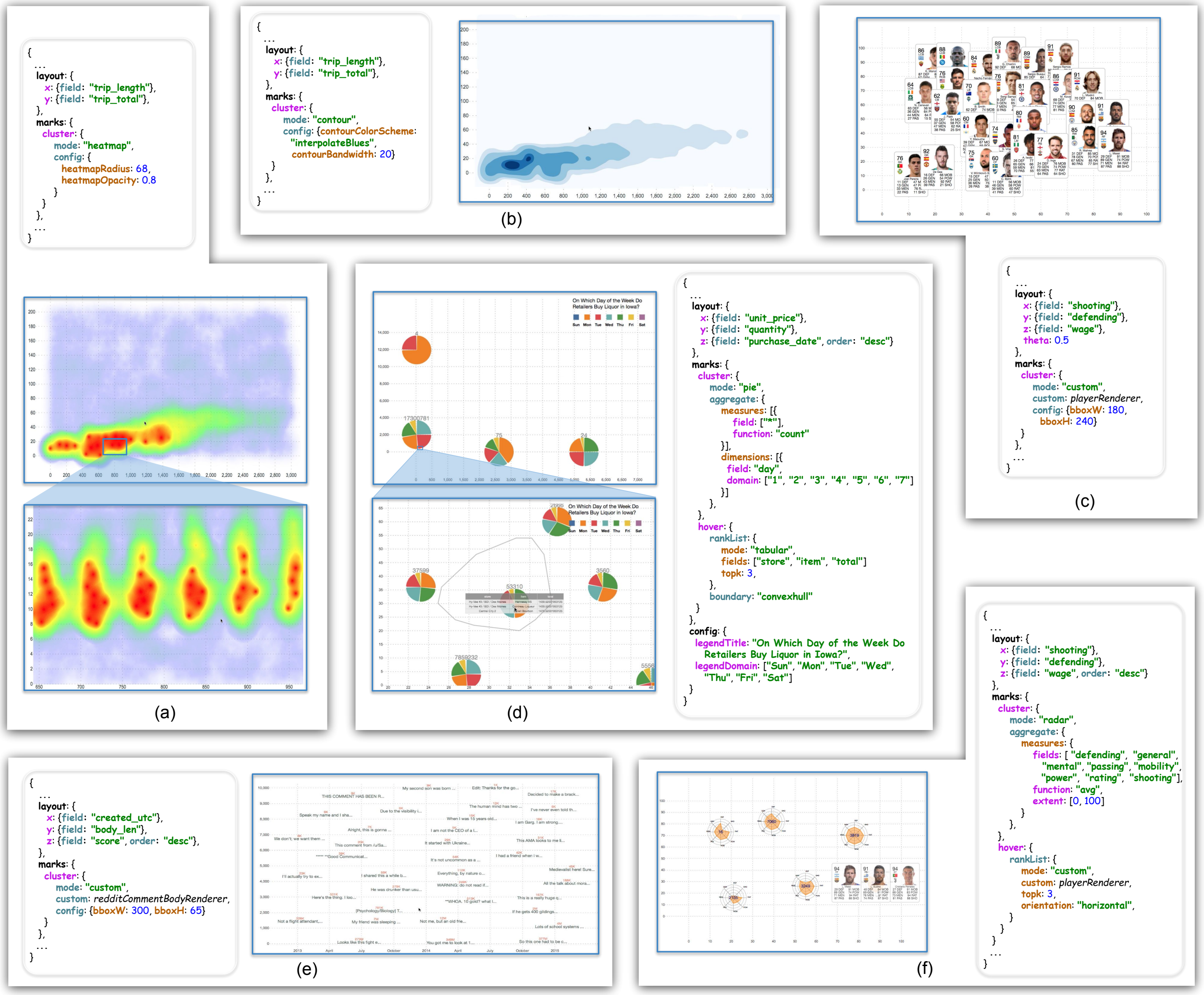
In the rest of Chapter 3, we justify the design choices we make by referencing the above goals when appropriate.

3.4 Declarative Language

In this section, we present Kyrix-S's declarative language. We start with showing a gallery of example SSVs authored with Kyrix-S (Section 3.4.1), which we then use to illustrate the design of the language in Section 3.4.2.

3.4.1 Example SSVs

Figure 3-2 shows a gallery of SSVs and their specifications.



```
{
  ...
  layout: {
    x: {field: "trip_length"},
    y: {field: "trip_total"},
  },
  marks: {
   cluster: {
      mode: "heatmap",
      config: {
        heatmapRadius: 68,
        heatmapOpacity: 0.8
      }
    }
  },
  ...
}
```

```
{
  ...
  layout: {
    x: {field: "trip_length"},
    y: {field: "trip_total"},
  },
  marks: {
   cluster: {
      mode: "contour",
      config: {contourColorScheme:
        "interpolateBlues",
         contourBandwidth: 20}
    }
  },
  ...
}
```

```
{
  ...
  layout: {
    x: {field: "shooting"},
    y: {field: "defending"},
    z: {field: "wage"},
    theta: 0.5
  },
  marks: {
   cluster: {
      mode: "custom",
      custom: playerRenderer,
      config: {bboxW: 180,
        bboxH: 240}
    }
  },
  ...
}
```

```
{
  ...
  layout: {
    x: {field: "unit_price"},
    y: {field: "quantity"},
    z: {field: "purchase_date", order: "desc"}
  },
  marks: {
   cluster: {
      mode: "pie",
      aggregate: {
        measures: [{
          field: ["*"],
          function: "count"
        }],
        dimensions: [{
         field: "day",
         domain: ["1", "2", "3", "4", "5", "6", "7"]
        }]
      },
   },
   hover: {
      rankList: {
        mode: "tabular",
        fields: ["store", "item", "total"]
        topk: 3,
      },
      boundary: "convexhull"
   }
  },
  config: {
   legendTitle: "On Which Day of the Week Do
     Retailers Buy Liquor in Iowa?",
   legendDomain: ["Sun", "Mon", "Tue", "Wed",
     "Thu", "Fri", "Sat"]
  }
}
```

```
{
  ...
  layout: {
    x: {field: "created_utc"},
    y: {field: "body_len"},
    z: {field: "score", order: "desc"},
  },
  marks: {
   cluster: {
      mode: "custom",
      custom: redditCommentBodyRenderer,
      config: {bboxW: 300, bboxH: 65}
   }
  },
  ...
}
```

```
{
  ...
  layout: {
    x: {field: "shooting"},
    y: {field: "defending"},
    z: {field: "wage", order: "desc"}
  },
  marks: {
   cluster: {
      mode: "radar",
      aggregate: {
        measures: {
          fields: [ "defending", "general",
            "mental", "passing", "mobility",
            "power", "rating", "shooting"],
          function: "avg",
          extent: [0, 100]
        }
      },
   },
   hover: {
      rankList: {
        mode: "custom",
        custom: playerRenderer,
        topk: 3,
        orientation: "horizontal",
      }
    }
  }
  ...
}
```

Figure 3-2: A gallery of SSVs authored with Kyrix-S and their specifications. (a): a heatmap of 178.3 million taxi trips in Chicago since 2013, X: trip length (seconds), Y: trip total (dollars); (b): the same dataset/axes as (a) in contour plots; (c): an SSV of 18,207 soccer players in the video game FIFA19, X: shooting rating, Y: defense rating, Z: wage (i.e. highly-paid players appear on top zoom levels); (d): a pie-based SSV of 17.3 million liquor purchases by retailers in Iowa, X: unit price (dollars), Y: quantity (# of bottles), Z: purchase date; (e): a text visualization of the dataset of one billion Reddit comments in Figure 3-1 with the same axes; (f): the same dataset/axes as (c) in a radar-chart.

Taxi. In Figure 3-2a, a multi-scale heatmap shows the distribution of 178.5M taxi trips in Chicago since 2013, where X is trip length (in seconds) and Y is trip total (in dollars). In the overview (upper), the long thin "heat" region suggests that most trips have a similar total-length ratio. In a zoomed-in view (lower), we see vertical "heat" regions around entire minutes. In fact, more than 70% of the trips have a length of entire minutes, indicating the possible prevalent use of minute-precision timers. Figure 3-2b is the same representation of this dataset in contour lines.

FIFA. The SSV in Figure 3-2c visualizes 18,207 soccer players in the video game FIFA 19. X and Y are respectively the shooting and defensive rating of players. Players with the highest wages are shown at top levels. Lesser-paid players are revealed as one zooms in. Figure 3-2f is a radar-based SSV with the same X and Y. Each radar chart shows the averages of eight ratings (e.g. passing, power) of a cluster of players. When hovering over a radar, three players from that cluster with the highest wages are shown.

Liquor. Figure 3-2d is an SSV of 17.3M liquor purchases by retailers in Iowa since 2012. X and Y axes are the unit price (dollars) and quantity (# of bottles) of the purchases. Each pie shows a cluster of purchases grouped by day of the week. One can hover over a pie to see a tabular visualization of the three most recent purchases, as well as a convex hull showing the boundary of the cluster.

Reddit. Figure 3-2e is another representation of the one-billion Reddit comments dataset. Different from Figure 3-1, comments are directly visualized as non-overlapping texts. The number above each comment represents how many comments are nearby, giving the user an understanding of the data distribution hidden underneath.

3.4.2 Language Design

The primary goal of Kyrix-S's declarative language is to help the developer quickly navigate a large SSV design space (**G1** and **G2**). The high-level design of the language closely follows a survey of scatterplots designs and tasks by Sarikaya et al. [110], which outlined four common design variables of scatterplot visualizations: *point encoding* (i.e. visual representation of one object), *point grouping* (i.e. visual representation of

$$\langle\text{SSV}\rangle ::= \langle\text{Marks}\rangle\langle\text{Layout}\rangle\langle\text{Data}\rangle[\text{Config}] \tag{3.1}$$

; marks

$$\begin{align}
\langle\text{Marks}\rangle &::= \langle\text{Cluster}\rangle[\text{Hover}] \tag{3.2}\\
\langle\text{Cluster}\rangle &::= \langle\text{Mode}\rangle\langle\text{Aggregate}\rangle[\text{Config}] \tag{3.3}\\
\langle\text{Hover}\rangle &::= \langle\text{Ranklist}\rangle\langle\text{Boundary}\rangle[\text{Config}] \tag{3.4}\\
\langle\text{Mode}\rangle &::= \textit{Circle} \mid \textit{Contour} \mid \textit{heatmap} \mid \nonumber\\
&\qquad \textit{Radar} \mid \textit{Pie} \mid \langle\text{Custom}\rangle \tag{3.5}\\
\langle\text{Aggregate}\rangle &::= \langle\text{Dimension}\rangle * \langle\text{Measure}\rangle + \tag{3.6}\\
\langle\text{Ranklist}\rangle &::= \langle\text{Topk}\rangle(\textit{Tabular} \mid \langle\text{Custom}\rangle) \tag{3.7}\\
\langle\text{Boundary}\rangle &::= \textit{Convex Hull} \mid \textit{BBox} \tag{3.8}\\
\langle\text{Custom}\rangle &::= \text{Custom JS mark renderer} \tag{3.9}\\
\langle\text{Dimension}\rangle &::= \langle\text{Field}\rangle[\text{Domain}] \tag{3.10}\\
\langle\text{Measure}\rangle &::= \langle\text{Field}\rangle\langle\text{Function}\rangle[\text{Extent}] \tag{3.11}\\
\langle\text{Topk}\rangle &::= \text{A positive integer} \tag{3.12}\\
\langle\text{Domain}\rangle &::= \text{A list of string values} \tag{3.13}\\
\langle\text{Function}\rangle &::= \textit{Count} \mid \textit{Sum} \mid \textit{Avg} \mid \textit{Min} \mid \nonumber\\
&\qquad \textit{Max} \mid \textit{Sqrsum} \tag{3.14}
\end{align}$$

; layout

$$\begin{align}
\langle\text{Layout}\rangle &::= \langle\text{X}\rangle\langle\text{Y}\rangle\langle\text{Z}\rangle[\text{Theta}] \tag{3.15}\\
\langle\text{X}\rangle &::= \langle\text{Field}\rangle[\text{Extent}] \tag{3.16}\\
\langle\text{Y}\rangle &::= \langle\text{Field}\rangle[\text{Extent}] \tag{3.17}\\
\langle\text{Z}\rangle &::= \langle\text{Field}\rangle\langle\text{Order}\rangle \tag{3.18}\\
\langle\text{Theta}\rangle &::= \text{A number between 0 and 1} \tag{3.19}\\
\langle\text{Field}\rangle &::= \text{A database column name} \tag{3.20}\\
\langle\text{Extent}\rangle &::= \text{A pair of float numbers} \tag{3.21}\\
\langle\text{Order}\rangle &::= \textit{Ascending} \mid \textit{Descending} \tag{3.22}
\end{align}$$

; data

$$\langle\text{Data}\rangle ::= \text{a database query} \tag{3.23}$$

; config

$$\langle\text{Config}\rangle ::= \text{Key value pairs} \tag{3.24}$$

Figure 3-3: Kyrix-S's declarative language in the BNF notation. Inside ⟨⟩ or [] is a component. Every rule (1-24) defines what the left-hand side component is composed of. On the right hand side of a rule, | means OR, * means zero or more, + means one or more and [] means that a component is optional.

a group of objects), *point position* (e.g. subsampling, zooming) and *graph amenities* (e.g. axes, annotations).

These design variables map to the highest-level components in Kyrix-S's language, i.e., *Marks*, *Layout*, *Data* and *Config*, as illustrated in Figure 3-3 using the BNF notation [79]. We elaborate the design of them in the following subsections.

Marks: Templates + Extensible Components

The *Marks* component (Rules 2-14[3]) defines the visual representation of one or more objects, and covers both *point encoding* and *point grouping* in [110].

Visual marks of a single or a cluster of objects span a huge space of possible visualizations. To keep our language high-level (**G1**), we adopt a *templates+extensible components* methodology, where we provide a diverse library of template mark designs, and offer extensible components for authoring custom marks.

We divide the *Marks* component into two subcomponents: *Cluster* (Rule 3) and *Hover* (Rule 4).

Cluster: cluster marks are static marks rendering one or a group of objects. Currently, Kyrix-S has five built-in *Cluster* marks including CIRCLE (Figure 3-1), CONTOUR (Figure 3-2b), HEATMAP (Figure 3-2a), RADAR (Figure 3-2f) and PIE (Figure 3-2d). The developer can choose one of these marks by specifying just a name (**G1**). These built-in *Cluster* marks are carefully chosen to cover a range of aggregate-level SSV tasks [110]. For example, heatmaps and contour plots enable the user to *characterize distribution* and *identify correlation* between the two axes. The user can perform *numerosity comparison* and *identify anomalies* with circle-based SSVs. Radar-based and pie-based SSVs allow for *exploring object properties within a neighborhood.* For fast authoring, Kyrix-S sets reasonable default values for many parameters (**G1**), e.g., inner/outer radius of a pie and bandwidth of heatmaps. The developer can also make customizations (**G2-b**) using a *Config* component (Rules 3 and 24).

With the *Custom* component (Rules 5 and 9), the developer can specify custom

[3]Hereafter, rules referenced inside parentheses implicitly refer to rules in Figure 3-3. A rule defines the composition logic of one component in the language.

visual marks easily. For example, player profiles in Figure 3-2c are specified as a custom visual mark. Kyrix-S currently supports arbitrary Javascript-based renderers (e.g. D3 [28] or Vega-lite-js [14]). For increased expressivity, a custom mark renderer is passed all useful information about a cluster of objects, including aggregation information in both *Aggregate* and *Hover*. As an example, the custom renderer in Figure 3-2e displays both an example comment and the size of the cluster. More importantly, *Custom* also facilitates rapid future extension of Kyrix-S, allowing easy addition of built-in mark types.

The *Aggregate* component (Rule 6) informs Kyrix-S details of aggregations statistics shown by a *Cluster* mark. This component is composed of *Dimension*s (Rule 10) and *Measure*s (Rule 11). A *Dimension* is a categorical field of the objects that indicates how objects are grouped (e.g. by day of the week in Figure 3-2d). A *Measure* defines an aggregation statistic (e.g. average of a rating in Figure 3-2f). Currently Kyrix-S supports six aggregation functions: count, average, min, max, sum and square sum (Rule 14).

Hover: Hover marks add more expressivity into the language by showing additional marks when the user hovers over a *Cluster* mark. For example, in Figure 3-1 three example comments are shown upon hovering a circle. The motivation for adding this component is two-fold.

First, as outlined in **G2**, we want to enable tasks that require inspection of individual objects in addition to showing visual aggregates with *Cluster* marks. To this end, we design a *Ranklist* component which visualizes objects with top-k importance (Rule 7). The importance of objects is defined in the *layout* component as a field from the objects. We offer a default tabular visualization template (e.g. Figure 3-2d), and allow custom marks via *Custom* (e.g. player profiles in Figure 3-2f).

Secondly, multi-scale visualizations often suffer from the "desert fog" problem [74], where the user is lost in the multi-scale zooming space and not sure what is hidden underneath the current zoom level. *Boundary* is designed to aid the user in the navigation process (**G3**) by showing the boundaries of a cluster of objects (Rule 8), using either the convex hull (Figure 3-2d) or the bounding box (Figure 3-1). By hinting that there is more to see by zooming in, more interpretability is added to the *Cluster* marks [54].

Layout: Configuring All Zoom Levels at Once

The *Layout* component (Rules 15-22) controls the placement of visual marks[4] on zoom levels, which corresponds to the *point position* design variable in [110]. We aim to assist the developer in specifying the layout for all zoom levels together rather than independently, motivated by the limitation of general DoD visualization systems [23, 22] that mark placements are manually configured for every zoom level.

X and Y (Rules 16 and 17) define the two spatial dimensions. The only specifications required are two raw data columns that map to the two dimensions (e.g. trip length and total in Figures 3-2a and 3-2b). An optional *Extent* component (Rule 21) can be used to indicate the visible range of raw data values on the top zoom level.

The Z component (Rule 18) controls how visual marks are distributed across zoom levels. Drawn from prior works [45, 62, 105], we use a usability heuristic that makes objects with higher importance more visible on top zoom levels. The importance is defined by a field of the objects. For example, in Figure 3-2e, highest-scored comments are displayed on top zoom levels.

Optionally, *Theta* is a number between 0 and 1 indicating the amount of overlap allowed between *Cluster* marks (Rule 19), with 0 being arbitrary overlap is allowed and 1 being overlap is not allowed. For instance, *Theta* is 0.5 in Figure 3-2c, making the player profiles overlap to a certain degree.

The above layout-related parameters serve as inputs to the layout generator, which we detail in Section 3.6.

Data and Config

We assume that the raw spatial data exists in the database, and can be specified as a SQL query (Rule 23).

[4]For KDE-based SSVs (e.g. heatmaps and contours), a visual mark here refers to the kernel density estimates generated by a weighted object.

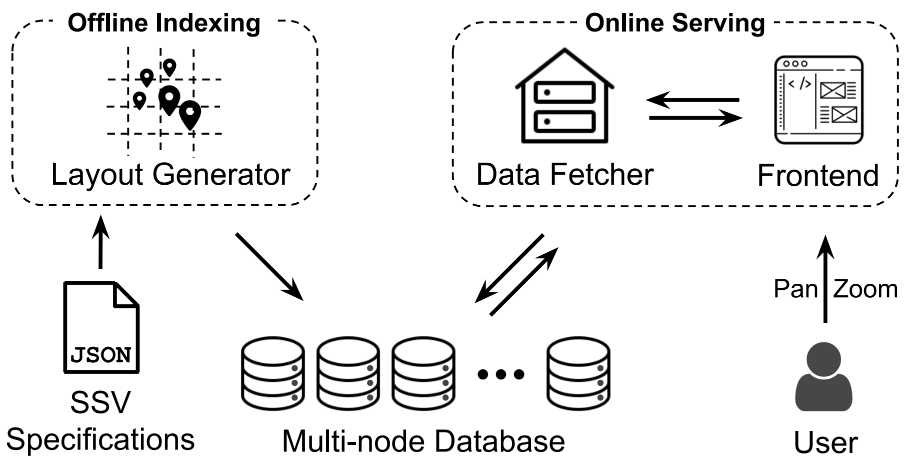


Figure 3-4: Kyrix-S optimization framework.

The highest-level *Config* component corresponds to the design variable *graph amenities* in [110]. The developer can use it to specify global rendering parameters such as the size of the top zoom level, number of zoom levels, as well as annotations such as axes, grid lines and legends.

3.5 Optimization Framework

Figure 3-4 illustrates the optimization framework adopted by Kyrix-S to scale to large datasets(**G4**). There are two main phases: offline indexing and online serving. Specifically, given an SSV specification, the layout generator computes offline the placement of visual marks on zoom levels using several usability considerations (**G3**), e.g., bounded visual density, free of clutter, etc. Along the way, useful aggregation information (e.g. statistics and cluster boundaries) is also collected. The computed layout information is stored in a multi-node database with multi-node spatial indexes. Online, the data fetcher communicates with the frontend and fetches data in user's viewport from the multi-node database with interactive response times (**G4-b**). In the next section, we describe these two components in greater detail.

3.6 Layout Generation and Data Fetching

Here, we first describe how we model the layout generation problem (Section 3.6.1). We then describe a single-node layout algorithm (Section 3.6.2), which is the basis of a distributed algorithm detailed in Section 3.6.3. Lastly, Section 3.6.4 describes the design of the data fetcher.

3.6.1 Layout Generation: Problem Definition

We assume that there is a discrete set of zoom levels numbered 1, 2, 3... from top to bottom with a constant zoom factor between adjacent levels (e.g. 2 as in many web maps). The layout generation problem concerns how to, in a scalable manner, place visual marks onto these zoom levels in a general way that works for any SSV that Kyrix-S's declarative language can express (**G2**).

To aid the formulation of the layout generation problem, we collect a set of existing layout-related usability considerations from prior SSV systems and surveys [62, 24, 32, 45, 54, 83], and list them as subgoals of **G3: Usable SSVs**.

G3-a. Non/partial overlap. *Cluster* visual marks (Rule 3) should not overlap or only overlap to a certain degree (if specified by *Theta* in Rule 19). For simplicity, we assume that *Cluster* marks have a fixed-size bounding box, which is either decided by Kyrix-S or specified by the developer (see Figure 3-2e for an example). We then only check the overlap of bounding boxes.

G3-b. Bounded visual density. Mark density in any viewing region should not exceed an upper bound. Excessive density stresses the user and slows down both the client and the server. Kyrix-S sets a default upper bound K on how many marks should exist in any viewport-sized region based on empirical estimates of the processing capability of the database and the frontend.

We should also avoid very low visual density, which often leads to too many zoom levels and thus increased navigation complexity. We therefore try to maximize spatial fullness without violating the overlap constraint and the density upper bound.

G3-c. Zoom consistency. If one object is visible on zoom level i, either through a custom *Cluster* mark or a *Ranklist* mark (Rule 7), it should stay visible on all levels $j > i$. This principle is adopted by many SSV systems that support inspection of individual objects (e.g. [45, 32, 62]). The rationale is to aid object-centric tasks where keeping track of locations of objects is important.

G3-d. Data abstraction quality. Data abstraction characterized by visual marks should be interpretable and not misinform the user. For *Cluster* marks, it is important to reduce *within-cluster variation* [41, 142, 54], which can be characterized by average distance of objects to the visual mark that represent them [41]. We also adopt an *importance policy*, where objects with high importance (Rule 18) should be more likely to be visible on top zoom levels than objects with low importance. This is a commonly adopted principle to help the user see representative objects early on in the navigation process [62, 45].

Discussion. Despite that subgoals **G3-a∼d** are all from existing works, we are not aware of any prior system that addresses all of them. As mentioned in Section 3.2, a key distinction of Kyrix-S's layout generation is the broad design requirements of building a general and scalable SSV authoring system. Due to this broad focus, finding an "optimal layout" with the objectives and constraints in **G3-a∼d** is hard. In fact, one prior work [45] proves that with only a subset of **G3-a∼d**, finding the optimal layout is NP-hard (for an objective function they define). Therefore, we do not attempt to define a formal constraint solving problem. Instead we keep our goals qualitative and look for heuristic solutions.

3.6.2 A Single-node Layout Algorithm

Here, we describe a single-node layout algorithm which assumes that data fits in the memory of one computer.

We assume that the X/Y placement of a *Cluster* mark comes from an object it represents. Alternatively, one could consider inexact placement of the marks (e.g. "median location" or binned aggregation), which we leave as our future work. Additionally, we assume that the X/Y placement of a *Hover* mark is the same as the corresponding *Cluster* mark. So in the rest of Section 3.6, any mention of mark refers

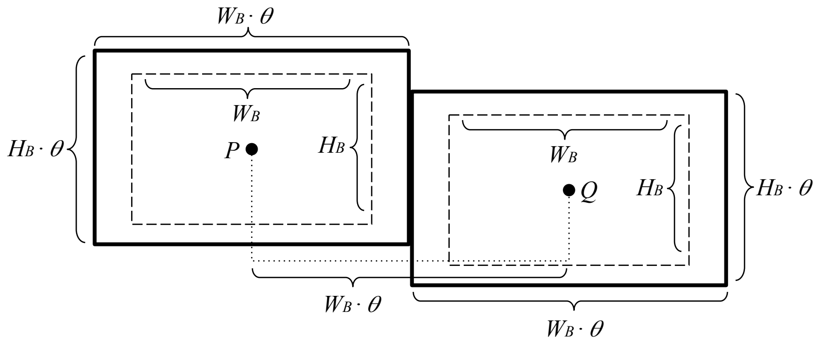


Figure 3-5: Marks P and Q with an *ncd* of θ. Inner boxes (dashed) are the bounding boxes of the marks. Outer boxes (solid) are bounding boxes scaled by a factor of θ. Scaled boxes do not overlap and touch on one side. In general, for any two marks that have *ncd* greater than θ, their bounding boxes do not overlap after being scaled by a factor of θ.

to a *Cluster* mark if not explicitly stated.

We make two important algorithmic choices. First, we enforce a minimum distance between marks in order to cope with the overlap and density constraints (**G3-a** and **G3-b**). Second, we use a hierarchical clustering algorithm to ensure zoom consistency (**G3-c**) and data abstraction quality (**G3-d**).

Enforcing a minimum distance between marks. For overlap and density constraints, we make use of the *normalized chessboard distance* (*ncd*) between two marks P and Q:

$$ncd(P, Q) = \max(\frac{|P_x - Q_x|}{W_B}, \frac{|P_y - Q_y|}{H_B})$$

where $P_x(P_y)$ is the $x(y)$ coordinate of the centroid of P in the pixel space and $W_B(H_B)$ is the width (height) of the bounding box of a mark (note that bounding boxes of marks are of the same size).

ncd helps us reason about non/partial overlap constraints. If *ncd*$(P, Q) \geq 1$, P and Q do not overlap because they are at least one bounding box width/height away on X or Y. Even if *ncd* is smaller than one, the degree of overlap is bounded. For example, if *ncd*$(P, Q) = 0.5$, the centroids of P and Q remain visible despite the potential overlap.

To this end, we set a lower bound θ on the *ncd* between any two visual marks, which is specified through the *Theta* component (e.g. Figure 3-2c) or built-in with

Cluster marks.

We also use θ to enforce the visual density upper bound K (**G3-b**). Intuitively, the smaller θ is, the closer marks are, and thus the denser the visualization is. We search for the smallest θ (for maximum spatial fullness, **G3-b**) that does not allow more than K marks in any viewport-sized region ($W_V \times H_V$). To find this θ value, we show in Figure 3-5 another perspective on how θ controls the placement of marks: enforcing that any $ncd \geq \theta$ is equivalent to scaling the bounding boxes of marks by a factor of θ, and then enforcing that none of these scaled bounding boxes overlap. So we are left with a simple bin-packing problem. For a given θ, the maximum number of marks that can be packed into a viewport is:

$$\mathcal{P}(\theta) = \left\lceil \frac{W_V}{W_B \cdot \theta} \right\rceil \cdot \left\lceil \frac{H_V}{H_B \cdot \theta} \right\rceil$$

With this, we can find the smallest θ such that $\mathcal{P}(\theta) \leq K$ using a binary search on θ.

We take the larger θ calculated/specified for the overlap and density constraints. By imposing this lower bound on *ncd*, these two constraints are strictly satisfied.

Hierarchical clustering. The key part of the algorithm is a bottom-up hierarchical clustering process. Suppose there are η zoom levels. We start with a fake bottom level $\eta + 1$ where every object is in its own cluster. Each cluster's aggregation information (e.g. aggregated stats and cluster boundaries) is initialized using the only object in it, which we call the "representative object" of a cluster in the following.

Then we build the clusters level by level. For each zoom level $i \in [1, \eta]$, we construct a new set of clusters by merging the clusters on level $i+1$. Zoom consistency (**G3-c**) is then guaranteed because each zoom level merges clusters from the one level down. By mathematical induction, we can show that if an object is visible on level i, it is visible on any level $j > i$.

Specifically, we iterate over all clusters on level $i+1$ in the order of the importance of their representative objects, which is a greedy strategy to make important objects more visible (**G3-d**). For each cluster α on level $i+1$, we search for a cluster β on the current level i with the closest *ncd*. If this *ncd* is smaller than θ, we merge α

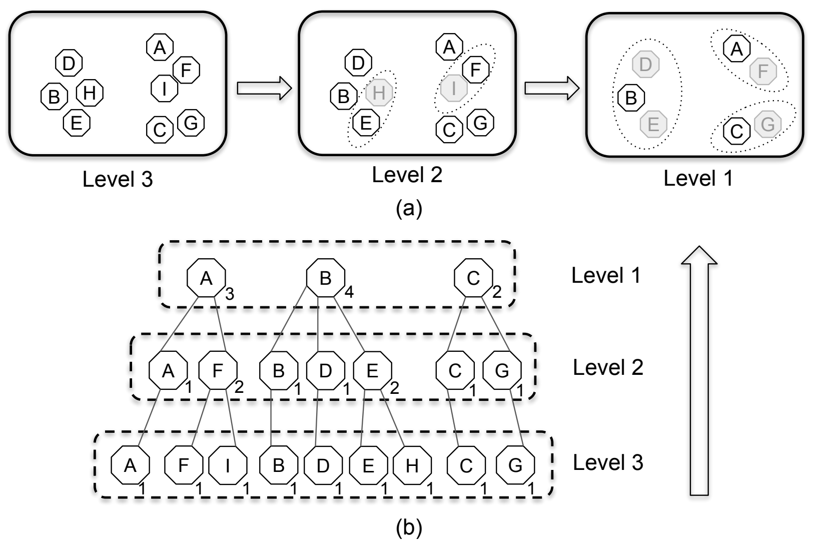


Figure 3-6: An illustration of the hierarchical clustering. There are 9 objects A-I, in decreasing order of importance. Each octagon is a cluster, with the representative object inside it. (a): Three zoom levels constructed. A dashed ellipse indicates the merging of the lighter cluster into the darker one. (b): A tree representation of the hierarchical clusters. The number next to a cluster is the number of objects this cluster represents. These numbers, along with other possible aggregation information, are computed when clusters merge.

into β; otherwise we add α to level i. By merging a cluster into its nearest neighbor (measured in *ncd*), within-cluster variances can be reduced (**G3-d**). Figure 3-6 shows an example with 9 objects and 3 zoom levels.

Identifying outliers. The single-node algorithm preserves an outlier if it is not within θ *ncd* of any other object. To identify less isolated outliers, one would need to assign to each object a score (i.e. the importance field) indicating how distant an object is from other objects. Kernel density estimations would be an example of such type of score.

Optimizations and complexity analysis. Let n be the total number of objects. When constructing clusters for level i, sorting the clusters on level $i+1$ takes $O(n \log n)$. We maintain a spatial search tree (e.g. R-tree) of the clusters on level i so that nearest neighbor searches can be done in $O(\log n)$. Inserting a new cluster into the tree also takes $O(\log n)$. Therefore, the overall time complexity of this algorithm is $O(n \log n)$ if we see the number of zoom levels as a constant.

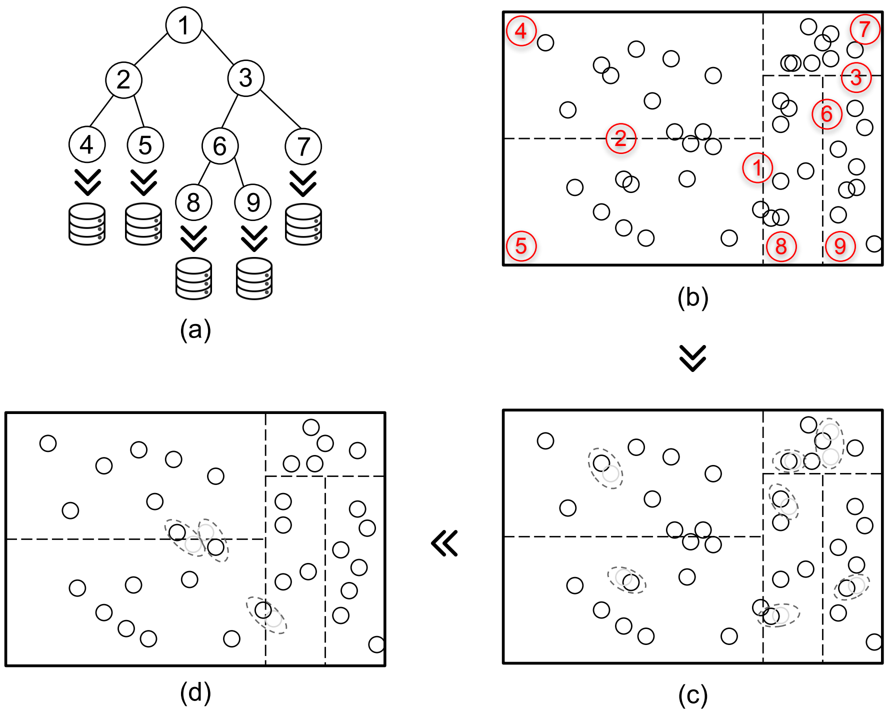


Figure 3-7: An illustration of the distributed clustering algorithm for zoom level i. (a), (b): clusters on zoom level $i+1$ are spatially partitioned and stored on multiple database nodes. KD-tree is used for skew-resilient partitioning. In (a), non-leaf tree nodes (1, 2, 3 and 6) represent KD-tree splits, while leaf tree nodes (4, 5, 7, 8 and 9) correspond to actual partitions. Each circle in (b) is a mark/cluster; (c): the single-node algorithm is run for each partition in parallel, merging clusters that have an ncd smaller than θ; (d): merging clusters close to partition boundaries.

3.6.3 A Multi-node Distributed Layout Algorithm

The algorithm presented in Section 3.6.2 only works on a single machine which has limited memory. Here, we extend it to work with a multi-node database system.[5]

Given the sequential nature of the single-node algorithm, one major challenge here is how to utilize the parallelism offered by the multi-node database. Our idea is to spatially partition a zoom level, perform clustering in each partition independently in parallel and then merge the partitions. Figure 3-7 shows an illustration of the three steps. We detail them in the following, assuming the context of constructing clusters on zoom level i from the clusters on level $i+1$.

[5]The distributed algorithm proposed here works with any multi-node database that supports basic data partitioning (e.g. Hash-based) and 2D spatial indexes.

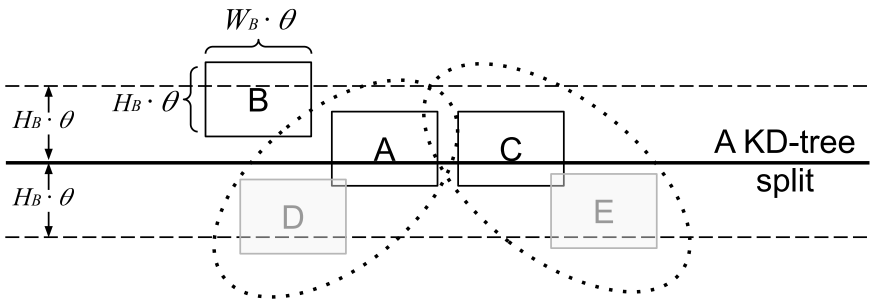


Figure 3-8: An example of merging clusters along a KD-tree split.

Step 1: skew-resilient spatial partitioning. We use a KD-tree [25] to spatially partition the 2D plane so that each resulting partition has similar number of clusters from zoom level $i+1$. Note that each cluster belongs to exactly one partition according to its centroid. A KD-tree is a binary tree (Figure 3-7a) where every non-leaf tree node represents a split of a subplane, and every leaf tree node is a final partition stored as a table in one database node. KD-tree splits are axis-aligned and alternate between horizontal and vertical as one goes down the hierarchy. For each split, the median value of the corresponding axis is used as the split point. We stop splitting when the number of clusters in a partition can fit into the memory of one database node.

Step 2: processing partitions in parallel. Since each partition fits in the memory of one database node, we can efficiently run the single-node clustering algorithm on each partition in parallel. As a result, a new set of clusters is produced in each partition where no two clusters have an *ncd* smaller than θ (Figure 3-7c).

Step 3: merging clusters on partition boundaries. After Step 2, some clusters close to partition boundaries may have an *ncd* smaller than θ. Step 3 resolves these border cases by merging clusters along KD-tree splits. We "process" (i.e. merging clusters along) KD-tree splits in a bottom-up fashion, starting with splits that connect two leaf partitions. After the KD-tree root is processed, we finish the layout generation for level i.

When processing a given split, we make use of the fact that only clusters whose centroid is within a certain distance to the split ($W_B \cdot \theta$ or $H_B \cdot \theta$ depending on the orientation of the split) need to be considered. Consider the horizontal split in Figure

3-8. The two horizontal dashed lines indicate the range of cluster centroids that we need to consider. Any cluster whose centroid is outside this range is at least θ away (in *ncd*) from any cluster on the other side of the split.

We use a greedy algorithm to process a KD-tree split. We iterate over all clusters in the aforementioned range in the order of their x coordinates (y if the split is vertical). We keep track of the last added/merged cluster α. Let β be the currently considered cluster. If $ncd(\alpha, \beta) \geq \theta$, we add β and set α to β; otherwise we merge α and β. The one with the less important representative object is merged into the other (**g3-d**). Then we update α accordingly.

Consider again Figure 3-8. There are five clusters A-E in decreasing importance order. The boxes around clusters are their bounding boxes scaled by a factor of θ. So if two boxes overlap, two corresponding clusters have an *ncd* smaller than θ (see Figure 3-5). The above algorithm iterates over the clusters in the following order: B, D, A, C, E. When $\beta = A$, $\alpha = D$. D is then merged into A because $ncd(A, D) < \theta$ and D has a less important representative object. For the same reason, E is merged into C.

Optimizations and complexity analysis. Let M be the upper bound on the number of clusters that can fit in memory. Hence there are roughly $T = \frac{n}{M}$ partitions, which means there are $O(T)$ KD-tree nodes. Determining the splitting point can be done in $O(\log n)$, thus constructing the spatial partitions takes $O(T \cdot \log n)$. Step 1 also involves distributing the clusters to the correct database node, which is often an expensive I/O bound process. So we do spatial partitioning only once based on the bottom level, and reuse the same partition scheme for other levels to avoid moving data around database nodes. Step 2 runs in $O(M \log M)$ because the single node algorithm is run in parallel across partitions. Step 3 takes $O(n \log T)$ because there are $\log T$ KD-tree levels in total, and we need to consider for each KD-tree level n clusters in the worst case. However, Step 3 is expected to run very fast in practice because most clusters are out of the range in Figure 3-8.

Other partitioning strategies. One could partition the data using fields other than x and y and then in a similar fashion, run the single-node algorithm on the resulting partitions in parallel. However, since the two spatial attributes are not involved in partitioning, objects in each partition would span the whole 2D space. So even though overlap and density constraints are satisfied within each partition, when merged together, they will very likely be violated unless extra spatial postprocessing are in place. We therefore choose to perform spatial partitioning throughout to guarantee **G3-a** and **G3-b**.

3.6.4 Data Fetching

The data fetcher's job is to efficiently fetch data in the user's viewport (**G4-b**). We make use of multi-node spatial indexes, which can help fetch objects falling in a viewport-sized region with interactive response times.

Creating multi-node spatial indexes. Suppose the j-th $(1 \leq j \leq T)$ partition on zoom level i is stored in the database table $t_{i,j}$, which has roughly M clusters. We augment all such $t_{i,j}$ with a box-typed column `bbox`, which stores the bounding box of cluster marks. We then build a spatial index on column `bbox`, by issuing the following query:

```
CREATE INDEX sp_idx ON t_{i,j} using gist(bbox);
```

where `gist` is the spatial index based on the generalized search tree [8]. In practice, these `CREATE INDEX` statements can be run in parallel by the multi-node database.

Fetching data from relevant partitions. Given a user viewport V on zoom level i, clusters from partition $t_{i,j}$ that are inside V can be fetched by the following query:

```
SELECT * FROM t_{i,j} WHERE bbox && V;
```

where `&&` is the intersection operator. The spatial index on `bbox` ensures that this query runs fast. We traverse the KD-tree to find out partitions that intersect V, run the above query on these partitions and union the results. Note that for top zoom levels that are small in size, there can be too many partitions that intersect with the viewport, which can be harmful for data fetching performance because we need to wait for sequential network trips to many database nodes. Therefore, we merge all partitions on each of the top L levels into one database table. L is an empirically determined constant based on the relative size of the zoom levels to the viewport size.

3.7 Implementation

We implement Kyrix-S as an extension to Kyrix, the general DoD system introduced in Chapter 2. This enables the developer to both rapidly author SSVs and reuse features of a general DoD system in one integrated system. For example, Kyrix supports multiple coordinated views. Without switching tools, the developer can construct a multi-view visualization in which one or more views are SSVs authored with Kyrix-S. As another example, the developer can augment SSVs with the jump functionality provided by Kyrix, where the user can click on a visual mark and jump to another SSV. Furthermore, Kyrix provides APIs for integrating a DoD visualization into a web application, which are highly desired by the SSV developers we collaborate with. Examples include programmatic interaction control, notifications of pan/zoom/jump events and getting current visible data items.

Specification compilation. Kyrix-S uses a *Node.js* module to validate the JSON-based SSV specification. Validated specifications are compiled into low-level Kyrix specifications so that part of Kyrix's frontend code can be reused to handle rendering and pan/zoom interactions. Specifically, we construct a *canvas* (see Section 2.4.2) for each zoom level, and use the *zoom* connection (see Section 2.4.3) to connect adjacent zoom levels.

Layout generator and data fetcher. Kyrix-S's layout generator and data fetcher override respectively Kyrix's index generator and data fetcher. Both components are written in the same Java application, using the Java Database Connectivity (JDBC) to talk to Citus[6], an open-source multi-node database built on top of PostgreSQL. The layout generator uses PLV8[7], a PostgreSQL extension that enables implementation

[6]https://www.citusdata.com/
[7]https://plv8.github.io/

of algorithms in Section 3.6 in Javascript functions, along with parallel execution of those functions directly inside each Citus database node.

Database deployment and orchestration. Kyrix-S provides useful scripts for one-command deployment of Kyrix-S and database dependencies (**G1**). We use Kubernetes[8] to orchestrate a group of nodes running containerized Citus and Kyrix-S built with Docker[9].

3.8 Evaluation

We conducted extensive experiments to evaluate two aspects of Kyrix-S: 1) performance and 2) authoring effort.

3.8.1 Performance

We conducted performance experiments to evaluate the online serving and indexing performance of Kyrix-S. We used both example SSVs in Figures 3-1 and 3-2 and a synthetic circle-based SSV Syn that visualizes a skewed dataset where 80% of the objects are in 20% of the 2D plane, and the rest of the 20% are uniformly distributed across the 2D plane. For database partitioning, we set $M = 2$ million, i.e., each partition has roughly 2 million objects. So for a dataset with N objects, there are $K = \lceil \frac{N}{M} \rceil$ partitions. Based on the number of partitions, we provision a Google Cloud Kubernetes cluster with $\lceil \frac{K}{8} \rceil$ `n1-standard-8` PostgreSQL nodes (8 vCPUs, 30GB memory), each serving 8 partitions.

Online Serving Performance

To measure the online response times, we used a user trace where one pans around to find the most skewed region on a zoom level, zooms in, repeats until reaching the bottom level and then zooms all the way back to the top level. We measured the

[8]https://cloud.google.com/kubernetes-engine/
[9]https://www.docker.com/

Table 3.2: Online serving time (95-th percentile, in milliseconds).

	Reddit Text (Figure 3-2e, 1B objects)	Reddit Circle (Figure 3-1, 1B objects)	Taxi Heatmap (Figure 3-2a, 178.3M objects)	Taxi Contour (Figure 3-2b, 178.3M objects)	Liquor (Figure 3-2d, 17.3M objects)
Data Fetching	14	17	32	32	14
Network	1	1	223	254	1

Table 3.3: Offline indexing time (in minutes).

	Reddit Text (Figure 3-2e, 1B objects)	Reddit Circle (Figure 3-1, 1B objects)	Taxi Heatmap (Figure 3-2a, 178.3M objects)	Taxi Contour (Figure 3-2b, 178.3M objects)	Liquor (Figure 3-2d, 17.3M objects)
Building KD-tree (Step 1)	11.8	10.5	2.7	2.4	0.7
Redistributing data (Step 1)	94.3	100.0	8.5	8.4	1.3
Parallel clustering (Step 2)	9.9	3.7	6.9	9.0	4.7
Merge partitions (Step 3)	61.3	18.2	1.1	0.8	0.1
Creating Spatial Indexes	2.4	1.3	1.2	1.2	1.3
Total	179.7	133.8	20.3	21.8	8.2

95-th percentile[10] of all data fetching time and network time.

Table 3.2 shows the results on five SSVs. The 95-percentile data fetching times were all below 32 ms. The reason was because we only fetched data from the partitions that intersect with the viewport and we built spatial indexes which sped up the spatial queries. Network times were mostly negligible except for Taxi Heatmap and Taxi Contour, where many more data items were fetched due to smaller θ values.

Figure 3-9 shows the response times on different sizes of the synthetic SSV Syn. We can see that the response times remained stably under 20 ms for data sizes from 32 million to 1 billion.

Offline Indexing Performance

Table 3.3 shows the indexing performance of the layout generator on five example SSVs. We make the following observations. First, the indexing phase finished in reasonable time: every example finished in less than 3 hours. Second, redistributing the data to the correct spatial partition was the most time consuming part since it was an I/O bound process. Fortunately, the same spatial partitions can be reused for updatable data if the spatial distribution does not change drastically. Third, parallel

[10]A 95-percentile says that 95% of the time, the response time is equal to or below this value. This is a common metric for measuring network latency of web applications.

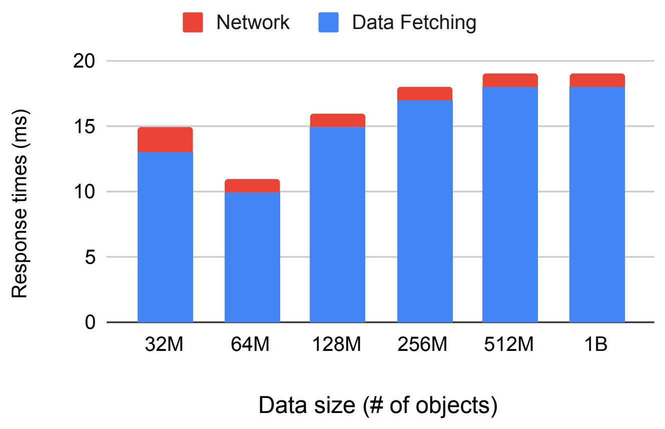


Figure 3-9: Serving scalability on the synthetic SSV Syn.

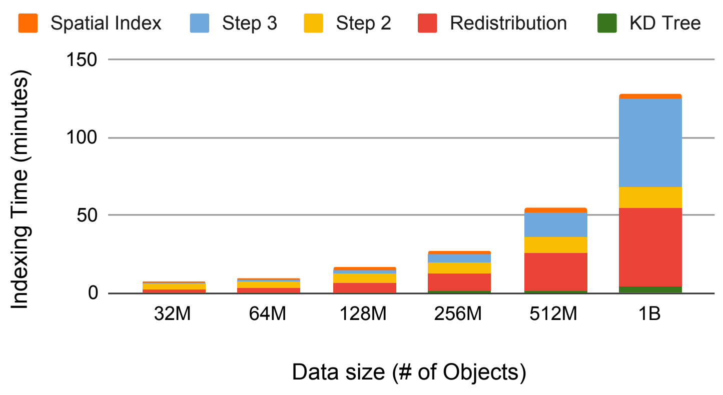


Figure 3-10: Indexing scalability on the synthetic SSV Syn.

clustering and spatial index creation took the least time because they could be run in parallel across partitions. Fourth, merging clusters along KD-tree splits was mostly a cheap process. In fact, the largest number of clusters along a KD-tree split was 16,647. The reason that this step took longer on Reddit Text than on Reddit Circle was because it had more zoom levels (20 vs. 15) due to larger mark size (text vs. circle). Moreover, iterating through objects along KD-tree splits were much more time-consuming on the bottom five levels.

Figure 3-10 shows how indexing time changed for different sizes of the synthetic SSV Syn. We can see that the indexing time scaled well as the data size grew: as data size doubled, indexing time roughly doubled as well.

Table 3.4: Comparison of lines of specifications when using Kyrix-S and Kyrix to author the two example SSVs in Figure 3-2d and Figure 3-2f.

	Kyrix-S	Kyrix	Kyrix-S's saving over Kyrix
Figure 3-2d	62 lines	568 lines	9.2×
Figure 3-2f w/ custom renderer	164 lines	610 lines	3.7×
Figure 3-2f w/o custom renderer	68 lines	514 lines	7.6×

3.8.2 Authoring Effort

In this experiment, we compared the authoring effort of Kyrix-S with Kyrix. To our best knowledge, Kyrix is the only system that offers declarative primitives for programming general DoD visualizations. Former systems/languages such as D3 [28], Pad++ [22], Jazz [23] and ZVTM [106] require procedural programming which generally takes more authoring effort as we discussed in Chapter 2. We measured lines of specifications using both systems for the two examples SSVs in Figures 3-2d and 3-2f. We used a code formatter[11] to standardize the specifications, and only counted non-blank and non-comment lines.

Table 3.4 shows the results. We can see that when authoring the two example SSVs, Kyrix-S achieved respectively 9.2× and 3.7× saving in specifications compared to Kyrix. In the second example, when we excluded the custom renderer for soccer players (which has 96 lines), the amount of savings was 7.6×. These savings came from Kyrix-S abstracting away low-level details such as rendering of visual marks, configuring zoom levels, etc.

The above comparison did not include the code for layout generation. To enable the comparison, we stored the layouts generated by Kyrix-S as database tables so that Kyrix could directly use them. However, programming the layout was in fact a challenging task, as indicated by the total lines of code of Kyrix-S's layout generator (1,439). Therefore, we conclude that Kyrix-S greatly reduced the user's effort in authoring SSVs compared to general DoD visualization systems.

[11]https://prettier.io/

3.9 Discussions

3.9.1 Limitations and Future Work

Other layout strategies. Kyrix-S's layout generator assumes that the location of a mark comes from an object. This can be relaxed to diversify our layout generator. For example, supporting inexact placement of marks such as binned aggregation [71] in SSVs is one future direction. We also plan to investigate layout strategies that concern multi-class scatterplots, e.g. how to preserve relative density orders among multiple classes [32, 36].

More built-in templates. Our declarative language is designed to enable rapid extension of the system with custom marks. This motivates us to engage more with the open-source community and enrich our built-in mark gallery with templates commonly required/authored by developers.

Incremental updates. Currently, Kyrix-S assumes that data is static and pre-materializes mark layouts. To interactively debug, the developer needs to either use a sample of the data or reduce the number of zoom levels. It is our future work to identify ways to incrementally update our mark layout upon frequent changes of developer specifications, as well as when the data itself is updated dynamically.

Animated transitions. A discrete-zoom-level model simplifies layout generation, but can potentially lead to abrupt visual effect upon level switching, especially for KDE-based renderers such as heatmaps. As future work, we will use animated transitions to counter this limitation.

Raster Images-based SSVs. The visual density constraint, partly due to limited processing capabilities of the frontend and the database, forbids the creation of dense visualizations such as point clouds [105]. We envision the use of raster images to remove this constraint for these visualizations where interaction with objects is not required.

Table 3.5: A tabular comparison of systems/toolkits for authoring SSVs. This is a more detailed version of Table 3.1.

		Scalability		Authoring Capability		Expressivity & Usability				
		Support data that cannot fit in memory	Distributed architecture (billion-object scalability)	Declarative authoring in 10s of lines of code	Automatic layout generation	Inspection of important objects	Arbitrary mark types	Show aggregation information	Support both partial & non-overlap	Automatic bounded visual density
Our work	Kyrix-S	✓	✓	✓	✓	✓	✓	✓	✓	✓
General DoD systems	Kyrix	✓				✓	✓	✓	✓	
	Pad++ [22]					✓	✓	✓	✓	
	Jazz [23]					✓	✓	✓	✓	
	ZVTM [106]					✓	✓	✓	✓	
Specialized SSV systems	Cartolabe [105]				✓	✓		✓		
	Leaflet [2]			✓	✓			✓		✓
	Beilschmidt et al. [24]				✓			✓		
	Chen et al. [32]			✓	✓				✓	✓
	Sarma et al. [45]			✓	✓	✓				✓
	Delort et al. [46]				✓			✓		
	Derthick et al. [47]			✓	✓	✓		✓		
	Disc [51]				✓	✓			✓	
	Guo et al. [62]				✓	✓			✓	✓
	Kefaloukos et al. [76]			✓	✓	✓			✓	✓
	Lekschas et al. [81]				✓	✓		✓		
	Nanocubes [84]	✓		✓	✓			✓		
	Splatterplots [90]			✓	✓	✓		✓		✓
	Nutanong et al. [97]			✓	✓	✓			✓	
	Hashedcubes [100]	✓		✓	✓			✓		
	Perrot et al. [104]	✓	✓	✓	✓			✓		

3.9.2 A Tabular Comparison

Table 3.5 shows a more detailed version of Table 3.1 introduced in Section 3.1. We include 21 systems as subrows, and 9 detailed features as subcolumns. Check marks are used to indicate that a system has the corresponding feature.

We can use this table to illustrate the claims we have made on the limitations of existing systems:

- **Both general DoD systems and specialized SSV systems have limited scalability**. We can see from the *Scalability* column that only four systems can support data beyond memory size, and only Perrot et al. [104] scales to billion-object scatterplots.

- **General DoD systems incur too much developer work**. As can be seen, none of the general DoD systems support declarative authoring in 10s of lines of specifications. Also, all of them put the burden of mark layout generation on the developers. Specialized SSV systems, in contrast, mostly provide a concise authoring interface and all have an automatic layout generator.

- **Specialized SSV systems have low flexibility**. As Table 3.5 shows, none of the specialized SSV systems support all features under "Expressivity & Usability". In fact, the number of such features supported by any specialized SSV system is at most three (Guo et al. [62], Kefaloukos et al. [76] and Splatterplots [90]). In particular, every specialized SSV system is focused on very specific visual marks, as indicated by the empty "arbitrary mark types" column. The inflexible nature of these systems makes it hard to extend them to general scenarios.

Kyrix-S, on the other hand, supports all features.

3.9.3 More on Layout Generators

The layout generator in Kyrix-S decides how marks are placed on each zoom level, and serves as the key component to make Kyrix-S scale to big skewed datasets. As discussed briefly in Section 3.2.2, Kyrix-S's layout generator is algorithmically similar in some ways to the ones in existing systems, but also significantly different in many aspects due to the broader design requirements.

Here, we expand the discussion in Section 3.2.2 to elaborate how Kyrix-S's layout generator compares to existing ones.

Offline vs. online. Some systems [62, 51, 97] use an online approach which computes the layout of visual marks on the fly as user's viewport changes. Although being more amenable to updating data, this approach has two inherent drawbacks. First, it is susceptible to data skew as it often needs to iterate through a great portion, if not all of the objects in the viewport. In our examples, there can be tens of millions of objects in a viewport which would make these system unable to respond within 500 ms (**G4-b**). Second, these systems cannot support aggregation-based marks (e.g. pie or bar) because without any precomputed indexes, computing the aggregation statistics requires going through all data in the current viewport. As shown in Table 3.5, none of [62, 51, 97] can show aggregation information. In contrast, Kyrix-S precomputes the mark layouts offline, which makes it possible to support aggregation marks and respond to user interactions under 500 ms despite data skew.

Greedy-based hierarchical sampling/clustering. Kyrix-S's single-node layout algorithm is inspired by prior systems [62, 32, 76, 104]. Specifically, the basic algorithmic flow of iterating over all objects in importance order and then maintaining a minimum distance between objects is drawn from those systems. Nevertheless, Kyrix-S' goes two steps further by 1) generalizing this algorithmic framework to support more usability requirements (last three columns of Table 3.5), and 2) using spatial partitioning techniques to extend the algorithm to support scatterplots of billions of objects (Section 3.6.3).

3.10 Conclusion

In this chapter, we presented the design of Kyrix-S, a system for easy authoring of SSVs at scale. Kyrix-S contributed a declarative language that enabled concise specification of a wide range of SSVs and rapid authoring of custom marks. Behind the scenes, Kyrix-S automatically generated layout of visual marks on zoom levels using a range of usability guidelines such as maintaining a visual density budget and high data abstraction quality. To scale to big skewed datasets, Kyrix-S worked with a multi-node parallel database system to implement the layout algorithm in a distributed setting. Multi-node spatial indexes were built to achieve interactive response times. We demonstrated the expressivity of Kyrix-S with a gallery of example SSVs. Experiments on real and synthetic datasets showed that Kyrix-S scaled to big skewed datasets with billions of objects and reduced the authoring effort significantly compared to general DoD visualization systems such as Kyrix.

Chapter 4

Kyrix-J: Supporting Pivot Jumps between Visualizations for Many Tables in a Relational Database

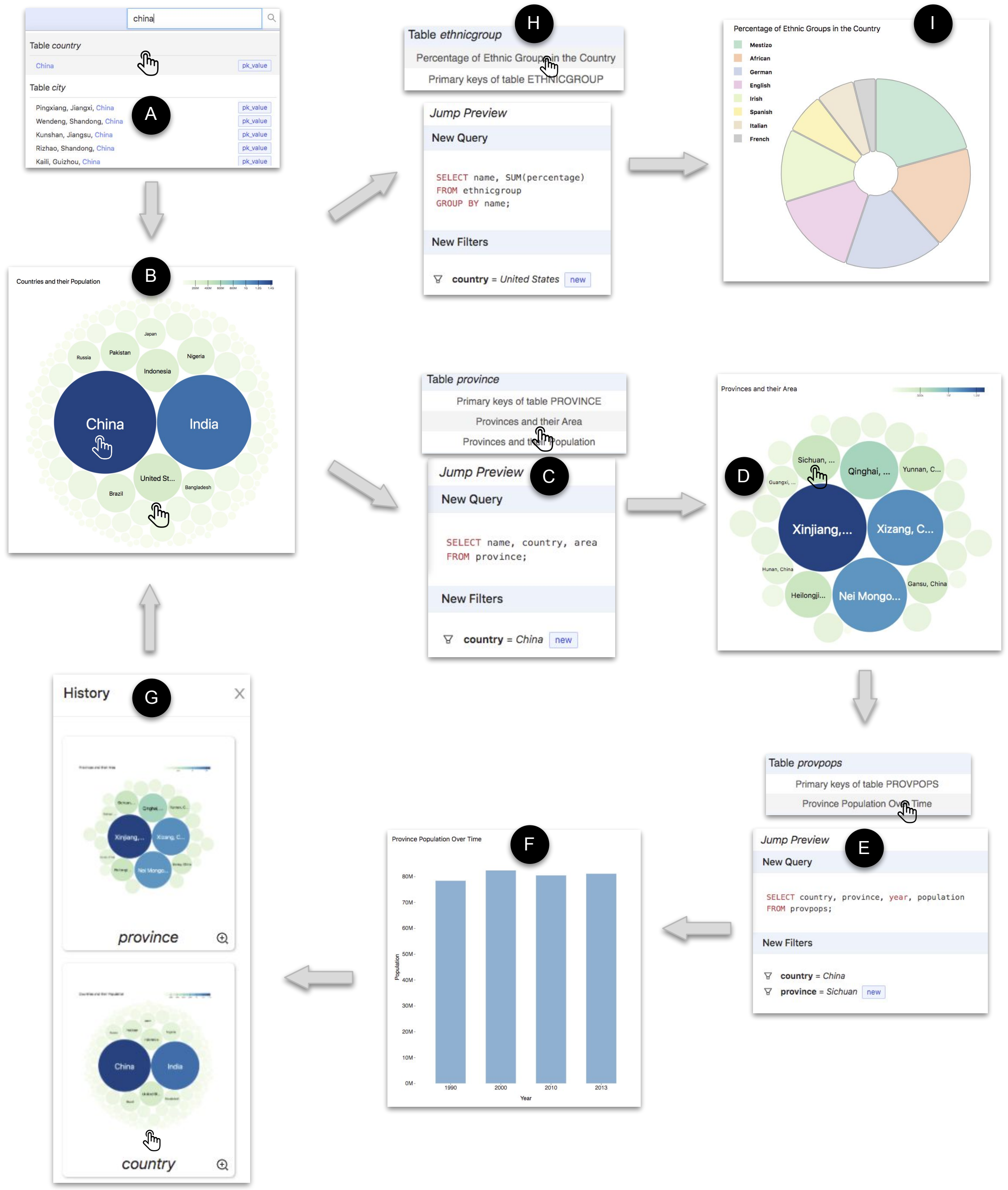
4.1 Introduction

In Chapter 2, we introduced the *jump* connection (Section 2.4.3), which connects two Kyrix canvases with different coordinated systems and visual representations using smooth transitions. It is a powerful operation that allows a user to select an object in a visualization and see a new visualization about this selected object. Essentially, the user "jumps to" the new visualization, whose data is filtered to match the selected object. Consider a series of jumps in Figure 4-1 supported by the system Kyrix-J introduced in this chapter. Figures 4-1b-d describe one jump where a user selects an object (`China`) from a visualization showing countries and jumps to a new visualization showing the provinces of the selected country (`China`) and their areas. In the jump described by Figures 4-1d-f, the user selects the `Sichuan` province and then jumps to a new visualization showing the populations of `Sichuan` over time. These chained jumps allow the user to drill into specifics of the data based on objects of interest, which embody the spirit of the *details on demand* design principle. Jumps are supported in many academic and commercial systems such as GraphTrail [52], PivotPaths [50], Spotfire [18] and Tableau [125], and are often alternatively termed as *pivoting*, *pivot search* or *pivot jump*. In the rest of this chapter, we will use *pivot jump* to refer to a jump which is a more common terminology used in the literature.

While prior systems support users in performing pivot jumps, a common limitation is that they only allow pivot jumps between visualizations for a *single data table* [122, 50, 52, 18, 4, 125]. In practice, data is often scattered in multiple data tables managed by a relational database system (RDBMS) [96, 56]. Enabling pivot jumps in a multi-table RDBMS setting brings two new challenges that existing systems cannot easily address.

First, data attributes across different tables are highly interconnected in an RDBMS, which can lead to many possible pivot jump paths (i.e. pairs of visualizations connected by a pivot jump). Existing pivot jump systems require significant manual effort to label these relationships by either writing code (e.g. Kyrix and [4]) or specification via a visualization interface [18, 52, 96] in order to create visualization filtered on some object. This makes the use of existing pivot jump systems to explore multiple tables tedious and impractical.

Second, it can be overwhelming and disorienting for a user to perform pivot jumps across many tables because these jumps involve joining columns from multiple tables. For example, the pivot jump in Figures 4-1b-d corresponds to a join between the `name` column in the `country` table and the `country` column in the `province` table. Without support, the user can become lost in the exploration process, not knowing where they are and how to locate where they want to go [133]. Existing systems [18, 50, 52, 122] are often hardcoded for a single table and lack the multi-table semantics to help users stay oriented, e.g., to keep track of which table they are visually examining and to understand what the current query and filters are. In this chapter, we present Kyrix-J, a system to support pivot jumps between visualizations for multiple tables in an RDBMS. To free users from manually joining columns to create filtered visualizations, Kyrix-J automatically generates pairwise pivot jump paths between data visualizations based on primary key and foreign key (PK-FK) [13] relationships between data tables.



```
SELECT name, SUM(percentage)
FROM ethnicgroup
GROUP BY name;
```

```
SELECT name, country, area
FROM province;
```

```
SELECT country, province, year, population
FROM provpops;
```

Figure 4-1: A typical usage flow of deployed on the MONDIAL database [89] which involves identifying a table of interest using the keyword search (a), inspecting data visualizations (b, d, f and i), performing pivot jumps between visualizations (b→c→d, d→e→f or b→h→i) and using the history panel to go back to a visited visualization (g).

The pivot jump paths are presented in a popover window after a user clicks on an object (Figure 4-3g), enabling the user to browse and navigate a large number of pivot jump paths in an efficient manner. In one case study we identify more than 1,000 pairwise pivot jump paths between 70 visualizations in an RDBMS with 40 tables, which would be infeasible to explore with tools that require significant manual effort to enable a pivot jump.[1]

To help users stay oriented while performing pivot jumps, Kyrix-J contributes a novel UI (Figure 4-3) with carefully designed components. To start a pivot jump, a user can identify a table of interest using keyword search. We make use of a network graph of tables to signal where the user is, what the neighboring tables are and what the relationships between tables are. Additional panels enable the user to easily inspect information such as the current table, SQL query and filters. We make use of animated transitions in coordinated views to help the user preview the information (e.g. table and query) of the next visualization. Bookmarking and history functionalities are also provided to help the user recall and share exploration history.

We conduct a first-use study with eight users to evaluate the usability of the Kyrix-J UI. Results of the study suggest that participants could perform pivot jumps with Kyrix-J to quickly and accurately complete search tasks in an RDBMS. Participants reported that they were able to stay oriented during their exploration of the RDBMS. They rated Kyrix-J as easy to use and gave positive feedback about the system.

To summarize, we make the following contributions:

- We design an end-to-end system called Kyrix-J to support effective and efficient pivot jumps between visualizations for multiple tables in an RDBMS.[2]

- Kyrix-J automatically generates pairwise pivot jump paths between data visualizations in an RDBMS based on PK-FK relationships between data tables

[1]Kyrix requires manual coding for performing a pivot jump, which is the main motivation behind Kyrix-J. Kyrix-J is built on top of Kyrix to automatically generate Kyrix code for the pivot jump paths, and generates roughly 30,000 lines of Kyrix code (in Javascript) for 1,000+ pairwise pivot jump paths. The name Kyrix-J here suggests that Kyrix-J is built as an extension of Kyrix (rather than a replacement) to mainly support pivot jumps in an RDBMS. The J in the name refers to *jumps*.

[2]The source code for the system can be found at `https://github.com/tracyhenry/kyrix-j`

(Section 4.4). As a result, exploring a large number of pivot jump paths becomes possible.

- We contribute a novel user interface which features the use of coordinated views, animated transitions, bookmarking, text search and other visual aids to help users stay oriented when performing pivot jumps in a multi-table space (Section 4.5).

- We conducted a user study and confirm that Kyrix-J enables effective and efficient pivot jumps in an RDBMS setting with 40 tables, 70 visualizations and more than 1,000 pairwise pivot jump paths between visualizations (Section 4.6).

4.2 Related Works

4.2.1 Pivot Jump Systems

A number of systems [50, 52, 18, 4, 125, 96, 102, 122] have been developed to support pivot jumps. As mentioned in Section 4.1, these systems mostly focus on pivot jumps between visualizations for one data table. As such, *pivoted search* or simply *pivoting* are commonly used terms for describing pivot jumps in those systems. In our multi-table RDBMS setting, we use the term *pivot jumps* to indicate that the interaction "jumps" across the table boundaries. In Section 4.1 we have detailed two common limitations of existing pivot jump systems when applied in an RDBMS setting, namely 1) too much manual effort to support a large number of pivot jump paths and 2) lack of multi-table semantics to help users stay oriented.

Multi-view cross-filter systems such as Snap [96] and Falcon [94] also support our definition of pivot jumps in that a user can select an object in a view and as a result see other views filtered to match the selected object. However, since the underlying coordinations between the views need to be manually configured, they also suffer from limitation 1. Note that Kyrix-J is not a replacement to those systems as they support more types of selections, e.g. brushing and linking in Falcon and synchronized scrolling in Snap, whereas Kyrix-J is limited to click-based selection of a single object.

Supporting more types of selections is part of our future work as we discuss more in Section 4.7.

Kyrix introduced in Chapter 2 also supports pivot jumps in an RDBMS. However, it requires writing tens of lines of Javascript to create a pivot jump path. Also, it does not offer UI support to help users stay oriented. Thus Kyrix has both limitations 1 and 2. Kyrix-J advances Kyrix by automatically generating pivot jump paths based on PK-FK relationships between tables, and offering a novel UI to enable effective RDBMS exploration while performing pivot jumps.

Despite the limitations in these prior systems, they have inspired the design of Kyrix-J in several ways. For example, prior pivot jump systems [52, 146] allow users to easily go back to a visited visualization, which inspires the design of the bookmarking and history functionality in Kyrix-J. As another example, automatic generation of pivot jump paths based on PK-FK is inspired by the categorization of view coordinations in an RDBMS proposed by Snap [96].

4.2.2 Understanding Relationships between Visualizations

Several works [123, 131, 40, 146] tackle the problem of understanding the relationships between data items in multiple visualizations. These systems typically propose new visual representations that explicitly display the relationships between data items in two or more visualizations [35]. While Kyrix-J is related to these systems in that pivot jumps also help users to understand the relationships between data items in a dataset, it has a different focus than these systems. Kyrix-J focuses on supporting "jumping" to a filtered visualization, while the aforementioned systems are concerned with explicitly displaying the connections between data items such as drawing an arrow between related data items.

4.2.3 Data Exploration in an RDBMS

An RDBMS typically offers a command line (e.g. [7]) or a graphical interface (e.g. [5]) that allows users to access the database content using the SQL language. Although proven to be useful, the SQL language has its limitations when used to explore complex datasets. For example, it is often hard to get an overview of the tables and their relationships, leading to slow query construction [138]. Over the years, there have been multiple lines of research efforts in improving data exploration in an RDBMS. Many works (e.g. [138] and [42]) utilize the Entity-Relationship diagram [34], which visualizes the tables and their connections in a graph network, to facilitate RDBMS schema comprehension. The Kyrix-J UI also has a variant of an entity relationship diagram, whose design is inspired by prior works on this topic. Other interesting approaches include allowing users to pose natural language-based queries [44, 82, 117], enabling RDBMS exploration in a computational notebook environment [58] and automatic construction of queries based on example query results [119] (a.k.a *query by example*).

Kyrix-J's approach is to use interactive data visualizations to assist RDBMS data exploration. While there are prior data visualization-based RDBMS exploration systems (e.g. Snap [96], Kyrix and Kyrix-S), Kyrix-J's distinctions lie in its support for effective pivot jumps in a multi-table setting.

4.3 Kyrix-J Overview

In this section, we provide an overview of the Kyrix-J system.

4.3.1 A Usage Scenario

We illustrate the usage of Kyrix-J with a motivating example (Figure 4-1). This example is based on a public relational database called MONDIAL [89], which has 40 tables that correspond to real world entities (e.g. `country`, `lake`, `island`) and their relationships (e.g. `lakeonisland`). Examples in this chapter are all based on this dataset.

To start, a user types the search term `China`, a keyword of interest, into the search box, sees a list of tables matching this keyword and then clicks on the table `country` (Figure 4-1a). The visualization view (Figure 4-3b) then switches to the default visualization for the table `country`, which is a circle pack visualization of countries where the color and size of circles encode the population of the countries (Figure 4-1b).

The user then clicks on the circle with the label `China`, which renders a menu showing a list of pivot jump paths from the current visualization (Figure 4-1c) which are automatically generated by Kyrix-J. The user chooses to perform a pivot jump to the visualization called `Provinces and their Area` for the table `province`. Prior to jumping, another popover window appears upon hovering over the pivot jump path, allowing the user to examine the SQL query and filters of the new visualization (Figure 4-1c).

The target visualization (Figure 4-1d) is again a circle pack visualization showing provinces in China and their area, in which the user is interested in the `Sichuan` province. After clicking on `Sichuan`, the user decides to jump to a new visualization `Province Population Over Time` for the `provpops` table (Figure 4-1e) to see the population of `Sichuan` over time in a bar chart (Figure 4-1f).

Next, the user decides to see something about `United States`. Recalling seeing `United States` in the visualization in Figure 4-1b, the user opens up the history panel (Figure 4-1g) and clicks on the item containing that visualization to go back to Figure 4-1b. From there, the user clicks on `United States` and goes to the visualization `Percentage of Ethnic Groups in the Country` for the `ethnicgroup` table. The new visualization shows what ethnic groups there are in the `United States` and what their percentages are.

In this example, the user has performed two chains of pivot jumps (b $\rightarrow$ c $\rightarrow$ d $\rightarrow$ e $\rightarrow$ f and b $\rightarrow$ h $\rightarrow$ i in Figure 4-1), each of which allows drilling into details of the data tables and seeing different facets of the visualizations. At any point during exploration, the user can utilize the graph view (Figure 4-3c) to inspect what the current table is, what the neighboring tables are and their schema information. The user can also use the informational views (Figures 4-3d-f) to understand the details of the current SQL query and visualization. If the user wants to save a visualization and later shares it with colleagues, they can use the bookmark functionality (Figure

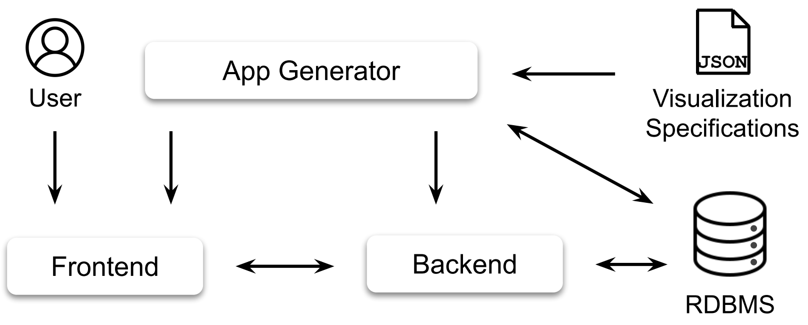


Figure 4-2: The Kyrix-J system architecture.

4-3h).

4.3.2 Design Requirements

The design of Kyrix-J follows an iterative design methodology. We surveyed existing pivot jump visualization systems and visualizations that support RDBMS exploration, and synthesized the findings based on our combined experience in developing database systems and visualization systems. Iterations of the design of Kyrix-J were presented to colleagues, collaborators and students, whose feedback informed the list of design requirements specified below.

R1. Make pivot jumps effortless. In an RDBMS there can be many pivot jump paths between visualizations due to complex relationships between data tables. To enable efficient pivot jumps, we must minimize the human effort involved. This requires that:

- **R1-a**. The pivot jump paths need to be automatically generated since operations to construct filtered visualizations are often tedious and time-consuming.
- **R1-b**. There is an easy way to browse a large number of pivot jump paths from a given visualization.

Theses requirements are also informed by the limitations we see in prior pivot jump systems, which typically require significant manual effort to perform even one pivot jump (see discussions in Sections 4.1 and 4.2).

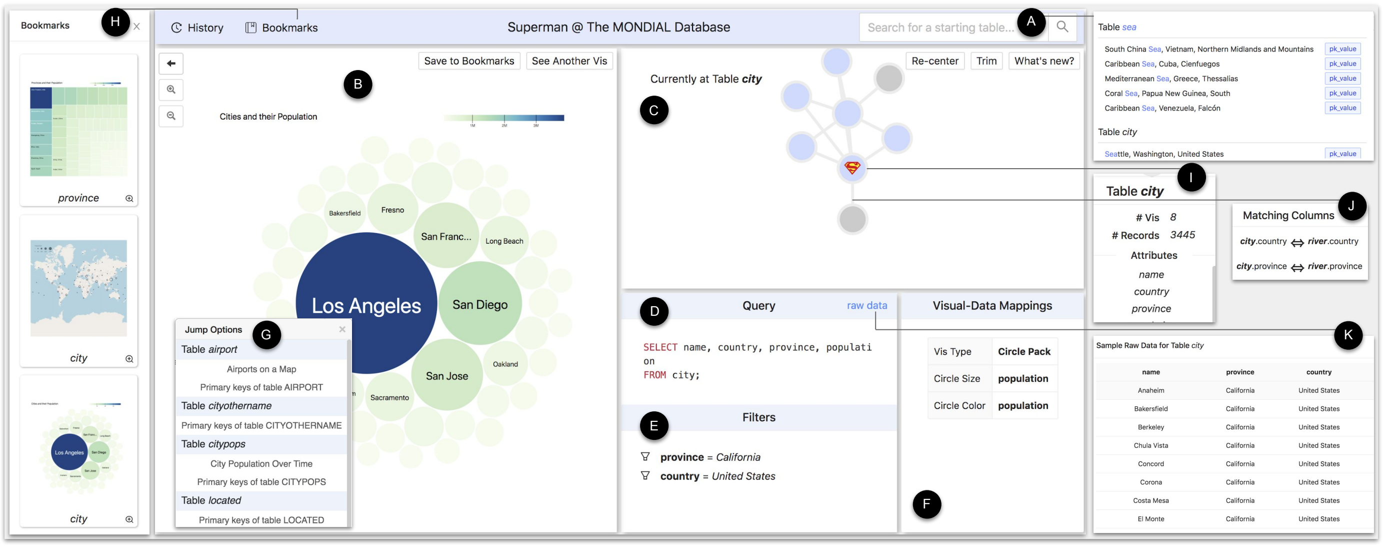


Table *sea*

South China Sea, Vietnam, Northern Midlands and Mountains	pk_value
Caribbean Sea, Cuba, Cienfuegos	pk_value
Mediterranean Sea, Greece, Thessalias	pk_value
Coral Sea, Papua New Guinea, South	pk_value
Caribbean Sea, Venezuela, Falcón	pk_value

Table *city*

Seattle, Washington, United States	pk_value

Table ***city***

Vis *8*
Records *3445*
Attributes
name
country
province

Matching Columns

city.country ⇔ ***river***.country
city.province ⇔ ***river***.province

Jump Options

Table *airport*
Airports on a Map
Primary keys of table AIRPORT
Table *cityothername*
Primary keys of table CITYOTHERNAME
Table *citypops*
City Population Over Time
Primary keys of table CITYPOPS
Table *located*
Primary keys of table LOCATED

Query

```
SELECT name, country, province, populati
on
FROM city;
```

Filters

province = *California*
country = *United States*

Visual-Data Mappings

Vis Type	**Circle Pack**
Circle Size	**population**
Circle Color	**population**

Sample Raw Data for Table *city*

name	province	country
Anaheim	California	United States
Bakersfield	California	United States
Berkeley	California	United States
Chula Vista	California	United States
Concord	California	United States
Corona	California	United States
Costa Mesa	California	United States
El Monte	California	United States

Figure 4-3: The Kyrix-J user interface: (a) the keyword search box which allows a user to identify a table to start their exploration; (b) the visualization view; (c) the graph view where a simplified Entity-Relationship diagram shows each table as a graph node and PK-FK relationships between tables as graph edges; (d-f) informational views showing the current SQL query, filters and mappings from visual properties to data attributes; (g) a popover appearing after the user clicks on an object, which contains a list of new visualizations to "jump" to. These pivot jump paths between pairs of visualizations are automatically generated by Kyrix-J based on PK-FK links; (h) the list of bookmarked visualizations; (i-j) when hovering over a node/edge in the graph view, more information shows up in a popover; (k) the user clicks on the "raw data" button to see the data items in a tabular format.

R2. Help users stay oriented. Disorientation is a common problem in interactive interfaces where a user needs to find a path to navigate through (e.g. a complex website) [133]. In a pivot jump setting where each step involves joining columns from two tables, it is easy for the user to get lost. We need to provide navigation support to help the user build location and analysis-context awareness which has the potential to boost both performance and engagement [133, 98]. More specifically,

- **R2-a**. Enable common user task during RDBMS exploration such as finding a starting table, understanding the schema of the tables, the current query and relationships between tables, etc.

- **R2-b**. Help the user keep track of which table they are currently examining, which table they come from and which table they will go to.

- **R2-c**. Support quick navigation to a previously visited visualization. This is a common requirement adopted by many prior systems that allow the user to navigate between multiple visualizations [139, 52, 146].

In the rest of the chapter, when describing the design of Kyrix-J, we will refer back to this list of requirements.

4.3.3 System Architecture

The architecture of Kyrix-J is illustrated in Figure 4-2. There are five key components involved: the RDBMS, visualization specifications, the app generator, the backend and the frontend.

RDBMS. In our prototype, we use PostgreSQL [6] as the RDBMS.

Visualization Specifications. Kyrix-J accepts as input a set of visualizations specified in a JSON-based visualization language offered by Kyrix, which can support bar charts, pie charts, treemaps, circle packs and word clouds.[3] We also support scatter-

[3]For the documentation for this language, see `https://github.com/tracyhenry/Kyrix/wiki/Template-API-Reference#static-aggregations`

plots using the JSON-based language offered by Kyrix-S.[4] Additional visualizations can be authored and added to Kyrix-J using the JSON-based authoring interface. In Section 4.7 we describe more about our future plans to better support authoring visualizations in Kyrix-J.

App Generator. The app generator takes in two pieces of information as input: 1) the RDBMS schema (e.g. table/column names and PK-FK relationships) and 2) the set of visualization specifications. The output of the app generator contains two parts: 1) a Kyrix application specification containing auto-generated definitions of pivot jumps which is used by the backend to build necessary database indexes and 2) JSON-based metadata about the application (e.g. mappings from pivot jump paths to pairs of tables, database schema, etc), which is used by the frontend to generate the UI and the visualizations.

Backend. The backend extends the Kyrix backend with one added capability: to receive keyword queries from the client and return the database content (tables, columns, primary keys) that matches the keyword(s). The backend talks to the PostgreSQL RDBMS and uses the Generalized Inverted Indexes (GIN) [9] offered by PostgreSQL to support fast keyword search queries. The backend is also responsible for parsing the application specification generated by the app generator and building necessary database indexes to support efficient data serving for the frontend.

Frontend. The frontend is the user interface and is responsible for interacting with the backend to fetch data as a user performs pivot jumps. It is currently implemented as a React JS [10] application.

4.3.4 Organization

The rest of the chapter is organized as follows. Section 4.4 shows how the app generator automatically generates pivot jump paths. Section 4.5 presents the design of the Kyrix-J UI. In Section 4.6 we describe a user study we conduct to evaluate the

[4]For our prototype we support these six types of visualizations. In the future we will consider integrating with established visualization libraries such as Vega-lite [112] to support more visualization types.

usability of Kyrix-J. Section 4.7 discusses the limitations of the current system and future work. Section 4.8 concludes the paper.

4.4 Automatic Generation of Pivot Jump Paths

To free users from manually creating filtered visualizations, we automatically generate pivot jump paths (**R1-a**). In this section, we describe how the app generator achieves this goal. We first formally define the problem in Section 4.4.1 and then describe in Section 4.4.2 a solution that utilizes the PK-FK relationships in the RDBMS.

In the rest of this chapter, we will use path, jump path and pivot jump path interchangeably for simplicity.

4.4.1 Problem Definition

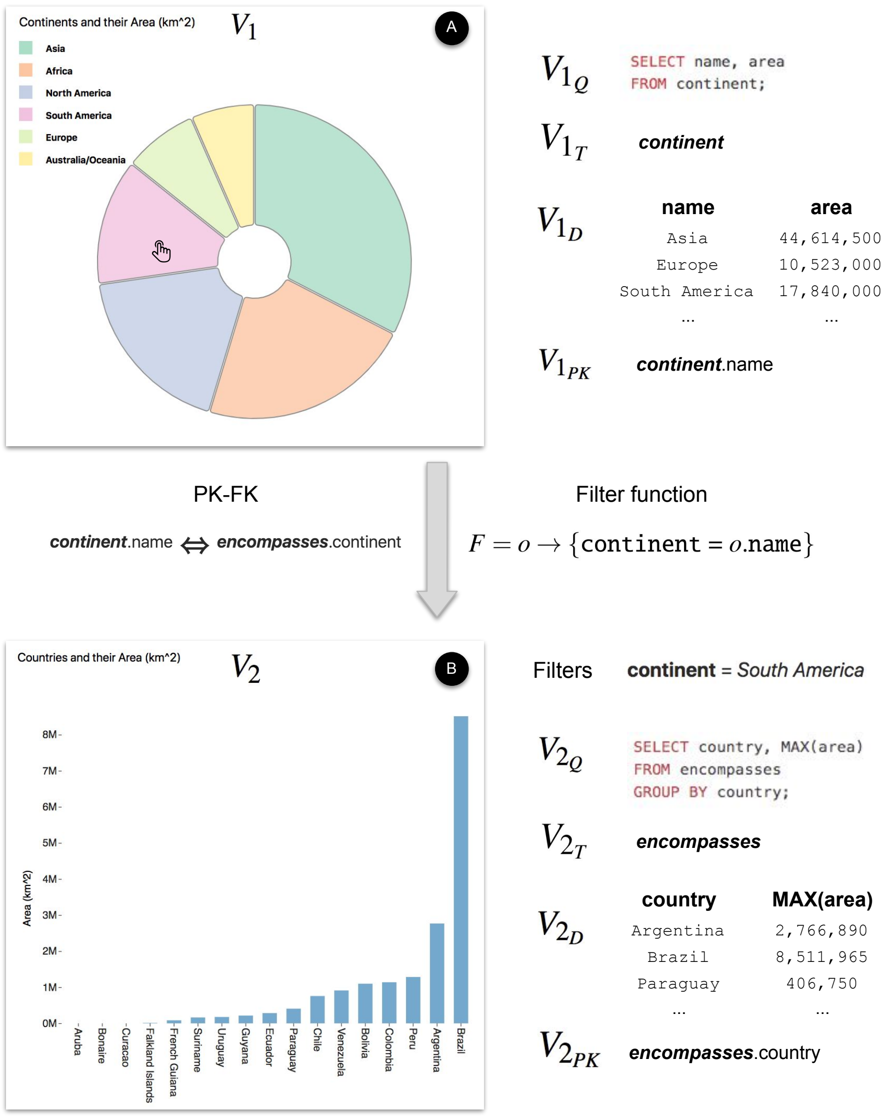
Here, we use the *auto jump problem* to refer to the problem of automatically identifying jump paths between visualizations for multiple tables in an RDBMS. To formally define the auto jump problem, we first introduce a few definitions.

Definition 1 (Visualizations)**.** *A visualization V is a tuple (V_Q, V_D, V_T, V_{PK}). V_Q is the SQL query used to fetch the data items for V from the RDBMS. V_D represents the query results of V_Q. V_T denotes the table used in V_Q. The primary key of a visualization V, denoted as V_{PK}, is the set of data fields in V_D which uniquely identify each object in V.*

To compute V_{PK}, i.e., the primary key for each visualization V, we use V_Q, i.e., the SQL query of V. If V_Q is a SQL `GROUP BY` aggregation query, we use the `GROUP BY` columns in V_Q as V_{PK}; otherwise we use the primary key of V_T as V_{PK}.[5]

We illustrate Definition 1 with Figure 4-4a. V_1 is a pie chart showing continents and their area. V_{1_Q} is a simple `SELECT` query, therefore $V_{1_{PK}}$ is the primary key of

[5]We make two assumptions to simplify the presentation. First, each RDBMS table has exactly one primary key. Our techniques easily extend to the case where a table has multiple primary keys. Second, the SQL query for a visualization is either a `GROUP BY` query, or simply a `SELECT` query which selects some fields from a table. If a SQL query is too complex for this assumption, we assume that an expert (e.g. a database administrator) materializes the query result into a table and rewrites the SQL query to fit this assumption.



V_{1_Q}

```
SELECT name, area
FROM continent;
```

V_{1_T} ***continent***

V_{1_D}

name	area
Asia	44,614,500
Europe	10,523,000
South America	17,840,000
...	...

$V_{1_{PK}}$ ***continent***.name

continent.name ⇔ ***encompasses***.continent

$F = o \rightarrow \{\texttt{continent} = o.\texttt{name}\}$

Filters **continent** = *South America*

V_{2_Q}

```
SELECT country, MAX(area)
FROM encompasses
GROUP BY country;
```

V_{2_T} ***encompasses***

V_{2_D}

country	MAX(area)
Argentina	2,766,890
Brazil	8,511,965
Paraguay	406,750
...	...

$V_{2_{PK}}$ ***encompasses***.country

Figure 4-4: An example pivot jump path with annotations of the SQL queries, data items and primary keys of the visualizations, the filter function and the PK-FK relationship used to identify this path.

V_{1_T}, i.e., the `name` column in table `continent`. This suggests that the each object (pie) in V_1 is a continent.

As another example, consider V_2 in Figure 4-4b which is a bar chart showing countries in South America and their area. V_{2_T} is a table called `encompasses` where each data item denotes one continent encompassing a country and has a corresponding attribute `area` indicating the area of the country. V_{2_Q} is a SQL `GROUP BY` query which groups all data items by country and also computes the area of the country using the `MAX` aggregate.[6] Thus $V_{2_{PK}}$ is the `GROUP BY` columns of V_{2_Q}, i.e., the `country` column in table `encompasses`. This suggests that each object (bar) in V_2 is a country.

Next, we define a filter function.

Definition 2 (Filter Functions). *A filter function F takes a visual object o as input, and outputs a set of filters. Each filter is in the form $b = o.a$ where b is one column and $o.a$ is the value of the a field of o.*

A pivot jump path is then defined as follows.

Definition 3 (Pivot Jump Paths). *A jump path from V_1 to V_2 is a tuple (V_1, V_2, F) that satisfies the following: 1) V_1 and V_2 are visualizations; 2) F is a filter function; and 3) the output of F when applied on an object in V_1 satisfies the following: a) there are $|V_{1_{PK}}|$ filters in total; b) each column in $V_{1_{PK}}$ appears on the right-hand side of exactly one filter; and c) the left-hand side of each filter is a column in V_{2_T}.*

In other words, requirement 3) says that the resulting filters of F applied on an object in V_1 form a one-to-one mapping from columns in $V_{1_{PK}}$ to a subset of columns in V_{2_T}.

Consider the example path in Figure 4-4 where a user selects `South America` in the pie chart V_1 and jumps to V_2 to see countries in `South America` and their area. The filter function returns `continent` $= o.$`name` when applied on an object o in V_1, which forms a one-to-one mapping from the only column in $V_{1_{PK}}$ (the `name` column in table `continent`) to one column in V_{2_T} (the `continent` column in table

[6]Data items in the same group have the same area. The `MAX` aggregate is only used to get that number, and could be replaced by `MIN` or `AVG`.

`encompasses`). When the user selects the object `South America` to start the pivot jump, the filter function returns the filter `continent` = `South America`.

While Definition 3 is useful in characterizing a jump path, note that there could be many possible filter functions that map columns in $V_{1_{PK}}$ to columns in V_{2_T}. Not every mapping is useful. Therefore, we need to define what constitutes a meaningful jump path.

Definition 4 (Meaningful Pivot Jump Paths)**.** *We say that a jump path* (V_1, V_2, F) *is meaningful, if and only if any filter generated by* F *connects two columns from* V_1 *and* V_2 *that represent the same type of object (referred to as matching columns subsequently).*

For example, the filter `continent` $=$ o.`name` makes the pivot jump path in Figure 4-4 meaningful because both two columns represent continents. Yet if the filter function is changed to return `country` $=$ o.`name`, the path would not be meaningful.

With Definitions 1-4, we can now define the auto jump problem.

Definition 5 (The Auto Jump Problem)**.** *Given a set of visualizations* $\mathbb{V}$*, identify all meaningful pivot jump path* (V_1, V_2, F) *such that* $V_1 \in \mathbb{V}$ *and* $V_2 \in \mathbb{V}$*.*

4.4.2 A Solution based on RDBMS PK-FK Relationships

Given two visualizations V_1 and V_2, the key in identifying meaningful jump paths from V_1 to V_2 is to look for matching columns between $V_{1_{PK}}$ and V_{2_T}. With tools that require manual creation of the jump paths (e.g. [52, 18]), users typically look for matching columns using common sense and constructs the filters manually.

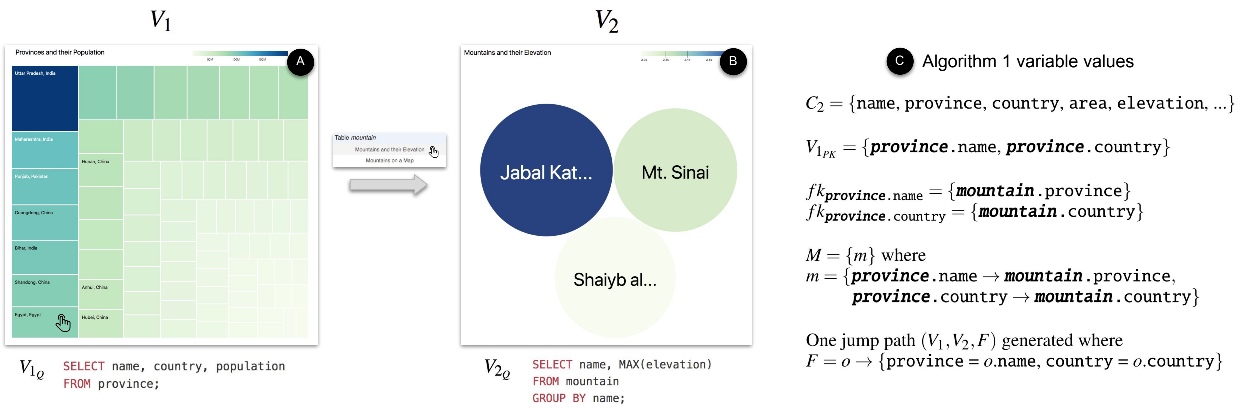
Our idea is based on the fact that matching columns are readily available in an RDBMS in the form of PK-FK relationships between tables [58, 96], which can be used to automatically identify meaningful jump paths. Consider again the jump path in Figure 4-4. the `continent` column in the table `encompasses` is a foreign key referencing the primary key column `name` in the table `continent`, which is the only column in $V_{1_{PK}}$. Therefore, we can match these two columns up to get the filter function that satisfies the requirements in Definition 3.

Algorithm 1: Automatic Generation of Meaningful Jump Paths

Input: A set of visualizations $\mathbb{V}$ and the PK-FK relationships in the RDBMS.

Output: A set of meaningful pivot jump paths between visualizations in $\mathbb{V}$.

```
1  $P \leftarrow \varnothing$;
2  for each $V_1 \in \mathbb{V}$ do
3      for each $V_2 \in \mathbb{V}$, $V_2 \neq V_1$ do
4          $C_2 \leftarrow$ all columns in $V_{2_T}$;
5          for each $c \in V_{1_{PK}}$ do
6              $fk_c \leftarrow$ the set of foreign keys in $C_2$ that reference $c$;
7          $M \leftarrow \{m \mid m : V_{1_{PK}} \rightarrow C_2$
8                      and $\forall c \in V_{1_{PK}}, m(c) \in fk_c$
9                      and $\forall c_x, c_y \in V_{1_{PK}}, c_x \neq c_y \Leftrightarrow m(c_x) \neq m(c_y)\}$;
10         for each $m \in M$ do
11             $F = o \rightarrow \{m(c) = o.c \mid c \in V_{1_{PK}}\}$;
12             Insert $(V_1, V_2, F)$ into $P$;
13 return $P$;
```



V_{1_Q}
```
SELECT name, country, population
FROM province;
```

V_{2_Q}
```
SELECT name, MAX(elevation)
FROM mountain
GROUP BY name;
```

$C_2 = \{$`name`, `province`, `country`, `area`, `elevation`, ...$\}$

$V_{1_{PK}} = \{$**`province`**`.name`, **`province`**`.country`$\}$

$fk_{\texttt{province.name}} = \{$**`mountain`**`.province`$\}$
$fk_{\texttt{province.country}} = \{$**`mountain`**`.country`$\}$

$M = \{m\}$ where
$m = \{$**`province`**`.name` $\rightarrow$ **`mountain`**`.province`,
`province``.country` $\rightarrow$ **`mountain`**`.country`$\}$

One jump path (V_1, V_2, F) generated where
$F = o \rightarrow \{$province = o.name, country = o.country$\}$

Figure 4-5: Another example jump path with annotations on the right to illustrate Algorithm 1.

Using this idea, our solution is presented as pseudo code in Algorithm 1. The high-level strategy is to search for meaningful jump paths between every possible pair of visualizations (Lines 2-3). For each pair of visualizations V_1 and V_2, we first identify for each column c in $V_{1_{PK}}$, all foreign keys in V_{2_T} that reference it (Lines 5-6).[7] Next, we use this information to get all mappings from $V_{1_{PK}}$ to columns in V_{2_T} that connect columns in $V_{1_{PK}}$ to distinct foreign keys in V_{2_T} (Line 9). Each of these mappings corresponds to one filter function and thus one pivot jump path, which we then add to a global jump path array (Lines 10-12).

We illustrate this solution using another example jump path in Figure 4-5. A user selects `egypt` from V_1, a visualizations showing the largest provinces in the world (Figure 4-5a), and jumps to V_2, a visualization showing the mountains in `egypt` and their elevation (Figure 4-5b). Figure 4-5c shows what value each variable in Lines 4-12 holds for this given pair of V_1 and V_2. There are two columns in $V_{1_{PK}}$, which respectively match one foreign key in V_{2_T}. This leads to one match in M, which results in one jump path.

Complexity Analysis. The average time complexity of Algorithm 1 is $O(|\mathbb{V}|^2 \cdot |\bar{M}|)$ where $|\bar{M}|$ is the average size of the variable M on Line 9 of Algorithm 1, which holds all possible matches from $V_{1_{PK}}$ to columns in V_{2_T}. In practice M should have a small size since it is not common to have multiple foreign keys for the same column. Therefore this algorithm should run reasonably fast (e.g. finishes within a few minutes) on a single machine on tens of thousands of visualizations.[8] In the case where new visualizations are added to the system dynamically, the time complexity of identifying the new jump paths from/to a new visualization is $O(|\mathbb{V}| \cdot |\bar{M}|)$.

Discussions. Although PK-FK relationships can help identify meaningful jump paths, they only represent a subset of all pairs of matching columns. However, Algorithm 1 can take in any input set of matching columns (e.g. those generated by a data discovery system [58]) and therefore is not limited to PK-FK relationships. Nevertheless, we expect that PK-FK relationships can generate a large number of

[7]To enable pivot jumps between visualizations for the same table, i.e., when $V_{1_T} = V_{2_T}$, we see each column in $V_{1_{PK}}$ as a foreign key of itself. Therefore in Line 6, each fk_c is simply $\{c\}$.

[8]For the example application with 70 visualizations, the algorithm finishes in a few seconds.

paths that are sufficient for initial RDBMS exploration uses. For example, we have identified over 1,000 jump paths between 70 visualizations for 40 tables in the example MONDIAL database.

4.5 User Interface

The high-level goal of the Kyrix-J UI is to present the jump paths generated by the algorithm in Section 4.4 using intuitive UI elements that users can understand and use. Specifically, we should make it easy to browse a large number of such jump paths (**R1-b**) and also help users stay oriented during their exploration with pivot jumps (**R2**).

Figure 4-3 shows an overview of the UI, which features multiple coordinated views. In this section we describe the design of each UI element in detail.

4.5.1 The Keyword Search Box

The search box (Figure 4-3a) enables a user to identify a starting table for exploration (**R2-a**) using keywords of interests. The rationale for this choice comes from prior works on using keywords to search through RDBMS content, which usually assume that even if the user is not familiar with the database schema (e.g. names of the tables and columns), they generally have something of interest expressible as keywords [58, 86, 72].

There are three types of search results: a table name match, a column name match and a primary key value match, which are indicated using a tag to the right of each result. The search results are grouped by tables. Clicking on a group brings the user to the table and updates several other views, including showing a default visualization in the visualization view (Figure 4-3b) and updating the graph view (Figure 4-3c) to showing the table and its direct neighbors.

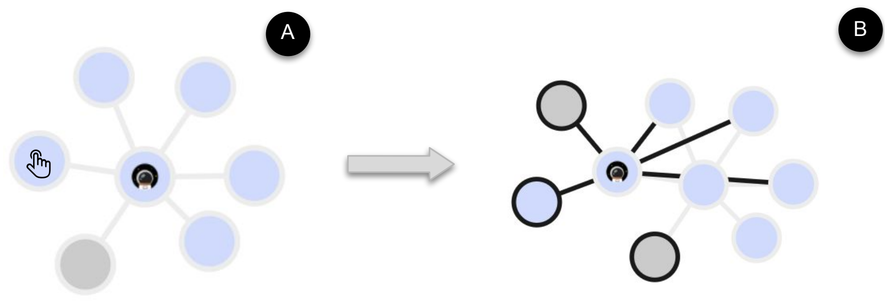


Figure 4-6: Expanding the graph view: a) one can click on a node/table to go to that table; b) an animation highlighting new nodes and edges added to the graph.

4.5.2 The Graph View

To support RDBMS exploration tasks such as inspecting table schema and relationships between tables (**R2-a**), we present a variant of the Entity-Relationship (ER) diagram [34] where each node is a table and each edge represents a set of PK-FK relationships between two tables (Figure 4-3c). We use an astronaut icon[9] to indicate which table the user is currently examining (**R2-b**). In the top left-hand corner there is also a text saying what the current table is.

This graph view offers a rich set of interactions that enable useful exploration of the RDBMS by itself. The user can click on a node/table and go to that table (Figure 4-6a). Any new direct neighbors of the clicked table and corresponding edges are added to the graph and highlighted with an animation (Figure 4-6b). The user can click on the *What's new* button to replay the highlight animation. To reduce the density of the graph, the user can hit the *Trim* button to see only the current table and its direct neighbors. The user can also pan the graph and use the *Re-center* button to bring the current table back to the center of the graph view.

Note that in a traditional ER diagram[10], data columns and PK-FK relationships are always visible. Yet this generates visual clutter when there are tables with many columns or too many PK-FK relationships. While many have worked on addressing

[9]We use a galaxy setting for our UI where the user explores a database as an astronaut jumping between planets that represent tables.

[10]The ER diagram of the example MONDIAL database is available here: `https://www.dbis.informatik.uni-goettingen.de/Mondial/mondial-ER.pdf`

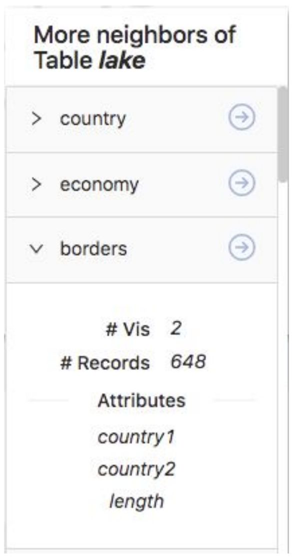


Figure 4-7: Hovering over a meta node.

this issue of ER diagrams (e.g. [129, 118, 141]), we take the simple approach of only revealing the columns or PK-FKs upon hovering a node or edge (Figures 4-3i and 4-3j).

The rationale is that understanding schema information is a relatively secondary task and can be completed on demand. To cope with "wild datasets" where a table has many neighbors, we implement a strategy showing only a sample of neighbors at the beginning[11], and use a *meta* node in grey to represent the rest of the neighbors. The user can hover over a meta node and see the rest of the neighbors in collapsed panels (Figure 4-7). Each panel can be expanded which reveals the detailed information about a table. Clicking on the *go* icon on the right brings the user to that neighboring table, similar to clicking on a normal node. After that, the meta node becomes a normal node representing the clicked neighbor and a new meta node is generated for the last table.

[11]There are numerous ways to choose which tables to sample, e.g., based on popularity, PK-FK connections a table has, etc.

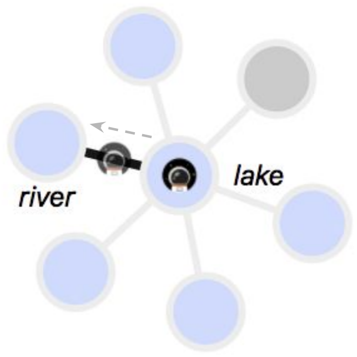


Figure 4-8: When hovering over a jump path, an animated astronaut icon signals the move from one table to the next.

4.5.3 The Visualization View

The visualization view (Figure 4-3b) presents one visualization at a time. We reuse some functionalities from the Kyrix frontend including rendering the visualizations and navigation buttons (top left-hand corner in Figure 4-3b), but also add several new functionalities.

First, we group the jump paths based on the table (V_T) of the target visualization V (Figure 4-3g). This enables quick browsing of similar jump paths on the same data (**R1-a)**. The table names are sorted alphabetically to make it easy to locate a destination table based on its name.

Second, when hovering over a jump path, a popover shows up to help the user preview the query of the next visualization and the generated filters (Figures 4-1c, 4-1e and 4-1h) which facilitates the understanding of the queries and relationships between tables (**R2-a**). Further, in the graph view, we show an animation of a semi-transparent astronaut icon moving from the current table to the table of the target visualization (Figure 4-8) to help the user be aware of which table they will go to (**R2-b**). Labels of the table names are also shown at this time.

Third, when the user decides to take a jump path, we show an animation of a flying rocket (Figure 4-9) before switching to the next visualization to help the user stay aware of an ongoing jump (**R2-b**). At the same time, there is also a animation

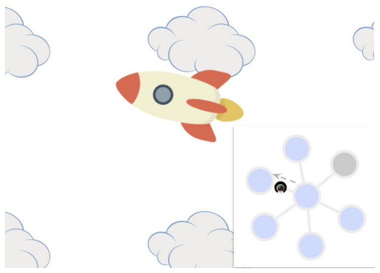


Figure 4-9: The visualization view uses an animated rocket to indicate a jump to the next visualization, which is coordinated with the astronaut icon movement in the graph view.

of the astronaut icon in the graph view moving along the corresponding edge. We coordinate these two animations so that the rocket flies in the same direction as the astronaut icon moves, to help the user achieve a stronger sense of the movement from one table to another. After the animation ends, the astronaut icon ends up at the new table. New direct neighbors and corresponding edges are added if not exist, similar to clicking on a node to expand the graph.

Lastly, the user can hover over the *See Another Vis* button to see a list of all visualizations for the current table (Figure 4-10). Clicking on a visualization brings the user to that visualization and updates other views accordingly. This enables quick switch to visualizations on the same data without going through pivot jumps.

4.5.4 Informational Views

The informational views display relevant information about the current state (**R2-a**), including the current query (Figure 4-3d), the current filters (Figure 4-3e) and mappings from data columns to visual properties (Figure 4-3f).

In the query view, the user can click on the *raw data* button to see a sample of raw data items in a tabular format. We separate filters from the query because filters help signal what objects have triggered a pivot jump, therefore helping keep track of the user's location (**R2-b**). For example, in Figure 4-3, the filter `province`

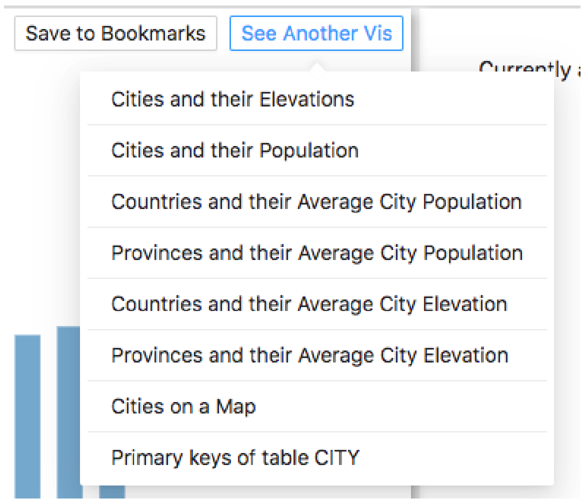

Save to Bookmarks
See Another Vis
Currently
Cities and their Elevations
Cities and their Population
Countries and their Average City Population
Provinces and their Average City Population
Countries and their Average City Elevation
Provinces and their Average City Elevation
Cities on a Map
Primary keys of table CITY

Figure 4-10: Hovering over *See Another Vis* reveals a list of all visualization for the current table.

`= California` suggests that the user selected the object `California` to trigger the last pivot jump.

4.5.5 Bookmarking and History

To support going back to a previously visited visualization (**R2-c**), Kyrix-J allows the user to bookmark a visualization by clicking on the *Save to Bookmarks* button in the visualization view. All bookmarks can be browsed in the bookmark tab (Figure 4-3h). Each bookmark item contains a thumbnail of the visualization, as well as the corresponding table name at the bottom which serves as the "address" of the bookmark in the data space. Clicking on the thumbnail brings the user back to the bookmarked visualization. Clicking on the magnifier icon in the bottom right reveals a full screenshot of the bookmarked visualization which the user can download, edit and share with others.

The history tab (Figure 4-1g) logs all visualizations visited. Each item works similarly as a bookmark item. For simplicity we only show in our prototype a linear history where visualizations are sorted by the order in which they are visited. In the future we will investigate more complicated mechanisms (e.g. support branching [70]).

4.6 Evaluation: A First-use Study

We conducted an observational first-use study to evaluate the usability of Kyrix-J. Specifically, we want to answer the following questions through the study: 1) Can Kyrix-J help users perform pivot jumps in an efficient and accurate manner? 2) How much can Kyrix-J help users stay oriented during exploration? 3) How helpful is each UI element? 4) What are possible improvements for the current system?

We recruited eight participants (5 females, 3 males, age range 21-55) through posting ads in online forum and mailing lists. Participants reported a diverse background in database and visualization technologies. Six participants reported more than one year of experience with RDBMS related tools while four reported more than one year of experience with data visualization. The Kyrix-J application used in this study was based on the MONDIAL database (which is the example RDBMS used throughout the chapter) and had 40 tables, 70 visualizations and over 1,000 pairwise jump paths between visualizations.

4.6.1 Procedure and Tasks

At the start of each study session, we asked the participant to fill out a consent form. We then gave the participant a tutorial on Kyrix-J and demonstrated how to use Kyrix-J to complete simple search tasks. The participant then went through a training phase where they were asked to complete two example search tasks to gain familiarity with the UI and the data. The participant was also encouraged to spend as much time with the system and ask as many questions as they need. In the main experiment we asked the participant to complete five search tasks (Table 4.1) using Kyrix-J and recorded the time taken and the accuracy. During task completion, the participants were allowed to ask the experimenter general questions about the UI and the dataset. After the completion of the tasks, we collected free-form feedback from the participant in a semi-structured interview. The study concluded with the participants filling out a post-study questionnaire where they rated how much they agree with statements about the usability of Kyrix-J on a 5-point Likert scale (1 –

Table 4.1: Search tasks used in the user study.

Task 1	Is the following statement true? The 3rd largest ethnic group in the 4th largest country in the South America continent is European.
Task 2	a: Find the 4th most populated country in the world. b: Find the most populated city in that country. c: Find one organization headquartered at that city.
Task 3	Find the population (in 2011) of the 3rd most populated city in the most populated province in the world.
Task 4	Is the following statement true? Ukerewe is an island in the 3rd largest lake in the world.
Task 5	a: Find the country with the 2nd largest average city elevation. b: Find the 2nd most spoken language in that country.

strongly disagree and 5 – strongly agree).

Each session was conducted on a video conferencing platform and lasted roughly one hour. We asked the participant to share their screen when using Kyrix-J to complete the search tasks. With the consent from the participants, we recorded the session including all the interviews for post-study analysis. Each participant was compensated $20 for their time.

The tasks in Table 4.1 were designed so that the participants could start from a visualization and perform one or more pivot jumps to another visualization containing the answer. We varied the descriptions of the tasks (checking if a statement is true/false, one-sentence question and subtask-based question) to account for potential human bias in understanding the tasks.

4.6.2 Results and Discussions

Task completion. All participants were able to complete all tasks with minimal guidance. The completion times for the tasks are shown in Table 4.2. The average total completion time for all tasks was 11.4 minutes. It is worth noting that Task 5 took longer to complete because we designed the task so that it was not straightforward to search for the starting visualization, which required using the *See Another*

Vis button (Figure 4-10).

Completions of the tasks were mostly accurate. Only two participants got one task wrong on the first try, which was due to inaccurate numerical comparison (e.g. mistaking the 4th lengthiest bar for the 3rd) and misunderstanding of the task. Both were able to quickly correct themselves after realizing the mistakes.

Table 4.2: Average task completion times and standard deviations (in minutes).

	Task 1	Task 2	Task 3	Task 4	Task 5
μ	2.2	1.7	1.8	1.7	4.1
σ	0.6	0.7	0.6	1.0	2.0

Ease of use and learning. In the questionnaire, participants rated that Kyrix-J was easy to use ($\mu = 4.63, \sigma = 0.52$) and easy to learn ($\mu = 4.75, \sigma = 0.46$). Many participants gave positive comments on the overall experience: *"it's very natural to jump from thing to thing* (P1)*","this (Kyrix-J) helps me figure out what tables are related to each other, what are the PK-FK relationships, where am I in filtering* (P2)*","the jumps help data discovery* (P7)*","I want to use this tool for my personal use if I can* (P8)*"*. Participants especially liked the ability to browse a lot of jump paths: *"I like the options for drill down (jumps), you can have multiple options so you are not limited* (P6)*","it helps showcase the possible ways you can filter different tables* (P7)*","the pace at which I can quickly jump from one visualization to another is amazing* (P8)*"*.

Helping users stay oriented. In the questionnaire, participants rated that they could effortlessly navigate the database content and be aware of their current location in the space ($\mu = 4.25, \sigma = 0.46$). When asked how much they were able to "stay oriented" and keep track of things like the current table and query, participants reported that *"I have a pretty good feel for it* (P1)*","it was pretty easy to keep track of where I was* (P3)*","I was able to do that (stay oriented) a lot* (P4)*","fairly well* (P6)*"*. They commented that they could understand what table they were at by looking at the SQL query (P5) and where they jumped from/to by looking at the filters (P3 and P5). On the other hand, one participant (P7) felt that *"after you spend sometime digging into the visualizations, it's more difficult to remember exactly where you came from."*.

In a post-study interview, P7 and the experimenter discussed possible improvements such as visualizing the current jump chain in the graph view using several astronaut icons with decreasing opacity. We will investigate these possible designs in future work.

The keyword search box. All participants rated the keyword search functionality to be helpful ($\mu = 4.88, \sigma = 0.35$) and gave many positive comments: *"I really like being able to search for column names and keys* (P3)*","search and vis work really well together* (P8)*","I found it really easy to search for my starting point* (P8)*".* One observation was that the participants used the search box more often than alternatives (e.g. the graph) to find a starting visualization or table. The reason could be that the search functionality was similar to using a web search engine which people nowadays are very familiar with.

The graph view. The Likert score on the helpfulness of the graph view is relatively high ($\mu = 4.00, \sigma = 1.60$). Most participants found the graph view to be useful, e.g., *"graph is easy to pan and see three or four jumps, it might get too busy, but still pretty easy to follow* (P3)*".* However, two participants (P1 and P5) expressed that they did not find the graph to be helpful. They explained that this was in part because the search tasks are specific enough to be accomplished without much interaction with the graph view. As P7 pointed out, *"if I'd do more free form exploration the graph would be more useful".*

The informational views. In the questionnaire, participants rated that the informational views (i.e. query, filters and visual-data mappings) were helpful ($\mu = 4.25, \sigma = 0.71$). One participant (P6) who is an experienced RDBMS user commented that *"The fact that the query is there and the filter is there is helpful, especially since I'm used to SQL, so it feels like home".* P1 also said that *"The query box is nice to see what's going on".* Others noted that *"the information presented is straightforward and can be something that is toggled on* (P6)*".* P7 suggested that *"query and filters view might not be super helpful for free explorations, but useful if my end goal is to write a query".* Based on these feedback, we can offer the option to toggle off the informational views as a possible improvement.

The animations. Participants had diverging opinions on the helpfulness of the animations in the UI as indicated by the Likert score ($\mu = 3.25, \sigma = 1.66$). P3 commented *"animations might not be helpful if I know the schema very well"*. P6 suggested that the rocket animation *"needed to be shorter"*. Others who liked the animations noted that they enjoyed the interactivity the animations brought. As an improvement, we can make the animations and their duration configurable and shorter by default.

Bookmarks and history. Participants feedback on the helpfulness of the bookmarking and history functions are mostly positive ($\mu = 3.63, \sigma = 1.06$ for bookmarks, and $\mu = 4.00, \sigma = 0.93$ for history). Although the Likert scores are not very high, it was largely due to the fact that it was hard to justify the need to recall history in a short study session. As P7 noted, *"If I spend a lot of time with the tool, the bookmark will be helpful"*.

Comparison with alternative techniques. Not all participants had prior experiences with pivot jumps, but they still provided insights on what alternative techniques they might use to complete similar tasks. Many participants mentioned using the SQL command-line interface offered by the RDBMS as an alternative. However, they also pointed out using a graphical interface like Kyrix-J would be easier and more intuitive: *"that (using a SQL command-line interface) would require knowing the schema a lot better. I need to pay a lot more attention* (P1)*","it (Kyrix-J) automates all the painful coding you have to do* (P2)*","I don't have to write 10 to 20 lines of code, everything (in Kyrix-J) is a click away. It's like shopping on Amazon* (P4)*"*.

One participant (P6) had used a variety of tools (e.g. Tableau [125], Chartio [1] and SSIS [12]) to do pivot jumps in professional settings. She noted that a common limitation of the tools she used was that they only supported pivot jumps in the same table, which is inline with our discussions in Sections 4.1 and 4.2: *"going to another chart is challenging because when they (tool creators) are building it, they often will grab the only the values you can find in that chart (table). When you want to grab something outside that chart (table), it's not easy to do"*. P6 further commented that *"Having drill down (pivot jump) like Kyrix-J in one of those tools I'm using would be* *amazing."* P8 also compared using Tableau to do pivot jumps versus using Kyrix-J and echoed that Kyrix-J offers a better experience.

More suggestions on improvements. A common feature suggested by the participants was to enable editing of the visualizations in the UI. They thought *"being able to edit the query and filters would be nice* (P4)*"*, and many wanted to create their own SQL query and visualization directly in the UI (P1 and P4). Although all tasks in the study were doable with the 70 available visualizations, being able to create visualizations in situ could make the system more powerful. P2 commented *"it (Kyrix-J) is limiting when the thing should show up doesn't show up"*, which was referring to the case where a desired jump path does not exist from a visualization and she wanted to write custom queries and visualizations instead of searching for another visualization. Currently, Kyrix-J offers a JSON interface for authoring visualizations. Bring that functionality into the UI is an obvious and straightforward improvement.

The rest of the suggestions were more minor, which included examples such as using the right mouse click instead of the left click to open up the jump paths popover window (Figure 4-3g), which resembled a menu that people are more accustomed to open using right clicks. We have digested these feedback and are preparing a more polished version of the system in a future release.

4.7 Discussions on Limitations and Future Work

In this section, we discuss the limitations we see in the current system and outline potential future work.

Enabling visualization authoring. As suggested by the user study results, a desirable feature is to enable authoring/editing visualizations in the UI. Although it is straightforward to bring the current JSON-based command-line authoring paradigm into the UI, it currently only supports simple `SELECT` and `GROUP BY` queries as discussed in Section 4.4.1. We plan to lift this restriction in the future to support more types of SQL queries. The challenge here is how to support large query results in the process of visualization authoring. It is undesirable if a user has to wait for a long time for the query result to return every time a query is changed. We will look into using sampling strategies to enable interactive authoring on large datasets, as well as design an asynchronous workflow to query the database if a user decides to run a full query.

Tackling databases in the wild. RDBMSes in the wild are very complex, consisting of a large number of tables that often lack maintenance [126] which leads to data lakes [66, 92] or even data swamps [29] that are highly unstructured and hard to understand. To deploy Kyrix-J in the wild, we face the following challenges.

First, table and column names are often machine-generated. The quality of visualization titles may not be high if they are authored by users in a multi-user environment. These will make it harder to comprehend the meaning of a pivot jump. We plan to address this challenge by integrating with metadata management systems (e.g. Google's Goods [65] and Lyft's Amundsen [3]) to ensure that popular tables are more visible to the users and get more quality crowd-sourced annotations of tables and columns. We will also investigate quality control mechanisms such as allowing custom-made data visualizations by users to be rated by a pool of users so that we can make highly-rated jump paths more visible and vice versa.

Second, the amount of jump paths will grow tremendously at scale when there are hundreds of even thousands of visualizations. The current way of browsing the jump paths is limiting in that it requires a user to scroll through a list of paths. To enable more efficient browsing of the jump paths, we plan to offer a functionality to allow the user to search a jump path using keywords.

Supporting more types of filters. Currently the filter function of a jump path (Definition 2) supports only the selection of one input object and outputs only equality filters. One avenue of future research is to support more powerful types of filter functions, e.g., ones that allow the selection of multiple objects or even support more numerical comparisons such as *greater than*. More powerful filter functions will make Kyrix-J able to answer questions such as: what countries have an area between United States and India? Note, however, that one potential challenge is that the space of possible jump paths will be significantly larger because of more ways to compare values and combine the filters. We envision that there needs to be a mechanism to intelligently filter out "unexciting" jump paths or guide users through constructing their own jump paths.

4.8 Conclusion

In this chapter, we present the design of Kyrix-J, an end-to-end system for supporting effective pivot jumps between visualizations for multiple tables in an RDBMS environment. Kyrix-J addresses two major challenges raised by a multi-table RDBMS setting which prior systems fail to address. First, data attributes are highly connected in an RDBMS, leading to a large number of possible jump paths. Kyrix-J automatically generates meaningful jump paths based on PK-FK relationships between tables, thereby freeing users from manually creating filtered visualizations. Second, performing pivot jumps between multiple tables can be disorienting to users because it involves comparing and joining columns from multiple tables. To support the users in staying oriented, Kyrix-J contributes a novel UI which features the use of coordinated multiple views to enable users to quickly navigate between visualizations using pivot jumps, while also being able to keep track of information such as what table they are visually examining, where they come from and what the current SQL query is. To evaluate the effectiveness of Kyrix-J, we conducted a user study where eight participants performed search tasks using Kyrix-J in an RDBMS with 40 tables, 70 visualizations and over 1,000 pairwise jump paths between the visualizations. The results showed that all participants completed the tasks fast and accurately and rated Kyrix-J to be easy to use and help them stay oriented during their exploration. Participants also gave positive feedback on the system as well as suggestions which inspired possible improvements.

Chapter 5

More Discussions

Systems research is coupled with a lot of decision making, usually as attempts to optimize for both generality and applicability in the real world. The three systems described in previous chapters are no exception. Over the course of the development, we constantly prioritize certain features that might be of more interest to the visualization and database community than others, try different directions and implementation methods, and optimize between trying a lot of novel ideas and making a more usable research prototype system in order to see more applications of completed research. After the thesis is done, we have collected a list of things that we have tried but did not end up in publications as well as things that are worth being worked on in the future. This chapter serves as a documentation of such things, including lessons learned, alternative designs, limitations and future work.

5.1 The Multi-dimensional Performance Problem

We have tackled the performance problems in the systems, Kyrix and Kyrix-S in particular, along many dimensions. First, we need to optimize for both the online performance and the offline performance. Second, we also need to consider both the vertical scalability (i.e. the ability to scale on a single computer) as well as the horizontal scalability (i.e. the ability to scale with more computers). Lastly, different types of databases also require different scaling strategies. In the following, we elaborate on these performance dimensions.

5.1.1 Online Performance

In both Kyrix and Kyrix-S, we have required the systems to respond to any user interaction under 500 ms, en empirical upper bound used widely by popular websites [11]. While in the performance experiments we showed that the systems could satisfy this requirement in a variety of settings, we do note that we made a few assumptions about the experiment settings and that to relax these assumptions we need to improve the systems in certain ways.

First, we used cloud instances with decent internet connections to host our web server. The client was in the same country (the United States). This setup ensured that the network transmission times were only a small fraction of the end-to-end latency. However, in the real world not everyone has good internet connections. We probably need to assume that network time is a big fraction of the 500 ms budget and further constrain the latency budget for fetching the data from the database. Fortunately, experiments in Sections 2.8 and 3.8 have shown that the data fetching response times were in the order of tens of milliseconds, leaving plenty of room for potentially long network transmission times. Further, in really bad situations, we can progressively send data (e.g. one object at a time) from the backend to the frontend so that the user could see partial visualizations quickly.

Second, we assumed that the number of objects in any viewport was not too high (e.g. thousands at most) and thus the frontend rendering time was negligible. Bounded visual density is guaranteed by Kyrix-S. In Kyrix, it is the developer's job to ensure that the density of objects is low otherwise both the frontend rendering and the network transmission can be more time consuming. However, as the field of information visualization advances, a few visual representations emerge where high visual densities are in fact desirable. For example, Philippe et. al visualizes a large collection of documents as point clouds [105] in a galaxy-like environment which allows a user to both grab high-level object distributions as well as inspect individual objects. The current object-based data fetching paradigm is limiting in these high-density cases because it is prohibitive to both transmit a lot of objects to the client and render them in the frontend. To make high-density visual representations work, we can explore a few directions. First, we can pre-render the visualizations as images on the server-side, and only fetch the images into the client. This approach has several drawbacks which are discussed in Section 2.5.3. Second, we can look into advanced compression mechanisms to make the network transmission times acceptable, and then use GPU-based renderers to render a massive amount of objects in parallel.

5.1.2 Offline Performance

Both Kyrix and Kyrix-S use an offline phase to compute object layouts and spatial indexes. As noted in the developer study results (Section 2.7), this can hinder the iterative workflow of visualization developers. We have made some attempts to reduce this turnaround time, both in a single-node setting (vertical scaling) and using multiple nodes (horizontal scaling). Here, we describe the approaches we have tried and also other potential solutions.

Vertical Scaling Approach 1: the "Funny" R-tree

The most time-consuming part of the indexing phase in Kyrix (and also in Kyrix-S) is building the database spatial indexes, which are often in the form of 2D data structures such as R-trees [63]. Building a disk-based R-tree for a large set of data points is expensive because the database needs to write out many new pages to store R-tree nodes, which are linear in the number of data points. A post-sorting process is also needed to "cluster" the index pages in Z order [109] or Hilbert order [75] so that online queries incur less disk seeks, which is another process with a running time directly related to the number of data points.

The "funny R-tree" solution[1] is based on a simple idea: we compress nearby data points into one row (e.g. with a big string field) in the database table instead of multiple rows, and then build the indexes on the new table with much fewer rows.

[1]The name "funny" R-tree was how Professor Michael Stonebraker was referring to this approach.

The intuition is that by reducing the number of rows, we essentially reduce the size of an R-tree, which leads to shorter build time. Each of these "super" rows now represents a set of data points, whose bounding box column (the `bbox` column in Figure 2-8) is the bounding box of these data points.

The runtime logic is also simple: the backend fetches from the database all super rows whose bounding boxes intersect the viewport, decompresses the rows into normal rows and then sends them to the client. Note that there can be rows fetched that contain objects not in the viewport which can be simply discarded. It is, however, guaranteed that objects in the viewport will always be fetched.

Now the "magical aspect" of this approach is that the runtime performance is also improved because the runtime performance is linear in the depth of the R-tree and that we now have a shallower R-tree because of the reduction in the number of rows. This means that we are not trading off runtime performance for shorter build time.

A key question is how we define what "nearby points" are when performing the row compression. While many clustering algorithms can be applied, only ones with a relatively short running time are acceptable otherwise it might lead to even longer build time. Note that we do not need exact clustering algorithms such as kNN which tend to have a quadratic time complexity. Therefore, approximate algorithms with a linear time complexity such as Birch [144] can be applied.

In our experimentation, we used an even simpler approach which partitions a Kyrix canvas into equal-sized grids, and compress all objects in each grid as one database row. This approach is simple yet effective. It reduced the build time of Kyrix on one billion objects in a single-node PostgreSQL database from one week down to 12 hours. The runtime performance was on the order of single-digit milliseconds, which was also a big improvement over the original one-row-per-object approach.

Note that this approach is not without limitations. By serializing a set of data points into one compressed row, we lose the ability to apply filters on the raw data attributes. Therefore jumps may not be applicable in canvases that adopt this optimization. To enable filtering, we can apply the filters in the backend or in the client. Yet that requires the selectivity of the filter to be low to prevent fetching a large number of unfiltered objects into the memory.

Vertical Scaling Approach 2: Spatial Partitioning

Our second single-node approach is based on database partitioning. The idea is to partition the set of objects in one Kyrix layer into multiple database tables. The original big R-tree index thus turns into a set of smaller R-trees, one on each partition table, which can lead to a shorter aggregated build time. The reason is that the disk thrashing involved when the R-tree index could not fit in the memory can be much more expensive as the number of objects increases.

We can partition the table spatially (e.g. similar to the spatial partitioning used in Kyrix-S) so that runtime queries tend to hit the same table more often. The only difficult case is when a user viewport spans multiple partition tables where the database needs to query different tables.

In one case study using PostgreSQL, we observed that the build time was improved by five times using this approach compared to the original approach. The dataset used had 100 million objects. The response times still stayed around 100ms-200ms even when multiple queries were involved.

Moreover, this approach can be combined with the first vertical scaling approach to further optimize both build time and runtime performance.

Vertical Scaling Approach 3: Pre-sorting the Data

Another approach which could in theory reduce the build time is to pre-sort the 2D objects using Z order [109] or Hilbert order [75] outside the database. This observation comes from the fact that we need the objects in the database to layout in Z order or Hilbert order to reduce disk seeks at runtime, and that currently we rely on the database to perform this sorting, which is often not optimized. For example, in PostgreSQL, the data is sorted by traversing the 2D spatial index tree, which could lead to large amounts of disk seeks and thrashing. Nowadays, there are very powerful external disk-based sorting algorithms that fully leverage the underlying hardware for high performance [15]. By utilizing these algorithms, it is possible that build time can be reduced without affecting runtime performance.

Horizontal Scaling: Multi-node Approaches

As shown in the Kyrix-S Chapter, our index paradigm is amenable to scaling horizontally, which means utilizing a set of distributed database instances to perform indexing in parallel. For Kyrix we also implemented distributed indexing using Citus, the distributed extension for PostgreSQL.

The key in making Kyrix work with a distributed database lies in how we partition the data. We have two options: hash-based partitioning and spatial partitioning. Both strategies have their pros and cons. Hash-based partitioning involves little build-time overhead after the data is loaded (often in parallel), but may have poor runtime performance especially in a multi-user setting. The reason is that in a distributed database with a coordinator-worker model, the coordinator needs to communicate with multiple workers for one data fetching request, which can incur lots of CPU and I/O costs. On the other hand, partitioning the data spatially can avoid this problem, but can involve expensive build-time overhead as the coordinator needs to reshuffle the data across the cluster in order to put objects in the right partitions, which is often an expensive I/O bound process as shown in Figure 3-10. More experimentation is needed in order to decide which approach to use for a given use case.

Sampling for Interactive Debugging

It is conceivable that even with extensive optimizations, it might still be hard to achieve interactive debugging which requires sub-second build time. As one possible future work, we can investigate the use of sampling to allow a developer to see partial visualizations interactively for debugging, and only commit for a full indexing once they are sure that no more change/tweak of indexing-related parameters is needed.

Alternative Databases

The prototype systems are "married" to the relational database PostgreSQL, which is decision we make based on its popularity and also after experimenting with other databases. For example, we have tested the integration with Vertica, a columnar database. However, the spatial indexes were not fast enough to support 500 ms response times. We have also tried SciDB, an array database, but failed to integrate because the array database data model did not allow objects to have any overlap.

Nevertheless, there are alternative databases that are worth exploring in the future. For example, the Citus distributed extension of PostgreSQL used in Kyrix and Kyrix-S, while being powerful, uses a coordinator-worker model which incurs significant build-time overhead for spatial partitioning. A distributed database with a more decentralized model could alleviate this problem. Also, we can extend Kyrix-J to support document databases [67] where there are lots of relationships between objects that can be better understood with the help of jumps.

5.2 Updating Data

Frequently updating data is very common in a variety of domains such as transportation, finance and web monitoring. Yet due to potentially limited outreach, we unfortunately did not meet a real use case that required the support for data that is frequently updated. Oftentimes, daily/weekly updates of the data were performed and therefore re-running the offline indexing phase was sufficient for the use cases we met. Since we drove our research based on the use cases at hand, we did not prioritize the support for frequent data updates.

To support streaming data updates in the future, we outline here a few possible strategies. In Kyrix, supporting updating data is actually straightforward. Since Kyrix builds an index table for each layer, we only need to use database triggers to listen to updates on the raw data table, and then propagates the updates to raw data tables to the index tables. The spatial indexes built will typically be updated by the database automatically in logarithmic time.

Supporting data updates in Kyrix-S is more challenging due to the inflexible placement requirements used in the layout algorithm: 1) objects can only overlap to a certain degree, 2) object density cannot be higher than a given threshold in any regions and 3) we need to place more important objects on top zoom levels. The layout algorithm is expensive so we cannot afford re-running it on every change to the placement of one object. Without rerunning the algorithm, the users will possibly experience undesirable visual clutter, see some unimportant objects on top levels, and miss important objects that are buried in lower levels as the updates come in. A potential way to alleviate this problem is to re-run the algorithm periodically, or maybe use a profiler to monitor the overlap, density, and the visibility of importance objects, and only rerun the algorithm if the metrics being monitored become too low to be acceptable.

Fortunately, it is still straightforward to update attributes that do not affect the placement of the objects in Kyrix-S. Similar to supporting updates in Kyrix, we listen to data updates to the raw data tables using database triggers. Then starting from the bottom zoom level, we propagate the changes according to how clusters are merged and update the aggregations at the same time.

Note that even if the database indexes are up-to-date, a user might still be looking at stale data. We can use modern web technologies such as web sockets to notify users that the data is out of date and that they can refresh the browser to see new data.

Another way to look at data updates is that the systems themselves can be used as interfaces for performing data updates. We refer interested readers to Peter Griggs's masters thesis [17] which includes several promising proposals on how to use Kyrix and Kyrix-S as interfaces for updating the database.

5.3 Continuous Zooming

A common question we get is: given the discrete canvas model of Kyrix, can Kyrix support continuous zooming? Continuous zooming often means that a user can seemingly infinitely scroll to zoom in and see more objects appear in a smooth and continuous way.

The answer to this question is yes, despite that the canvas model is discrete. The reason is that for any continuous zooming application, the set of altitudes of objects (i.e. a real number indicating how deep one object is down the zoom hierarchy) is always finite and discrete. We can create a canvas for each distinct altitude. The resulting application will then be a continuous zooming application.

A more relevant question is: whether the canvas model is the best way to author a continuous zoom application? The answer is probably no. Ideally, a developer only needs to specify what objects are on the same altitude, and the exact altitude on which those objects appear, without worrying about canvases and their sizes. Yet, we can easily build a higher-level language on top of the current canvas-based language which allows such specifications and translate them to canvases behind the scenes.

Implementation wise, a related challenge in Kyrix-S is that it is hard to achieve continuous zooming for some renderers that are based on Kernel Density Estimations (KDE), e.g., heatmap and contour in Figures 3-2a and 3-2b, which need to be re-rendered on every zoom-in (scroll) event in order to see continuous changing of the visualizations. These renderers, which require computations that are on the order of total available pixels, are too compute-intensive for the web SVG model we use. We plan to investigate GPU-based renderers such as WebGL to address this issue in the future.

5.4 Open Source

We have released the prototype systems as open source software. Kyrix and Kyrix-S are in the same Github repository at `https://github.com/tracyhenry/kyrix`. Kyrix-J is in this Github repository at `https://github.com/tracyhenry/kyrix-j`.

Although we have tried extensively to create a community around open source Kyrix, we have not succeeded much in doing so as of writing this thesis. Although we could use the excuse that we have been shorthanded (with only one full-time PhD student and master/undergraduate students who come and go), there are things that we could have done better at. Here, we want to document these learnings which hopefully can inspire interested readers.

First, we have not been super committed in helping our users deploy their applications. Due to the need to write papers and theses as well as the desire to work on more general research problems, we graduate students could easily get bored by the endless minor revision requirements proposed by our users. This has contributed in part to some users deciding not to use the applications built by us. In fact, as shared by maintainers of many successful open-source research projects, it is critically important to “over” deliver on the first few users so that they are willing to help spread the words. More adopters could lead to more use cases, more collaborators and thus more interesting research ideas, which are often worth the grunt work that needs to get done.

Second, it is important to choose the technology stack that integrates well with the rest of the world. We were not super careful in choosing our tech stack, partly because of the inexperienced lead maintainer, the author of this thesis, me. I had written very little full-stack code before we started the Kyrix project in 2018. As a result, many immature engineering decisions were made that did not follow industry best practices/standards. For example, I chose to write the backend in Java in part because of personal familiarity, which is arguably not one of the top choices for a backend language. As another example, although I made the right call that Kyrix should offer mechanisms to easily embed a Kyrix app in other web applications, the implementation required the target application to include many large dependency files. The industry standard seems to be using HTML iFrame, which is more convenient and lightweight.

In hindsight, there are countless ways we can architecture the systems differently so that they are easier to be integrated with other systems such as popular frontend frameworks (e.g. React and Vue) and data science platforms (e.g. Jupyter notebooks), and potentially will attract more users and collaborators. Fortunately, we have learned the lessons and will try harder in the future.

Chapter 6

Conclusion

In this thesis, we contributed three integrated systems Kyrix, Kyrix-S and Kyrix-J to facilitate the creation of scalable details-on-demand (DoD) data visualizations. As datasets get increasingly larger, the need for scalable tools that bring insight into big data is increasing rapidly. With the release of these systems as open source software, we hope that we have equipped visualization developers another powerful weapon to tame their complex datasets.

In addition to being useful tools on their own, the systems open up a lot of future possibilities. With Kyrix being the foundation system, we can explore high-level designs for a variety of other use cases. For example, a large graph network is inherently amenable to being visualized in multiple level of details to mitigate the potential visual clutter generated by lots of nodes and edges. It is an interesting and open research question as to how to construct the levels of details to satisfy similar usability requirements in Kyrix-S. The problem should be harder since it also involves the display of edges. An even harder case would be to visualize a large knowledge graph, where the nodes and edges can have different categories. Nonetheless, regardless of how the levels of details are constructed, Kyrix can serve as the foundation that a high-level extension is built on top of where its data fetching pipeline and rendering engine can be reused. In fact, the research team behind Kyrix is working on building such an extension system of Kyrix for graph networks. As other examples, the DoD paradigm can facilitate the understanding of hierarchical data and time series data. It would make sense to build on top of Kyrix to support these types of data as well.

Looking forward, we hope that our research can inspire researchers to explore more scalability problems in the field of data visualization. Historically, the visualization field is largely concerned with designing new visual representations for small-scale data that usually can fit in the client memory. As datasets get bigger, it makes sense to remove this assumption and explore ways to scale novel visualization methods to practical data size.

Further, our systems have shown that the meaning of scalability could be different in different settings. In Kyrix, we aim to scale to large data tables. In Kyrix-S, we have handled "perceptual scalability", which means designing algorithms to make visualizations easier to be consumed by a human. In Kyrix-J, we need to scale to a large number of relationships between pairwise visualizations. Similar scalability problems exist in every corner of the rapidly evolving field of data visualization. For instance, one can explore the problem of multi-user scalability, i.e., supporting a large number of concurrent users who are viewing/editing the visualization collaboratively. As another example, data visualizations are scattered all over the Internet. We can explore how to identify interesting visualizations (e.g. ones that match a natural language description or match a given visualization) from a potentially large number of sources on the Web.

To solve these scalability problems, we believe the design principles proposed in this thesis, namely 1) offer declarative visualization languages to facilitate rapid authoring and 2) work with a database system to transparently handle the optimizations needed for interactivity, form a framework that future scalable visualization systems adopt or refer to. With more emphasis on scalability, our visualization community will be able to contribute significantly to the democratization of big data.

Bibliography

[1] Chartio. https://chartio.com/. accessed: 2021/03.

[2] Leaflet markercluster plugin. https://github.com/Leaflet/Leaflet.markercluster. accessed: 2019/11.

[3] Lyft amundsen. https://eng.lyft.com/amundsen-lyfts-data-discovery-metadata-engine- accessed: 2021/03.

[4] More powerful data drilling. https://help.looker.com/hc/en-us/articles/360023589613--More-Powerful-Data-Drilling. accessed: 2021/03.

[5] Mysql workbench. https://www.mysql.com/products/workbench/. accessed: 2021/03.

[6] Postgresql. https://www.postgresql.org/. accessed: 2021/03.

[7] Postgresql documentation: Accessing a database. https://www.postgresql.org/docs/current/tutorial-accessdb.html. accessed: 2021/03.

[8] Postgresql generalized search tree. https://www.postgresql.org/docs/12/textsearch-indexes.html. accessed: 2020/07.

[9] Postgresql gin. https://www.postgresql.org/docs/13/gin-intro.html. accessed: 2021/03.

[10] React js. https://reactjs.org/. accessed: 2021/03.

[11] Site performance for webmasters. https://youtu.be/OpMfx_Zie2g. accessed: 2021/05.

[12] Sql server integration services. https://docs.microsoft.com/en-us/sql/integration-services/sql-server-integration-services?view=sql-server-ver15. accessed: 2021/03.

[13] Sqlite foreign key. https://sqlite.org/foreignkeys.html. accessed: 2021/03.

[14] Vega-lite javascript api. https://observablehq.com/@vega/vega-lite-api. accessed: 2019/11.

[15] Sort benchmark. `http://sortbenchmark.org/`, 2008. Accessed: 2018-09-19.

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

[17] Masters thesis: Database updates using interactive pan and zoom visualizations. `https://peterg17.github.io/files/thesis.pdf`, 2021. Accessed: 2021-05.

[18] Christopher Ahlberg. Spotfire: an information exploration environment. *ACM SIGMOD Record*, 25(4):25–29, 1996.

[19] 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.

[20] 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.

[21] Benjamin B. 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.

[22] Benjamin B Bederson and James D 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.

[23] Benjamin B 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.

[24] Christian Beilschmidt, Thomas Fober, Michael Mattig, and Bernhard Seeger. A linear-time algorithm for the aggregation and visualization of big spatial point data. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 73. ACM, 2017.

[25] Jon Louis Bentley. K-d trees for semidynamic point sets. In *Proceedings of the sixth annual symposium on Computational geometry*, pages 187–197. ACM, 1990.

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

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

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

[29] Will Brackenbury, Rui Liu, Mainack Mondal, Aaron J Elmore, Blase Ur, Kyle Chard, and Michael J Franklin. Draining the data swamp: A similarity-based approach. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, pages 1–7, 2018.

[30] 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.

[31] 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.

[32] Haidong Chen, Wei Chen, Honghui Mei, Zhiqi Liu, Kun Zhou, Weifeng Chen, Wentao Gu, and Kwan-Liu Ma. Visual abstraction and exploration of multi-class scatterplots. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1683–1692, 2014.

[33] Helen Chen, Sophie Engle, Alark Joshi, Eric D Ragan, Beste F Yuksel, and Lane Harrison. Using animation to alleviate overdraw in multiclass scatterplot matrices. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, page 417. ACM, 2018.

[34] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM transactions on database systems (TODS)*, 1(1):9–36, 1976.

[35] Xi Chen, Wei Zeng, Yanna Lin, Hayder Mahdi Al-Maneea, Jonathan Roberts, and Remco Chang. Composition and configuration patterns in multiple-view visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[36] Xin Chen, Tong Ge, Jian Zhang, Baoquan Chen, Chi-Wing Fu, Oliver Deussen, and Yunhai Wang. A recursive subdivision technique for sampling multi-class scatterplots. *IEEE transactions on visualization and computer graphics*, 2019.

[37] 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.

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

[39] 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.

[40] Christopher Collins and Sheelagh Carpendale. Vislink: Revealing relationships amongst visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1192–1199, 2007.

[41] Qingguang Cui, Matthew Ward, Elke Rundensteiner, and Jing Yang. Measuring data abstraction quality in multiresolution visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):709–716, 2006.

[42] Bogdan Czejdo, Ramez Elmasri, Marek Rusinkiewicz, and David W. Embley. A graphical data manipulation language for an extended entity-relationship model. *Computer*, 23(3):26–36, 1990.

[43] 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.

[44] Shaul Dar, Gadi Entin, Shai Geva, and Eran Palmon. Dtl's dataspot: Database exploration using plain language. In *VLDB*, volume 98, pages 24–27, 1998.

[45] Anish Das Sarma, Hongrae Lee, Hector Gonzalez, Jayant Madhavan, and Alon Halevy. Efficient spatial sampling of large geographical tables. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 193–204. ACM, 2012.

[46] Jean-Yves Delort. Vizualizing large spatial datasets in interactive maps. In *2010 Second International Conference on Advanced Geographic Information Systems, Applications, and Services*, pages 33–38. IEEE, 2010.

[47] Mark Derthick, Michael G 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.

[48] 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.

[49] 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.

[50] Marian Dörk, Nathalie Henry Riche, Gonzalo Ramos, and Susan Dumais. Pivotpaths: Strolling through faceted information spaces. *IEEE transactions on visualization and computer graphics*, 18(12):2709–2718, 2012.

[51] Marina Drosou and Evaggelia Pitoura. Disc diversity: result diversification based on dissimilarity and coverage. *Proceedings of the VLDB Endowment*, 6(1):13–24, 2012.

[52] Cody Dunne, Nathalie Henry Riche, Bongshin Lee, Ronald Metoyer, and George Robertson. Graphtrail: Analyzing large multivariate, heterogeneous networks while supporting exploration history. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 1663–1672, 2012.

[53] 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.

[54] 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, 2009.

[55] 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.

[56] Wu Eugene, Battle Leilani, and R Madden Samuel. The case for data visualization management systems. *Proceedings of the VLDB Endowment*, 7(10):903–906, 2014.

[57] J-D Fekete and Catherine Plaisant. Interactive information visualization of a million items. In *IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.*, pages 117–124. IEEE, 2002.

[58] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1001–1012. IEEE, 2018.

[59] Michael Glueck, Azam Khan, and Daniel J 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.

[60] Jade Goldstein and Steven F 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.

[61] Google, Inc. Google maps. `https://www.google.com/maps`.

[62] Tao Guo, Kaiyu Feng, Gao Cong, and Zhifeng Bao. Efficient selection of geospatial data on maps for interactive and visualized exploration. In *Proceedings of the 2018 International Conference on Management of Data*, pages 567–582. ACM, 2018.

[63] 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.

[64] Carl Gutwin and Saul Greenberg. The effects of workspace awareness support on the usability of real-time distributed groupware. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 6(3):243–281, 1999.

[65] Alon Halevy, Flip Korn, Natalya F Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Goods: Organizing google's datasets. In *Proceedings of the 2016 International Conference on Management of Data*, pages 795–806, 2016.

[66] Alon Y Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Managing google's data lake: an overview of the goods system. *IEEE Data Eng. Bull.*, 39(3):5–14, 2016.

[67] Jing Han, Ee Haihong, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th international conference on pervasive computing and applications*, pages 363–366. IEEE, 2011.

[68] 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.

[69] Jeffrey Heer and Danah Boyd. Vizster: Visualizing online social networks. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.*, pages 32–39. IEEE, 2005.

[70] Jeffrey Heer, Jock Mackinlay, Chris Stolte, and Maneesh Agrawala. Graphical histories for visualization: Supporting analysis, communication, and evaluation. *IEEE transactions on visualization and computer graphics*, 14(6):1189–1196, 2008.

[71] Florian Heimerl, Chih-Ching Chang, Alper Sarikaya, and Michael Gleicher. Visual designs for binned aggregation of multi-class scatterplots. *arXiv preprint arXiv:1810.02445*, 2018.

[72] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 670–681. Elsevier, 2002.

[73] Jaemin Jo, Frédéric Vernier, Pierre Dragicevic, and Jean-Daniel Fekete. A declarative rendering model for multiclass density maps. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):470–480, 2018.

[74] Susanne Jul and George W Furnas. Critical zones in desert fog: aids to multiscale navigation. In *Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 97–106. ACM, 1998.

[75] Ibrahim Kamel and Christos Faloutsos. Hilbert r-tree: An improved r-tree using fractals. Technical report, 1993.

[76] Pimin Konstantin Kefaloukos, Marcos Vaz Salles, and Martin Zachariasen. Declarative cartography: In-database map generalization of geospatial datasets. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1024–1035. IEEE, 2014.

[77] Daniel A Keim and Annemarie Herrmann. *The gridfit algorithm: An efficient and effective approach to visualizing large amounts of spatial data.* IEEE, 1998.

[78] Peter Kerpedjiev, Nezar Abdennur, Fritz Lekschas, Chuck McCallum, Kasper Dinkla, Hendrik Strobelt, Jacob M Luber, Scott B Ouellette, Alaleh Azhir, Nikhil Kumar, et al. Higlass: web-based visual exploration and analysis of genome interaction maps. *Genome biology*, 19(1):1–12, 2018.

[79] Donald E Knuth. Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12):735–736, 1964.

[80] Robert Kosara, Silvia Miksch, and Helwig Hauser. Focus+ context taken literally. *IEEE Computer Graphics and Applications*, 22(1):22–29, 2002.

[81] Fritz Lekschas, Michael Behrisch, Benjamin Bach, Peter Kerpedjiev, Nils Gehlenborg, and Hanspeter Pfister. Pattern-driven navigation in 2d multiscale visualizations with scalable insets. *IEEE transactions on visualization and computer graphics*, 2019.

[82] Fei Li and HV Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.

[83] Hongsen Liao, Yingcai Wu, Li Chen, and Wei Chen. Cluster-based visual abstraction for multivariate scatterplots. *IEEE transactions on visualization and computer graphics*, 24(9):2531–2545, 2017.

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

[85] Can Liu, Cong Wu, Hanning Shao, and Xiaoru Yuan. Smartcube: An adaptive data management architecture for the real-time visualization of spatiotemporal datasets. *IEEE transactions on visualization and computer graphics*, 2019.

[86] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 563–574, 2006.

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

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

[89] Wolfgang May. Information extraction and integration with FLORID: The MONDIAL case study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from `http://dbis.informatik.uni-goettingen.de/Mondial`.

[90] Adrian Mayorga and Michael Gleicher. Splatterplots: Overcoming overdraw in scatter plots. *IEEE transactions on visualization and computer graphics*, 19(9):1526–1538, 2013.

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

[92] Natalia Miloslavskaya and Alexander Tolstoy. Big data, fast data and data lake concepts. *Procedia Computer Science*, 88:300–305, 2016.

[93] Fabio Miranda, Lauro Lins, James T Klosowski, and Claudio T Silva. Topkube: a rank-aware data cube for real-time exploration of spatiotemporal data. *IEEE Transactions on visualization and computer graphics*, 24(3):1394–1407, 2017.

[94] Dominik Moritz, Bill Howe, and Jeffrey Heer. Falcon: Balancing interactive latency and resolution sensitivity for scalable linked visualizations. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, page 694. ACM, 2019.

[95] Randall Munroe. Zoomcharts. https://zoomcharts.com/en/, May 2010.

[96] Chris North and Ben Shneiderman. Snap-together visualization: a user interface for coordinating visualizations via relational schemata. In *Proceedings of the working conference on Advanced visual interfaces*, pages 128–135, 2000.

[97] Sarana Nutanong, Marco D Adelfio, and Hanan Samet. Multiresolution select-distinct queries on large geographic point sets. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, pages 159–168. ACM, 2012.

[98] Heather L O'Brien and Elaine G Toms. What is user engagement? a conceptual framework for defining user engagement with technology. *Journal of the American society for Information Science and Technology*, 59(6):938–955, 2008.

[99] Cicero A. L. Pahins, Sean A. Stephens, Carlos Scheidegger, and Joao 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.

[100] Cicero AL Pahins, Sean A Stephens, Carlos Scheidegger, and Joao LD Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *IEEE transactions on visualization and computer graphics*, 23(1):671–680, 2016.

[101] Deokgun Park, Steven M Drucker, Roland Fernandez, and Niklas Elmqvist. Atom: A grammar for unit visualizations. *IEEE transactions on visualization and computer graphics*, 24(12):3032–3043, 2017.

[102] Christian Partl, Alexander Lex, Marc Streit, Hendrik Strobelt, Anne-Mai Wassermann, Hanspeter Pfister, and Dieter Schmalstieg. Contour: data-driven exploration of multi-relational datasets for drug discovery. *IEEE transactions on visualization and computer graphics*, 20(12):1883–1892, 2014.

[103] 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.

[104] Alexandre Perrot, Romain Bourqui, Nicolas Hanusse, Frédéric Lalanne, and David Auber. Large interactive visualization of density functions on big data infrastructure. In *2015 IEEE 5th Symposium on large Data Analysis and Visualization (lDAV)*, pages 99–106. IEEE, 2015.

[105] Caillou Philippe, Renault Jonas, Fekete Jean-Daniel, Letournel Anne-Catherine, and Sebag Michèle. Cartolabe: A web-based scalable visualization of large document collections. *arXiv preprint arXiv:2003.00975*, 2020.

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

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

[108] 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.

[109] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer. Integrating the ub-tree into a database system kernel. In *VLDB*, volume 2000, pages 263–272. Citeseer, 2000.

[110] Alper Sarikaya and Michael Gleicher. Scatterplots: Tasks, data, and designs. *IEEE transactions on visualization and computer graphics*, 24(1):402–412, 2017.

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

[112] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, 23(1):341–350, 2016.

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

[114] 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.

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

[116] 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.

[117] Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. Athena++ natural language querying for complex nested sql queries. *Proceedings of the VLDB Endowment*, 13(12):2747–2759, 2020.

[118] Muhammad Shahbaz, Syed Ahsan, Muhammad Shaheen, Rao Muhammad Adeel Nawab, and Syed Athar Masood. Automatic generation of extended er diagram using natural language processing. *Journal of American Science*, 7(8):1–10, 2011.

[119] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. Discovering queries based on example tuples. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 493–504, 2014.

[120] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE symposium on visual languages*, pages 336–343. IEEE, 1996.

[121] 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.

[122] Michael Spenke and Christian Beilken. Infozoom-analysing formula one racing results with an interactive data mining and visualisation tool. *WIT Transactions on Information and Communication Technologies*, 25, 2000.

[123] John Stasko, Carsten Görg, and Zhicheng Liu. Jigsaw: supporting investigative analysis through interactive visualization. *Information visualization*, 7(2):118–132, 2008.

[124] Jason Stewart, Elaine M Raybourn, Ben Bederson, and Allison Druin. When two hands are better than one: Enhancing collaboration using single display groupware. In *CHI 98 conference summary on Human factors in computing systems*, pages 287–288, 1998.

[125] 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.

[126] Michael Stonebraker, Dong Deng, and Michael L Brodie. Database decay and how to avoid it. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 7–16. IEEE, 2016.

[127] Kenneth L Summers, Timothy E Goldsmith, Steve Kubica, and Thomas P 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.

[128] Marjan Trutschl, Georges Grinstein, and Urska Cvek. Intelligently resolving point occlusion. In *IEEE Symposium on Information Visualization 2003 (IEEE Cat. No. 03TH8714)*, pages 131–136. IEEE, 2003.

[129] Yannis Tzitzikas and Jean-Luc Hainaut. How to tame a very large er diagram (using link analysis and force-directed drawing algorithms). In *International Conference on Conceptual Modeling*, pages 144–159. Springer, 2005.

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

[131] Christophe Viau and Michael J McGuffin. Connectedcharts: explicit visualization of relationships between data graphics. In *Computer Graphics Forum*, volume 31, pages 1285–1294. Wiley Online Library, 2012.

[132] Carsten Waldeck and Dirk Balfanz. Mobile liquid 2d scatter space (ml2dss). In *Proceedings. Eighth International Conference on Information Visualisation, 2004. IV 2004.*, pages 494–498. IEEE, 2004.

[133] Jane Webster and Jaspreet S Ahuja. Enhancing the design of web navigation systems: The influence of user disorientation on engagement and performance. *Mis Quarterly*, pages 661–678, 2006.

[134] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009.

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

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

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

[138] Harry KT Wong, Ivy Kuo, et al. Guide: Graphical user interface for database exploration. In *VLDB*, pages 22–32. Citeseer, 1982.

[139] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE transactions on visualization and computer graphics*, 22(1):649–658, 2015.

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

[141] Shuyun Xu, Yu Li, and Shiyong Lu. Erdraw: An xml-based er-diagram drawing and translation tool. In *Computers and Their Applications*, pages 143–146. Citeseer, 2003.

[142] Jing Yang, Matthew O Ward, and Elke A Rundensteiner. Interactive hierarchical displays: a general framework for visualization and exploration of large multivariate data sets. *Computers & Graphics*, 27(2):265–283, 2003.

[143] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M Procopiuc, and Divesh Srivastava. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment*, 3(1-2):805–814, 2010.

[144] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. *ACM sigmod record*, 25(2):103–114, 1996.

[145] Jian Zhao, Christopher Collins, Fanny Chevalier, and Ravin Balakrishnan. Interactive exploration of implicit and explicit relations in faceted datasets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2080–2089, 2013.

[146] Jian Zhao, Christopher Collins, Fanny Chevalier, and Ravin Balakrishnan. Interactive exploration of implicit and explicit relations in faceted datasets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2080–2089, 2013.

[147] Zoomify, Inc. Zoomify, 1999. Accessed: 2018-09-19.