Table of Contents

Introduction

For data professionals, the initial data exploration phase is vital to inform the direction of probes and analysis. Statistical analysis alone is often not enough, and visual exploration is holding an increasingly key influence towards understanding data, its underlying trends, relationships, and other traits of significance. However, this process is not only tedious due to the need to generate and inspect potentially hundreds of graphs to search for interesting variable relationships - but only becomes worse as data grows. To address this problem came Lux, an open source data exploration system that streamlines users' data science workflows by automating aspects of data exploration. The Lux API supports in-situ use within Jupyter Notebooks, a common environment used by data scientists and analysts. Through an interactive widget and a powerful intent driven system, Lux allows users to easily and automatically generate visualizations that demonstrate trends in their data right within their Jupyter notebooks.

Upon a simple import of data into a Lux-specific object within their Jupyter Notebook, the Lux visualization recommendation system works its magic to automatically generate collections of graphs for users to browse. One of the core components of this system is an execution backend which collects and aggregates data necessary to render visualizations. Lux consists of the original PandasExecutor, the backend that reads in graph specifications and gathers the necessary data from a user's source dataframe. This data is then formatted and passed on to the Lux Renderer which generates an appropriate visualization.

The original target audience for the Lux system included users working with smaller sets of data within DataFrames, but for many professionals, it is common to be working with data stored in relational databases such as Postgres. Previously, in order to use Lux' visual exploration system with database data, users would need to load the data into a local Pandas DataFrame object. However fetching all of the data from a database may not always be an option for users. Storing entire data tables locally may not be feasible due to the prohibitive size of data or a lack of permissions due to the sensitive nature of industry-scale data.

To make Lux more accessible to database users and allow them to leverage all of their data in the exploration process, our capstone project focused on extending the Lux feature set by enabling it to work on a Postgresql database directly by creating a new SQL Execution backend. The execution engine we have created lets users connect Lux to a relational database and use its visualization recommendation system on top of their database systems. To make this work, we needed to make a new SQL Execution engine that automatically creates and pushes queries

to users' databases to gather visualization data. Furthermore, to ensure that the new Lux SQL capabilities meet database users' needs, we conducted interviews to validate and inform our engineering work. Talking with these users helped us understand their workflows and what would be needed for Lux to better fit into them. The interviews also highlighted other API features that would greatly improve users' experience.

In this report, we first outline our interviewing methodology to understand our targeted users' needs, and identify additional functionality that would give them a smoother experience when using Lux. We found that users want a visualization tool that provides transparency over underlying decisions, customization of over visuals, easy handoff for seamless collaboration, and industry wide interoperability both in terms of databases and tools. The User Interview Discovery section is a walkthrough of key takeaways regarding Lux's usability within the context of the user's workflow, as well as our proposed additions to the Intent path with insights from the new intent prototype. Among key takeaways, frequent user requests for increased variety of types of visualizations critical to exploratory data analysis, such as boxplot or violin plot, and display of summary statistics, are valuable learnings for future enhancements.

In the Backend Engineering section, we cover our design considerations when implementing new features that we discovered through our interviews, and the steps we took to optimize performance for a smoother user experience. These include allowing users to perform basic join operations through Lux, increasing the transparency of Lux' data handling through code exporting, and extending Lux to a wider audience through a more general database executor. Finally, we end our report with a discussion of future work that aimed at further improving database users' experience with Lux.

User Experience Research

The initial iteration of Lux provided limited support for databases intended for larger datasets such as Postgres. To identify and prioritize additional SQL Executor features that would make Lux more suitable for database users, we conducted qualitative user research by leveraging remote interviews of potential Lux users. Additionally, a secondary objective of the study was to understand the workflow of Lux's target and adjacent user personas to uncover the pain points that their current visualizations tools are unable to alleviate, thereby increasing the usability and applicability of Lux to a broader audience.

To better inform the technical work of our Capstone, which aimed at fleshing out Lux' database functionality, we wanted to have a clearer idea of what potential database users would want out of a data exploration tool and their feedback on our initial prototype. Thus, the goal of our user experience studies was to identify users' pain points in their data exploration workflows and validate potential features that could be integrated with the existing version of Lux.

Research Questions

The study aimed to answer the following questions:

1. How do data analysts and scientists interact with their databases, and what are their typical workflows?
2. How can we tailor the Lux API and interface to better suit the workflows of the aforementioned target personas?
3. Are there data exploration actions Lux users want to perform but are unable to accomplish while using the API syntax?

Methodology

The project incorporated a design thinking approach to identify and address user pain points through the following steps:

1. **Empathize**: Through exploratory interviews of Lux users and similar personas, we gathered intimate knowledge about gaps in the industry, the goals of the users, and their reservations against Lux.
2. **Define & Ideate**: The insights and findings from interviews informed the ideation process where the team gathered to scope the problem and brainstorm solutions.

3. **Prototype & Test**: Using Figma and ipywidgets initial prototypes were designed to gather feedback from the users.

Participants

During the course of the Lux UX study, we interviewed six participants. The coordinators leveraged three channels: UC Berkeley Information School Slack groups, alumni, and professional networks, to recruit potential interviewees. We sought individuals who had some experience with data science, worked with databases in their day-to-day or were interested in having an easy-to-use visualization solution. For this study, there was no screener to validate participants. However, to garner interest from target participants, a blurb with potential characteristics of Lux's intended audience, its value proposition, and a GIF of Lux was broadcasted across the aforementioned slack groups. In addition, before the interviews, participants were given a brief overview of the Lux to familiarize them with its capabilities and features. Recruited participants had a mix of roles, but each participant demonstrated advanced coding proficiency.

User	Role	Role Description	Company Size (number of employees)	Lux Experience	Coding Proficiency
P1	Growth Marketing Team Manager	Multidisciplinary, managerial role responsible for overseeing the growth of marketing metrics, from AB testing experiment design to iterative technical improvements	~5,000	Beginner	Advanced
P2	Senior Data Science Manager	Managerial role focusing on the end-to-end spectrum of data science technologies, from developing machine-learning models to launching production-ready ML/AI methods to support business goals	~5,000	Beginner	Advanced

P3	Data Engineer	Data engineer focused on developing, maintaining, and analyzing data pipelines for accounting and revenue teams.	~ 575,000	No Experience	Advanced
P4	Machine Learning Engineer	Implements ML models to prevent toxic behavior on gaming platforms.	(~11-50)	No Experience	Advanced
P5	Head of Data Science	Participant's role revolves around analyzing data to share insights with non-technical stakeholders in a presentable and easy to understand format.	~100	Intermediate	Advanced
P6	User Research Manager for Analytics	Manages a team of UX researchers working on a leading visualization tool.	~ 4,000	No Experience	Advanced

Table 1: Details around role, company size, Lux experience and coding proficiency of interviewed participants.

User Personas

While Lux has been developed to streamline the exploratory data process (EDA) of data scientists, we learned that an increasingly diverse range of data professionals -- ranging from analyst, product manager, and software engineering professionals have performed similar functions respectively in their engineering teams. To gain clarity on whether the target user of EDA has expanded beyond data scientists, we included a few questions on their functional roles in our interviews script and early usability testing.

- What is your role inside the company?
  - Describe your main responsibilities.
  - What teams do you interact with during your day-to-day functions?

- What are your primary day-to-day activities as a (insert job title)?
  - Describe the problem that you're trying to solve.
  - Describe the workflow that you currently use to analyze and make sense of your data. What tools do you use in each step of the workflow?

What emerged was a set of target users who are primarily embedded in the technology team and shared similar patterns of data-driven workflows without the formal job title as "data scientist."

As we learned about their day-to-day tasks and identified their main responsibilities, it became clear:

1. Data analysis was no longer a job exclusive to the explicit role of data scientists.
2. Basic data queries are often conducted within each functional team without assistance from the company's core data science department.
3. For companies of 5,000 and above, incorporating data-driven methods in decision-making or product development cycles is today's cultural norm.

The clear, direct articulation of data-driven processes showed how no single job title focused solely on data preparation nor analysis. Instead, today's widespread culture is implicitly built on rigorous use of data analysis and visualization throughout each core functional team. Be it growth marketing, product management, or accounting teams, there are data-centric tasks completed by roles outside of what is traditionally understood to be a data scientist. We realized this expansion and fluidity of data-related professionals across different company departments,

from individual contributors to managerial levels, to be a major opportunity for Lux to serve and address in our scope of work. We also saw similar validation from Tableau's research team when they surveyed their end-users and identified the nine roles (illustrated by the table below) with different degrees of proficiency in statistics, computer science, and domain knowledge who leverage Tableau's visualization tool in their day to day activities[1]. However, during the recruitment for study, only participants with a high degree of coding proficiency showed significant interest in using Lux. Therefore, given a lack of interest from non-coders, the study excluded personas who may lack any coding proficiency and narrowed down on the following personas of Data Scientists and Business Analysts that Lux will serve.

Process	Role	Role Description	In prior Studies	Level of Expertise: Statistics	Computer Science	Domain Knowledge	Human Centered Design
Preparation	Data Steward	Domain expert responsible for governing access and use of data	Data Broker [87]; Data Owners [84]	☐	☐	■	☐
	Data Shaper	Developer responsible for supporting the curation and preparing data for analysis	Data Shaper [38]; Data Preper [38];	▒	▒	▒	☐
Deployment & Engineering	Data Engineer	Engineer proficient in developing Data Science technologies, including data preparation and analysis pipelines	Platform Builder [38]; Data Developer [27]; Hacker [35]; Scripter [34]; Engineer [57]; BI engineer [84]	▒	■	▒	☐
	ML / AI Engineer	Engineer proficient in developing and deploying machine learning / artificial intelligence methods to support data science processes	Hacker [33]; Modeling Specialists [38]	■	▒	▒	☐
Analysis	Generalist	Multidisciplinary individual focused solely on data science	Polymath [39]; Data Creative [27]	▒	▒	▒	▒
	Research Scientist	A domain expert involved in research typically with technical expertise in 'Data Science' technologies	Data Researcher [27, 57]; X-informatician [12]	▒	▒	■	▒
	Technical Analyst	A technical individual from whom data science is not core to their job but occurs only at the margins of other work	Data Analyst [4, 38, 39, 57, 84]; Application User [35]; Business Analyst [36, 57]; Data Business Person [27]; Analysis Team Members [84]	▒	▒	■	☐
	Moonlighter	Non-technical individual tasked to perform data science duties, either voluntarily or through necessity	Moonlighter [39]	☐	☐	■	☐
Communication	Evangelist	Manager, team leader, or analyst tasked with disseminating findings from data science work	Data Evangelist [39]; Communicator [87]; Insight Actor [39]	▒	▒	▒	☐

Table 2. Summary of data science roles and an illustration of their skill sets. Data science skills were classified along four axes: statistics, computer science, domain knowledge, and human centred design. We use a color gradient to illustrate a level of expertise: proficient ■; knowledgeable ▒; working ▒; and little to none.

Figure 1: The table outlines the different data-centric roles who use Tableau along with their proficiency in statistics, computer science, domain knowledge, and human centered design[1]

[1]1 https://research.tableau.com/sites/default/files/Crisan_DataScience.pdf

Data Scientist:

A Data Scientist creates machine learning models using data gathered from different product teams.

Goals:

1. Interprets model outputs to inform the product team.
2. Share insights with stakeholders.

Common Frustrations:

1. Exploratory Data Analysis takes ~ 10% of work time.
2. Libraries like Pandas don't offer ease of use and granularity at the same time.

Business Analyst:

Image source[2]

A Business Analyst identifies insights from the data to help stakeholders make business decisions.

Goals:

1. Identify data trends to validate & inform business decisions.
2. Share insights with stakeholders.

Frustrations:

1. Creating situation specific presentable visualizations.
2. Limited integration across different tools.

[2]Image Source: https://icons8.com/illustrations/author/5eb2a7bd01d0360019f124e7 by Olha Khomich from icons8.com

Key Takeaways from User interviews:

Based on generative interviews and early usability testing results, we identified four action areas around transparency, customization, handoff, and interoperability of existing Lux's approaches that are included in our current scope of work.

Transparency:

The participants emphasized their preference for knowing how specific data points inform particular visualizations. For instance, a user must have the ability to view whether a sample of the data-set or if the entire data set was used to create the visualization. Users also wanted greater transparency regarding how Lux recommends visualizations; they wanted to understand the ranking algorithms used by the API.

Participant quotations:

"Off the bat, I don't always trust visualizations. I want to look at the computations and samples of data behind a particular visualization." (P1)

"It was not clear the visualizations were ranked. So maybe, even like, when those plots are being produced, you know, where it says shows the relationship between two quantitative attributes, just literally saying, like, ranked by correlation, or just literally telling me like, what, what the ranking method is, will help me understand." (P5)

Action item: Allow greater transparency of underlying calculations, data, and details for visualizations.

Customization:

Given the participants work in organizations where they need to collaborate and share insights with multiple stakeholders. They consistently emphasized the importance of a tool that can provide them adequate control over the aesthetics and other aspects of a visualization (legends, bin size, color, etc.).

Participant quotations:

"One of the hassles during visualization is you make a plot in Pandas but that not everything is controllable by Pandas. If you have to move the legend, like outside the box, you have to get the axis object. And then you have to define that you want your legend, like these coordinates, and

move it outside of the box. So Pandas is doing some nice plotting for you, but you can't control everything" (P5)

“The stakeholders are not interested in looking at co-lab or Python notebooks, or code or anything like that. I think they want a more cleaned up, you know, sort of doc version of the stuff.” (P5)

"There are always issues with histogram scale. You want zero to 100. But then, the histogram only does, by default, zero to 10. Then you make it 100, but it doesn't quite catch the granularity. So I never have a good solution for frequency or the number of bins that I need to put in the histograms." (P5)

"Redash creates beautiful visualization and offers a lot of functionalities. You can make a Sankey plot, box pot with very consistent colors. And you don't necessarily need to understand Matplotlib, which works at the backend." (P2)

“Technical folks want the ability to customize visualizations and choose graphs based on their needs, even if it takes a few more steps or lines of code.” (P4)

Action item: Allow greater flexibility and control over customization of aesthetics and other parameters (scale, frequency, etc.) of visualizations without sacrificing ease of use.

Handoff:

The participants who align with the Data Scientist Persona run into scenarios where they have to share visualizations with their teammates or other stakeholders with underlying data samples and provide the context necessary to understand visualization. For instance, P5 mentioned that they prefer using Google sheets to add visualization and the context required to understand them when they want to share their insights with teammates.

Participant quotations:
"So a lot of what I do literally nowadays do a plot in Matplotlib, right-click on it, copy it and put in Slack, or right-click on it, copy it and put it into Google sheets so that I can add some description and kind of scaffolding around the thing that I'm talking about" (P5)

Action item: Allow greater shareability of underlying visualization code to foster collaboration.

Interoperability:

Lux must extend its support to a greater variety of databases used in the industry so that users don't have to learn a new platform each time they move from a company. Additionally, Lux must provide sufficient control over its supported databases, such as Postgres, to allow all the necessary operations required to interact with the database from the Lux interface.

Participant quotations:

“If Lux can cover GCP, and also common databases, then it’ll be able to serve most of the industry, except some of the big techs who use their internal tools.” (P2)

“ 80% of the companies either leverage Databricks, or the Google Cloud platform.” (P2)

“Users transition to different platforms based on team and job changes. They prefer a data visualizer that is not constrained to a specific platform.” (P4)

“Lux doesn’t allow crucial relational DB operations like the “JOIN” table in SQL. Data pipelines of industry professionals merge data from different teams resulting in different SQL DB tables. Therefore, these individuals prefer the ability to combine multiple tables directly through Lux to visualize relations between features across tables.” (P4)

“I think 90% of Lux users would be analysts who are dealing with SQL, and very small portion analysts will be able to throw the data into Spark SQL, where they are free to do further iteration on top of data like modeling. So maybe catering to the needs of the analysts might be a quick win for Lux.” (P2)

Action item: Allow support of Lux for common databases, and provide sufficient interoperability and control over the databases from the Lux interface so that the user does not have to rely on the default methods (ex: SQL queries) to interact with the databases.

Define and Ideate Potential Solutions

Through the exploratory interviews, the team was able to empathize with the user and identify their major concerns with incorporating Lux in their day-to-day work. These insights were coalesced into following “How might we” questions to narrow the scope and help guide us on what additional Lux features would be useful for database users.

1. How might we enable greater control and transparency over Lux visualizations?

Solutions: Provide users the ability to view, sort/filter visualizations, and export the code used to create a particular visualization.

2. How to make it easier for the user to integrate Lux into their existing workflow?
   Solutions: Expand Lux's support to other popular database systems, making it accessible to a wider range of industry users. Extend support for critical commands like "JOIN" tables so that users can seamlessly integrate Postgres with Lux.

3. How might we develop new ways to support a more intuitive onboarding process for new users from non-coding backgrounds?
   Solutions: Provide new interaction methods embedded within the existing Lux interface to support new users, either question-and-answer based questionnaire or visual selection interface. Let the user do the heavy-lifting but with an intuition-based interaction interface and add delight to the often solo data exploration process.

New Intent Prototype

"Exploratory data analysis is an attitude, a state of flexibility, a willingness to look for those things that we believe are not there, as well as those we believe to be there."

--John W. Tukey

Goals

Learning from user interviews, we concluded not only are non-data-scientists conducting routine data analysis and summarizing their findings with data visualizations but they are turning to exploratory data analysis as a source of ideation for AB testing experiments, new product developments, user segmentation, or cost-saving business goals. In between the request for dashboarding reports and custom investigative queries, we identified a state of data exploration that is more in flux and prototyped ways to solve the problems identified earlier.

On the spectrum of intent and specificity needed to create a detailed visualization, users are often adapting their hypothesis on the fly. They do not have tangible mental models to pinpoint the exact visualization they want although they are eager to interact with data and narrow down the list of attributes to further dive into. More commonly, a combination of skill sets and level of expertise across statistics, computer science, domain knowledge, and human centered design is applied when these data professionals are working with their data on a regular basis. Thus, we began the usability testing of Lux's current features with a few hypothesis on whether:

Hypothesis

1. There is a spectrum of intent specificity, ranging from abstract to specific, that is currently unmet by Lux's current technical capabilities
2. There is room for improvement regarding Lux's current interface interactions. These can be more natural language based and less code dependent
3. There is a need for greater control over the sets of recommended visualizations through customized interestingness metrics or user controls

Design Principles

From the hypothesis, we formulated a few ideas and designed the prototypes around these design principles:

- Seamless integration with Lux
- Language-based or visual-based interface interaction

- Clarity of user progress
- No coding required
- Greater user control and accessibility

Moreover, while Lux's initial goals for the intent path were meant to make EDA easy and fast, there are potential consequences that may have been overlooked. Based on recurring themes raised by interviewees P1, P5, and P6, we realized that a different method of intent specification could assist in addressing these prevailing problems within the EDA process. Specifically, we learned how issues related to multiple-comparison data modeling could frequently occur. This informed our decision to implement a more meticulous user interaction method when making the intent prototypes.

Multiple-Comparison Data Modeling

Researchers' claims of statistical significance often fall prey to fishing expeditions or p-hacking even when there are no conscious procedural lapses.[3] Building on the ways that p-hacking may erode research validity, Gelman and Loken propose using multilevel modeling and thorough analysis of relevant comparisons of the dataset from the outset of the data exploration process to resolve these multiple-comparison issues.[4] The authors state, "A starting point would be to analyze all relevant comparisons, not just focusing on whatever happens to be statistically significant."[5] For the context of EDA on largely observational studies in political science or economics, having a rigorous approach to analyzing data with a clear statistical framework of multilevel modeling alongside hypothesis formulation would mitigate errors arising from "insufficient modeling of the relationship between the corresponding parameters of the model."[6]

In many ways, while we understand one of Lux's primary goals is to automate EDA quickly and present visualizations instantly to the user, there is potential in having a more methodical interaction method that can help researchers reduce missteps in procedure prematurely. The more deliberate approach behind the iterative phases of the following intent prototypes attempts to erect guards against aforementioned examples of misuse. Additionally, adding more structure to the EDA process that Lux automates invisibly could complement Lux by giving the user more

---

[3] The garden of forking paths: Why multiple comparisons can be a problem, even when there is no "fishing expedition" or "p-hacking" and the research hypothesis was posited ahead of time∗ Andrew Gelman† and Eric Loken
14 Nov 2013

[4] Same as above

[5] Same as above, page 14

[6] Andrew Gelman, Jennifer Hill & Masanao Yajima (2012) Why We (Usually) Don't Have to Worry About Multiple Comparisons, Journal of Research on Educational Effectiveness, 5:2, 189-211, DOI: 10.1080/19345747.2011.618213, page 190

intuitive, visible control and precision over their own goal-driven actions. As summarized and defined by Dimara and Perin, “good interaction minimizes error and distance to user goal, and provides rapid and stable convergence to the target state.”[7]

Iterative Process

Prototype 1 - Interactive Text-Based Python Notebook

The first prototype was born out of inspiration from coursework from Information Visualization and Presentation (INFO 247 Spring 2021) taught by Professor Marti Hearst and the diagram from Andrew V. Abela redrawn by Berinato.

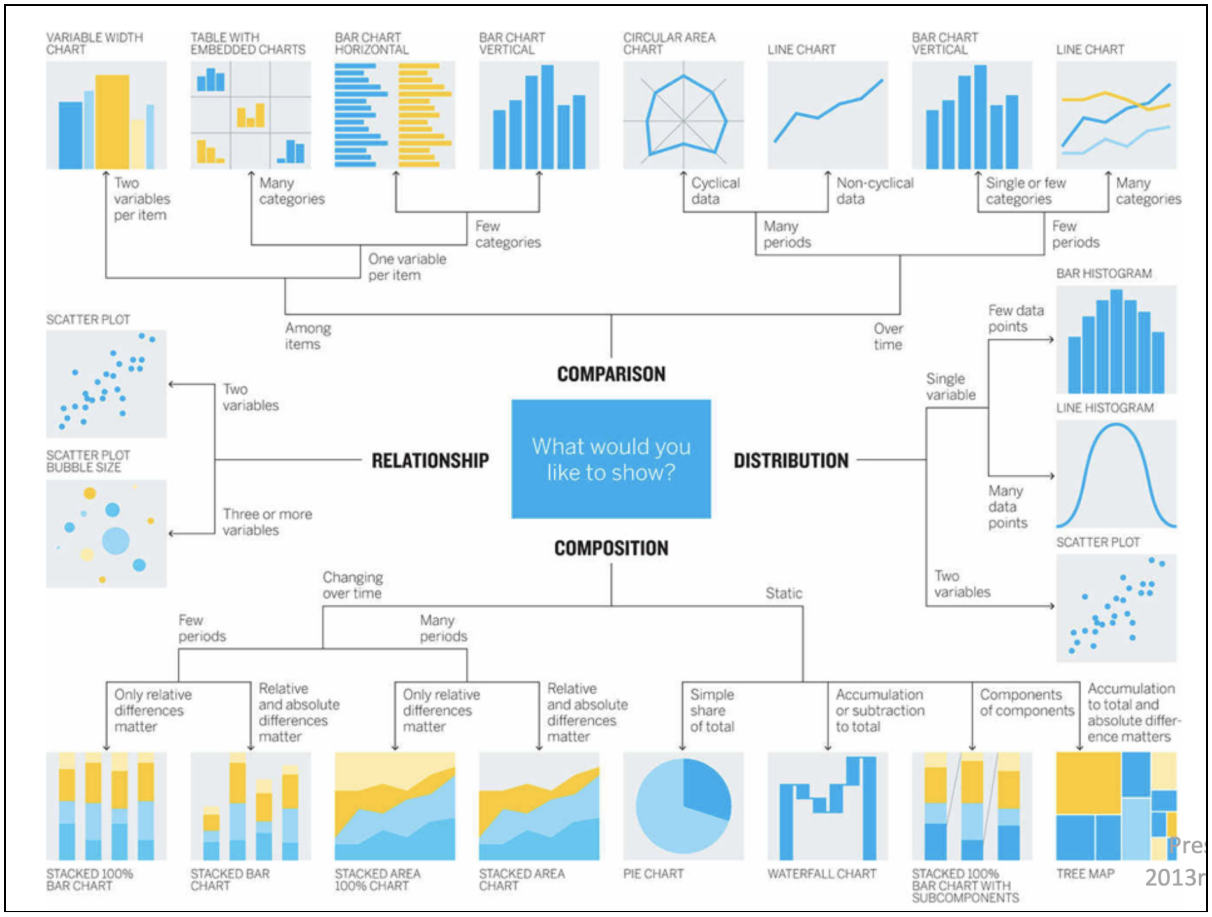


Figure 2: Charts of visualizations by relationships [8]

[7] Evanthia Dimara, Charles Perin. What is Interaction for Data Visualization?. IEEE Transactions on Visualization and Computer Graphics, Institute of Electrical and Electronics Engineers, 2020, 26 (1), pp.119 - 129. ff10.1109/TVCG.2019.2934283ff. ffhal-02197062

[8] Andrew V. Abela, Advanced Presentations by Design, 2013 Redrawn by Berinato. Good Charts

We started to converge around the idea of how we might translate user intent into a series of specific data gathering requirements to map it to a set of recommended visualizations. After consulting with Professor Hearst, we agreed that the prototype could be limited to the scope of Distributions (center right section of the diagram) because a) the visualizations under distributions are all visualizations supported by Lux's current capabilities and b) any interface interaction introduced anew could be compared against Lux's current Distribution tab feature.

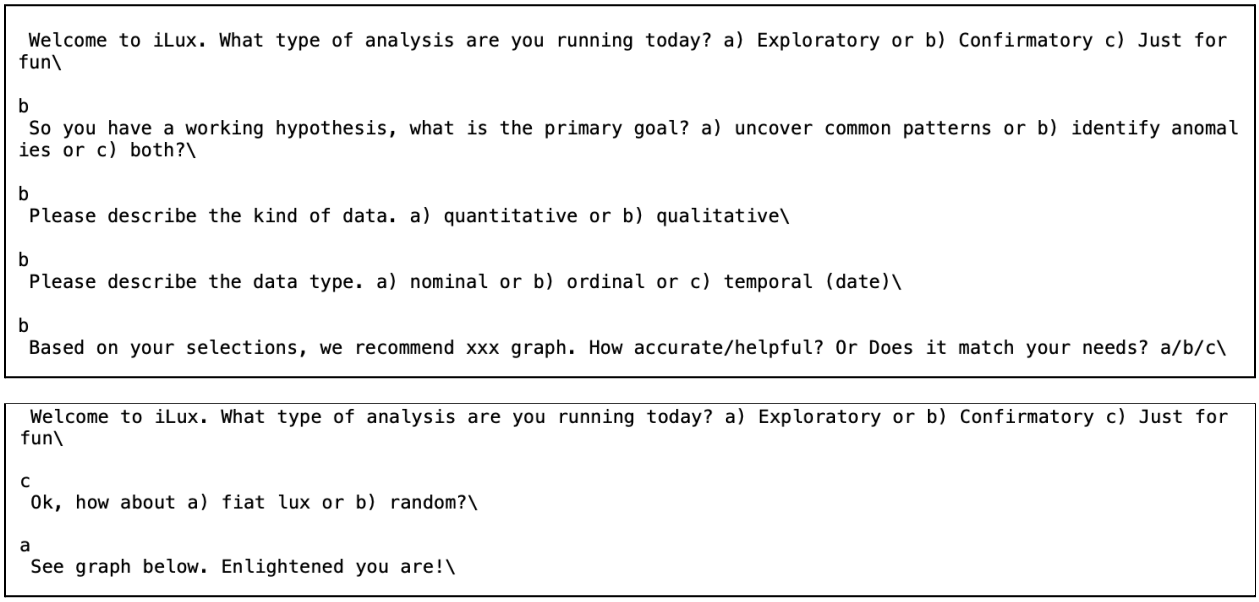


Figure 3: Interactive Text-based Python Notebook

For the first prototype, we developed a text-based questionnaire in a python notebook to gauge how the newly broadened set of target users would respond to an elongated format and progressive approach to EDA with guided prompts. During this stage, the goal is less about identifying technical gaps of the intent's path to visualization generation, and more about determining if the questionnaire is a feasible and efficient way to create mappings of visualizations. The content of the questions are adaptations from courses lectures of INFO 247 for what makes up a visualization and meant to be placeholder content until mapping of questions to visualizations are finalized.

Prototype 2 - Embedded into Lux's current notebook Interface

Using the same set of questions in the earlier prototype, we wanted to find a way to integrate them seamlessly with Lux's current python notebook environment. To this end, we developed the questionnaires nested in Lux' current tabs and leveraged Lux's existing design elements. Ensuring that we stay consistent with Lux's current design language, we integrated the

questions using the ipywidget[9] that Lux is developed from. At this stage we were primarily interested in how users would respond to this approach of using a questionnaire, and identifying this method's potential limitations.

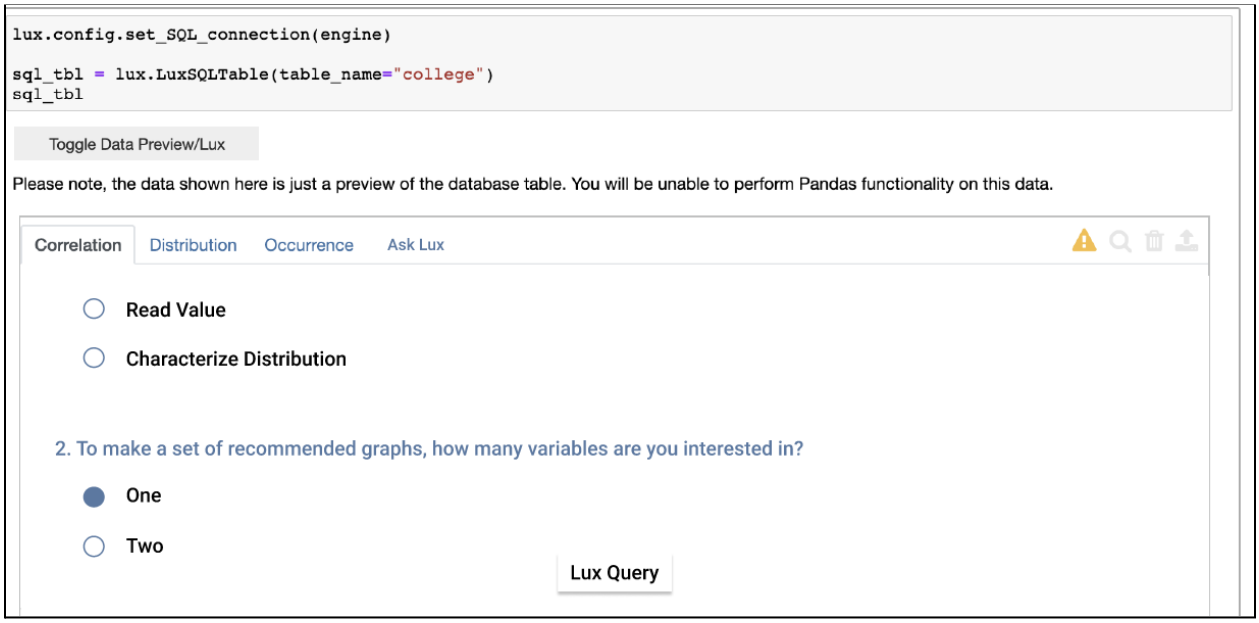


Figure 4: Interactive Questionnaire Embedded into Lux's System

P1 and P6 expressed how they like how natural it feels to see the questionnaire embedded in Lux's current design system. P1 stated he would prefer to have the questions stated with a more conversational and friendly tone; more specifically, natural language forms are good references to build from.[10]

Some of the general concerns that we learned regarding this prototype were that the scope of variables which users could declare may be too limited and might require laborious repetition of form-filling for multiple sets of visualizations interested. For example, if they are interested in creating multiple visualizations for more than 3 variables and analyzing multiple comparisons, they would need to complete the questionnaire a few times from the beginning to end. To lessen the repetitive workflow, this would require more time and effort spent upfront on the users' part to formulate smart hypotheses and design more complicated levels of data modeling, rather than relying on Lux's automation or abstraction methods. Another limitation of this prototype was the treatment of user control and refinement mechanism after the set of recommended visualizations were generated and when it failed to match the user's initial expectations. Users

[9] https://ipywidgets.readthedocs.io/en/stable/
[10]https://wpforms.com/natural-language-form-examples/#:~:text=A%20Natural%20Language%20Form%20is,human%2C%20and%20people%20love%20that

wonder if and how will they be able to refine the sets of visualizations presented in front of them afterwards. In addition, users felt uncertain about how many of the questions could be made optional and populated automatically to reduce steps to completion when the first few questions have been answered earlier.

Prototype 3 - Complete User Flow

For this version, we incorporated formats of natural language forms with new interaction components, such as text fields or dropdowns, to improve ease of use while completing the user flow with the generation of a set of visualizations to select from. We also translated each fork of decision-tree-like questions from Abela's diagram of Advanced Presentations by Design and mapped each path of data relationships into data visualizations through a series of questions.

Mapping of Questions to Visualizations

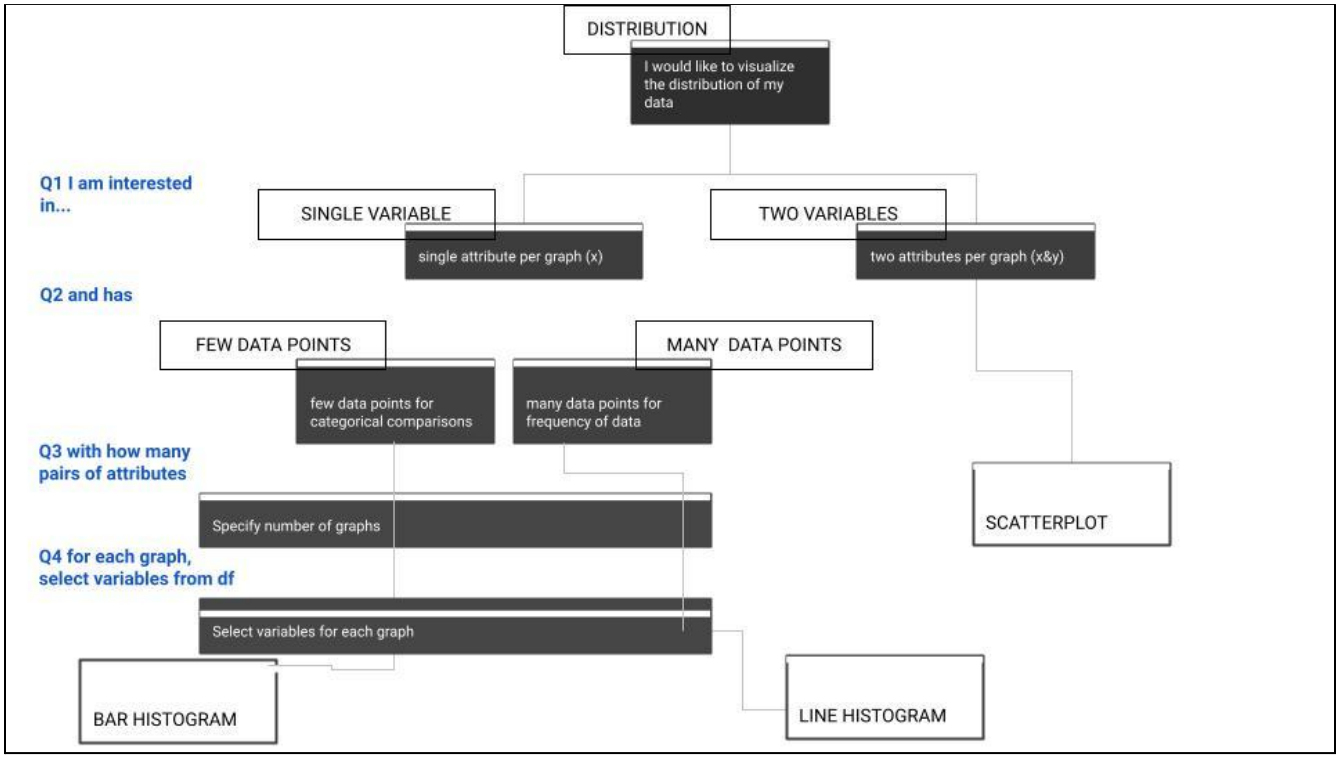


Figure 5: Diagram Mapping of Questionnaire to Visualizations

The diagram above is the mapping of questions to retrieve visualizations:

- The questions worded in the questionnaires are labeled in blue on the left
- The boxed grey labels are from Abela's diagram of Advanced Presentations by Design
- The grey background labels are the selected inputs for each question in the questionnaire

Both Figure 6a and 6b are developed from the diagram above with different interface and interaction controls within the same context of the questionnaire.

Version a) the form uses complete-the-sentences format with dropdown menus and text field

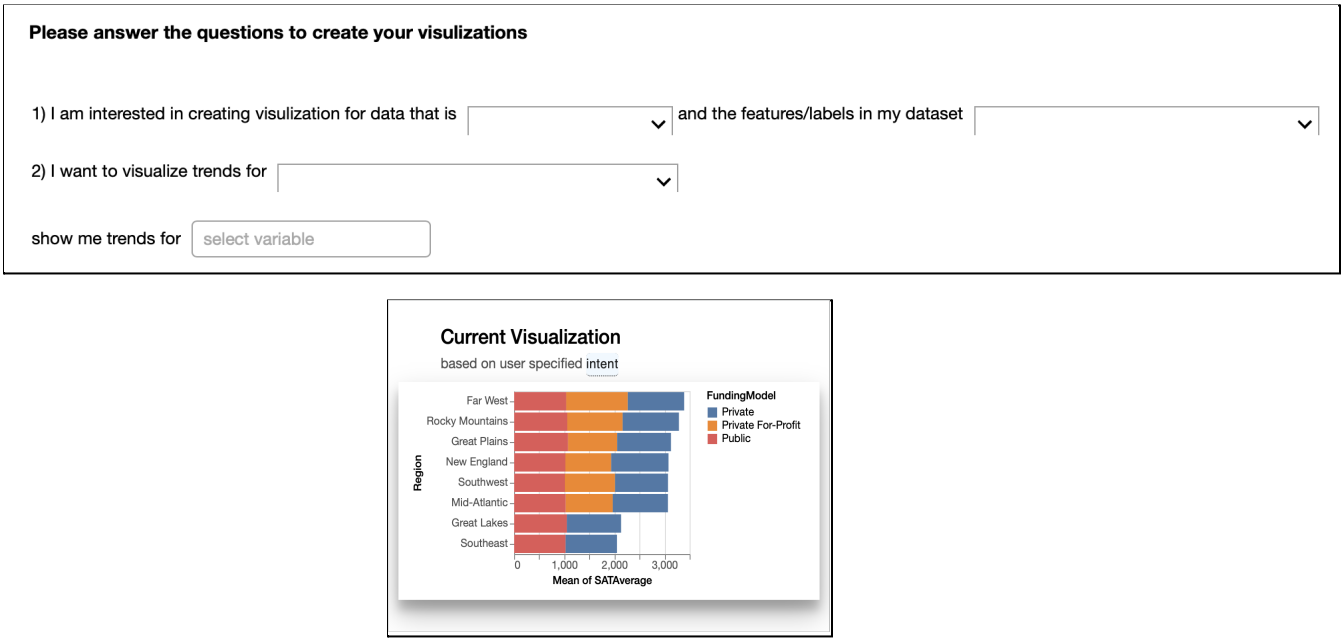


Figure 6a: Complete Interactive Questionnaire (Horizontal View)

Version b) the form uses a standard survey format with display of options upfront/ pre-populated in its default state with multiple choice views, text fields, and a numeric input stepper

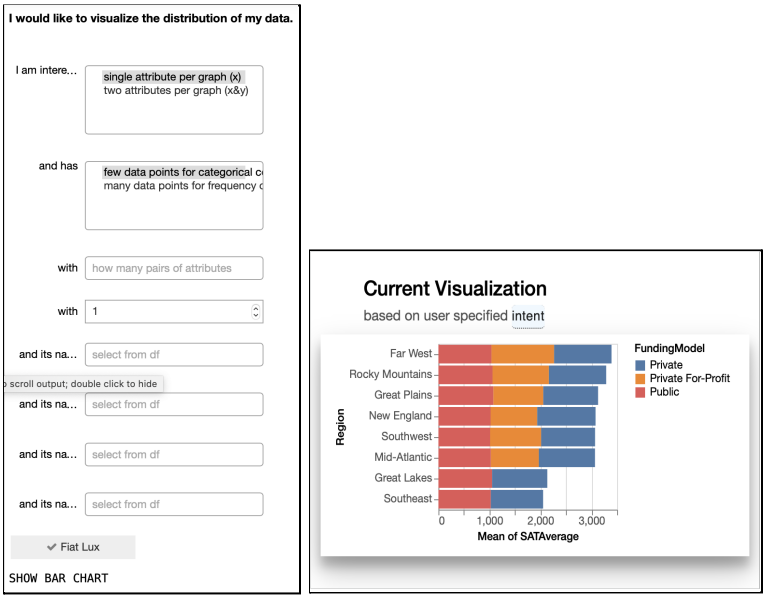


Figure 6b: Complete Interactive Questionnaire (Vertical View)

From initial results of three user tests, two out of the three users prefer prototype version 3, and each chose figure 6a and figure 6b respectively as their preferred method of EDA.

Both highlighted that these sets of questions force them to slow down and ponder more critically about what hypothesis or relationship between data they were looking for before interacting with the questionnaire. From a workflow perspective, all three users remarked that this questionnaire assumes the user has already performed common operations of EDA using panda functions and gained an overall understanding of its descriptive summary statistics, data types, or percentage of missing data.[11] They cautioned this is not always the case before doing EDA and creating visualizations, primarily when there are many steps to data pre-processing, including data formatting, wrangling, transformation, grouping, filtering, and addition of newly created attributes that could be used in tandem with a quick view of visualizations. Typically, they make iterative decisions to keep, clean, reorganize, or remove data either before making visualizations or during the visualization generation process to identify missing gaps and detect anomalies of the dataset. During this initial phase, the goal is to determine what are the key variables to examine and develop intuition for the types of relationships between variables. In terms of clarifying intent specificity and technical constraints of Lux's current capabilities, users suggested a few missing features from Lux currently:

- Boxplot or violin plot visualization with descriptive statistics (P1)
- Quickly specify data for formatting and easier ways to handle data wrangling or pre-processing or modeling, especially ease of converting temporal data and time formats from Trifacta (P1, P5, P6)
- Transparency and user control of ranking algorithm for correlation, distribution, occurrence, and temporal visualizations, i.e. through code declarative statements or addition of user interaction components (P1, P5, P6)
- Pre-emptive alert notification after the dataframe uploads to identify potential areas where issues may arise, i.e. misclassified data types between strings and numbers (P5)

All users said regardless of the horizontal or vertical view of the questionnaire, they like how they could see the entire set of questions in a single view in its entirety. They also said the expectation of output is clear; they will receive either one or multiple visualizations when they finish filling out the questionnaire. Although there are no specific comments about the length of

[11] Top 20 Pandas Functions which are commonly used for Exploratory Data Analysis. https://medium.com/analytics-vidhya/top-20-pandas-functions-which-are-commonly-used-for-exploratory-data-analysis-3cb817a60f46

the questionnaire, all gave general direction about finding the right balance between the length of the questionnaire and the utility and natural tone of each individual questions such that users can expect to respond only to the first few questions with built-in autodetection or auto-population for latter questions. Two out of three interviewees suggested this may be an easier way to onboard users who are less familiar with coding in their jobs though they may be savvy enough to figure out basics and are already comfortable enough to be in a python notebook environment. For the horizontal scrolling window to see resulting visualizations generated, all users suggested they would like to see more visualizations presented without scrolling right and left, and prefer to see as many visualizations retrieved with the option to see more if it does not fit into the size of the current window.

Prototype 4 - Combining Question-and-Answer Approach with Visual-based Interaction Method (Attached Python Notebook)

From prototype version 3, we also wanted to experiment with a visual interface that provides the user instant feedback and validation in real-time, alongside the question-and-answer style formats of the prior prototypes.

Based on the user tests, none preferred this prototype. Both P1 and P6 commented how the look-and-feel is similarly comparable to Tableau.

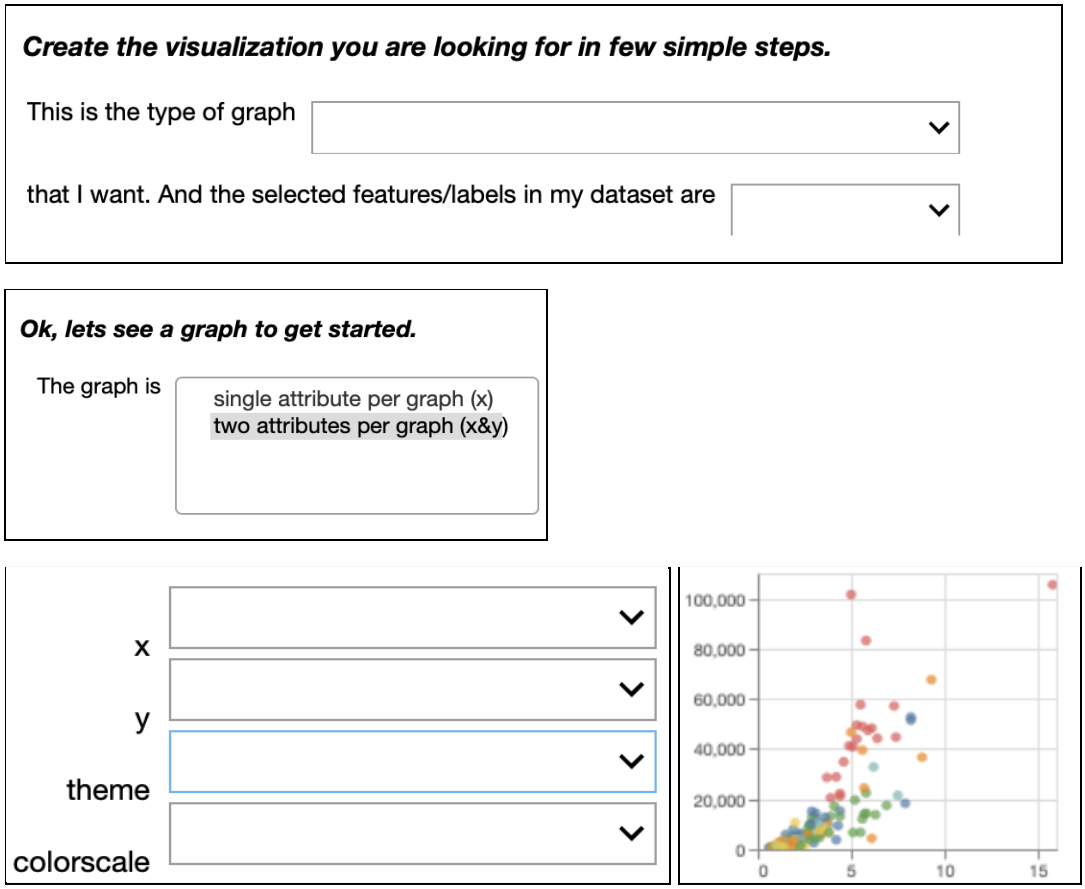


Figure 7: Combined Methods of Interactive Questionnaire

Key Takeaways on New Intent Prototype

Overall, all users expressed excitement about the questionnaire approach to EDA. Among the highlights, its seamless integration into Lux's existing design systems, intuitive interactions, and series of natural language questions mapped to paths of visualization generation offered another complement to strengthen Lux's adoption usage beyond traditional circles of data scientists. More importantly, users' expression of intent have become much more defined, from specifying the variables of interest to formulating sound hypothesis structure to orchestrating multiple-comparison modeling even before they begin EDA, afforded by the sequence of guided prompts. While we did not find out how the new questionnaire approach fared over comparison with the current auto-generated sets of visualizations underneath the Distribution tab, we learned major areas of improvements that could be incorporated into future enhancements:

1) Increase variety of types of visualizations that are critical and common to EDA i.e. Boxplot or Violin plot.

2) Simpler handling method or streamlining of data preprocessing and wrangling helps users get to making visualizations faster. Widespread frustration with unavoidable, conventional, and repetitive operations for multiple variables within the same dataset is a huge pain point for EDA**.** To double-down on Lux's original value proposition of fast and simple EDA, it may be valuable to look into conventional operations of EDA and display summary statistics alongside visualizations.
3) Greater transparency and user control on algorithmic rank with code statements or interactive refining components. Some proposed solutions were more details for calculation of the interestingness function and its equation, or slider of the range of numeric values of the function.
4) Better navigation and larger display window to see resulting outcomes of visualizations in a single view, and with the option to see more, e.g. an accordion signifier that opens up to more visualizations when they don't fit in a single view.
5) Automatic detection of data types and communication of potentially problematic subsets of data upfront.

Backend Engineering

The core motivation behind extending Lux to support Database querying out-of-the-box was to provide users the ability to work with larger datasets that cannot be loaded to in-memory storage. Another use-case we considered involved users working with secure data in the industry that was available to employees for aggregate level data analysis but not for download/import into their local machines/Jupyter Notebooks. To make Lux compatible with these use cases, we created the SQL Executor, which performs all of the data aggregation functions necessary to run Lux' recommendation system on the database end. Our next step was to give users a smoother experience with the best possible performance from the Lux side by optimizing the code base, and adding new features based off of user interviews.

Lux SQL Executor

The core of the technical portion of our Capstone project revolves around the SQL Executor that we have created. This executor engine acts as the interface layer between the Lux recommendation system and the Database server. It allows Lux to automatically query and format database data that is required for the rendering of visualizations. Devoid of the ability to

import data locally and use Pandas functions to aggregate and format, as is the case with the Pandas Executor, we require the SQL Executor to map and fire equivalent SQL queries on the Database server to gather the data required for these functions.

The idea is to offload the required aggregation operations to the SQL Database Server - a system designed and equipped to work with that scale of data - while orchestrating the querying, recommendation, and visualization generation on the user's local machine. The SQL Executor is at the heart of this orchestration as it sequentially fires queries to obtain the necessary (aggregate) data from the Database server. Post the data gathering, Lux can generate and recommend appropriate and potentially interesting visualizations to the end user.

The SQL executor automatically fills in the queries with necessary parameters; selected column names, table name, filters, etc; and fires them to pull aggregated data from the database. This data is then formatted to work with the Lux visualization renderer, and used to create specific relevant visualizations. In order to work seamlessly with the rest of the Lux recommendation system, we designed the SQL Executor to take the same inputs and release the same outputs as the original Pandas Executor. This allowed us to essentially plug in the SQL Executor in place of the Pandas Executor, without making major changes to the rest of the code base.

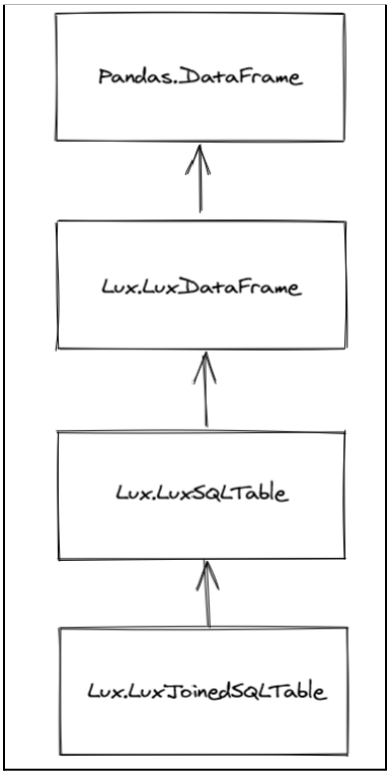
LuxSQLTable Object

Originally the SQL Executor was intended to work with the already existing LuxDataFrame class, which is a child of the Pandas Dataframe with Lux' recommendation system and interactive widget wrapped on top. To use Lux' SQL functionality, users would set a database connection within the Lux configuration and create a new LuxDataFrame object associated with a specified database table or view allowing them to use the Lux recommendation system to explore their database.

On some testing however, we found that the affordances provided by the SQL Executor would have to differ greatly from those provided by the Pandas Executor. When working with data originally residing in a Dataframe, users expect to be able to manipulate and alter the data. There are a number of data transformation and manipulation capabilities that are provided natively by Pandas on its Dataframes. On the other hand, these operations are not always supported by SQL Databases. Hence, when using the SQL Executor, users will not be able to perform the same data wrangling operations considering the data is not stored locally and instead resides on the database system. Thus, the LuxDataFrame objects would need to

support Pandas specific capabilities when in use with the PandasExecutor and silence these capabilities when in use with a SQL Database. This implementation would be unintuitive, and to avoid greater confusion for the end-user we decided to have a clearer demarcation through the creation of a new class.

To better differentiate between the Pandas Dataframe and SQL database use cases, we created a new LuxSQLTable class. The LuxSQLTable object uses the SQL Executor to collect data for visualizations, while the original LuxDataFrame uses the Pandas Executor. Separating these two classes makes it easier for us to control what functionality users have access to according to the context of their work.



Internally, LuxDataFrame, the core object behind all of Lux' core features is a child class with Lux' recommendation system and interactive widget wrapped on top of it's parent - the Pandas native Dataframe. The LuxDataFrame was thus the foundation of the LuxSQLTable, which inherited many of LuxDataFrame's core functionalities. This design allowed for code modularity and reuse while allowing us to override and hence silence other pandas specific functionality that we needed to subdue.

To use Lux' SQL functionality, users would set a database connection within the Lux configuration and create a new LuxSQLTable (a subtype of LuxDataFrame) object associated with a database table or view that they specify. They would then be able to use the Lux recommendation system to explore their database but not Pandas specific functionality.

Figure 8: Class Hierarchy Diagram for Lux Data object

Additionally we made changes to the initial interface of the LuxSQLTable object to provide a better user experience. While none of the database data is stored within the object itself, we wanted to keep providing a preview similar to the Pandas head() function call so that users can get a sense of their database data. On top of this preview, we also added a message indicating the table name and database connection details that the LuxSQLTable preview corresponds to, allowing users to know that this is a Database-specific object that won't accept any data modification (as the LuxDataFrame does). These changes can be seen in the figure below.

```
import psycopg2
connection = psycopg2.connect("host=localhost dbname=testdb user=testuser password=testpass")
lux.config.set_SQL_connection(connection)

tbl = lux.LuxSQLTable()
tbl.set_SQL_table("employee")
tbl
```

Toggle Table/Lux

Here is a preview of the **employee** database table: **localhost:5432/testdb**

	Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField	EmployeeCount	EmployeeNumber	...	Relationsh
0	41	Yes	Travel_Rarely	1102	Sales	1	2	Life Sciences	1	1	...	
1	49	No	Travel_Frequently	279	Research & Development	8	1	Life Sciences	1	2	...	
2	37	Yes	Travel_Rarely	1373	Research & Development	2	2	Other	1	4	...	
3	33	No	Travel_Frequently	1392	Research & Development	3	4	Life Sciences	1	5	...	
4	27	No	Travel_Rarely	591	Research & Development	2	1	Medical	1	7	...	

Figure 9: Preview interface when using the SQL Executor

Optimizing/Stabilizing Lux Performance

Lux's performance speed when using the SQL Executor is directly proportional to the time it takes to run a query, which increases proportionally to the sizes of the database tables we connect to. While we had tested the robustness and performance of our SQL Executor on a number of databases of varying sizes, we were aware that we need the system to be as optimal as possible for industry-scale databases that can reach billions of rows and multiple terabytes on occasion. Considering the sizes of industry-scale databases, there was a need to find ways that allowed us to keep our performance optimal as the database sizes increased.

We took two steps to reduce wait times for users. First, we did some preliminary benchmarking to compare the performance of the LuxSQLExecutor v/s the PandasExecutor. Doing this helped us pinpoint some areas that could be most easily improved. Once we had identified the areas which could be improved quickly, we streamlined the SQL Executor so that it could gather the data required for maintaining Lux' recommendation system with greater efficiency.

We started by analyzing the number and performance of the individual queries used in supplying the necessary functionality to our system. There were a number of avenues we identified for improvement.

1. The first step was to minimize the number of database round trips that were made or, in other words, the number of queries fired. We worked here to identify ways to retrieve the

required data or metadata by combining multiple queries into one. We went through the whole workflow to also identify places where duplicate data reads were being made – moving forward to eliminating them by reusing the available data. We have more on this in the upcoming section.

2. Next, we started the runtime analysis of individual queries to identify possible optimizations. Individually on each query, these optimizations might yield small improvements but collectively, these optimizations should be able to make a noticeable difference for the users that use Lux to work with industry-scale databases. Again, more on this in the upcoming section.

Following the optimization on the query side, we went back to the application code to analyze the algorithms and data structures that were put in place for the proof-of-concept and initial prototype. Looking through the code, we replaced various data structures and fine tuned algorithms to also improve performance in that regard through the asymptotic analysis and comparison of the performance of data structures in use in our application code.

Then, we worked on extensive testing of our code. We searched for, and imported databases with varying sizes, data types, and designs to check both the performance and robustness of our implementation. This phase of testing helped us identify a sizable number of bugs that had so far managed to slip through. Before the initial release, we resolved these issues and bugs to ensure the best possible experience for new users.

Benchmarking and Streamlining

At the beginning of our Capstone, the PostgreSQL executor was written as a proof-of-concept and had certain inefficiencies that we worked hard to identify and streamline. We aimed to develop an effective benchmarking solution to quantify the progress we made in terms of making (and keeping) Lux highly robust and performant. These benchmarking exercises allowed us to identify key opportunities to make the system more efficient.

To benchmark Lux when using the SQL Executor we used variously sized database tables created by sampling an Airbnb dataset to 500,000 to 2.25 million rows with a 250,000 row increment. This data contains sixteen columns with a good variety of data types; qualitative, quantitative, date-time. For each differently sized dataset we created a different Postgresql database table, connected each table to Lux, and measured the time it took for Lux to compute

the metadata required for recommendations and the time it took to generate an initial set of visualization recommendations.

There are two main computational components to the Lux recommendation system, first is the collection of dataset metadata which will be used by Lux, and next is the actual visualization creation and analysis for the recommendation system. After performing the initial benchmarking we see that the SQL Executor performed significantly worse than the Pandas executor in both of these areas. This was to be expected since querying a database incurs a significantly greater overhead cost compared to local pandas operations due to server round trips, query processing overheards, etc. With this in mind, we wanted to improve performance by reducing the total number of queries performed.

During the metadata collection stage, there was a good amount of improvement to be made on the SQL Executor. This stage involved the system firing SQL queries to gather the following information:

- Unique values for each table column
- Cardinality for each table column
- Min and Max values for each quantitative table column

For these required steps, we were able to identify some opportunities for optimizations. For instance, both the cardinality and min/max values of a column could be derived from a table's list of unique values. Thus rather than querying the database again for all of these, we can get away with just using querying for the unique values per column. We then used python code to derive the cardinality and min/max values for each column. Since this essentially reduced the number of queries performed in this step by a third, the run time for metadata collection when using the SQL Executor sped up considerably and even became comparable to the performance of the Pandas Executor as shown in the graph below.

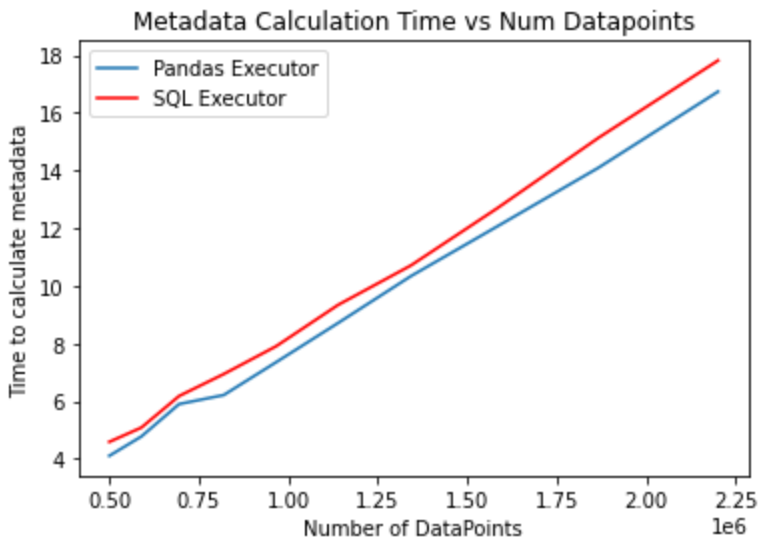


Fig 10: Comparison of Pandas and Postgresql Executor time to calculate metadata vs the number of datapoints in differently sized Airbnb datasets

While we were able to address the issues with SQLExecutor's metadata gathering to make it on par with the Lux Pandas Executor, there was still potential for improvement on the recommendation generation side. As shown in figure 11 below, the amount of time it takes to generate recommended visualizations when using the SQL Executor is significantly higher than when using the Pandas Executor.

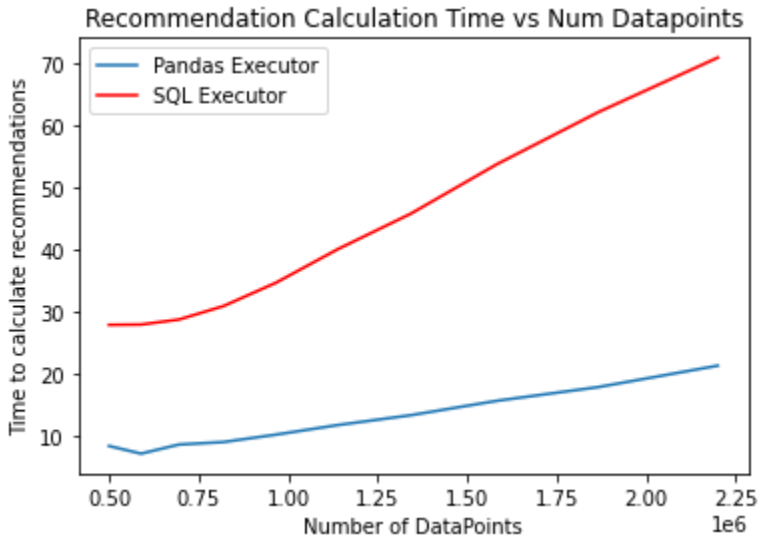


Fig 11: Comparison of Pandas Executor and Postgresql Executor time to calculate recommendations vs the number of datapoints in differently sized Airbnb datasets

From our benchmarking we know that the time to query data for charts using the SQL Executor is several times longer than the time it takes to gather the same amount of data when using the

Pandas Executor. Furthermore, this held true across all sizes of datasets we used in our testing. Thus the performance difference between executors may likely be attributed to the additional overhead incurred while querying a database server, rather than an in-memory dataframe. However there is some more testing we can do in the future to verify this claim. We would like to perform granular benchmarking exercises to verify if running SQL queries is the main bottleneck to better performance. If this is the case, there are some possible directions that we can explore in the future. These solutions are discussed more in the Future Work section of our report.

New Lux Features

We also worked on a set of new features that were informed through a mixture of usability studies, user interviews, and internal testing.

Improved Datetime Datatype Detection

The first feature to be picked up was the implementation of a new, more robust, datetime data type detection algorithm. This came as a result of one of our internal testing rounds. We observed that, for some databases, our system was not able to correctly identify the columns as a “Datetime” type – the default datatype for temporal data - in SQL. Our initial debugging indicated a need for a better detection algorithm to work with more databases that might have the datetime formatted differently. We worked on a new regular expression-based algorithm in a bid to make the data type detection more robust. However, post implementation, we continued to see failures. On closer inspection, we discovered that the Postgres system actually stores all Datetime data type columns in a single format internally. It is at read-time that these values are transformed to the format dictated by user preferences. These failures, we discovered, were hence a result of incorrect database design and configuration issues outside of the scope of Lux and not algorithmic issues on Lux end. This feature was abandoned due to these findings.

Joins on Multiple Tables

Based on collective prior work experience, the team discussed a hypothesis around making the Lux SQL compatibility more rewarding for users. The best practices around the use of Relational Databases dictate that the data be scattered over multiple tables that are ‘related’ by particular column(s). Thus, we hypothesized the following about the use of Relational Databases in the industry: a lot of exploratory data analysis and insight gathering from data stored in relational databases comes from across multiple tables in the databases and not just

one table (as the then current version of Lux supported). We decided to validate this hypothesis through user interviews with industry professionals experienced with Relational Databases.

In multiple interviews we conducted, we were able to successfully validate our hypothesis. Users mentioned how JOIN functionality was essential to their work, and that they would prefer to perform basic SQL JOIN functions through Lux . Having this functionality would make the API easier to use within their workflows. Our interviews helped us finalize the high-level requirements with regards to the JOIN functionality and we decided to move forward into building a proof-of-concept. We started by having an internal discussion focused on scoping the functional and technical requirements of such a feature. For our approach, we decided to create a view within the Postgres Database that consisted of user specified tables joined on users' supplied conditions.

We created a brand-new class 'LuxSQLJoinedTable', a child of the LuxSQLTable class to:

1. allow users to be able to differentiate between the two.
2. allow us to build methods for LuxSQLJoinedTable on top of LuxSQLTable's existing methods (which can be inherited)

This approach allowed us to utilize Postgres' existing affordance of treating views as a named query for use in a syntax that is synonymous to actual tables.

The first question this brought up then, was around the user-interface we provide for this feature. At the onset, we experimented around setting up another class of objects to signify "JoinType" (InnerJoin, LeftOuterJoin, RightOuterJoin, and FullOuterJoin) that could be used by the user to supply the required table names and conditions to the system synonymously to how the Clause object does for Vis. The user can create and supply multiple of these JoinType objects as arguments to the LuxSQLJoinedTable object constructor. Each JoinType object constructor would, in turn, take as arguments a pair of table_name.column_name to join on appropriately - based on the type of the JoinType object itself.

```
import lux
import psycopg2
import pandas as pd

connection =  psycopg2.connect(DB_CONNECT_STRING)
join_1 = InnerJoin(left_table = "table1_name", left_column = "col1_name", right_table = "table2_name", right_column = "col2_name")
join_2 = FullOuterJoin(left_table = "table1_name", left_column = "col1_name", right_table = "table3_name", right_column = "col3_na
join_3 = LeftOuterJoin(left_table = "table2_name", left_column = "col2_name", right_table = "table4_name", right_column = "col4_na
joinedDf = lux.JoinedSQLTable(joins = [join_1, join_2, join_3])
```

Figure 12: Code Interface with multiple JoinType support

After setting this up and performing some experimentation, however, this interface proved to be too verbose – code-wise – for end users. On discussion, we decided to limit our functionality to InnerJoin alone. Doing this enabled us to do away with most of the verbosity in the code – simplifying it greatly – whilst retaining the majority of the value that (inner-) joins provide in data analysis. The user should be able to supply pairs of table_name.column_name directly to the LuxSQLJoinedTable object separated by the “=” operator, allowing the system to perform a Join on these specifications.

```
import lux
import psycopg2
import pandas as pd

connection =  psycopg2.connect(DB_CONNECT_STRING)
joinedDf = lux.JoinedSQLTable(joins = ["table1.col1 = table2.col2", "table2.col2 = table3.col3", "table2.col2 = table4.col4"])
```

Figure 13: Simplified code interface supporting InnerJoins only

Lastly, we decided to limit the number of tables that can be supplied to each LuxSQLJoinedTable object to 4 (Four) distinct tables (i.e., 3 joins) for initial performance benchmarking and proof-of-concept reasons. We will be studying this prototype, writing in-depth unit and functional test-cases to help dictate future design decisions around these limitations.

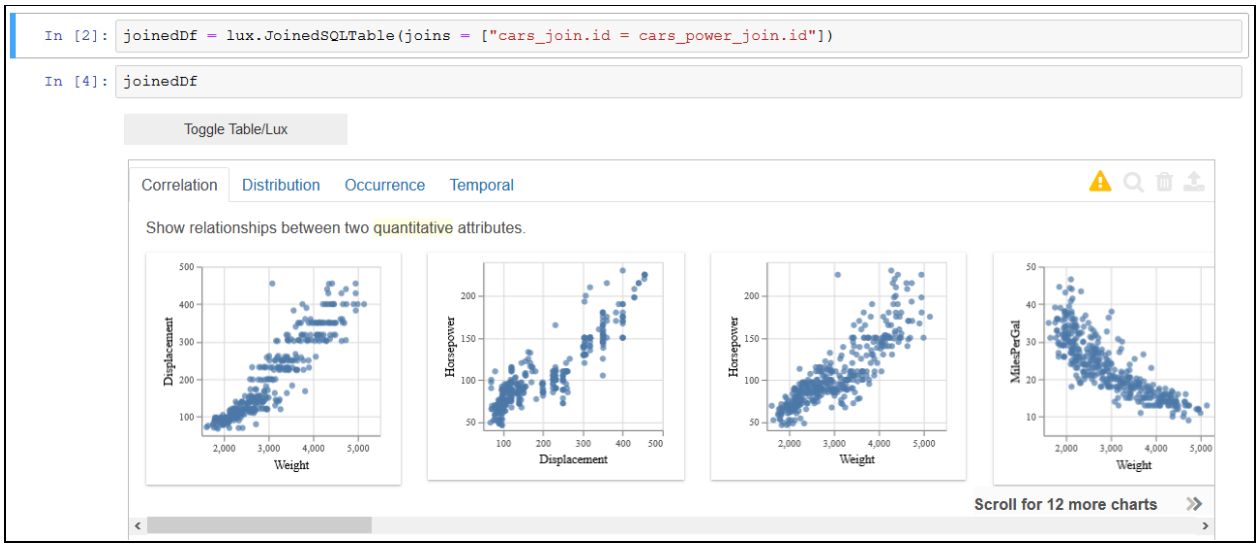


Figure 14: The Lux Widget displaying generated graphs on joined tables

An additional point of future consideration that we will be looking to address is regarding access rights of our end-users with respect to the database. With the goal of ensuring an optimal user experience in mind, we have a number of directions that we wish to explore around these considerations before we will be able to include this feature for a general release. More on these potential opportunities in the Future Works section.

Lux Code Exporting

From our user interviews we received feedback that users want greater transparency in how their data exploration tools aggregate and use data. When connecting to databases in particular, users P1 and P5 mentioned how they would prefer to know more about how tools like Tableau query their databases. With this information, users could re-use the database queries elsewhere or customize them to aggregate other data that they may want to use. To address this, we have implemented a code tracing tool that will help expose the SQL queries and pandas code that Lux uses to aggregate data for its visualizations.

This code tracer uses the Python inspect module[12] to keep track of what lines of code are run when Lux tries to execute an individual visualization. When a vis is generated the tracer keeps track of the python code run in the Lux Executor, stores this information, and processes it into a readable format. A user is then able to view this code by using the to_code() function of a Vis object as shown in the figure 15 below:

[12] https://docs.python.org/3/library/inspect.html

```
vis = sql_tbl.recommendation['Correlation'][0]
print(vis.to_code(language = "python"))
```

```
from lux.utils import utils
from lux.executor.SQLExecutor import SQLExecutor
import pandas
import math
tbl = 'insert your LuxSQLTable variable here'
vis = 'insert the name of your Vis object here'
filters = utils.get_filter_specs(view._inferred_intent)
attributes = set([])
for clause in view._inferred_intent:
    attributes.add(clause.attribute)
where_clause, filterVars = SQLExecutor.execute_filter(view)
length_query = pandas.read_sql("SELECT COUNT(1) as length FROM {} {}".format(tbl.table_name, where_clause),lux.config.SQLconnec
tion,)
def add_quotes(var_name):
        return '"' + var_name + '"'
required_variables = attributes | set(filterVars)
required_variables = map(add_quotes, required_variables)
required_variables = ",".join(required_variables)
row_count = list(pandas.read_sql(f"SELECT COUNT(*) FROM {tbl.table_name} {where_clause}",lux.config.SQLconnection,)["count"])
[0]
query = "SELECT {} FROM {} {}".format(required_variables, tbl.table_name, where_clause)
data = pandas.read_sql(query, lux.config.SQLconnection)
view._vis_data = utils.pandas_to_lux(data)
view._query = query

vis
```

Fig 15: Example output of of the python to_code() function with a LuxSQLTable

This code can be copied and pasted into a Jupyter notebook cell, then run to recreate the original visualization. With this feature users can better understand how Lux generates its visualizations and they can easily repurpose this existing code for their own needs. When using the Lux SQL Executor, users can also call the to_code() function to view the original SQL query used prior to data processing in the following diagram. This query can then be used to perform further analysis on whatever data points are included within a Lux Visualization.

```
vis = sql_tbl.recommendation['Correlation'][0]
print(vis.to_code(language = "SQL"))
```

```
SELECT "Weight","Displacement" FROM car WHERE "Weight" IS NOT NULL AND "Displacement" IS NOT NULL
```

Fig 16: Example output of of the SQL to_code() function with a LuxSQLTable

One downside to implementing code tracing, however, is the additional overhead imposed during the creation of the target charts. Furthermore, since Lux usually generates dozens to hundreds of charts in the process of making recommendations, performing the code tracing incurs a significant amount of time loss for users. To avoid unnecessarily keeping track of and formatting python code for all charts, we instead only run the code tracing when a user selects a specific Vis object and runs the to_code() method. When this occurs, we simply take the Vis object's specifications and re-execute the data with the tracer on. This ensures that the tracer

has no impact on Lux' performance when generating visualization recommendations, but still makes the Lux code transparent to users.

General Database Executor Template

At the current state of the project we have stabilized and released the Postgresql compatible Lux build to production as of v0.3.0[13]. This allows Lux to support connections with Postgresql databases to deliver all of the API's visual recommendation features on top of relational tables. This has been made possible by extending our internal executor to support data querying, aggregation, and analysis on Postgresql databases directly. However users in our interviews also mentioned a desire for greater database and ETL support, as there are a variety of other storage systems that our interviewees use in their day to day functions. From our talks users specified an interest in connecting Lux with systems like Google BigQuery, MySQL, Oracle etc.

To meet these users' needs, one possible solution would be to create a new querying executor for each of the requested database systems. However this process comes with some significant challenges that affect its practicality and maintainability. It would be cumbersome to have a completely different executor object for each potential storage system, and ensuring the stability and robustness of these executors would be very difficult in the future. Instead of making many specific execution engines, we propose a more general database executor that can be made compatible with a variety of relational database systems. The idea behind this is based on the fact that the process of executing a visualization in Lux is generally the same no matter the database backend is. The executor needs to gather the metadata necessary to run Lux' recommendation engine, as well as gather and format data that is compatible with the Lux visualization renderer. Rather than rewriting all of these functions, we can have a more general executor that keeps most of the same code but changes the syntax of queries that it uses.

[13] https://github.com/lux-org/lux/releases/tag/v0.3.0

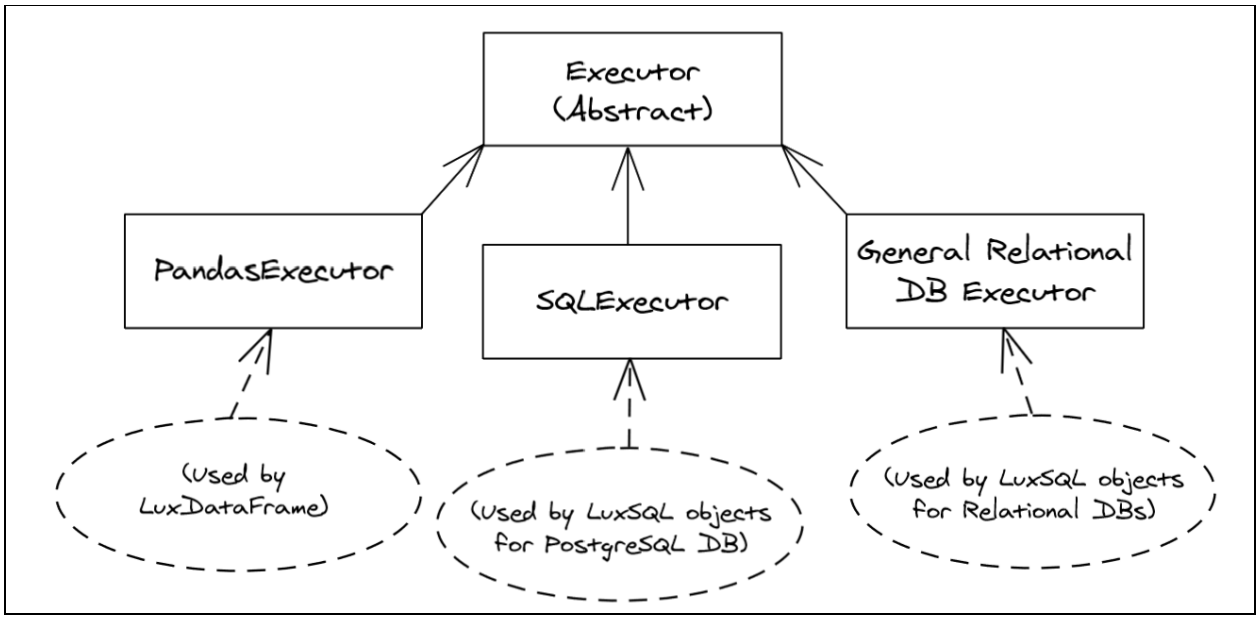


Figure 17: The Executor hierarchy and usage

Instead of hard coding database queries into the executor itself, we propose a system that allows the user to easily supply the necessary queries. The types of queries templates required by this Lux database executor are fully outlined in Appendix A. The user can refer to this template and provide an appropriate ‘query template file’ for their choice of database. Lux can use this external text file as a configuration file.

Once the query template file has been read into a dictionary, the Lux executor can use the dictionary to retrieve and populate whatever query template it needs. These queries do require a specific output format in order to be compatible with existing Executor code. However as long as the outputs of these queries are consistent, then no matter the database system, the existing executor code that is used to format the data for the renderer can remain unchanged. This would make it easier for future Lux developers to connect the API to new database systems and maintain the Lux execution backend. Furthermore, users of Lux could even implement these changes themselves as they would not have to rewrite the entirety of the Lux executor class and simply adjust the syntax of the database queries.

When a user has a new query template, they would be able to read it into the Lux config and use the generalized Database Executor to leverage Lux’ recommendation system on their database. For example, if a user would like to have Lux work with their MySQL database they

could use the GeneralDatabase Executor and specify a MySQL query template like so:

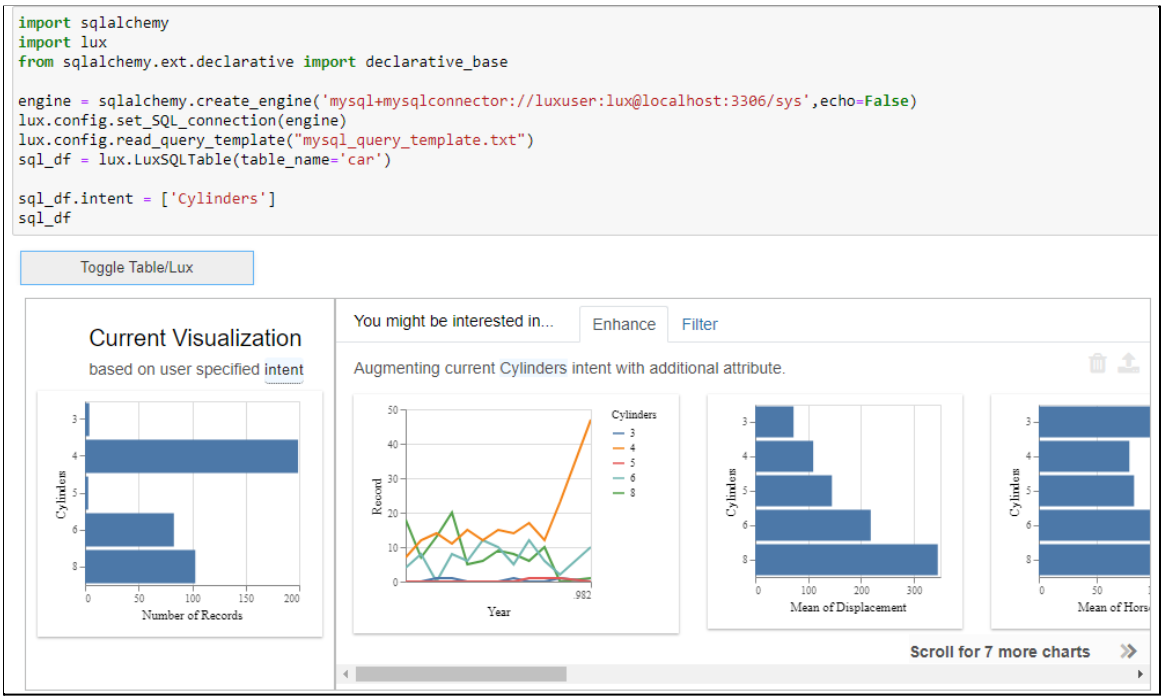


Figure 18: Example of using a MySQL query template with the General Database Executor

Future Work

While the core Postgresql functionality has been rigorously tested and fleshed out, there are still more optimizations that can be explored and other database features to be worked on. As shown in the figure eleven above, the time it takes to generate recommendations when using the Postgresql Executor is significantly longer compared to the time it takes when using the Pandas Executor. In future iterations of the project we would like to explore some options that can further optimize the database use case.

Future Usability Improvements

To further improve the user experience on Lux, future work on usability should prioritize tutorials and make exploratory data analysis easier. It was noted with P5 that even with significant interest and expertise on Lux, he wasn't able to create trend graphs for two variables with his dataset. The current Lux tutorial is formulated in a step-by-step instruction guide. However, there seems to be a need for a more exhaustive tutorial or library of plots providing examples of popular plots with the code that users can replicate; similar to examples provided below the webpage of popular libraries like Matplotlib.

Additionally, every participant indicated significant effort being wasted in exploratory data analysis to manually identify missing values, correct formatting, change features to correct data types, etc. Therefore, this is an opportunity for Lux to identify ways to leverage visualizations to point out the aforementioned inconsistencies in the data. Moreover, tools such as Trifacta have built-in machine learning to specify data for a strange format quickly. For instance P5 said, "*Suppose you need to extract the zip code. Now, if someone just gave you a blob of text, but it's mostly like an address. Trifacta gives you a quick and easy way to understand that this is the city zip code*". In the future, the Lux UX team must study these similar tools to identify the available EDA-specific features and prioritize the subset that can be incorporated into Lux.

In this project, the new intent prototype is focused mostly on evaluating whether a set of questions - coupled with an interactive questionnaire interface - can be used to generate the set of recommended graphs for understanding the features of the data. In the next design phases of the intent prototype, more experimentation with the wording, flow and order of the questions, and the interface or interaction aesthetics on different relationships between data would improve adaptations needed for specific visualization paths and use cases.

In addition, while customizability around the look and subtle details of the visualization, such as color, scale, bin, font size, position of labels, may not be part of the scope of work here. P2, P5, and P6 have all shared an interesting observation: quickly generated templates of graphs take work to achieve a professional level of presentation and pretty graphs take a long time and patience to codify each tiny aspect. In the next phase of the customizability for graphs, both the pane of interaction controls and interface aesthetics of graphs could be areas of opportunities.

Future Optimization Improvements

Data Caching

One method to reduce the overall number of queries that Lux needs to perform is by implementing a Data Caching layer using the Key-Value datastore Redis that works efficiently as a cache. This would allow us to offload the query operations and aggregation to the database and to store the commonly used metadata in the Redis cache to allow repeated efficient access of these computed relations.

However there would still be some trade offs to this implementation. First of all we would need to understand how best to handle the space vs efficiency tradeoff. For the database use case,

we expect users to be working with very large sets of data. This means that it is likely that they would not be able to pull in all or even significant chunks of the data they are working with onto their local machine. Thus to make the cache work, we would have to be careful about how much and which data we keep locally.

Asynchronous queries

In the future we could also explore how to efficiently query the databases in an asynchronous manner. Running all the queries in one go after receiving a user's intent input is inefficient. Once we have information about the tables or columns that a user is most interested in, we can, in the backend, start the aggregation process on the database engine for some basic data points for future use. These data points can be stored in a cache for more efficient retrieval as well. The overall goal here would be to perform querying actions while the user is interfacing with the Lux widget, so the wait time for these queries are less apparent.

Future Database Support

With the general database executor template we have created a more generalizable way to connect Lux to other relational databases. However, there are many other data storage systems being used in industry that do not fit with this approach. NoSQL databases are commonly used, and finding ways to have Lux support these types of systems would make the API a more attractive data exploration option for professionals.

Future Work on Joins

We plan to work on re-visiting the number of tables allowed based on feedback from future testing, usability studies, and user interviews. We have another open question around allowing the recursive joining of one LuxSQLJoinedTable object to other LuxSQLJoinedTable object(s). The ability to join on views is an affordance that Postgres does provide, but we will be revisiting this once we are able to get a handle on the performance of the Joined views currently.

There are also potential roadblocks caused by Database access control. In many cases, especially in the industry, professionals do not have the necessary permissions to fire DDL (Data Definition Language) queries such as CREATE which the LuxSQLJoinedTable class depends on. In such cases, we need Lux to be able to find a work around. We have a potential path in mind we'd like to explore in the future. For example, in instances of such an error, we

could use the supplied join conditions within a subquery that replaces the ‘table_name’ parameter within Lux. This can allow normal operations.

Conclusion

With our Capstone project we hoped to make Lux more accessible to a wider range of data professionals. While the original Pandas use case for Lux is really powerful, users could still be limited by their inability to pull data onto their local machines due to a variety of reasons. This made the API more difficult to insert into database analysis workflows. With our capstone, we have set the groundwork to have Lux be integrated with relational databases allowing industry users to leverage the visual recommendation system.

For our team, this project has brought together much of the interdisciplinary teachings we have learned throughout the MIMS program. Not only did we work to implement technical software to support Lux’ new database integration and features, we made use of design principles and qualitative interviews revolving around data science to better inform our work. To summarize, our capstone team contributed the following to the ongoing Lux project:

1. Qualitative insights around transparency, customization, handoff and interoperability distilled from interviews of six data practitioners into actionable insights and implementable features.
2. A novel SQL execution backend that automatically queries relational databases and formats data for visualizations, letting users easily create and browse sets of visualizations.
3. Improved interface and additional quality of life features to improve user experience when using the Lux API.

Appendix A

Outline of the query template required to use the Lux General Database Executor. This table describes each of the queries that need to be written and included in the template file.

Query Template	**Query Purpose**
preview_query:	Gets a preview of a database table
length_query:	Gets the number of rows in a database table
sample_query:	Gets a random sample of specified size from a database table
scatter_query:	Gets the data required for a scatter plot
colored_barchart_counts:	Gets the data required for a color bar chart which uses counts as the aggregate
colored_barchart_average:	Gets the data required for a color bar chart which uses average as the aggregate
colored_barchart_sum:	Gets the data required for a color bar chart which uses sum the aggregate
colored_barchart_max:	Gets the data required for a color bar chart which uses max the aggregate
barchart_counts:	Gets the data required for a bar chart which uses counts the aggregate
barchart_average:	Gets the data required for a bar chart which uses average the aggregate
barchart_max:	Gets the data required for a bar chart which uses max the aggregate
histogram_counts:	Gets the data required for a histogram
heatmap_counts:	Gets the data required for a heatmap
table_attributes_query:	Gets the column attributes of a database table

min_max_query:	Gets the min and max values of a quantitative database column
cardinality_query:	Gets the cardinality of a database column
unique_query:	Gets all unique values of a database column
datatype_query:	Gets the datatype of a database column