LUX: Always-on Visualization Recommendations for Exploratory Data Science

[Technical Report]

Doris Jung-Lin Lee, Dixin Tang, Kunal Agarwal, Thyne Boonmark, Caitlyn Chen, Jake Kang,
Ujjaini Mukhopadhyay, Jerry Song, Micah Yong, Marti A. Hearst, Aditya G. Parameswaran
UC Berkeley
dorislee | totemtang | kagarwal2 | thyneboonmark | caitlynachen | cjache | ujjaini | jerrysong | micahtyong | hearst | adityagp @berkeley.edu

ABSTRACT

Exploratory data science largely happens in computational notebooks with dataframe API, such as pandas, that support flexible means to transform, clean, and analyze data. Yet, visually exploring data in dataframes remains tedious, requiring substantial programming effort for visualization and mental effort to determine what analysis to perform next. We propose LUX, an *always-on* framework for accelerating visual insight discovery in data science workflows. When users print a dataframe in their notebooks, LUX recommends visualizations to provide a quick overview of the patterns and trends and suggests promising analysis directions. LUX features a high-level language for generating visualizations on-demand to encourage rapid visual experimentation with data. We demonstrate that through the use of a careful design and three system optimizations, LUX adds no more than two seconds of overhead on top of pandas for over 98% of datasets in the UCI repository. We evaluate LUX in terms of usability via a controlled first-use study and interviews with early adopters, finding that LUX helps fulfill the needs of data scientists for visualization support within their dataframe workflows. LUX has already been embraced by data science practitioners, with over 1.9k stars on Github within its first 15 months.

1 INTRODUCTION

Data science is an iterative, trial-and-error process, involving many interleaved stages of data cleaning, transformation, analysis, and visualization. Data scientists typically use a dataframe library [30, 48], such as pandas [45], which offers a flexible and rich set of operators to transform, analyze, and clean tabular datasets. They manipulate dataframes within a computational notebook such as Jupyter, which offers a flexible medium to write and execute snippets of code; nearly 75% of data scientists use them everyday [13]. In between these dataframe transformation operations, users visually inspect intermediate results, either by printing the dataframe, or by using a visualization library to generate visual summaries. This visual inspection is *essential* to validate whether the prior operations had their desired effect and determine what needs to be done next. However, ***visualizing dataframes is a cumbersome and error-prone process, adding substantial friction to the fluid, iterative process of data science***, for two reasons: cumbersome boilerplate code and challenges in determining the next steps.

Cumbersome Boilerplate Code. Substantial boilerplate code is necessary to simply generate a visualization from dataframes. In a formative study, we analyzed a sample of 587 publicly-available notebooks from Rule et al. [49] to understand current visualization practices. A surprising number of notebooks apply a series of *data processing* operations to wrangle the dataframe into a form amenable to visualization, followed by a set of highly-templatized *visualization specification* code snippets copy-and-pasted across the notebook. Our findings echo a recent study of 6386 Github notebooks [36], where visualization code was the most dominant category of duplicated code (21%). On top of the high cognitive cost when writing "glue code" to go from dataframes to visualizations [17, 62], users have to context-switch between thinking about data operations and visual elements. These barriers hinder exploratory visualizations and, as a result, users often only visualize during the "*late stages of [their] workflow*" [18, 33], rather than for experimenting with possible analyses—which is precisely when visualization is likely to be most useful.

Challenges in Determining Next Steps. Beyond writing code to generate a given visualization, there are challenges in determining which visualizations to generate in the first place. Dataframe APIs support datasets with millions of records and hundreds of attributes, leading to many combinations of visualizations that can be generated. The many choices make it hard for the data scientist to determine what visualization to generate to advance analysis. They receive no automated guidance on what may be valuable visualizations to examine next. While there has been some work on automated visualization recommendation in the context of interactive visual analytics tools [39, 56, 67, 68], targeting identification of "interesting" patterns, trends, or insights, none of this work has impacted typical data science workflows in computational notebooks. The former is easier since datasets are static; in a computational notebook, the dataframes are continuously evolving as data scientists perform data cleaning and transformation operations.

Always-On Visualization Recommendations with LUX. To address the above challenges, we introduce LUX, a seamless extension to pandas that retains its convenient and powerful API, but enhances the tabular outputs with automatically-generated visualizations highlighting interesting patterns and suggesting next-steps for analysis (`https://github.com/lux-org/lux`). LUX has already been adopted by data scientists from a diverse set of industries, and has gained traction in the open-source community, with the number of ***monthly downloads around 280 (with a total of 7943 downloads), and over 1900 stars on Github***, as of March 2021. Multiple industry users have created blog posts or YouTube videos extolling the virtues of LUX [8–10, 24, 47, 64].

Challenges of Always-On Visualization Recommendations. Prior work has examined supporting automatic recommendations of interesting summaries in an OLAP setting, e.g., [31, 37, 52, 56, 60, 61, 68, 70], and automatically picking the right visualization modality, given attributes of interest [41, 42, 59, 67]. However, providing always-on visualization recommendations while data scientists perform ad-hoc exploration of dataframes is non-trivial and presents its own unique research challenges:

What and how do we recommend? Data scientists using dataframes are unlikely to use a visualization tool that causes any disruption to their workflow. How do we make visualization recommendations as easy to peruse as the tabular view provided on printing the dataframe within a computational notebook? What types of useful visualization recommendations do we show? There are lots of visualizations that could be generated on a given dataset.

How do we support dataframe evolution? Unlike traditional visual analytics, dataframes are continually evolving over the course of data science. Operations involving pivots or grouping can drastically change the shape of the dataframe. How do we provide visualization recommendations as the dataframe metadata (cardinalities and data types for columns) is changing rapidly? The cost of updating the metadata and recommendations at every point in a dataframe workflow to keep the recommendations "always-on" is often high.

How can we be informed by the dataframe operations users are performing? How do we ensure that the visualization recommendations are relevant and useful, based on the operations that the users have performed? For example, if a user has just performed a grouping, that is an indication that the group-by column is of interest.

How do we allow users to steer the visualizations they want to see? Simply providing users the ability to passively receive visualization recommendations without any power to indicate their interests to Lux is not useful. How do we allow users to provide their "fuzzy intent" in a lightweight manner quickly and without having to write a lot of code—with the system filling in the gaps as needed?

How do we keep it interactive? Visualization recommendation involves traversing through a large search space of candidate visualizations to select ones that would be most interesting to the user. It is critical to provide interactive feedback—even seconds of latency substantially discourages users from visually inspecting their dataframes altogether. How do we ensure that the overhead of visualization recommendations are not substantial?

How do we allow users to export and edit? Often users want to be able to take the visualizations and further customize it to their needs. How do we enable users to export visualizations and edit it in their favorite visualization specification language?

How do we continue to support the rich `pandas` *API?* How do we provide this experience when continuing to support `pandas`' 200+ operators—without compromising the ease and flexibility of programmatic data transformation and preparation as is done presently?

The Lux Approach. We address the aforementioned challenges in developing Lux. Lux preserves all the functionalities of present-day dataframes, while augmenting the default tabular dataframe view with a toggle button to switch to visualization recommendations. Lux is a lightweight wrapper around `pandas` that intelligently caches and lazily evaluates the metadata and recommendations associated with a dataframe. At any point during the dataframe workflow, Lux offers an intuitive way of visualizing the dataframe. These include metadata and intent-based visualizations common in past visualization recommendation systems, as well as novel dataframe visualizations based on structural (Series, Index) and history information. Lux additionally offers a powerful, intuitive and succinct intent language powered by a formal, expressive algebra that allows users to specify their fuzzy intent at a high-level. Lux implements an intent processing stack that compiles the declarative specification into appropriate visualization mappings. Overall, users can use Lux to quickly compose one or more visualizations, and get visualization recommendations for the next steps in their analysis. A naive implementation of recommendation on top of dataframes can be extremely costly incurring up to 575× slowdown relative to `pandas`. Lux ensures interactive visual feedback through a series of optimization strategies that minimize the overhead incurred on top of a dataframe workflow. Lux adds no more than two seconds of overhead on top of medium-to-large real-world datasets with characteristics covering around 98% of datasets in the UCI repository. Finally, Lux has intuitive ways to export one or more visualizations, as well as edit the underlying code for customization. Our **contributions** are as follows:

- We show how Lux supports visual interactions with dataframes, and introduce a *dataframe interaction framework* (§2).
- We introduce *intent* as a high-level mechanism to convey aspects of interest to Lux, with a grammar and query language (§3).
- We introduce four classes of recommendations based on the metadata, intent, structure, and history. The latter two are dataframe-specific ones that have not been explored in prior work (§4).
- We develop a modular system, Lux, that interprets intent and generates recommendations (§5) with an efficient execution engine for metadata and visualization computation (§6).
- Finally, we evaluate the interactive performance of Lux (§7) and conduct usability studies with data scientists and early adopters (§8).

2 VISUAL DATAFRAME WORKFLOWS

We first demonstrate how always-on visualization support for dataframes accelerates exploration and discovery.

2.1 Lux Example Workfow

We present a workflow of Alice, a public policy analyst, exploring the relationship between world developmental indicators (such as life expectancy, inequality, and wellbeing) and the country's early effort in COVID-19 response. A live demo of the example notebook can be found here[1].

Always-on dataframe visualization. Alice opens up a Jupyter notebook and imports `pandas` and Lux. Using `pandas`'s `read_csv` command, Alice loads the Happy Planet Index (HPI) [2] dataset of country-level data on sustainability and well-being. To get an overview, Alice prints[2] the dataframe `df` and Lux displays the default `pandas` tabular view, as shown in Figure 1 (top, orange box). By clicking on the toggle button, Alice switches to the Lux view that displays a set of univariate and bivariate visualizations (bottom), including scatterplots, bar charts, and maps, showing an overview of the trends. Visualizations are organized into sets called *actions*, displayed as tabs. The one displayed currently is the `Geographic` action. By inspecting the `Correlation` tab in Figure 1 (not displayed here), she learns that there is a negative correlation between `AvrgLifeExpectancy` and `Inequality` (same chart as Figure 2 left); in other words, countries with higher levels of inequality also have a lower average life expectancy. She also examines the other tabs, which show the `Distribution` of quantitative attributes and the `Occurrence` of categorical attributes.

Steering analysis with intent. Next, Alice wants to investigate whether any country-level characteristics explain the observed negative correlation between inequality and life expectancy. As

[1] https://mybinder.org/v2/gh/lux-org/lux-binder/master?urlpath=tree/demo/hpi_covid_demo.ipynb

[2] We refer to any operations that result in a dataframe in the output cell of a notebook as *printing the dataframe*, not the literal 'print (df)'.

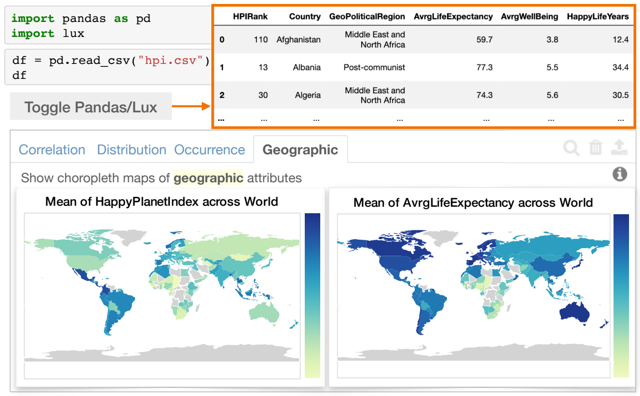

Figure 1: By printing out the dataframe, the default pandas tabular view is displayed (orange box) and users can toggle to browse through visualizations recommended by Lux.

in Figure 2, she specifies her analysis *intent* to Lux as: `df.intent = ["AvrgLifeExpectancy", "Inequality"]`. On printing the dataframe again, Lux employs the specified analysis intent to steer the recommendations towards what Alice might be interested in. On the left, Alice sees the visualization based on her specified intent. On the right, Alice sees two sets of recommendations that add an additional attribute (`Enhance`) or add an additional filter (`Filter`) to her intent. By looking at the colored scatterplots in the `Enhance` action, she learns that most G10 industrialized countries (Figure 2 center) are on the upper left quadrant on the scatterplot (low inequality, high life expectancy). In the breakdown by `Region` (Figure 2 right), she finds countries in Sub-Saharan Africa (yellow points) tend to be on the bottom right, with lower life expectancy and higher inequality.

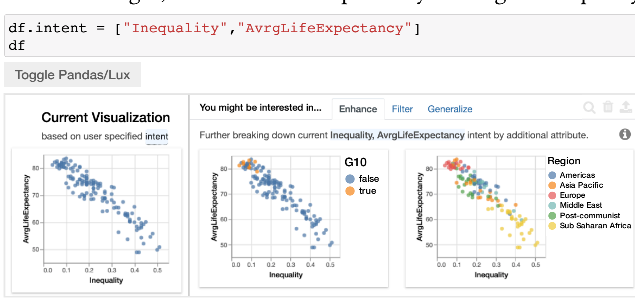

Figure 2: Alice sets the intent based on the attribute `AvrgLifeExpectancy` and `Inequality`, and Lux displays visualizations that are related to the intent.

Seamless integration with cleaning and transformation. Alice is interested in how a country's development indicators relate to their early COVID-19 response as of March 11, 2020. To investigate this, she imports a new dataset that characterizes how strict a country's response is, via `stringency` [27], a number from 0-100, with 100 being the highest level of responses. As shown in Figure 3, (I) Alice loads and joins the newly-cleaned dataframe with the earlier HPI dataset. (II) When she sets the intent on `stringency`, she finds that China and Italy have the strictest measures (dark blue on map Figure 3 center). She also learns that the histogram of `stringency` is heavily right-skewed (Figure 3 left), revealing how many countries had low levels of early pandemic response. (III) To better discern country-level differences, Alice bins `stringency` values into a binary indicator, `stringency_level`, showing whether a country had `Low` or `High` levels of early response. With the modified dataframe, Alice revisits the negative correlation she observed previously by setting the intent as average life expectancy and inequality again. The resulting recommendations are similar to Figure 2, with one additional visualization showing the breakdown by `stringency_level` (Figure 4 right). Alice finds a strong separation showing how stricter countries (blue) corresponded to countries with higher life expectancy and lower levels of inequality. This visualization indicates that these countries have a more well-developed public health infrastructure that promoted the early pandemic response. However, we observe three outliers (red arrow on Figure 4 right) that seem to defy this trend. When she filters the dataframe to learn more about these countries (Figure 4 left), she finds that these correspond to Afghanistan, Pakistan, and Rwanda—countries that were praised for their early pandemic response despite limited resources [5, 7, 19]. She clicks on the visualization in the Lux widget and the button to export the visualization from the widget to a `Vis` object. Alice can access the exported `Vis` via the `df.exported` property and print it as code, following which she can tweak the plotting style before sharing Figure 4 (right) with her colleagues.

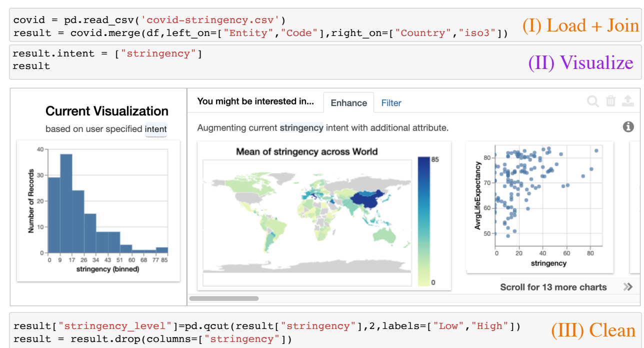

Figure 3: Tabular operations (orange, steps I & III) to load, clean, and transform the data, while visualizing with Lux (purple, step II).

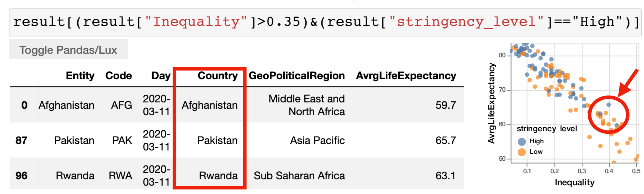

Figure 4: The scatterplot shows a separation between countries with high and low stringency in their COVID response. By filtering the dataframe (left), we see that Afghanistan, Pakistan, and Rwanda correspond to the three outliers (red boxed) that defies the trend.

Overall, this example demonstrates the value of always-on visualization support within a dataframe workflow: the tight integration between Lux and dataframes enabled Alice to seamlessly perform ETL with her familiar pandas API and notebook environment.

2.2 Dataframe Interaction Framework

The demo illustrates the many flexible ways that users can interact with a dataframe to achieve their analytical goal. We outline this different interaction[3] modalities in Figure 5.

[3]We use the term "interaction" loosely in describing this framework. The interaction with a dataframe not only refers to operations specified programmatically by user, but also ones that are synthesized by the system. For example, Lux can automatically recommend appropriate actions that generate views to display to users.

Dataframe API 1: Users can operate on the dataframe directly to perform any desired transformation or analysis. For example, Alice loaded the CSV, performed a join with another dataframe, and filtered to a data subset all via the familiar `pandas` dataframe API.
Intent 2: Users can "attach" an intent to a dataframe to indicate aspects of the dataframe that they are interested in. The intent drives the actions and views that are generated in the levels above. In the demo, Alice indicated that she wants to learn more about `AvrgLifeExpectancy` and `Inequality`; Lux displayed visualizations related to the variable of interest. This intent is *virtual* in that as the dataframe changes, the intent can still be used to recompute visualizations on the updated dataframe; in some cases, this may result in a different visualization being computed, e.g., if the data type for a given intent column is modified. In Section 3.1, we describe a flexible intent language for specifying user interest.
Actions 3: Lux displays a default set of system-recommended actions that the users can interact with, e.g., `Enhance` or `Correlation`. Users can also register UDF-based actions for domain-specific needs[4]. In either case, these actions are written in terms of the intent language but also leverage metadata and history. They instantiate a set of views displayed to users (described next).
Views 4: A *view* is an operationalization of intent when coupled with a specific dataframe instance. Users can directly create view(s) via `Vis`/`VisList` by specifying the intent applied to a given dataframe, resulting in one (or more) visualization(s). Actions instantiate one or more views—e.g., for a collection of visualizations (`VisList`) formed by plotting correlations across various attributes, each individual visualization (`Vis`) is an intent operating on a specific dataframe instance.

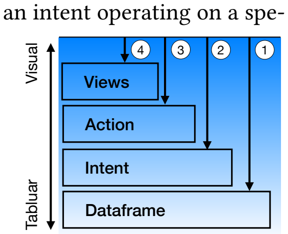

Figure 5: Visual dataframe interaction framework. 1-4 denotes four different modalities.

In this multi-tiered framework, changes in the bottom levels propagate to those above. Moreover, the settings at each level are retained across the session, so users can interact with the dataframe in a consistent and controllable manner. For example, when a user modifies the dataframe at the bottommost level, the same intent and actions are kept fixed and are used to update the views.

In addition to outlining different ways of interacting with a dataframe, the framework in Figure 5 from top to bottom spans a spectrum of interactions from visual-oriented to tabular-oriented ways of thinking, as exemplified by the orange and purple cells in Figure 2. During visual data exploration, some analytical tasks are better expressed as tabular operations (e.g., *convert the temperatures column to Fahrenheit*), while others are better expressed visually (e.g., *inspect correlation between sales and order volume* as a `Vis`). Many tasks are somewhere in between. Yet existing data querying languages and visualization grammars often create an artificial separation between the two, necessitating expensive "glue" code described earlier. By jointly considering and operating over visual and tabular aspects of dataframes, Lux supports a flexible and intuitive experience for interacting with data.

[4]https://lux-api.readthedocs.io/en/latest/source/advanced/custom.html

3 INTENT LANGUAGE FORMALIZATION

As shown in the framework in Figure 5, above the dataframe layer, users can specify their analysis intent, create custom actions, and generate desired views. This is all made possible through the intent language. The *intent language* is a lightweight, succinct means for users to programmatically and declaratively specify their high-level analysis interests and goals. Its capabilities are inspired by work on visualization query languages, such as ZQL [57] and CompassQL [66]. Unlike those languages, which are largely meant to be used internally within the corresponding interactive visual analytics systems (Zenvisage and Voyager) operating on static datasets, our intent language is tailored for programmatic specification coupled with a dynamically-evolving dataframe. To lower the barrier for composing an intent, our intent language is applicable even for sloppy, underspecified queries, such as when the channels or aggregation properties are missing from the user specification. The intent processing layer (described in Section 5) automatically infers the necessary details to transform user-specified intent into complete specifications that can be operationalized. In this section, we introduce the syntax of this intent language, and the underlying formal grammar. The grammar is decoupled from our specific implementation, which uses syntactic sugar for expressing the intent in a convenient Python-based API.

3.1 Intent Grammar

The *intent* grammar describes what the user is interested in within a dataframe. The intent is composed of one or more *clauses*, each of which is either an *axis* or a *filter* of interest.

$$\begin{aligned} \langle Intent \rangle &\rightarrow \langle Clause \rangle^{+} \\ \langle Clause \rangle &\rightarrow \langle Axis \rangle \;\mid\; \langle Filter \rangle \end{aligned} \tag{1}$$

An *axis* defines one or more attribute(s), mapped appropriately to a specific encoding or channel of the corresponding visualizations.

$$\langle Axis \rangle \rightarrow \langle attribute \rangle \langle channel \rangle^{?} \langle aggregation \rangle^{?} \langle bin\_size \rangle^{?} \tag{2}$$

For the axis, apart from the mandatory attribute(s), specified under $\langle attribute \rangle$, the remaining properties are optional—and can be automatically inferred. The axis construct is inspired by the grammar of graphics (GoG) [65] underlying visualization packages such as Vega-Lite [53] and ggplot [63]. Unlike GoG, our intent grammar doesn't require users to specify mark and channel properties. In GoG, users explicitly specify which encoding channel (e.g., x or y) each attribute is plotted—this is not necessary in our case.

Filters define a subset of data that the user is interested in. To specify a filter, the attribute being filtered, the operation, and the value, are required.

$$\langle Filter \rangle \rightarrow \langle attribute \rangle \;\; [=\, >\, <\, \leq\, \geq\, \neq] \;\; \langle value \rangle \tag{3}$$

Consider the simple case when $\langle attribute \rangle$ refers to a single attribute and $\langle value \rangle$ refers to a single value in Equations 2 and 3; then, an intent with multiple clauses (axis or filter) represents a user preference to see each of the axis attributes visualized, for the subset of data corresponding to the conjunction of the filters.

In the more general case, $\langle attribute \rangle$ can correspond to a union of attributes, or a special wildcard value ? (with an optional constraint to define the subset of attributes), while the $\langle value \rangle$ can refer to a union of values, or a special wildcard value ?.

$$\langle attribute \rangle \rightarrow \text{attribute} \cup \langle attribute \rangle^{*} \mid ? \; \langle constraint \rangle^{?} \tag{4}$$

$$\langle value \rangle \rightarrow \text{value} \cup \langle value \rangle^{*} \;\; \mid ? \tag{5}$$

The use of unions in either case (as well as ? which implicitly is a union of all alternatives) admits a disjunction of options for the axis or filter clause. If there are $n_i \geq 1$ alternatives for the i^{th} clause, we can construct a collection of $n_1 \times n_2 \times \ldots \times n_k$ visualizations by taking the cross-product of alternatives per clause. Constructing a collection of visualizations via partial specification of this sort has been explored in ZQL [56] and CompassQL [66].

3.2 Specifying Intent

As described in Section 2.2, users can specify an intent indicating their analysis interests (2). Users can also create desired views by applying the intent to a specific dataframe (4). For the creation of actions (3), Lux makes use of the same view constructs as in (4) to enumerate one or more visualizations; however, the intent for these actions is often specified by Lux internally, instead of explicitly specified by the user.

3.2.1 *Attaching an Intent to a Dataframe* (2).

Building on the grammar described above, within Lux, a `Clause` can specify one or more columns (i.e., `Axis`) or rows (i.e., `Filter`) of interest.

Query 1. To set `Age` and `Education` as columns of interest for a given dataframe `df`, one can state:

```
axis1 = lux.Clause(attribute="Age")
axis2 = lux.Clause(attribute="Education")
df.intent = [axis1,axis2]
```

Or one can also use the equivalent shortcut:

```
df.intent = ["Age", "Education"]
```

Once the intent is set, whenever `df` is printed, the Lux widget will use the intent to determine what visualizations to show to the user. Here, Lux would display visualizations related to attributes `Age` and `Education` from `df`. In the following, we will showcase the Lux intent syntax as part of `Vis` and `VisList`, but the syntax can also be used to simply set intent as in `df.intent` above.

3.2.2 *Constructing a Single Intent-driven View* (4).

As mentioned in Section 2.2, a view operationalizes an intent on a dataframe. The intent serves as a blueprint that describes while the source dataframe provides the underlying data to drive the visualization. A view is specified using the `Vis` keyword within Lux, and results in `Vis` object that is rendered as a single visualization. Users can either edit the intent or refresh the source dataframe to modify the visualization.

Query 2. Compare average `Age` across different `Education` levels.

```
axis1 = lux.Clause(attribute="Age")
axis2 = lux.Clause(attribute="Education")
Vis([axis1,axis2],df)
```

Query 2 is similar to Query 1, except that the intent is applied to the dataframe `df` to create a visualization via `Vis`, rather than changing the intent associated with the dataframe (to be used when the dataframe is eventually printed). Given that the intent involves one measure (`Age`) and one dimension (`Education`), Lux will display a bar chart. By default, average is the function used for aggregation.

Aggregation is one of three optional properties for `Axis` (Equation 2); others are channel and binning. If any of these are explicitly specified, they override Lux's defaults, as in the following query.

Query 3. Compare the variance of `MonthlyIncome` based on employee `Attrition`.

```
axis1 = lux.Clause("MonthlyIncome", aggregation=numpy.var)
axis2 = "Attrition"
Vis([axis1,axis2],df)
```

Finally, we can compose `Axis` and `Filter` together, as follows.

Query 4. Visualize the `Age`s for employees in the Sales `Department`.

```
axis = "MilesPerGal"
filter = "Department=Sales"
Vis([axis, filter],df)
```

3.2.3 *Constructing Many Intent-driven Views* (4).

`VisList` represents a collection of visualizations, which can either be constructed indirectly by setting `df.intent` as in Section 3.2.1, or as an input intent to a `VisList`, as in the following query.

Query 5. Show how factors related to the rate of compensation differ for employees with different `EducationField`s.

```
rates = ["HourlyRate","DailyRate","MonthlyRate"]
VisList(["EducationField",rates],df)
```

Here, there is one `Vis` corresponding to `EducationField` combined with each of `HourlyRate`, `DailyRate`, and `MonthlyRate`. The wildcard character ?, when used as part of an `Axis`, can be used to enumerate over *all* attributes in a dataframe; constaints may be used to restrict them to a certain type.

Query 6. Browse through relationships between any two quantitative columns in the dataframe.

```
any = lux.Clause("?",data_type = "quantitative")
VisList([any, any],df)
```

This `VisList` corresponds to the search space for the `Correlation` action; the `Correlation` action additionally ranks and sorts each `Vis` in the `VisList` based on their Pearson's correlation score.

`Filter` values can also be specified as a list or via wildcards across all possible values for a fixed filter attribute.

Query 7. Examine `Age` distributions across different `Countries`.

```
VisList(["Age", "Country=?"],df)
```

The generated `VisList` contains histograms of `Age`, one each for individuals where `Country` is `USA`, `Japan`, `Germany`, and so on.

4 VISUAL RECOMMENDATIONS

In the previous section, we have seen how users can either attach an intent to a dataframe, or this intent can be programmatically generated as part of Lux's recommendations. We discuss the latter in this section. In Lux, an *action* describes a set of visualization recommendations based on a predefined search space. These recommendations are designed to inspired next steps in the user's analysis. Lux supports four major classes of actions, as summarized in Table 1. Metadata- and intent-based ones are akin to those used in past visualization recommendation systems [29, 69]—see Lee et al. [39] for details. We then introduce two novel classes based on the use of Lux within dataframe-based data science workflows, based on dataframe structure and history.

Metadata-based Recommendations. Lux maintains dataframe metadata, including attribute-level statistics such as min/max and cardinality to determine the semantic data type of each column and to automatically populate visualization settings. For example, based on data type, Lux can generate univariate and bivariate overviews. In Figure 1, Distribution, Occurrence, Temporal, and Geographical actions provide univariate overviews of columns, while the Correlation action provides bivariate overviews of all possible pairs of quantitative attributes, ranked based on Pearson's correlation. Metadata-based recommendations have been used extensively in past visualization recommendation systems [29, 69].

Metadata	Distribution	Univariate vis of quantitative attributes (histogram)
	Occurence	Univariate vis of categorical attributes (bar chart)
	Temporal	Univariate vis of temporal attributes (line chart)
	Geographic	Univariate vis of geographical attributes (chlorepleth map)
	Correlation	Bivariate vis between quantitative attributes (scatterplot)
Intent	Enhance	Add 1 additional attribute to current vis
	Filter	Add 1 additional filter to current vis or change its value
	Generalize	Removing one or more selected attribute/filter
Structure	Series	1D versions of dataframe visualizations
	Index	Vis based on values grouped by row/column indexes
History	Pre-aggregate	Vis based on dataframes that have already been aggregated
	Pre-filter	Vis based on dataframes that have already been filtered

Table 1: Different types of default recommendations in Lux

Intent-based Recommendations. Lux displays recommendations based on the user-specified intent. On printing the dataframe, Lux displays a visualization based on the user-specified intent as in Figure 2, as the *Current Visualization*. In addition, Lux provides recommendations based on valuable next analysis steps starting from that visualization. For example, the Enhance action recommends visualizations formed by adding an additional attribute to the current visualization.

Structure-based recommendations. During the process of data science, data scientists often reshape their dataframes in ways that are more amenable to downstream analysis, discovery, and machine learning. Our formative study of existing notebooks indicates that the dataframe "structure" reveals strong signals for what the users subsequently visualize; Lux can use the same information to provide recommendations automatically:

Index-based visualizations: Dataframe indexes provide a natural way to order and label dataframe rows and columns. Indexes are typically created as a result of grouping and aggregation through operations such as `groupby`, `pivot`, `crosstab`. For any *pre-aggregated* dataframe (i.e., dataframes resulting from an aggregation operation), Lux creates visualizations by grouping the values either row or column-wise. For example, Figure 6 displays the result of a pivot operation, where each row is visualized as a time series line chart. Lux currently only supports single-level indexes, visualization of multi-level indexes is a potential direction for future work.

Series visualizations: Series are dataframes with a single column. Lux leverages the same dataframe visualization mechanism for Series, displaying univariate, metadata-based visualization, such as a bar chart for categorical and histogram for quantitative Series. By visualizing dataframe structure, Lux provides a natural and intuitive representation of dataframes and their derivative products. These visual representations can be extended to other dataframe-derived structures (e.g., `GroupBy`, `Offset`, or `Interval`) to help novices learn, debug, and validate complex dataframe operations.

History-based recommendations. Our formative study of notebooks also revealed that there is a strong connection between the operations performed by users and subsequent visualizations generated. For example, if the user cleaned up a particular column and renames it, it is likely that they would want to visualize the same column soon thereafter. Lux displays history-based recommendations based on whether the dataframe has been filtered or aggregated in its recent history. For example, when a filtering-based

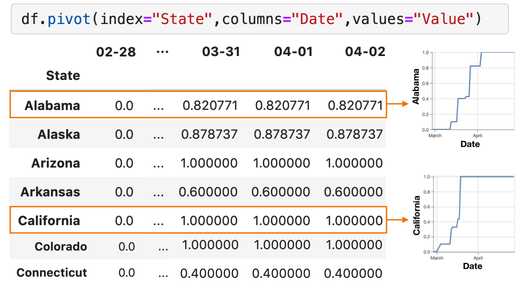

Figure 6: Row-wise index visualization displaying the normalized percentage of COVID-19 cases across different States.

operation leads to a small dataframe (such as when a `head` or `tail` is performed), Lux visualizes the previous unfiltered dataframe since there are too few tuples for generating recommendations in the filtered dataframe. Lux also uses history to determine if an aggregation has been performed, helping identify the structure-based recommendations described earlier.

To collect this history, since Lux acts as a wrapper around `pandas` (described in the next section), we instrument each dataframe function and track each one with minimal overhead and store it as part of the dataframe, instead of requiring program analysis, which is prone to false positives [72]. Given that new dataframes or intermediate objects (e.g., `GroupBy`, `Series`) are often created when the user performs an operation, Lux propagates the history over to derived objects so that the history is not lost. A key challenge for leveraging dataframe history to infer better recommendations would be around surfacing the inferred implicit intent in a way that is interpretable and explains resulting recommendations choices.

5 LUX SYSTEM DESCRIPTION

Lux implements the visual dataframe framework described in Section 2.2, and is currently used by data scientists in real-world exploratory workloads.

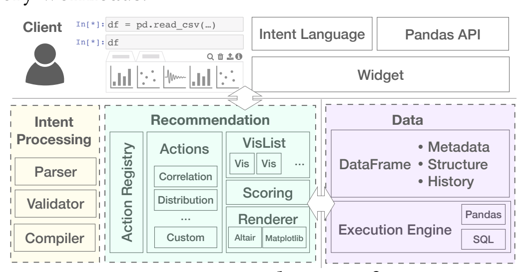

Figure 7: System architecture for Lux

5.1 Architecture

Lux employs a client-server model, leveraging computational notebooks as a frontend client. Lux currently supports Jupyter Notebooks, Jupyter Lab, Jupyter Hub, Microsoft Visual Studio Code, and Google Colab. Once users import Lux, they can interact with a `LuxDataFrame` instead of a regular `pandas` dataframe. `LuxDataFrame` acts as a wrapper around `pandas`, and supports all existing `pandas` operations, while storing additional information, such as the intent, metadata, structure, and history, for generating visual recommendations. As shown in Figure 7, the server side logic is largely separated into two distinct layers: 1) the *intent processing* layer is responsible for processing intent into executable instructions (Section 5.2) and 2) the *recommendation* layer is responsible for generating the displayed visualizations (Section 5.3). To generate the visualization recommendations, as well as compute metadata that is used in various stages, the execution engine performs the required data processing and optimization, either as a series of dataframe operations in pandas or equivalently in SQL queries executed in relational databases (Section 6).

The overall workflow is as follows:

parse §5.2.1 → compute metadata §6 → validate §5.2.1 → compile §5.2.2 → select recs. §5.3 → compute recs. §6

Metadata is memoized and only computed when needed. Finally, the system design is intended to be modular and extensible so that alternatives can be swapped in at different layers, e.g., Altair and Matplotlib visualization rendering libraries.

5.2 Intent Processing

Here, we discuss how Lux processes user intent to automatically infer missing details and determine appropriate visualization mappings. The intent processing layer parses, validates, and compiles the user's underspecified intent into complete specifications.

5.2.1 Parser and Validator. In Section 3, we saw how `Axis` and `Filter` can be be used to compose `Clauses`; the parser parses the user-inputted strings into an internal `Clause` representation. Subsequently, the validator checks for any inconsistencies between user-specified `Clauses` and the dataframe content. To do so, it leverages the dataframe's pre-computed metadata to verify the input intent. If the user's input does not align with the data present in the dataframe, the validator provides early warnings and suggests corrections to the input intent.

5.2.2 Compiler. During intent specification, users have the ability to omit certain optional details, making them *partial specifications*. Users also implicitly construct a collection of visualizations by using a union or wildcard character for `Axis` or `Filter`. Post validation, the compiler expands the `Clauses` into multiple visualizations and adds in defaults for the omitted details, making the `Clauses` *complete*. This transformation is performed in three steps.

1) Expand: If the input intent implicitly encodes multiple visualizations, the compiler "unrolls" these visualizations into individual `Vis` objects as a cross-product of the specified `Clauses`, leading to a `VisList` containing the resulting visualization specifications.

2) Lookup: For each `Vis` in the `VisList`, Lux populates the omitted details using the dataframe's pre-computed metadata. The compiler also removes any invalid visualizations generated that are either not supported in Lux or use ineffective encodings.

3) Infer: Finally, Lux infers the visualization encodings, including the marks, channels, and transforms (sort, aggregation, binning) required for generating the visualizations. The compiler implements rule-based heuristics drawn from best practices in design [26, 42].

After intent processing, Lux can now use the complete intent specification to either generate a `Vis` directly or generate a set of appropriate recommendations (described next).

5.3 Recommendation Generation

As described in the framework in Figure 5, actions organize collections of views into recommendations displayed to the users. The *action registry* in Lux keeps tracks of a list of possible actions that could be applicable for generating recommendations at any point in the analysis. On initialization, Lux registers a set of default actions (described in Section 4) applied to all dataframes. Users can also register their own custom actions programmatically by writing a Python-based UDF. The UDF generates a `VisList` of possible visualizations and optionally scores and ranks each `Vis`. The custom action is "triggered" whenever the dataframe satisfies the user-specified condition on when the action is applicable; Lux recommends visualizations based on the action.

6 EXECUTION AND OPTIMIZATION

We now describe Lux's execution engine that is responsible for computing metadata and generating visualizations. We first describe the two major tasks performed by this execution engine. Then, we describe three optimizations aimed at speeding up these tasks.

6.1 Execution Engine

We now discuss how we computes metadata and visualizations.

Metadata Computation: The metadata computed includes attribute-level statistics and data types. The statistics include the list of unique values, cardinality, and min/max of the attribute. The unique values is used to determine the candidates generated by a wildcard for a filter on the column, or for validating filter input for the column, and for computing the cardinality. The cardinality information is used to determine the data type, while min/max is used for determining the limits on the visualization axes. Next, the execution engine infers the semantic data type based on the internal data type and cardinality information. Lux supports nominal, quantitative, geographic, and temporal data types. If the data type is misclassified, users can override the automatically-inferred data type.

Visualization Processing: After the user or system-specified intent has been transformed into one or more `Vis` objects with a complete specification, the execution engine translates each `Vis` to queries responsible for processing the data required for the visualizations. First, the engine applies any filters and retrieves relevant attributes. Next, the execution engine performs different visualization-specific operations depending on the mark type. For example, to process the data for a histogram, the engine bins an attribute into fixed-sized bins and performs a count aggregation for each bin. Table 2 summarizes the relational operations that corresponds to processing different visualization types.

6.2 Optimization

Next, we describe several optimizations aimed at minimizing the overhead incurred by Lux.

Intelligent workflow-based optimizations (WFLOW): During an analysis session, users constantly modify and operate on dataframes, which means that the metadata and associated recommendations can change throughout a session, especially during reshaping and type-modifying operations. Thus, unlike conventional visual analytics, where metadata can be computed upfront and stays fixed throughout, here, metadata needs to be constantly updated to ensure that recommendations are generated correctly. As a result, the computation associated with keeping the metadata "fresh" after each dataframe operation can be computationally expensive. We propose two techniques to reduce this overhead: 1) lazily compute the metadata and recommendations only when users explicitly print dataframes; 2) cache and reuse results later on in the session.

Since users often interspersed dataframe printing with several dataframe operations, it is likely that the computed metadata and recommendations would be outdated before users see the results. As a result, we can delay computation and compute the metadata and recommendations only after the user has explicitly requested to print a dataframe. Each `LuxDataFrame` keeps track of how fresh the metadata and recommendations are and expires them when an operation makes a change to the dataframe. In particular, we leverage `pandas`'s internal functions that are triggered when:

- the dataframe is modified inplace instead of returning a new dataframe, e.g., `df.dropna(inplace=True)`
- columns in the dataframe are updated, either through the bracket or dot notation, e.g., `df.Frac` or `df["Frac"] = df["value"]/100`
- the row or column labels are changed, e.g., `df.rename(columns="val":"value")`

Additionally, recommendations are expired when the intent is modified. On printing the dataframe, Lux recomputes the metadata as needed and generates the recommendations accordingly. This lazy strategy ensures no overhead on any non-print operations. Future work on more intelligent, fine-grained maintenance and expiration strategies can improve system performance (e.g., only refresh metadata and recommendation relevant to a specific column instead of entire dataframe for a single column update).

Lux further memoizes the metadata and recommendations so that any subsequent prints to an unmodified dataframe do not require recomputation. While this may sound like an overly specific use case, such operations are, in fact, very common. In dataframe sessions, users frequently perform "non-committal" operations that do not make changes to the dataframe to be used in subsequent analyses. These non-commital actions often involve printing dataframes as intermediate results to facilitate quick experimentation and debugging. As shown in In[3-5] in Figure 8, users may try to print out a column, perform grouping and aggregation, or print out descriptive summaries, all without modifying the original dataframe. In this case, when the user revisits the original dataframe, the memoized recommendations are immediately accessible to them.

```
# Modifying operation (Deferred Computation)                          [1]
df['review_date'] = pd.to_datetime(df["review_date"],format="%Y")
df.drop(columns=['Unnamed: 0'],inplace=True)

df # Recommendations computed for the first time here                 [2]

df.groupby("company_location").mean() # Non-committal operation       [3]

df.info() # Non-committal operation                                   [4]

df # Memoized results, no extra work done!                            [5]
```

Figure 8: Example workflow demonstrating the applicability of WFLOW optimizations.

Approximate, early pruning of search space (PRUNE): As described in Section 5.3, Lux searches through a `VisList` of candidates during recommendation generation phase to displays the most interesting visualizations to users. Dataframes that are wide or contain high-cardinality attributes can often result in large visualization search spaces. For instance, the Correlation action scales quadratically with the number of quantitative attributes in the dataframe. With PRUNE, Lux first performs a preliminary pass over `VisList` to approximate the score of each visualization and then proceeds to recompute the top-k selected visualizations in a second pass to process each of the displayed visualizations *exactly*.

Lux leverages a cached sample of the dataframe to approximate visualization scores (e.g., approximating correlation on a scatterplot

Vis Type	Relational Operation
Scatterplot	Selection on 2 columns
Color Scatterplot	Selection on 3 columns
Line/Bar	Group-By aggregation
Colored Line/Bar	2D Group-By aggregation
Histogram	Binning + Count
Heatmap	2D Binning + Count
Color Heatmap	2D Binning + Count + Group-By aggregation

Table 2: Table summarizing the relational operations performed for processing different visualizations. Primary operations that accounts for the bulk of the visualization processing costs are listed.

by using only 30k rows on a dataframe with 1M rows), although other approximate query processing (AQP) methods could be applied.

Given that the PRUNE optimization performs two passes over the `VisList` (first pass for pruning, followed by an exact recomputation for the top-k), the additional recomputation cost incurred can be higher than doing a single pass over the `VisList`. Therefore, this optimization should only be applied when the approximate savings are larger than the recomputation cost of the top k visualizations: $N \times t_{exact} \gg N \times t_{approx} + k \times t_{exact}$, where N represents the number of candidate visualizations, t_{exact} and t_{approx} are the cost of computing the exact and approximate scores, respectively. Intuitively, in the ideal case where t_{approx} is close to zero, N needs to be at least greater than k as a minimum requirement for the PRUNE optimization to provide meaningful savings. The cost of scoring a visualization is dominated by the relational operations for extracting the required visualization data (e.g., selecting two columns from a dataframe for scatterplots as shown in Table 2). Therefore, we calculate t_{exact} and t_{approx} using the estimated cost of these operations (described in Section 6.3).

Cost-based scheduling of actions (STREAM): We find that users generally spend an average of 28 seconds[5] skimming through the `pandas` table view before toggling to the Lux view. To ensure interactive responses, recommendation results can be streamed into the frontend widget as the computation for each action completes without having to wait for all of the actions to finish rendering. After compiling the visualizations for each action, we estimate the cost of the action as the sum of the visualization costs in the `VisList`, using the cost model describe next. This estimate is then used for scheduling the cheapest action to compute first, followed by computing the remaining in the background. In datasets where a few "laggard" actions dominate the overall recommendation generation (e.g., `Correlation` for a wide and highly quantitative dataset), the STREAM optimization provides users with early results and returns interactive control back to the user, instead of incurring a high wait time during their analysis session.

6.3 Cost Models for Visualization Types

We now discuss the latency cost estimation of different visualization types (e.g., bar chart or scatterplot) used for PRUNE and STREAM. The visualization cost is dominated by operations that Lux performs for each visualization type, as summarized in Table 2. We outline the functional form of the cost model and note that the coefficients A-D can be empirically fitted offline.

To develop a cost model for each visualization type, we profile the visualization processing time for a set of visualizations with

[5] Based on 514 collected logs of Lux usage, the time spent on the initial `pandas` table follows a long-tail distribution, with a median of 2.8 seconds and standard deviation of 183.4 seconds.

different parameters that could affect the runtime[6]. Given the time (measured in milliseconds) on a machine with parameters as discussed in Section 7, we perform a polynomial fit with the relevant parameters[7].

Scatterplots require selecting two columns (i.e., X/Y), so the cost of visualizing a scatterplot is linear in the number of points (N). In practice, the cost of selecting a column in `pandas` is dependent on the numpy datatype (`dtype`) of the selected column, with integer columns being the fastest, followed by float and object `dtypes`. We account for the effect of data types via the channel cardinality $C_{channels}$, which is the sum of the cardinality across all channel attributes (e.g., X/Y/color).

$$\mathrm{cost(scatter)} = A \cdot C_{channel} + B \cdot N + C \tag{6}$$

where A = -1.75×10^{-4} , B = 6.04×10^{-6}, C = 2.85. Colored scatterplots select one additional color column, with A = -1.75×10^{-4}, B= 6.04×10^{-6}, C = 2.27 for quantitative colorbar and A = -3.46×10^{-4}, B = 1.59×10^{-5}, C = 3.31 for categorical colorbar.

We find that the channel cardinality and data type is a stronger determinant to the computation cost compared to the number of rows (N) in the dataframe. The coefficient for channel cardinality in the linear equation is negative, which implies that the larger the channel cardinality, the greater the runtime. In other words, a scatterplot plotting longitude and latitude containing floats with high cardinality will take longer than plotting low-cardinality integer columns.

For bar charts, Lux essentially performs a group-by aggregation. The cost is therefore dependent on the number of unique values in group-by dimension (G_{bar}), i.e., the number of bars:

$$\mathrm{cost(bar)} = A \cdot G_{bar} + B \cdot N + C \tag{7}$$

where A = 2.86×10^{-3}, B = 9.90×10^{-5}, C = 20.83. For colored bar charts, Lux performs group-by on both the bar dimension and the color attribute. Hence the dependence on the number of unique colors, G_{color}.

We also observe a weak non-linear interaction effect between G_{color} and G_{bar}.

$$\begin{aligned}\mathrm{cost(color\ bar)} = A \cdot G_{bar} + B \cdot G_{color} \\ + C \cdot G_{color} \cdot G_{bar} + D \cdot N + E\end{aligned} \tag{8}$$

where A = 2.01×10^{-2}, B = -1.81×10^{-2}, C =1.88×10^{-3}, D =1.27×10^{-4}, E = 37.12.

For histograms and heatmaps, Lux performs a binning of the data points into a number of bins or a two-dimension grid, followed by aggregation. For both chart types, the visualization cost is linear to the number of rows:

$$c(\mathrm{histogram/heatmap}) = A \cdot N + B \tag{9}$$

where A = 1.38×10^{-5}, B = 5.31 for histograms and A = 9.17×10^{-5}, B = 58.68 for heatmaps. For colored heatmaps, the aggregation function for the color attribute is an average of the datapoints that lie in the cell for color bars with quantitative attributes (A = 2.63×10^{-4}, B = 126.06), while the aggregation is based on the majority vote (i.e., mode) of the datapoints for color bars with categorical attributes (A = 1.11×10^{-4}, B = 92.38).

The cost is largely independent of the number of bins or grid size because the number of total rows that the group-by aggregation needs to be processed is the same regardless of the number of buckets the data is divided among.[8] We also note that the heatmap's coefficients differs for different aggregation functions. For example, for heatmaps without color, the aggregation is based on the counts in each bin, and for quantitative colored heatmaps, the aggregation is an average of the data points that lie in the cell.

[6]The time measured is solely based on the data operations performed to generate the data required for each visualization. This does not account for the time it takes to *render* the visualization (which is largely dependent on the specific visualization libraries and the number of graphical marks that is drawn).

[7]Fitting coefficients with values below 10^{-5} are discarded as they have little influence on the overall cost. The terms in the equation are arranged in descending order of importance of each term, based on the value of the coefficient.

7 PERFORMANCE EVALUATION

We evaluate Lux to measure its performance on large real-world datasets and notebook sessions, along the following dimensions:

- RQ1: What is the overall performance of Lux? Can Lux achieve interactive latency during a typical dataframe workflow?
- RQ2: What is the effect of the number of columns on Lux's performance?
- RQ3: How does the approximation-based PRUNE condition affect the quality of the recommendations relative to no approximation?

We focus on evaluating the interactive latency in this section; we describe the usability evaluation in the following section. Source code for experiments and analysis can be found here [9].

7.1 Data and Methodology

Data: We use two real-world datasets to evaluate the performance of Lux. The `Airbnb` dataset [23] contains 12 columns while the `Communities` [35] dataset contains 128 columns. For both datasets, we duplicated the dataset multiple times (up to 10M rows for `Airbnb` and up to 100k rows for `Communities`) to investigate the effects of scaling with the number of rows. After duplication, `Airbnb` exemplifies datasets with a moderate number of columns and a large number of rows, while `Communities` exemplifies those with a large number of columns. The upper limits on the two datasets cover around 98% of the datasets in the UCI repository [15].

Setup: All of our experiments were conducted on a Macbook Pro with 32GB of RAM and an Intel Core i9 processor running macOS 10.15.6. The experiments were run using Python 3.7.7, pandas 1.2.1, and a version of lux-api 0.2.3 adapted for purpose of the experiments. We used `papermill` [11] to programmatically execute each notebook cell. We set k for top k as 15 and apply PRUNE for any action where the number of visualizations exceeds k. For the sampling policy, we used cached random samples capped at 30k rows for approximating the visualization interestingness of dataframes over 30k rows (the choice of this parameter is justified in Section 7.4). For the runtimes reported, we exclude the frontend drawing time for each visualization given that it is constant and highly dependent on the chosen visualization library and frontend.

Conditions: Our experiment measures the time it takes to execute every cell in the notebook across five different conditions:

- no-opt: Baseline condition with no optimization applied, representing a naive implementation of Lux where the results are explicitly computed at the end of every cell involving a reference

[8]Even though heatmaps and histograms can be an arbitrarily high resolution without affecting the processing speed in the execution engine, in practice, the bin resolution still needs to be capped at a reasonable limit, since increasing bin size impacts the rendering speed (i.e., more marks that needs to be drawn on the frontend).

[9]https://github.com/lux-org/lux-benchmark

to the dataframe. This condition is akin to the naive implementation in most visualization recommendation systems, where the results are updated whenever the dataset is operated on.

- wflow: Condition with the wflow optimization applied.
- wflow + prune: Condition with wflow and prune applied.
- all-opt: Condition with wflow, prune, and stream applied, representing the best achievable performance within Lux.
- pandas: Condition with only pandas and *without* using Lux, representing the raw performance of dataframe workflows without the benefits of always-on visualizations.

7.2 Overall workflow performance (RQ1)

To evaluate the overall performance of Lux with a dataframe-based workflow, we measured the runtime for executing an example notebook involving pandas.

Workload: The workload is based on publicly available notebooks on Kaggle for Airbnb and Communities. These notebooks follow a typical exploratory analysis of a dataframe that includes loading, transformation, cleaning, computing statistics, and machine learning. We modified these notebooks to print out dataframes and series at various points in the notebook akin to what a user would typically do for validating the results of operations. In addition, we label each cell in the notebook as either a print of a dataframe, print of a series, or neither (i.e., any non-Lux Python command) to separately measure the runtime for different cell types. Table 3 shows the breakdown of the two notebook workloads by different cell types. We define *overhead* as the difference in runtime between the all-opt and pandas condition, i.e., the additional time required to support always-on visualizations via Lux.

	Airbnb		Communities		
	N	overhead [s]	N	overhead [s]	Distr.
Print df	14	21.18	14	1.41	
Print Series	7	0.61	4	0.07	
Non-Lux	17	0	25	0	

Table 3: Table reports the number of cells for each type (N), the additional time incurred on top of pandas for 10M Airbnb and 100k Communities (overhead), and the relative shape of the runtime distribution similar to Figure 9,10, (Distr.).

Overall runtime: To understand the overall performance of Lux on dataframes with varying sizes, we varied the dataframe size from 10k to 10M rows. Figure 9 displays the overall runtime averaged over all cells in the notebook. We find that the best achievable performance with Lux led to significant speedup with up to 11X improvement in overall runtime for the Airbnb dataset (and up to 345X for Communities) compared to the no-optimization baseline.

Printing Dataframes and Series: We measure the performance of each cell that prints a dataframe or series to understand the overheads associated with Lux. Figure 10 shows the average time it takes for printing a dataframe for Airbnb and Communities. In particular, the overhead of Lux for each print can be determined by comparing against the cost for a print in pandas. When the dataframe contains fewer than 1M rows for Airbnb, each print incurs no more than 2 seconds in addition to pandas (in the 10M case, each print incurred an overhead of 21 seconds). For Communities, the overhead was no more than 1.5 seconds.

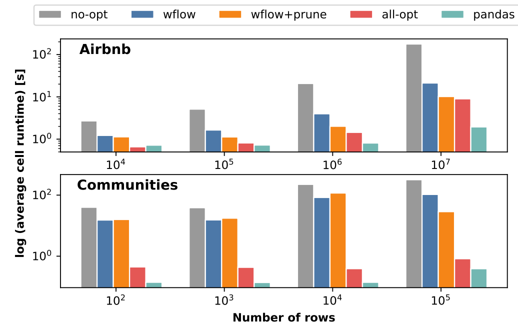

Figure 9: Average runtime of a notebook cell across the workload for different dataframe size and conditions.

As shown in the sparkline visualization in Table 3 row 2, the performance for printing series follows the same pattern as that of the dataframe. However, since series only involves a single column, it effectively avoids the costly procedure of traversing through a large search space. The overhead on top of pandas is no more than 1 second for each series print even on the largest datasets.

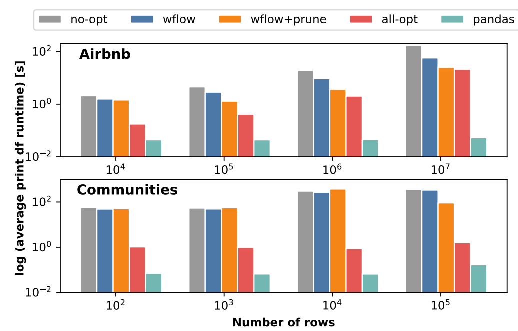

Figure 10: Average time for printing a single dataframe for different dataframe size and conditions.

Non-Lux operations: Across all conditions except the baseline, the runtime for non-Lux operations (Table 3 row 3) is the same—demonstrating how Lux incurs zero overhead on any Python operations in a notebook session. When compared against the baseline, Lux is over 100X faster for 100k Airbnb and over 650X faster for 10M Communities. The performance improvement for non-Lux operations demonstrates how wflow's lazy evaluation strategy avoids unnecessary computation.

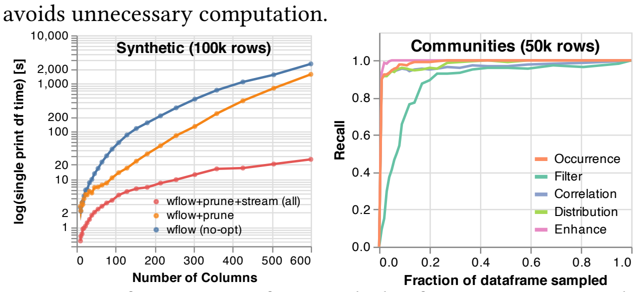

Figure 11: Left: Time spent for a single dataframe print varying the number of columns in a synthetic dataframe. Right: Recall curve for different actions varying fractional samples of rows in the 50k Communities dataset.

7.3 Effect of dataframe width (RQ2)

We investigate how the performance of Lux varies depending on the number of columns in the dataframe. To understand the effect of the width of a dataframe (w), we measure the processing time for a single dataframe print (after the metadata has already been precomputed). Given the dependence of actions on data types, we leverage a synthetic dataset generated using the `faker`[14] library to vary the number of columns in the dataframe, while fixing the proportion of data types. The simulated dataframe contains 100k rows with 78% quantitative columns, 20% nominal columns, and 2% as temporal. Across the quantitative columns, half of the columns are integers, while the other half are floats. For the nominal columns, we generate columns of strings with varying cardinalities chosen based on a geometric series between 1 to 10000.

Figure 11 left shows the runtime for different dataframe widths[10]. We note that the blue no-opt curve (power=2.53) scales exponentially with the number of columns. By applying the PRUNE and STREAM optimizations (red), LUX effectively lowers the cost of printing a dataframe by bringing the runtime closer to linear (power=1.07).

7.4 Effect on recommendation accuracy (RQ3)

To understand how the approximation-based PRUNE condition affects the recommended results, we experimented with different fractional sizes of the dataframe to be used in the sample and its effect on the recommendation ranking. We compared the list of recommendations generated with and without the optimization applied. We computed Recall@15 of the top k results against the ground truth rankings. We chose recall, instead of other rank position-dependent measures, because the top-k visualizations are computed exactly and re-ranked after selection, so the metric only needs to capture how accurately the top-k visualizations are retrieved.

The recall curves in Figure 11 right shows that for most actions 10% (5k rows) is required in the sample for achieving over 90% accuracy. For the 100k `Airbnb` dataset, the sample requirement is around 20-40% (i.e., 20-40k rows). As a result, we chose the sampling cap in our experiment to be 30k rows to reach an average of 90.5% on `Airbnb` dataset and near perfect ($\geq$ 95%) on `Communities`. Compared to other actions, since Filter (light green in Figure 11 right) enumerates over data subsets, it requires more samples to ensure enough data points per stratum to achieve the same accuracy.

8 ASSESSMENTS WITH USERS

To understand the effectiveness and usage of LUX in typical data science workflows, we performed two usability studies: a controlled study with new users and a field study with existing users of LUX.

8.1 First-use Controlled Study

We performed a study to understand participants' initial impressions of LUX and whether they are able to use LUX effectively in a controlled setting. This study was performed remotely from October to November 2020 using lux-api 0.2.0. This study was part of a 90-minute interactive session where participants were first introduced to the basics of LUX and guided through a set of hands-on exercises on how to use LUX. The study was conducted with two focus groups: the first was a bootcamp for industry data practitioners (N=20) and the second was an online lecture for students in a graduate-level data visualization course (N=15). Both groups engaged in the same set of instructions and tasks. The instructions and tasks were made available to participants via a web link to a live Jupyter notebook. Participants were led through three notebooks in sequence. Each notebook contained examples and exercises covering the key concepts in LUX using three datasets (College [4], Happy Planet Index [2], and Olympics [1]). Interactions on the LUX widget and actions performed on the notebook were logged via a custom extension [40]. The session concluded with a short survey documenting participants' experience. Due to the remote and unsupervised study setting, not all participants submitted survey responses or performed notebook operations that were logged.

Study Findings. We collected 16 survey responses (6 from bootcamp, 10 from lecture). The results were thematically coded and classified by one of the authors. In response to background questions regarding the existing exploration workflows of the participants, their concerns echoed the pain points that LUX aims to address, including difficulty in determining the "right" visualization to plot (5/16), modifying and iterating on visualizations (4/16), and determining where to begin an analysis (4/16). When asked to comment on aspects of LUX that they liked, 9/16 participants cited how the ability to print and visualize dataframes was the most useful. Participants also noted how the integration of LUX with their data science workflow was seamless and intuitive. When asked to comment on aspects of LUX that they found challenging, 8/16 participants described unfamiliarity and the learning curve associated with the intent syntax. When asked about what they would like to see most in future versions, participants were most interested in improving LUX's latency on large datasets (12/16)[11], followed by support for a wider and more useful set of recommendations (8/16) and making the intent language more customizable (7/16). At the end of the survey, 13/16 participants signed up for follow-ups and expressed interest in continuing to use LUX.

To evaluate whether participants were able to accomplish controlled tasks with LUX, we collected 23 unique logs of the participants' interaction with the notebooks. We qualitatively graded how well participants performed across the three exercises. The task success rate for the three exercises was 68% (for composing an intent indicating multiple views), 87% (for specifying a desired `Vis`), and 71% (for creating a `VisList`). By inspecting the trace of attempts, on average participants were able to obtain the first successful answer within their first five tries. Participants' most common mistakes involved confusion around the syntax for specifying multiple visualizations via union. Finally, participants were encouraged to try out one of the provided datasets for open-ended exploration. While participants successfully used LUX to print and visualize their dataframes, due to the setting and time constraints, their interactions with LUX were brief. The limited insight into how users performs open-ended exploration with LUX motivated the need for the following study.

8.2 Field Study Interviews

To understand how LUX is used in real-world analytical workflows, from December 2020 to January 2021, we conducted semi-structured interviews with participants who used LUX in their data science work. We interviewed two industry data scientists in an insurance (P1) and retail company (P3), and a researcher in education (P2). Given that participants had extended exposure to LUX, our questions largely focused on understanding how LUX fits into their

[10] We note that the no-opt condition is the same as WFLOW in this case since we are only measuring a single print dataframe cell.

[11] We note that the study was performed using the latest version of LUX at that point, which did not include many of the scalability improvements described earlier (WFLOW was included, but not STREAM and PRUNE).

To what extent do you find the following functionalities in Lux useful?	P1	P2	P3
Printing dataframe and inspecting recommended visualizations	Very useful	Very useful	Extremely Useful
Expressing analysis intent to steer recommendations	Extremely Useful	Extremely Useful	Very useful
Specifying visualization of interest via `Vis`	Moderately useful	N/A (Did not use)	Very useful
Specifying collections of visualizations of interest via `VisList`	Very useful	N/A (Did not use)	Very useful
Exporting selected visualizations from Jupyter widget	Very useful	Extremely Useful	N/A (Did not use)
Lux makes it easier to ...	**P1**	**P2**	**P3**
Visualize my data across different stages in the data science workflow	Agree	Strongly agree	Agree
Plot a single visualization that I have in mind	Strongly agree	Strongly agree	Strongly agree
Identify what aspects of data I should visualize	Strongly agree	Agree	Agree
Determine what to do next in my exploration	Agree	Somewhat agree	Neutral

Table 4: Table of Likert scale ratings across the three field study participants.

existing workflows. Before the interviews, participants used Lux over the span of 1-2 months in their professional data science work. Their usage frequency varied: P1 used Lux daily, P2 used Lux once every one or two weeks, P3 used Lux around ten times in total. Unlike the first-use study where participants were led through instructions dedicated to how to create `Vis` and `VisList`, field study participants learned how to use Lux on their own through tutorials and documentation on our website. We performed a walk-through of real-world notebooks in which participants had used Lux.

Study Findings. All three participants expressed that understanding their data was a challenge during exploration. In fact, two of the participants have developed their own homegrown solutions for past projects (echoing findings from Alspaugh et al. [17]), ranging from for loops across `matplotlib` charts in notebooks to VBA scripts that generate plots in Excel. In their existing workflows, P1 and P2 visualized their data programmatically via `matplotlib`, while P3 largely on Tableau's GUI for creating visualizations.

On dataframe visualizations: All three participants expressed that they appreciated how the automatic visualizations provided by Lux afforded them quick insight into their dataframes without the need for code. P2 typically examines over 100 columns of data as part of an educational course survey, and stated that Lux sped up the amount of time for EDA by at least two-fold: *" it really helps speed up my exploratory analysis. If not, it will take me forever to go through these many variables."* When asked about the scenarios for which they would toggle to the Lux view versus the default `pandas` table, most participants preferred seeing the Lux view for the purposes of EDA. Participants described how they only use the `pandas` table to quickly check if "*the data looks okay*" (P1) and rarely toggle back to it unless they observe anomalous trends in the visualizations. During the study, P2 adopted a workflow where they sampled a single row to display the `pandas` table in one notebook cell, then printed the Lux view in the cell below to check that the data falls in the expected ranges as displayed in the visualizations.

On dataframe intents: Participants indicated that the concept of intent was an intuitive way for steering the course of their analysis. P1 and P2 leveraged intent as a way of systematically exploring groups of variables they were interested in. To investigate their research questions, P2 listed groups of independent and dependent variables as their intent to explore each group one at a time. P1 and P3 used intent as a way of exploring predictive variables of interest, such as whether a customer purchased accessories alongside their orders, to help inform feature engineering for downstream machine learning. However, challenges in specification sometimes prevented them from making use of intent fully. In particular, P2 and P3 both described that they were interested in exploring alternative data subsets for an attribute of interest (a query that is expressible in Lux's intent language); however, they were unaware that they could specify filter intent with wildcards. Improving the API for intent specification remains an important direction for future work.

On custom actions: Participants noted how the default Lux actions largely covered the basic sets of analyses that they would typically perform on their own. While most participants were unaware that Lux supported the ability to create custom actions, during various points in the interview, they described additional actions that they would find useful. For example, P3 described how they wanted to create a custom action that lists the top ten dataframe columns with the most influence over a desired predictive variable. Other participants described actions that are similar to the default Lux actions, but with a different ranking. For example, P2 was interested in categorical variables that involved bar charts that looked very even, since that means that it has a closer-to-equal likelihood of being in either categories, so the trend is potentially interesting.

On user-specified views: Somewhat surprisingly, while `Vis` and `VisList` were highly favored in the first-use study, they were rarely brought up in the field study interviews. Possible explanations for their limited use include the unfamiliarity with these concepts and their usage of Lux in conjunction with other visualization tools. All participants used an existing visualization tool (e.g., matplolib or Tableau) while exploring their data with Lux. As a result, they simply used their familiar tools for specific visualizations when they knew exactly what to plot. To fully leverage `Vis` and `VisList` in their work, participants often asked for ways to extend or customize the visualization type for a user-specified view. For example, P3 explained how market share data was best visualized as a top-k pie chart, while P2 was interested in examining overlaid histogram distribution of different measures for binary variables, such as whether or not a course was open-ended. These findings indicate that increased flexibility in the intent language could afford the familiar visualization capabilities for users when creating specified views.

Usage of Lux in data science workflows: All three participants described using Lux explicitly in the exploration stage after data loading and cleaning, but before advanced analysis or modeling. P1 and P2 used Lux in conjunction with custom `matplotlib` code that they repurposed for their analysis. When asked why participants did not print the dataframe for visualizations during the data transformation and cleaning phase, P1 and P3 answered that since the dataframe prints resulted in a few seconds of latency, they were hesitant to do it until they were ready to *"chuck in [their] data and get the charts out"* (P3). Participants also described how Lux needed to be more robust in visualizing dirty or ill-formatted data.

Post-interview survey results: Table 4 details participants' Likert scale ratings of the functionalities and benefits of Lux. Participants found the use of intent and the ability to print and visualize dataframes to be the most useful features. Participants reported that they either did not make use of `Vis`, `VisList`, and export functionalities or found them to be less useful. Participants described how Lux made it easier to plot a single visualization that they had in mind, identify aspects of data they should visualize, perform visualization across different stages of the data science workflow, and determine what to do next. The average System Usability Scale (SUS) [21] score across participants is 70/100. All three participants were interested in continuing to use Lux in their data science work.

Limitations and future work: The discrepancy between the usage of views in the first-use and field study indicates that even though `Vis` and `VisList` are could be learned with a focused tutorial and exercise, they are not as discoverable and easy-to-use as the dataframe visualizations. Despite the enthusiasm around Lux, we find participants still attached to their existing visualization tool for this functionality. They shared concerns around customizability and the inability to express their desired visualizations in Lux, pointing to the need for improving the flexibility of the intent language. Given that our participants often work with data in commercial cloud data warehouses, it is not only important for Lux to speed up processing for recommendations (as in Section 6), but also account for data that doesn't fit in memory in the future.

9 RELATED WORK

Visual Analytics. To visualize data, data scientists need to subselect the aspects of data, and then define a mapping from data to graphical encodings. This is done via one of two paradigms: code or interactive interfaces. Interactive interfaces, such as Tableau [3, 59] and PowerBI [12], offer easy-to-use interfaces for visualization construction. Some systems also offer visualization recommendations (VisRec). VisRec systems can either suggest interesting portions of the data to visualize based on statistical properties [22, 32, 38, 43, 56, 60, 61] or better ways to visualize attributes that users have selected [28, 41, 42, 44, 66]. Similarly, there has been research on recommending interesting attributes or filters to avoid manual data exploration during OLAP [31, 37, 50–52, 70]. While such GUI tools have gained adoption among business analysts, they are not as widely used by data scientists with programming expertise, due to their lack of customizability and integration with the rest of the data science workflow. That said, we draw on recommendation principles from this work, as discussed in Section 4.

On the other hand, data scientists often leverage plotting libraries, such as Altair, `matplotlib`, Plotly, or ggplot, to programmatically generate visualizations, leveraging various visualization design principles [20, 53, 54, 58, 63]. These libraries often take a dataframe as input, attempting to isolate visualization decisions from data processing ones, requiring users to translate their desired visualization goals into executable code across different visualization and data processing libraries. Lux instead reduces the burden on users, allowing them to provide lightweight intent as opposed to writing long code fragments for visualization.

Given these aforementioned tradeoffs, recent projects have explored ways to integrate code and interactions in computational notebooks [34, 71]. For example, B2 [71] facilitates bi-directional exchange between code and interaction, and produces a persistent trace of interaction history. Likewise, mage [34] demonstrates how similar techniques could be generalized to other interactive data science applications. While we draw on similar principles, our focus is on visualization recommendation as a replacement for cumbersome visualization code, and for suggesting promising next steps.

Visual Data Exploration with Dataframes. Of late, dataframes have become the de-facto framework for exploratory data science. The comprehensive, incremental set of operators make it easy to do sophisticated data transformation, while also allowing rapid validation after each step. Dataframes are also used as an exchange format for interoperability across data science libraries. However, exploring dataframes is challenging, requiring substantial programming and analytical know-how. Many visualization tools have been developed for dataframes [6, 16, 25, 46, 55]. These tools generate interactive reports of a dataframe, covering analyses spanning from missing values, outliers, to attribute-level visualizations and associated statistics. In addition, bamboolib [6], pandas-profiling [46], dataprep [55], sweetviz [25], and pandasgui [16] offer a GUI for constructing visualizations and data transformation.

Instead, Lux adopts an always-on approach so that dataframe visualizations are generated for free whenever the dataframe is printed, instead of relying on users to explicitly call external commands to *plot* or *profile* as needed. This *always-on* approach lowers the barrier to visualizing dataframes and encourages exploration. Moreover, while these interactive reports contain a lot of information, similar to GUI tools, there is no way to customize or "dig further" to investigate an interesting visual insight. Users cannot suggest their intent, and nor can the system use this to recommend appropriate visualizations and next steps. Finally, as we saw in Section 6, Lux's optimizations allow for it to be interactive on a range of real-world datasets.

10 CONCLUSION

We propose Lux, an always-on visualization framework for accelerating insight exploration and discovery. Lux is a lightweight wrapper around dataframes that reduces the barrier of visualizing data and guiding the process of determining analysis next-steps. To support automated visualizations of dataframes, dataframes can be enriched with information from the user, such as user's intent and history, as well as structural information and metadata. We develop and evaluate effective optimization strategies that intelligently cache and maintain metadata and recommendations. Lux's initial adoption and success of user evaluation points to its importance for exploratory data science — presenting a scalable solution towards bridging the gap between users and insight discovery.

REFERENCES

[1] 2019. 120 years of Olympic history: athletes and results. https://www.kaggle.com/heesoo37/120-years-of-olympic-history-athletes-and-results. (2019).
[2] 2019. Happy Planet Index. http://happyplanetindex.org/. (2019).
[3] 2019. Tableau. https://www.tableau.com/. Accessed: 2019-09-11.
[4] 2019. US Department of Education: College Scorecard Data. https://collegescorecard.ed.gov/data/documentation/. (2019).
[5] 2020. Afghanistan: WHO mission reviews COVID-19 response. https://www.who.int/news-room/feature-stories/detail/afghanistan-who-mission-reviews-covid-19-response. *World Health Organization* (2020).
[6] 2020. bamboolib. https://bamboolib.8080labs.com/
[7] 2020. COVID-19 in Pakistan: WHO fighting tirelessly against the odds. *World Health Organization* (2020). https://www.who.int/news-room/feature-stories/detail/covid-19-in-pakistan-who-fighting-tirelessly-against-the-odds
[8] 2020. Faster data exploration in Jupyter through Lux. https://blog.dominodatalab.com/faster-data-exploration-in-jupyter-through-the-lux-widget/
[9] 2020. LUX Exploratory Data Analysis (EDA). https://www.youtube.com/watch?v=00BEjUzKDmY
[10] 2020. LUX Library: Matplotlib replacer? https://www.youtube.com/watch?v=m41h4mGzwwE

[11] 2020. papermill 2.3.3 documentation. https://papermill.readthedocs.io/
[12] 2020. Power BI | Interactive Data Visualization BI Tools. https://powerbi.microsoft.com/en-us/.
[13] 2020. State of Data Science and Machine Learning 2020. https://www.kaggle.com/kaggle-survey-2020
[14] 2021. Faker. https://github.com/joke2k/faker
[15] 2021. UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/datasets.php
[16] adamerose. [n. d.]. PandasGUI. https://github.com/adamerose/pandasgui
[17] S. Alspaugh, N. Zokaei, A. Liu, C. Jin, and M. A. Hearst. 2019. Futzing and Moseying: Interviews with Professional Data Analysts on Exploration Practices. *IEEE Transactions on Visualization and Computer Graphics* 25, 1 (2019), 22–31.
[18] Andrea Batch and Niklas Elmqvist. 2018. The Interactive Visualization Gap in Initial Exploratory Data Analysis. *IEEE Transactions on Visualization and Computer Graphics* 24, 1 (2018), 278–287. https://doi.org/10.1109/TVCG.2017.2743990
[19] Jason Beaubien. 2020. Why Rwanda Is Doing Better Than Ohio When It Comes To Controlling COVID-19. *NPR* (Jul 2020). https://www.npr.org/sections/goatsandsoda/2020/07/15/889802561/a-covid-19-success-story-in-rwanda-free-testing-robot-caregivers
[20] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D³ Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics* 17, 12 (Dec. 2011), 2301–2309. https://doi.org/10.1109/TVCG.2011.185
[21] John Brooke. 1996. *"SUS-A quick and dirty usability scale." Usability evaluation in industry.* CRC Press. https://www.crcpress.com/product/isbn/9780748404605 ISBN: 9780748404605.
[22] Çağatay Demiralp, Peter J Haas, Srinivasan Parthasarathy, and Tejaswini Pedapati. 2017. Foresight: Rapid data exploration through guideposts. *arXiv preprint arXiv:1709.10513* (2017).
[23] Dgomonov. 2020. Data Exploration on NYC Airbnb. https://www.kaggle.com/dgomonov/data-exploration-on-nyc-airbnb
[24] Pranavi Duvva. 2021. Speed up EDA With the Intelligent Lux. https://pub.towardsai.net/speed-up-eda-with-the-intelligent-lux-37f96542527b
[25] Fbdesignpro. [n. d.]. sweetviz. https://github.com/fbdesignpro/sweetviz
[26] Stephen Few. 2012. *Show Me the Numbers: Designing Tables and Graphs to Enlighten.* Analytics Press.
[27] Thomas Hale, Noam Angrist, Rafael Goldszmidt, Beatriz Kira, Anna Petherick, Toby Phillips, Samuel Webster, Emily Cameron-Blake, Laura Hallas, Saptarshi Majumdar, and Helen Tatlow. 2021. A global panel database of pandemic policies (Oxford COVID-19 Government Response Tracker). *Nature Human Behaviour* (2021). https://doi.org/10.1038/s41562-021-01079-8
[28] Kevin Hu, Michiel A. Bakker, Stephen Li, Tim Kraska, and César Hidalgo. 2019. VizML: A Machine Learning Approach to Visualization Recommendation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300358
[29] Kevin Hu, Diana Orghian, and César Hidalgo. 2018. DIVE: A Mixed-Initiative System Supporting Integrated Data Exploration Workflows. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics.* ACM, 5.
[30] Alekh Jindal, K. Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas C. Müller, Wentao Wu, and Hiren Patel. 2021. Magpie: Python at Speed and Scale using Cloud Backends. In *CIDR*. https://www.microsoft.com/en-us/research/publication/magpie-python-at-speed-and-scale-using-cloud-backends/
[31] Manas Joglekar, Hector Garcia-Molina, and Aditya Parameswaran. 2015. Smart Drill-Down : A New Data Exploration Operator. *Proceedings of the 41st International Conference on Very Large Data Bases* 8, 12 (2015), 1928–1931.
[32] Niranjan Kamat, Eugene Wu, and Arnab Nandi. 2016. TrendQuery: A System for Interactive Exploration of Trends. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics (HILDA '16)*. ACM, New York, NY, USA, Article 12, 4 pages. https://doi.org/10.1145/2939502.2939514
[33] S Kandel, A Paepcke, J M Hellerstein, and J Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE transactions on visualization and computer graphics* 18, 12 (2012), 2917–26. https://doi.org/10.1109/TVCG.2012.219
[34] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. 2020. Mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/3379337.3415842
[35] Kkanda. 2020. Analyzing UCI Crime and Communities Dataset. https://www.kaggle.com/kkanda/analyzing-uci-crime-and-communities-dataset
[36] A. P. Koenzen, N. A. Ernst, and M. A. D. Storey. 2020. Code Duplication and Reuse in Jupyter Notebooks. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–9. https://doi.org/10.1109/VL/HCC50065.2020.9127202
[37] Doris Jung-Lin Lee, Himel Dev, Huizi Hu, Hazem Elmeleegy, and Aditya Parameswaran. 2019. Avoiding Drill-down Fallacies with VisPilot: Assisted Exploration of Data Subsets. In *Proceedings of the 24th International Conference on Intelligent User Interfaces (IUI '19)*. ACM, New York, NY, USA, 186–196. https://doi.org/10.1145/3301275.3302307
[38] Doris Jung-Lin Lee and Aditya Parameswaran. 2018. The Case for a Visual Discovery Assistant: A Holistic Solution for Accelerating Visual Data Exploration. *IEEE Bulletin of Technical Committee on Data Engineering* 41, 3 (2018), 3–14.
[39] Doris Jung-Lin Lee, Vidya Setlur, Melanie Tory, Karrie Karahalios, and Aditya Parameswaran. 2021. Deconstructing Categorization in Visualization Recommendation: A Taxonomy and Comparative Study. *arXiv e-prints*, Article arXiv:2102.07070 (Feb. 2021), arXiv:2102.07070 pages. arXiv:2102.07070
[40] lux org. [n. d.]. lux-logger. https://github.com/lux-org/lux-logger
[41] Jock Mackinlay. 1986. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics* 5, 2 (1986), 110–141. https://doi.org/10.1145/22949.22950
[42] Jock D. Mackinlay, Pat Hanrahan, and Chris Stolte. 2007. Show Me: Automatic presentation for visual analysis. *IEEE Transactions on Visualization and Computer Graphics* 13, 6 (2007), 1137–1144. https://doi.org/10.1109/TVCG.2007.70594
[43] Miro Mannino and Azza Abouzied. 2018. Expressive Time Series Querying with Hand-Drawn Scale-Free Sketches. , Article 388 (2018), 13 pages. https://doi.org/10.1145/3173574.3173962
[44] Dominik Moritz, Chenglong Wang, Greg L Nelson, Halden Lin, Adam M Smith, Bill Howe, and Jeffrey Heer. 2019. Formalizing visualization design knowledge as constraints: Actionable and extensible models in Draco. *IEEE transactions on visualization and computer graphics* 25, 1 (2019), 438–448.
[45] The pandas development team. 2020. pandas-dev/pandas: Pandas. https://doi.org/10.5281/zenodo.3509134
[46] Pandas-Profiling. [n. d.]. pandas-profiling. https://github.com/pandas-profiling/pandas-profiling
[47] Parul Pandey. 2021. Intelligent Visual Data Discovery with Lux: A Python library. https://towardsdatascience.com/intelligent-visual-data-discovery-with-lux-a-python-library-dc36a5742b2f
[48] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 12 (July 2020), 2033–2046. https://doi.org/10.14778/3407790.3407807
[49] Adam Rule and U C San Diego. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. 2, November (2018).
[50] Sunita Sarawagi. 1999. Explaining differences in multidimensional aggregates. *Proceedings of the VLDB Endowment* (1999), 42–53.
[51] S. Sarawagi. 2000. User-adaptive exploration of multidimensional data. *Proc of the 26th Intl Conference on Very Large* (2000), 307–316. citeseer.ist.psu.edu/sarawagi00useradaptive.html
[52] S Sarawagi, R Agrawal, N Megiddo, Generalitat Valenciana Ajuntament Valencia Univ Politecn Valencia, and E T H Zurich Oracle Sybase Softlab Iberia Edbt Fdn. 1998. Discovery-driven exploration of OLAP data cubes. *6th International Conference on Extending Database Technology (EDBT 98)* (1998), 168–182. http://dl.acm.org/citation.cfm?id=645338.650401
[53] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017), 341–350.
[54] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. 2016. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics* 22, 1 (2016), 659–668.
[55] sfu db. [n. d.]. dataprep. https://github.com/sfu-db/dataprep
[56] Tarique Siddiqui, Albert Kim, John Lee, Karrie Karahalios, and Aditya Parameswaran. 2016. Effortless data exploration with zenvisage: an expressive and interactive visual analytics system. *Proceedings of the VLDB Endowment* 10, 4 (Nov. 2016), 457–468. https://doi.org/10.14778/3025111.3025126
[57] Tarique Siddiqui, John Lee, Albert Kim, Edward Xue, Xiaofo Yu, Sean Zou, Lijin Guo, Changfeng Liu, Chaoran Wang, Karrie Karahalios, et al. 2017. Fast-Forwarding to Desired Visualizations with Zenvisage.. In *CIDR*.
[58] Chris Stolte, Diane Tang, and Pat Hanrahan. 2002. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics* 8, 1 (2002), 52–65.
[59] C. Stolte, D. Tang, and P. Hanrahan. 2002. Polaris: a system for query, analysis, and visualization of\nmultidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics* 8, 1 (2002), 1–14. https://doi.org/10.1109/2945.981851
[60] Manasi Vartak, Silu Huang, Tarique Siddiqui, Samuel Madden, and Aditya Parameswaran. 2017. Towards Visualization Recommendation Systems. *SIGMOD Records* 45, 4 (May 2017), 34–39. https://doi.org/10.1145/3092931.3092937
[61] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. 2015. SeeDB: efficient data-driven visualization recommendations to support visual analytics. *Proceedings of the VLDB Endowment* 8, 13 (2015), 2182–2193. https://doi.org/10.14778/2831360.2831371
[62] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by Example. *Proc. ACM Program. Lang.* 4, POPL, Article 49 (Dec. 2019), 28 pages. https://doi.org/10.1145/3371117
[63] Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis.* Springer-Verlag New York. https://ggplot2.tidyverse.org

[64] Cornellius Yudha Wijaya. 2021. Quick Recommendation-Based Data Exploration with Lux. https://towardsdatascience.com/quick-recommendation-based-data-exploration-with-lux-f4d0ccb68133
[65] Leland Wilkinson. 2005. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg.
[66] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2016. Towards a general-purpose query language for visualization recommendation. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM, 4.
[67] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2016. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (2016), 649–658. https://doi.org/10.1109/TVCG.2015.2467191
[68] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2017. Voyager 2 : Augmenting Visual Analysis with Partial View Specifications. (2017). https://doi.org/10.1145/3025453.3025768
[69] Kanit Wongsuphasawat, Daniel Smilkov, James Wexler, Jimbo Wilson, Dandelion Mane, Doug Fritz, Dilip Krishnan, Fernanda B Viégas, and Martin Wattenberg. 2017. Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE transactions on visualization and computer graphics* 24, 1 (2017), 1–12. https://doi.org/10.1109/TVCG.2017.2744878
[70] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *Proceedings of the VLDB Endowment* 6, 8 (2013), 553–564. https://doi.org/10.14778/2536354.2536356
[71] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. 2020. B2: Bridging Code and Interactive Visualization in Computational Notebooks. In *ACM User Interface Software & Technology (UIST)*. http://vis.csail.mit.edu/pubs/b2
[72] Cong Yan and Yeye He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *International Conference on Management of Data (SIGMOD)*.