



Lux: Always-on Visualization Recommendations for Exploratory Dataframe Workflows

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 APIs, 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 dataframe 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 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 3.1k stars on Github.

PVLDB Reference Format:

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. Lux: Always-on Visualization Recommendations for Exploratory Dataframe Workflows. PVLDB, 15(3): 727 - 738, 2022.
doi:10.14778/3494124.3494151

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/lux-org/lux>.

1 INTRODUCTION

Exploratory 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 [35, 53], such as pandas [65], 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 [12]. 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 an unwieldy 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. [54] 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-templated *visualization specification* code snippets copy-and-pasted across the notebook. Our findings echo a recent study of 6386 Github notebooks [41], 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 [16, 69], 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*” [17, 38], 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, and automated assistance is not provided.

Always-On Dataframe Visualizations 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. 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 9k (with a total of 62k downloads), and over 3.1k stars on Github, as of November 2021*. Multiple industry users have created blog posts or YouTube videos extolling the virtues of Lux [7–9, 25, 52, 71].

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494151

Contributions. Our contributions are as follows.

First, we introduce a novel, always-on framework that provides visualizations for the dataframe as it stands at any point in the workflow (§3). This is in contrast with existing visualization specification libraries [33, 66, 70] that require users to write substantial code to generate visualizations. This multi-tiered dataframe interaction framework supports pandas’ 600+ operators without compromising the ease and flexibility of data transformation and analysis (§4).

Second, we introduce an expressive and succinct *intent* language powered by a formal, algebra that allows users to specify their fuzzy intent at a high level. Compared to existing languages [46, 51, 61, 72], the intent language in Lux not only allows users to create one or more visualizations but also flexibly indicate their high-level analysis interest, without worrying about how the data elements map onto aspects of the visualization (§5).

Third, we introduce a novel recommendation system that uses automatically extracted information about dataframes to implicitly infer the appropriate visualizations to recommend. This is in contrast with most existing visualization recommendation systems, which are situated in GUI-based charting tools, whereas Lux is one of the first of such systems that is designed to fit into a programmatic dataframe workflow. In particular, we introduce two novel classes of recommendations based on dataframe structure and history specific to such workflows (§6).

Fourth, we identify opportunities wherein we can adapt techniques from approximate query processing [22, 27], early pruning [39, 47, 68], caching and reuse [29, 64], and asynchronous computation [19, 76] to provide interactive feedback, which is critical for usability; Lux adds no more than two seconds of overhead on top of medium-to-large real-world datasets (§8).

Finally, we evaluate the interactive latency of Lux (§9) and usability with early adopters (§10) that assess the effectiveness of this lightweight, always-on approach to visualizing dataframes.

2 RELATED WORK

Lux draws from work on visualization recommendation systems, visualization specification, and visual dataframe tools.

Visualization Recommendation (VisRec). To visualize data, data scientists need to subselect the aspects of data, and then define a mapping from data to graphical encodings. Interactive interfaces, such as Tableau [3, 63] and PowerBI [11], offer easy-to-use interfaces for visualization construction. Some systems offer suggestions on other possible visualizations for users to browse through, as visualization recommendations. VisRec systems can either suggest interesting portions of the data to visualize based on statistical properties [23, 37, 43, 50, 61, 67, 68] or better ways to visualize attributes that users have selected [31, 48, 49, 51, 72]. Similarly, there has been research on recommending interesting attributes or filters to avoid manual data exploration during OLAP [36, 42, 55–57, 74]. While interactive GUI-based 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. Lux draws on recommendation principles from this literature and explores how visualization recommendations can support a dataframe workflow. Moreover, Figure 5 outlines a novel, multi-tiered framework that Lux employs to support flexible visual and programmatic

interactions with a dataframe, overcoming the limitation in expressiveness of existing GUI-based VisRec tools.

Visualization Specification (VisSpec). VisSpec frameworks codify visualization design principles and best practices to simplify the task of creating a visualization [20, 58, 59, 62, 70]. These frameworks encompass a range of abstractions depending on the degree to which users are required to specify low-level details associated with the visualization definition. For example, *imperative* visualization libraries, such as plotly [34], D3 [20], and matplotlib [33], require users to manually compute the data associated with the graphical elements (e.g., position or size of marks) before defining the visualization characteristics. *Declarative* visualization languages, such as Altair [66] and Vega-Lite [58], enable rapid specification of visualizations by applying smart defaults to synthesize low-level visualization details, so that users are not required to specify common chart components, such as axes, ticks, and labels. Lux is built on top of these imperative and declarative frameworks and synthesizes visualization code to enable users to customize as needed.

Partial specification languages, such as Draco [51] and CompassQL [72], commonly employed in VisRec systems, support reasoning based on a partial specification provided by the user and design constraints encoded in the system. A partial specification can be thought of as a “query”, with the system automatically ranking a set of perceptually-effective visualizations that match the query. As we will see in Section 5, the intent language in Lux is more convenient to specify than these existing languages in that it only requires users to specify data aspects of interest (or omit them entirely), instead of having to worry about visualization encodings. Lux is also more versatile in that it supports functionalities beyond visualization creation for steering the recommendations generated. That said, as a promising direction for future work, Lux could make use of Draco’s sophisticated reasoning around visualization design to improve which visualizations are displayed, going beyond the rule-based heuristics in its current implementation.

Compared to imperative, declarative, and partial VisSpec frameworks, Figure 6 illustrates how Lux’s intent language further reduces the specification burden on users, allowing them to provide lightweight intent as opposed to writing long code fragments for visualization; we will elaborate on this in Section 5.

Visual Data Exploration with Dataframes. Of late, dataframes have become the de-facto framework for interactive data science. The comprehensive, incremental set of operators make it easy to do sophisticated data transformation, while also allowing validation after each step. However, exploring dataframes is challenging, requiring substantial programming and analytical know-how. Many visualization tools have been developed for dataframes [1, 5, 15, 26, 60]. These tools generate summaries, covering analyses spanning missing values, outliers, attribute-level visualizations, and associated statistics. In addition, bambolib [5] and pandasgui [15] offer a GUI for constructing visualizations and data transformations. Unlike these existing tools, Lux lowers the barrier to visualizing dataframes by adopting an always-on approach so that dataframe visualizations are always recommended to users at all times.

3 EXAMPLE WORKFLOW

In this example workflow, we demonstrate how always-on visualization support for dataframes accelerates exploration and discovery.

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 at <http://tinyurl.com/demo-lux>.

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 prints¹ 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 `AvgLifeExpectancy` 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.

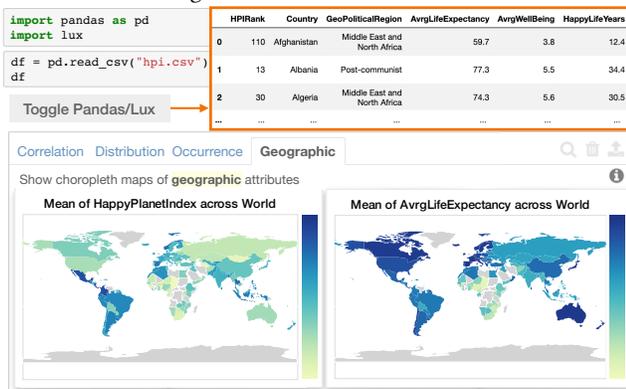


Figure 1: By printing out the dataframe, the default pandas tabular view is displayed (orange box) and users can toggle to browse through visualizations recommended by Lux.

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 in Figure 2, she specifies her analysis *intent* to Lux as: `df.intent = ["AvgLifeExpectancy", "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.

¹We 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)'.

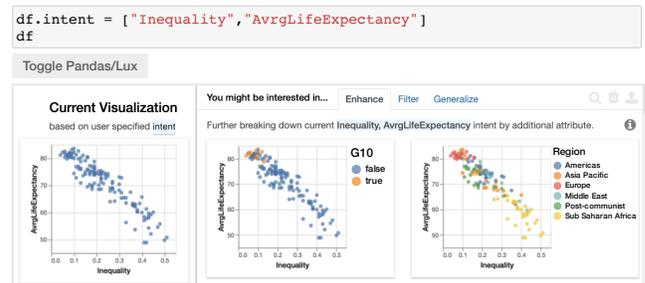


Figure 2: Alice sets the intent based on the attribute `AvgLifeExpectancy` and `Inequality`, and Lux displays visualizations that are related to the intent.

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` [30], 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 modi-

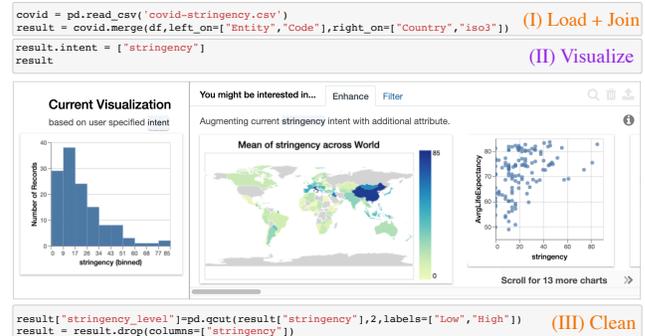


Figure 3: Tabular operations (orange, steps I & III) to load, clean, and transform the data, while visualizing with Lux (purple, step II).

fied 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 [4, 6, 18]. 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.

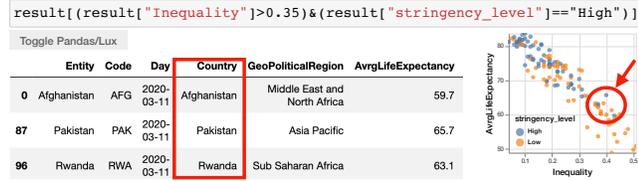


Figure 4: The scatterplot shows a separation between countries with high and low stringency in their COVID response. By filtering the dataframe (left), we see that Afghanistan, Pakistan, and Rwanda correspond to the three outliers (red boxed) that defies the trend.

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 data cleaning via a familiar API and notebook environment.

4 INTERACTING WITH DATAFRAMES

In this section, we propose a novel always-on framework for visual interaction with dataframes as outlined in Figure 5. The example workflow illustrated the many flexible ways users interact with a dataframe to achieve their analytical goal. Here, we summarize these ways and contrast it to existing visualization specification approaches in dataframe workflows.

As shown in Figure 5, in both existing dataframe workflows (a) and the always-on framework (b), users can work with the dataframe API and see the table view by default (grey). For creating visualizations in an existing workflow, as shown in Figure 5a, users would typically need to explicitly write visualization specification code in a language such as `matplotlib` or `altair` (orange) to create individual visualizations (blue). In our always-on framework, as shown in Figure 5b, users further inspect a dashboard of recommended visualizations, as part of a multi-tiered framework (blue), all of which is driven by a user- or system-specified intent (orange), described below.

Intent: Users can indicate aspects of the dataframe that they are interested in via a lightweight intent specification (§5). The intent drives the visualizations, actions, and dashboard. In the example, Alice indicated that she wants to learn about `AvrgLifeExpectancy` and `Inequality`; Lux displayed visualizations related to these variables. Unlike existing visualization libraries, intent can also be system-specified—meaning that the visual display will be always-on, even if the user does not explicitly specify intent. We now describe the different layers in our always-on framework, following the notation in Figure 5.

(A) Visualization: Visualizations are created by applying the intent to a given dataframe. Each visualization, i.e., `Vis`, is an intent operating on a specific dataframe instance; a collection of visualizations is known as a `VisList`.

(B) Actions: Each action is an ordered collection of visualizations (`VisList`), e.g., the `Correlation` action plots pairwise relationships ranked by Pearson’s correlation.

(C) Dashboard: A dashboard is composed of one or more actions that may be relevant to the user.

Users can either make changes to the dataframe or the intent in order to fulfill different analytical needs. Dataframe operations are

exact, leveraging the expressiveness of the dataframe API. On the other hand, the intent is a high-level specification of user interest, either explicitly provided by the user or system-inferred, steering Lux’s recommendations.

By working with both the intent and dataframe API, Lux supports a flexible and intuitive experience for interacting with data. Next, we describe the intent grammar that underlies the always-on framework of Lux.

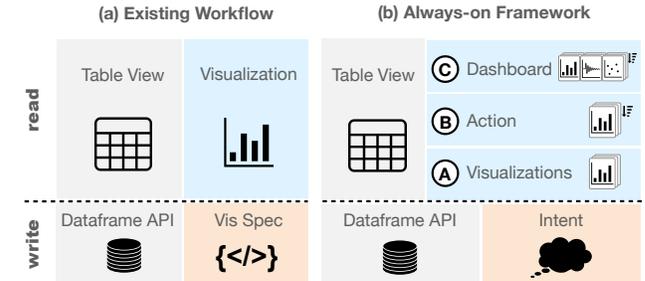


Figure 5: Conceptual framework for dataframe interaction. Users can make changes to anything below the dotted line (write), elements displayed to the user are shown above the dotted line (read). (a) In existing workflows, users write visualization specification code to create one or more visualizations. (b) In Lux’s always-on framework, users can optionally make changes to the intent, which steers the recommended visualizations (Visualizations, Action, Dashboard).

5 INTENT LANGUAGE FORMALIZATION

The *intent language* is a lightweight, succinct means for users to declaratively specify their high-level interests. In this section, we introduce this language and its underlying grammar, and how it differs from existing approaches.

5.1 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.

$$\begin{aligned} \langle \text{Intent} \rangle &\rightarrow \langle \text{Clause} \rangle^+ \\ \langle \text{Clause} \rangle &\rightarrow \langle \text{Axis} \rangle \mid \langle \text{Filter} \rangle \end{aligned} \quad (1)$$

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

$$\langle \text{Axis} \rangle \rightarrow \langle \text{attribute} \rangle^* \langle \text{channel} \rangle \langle \text{aggregation} \rangle \langle \text{bin_size} \rangle \quad (2)$$

For the axis, apart from the mandatory attribute(s), specified under `attribute`, the remaining properties are optional—and can be automatically inferred.

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.

$$\langle \text{Filter} \rangle \rightarrow \langle \text{attribute} \rangle [= > < \leq \geq \neq] \langle \text{value} \rangle \quad (3)$$

Consider the simple case when `attribute` refers to a single attribute and `value` 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, `attribute` can correspond to a union of attributes, or a special wildcard value `?` (with an optional constraint to define the subset of attributes), while the `value` can

refer to a union of values, or a special wildcard value (?).

$\langle attribute \rangle \rightarrow attribute \cup \langle attribute \rangle^* \mid (?) \langle constraint \rangle$ (4)

$\langle value \rangle \rightarrow value \cup \langle value \rangle^* \mid (?)$ (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 $n_1 \times n_2 \times \dots \times n_k$ visualizations by taking the cross-product of alternatives per clause.

5.2 Specifying Intent

Users can specify an intent indicating their analysis interests or create desired visualizations by applying the intent to a specific dataframe. We note that while the focus here is describing user-specified intent, the same intent language is used by the system for generating recommendations as will be described in Section 6.

5.2.1 Attaching an Intent to a Dataframe. 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.

Q1. 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.

We can compose Axis and Filter together, as follows.

Q2. Explore the Ages for employees in the Sales Department.

```
axis = "Age"
filter = "Department=Sales"
df.intent = [axis, filter]
```

Based on the specified intent, Lux not only shows the Age distribution filtered to the Sales department, but also displays a set of related visualizations, such as visualizations involving one additional attribute or one additional filter.

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.

5.2.2 Constructing Visualizations Directly via Intent. Instead of attaching an intent to a dataframe, one can use the Vis and VisList keyword to directly generate specific visualizations.

Q3. Compare average Age across different Education levels.

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

Query 3 is similar to Query 1, except that the intent is immediately 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.

Q4. Compare the variance of MonthlyIncome based on employee Attrition.

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

To generate multiple visualizations, one could either set df.intent as in Section 5.2.1, which would generate a collection of visualizations related to the intent, or specify intent as an input to a VisList, as in the following query.

Q5. Show how factors related to the rate of compensation differ for employees with different EducationFields.

```
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; constraints may be used to restrict them to a certain type.

Q6. 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.

Q7. 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.

Required Specification	
Legend	
■ Data source	■ Field Name
■ Data operation	■ Channel
■ Field Type	■ Mark
Intent	Imperative
<pre>Vis(["Age", "Education"],df)</pre> <p>Lux</p>	<pre>bar=df.groupby("Education").mean()["Age"] y_pos=range(len(bar)) plt.barh(y_pos,bar,align='center') plt.yticks(y_pos,list(bar.index)) plt.xlabel('Mean of Age') plt.ylabel('Education')</pre> <p>matplotlib</p>
Declarative	Partial
<pre>{ "mark": "bar", "data": "...", "encoding": { "x": { "type": "quantitative", "field": "Age", "aggregate": "average" }, "y": { "type": "nominal", "field": "Education", } } }</pre> <p>Vega-Lite</p>	<pre>data(...). encoding(e0). channel(e0,x). type(e0,quantitative). field(e0,"Age"). aggregate(e0,mean). encoding(e1). channel(e1,y). type(e1,nominal). field(e1,"Education").</pre> <p>Draco</p>

Figure 6: Comparison between the level of specification required from Lux versus other existing approaches for Query 3.

5.3 Rationale for Design Choices

Due to the heavy cognitive cost of writing glue code to visualize their data [16, 69], users often opt to visualize in the later stages of their workflow [17, 38]. Instead, our goal with Lux's intent language has been to support visualization to be used throughout; and for this, users should not have to expend too much effort in thinking about what and how to visualize. There are two key characteristics of our

intent language that support this quick and flexible programmatic specification, described next.

Versatility: Our intent language is *versatile* in that it serves both as a mechanism for steering recommendations (Q1-2) and as a way of directly creating visualizations on top of dataframes (Q3-7). This is unlike existing specification approaches whose sole focus is the creation of one or more visualizations. This versatility means that whenever users specify their intent, they are not committing to a pre-defined set of operations. Instead, the system leverages explicit user input (in the form of intent), as well as implicit signals to determine what to display to users.

Consider Q2, which demonstrates the versatility of intent beyond the specification of a single visualization. Here, the user simply specifies the data-specific aspects they are interested in, i.e., the attribute Age, and the Sales Department filter; these are used as cues by LUX to generate visualizations, including those that wouldn't ordinarily be picked if we were using a conventional visualization specification framework (such as those with a different filter). This versatility makes it easy for users to communicate their analysis intent even when they do not have a specific visualization in mind.

Convenience: Our intent language only requires specification of data-oriented aspects, while existing approaches also require users to specify visual encoding-oriented aspects to generate visualizations. Our minimalistic language design is intended to alleviate the common challenge in exploratory analysis where users struggle to translate their high-level data questions to exact visualization specifications [28]. LUX supports convenient specification short-hands and defaults and automatically infers the necessary details to transform user-specified intent into complete specifications.

As shown in Q3-7, where the target is one or more specific visualizations, LUX enables users to visualize their data with only a single line of code, effectively lowering the barrier to visual exploration. In Figure 6, we outline the code required to create a single visualization based on Query 3, and compare the key differences in the required specification across various languages, including Draco [51], matplotlib [33], and Vega-Lite [58]. Other languages often require users to specify the field type, channel, and marks, while LUX can reason over underspecified intent. This reduces the effort required on the part of the user.

6 DATAFRAME 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 ranked list of visualization recommendations based on a predefined search space. LUX supports four major classes of actions. Metadata- and intent-based ones are akin to those used in past visualization recommendation systems [32, 44, 73]—see our technical report [45] for details. As described in Section 2, most existing VisRecs are situated in GUI-based charting tools; our key novelty is that LUX is one of the first visualization recommendation systems that is designed to fit into a programmatic dataframe workflow. Specifically, here, we introduce two novel classes of recommendations specific to dataframe-based workflows, based on dataframe structure and history.

Structure-based recommendations. Data scientists often reshape their dataframes in ways that are more amenable to downstream

analysis, modeling, or presentation. One of our key insights is that the dataframe “structure” reveals strong signals for what the users subsequently choose to visualize, thus providing implicit information on what recommendations to display automatically by LUX.

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 row or column-wise. For example, Figure 7 displays the result of a `pivot` operation, where each row is visualized as a time series line chart.

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.

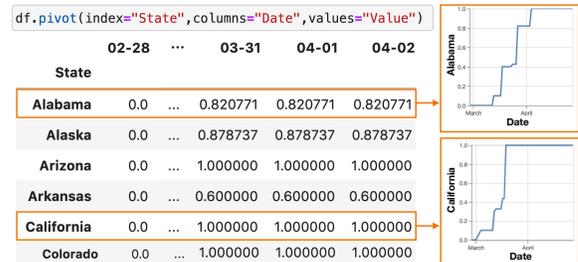


Figure 7: Row-wise index visualization displaying the normalized percentage of COVID-19 cases across different States.

History-based recommendations. Apart from dataframe structure, another source of implicit information from the dataframe is the historical set of operations performed by users. 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 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.

7 SYSTEM DESCRIPTION

LUX employs a client-server model, leveraging computational notebooks as a frontend client. 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 8, the server side logic is largely separated into two distinct layers: 1) the *intent processing* layer is responsible for processing intent into executable instructions, and 2) the *recommendation* layer is responsible for generating the displayed visualizations. 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 in relational databases (§8). Finally, the system design is modular and extensible so that alternatives can be swapped in at different layers, e.g., Altair and Matplotlib visualization rendering libraries.

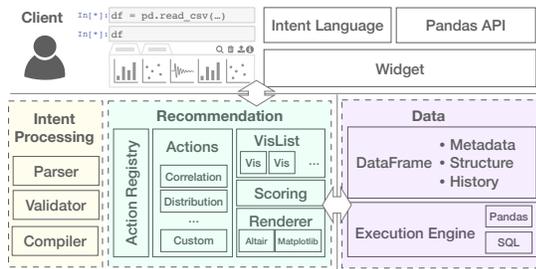


Figure 8: System architecture for Lux

8 EXECUTION AND OPTIMIZATION

LUX’s execution engine performs two major tasks: 1) compute the metadata required to generate candidate visualizations, and 2) extract the actual data for each visualization.

Metadata Computation: The metadata computed includes attribute-level statistics and data types. The statistics include the list of unique values, cardinality, and min/max values for each attribute.

Visualization Processing: After the user or system-specified intent has been transformed into one or more visualizations with a complete specification, the execution engine translates each visualization to queries responsible for processing the data required for the visualizations. The engine applies any filters and projections on the data before performing different visualization-specific operations, such as binning and aggregation. Computing metadata and processing data for visualizations can be time consuming, even for a moderately-sized dataframe. Therefore, we adapt optimizations from approximate query processing [22, 27], early pruning [39, 47, 68], caching and reuse [29, 64], and asynchronous computation [19, 76], to improve the interactivity of 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 intersperse 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["val2"] = df["val"]/5`

- 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 metadata and generates the recommendations accordingly. This lazy strategy ensures no overhead on any non-print operations.

LUX further memoizes the metadata and recommendations so that any subsequent prints to an unmodified dataframe do not require re-computation. Users frequently perform “non-committal” operations that do not make changes to the dataframe to be used in subsequent analyses, involving printing dataframes as intermediate results to facilitate quick experimentation and debugging. For instance, users may print a column, perform grouping and aggregation, or print descriptive summaries, all without modifying the dataframe. In this case, when the user revisits the original dataframe, the memoized recommendations are immediately accessible to them.

While lazy computation and caching and reuse are well-studied [29, 64, 75], identifying common dataframe usage patterns and determining when and how to expire metadata and recommendations in a dataframe workflow are novel contributions.

Approximate, early pruning of search space (PRUNE): As described in Section 4, each visualization in an action is ranked based on a scoring function, computed based on the data associated with each visualization. Inspired by existing work in early pruning [39, 47, 68], LUX estimates the visualization score to speed up the retrieval of top-k visualizations for each action. We employ approximate query processing to reduce the cost by estimating the scores using sampled data. Specifically, LUX first performs a preliminary pass over the `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*. Currently LUX leverages a cached sample of the dataframe to approximate visualization scores (e.g., for a dataframe with 1M rows, approximating correlation score by using only 30k rows), although other approximate query processing methods could be applied.

Table 1: Table summarizing the relational operations performed for processing different visualizations.

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	Bin + Count
Heatmap	2D Bin + Count
Color Heatmap	2D Bin + Count + Group-By Aggregation

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`. For example, dataframes that are wide or contain high-cardinality attributes can often result in actions involving large visualization search spaces. 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. The cost of scoring a visualization, t_{exact} and t_{approx} , is

determined by the relational operations for extracting the required visualization data (as shown in Table 1).

Here, while the use of approximate samples to rank and identify top-k visualizations is not new [47, 68], our use of approximation in conjunction with a cost model to determine its potential interestingness is a novel application of the technique.

Cost-based scheduling of actions (ASYNC): We find that users generally spend an average of 28 seconds² skimming through the pandas table view before toggling to the Lux view. Leveraging past work on asynchronous query execution [19, 76], recommendation results can be streamed into the frontend widget as the computation for each action completes to ensure interactive responses, 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 described in our technical report [45]. 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 `ASYNC` 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.

The idea of exploiting asynchronous execution during user wait-time has been well-established [19, 76], but our work is the first to apply this technique in a visualization recommendation context, by leveraging cost estimates to prioritize cheaper-to-compute visualizations. Our cost model across different visualization types is an independent valuable contribution.

9 PERFORMANCE EVALUATION

We evaluate Lux to measure its performance on large real-world datasets and notebook sessions. 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 are available online³.

Data: We use two real-world datasets to evaluate the performance of Lux. The `Airbnb` dataset [24] contains 12 columns while the `Communities` [40] 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 [14].

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` [10] 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

²Based 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.

³<https://github.com/lux-org/lux-benchmark>

for approximating the visualization interestingness of dataframes over 30k rows. 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 to the dataframe⁴.
- **wflow:** Condition with the `WFLOW` optimization applied.
- **wflow+prune:** Both `WFLOW` and `PRUNE` applied.
- **all-opt:** All `WFLOW`, `PRUNE`, and `ASYNC` applied, representing the best achievable performance.
- **pandas:** Condition with only pandas and *without* using Lux, representing the raw performance of dataframe workflows without the benefits of always-on visualizations.

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 2 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.

Table 2: 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.).

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

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

⁴This condition is akin to the naive implementation in most visualization recommendation systems, where results are updated whenever the dataset is operated on.

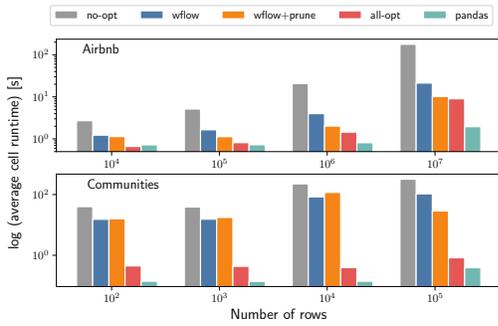


Figure 9: Average runtime of a notebook cell across the workload for different dataframe size and conditions.

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.

As shown in the sparkline visualization in Table 2 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.

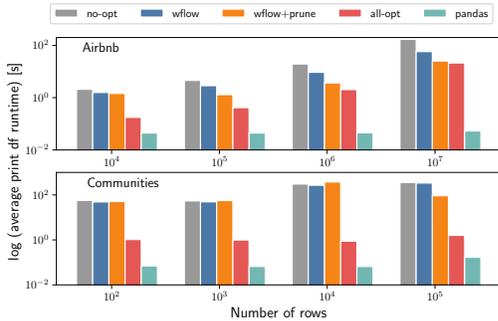


Figure 10: Average time for printing a single dataframe for different dataframe size and conditions.

Non-Lux operations: Across all conditions except the baseline, the runtime for non-Lux operations (Table 2 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 10M Airbnb and over 650X faster for 100k Communities. The performance improvement for non-Lux operations demonstrates how WFLOW’s lazy evaluation strategy avoids unnecessary computation.

Additional experiments: Details of additional performance experiments investigating how LUX performs on dataframes of varying sizes, as well as the recommendation accuracy for the PRUNE optimization can be found in the technical report [45].

10 POST-DEPLOYMENT EVALUATION

LUX was first released in October 2020 and gained substantial traction in the open-source community since then. In this section, we report on a field study with existing users of LUX and lessons

learned from developing LUX. We summarize the key findings that would potentially be informative to future tool designers. Details of other post-deployment evaluations, including a first-use study, can be found in the technical report [45].

Field Study Interviews 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 existing workflows. Before the interviews, participants used LUX over the span of 1-2 months in their professional data science work. We performed a walk-through of real-world notebooks in which participants had used LUX.

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. [16]), 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 user-specified views: Somewhat surprisingly, the use of Vis and VisList 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., matplotlib 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.

Summary and Limitations: 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. We learned that `Vis` and `VisList` are not as discoverable and easy to use as always-on dataframe visualizations. Despite the enthusiasm around LUX, we find that participants are 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. Furthermore, a controlled study comparing LUX with a manual baseline approach would further quantify the expected benefits of the tool.

Lessons from Developing LUX. We summarize the implementation challenges and lessons learned from the longitudinal open-source deployment of LUX over 15 months, with over 62k downloads. Given that the nature of such engagement is largely organic, ranging from feature requests stemming from Github Issues to questions and discussions with users in our Slack community, these observations and findings will remain qualitative.

Metadata Propagation: To preserve the comprehensive array of convenient operators offered by the dataframe API, we aimed to natively support any possible pandas operations. This led to our design of `LuxDataFrame` as a wrapper around the pandas dataframe. However, dataframes can often get transformed to other intermediate data structures such as `GroupBy`, `Series`, or `Index` objects when users are working with dataframes. To this end, we extended specific pandas functions and classes to ensure that the metadata and associated information is propagated across a workflow, so that the context does not get lost in intermediate operations.

Failproofing always-on dataframe display: One of the reasons why dataframes have become popular is the ease and flexibility of being able to work with the data in a schema-free manner [53]. However, this can be problematic for generating recommendations, since

underlying operations for visualization recommendations can lead to errors. For instance, dataframes can often be ill-formatted in a way that is not amenable to visualizations. One example of this is dataframes with missing values, or when dataframes contains mixed datatypes (a common issue when loading in spreadsheets). To this end, we provide pandas-consistent output behavior by safeguarding LUX with informative warnings, and falling back to pandas upon internal errors, to always ensure that LUX provides at least the pandas table as the default display. To allow users to effectively operate on their data, it is crucial that the system provides users with the *unmodified, consistent* state of the dataframe. In other words, LUX should not modify and change the user’s dataframe in the process of visualization to maintain “What You See Is What You Get” (WYSIWYG) behavior. As a result, all recommendations in LUX are generated as views that are decoupled from the dataframe content.

Integration with Downstream Reports: One common use case for LUX is to get a quick overview of insights on a new dataset. We found that users often wanted a way to share their findings in their organization. Our initial use case for supporting downstream reporting was to allow users to select one or more visualizations and export it as matplotlib or altair code. However, this approach quickly became unsustainable when users wanted to share many visualizations from their dashboard at the same time. To support presentation and collaboration, we implemented various options for export, from static HTML reports to integration with popular libraries for creating interactive “data apps”, such as Streamlit [13]. Given that many LUX users often share their findings with business stakeholders without a Python development setup, future work might include supporting exports to readily-accessible presentation formats, such as PDF or Powerpoint.

Another lesson that we learned is that ease of initial installation and setup is a primary driver impacting the adoption of tools like LUX. In a similar vein, our user surveys and online discussions suggest that the minimal API is attractive to many data practitioners.

11 CONCLUSION

We propose LUX, an always-on visualization framework for exploratory dataframe workflows. LUX is a lightweight wrapper that reduces the barrier of visualizing data, enabling seamless exploration and visual discovery in-situ. To provide better visualization recommendations, we make use of user-provided intent and history, as well as structural information and metadata. We extend and evaluate various optimization strategies that minimize the overhead of LUX, including approximate query processing, lazy computation, and caching and reuse. LUX’s adoption over the last year and success of user evaluation points to its importance for dataframe workflows—steering users towards valuable insights as they ponder what to do next with their data.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work is supported by a Facebook Fellowship, grants IIS-2129008, IIS-1940759, and IIS-1940757 awarded by the National Science Foundation, and funds from the Alfred P. Sloan Foundation. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies and organizations.

REFERENCES

- [1] pandas-profiling. <https://github.com/pandas-profiling/pandas-profiling>.
- [2] Happy Planet Index. <http://happyplanetindex.org/>, 2019.
- [3] Tableau. <https://www.tableau.com/>, 2019. Accessed: 2019-09-11.
- [4] Afghanistan: WHO mission reviews COVID-19 response. *World Health Organization*, 2020.
- [5] bamboolib. <https://bamboolib.8080labs.com/>, 2020.
- [6] COVID-19 in Pakistan: WHO fighting tirelessly against the odds. *World Health Organization*, 2020.
- [7] Faster data exploration in Jupyter through Lux, 2020.
- [8] LUX Exploratory Data Analysis (EDA). *YouTube*, Dec 2020.
- [9] LUX Library: Matplotlib replacer? *YouTube*, Dec 2020.
- [10] papermill 2.3.3 documentation. <https://papermill.readthedocs.io/>, 2020.
- [11] Power BI: Interactive data visualization BI Tools. 2020.
- [12] State of Data Science and Machine Learning 2020. *Kaggle*, 2020.
- [13] Streamlit. <https://streamlit.io/>, 2021.
- [14] UCI Machine Learning Repository, Mar 2021.
- [15] adamerose. PandasGUI. <https://github.com/adamerose/pandasgui>.
- [16] S. Alspaugh, N. Zokaie, A. Liu, C. Jin, and M. A. Hearst. Futing and moseying: Interviews with professional data analysts on exploration practices. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):22–31, 2019.
- [17] A. Batch and N. Elmqvist. The Interactive Visualization Gap in Initial Exploratory Data Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):278–287, 2018.
- [18] J. Beaubien. Why Rwanda Is Doing Better Than Ohio When It Comes To Controlling COVID-19. *NPR*, July 2020.
- [19] M. Bendre, T. Wattanawaroon, K. Mack, K. Chang, and A. G. Parameswaran. Antifreeze for large and complex spreadsheets: Asynchronous formula computation. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1277–1294. ACM, 2019.
- [20] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents.
- [21] J. Brooke. *"SUS-A quick and dirty usability scale."* Usability evaluation in industry. CRC Press, June 1996. ISBN: 9780748404605.
- [22] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends Databases*, 4(1-3):1–294, 2012.
- [23] Ç. Demiralp, P. J. Haas, S. Parthasarathy, and T. Pedapati. Foresight: Rapid data exploration through guideposts. *arXiv preprint arXiv:1709.10513*, 2017.
- [24] Dgomonov. Data Exploration on NYC Airbnb. *Kaggle*, Aug 2020.
- [25] P. Duvva. Speed up EDA With the Intelligent Lux. *Medium*, Mar 2021.
- [26] Fbdesignpro. sweetviz. <https://github.com/fbdesignpro/sweetviz>.
- [27] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 2001.
- [28] L. Grammel, M. Tory, and M. Storey. How information visualization novices construct visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):943–952, Nov. 2010.
- [29] H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.
- [30] T. Hale, N. Angrist, R. Goldszmidt, B. Kira, A. Petherick, T. Phillips, S. Webster, E. Cameron-Blake, L. Hallas, S. Majumdar, and H. Tatlow. A global panel database of pandemic policies (Oxford COVID-19 Government Response Tracker). *Nature Human Behaviour*, 2021.
- [31] K. Hu, M. A. Bakker, S. Li, T. Kraska, and C. Hidalgo. Vizml: A machine learning approach to visualization recommendation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI '19*, page 1–12, New York, NY, USA, 2019. Association for Computing Machinery.
- [32] K. Hu, D. Orghian, and C. Hidalgo. Dive: A mixed-initiative system supporting integrated data exploration workflows. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 5. ACM, 2018.
- [33] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.
- [34] P. T. Inc. Collaborative data science, 2015.
- [35] A. Jindal, K. V. Emani, M. Daum, O. Poppe, B. Haynes, A. Pavlenko, A. Gupta, K. Ramachandra, C. Curino, A. C. Müller, W. Wu, and H. Patel. Magpie: Python at speed and scale using cloud backends. In *CIDR*, February 2021.
- [36] M. Joglekar, H. Garcia-Molina, and A. Parameswaran. Smart Drill-Down: A New Data Exploration Operator. *Proceedings of the 41st International Conference on Very Large Data Bases*, 8(12):1928–1931, 2015.
- [37] N. Kamat, E. Wu, and A. Nandi. Trendquery: A system for interactive exploration of trends. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA '16*, pages 12:1–12:4, New York, NY, USA, 2016. ACM.
- [38] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE transactions on visualization and computer graphics*, 18(12):2917–26, 2012.
- [39] A. Kim, E. Blais, A. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld. Rapid sampling for visualizations with ordering guarantees. *Proceedings of the VLDB Endowment*, 8(5):521–532, 2015.
- [40] Kkanda. Analyzing UCI Crime and Communities Dataset. *Kaggle*, Mar 2020.
- [41] A. P. Kozen, N. A. Ernst, and M. A. D. Storey. Code duplication and reuse in jupyter notebooks. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–9, 2020.
- [42] D. J.-L. Lee, H. Dev, H. Hu, H. Elmeleegy, and A. Parameswaran. Avoiding Drill-down Fallacies with VisPilot: Assisted Exploration of Data Subsets. In *Proceedings of the 24th International Conference on Intelligent User Interfaces, IUI '19*, pages 186–196, New York, NY, USA, 2019. ACM.
- [43] D. J.-L. Lee and A. Parameswaran. 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):3–14, 2018.
- [44] D. J.-L. Lee, V. Setlur, M. Tory, K. Karahalios, and A. Parameswaran. Deconstructing Categorization in Visualization Recommendation: A Taxonomy and Comparative Study. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–15, 2021.
- [45] D. J.-L. Lee, D. Tang, K. Agarwal, T. Boonmark, C. Chen, J. Kang, U. Mukhopadhyay, J. Song, M. Yong, M. A. Hearst, and A. G. Parameswaran. Lux: Always-on visualization recommendations for exploratory data science, 2021.
- [46] H. Lin, D. Moritz, and J. Heer. Dziban: Balancing Agency & Automation in Visualization Design via Anchored Recommendations. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems - CHI '20*, 2020.
- [47] S. Macke, Y. Zhang, S. Huang, and A. Parameswaran. Fastmatch: Adaptive algorithms for rapid discovery of relevant histogram visualizations. *PVLDB*, 2017.
- [48] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, 1986.
- [49] J. D. Mackinlay, P. Hanrahan, and C. Stolte. Show Me: Automatic presentation for visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1137–1144, 2007.
- [50] M. Mannino and A. Abouzied. Expressive Time Series Querying with Hand-Drawn Scale-Free Sketches. pages 1–12, 2018.
- [51] D. Moritz, C. Wang, G. L. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer. Formalizing visualization design knowledge as constraints: Actionable and extensible models in draco. *IEEE transactions on visualization and computer graphics*, 25(1):438–448, 2019.
- [52] P. Pandey. Intelligent visual data discovery with lux: A python library. *Medium*, Mar 2021.
- [53] D. Petersohn, S. Macke, D. Xin, W. Ma, D. Lee, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran. Towards scalable dataframe systems. *Proc. VLDB Endow.*, 13(12):2033–2046, July 2020.
- [54] A. Rule and U. C. S. Diego. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. 2(November), 2018.
- [55] S. Sarawagi. Explaining differences in multidimensional aggregates. *Proceedings of the VLDB Endowment*, pages 42–53, 1999.
- [56] S. Sarawagi. User-adaptive exploration of multidimensional data. *Proc of the 26th Intl Conference on Very Large*, pages 307–316, 2000.
- [57] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-Driven Exploration of OLAP Data Cubes. In *Proceedings of the 6th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '98*, page 168–182, Berlin, Heidelberg, 1998. Springer-Verlag.
- [58] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2017.
- [59] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE transactions on visualization and computer graphics*, 22(1):659–668, 2016.
- [60] sfu-db. dataprep. <https://github.com/sfu-db/dataprep>.
- [61] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. Parameswaran. Effortless data exploration with zenvisage: an expressive and interactive visual analytics system. *Proceedings of the VLDB Endowment*, 10(4):457–468, Nov. 2016.
- [62] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.
- [63] C. Stolte, D. Tang, and P. Hanrahan. Polaris: a system for query, analysis, and visualization of n/multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):1–14, 2002.
- [64] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Interimittent query processing. *Proc. VLDB Endow.*, 12(11):1427–1441, 2019.
- [65] The pandas development team. pandas-dev/pandas: Pandas, Feb. 2020.
- [66] J. VanderPlas, B. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert. Altair: Interactive statistical visualizations for python. *Journal of Open Source Software*, 3(32):1057, 2018.
- [67] M. Vartak, S. Huang, T. Siddiqui, S. Madden, and A. Parameswaran. Towards visualization recommendation systems. *SIGMOD Records*, 45(4):34–39, May 2017.
- [68] M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis. Seedb: efficient data-driven visualization recommendations to support visual analytics. *Proceedings of the VLDB Endowment*, 8(13):2182–2193, 2015.

- [69] C. Wang, Y. Feng, R. Bodik, A. Cheung, and I. Dillig. Visualization by example. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019.
- [70] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016.
- [71] C. Y. Wijaya. Quick recommendation-based data exploration with lux. *Medium*, Feb 2021.
- [72] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Towards a general-purpose query language for visualization recommendation. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 4. ACM, 2016.
- [73] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mane, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg. Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE transactions on visualization and computer graphics*, 24(1):1–12, 2017.
- [74] E. Wu and S. Madden. Scorpion: Explaining Away Outliers in Aggregate Queries. *Proceedings of the VLDB Endowment*, 6(8):553–564, 2013.
- [75] D. Xin, S. Macke, L. Ma, J. Liu, S. Song, and A. Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. *Proceedings of the VLDB Endowment*, 12(4):446–460, 2019.
- [76] D. Xin, D. Petersohn, D. Tang, Y. Wu, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. G. Parameswaran. Enhancing the interactivity of dataframe queries by leveraging think time. *CoRR*, abs/2103.02145, 2021.