arXiv:2105.00121v2 [cs.DB] 22 Dec 2021

# Lux: Always-on Visualization Recommendations for Exploratory Dataframe Workflows

## [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 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 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 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

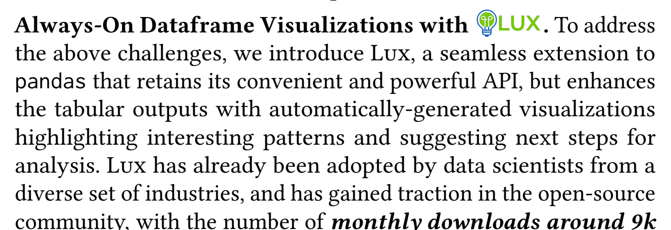
**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/lux-org/lux.

## 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 [41, 60], such as pandas [72], 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 [14]. 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. [61] 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 [47], 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 [21, 76], 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*" [22, 44], 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***

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

***(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 [9–11, 30, 59, 78].

**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 [39, 73, 77] 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 for partial specification [52, 58, 68, 79], 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 [27, 33], early pruning [45, 54, 75], caching and reuse [35, 71], and asynchronous computation [24, 83] 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 [4, 70] and PowerBI [13], 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 [28, 43, 49, 57, 68, 74, 75] or better ways to visualize attributes that users have selected [37, 55, 56, 58, 79]. Similarly, there has been research on recommending interesting attributes or filters to avoid manual data exploration during OLAP [42, 48, 62–64, 81]. 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 [25, 65, 66, 69, 77]. 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 [40], D3 [25], and matplotlib [39], 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 [73] and Vega-Lite [65], 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 [58] and CompassQL [79], 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, 7, 20, 31, 67]. These tools generate summaries, covering analyses spanning missing values, outliers, attribute-level visualizations, and associated statistics. In addition, bamboolib [7], pandas-profiling [1], dataprep [67], sweetviz [31], and pandasgui [20] 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, instead of relying on users to explicitly call external commands to *plot* or *profile* as needed.

## 3 EXAMPLE WORKFOW

In this example workflow, we demonstrate how always-on visualiz-
tion support for dataframes accelerates exploration and discover
We present a workflow of Alice, a public policy analyst, explorin
the relationship between world developmental indicators (such a
life expectancy, inequality, and wellbeing) and the country's earl
effort in COVID-19 response. A live demo of the example noteboo
can be found at http://tinyurl.com/demo-lux.

**Always-on dataframe visualization.** Alice opens up a Jupyte
notebook and imports pandas and Lux. Using pandas's read_cs
command, Alice loads the Happy Planet Index (HPI) [3] datase
of country-level data on sustainability and well-being. To get a
overview, Alice prints[1] the dataframe df and Lux displays the d
fault pandas tabular view, as shown in Figure 1 (top, orange box
By clicking on the toggle button, Alice switches to the Lux view tha
displays a set of univariate and bivariate visualizations (bottom
including scatterplots, bar charts, and maps, showing an overvie
of the trends. Visualizations are organized into sets called *action*
displayed as tabs. The one displayed currently is the Geographi
action. By inspecting the Correlation tab in Figure 1 (not di
played here), she learns that there is a negative correlation betwee
AvrgLifeExpectancy and Inequality (same chart as Figure 2 left

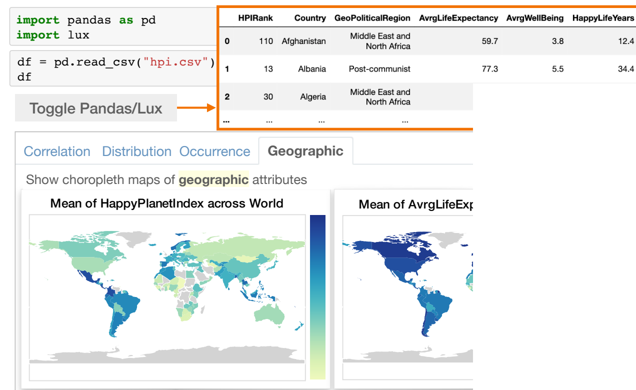



**Figure 1: By printing out the dataframe, the de lar view is displayed (orange box) and users car through visualizations recommended by Lux.**

**Steering analysis with intent.** Next, Alice wa
if any country-level characteristics explain th
tive correlation between inequality and life exp
ure 2, she specifies her analysis *intent* to Lux
["AvrgLifeExpectancy","Inequality"]. On print
again, Lux employs the specified analysis intent
mendations towards what Alice might be interes
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

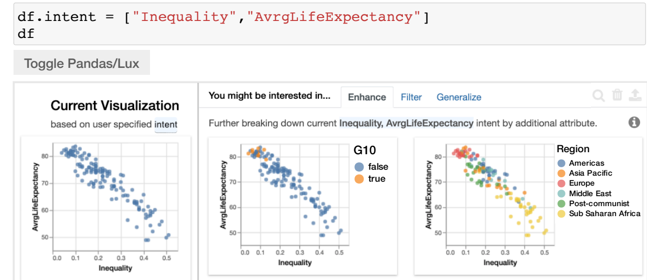



**Figure 2: Alice sets the intent based on the attribute AvrgLifeExpectancy and Inequality, and Lux displays visualizations that are related to the intent.**

**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 [36], 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

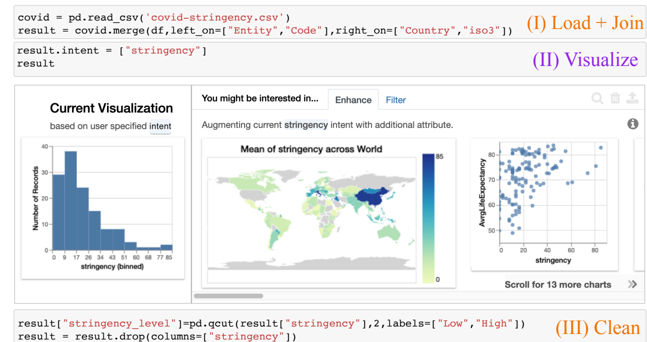
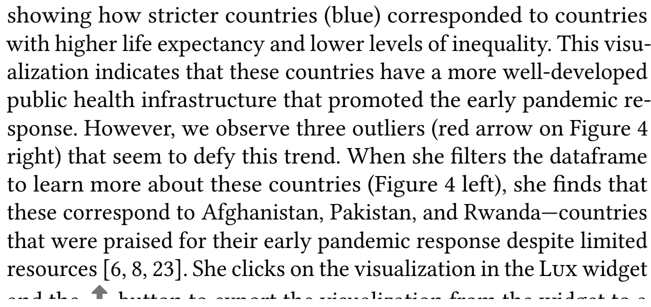



**Figure 3: Tabular operations (orange, steps I & III) to load, clean, and transform the data, while visualizing with Lux (purple, step II).**

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

[1]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)'.

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 [6, 8, 23]. She clicks on the visualization in the Lux widget and the button to export the visualization from the widget to a



```
result[(result["Inequality"]>0.35)&(resu
```

Toggle Pandas/Lux

| | Entity | Code | Day | Country | GeoPoliticalRegion | A |
|---|---|---|---|---|---|---|
| 0 | Afghanistan | AFG | 2020-03-11 | Afghanistan | Middle East and North Africa | |
| 87 | Pakistan | PAK | 2020-03-11 | Pakistan | Asia Pacific | |
| 96 | Rwanda | RWA | 2020-03-11 | Rwanda | Sub Saharan Africa | |

**Figure 4: The scatterplot shows a separ high and low stringency in their COV dataframe (left), we see that Afghan correspond to the three outliers (red l**

Overall, this example demonstrates ization support within a dataframe w between Lux and dataframes enablec data cleaning via a familiar API and r

# 4 INTERACTING WITH I

In this section, we propose a novel alv interaction with dataframes as outlir workflow illustrated the many flexil a dataframe to achieve their analytic these ways and contrast it to existir approaches in dataframe workflows.

As shown in Figure 5, in both existin the always-on framework (b), users c API and see the table view by default tions 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.

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

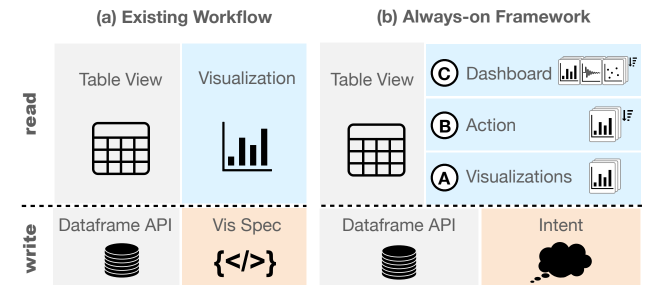



**Figure 5: Conceptual framework for dataframe interaction. Users can make changes to anything below the dotted line (write), elements displayed to the user are shown above the dotted line (read). (a) In existing workflows, users write visualization specification code to create one or more visualizations. (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.

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

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.

*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.

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

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 ?.

$$\langle attribute \rangle \rightarrow \text{attribute} \cup \langle attribute \rangle^* \mid ? \ \langle constraint \quad (4)$$

$$\langle value \rangle \rightarrow \text{value} \cup \langle value \rangle^* \mid ? \quad (5)$$

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 [68] and CompassQL [79].

## 5.2 Specifying Intent

The aforementioned grammar is decoupled from our specific implementation, which uses syntactic sugar for expressing the intent in a convenient Python-based API. 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 `Age`s 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; constaints 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.

## 5.3 Rationale for Design Choices

Due to the heavy cognitive cost of writing glue code to visualize their data [21, 76], users often opt to visualize in the later stages of their workflow [22, 44]. 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 [34]. Lux supports convenient specification shorthands 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

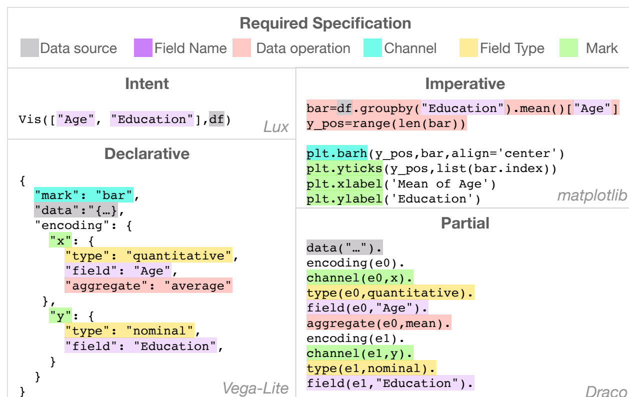



**Figure 6: Comparison between the level of specification required from Lux versus other existing approaches for Query 3.**

## 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, as summarized in Table 1. Metadata- and intent-based ones are akin to those used in past visualization recommendation systems [38, 50, 80]—see Lee et al. [50] 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.

| | | |
|---|---|---|
| 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**

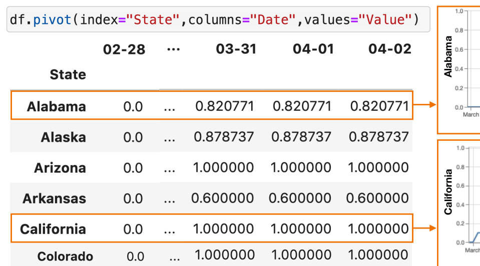
**Metadata-based Recommendations.** Lux maintains dataframe metadata, including attribute-level statistics such as min/max and cardinality to determine the semantic data type of each column and to automatically populate visualization settings. For example, based on data type, Lux can generate univariate 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 [38, 80].

**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.** 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. 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.



```
df.pivot(index="State",columns="Date",values="Value")
```

| State | 02-28 | ··· | 03-31 | 04-01 | 04-02 |
|---|---|---|---|---|---|
| Alabama | 0.0 | ... | 0.820771 | 0.820771 | 0.820771 |
| Alaska | 0.0 | ... | 0.878737 | 0.878737 | 0.878737 |
| Arizona | 0.0 | ... | 1.000000 | 1.000000 | 1.000000 |
| Arkansas | 0.0 | ... | 0.600000 | 0.600000 | 0.600000 |
| California | 0.0 | ... | 1.000000 | 1.000000 | 1.000000 |
| Colorado | 0.0 | ... | 1.000000 | 1.000000 | 1.000000 |



**Figure 7: Row-wise index visualization displaying th 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. 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 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 [84]. 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.

# 7 SYSTEM DESCRIPTION

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. The `ipywidgets` library is used for rendering an interactive HTML widget as the cell output. 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

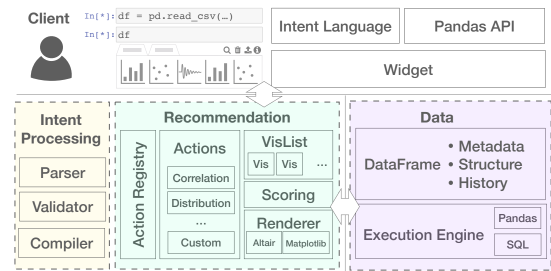




**Figure 8: System architecture for Lux**

## 7.1 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.

### 7.1.1 *Parser and Validator.*

In Section 5, 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.

### 7.1.2 *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 [32, 56].

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).

## 7.2 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 6) 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.

# 8 EXECUTION AND OPTIMIZATION

We now describe Lux's execution engine. We first describe the two major tasks performed by this execution engine. Then, we describe three optimizations aimed at speeding up these tasks.

## 8.1 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 are 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.

## 8.2 Optimization

Next, we describe several optimizations aimed at minimizing the overhead incurred by Lux. 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 [27, 33], early pruning [45, 54, 75], caching and reuse [35, 71], and asynchronous computation [24, 83], 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. Figure 9 shows an example of one such workflow. 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. 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. 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. As shown in cells labeled [3-5] in Figure 9, 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.

Note that while lazy computation and caching and reuse are well-studied in the database literature [35, 71, 82], identifying that lazy computation may be beneficial non-committal dataframe opera-

```
# 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 9: Example workflow demonstrating the applicability of WFLOW optimizations.**

**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 [45, 54, 75], 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 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.**

| 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. 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, $t_{exact}$ and $t_{approx}$, is determined by the relational operations for extracting the required visualization data (as shown in Table 2).

Here, while the use of approximate samples to rank and identify top-k visualizations is not new [54, 75], 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 seconds[2] skimming through the pandas table view before toggling to the Lux view. Leveraging past work on asynchronous query execution [24, 83], 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 [51]. 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 [24, 83], 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, 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 are available online[3].

[2]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.

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

## 9.1 Data and Methodology

**Data:** We use two real-world datasets to evaluate the performance of Lux. The Airbnb dataset [29] contains 12 columns while the Communities [46] 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 [19].

**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 [12] 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 9.4). 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 to the dataframe[4].
- **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.

## 9.2 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.

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

**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 10,11, (Distr.).**

| | 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 | |

**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 10 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.

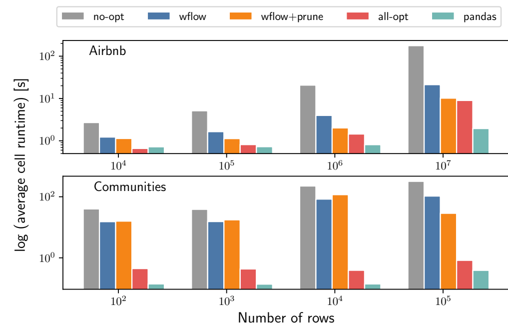



**Figure 10: Average runtime of a notebook cell across the workload for different dataframe size and conditions.**

**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 11 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.

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.

**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 10M Airbnb and over 650X faster for 100k Communities. The performance improvement for non-Lux

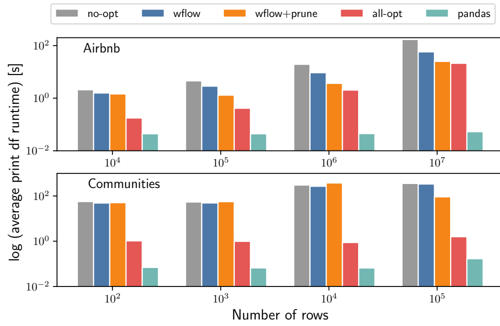



**Figure 11: Average time for printing a single dataframe for different dataframe size and conditions.**

operations demonstrates how WFLOW's lazy evaluation strategy avoids unnecessary computation.

## 9.3 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`[16] 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 12 left shows the runtime for different dataframe widths[5]. We note that the blue no-opt curve (power=2.53) scales exponentially with the number of columns. By applying the PRUNE and ASYNC optimizations (red), LUX effectively lowers the cost of printing a dataframe by bringing the runtime closer to linear (power=1.07).

## 9.4 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 12 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 12 right) enumerates over data subsets, it requires more samples to ensure enough data points per stratum to achieve the same accuracy.

[5] 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.

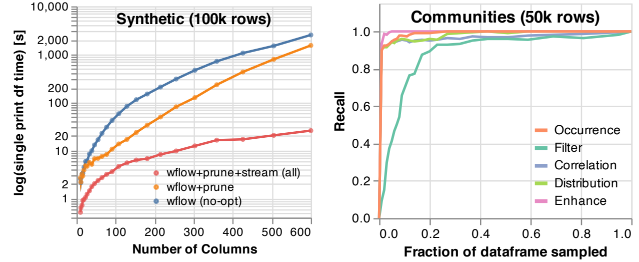



**Figure 12: Left: Time spent for a single dataframe print varying the number of columns in a synthetic dataframe. Right: Recall curve for different actions varying fractional samples of rows in the 50k `Communities` dataset.**

# 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.

## 10.1 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 [5], Happy Planet Index [3], and Olympics [2]). Interactions on the LUX widget and actions performed on the notebook were logged via a custom extension [53]. 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)[6], 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.

## 10.2 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. 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.

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. [21]), 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, 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., 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

[6]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 async 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.**

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:* Table 4 details participants' Likert scale ratings of the functionalities and benefits of Lux. The average System Usability Scale (SUS) [26] 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.

## 10.3 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 [60]. 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", including DatPane [15], Panel [17], and Streamlit [18]. 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 as demonstrated in our documentation and tutorials 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] 120 years of olympic history: athletes and results. 2019.
[3] Happy Planet Index. http://happyplanetindex.org/, 2019.
[4] Tableau. https://www.tableau.com/, 2019. Accessed: 2019-09-11.
[5] Us department of education: College scorecard data. 2019.
[6] Afghanistan: WHO mission reviews COVID-19 response. *World Health Organization*, 2020.
[7] bamboolib. https://bamboolib.8080labs.com/, 2020.
[8] COVID-19 in Pakistan: WHO fighting tirelessly against the odds. *World Health Organization*, 2020.
[9] Faster data exploration in Jupyter through Lux, 2020.
[10] LUX Exploratory Data Analysis (EDA). *YouTube*, Dec 2020.
[11] LUX Library: Matplotlib replacer? *YouTube*, Dec 2020.
[12] papermill 2.3.3 documentation. https://papermill.readthedocs.io/, 2020.
[13] Power BI: Interactive data visualization BI Tools. 2020.
[14] State of Data Science and Machine Learning 2020. *Kaggle*, 2020.
[15] datapane. https://datapane.com/, 2021.
[16] Faker. https://github.com/joke2k/faker, Mar 2021.
[17] Panel. https://panel.holoviz.org/, 2021.
[18] Streamlit. https://streamlit.io/, 2021.
[19] UCI Machine Learning Repository, Mar 2021.
[20] adamerose. PandasGUI. https://github.com/adamerose/pandasgui.
[21] S. Alspaugh, N. Zokaei, A. Liu, C. Jin, and M. A. Hearst. Futzing and moseying: Interviews with professional data analysts on exploration practices. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):22–31, 2019.
[22] 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.
[23] J. Beaubien. Why Rwanda Is Doing Better Than Ohio When It Comes To Controlling COVID-19. *NPR*, Jul 2020.
[24] M. Bendre, T. Wattanawaroon, K. Mack, K. Chang, and A. G. Parameswaran. Anti-freeze 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.
[25] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents.
[26] J. Brooke. *"SUS-A quick and dirty usability scale." Usability evaluation in industry.* CRC Press, June 1996. ISBN: 9780748404605.
[27] 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.
[28] Ç. Demiralp, P. J. Haas, S. Parthasarathy, and T. Pedapati. Foresight: Rapid data exploration through guideposts. *arXiv preprint arXiv:1709.10513*, 2017.
[29] Dgomonov. Data Exploration on NYC Airbnb. *Kaggle*, Aug 2020.
[30] P. Duvva. Speed up EDA With the Intelligent Lux. *Medium*, Mar 2021.
[31] Fbdesignpro. sweetviz. https://github.com/fbdesignpro/sweetviz.
[32] S. Few. *Show Me the Numbers: Designing Tables and Graphs to Enlighten.* Analytics Press, 2012.
[33] 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.
[34] 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.
[35] 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.
[36] 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.
[37] 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.
[38] 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.
[39] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.
[40] P. T. Inc. Collaborative data science, 2015.
[41] 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.
[42] 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.
[43] 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.
[44] 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.
[45] 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.
[46] Kkanda. Analyzing UCI Crime and Communities Dataset. *Kaggle*, Mar 2020.
[47] A. P. Koenzen, 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.
[48] 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.
[49] 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.
[50] 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.
[51] 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.
[52] 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.
[53] lux org. lux-logger. *GitHub*.
[54] S. Macke, Y. Zhang, S. Huang, and A. Parameswaran. Fastmatch: Adaptive algorithms for rapid discovery of relevant histogram visualizations. *PVLDB*, 2017.
[55] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, 1986.
[56] 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.
[57] M. Mannino and A. Abouzied. Expressive Time Series Querying with Hand-Drawn Scale-Free Sketches. pages 1–12, 2018.
[58] 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.
[59] P. Pandey. Intelligent visual data discovery with lux: A python library. *Medium*, Mar 2021.
[60] 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.
[61] A. Rule and U. C. S. Diego. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. 2(November), 2018.
[62] S. Sarawagi. Explaining differences in multidimensional aggregates. *Proceedings of the VLDB Endowment*, pages 42–53, 1999.
[63] S. Sarawagi. User-adaptive exploration of multidimensional data. *Proc of the 26th Intl Conference on Very Large*, pages 307–316, 2000.

[64] 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.

[65] 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.

[66] 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.

[67] sfu-db. dataprep. https://github.com/sfu-db/dataprep.

[68] 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.

[69] 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.

[70] C. Stolte, D. Tang, and P. Hanrahan. Polaris: a system for query, analysis, and visualization of\nmultidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):1–14, 2002.

[71] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Intermittent query processing. *Proc. VLDB Endow.*, 12(11):1427–1441, 2019.

[72] The pandas development team. pandas-dev/pandas: Pandas, Feb. 2020.

[73] 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.

[74] M. Vartak, S. Huang, T. Siddiqui, S. Madden, and A. Parameswaran. Towards visualization recommendation systems. *SIGMOD Records*, 45(4):34–39, May 2017.

[75] 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.

[76] C. Wang, Y. Feng, R. Bodik, A. Cheung, and I. Dillig. Visualization by example. *Proc. ACM Program. Lang.*, 4(POPL), Dec. 2019.

[77] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis.* Springer-Verlag New York, 2016.

[78] C. Y. Wijaya. Quick recommendation-based data exploration with lux. *Medium*, Feb 2021.

[79] 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.

[80] 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.

[81] E. Wu and S. Madden. Scorpion: Explaining Away Outliers in Aggregate Queries. *Proceedings of the VLDB Endowment*, 6(8):553–564, 2013.

[82] 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.

[83] 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.

[84] C. Yan and Y. He. Auto-suggest: Learning-to-recommend data preparation steps using data science notebooks. In *International Conference on Management of Data (SIGMOD)*, June 2020.