

Accelerating Visual Data Exploration via Sampling: A Case Study with Lux

Kunal Agarwal



Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-80

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-80.html>

May 12, 2022

Copyright © 2022, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**Accelerating Visual Data Exploration via Sampling: A Case Study with
Lux**
by Kunal Agarwal

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,
University of California at Berkeley, in partial satisfaction of the requirements for the
degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

Committee:

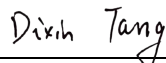


Professor Aditya Parameswaran
Research Advisor

05/11/2022

(Date)

* * * * *



Dixin Tang
Second Reader

05/11/2022

(Date)

Accelerating Visual Data Exploration via Sampling: A Case Study with Lux

by

Kunal Agarwal

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Aditya Parameswaran, Chair

Spring 2022

Accelerating Visual Data Exploration via Sampling: A Case Study with Lux

Copyright 2022
by
Kunal Agarwal

Abstract

Accelerating Visual Data Exploration via Sampling: A Case Study with Lux

by

Kunal Agarwal

Masters of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Aditya Parameswaran, Chair

Exploratory Data Analysis (EDA) is a necessary and vital part of data science that usually occurs in computational notebooks with tools such as pandas. One of the most popular tools for EDA is *Lux* which visualizes data stored in pandas DataFrames in a dashboard displayed in a Jupyter Notebook. However, as datasets become larger in size, the computation necessary to compute these visualizations becomes larger as well, slowing down *Lux*. We consider the use of sampling to accelerate the computation required for generating visualizations. We analyzed how *Lux* performs on large datasets and determined what parts of *Lux* could be accelerated using data sampling. We then integrate our sampling method into *Lux* and demonstrate a significant speedup while not compromising the quality of the visualizations produced by *Lux*.

To my sister, Anya

Contents

Contents	ii
List of Figures	iii
List of Tables	iv
1 Introduction	1
2 Related Work	6
3 Benchmarking <i>Lux</i> on Large Datasets	7
3.1 General Scalability Benchmarking	7
4 New Sampling Method	13
4.1 Sampling In <i>Lux</i>	13
4.2 Sampling Configuration	16
4.3 New Sampling Method	16
5 Benchmarking the New Sampling Strategy	19
5.1 Benchmarking Old vs New Sampling Strategy	19
5.2 Fidelity of Visualizations Generated	26
6 Incorporating Sampling Beyond <i>Lux</i>	32
6.1 Design Principles for Incorporating Sampling	32
6.2 Applications of Sampling	34
7 Conclusion	35
Bibliography	36

List of Figures

1.1	Example of <i>Lux</i> widget display	2
1.2	Diagram of <i>Lux</i> architecture simplified	3
3.1	Experiment 1: Total Time	9
3.2	Experiment 1: Computation Time Breakdown	9
3.3	Experiment 2: Total Time	11
3.4	Experiment 2: Computation Time Breakdown	11
4.1	Diagram of <i>Lux</i> architecture utilizing old sampling method	14
4.2	Example of sampling warning in the <i>Lux</i> user interface	15
4.3	Example of sampling configurations in practice	16
4.4	Diagram of <i>Lux</i> architecture utilizing the new sampling method	18
5.1	Old vs New Sampling Strategy on Categorical Data: Total Time	20
5.2	Old vs New Sampling Strategy on Categorical Data: Metadata Computation Time	20
5.3	Old vs New Sampling Strategy on Categorical Data: Execute Time	21
5.4	Old vs New Sampling Strategy on Categorical Data: <code>create_vis</code> Time	21
5.5	Old vs New Sampling Strategy on Numerical Data: Total Time	24
5.6	Old vs New Sampling Strategy on Numerical Data: Metadata Computation Time	24
5.7	Old vs New Sampling Strategy on Numerical Data: Execute Time	25
5.8	Old vs New Sampling Strategy on Numerical Data: <code>create_vis</code> Time	25
5.9	Correlation Visualizations for Airbnb Dataset using the old sampling method . .	27
5.10	Occurrence Visualizations for Airbnb Dataset using the old sampling method . .	27
5.11	Correlation Visualizations for Airbnb Dataset using the new sampling method .	28
5.12	Occurrence Visualizations for Airbnb Dataset using the new sampling method .	28
5.13	Correlation Visualizations for Communities Dataset using the old sampling method	29
5.14	Occurrence Visualizations for Communities Dataset using the old sampling method	29
5.15	Correlation Visualizations for Communities Dataset using the new sampling method	30
5.16	Occurrence Visualizations for Communities Dataset using the new sampling method	30

List of Tables

3.1	Subset of full dataset for Figures 3.1 and 3.2. All time is in seconds	8
3.2	Subset of full dataset for Figures 3.3 and 3.4. All time is in seconds	10
5.1	Subset of full dataset for the old sampling method for Figures 5.1 through 5.4. All time is in seconds.	22
5.2	Subset of full dataset for the new sampling method for Figures 5.1 through 5.4. All time is in seconds.	22
5.3	Subset of full dataset for the old sampling method for Figures 5.5 through 5.8. All time is in seconds.	23
5.4	Subset of full dataset for the new sampling method for Figures 5.5 through 5.8. All time is in seconds.	26

Acknowledgments

Thank you to everyone listed below for your help and support.

- Doris Lee, thank you for introducing me to the research process and answering all my questions over the past two years. It has been a great experience working on *Lux* with you!
- Aditya Parameswaran, thank you for advising me on my research project throughout the past year. From outlining the big picture of the project to helping debug small edge cases, you've been immensely helpful and I've learned so much.
- Dixin Tang, thanks for providing valuable insight and tips in our weekly meetings and for helping edit and refine the thesis.
- The Lux team, thank you for your help in answering my questions and assisting me on implementing new features. I appreciate everyone's help throughout this process.
- All my friends, thank you for making this last year memorable. Your support and encouragement has helped me get to this point, and I couldn't have done it without all of your help.

Chapter 1

Introduction

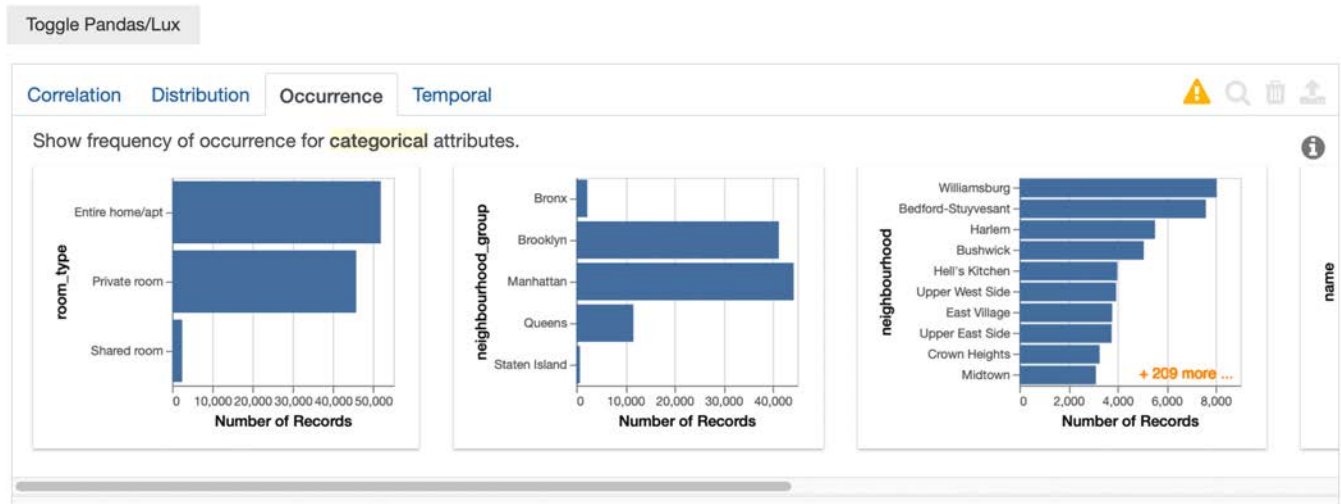
It is often difficult and cumbersome for data scientists to draw meaningful insights from large datasets. They typically explore the dataset via *pandas* [16], a popular data science library for manipulating tabular data. This exploration is usually done in a computational notebook, a popular one being Jupyter [7]. This exploration process, called Exploratory Data Analysis (EDA), is done by repeatedly issuing `pandas` commands to transform the data and visualizing the result. In conjunction with `pandas`, many different visualization libraries like `matplotlib` [5] and `altair` [17], are used to make sense of data visually to gather insights and determine subsequent steps.

Unfortunately, these libraries can be complicated and cumbersome to use, with many different steps and lines of code needed to achieve the desired results. For example, creating visualizations in a library like `matplotlib` takes many lines of code, and can be complicated to manipulate if a small change to the visualization is necessary to continue with EDA. *Lux* tries to solve these issues by streamlining much of the EDA process, particularly in visualizing datasets in a simple way.

What is *Lux*?

Lux is an always-on visualization recommendation system built as a wrapper over `pandas`, so it retains much of the same functionality that `pandas` offers [10]. *Lux* both uses `pandas` metadata and also generates its own metadata to intelligently generate visualizations that describe potentially interesting and relevant trends and correlations within the data. *Lux* generates these visualizations on the fly and automatically whenever a `pandas` DataFrame is printed, helping to narrow the scope of the EDA that the data scientist must conduct. These visualizations are populated as a widget shown to the user within a Jupyter notebook.

Lux displays many different types of visualizations, from univariate to multivariate visualizations. It displays bar graphs, histograms, scatter plots, heat maps, geographical maps, line plots, and time series plots. Each of these visualizations are categorized into different

Figure 1.1: Example of *Lux* widget display

tabs on the widget, each representing a different analytical action [9]. For example, for a Correlation action, we may see a bivariate visualization like a scatter plot displaying the correlation between two features in the dataset. For a Distribution action, we may see a univariate visualization like a histogram displaying the distribution of a particular attribute. There are a few default actions that *Lux* provides, but it is simple for users to design their own actions as well. We see an example of the display widget that *Lux* generates in Figure 1.1.

The computation necessary to generate these different visualizations using the data is quite complex and can be separated into a three key components. The first is **computing metadata**. Once *Lux* is given a dataset, it first computes a variety of statistics and metadata to understand the structure of the data to inform future decisions it needs to make. Some examples include calculating the cardinality, or the data type of each column. Many of these calculations are expensive and grow as the size of the dataset increases. The second step is the **execute function**. The execution engine retrieves all the metadata calculated in the previous step and uses it to prepare the data for visualization generation. It filters the data as necessary, and then performs different operations depending on the visualization type generated. For example, the execution engine will aggregate the data in preparation for a bar plot, or bin the data in preparation for a histogram. The metadata will come in handy for these operations. However, the larger the dataset used, the slower it will be to perform this computation to prepare the data for the visualizations. Finally, the third step is **creating the visualizations**. This involves using the prepared data from the execution engine and building the visualizations in the selected visualization engine (eg. `altair` or

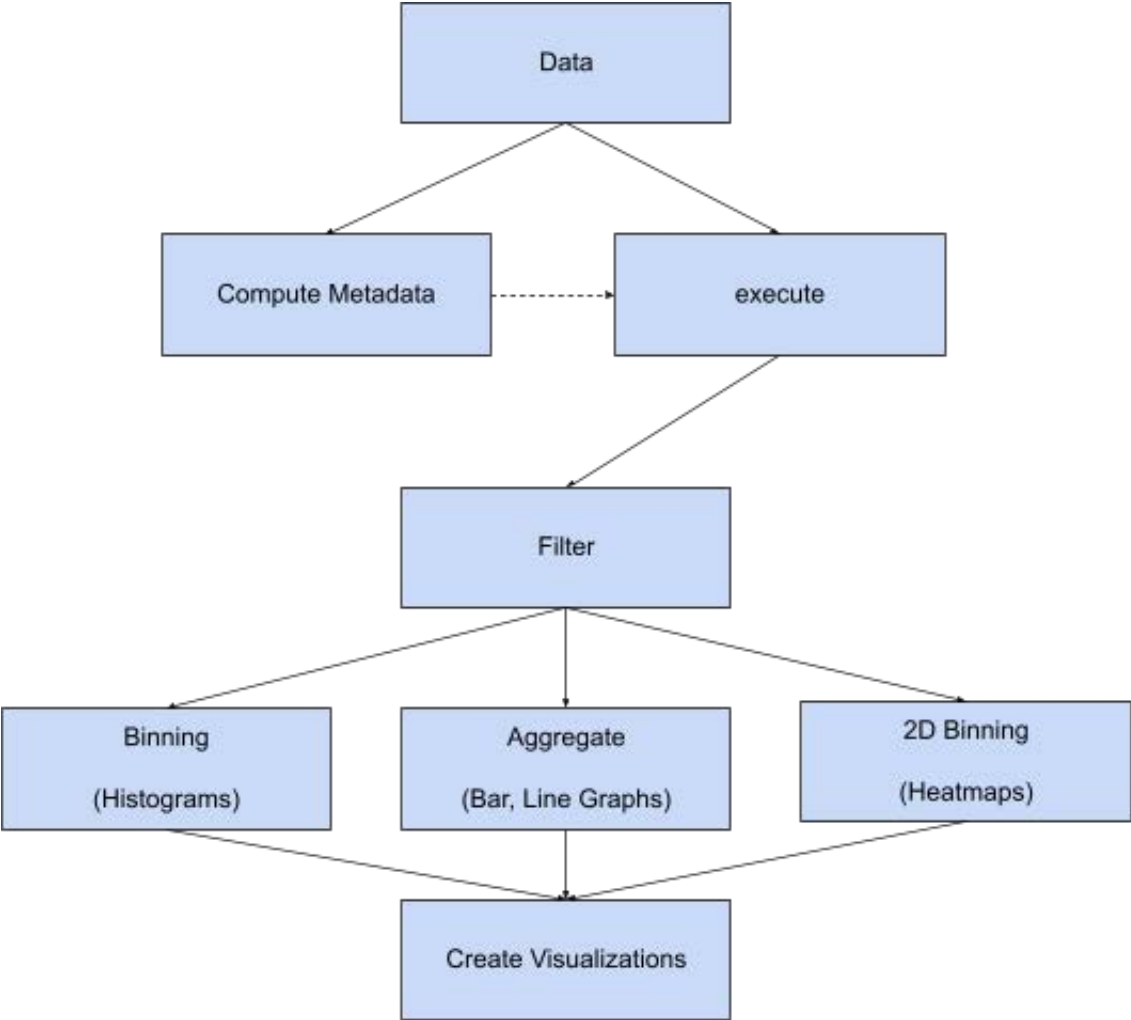


Figure 1.2: Diagram of *Lux* architecture simplified

`matplotlib`). The visualizations are built and sent to the front-end to be displayed to the user. The more data that is used for the visualizations, the longer it will take to generate, especially for visualizations like scatter plots. [10]. The workflow described can be seen in Figure 1.2.

For large datasets, to reduce latencies, *Lux* samples datasets that are over a certain size. Visualizations are then generated based on this smaller sample. However, this approach still has problems, which will be evaluated throughout this report as we propose a new approach to sampling that improves upon the current one.

Research Questions and Contributions

Visualization libraries, like *Lux*, work well on smaller datasets, but at scale they struggle to provide interactive results. Visualizations with thousands to millions of data points tend to take a long time to be generated, defeating the purpose of an ‘always on’ visualization software like *Lux*. To help *Lux* scale, we need to answer the following research questions:

How does Lux scale when run on large datasets, and where are most of the bottlenecks located? *Lux*, like most visualization libraries, becomes slower when run on large datasets. Our objective is to determine where the bottlenecks in the computation are. Once we are able to identify these areas for improvement, we can determine if sampling or other optimizations would be appropriate.

How can sampling be used effectively in generating visualizations on large datasets? The data *Lux* receives is used in a variety of ways from the computation of metadata to the calculations surrounding potential interestingness of visualization recommendations to sending data to the front-end. These are all integral aspects of the *Lux* pipeline and take longer to complete on larger datasets. Our goal is to determine which areas would benefit from adding sampling.

How do new sampling strategies provide faster functionality without compromising on the accuracy and quality of the generated visualizations? While sampling can allow *Lux* and other visualization libraries to run on a small subset of the data, this may compromise the quality of the visualizations in that they may not reflect true relationships in the data. It is important to note if the additional sampling creates inaccurate representations of the data, which would make the new sampling strategy pointless.

How does our experience in using sampling to optimize an EDA tool like Lux provide lessons for other EDA tools? There are many EDA tools beyond *Lux* that also suffer similar constraints around large datasets. We provide a framework based on our experience that can guide others in adding sampling to their EDA tools without compromising the utility of

the tool itself.

Our contributions are as follows:

- We survey previous work in the area of generating approximate visualizations from sampled data. (Chapter 2)
- We analyze how *Lux* does on large datasets, and where primary bottlenecks lie. (Chapter 3)
- We build a new method of sampling with *Lux* that encourages sampling of important metadata used in the computation of visualizations to speed up computation. (Chapter 4)
- We analyze how the new method affects the speed of computing the visualizations as well as ensuring whether the accuracy and fidelity of the visualizations generated were up to par. (Chapter 5)
- We surveyed other visualization methods and their sampling strategies to create a framework for incorporating sampling in an EDA system in general. (Chapter 6)

Chapter 2

Related Work

The field of Approximate Query Processing (AQP), targets returning approximate results by using samples collected either offline or online [4]. Some recent work focuses on giving users the ability to control the approximation for their query. There isn't a perfect solution to AQP, but the user should be able to decide for themselves the tradeoff between speed and accuracy of the query [1]. For example, with the use of sampling to provide AQP, the users can specify exactly the method of sampling to be used that best fits their needs.

Narrowing down the scope of use of AQP, we focus on approximating queries that generate visualizations. One example use of AQP is in *SeeDB* [14], which partially automates the task of finding visualizations that shows high deviation for a given query. *SeeDB* samples the query output data as well as the original data to speedup the overall workflow and also uses sampling to prune uninteresting visualizations.

Sampling can help to speed up query processing, but allowing the user to understand the potential loss of accuracy in visualization is just as important. One example of an AQP that visualizes the confidence intervals that result from sampling is *Pangloss* [12]. *Pangloss* allows users to understand the difference in visualization results between the sampled and non-sampled versions.

While *Pangloss* and *SeeDB* use sampling to generate visualizations, there are other uses of sampling for the purposes of EDA. Another example of previous work in utilizing sampling to improve visualization recommendation is *FastMatch* [11]. *FastMatch* uses adaptive sampling methods to retrieve histogram visualizations from a large set of possible candidate histograms that are closest to a user specified target distribution. As we discuss in this thesis, *Lux* also employs sampling as part of its visualization recommendation system.

Chapter 3

Benchmarking *Lux* on Large Datasets

In this chapter, we describe the processes we use to benchmark *Lux* across different experiments measuring various scalability-centric metrics and discuss insights gained from these experiments. Throughout this process, we gained valuable knowledge on where certain bottlenecks may exist within the current framework that are amplified on larger datasets.

All experiments run in this chapter and throughout this paper were run on a 2021 MacBook Pro with an Apple M1 Pro chip and 32GB of memory and 10 cores.

Data

We use two real world datasets for our experiments. The first is an Airbnb dataset, which contains many different rental listings on the popular app Airbnb [2]. The dataset contains 16 columns and about 49,000 rows. For this dataset we duplicated the data to create a dataset that contains 10 million data points that we can use for experiments. The second dataset used is the Communities dataset, which contains information about the community surrounding UC Irvine [6]. This dataset contains 128 columns and about 2,000 rows. Again, for this dataset we duplicated the data to create a dataset that contains 1 million data points for experiments.

3.1 General Scalability Benchmarking

The first experiment we ran was to measure how different parts of *Lux* scaled as the number of data points grew. The computation underlying *Lux* can be divided into a number of different components:

- `t_meta`: time for computing the metadata

Num Pts	t_meta	t_recs	t_render	t_1st_print	t_2nd_print	t_total
50000	0.31	2.08	1.05	0.004	0.004	3.448
86975	0.50	0.37	0.79	0.004	0.004	1.668
151293	0.87	0.49	0.63	0.004	0.004	1.998
263175	1.61	0.68	0.81	0.004	0.004	3.108
457795	2.93	1.16	0.94	0.004	0.004	5.038
796336	5.13	1.60	1.26	0.004	0.004	7.998
1385231	8.29	2.80	1.24	0.006	0.005	12.341
2409614	15.96	4.97	2.43	0.004	0.004	23.368
4191534	27.36	7.75	3.00	0.004	0.004	38.118
7291189	46.12	12.64	3.71	0.004	0.004	62.478

Table 3.1: Subset of full dataset for Figures 3.1 and 3.2. All time is in seconds

- `t_recs`: time for computing and generating the visualization recommendations
- `t_render`: time for renderer (e.g., `matplotlib`, `altair`) to build the visualization in the backend

`t_meta` represents the time it takes to compute metadata, which includes calculating important information about the dataset like the cardinality of each column, the unique values in each column, and more. It also uses this information to determine what type of data each column represents (e.g, temporal, numerical, or geographic). `t_recs` is the time it takes to compute and generate the visualization recommendations. This involves generating many different types of visualizations and calculating an interestingness score for each one to determine which ones should be displayed to the user [10]. Finally, `t_render` is the amount of time it takes for the renderer (e.g, `matplotlib` or `altair`) to build the actual visualization in the backend so that it can be sent to the front end in the notebook. `t_render` provides a good representation of the runtime on the front-end, since after the visualizations are rendered, it takes very little time to print the visualizations in the front end. We’ve tracked this time to print as `t_1st_print`. We print the widget twice to ensure there aren’t any discrepancies between the two prints due to caching optimizations within *Lux*. The time it takes to print a second time is tracked as `t_2nd_print`.

Experiment 1: Airbnb Dataset

The first experiment we run is in calculating the above metrics on increasing samples of a dataset on a logarithmic scale. Since *Lux* natively uses sampling in its computation as mentioned in Chapter 1, we turn off any sampling that *Lux* does in order to truly test how *Lux* scales in computation on large datasets. The number of points in each run is selected from between 5,000 data points and 10 million data points where each run represents a value

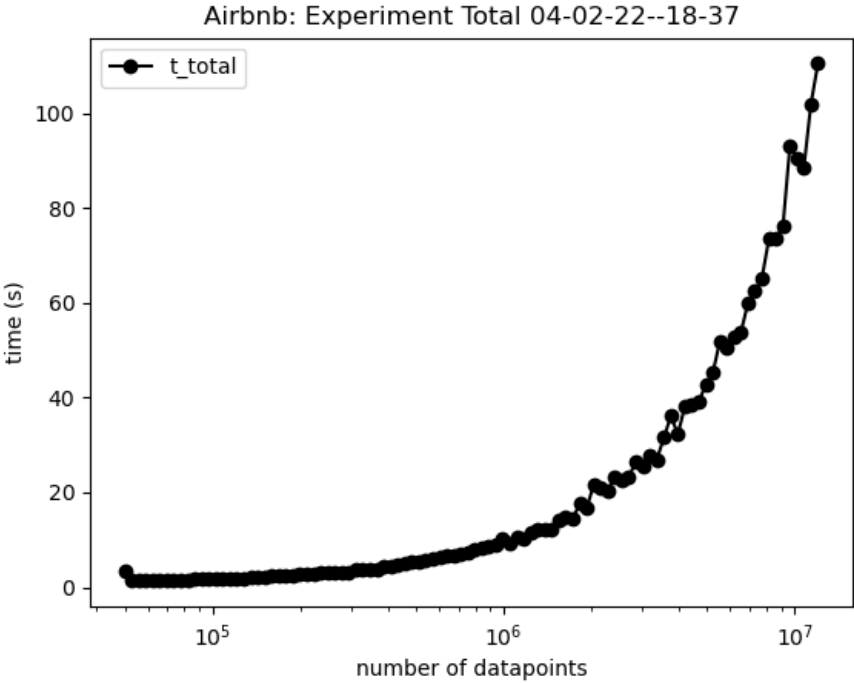


Figure 3.1: Experiment 1: Total Time

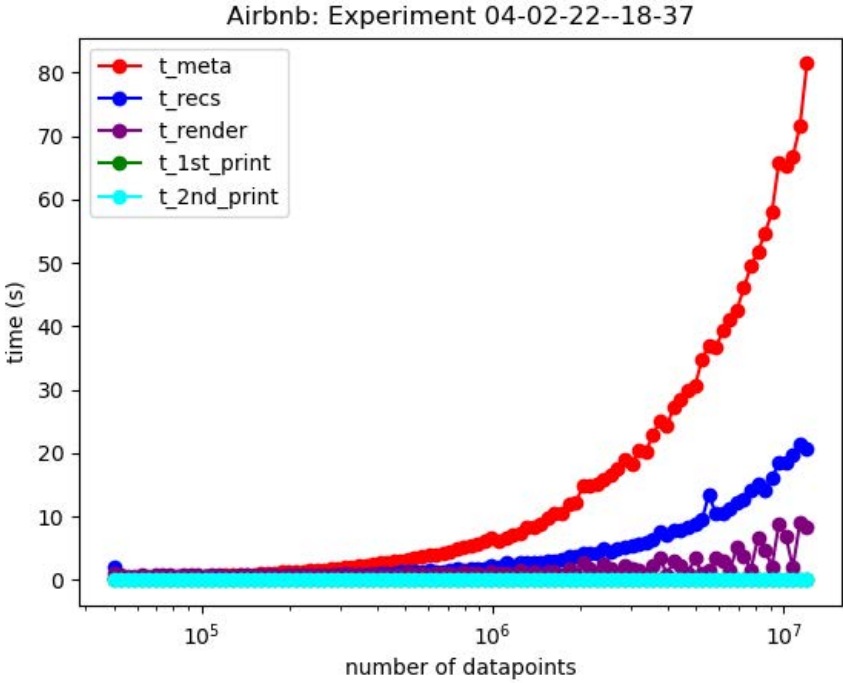


Figure 3.2: Experiment 1: Computation Time Breakdown

Num Pts	t_meta	t_recs	t_render	t_1st_print	t_2nd_print	t_total
5000	0.08	39.08	3.01	0.026	0.025	42.221
8538	0.09	41.44	3.10	0.028	0.028	44.686
14582	0.15	45.92	3.35	0.034	0.033	49.487
24903	0.19	49.71	3.95	0.040	0.039	53.929
42528	0.29	56.30	4.18	0.067	0.052	60.889
72628	0.445	46.32	5.88	0.088	0.068	52.801
124031	0.73	55.89	7.79	0.146	0.102	64.658
211816	1.39	70.65	11.41	0.420	0.154	84.02
361732	2.14	88.18	17.54	0.614	0.230	108.704
617753	3.50	171.56	31.25	1.988	0.415	208.713

Table 3.2: Subset of full dataset for Figures 3.3 and 3.4. All time is in seconds

equidistant to each other on a logarithmic scale. Figure 3.1 shows the total time it takes to fully run *Lux* on the dataset provided. We notice that as the number of datapoints in our dataset increases, represented by the x-axis, the time in seconds it takes to fully run *Lux* on the dataset, represented by the y-axis, increases dramatically. In Figure 3.2, we see the data used in Figure 3.1 broken down into the components described earlier. `t_meta` takes the longest amount of time and grows the fastest with the size of the dataset in comparison to the other metrics. The detailed data for these figures can be found in Table 3.1.

Overall, we notice that the time it takes to calculate the metadata is the most compared to the other metrics, and that difference widens as the size of the dataset increases. This is clearly where much of the slowdown comes from. When looking at the `t_recs`, we see that there is also a noticeable increase in the computation time as the dataset size increases, but not as much. The rendering time and the printing time stay largely constant.

Experiment 2: Communities Dataset

We run the same experiment on the Communities dataset. This dataset has a large number of columns, so we're interested to see how the benchmarking test changes with the differing structure of this dataset. As in the previous experiment, we switch off any sampling that *Lux* does to test how *Lux* scales in computation on large datasets. The number of points in each run is selected from between 5,000 data points and 1 million data points where each run represents a value equidistant to each other on a logarithmic scale. Figure 3.3 shows the total time it takes to fully run *Lux* on the dataset provided. We notice that as the number of datapoints in our dataset increases, represented by the x-axis, the time in seconds it takes to fully run *Lux* on the dataset, represented by the y-axis, increases dramatically. In Figure

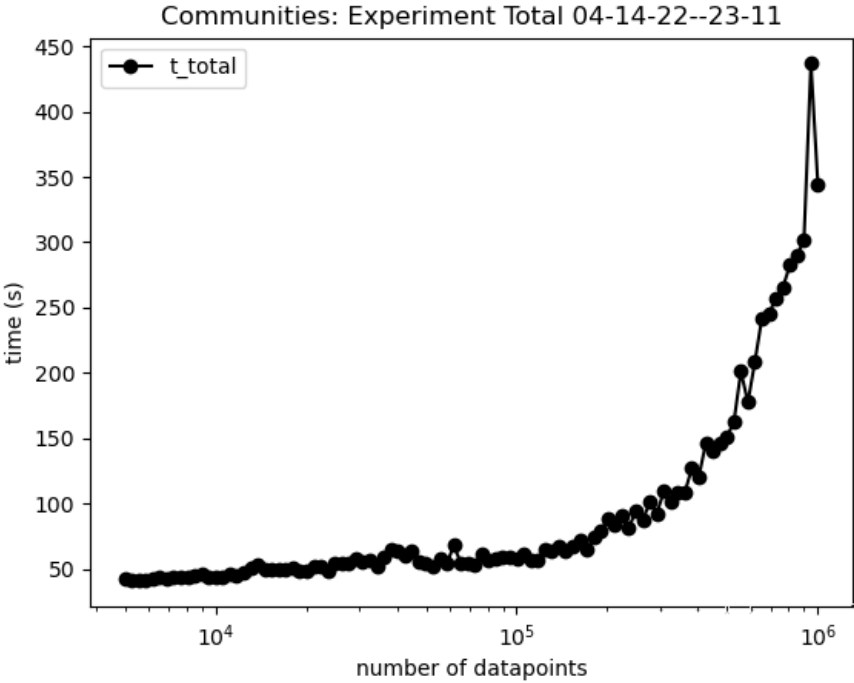


Figure 3.3: Experiment 2: Total Time

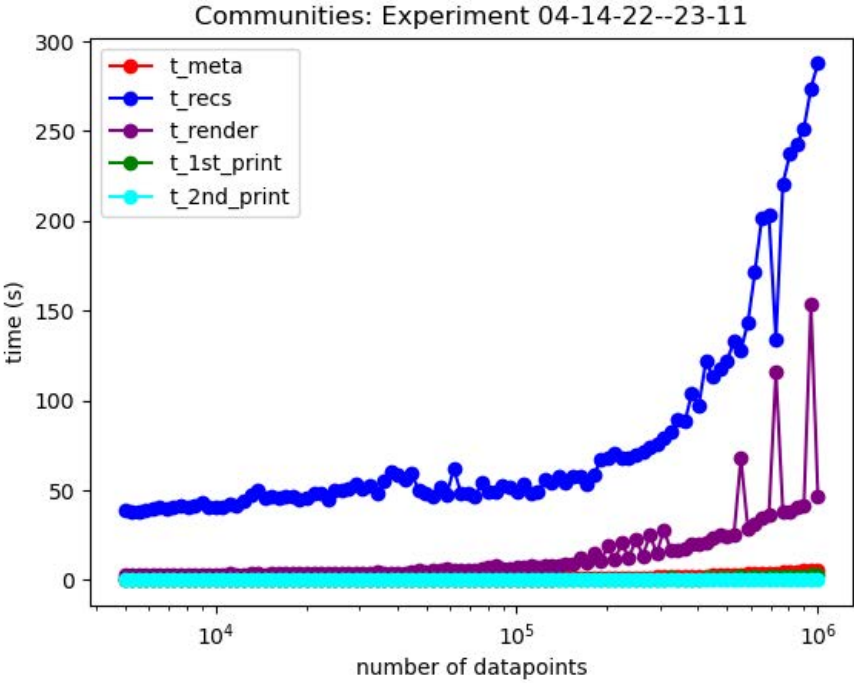


Figure 3.4: Experiment 2: Computation Time Breakdown

3.4, we see the data used in Figure 3.3 broken down into the components described earlier. `t_recs` takes the longest amount of time and grows the fastest with the size of the dataset in comparison to the other metrics. Any spikes seen in the experiment are due to noise in running the experiments. The detailed data for these figures can be found in Table 3.2.

Since the Communities dataset has a large number of columns, the time it takes to create visualizations (`t_recs`) is much larger than the other measured areas of *Lux*, as we see in Figure 3.4. There are a lot more potential visualizations under consideration for recommendation, and thus there is more computation required. In general, just like the Airbnb dataset, the computation time increases as the size of the dataset increases, as we see in Figure 3.3. The rendering time also increases with the size of the dataset, but the metadata computation time and the printing time are miniscule compared to the other parts of *Lux*, and stay relatively the same as the size of the dataset increases.

Key Takeaways: These experiments show us how the size of the dataset can massively impact the speed at which *Lux* can properly function. We also note the specific areas of *Lux* that capture most of the computation time. Those are the areas we want to focus on in our efforts to scale *Lux* for larger datasets. We also note the major differences between the Airbnb and Communities benchmarking: for the Airbnb dataset, with fewer columns, most of the time is taken calculating the metadata and for the Communities dataset, with many columns, the metadata computation time is small compared to the time taken to generate the recommendations. We aim to use sampling to solve both of these issues.

Chapter 4

New Sampling Method

In this chapter, we discuss how sampling works currently in *Lux*. We also introduce a new way to further optimize *Lux* by using additional sampling throughout *Lux*.

4.1 Sampling In *Lux*

We take a detailed look at the system architecture of *Lux* in Figure 4.1 and how sampling plays a role in each component.

Computing Metadata

First, the metadata is computed on the full dataset. As we saw in our benchmarking on the Airbnb dataset, the computation time for the metadata increased along with the size of the data. This is due to the fact that the calculations grow in complexity with the size of the dataset, such as calculating the cardinality of the dataset, or inspecting the `dtype` of each value in a column to find the correct data type for further calculations.

Execution Engine (`execute`)

The `execute` function is also called on the data, but makes use of the metadata calculated in the previous step to grab all the relevant data and metadata needed for the creation of a visualization. If the data is large, then *Lux* will collect only a sample of the full dataset for the visualization. Since the sample is not large, generating the visualization and sending the data to the front-end doesn't require a large amount of time which speeds up the visualization process. It's important to note that the actual DataFrame is never edited in any way, it's just the data used in the visualizations would be a sample of the full dataset.

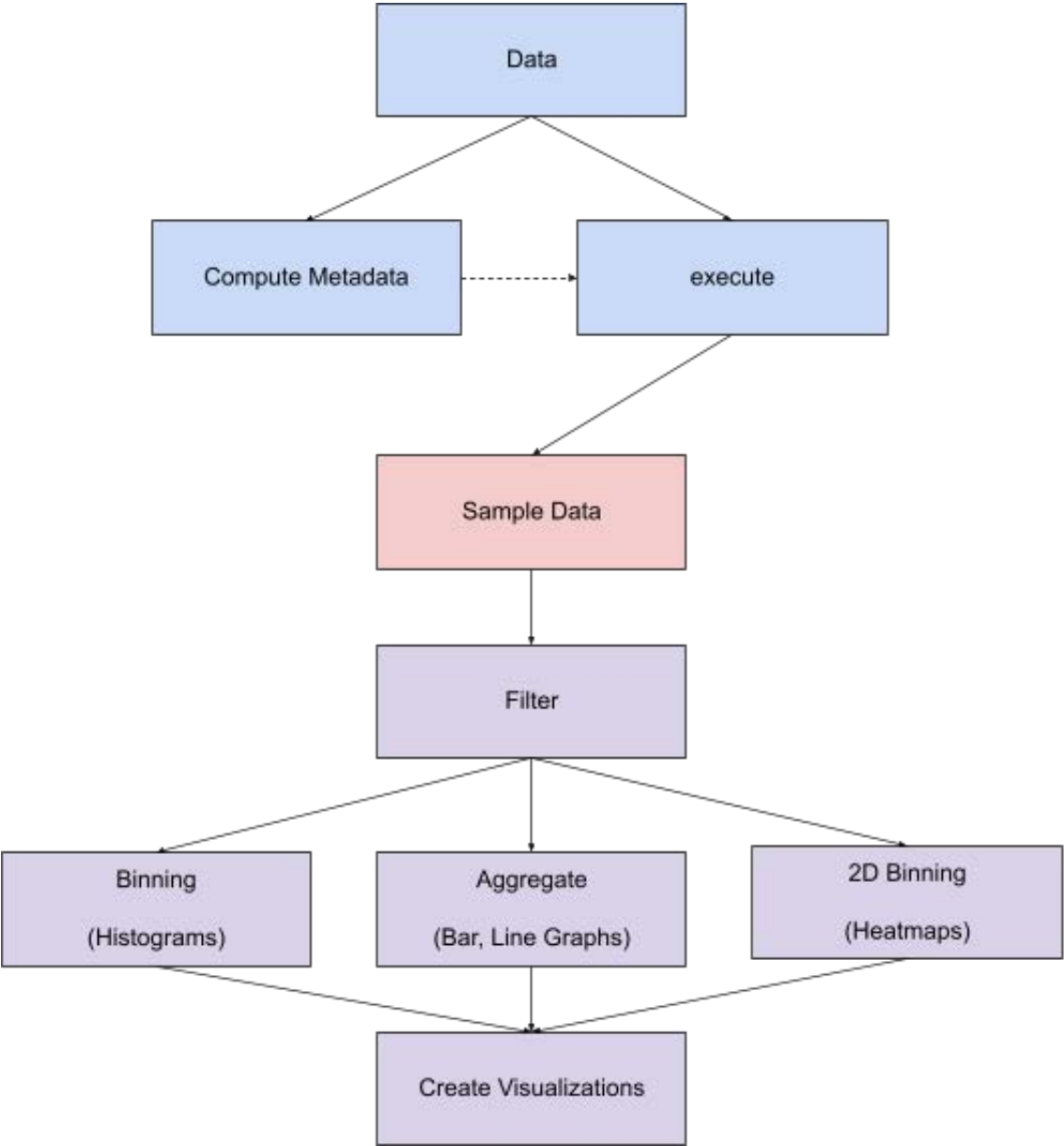


Figure 4.1: Diagram of *Lux* architecture utilizing old sampling method

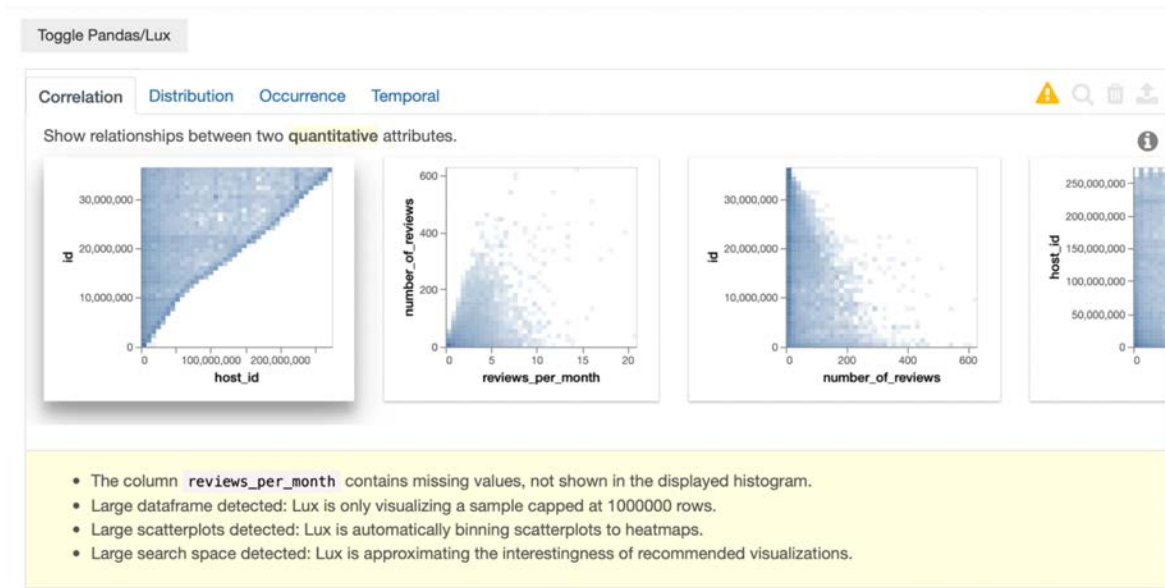


Figure 4.2: Example of sampling warning in the *Lux* user interface

The `execute` component will generate many different types of visualizations that are the result of aggregating and binning the data in various ways, and to generate the more interesting or salient visualization that best describes the data, it must use the metadata computed previously.

Create Visualizations (`create_vis`)

Finally, once the `execute` component has determined which visualizations to generate and has collected the necessary data for each visualization, the visualization will finally be generated and sent to the front-end. If the data is too large and needs to be sampled, then there will be a flag letting the user know that the visualizations are based on sampled data instead of the full dataset, as seen in Figure 4.2.

Key Takeaway: The current system architecture of *Lux* generates visualizations on sampled data. This sampling currently in the architecture helps address the issue seen when benchmarking the Communities dataset, where the recommendation generation takes up a majority of the computation time. However, for datasets like the Airbnb dataset, a large amount of computation in the form of metadata computation is necessary to generate these visualizations. This forms a bottleneck in the architecture. Adding sampling to metadata computation as well would speed up *Lux* as a whole. This is bolstered by the benchmark on the Airbnb dataset where we notice a significant portion of the total compute time is being

```
lux.config.sampling_thresh = 100000|
lux.config.sampling = False
lux.config.heatmap = False
pd.read_csv('data/airbnb_1000000.csv')
```

Figure 4.3: Example of sampling configurations in practice

taken to calculate the metadata, which grows with the size of the dataset. By calculating the metadata based on a sampled of the data, we can see a significant speedup.

4.2 Sampling Configuration

As we consider using sampling to cut down computation on large datasets, the question arises as to what we consider to be a dataset large enough for sampling to be needed. We allow the user to determine this using user-settable configuration parameters. The user can set a certain number of rows that they deem to indicate a ‘large’ DataFrame, and *Lux* will incorporate sampling for any DataFrames larger than this threshold. That is, sampling is not used if the dataset is smaller than this value. This configuration places the decisions in the hands of the user regarding whether they want to wait longer for more accurate visualizations, or if they’re satisfied with approximate visualizations that are available immediately. There are other user configuration parameters that can be used to speed up or slow down *Lux* in exchange for accuracy. The user may completely switch off sampling, or disallow the *Lux* from using heatmaps as a replacement for scatter plots for large DataFrames. The user can also set the size of the heatmap bins.

We see an example of these configurations in use in Figure 4.3. By default, *Lux* keeps the sampling threshold at 100,000 rows with sampling switched on. Heatmaps are also used by default for datasets with more than 5,000 rows. *Lux* defaults to a bin size of 40 x 40 for the heatmaps. Future versions of *Lux* may allow users to control bin size as well.

Key Takeaway: While it is important to be able to scale *Lux* to larger datasets, it is also important to understand the need for flexibility for the user. Ultimately, the user decides the best use of *Lux* for their purposes, and so it is important to provide this flexibility to the user to let them change how the different optimizations and sampling strategies are used.

4.3 New Sampling Method

As we saw with the current sampling strategy for *Lux* in Section 4.1 and in the previous benchmarking in Chapter 3, there is a bottleneck in the *Lux* architecture. The computation

done by *Lux* to compute the metadata for the dataset is necessary for any future computation and visualization generation, but is done on the full dataset. This slows down the overall computation time for *Lux* for larger datasets. We propose computing metadata over a sample of the dataset, sampled without replacement, rather than the full dataset to speed up computation across the board.

New Changes

Originally, *Lux* would sample the data before pulling the data for the different visualizations it wanted to generate. Now, this sampling is pushed back to before the metadata is computed. We use the same sampling method *Lux* currently uses, which samples a specified number of rows without replacement from the dataset.

As we see in the diagram in Figure 4.4, the sampling is pushed to the beginning of all computation within *Lux*, so most of the computation is done on sampled data. If you compare this diagram to Figure 4.1, we see only a portion of the total computation is operated on sampled data.

The sampled data is now used for all of the computation of the metadata used throughout *Lux* including calculating the cardinality of each column, finding the unique values in each column, and other important features about the data used in generating the visualizations. The sampled data is then saved and used later for generating visualizations in the `execute` function.

Future Work

While this implementation is a major improvement to *Lux*'s efficiency, there are still many other improvements that can be made.

One idea would be to lazily render the visualizations in the front end. A large portion of the computation time is spent sending the data to the front end. If the visualizations could be loaded in as they are generated, this can provide the best visualizations right away while less optimal options are loaded in only a few seconds later.

Another option would be to utilize parallel computation to analyze parts of the dataset and generate visualizations all in parallel. Incorporating libraries like `modin` [15] or `ray` [13] to parallelize some of the tasks that are done would provide a tremendous speedup.

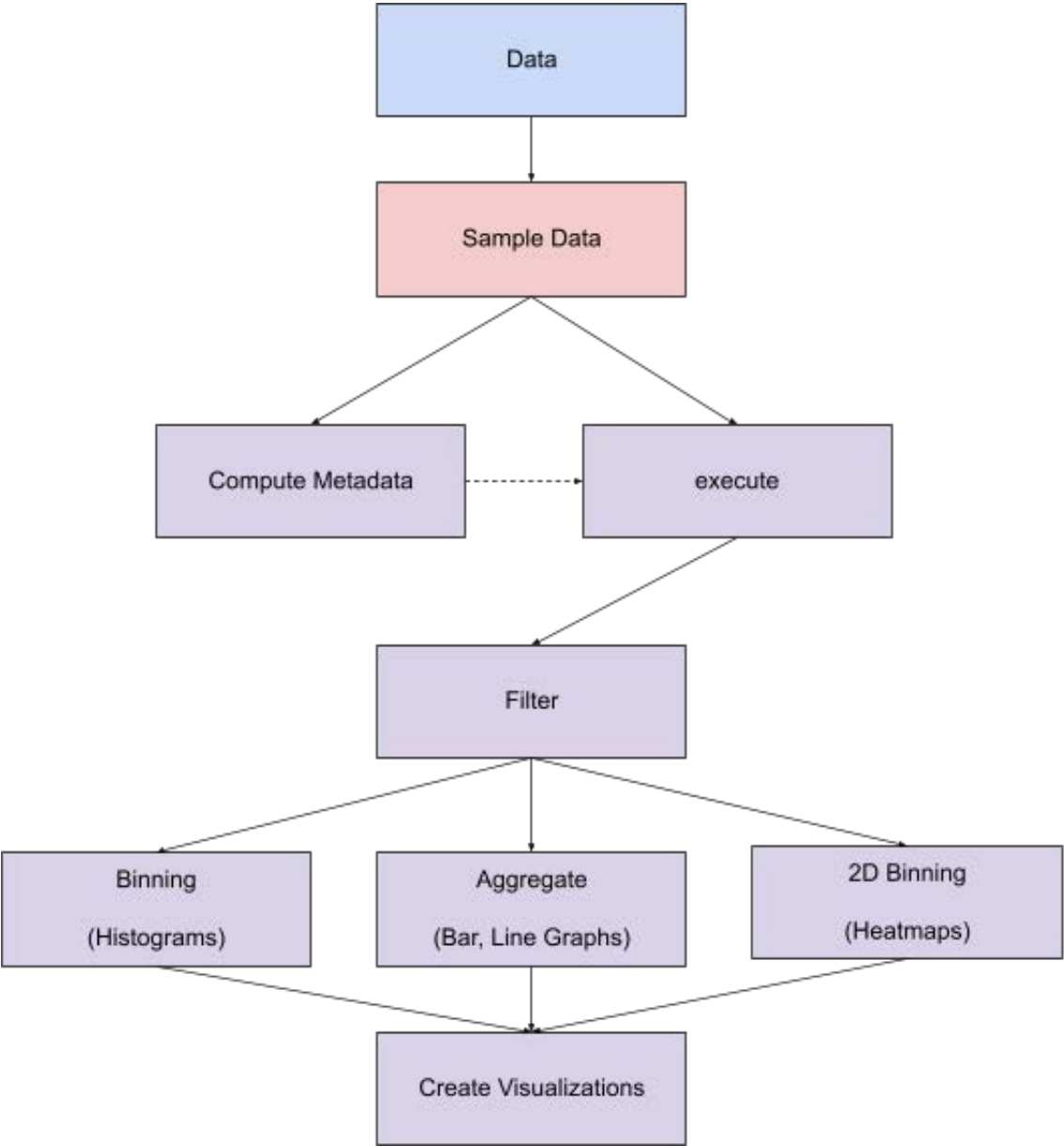


Figure 4.4: Diagram of *Lux* architecture utilizing the new sampling method

Chapter 5

Benchmarking the New Sampling Strategy

Next, we wanted to assess the benefits of the new sampling scheme in *Lux*. We run a number of different experiments comparing the old version of *Lux* with the new version that includes the addition of metadata being calculated on sampled data instead of the full dataset.

5.1 Benchmarking Old vs New Sampling Strategy

The first experiment we run is benchmarking both the old sampling strategy and the new one by comparing the computation time for various different sampling thresholds. To get an accurate comparison, we generate a categorical dataset and a numerical dataset and run *Lux* on each dataset for many different sampling thresholds.

For the categorical dataset, we generate a dataset with 10 columns and 1 million rows. Each column has an increasing cardinality with the first column being a column with only one value. We use `faker` [3] to generate the data with values being generated at random, making it a categorical dataset. We run *Lux* on this dataset using both the old sampling method and the new one which involves using sampling on metadata calculation. We increase the sampling threshold starting at 10,000 in intervals of 10,000 up to the size of the dataset itself (1 million). The results are seen in Figures 5.1-5.4, where the sampling threshold is represented by the x-axis and the time, in seconds, is represented by the y-axis. Each figure represents a different metric tracked, represented by the y-axis. In Figure 5.1, we track the time it takes to fully run *Lux* on the dataset with different sampling thresholds. In Figure 5.2, we measure the time it takes to compute the metadata. In Figure 5.3, we track the time it takes to run the executor engine. Finally, in Figure 5.4, we track the time it takes to create the visualizations. Each of these charts compare the time taken to run the old sampling method (yellow) with the new sampling method (blue). The detailed data for these figures

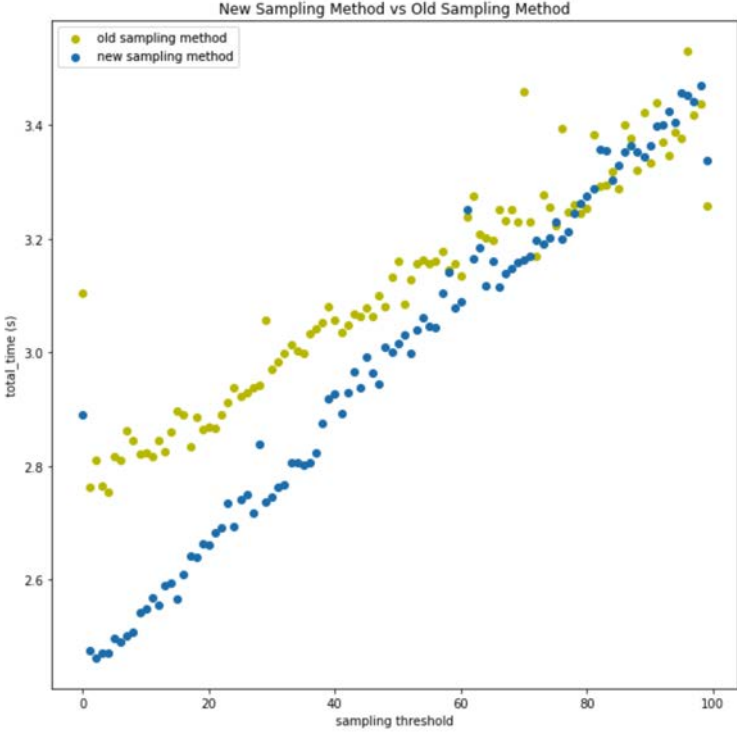


Figure 5.1: Old vs New Sampling Strategy on Categorical Data: Total Time

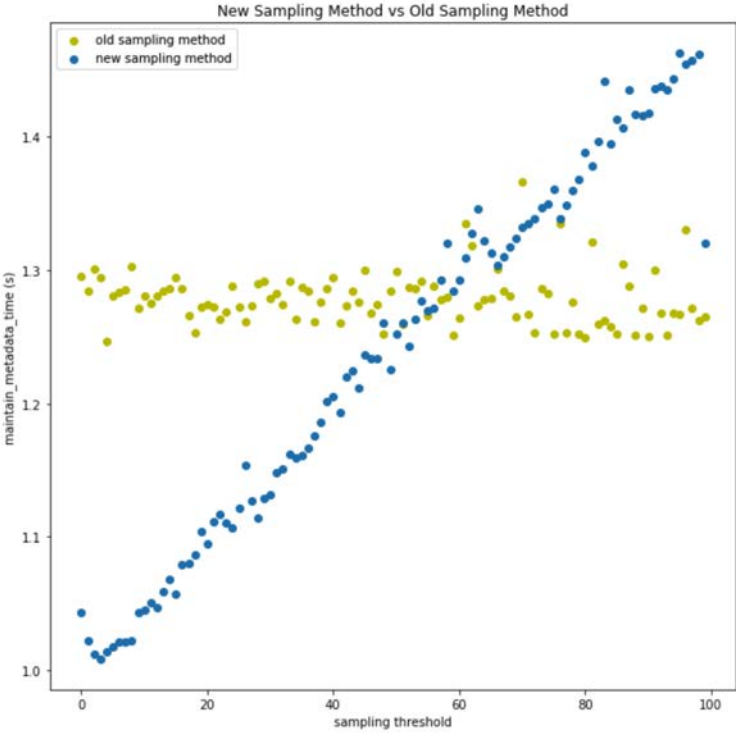


Figure 5.2: Old vs New Sampling Strategy on Categorical Data: Metadata Computation Time

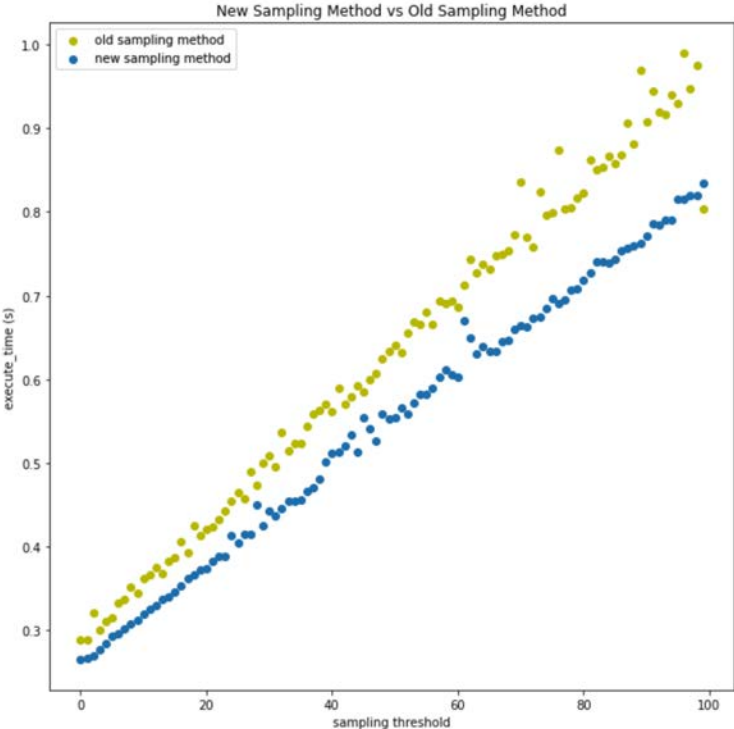


Figure 5.3: Old vs New Sampling Strategy on Categorical Data: Execute Time

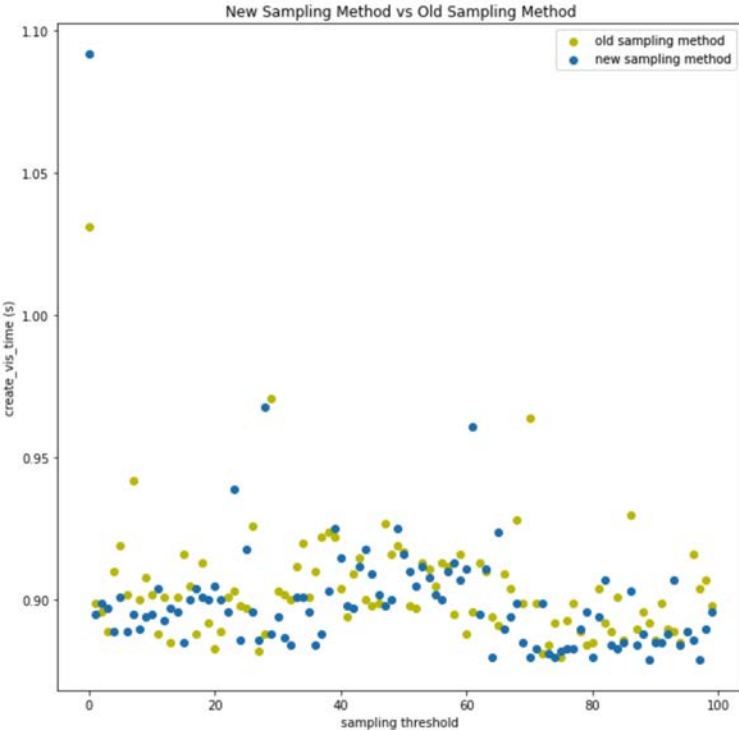


Figure 5.4: Old vs New Sampling Strategy on Categorical Data: create_vis Time

Sampling Threshold	create_vis	maintain_metadata	display	execute	Total
10000	1.031	1.295	0.085	0.288	3.105
110000	0.902	1.281	0.071	0.362	2.824
210000	0.883	1.274	0.072	0.42	2.868
310000	0.903	1.279	0.071	0.508	2.97
410000	0.904	1.294	0.075	0.562	3.056
510000	0.917	1.299	0.076	0.64	3.161
610000	0.888	1.264	0.076	0.686	3.134
710000	0.964	1.366	0.073	0.836	3.459
810000	0.885	1.249	0.073	0.823	3.253
910000	0.886	1.25	0.073	0.907	3.334

Table 5.1: Subset of full dataset for the old sampling method for Figures 5.1 through 5.4. All time is in seconds.

Sampling Threshold	create_vis	maintain_metadata	display	execute	Total
10000	1.092	1.043	0.086	0.265	2.891
110000	0.895	1.045	0.07	0.319	2.548
210000	0.905	1.095	0.071	0.373	2.661
310000	0.894	1.131	0.07	0.442	2.745
410000	0.915	1.205	0.071	0.511	2.927
510000	0.916	1.252	0.073	0.554	3.015
610000	0.911	1.293	0.07	0.602	3.09
710000	0.88	1.332	0.07	0.664	3.162
810000	0.88	1.388	0.07	0.719	3.275
910000	0.885	1.418	0.07	0.771	3.363

Table 5.2: Subset of full dataset for the new sampling method for Figures 5.1 through 5.4. All time is in seconds.

Sampling Threshold	create_vis	maintain_metadata	display	execute	Total
10000	5.414	0.798	0.109	0.32	14.765
110000	6.034	0.778	0.109	0.505	15.288
210000	5.993	0.847	0.11	0.501	15.087
310000	6.437	0.9	0.109	0.509	15.655
410000	6.295	0.913	0.116	0.686	16.043
510000	5.979	0.893	0.11	0.836	16.007
610000	5.837	0.833	0.109	0.848	15.951
710000	5.99	0.899	0.111	0.721	15.803
810000	5.823	0.857	0.11	0.999	15.95
910000	6.22	0.923	0.114	0.946	16.067

Table 5.3: Subset of full dataset for the old sampling method for Figures 5.5 through 5.8. All time is in seconds.

can be found in Tables 5.1 and 5.2.

As we can see in Figure 5.1, the old sampling method is slower overall for all sampling thresholds when compared to the new sampling method. As the sampling threshold moves farther away from the size of the dataset, the speedup increases as well. Figure 5.2 shows that while the old sampling method has a constant expensive computational cost for computing the metadata, the new sampling method’s metadata computation time increases with the size of the sample. The executor engine sees significant speedup with the new sampling method, as seen in Figure 5.3. Finally, in Figure 5.4, we see that the sampling method doesn’t change the time it takes to create the visualizations.

For the numerical dataset, we generate a dataset with 10 columns and 1 million rows. Half of the columns have integer values and the other half have float values. We run *Lux* on this dataset using both the old sampling method and the new one which involves using sampling on metadata calculations. The results are seen in Figures 5.5-5.8 and Tables 5.3-5.4. The results are seen in Figures 5.5-5.8, where the sampling threshold is represented by the x-axis and the time, in seconds, is represented by the y-axis. Each figure represents a different metric tracked, represented by the y-axis. In Figure 5.5, we track the time it takes to fully run *Lux* on the dataset with different sampling thresholds. In Figure 5.6, we measure the time it takes to compute the metadata. In Figure 5.7, we track the time it takes to run the executor engine. Finally, in Figure 5.8, we track the time it takes to create the visualizations. Each of these charts compare the time taken to run the old sampling method (yellow) with the new sampling method (blue). The detailed data for these figures can be found in Tables 5.3 and 5.4.

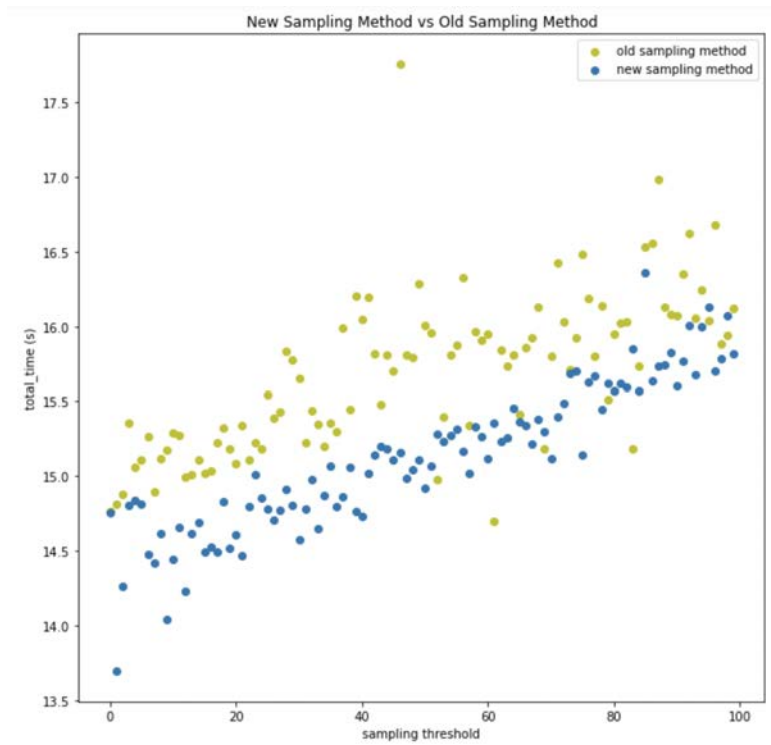


Figure 5.5: Old vs New Sampling Strategy on Numerical Data: Total Time

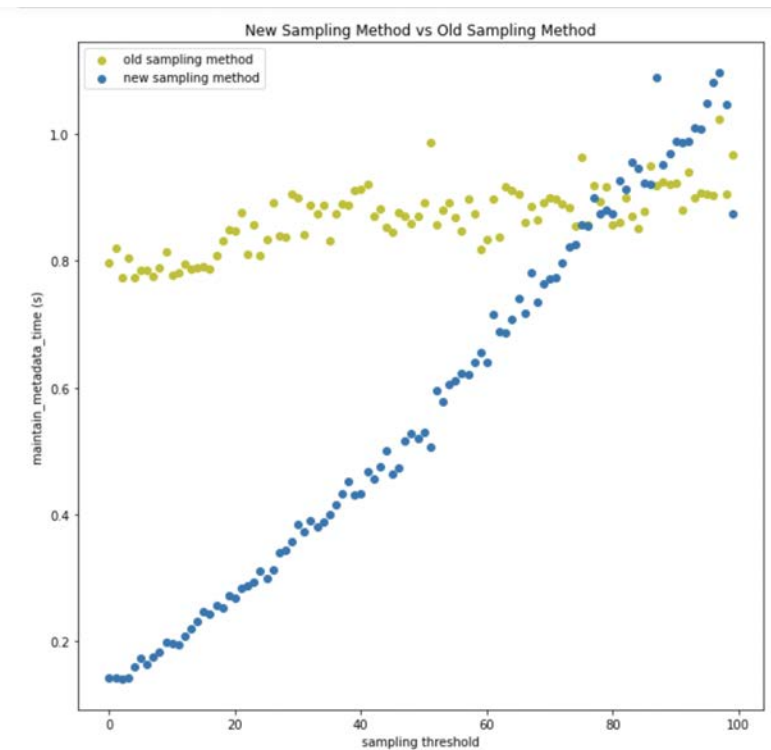


Figure 5.6: Old vs New Sampling Strategy on Numerical Data: Metadata Computation Time

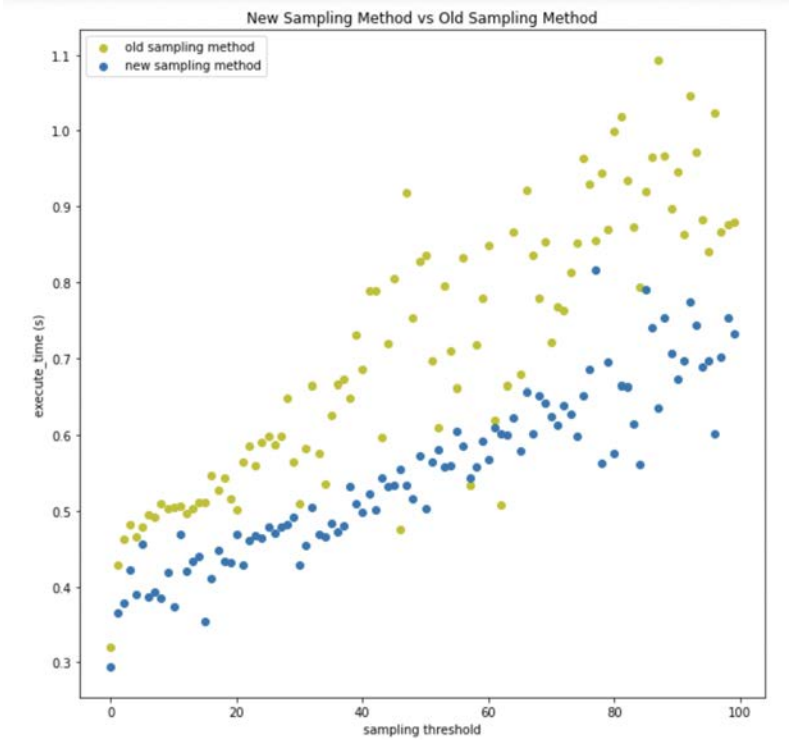


Figure 5.7: Old vs New Sampling Strategy on Numerical Data: Execute Time

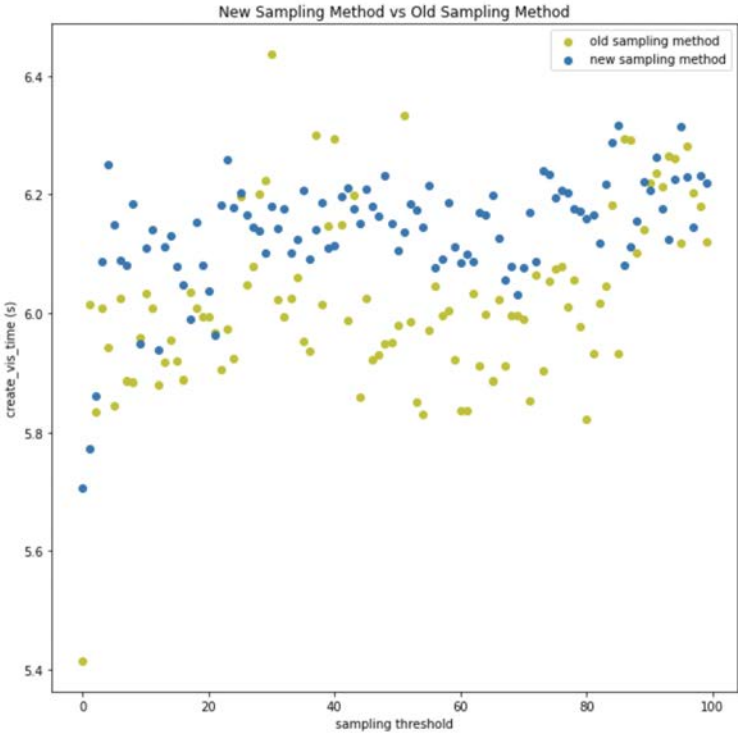


Figure 5.8: Old vs New Sampling Strategy on Numerical Data: create_vis Time

Sampling Threshold	create_vis	maintain_metadata	display	execute	Total
10000	5.706	0.142	0.117	0.294	14.751
110000	6.109	0.195	0.112	0.374	14.439
210000	6.037	0.267	0.107	0.469	14.606
310000	6.18	0.384	0.109	0.428	14.575
410000	6.113	0.433	0.112	0.498	14.732
510000	6.105	0.53	0.111	0.502	14.921
610000	6.085	0.64	0.112	0.568	15.117
710000	6.076	0.772	0.112	0.624	15.119
810000	6.159	0.875	0.116	0.575	15.57
910000	6.206	0.989	0.113	0.672	15.599

Table 5.4: Subset of full dataset for the new sampling method for Figures 5.5 through 5.8. All time is in seconds.

As we can see in Figure 5.5, the old sampling method is slower overall for all sampling methods when compared to the new sampling method, just like with the categorical dataset. Figure 5.6 shows that while the old sampling method has a constant expensive computational cost for computing the metadata, the new sampling method’s metadata computation time increases with the size of the sample. The executor engine sees significant speedup with the new sampling method, as seen in Figure 5.7. Finally, in Figure 5.8, we see that the sampling method doesn’t change the time it takes to create the visualizations.

5.2 Fidelity of Visualizations Generated

The previous experiments showed that sampling allows results to be generated faster. Next, we want to assess if these visualizations have high fidelity when used by data scientists in a real world context. For this experiment, we test the fidelity of the visualizations generated using the Airbnb and Communities datasets.

For the Airbnb dataset, we run *Lux* on the 1 million size dataset we generated with the default sampling threshold being 100,000. In Figure 5.9 and Figure 5.10, we see the visualizations generated for two different categories of visualizations (Correlation and Occurrence) using the old sampling method which doesn’t sample to calculate the metadata. It takes around 9.44 seconds to generate the visualizations with the old sampling method.

We then take a look at the visualizations generated by the new sampling method (Figure 5.11 and Figure 5.12). It takes around 9.03 seconds to generate the visualizations with the

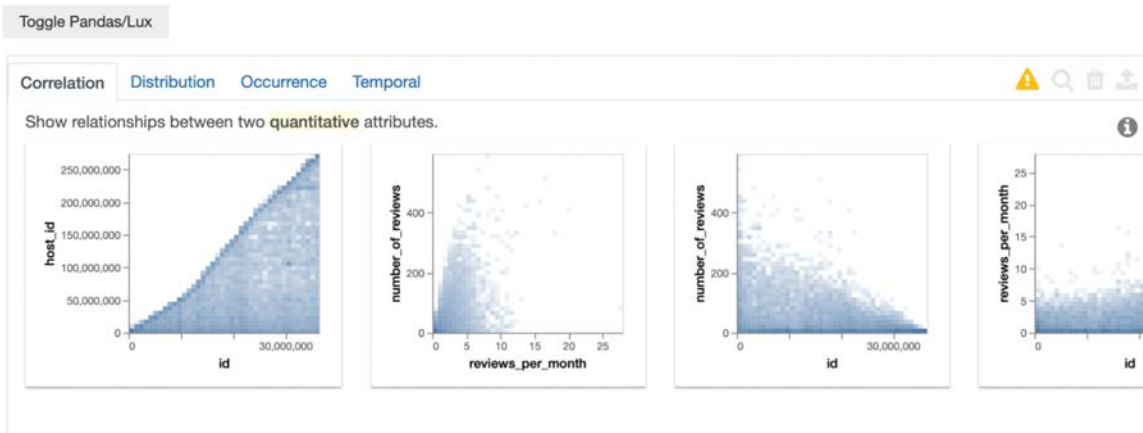


Figure 5.9: Correlation Visualizations for Airbnb Dataset using the old sampling method

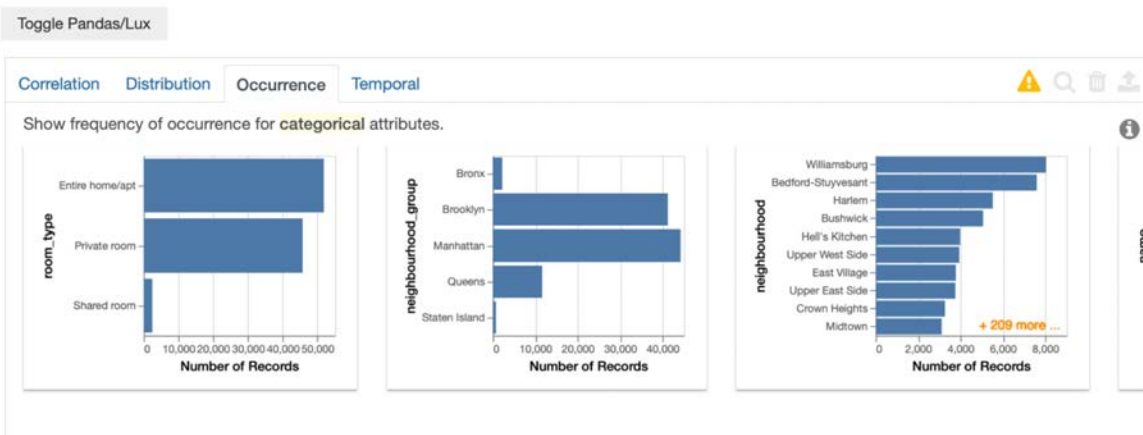


Figure 5.10: Occurrence Visualizations for Airbnb Dataset using the old sampling method

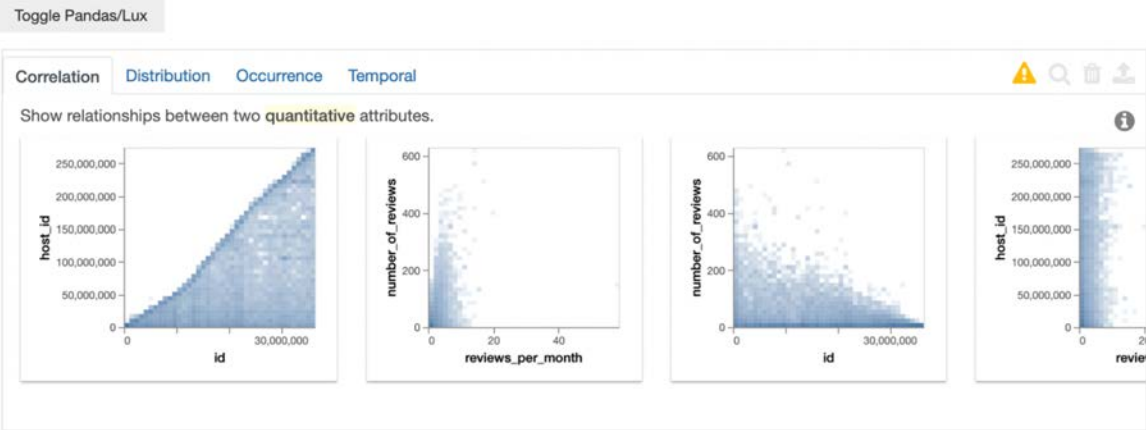


Figure 5.11: Correlation Visualizations for Airbnb Dataset using the new sampling method

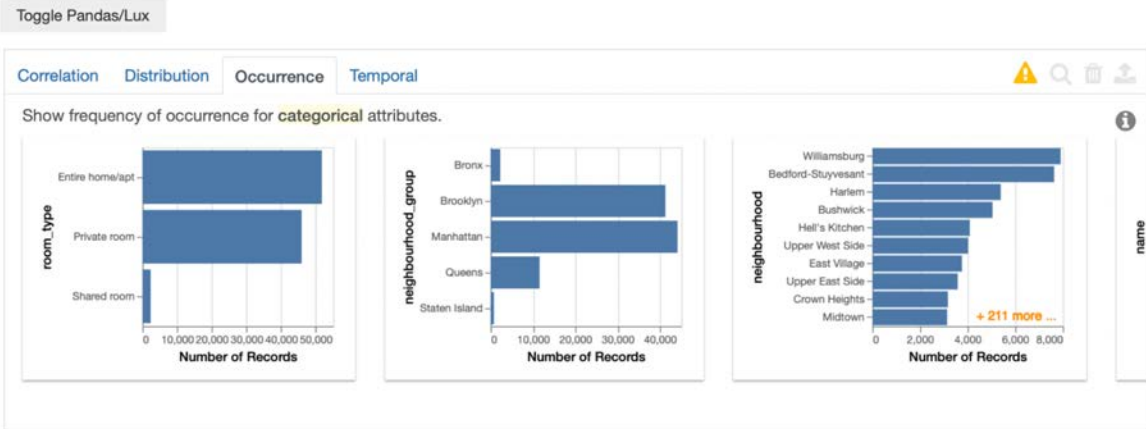


Figure 5.12: Occurrence Visualizations for Airbnb Dataset using the new sampling method

new sampling method.

When comparing the visualizations among all categories, we see the new sampling method makes no meaningful changes to the visualizations generated. The ordering of the visualizations stays mostly the same and the information the generated visualizations conveys is the same. As we can see in Figures 5.9 to 5.12, the top three visualizations for each of the categories shown (Correlation and Occurrence) are effectively the same and convey the same information. We can conclude that the new sampling method has no impact on the actual fidelity of the visualizations generated and no information is lost from using approximated metadata rather than fully accurately calculated metadata.

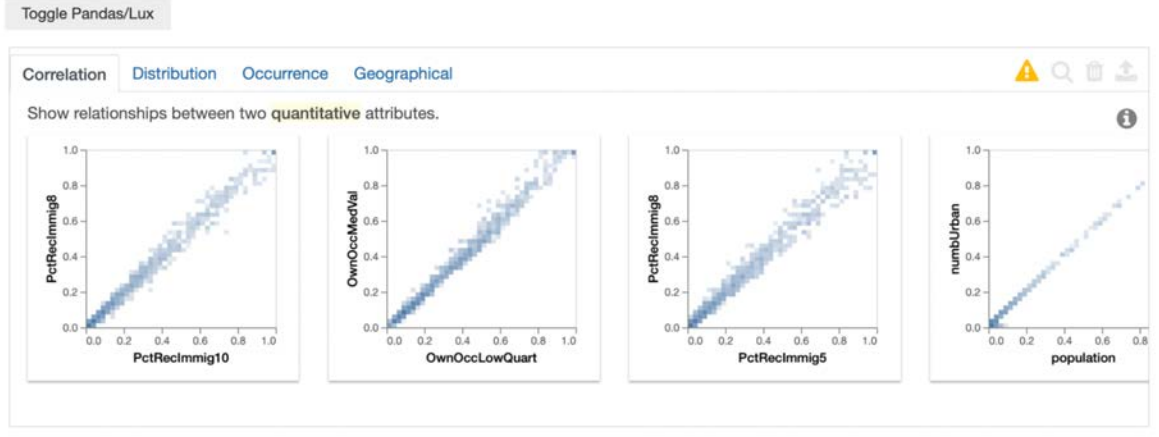


Figure 5.13: Correlation Visualizations for Communities Dataset using the old sampling method

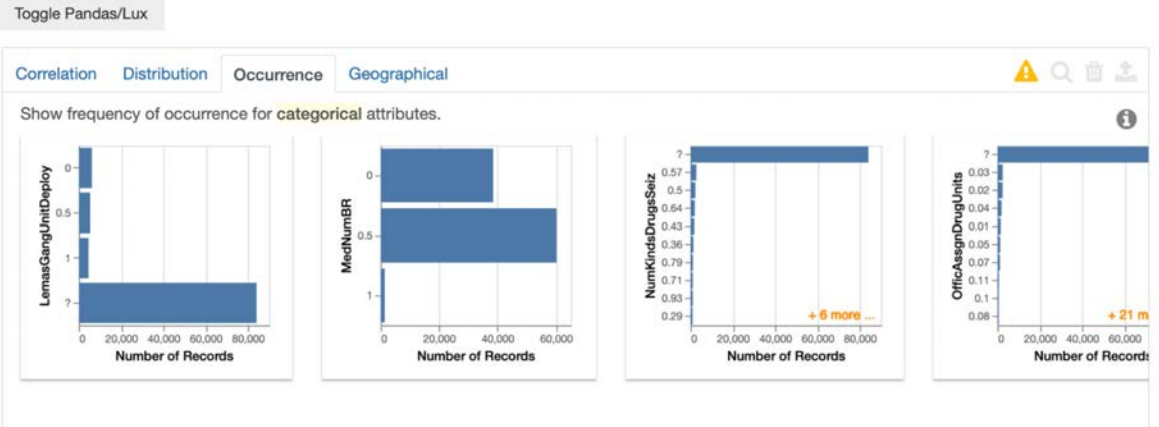


Figure 5.14: Occurrence Visualizations for Communities Dataset using the old sampling method

We run the same experiment on the Communities dataset, specifically the generated dataset with 1 million data points. Again, we use the default sampling threshold of 100,000 to simulate how a typical data scientist may utilize *Lux*.

For the old sampling method, we see the visualizations generated for the Correlation and Occurrence categories in Figures 5.13 and 5.14. It took around 61.57 seconds to generate the visualizations with the old sampling method.

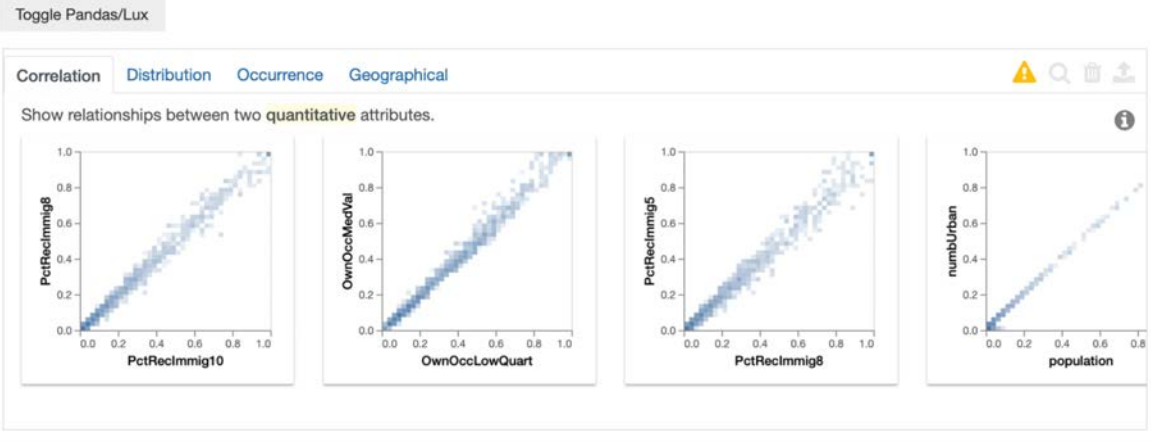


Figure 5.15: Correlation Visualizations for Communities Dataset using the new sampling method

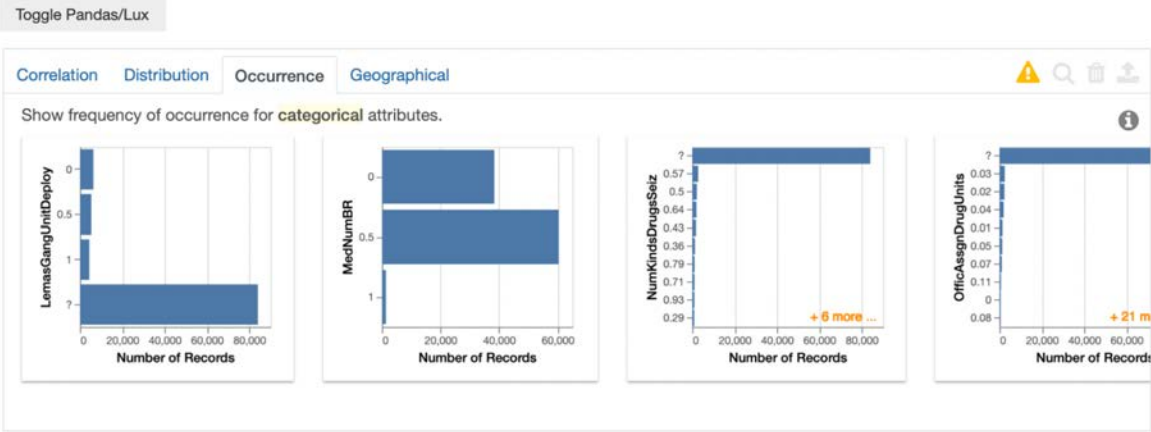


Figure 5.16: Occurrence Visualizations for Communities Dataset using the new sampling method

We then take a look at the visualizations generated by the new sampling method, seen in Figure 5.15 and Figure 5.16. It took around 57.74 seconds to generate the visualizations with the new sampling method.

Just like with the Airbnb dataset, when comparing the visualizations among all categories, we see the new sampling method makes no meaningful changes to the visualizations generated. The ordering of the visualizations stays mostly the same and the information the

generated visualizations conveys is the same. As we can see in Figures 5.13 to 5.16, we see the top three visualizations generated for both the categories shown are effectively the same.

Chapter 6

Incorporating Sampling Beyond *Lux*

Through our work on making *Lux* more efficient, we learned the usefulness of sampling as a strategy for efficient computation for the purposes of ‘always on’ EDA software. As new EDA tools are developed and introduced, we propose some guidelines on how to use sampling to make these tools more efficient.

6.1 Design Principles for Incorporating Sampling

When designing an EDA system, there are a few principles that we recommend following to incorporate sampling into the software.

Fidelity of the Analysis

As sampling is added to an EDA system, any analysis done on the data is no longer fully accurate, it is merely an approximation. The sampling should be done in a way that is as random as possible in order to maintain as accurate a representation of the full dataset. The analysis resulting from sampled data that is outputted from the EDA system should convey the same or similar information as analysis that results from the full dataset. In *Lux* this came in the form of generated recommended visualizations, and while the visualizations weren’t exactly the same as those generated for a full dataset, they were similar enough that the information conveyed by the data was the same. As we saw in Chapter 5, the visualizations generated were very similar between the old sampling method and the new sampling method.

Transparency of Sampling

It is important for the user to know if the EDA system they’re using is employing sampling methods to generate approximate analyses. Otherwise the user may take away conclusions

from the system that they believe to be true regarding the entire dataset, when in fact it's derived from a random sample of the data. Having a warning label, similar to what *Lux* uses in Figure 4.2, is a viable solution to keeping the use of sampling transparent to the user.

In addition to transparency when sampling is occurring, it is important to never change the data the user gives to the system with sampled data. The EDA system should continue to act as if it is operating on the full dataset, but generate analyses based on the sampled data. This allows the user to continue to analyze their full dataset alongside looking at the generated visualizations which provide a good approximation to their data.

User Configurable Sampling Settings

While an EDA system may correctly choose sampling methods to be the best way to make their software more efficient, users may want to conduct their analysis on the entire dataset, even if it comes at the cost of inefficient computation. It is important for the user to be able to easily switch off sampling within the structure of the software itself. The user should also be able to adjust the extent to which sampling is used. For example, in *Lux*, we make it very simple for the user to adjust the threshold for the size of the dataset wherein sampling would begin (effectively setting the maximum size for dataset that *Lux* will compute analysis on, and sampling the dataset if it is bigger than that threshold). The user has full autonomy over that threshold, and they can get more accurate results by increasing that threshold at the cost of slower computation. Kwon et. al. discuss the importance of user driven sampling methods, mentioning that “users may want to steer the sampling process by explicitly specifying sampling characteristics or criteria” [8]. The EDA system can have a default setting for sampling strategies, but an effective EDA system should allow the user to adjust sampling settings to better fits their needs.

Pre-computation of Sampled Data

Sampling itself can be an expensive process, as it requires a full pass through the data to truly achieve a random sample of the data. It is important that in any EDA system that utilizes sampling that the sampling of the data itself is only done once. All further calculations should be then computed using the saved sampled data, rather than continuously sampling the data. This would defeat the purpose of using sampling as there wouldn't be any measurable speedup in the computation. *Lux* does this particularly well, as the sampled data is saved as an attribute in the `LuxDataFrame` which represents the full dataset. Any further computation, from metadata calculations to visualization generation then uses this sampled attribute.

6.2 Applications of Sampling

While sampling seems to be a foolproof method to speed up any EDA system, it's important to understand where sampling is useful and where it should be avoided.

Approximations

Sampling inherently creates an approximation of the data at a cost of the size of the data itself. Therefore, the developer should be aware how this may affect their results. Oftentimes, depending on how the analysis is conducted by the EDA system, sampling can have a major impact on the results. In fact, even different samples may produce vastly different results.

It's also important to recognize that approximate results due to sampling may produce more desirable results. For example, a multivariate quantitative correlation plot may be difficult to read with too many points. However, taking a sample of the data may produce a far cleaner plot that is simpler to read.

Efficiency

As was discussed earlier in the paper, having an EDA system operate on a smaller sampled dataset rather than the full dataset can improve the efficiency of the system. However, it is also important to note that if much of the computation is not done in terms of the data, but is happening elsewhere in the system, then sampling may not provide a speedup that makes the drop in accuracy of the resulting analysis worthwhile. For example, in *Lux*, if the time it takes to build the dashboard skeleton in the front-end took much longer than it currently does, any gains in speed from sampling would not be useful. Instead, efforts should be made to speed-up the building of the dashboard skeleton, which has nothing to do with the dataset itself.

Chapter 7

Conclusion

To make exploratory data analysis more efficient on large datasets we look to sampling as a form of approximating the data to provide fast analysis results. We examined *Lux* which already incorporates sampling in parts of its architecture and determined how we could incorporate additional sampling to provide reliable analysis quickly to the user. By sampling the metadata computed to assist with further calculations necessary for the generation of visualizations, we were able to make *Lux* more efficient on large datasets. As we saw in the experiments we ran in Chapter 5, there was a significant speedup in *Lux* when utilizing the new sampling method. Using what we learned from this process, we constructed some guiding principles to aid future developers as they incorporate sampling into their own EDA systems.

Bibliography

- [1] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. “Approximate query processing: No silver bullet”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 511–519.
- [2] Dgomonov. “Data Exploration on NYC Airbnb”. In: *Kaggle* (2020).
- [3] Daniele Faraglia and Other Contributors. *Faker*. URL: <https://github.com/joke2k/faker>.
- [4] Minos N Garofalakis and Phillip B Gibbons. “Approximate Query Processing: Taming the TeraBytes.” In: *VLDB*. Vol. 10. 2001, pp. 645927–672356.
- [5] John D Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in science & engineering* 9.03 (2007), pp. 90–95.
- [6] Kkanda. “Analyzing UCI Crime and Communities Dataset”. In: *Kaggle* (2020).
- [7] Thomas Kluyver et al. *Jupyter Notebooks-a publishing format for reproducible computational workflows*. Vol. 2016. 2016.
- [8] Bum Chul Kwon et al. “Sampling for scalable visual analytics”. In: *IEEE computer graphics and applications* 37.1 (2017), pp. 100–108.
- [9] Doris Jung-Lin Lee et al. “Deconstructing Categorization in Visualization Recommendation: A Taxonomy and Comparative Study”. In: *IEEE Transactions on Visualization and Computer Graphics* (2021).
- [10] Doris Jung-Lin Lee et al. “Lux: always-on visualization recommendations for exploratory dataframe workflows”. In: *Proceedings of the VLDB Endowment* 15.3 (2021), pp. 727–738.
- [11] S Macke et al. *Fastmatch: Adaptive algorithms for rapid discovery of relevant histogram visualizations*. Tech. rep. Technical report, Available at: [http://data-people.cs.illinois.edu/papers ...](http://data-people.cs.illinois.edu/papers...), 2017.
- [12] Dominik Moritz et al. “Trust, but verify: Optimistic visualizations of approximate queries for exploring big data”. In: *Proceedings of the 2017 CHI conference on human factors in computing systems*. 2017, pp. 2904–2915.

- [13] Philipp Moritz et al. “Ray: A distributed framework for emerging {AI} applications”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 561–577.
- [14] Aditya Parameswaran, Neoklis Polyzotis, and Hector Garcia-Molina. “Seedb: Visualizing database queries efficiently”. In: *Proceedings of the VLDB Endowment 7.4* (2013), pp. 325–328.
- [15] Devin Petersohn et al. “Towards scalable dataframe systems”. In: *arXiv preprint arXiv:2001.00888* (2020).
- [16] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: <https://doi.org/10.5281/zenodo.3509134>.
- [17] Jacob VanderPlas et al. “Altair: interactive statistical visualizations for Python”. In: *Journal of open source software 3.32* (2018), p. 1057.