A Spreadsheet Interface for Dataframes

Richard Lin
Aditya Parameswaran, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-107
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-107.html

May 14, 2021

Copyright © 2021, 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.

Acknowledgement

Please see page vi for a dedicated acknowledgement.

A Spreadsheet Interface for Dataframes

by Richard Lin

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:

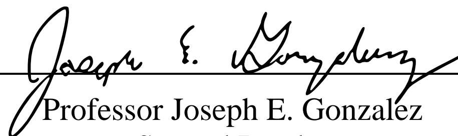
Committee:

Professor Aditya Parameswaran
Research Advisor

05/14/2021

(Date)

* * * * * * *



Professor Joseph E. Gonzalez
Second Reader

05/14/2021

(Date)

A Spreadsheet Interface for Dataframes

Copyright 2021
by
Richard Lin

Abstract

A Spreadsheet Interface for Dataframes

by

Richard Lin

Master of Science in Computer Science

University of California, Berkeley

Professor Aditya Parameswaran, Co-chair

Professor Joseph E. Gonzalez, Co-chair

Spreadsheets are easy to learn and provide intuitive controls for exploring data, but they scale poorly and make it hard to perform certain tasks in compared to code. Dataframes are more performant than spreadsheets and can support significantly larger datasets, but have a steeper learning curve and are less interactive. The respective advantages and disadvantages of spreadsheets and dataframes lead data scientists to switch between the two for various steps. Rather than regarding them as separate workflows, we want to integrate them into a single workflow so that users can take advantage of both tools.

We present Modin Spreadsheet, a spreadsheet UI for dataframes with a specific implementation choice based on the popular Modin dataframe system. Modin Spreadsheet builds off of Qgrid in modeling spreadsheet data as a dataframe and improves on traditional spreadsheet software in the aspects of interactivity, scalability, and reproducibility. Modin Spreadsheet's integration of spreadsheets into a coding interface allows it to create a new form of reproducibility for spreadsheets through the representation of spreadsheet changes as dataframe code.

Contents

Contents i

List of Figures iii

List of Tables v

1 Introduction 1
- 1.1 Scalability Issues with Spreadsheets 2
- 1.2 Problems tackled in Modin Spreadsheet 2

2 Technical Background 6
- 2.1 Modin 6
- 2.2 Qgrid 7

3 Modin Spreadsheet Overview 10
- 3.1 User Interface 12
- 3.2 Scalability 13
- 3.3 Clearing Operations 13

4 Recording Spreadsheet Changes 15
- 4.1 Auto-display of Reproducible Code 18
- 4.2 Displaying Multiple Spreadsheets 20
- 4.3 Condensing Reproducible Code 20

5 Related Work 23
- 5.1 Bamboolib 23
- 5.2 Ipysheet 23

6 Future Work 26
- 6.1 Formula Computation & Cell References 26
- 6.2 Undo & Redo Operations 27
- 6.3 Bidirectional Spreadsheet to Code interface 28
- 6.4 Update Slickgrid Version 29

6.5 Additional Spreadsheet Operations 29

7 Conclusion **31**

Bibliography **32**

List of Figures

1.1 The experiment was run on a 6-Core Intel Core i7 2019 Macbook Pro. The y-axis measures the mean of 5 trials for the time taken to run Pandas `read_excel` on a given size dataset. The base dataset is a 3MB weather dataset consisting of 50,000 rows and 13 columns with a mix of values and formulas. The dataset is scaled up for 250,000 rows and 1,000,000 rows while keeping the number of columns constant. . . . 1
1.2 Version snapshots from Google Sheets Version History. . . . 4
1.3 Examples of Excel features for tracking changes. . . . 5

2.1 Replace Pandas with Modin by just changing one line of code. . . . 6
2.2 High-Level Architectural View of Modin [11]. Modin logically separates their architecture into the layers of APIs, Query Compiler, Middle Layer, and Execution. The spreadsheet UI is a component in the APIs layer. . . . 7
2.3 Screenshot of Qgrid's Architecture from a demo video [16]. Only rows in the spreadsheet viewport are sent to the Javascript frontend for the browser to render. Operations are performed on the Python back-end and new rows are sent to the browser if changes occur. . . . 8

3.1 Example of the Modin Spreadsheet UI. . . . 10
3.2 Example of using Modin Spreadsheet to convert between a dataframe and the spreadsheet interface. . . . 13
3.3 Example code of a filter operation. . . . 14

4.1 Filtering a column in Google Sheets requires clicking on the column's filter button and then selecting the filter criteria. . . . 15
4.2 Version snapshots from Google Sheets Version History. . . . 16
4.3 Examples of Excel features for tracking changes. . . . 17
4.4 Screenshot of example Python code that was output in the history cell after several spreadsheet operations. . . . 19
4.5 Example of reproducible code before and after being condensed. . . . 21

5.1 Spreadsheet interface of Bamboolib. . . . 24
5.2 Plotting interface of Bamboolib. . . . 25

5.3 Example of an Ipysheet spreadsheet in Jupyter Notebook. 25

List of Tables

1.1 Pros and cons of spreadsheet systems. 3

4.1 Mapping between spreadsheet operations and Python code snippet that is output to the history cell. Some variable names are altered for readability. The code inside curly braces is evaluated and converted into string format before being printed. For example, the user would end up seeing `df.drop([0,1,2], inplace=True)` for the remove row(s) operation. 19

Acknowledgments

I would like to thank Professor Aditya Parameswaran for advising me during my candidacy. He has fostered a welcoming research community, provided me with countless learning opportunities, and helped turn a year of unfortunate circumstances into an enjoyable one.

I would like to thank Devin Petersohn for introducing me to the opportunities in Modin and supporting me immensely throughout my candidacy. His advice and guidance have empowered me to accomplish a number of feats not described in this report and have made me feel supported during this project.

I would like to thank Doris Junglin Lee for acting as a great source of advice and inspiration during this project. I would like to thank the whole DataSpread team, who I have happily worked with over the past year. I would also like to thank Sajjadur Rahman and Doris Xin for their past mentorship and for helping me improve as a researcher.

Finally, I would like to thank my partner, Elisa. Her continued support has pulled me through difficult times and has been integral to my success. I am extremely grateful for her and her support.

Chapter 1

Introduction

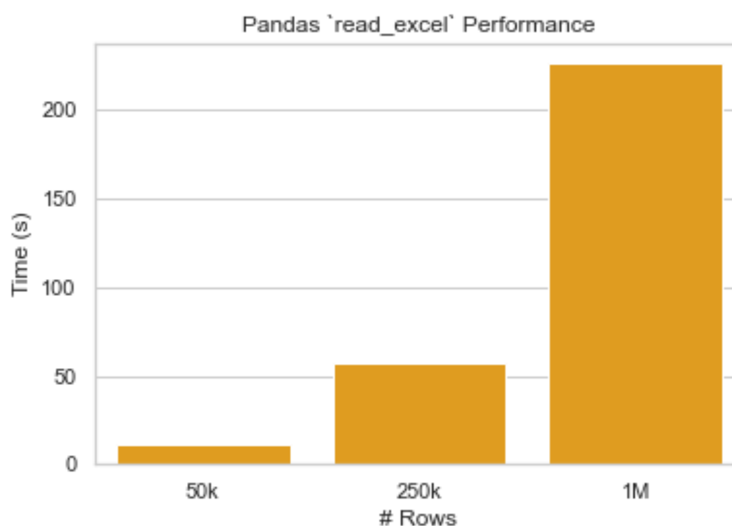


Figure 1.1: The experiment was run on a 6-Core Intel Core i7 2019 Macbook Pro. The y-axis measures the mean of 5 trials for the time taken to run Pandas `read_excel` on a given size dataset. The base dataset is a 3MB weather dataset consisting of 50,000 rows and 13 columns with a mix of values and formulas. The dataset is scaled up for 250,000 rows and 1,000,000 rows while keeping the number of columns constant.

As datasets have grown larger and gigabytes of data have become commonplace, many data scientists and analysts have had to move from using spreadsheets to tools like Pandas [13] to manipulate and explore their data [15]. However, learning how to write code proficiently in Pandas requires considerable effort, and Pandas may not provide the interactivity that data analysts are used to. As a result, many will still switch between using Pandas and Excel (or other spreadsheet systems) because spreadsheets support direct manipulation

and are intuitive for certain types of operations as opposed to writing code [8]. This back-and-forth between Pandas and Excel, however, is messy and lossy. The time to export from Excel and import into Pandas and vice-versa can often be a major bottleneck to productivity, not to mention the overhead from context-switching between the two tools. From the measurements shown on Figure 1.1, reading in an Excel file into Pandas takes up to a few minutes.

In addition to the switching overhead, spreadsheets can be unresponsive with as little as 20,000 rows [10], which can be as little as 1MB of data, a small fraction of the size of typical datasets. To address this problem, we propose Modin Spreadsheet, a spreadsheet UI to complement dataframe workflows and allow users to take advantage of the interactivity of spreadsheets without having to switch back and forth between spreadsheet software and typical development environments, such as computational notebooks.

1.1 Scalability Issues with Spreadsheets

Spreadsheets are the tool of choice for billions of people for analyzing and managing data. Its usage spans across a wide range of industries and institutions for both novice and advanced users [2]. However, the increase in dataset sizes have rendered spreadsheets unusable for many applications. At larger data sizes, common spreadsheet operations and even scrolling can become unresponsive [19]. Spreadsheet software have tried to accommodate these size increases; Excel changed its worksheet size limit from 65k to one million rows [5] and Google Sheets expanded its size limit from two million to five million cells [6]. These problems, however, can occur even at a fraction of the size limits set by current spreadsheet systems. Mack et al. notes that spreadsheets with as few as 18,000 rows (<2% of Excel's row limit) can have issues with computation related lagging [10].

1.2 Problems tackled in Modin Spreadsheet

As shown in Table 1.1, which describes the pros and cons of spreadsheet systems, spreadsheets have advantages over dataframes that can make it more preferable for Exploratory Data Analysis (EDA). While preferable for iterative processes, spreadsheets are not applicable in every scenario due to their no-code direct manipulation interface and limited scalability. Rather than choose one or the other, it would be ideal to bring both tools together in an integrated experience and receive the benefits of both spreadsheets and dataframes. In the following sections, we elaborate how Modin Spreadsheet improves on traditional spreadsheet software in the aspects of interactivity, scalability, and reproducibility.

Scalability

Spreadsheets are the most popular data analytics tool, but their size limits are well below the size of a growing number of datasets used by businesses and institutions [14]. Rather than

Pros	Cons
Graphical user interface with intuitive controls	Scales poorly; unresponsive even at small scale data
Immediate feedback	Slow and complex to perform certain tasks compared to code
Flexible data structure	Difficult to audit and reproduce changes

Table 1.1: Pros and cons of spreadsheet systems.

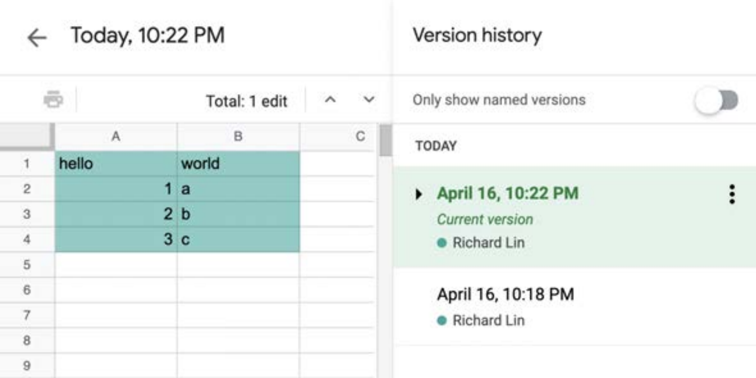
completely replace spreadsheets with another tool, it would be ideal to improve the scalability of spreadsheets so that they can continue to be used for these larger datasets [3]. Unlike prior work which has used databases to increase scalability [2], we use a dataframe to represent the underlying data in the spreadsheet. While dataframes are less scalable than databases due to their increased flexibility, their use in the spreadsheet backend still improves on traditional spreadsheet systems in supporting a greater number of rows and by speeding up spreadsheet operations. One drawback of our dataframe backend implementation is that we constrain the spreadsheet to a dense tabular structure, which takes away from the flexibility of data representation in spreadsheets. There is already prior work on using a relational database for storing sparse data in a spreadsheet [2], so it may be possible for a dataframe to also store spreadsheet data similarly rather than enforcing a tabular data scheme. In that sense, we do not believe a dense tabular view of data is a fundamental constraint of Modin Spreadsheet.

Interactivity

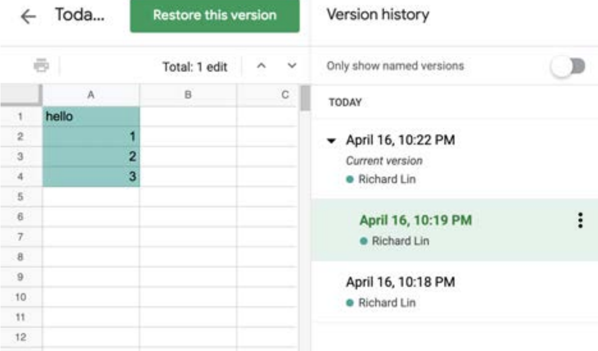
Spreadsheets can be good tools for EDA because of their simplicity, but they are traditionally slow and make it complex to perform certain tasks compared to code, such as joins or custom functions [8]. Some systems, e.g., DataSpread [2] and SIEUFERD [1], have addressed this by supporting relational querying in spreadsheets, but they do not solve the issue of easily performing operations that involve a complex chain of conditionals and function calls, which are commonly found in data engineering pipelines. To get around this limitation, some people convert between spreadsheets and dataframes in order to perform operations that are easier to do in the respective tools. This back-and-forth process leads to unproductive time from importing and exporting data and takes away from the interactivity that is important to iterative processes [9][20]. Modin Spreadsheet removes this barrier by enabling seamless conversion between the direct manipulation interface of spreadsheets and the programmatic interface of dataframes, thereby increasing speed at which users can iterate.

Reproducibility

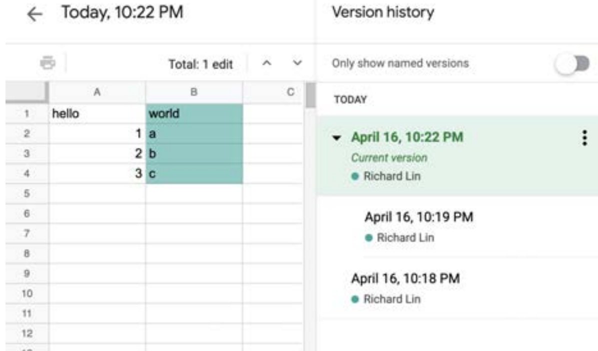
Reproducibility in spreadsheets is complex. In traditional spreadsheet software, the main method of interacting with spreadsheets is through the spreadsheet's graphical user interface [19]. As a result, changes are often represented visually on the spreadsheet, such as in Figures 1.2 and 1.3, which show change tracking in Google Sheets and Excel respectively. While intuitive to understand, the visual representation makes reproducing changes difficult or labor-intensive. Code, on the other hand, models changes in data with each line of code and is easily exported. Since execution of code is programmatic rather than manual, reproducibility just requires running the same code along with any additional state. Modin Spreadsheet takes advantage of the programmatic nature of development environments and brings about a new form of reproducibility for spreadsheets by recording spreadsheet operations as dataframe code.



(a) Coarse-grain data snapshot in Google Sheets.

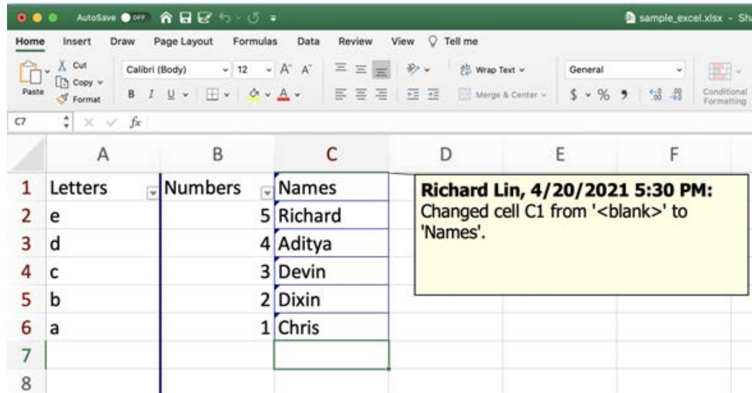


(b) First of two finer-grain snapshots from expanding the snapshot in Figure 4.2a.

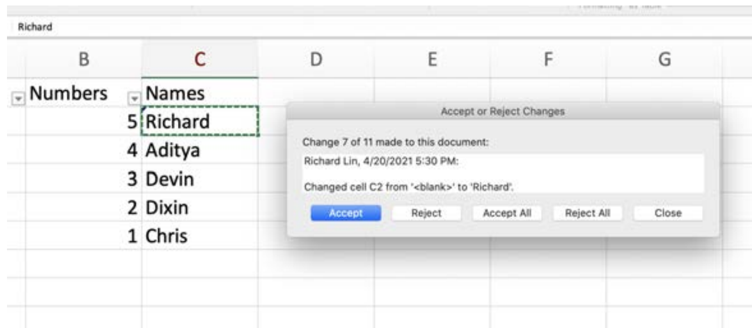


(c) Second of two finer-grain snapshots from expanding the snapshot in Figure 4.2a.

Figure 1.2: Version snapshots from Google Sheets Version History.



(a) Changes are highlighted on the worksheet and describe the operation.



(b) Users can accept or reject changes.

Action Number	Date	Time	Who	Change	Sheet	Range	New Value	Old Value	Action Type	Losing Action
11	4/20/21	5:37 PM	Richard Lin	Cell Change	Sheet1	B5	'=A5*2	<blank>		
12	4/20/21	5:37 PM	Richard Lin	Cell Change	Sheet1	B6	'=A6*2	<blank>		
13	4/20/21	5:37 PM	Richard Lin	Cell Change	Sheet1	B7	'=A7*2	<blank>		
14	4/20/21	5:40 PM	Richard Lin	Row Insert	Sheet1	'4:4				
15	4/20/21	5:40 PM	Richard Lin	Column Delete	Sheet1	'B:B				
16	4/20/21	5:40 PM	Richard Lin	Worksheet Insert	Sheet2					
18	4/20/21	5:41 PM	Richard Lin	Cell Change	Sheet1	A2	10	1		

The history ends with the changes saved on 4/20/2021 at 5:41 PM.

(c) Changes are compactly recorded in a history log with different details.

Figure 1.3: Examples of Excel features for tracking changes.

Chapter 2

Technical Background

Modin Spreadsheet is an open-source system built on top of several pre-existing libraries and systems. In this chapter, we describe the systems that provided the technical foundation for Modin Spreadsheet, Modin and Qgrid, and how they differ from the work presented in this report.

2.1 Modin

```
# import pandas as pd
import modin.pandas as pd
```

Figure 2.1: Replace Pandas with Modin by just changing one line of code.

Modin [15] is an open-source distributed dataframe library that aims to bridge the solutions between dataframes for small data and big data. It automatically distributes data and computation across available resources and is completely compatible with Pandas, providing an effortless way to speed up Pandas code by changing a single line of code as demonstrated in Figure 2.1 [11]. According to the Modin documentation [11], "Modin is logically separated into different layers that represent the hierarchy of a typical Database Management System. Abstracting out each component allows us to individually optimize and swap out components without affecting the rest of the system."

Modin's abstractions enable the Spreadsheet UI to be built as an additional component in the API layer and allow it to continually receive performance benefits as the lower layers are changed [15]. Furthermore, the API abstraction layer provides multiple methods for interacting with the same data, increasing flexibility in accomplishing certain tasks and boosting overall developer productivity.

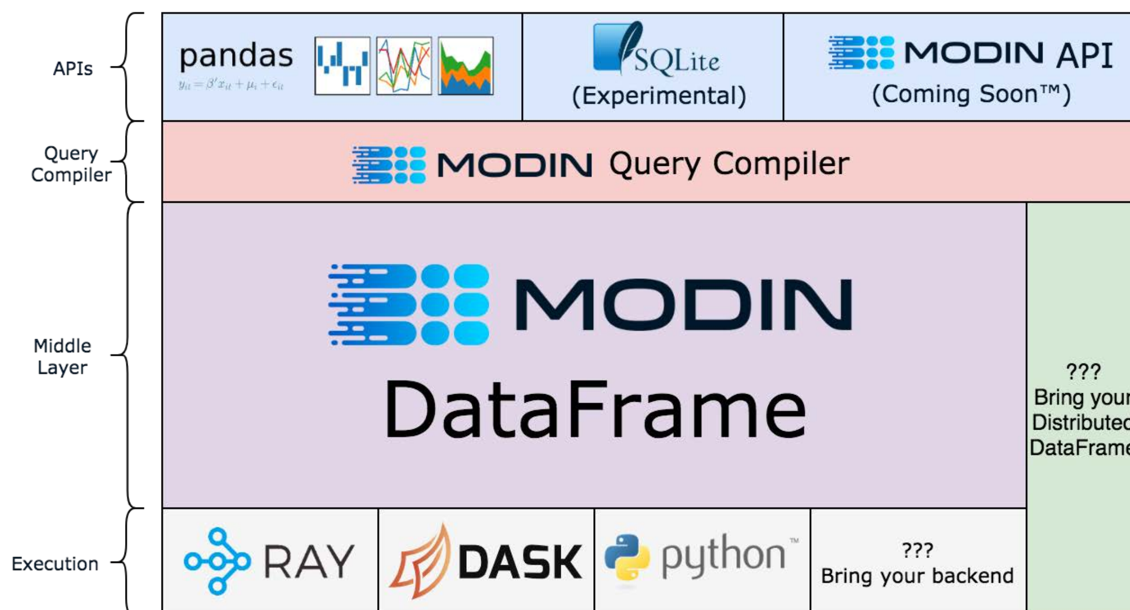


Figure 2.2: High-Level Architectural View of Modin [11]. Modin logically separates their architecture into the layers of APIs, Query Compiler, Middle Layer, and Execution. The spreadsheet UI is a component in the APIs layer.

2.2 Qgrid

Modin Spreadsheet builds off of Qgrid, an open-source tool to render Pandas dataframes as a spreadsheet [17]. Qgrid is no longer maintained because Quantopian, the company that developed it, shut down in November 2020 [4].

Modin Spreadsheet was not built with the intention to replace Qgrid as a spreadsheet UI for Pandas dataframes. Modin Spreadsheet currently uses the Pandas API in implementing spreadsheet operations and can support Pandas dataframes, but this is not the only implementation option. For performance reasons, we may want to change the underlying execution of spreadsheet operations by using a different API, such as Modin's dataframe API, instead of the Pandas API. This change might limit support to Modin dataframes, which would diverge from Qgrid's goal of visualizing Pandas dataframes.

Architecture

As shown in Figure 2.3, the architecture of Qgrid [17] splits into a HTML/Javascript front-end and a Python back-end, which is connected through the Jupyter Widget class. The front-end is what displays the spreadsheet and processes interactions on the graphical user interface. Whenever a user performs an operation on the graphical interface, the front-end will send a message containing the type of operation and other relevant information to the

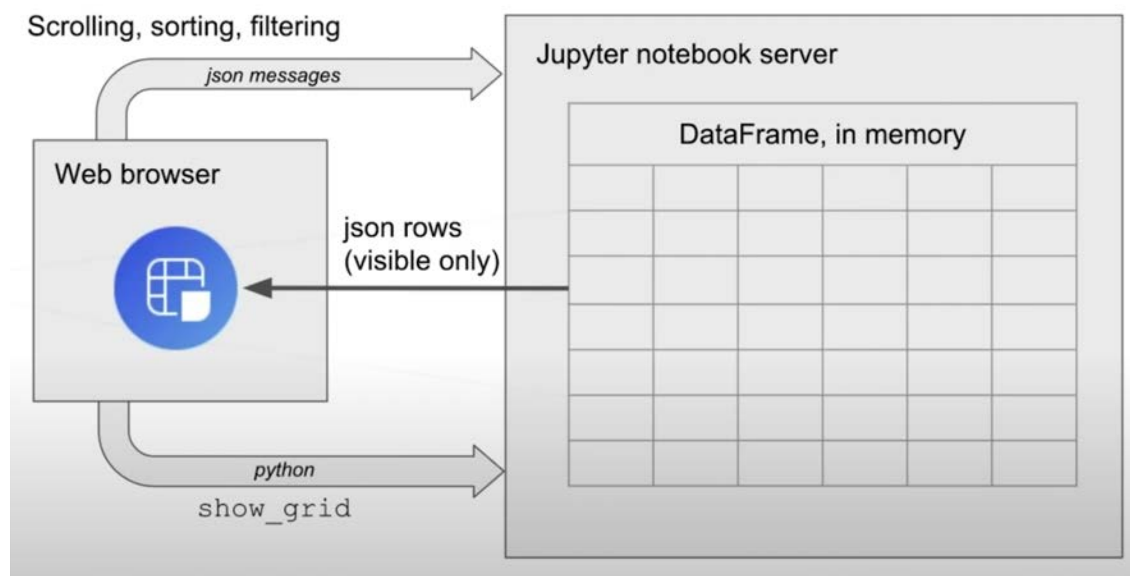


Figure 2.3: Screenshot of Qgrid's Architecture from a demo video [16]. Only rows in the spreadsheet viewport are sent to the Javascript frontend for the browser to render. Operations are performed on the Python back-end and new rows are sent to the browser if changes occur.

back-end, where it is processed on the underlying dataframe. If necessary, the back-end will send a message back to the front-end, such as to notify the completion of an operation [17].

The separation between the graphical interface and the data model enables Qgrid to stay responsive even as data scales. Whenever a user moves their viewport in the spreadsheet, the front-end will request from the back-end the range of rows corresponding to the viewport e.g., row 200 to row 300 [17]. Thus, the whole dataset does not need to be stored in the browser memory. This mechanism limits the rendering latency even as the dataset size increases, alleviating the unresponsive display issues that often pop up in spreadsheet software [10]. Overall, this leads to the front-end being responsive to scrolling and scalable in the number of rows it can support.

Limitations

There lies a disconnect between the reproducibility of computational notebooks and the impermanent nature of Qgrid's spreadsheet widget. While code in a Jupyter Notebook can be shared and re-run, Jupyter widgets are stored in-memory and are not persisted to the Notebook file. Qgrid offers the ability to export the dataframe underlying the spreadsheet interface, but unless the dataframe is saved to disk, changes made to the spreadsheet cannot be saved [17]. Modin Spreadsheet addresses this by exporting changes to the spreadsheet

as Python code, which persists with the Notebook file, so any changes can be shared with collaborators and re-run. This process is described in detail in Chapter 4.

While Qgrid's front-end is scalable due to its architecture, its back-end is limited by the scalability and performance of Pandas dataframes. Pandas' scalability limitations include single-thread execution and excessive memory usage, which are addressed by the work on Modin [15]. We tackle the scalability issue of Qgrid by adapting it to use Modin dataframes instead of Pandas dataframes.

Lastly, the possible spreadsheet operations on Qgrid are relatively small compared to those of popular spreadsheet software e.g., Excel and Google Sheets. For example, Qgrid doesn't support formula computation and cell references. Modin Spreadsheet implements a few more basic spreadsheet operations that are described in Chapter 3. Additional limitations not addressed in Modin Spreadsheet and future work are described in detail in Chapter 6.

Chapter 3

Modin Spreadsheet Overview

Like Qgrid, the Modin Spreadsheet UI [12] builds a spreadsheet out of a dataframe and allows you to seamlessly switch between operating on a dataframe using a spreadsheet and operating on a dataframe using code. The ability to easily convert between spreadsheet and code interfaces removes the need to export data to and from spreadsheet software—a process that can create countless copies, waste time, and lead to hard-to-detect errors. Modin Spreadsheet reuses Qgrid's visualization capabilities and spreadsheet operations, but adds features for reproducibility, additional spreadsheet operations, and a modified back-end, all of which are explained in further detail below.

```
import modin.spreadsheet as mss
spreadsheet = mss.from_dataframe(df)
spreadsheet
```

Add Row | Remove Row | Clear History | Filter History | Reset Filters | Reset Sort

	trip_id	vendor_id	pickup_datetime	dropoff_datetime	store_and_fwd_flag	rate_code_id
0	1	2	2013-08-01 08:14:37	2013-08-01 09:09:06	N	1
1	2	2	2013-08-01 09:13:00	2013-08-01 11:38:00	N	1
2	3	2	2013-08-01 09:48:00	2013-08-01 09:49:00	N	5
3	4	2	2013-08-01 10:38:35	2013-08-01 10:38:51	N	1
4	5	2	2013-08-01 11:51:45	2013-08-01 12:03:52	N	1
5	6	2	2013-08-01 14:33:39	2013-08-01 15:49:00	N	1
6	7	2	2013-08-01 17:19:00	2013-08-01 17:19:00	N	1
7	8	2	2013-08-01 17:22:00	2013-08-01 17:22:00	N	1
8	9	2	2013-08-01 17:24:00	2013-08-01 17:25:00	N	1
9	10	2	2013-08-01 19:21:09	2013-08-01 19:22:30	N	1
10	11	2	2013-08-01 19:29:27	2013-08-01 19:32:38	N	4
11	12	2	2013-08-01 19:33:28	2013-08-01 19:35:21	N	1
12	13	2	2013-08-02 09:37:44	2013-08-02 09:38:08	N	1
13	14	2	2013-08-02 09:43:58	2013-08-02 09:44:13	N	1
14	15	2	2013-08-02 09:44:43	2013-08-02 09:45:15	N	1

```
# ---- spreadsheet transformation history ----
unfiltered_df = df.copy()
```

Figure 3.1: Example of the Modin Spreadsheet UI.

Figure 3.1 shows an example of the Modin Spreadsheet UI, which only requires a few lines to get started. The spreadsheet UI is displayed as a Jupyter Notebook widget inline with other code so that switching between the spreadsheet interface and other code is swift and easy.

Modin Spreadsheet [12] offers a number of capabilities with a focus on enabling easy exploration/manipulation of data and providing reproducibility:

Exploration Features

- Responsive scrolling: Whenever a user scrolls, or moves their spreadsheet viewport vertically, the Modin Spreadsheet front-end will retrieve the data in the visible rows plus additional data contained in a number of rows above and below the viewport. Since the front-end only needs to render the visible rows, the time it takes to render data is independent from the size of the dataset. Thus, scrolling around the spreadsheet is responsive no matter the size of the dataset.

- Simple controls for manipulating data: The user can easily filter data by column value, sort data based on column values in ascending and descending order, and reorder columns in order to better understand the data. These controls are available on the graphical interface and can be performed in under a second.

Operations available on the GUI

- Edit cell value: This operation corresponds to modifying the value of a cell at a specific position on the spreadsheet. The user can perform this operation by double-clicking on the specific cell that they want to update.

- Add/Remove row(s): This operation corresponds to adding or removing row(s) at a specific location on the spreadsheet. The index of subsequent rows are shifted to account for the addition/removal.

- Filter columns based on cell value: This operation corresponds to applying a filter criteria on the data such that only rows with values in the corresponding column that satisfy the criteria are kept. Rows with values in the filtered column that don't satisfy the filter criteria are removed. The user can perform this operation by clicking on a column's filter button and selecting the filter criteria.

- Sort columns based on cell value: This operation corresponds to sorting data in ascending or descending order by the cell values in a specific column of the spreadsheet. The user can perform this operation by clicking on a specific column's header.

- Reorder columns with drag and drop: This operation corresponds to modifying the order of columns by inserting a column into a new position and shifting any subsequent columns accordingly. The user can perform this operation by dragging a specific column's header to the desired location relative to other columns.

- Clear Filter/Sort(s): This operation corresponds to reverting the data such that the filter/sort(s) are no longer applied. The user can perform these operations by clicking on the corresponding buttons in the spreadsheet toolbar.

Reproducibility Features

- Exported Python code (specifically, Pandas) needed to reproduce changes in the spreadsheet: The spreadsheet operations that are performed are recorded and exported as Python code, which can be rerun on the source dataframe to match the data shown on the spreadsheet.
- Replay spreadsheet operations on a dataframe: Given a dataframe, this feature will take the history of performed spreadsheet operations and apply a series of operations on the dataframe that are logically equivalent to the spreadsheet operations. The resultant dataframe is returned.
- Condense exported Python code: This operation corresponds to removing any operations from the exported Python code that do not affect the final state of the dataframe. The resulting Python code contains a minimal number of spreadsheet operations needed to achieve the final state of the data. Additional condense operations without performing new spreadsheet operations will not change the exported Python code.

The contributions that Modin Spreadsheet innovates on top of Qgrid are the following:

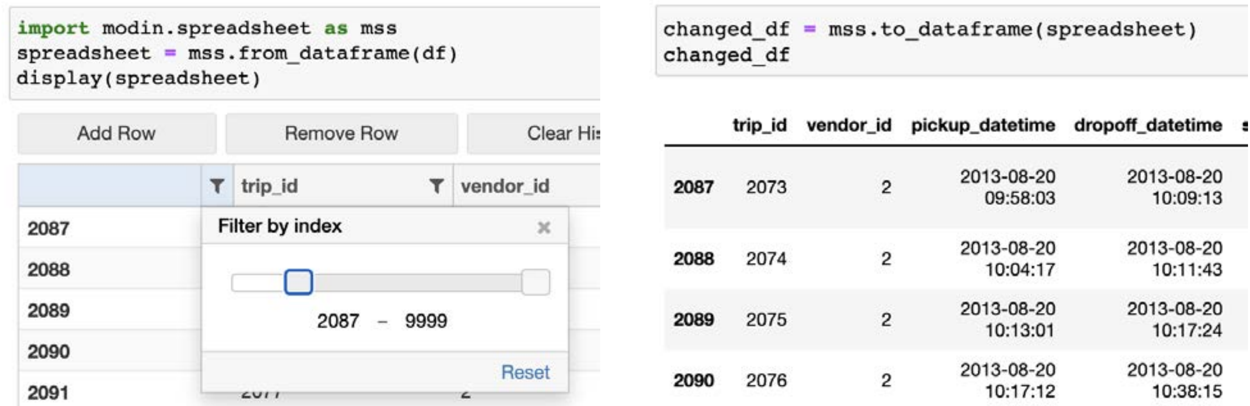
- Adapting the core logic to use Modin dataframes, allowing it to scale better
- Adding new spreadsheet operations - clear filters, clear sorts, reorder columns
- Introducing reproducibility features - exporting changes as code and condensing code

Sections 3.2, 3.3 and Chapter 4 describe these contributions in more detail.

3.1 User Interface

Displaying the spreadsheet UI view is straightforward; Modin Spreadsheet exposes an API that takes in a dataframe and optional arguments for configuring the spreadsheet appearance and additional features [12]. The API returns a spreadsheet widget object which can be displayed using the IPython display module [12].

The spreadsheet widget is generated from the dataframe input to the `from_dataframe` API. The spreadsheet widget stores a copy of the input dataframe and uses the copy to model the spreadsheet data [12]. Changes to the spreadsheet are performed on a copied dataframe so that the input dataframe remains unaltered. Since changes to the spreadsheet do not persist to the input dataframe, the user can retrieve the changed data using the



(a) Example of `from_dataframe` being used to generate the spreadsheet UI. A filter is applied on the index for values within 2087 to 9999 inclusive.

(b) Example of `to_dataframe` being used on the spreadsheet in Figure 3.2a. The resultant dataframe only contains rows with index values within 2087 to 9999 inclusive.

Figure 3.2: Example of using Modin Spreadsheet to convert between a dataframe and the spreadsheet interface.

`to_dataframe` API, which will return a dataframe that is equivalent to the current data in the spreadsheet view.

There exist additional functions on the spreadsheet object, such as for editing cells or changing the grid configuration, but these two functions are the only ones currently exposed on the Modin Spreadsheet API to keep the spreadsheet UI interactions simple.

3.2 Scalability

While Modin Spreadsheet supports more rows than the Excel worksheet row limit of one million [5] and more cells than the Google Sheets worksheet cell limit of five million [6], it runs into usability issues at sizes closer to 1GB. Modin Spreadsheet often creates copies of dataframes and repeats operations because it simplifies the implementation for applying filters. However, frequently creating copies of dataframes and repeating operations unnecessarily increases computation and limits the scalability of Modin Spreadsheet. Work is currently being done to improve Modin Spreadsheet's latency and look at trading implementation simplicity for increased scalability.

3.3 Clearing Operations

The spreadsheet operations that modify the dataframe can be separated into two categories:

1. Clearable operations: Sort, Filter

2. Not-clearable operations: Edit cell, Add/Remove rows, Reorder columns

The distinction between clearable and not-clearable operations is dependent on whether there is a clearing functionality implemented for the specific operation, rather than any fundamental distinction.

To implement a clearing functionality for an operation, Modin Spreadsheet would need to store the state prior to the operation that the spreadsheet can revert back to. For example, a filter may remove rows from the spreadsheet, so the filter cannot be undone without storing the filtered rows and then adding the filtered rows back into the spreadsheet. The spreadsheet UI implements the clearing functionality in general by using two dataframes, a checkpoint dataframe, which contains the state that is unaltered by any clearable operations, and a spreadsheet dataframe, which contains the data the user views. These are shown as the `unfiltered_df` and `df` to the user respectively in the history cell. For the checkpoint dataframe, all not-clearable operations are performed on it, but not clearable operations. For the spreadsheet dataframe, both not-clearable and clearable operations are performed on it. This allows us to clear clearable operations, i.e., filter and sort, using the data stored in the checkpoint dataframe.

The filter is a special operation because it removes rows, so clearing filters would either require adding the differential rows back to the checkpoint dataframe or reassigning the spreadsheet dataframe to a copy of the checkpoint dataframe. Both cases require replaying any active, non-filter clearable operations on the checkpoint dataframe i.e., sorts, so we go with the latter implementation because it is less complex to implement and it is more computationally efficient.

```
# Filter columns
df = unfiltered_df[(unfiltered_df['trip_id'] >= 10197)&(unfiltered_df['vendor_id'] >= 2)].copy()
```

Figure 3.3: Example code of a filter operation.

While there currently only exists clearing operations for filtering and sorting, it is possible to implement them for every operation that the spreadsheet UI current provides. However, is enabling clearing functionality for these operations something the user would want? With regard to how it is done for filtering and sorting, the clearing functionality removes in bulk all filters or all sorts, which may not be as valuable for operations like cell edits. Operations like editing cell values and adding/removing rows are typically meant to be permanent, so enabling the ability to revert all operations of one of these types would be unnecessary. Instead, it makes more sense to offer an undo functionality, which would revert a single operation, to accomplish the goal of reverting changes one by one. The implementation details of undo operations are explored in further detail in Section 6.2.

Chapter 4

Recording Spreadsheet Changes

One key issue of spreadsheets is that reproducibility is complex. In a traditional spreadsheet system, the main method of changing the spreadsheet is through clicking on the graphical user interface to navigate to an action and possibly specifying additional information through text. For example, filtering a column in Google Sheets would require clicking on the column's filter button and then selecting the filter criteria.

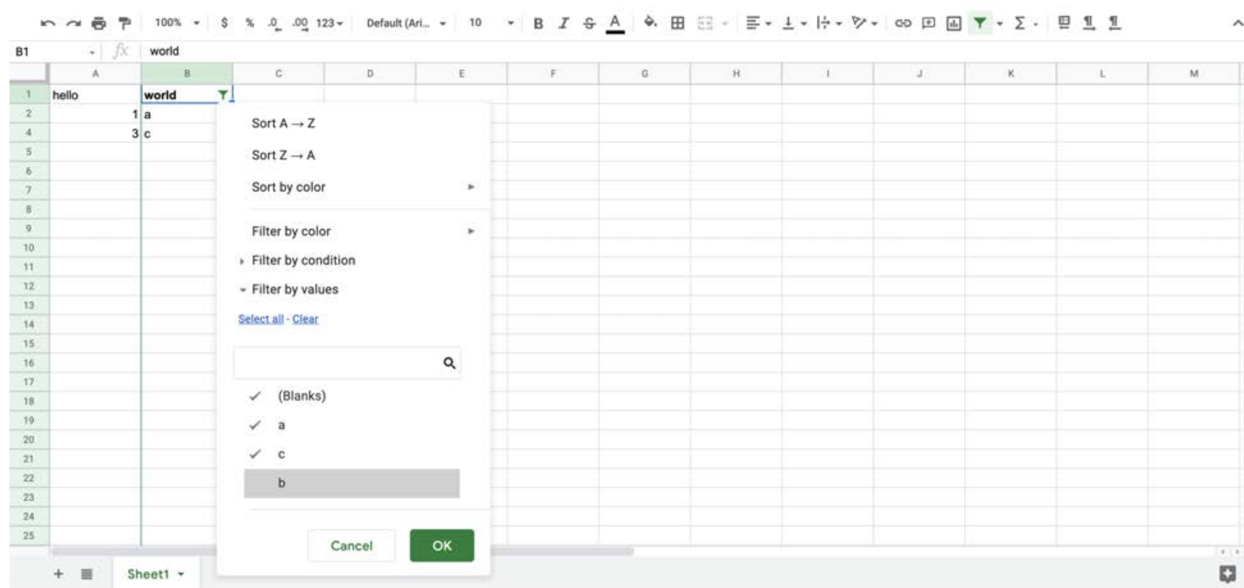
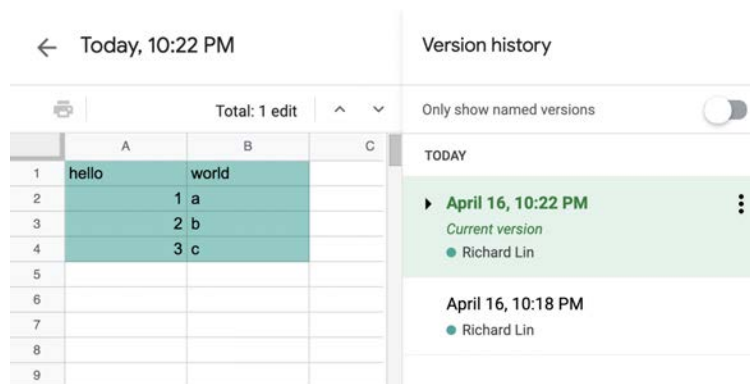


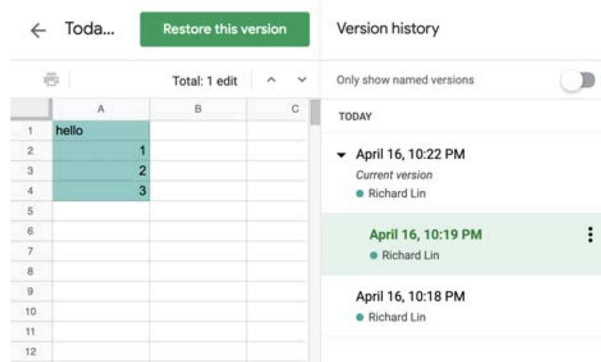
Figure 4.1: Filtering a column in Google Sheets requires clicking on the column's filter button and then selecting the filter criteria.

This process is manual, making it difficult to represent changes in a way that is easily reproduced. Google Sheets tracks changes in its "Version History" by taking snapshots in between two groups of changes and highlighting the changeset. The number of versions

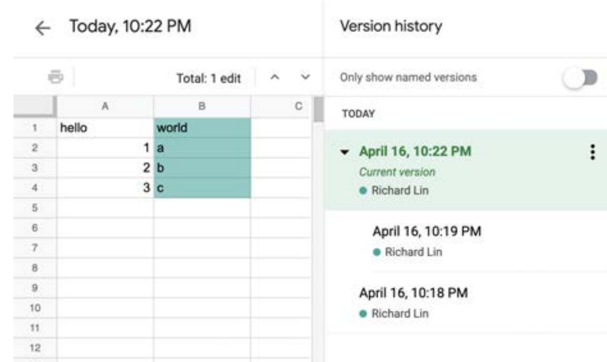
is limited by only taking snapshots whenever there is a significant gap in time between changes or when there is a significant number of changes. Limiting the number of snapshots with these criteria likely helps to reduce the amount of data that needs to be stored and prevents the user from having to search through an excessive number of snapshots, reducing navigational overhead and cognitive strain.



(a) Coarse-grain data snapshot in Google Sheets.



(b) First of two finer-grain snapshots from expanding the snapshot in Figure 4.2a.

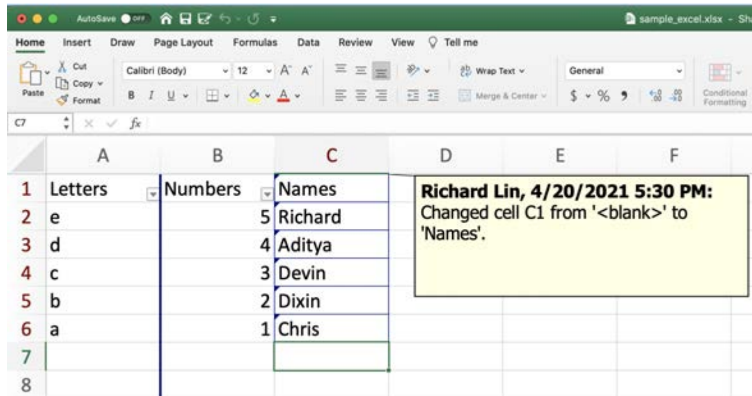


(c) Second of two finer-grain snapshots from expanding the snapshot in Figure 4.2a.

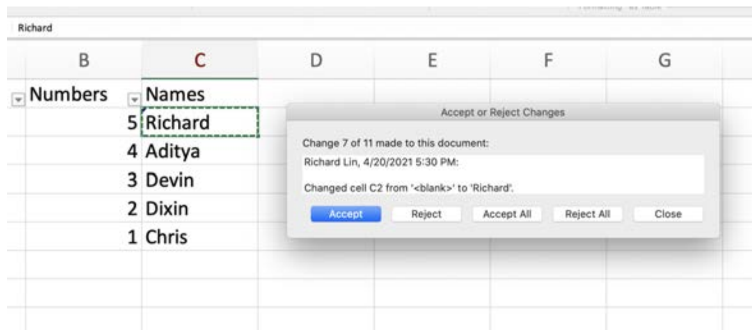
Figure 4.2: Version snapshots from Google Sheets Version History.

As shown in Figure 4.2, Google Sheet's version control lists the snapshots and displays the snapshot that is selected. Edits are grouped under a snapshot, and snapshots can also be grouped into a coarser snapshot. Coarse snapshots can be expanded to see finer grain edits, but are still limited to smaller groups of edits rather than individual edits. For example, the snapshot in Figure 4.2b groups the changes in range A1:A4 as one edit when they were actually edited individually. However, the types of operations performed are not apparent and versions only show the before and after case, making it difficult for someone to follow how changes were made.

Excel provides a more informative tracking feature in addition to an operation history log and the ability to accept or reject changes.



(a) Changes are highlighted on the worksheet and describe the operation.



(b) Users can accept or reject changes.

Action Number	Date	Time	Who	Change	Sheet	Range	New Value	Old Value	Action Type	Losing Action
11	4/20/21	5:37 PM	Richard Lin	Cell Change	Sheet1	B5	'=A5*2	<blank>		
12	4/20/21	5:37 PM	Richard Lin	Cell Change	Sheet1	B6	'=A6*2	<blank>		
13	4/20/21	5:37 PM	Richard Lin	Cell Change	Sheet1	B7	'=A7*2	<blank>		
14	4/20/21	5:40 PM	Richard Lin	Row Insert	Sheet1	'4:4				
15	4/20/21	5:40 PM	Richard Lin	Column Delete	Sheet1	'B:B				
16	4/20/21	5:40 PM	Richard Lin	Worksheet Insert	Sheet2					
18	4/20/21	5:41 PM	Richard Lin	Cell Change	Sheet1	A2	10	1		

The history ends with the changes saved on 4/20/2021 at 5:41 PM.

(c) Changes are compactly recorded in a history log with different details.

Figure 4.3: Examples of Excel features for tracking changes.

While more useful than the Google Sheet's Version History, Excel's changes are linked to a workbook and cannot be exported and applied elsewhere. Excel's history provides a detailed view on operations that were performed, but does not provide a way to programmatically apply these changes elsewhere, which could be useful if you had a similarly structured spreadsheet with different data values that you wanted to transform in an analogous manner.

With the approaches of Excel and Google Sheets in mind, reproducibility seems to be difficult because the representation of changes do not match up with the method of interaction. However, this is not the case with code. With code, the representation of changes can often be the same as the method of interaction because of its linear nature. Each line of code may represent a snapshot of data and sequential lines of code represent a chronological progression of how data is changing. Therefore, code becomes both how we interact with data and the representation of changes in data.

Both Excel and Google Sheets have programmatic interfaces, Excel Visual Basic for Applications and Google Apps Script respectively. However, there are no native solutions for converting changes into code for these interfaces to the best of our knowledge.

When spreadsheets are combined with a coding interface like Jupyter Notebook, the opportunity to both execute using code and represent the changes as code brings an exciting new form of reproducibility to spreadsheets. The Modin Spreadsheet UI operates on an underlying dataframe whenever a user edits the spreadsheet. The spreadsheet logic consists of a mixture of Javascript and Python code. For each spreadsheet operation, there is a corresponding code snippet that is generalized and logically equivalent to the operation, and this code snippet is recorded into the spreadsheet history whenever a change occurs. The code snippet is generalized by populating the snippet with information corresponding to the changes so that running the code doesn't require any additional variable state; in a new notebook, a similar dataframe and the code snippet are all they need to reproduce the changes. This means any mentions of internal Modin Spreadsheet variables or helper functions are removed unless they are also defined in the code snippet.

Although both Javascript and Python code are run during spreadsheet operations, the code snippets only contain Python code because Python is the standard language used for executing code in Jupyter Notebook and for use with Pandas and Modin dataframes.

The user can retrieve the reproducible code via two methods: API or Graphical User Interface. The second method will be described in more detail in the following section.

4.1 Auto-display of Reproducible Code

The synthesis of reproducible code is beneficial, but the method through which the user retrieves this information can make a big difference in streamlining developer productivity. Since we predict that the reproducible code may be a heavily used feature for some users, we implemented the spreadsheet UI such that it would automatically generate a code cell beneath the spreadsheet, referred to as the history cell, and populate the cell with the reproducible code as edits are made to the spreadsheet. This approach would also provide the most interactive interface for seeing and retrieving the reproducible code.

The population of code into the history cell also provides the additional benefit of persisting to the notebook file even when the Jupyter Notebook server is shut down. If the SpreadsheetWidget state were to be erased because the object was overwritten or the notebook kernel was restarted, the history would not be able to be retrieved using an API, but

Operation	**Code Snippet**
Create spreadsheet	unfiltered_df = df.copy()
Sort index	df.sort_index(ascending={sort_ascending}, inplace=True)
Sort column	df.sort_values({sort_field}, ascending={sort_ascending}, inplace=True)
Sort mixed datatype column	df[{helper_col}] = df[{sort_field}].map(str) df.sort_values({helper_col, ascending={sort_ascending}, inplace=True) df.drop(columns={helper_col}, inplace=True)
Clear sort	df.reindex(index={unsorted_index})
Filter columns	df = unfiltered_df[{filter_conditions}].copy()
Clear filter	df = unfiltered_df.copy()
Reorder columns	df.reindex(columns={new_order})
Edit cell	df.loc[{location}]={repr(val_to_set)} unfiltered_df.loc[{location}]={repr(val_to_set)}
Add row	last = df.loc[max(df.index)].copy() df.loc[last.name+1] = last.values unfiltered_df.loc[last.name+1] = last.values
Remove row(s)	df.drop({selected_rows}, inplace=True) unfiltered_df.drop({selected_rows}, inplace=True)

Table 4.1: Mapping between spreadsheet operations and Python code snippet that is output to the history cell. Some variable names are altered for readability. The code inside curly braces is evaluated and converted into string format before being printed. For example, the user would end up seeing `df.drop([0,1,2], inplace=True)` for the remove row(s) operation.

```
# ---- spreadsheet transformation history ----
unfiltered_df = df.copy()
# Filter columns
df = unfiltered_df[(unfiltered_df['trip_id'] >= 9455)].copy()
# Sort column
df.sort_values('pickup_datetime', ascending=True, inplace=True)
# Edit cell
df.loc[(9781, 'trip_id')]=5001
unfiltered_df.loc[(9781, 'trip_id')]=5001
```

Figure 4.4: Screenshot of example Python code that was output in the history cell after several spreadsheet operations.

the permanence of the history cell allows the code to continue to be used independently of the spreadsheet UI. This also allows the code and in effect, the history of your changes, to be shared by sharing the notebook file with someone.

Another benefit of synthesizing reproducible code is that for someone who is not proficient with Pandas, they can perform the spreadsheet operations that they want to understand how to do in Pandas and copy-and-paste the code from the history cell to use elsewhere. This avoids the trial-and-error process of searching through documentation or forums on how to write code for a certain task and potentially helps these people learn Pandas faster for spreadsheet tasks.

4.2 Displaying Multiple Spreadsheets

A SpreadsheetWidget object is generated using the Modin `from_dataframe` API call. Each SpreadsheetWidget object when instantiated will generate a UUID that it uses as metadata for the history cell. When the SpreadsheetWidget is displayed, it will scan all the cells in the Jupyter Notebook to see if there is a cell metadata tag that matches the UUID. If a matching tag is not found, the SpreadsheetWidget will generate a code cell with the UUID in its metadata. Otherwise, the SpreadsheetWidget will do nothing. This will prevent the Spreadsheet UI from creating duplicate history cells, but it also means that if you display the SpreadsheetWidget in two different locations, only one will have a history cell below it. Even though there is only one history cell, changes to either displayed spreadsheet will affect the single history cell. The history cell metadata tag also means that a Jupyter Notebook can have multiple distinct spreadsheets, each with an independent history cell. Any updates to one spreadsheet will not influence the other spreadsheet's history cell because updates are linked by the SpreadsheetWidget's UUID.

4.3 Condensing Reproducible Code

As the user performs spreadsheet operations, the history code will continue to expand until it hits a physical limit and this can cause two problems:

1. The history cell takes up excessive space on the screen, reducing productivity by increasing the time to travel to other cells.

2. The history code is too long such that it becomes difficult to manipulate or understand.

We present some potential suggestions to solve these issues. For problem 1, we can:

- Move the history cell to a designated area such that it doesn't occupy any space between the spreadsheet and cells below it

- Constrain the size of the cell and use scrolling to navigate the cell

- Enable the ability to collapse the code history

Moving the history cell to a sidebar or another area would require significant development for use in Jupyter Notebook, but this may be more easy to do in a system like JupyterLab. Constraining the cell size is practical but may limit the usability to a small extent. Enabling the functionality to collapse the cell is helpful for allowing the user to display the code only when needed, but this could also become an annoyance when expanded. A combination of the second and third suggestions seem to be the most practical until the first suggestion can be implemented. At the moment, none of these suggestions are implemented into the spreadsheet UI.

Instead, we believe problem 2 to be of higher priority because it affects the usability of the reproducible code and the ease of collaboration. We address this problem by condensing the code such that any statements that don't affect the final state of the spreadsheet are removed. Since this process leads to information loss on the changes as they were performed, we provide two modes for displaying the history: All or Condensed. The "All" history mode will show all operations that are performed, and the "Condensed" history mode will remove any lines of code that don't affect the final state of the spreadsheet after every operation. By default, the "Condensed" history mode is shown as we believe users would prefer the minimal set of code for sharing changes or reproducing the operations in code themselves.

```
# ---- spreadsheet transformation history ----
unfiltered_df = df.copy()
# Edit cell
df.loc[(3, 'trip_id')]=150
unfiltered_df.loc[(3, 'trip_id')]=150
# Filter columns
df = unfiltered_df[(unfiltered_df['trip_id'] >= 10197)&(unfiltered_df['pickup_datetime'] >= pd.to_datetime(137816640000
# Sort column
df.sort_values('trip_id', ascending=True, inplace=True)
# Add row
last = df.loc[max(df.index)].copy()
df.loc[last.name+1] = last.values
unfiltered_df.loc[last.name+1] = last.values
# Filter columns
df = unfiltered_df[(unfiltered_df['trip_id'] >= 10197)].copy()
# Sort column
df.sort_values('trip_id', ascending=True, inplace=True)
# Filter columns
df = unfiltered_df[(unfiltered_df['trip_id'] >= 10197)&(unfiltered_df['vendor_id'] >= 2)].copy()
# Sort column
df.sort_values('trip_id', ascending=True, inplace=True)
```

(a) History cell on "All" history mode before condensing.

```
# ---- spreadsheet transformation history ----
unfiltered_df = df.copy()
# Edit cell
df.loc[(3, 'trip_id')]=150
unfiltered_df.loc[(3, 'trip_id')]=150
# Add row
last = df.loc[max(df.index)].copy()
df.loc[last.name+1] = last.values
unfiltered_df.loc[last.name+1] = last.values
# Filter columns
df = unfiltered_df[(unfiltered_df['trip_id'] >= 10197)&(unfiltered_df['vendor_id'] >= 2)].copy()
# Sort column
df.sort_values('trip_id', ascending=True, inplace=True)
```

(b) History cell from top image after condensing the history.

Figure 4.5: Example of reproducible code before and after being condensed.

Condensing Rules

As described in Section 3.3, there are two types of spreadsheet operations in Modin Spreadsheet, clearable and not-clearable. Based on these types, we propose rules for condensing the reproducible code to the minimal set of equivalent code that results in the same changes:

1. Keep all non-clearable operations
2. Keep all clearable operations of its type that occur after the most recent clearing or reassignment to the checkpoint dataframe
3. Keep code needed for setting up helper variables
4. Remove everything else

Each filter operation stores the complete filter criteria across the dataframe and applies it to the checkpoint dataframe. This means that filter operations are independent from each other and we only need to keep the most recent filter operation. As a result of the reverted state, sorts need to be reapplied after the filter operation. This could mean numerous sorts need to be applied again every time the filter criteria changes, leading to an excessive amount of computation. As a performance optimization, we only reapply the most recent sort even though it may no longer result in the same dataframe as one sorted on multiple columns.

Chapter 5

Related Work

We now describe other related tools. While there exists several tools for rendering data as a table in computational notebooks, we focus on Bamboolib [7] and Ipysheet [18] due to their relative popularity and similar capabilities. Qgrid, the most related tool to Modin Spreadsheet, is described in detail in Chapter 2.

5.1 Bamboolib

Bamboolib is a commercial data analysis tool for Jupyter that exposes a spreadsheet widget for pandas dataframes [7]. It describes itself as "a GUI for Pandas" and has an extensive feature set that includes plotting, code exporting, and a number of spreadsheet transformations not in Qgrid. The spreadsheet and plotting interfaces of Bamboolib are shown in Figures 5.1, 5.2. Bamboolib allows users to try the tool for free with the titanic dataset, which is 891 rows and 12 columns. We scaled the dataset to 10 times the length to around 90,000 rows and the plotting tool was laggy and often unresponsive. At 1000 times the length or approximately 900,000 rows, sorting would take almost 20 seconds and other common spreadsheet transformations were also slow.

While Bamboolib provides many features, it doesn't seem to be built for scale. Interestingly, Bamboolib lists Qgrid as a package dependency, which would suggest it uses Qgrid as a foundation for its spreadsheet interface. The Bamboolib product can be found here: bamboolib.8080labs.com [7].

5.2 Ipysheet

Ipysheet is another open-source spreadsheet widget library for Jupyter [18]. The spreadsheet interface of Ipysheet is shown in Figure 5.3. We looked into using Ipysheet as a foundation for Modin Spreadsheet, but we decided against using it after seeing that it doesn't scale well. Some of the major limitations included:

Figure 5.1: Spreadsheet interface of Bamboolib.

- Operations are performed on an internal data structure rather than a dataframe, so it doesn't receive any of the benefits from using Modin.
- It renders each cell as a different widget, resulting in a relatively large overhead.
- It sends all the data in spreadsheet to the frontend interface, making it prohibitively expensive in memory for larger datasets.
- It is less developed and popular.

One advantage Ipysheet has is that it provides an easy API for editing values in a row, column, or cell range. This functionality is something that can be built in Modin Spreadsheet as well. The Ipysheet Github can be found here: github.com/QuantStack/ipysheet [18].

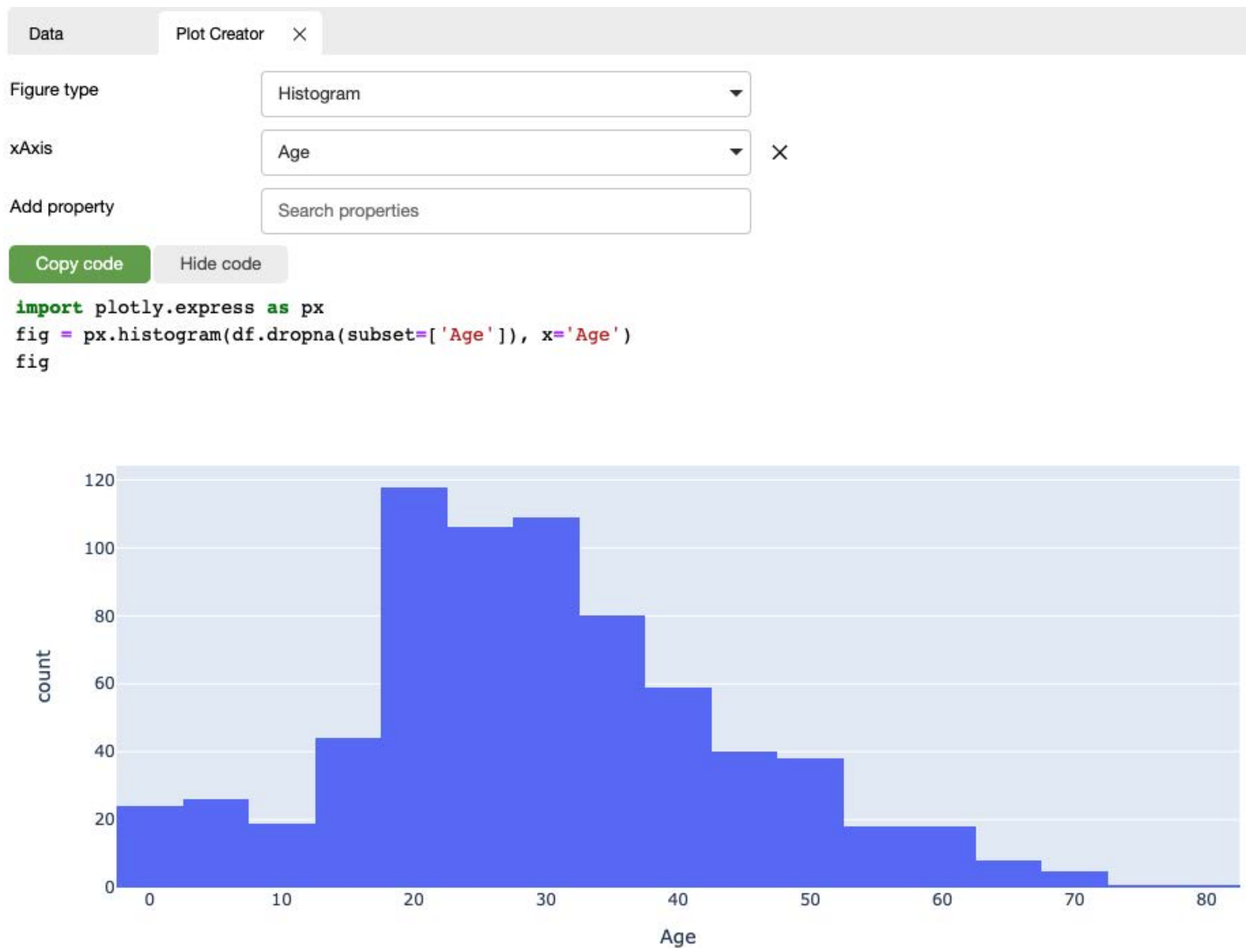
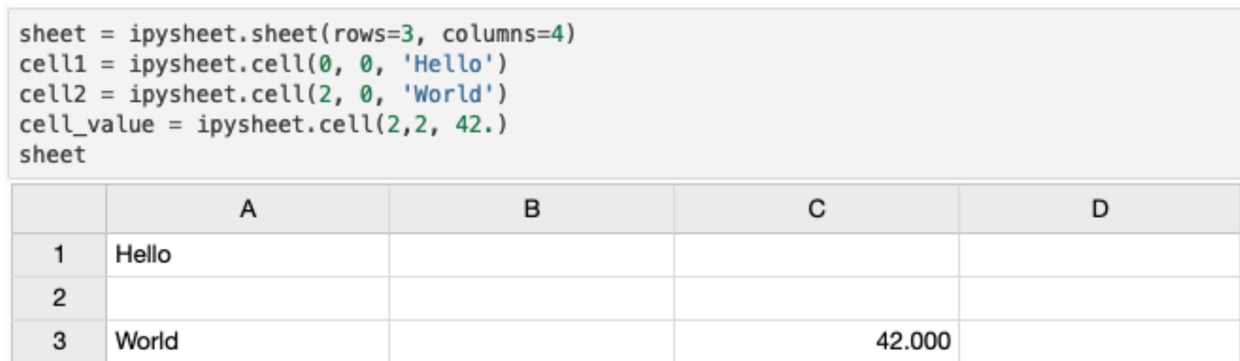


Figure 5.2: Plotting interface of Bamboolib.



```
sheet = ipysheet.sheet(rows=3, columns=4)
cell1 = ipysheet.cell(0, 0, 'Hello')
cell2 = ipysheet.cell(2, 0, 'World')
cell_value = ipysheet.cell(2,2, 42.)
sheet
```

	A	B	C	D
1	Hello			
2				
3	World		42.000	

Figure 5.3: Example of an Ipysheet spreadsheet in Jupyter Notebook.

Chapter 6

Future Work

We now describe major areas of future work for Modin Spreadsheet along with any relevant challenges, implementation details, and impact on usability.

6.1 Formula Computation & Cell References

Spreadsheet formulae allow users to define a cell value based on constants and references to cells. Pandas and Modin Dataframes do not natively support formulae. In Pandas and Modin dataframes, one could reference a cell value in defining another cell value, but the referenced cell value is copied rather than linked and updated. Implementing native support for formulae would simplify a wide range of applications that are more complex to perform with code and provide a no-code method for conducting a large subset of DataFrame operations.

However, the ability to reference other cells and update formula evaluations efficiently is not straightforward. Many spreadsheets today have formulae that contain millions of dependencies. As the number of references in a formula grows, the overhead to update the DataFrame grows with it. One notable difference is the scalability of spreadsheets and DataFrames. Microsoft Excel has a physical limit of 1 million rows and can experience significant performance degradations even at 100,000 rows, whereas Modin DataFrames can handle billions of rows. Research initiatives like DataSpread [2] have made valuable contributions to address the limitations of traditional spreadsheet systems, but addressing the scalability issue of formula computation will require considerable effort before formulae can be fully integrated into dataframes. Questions that will need to be investigated further include:

- How do we present an intuitive interface for spreadsheet formulae while maintaining an easy transition between spreadsheets and code?

- How do we represent formulae in DataFrames such that it is compatible with the Pandas API?

- Does the ability to reference arbitrary cells actually lead to a higher likelihood of user errors? Should we limit formulae to a more constrained form, such as derivative columns, e.g., df['double'] = df['age'] * 2?

6.2 Undo & Redo Operations

The ad-hoc nature of spreadsheets makes it easy to perform a wide range of operations quickly and easily, but this ease also increases the frequency of unintentional operations. One method for fixing a unintentional operation is manually perform a series of operations to revert the state, but this is tedious and not always feasible. The solution used in many spreadsheet software is to expose an undo and redo capability that allows users to revert state backward forward, whenever in a reverted state. This feature is simple, intuitive, and would significantly increase the usability of Modin Spreadsheet.

To revert backward in state, Modin Spreadsheet could either store a copy of the state at each point to reset to, or perform a series of operations that result in the reverted state. For most data sizes, storing a snapshot of the state at each change would be prohibitively expensive in storage costs. The storage cost can be reduced by constraining the number of snapshots. Two potential approaches we propose are: first, store snapshots for each of the last n operations, which would allow the user to undo up to n operations. Second, take a snapshot every x operations and use the history log to replay operations up to the desired point from the most recent valid snapshot. Even then, typical dataset sizes in Modin are large enough that a handful of snapshots would be too expensive to store.

The alternative of performing a series of operations that results in the reverted state is tedious and complex to do manually, which leaves the responsibility on Modin Spreadsheet. Enabling operations to be reversible is more complex to implement than using snapshots because each operation requires an unique implementation, which results in the complexity scaling with the number of available operations. However, this approach would be drastically more space-efficient and likely the most practical for typical Modin dataset sizes. We will reference this approach as performing a reversing operation.

Fortunately, the implementation for a reversing operation would likely be similar to the implementation of the operation being undone, just with different data. Modin Spreadsheet also already stores the history of spreadsheet operations as code, making it easy to perform redo operations. With regards to storage, allowing all operations to be undone would require storing the combination of all previously created cells, all versions of these cells, and every version of the row and column indices. More specifically, the following data would need to be stored for each spreadsheet operation:

- Filter: Old filter conditions
- Sort: Old row index
- Reorder Columns: Old column index

- Edit Cell: Old cell value
- Add Row: Index of the added row
- Remove Rows: Indices and data of the removed rows

The drawbacks to the reversing operation approach are that it is computationally intensive and that operations can only be undone one by one. The respective drawbacks of using reversing operations and using snapshots can be addressed by combining the two. Using occasional snapshots would make it faster and less computationally expensive to revert multiple changes. Using reversing operations helps reduce the number of snapshots that need to be stored. The optimal parameters for when and how to take snapshots will depend on a number of factors that have not been investigated, including typical storage constraints, reversing operation latencies, and common user behavior for reverting state.

6.3 Bidirectional Spreadsheet to Code interface

One unique feature of Modin Spreadsheet is the history cell, which automatically populates with Python code that can be used to reproduce spreadsheet changes. Changes to the history cell currently do not affect the spreadsheet, but enabling a bidirectionality between the spreadsheet and the history cell would make operating on the dataframe even more seamless.

With the current implementation, one needs to export the changed dataframe from the spreadsheet and perform operations on the output dataframe. Although this overhead is significantly smaller than that of exporting to spreadsheet software, removing this barrier would remove the remaining barrier between using the spreadsheet interface and dataframe API. However, enabling users to input arbitrary code into the history cell significantly increases the complexity of its implementation. To break down the complexity, we considered the following implementation details.

How should the history code text be processed?
To initiate the processing, users can press a button on the GUI that tells the spreadsheet to run the history cell code. In order to ensure the spreadsheet is left in a valid state, Modin Spreadsheet would need to check the syntax of the history cell code is correct. The code would be parsed to differentiate between spreadsheet operations and externally added code. Parsing spreadsheet operations allows the corresponding metadata to be changed correctly and enable those operations to be natively undone. Executing spreadsheet operations and the externally added code is described in greater detail below.

How should modifications to previously executed operations be handled?
To perform a new set of spreadsheet operations, executing all the operations on an unaltered dataframe would be the most straightforward. This is computationally expensive and also

requires a copy of the unaltered dataframe. Another option is to revert the spreadsheet to the state before any modified operations by using undo operations and then executing the new set of operations, but this option could also be computationally expensive. A hybrid approach could choose from either option depending on which is estimated to complete faster. We believe the first implementation would be more simple and could be used to test whether users find bidirectionality helpful. The latter implementation or the hybrid approach would be good improvements if the feature was found to be useful.

How should externally additional code be handled?
After being pased, externally added code can be run using Python's 'exec' API. Running arbitrary code on the dataframe might lead to mismatches in the spreadsheet metadata and the dataframe. Resolving this disconnect requires further investigation. One possible solution would be to store all metadata in the dataframe so that the spreadsheet state is fully derived from the dataframe, but this also doesn't seem straightforward.

6.4 Update Slickgrid Version

Qgrid was built using the Slickgrid library developed by MLeibman, which is no longer being maintained. A Slickgrid fork maintained by 6pac has been acknowledged as the current master repository and the most active non-customized fork. Changing Modin Spreadsheet to use the 6pac fork would help it benefit from all the enhancements and updates that have been contributed. The original Slickgrid GitHub repository can be found here: github.com/mleibman/SlickGrid. The 6pac fork can be found here: github.com/6pac/SlickGrid.

6.5 Additional Spreadsheet Operations

Modin Spreadsheet is limited in the number of available data transformations. Here are some transformations that we believe would provide additional usability to the spreadsheet interface:

- Column Removal: This operation corresponds to removing a specific column in the spreadsheet. The user might perform this operation by right-clicking on the specific column header and selecting the "Remove" option on the context menu.
- Column Renaming: This operation corresponds to modifying the name of a specific column in the spreadsheet. The user might perform this operation by right-clicking on the specific column header, selecting the "Rename" option on the context menu, and inputting the desired name.
- Column Summary and Insights: This feature corresponds to displaying an area containing statistics or derived information about data in a specific column. This display could include, but is not limited to, the following information.

  - Unique values in the column
  - Number of missing values e.g., NaN, ‘’
  - Column datatype
  - Frequency chart or histogram of column values

- Change column data type: This operation corresponds to changing the datatype of a specific column in the spreadsheet. For example, a column containing numbers could be converted into a text column to satisfy a type constraint. The user might perform this operation by right-clicking on the specific column header, selecting the “Change datatype” option on the context menu, and selecting the desired datatype.

- Replace values in a column: The operation corresponds to replacing all values in a column that satisfy a specified criteria with a specified value. For example, all NaN values in a column could be replaced with 0. The user might perform this operation by right-clicking on the specific column header, selecting the “Replace value” option on the context menu, and specifying the values to be replaced and the value to replace with.

Chapter 7

Conclusion

In this report, we introduced Modin Spreadsheet, an open-source spreadsheet UI for dataframes implemented on the Modin dataframe system. Modin Spreadsheet integrates spreadsheets and dataframes into a single workflow to address the need for data scientists to switch between the two for various steps of their day-to-day work. This integration increases developer productivity by removing the overhead of switching between spreadsheet software and using dataframes, and exposes a low-code method for manipulating dataframes.

By modeling spreadsheet data as a dataframe, Modin Spreadsheet is able to improve on traditional spreadsheet software in the aspects of interactivity, scalability, and reproducibility. Modin Spreadsheet also brings a new form of reproducibility for spreadsheets by exporting spreadsheet changes as Python code that the user can run to reproduce changes on a dataframe. The code for Modin Spreadsheet can be found on the Modin Spreadsheet GitHub page: github.com/modin-project/modin-spreadsheet.

Bibliography

[1] Eirik Bakke and David R. Karger. "Expressive Query Construction through Direct Manipulation of Nested Relational Results". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1377–1392. ISBN: 9781450335317. DOI: 10.1145/2882903.2915210. URL: https://doi.org/10.1145/2882903.2915210.

[2] Mangesh Bendre et al. "DataSpread: Unifying Databases and Spreadsheets". In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 2000–2003. ISSN: 2150-8097. DOI: 10.14778/2824032.2824121. URL: https://doi.org/10.14778/2824032.2824121.

[3] Mangesh Bendre et al. "Towards a Holistic Integration of Spreadsheets with Databases: A Scalable Storage Engine for Presentational Data Management". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 113–124. DOI: 10.1109/ICDE.2018.00020.

[4] Sarah Butcher. *Howls of pain as quant site 'unexpectedly' shuts down.* Nov. 2020. URL: https://www.efinancialcareers.com/news/2020/11/quantopian-shutdown.

[5] *Excel specifications and limits.* URL: https://support.microsoft.com/en-us/office/excel-specifications-and-limits-1672b34d-7043-467e-8e27-269d656771c3?ui=en-us&rs=en-us&ad=us#ID0EBABAAA=Newer_versions.

[6] *Files you can store in Google Drive.* URL: https://support.google.com/drive/answer/37603.

[7] 8080 Labs. *Python Data Science for Everyone.* URL: https://bamboolib.8080labs.com/.

[8] Bin Liu and H. V. Jagadish. "A Spreadsheet Algebra for a Direct Data Manipulation Query Interface". In: *2009 IEEE 25th International Conference on Data Engineering.* 2009, pp. 417–428. DOI: 10.1109/ICDE.2009.34.

[9] Zhicheng Liu and Jeffrey Heer. "The Effects of Interactive Latency on Exploratory Visual Analysis". In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (2014), pp. 2122–2131. DOI: 10.1109/TVCG.2014.2346452.

[10] Kelly Mack et al. "Characterizing Scalability Issues in Spreadsheet Software Using Online Forums". In: *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI EA '18. Montreal QC, Canada: Association for Computing Machinery, 2018, pp. 1–9. ISBN: 9781450356213. DOI: 10.1145/3170427.3174359. URL: https://doi.org/10.1145/3170427.3174359.

[11] Modin. *Scale your pandas workflow by changing a single line of code*. URL: https://modin.readthedocs.io/en/latest/.

[12] Modin-Project. *modin-project/modin-spreadsheet*. URL: https://github.com/modin-project/modin-spreadsheet.

[13] Pandas-Dev. *pandas-dev/pandas*. URL: https://github.com/pandas-dev/pandas.

[14] Aditya Parameswaran. "Enabling Data Science for the Majority". In: *Proc. VLDB Endow.* 12.12 (Aug. 2019), pp. 2309–2322. ISSN: 2150-8097. DOI: 10.14778/3352063.3352148. URL: https://doi.org/10.14778/3352063.3352148.

[15] Devin Petersohn et al. "Towards Scalable Dataframe Systems". In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2033–2046. ISSN: 2150-8097. DOI: 10.14778/3407790.3407807. URL: https://doi.org/10.14778/3407790.3407807.

[16] Quantopian. July 2018. URL: https://youtu.be/AsJJpgwIX0Q.

[17] Quantopian. *quantopian/qgrid*. URL: https://github.com/quantopian/qgrid.

[18] QuantStack. *QuantStack/ipysheet*. URL: https://github.com/QuantStack/ipysheet.

[19] Sajjadur Rahman et al. "Benchmarking Spreadsheet Systems". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 1589–1599. ISBN: 9781450367356. DOI: 10.1145/3318464.3389782. URL: https://doi.org/10.1145/3318464.3389782.

[20] Vijayshankar Raman, Andy Chou, and Joseph M. Hellerstein. "Scalable Spreadsheets for Interactive Data Analysis". In: *In ACM SIGMOD Wkshp on Research Issues in Data Mining and Knowledge Discovery*. 1999, pp. 1–11.