"Don't Step on My Toes": Resolving Editing Conflicts in Real-Time Collaboration in Computational Notebooks

ANONYMOUS AUTHOR(S)

Real-time collaborative editing in computational notebooks can improve the efficiency of teamwork for data scientists. However, working together through synchronous editing of notebooks introduces new challenges. Data scientists may inadvertently interfere with each others' work by altering the shared codebase and runtime state if they do not set up a social protocol for working together and monitoring their collaborators' progress. In this paper, we propose a real-time collaborative editing model for resolving conflict edits in computational notebooks that introduces three levels of edit protection to help collaborators avoid introducing errors to both the program source code and changes to the runtime state. Through a set of evaluations, we found that these three levels of edit protection make collaborators more satisfied with their collaboration experience compared with notebooks without these features and are perceived to be useful in a variety of collaborative settings.

CCS Concepts: • **Human-centered computing** → **Interactive systems and tools**.

Additional Key Words and Phrases: computational notebooks, data science, synchronous editing, collaboration

ACM Reference Format:

Anonymous Author(s). 2018. "Don't Step on My Toes": Resolving Editing Conflicts in Real-Time Collaboration in Computational Notebooks. In *CSCW '23: The 26th ACM Conference On Computer-Supported Cooperative Work And Social Computing, October 14–18, 2023, Minneapolis, MN, USA.* ACM, New York, NY, USA, 27 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

"You work on *this* section, and I'll work on *that* one" is a familiar refrain for authors who work in teams. Working on different portions of the same document is a natural way to combine collaborators' work while preventing conflicts [30]. In the context of data science programming, collaborators use a variety of collaborative strategies including "divide and conquer" (splitting work between team members), "competitive authoring" (working on the same sub-problem simultaneously), and more [39].

However, computational notebooks like Jupyter, which are often used by data scientists, introduce new challenges for collaboration. Although some version control tools (e.g., Git) can be used for computational notebooks, they mostly support the collaboration strategies for dividing work (e.g., working in separate, interdependent files). Further, data scientists sometimes collaborate *synchronously*, with tools like JupyterLab[1], Google Colab[2], Deepnote[3], and RStudio Cloud[4] that broadcast code and runtime updates to collaborators in real-time [39]. Consequently, dependencies between different authors' code can hinder collaboration, as upstream changes can break downstream dependent code [13, 14]. This includes changes to code that defines variables that will be subsequently referenced or even direct changes to the

[1] https://jupyter.org/
[2] https://colab.research.google.com/
[3] https://deepnote.com/
[4] https://www.rstudio.com/products/cloud/

Manuscript submitted to ACM

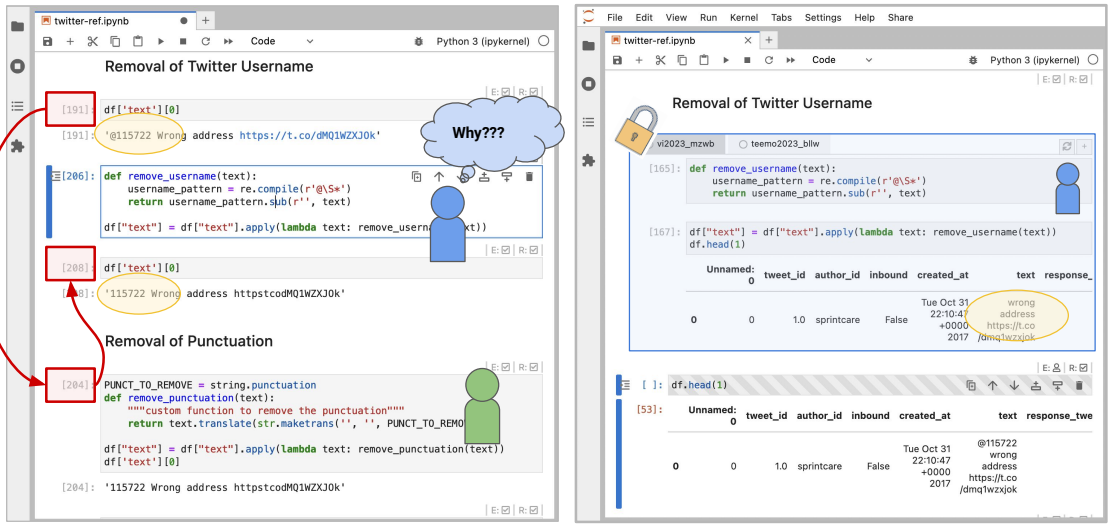

Fig. 1. Editing conflicts in real-time collaborative notebooks can be implicit. As shown on the left, one can get an unexpected execution result because the collaborator accidentally changed the shared variable. As shown on the right, PADLOCK helps data scientists resolve editing conflicts in real-time collaborative editing in computational notebooks.

shared runtime state (as shown in Figure 1). Finally, data science work is often exploratory, so the notebooks produced might be unorganized, fragmented, and poorly documented draft code [33].

Computational notebooks also introduce new opportunities for improving collaboration between data scientists. Synchronized collaborative computational notebooks allow data scientists to see each other's edits as they happen and share documentation, code, and an interpreter runtime state. Real-time collaborative editing improves data science teamwork by creating a shared context, encouraging more explanation, and reducing communication costs [39]. Real-time collaborative editing may also benefit presenting and reproducing results in the workplace or educational settings [21]. However, editing these shared computational narratives together comes with unique challenges [39]. To address these issues, data scientists establish social protocols similar to those found in other collaborative writing situations [28] but contextualized to the computational narrative workflow. For instance, an author may take ownership of a code cell and use formatted Markdown or code comments to indicate to collaborators they don't want others to edit the contents. Depending on the workflow of collaboration, this might result in multiple variants of an analysis which then need to be managed and potentially reduced to a single narrative. For instance, after two data scientists work together on a problem, they may decide to only keep the most efficient solution found and delete one. Alternatively, two competing solutions have unique characteristics which warrant keeping them both. For instance, if different machine-learned models highlight unique aspects of the underlying data, the authors might wish to integrate both into a single narrative.

Inspired by these challenges and opportunities, we propose a set of interactive techniques to minimize collaboration friction while maintaining the readability of the shared notebook. We instantiate these techniques in PADLOCK[5], an extension to the open source JupyterLab platform that provides three different mechanisms to support conflict free collaboration. PADLOCK leverages the context of data science development to provide three domain-relevant mechanisms to improve collaboration on computational narratives. The first, *cell-level access control*, prevents collaborators from

[5]PADLOCK is short for "**P**arallelization **A**nd **D**ata **L**ocks **O**ffset **C**ollaboration **K**inks"

viewing or editing a collection of cells. This mechanism aims to allow ad-hoc locking of code cells to support volatile collaboration patterns. The second mechanism, *variable-level access control*, extends the access control from cell-level to shared variables. This mechanism is designed to prevent implicit editing conflicts and allow collaborators to protect important shared variables. The third, *parallel cell groups*, leverages the familiar programming concept of encapsulation through scoping with the Jupyter cell user interface control. This mechanism allows individuals to pursue exploratory solutions while not having to be concerned about interference with others. Our evaluation of PADLOCK has shown that these mechanisms can effectively prevent editing conflicts in shared notebooks; and they support a wide range of ad-hoc and volatile collaborative workflows.

This work makes several contributions that advance the state of the art for collaborative data science tools:

- We introduce a mechanism (cell-level access control) that can support collaborative data science work by giving authors more control over who can view or edit sensitive cells
- We define variable-level access control mechanisms that give data scientists more control over the runtime state of shared notebooks
- We enable "parallel cell groups", designated areas where data scientists can manipulate and share their own ideas
- A system (PADLOCK) that instantiates all these features in a JupyterLab plugin
- We present three evaluations of PADLOCK (paired sessions, planning sessions, and group sessions) to better understand how these features can be used in collaborative data science.

Further, to the best of our knowledge, PADLOCK is the first system to:

- Give users the ability to specify access control constraints at the level of individual cells in computational notebooks
- Allow programmers to specify which collaborators (as opposed to which code fragments) can access or overwrite specific variables
- Allow data scientists to work in "parallel cell groups", that are scoped in a way where they can access and reference each other's work without worrying about introducing conflicting code

2 BACKGROUND AND RELATED WORK

In this section, we review previous work in version control systems, real-time collaborative programming tools, and real-time collaborative editing tools more broadly, such as used in art and game design. We summarize the features of previous systems that support collaboration and identify the gaps in handling editing conflict in data science collaboration.

2.1 Version Control Systems

One type of tools commonly used in programming collaboration are version control systems. Version control systems track changes on files, allow collaborators to build on top of each other's work, and help collaborators to handle the conflicts between versions [44]. Version control systems are widely used especially in software engineering, where multiple software engineers develop different features in parallel. Popular version control systems include Git [9], Apache Subversion [1], and Mercurial [3]. They are also adopted to support other collaborative tasks, such as collaborative art, game design [6], AR development [36], and Ontology engineering [32].

The workflow when using version control systems for one collaborator usually involves checking out a version from the project into their own working directory, making changes on their copy of the version, and committing and merging the change back to the shared workspace [20, 44]. This use limits the tracking and merging of iterations made by a single developer at a time.

Different from software engineering, programming tasks in data science are often highly exploratory and open-ended [18, 39], including data analysis in academic research [25]. Thus, traditional version control systems are not commonly used in data science programming. According to the design space of computational notebooks formulated by Lau et al. [22], existing design of versioning in notebook ranges from versioning of the complete notebook files, notebook file with dependencies (including data and packages), and cell-level versioning. With the exploratory nature of data science programming in mind, prior studies have proposed lightweight versioning to help data scientists explore alternatives in a more structured way [16, 26]. Some work further explores versioning kernels to provide support for an individual data scientist to store and recover kernel states [5, 42]. However, this work on data science only tracks and merges the code for a single developer to iterate on ideas. Previous work has not been evaluated in real-time collaborative editing scenarios, which involve multiple data scientists editing the same file and working on the same kernel synchronously.

2.2 Real-Time Collaborative Programming

Data science programming can benefit from both synchronous and asynchronous collaboration. Zhang et al. [43] conducted a large-scale survey of data science workers and found that data science work is "extremely collaborative", and the collaboration practices of these workers vary depending on the type of tool they use. Wang et al. [39] found that compared to individual programming contexts, real-time collaboration in computational notebooks can encourage more exploration and provide a shared context for communication. They have proposed four collaboration styles to characterize how data scientists work together, including *single authoring* where one collaborator does the majority of the work, *pair authoring* where one collaborator contributes to the implementation while the other collaborator participates in the discussion, *divide and conquer* where collaborators divide the task into subgoals and assign to each other, and *competitive authoring* where collaborators implement independently toward the same goal.

Researchers have proposed different systems to support programmers in working on the same code file synchronously. Targeting novice programmers, Warner and Guo created CodePilot [41], which is the first real-time collaborative programming tool that embeds coding, testing, bug reporting, and VCS features. Rädle et al. created Codestrates [31] to embed literate computing based on a shareable dynamic media system [19] and enables users to collaboratively work on authoring and debugging. There are also other tools provided in the form of IDE plugins [4, 7]. For example, Microsoft Live Share in VSCode [7] allows users to set the code to read-only for collaborators or enable server sharing for collaborating with the same variables. On a more fine-grained level, some researchers focused on resolving editing conflict of collaborative real-time editing in rich text with Conflict-Free Replicated Data Types (CRDTs) [24]; others examined collaboration in a broader context of peer assessment in programming classes for lightweight test cases [37]. Real-time collaboration in programming also brings unique challenges. Goldman [13] identified that syntax errors introduced by other collaborators might block one programmer's work. To address the issue, Goldman et al. proposed Collabode [14] which uses error-mediated integration that only integrate edits that do not cause compile errors.

Many tools have been released to provide real-time collaboration environments for data scientists. Table 1 compares the collaboration features provided by four tools that are commonly used by data scientists. Although there are many features that aims at improving awareness between collaborators and enhance communication, there has been limited features regarding preventing conflicts or interference in the collaboration process.

	Awareness	Permissions	History	Communication	Others
JupyterLab	Shared cursor	-	-	-	-
Google Colab	Shared cursor; Presence of active users;	Notebook-level	Execution history	Cell commenting	Deprecated [6]
Observable[7]	Shared cursor; Presence of active users	Notebook-level	Editing history	Cell Commenting	Notebook forking
Deepnote	Shared cursor; Presence of active users	Notebook-level	Editing history; Execution history	Cell Commenting	Cell take-over
Hex[8]	Shared cursor; Presence of active users	Notebook-level	Editing history	Cell Commenting	Cell take-over
Databricks Notebooks[9]	-	Notebook-level	Editing history	Cell commenting	-

Table 1. A comparison of collaborative features between popular data science platforms.

2.3 Real-Time Collaborative Editing Tools

Previous research has also explored real-time collaborative editing tools in different aspects. The most common scenario for collaborative editing is collaborative writing. Early in the 1990s, researchers have proposed ShrEdit [29], a shared editor that allows multiple group members to edit a document synchronously. Real-time collaborative editing tools and prototypes that introduce different features to enhance collaboration have been developed ever since, and commercial tools such as Google Docs [2] have become an important part in collaborative workflows. One focus of improving collaborative editing experience is to support collaboration awareness. Lee et al. designed CollabAlly [23], which makes collaboration awareness information accessible to blind users. Aside from writing and programming, real-time collaborative editing tools have also been applied to other contexts, such as map editing system [10], 3D CAD system [27], and phylogenies editing system [8].

Some studies focused on access control and conflicts in the context of real-time collaboration. Targeting real-time collaborative web applications, Gaubatz et al. integrated role-based access control to the system to generate access constrained UI elements for different users [11]. From a different perspective of exploring the positive effect of conflicts, Sun et al. proposed a creative conflict resolution approach to help collaborators to resolve the conflicts and generate alternative solutions [35]. Much previous work on conflicts aims to solve the editing conflicts caused by multiple users attempting to edit some content at the same time [15, 34]. Greenberg and Marwood [15] discussed the choice of concurrency control methods should come from both human and technical considerations. While it provides valuable insight into preventing editing conflicts, it does not apply to the scenario in data science where kernel states can be changed while users are working on different parts of the computational narrative.

3 DESIGN MOTIVATIONS

To motivate our design, we identified three typical real-time collaboration scenarios in data science that would cause conflict, synthesized by the challenges in real-time collaboration by Wang et al. [39].

[9]Currently, Google Colab has stopped supporting RTC. See https://github.com/googlecolab/colabtools/issues/355 for details.
[9]https://observablehq.com/
[9]https://hex.tech/
[9]https://www.databricks.com/product/collaborative-notebooks

3.1 Implementing the Same-Purpose Code at the Same Time

Data scientists would adopt different collaboration styles in their work, including competitive authoring, divide and conquer, single authoring, and pair authoring [39]. Editing conflicts can arise when collaborators are adopting the competitive authoring collaboration style, editing cells that attempt to solve the same problems to explore different alternatives. When multiple people are attempting to edit the same cell, they must be in close collaboration and frequent communication to avoid conflicts. Thus, in competitive authoring scenarios, some people would choose to make copies and only edit their own copy as a way of claiming ownership of some cells. For example, Alice and Bob are experimenting with different ways to pre-process the data. They each created a few cells to finish their explorations. Although they are working on separate cells and "claim" these cells, it does not eliminate the possibility of other users accidentally making edits. Although some commercial platforms such as Deepnote and Hex prevent two users from concurrently editing the same cell, the solution is not perfect - once the "cell owner" stops editing the cell, other users can start editing, even when the owner was only pausing their activity in that cell and would come back shortly after, not expecting changes from other people. In many situations, after each collaborator generates a solution, they would want to save previous exploration results and keep a clean computational narrative. However, these two needs are often in conflict.

3.2 Using or Changing the Same Variable at the Same Time

Even when data scientists are adopting collaboration styles such that the tasks they are working on are different, such as divide and conquer, conflicts can happen when multiple users are using or changing the same variable at the same time. In the second scenario, we demonstrate that when people are running code that affects the same variable at the same time, unexpected conflicts can happen.

Imagine Alice and Bob are doing different analyses on the same variable that contains the dataframe. In the working process, it is easy for collaborators to forget they are sharing the same variable, and start to make edits that are unexpected to other collaborators. When a person changes the variable, it could cause conflict in other people's exploration. In a more extreme situation, they might both give a new variable the same name, unaware that their actions on the variable would influence other collaborators.

Aside from protecting variables from being changed unexpectedly, there are also occasions where data scientists need to sync the changes made by their collaborators working upstream. In this case, Carol might be working on data cleaning that is upstream of Alice and Bob, and they would need to sync the variable from Carol once they finish. In terms of preventing conflict in variable change, we would also need to provide flexibility in syncing from other collaborators.

3.3 Other Needs for Access Control in RTC

The need for access control not only comes from preventing collaborators from unconsciously messing up other collaborator's code or variable, but can also come from other aspects, such as concern for social image or project management.

For example, in a team with different expertise, people may shy away from attempting exploratory work when they are aware that their progress is visible to other collaborators, especially more senior ones. In an educational scenario, novices might be self-conscious about their half-finished code in a collaborative tool being viewed by peers, and might choose to explore their code in a separate environment and paste the results back after they finish; or instructors might want to hide the solution from students before they finish the exploration.

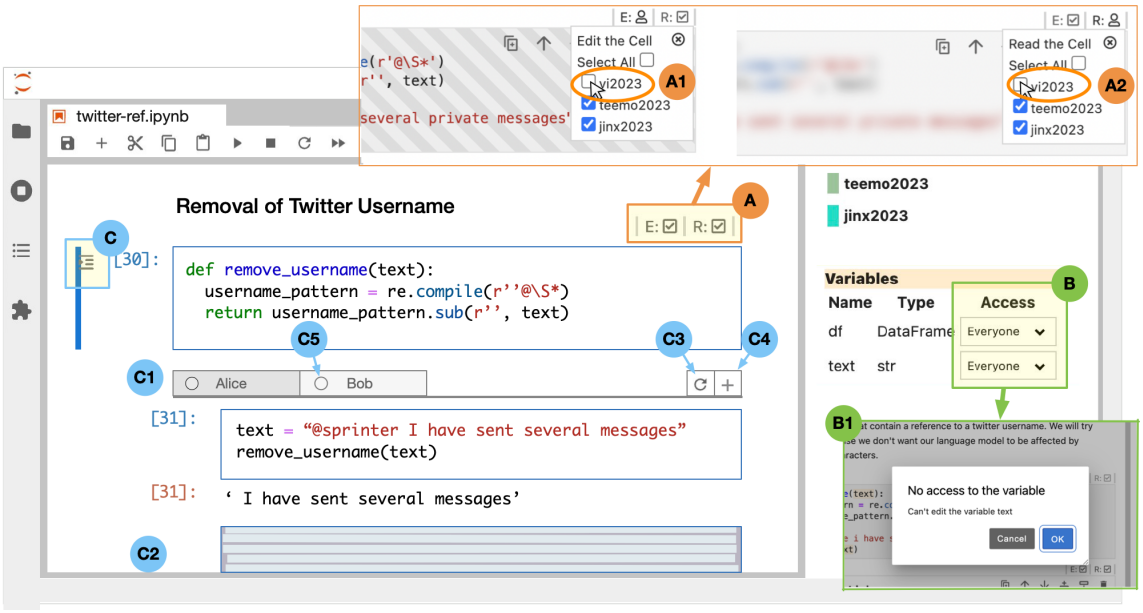

Fig. 2. Overview of the three conflict-free mechanisms in PADLOCK. (A) Cell-level access control allows collaborators to claim ownership of the code cells and restrict others from editing or viewing them: (A1) Unchecking edit access for a user will change the cell background and disable editing; (A2) Unchecking read access for a user will blur the cell. (B) Variable-level access control extends the idea of access control from cells to shared variables: (B1) Unchecking a user's variable access; (B2) After losing access, they will not be able to edit the variable and will receive warning when attempting to do so. (C) Parallel cell groups define a designated area where changes of the code and runtime state stay inside its own scope: (C1) One parallel cell group can contain multiple tabs; (C2) Each tab can contain multiple cells; (C3) Sync the variables from the global scope to the current active tab; (C4) Add a new tab; (C5) Click the radio button to mark it as the "main" tab.

Another reason that calls for access control is from the management perspective. In a larger team with multiple collaborators and a leader, the leader might want to manage the tasks for individuals and give different levels of access. For example, they might want to freeze the code that is ready and does not want any change, such as importing data. Or they might only want a subset of people to edit a certain part of the notebook, such as letting machine learning experts work on training models, and domain experts work on exploratory data analysis to provide more insights. Although many commercial real-time collaborative platforms support notebook-level access control, more fine-grained access control would be appreciated in many scenarios.

4 SYSTEM OVERVIEW

Our system uses three complementary techniques for conflict-free protection: cell-level access control, variable-level access control, and parallel cell groups.

4.1 Cell-Level Access Control

Computational notebooks consist of *cells*. Each cell typically represents a conceptual unit within the larger notebook. For example, a notebook might consist of one cell to fetch data from a remote API, another to clean those data, and other cells for various transformations and visualizations of the data. In PADLOCK, we leverage the structure of cells

in order to allow collaborators to claim ownership of parts of larger collaborative notebooks. This helps address of the challenges of synchronous editing in traditional text-based programming tools where there are no clear "dividing lines" between different parts of the shared codebase and unclear how to localize the scope and effects of collaborators' changes.

Specifically, PADLOCK enables *cell-level access control* where users can prevent collaborators from viewing or editing a collection of cells. As Figure 2.A shows, users can select a code cell and specify who can read or edit the code. As prior studies have found, there are many collaboration styles [39], and cell-level access control benefits multiple collaboration styles. In a "single authoring" style [39]—where one collaborator contributes the majority of ideas and code—setting cells to be only editable by the main contributor can prevent others from accidentally introducing errors. In a "divide and conquer" style [39]—where collaborators split up work—restricting view access might ease feelings of self-consciousness that authors sometimes feel when collaborators can see their writing in real-time (which might otherwise lead them to work in a private editor and then copy its contents to the main notebook, as prior work found in collaborative writing [40]).

When an author is restricted from editing a cell, the background of the cell (grey striping) indicates that edit access is not permitted. When an author is restricted from reading a cell, the content of the cell is blurred but activity (and thus awareness of contributors' location in the narrative) is supported.

Thus, the view control of the cell can allow them to focus on early explorations of ideas while let others aware of where they are working on.

4.2 Variable-Level Access Control

Restricting cell access gives collaborators control of how the runtime state is determined (as determined by the code that computes variables' values). However, it does not prevent other cells from subsequently modifying the runtime state. For example, a user might create a cell that defines `df` as a data frame (data in a table-like structure) and restrict write access to the cell. Other collaborators cannot edit the cell that declares `df`. However, they could create a new cell that either re-declares or mutates the value of `df` and breaks downstream code that references the variable.

Thus, PADLOCK also introduces *variable-level access control*. Variable-level access control extends the idea of access control from cells to shared variables—authors can determine if collaborators' code can view or modify the values of runtime variables. PADLOCK tracks the runtime state of the notebook kernel and extracts the variable information. Users can specify the access control of every variable in a side panel (as shown in Figure 2.B). On the other collaborators' side, the protected variable is highlighted throughout the notebook. When an individual attempts to execute a code cell a static analysis on the abstract syntax tree (AST) of the program is done to determine whether the execution would impact the value of protected variables and, if so, the execution is halted with an error.

Variable-level access control is especially beneficial for scenarios where there is a lead collaborator in charge of managing important data tables. Setting variable-level access control can encourage collaborators to either make a copy or use parallel cell groups before they do any risky explorations.

4.3 Parallel Cell Groups

Data science work is often exploratory. Authors might write code to explore an idea or approach. In the context of teams, multiple team members might simultaneously work through different approaches for the same problem [39]. In

these situations, authors might want to write code that manipulates *their own version of* some subset of variables in the notebook as they explore.

PADLOCK thus also introduces *parallel cell groups* (which we will call "parallel cells"). Parallel cells define a designated area where changes of the code and runtime state stay inside its own scope. As Figure 2.C shows, users can split a regular code cell into parallel cell groups. Collaborators can create new cell groups to branch off and explore alternatives; add multiple cells to a cell group to write larger and more complex alternative code; and work individually in each cell group. The parallel cell groups are folded together into the same area in the notebook, helpings collaborators to maintain an overall coherent structure of the narrative. In addition, when collaborators are settled on a solution, they can mark a cell group as "primary", which merges the execution result into the main runtime state. Note that the parallel cell groups designed in PADLOCK are different notions than the forked cells in [42] in several ways. In terms of the usage scenario, [42] is designed for a single developer to explore alternative ideas, whereas PADLOCK is designed for synchronous computational notebooks. For the implementation, PADLOCK uses a scoping mechanism instead of spawning multiple kernels, making it easier for managing different versions of the same variable.

A key difference from prior tools for branching and managing local versions [16] is that each cell group has its own execution scope, which means changing the variables in cell group A would not affect the value in cell group B. For example, suppose there is a parallel cell group is named `plel`, and within those cells, code creates variables named `x` and `y`. Inside of the cell group, `x` and `y` are defined as normal (referencing them returns the value that they were set to in `plel`). Outside of the cell group, code that references variables `x` and `y` get their 'old' values (or an error if they were not set outside of the cell group). However, these variables can be referenced outside of the group if the user explicitly specifies which scope they want to reference. So while `x` and `y` are not affected outside of `plel`, collaborators can refer to `_plel.x` and `_plel.y` to access the values that were set inside of `plel`.

Parallel cell groups allow collaborators to flexibly split the notebook for exploring alternatives. It is particularly designed for the "competitive authoring" collaboration style [39] where team members competitively write code for the same purpose and reach consensus when an acceptable solution is found. This allows collaborators to work independently while making concurrent edits and executions, preventing costly mismatches between programmers' mental state and the actual state of the runtime. It also provides collaborators with the shared context so they do not work too "far" away from one another, thus supporting awareness of the others' actions (e.g., working on an individual notebook for exploration). Finally, this feature preserves the structure of the narratives by grouping and folding parallel alternatives together.

In the PADLOCK user interface, parallel cell groups are represented as indented cells. Conceptually, this matches the semantic meaning of indentation in Python (specifying the bounds of a code block and potentially creating a new scope), which makes the UI more intuitive and easier to remember for Python programmers.

4.4 Implementation

Although the three features of PADLOCK are conceptually related, the implementation of each feature's implementation is substantially different, as detailed below.

4.4.1 Cell-Level Access Control. Cell-level access control restricts users' ability to view or edit cells and is thus implemented primarily in the JupyterLab UI. In order to store and sync access control specifications (who can edit or view which cells), PADLOCK stores access control permissions in the metadata for each cell. This information is automatically shared in real-time with every collaborator of the notebook through JupyterLab's Conflict-free Replicated

Data Type (CRDT) syncing mechanism. When appropriate, PADLOCK prevents edits by adding a 'read-only' flag to the CodeMirror editor (the underlying Web-based editor used by each JupyterLab cell). When appropriate, PADLOCK prevents viewing a cell by modifying the cell's CSS style to add a "`blur`" gaussian blur filter to the cell element.

One limitation of our implementation of cell-level access control is that it would not prevent technically savvy bad actors from bypassing the PADLOCK UI (for example, using a browser's debugging panel) and violating the notebook's access control specification. However, we believe it is reasonable to assume that collaborators *intend* to act in the best interest of the larger team and thus would not go out of their way to sabotage the larger project. Otherwise, there would be many other attack vectors to address, such as text spamming or executing computationally expensive code.

4.4.2 Variable-Level Access Control. The variable-level access control feature stores and syncs access specifications in the metadata for the notebook (similar to how the cell-level access control specifications are stored). However, unlike cell-level access control, which is implemented primarily as a UI feature, variable-level access control is implemented primarily as a backend feature. PADLOCK implements variable-level value locks by performing a static analysis on each code cell before it is executed. This static analysis searches the Abstract Syntax Tree (AST) of the code to detect if any restricted variables appear in code and what context they appear in (assignment vs. reference). If the code uses the variable in a way that it should not have access to, PADLOCK prevents the entire cell from executing (before any part of the code runs) and throws an error. If the user has permission to execute the cell, PADLOCK performs a post-execution query to track every variable and its value. This allows every variable to appear in the UI for specifying access control rules. After every cell execution, PADLOCK performs a query to track every variable and its value.

As is the case with cell-level access control, a savvy bad actor could bypass PADLOCK's variable-level access control rules by directly inspecting and executing custom code outside of the JupyterLab UI. However, this is acceptable under the assumption that collaborators are trustworthy in their intentions. Although variable locks are currently implemented through static analyses, future work could consider instead using dynamic analysis to avoid only the portions of code that would actually reference or modify restricted variables.

4.4.3 Parallel Cell Groups. Parallel cell groups are implemented by dynamically transforming each cell's code before it is executed. PADLOCK takes a different approach from ForkIt [42], which spawns multiple kernels for its variations. PADLOCK instead executes parallel cell groups within the same kernel, but uses a scoping mechanism that limits the scope of any variables that are declared in the cell. The benefit of PADLOCK's approach is that collaborators can cross-reference variables in other parallel cells (for instance, they can reference the variable `test` in `fragmentA` using `_fragmentA.test`). Cross referencing variables allows users to directly compare different versions of the variables as needed. However, a limitation of PADLOCK's single kernel approach is that computationally expensive code can block the kernel for other users if it takes too long to run.

Through dynamic code execution, PADLOCK converts a parallel cell's code into code that is functionally equivalent but limits the scope of any variables that are declared in the cell. Specifically, PADLOCK defines a `Fragment` class that executes code in a parallel group privately. When a new parallel cell group is created, an instance of `Fragment` is created and the variables in the main notebook are deep copied through pickle serialization. When a user executes a cell, PADLOCK will send the code content of the cell through the `Fragment` execution function. The execution function runs the code using Python dynamic execution (`exec`). We create an execution scope that merges the global scope with the local scope of variables inside the `Fragment` instance. This allows the execution function to use the cloned version of the variables under the scope of the `Fragment` instance. Lastly, PADLOCK updates variables under the scope of the `Fragment` instance to variables from the dynamic execution.

Finally, there are subtle aspects to how JupyterLab executes cells that PADLOCK handles. For example, JupyterLab typically displays the value of the last expression in the cell as the 'output' of the cell. This would be lost in PADLOCK's rewritten `Fragment` instances so PADLOCK re-routes system IO in a way that matches JupyterLab's 'standard' behavior.

To illustrate on a high-level how this works, consider the following example, where the 'main' notebook defines a variable named `foo` and a parallel cell group named `plel` re-assigns `foo` and declares a new variable named `bar`:

```
# main notebook
foo = 123
```

```
# parallel cell group named 'plel'
foo = foo +  5
bar = 'hello'
foo # note: in "standard" Jupyter, this line outputs '128' when the user executes this cell
```

PADLOCK first uses a hidden "cell magic"[10] command (`_privateCell`) to convert the second cell's text (omitting the original comments in subsequent code samples):

```
%%_privateCell plel
foo = foo +  5
bar = 'hello'
foo
```

The `_privateCell` cell magic function further transforms the code to new code that:

- Creates a new `Fragment` instance for this cell group, if it does not exist
- Copies all the global variables into this `Fragment` instance
- Executes a transformed version of the cell code

This is illustrated in the code below. Note that several function calls (`_copyglobal` and `_execute`) are expanded to demonstrate what they do:

```
# STEP 1. create an instance of the _Fragment class if it does not exist
# ======================================================================
if not '_plel' in dir():
    _plel = _Fragment('_plel')
```

```
# STEP 2. deep clone the global variables through pickle
# ======================================================
_plel._copyglobal() # this method does the functions described below (2.1 & 2.2)
#     2.1. get all the global variables by dir(),
```

[10] https://ipython.readthedocs.io/en/stable/interactive/magics.html

```
#          the _filter_var function omits variables with some prefix (default: '_')
_global_vars = _filter_var(dir())               # note: this happens inside _copyglobal()
#     2.2. loop through all the global variables
#          _vars = ['foo']
_plel.foo = pickle.loads(cPickle.dumps(foo, -1)) # note: this happens inside _copyglobal()
#           ... repeat for other global variables in _global_vars
```

```
# STEP 3. execute the code in local scope
# =======================================
_plel._execute(''' # note: this example ignores spacing for clarity
    foo = foo +  5
    bar = 'hello'
    foo
''')
# the _execute method does the functions described below (3.1, 3.2, & 3.3):
#     3.1. Construct an appropriate scope
_global_scope = globals()           # note: this happens inside _execute
_local_scope = {'foo': _plel.foo }  # note: this happens inside _execute
                                    # use the *cloned* version of foo
_merged_scope = dict()              # note: this happens inside _execute
# add in global and local variables:
_merged_scope.update(_global_scope) # note: this happens inside _execute
_merged_scope.update(_local_scope)  # note: this happens inside _execute

#     3.2. Execute a transformed version of the original cell code
#          within the constructed scope using exec()
exec(code, _merged_scope, _merged_scope) # note: this happens inside _execute

#          this is equivalent to executing (through exec):
#          _plel.foo = _plel.foo + 5
#          bar = 'hello' # note that this assigns 'bar' within _merged_scope
#          display(_plel.foo) # A call to display() needed to be added for consistency

#     3.3. Clean up and store variables that were declared.
#          _merged_scope.foo has been updated but _plel.foo has not.
#          (same with .bar) so store results back in _plel
#              _plel.foo = foo # note: this happens inside _execute
#              _plel.bar = bar # note: this happens inside _execute
#              ... merge anything else in _merged_scope
```

5 EVALUATION OVERVIEW

To validate the effectiveness of PADLOCK, we designed a three-stage evaluation—a laboratory study with individual participants working with a paired collaborator, a laboratory study with individual participants planning for various collaboration scenarios, and a case study with groups of participants. In the first stage (paired session), participants joined a laboratory study to work on a structured data science task with designed situations of editing conflicts. After the first stage, participants are optionally allowed to proceed to the second and third stages. In the second stage (planning session), participants were given three hypothetical collaboration setups and work on planning the notebook for these collaborations. In the third stage (group session), participants were paired with each other into groups and worked on open-ended data science tasks. Through this three-stage evaluation, we aim to answer the following question: How do features of PADLOCK assist or hinder real-time collaboration among data scientists?

5.1 Participants

We recruited data science students and alumni from data science programs and interest groups in a university. We required participants to have experience with Jupyter and Python, and preferred participants to have experience using real-time collaborative notebooks. To match the task difficulty, we required participants to be familiar with Pandas, but not necessarily with Regex. As Table 2 shows, the paired session contains 14 participants (3 undergraduate students, 9 graduate students, 2 alumni), where all of them have used real-time collaborative notebooks. 8 participants volunteered to join the additional evaluation for planning notebooks for various collaboration scenarios, while 7 of them signed up for the group session. Based on their time availability for the group session, we assigned them to two groups, where G1 has 4 participants and G2 has 3 participants.

6 PAIRED SESSION: HANDLING SITUATIONS OF EDITING CONFLICTS

We first conducted a laboratory study on handling situations of editing conflicts with a paired collaborator. We are interested in observing how participants use the features in a common conflict editing scenario. This conflict editing scenario is synthesized from literature [39] and reproduced by pairing participants with a member of the research team who plays the role of a "clumsy collaborator".

6.1 Study Setup

The study session was conducted remotely through a video-conferencing application. We deployed the extension on a JupyterHub instance to support multiple users accessing the collaborative editing infrastructure of JupyterLab. In addition, we implemented a basic chat interface for the clumsy collaborator to communicate with the participant. We chose to use text chat instead of voice communication because it was easier for the clumsy collaborator to control how they talked to the participants.

Each session lasted 60 minutes. When participants joined the remote meeting, we first showed them how to use the real-time collaboration feature provided by JupyterLab. We then introduced them to the clumsy collaborator and asked them to greet each other through the chat tool. Participants were informed that not all the study procedures would be explained until the end of the study, and we did not reveal the clumsy collaborator being a member of the research team. The clumsy collaborator would then introduce his background as a data science student who knew Python and regex, but was not experienced in Pandas. Next, we explained the task and the dataset. We divided the task into three sub-goals (noted as T1, T2, and T3), which we will discuss later in the task description. We then asked the participant to

Table 2. We recruited students and alumni from data science programs in our institution. There are 14 participants who finished the paired session, 8 of them chose to participate in the individual session (S2), and 7 of them chose to participate in the group session (S3). Grads. refers to graduate students; Ugrd. refers to undergraduate students.

PID	S2	S3	Bg.	RTC Exp.	PID	S2	S3	Bg.	RTC Exp.
P1	-	-	Grads.	Deepnote	P8	-	-	Grads.	Deepnote
P2	-	-	Alumni	Google Colab	P9	✓	G2	Grads.	Google Colab
P3	✓	G2	Ugrd.	Deepnote	P10	✓	G1	Ugrd.	Deepnote
P4	-	G1	Grads.	Deepnote; Google Colab	P11	-	-	Grads.	Deepnote
P5	-	-	Grads.	Google Colab	P12	✓	G1	Alumni	Deepnote; Google Colab
P6	✓	-	Grads.	Deepnote	P13	✓	-	Grads.	Collaborative JupyterLab
P7	✓	G1	Grads.	Deepnote	P14	✓	G2	Ugrd.	Deepnote

work with the clumsy collaborator to solve T1, where the clumsy collaborator will not disturb the participant's work. This is to get participants familiar with the built-in collaborative editing feature and the clumsy collaborator. Next, we asked the participant to solve T2 without the conflict editing feature; the clumsy collaborator would follow a script to disturb the participant's work. We would observe how participants reacted to the unexpected execution results. After T2, we would conduct a debrief to ask participants what went wrong in the previous session and how they handled it. Followed by a live demo on PADLOCK, we asked the participant to solve T3 with the conflict editing feature; the clumsy collaborator would follow the participants' suggestion to use the notebook. Lastly, we would debrief the whole research process, reveal the study setup about the clumsy collaborator, and discuss the conflict editing features with the participants in a reflective interview.

6.2 Task Description

The task and the dataset were adapted from a Kaggle challenge to preprocess customer support twitter contents. The reference solution on Kaggle contains several steps, including lower casing, removing twitter user name, removing frequent words, etc. We chose lower casing as T1 for warm-up, removing twitter user name as T2, and removing URL as T3. In the notebook, we also inserted sections on removing punctuation and removing frequent words after T3, with code already implemented.

6.3 The Clumsy Collaborator

A member of the research team played the role of clumsy collaborator followed by the following heuristics. First, the collaborator would greet the participant in the chat message at the beginning of the study. For T1, the clumsy collaborator would first ask participants how they want to solve the problem. Then, the clumsy collaborator would tell the participant that she is going to google some API. If the participant got stuck on the task for more than 5 minutes, the collaborator would come back and send a reference code or documentation in the chat (not a direct solution).

For T2, the clumsy collaborator would inform the participant that she would explore the regex in a different cell. Then, if the collaborator got stuck on the task, the collaborator would message an example regex string (not directly how it can be applied to the dataframe) to the participant. Next, the clumsy collaborator would remove the `@` symbol at a reasonable time — before the participant executes the code to remove the Twitter username. This operation would interfere with the participant's action to remove the Twitter username since the `@` symbol was no longer in the tweet to indicate the Twitter username.

For T3, the clumsy collaborator would first ask the participant which cell she should work on. Then, the clumsy collaborator would follow the participant's suggestion to work together. The clumsy collaborator would still pretend to "accidentally" execute the removal punctuation code, which would disturb the regex matching for URLs.

6.4 Data Analysis

For each study session, two members of the research team took notes on how participants responded to the clumsy collaborator in T1, T2, and T3. The research team also used screen recording to reflect on the observations. For the reflective interview, one member of the research team took an inductive approach to identify common feedback and representative comments.

6.5 Result

6.5.1 Conflict editing is hard to notice and prevent. After the second task, most participants (13/14) were not able to correctly find out what caused the code cell not to return the expected results until we explained it to them. This aligned with our observations that many participants (12/14) switched their browsers to search for API documentation and did not stay on the shared notebooks all the time. Moreover, there are several participants (P5, P10) who did not even notice that the output was wrong. There is an exceptional case where P9 ran the data loading cell right before executing the cell for removing the twitter username, leaving no chance for the clumsy collaborator to modify the shared variable. P9 noted,

> Yes, I do prefer to reload the data every time before running a new cell, unless the data frame is very large, in which case it takes a lot of time to load. So then I would avoid doing it, but otherwise yes, I do.

Interestingly, although several participants (4/14) were able to recover from the issue by reloading the dataframe, they still did not find the source of the problem. The majority of participants (12/14) did not doubt their collaborators' actions or question what they did. Instead, they blamed themselves and looked into their own code to debug. For example, P3 said:

> I am familiar with Jupyter Notebook, but I just don't have the confidence... I felt like I had the correct code. But I assumed something was wrong with it. I just didn't even think that it could have been the collaborator's code (that causes the issue).

6.5.2 Perceptions of PADLOCK for preventing conflict editing. After debriefing the second task and walking through the three features of PADLOCK, participants were asked to finish the third task with the clumsy collaborator. All participants (14/14) chose to create parallel cell groups and suggested the clumsy collaborator to write their code in a parallel cell. After they finished the task, some participants (8/14) cleaned up the notebook by unindenting the parallel cell groups. Several participants (2/14) chose to keep the clumsy collaborator's parallel cell and merge their solution into the notebook by marking their solution as main.

Overall, participants reported that they felt confident about not messing up with the shared notebook. For example, P12 reported:

> The parallel cells are very useful. In the case of removing punctuations and removing twitter username, as long as I check the value of `df` when I start the indent, that would be okay.

Participants also mentioned that the parallel cell groups made the shared notebook "neat" (P4), "organized" (P11), and "structured" (P6).

Although participants did not use the cell-level access control and variable-level access control, they described scenarios where these features could be useful. P10 mentioned that both features could be helpful in the large classroom setting, and she recalled an experience of messing up shared notebooks:

> I definitely think the cell-level access control and variable-level access control can be useful in a classroom setting where maybe you have an instructor with a sample notebook. Maybe they'd want to obscure and not have you be able to read like a possible solution that they have, or be able to accidentally edit and screw up some steps that they had put in just to show everyone. Because I remember that happened a couple of times in my class. Last semester students would sometimes accidentally edit the wrong thing, and then the professor would have to backtrack and just make sure his starter code was fixed before we could continue...

P5 said that variable-level access control can be useful when the cost of restarting the kernel and running previous code cells is expensive. He described the scenario where a data science manager would not want interns to accidentally modify large-scale data tables and had to restart the kernel to recover the results. In addition, P10 mentioned that she would use the cell-level access control on finished code cells, and use parallel cell groups on work-in-progress cells. Noticeably, several participants mentioned that read access in cell access control was not necessary for themselves, but they could see it being used by other people. For example, P4 said:

> For blurring the cells, some of my friends are shy so I could see that this would be very useful for them. But I personally would not use it. I think being able to see what your collaborator is doing is a part of that collaboration experience.

6.5.3 Improvement of the Parallel Cell. Participants shared several ideas on improving the parallel cell feature in PADLOCK. Several participants (P6, P9) mentioned adding notifications or activity histories to track if others have unindented a code cell. P9 illustrated this need by comparing the experience with git:

> When you merge the selected tab with the main thread, that's like a commit to the main repository in GitHub. So then, you know, you need to also tell others that I have launched this, maybe a notification. I was hoping there would be some way to track that, like GitHub provides a history of commits that somebody has made to other changes.

In addition, P2 asked for a merging process where she could pull cells from various fragments:

> I wish there is an option to maybe merge different parts of the cell, like maybe one collaborator has one cell, and then you merge the second part of another collaborator.

6.5.4 Resonate with prior experience. The instance of editing conflict in task 2 resonated with participants' prior experience with real-time collaborative editing. P1 mentioned a different collaborative setting in a data science classroom. The data science classroom had around 100 students and the instructor asked everyone to join the same notebook in Deepnote. However, the instructor asked students to not directly run code cells in the notebook. Instead, students typed out solutions and commented at the same time. Several participants (P9, P13) mentioned that their prior experience with shared notebooks was mostly asynchronous collaborating. To further understand how PADLOCK may improve the issues that participants mentioned in their prior experience, we conducted an additional study where we asked participants to plan for a future collaboration scenario, as discussed in the next section.

7 PLANNING SESSION: PLANNING FOR VARIOUS COLLABORATION SCENARIOS

Noticeably, none of the participants used the cell-level access control and variable-level access control features when working with the clumsy collaborator in the paired session, although they have mentioned the potential of using these features in other collaboration setups. To better understand the usefulness of PADLOCK under various collaboration scenarios, we conducted an additional evaluation where we asked participants to plan a future collaboration session by configuring a collaborative notebook. This additional evaluation allowed us to explore the usefulness of the PADLOCK features in different collaboration scenarios. By asking participants to configure a collaborative notebook for planning a future collaboration session, we were able to see how they would use the collaborative features in a more proactive manner. This can provide valuable insights into how users might use the features of PADLOCK in a variety of collaborative environments.

Overall, our additional evaluation showed that the PADLOCK features can be useful in various collaboration scenarios, including working on an asynchronous paired programming session, working on tasks with high cost of error recovery, and using collaborative notebooks as lecture notes. By providing a way to prevent conflicts and improve organization, PADLOCK can help make collaboration more efficient and effective.

7.1 Study Setup

In the individual study sessions, we used the same deployment as in the paired sessions. Based on participants' responses from the previous study, where they mentioned their prior or future use of collaborative notebooks, we synthesized three collaboration scenarios designed to be representative of various realistic situations. For each scenario, we gave participants an initial notebook containing skeleton code and a written description of the collaboration scenario. We asked participants to plan for the collaboration by leveraging the features of PADLOCK and modifying the notebook content as necessary. When they finished planning, we asked participants to verbally describe their plans to the study coordinators, who would then ask a set of semi-structured interview questions to gain further insight into the participants' plans and the features they used or did not use, as well as any potential problems that might arise during the collaboration and any additional features that they felt would be useful. Participants were given as much time as they needed to think about their plans, with most individual sessions lasting around 30 minutes.

7.2 Collaboration Scenarios

In the individual session, participants were asked to plan for the following three collaboration scenarios:

7.2.1 Scenario 1: Asynchronous Collaboration with a Peer. For the first scenario, participants are asked to plan a paired programming session with a friend named Bob who is inexperienced with libraries like Pandas and Numpy, similar to the clumsy collaborator case in the previous study. However, due to conflicting schedules, Bob and the participant are unable to find a synchronous time to work together. This means that both of them may join and leave the session with incomplete code, and the participant cannot guarantee how Bob will use the collaborative editing features without being able to observe and intervene in Bob's actions.

7.2.2 Scenario 2: Working with Trainees and a High Cost of Error Recovery. In this scenario, we asked participants to plan a collaboration session as a full-time employee distributing tasks to interns on visualizing different aspects of the data. We provided participants with a notebook skeleton where the data takes a long time to load. In this scenario, participants needed to avoid the shared dataframe being polluted by any accidental changes, which would be costly to

recover from. To do this, they could leverage the features of the PADLOCK system to ensure that their collaboration was organized and efficient, and that the data was protected from any unintended changes. This scenario simulates the case where error recovery could be costly in a collaboration setting.

7.2.3 Scenario 3: Classroom Sharing with Hierarchical Permissions. In this scenario, we provided participants with an educational notebook on the topic of linear regression, taken from a data science handbook. The first half of the notebook demonstrated the concepts, while the second half contained an exercise for students to practice what they learned. Participants were asked to plan the use of the notebook as an instructor, taking into account the needs of the entire class during a lecture. They needed to ensure that they could effectively explain the concepts to the students, while also providing them with an opportunity to practice individually on the exercise. Additionally, the exercise included a standard solution that the instructor may want to go over with the class after they have explored their own solutions. Participants were asked to plan how they would like to set up the collaborative notebook at the beginning of the lecture, and how they would modify the configurations as the lecture progressed.

7.3 Data Analysis

In each of the above scenarios, participants were asked to configure a shared notebook. We used three metrics to understand and evaluate the effectiveness of the participants' specified notebook configurations. First, we summarize the patterns in how participants set up collaborative notebooks for each scenario. The second data source was a list of potential editing conflicts that may occur in each scenario, which was identified by the research team. We used this list to test participants' configurations against the potential conflicts and analyze their reliability in addressing them. Lastly, we recorded and transcribed participants' post-task reflections, in which they discussed their choices, potential problems, and suggestions for additional features. These data sources gave us a comprehensive view of the effectiveness of the configuration in different collaboration scenarios.

7.4 Results

7.4.1 Usage of the Collaborative Features for Each Scenario. In Table 3, we summarize the strategies that participants described for each scenario and listed the features of PADLOCK that are involved. For the first scenario of working in an asynchronous paired programming session, three participants chose to ask the collaborator to use the parallel cells for exploration, while the other participants chose to use cell-level or variable-level access controls. P12 explained why their strategy changed compared to the clumsy collaborator scenario in the previous paired session:

> I can't really trust that Bob is going to use parallel cells because we are not working together at the same time. I want to set up everything for him so he can only access the things he need[s].

For the second scenario, most participants except P10 chose to lock the shared variable to prevent it from being polluted. Participants suggested that the interns could use parallel cells or make a copy of the dataframe if needed. In addition, most participants (6 out of 8) chose to restrict the editing access of the code cell for loading the data, while two of them also chose to set up the editing access for the code cells that are assigned to each individual intern.

In the last scenario, participants described a mixed strategy for planning the collaborative notebook for a data science classroom. All participants decided to turn off the editing access for code cells in lecture notes, as P14 explained:

> I would turn off the cell editing for the class. Otherwise, if there's so many students, it is easy for someone to accidentally hit a backspace somewhere or something and mess things up.

Session	Action or Strategy	Feature	PID
Paired	Nudge the clumsy collaborator to create parallel cells	Parallel Cell	P1-14
Paired	Merge parallel cells	Parallel Cell	P1, P2, P3, P5, P9, P11, P12, P14
Planning (1)	Lock cells for loading package and data	Cell-Level Access Control	P3, P7, P10, P12, P13, P14
Planning (1)	Lock future cells	Cell-Level Access Control	P6, P10
Planning (1)	Lock the original or a copied dataframe	Variable-Level Access Control	P3, P10, P12, P13, P14
Planning (1)	Create a copy of the dataframe	–	P3, P7, P12
Planning (1)	Ask Bob to use parallel cells	Parallel Cell	P6, P9, P13
Planning (2)	Lock cells for loading data	Cell-Level Access Control	P3, P6, P10, P12, P13, P14
Planning (2)	Change cells' edit access so that interns cannot change each others' code	Cell-Level Access Control	P9, P12
Planning (2)	Lock the shared dataframe	Variable-Level Access Control	P3, P6, P7, P10, P12, P13, P14
Planning (2)	Ask the interns to use the parallel cells	Parallel Cell	P3, P6, P9
Planning (2)	Create a copy of the dataframe if needed	–	P7, P10
Planning (3)	Lock editing access for code cells in lecture notes	Cell-Level Access Control	P3, P6, P7, P9, P10, P12, P13, P14
Planning (3)	Lock reading access for code cells in lecture notes that are not covered yet	Cell-Level Access Control	P6, P10
Planning (3)	Lock reading access for the standard solution code cell in the exercise	Cell-Level Access Control	P3, P6, P9, P10, P12, P14
Planning (3)	Lock access for variables generated in lecture notes	Variable-Level Access Control	P13
Planning (3)	Lock access for the variable for storing results in the exercise	Variable-Level Access Control	P14
Planning (3)	Create parallel cells for students to work on the exercise	Parallel Cell	P3, P7, P9, P10, P12, P13, P14
Planning (3)	Ask students to use parallel cells if they want to explore the code (e.g., change parameters) in lecture notes	Parallel Cell	P9, P12
Group	Create parallel cells	Parallel Cell	P3, P4, P7, P9, P10, P14
Group	Merge parallel cells	Parallel Cell	P4, P7, P9
Group	Sync parallel cells	Parallel Cell	P3

Table 3. Features that participants have used for tasks in each study.

Some participants (P9 and P12) mentioned that they would create a copy of the cell below each code cell for lecture notes and make them into parallel cells, in case students want to explore the code cells in lecture notes. For the same consideration, P13 wanted to restrict the access for variables generated in lecture notes, in case students modify the shared runtime in the lecture. Other participants did not worry about protecting the shared variables, as one participant explained (P6):

> Since the content of the code cell is locked, I can always restart the kernel and run the notebook from the beginning if anything goes wrong.

Scenario	Editing Conflicts	Category	Pass Rate
Scenario 1	Bob dropped all the NA values in the dataset.	Common	0.75
Scenario 1	Bob changed the importing package cell to only include a subset of a package.	Common	0.75
Scenario 1	Bob missed the instruction and started to work on the last code cell.	Rare	0.25
Scenario 2	Intern A changed the shared dataframe.	Common	0.88
Scenario 2	Intern A run the code cell for loading the data frame.	Common	0.88
Scenario 2	Intern A and intern B have the same naming of a variable.	Rare	0.38
Scenario 3	A student edited the code cells for demonstration.	Common	1
Scenario 3	Students directly assigned the prediction results to the shared data frame.	Common	0.88
Scenario 3	Students executed a code cell multiple times.	Rare	0.13

Table 4. For each scenario, the research team solicited three potential cases for editing conflicts and run through participants' notebooks through these cases.

In addition, two participants (P6 and P10) mentioned that they would like to change the code cells' reading access as the lecture progresses so that students can stay focused. For the exercise part, most participants planned to hide the reference solution code (6 out of 8) and ask students to work on parallel cells for their own practice (7 out of 8).

7.4.2 Effectiveness of the Collaboration Plan. For each scenario, we solicited three potential conflicts that may arise during collaboration: two that we expect to be common and one that is less likely to occur. We then evaluated each participant's use of the collaborative notebook to determine if their configuration could effectively address these conflicts. Two members of the research team carefully calibrated and discussed the ratings until they reached a consensus. The results, shown in Table 4, indicate that most participants (more than 6 out of 8) were able to utilize the features in PADLOCK to successfully address common collaboration issues. Even for the less likely problems (e.g., Bob missing the instruction and starting work on the last code cell in the first scenario), a small number of participants (1–3) were still able to successfully handle these rare cases.

7.4.3 Improving Access Control. In the reflective interview, participants mentioned several potential problems and suggested improvements for the collaboration scenarios in the current system design.

First, participants brought up the need for better access control on the notebook level. For example, P13 mentioned that restarting or interrupting the notebook kernel in the third collaboration scenario could cause problems when the instructor is demonstrating concepts. Other types of access control mentioned by participants included preventing collaborators from executing a code cell (P6) or copying and pasting the content from a code cell (P13).

Participants also suggested ways to improve the process of configuring access control. For example, in the first scenario, P6 and P10 mentioned that they would like to restrict cell edit access for their collaborator for every new code cell that they create under a section. One participant also suggested adding a "run and lock all cells and variables above" button (P13) to avoid the need to manually lock all code cells in the lecture notes in the third scenario.

Lastly, participants (P9 and P12) mentioned the potential benefits of combining cell access control with parallel cells. This would be particularly useful in the third scenario, where the instructor may not want students to see each other's solutions in the parallel cell.

8 GROUP SESSION: CASE STUDY ON OPEN-ENDED COLLABORATION

The paired session and the individual session demonstrated the usefulness of PADLOCK in designated collaboration setups. To further understand how PADLOCK would be used in open-ended and less-structured collaboration tasks, we conducted two case studies by observing participants as they worked together in groups on a collaborative task.

Overall, our case studies showed that PADLOCK can be useful in open-ended and less-structured collaboration tasks. By providing a framework for organizing the collaboration, it can help them divide up the work and ensure that each person is working on a specific part of the task. This can help make the collaboration more efficient and reduce the risk of overlap or duplication of effort. Additionally, PADLOCK can help prevent conflicts between collaborators, save time and reduce frustration, leading to a more productive collaboration.

8.1 Study Setup

The group study sessions were also conducted remotely, and used the same deployment as the individual sessions. In the group sessions, participants communicated by talking to each other through the video-conferencing application.

As all participants had already used the extension in individual sessions, we started with a brief reminder about its features. Then, one researcher would explain the collaborative task to the participants, and ask participants to introduce themselves to their collaborators. Next, we explained the open-ended tasks and dataset. In the first group (4 participants), we assigned the participant who is most experienced in data science as the team leader to mitigate collaboration during the task. We did not assign a group leader in group two, which only had 3 participants. Each session lasted 60 minutes in total, and participants were given 40 minutes to complete the task. We would remind the participants to clean up the notebook to provide final results when there were 5 minutes left. Finally, we asked each participant to fill out a post-task survey regarding their collaboration experience.

8.2 Task Description

In the group task, participants were asked to conduct an exploratory analysis of papers accepted by CHI 2022. We provided the participants with two datasets: the first one contains the paper name, link, and author list of full papers; the second one contains the name, link, and affiliations of authors. To make the task more authentic, we collected the raw data by scraping the actual conference program from 2022. Participants were given a list of open-ended tasks regarding ranking authors and institutions by the number of full papers they published. To complete the task, they would need to conduct a series of data pre-processing and cross-referencing between two datasets, and the process is highly open-ended. For example, the format for authors' names in two datasets are different; authors from different institutions might have the same name; an author may have multiple institutions; or even the name of the same institution can be put differently in the system. We made participants aware of the complication in the dataset, but did not provide detailed instructions to encourage open-ended exploration.

8.3 Data Analysis

During each group session, two members of the research team closely monitored collaboration activities and took detailed observational notes. We also recorded participants' screens and captured key activity logs from the Jupyter Notebook to help us understand the collaborative workflow of each group. In addition, we conducted a second pass on the video recordings to identify instances of collaboration challenges and undesired behaviors. This allowed us to carefully analyze the collaboration process and identify areas for improvement.

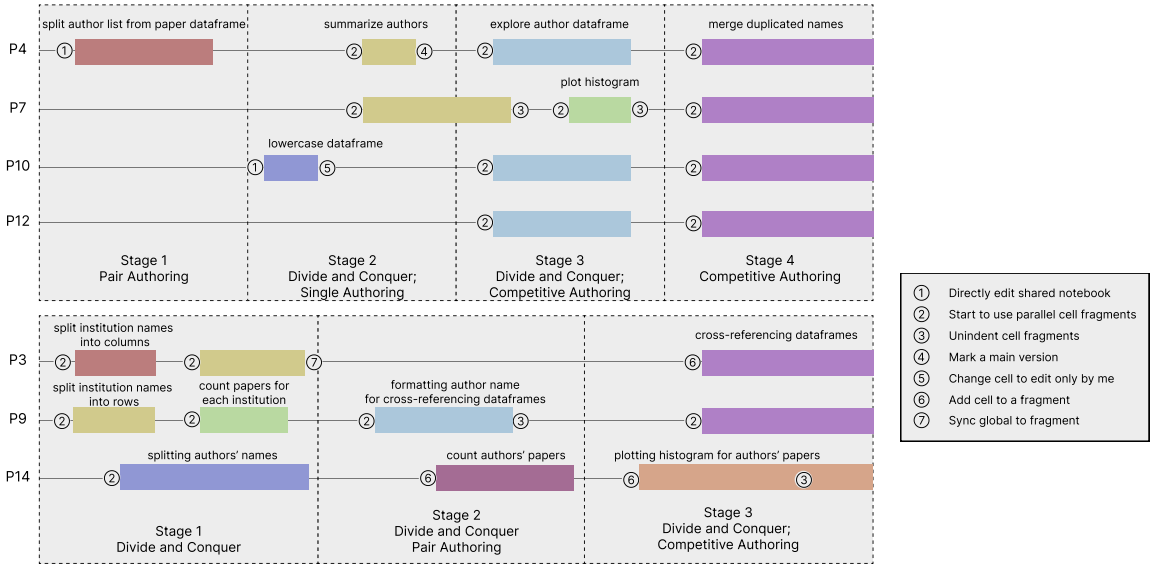

Fig. 3. The collaborative workflow for G1 and G2.

8.4 Result

Overall, we observed different collaboration patterns among the two groups. We report how the two groups leveraged PADLOCK to support their workflow.

8.4.1 G1: Starting from Pair Authoring. We observed that participants in G1 used a mixed of collaboration styles, including pair authoring, divide and conquer, and competitive authoring. With a group leader moderating the discussions, the team naturally started the task by a pair authoring mode. Figure 3 illustrates the collaborative workflow of G1. We broke down their exploration process into four stages.

- **Stage 1:** To begin with, the four participants (P4, P7, P10, P12) discussed the task and decided to split paper authors first. Then, P4 proposed to begin work and they used a pair authoring mode where P4 directly wrote code in the notebook. The rest of the team closely watched him coding and helped him with debugging.
- **Stage 2:** Next, the team leader P12 proposed two tasks – lower casing the authors' names and summarizing authors' paper counts. The team then split into two groups where P10 worked on lower casing, P4 and P7 worked on summarizing the counts, and P12 observed the process. P10 quickly implemented lower casing by directly working on the code cell while the rest of the team is still discussing how to summarize the counts. P10 then turned on the cell editing lock to her finished code cells. In the meanwhile, P4 and P7 created two parallel cell groups and started to work on summarizing the counts in competitive authoring style. After P4 figured out the correct solution, he reported to P12 and they decided to merge his solution into the main notebook. However, noticing that P7 was still working on his solution, P4 chose to mark his cell group as the main version in the parallel groups and moved on to the next task with P12.
- **Stage 3:** Moving on to the next stage, P4, P10, and P12 started to examine the author—affiliations table using parallel cell groups. Then, P7 finished his implementation of summarizing author counts and reported to the team. P7 proposed to unindent the parallel cell group to keep P4's solution. After they merged the cells for

summarizing authors count, P7 proposed to branch off and work on creating a histogram visualization of the authors' paper count while the rest of the team kept exploring the author—affiliations table. After P7 finished the histogram visualization in a parallel cell, he notified the rest of the team and merged his exploration code.

- **Stage 4:** Lastly, all the team members worked together on merging duplicates in institution names since they realized that there was not much time left. They created a parallel cell group with four tabs where everyone worked on their own tab. They were not able to finish this task towards the end of the study.

8.4.2 G2: Starting from Divide and Conquer. G2 started with the divide and conquer style where the three members decided to work on three different tasks — authors' paper count, institutions' paper count, and institutions' paper count without duplicates. The team also discussed and agreed on using parallel cell groups whenever they started to work, and merging them to the main notebook after they verified the results. P3 proposed to lock the shared variables as well. The team decided to stick with the protocol of using parallel cell groups because they agreed that "If we are all working in indented cells, we don't need to lock the dataframes" (P3). We broke down their exploration process into three stages:

- **Stage 1**: The three participants started by indenting three cells and working individually on the three tasks. However, the tasks for P3 and P9 both required splitting institution lists. P3 and P9 took two different approaches for the splitting, where P3 splitting the institutions into columns and P9 splitting the institutions into rows. P9 realized it and started a discussion with P3. In the meanwhile, P14 was working individually on splitting the list of authors for each paper.
- **Stage 2**: In this stage, P3 and P9 turned into pair authoring mode after realizing that their tasks need the same pre-processing. After the discussion, P3 was convinced to split the institution list into rows and she attempted to do it in her previous cell. After she finished, she continued to discuss with P9 and observed P9 coding. P9 first attempted to count institutions' papers. After realizing the need to cross reference two tables, P9 and P3 turned to work on formatting authors' name in the author–affiliations table. At this stage, P9 did most of the programming and P3 actively participated in discussions. On the other hand, P14 continued to work individually on counting authors' paper numbers.
- **Stage 3**: In this stage, P3 and P9 decided to both try to create a cross-reference between the two tables. P3 asked P9 to merge his previous code into the main notebook, and performed a sync to update her parallel cell group. Then, they began to work competitively on cross reference. Meanwhile, P14 continued to work individually on creating a histogram visualization of the authors' papers.

8.4.3 Reflections on the Collaborative Session. By observing two groups attempting the task with different collaboration strategies, we found that PADLOCK was stable and effective in preventing conflict edits while being flexible to support various ad-hoc combinations of collaboration models. We did not observe instances of the abused copying of data frames. We also did not observe instances where participants messed up with the shared notebook and needed to restart the kernel. In the reflection questionnaire, participants commented that "it's a very nice tool for collaboration" (P7); "it helps a lot when we are exploring the dataset and trying to test some functions." (P10). In particular, P9 mentioned:

> This tool is very helpful. It allows us to split the tasks amongst the team, work independently without having to worry about interference, but still be able to discuss problems with each other's solutions if needed.

In addition, participants also reported things that did not work well in the collaboration session. For example, P14 in G2 who spent most of the session working independently on the authors' paper count complained that the

pre-processing approach in her task (e.g., split author list) is similar to her collaborators' tasks (e.g., split institution list). But they did not collaboratively work on the pre-processing:

> It helped us avoid running cell blocks that would influence each other's work and allowed us to work on our own parts. However, it also made us communicate less and made us focus on our own parts when a lot of the questions could've been solved with the majority of the data cleaning work being the same.

9 DISCUSSION AND FUTURE WORK

9.1 Lightweight Collaboration Support for Data Science

Compared to collaboration in traditional software engineering, collaborative data science is more exploratory, ad-hoc, and volatile. Thus, data scientists benefit from real-time collaboration to quickly exchange ideas and plan the next step. Real-time collaborative editing improves data science teamwork by creating a shared context, encouraging more explanation, and reducing communication costs [39]. Although data scientists benefit from writing code that is exploratory, experimental, and messy, they need to be careful about their programming practices when working on a shared notebook. This limits data scientists from harnessing the advantage of computational notebooks as they have to manage the invisible state of the shared kernel through careful practices. We believe that data science collaboration benefits from lightweight and ad-hoc support in computational notebooks. Our evaluation of PADLOCK shows that such a design can effectively help data science teams avoid both implicit and explicit editing conflicts.

9.2 From Small Groups to Collaboration at Scale

Collaborative editing in computational notebooks can benefit not just small team collaboration, but also collaboration at scale. For example, instructors can share a collaborative notebook with a classroom of students; researchers can share a collaborative notebook with a broader audience for open collaboration; data science hobbyists can make their live streaming session more engaged by sharing the collaborative notebook session. Our work suggests exciting design opportunities for supporting collaborative editing at scale. For example, our current design of the parallel cell would horizontally display the parallel tabs as the number grows. Future work can use mechanisms like searching, tagging, and filtering for managing multiple parallel cells. Our current design of cross-referencing allows participants to computationally compare versions of variables from different parallel cells, which could be improved by integrating visualization techniques to compare data changes [38], or clustering techniques to explore variance [12]. Additionally, we are interested in incorporating domain-specific features and needs for large-scale collaborative editing, such as allowing students to test their code cells with shared test cases provided by peers [37].

9.3 Blending Sync and Async Collaboration in Long Terms

One area of interest in our future work on PADLOCK is exploring how the system might be adapted to support different levels of synchronicity in collaborative editing. For example, in asynchronous collaboration, collaborators may not work on the same shared kernel even with a shared notebook. In this case, cell access control may still work to protect the ownership of the code cell; parallel cell groups may be used to improve the readability of the notebook and make it more structured. However, managing access control in a hybrid synchronous-asynchronous collaboration presents unique challenges, such as deciding who should be in charge of setting and enforcing access rules. The current design of PADLOCK allows any collaborators to change the cell-level and variable-level access control. It remains to be explored who should be in charge of managing the access control. Lastly, it is worth exploring how PADLOCK would affect the

presentation of the shared notebook. In a hybrid synchronous and asynchronous collaboration, collaborators may leave staled configuration over a code cell or shared variable, making it difficult for others to understand the context or history of the notebook.

9.4 Improving Awareness of Collaborators' Activities

PADLOCK focused on the perspective of editing conflicts in real-time collaborative computational notebooks. We believe that effective collaboration can also benefit from combining conflict-free mechanisms with awareness design. For example, the use of parallel cell groups allows multiple users to work on different versions of the same document concurrently, but makes it difficult for users to see each other's cursor movements or edits. By highlighting the active tab for each collaborator, PADLOCK can improve awareness and help users understand who is working on what. Similarly, notifications can alert users when others make changes that affect their work, such as unindenting a cell group or removing tabs. In addition, the design of PADLOCK brings up the unique challenges in helping collaborators track and forage editing history. With the notebook structure being not linear anymore, it is worth exploring how notebook history foraging designs [17] can be extended to support the awareness of complex cell editing.

9.5 Limitations

9.5.1 Limitations of the Evaluation. It is worth noting that the results of the evaluation may not be generalizable to all collaboration scenarios. The specific tasks in the three study sessions of working with a clumsy collaborator, configuring a collaborative notebook for planning a future collaboration session, and working in small groups to solve an exploratory analysis task together, may not be representative of all collaboration scenarios, and further research would be needed to explore the usefulness of PADLOCK in a wider range of collaboration contexts. Additionally, the results of this evaluation may have been influenced by other factors, such as the participants' individual preferences and experiences with collaborative tools. As such, it is important to interpret the results of this evaluation with these limitations in mind.

9.5.2 Limitations of PADLOCK. While PADLOCK is designed to address editing conflicts and support flexible explorations between small-size teams during synchronous editing sessions, it may be limited in its ability to support the needs of large-scale and long-term collaborations. For example, PADLOCK may not provide the necessary tools and features for comparing variance across multiple parallel cells, or maintaining the readability of notebooks and keeping the access controls updated over long periods of use. However, we are interested in understanding how features of PADLOCK are effective for large-scale and long-term usage, as well as ways to extend and adapt PADLOCK to better support these types of collaborative work in the future.

10 CONCLUSION

Real-time collaborative editing in computational notebooks requires strategic coordination between collaborators. We investigated common obstacles in real-time notebook editing and proposed a set of access control mechanisms to support conflict-free editing: cell-level access control (which restricts collaborators' ability to see or edit cells), variable-level access control (which protects runtime variables from being referenced or modified, and parallel cell groups (which allow collaborators to work in their own space while staying connected to the larger notebook). As we found in our user studies with PADLOCK, these features can improve collaboration within data science teams.

REFERENCES

[1] 2022. Apache® Subversion®. https://subversion.apache.org/

[2] 2022. Google Docs: Online Document Editor | Google Workspace. https://www.google.com/docs/about/

[3] 2022. mercurial. https://www.mercurial-scm.org/

[4] 2022. Saros. https://www.saros-project.org/

[5] 2022. Scratchpad Notebook Extension. https://jupyter-contrib-nbextensions.readthedocs.io/en/latest/nbextensions/scratchpad/README.html

[6] 2022. Source Control | Unreal Engine 4.27 Documentation. https://docs.unrealengine.com/4.27/en-US/Basics/SourceControl/

[7] 2022. Use Microsoft Live Share to collaborate with Visual Studio Code. https://code.visualstudio.com/learn/collaboration/live-share

[8] Floréal Cabanettes, Jean-François Dufayard, Vincent Berry, Vincent Lefort, Frederic de Lamotte Guéry, and Anne-Muriel Arigon Chifolleau. 2018. CompPhy2: a flexible and real time collaborative web platform for editing and comparing phylogenies. *Bioinformatics* (2018), 1–3.

[9] Scott Chacon and Ben Straub. 2014. *Pro git*. Apress.

[10] Thore Fechner, Dennis Wilhelm, and Christian Kray. 2015. Ethermap: Real-Time Collaborative Map Editing. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*. Association for Computing Machinery, New York, NY, USA, 3583–3592. https://doi.org/10.1145/2702123.2702536

[11] Patrick Gaubatz, Waldemar Hummer, Uwe Zdun, and Mark Strembeck. 2013. Supporting customized views for enforcing access control constraints in real-time collaborative web applications. In *International Conference on Web Engineering*. Springer, 201–215.

[12] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 1–35.

[13] Max Goldman et al. 2012. *Software development with real-time collaborative editing*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[14] Max Goldman, Greg Little, and Robert C Miller. 2011. Real-time collaborative coding in a web IDE. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. 155–164.

[15] Saul Greenberg and David Marwood. 1994. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*. 207–217.

[16] Mary Beth Kery, Amber Horvath, and Brad A Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists.. In *CHI*, Vol. 10. 3025453–3025626.

[17] Mary Beth Kery, Bonnie E John, Patrick O'Flaherty, Amber Horvath, and Brad A Myers. 2019. Towards effective foraging by data scientists to find past analysis choices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–13.

[18] Mary Beth Kery and Brad A Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 25–29.

[19] Clemens N Klokmose, James R Eagan, Siemen Baader, Wendy Mackay, and Michel Beaudouin-Lafon. 2015. Webstrates: shareable dynamic media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 280–290.

[20] Ali Koc and Abdullah Uz Tansel. 2011. A survey of version control systems. *ICEME 2011* (2011).

[21] Sean Kross and Philip J Guo. 2019. Practitioners teaching data science in industry and academia: Expectations, workflows, and challenges. In *Proceedings of the 2019 CHI conference on human factors in computing systems*. 1–14.

[22] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. 2020. The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–11. https://doi.org/10.1109/VL/HCC50065.2020.9127201

[23] Cheuk Yin Phipson Lee, Zhuohao Zhang, Jaylin Herskovitz, JooYoung Seo, and Anhong Guo. 2022. CollabAlly: Accessible Collaboration Awareness in Document Editing. In *CHI Conference on Human Factors in Computing Systems*. 1–17.

[24] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. 2022. Peritext: A CRDT for Collaborative Rich Text Editing. *Proceedings of the ACM on Human-Computer Interaction* 6, CSCW2 (2022), 1–36.

[25] Yang Liu, Tim Althoff, and Jeffrey Heer. 2020. Paths Explored, Paths Omitted, Paths Obscured: Decision Points amp; Selective Reporting in End-to-End Data Analysis. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3313831.3376533

[26] Hiroaki Mikami, Daisuke Sakamoto, and Takeo Igarashi. 2017. Micro-versioning tool to support experimentation in exploratory programming. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 6208–6219.

[27] Tek-Jin Nam and David Wright. 2001. The development and evaluation of Syco3D: a real-time collaborative 3D CAD system. *Design Studies* 22, 6 (2001), 557–582.

[28] Sylvie Noël and Jean-Marc Robert. 2004. Empirical study on collaborative writing: What do co-authors do, use, and like? *Computer Supported Cooperative Work (CSCW)* 13, 1 (2004), 63–89.

[29] Judith S Olson, Gary M Olson, Marianne Storrøsten, and Mark Carter. 1993. Groupwork close up: A comparison of the group design process with and without a simple group editor. *ACM Transactions on Information Systems (TOIS)* 11, 4 (1993), 321–348.

[30] Ilona R Posner and Ronald M Baecker. 1992. How people write together (groupware). In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, Vol. 4. IEEE, 127–138.

[31] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R. Eagan, and Clemens N. Klokmose. 2017. Codestrates: Literate Computing with Webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 715–725. https://doi.org/10.1145/3126594.3126642

[32] Timothy Redmond, Michael Smith, Nick Drummond, and Tania Tudorache. 2008. Managing Change: An Ontology Version Control System.. In *OWLED*.

[33] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

[34] David Sun, Chengzheng Sun, Agustina Ng, and Weiwei Cai. 2020. Real Differences between OT and CRDT in Correctness and Complexity for Consistency Maintenance in Co-Editors. *Proc. ACM Hum.-Comput. Interact.* 4, CSCW1, Article 21 (may 2020), 30 pages. https://doi.org/10.1145/3392825

[35] David Sun, Chengzheng Sun, Steven Xia, and Haifeng Shen. 2012. Creative Conflict Resolution in Realtime Collaborative Editing Systems. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work* (Seattle, Washington, USA) *(CSCW '12)*. Association for Computing Machinery, New York, NY, USA, 1411–1420. https://doi.org/10.1145/2145204.2145413

[36] Ana Villanueva, Zhengzhe Zhu, Ziyi Liu, Kylie Peppler, Thomas Redick, and Karthik Ramani. 2020. Meta-AR-app: an authoring platform for collaborative augmented reality in STEM classrooms. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–14.

[37] April Yi Wang, Yan Chen, John Joon Young Chung, Christopher Brooks, and Steve Oney. 2021. PuzzleMe: Leveraging Peer Assessment for In-Class Programming Exercises. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW2, Article 415 (oct 2021), 24 pages. https://doi.org/10.1145/3479559

[38] April Yi Wang, Will Epperson, Robert A DeLine, and Steven M Drucker. 2022. Diff in the Loop: Supporting Data Comparison in Exploratory Data Analysis. In *CHI Conference on Human Factors in Computing Systems*. 1–10.

[39] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. 2019. How data scientists use computational notebooks for real-time collaboration. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–30.

[40] Dakuo Wang, Haodan Tan, and Tun Lu. 2017. Why users do not want to write together when they are writing together: Users' rationales for today's collaborative writing practices. *Proceedings of the ACM on Human-Computer Interaction* 1, CSCW (2017), 1–18.

[41] Jeremy Warner and Philip J Guo. 2017. Codepilot: Scaffolding end-to-end collaborative software development for novice programmers. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 1136–1141.

[42] Nathaniel Weinman, Steven M Drucker, Titus Barik, and Robert DeLine. 2021. Fork It: Supporting stateful alternatives in computational notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–12.

[43] Amy X. Zhang, Michael Muller, and Dakuo Wang. 2020. How Do Data Science Workers Collaborate? Roles, Workflows, and Tools. *Proc. ACM Hum.-Comput. Interact.* 4, CSCW1, Article 22 (may 2020), 23 pages. https://doi.org/10.1145/3392826

[44] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. 2018. Version control system: A review. *Procedia Computer Science* 135 (2018), 408–415.