University of Toronto

CSCD01H3 Engineering Large Software System

Deliverable #4: Pandas Open Source Contribution

Final Version

Winter 2023

Instructor: Prof. Rawad Abou Assi

Group Name: Timbits

Group Members:

Xuen Shen

Xu Zheng

Lingfeng Su

Yawen Zhang

Megan Mujia Liu

Shaopeng Lin

Runyu Yue

April.06, 2023

Issues

Issue 1: [ENH: Allow easy selection of ordered/unordered categorical columns #46941]

Member Contribution

Yawen Zhang:

Completed the implementation for functions to check whether a pandas dtype is ordered.
Composed the user guide for the two functions.

Xuen Shen:

Implemented categories property to the accessor.
Added unit and acceptance tests to the new pandas dtype functionality and categories property.

Shaopeng Lin:

Designed and participated in implementing the basic DataFrame level categorical accessor. Added the basic set of unit and acceptance tests. Created the accessor portion of the user guide.

Implementation Description and Reasoning

Issue Description

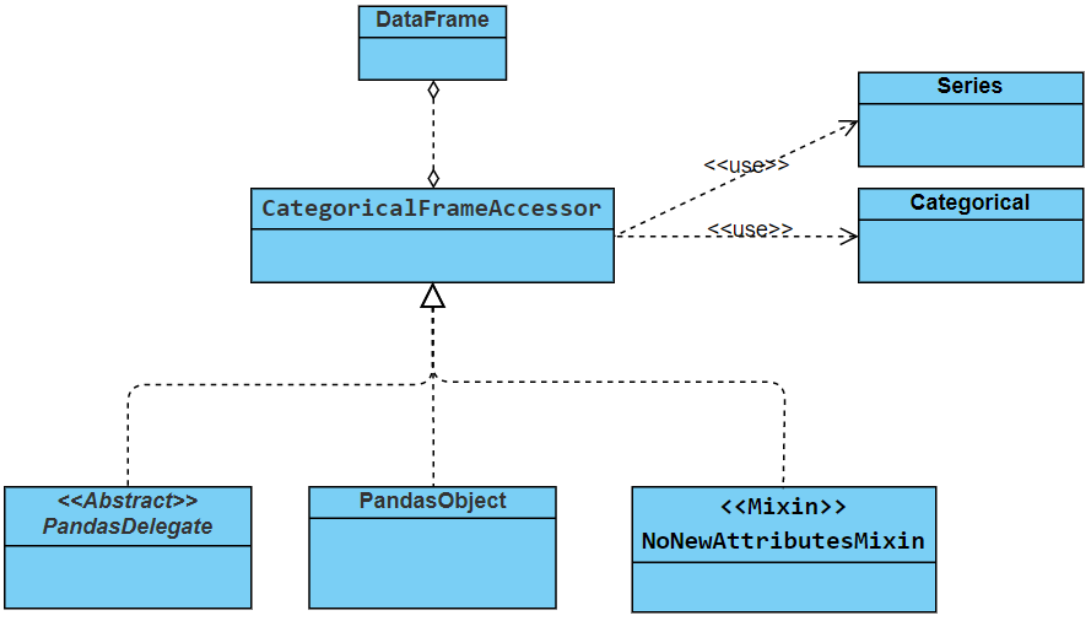
The issue is an enhancement of the complexity of the extraction of ordered and unordered categorical columns in a DataFrame. We need a cleaner interface for DataFrame for users to easily select only ordered or unordered categorical columns.
As you can see below, the current interface is not as abstracted, and every time a user need to do this operation they will have to write these two long lines of code.

```
categories = people.select_dtypes("category")
categories[[col for col in categories.columns if categories[col].cat.ordered]]
```

Implementation and Reasoning

We implemented the CategoricalFrameAccessor as a DataFrame level category accessor similar to the one in Series. The accessor contains similar methods as the one in Series and in addition, added all() and unordered() in response to the issue requirement.
First, this implementation resolves the issue with unordered and ordered as @property of the accessor class. The user can call *df.cat.ordered* or *df.cat.unordered* to retrieve only ordered or

unordered columns in a DataFrame which is a significantly shorter snippet compared to the original solution.
In addition, to ensure consistency in the interface, we delegate the methods in the Categorical class and apply them to each column of the DataFrame.
This design does not break any existing functionalities in Pandas as the accessor and type functions are new features and are used nowhere in the source code. The accessor might break libraries that extend pandas, fortunately, we did not find any conflict in the extensions listed in Pandas documentation.



Major Design Pattern
The accessor incorporates a Delegation Pattern or as we have seen in the lecture, replaces inheritance with delegation. The categorical accessor creates a namespace/property in DataFrame to prevent conflicting method names and delegates the operation on categorical columns to the Categorical class. This is an existing pattern in Pandas often referred to as an accessor to decouple the responsibility of DataFrame from knowing how to handle its categorical columns. @delegate_names is used to add the set of methods delegated to the df.cat property.

Since Categorical class in nature provides a one-dimensional interface, when we created the accessor, we intuitively apply the delegated methods in a column-wise fashion. Methods such as remove_unused_categories are direct beneficiaries of this functionality as users no longer need to write manual loops. If users want to have the function applied on a more refined subset,

they can always use filter() on the DataFrame. We feel this is an intuitive addition to ensure consistency in the interface.

In addition, we added the is_ordered_categorical_dtype and is_unordered_categorical_dtype for both as a helper function in the accessor and also as a more flexible option for the users. DataFrame column selections usually use boolean arrays by & or | to apply filters. The user now can easily create a boolean array of the ordered or unordered columns by using apply() function on DataFrame with the type checking functions `df.apply(is_ordered_categorical_dtype)`.

Detailed Changes

A new accessor class is created in ***\core\arrays\categorical.py*** to handle the common operations on categorical types, including the new functionalities of order/unordered column extractions.
The delegated class is added to DataFrame via the CachedAccessor using namespace ‘cat’.

```
    # ------------------------------------------------------------------------
    # Add extra accessors to DataFrame
    cat = CachedAccessor("cat", CategoricalFrameAccessor)
```

```
@delegate_names(
    delegate=Categorical,
    accessors=[
        "rename_categories",
        "reorder_categories",
        "add_categories",
        "remove_categories",
        "remove_unused_categories",
        "set_categories",
        "as_ordered",
        "as_unordered",
    ],
    typ="method",
)
class CategoricalFrameAccessor(PandasDelegate, PandasObject, NoNewAttributesMixin):
```

The extraction methods are defined as @property in order to be a cached result. The all() property is added as a convenience for both user and the internal implementation.
__check_ordered is used to reduce duplicates in ordered and unordered as they only differ by a type checking function.

```python
    @property
    def all(self) -> DataFrame:
        """
        Return all categorical columns in the Dataframe
        """
        return self._parent.select_dtypes("category")

    def __check_ordered(self, is_ordered: bool=False) -> DataFrame:
        """
        Helper function for ordered and unordered to avoid duplication
        """
        all_cat_cols = self.all
        order_cond = is_ordered_categorical_dtype if is_ordered else is_unordered_categorical_dtype
        return all_cat_cols[[col for col in all_cat_cols.columns if order_cond(all_cat_cols[col])]]

    @property
    def ordered(self) -> DataFrame:
        """
        Return all ordered categorical columns in the Dataframe
        """
        return self.__check_ordered(is_ordered=True)

    @property
    def unordered(self) -> DataFrame:
        """
        Return all unordered categorical columns in the Dataframe
        """
        return self.__check_ordered()

```

The categories property that was in the Series accessor is a commonly used feature, and we will use a dictionary to represent columns and their corresponding categories index.

```python
    @property
    def categories(self) -> DataFrame:
        """
        Return all categorical type in the Dataframe
        """
        cat_col = self._parent.select_dtypes("category")
        dict = {}
        for cat in cat_col:
            dict[cat] = cat_col[cat].cat.categories
        return dict
```

In order to delegate the Categorical methods, we change the inherited _delegate_method to apply the method retrieved to each categorical column. A problem arises when no categorical columns exist and the parameter to the delegated method will not be verified. To ensure correct

error reporting, we have to manually apply this on a dummy Series to ensure correctness.

```python
    def _delegate_method(self, name, *args, **kwargs):
        """
        Return the result of delegated method on all the categorical columns
        in the DataFrame.
        """
        from pandas import Series

        # We might not apply to any column. This prevents error messages in
        # parameters not being passed in. User can be unaware of this
        # until there has been a categiorical column and this is unwanted.
        try:
            bool_cat = CategoricalDtype(categories=[0])
            sr = Series(dtype=bool_cat)
            method = getattr(sr.cat, name)
            method(*args, **kwargs)
        except (TypeError, AttributeError) as ex:
            raise ex
        except:
            pass

        # Apply method on all
        cat_df_res = self._parent.cat.all

        for column in cat_df_res:
            method = getattr(cat_df_res[column].values, name)
            cat_df_res[column] = method(*args, **kwargs)

        return cat_df_res
```

Two functions ‘is_ordered_categorical_dtype’ and ‘is_unordered_categorical_dtype’ are implemented in ***/core/dtypes/common.py*** to check whether the input dtype is ordered. The type-check function takes any dtype as input and returns a boolean. In order to be compatible with all dtypes, the functions can take any dtype as the parameter and will return false if the input dtype is not a categorical type.

Given that the implementation of the two functions is similar, another helper function ‘check_ordered_categorical_dtype’ is created:

```python
def check_ordered_categorical_dtype(arr_or_dtype, is_ordered: bool=False) -> bool:
    """
    Helper function that check if the provided array or dtype is of an ordered categorical dtype.
    """
    if not is_categorical_dtype(arr_or_dtype):
        return False
    if isinstance(arr_or_dtype, ExtensionDtype):
        return arr_or_dtype.ordered if is_ordered else not arr_or_dtype.ordered
    else:
        cat_dtype = CategoricalDtype._from_values_or_dtype(values=arr_or_dtype)
    return cat_dtype.ordered if is_ordered else not cat_dtype.ordered
```

The flag ‘is_ordered’ is set to check which type-checking function is calling the helper.

Three cases are considered in this helper function. As mentioned above, the function returns False if the input dtype is not categorical. The function 'is_categorical_dtype' is used to check whether the input data is categorical. If a dtype is categorical, the function checks the value of the ordered field.

The case where the input dtype is an extension type is also considered. We call the method '_form_values_or_dtype' to form a new CategoricalDtype then check whether the input dtype is ordered.

New test cases for accessors were added to ***pandas/tests/frame/accessor/test_cat_accessor.py***, similar to the accessor tests done for Series.

New test cases for type checking functions were added to ***pandas/tests/dtypes/test_common.py*** along with the other type checking functions.

Acceptance Tests/Description

According to the changes made to ***\core\arrays\categorical.py*** and ***\core\dtypes\common.py***, acceptance tests are added to test the accessor and type checking functions' functionality on DataFrame and Categorical dtypes.

Acceptance tests and guides are located at:

pandas/tests/Issue46941_Accessor_AcceptanceTests
pandas/tests/Issue46941_OrderAndUnorderType_AcceptanceTests

The details are in their respective tests.

User guide

The user guide is modified in ***pandas/doc/source/user_guide/categorical.rst.***
We provided a pdf version of the changes **Categorical Data User Guide.pdf** in the folder deliverable4, the same as this documentation file to avoid the trouble of building.
The updated sections are in **Working with categories** and **Sorting and order.** Marked with ****Update D4 type checking**.**

Discussion timeline

2023.3.23 (2 hours):
Summary:
Decided on issue 46941 since it requires an entire accessor dedicated to DataFrame. We deemed it complex as understanding the delegation methods and the inheritance needed in pandas is nowhere close to obvious. We decided on the basis that we will implement an accessor similar to Series to ensure consistency in the code base. We decided to implement the

type checking functions for a better user experience and avoid code duplication in our implementation.

2023.3.30 (2 hours):
Summary:
Bug found in type checking function as it was supposed to handle both array types and dtypes. Spent a significant amount of time discovering existing solutions to this problem.
We discovered that implementing just categories and ordered/unordered is not exactly following the Series categorical accessor. We have to find a way to also delegate the Categorical class methods to the DataFrame. We decided to apply the delegated method across all categorical columns.

2023.4.6 (2 hours):
Summary:
Came together to code review the existing implementation. Found an error in Setup the environment for everyone to create a user guide for our implementation.

Issue 2: [ENH: Row-wise dataframe builder #50582]

Member Contribution

Xu Zheng:

Analyze the root of the issue and develop the __init__, appendRow, and build functions.
Developed acceptance tests

Megan Mujia Liu:

Analyze the root of the issue and develop the __init__, appendRow, and build functions.
Developed appendDict, asType function

Runyu Yue:

Analyze the root of the issue and develop the __init__, appendRow, and build functions.
Developed unit tests

Lingfeng Su:

Analyze the root of the issue and develop the __init__, appendRow, and build functions.

Implementation Description and Reasoning

Issue Description

The issue is a lack of behavior in the current DataFrame. The user wants to build a DataFrame row-by-row instead of providing all data at once. Pandas implement a columnar memory model, such that data is stored in memory as columns. Moreover, the current solution doesn't allow the user to specify the dtype for each column, instead, the user is only able to give one dtype for the whole DataFrame.

```
columns = ["a", "b", "c"]
dfs = []
dfs.append(DataFrame([[1, 2, 3]], columns=columns, dtype=object)
dfs.append(DataFrame([[4, 5.5, "potato"]], columns=columns, dtype=object)
df_tmp = pd.concat(dfs, ignore_index=True)
df = df_tmp.astype({"a": "uint64", "b": "float32", "c": object}, copy=False)
```

Now, if the user wants to use rows to build data, the user must create two DataFrame and then concatenate them together. Then they are able to specify the type of each column by using a dictionary.

Implementation

To provide the desired functionality to the user, we decided to create a new class dfBuilder for building a DataFrame row-wisely. A dfBuilder object is initialized by given column names and optional dtypes for each column. It can take a list as a row, or a dictionary with a column and the data as key-value-pair. It provides a build function to finalize the information a user gives to a DataFrame. It also allows for type checking that is consistent with pandas' implementation.

Detailed Changes

We reached an agreement to solve the problem by creating a new class called dfBuilder to provide a way to create complex objects step by step. It allows users to construct an object incrementally, with fine-grained control over each step of the construction process, which conforms to the builder design pattern. In specific, a class called dfBuilder is created which includes the constructor, the asType() method, the appendRow(), appendDict() method, and the build() method.

The constructor is used to initialize the column name and the types for each column of the dataframe, in which the types being passed are optional.

```python
class dfBuilder:
    __rows = []

    def __init__(
        self,
        columns: list,
        dtypes: list = None,
    ):
        self.__rows=[]
        self.columns = columns
        if dtypes is not None:
            self.dtypes = list(dtypes)
            if len(columns) != len(dtypes):
                raise ValueError("Given columns and dtypes length do not match")
        else:
            self.dtypes = None
```

appendRow() and appendDict() are used to append rows to the dataframe rows, if the data is not the same as dtype(). It will try to convert the data to the corresponding type and then append it.

```python
def appendDict(self, row: dict):
    new_row = []
    for i in range(len(self.columns)):
        if self.columns[i] not in row:
            raise ValueError("Missing data of column " + self.columns[i])

        if self.dtypes is not None:
            r = np.array([row[self.columns[i]]], dtype= np.dtype(self.dtypes[i]))
            new_row.append(r[0])
        else:
            new_row.append(row[self.columns[i]])

    self.__rows.append(new_row)

    return self
```

```python
def appendRow(self, row: list):
    if len(row) != len(self.columns):
        raise ValueError("Given row length not match with columns length")

    if self.dtypes is not None:
        new_row = []
        for i in range(len(self.columns)):
            r = np.array([row[i]], dtype=np.dtype(self.dtypes[i]))
            new_row.append(r[0])
        self.__rows.append(new_row)

    else:
        self.__rows.append(row)

    return self
```

asType() function is used to cast the type of the columns in the dataframe after the instantiation

```
def asType(self, dtype: list):
    if len(self.columns)!= len(dtype):
        raise ValueError("Given dtypes length do not match with columns length")

    if len(self.__rows) ==0:
        self.dtypes = list(dtype)
    else:
        for i in range(len(self.columns)):

            for j in range(len(self.__rows)):

                column_check=[(self.__rows[j][i])]
                r = np.array(column_check, dtype=np.dtype(dtype[i]))
                self.__rows[j][i] = r[0]
        self.dtypes=list(dtype)

    return self
```

build() function is used to build and return the dataframe generated according to the given parameters.

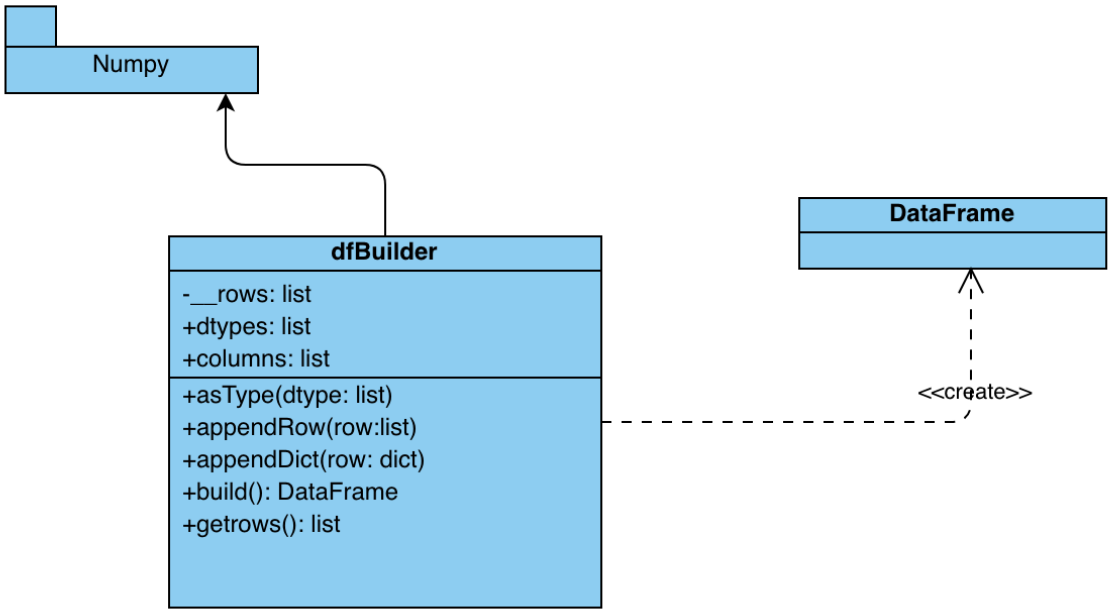
```
def build(self):
    df = DataFrame(self.__rows, columns=self.columns)
    if self.dtypes is not None:
        for col, dtype in zip(self.columns, self.dtypes):
            df[col] = df[col].astype(dtype)
    return df
```

Added Files:
pandas/pandas/core/framebuilder.py
pandas/pandas/tests/frame/test_dfBuilder_part1.py
pandas/pandas/tests/frame/test_dfBuilder_part2.py
pandas/pandas/tests/Issue50582_DfBuilder_AcceptanceTests/README.md
pandas/pandas/tests/Issue50582_DfBuilder_AcceptanceTests/test_dfBuilder.py

Modified Files:
pandas/pandas/__init__.py
pandas/doc/source/user_guide/dsintro.rst

Design Pattern

Our design conformed to the Builder Design Pattern. A builder pattern provides flexible solutions to various object creation problems, such that this pattern separates the construction of a complex object from the representation. Class dfBuilder is a builder for constructing DataFrame objects. It provides an alternative way for constructing DataFrame, and this construction functionality is independent of the class DataFrame. Our design promotes loose coupling.



Unit Tests Description

Unit tests are divided into 2 files.
The first file is located in:pandas/pandas/tests/frame/test_dfBuilder_part1.py.
The second test is in pandas/pandas/tests/frame/test_dfBuilder_part2.py.
To run the tests, go to the frame folder and run “pytest filename”.
The unit tests are fully elaborated on each sub-method inside the dfBuilder class. Each method has separate tests for the normal cases, the edge cases, and the error cases if the cases are issued in our code.

Acceptance Tests Description

According to the enhancement made on class ***DataFrame***, acceptance tests are in a newly created folder, and there’s also a README file that gives clear instructions to users on how to run acceptance tests and what’s the expected result.

Acceptance tests and guides are located at:
pandas/pandas/tests/Issue50582_DfBuilder_AcceptanceTests/

We basically test all user cases in different situations, simulating a real working environment. Specific details can be found in
pandas/pandas/tests/Issue50582_DfBuilder_AcceptanceTests/test_dfBuilder.py

User guide

The user guide is modified in ***pandas/doc/source/user_guide/dsintro.rst.***
We provided a pdf version of the changes in the same directory as this documentation file to avoid the trouble of building.

Discussion Timeline

2023.3.23
We searched in the pandas repo and found several potential issues for this deliverable. After reading through several, we decided on another issue #51478. However, this issue was suggested by the TA that it is not significant enough for this deliverable.

2023.3.30
We looked into different kinds of issues again and decided on #50582. We ran several tests on pandas version 1.5.3 to see the current behavior of the DataFrame. We discussed several solutions on how to implement the new features. The solution includes: using existing DataFrame functionalities which are similar to the alternative solution in the issue description, building a new class for saving row data than building to DataFrame, etc. We reached an agreement to solve the problem by creating a new class called dfBuilder to build the DataFrame in a more convenient way. We also realized that using an NP array is a convenient solution to checking our types to see if it's compatible with pandas since pandas use Numpy under the hood anyways, this allowed us to check the types of user input and convert types in a way that is consistent to pandas' behavior. Based on these ideas, we finished the basic implementation of the class including the constructor, the appendRow() method, and the build() method.

2023.4.2
We realized the pandas can also create a dataframe from a dictionary and we do not have this feature, so we created new methods appendDict() to add rows by directly passing in a dictionary. Besides this, we also add a method asType() to add the type to the dataframe even if the user does not initialize it in the constructor. We reviewed our code and fixed the build()

function which did not work as expected because it did not give type to the returning dataframe. Then we added the unit test and acceptance tests to fully test our new class.

Group Development Processes

Trello Invitation Link:
https://trello.com/invite/b/Uq5KhMEH/ATTIf6e2d135724a0bccadc2f567583a0340CB02AB10/scrum-board

Group-wise Meeting 1 (March 23, 2023: 2 hours)

Summary:
Created the deliverable 4 repositories based on pandas 1.5.x as the issues we selected were new features that did not exist at this point. Discussed the difficulty of selected issues and selected 51478 and 46941 initially. Each group is then separated to investigate their respective issue and come up with a UML.

Group-wise Meeting 2 (March 30, 2023: 1 hour)

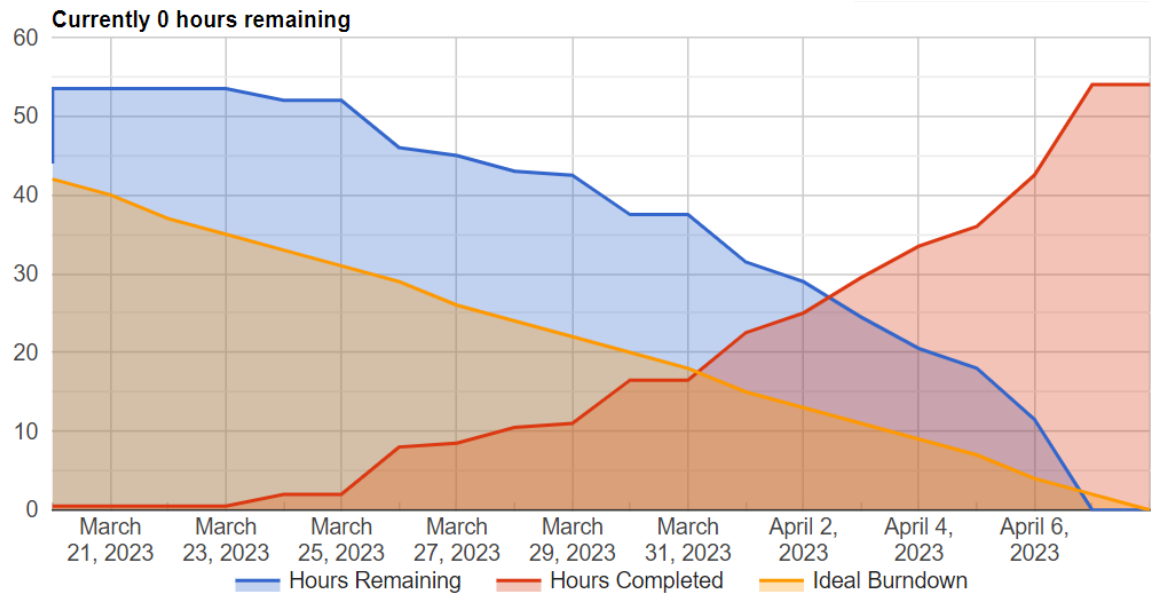
Summary:
Group 2 switched to 50582 after TAs suggested that 51478 is too easy. Discussions are done in both teams as per the meeting notes in the respective sections.

Group-wise Meeting 3 (April 6, 2023: 1 hour)

Summary:
Discussed user guide documentation, making sure that both groups follow the same guidelines. In addition, make sure both groups make good progress so that we can finish `deliverable 4` on time.

Task Time Estimation and Burndown Result

Burndown Chart



Our estimated total hours is 54 hours. Looking at the artifacts below, our estimations are mostly correct with a few overestimated tasks. Different from last time, we are definitely more familiar with the Pandas system as we see ourselves solving the tasks a lot faster than before.

Time Estimation

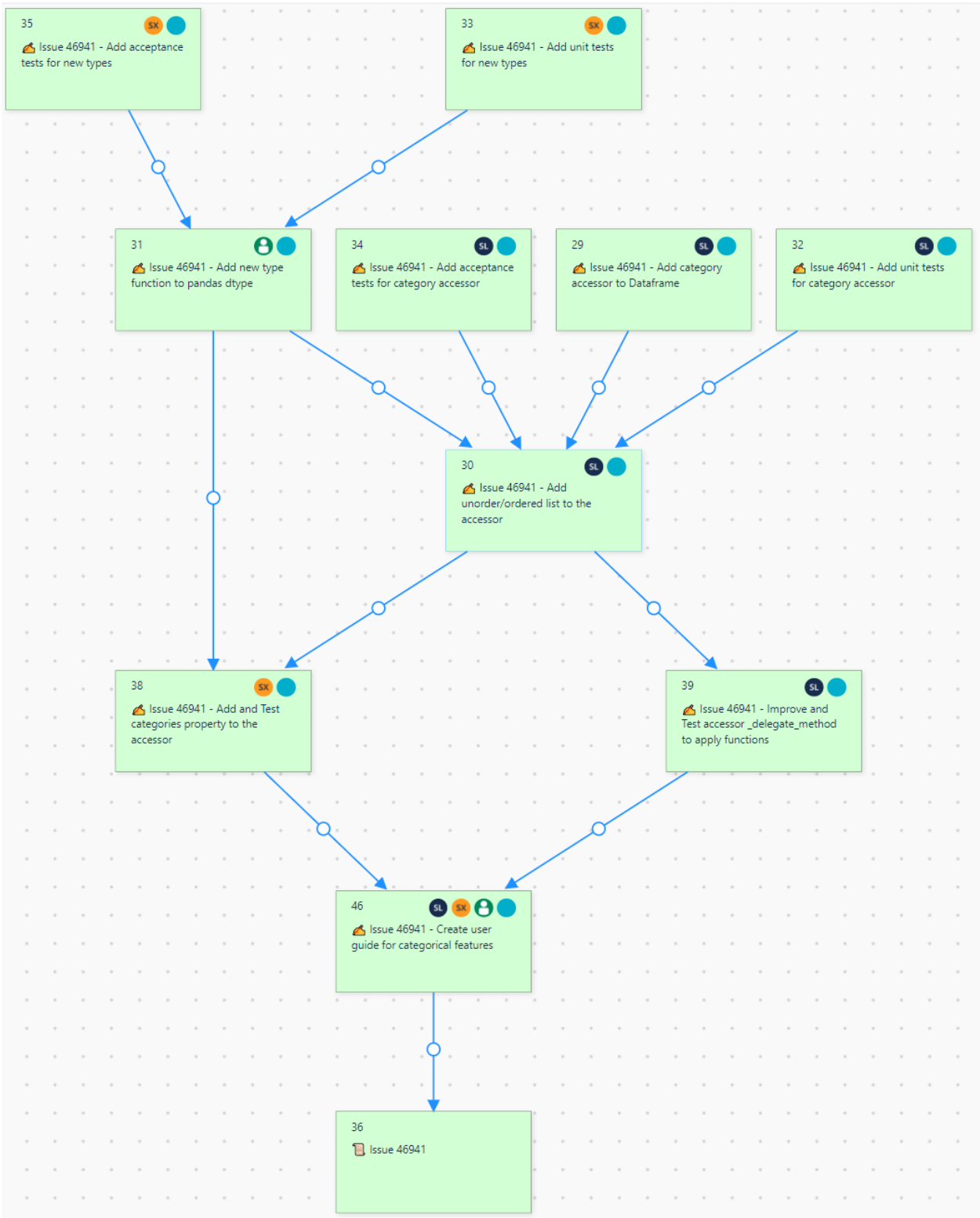
Card	Card labels	Time	Estimate
Issue #50582-4.2-add unit tests for asType, build function		1h	1h
Issue #50582-3.5-implement the asType function		1h 30m	2h
Issue #50582 - TASK - pass all tests without affecting old tests		30m	1h
Issue #50582 - TASK - finish up the doc		2h	2h
Issue #50582 - 5.2- add acceptance tests for asType and build function		1h 30m	2h
Issue #50582 - 5.1 - add acceptance tests for framebuilder		1h 30m	2h
Issue #50582 - 4.1 - add unit tests for __init__, appendRow, and appendDict		1h	1h
Issue #50582 - 3.4- implement the appendDict function		1h	1h
Issue #50582 - 3.3 - implement the build function		30m	1h
Issue #50582 - 3.2- implement the appendRow function		3h	4h
Issue #50582 - 3.1 - implement the __init__ function		3h 30m	3h
Issue #50582 - 2 - identify the location to properly implement new feature		2h	3h
Issue #50582 - 1 - set up dev environment		2h	1h
✍️ Issue 46941 - Improve and Test accessor _delegate_method to apply functions	Task	8h	8h
✍️ Issue 46941 - Create user guide for categorical features	Task	4h 6m	6h
✍️ Issue 46941 - Add unorder/ordered list to the accessor	Task	1h 40m	2h
✍️ Issue 46941 - Add unit tests for new types	Task	2h	2h
✍️ Issue 46941 - Add unit tests for category accessor	Task	2h 54m	3h
✍️ Issue 46941 - Add new type function to pandas dtype	Task	2h 12m	2h
✍️ Issue 46941 - Add category accessor to Dataframe	Task	50m	1h
✍️ Issue 46941 - Add and Test categories property to the accessor	Task	2h	2h
✍️ Issue 46941 - Add acceptance tests for new types	Task	2h	2h
✍️ Issue 46941 - Add acceptance tests for category accessor	Task	1h 46m	2h

Tracking/Assignment Artifacts

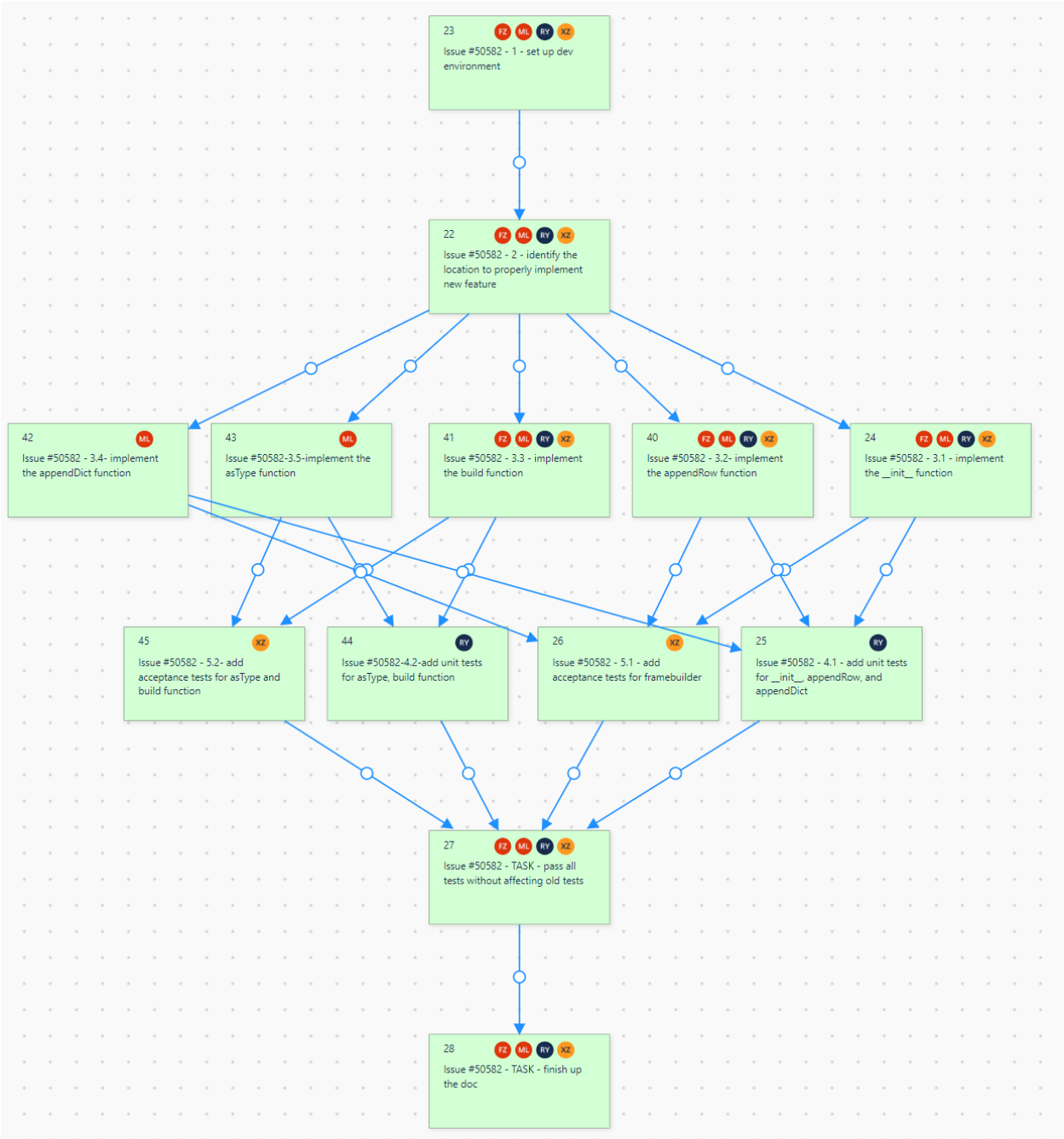
We develop the priority of the dependency relation and task assignment below.

Dependency Graph

Issue #46941



Issue #50582



Task assignments

Xuen Shen:

SX Shen Xuen	✍ Issue 46941 - Add unit tests for new types
SX Shen Xuen	✍ Issue 46941 - Add acceptance tests for new types
SX Shen Xuen	✍ Issue 46941 - Add and Test categories property to the accessor

Xu Zheng:

XZ xu zheng	Issue #50582 - 5.1 - add acceptance tests for framebuilder
XZ xu zheng	Issue #50582 - 1 - set up dev environment
XZ xu zheng	Issue #50582 - 5.2- add acceptance tests for asType and build function
XZ xu zheng	Issue #50582 - 3.3 - implement the build function
XZ xu zheng	Issue #50582 - 2 - identify the location to properly implement new feature
XZ xu zheng	Issue #50582 - 3.1 - implement the __init__ function
XZ xu zheng	Issue #50582 - 3.2- implement the appendRow function

Lingfeng Su:

FZ flying zambie	Issue #50582 - 1 - set up dev environment
FZ flying zambie	Issue #50582 - 2 - identify the location to properly implement new feature
FZ flying zambie	Issue #50582 - 3.1 - implement the __init__ function
FZ flying zambie	Issue #50582 - TASK - pass all tests without affecting old tests
FZ flying zambie	Issue #50582 - TASK - finish up the doc

Yawen Zhang:

嗡嗡嗡	✍ Issue 46941 - Add new type function to pandas dtype
嗡嗡嗡	✍ Issue 46941 - Create user guide for categorical features

Megan Mujia Liu:

ML Megan Liu	Issue #50582 - 2 - identify the location to properly implement new feature
ML Megan Liu	Issue #50582 - 3.1 - implement the __init__ function
ML Megan Liu	Issue #50582 - 3.4- implement the appendDict function
ML Megan Liu	Issue #50582-3.5-implement the asType function
ML Megan Liu	Issue #50582 - 1 - set up dev environment
ML Megan Liu	Issue #50582 - 3.3 - implement the build function
ML Megan Liu	Issue #50582 - 3.2- implement the appendRow function

Shaopeng Lin:

SL Shaopeng Lin	✍️ Issue 46941 - Add category accessor to Dataframe
SL Shaopeng Lin	✍️ Issue 46941 - Add unorder/ordered list to the accessor
SL Shaopeng Lin	✍️ Issue 46941 - Add unit tests for category accessor
SL Shaopeng Lin	✍️ Issue 46941 - Add acceptance tests for category accessor
SL Shaopeng Lin	✍️ Issue 46941 - Improve and Test accessor _delegate_method to apply functions
SL Shaopeng Lin	✍️ Issue 46941 - Create user guide for categorical features

Runyu Yue:

RY Runyu Yue	Issue #50582 - 4.1 - add unit tests for __init__, appendRow, and appendDict
RY Runyu Yue	Issue #50582-4.2-add unit tests for asType, build function
RY Runyu Yue	Issue #50582 - 3.1 - implement the __init__ function
RY Runyu Yue	Issue #50582 - 3.2- implement the appendRow function
RY Runyu Yue	Issue #50582 - 3.3 - implement the build function
RY Runyu Yue	Issue #50582 - 2 - identify the location to properly implement new feature
RY Runyu Yue	Issue #50582 - 1 - set up dev environment
RY Runyu Yue	Issue #50582 - TASK - finish up the doc