Revisiting PySpark pandas UDF

In the past two years, the pandas UDFs are perhaps the most important changes to Spark for Python data science. However, these functionalities have evolved organically, leading to some inconsistencies and confusions among users. This document revisits UDF definition and naming.

Table of contents

Existing pandas UDFs

As of Dec. 30, 2019, we have the following types of pandas UDFs in Spark. There are a few issues with the existing UDFs:

- There are a lot of different types of pandas UDFs that are difficult to learn. This is the result of the next bullet point.
- The type names in most cases describe the Spark operations the UDFs can be used with, rather than describing the UDF itself. An example here is `SCALAR` and `GROUPED_MAP`. The two are almost identical, except SCALAR can only be used in select, while `GROUPED_MAP` can only be used in `groupby().apply()`. As we implement more operators in which these UDFs can be applied, we will add more and more different types.
- I believe the initial "SCALAR" type specifies that the UDF returns a single column, rather than a DataFrame. That convention is now broken with the new `SCALAR_ITER` feature. The "SCALAR" name is also confusing to many users, because none of the operations are scalar. They are all vectorized (as in operating on arrays of data) in some form or another.
- It is unclear whether a UDF's input should be a number of `Series`, or a single `DataFrame`. `SCALAR_ITER` with `mapInPandas` and `GROUPED_MAP` accept `DataFrame`, but everything else accepts `Series`.
- There are two different ways to encode struct columns. In `SCALAR_ITER`, a struct column is encoded as a `pd.DataFrame`, and normal columns are encoded as a `pd.Series`. In other types, a struct column is encoded as a `pd.Series` where each element is a dictionary. The former does not support multiple levels of structs, while the latter leverages pandas as purely a serialization mechanism, as pandas itself has virtually no functionality to operate on these dictionaries.
- (Maybe more controversial) `GROUPED_MAP`, `GROUPED_AGG` and `COGROUPED_MAP` are potentially a recipe for disaster, because they require materializing the entirety of each group in memory as a single pandas DataFrame. When data grows for a specific group, users' programs would run out of memory.

SCALAR

DataFrame or Series, ... -> DataFrame or Series

The UDF must ensure output cardinality is the same as input cardinality.

```
@pandas_udf("long", PandasUDFType.SCALAR)
def multiply(a, b):
    return a * b

df.select(multiply(col("x"), col("x"))).show()
```

SCALAR_ITER (part of Spark 3.0)

Iterator[Tuple[Series or DataFrame, ...]] -> Iterator[Series or DataFrame]

The UDF must ensure output cardinality is the same as input cardinality. This version is added to give the UDF an opportunity to manage its own life cycle, e.g. loading a machine learning model once and applying that model to all batches of data.

```
@pandas_udf("long", PandasUDFType.SCALAR_ITER)
def plus_one(batch_iter):
    for x in batch_iter:
        yield x + 1

df.select(plus_one(col("x"))).show()
```

MAP_ITER with mapInPandas (part of Spark 3.0)

Iterator[DataFrame] -> Iterator[DataFrame]

The UDF can change cardinality. For example, it can apply filtering operations.

```
@pandas_udf(df.schema, PandasUDFType.MAP_ITER)
def filter_func(batch_iter):
    for pdf in batch_iter:
        yield pdf[pdf.id == 1]

df.mapInPandas(filter_func).show()
```

GROUPED_MAP

DataFrame -> DataFrame

Without grouping key in the function:

```
@pandas_udf("id long, v double", PandasUDFType.GROUPED_MAP)
def subtract_mean(pdf):
    v = pdf.v
    return pdf.assign(v=v - v.mean())

df.groupby("id").apply(subtract_mean).show()
```

With grouping key in the function:

```
@pandas_udf("id long, v double", PandasUDFType.GROUPED_MAP)
def subtract_mean(key, pdf):
    v = pdf.v
    return pdf.assign(v=v - v.mean())

df.groupby("id").apply(subtract_mean).show()
```

GROUPED_AGG

DataFrame or Series, ... -> single value

```
@pandas_udf("double", PandasUDFType.GROUPED_AGG)
def mean_udf(ser):
    return ser.mean()

df.groupby("id").agg(mean_udf).show()
```

GROUPED_AGG also works with window functions:

```
w = Window ¥
    .partitionBy('id') ¥
    .rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
df.withColumn('mean_v', mean_udf(df['v']).over(w)).show()
```

COGROUPED_MAP (part of Spark 3.0)

DataFrame -> DataFrame

Without grouping key in the function:

```
@pandas_udf('id long, k int, v int, v2 int', PandasUDFType.COGROUPED_MAP)
def merge_pandas(left, right):
    return pd.merge(left, right, how='outer', on=['k'])

df.groupby(l.id).cogroup(r.groupby(r.id)).apply(merge_pandas)
```

With grouping key in the function:

```
@pandas_udf('time int, id int, v double', PandasUDFType.COGROUPED_MAP)
def asof_join(key, left, right):
    if key == (1,):
        return pd.merge_asof(left, right, how='outer', on=['k'])
    else:
        return pd.DataFrame(columns=['time', 'id', 'v'])

df.groupby("id").cogroup(df2.groupby("id")).apply(asof_join).show()
```

New Proposal

Rather than focusing on a single dimension called "type" and an output schema, I would like to propose to infer such cardinality by leveraging Python type hint (see PEP 484).

The type hints, `pd.Series`, `pd.DataFrame`, `Tuple` and `Iterator`, are handled mainly and other types are simply ignored. Such details could vary at implementation level. If the type hint is not given, an exception will be thrown. This proposal does not cover deprecated Python 2 and Python 3.4 & 3.5 supports in PySpark.

Given the analysis of the previous proposal, we figured out the current pandas UDFs can be classified by cardinality and input type. This proposal keeps this classification but uses Python type hint to express each case. Please see the previous proposal to see the definitions of proposed attributes mentioned below.

- `schema`: output schema. Same as the current "schema" field.
- `input`: instead of `input` attribute, we can infer input types from type hint.

  pandas Series or DataFrame that represent multiple Spark columns (`cols` in the previous proposal)

  ```
  def func(c1: Series, c2: DataFrame, ...):
      pass
  ```

  Same as above but iterator version (`cols iter` in the previous proposal)

  ```
  def func(iter: Iterator[Tuple[Series, DataFrame, ...]]):
      pass
  ```

  pandas DataFrame that represents the Spark DataFrame (`df` in the previous proposal)

  ```
  def func(df: DataFrame):
      pass
  ```

Same as above but iterator version (`df iter` in the previous proposal)

```
def func(iter: Iterator[DataFrame]):
    pass
```

- `cardinality`: instead of `cardinality` attribute, we can infer it from input and output type hints

  Many-to-many or one-to-one cardinality (`n to n` and `n to m` in the previous proposal)

  ```
  def func(c1: Series) -> Series:
      pass
  ```

  Many-to-one cardinality (`n to 1` in the previous proposal)

  ```
  def func(c1: Series) -> int:
      pass
  ```

Therefore, the complete examples of the new proposal would be as below:

```
@pandas_udf(schema='...')
def func(c1: Series, c2: Series) -> DataFrame:
    pass
```

```
@pandas_udf(schema='...')
def func(iter: Iterator[Tuple[Series]]) -> int:
    pass
```

```
@pandas_udf(schema='...')
def func(df: DataFrame) -> DataFrame:
    pass
```

```
@pandas_udf(schema='...')
def func(iter: Iterator[Tuple[Series, DataFrame, ...]]) -> Iterator[Series]:
    pass
```

```
@pandas_udf(schema='...')
def func(iter: Iterator[DataFrame]) -> Iterator[DataFrame]:
    pass
```

Discussions

Many benefits and justification are inherited from the previous proposal, for instance, both proposals still can decouple UDF type from input type, and can cover the same functionalities as before (see **Appendix** for this mapping of old and new styles, and the previous proposal).

Nevertheless, there are major differences specifically for this proposal, for instance, many benefits are also inherited from Python hint (see also PEP 484 and this blog).

In this section, it targets to discuss the major benefits, tradeoff compared to the previous proposal, downside and abandoned alternatives.

Benefits

"Pythonic" PySpark APIs

Python type hints seem to be encouraged in general. For instance, the usage of mypy, Python type hint linter, has been rapidly increased lately. Python libraries such as pandas or NumPy started to add and fix such type hints (see here for pandas and here for NumPy). Type hinting seems to be used in production as well sometimes given this blog. As a long term design, leveraging Python type hints seems a reasonable way to make PySpark APIs more "Pythonic".

Clear definition for supported UDFs

One benefit of using Python type hints is the easy understanding of codes and clear definition of what the function is supposed to do. As an example, SCALAR UDF requires always to return a Series or a DataFrame. None is disallowed. With the explicit type hint, we can avoid such many subtle cases to document with a bunch of test cases and/or for users to test and figure out by themselves.

Allowing easier static analysis

IDEs and editors such as PyCharm and Visual Studio Code can leverage type annotations to provide code completion, for instance, to show errors, and to support better go to definition functionality. See also mypy#ide-linter-integrations-and-pre-commit.

Tradeoff

Missing notation of many-to-many vs one-to-one

As mentioned earlier, there is a conflict in the notion of many-to-many vs one-to-one.

```
@pandas_udf(schema='...')
def func(c1: Series) -> Series:
    pass
```

This can mean both many-to-many and one-to-one relations. The size of the output Series can be the same as its input or different.

In the previous proposal, this was able to be distinguished by `n to m` and `n to n`. The current proposal defers to the runtime checking.

Two places to define the UDF type

In the previous proposal, one decorator can express the UDF execution as below:

```
@pandas_udf(schema='...', cardinality="...", input="...")
def func(c1):
    pass
```

However, in the current proposal, now the function specification affects the UDF execution as below:

```
@pandas_udf(schema='...')
def func(c1: Series) -> Series:
    pass
```

In a way, the former can be simpler with simple arguments in single place but the latter could look too verbose on the other hand.

Downsides

Premature Python type hint

Arguably, the Python type hint is yet premature. Python type hint was introduced from Python 3.5 as of PEP 484. Although it has been several years, still new type hint APIs are being added. See also "Why optional type hinting in python is not that popular?" (although it was 2 years ago).

Considering the stability and arguably prematurity, it might lead to forcing users to use what they are not used to, and/or unstable support in pandas UDFs.

Enforcing optional Python type hint

This is related to the prematurity discussed above. Python type hint is completely optional at this moment, and the current proposal enforces users to specify the type hints. If users are not familiar with type hinting, likely they would feel the new design of pandas UDFs is more difficult on the other hand.

Nevertheless, note that in some cases the Python type hint can stay as optional. This is left as a future improvement. See Appendix for more discussions

Abandoned Alternatives

There have been several alternatives and options as below. They have been abandoned.

Merging to the regular UDF interface

It seems now feasible to merge `pandas_udf` to `udf` because the type hint can specify the input and output, and it is able to distinguish each:

```
@udf(schema='...')
```

```
def func(c1: Series) -> Series:
    pass
```

```
@udf(schema='...')
def func(value):
    pass
```

However, there are several usages specific to pandas UDFs, for example,

```
df.groupby(...).cogroup(udf))
```

```
df.groupby(...).apply(udf))
```

```
df.mapInPandas(udf)
```

If both are merged, it would imply regular UDFs should also work with the cases above. It could work if we explicitly throw exceptions in some cases but sounds incoherent.

To work around, we might be able to do below:

- Merge `pandas_udf` to `udf` only where the usage is the same as the regular UDF (e.g., `df.select(udf(col))`)
- Have another API called, for instance, `pandas_func` to cover pandas UDF specific usages.

However, it was abandoned as this looks somewhat over-complicated, and introducing new APIs with mixing the existing APIs could easily confuse, in particular, the existing users.

Having two APIs to express many-to-many vs one-to-one

To work around the conflict between many-to-many cardinality (`n to m`) and one-to-one cardinality (`n to n`) in the current proposal, It was considered for `pandas_udf` to have two new categories:

- `pandas_transform`: one-to-one cardinality
- `pandas_apply`: many-to-many cardinality

The terms `transform` and `apply` are from pandas. See [DataFrame.transform]() and [DataFrame.apply]() in pandas documentation.

This was feasible but looked also over-complicated. It does not look worth enough to introduce two different names only to work around one case of car dinality, which can be already worked around by runtime checking.

Appendix

Old style

SCALAR

```
@pandas_udf(schema='...', functionType=SCALAR)
def func(c1, c2):
    pass
```

SCALAR_ITER

```
@pandas_udf(schema='...', functionType=SCALAR_ITER)
def func(iter):
    pass
```

MAP_ITER

```
@pandas_udf(schema='...', functionType=MAP_ITER)
def func(iter):
    pass
```

GROUPED_MAP

```
@pandas_udf(schema='...', functionType=GROUPED_MAP)
def func(df):
    pass
```

```
@pandas_udf(schema='...', functionType=GROUPED_MAP)
def func(key, df):
    pass
```

GROUPED_AGG

```
@pandas_udf(schema='...', functionType=GROUPED_AGG)
def func(c1, c2):
    pass
```

COGROUPED_MAP

```
@pandas_udf(schema='...', functionType=COGROUPED_MAP)
def func(left_df, right_df):
    pass
```

```
@pandas_udf(schema='...', functionType=COGROUPED_MAP)
def func(key, left_df, right_df):
    pass
```

New style

SCALAR

```
@pandas_udf(schema=’...’)
def func(c1: Series, c2: DataFrame) -> Series:
    pass  # DataFrame represents a struct column
```

```
@pandas_udf(schema=’...’)
def func(c1: Series, c2: DataFrame) -> DataFrame:
    pass  # DataFrame represents a struct column
```

SCALAR_ITER

```
@pandas_udf(schema=’...’)
def func(iter: Iterator[Tuple[Series, DataFrame, ...]]) -> Iterator[Series]:
    pass  # Same as SCALAR but wrapped by Iterator
```

MAP_ITER

```
@pandas_udf(schema=’...’)
def func(iter: Iterator[DataFrame]) -> Iterator[DataFrame]:
    pass
```

GROUPED_MAP

```
@pandas_udf(schema=’...’)
def func(df: DataFrame) -> DataFrame:
    pass
```

```
@pandas_udf(schema=’...’)
def func(key: Tuple[...], f: DataFrame) -> DataFrame:
    pass
```

GROUPED_AGG

```
@pandas_udf(schema=’...’)
def func(c1: Series, c2: DataFrame) -> int:
    pass  # DataFrame represents a struct column
```

COGROUPED_MAP

```
@pandas_udf(schema=’...’)
def func(left_df: DataFrame, right_df: DataFrame) -> DataFrame:
    pass
```

```
@pandas_udf(schema=’...’)
def func(key: Tuple[...], left_df: DataFrame, right_df: DataFrame) -> DataFrame:
```

```
    pass
```

Side discussions and future Improvement

There have been several future improvements suggested.

A couple of less-intuitive pandas UDF types by Maciej Szymkiewicz

In short, the current usage of pandas UDFs are inconsistent with regular UDFs. For example, MAP_ITER, GROUPED_MAP, and COGROUPED_MAP as below.

```
df.mapInPandas(filter_func).show()
df.groupby("id").apply(subtract_mean_func).show()
df.groupby(l.id).cogroup(r.groupby(r.id)).apply(merge_func)
```

The problem is that, it requires to take a function wrapped by `pandas_udf`. It might make more sense to directly accept a Python native function, and the APIs should take the output schema as these APIs do not really take column instances unlike other pandas and regular UDFs. See here for more details.

Optional type hints by Li Jin

In some cases, we can avoid enforcing the type hint. As an example, we can assume such type hints as below by default.

```
@pandas_udf(schema='...')
def func(c1: Series, c2: DataFrame) -> Series:
    pass  # DataFrame represents a struct column
```

Furthermore, we could just throw a runtime exception in the case below as the API itself only requires a single type of pandas UDF:

```
df.mapInPandas(filter_func).show()
df.groupby("id").apply(subtract_mean_func).show()
df.groupby(l.id).cogroup(r.groupby(r.id)).apply(merge_func)
```

This proposal leaves this as a future improvement.

NumPy array consideration and extensibility by Li Jin

In some cases, NumPy array can be much more robust compared to pandas instances (up to 10 times). We might have to officially support it and type hints should be supported accordingly.