Apache Spark 3.0 新的 Pandas UDF 及 Python Type Hints

Pandas 用户定义函数（UDF）是 Apache Spark
中用于数据科学的最重要的增强之一，它们带来了许多好处，比如使用户能够使用 Pandas API和提高性能。

但是，随着时间的推移，Pandas UDFs
已经有了一些新的发展，这导致了一些不一致性，并在用户之间造成了混乱。即将推出的 Apache Spark 3.0 完整版将为 Pandas UDF 引入一个新接口，该接口利用 Python
类型提示（Python type hints）解决 Pandas UDF 类型的泛滥，并使得它变得更符合 Python 风格。

这篇文章将介绍带有 Python 类型提示的新 Pandas UDF，以及新的 Pandas Function API，包括 grouped map、map 以及 co-grouped map。

Pandas UDFs

在Spark 2.3中引入了Pandas UDF，请参见PySpark中对panda UDF的介绍。panda是数据科学家所熟知的，它与许多Python库和包(如NumPy、statsmodel和scikit-learn)进行了无缝集成，并且panda
udf不仅允许数据科学家扩展工作负载，还允许他们利用Apache Spark中的panda api。

Pandas UDF 是从 Spark 2.3 版本开始引入的。 Pandas 是数据科学家所熟知的，并且与许多 Python 库和软件包（例如 NumPy，statsmodel 和 scikit-learn）无缝集成，Pandas UDF 使数据科学家不仅可以扩展工作负载，还可以在 Apache Spark 中利用 Pandas API。

用户定义的函数通过以下方式执行：

- Apache Arrow，用于在 JVM 和 Python driver/executors 之间直接交换数据，序列化和反序列化成本几乎为零。
- 函数内部的 Pandas，可与 Pandas 实例和 API 配合使用。

Pandas UDF 与函数内部的 Pandas API 和 Apache Arrow 一起使用来交换数据。它允许向量化的操作，相比一行一行的执行性能提高多达100倍。

下面的示例展示了使用 Pandas UDF 实现给某个值加一的场景，它由 pandas_udf 装饰的名为 pandas_plus_on 的函数定义，且 Pandas UDF 类型指定为 PandasUDFType.SCALAR。

```
from pyspark.sql.functions import pandas_udf, PandasUDFType

@pandas_udf('double', PandasUDFType.SCALAR)
def pandas_plus_one(v):
    # `v` is a pandas Series
    return v.add(1)  # outputs a pandas Series

spark.range(10).select(pandas_plus_one("id")).show()
```

Python 函数的输入和输出是 Pandas Series。
在这个函数中我们可以执行矢量化操作，并利用丰富的 Pandas API 为每个值加一。

Python Type Hints

Python type hints 是在 Python 3.5 引入的，参见 PEP 484。类型提示是在 Python 中静态指示值类型的一种官方方法。 请参见下面的示例：

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

name: str 显示 name 参数是 str 类型的，-> 语法显示 greeting() 函数返回 string 类型。

Python type hints 为 PySpark 和 Pandas UDF context 带来了两个明显的好处：

- 它给出了函数应该做什么的明确定义，使用户更容易理解代码。例如，除非有文档记录，否则在没有类型提示时，用户无法知道 greeting 函数是否可以接受 None。它可以避免用一堆测试用例来记录这些微妙的用例，或者让用户自己测试和解决。

- 它可以使静态分析更加容易。 诸如 PyCharm 和 Visual Studio Code 之类的 IDE 可以利用类型注释来提供代码提示、显示错误并支持更好的定位功能。

Proliferation of Pandas UDF Types

自 Apache Spark 2.3 发布以来，已实现了许多新的 Pandas UDF，这使用户很难了解新规范及其使用方法。 例如，下面是三个 Pandas UDF 输出的结果几乎相同：

```
from pyspark.sql.functions import pandas_udf, PandasUDFType



@pandas_udf('long', PandasUDFType.SCALAR)
def pandas_plus_one(v):
    # `v` is a pandas Series
    return v + 1  # outputs a pandas Series

spark.range(10).select(pandas_plus_one("id")).show()
```

```
from pyspark.sql.functions import pandas_udf, PandasUDFType


# New type of Pandas UDF in Spark 3.0.
@pandas_udf('long', PandasUDFType.SCALAR_ITER)
def pandas_plus_one(itr):
    # `iterator` is an iterator of pandas Series.
    return map(lambda v: v + 1, itr)  # outputs an iterator of pandas Series.

spark.range(10).select(pandas_plus_one("id")).show()
```

```
from pyspark.sql.functions import pandas_udf, PandasUDFType

@pandas_udf("id long", PandasUDFType.GROUPED_MAP)
def pandas_plus_one(pdf):
    # `pdf` is a pandas DataFrame
    return pdf + 1  # outputs a pandas DataFrame
```

```
# `pandas_plus_one` can _only_ be used with `groupby(...).apply(...)`
spark.range(10).groupby('id').apply(pandas_plus_one).show()
```

尽管每种 UDF
类型都有不同的用途。在这个简单的例子中，您可以使用这三个中的任何一个。但是，每个 Pandas UDF 都希望输入和输出类型不同，并且以不同的方式工作，具有不同的语义和不同的性能。它让用户分不清要使用和学习哪一种，以及每一种是如何工作的。

此外，在使用常规 PySpark 列的情况下，可以使用第一和第二种情况中的 pandas_plus_one。最后一个 pandas_plus_one 只能与 groupby(...).apply(pandas_plus_one) 一起使用。

这种高度的复杂性引发了 Spark 开发人员的众多讨论，并推动通过官方提案引入带有 Python 类型提示的新 Pandas API。目的是使用户能够使用 Python 类型的提示自然表达 Pandas UDF，而不会出现上述问题情况那样的困惑。 例如，上面的情况可以写成如下形式：

```
def pandas_plus_one(v: pd.Series) -> pd.Series:
    return v + 1
```

```
def pandas_plus_one(itr: Iterator[pd.Series]) -> Iterator[pd.Series]:
    return map(lambda v: v + 1, itr)
```

```
def pandas_plus_one(pdf: pd.DataFrame) -> pd.DataFrame:
    return pdf + 1
```

支持 Python Type Hints 的新 Pandas APIs

为了解决旧版 Pandas UDF 的复杂性，Apache Spark 3.0 引入了 Python 3.6 及更高版本，可以使用诸如 pandas.Series、pandas.DataFrame、Tuple 和 Iterator 之类的 Python 类型提示来表示新的 Pandas UDF 类型。

此外，旧的 Pandas UDF 分为两类 API：Pandas UDF 和 Pandas Function API。尽管它们在内部以类似的方式工作，但仍存在明显差异。

我们可以像使用 PySpark 列实例一样的方式来处理 Pandas UDF。但是，不能将 Pandas Function

API 与这些列实例一起使用。下面有两个示例说明这些：

```
# Pandas UDF
import pandas as pd
from pyspark.sql.functions import pandas_udf, log2, col

@pandas_udf('long')
def pandas_plus_one(s: pd.Series) -> pd.Series:
    return s + 1

# pandas_plus_one("id") is identically treated as _a SQL expression_ internally.
# Namely, you can combine with other columns, functions and expressions.
spark.range(10).select(
    pandas_plus_one(col("id") - 1) + log2("id") + 1).show()
```

```
# Pandas Function API
from typing import Iterator
import pandas as pd

def pandas_plus_one(iterator: Iterator[pd.DataFrame]) -> Iterator[pd.DataFrame]:
    return map(lambda v: v + 1, iterator)

# pandas_plus_one is just a regular Python function, and mapInPandas is
# logically treated as _a separate SQL query plan_ instead of a SQL expression.
# Therefore, direct interactions with other expressions are impossible.
spark.range(10).mapInPandas(pandas_plus_one, schema="id long").show()
```

另外需要注意的是，Pandas UDF 需要使用 Python 类型提示，而 Pandas Function API 中的类型提示当前是可选的。类型提示目前也支持 Pandas Function API，因为可能以后需要这个。

New Pandas UDFs

新的 Pandas UDF 无需手动定义和指定每个 Pandas UDF 的类型，而是从 Python 函数的给定 Python 类型提示中推断 Pandas UDF 类型。目前 Python UDF 支持以下四种 Python 类型提示：

- Series to Series

- Iterator of Series to Iterator of Series
- Iterator of Multiple Series to Iterator of Series
- Series to Scalar (a single value)

在深入研究每种情况之前，让我们看一下使用新的 Pandas UDF 的三个关键点。

- 尽管在 Python 世界中，Python 类型提示通常是可选的，但是为了使用新的 Python UDF，我们必须为输入和输出指定 Python 类型提示。
- 用户仍然可以通过手动指定 Pandas UDF 类型来使用旧方法。 但是，在 Spark 3.0 鼓励使用 Python 类型提示。
- 在所有情况下，类型提示都应使用 pandas.Series。但是当输入或输出为 StructType 类型应该使用 pandas.DataFrame 作为其输入或输出的类型提示。比如下面的例子：

```
import pandas as pd
from pyspark.sql.functions import pandas_udf


df = spark.createDataFrame(
    [[1, "a string", ("a nested string",)]],
    "long_col long, string_col string, struct_col struct<col1:string>")

@pandas_udf("col1 string, col2 long")
def pandas_plus_len(
        s1: pd.Series, s2: pd.Series, pdf: pd.DataFrame) -> pd.DataFrame:
    # Regular columns are series and the struct column is a DataFrame.
    pdf['col2'] = s1 + s2.str.len()
    return pdf  # the struct column expects a DataFrame to return

df.select(pandas_plus_len("long_col", "string_col", "struct_col")).show()
```

Series to Series

Series to Series 是映射到 Apache Spark 2.3 中引入的标量（scalar）Pandas UDF 里面。类型提示可以表示为 pandas.Series, ... -> pandas.Series。它期望给定的函数接受一个或多个 pandas.Series 并输出一个 pandas.Series。输出长度应该与输入长度相同。

```
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf('long')
def pandas_plus_one(s: pd.Series) -> pd.Series:
    return s + 1
```

```
spark.range(10).select(pandas_plus_one("id")).show()
```

上面的实例可以使用旧的方式实现如下：

```
from pyspark.sql.functions import pandas_udf, PandasUDFType


@pandas_udf('long', PandasUDFType.SCALAR)
def pandas_plus_one(v):
    return v + 1

spark.range(10).select(pandas_plus_one("id")).show()
```

Iterator of Series to Iterator of Series

这是 Apache Spark 3.0 中引入的新型 Pandas UDF。它是 Series to Series 的变体，类型提示可以表示为 Iterator [pd.Series]-> Iterator [pd.Series]。该函数输入并输出 pandas.Series 的迭代器。

整个输出的长度必须与整个输入的长度相同。因此，只要整个输入和输出的长度相同，它就可以从输入迭代器中预取数据。给定的函数应以单列作为输入。

```
from typing import Iterator
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf('long')
def pandas_plus_one(iterator: Iterator[pd.Series]) -> Iterator[pd.Series]:
    return map(lambda s: s + 1, iterator)

spark.range(10).select(pandas_plus_one("id")).show()
```

当 UDF 的执行需要对某个状态进行昂贵的初始化时，它也很有用，下面的伪代码说明了这种情况。

```
@pandas_udf("long")
def calculate(iterator: Iterator[pd.Series]) -> Iterator[pd.Series]:
```

```
    # Do some expensive initialization with a state
    state = very_expensive_initialization()
    for x in iterator:
        # Use that state for the whole iterator.
        yield calculate_with_state(x, state)

df.select(calculate("value")).show()
```

Iterator of Series to Iterator of Series 可以使用旧的方式实现如下：

```
from pyspark.sql.functions import pandas_udf, PandasUDFType


@pandas_udf('long', PandasUDFType.SCALAR_ITER)
def pandas_plus_one(iterator):
    return map(lambda s: s + 1, iterator)

spark.range(10).select(pandas_plus_one("id")).show()
```

Iterator of Multiple Series to Iterator of Series

这种类型的 Pandas UDF 也是在 Apache Spark 3.0 中引入的。这种类型提示可以表示为 Iterator[Tuple[pandas.Series, ...]] -> Iterator[pandas.Series]。

这个使用限制和上面的 Iterator of Series to Iterator of Series 类似。给定函数的输入为 an iterator of a tuple of pandas.Series，输出为 an iterator of pandas.Series。

何时使用某些状态以及何时预取输入数据也很有用。
整个输出的长度也应该与整个输入的长度相同。
但是，给定的函数应该将多列作为输入，这与Series的Iterator到Series的Iterator不同。这个 UDF 使用如下：

```
from typing import Iterator, Tuple
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf("long")
def multiply_two(
        iterator: Iterator[Tuple[pd.Series, pd.Series]]) -> Iterator[pd.Series]:
    return (a * b for a, b in iterator)
```

```
spark.range(10).select(multiply_two("id", "id")).show()
```

旧的方式实现如下：

```
from pyspark.sql.functions import pandas_udf, PandasUDFType



@pandas_udf('long', PandasUDFType.SCALAR_ITER)
def multiply_two(iterator):
    return (a * b for a, b in iterator)

spark.range(10).select(multiply_two("id", "id")).show()
```

Series to Scalar

Series to Scalar 是从 Apache Spark 2.4 引入的。这种类型提示表示为 pandas.Series, ... -> Any。该函数输入为一个或多个 pandas.Series，输出原始数据类型。返回的标量可以是 Python 基本类型，例如 int，float 或NumPy 数据类型，例如 numpy.int64，numpy.float64 等。

```
import pandas as pd
from pyspark.sql.functions import pandas_udf
from pyspark.sql import Window

df = spark.createDataFrame(
    [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)], ("id", "v"))

@pandas_udf("double")
def pandas_mean(v: pd.Series) -> float:
    return v.sum()

df.select(pandas_mean(df['v'])).show()
df.groupby("id").agg(pandas_mean(df['v'])).show()
df.select(pandas_mean(df['v']).over(Window.partitionBy('id'))).show()
```

旧的方式实现如下：

```
import pandas as pd
from pyspark.sql.functions import pandas_udf, PandasUDFType
from pyspark.sql import Window

df = spark.createDataFrame(
   [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)], ("id", "v"))

@pandas_udf("double", PandasUDFType.GROUPED_AGG)
def pandas_mean(v):
   return v.sum()

df.select(pandas_mean(df['v'])).show()
df.groupby("id").agg(pandas_mean(df['v'])).show()
df.select(pandas_mean(df['v']).over(Window.partitionBy('id'))).show()
```

新的 Pandas 函数 APIs（New Pandas Function APIs）

Apache Spark 3.0 中的这一新功能使我们可以直接使用 Python 原生函数（过往记忆大数据，Python native function），该函数将输入输出为 Pandas 实例，而部署 PySpark DataFrame。Apache Spark 3.0 支持的 Pandas Functions API为：grouped map, map, 以及 co-grouped map.

注意，grouped map Pandas UDF 现在是归类为 group map Pandas Function API，正如前面说的，Pandas Function APIs 中的 Python 类型提示当前是可选的。

Grouped Map

Pandas Function API 中的 Grouped map 是 grouped DataFrame 中的 applyInPandas，例如 df.groupby(...)。这已映射到旧的 Pandas UDF 类型的 grouped map Pandas UDF。

```
import pandas as pd

df = spark.createDataFrame(
   [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)], ("id", "v"))

def subtract_mean(pdf: pd.DataFrame) -> pd.DataFrame:
   v = pdf.v
   return pdf.assign(v=v - v.mean())

df.groupby("id").applyInPandas(subtract_mean, schema=df.schema).show()
```

Grouped map type 是映射到 Spark 2.3 的 grouped map Pandas UDF：

```
import pandas as pd
from pyspark.sql.functions import pandas_udf, PandasUDFType

df = spark.createDataFrame(
    [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)], ("id", "v"))

@pandas_udf(df.schema, PandasUDFType.GROUPED_MAP)
def subtract_mean(pdf):
    v = pdf.v
    return pdf.assign(v=v - v.mean())

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

Map

Map Pandas Function API 是 DataFrame 中的 mapInPandas。它是 Apache Spark 3.0 中的新功能。该函数输入 pandas.DataFrame 的迭代器，输出 pandas.DataFrame 的迭代器。

```
from typing import Iterator
import pandas as pd

df = spark.createDataFrame([(1, 21), (2, 30)], ("id", "age"))

def pandas_filter(iterator: Iterator[pd.DataFrame]) -> Iterator[pd.DataFrame]:
    for pdf in iterator:
        yield pdf[pdf.id == 1]

df.mapInPandas(pandas_filter, schema=df.schema).show()
```

Co-grouped Map

Co-grouped map，co-grouped DataFrame 中的 applyInPandas 比如 df.groupby(...).cogroup(df.groupby(...))将在 Apache Spark 3.0 中引入。使用如下：

```
import pandas as pd

df1 = spark.createDataFrame(
    [(1201, 1, 1.0), (1201, 2, 2.0), (1202, 1, 3.0), (1202, 2, 4.0)],
    ("time", "id", "v1"))
df2 = spark.createDataFrame(
    [(1201, 1, "x"), (1201, 2, "y")], ("time", "id", "v2"))

def asof_join(left: pd.DataFrame, right: pd.DataFrame) -> pd.DataFrame:
    return pd.merge_asof(left, right, on="time", by="id")

df1.groupby("id").cogroup(
    df2.groupby("id")
).applyInPandas(asof_join, "time int, id int, v1 double, v2 string").show()
```

本文翻译自New Pandas UDFs and Python Type Hints in the Upcoming Release of Apache Spark 3.0™