Published in BiLD Journal

Diogo Veloso Follow

Dec 21, 2020 · 9 min read · Listen

Pyspark Data Types — Explained

The ins and outs — Data types, Examples, and possible issues

Data types can be divided into 6 *main different data types*:

Main Data Types	Data Types	Pyspark Data Type
Numeric	Byte	ByteType()
	Short	ShortType()
	Integer	IntegerType()
	Long Type	Long TypeType()
	Float	FloatType()
	Double	DoubleType()
	Decimal	DecimalType()
String	String	StringType()
Binary	Binary	BinaryType()
Boolean	Boolean	BooleanType()
Datetime	Timestamp	TimestampType()
	Date	DateType()
Complex	Array	ArrayType()
	Map	MapType()
	Struct	StructType()
	StructField	StructFieldType()

Numeric

ByteType()

Integer Numbers that has *1 byte,* ranges from -128 to 127.

ShortType()

Integer Numbers that has *2 bytes,* ranges from 32768 to 32767.

IntegerType()

Integer Numbers that has *4 bytes,* ranges from
-2147483648 to 2147483647.

LongType()

Integer Number that has *8 bytes,* ranges from
-9223372036854775808 to 9223372036854775807.

FloatType()

Rational Number (Floating-point) that have *4 bytes*

```
#Data representation
10.55
9.333
```

DoubleType()

Rational Number (Floating-point) that have *8 bytes*

```
#Data representation
11.445533
9.333
```

DecimalType() — `DecimalType(int precision, int scale)`

> *"Represents arbitrary-precision signed decimal numbers. Backed internally by* `java.math.BigDecimal`*. A* `BigDecimal` *consists of an arbitrary precision integer* ***unscaled value*** *and a 32-bit* ***integer scale****."*

But what it really means? Let's break it down:

DecimalType() stores two operands (Precision and Scale), this way avoids storing trailing zeros.

- Precision — Number of digits in the Unscaled value
- Unscaled value — Value without the floating-point (i.e 4.33 the unscaled value would be 433)
- Scale — Number of digits to the right of the decimal point (i.e 4.33 the scale is 2)

```
from pyspark.sql.types import DecimalType
from decimal import Decimal

#Example1
Value = 4333.1234
Unscaled_Value = 43331234
Precision = 6
Scale = 2

Value_Saved = 4333.12

schema = StructType([
            StructField('column1', IntegerType()),
            StructField('column2', DecimalType(Precision, Scale ))])
```

```
data = [(1 , Decimal(Value))]
df = spark.createDataFrame(data, schema=schema)
```

	column1	column2
1	1	4333.12

```
#Example2
Precision = 6
Scale = 3

# In this example it will throw an error, because if the precision is 6 we
have 4333.12, which only has 2 scale digits

schema = StructType([
            StructField('column1', IntegerType()),
            StructField('column2', DecimalType(Precision, Scale ))])

data = [(1 , Decimal(Value))]
df = spark.createDataFrame(data, schema=schema)
```

The precision can be up to 38, the scale can also be up to 38 (less or equal to precision)

Possible Issues with operations with decimal numbers

Let's take a multiplication example:

```
from pyspark.sql.types import DecimalType
from decimal import Decimal
import pyspark.sql.functions as F

schema = StructType([
            StructField('Exchange_Rate', DecimalType(38, 10 )),
            StructField('Value_Euro', DecimalType(38, 10 ))])
data = [(Decimal(1.21) , Decimal(4333.12 ))]
df = spark.createDataFrame(data, schema=schema)

# Multiplication
df = df.withColumn('Value_Dollar', F.col('Exchange_Rate') *
F.col('Value_Euro'))
```

```
df: pyspark.sql.dataframe.DataFrame
    Exchange_Rate: decimal(38,10)
    Value_Euro: decimal(38,10)
    Value_Dollar: decimal(38,6)
```

	Exchange_Rate	Value_Euro	Value_Dollar
1	1.21	4333.12	5243.0752

We can see that, we created a new column by multiplying 2 columns, each of the original ones have precision = 38 and scale = 10, but the result of that multiplication have precision = 38 and scale = 6. Shouldn't the new column have the same values for precision and scale as the columns that originated it?

Under the hood

Let's check the code that "calculates and propagates precision for fixed-precision decimals."

apache/spark

Apache Spark - A unified analytics engine for large-scale data processing - apache/spark

github.com

"If we have expressions e1 and e2 with precision/scale, p1/s1 and p2/s2 respectively, then the following operations have the following precision/ scale":

```
Operation    Result Precision                        Result Scale
------------------------------------------------------------------------
e1 + e2      max(s1, s2) + max(p1-s1, p2-s2) + 1     max(s1, s2)
e1 - e2      max(s1, s2) + max(p1-s1, p2-s2) + 1     max(s1, s2)
e1 * e2      p1 + p2 + 1                             s1 + s2
e1 / e2      p1 - s1 + s2 + max(6, s1 + p2 + 1)      max(6, s1 + p2 + 1)
e1 % e2      min(p1-s1, p2-s2) + max(s1, s2)         max(s1, s2)
e1 union e2  max(s1, s2) + max(p1-s1, p2-s2)         max(s1, s2)
```

Given the multiplication example we have:

```
p1 = 38
p2 = 38
s1 = 10
s2 = 10

Precision = p1 + p2 + 1
Scale = s1 + s2

Precision = 86
Scale = 20
```

The maximum value for precision is 38, in this case, it surpasses that value

apache/spark

Apache Spark - A unified analytics engine for large-scale data processing - apache/spark

github.com

```
val MAX_PRECISION = 38
val MAX_SCALE = 38
val SYSTEM_DEFAULT: DecimalType = DecimalType(MAX_PRECISION, 18)
val USER_DEFAULT: DecimalType = DecimalType(10, 0)
val MINIMUM_ADJUSTED_SCALE = 6
```

```scala
 * This method is used only when `spark.sql.decimalOperations.allowPrecisionLoss` is set to true.
 */
private[sql] def adjustPrecisionScale(precision: Int, scale: Int): DecimalType = {
  // Assumption:
  assert(precision >= scale)

  if (precision <= MAX_PRECISION) {
    // Adjustment only needed when we exceed max precision
    DecimalType(precision, scale)
  } else if (scale < 0) {
    // Decimal can have negative scale (SPARK-24468). In this case, we cannot allow a precision
    // loss since we would cause a loss of digits in the integer part.
    // In this case, we are likely to meet an overflow.
    DecimalType(MAX_PRECISION, scale)
  } else {
    // Precision/scale exceed maximum precision. Result must be adjusted to MAX_PRECISION.
    val intDigits = precision - scale
    // If original scale is less than MINIMUM_ADJUSTED_SCALE, use original scale value; otherwise
    // preserve at least MINIMUM_ADJUSTED_SCALE fractional digits
    val minScaleValue = Math.min(scale, MINIMUM_ADJUSTED_SCALE)
    // The resulting scale is the maximum between what is available without causing a loss of
    // digits for the integer part of the decimal and the minimum guaranteed scale, which is
    // computed above
    val adjustedScale = Math.max(MAX_PRECISION - intDigits, minScaleValue)

    DecimalType(MAX_PRECISION, adjustedScale)
  }
}
```

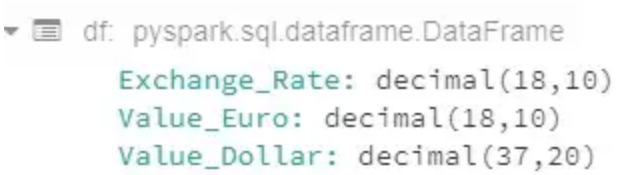
We have a precision > MAX_PRECISION and scale > 0:

```
precision = 86
scale = 20

inDigits = 66
minScaleValue = min(20, 6) = 6
adjustedScale = max(38 - 66, 6) = 6

DecimalType(38, 6)
```

That's why the new column, changes the scale, if we want to maintain the scale we need to have the sum of the two precisions ≤ MAX_PRECISION +1

```
e1 * e2      p1 + p2 + 1                              s1 + s2
```

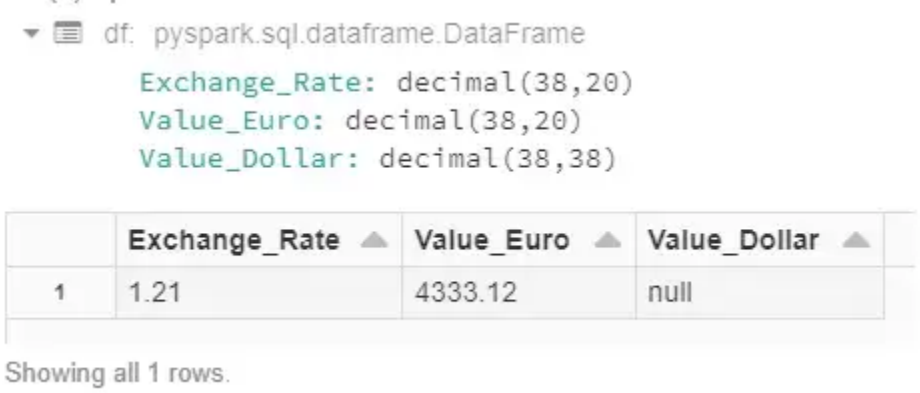
```
from pyspark.sql.types import DecimalType
from decimal import Decimal
import pyspark.sql.functions as F

Precision = (38/2) -1

schema = StructType([
         StructField('Exchange_Rate', DecimalType(Precision, 10 )),
         StructField('Value_Euro', DecimalType(Precision, 10 ))])

data = [(Decimal(1.21) , Decimal(4333.12 ))]

df = spark.createDataFrame(data, schema=schema)

# Multiplication
df = df.withColumn('Value_Dollar', F.col('Exchange_Rate') *
F.col('Value_Euro'))
```



Issue solved

— config spark.sql.decimalOperations.allowPrecisionLoss " if set to `false` , Spark uses previous rules, ie. it doesn't adjust the needed scale to represent the values and it returns NULL if an exact representation of the value is not possible.", as the example shows:

```
spark.conf.set("spark.sql.decimalOperations.allowPrecisionLoss", False)

schema = StructType([
            StructField('Exchange_Rate', DecimalType(38, 20 )),
            StructField('Value_Euro', DecimalType(38, 29))])
data = [(Decimal(1.21) , Decimal(4333.12 ))]
df = spark.createDataFrame(data, schema=schema)
df = df.withColumn('Value_Dollar', F.col('Exchange_Rate') *
F.col('Value_Euro'))
```



Double x Decimal

- Double has a certain precision
- Decimal is an exact way of representing numbers

If we sum values with various magnitudes(i.e 10000.0 and 0.00001), decimal digits can be dropped when summing them, with Decimal it will not happen

If precision is needed Decimal is the Data type to use, if not, Double will do the job. Decimal will be slower because it will store more data.

String

StringType()
Character Values, it's used to represent text, it can include letters, numbers, and characters symbols. Anything in quotes is treated as text data by the interpreter (except the escape sequences).

```
#Data representation
"Asdds!99"
"9998"
```

Double Quotes vs Single Quotes

You can use either Double(“”) or Single Quotes(‘), it will not make a difference most of the times, there is a specific case to use Single or Double Quotes, whenever you have a quoting in your text.

And then Jonh said: “I will buy that shirt!”

To have this phrase in a String value you would need to add quotes, the issue is that the text itself already has quotes, the way you can use quotes inside of a string is by **quoting with the opposite type of quotes** or **using triple quotes** or **using escape sequences.**

```
'''And then Jonh said: “I will buy that shirt!”'''
'And then Jonh said: "I will buy that shirt!"'
"And then Jonh said: \"I will buy that shirt!\""
```

String Interpolation (String Interpolation Method)

Whenever you need to add a variable to a string, you can use the prefix f (formating) and curly braces f”{}” (there are other methods to add a variable to a string, but this one is easy to use and increase the code readability) :

```
b = "John"
f"And then {b} said"
```

```
Out[3]: 'And then John said'
```

Escape Sequences

Escape Sequences starts with a backslash(\) and are interpreted differently.

```
"And then Jonh said: \"I will buy that shirt!\""
```

```
Out[4]: 'And then Jonh said: "I will buy that shirt!"'
```

You can check the link below for all Escape Sequences

https://www.python-ds.com/python-3-escape-sequences

Raw String

To ignore Escape Sequences just add r before the string

\t = ASCII Horizontal Tab (TAB) — adds a tab

```
print("And\then Jonh said")
```

```
And     hen Jonh said
```

```
print(r"And\then Jonh said")
```

```
And\then Jonh said
```

Binary

BinaryType()

binary values

```
#Data representation
bin(30)
```

```
0b11110
```

Binary representation of 30

Boolean

BooleanType()

Represents 2 values False or True, it can be also 0 (False) or 1 (True)

```
#Data representation
a =True
b = False
a == b
```

```
Out[15]: False
```

Datetime

TimestampType()

Represents values of the fields year, month, day, hour, minute, and second, with the local time-zone, according to your machine's local time zone, it can be changed by `spark.sql.session.timeZone`, ie:

```
spark.conf.set('spark.sql.session.timeZone', 'Europe/Paris')
```

The timestamp value represents an absolute point in time

```
import datetime

#Data representation
print(datetime.datetime.now())
```

```
2020-12-06 14:04:59.169814
```

Datetype()

Represents values of fields year, month, and day, without a local time-zone

```
import datetime

#Data representation
print(datetime.date(1994, 9, 8))
```

```
1994-09-08
```

Complex

ArrayType (elementType, containsNull)

The sequence of elements with the type of `elementType`. `containsNull` is used to indicate if elements in a `ArrayType` value can have `null` values. Is the equivalent in Python to a list.

```
#Data representation
["Attacking Midfielder"," Midfielder (Centre)"]
```

MapType(keyType, valueType, valueContainsNull)

The data type of keys is described by `keyType` and the data type of values is described by `valueType`. StrucType. Is the equivalent in Python to a dictionary.

```
#Data representation
{'Place of Birth':'Portugal', 'Squad\'s Country':'England'}
```

StructType(fields)

Represents values with the structure described by a sequence of StructFields (fields), StructType can be seen as the schema of a Dataframe

StructField(name, dataType, nullable)

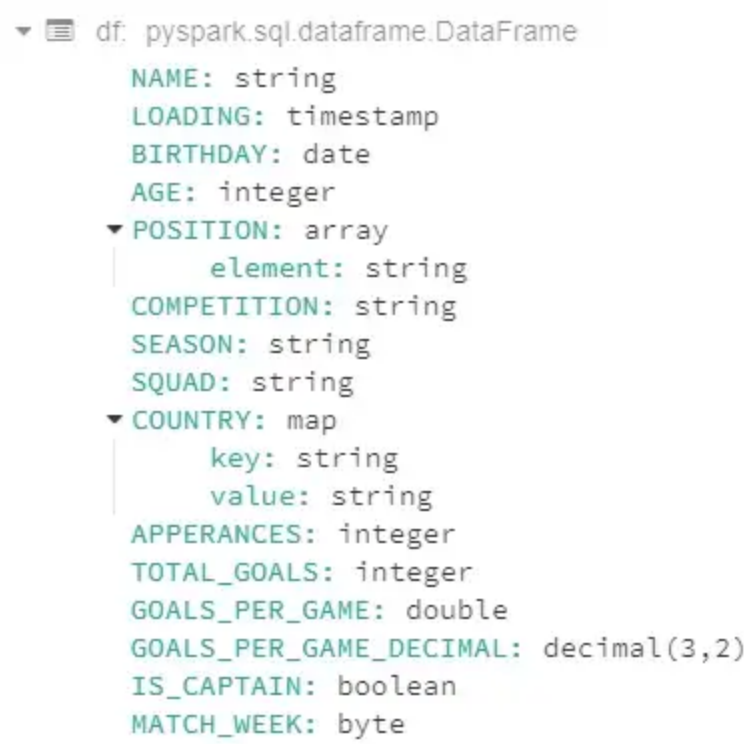
Represents a field in a StructType. The name of a field is indicated by name. The data type of a field is indicated by dataType. nullable is used to indicate if the values of these fields can have null values. StructField can be seen as the schema of a single column in a Dataframe.

Let's take some of the data types that we have learned and create a Dataframe, with the stats of the football player, Bruno Fernandes

Photo by Alex Motoc on Unsplash

```
import datetime
from decimal import *
from pyspark.sql.types import *

schema = StructType([\
          StructField("NAME", StringType(),True), \
          StructField("LOADING", TimestampType(),True), \
          StructField("BIRTHDAY", DateType(),True), \
          StructField("AGE", IntegerType(),True), \
          StructField("POSITION", ArrayType(StringType()),True), \
          StructField("COMPETITION", StringType(),True), \
          StructField("SEASON", StringType(),True), \
          StructField("SQUAD", StringType(),True), \
          StructField("COUNTRY", MapType(StringType(),StringType()),True), \
          StructField("APPERANCES", IntegerType(),True), \
          StructField("TOTAL_GOALS", IntegerType(),True), \
          StructField("GOALS_PER_GAME", DoubleType(),True), \
          StructField("GOALS_PER_GAME_DECIMAL", DecimalType(precision = 3,scale
= 2),True), \
          StructField("IS_CAPTAIN", BooleanType(),True), \
          StructField("MATCH_WEEK", ByteType(),True), \

])

data = [("Bruno Fernandes",
          datetime.datetime.now() ,
          datetime.date(1994, 9, 8) ,
          26 ,
          ["Attacking Midfielder"," Midfielder (Centre)"] ,
```

```
            "Premier League",
            "2019/2020",
            "Man Utd",
            {'Place of Birth':'Portugal', 'Squad\'s Country':'England'},
            9,
            7,
            9/7,
            Decimal(9)/Decimal(7),
            True,
            10
           )]

df = spark.createDataFrame(data=data,schema=schema)
```



Schema

	NAME	LOADING	BIRTHDAY	AGE	POSITION	COMPETITION	SEASON	SQUAD	COUNTRY
1	Bruno Fernandes	2020-12-06T15:18:17.848+0000	1994-09-08	26	["Attacking Midfielder", "Midfielder (Centre)"]	Premier League	2019/2020	Man Utd	{"Squad's Country": "England", "Place of Birth": "Portugal"}

APPERANCES	TOTAL_GOALS	GOALS_PER_GAME	GOALS_PER_GAME_DECIMAL	IS_CAPTAIN	MATCH_WEEK
9	7	1.2857142857142858	1.29	true	10

Cast

With Cast you can change the columns DataType, Cast can be used with:

- **selectExpr()**

- **withColumn()**
- **SQL**

```
import pyspark.sql.functions as F

# selectExpr()
df2 = df.selectExpr("cast(AGE as string) AGE")

# withColumn()
df2 = df.withColumn("AGE", F.col("AGE").cast(StringType()) ).select("AGE")

# SQL
df.createOrReplaceTempView("AGE_Cast")
df2 = spark.sql("SELECT STRING(AGE) from AGE_Cast")
```

InferSchema

Since we are talking about Schemas and Datatypes, I think it's important to talk about InferSchema.

InferSchema is an option in the *read function* where it guesses the datatypes in the file that you're reading, ie:

```
df = spark.read.format(file_type) \
  .option("inferSchema", True) \
  .option("header", True) \
  .option("sep", ";") \
  .load(file_location)
```

Now if you want to manually write the schema, we have learned that:

```
Schema = StructType([\
         StructField("NAME", StringType(),True), \
         StructField("LOADING", TimestampType(),True)]

df = spark.read.format(file_type) \
  .schema(Schema)
  .option("header", True) \
  .option("sep", ";") \
  .load(file_location)
```

NullType()

There is another data type that dataframes accept, when we use the InferSchema and the values are all null, Spark will use NullType to infer. If you force one column to be *null* it will also change the datatype to be *Null*

```
df2 = df.withColumn('AGE' , F.lit(None)).select('AGE')
```

Sources:

Data Types Spark SQL and DataFrames support the following data types: Numeric types ByteType: Represents 1-byte signed integer... spark.apache.org	

https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html#:~:text=Class%20BigDecimal&text=Immutable%2C%20arbitrary%2Dprecision%20signed%20decimal,right%20of%20the%20decimal%20point.

PySpark; DecimalType multiplication precision loss	

I think it is expected behavior. Spark's Catalyst engine converts an expression written in an input language (e.g...

stackoverflow.com

DecimalType (Spark 2.2.0 JavaDoc)

The precision can be up to 38, scale can also be up to 38 (less or equal to precision).

spark.apache.org

Spark SQL Upgrading Guide

Pyspark Databricks Technology