ADVANCED USAGE PATTERNS OF SCALA UDF IN PYSPARK

Andrey Titov

Обо мне

- Data Engineering
- Systems Engineering
- Apache Spark
- Scala
- Python

- Apache Spark – набор библиотек для распределенной обработки данных
- Один из самых популярных проектов в области Big Data
- SQL, включающий в себя:
  - *Оконные функции*
  - *Соединения*
  - *Встроенные функции*
  - *Пользовательские функции*
- Поддержка поточной обработки данных
- Можно писать на нескольких языках одновременно (Python, Scala)
- Наличие коннекторов для работы с большинством баз и форматов данных

О докладе

- Основные причины низкой производительности PySpark приложений
- Применимость пользовательских функций
- Виды и особенности UDF в Spark
- Недостатки «стандартного» способа использование Scala UDF
- Альтернативный способ создания Scala UDF и его преимущества

Почему тормозит PySpark приложение

- Использование RDD API
- Недостаточное количество партиций
- Перекосы данных
- Не хватает ресурсов
- Низкая утилизация ресурсов
- CartesianProduct
- Неоптимальное использование UDF

ПОЛЬЗОВАТЕЛЬСКИЕ ФУНКЦИИ

Общие сведения

- Любые UDF (даже Scala) замедляют обработку данных
- Данные DF всегда хранятся внутри JVM, даже когда используется PySpark
- По возможности необходимо использовать функции:
  - *pyspark.sql.functions*
  - *SQL built-in functions*

```
1 from pyspark.sql.functions import from_json, lit, udf, col
2 from pyspark.sql.types import *
3 import json
4
5 df = spark.range(1).select(lit("""{"foo": "bar"}""").alias("value"))
6 df.show()
7
8 r_type = StructType([StructField("foo", StringType())])
```

```
+--------------+
|         value|
+--------------+
|{"foo": "bar"}|
+--------------+
```

```
1 def parse_json(data):
2     return json.loads(data)
3
4 json_udf = udf(parse_json, r_type)
5
6 df.select(json_udf(col("value"))).show()
```

```
+-----------------+
|parse_json(value)|
+-----------------+
|            [bar]|
+-----------------+
```

```
1 f_json = from_json(col("value"), r_type)
2 df.select(f_json).show()
```

```
+--------------------+
|jsontostructs(value)|
+--------------------+
|                [bar]|
+--------------------+
```

Row-at-a-time UDF

- Каждый воркер форкает несколько **pyspark.daemon** процессов
- Данные на воркерах передаются между JVM и Python через пайп с использованием Pickle SerDe
- Функция и ее зависимости на драйвере передается на воркеры по сети с использованием Pickle SerDe
- Возвращаемая схема указывается явно

```
1  from pyspark.sql.functions import col, udf
2
3  my_udf = udf(lambda x: x * 2, IntegerType())
4  spark.range(10).select(my_udf(col("id"))).show()
```

```
+------------+
|<lambda>(id)|
+------------+
|           0|
|           2|
|           4|
|           6|
|           8|
|          10|
|          12|
|          14|
|          16|
|          18|
+------------+
```

```
== Physical Plan ==
*(2) Project [pythonUDF0#75 AS <lambda>(id)#73]
+- BatchEvalPython [<lambda>(id#70L)], [id#70L, pythonUDF0#75]
   +- *(1) Range (0, 10, step=1, splits=6)
```

Row-at-a-time UDF

```
case class BatchEvalPythonExec(udfs: Seq[PythonUDF], output: Seq[Attribute], child: SparkPlan)
  extends EvalPythonExec(udfs, output, child) {

  protected override def evaluate(
      funcs: Seq[ChainedPythonFunctions],
      argOffsets: Array[Array[Int]],
      iter: Iterator[InternalRow],
      schema: StructType,
      context: TaskContext): Iterator[InternalRow] = {
    EvaluatePython.registerPicklers()  // register pickler for Row

    val dataTypes = schema.map(_.dataType)
    val needConversion = dataTypes.exists(EvaluatePython.needConversionInPython)

    // enable memo iff we serialize the row with schema (schema and class should be memorized)
    val pickle = new Pickler(needConversion)
    ...
    val inputIterator = iter.map {...}.grouped(100).map(x => pickle.dumps(x.toArray))

    // Output iterator for results from Python.
    val outputIterator = new PythonUDFRunner(funcs, PythonEvalType.SQL_BATCHED_UDF, argOffsets)
      .compute(inputIterator, context.partitionId(), context)

    val unpickle = new Unpickler
    val mutableRow = new GenericInternalRow( size = 1)
    val resultType = if (udfs.length == 1) {...} else {...}

    val fromJava = EvaluatePython.makeFromJava(resultType)

    outputIterator.flatMap {...}.map {...}
  }
}
```

```
import net.razorvine.pickle.{Pickler, Unpickler}
```

```
EvaluatePython.toJava(row, schema)
```

Vectorized UDF

- На вход в функцию подаются вектора pandas.Series
- Для сериализации данных вместо Pickle используется Arrow
- Возвращаемая схема указывается явно

```
1  from pyspark.sql.functions import col, pandas_udf
2
3  my_udf = pandas_udf(lambda x: x * 2, IntegerType())
4  spark.range(10).select(my_udf(col("id"))).show()
```

```
+------------+
|<lambda>(id)|
+------------+
|           0|
|           2|
|           4|
|           6|
|           8|
|          10|
|          12|
|          14|
|          16|
|          18|
+------------+
```

```
== Physical Plan ==
*(2) Project [pythonUDF0#91 AS <lambda>(id)#89]
+- ArrowEvalPython [<lambda>(id#86L)], [id#86L, pythonUDF0#91]
   +- *(1) Range (0, 10, step=1, splits=6)
```

Vectorized UDF

```scala
case class ArrowEvalPythonExec(udfs: Seq[PythonUDF], output: Seq[Attribute], child: SparkPlan)
  extends EvalPythonExec(udfs, output, child) {

  private val batchSize = conf.arrowMaxRecordsPerBatch
  private val sessionLocalTimeZone = conf.sessionLocalTimeZone
  private val pythonRunnerConf = ArrowUtils.getPythonRunnerConfMap(conf)

  protected override def evaluate(
      funcs: Seq[ChainedPythonFunctions],
      argOffsets: Array[Array[Int]],
      iter: Iterator[InternalRow],
      schema: StructType,
      context: TaskContext): Iterator[InternalRow] = {

    val outputTypes = output.drop(child.output.length).map(_.dataType)

    // DO NOT use iter.grouped(). See BatchIterator.
    val batchIter = if (batchSize > 0) new BatchIterator(iter, batchSize) else Iterator(iter)

    val columnarBatchIter = new ArrowPythonRunner(...).compute(batchIter, context.partitionId(), context)

    new Iterator[InternalRow] {

      private var currentIter = if (columnarBatchIter.hasNext) {...} else {...}

      override def hasNext: Boolean = ...

      override def next(): InternalRow = currentIter.next()
    }
  }
}
```

Vectorized UDF

```
class ArrowPythonRunner(
    funcs: Seq[ChainedPythonFunctions],
    evalType: Int,
    argOffsets: Array[Array[Int]],
    schema: StructType,
    timeZoneId: String,
    conf: Map[String, String])
  extends BasePythonRunner[Iterator[InternalRow], ColumnarBatch](...) {

  protected override def newWriterThread(
      env: SparkEnv,
      worker: Socket,
      inputIterator: Iterator[Iterator[InternalRow]],
      partitionIndex: Int,
      context: TaskContext): WriterThread = {
    new WriterThread(env, worker, inputIterator, partitionIndex, context) {

      protected override def writeCommand(dataOut: DataOutputStream): Unit = {...}

      protected override def writeIteratorToStream(dataOut: DataOutputStream): Unit = {
        val arrowSchema = ArrowUtils.toArrowSchema(schema, timeZoneId)
        val allocator = ArrowUtils.rootAllocator.newChildAllocator(...)
        val root = VectorSchemaRoot.create(arrowSchema, allocator)

        Utils.tryWithSafeFinally {
          val arrowWriter = ArrowWriter.create(root)
          val writer = new ArrowStreamWriter(root, provider = null, dataOut)
          writer.start()

          while (inputIterator.hasNext) {
            val nextBatch = inputIterator.next()

            while (nextBatch.hasNext) {
              arrowWriter.write(nextBatch.next())
            }
```

val nextBatch: Iterator[InternalRow]

SCALA UDF

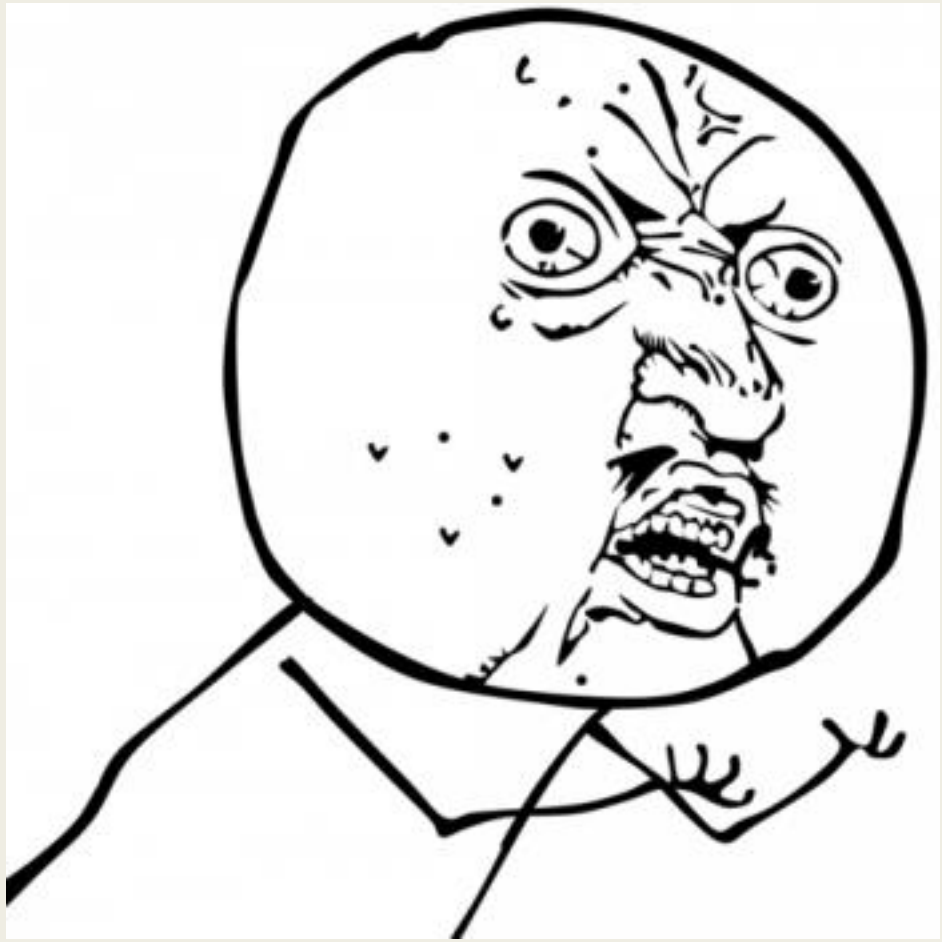
Стандартный вариант

- Нужно явно регистрировать
- Необходимо явно указать возвращаемый тип в трех местах
- Несовпадение типов выявляется только в рантайме
- Необходимо указать тип входных колонок в двух местах
- Использование:
  - *spark.sql(...)*
  - *expr(...)*
- Без использования select нельзя передать в функцию произвольную колонку **pyspark.sql.Column**

```python
from pyspark.sql.functions import expr
from pyspark.sql.types import *

spark.udf.registerJavaFunction(
    "my_udf",
    "local.spark.MyStandardFunc",
    IntegerType())

spark.range(10).select(
    col("id").cast("int").alias("a"),
    col("id").cast("int").alias("b"),
    col("id").cast("int").alias("c")
).select(expr("""my_udf(a, b, c)""")).show(2)
```

```
+------------------+
|UDF:my_udf(a, b, c)|
+------------------+
|                 0|
|                 3|
+------------------+
```

```scala
package local.spark

import org.apache.spark.sql.api.java.UDF3

class MyStandardFunc extends UDF3[Int, Int, Int, Int] {
  def call(t1: Int, t2: Int, t3: Int): Int = t1 + t2 + t3
}
```



А попроще нельзя было?

Альтернативный вариант

- Не нужно явно указывать возвращаемый тип в Python – определится автоматически
- Не нужно регистрировать – просто используйте когда надо
- Поддерживает **pyspark.sql.Column** – не нужны дополнительные обертки в виде spark.sql() и expr()
- Нет лишних нагромождений в Scala
- FQDN можно параметризовать через getattr()

```python
from pyspark.sql.column import Column, _to_seq, _to_java_column
from pyspark.sql.functions import col

def call(*args):
    java_function = sc._jvm.local.spark.MyAwesomeFunc.getSum()
    input_cols = _to_seq(sc, args, _to_java_column)
    udf_call = java_function.apply(input_cols)
    result_col = Column(udf_call)
    return result_col

spark.range(10).select(
    col("id").cast("int").alias("a"),
    col("id").cast("int").alias("b"),
    col("id").cast("int").alias("c")
).select(call(col("a"), col("b"), col("c"))).show(2)
```

```
+------------+
|UDF(a, b, c)|
+------------+
|           0|
|           3|
+------------+
```

```scala
package local.spark

import org.apache.spark.sql.expressions.UserDefinedFunction
import org.apache.spark.sql.functions.udf

object MyAwesomeFunc {
  def getSum: UserDefinedFunction = udf { (a: Int, b: Int, c: Int) => a + b + c }
}
```

```
== Physical Plan ==
*(1) Project [UDF(cast(id#112L as int), cast(id#112L as int), cast(id#112L as int)) AS UDF(a, b, c)#120]
+- *(1) Range (0, 10, step=1, splits=6)
```

ОСОБЕННОСТИ ПРИМЕНЕНИЯ

Настройка логирования

- log4j встроен в Spark
- Настройка через **conf/log4j.properties**
- Добавить **log4j.properties** на воркеры и драйвер
- Настроить опцию **-Dlog4j.configuration**

```scala
package local.spark

import org.apache.spark.internal.Logging
import org.apache.spark.sql.expressions.UserDefinedFunction
import org.apache.spark.sql.functions.udf

object MyAwesomeFunc extends Logging {
  log.debug("MyAwesomeFunc is awesome!")

  def getSum: UserDefinedFunction = udf { (a: Int, b: Int, c: Int) => a + b + c }
}
```

```
[spark-2.4.5-bin-hadoop2.7] cat conf/log4j.properties | grep MyAwesomeFunc
log4j.logger.local.spark.MyAwesomeFunc=DEBUG
```

```
spark.driver.extraJavaOptions -Dlog4j.configuration=file:logger/log4j.properties
spark.executor.extraJavaOptions -Dlog4j.configuration=file:logger/log4j.properties
spark.yarn.dist.archives /path/to/logger.zip#logger
```

Использование Singleton Pattern

- Контроль работы воркера
- Работа с БД
- Локальное хранилище фактов
- Локальный кеш
- Быстрый Stateful Streaming
- Альтернатива BroadcastHashJoin

```scala
package local.spark

import org.apache.spark.internal.Logging
import org.apache.spark.sql.expressions.UserDefinedFunction
import org.apache.spark.sql.functions.udf

class MyAweSomeDatabase() {
  def select(query: String): Int = ???
}

object MyAwesomeFunc extends Logging {
  log.debug("MyAwesomeFunc is awesome!")

  val myDb = new MyAweSomeDatabase()

  def getSum: UserDefinedFunction = udf { (a: Int, b: Int, c: Int) =>
    val foo = myDb.select( query = """SELECT foo FROM bar""")
    a + b + c + foo }
}
```

СПАСИБО!