# Spark SQL

## INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski Phd**
Data Scientist

datacamp

# Load a dataframe from file

```
df = spark.read.csv(filename)
```

```
df = spark.read.csv(filename, header=True)
```

# Create SQL table and query it

```
df.createOrReplaceTempView("schedule")
spark.sql("SELECT * FROM schedule WHERE station = 'San Jose'")
     .show()
```

```
+--------+--------+-----+
|train_id| station| time|
+--------+--------+-----+
|     324|San Jose|9:05a|
|     217|San Jose|6:59a|
+--------+--------+-----+
```

# Inspecting table schema

```
result = spark.sql("SHOW COLUMNS FROM tablename")
```

```
result = spark.sql("SELECT * FROM tablename LIMIT 0")
```

```
result = spark.sql("DESCRIBE tablename")
```

```
result.show()
```

```
print(result.columns)
```

# Tabular data

```
+--------+-------------+-----+
|train_id|      station| time|
+--------+-------------+-----+
|     324|San Francisco|7:59a|
|     324|  22nd Street|8:03a|
|     324|     Millbrae|8:16a|
|     324|    Hillsdale|8:24a|
|     324| Redwood City|8:31a|
|     324|    Palo Alto|8:37a|
|     324|     San Jose|9:05a|
|     217|       Gilroy|6:06a|
|     217|   San Martin|6:15a|
|     217|  Morgan Hill|6:21a|
|     217| Blossom Hill|6:36a|
|     217|      Capitol|6:42a|
|     217|       Tamien|6:50a|
|     217|     San Jose|6:59a|
+--------+-------------+-----+
```

# Loading delimited text

Loads a comma-delimited file `trainsched.txt` into a dataframe called `df` :

```
df = spark.read.csv("trainsched.txt", header=True)
```

# Loading delimited text

```
df = spark.read.csv("trainsched.txt", header=True)
df.show()
```

```
+--------+-------------+-----+
|train_id|      station| time|
+--------+-------------+-----+
|     324|San Francisco|7:59a|
|     324|  22nd Street|8:03a|
|     324|     Millbrae|8:16a|
|     324|    Hillsdale|8:24a|
|     324| Redwood City|8:31a|
|     ...|          ...|  ...|
|     217| Blossom Hill|6:36a|
|     217|      Capitol|6:42a|
|     217|       Tamien|6:50a|
|     217|     San Jose|6:59a|
+--------+-------------+-----+
```

# Window Function SQL

## INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# What is a Window Function SQL?

- Express operations more simply than dot notation or queries
- Each row uses the values of other rows to calculate its value

# A train schedule

| train_id | station | time |
|---|---|---|
| 324 | San Francisco | 7:59 |
| 324 | 22nd Street | 8:03 |
| 324 | Millbrae | 8:16 |
| 324 | Hillsdale | 8:24 |
| 324 | Redwood City | 8:31 |
| 324 | Palo Alto | 8:37 |
| 324 | San Jose | 9:05 |

# Column with time until next stop added

| train_id | station | time | time_to_next_stop |
|---|---|---|---|
| 324 | San Francisco | 7:59 | 4 min |
| 324 | 22nd Street | 8:03 | 13 min |
| 324 | Millbrae | 8:16 | 8 min |
| 324 | Hillsdale | 8:24 | 7 min |
| 324 | Redwood City | 8:31 | 6 min |
| 324 | Palo Alto | 8:37 | 28 min |
| 324 | San Jose | 9:05 | null |

# Column with time of next stop

| train_id | station | time | time (following row) |
|---|---|---|---|
| 324 | San Francisco | 7:59 | 8:03 |
| 324 | 22nd Street | 8:03 | 8:16 |
| 324 | Millbrae | 8:16 | 8:24 |
| 324 | Hillsdale | 8:24 | 8:31 |
| 324 | Redwood City | 8:31 | 8:37 |
| 324 | Palo Alto | 8:37 | 9:05 |
| 324 | San Jose | 9:05 | null |

# OVER clause and ORDER BY clause

```
query = """
SELECT train_id, station, time,
LEAD(time, 1) OVER (ORDER BY time) AS time_next
FROM sched
WHERE train_id=324 """
spark.sql(query).show()
```

```
+--------+-------------+-----+---------+
|train_id|      station| time|time_next|
+--------+-------------+-----+---------+
|     324|San Francisco|7:59 |    8:03 |
|     324|  22nd Street|8:03 |    8:16 |
|     324|     Millbrae|8:16 |    8:24 |
|     324|    Hillsdale|8:24 |    8:31 |
|     324| Redwood City|8:31 |    8:37 |
|     324|    Palo Alto|8:37 |    9:05 |
|     324|     San Jose|9:05 |    null |
+--------+-------------+-----+---------+
```

# PARTITION BY clause

```
SELECT
train_id,
station,
time,
LEAD(time,1) OVER (PARTITION BY train_id ORDER BY time) AS time_next
FROM sched
```

# Result of adding PARTITION BY clause

```
+--------+-------------+-----+---------+
|train_id|      station| time|time_next|
+--------+-------------+-----+---------+
|     217|       Gilroy|6:06 |    6:15 |
|     217|   San Martin|6:15 |    6:21 |
|     217|  Morgan Hill|6:21 |    6:36 |
|     217| Blossom Hill|6:36 |    6:42 |
|     217|      Capitol|6:42 |    6:50 |
|     217|       Tamien|6:50 |    6:59 |
|     217|     San Jose|6:59 |    null |
|     324|San Francisco|7:59 |    8:03 |
|     324|  22nd Street|8:03 |    8:16 |
|     324|     Millbrae|8:16 |    8:24 |
|     324|    Hillsdale|8:24 |    8:31 |
|     324| Redwood City|8:31 |    8:37 |
|     324|    Palo Alto|8:37 |    9:05 |
|     324|     San Jose|9:05 |    null |
+--------+-------------+-----+---------+
```

| train_id | station | time | time_to_next_stop |
|---|---|---|---|
| 324 | San Francisco | 7:59 | 4 min |
| 324 | 22nd Street | 8:03 | 13 min |
| 324 | Millbrae | 8:16 | 8 min |
| 324 | Hillsdale | 8:24 | 7 min |
| 324 | Redwood City | 8:31 | 6 min |
| 324 | Palo Alto | 8:37 | 28 min |
| 324 | San Jose | 9:05 | null |

# Dot notation and SQL

INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# Our table has 3 columns

```
df.columns
```

```
['train_id', 'station', 'time']
```

```
df.show(5)
```

```
+--------+-------------+-----+
|train_id|      station| time|
+--------+-------------+-----+
|     324|San Francisco|7:59 |
|     324|  22nd Street|8:03 |
|     324|     Millbrae|8:16 |
|     324|    Hillsdale|8:24 |
|     324| Redwood City|8:31 |
+--------+-------------+-----+
```

# We only need 2

```
df.select('train_id','station')
  .show(5)
```

```
+--------+-------------+
|train_id|      station|
+--------+-------------+
|     324|San Francisco|
|     324|  22nd Street|
|     324|     Millbrae|
|     324|    Hillsdale|
|     324| Redwood City|
+--------+-------------+
```

# Three ways to select 2 columns

- `df.select('train_id', 'station')`
- `df.select(df.train_id, df.station)`
- `from pyspark.sql.functions import col`
- `df.select(col('train_id'), col('station'))`

# Two ways to rename a column

```
df.select('train_id','station')
  .withColumnRenamed('train_id','train')
  .show(5)
```

```
+-----+-------------+
|train|      station|
+-----+-------------+
|  324|San Francisco|
|  324|  22nd Street|
|  324|     Millbrae|
|  324|    Hillsdale|
|  324| Redwood City|
+-----+-------------+
```

```
df.select(col('train_id').alias('train'), 'station')
```

# Don’t do this!

```
df.select('train_id', df.station, col('time'))
```

# SQL queries using dot notation

```
spark.sql('SELECT train_id AS train, station FROM schedule LIMIT 5')
     .show()
```

```
+-----+-------------+
|train|      station|
+-----+-------------+
|  324|San Francisco|
|  324|  22nd Street|
|  324|     Millbrae|
|  324|    Hillsdale|
|  324| Redwood City|
+-----+-------------+
```

```
df.select(col('train_id').alias('train'), 'station')
  .limit(5)
  .show()
```

# Window function SQL

```
query = """
SELECT *,
ROW_NUMBER() OVER(PARTITION BY train_id ORDER BY time) AS id
FROM schedule
"""
spark.sql(query)
     .show(11)
```

# Window function SQL

```
+--------+-------------+-----+---+
|train_id|      station| time| id|
+--------+-------------+-----+---+
|     217|       Gilroy|6:06 |  1|
|     217|   San Martin|6:15 |  2|
|     217|  Morgan Hill|6:21 |  3|
|     217| Blossom Hill|6:36 |  4|
|     217|      Capitol|6:42 |  5|
|     217|       Tamien|6:50 |  6|
|     217|     San Jose|6:59 |  7|
|     324|San Francisco|7:59 |  1|
|     324|  22nd Street|8:03 |  2|
|     324|     Millbrae|8:16 |  3|
|     324|    Hillsdale|8:24 |  4|
+--------+-------------+-----+---+
```

# Window function using dot notation

```
from pyspark.sql import Window,
from pyspark.sql.functions import row_number
df.withColumn("id", row_number()
                    .over(
                           Window.partitionBy('train_id')
                                 .orderBy('time')
                          )
  )
```

- ROW_NUMBER in SQL : `pyspark.sql.functions.row_number`
- The inside of the OVER clause : `pyspark.sql.Window`
- PARTITION BY : `pyspark.sql.Window.partitionBy`
- ORDER BY : `pyspark.sql.Window.orderBy`

# Using a WindowSpec

- The `over` function in Spark SQL corresponds to a `OVER` clause in SQL.
- The class `pyspark.sql.window.Window` represents the inside of an `OVER` clause.

```
window = Window.partitionBy('train_id').orderBy('time')
dfx = df.withColumn('next', lead('time',1).over(window))
```

- Above, type(window) is pyspark.sql.window.WindowSpec

# Loading natural language text

INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# The dataset

**The Project Gutenberg eBook of The Adventures of Sherlock Holmes,**

by Sir Arthur Conan Doyle.

Available from gutenberg.org

# Loading text

```
df = spark.read.text('sherlock.txt')
```

```
print(df.first())
```

```
Row(value='The Project Gutenberg EBook of The Adventures of Sherlock Holmes')
```

```
print(df.count())
```

```
5500
```

# Loading parquet

```
df1 = spark.read.load('sherlock.parquet')
```

# Loaded text

```
df1.show(15, truncate=False)
```

```
+------------------------------------------------------------------+
|value                                                             |
+------------------------------------------------------------------+
|The Project Gutenberg EBook of The Adventures of Sherlock Holmes  |
|by Sir Arthur Conan Doyle                                         |
|(#15 in our series by Sir Arthur Conan Doyle)                     |
|                                                                  |
|Copyright laws are changing all over the world. Be sure to check the|
|copyright laws for your country before downloading or redistributing|
|this or any other Project Gutenberg eBook.                        |
|                                                                  |
|This header should be the first thing seen when viewing this Project|
|Gutenberg file.  Please do not remove it.  Do not change or edit the|
|header without written permission.                                |
|                                                                  |
|Please read the "legal small print," and other information about the|
|eBook and Project Gutenberg at the bottom of this file.  Included is|
|important information about your specific rights and restrictions in|
+------------------------------------------------------------------+
```

# Lower case operation

```
df = df1.select(lower(col('value')))


print(df.first())
```

```
Row(lower(value)=
    'the project gutenberg ebook of the adventures of sherlock holmes')
```

```
df.columns
```

```
['lower(value)']
```

# Alias operation

```
df = df1.select(lower(col('value')).alias('v'))
```

```
df.columns
```

```
['v']
```

# Replacing text

```
df = df1.select(regexp_replace('value', 'Mr\.', 'Mr').alias('v'))
```

> "Mr. Holmes." ==> "Mr Holmes."

```
df = df1.select(regexp_replace('value', 'don\'t', 'do not').alias('v'))
```

> "don't know." ==> "do not know."

# Tokenizing text

```
df = df2.select(split('v', '[ ]').alias('words'))
df.show(truncate=False)
```

# Tokenizing text – output

```
+--------------------------------------------------------------------------------+
|words                                                                           |
+--------------------------------------------------------------------------------+
|[the, project, gutenberg, ebook, of, the, adventures, of, sherlock, holmes]     |
|[by, sir, arthur, conan, doyle]                                                 |
|[(#15, in, our, series, by, sir, arthur, conan, doyle)]                         |
|[]
.
.
.
|[please, read, the, "legal, small, print,", and, other, information, about, the]|
.
.
.
|[**welcome, to, the, world, of, free, plain, vanilla, electronic, texts**]      |
+--------------------------------------------------------------------------------+
```

# Split characters are discarded

```
punctuation = "_|.\?\!\",\'\[\]\*()"
df3 = df2.select(split('v', '[ %s]' % punctuation).alias('words'))
```

```
df3.show(truncate=False)
```

# Split characters are discarded – output

```
+--------------------------------------------------------------------------------+
|words                                                                           |
+--------------------------------------------------------------------------------+
|[the, project, gutenberg, ebook, of, the, adventures, of, sherlock, holmes]     |
|[by, sir, arthur, conan, doyle]                                                 |
|[, #15, in, our, series, by, sir, arthur, conan, doyle, ]                       |
|[]                                                                              |
.                                                                                .
.

|[please, read, the, , legal, small, print, , , and, other, information, about, the]    |
.
.
.
[, , welcome, to, the, world, of, free, plain, vanilla, electronic, texts, , ]         |
++-------------------------------------------------------------------------------+
```

# Exploding an array

```
df4 = df3.select(explode('words').alias('word'))
df4.show()
```

# Exploding an array – output

```
+----------+
|      word|
+----------+
|       the|
|   project|
| gutenberg|
|     ebook|
|        of|
|       the|
|adventures|
|        of|
|  sherlock|
|    holmes|
|        by|
|       sir|
|    arthur|
|     conan|
|     doyle|
+----------+
```

# Explode increases row count

```
print(df3.count())
```

```
5500
```

```
print(df4.count())
```

```
131404
```

# Removing empty rows

```
print(df.count())
```

```
131404
```

```
nonblank_df = df.where(length('word') > 0)
```

```
print(nonblank_df.count())
```

```
107320
```

# Adding a row id column

```
df2 = df.select('word', monotonically_increasing_id().alias('id'))


df2.show()
```

# Adding a row id column – output

```
+----------+---+
|      word| id|
+----------+---+
|       the|  0|
|   project|  1|
| gutenberg|  2|
|     ebook|  3|
|        of|  4|
|       the|  5|
|adventures|  6|
|        of|  7|
|  sherlock|  8|
|    holmes|  9|
|        by| 10|
|       sir| 11|
|    arthur| 12|
|     conan| 13|
|     doyle| 14|
|       #15| 15|
+----------+---+
```

# Partitioning the data

```
df2 = df.withColumn('title', when(df.id < 25000, 'Preface')
                             .when(df.id < 50000, 'Chapter 1')
                             .when(df.id < 75000, 'Chapter 2')
                             .otherwise('Chapter 3'))
```

```
df2 = df2.withColumn('part', when(df2.id < 25000, 0)
                            .when(df2.id < 50000, 1)
                            .when(df2.id < 75000, 2)
                            .otherwise(3))
                            .show()
```

# Partitioning the data – output

```
+----------+---+-----------+----+
|word      |id |title      |part|
+----------+---+-----------+----+
|the       |0  |    Preface|0   |
|project   |1  |    Preface|0   |
|gutenberg |2  |    Preface|0   |
|ebook     |3  |    Preface|0   |
|of        |4  |    Preface|0   |
|the       |5  |    Preface|0   |
|adventures|6  |    Preface|0   |
|of        |7  |    Preface|0   |
|sherlock  |8  |    Preface|0   |
|holmes    |9  |    Preface|0   |
```

# Repartitioning on a column

```
df2 = df.repartition(4, 'part')
```

```
print(df2.rdd.getNumPartitions())
```

```
4
```

# Reading pre-partitioned text

```
$ ls sherlock_parts
```

```
sherlock_part0.txt
sherlock_part1.txt
sherlock_part2.txt
sherlock_part3.txt
sherlock_part4.txt
sherlock_part5.txt
sherlock_part6.txt
sherlock_part7.txt
sherlock_part8.txt
sherlock_part9.txt
sherlock_part10.txt
sherlock_part11.txt
sherlock_part12.txt
sherlock_part13.txt
```

# Reading pre-partitioned text

```
df_parts = spark.read.text('sherlock_parts')
```

# Moving window analysis

INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# The raw text

```
ADVENTURE  I.  A SCANDAL IN BOHEMIA

I.



To Sherlock Holmes she is always the woman. I have seldom heard him mention
her under any other name. In his eyes she eclipses and predominates the whole
of her sex. It was not that he felt any emotion akin to love for Irene Adler.
All emotions, and that one particularly, were abhorrent to his cold, precise
but admirably balanced mind. He was, I take it, the most perfect reasoning
and observing machine that the world has seen, but as a lover he would have
placed himself in a false position. He never spoke of the softer passions,
save with a gibe and a sneer. They were admirable things for the observer--
excellent for drawing the veil from men's motives and actions. But for the
trained reasoner to admit such intrusions into his own delicate and finely
adjusted temperament was to introduce a distracting factor which might throw
a doubt upon all his mental results. Grit in a sensitive instrument, or a
crack in one of his own high-power lenses, would not be more disturbing than
a strong emotion in a nature such as his. And yet there was but one woman to
him, and that woman was the late Irene Adler, of dubious and questionable
memory.
```

# The processed text

```
+--------+---+----+
|    word| id|part|
+--------+---+----+
| scandal|305|   1|
|      in|306|   1|
| bohemia|307|   1|
|       i|308|   1|
|      to|309|   1|
|sherlock|310|   1|
|  holmes|311|   1|
|     she|312|   1|
|      is|313|   1|
|  always|314|   1|
|     the|315|   1|
|   woman|316|   1|
|       i|317|   1|
|    have|318|   1|
|  seldom|319|   1|
|   heard|320|   1|
|     him|321|   1|
| mention|322|   1|
|     her|323|   1|
|   under|324|   1|
+--------+---+----+
```

# Partitions

```
df.select('part', 'title').distinct().sort('part').show(truncate=False)
```

```
+----+--------------------+
|part|title               |
+----+--------------------+
|1   |Sherlock Chapter I  |
|2   |Sherlock Chapter II |
|3   |Sherlock Chapter III|
|4   |Sherlock Chapter IV |
|5   |Sherlock Chapter V  |
|6   |Sherlock Chapter VI |
|7   |Sherlock Chapter VII|
|8   |Sherlock Chapter VIII|
|9   |Sherlock Chapter IX |
|10  |Sherlock Chapter X  |
|11  |Sherlock Chapter XI |
|12  |Sherlock Chapter XII|
+----+--------------------+
```

| id | word |
|---|---|
| 0 | the |
| 1 | project |
| 2 | gutenberg |
| 3 | ebook |
| 4 | of |
| 5 | the |
| 6 | adventures |
| 7 | of |
| 8 | sherlock |
| 9 | holmes |
| 10 | by |
| 11 | sir |
| 12 | arthur |
| 13 | conan |

| id | word |
|---|---|
| 0 | the |
| 1 | project |
| 2 | gutenberg |
| 4 | of |
| 5 | the |
| 6 | adventures |
| 7 | of |
| 8 | sherlock |
| 9 | holmes |
| 10 | by |
| 11 | sir |
| 12 | arthur |
| 13 | conan |

| id | word |
|---|---|
| 1 | project |
| 2 | gutenberg |
| 3 | ebook |
| 5 | the |
| 6 | adventures |
| 7 | of |
| 8 | sherlock |
| 9 | holmes |
| 10 | by |
| 11 | sir |
| 12 | arthur |
| 13 | conan |

| id | word |
|---|---|
| 0 | the |
| 1 | project |
| 2 | gutenberg |
| 3 | ebook |
| 4 | of |
| 5 | the |
| 6 | adventures |
| 7 | of |
| 8 | sherlock |
| 9 | holmes |
| 10 | by |
| 11 | sir |
| 12 | arthur |
| 13 | conan |

| id | word |
|---|---|
| 0 | the |
| 1 | project |
| 2 | gutenberg |
| 3 | ebook |
| 4 | of |
| 5 | the |
| 7 | of |
| 8 | sherlock |
| 9 | holmes |
| 10 | by |
| 11 | sir |
| 12 | arthur |
| 13 | conan |

| id | word |
|---|---|
| 0 | the |
| 1 | project |
| 2 | gutenberg |
| 4 | of |
| 5 | the |
| 6 | adventures |
| 8 | sherlock |
| 9 | holmes |
| 10 | by |
| 11 | sir |
| 12 | arthur |
| 13 | conan |

| id | word |
|---|---|
| 0 | the |
| 1 | project |
| 2 | gutenberg |
| 3 | ebook |
| 4 | of |
| 6 | adventures |
| 7 | of |
| 8 | sherlock |
| 10 | by |
| 11 | sir |
| 12 | arthur |
| 13 | conan |

| id | word |
|---|---|
| 0 | the |
| 1 | project |
| 2 | gutenberg |
| 3 | ebook |
| 4 | of |
| 5 | the |
| 6 | adventures |
| 7 | of |
| 8 | sherlock |
| 9 | holmes |
| 10 | by |
| 12 | arthur |
| 13 | conan |

| id | word |
|---|---|
| 0 | the |
| 1 | project |
| 2 | gutenberg |
| 3 | ebook |
| 4 | of |
| 5 | the |
| 6 | adventures |
| 7 | of |
| 8 | sherlock |
| 9 | holmes |
| 10 | by |
| 11 | sir |
| 12 | arthur |
| 13 | conan |

# The words are indexed

```
+---+----------+
| id|      word|
+---+----------+
|  0|       the|
|  1|   project|
|  2| gutenberg|
|  3|     ebook|
|  4|        of|
|  5|       the|
|  6|adventures|
|  7|        of|
|  8|  sherlock|
|  9|    holmes|
| 10|        by|
| 11|       sir|
| 12|    arthur|
| 13|     conan|
| 14|     doyle|
| 15|       #15|
| 16|        in|
| 17|       our|
| 18|    series|
| 19|        by|
+---+----------+
```

# A moving window query

```
query = """
   SELECT id, word AS w1,
   LEAD(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
   LEAD(word,2) OVER(PARTITION BY part ORDER BY id ) AS w3
   FROM df
"""
spark.sql(query).sort('id').show()
```

```
+---+----------+----------+----------+
| id|        w1|        w2|        w3|
+---+----------+----------+----------+
|  0|       the|   project| gutenberg|
|  1|   project| gutenberg|     ebook|
|  2| gutenberg|     ebook|        of|
|  3|     ebook|        of|       the|
|  4|        of|       the|adventures|
|...|..........|..........|..........|
```

# Moving window output

```
+---+----------+----------+----------+
| id|        w1|        w2|        w3|
+---+----------+----------+----------+
|  0|       the|   project| gutenberg|
|  1|   project| gutenberg|     ebook|
|  2| gutenberg|     ebook|        of|
|  3|     ebook|        of|       the|
|  4|        of|       the|adventures|
|  5|       the|adventures|        of|
|  6|adventures|        of|  sherlock|
|  7|        of|  sherlock|    holmes|
|  8|  sherlock|    holmes|        by|
|  9|    holmes|        by|       sir|
| 10|        by|       sir|    arthur|
| 11|       sir|    arthur|     conan|
| 12|    arthur|     conan|     doyle|
+---+----------+----------+----------+
```

# LAG window function

```
lag_query = """
   SELECT
   id,
   LAG(word,2) OVER(PARTITION BY part ORDER BY id ) AS w1,
   LAG(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
   word AS w3
   FROM df
   ORDER BY id
"""

spark.sql(lag_query).show()
```

# LAG window function – output

```
+---+----------+----------+----------+
| id|        w1|        w2|        w3|
+---+----------+----------+----------+
|  0|      null|      null|       the|
|  1|      null|       the|   project|
|  2|       the|   project| gutenberg|
|  3|   project| gutenberg|     ebook|
|  4| gutenberg|     ebook|        of|
|  5|     ebook|        of|       the|
|  6|        of|       the|adventures|
|  7|       the|adventures|        of|
|  8|adventures|        of|  sherlock|
|  9|        of|  sherlock|    holmes|
| 10|  sherlock|    holmes|        by|
| 11|    holmes|        by|       sir|
| 12|       ...|       ...|       ...|
+---+----------+----------+----------+
```

# Windows stay within partition

```
lag_query = """
   SELECT
   id,
   LAG(word,2) OVER(PARTITION BY part ORDER BY id ) AS w1,
   LAG(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
   word AS w3
   FROM df
   WHERE part=2
"""

spark.sql(lag_query).show()
```

# Windows stay within partition – output

```
+----+----------+----------+----------+
|  id|        w1|        w2|        w3|
+----+----------+----------+----------+
|8859|      null|      null|     part2|
|8860|      null|     part2| adventure|
|8861|     part2| adventure|        ii|
|8862| adventure|        ii|       the|
|8863|        ii|       the|red-headed|
|8864|       the|red-headed|    league|
|....|..........|..........|..........|
```

# Repartitioning

- PARTITION BY
- `repartition()`

# Common word sequences

INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# Categorical vs Ordinal

- **Categorical:** he, hi, she, that, they
- **Ordinal:** 1, 2, 3, 4, 5

# 3-tuples

```
query3 = """
    SELECT
    id,
    word AS w1,
    LEAD(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
    LEAD(word,2) OVER(PARTITION BY part ORDER BY id ) AS w3
    FROM df
"""
```

# A window function SQL as subquery

```
query3agg = """
SELECT w1, w2, w3, COUNT(*) as count FROM (
   SELECT
   word AS w1,
   LEAD(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
   LEAD(word,2) OVER(PARTITION BY part ORDER BY id ) AS w3
   FROM df
)
GROUP BY w1, w2, w3
ORDER BY count DESC
"""

spark.sql(query3agg).show()
```

# A window function SQL as subquery – output

```
+-----+-----+-----+-----+
|   w1|   w2|   w3|count|
+-----+-----+-----+-----+
|  one|   of|  the|   49|
|    i|think| that|   46|
|   it|   is|    a|   46|
|   it|  was|    a|   45|
| that|   it|  was|   38|
|  out|   of|  the|   35|
|.....|.....|.....|.....|
```

# Most frequent 3-tuples

```
+-----+-----+-----+-----+
|   w1|   w2|   w3|count|
+-----+-----+-----+-----+
|  one|   of|  the|   49|
|    i|think| that|   46|
|   it|   is|    a|   46|
|   it|  was|    a|   45|
| that|   it|  was|   38|
|  out|   of|  the|   35|
| that|    i| have|   35|
|there|  was|    a|   34|
|    i|   do|  not|   34|
| that|   it|   is|   33|
| that|   he|  was|   30|
| that|   he|  had|   30|
| that|    i|  was|   28|
+-----+-----+-----+-----+
```

# Another type of aggregation

```
query3agg = """
SELECT w1, w2, w3, length(w1)+length(w2)+length(w3) as length FROM (
   SELECT
   word AS w1,
   LEAD(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
   LEAD(word,2) OVER(PARTITION BY part ORDER BY id ) AS w3
   FROM df
   WHERE part <> 0 and part <> 13
)
GROUP BY w1, w2, w3
ORDER BY length DESC
"""

spark.sql(query3agg).show(truncate=False)
```

# Another type of aggregation

```
+------------------+------------------+--------------+------+
|                w1|                w2|            w3|length|
+------------------+------------------+--------------+------+
|comfortable-looking|          building|   two-storied|    38|
|         widespread|comfortable-looking|      building|    37|
|      extraordinary|     circumstances|     connected|    35|
|      simple-minded|     nonconformist|     clergyman|    35|
|       particularly|         malignant| boot-slitting|    34|
|       unsystematic|       sensational|    literature|    33|
|       oppressively|       respectable|    frock-coat|    33|
|         relentless|       keen-witted|  ready-handed|    33|
|   travelling-cloak|               and| close-fitting|    32|
|        ruddy-faced|     white-aproned|      landlord|    32|
|  fellow-countryman|           colonel|      lysander|    32|
+------------------+------------------+--------------+------+
```

# Caching

## INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# What is caching?

- Keeping data in memory
- Spark tends to unload memory aggressively

# Eviction Policy

- Least Recently Used (LRU)
- Eviction happens independently on each worker
- Depends on memory available to each worker

# Caching a dataframe

### TO CACHE A DATAFRAME:

```
df.cache()
```

### TO UNCACHE IT:

```
df.unpersist()
```

# Determining whether a dataframe is cached

```
df.is_cached
```

```
False
```

```
df.cache()
df.is_cached
```

```
True
```

# Uncaching a dataframe

```
df.unpersist()
df.is_cached()
```

```
False
```

# Storage level

```
df.unpersist()
df.cache()
df.storageLevel
```

```
StorageLevel(True, True, False, True, 1)
```

In the storage level above the following hold:

1. `useDisk` = True
2. `useMemory` = True
3. `useOffHeap` = False
4. `deserialized` = True
5. `replication` = 1

# Persisting a dataframe

**The following are equivalent in Spark 2.1+ :**

- `df.persist()`
- `df.persist(storageLevel=pyspark.StorageLevel.MEMORY_AND_DISK)`
- `df.cache()` is the same as `df.persist()`

# Caching a table

```
df.createOrReplaceTempView('df')
spark.catalog.isCached(tableName='df')
```

```
False
```

```
spark.catalog.cacheTable('df')
spark.catalog.isCached(tableName='df')
```

```
True
```

# Uncaching a table

```
spark.catalog.uncacheTable('df')
spark.catalog.isCached(tableName='df')
```

```
False
```

```
spark.catalog.clearCache()
```

# Tips

- Caching is lazy
- Only cache if more than one operation is to be performed
- Unpersist when you no longer need the object
- Cache selectively

# The Spark UI

## INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# Use the Spark UI inspect execution

- **Spark Task** is a unit of execution that runs on a single cpu
- **Spark Stage** a group of tasks that perform the same computation in parallel, each task typically running on a different subset of the data
- **Spark Job** is a computation triggered by an action, sliced into one or more stages.

# Finding the Spark UI

1. http://[DRIVER_HOST]:4040
2. http://[DRIVER_HOST]:4041
3. http://[DRIVER_HOST]:4042
4. http://[DRIVER_HOST]:4043

   ...


Jobs | Stages | Storage | Environment | Executors | SQL

# Spark Jobs (?)

**User:** mark
**Total Uptime:** 3 s
**Scheduling Mode:** FIFO

▸ Event Timeline

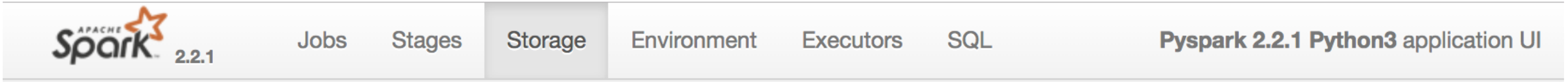
# Spark Jobs (?)

**User:** mark
**Total Uptime:** 9.1 min
**Scheduling Mode:** FIFO
**Completed Jobs:** 1

▸ Event Timeline

## Completed Jobs (1)

| Job Id ▾ | Description | Submitted | Duration | Stages: Succeeded/Total | Tasks (for all stages): Succeeded/Total |
|---|---|---|---|---|---|
| 0 | load at NativeMethodAccessorImpl.java:0 | 2018/12/23 19:56:18 | 0.5 s | 1/1 | 1/1 |




# Storage

## RDDs

| RDD Name | Storage Level | Cached Partitions | Fraction Cached | Size in Memory | Size on Disk |
|---|---|---|---|---|---|
| *FileScan parquet [word#9,id#10L,title#11,part#12] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/Users/mark/code/datacamp_py/sherlock_full_parts.parquet], PartitionFilters: [], PushedFilters: [], ReadSchema: struct<word:string,id:bigint,title:string,part:int> | Memory Deserialized 1x Replicated | 1 | 100% | 554.9 KB | 0.0 B |

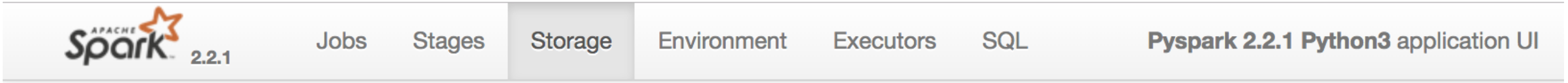
# Spark catalog operations

- `spark.catalog.cacheTable('table1')`
- `spark.catalog.uncacheTable('table1')`
- `spark.catalog.isCached('table1')`
- `spark.catalog.dropTempView('table1')`

# Spark Catalog

```
spark.catalog.listTables()
```

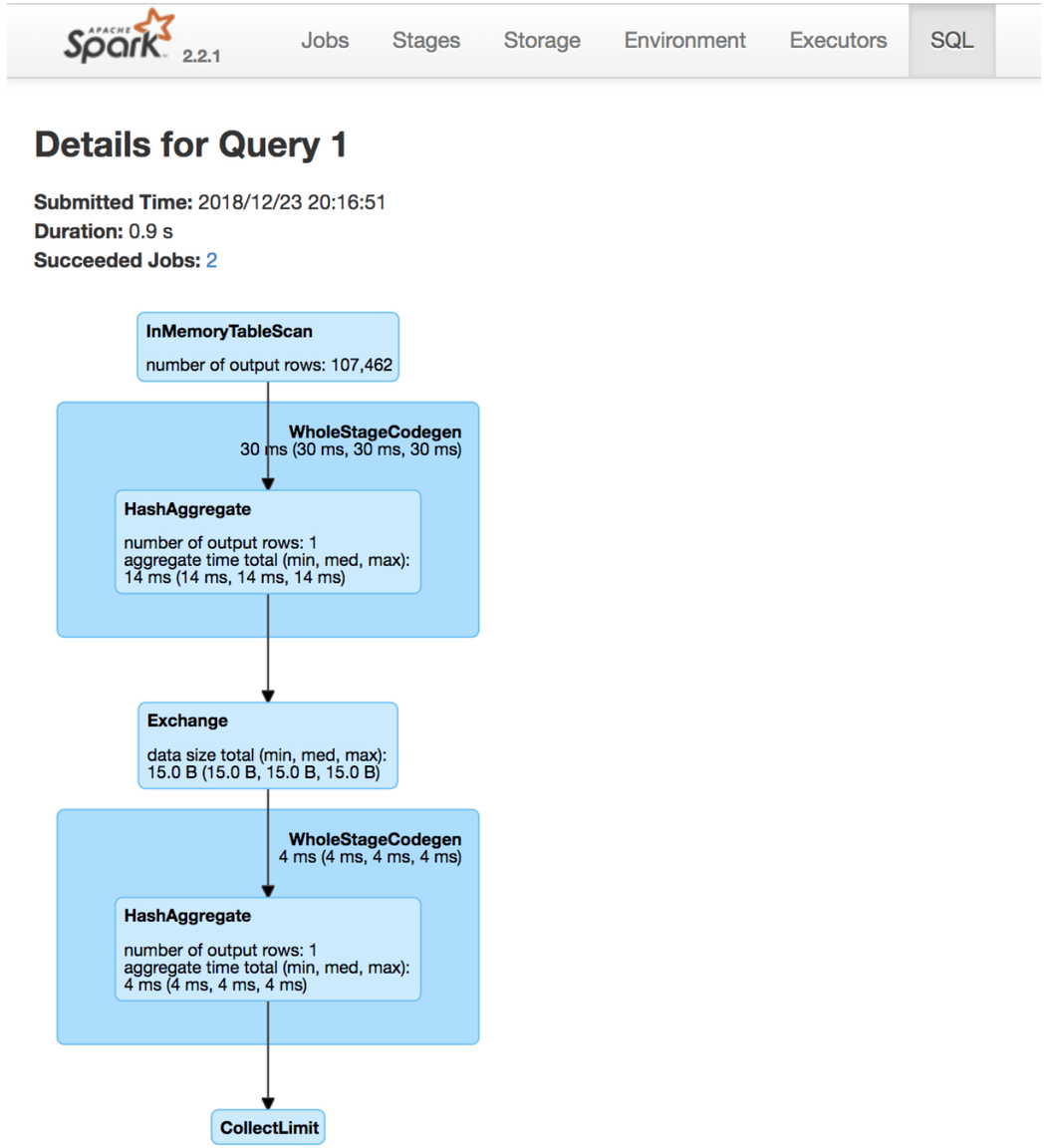
```
[Table(name='text', database=None, description=None, tableType='TEMPORARY', isTempor
```




# Storage

## RDDs

| RDD Name | Storage Level | Cached Partitions | Fraction Cached | Size in Memory | Size on Disk |
|---|---|---|---|---|---|
| In-memory table df | Memory Deserialized 1x Replicated | 1 | 100% | 554.9 KB | 0.0 B |


Spark 2.2.1
Jobs
Stages
Storage
Environment
Executors
SQL
Details for Query 1
Submitted Time: 2018/12/23 20:16:51
Duration: 0.9 s
Succeeded Jobs: 2
InMemoryTableScan
number of output rows: 107,462
WholeStageCodegen
30 ms (30 ms, 30 ms, 30 ms)
HashAggregate
number of output rows: 1
aggregate time total (min, med, max):
14 ms (14 ms, 14 ms, 14 ms)
Exchange
data size total (min, med, max):
15.0 B (15.0 B, 15.0 B, 15.0 B)
WholeStageCodegen
4 ms (4 ms, 4 ms, 4 ms)
HashAggregate
number of output rows: 1
aggregate time total (min, med, max):
4 ms (4 ms, 4 ms, 4 ms)
CollectLimit
Details

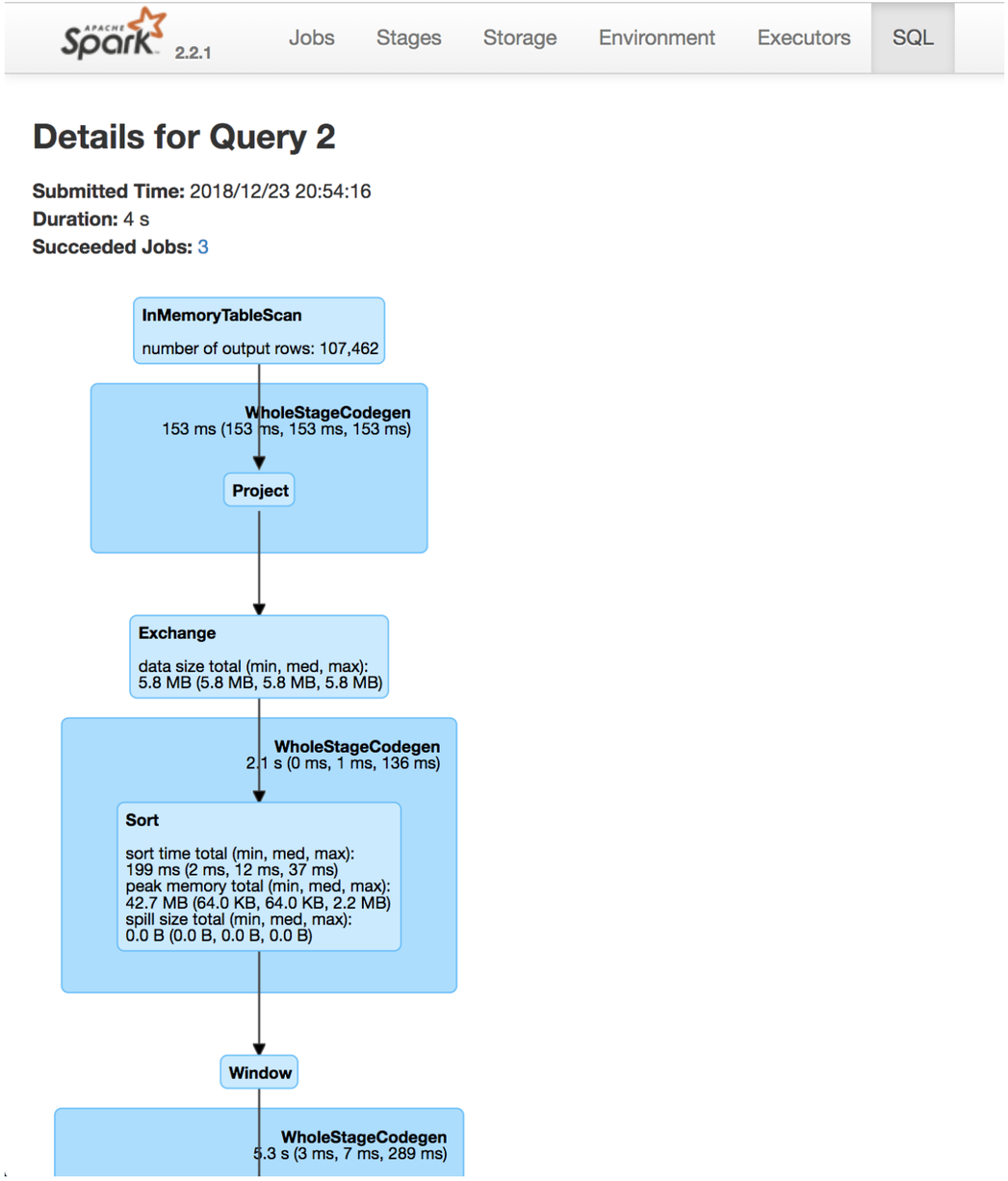
# Spark UI Storage Tab

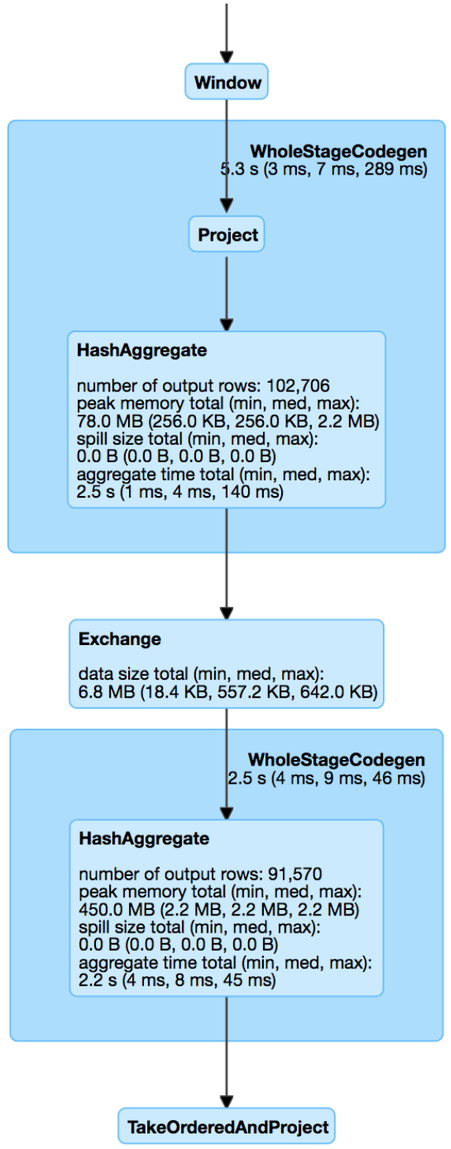
Shows where data partitions exist

- in memory,
- or on disk,
- across the cluster,
- at a snapshot in time.

# Spark UI SQL tab

```
query3agg = """
SELECT w1, w2, w3, COUNT(*) as count FROM (
   SELECT
   word AS w1,
   LEAD(word,1) OVER(PARTITION BY part ORDER BY id ) AS w2,
   LEAD(word,2) OVER(PARTITION BY part ORDER BY id ) AS w3
   FROM df
)
GROUP BY w1, w2, w3
ORDER BY count DESC
"""
spark.sql(query3agg).show()
```


Spark 2.2.1
Jobs
Stages
Storage
Environment
Executors
SQL
Details for Query 2
Submitted Time: 2018/12/23 20:54:16
Duration: 4 s
Succeeded Jobs: 3
InMemoryTableScan
number of output rows: 107,462
WholeStageCodegen
153 ms (153 ms, 153 ms, 153 ms)
Project
Exchange
data size total (min, med, max):
5.8 MB (5.8 MB, 5.8 MB, 5.8 MB)
WholeStageCodegen
2.1 s (0 ms, 1 ms, 136 ms)
Sort
sort time total (min, med, max):
199 ms (2 ms, 12 ms, 37 ms)
peak memory total (min, med, max):
42.7 MB (64.0 KB, 64.0 KB, 2.2 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
Window
WholeStageCodegen
5.3 s (3 ms, 7 ms, 289 ms)


Window
WholeStageCodegen
5.3 s (3 ms, 7 ms, 289 ms)
Project
HashAggregate
number of output rows: 102,706
peak memory total (min, med, max):
78.0 MB (256.0 KB, 256.0 KB, 2.2 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
2.5 s (1 ms, 4 ms, 140 ms)
Exchange
data size total (min, med, max):
6.8 MB (18.4 KB, 557.2 KB, 642.0 KB)
WholeStageCodegen
2.5 s (4 ms, 9 ms, 46 ms)
HashAggregate
number of output rows: 91,570
peak memory total (min, med, max):
450.0 MB (2.2 MB, 2.2 MB, 2.2 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
2.2 s (4 ms, 8 ms, 45 ms)
TakeOrderedAndProject

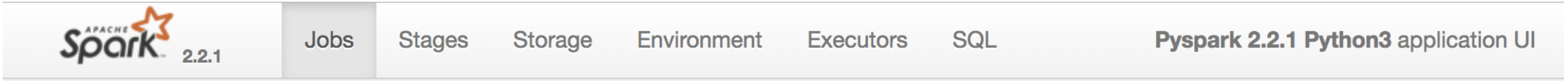
▸ Details


# Stages for All Jobs

**Completed Stages:** 6

## Completed Stages (6)

| Stage Id ▾ | Description | Submitted | Duration | Tasks: Succeeded/Total | Input | Output | Shuffle Read | Shuffle Write |
|---|---|---|---|---|---|---|---|---|
| 5 | showString at NativeMethodAccessorImpl.java:0 +details | 2018/12/23 20:54:19 | 0.6 s | 200/200 | | | 3.7 MB | |
| 4 | showString at NativeMethodAccessorImpl.java:0 +details | 2018/12/23 20:54:17 | 2 s | 200/200 | | | 1972.4 KB | 3.7 MB |
| 3 | showString at NativeMethodAccessorImpl.java:0 +details | 2018/12/23 20:54:16 | 0.8 s | 1/1 | 677.8 KB | | | 1972.4 KB |
| 2 | hasNext at NativeMethodAccessorImpl.java:0 +details | 2018/12/23 20:52:41 | 12 ms | 1/1 | | | | |
| 1 | hasNext at NativeMethodAccessorImpl.java:0 +details | 2018/12/23 20:52:41 | 11 ms | 1/1 | | | | |
| 0 | load at NativeMethodAccessorImpl.java:0 +details | 2018/12/23 20:52:33 | 0.3 s | 1/1 | | | | |




# Spark Jobs (?)

**User:** mark
**Total Uptime:** 18 min
**Scheduling Mode:** FIFO
**Completed Jobs:** 4

▸ Event Timeline

## Completed Jobs (4)

| Job Id ▾ | Description | Submitted | Duration | Stages: Succeeded/Total | Tasks (for all stages): Succeeded/Total |
|---|---|---|---|---|---|
| 3 | showString at NativeMethodAccessorImpl.java:0 | 2018/12/23 20:54:16 | 4 s | 3/3 | 401/401 |
| 2 | hasNext at NativeMethodAccessorImpl.java:0 | 2018/12/23 20:52:41 | 20 ms | 1/1 | 1/1 |
| 1 | hasNext at NativeMethodAccessorImpl.java:0 | 2018/12/23 20:52:41 | 18 ms | 1/1 | 1/1 |
| 0 | load at NativeMethodAccessorImpl.java:0 | 2018/12/23 20:52:33 | 0.5 s | 1/1 | 1/1 |

# Logging

## INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# Logging primer

```
import logging
logging.basicConfig(stream=sys.stdout, level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')
logging.info("Hello %s", "world")
logging.debug("Hello, take %d", 2)
```

```
2019-03-14 15:92:65,359 - INFO - Hello world
```

# Logging with DEBUG level

```
import logging
logging.basicConfig(stream=sys.stdout, level=logging.DEBUG,
                    format='%(asctime)s - %(levelname)s - %(message)s')
logging.info("Hello %s", "world")
logging.debug("Hello, take %d", 2)
```

```
2018-03-14 12:00:00,000 - INFO - Hello world
2018-03-14 12:00:00,001 - DEBUG - Hello, take 2
```

# Debugging lazy evaluation

- lazy evaluation
- distributed execution

# A simple timer

```
t = timer()
t.elapsed()
```

```
1. elapsed: 0.0 sec
```

```
t.elapsed() # Do something that takes 2 seconds
```

```
2. elapsed: 2.0 sec
```

```
t.reset() # Do something else that takes time: reset
t.elapsed()
```

```
3. elapsed: 0.0 sec
```

# class timer

```
class timer:
    start_time = time.time()
    step = 0

    def elapsed(self, reset=True):
        self.step += 1
        print("%d. elapsed: %.1f sec %s"
              % (self.step, time.time() - self.start_time))
        if reset:
            self.reset()

    def reset(self):
        self.start_time = time.time()
```

# Stealth CPU wastage

```
import logging
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')

# < create dataframe df here >

t = timer()
logging.info("No action here.")
t.elapsed()
logging.debug("df has %d rows.", df.count())
t.elapsed()
```

```
2018-12-23 22:24:20,472 - INFO - No action here.
1. elapsed: 0.0 sec
2. elapsed: 2.0 sec
```

# Disable actions

```
ENABLED = False


t = timer()
logger.info("No action here.")
t.elapsed()
if ENABLED:
    logger.info("df has %d rows.", df.count())
t.elapsed()
```

```
2019-03-14 12:34:56,789 - Pyspark - INFO - No action here.
1. elapsed: 0.0 sec
2. elapsed: 0.0 sec
```

# Enabling actions

Rerunning the previous example with ENABLED = True triggers the action:

```
2019-03-14 12:34:56,789 - INFO - No action here.
1. elapsed: 0.0 sec
2019-03-14 12:34:58,789 - INFO - df has 1107014 rows.
2. elapsed: 2.0 sec
```

# Query Plans

## INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# Explain

```
EXPLAIN SELECT * FROM table1
```

# Load dataframe and register

```
df = spark.read.load('/temp/df.parquet')
```

```
df.registerTempTable('df')
```

# Running an EXPLAIN query

```
spark.sql('EXPLAIN SELECT * FROM df').first()
```

```
Row(plan='== Physical Plan ==\n
*FileScan parquet [word#1928,id#1929L,title#1930,part#1931]
  Batched: true,
  Format: Parquet,
  Location: InMemoryFileIndex[file:/temp/df.parquet],
  PartitionFilters: [],
  PushedFilters: [],
  ReadSchema: struct<word:string,id:bigint,title:string,part:int>')
```

# Interpreting an EXPLAIN query

== Physical Plan ==

- `FileScan parquet [word#1928,id#1929L,title#1930,part#1931]`
- `Batched: true,`
- `Format: Parquet,`
- `Location: InMemoryFileIndex[file:/temp/df.parquet],`
- `PartitionFilters: [],`
- `PushedFilters: [],`
- `ReadSchema: struct<word:string,id:bigint,title:string,part:int>'`

# df.explain()

```
df.explain()
```

```
== Physical Plan ==
FileScan parquet [word#963,id#964L,title#965,part#966]
Batched: true, Format: Parquet,
Location: InMemoryFileIndex[file:/temp/df.parquet],
PartitionFilters: [], PushedFilters: [],
ReadSchema: struct<word:string,id:bigint,title:string,part:int>
```

```
spark.sql("SELECT * FROM df").explain()
```

```
== Physical Plan ==
FileScan parquet [word#712,id#713L,title#714,part#715]
Batched: true, Format: Parquet,
Location: InMemoryFileIndex[file:/temp/df.parquet],
PartitionFilters: [], PushedFilters: [],
ReadSchema: struct<word:string,id:bigint,title:string,part:int>
```

# df.explain(), on cached dataframe

```
df.cache()
df.explain()
```

```
== Physical Plan ==
InMemoryTableScan [word#0, id#1L, title#2, part#3]
   +- InMemoryRelation [word#0, id#1L, title#2, part#3], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
      +- FileScan parquet [word#0,id#1L,title#2,part#3]
         Batched: true, Format: Parquet, Location:
         InMemoryFileIndex[file:/temp/df.parquet],
         PartitionFilters: [], PushedFilters: [],
         ReadSchema: struct<word:string,id:bigint,title:string,part:int>
```

```
spark.sql("SELECT * FROM df").explain()
```

```
== Physical Plan ==
InMemoryTableScan [word#0, id#1L, title#2, part#3]
   +- InMemoryRelation [word#0, id#1L, title#2, part#3], true, 10000, StorageLevel(disk, memory, deserialized, 1 replicas)
     +- FileScan parquet [word#0,id#1L,title#2,part#3]
        Batched: true, Format: Parquet,
        Location: InMemoryFileIndex[file:/temp/df.parquet],
        PartitionFilters: [], PushedFilters: [],
        ReadSchema: struct<word:string,id:bigint,title:string,part:int>
```

# Words sorted by frequency query

```
SELECT word, COUNT(*) AS count
FROM df
GROUP BY word
ORDER BY count DESC
```

Equivalent dot notation approach:

```
df.groupBy('word')
  .count()
  .sort(desc('count'))
  .explain()
```

# Same query using dataframe dot notation

```
== Physical Plan ==
*Sort [count#1040L DESC NULLS LAST], true, 0
+- Exchange rangepartitioning(count#1040L DESC NULLS LAST, 200)
   +- *HashAggregate(keys=[word#963], functions=[count(1)])
      +- Exchange hashpartitioning(word#963, 200)
         +- *HashAggregate(keys=[word#963], functions=[partial_count(1)])
            +- InMemoryTableScan [word#963]
                  +- InMemoryRelation [word#963, id#964L, title#965, part#966],
                     true,10000, StorageLevel(disk, memory, deserialized,
                     1 replicas)
                        +- *FileScan parquet [word#963,id#964L,title#965,part#966]
                           Batched: true, Format: Parquet,
                           Location: InMemoryFileIndex[file:/temp/df.parquet],
                           PartitionFilters: [], PushedFilters: [],
                           ReadSchema: struct<word:string,id:bigint,title:string,part:int>
```

# Reading from bottom up

- `FileScan parquet`
- `InMemoryRelation`
- `InMemoryTableScan`
- `HashAggregate(keys=[word#963], ...)`
- `HashAggregate(keys=[word#963], functions=[count(1)])`
- `Sort [count#1040L DESC NULLS LAST]`

# Query plan

```
== Physical Plan ==
*Sort [count#1160L DESC NULLS LAST], true, 0
+- Exchange rangepartitioning(count#1160L DESC NULLS LAST, 200)
   +- *HashAggregate(keys=[word#963], functions=[count(1)])
      +- Exchange hashpartitioning(word#963, 200)
         +- *HashAggregate(keys=[word#963], functions=[partial_count(1)])
            +- *FileScan parquet [word#963] Batched: true, Format: Parquet,
               Location: InMemoryFileIndex[file:/temp/df.parquet], PartitionFilters: [],
               PushedFilters: [], ReadSchema: struct<word:string>
```

The previous plan had the following lines, which are missing from the plan above:

```
...
            +- InMemoryTableScan [word#963]
                  +- InMemoryRelation [word#963, id#964L, title#965, part#966], true, 10000,
                     StorageLevel(disk, memory, deserialized, 1 replicas)
...
```

# Extract Transform Select

## INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# Extract, Transform, and Select

- Extraction
- Transformation
- Selection

# Built-in functions

```
from pyspark.sql.functions import split, explode
```

# The length function

```
from pyspark.sql.functions import length
```

```
df.where(length('sentence') == 0)
```

# Creating a custom function

- User Defined Function
- UDF

# Importing the udf function

```
from pyspark.sql.functions import udf
```

# Creating a boolean UDF

```
print(df)
```

```
DataFrame[textdata: string]
```

```
from pyspark.sql.functions import udf
```

```
from pyspark.sql.types import BooleanType
```

# Creating a boolean UDF

```
short_udf = udf(lambda x:
                            True if not x or len(x) < 10 else False,
                            BooleanType())
```

```
df.select(short_udf('textdata')\
  .alias("is short"))\
  .show(3)
```

```
+--------+
|is short|
+--------+
|   false|
|    true|
|   false|
+--------+
```

# Important UDF return types

```
from pyspark.sql.types import StringType, IntegerType, FloatType, ArrayType
```

# Creating an array UDF

```
df3.select('word array', in_udf('word array').alias('without endword'))\
    .show(5, truncate=30)
```

```
+------------------------------+------------------------+
|                    word array|         without endword|
+------------------------------+------------------------+
|[then, how, many, are, there]||[then, how, many, are]|
|                   [how, many]||                 [how]|
|              [i, donot, know]||             [i, donot]|
|                   [quite, so]||                [quite]|
|    [you, have, not, observed]||      [you, have, not]|
+------------------------------+------------------------+
```

# Creating an array UDF

```
from pyspark.sql.types import StringType, ArrayType
```

```
# Removes last item in array
in_udf = udf(lambda x:
    x[0:len(x)-1] if x and len(x) > 1
    else [],
    ArrayType(StringType()))
```

# Sparse vector format

1. Indices
2. Values

Example:

- Array: `[1.0, 0.0, 0.0, 3.0]`
- Sparse vector: `(4, [0, 3], [1.0, 3.0])`

# Working with vector data

- `hasattr(x, "toArray")`
- `x.numNonzeros())`

# Creating feature data for classification

INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# Transforming a dense array

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType
bad_udf = udf(lambda x:
              x.indices[0]
              if (x and hasattr(x, "toArray") and x.numNonzeros())
              else 0,
              IntegerType())
```

# Transforming a dense array

```
try:
    df.select(bad_udf('outvec').alias('label')).first()
except Exception as e:
    print(e.__class__)
    print(e.errmsg)
```

```
<class 'py4j.protocol.Py4JJavaError'>
An error occurred while calling o90.collectToPython.
```

# UDF return type must be properly cast

```
first_udf = udf(lambda x:
                int(x.indices[0])
                if (x and hasattr(x, "toArray") and x.numNonzeros())
                else 0,
                IntegerType())
```

# The UDF in action

```
+-------+--------------------+-----+--------------------+--------------------+
|endword|                 doc|count|            features|              outvec|
+-------+--------------------+-----+--------------------+--------------------+
|     it|[please, do, not,...| 1149|(12847,[15,47,502...|  (12847,[7],[1.0])|
| holmes|[start, of, the, ...|  107|(12847,[0,3,183,1...|(12847,[145],[1.0])|
|      i|[the, adventures,...|  103|(12847,[0,3,35,14...| (12847,[11],[1.0])|
+-------+--------------------+-----+--------------------+--------------------+
```

```
df.withColumn('label', k_udf('outvec')).drop('outvec').show(3)
```

```
+-------+--------------------+-----+--------------------+-----+
|endword|                 doc|count|            features|label|
+-------+--------------------+-----+--------------------+-----+
|     it|[please, do, not,...| 1149|(12847,[15,47,502...|    7|
| holmes|[start, of, the, ...|  107|(12847,[0,3,183,1...|  145|
|      i|[the, adventures,...|  103|(12847,[0,3,35,14...|   11|
+-------+--------------------+-----+--------------------+-----+
```

# CountVectorizer

- ETS : Extract Transform Select
- CountVectorizer is a Feature **Extractor**
- Its input is an array of strings
- Its output is a vector

# Fitting the CountVectorizer

```
from pyspark.ml.feature import CountVectorizer

cv = CountVectorizer(inputCol='words',
                     outputCol="features")
model = cv.fit(df)
result = model.transform(df)
print(result)
```

```
DataFrame[words: array<string>, features: vector]

# Dense string array on left, dense integer vector on right
+------------------------+---------------------------------------+
|words                   |features                               |
+------------------------+---------------------------------------+
|[Hello, world]          |(10,[7,9],[1.0,1.0])                   |
|[How, are, you?]        |(10,[1,3,4],[1.0,1.0,1.0])             |
|[I, am, fine, thank, you]|(10,[0,2,5,6,8],[1.0,1.0,1.0,1.0,1.0])|
+------------------------+---------------------------------------+
```

# Text Classification

INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# Selecting the data

```
df_true = df.where("endword in ('she', 'he', 'hers', 'his', 'her', 'him')")\
            .withColumn('label', lit(1))
```

```
df_false = df.where("endword not in ('she', 'he', 'hers', 'his', 'her', 'him')")\
            .withColumn('label', lit(0))
```

# Combining the positive and negative data

```
df_examples = df_true.union(df_false)
```

# Splitting the data into training and evaluation sets

```
df_train, df_eval = df_examples.randomSplit((0.60, 0.40), 42)
```

# Training

```
from pyspark.ml.classification import LogisticRegression
logistic = LogisticRegression(maxIter=50, regParam=0.6, elasticNetParam=0.3)
model = logistic.fit(df_train)
print("Training iterations: ", model.summary.totalIterations)
```

# Predicting and evaluating

INTRODUCTION TO SPARK SQL IN PYTHON

**Mark Plutowski**
Data Scientist

datacamp

# Applying a model to evaluation data

```
predicted = df_trained.transform(df_test)
```

- prediction column: double
- probability column: vector of length two

```
x = predicted.first
print("Right!" if x.label == int(x.prediction) else "Wrong")
```

# Evaluating classification accuracy

```
model_stats = model.evaluate(df_eval)
```

```
type(model_stats)
```

```
pyspark.ml.classification.BinaryLogisticRegressionSummary)
```

```
print("\nPerformance: %.2f" % model_stats.areaUnderROC)
```

# Example of classifying text

- Positive labels:
  - ['her', 'him', 'he', 'she', 'them', 'us', 'they', 'himself', 'herself', 'we']
- Number of examples: **5746**
- Number of examples: **2873 positive**, **2873 negative**
- Number of training examples: **4607**
- Number of test examples: **1139**
- training iterations: **21**
- Test AUC: **0.87**

# Predicting the endword

- Positive label: **'it'**
- Number of examples: **438**
- Number of examples: **219 positive**, **219 negative**
- Number of training examples: **340**
- Number of test examples: **98**
- Test AUC: **0.85**

# Recap

- Window function SQL
- **E**xtract
- **T**ransform
- **S**elect
- Train
- Predict
- Evaluate