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

Let's practice

INTRODUCTION TO SPARK SQL IN PYTHON

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

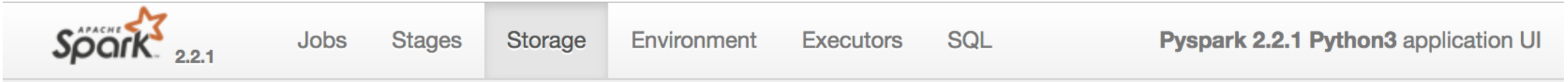
Spark Jobs (?)

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

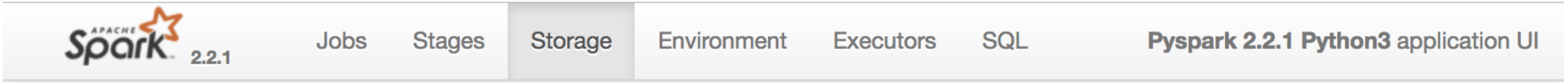
Spark catalog operations

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

Spark Catalog

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

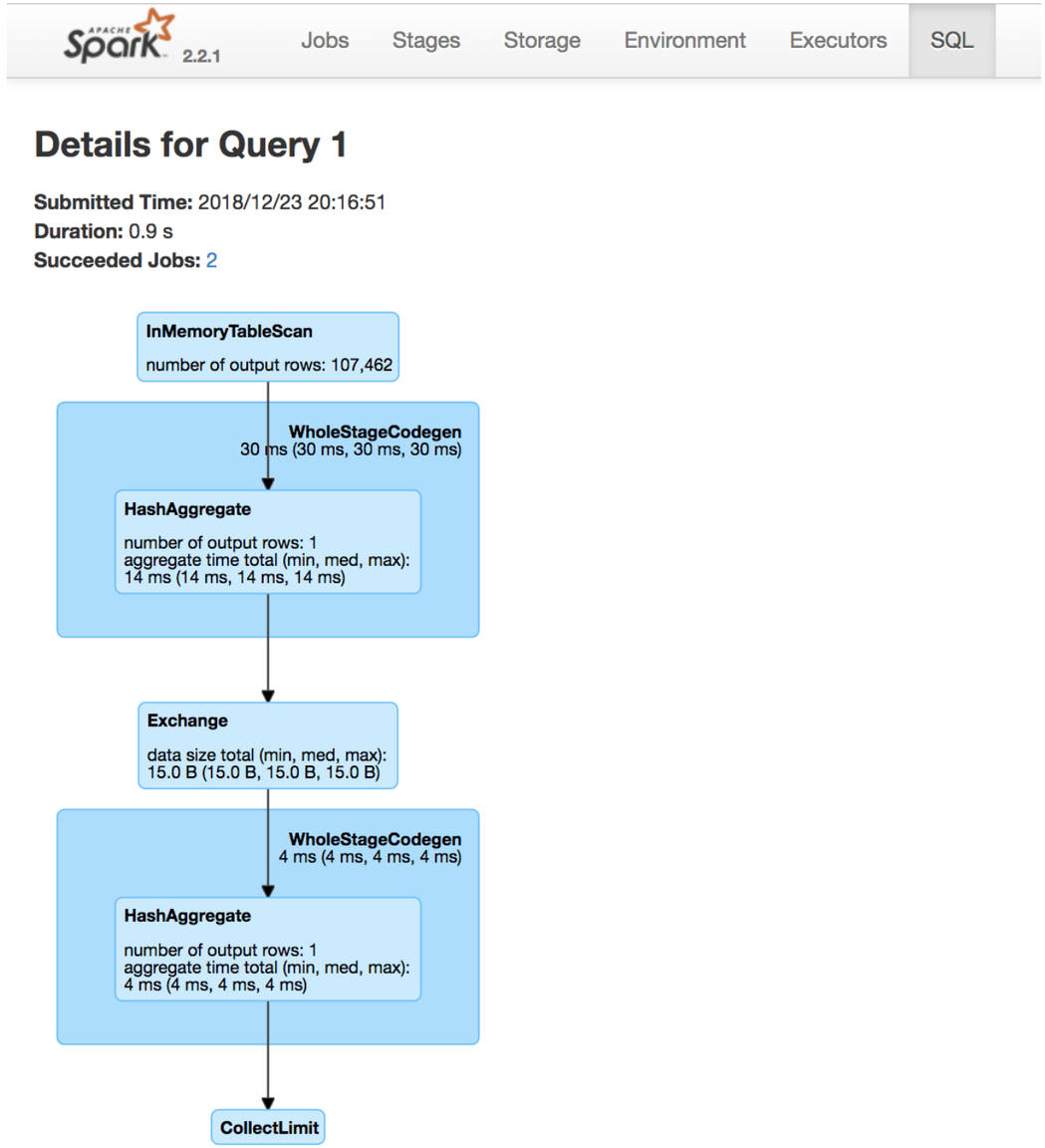
```
[Table(name='text', database=None, description=None, tableType='TEMPORARY', isTempor
```



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

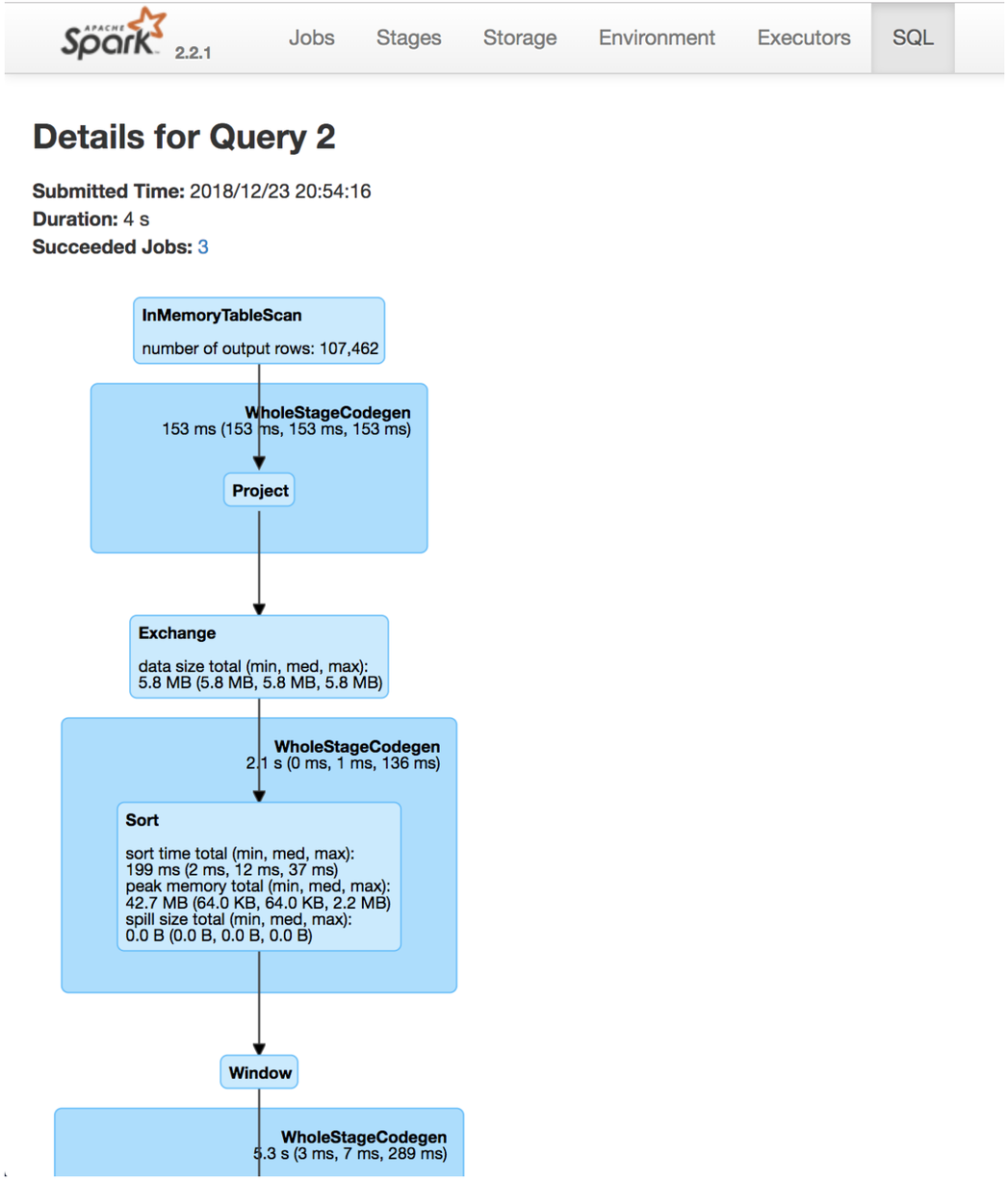
Spark UI Storage Tab

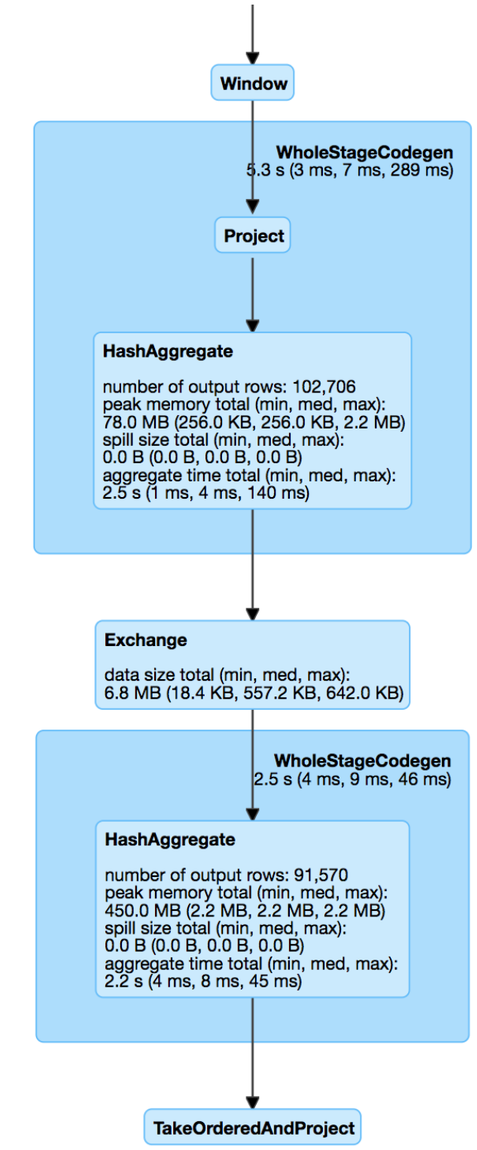
Shows where data partitions exist

- 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

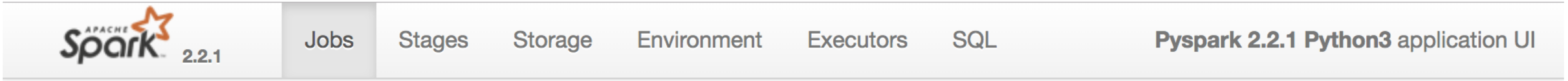
▸ Details

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

Let's practice

INTRODUCTION TO SPARK SQL IN PYTHON

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
```

Let's practice!

INTRODUCTION TO SPARK SQL IN PYTHON

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)
...
```

Let's practice

INTRODUCTION TO SPARK SQL IN PYTHON