

# COMP5349 – Cloud Computing

## Week 8: Spark DataFrame and Execution Plan

Dr. Ying Zhou  
School of Computer Science



# Week 7

## ■ We covered basics of Spark execution

- ▶ The driver program, which is the main script that creates SparkContext, defining the data flow DAG, etc needs to run somewhere, inside spark cluster or on an external host
- ▶ The actual data flow DAG runs inside spark in a number of executors
- ▶ The executor number and capacity can be configured by developer or use a preset default value

## ■ What happens when Spark application runs on YARN?

- ▶ Spark has its own ApplicationMaster, runs in a container
- ▶ Each executor runs in a container
- ▶ The driver program, in cluster mode, runs alongside AM in the same container
- ▶ The driver program, in client mode, runs on a client machine and communicate with the AM in the cluster.



# Week 7

## ■ What happens inside executor

- ▶ All executors stay through out the program life cycle
- ▶ An executor can run multiple threads, controlled by property *executor-cores*
- ▶ One executor runs SparkAM and/or driver program
- ▶ All others run tasks belonging to some stages
- ▶ Each task represents a sequence of transformations that can operate on single or limited parent partitions
- ▶ Executors can run tasks concurrently and subsequently
- ▶ Spark offers PROCESS\_LOCAL data locality



# Outline

## ■ Data Sharing in Spark

- ▶ Understanding Closure
- ▶ Accumulator and Broadcast Variables
- ▶ RDD persistence

## ■ Spark DataFrame

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

# Recap: Data Sharing in MapReduce

- The mappers and reducers are designed to run independently on different partitions of the input data on different nodes
  - ▶ They are written as separate classes/functions
- Common data sharing mechanisms:
  - ▶ Simple typed read-only data are shared as properties attached to configuration object
  - ▶ Small files can be put in distributed cache (not covered)
  - ▶ Global read/write data can be implemented as counter (not covered)



# Data Sharing in Spark

- Spark program may be written as a single script
  - ▶ But different pieces run in different places
    - Driver and executors
  - ▶ Data sharing happens implicitly through closure or just by referring to the variable by name
  - ▶ We need to make things explicit sometimes to ensure
    - Correctness of the result
    - Better performance

```
counter = 0          driver
rdd = sc.parallelize(data)  executor

# Wrong: Don't do this!!
def increment_counter(x):
    global counter
    counter += x
rdd.foreach(increment_counter)  executor

print("Counter value: ", counter)  driver
```

<https://spark.apache.org/docs/latest/rdd-programming-guide.html#understanding-closures->



# Understand Closure

- “Prior to execution, Spark computes the task’s closure. The closure is those variables and methods which must be visible for the executor to perform its computations on the RDD (in this case `foreach()`). This closure is serialized and sent to each executor.”
- Each executor gets a copy of the variables within the closure, they will update the copy independently
- The variables declared in the driver program stays in the memory of the node that runs the driver, they are not updated by executor
- In rare case when everything runs in the same executor, the variables may get updated.



# Understand Closure (cont'd)

The variables declared in the driver program

```
counter = 0
rdd = sc.parallelize(data)

# Wrong: Don't do this!!
def increment_counter(x):
    global counter
    counter += x
rdd.foreach(increment_counter)

print("Counter value: ", counter)
```

It is referenced in a function

When the function is sent to execute in an executor, this variable becomes part of the closure of the task, a copy of it is created and sent to the task. If there are many tasks, each gets its own copy

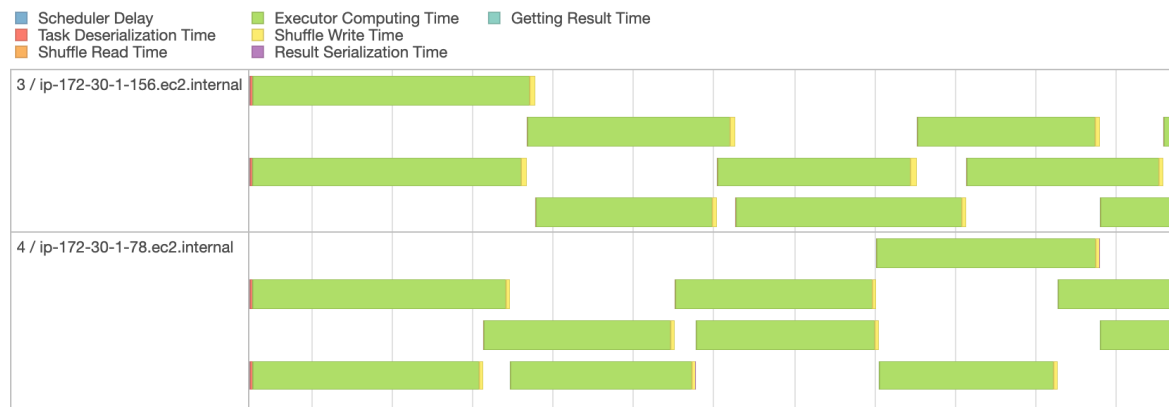
This is the original variable defined in the driver program





# Implications

- The implicit closure based share can be used to share small, read-only program parameters
- Some read-only data needs to be shared is relatively large
  - ▶ E.g. a dictionary for spelling check
- Sharing large read-only data through closure is not efficient
  - ▶ The data is copied for each task and shipped to the node running the task
  - ▶ If we have 20 tasks that needs to use the dictionary
    - 20 copies will be made
  - ▶ If we only use 10 executors, and they run on 6 nodes
    - Only 6 copies are required



# Broadcast Variables

- Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks
- The variables are distributed to each node using efficient broadcast algorithms
- Broadcast variables are created from a variable `v` by calling `SparkContext.broadcast(v)`. The broadcast variable is a wrapper around `v`, and its value can be accessed by calling the `value` method.

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
<pyspark.broadcast.Broadcast object at 0x102789f10>

>>> broadcastVar.value
[1, 2, 3]
```



# Accumulators

- Spark does not support general read-write shared variables
- Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.
  - ▶ Similar as counters in MapReduce
  - ▶ Spark natively supports accumulators of numeric types, and programmers can add support for new types.

# Sharing RDD

- In program code, RDDs can be assigned to variables
  - ▶ They are different to variables of programming language supported types
- RDD is a distributed data structure and is materialized in a lazy manner
  - ▶ All transformations in Spark are *lazy*, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program.
  - ▶ By default, each transformed RDD may be recomputed each time you run an action on it.
  - ▶ Spark is able to skip unnecessary re-computation sometimes

```
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```



# RDD persistence

- RDD persistence is a way to share RDD across jobs
- When you persist an RDD, each node stores any partitions of it that it computes in memory or storage and reuses them in other actions on that dataset (or datasets derived from it)
- An RDD can be persisted using the `persist()` or `cache()` methods on it.
  - ▶ The first time it is computed in an action, it will be kept in memory on the nodes.
  - ▶ A key tool for fast iterative algorithms



# Outline

## ■ Data Sharing in Spark

## ■ Spark DataFrame

- ▶ Basic concept
- ▶ A simple example with common operations
  - Query vs. Job
  - Filter-GroupBy query execution plan
  - One query two job example



# RDD API

- RDD based API is in the first release of Spark
- They are quite primitive compared with APIs of recent data analytic packages
  - ▶ Lack of sophisticated way to handle structured or semi structured data format
  - ▶ Lack of popular data manipulation operations, e.g. slicing
- Performance tuning requires largely manual efforts
  - ▶ Program should be carefully designed
    - Spark will execute the DAG as they are expressed as code
    - Combiners will be automatically used, which is an improvement compared with MapReduce
  - ▶ Execution parameters should be carefully chosen
    - Number of executors to use, number of partitions at each stage are either specified or default value is used
- More flexible, gives better control, allows better understanding of the Spark internals



# Spark SQL

- Spark SQL is Spark's interface for working with structured and semistructured data.
  - ▶ Data with schema
  - ▶ Known set of fields for each record, and their types
- Spark SQL makes loading and querying such data much easier and more efficient
  - ▶ Provides DataFrame abstraction
    - Can read data in a variety of formats
  - ▶ Query can be expressed using SQL
  - ▶ An optimization engine to pick the best physical query plan
- A Spark **DataFrame** contains an RDD of **Row** objects, each representing a record. A **DataFrame** also knows the schema of its rows.



# DataFrame

- The DataFrame concept comes from statistics software and is gradually accepted in other data analytic software
  - ▶ It is a way to organize tabular data: rows, named columns
- Spark's **DataFrame** concept is quite similar to those used in **R** and in **Pandas**
- A **DataFrame** can be created from an existing RDD, or from various data sources
- **DataFrame** is the common data structure in all spark supporting languages
- Java and Scala have another version called **DataSet**
  - ▶ DataSet provides strong typing supports



# DataFrame APIs

## ■ The entry point of Spark SQL is SparkSession

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

## ■ Spark DataFrame supports operations like filtering, slicing, groupBy, etc

- ▶ DataFrame support larger set of operations than those of RDD operations
- ▶ More restrictive in terms of parameters
- ▶ Most RDD operations take a user defined function as parameter



# Example DF API vs RDD API

## `filter(condition)`

Filters rows using the given condition.

**Parameters:** `condition` – a `Column` of `types.BooleanType` or a string of SQL expression.

```
>>> df.filter(df.age > 3).collect()
[Row(age=5, name='Bob')]
>>> df.where(df.age == 2).collect()
[Row(age=2, name='Alice')]
```

```
>>> df.filter("age > 3").collect()
[Row(age=5, name='Bob')]
>>> df.where("age = 2").collect()
[Row(age=2, name='Alice')]
```

## `filter(f)`

Return a new RDD containing only the elements that satisfy a predicate.

```
>>> rdd = sc.parallelize([1, 2, 3, 4, 5])
>>> rdd.filter(lambda x: x % 2 == 0).collect()
[2, 4]
```



# An Example

- We use the movie rating data set

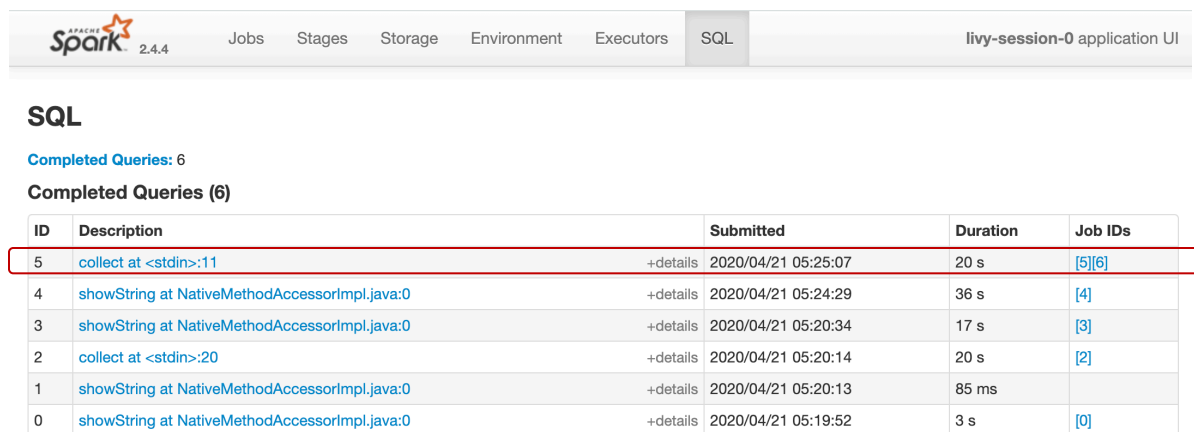
uid	mid	rate	ts
1	16	4.0	1217897793
1	24	1.5	1217895807
1	32	4.0	1217896246
1	47	4.0	1217896556
1	50	4.0	1217896523

```
1 from pyspark.sql import SparkSession
2 from pyspark.sql.types import StructType, StructField, StringType, IntegerType, FloatType
3
4 spark = SparkSession \
5     .builder \
6     .appName("Python Spark SQL basic example") \
7     .getOrCreate()
8
9 rating_data = 's3://comp5349-data/week6/ratings.csv'
10
11 rating_schema = StructType([
12     StructField("uid", StringType(), True),
13     StructField("mid", StringType(), True),
14     StructField("rate", FloatType(), True),
15     StructField("ts", IntegerType(), True)])
16
17 ratings = spark.read.csv(rating_data, header=False, schema=rating_schema)
18 ratings.show(5)
19 ratings.describe(['rate']).show()
20 ratings.filter("mid<=5").groupBy('mid').avg('rate').collect()
21 ratings.groupBy('mid').count().withColumnRenamed("count", "n").filter("n< 5").show()
```

# Query, Job, Stage, task in DataFrame

## ■ DataFrame is part of SparkSQL API

- ▶ At programming level there is the query concept
- ▶ At execution level, the concept of job and stage, tasks are still there and have the same definition
- ▶ There is no one to one mapping between query and job
  - All depends on programming logic
    - If we only need to return the results to driver after a few queries, we may have one job containing many queries
  - Optimization mechanism in SparkSQL
    - Spark optimization may break down a single query into multiple jobs



The screenshot shows the Spark SQL history page. At the top, there's a navigation bar with tabs: Jobs, Stages, Storage, Environment, Executors, and SQL (which is selected). To the right of the tabs, it says "livy-session-0 application UI". Below the navigation bar, the title "SQL" is displayed. Underneath, it says "Completed Queries: 6". Then, there's a section titled "Completed Queries (6)" which contains a table with 6 rows. The first row is highlighted with a red border. The table has columns: ID, Description, Submitted, Duration, and Job IDs.

ID	Description	Submitted	Duration	Job IDs
5	collect at <stdin>:11	+details 2020/04/21 05:25:07	20 s	[5][6]
4	showString at NativeMethodAccessorImpl.java:0	+details 2020/04/21 05:24:29	36 s	[4]
3	showString at NativeMethodAccessorImpl.java:0	+details 2020/04/21 05:20:34	17 s	[3]
2	collect at <stdin>:20	+details 2020/04/21 05:20:14	20 s	[2]
1	showString at NativeMethodAccessorImpl.java:0	+details 2020/04/21 05:20:13	85 ms	
0	showString at NativeMethodAccessorImpl.java:0	+details 2020/04/21 05:19:52	3 s	[0]

New query tab  
for SparkSQL  
applications in  
history server

# An example (job break down)

```
1 from pyspark.sql import SparkSession
2 from pyspark.sql.types import StructType, StructField, StringType, IntegerType, FloatType
3
4 spark = SparkSession \
5     .builder \
6     .appName("Python Spark SQL basic example") \
7     .getOrCreate()
8
9 rating_data = 's3://comp5349-data/week6/ratings.csv'
10
11 rating_schema = StructType([
12     StructField("uid", StringType(), True),
13     StructField("mid", StringType(), True),
14     StructField("rate", FloatType(), True),
15     StructField("ts", IntegerType(), True)])
16
17 ratings = spark.read.csv(rating_data, header=False, schema=rating_schema)
18 ratings.show(5)
19 ratings.describe(['rate']).show()
20 ratings.filter("mid<=5").groupBy('mid').avg('rate').collect()
21 ratings.groupBy('mid').count().withColumnRenamed("count", "n").filter("n < 5").show()
```

# An Example: show data frame content

```
rating_data = 's3://comp5349-data/week6/ratings.csv'

rating_schema = StructType([
    StructField("uid", StringType(), True),
    StructField("mid", StringType(), True),
    StructField("rate", FloatType(), True),
    StructField("ts", IntegerType(), True)])

ratings = spark.read.csv(rating_data, header=False, schema=rating_schema)
ratings.show(5)
ratings.describe(['rate']).show()
ratings.filter("mid<=5").groupBy('mid').avg('rate').collect()
ratings.groupBy('mid').count().withColumnRenamed("count", "n").filter("n< 5").show()
```

## ▼ Spark Job Progress

### ▼ Job [0]: showString at NativeMethodAccessorImpl.java:0

Progress for showString at NativeMethodAccessorImpl.java:0 Job Progress: 1/1 Tasks Complete

Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [0]: showString at Na...java:0	COMPLETE	1/1	2.244	

#### ▼ Completed Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output
0	Job group for statement 1 showString at NativeMethodAccessorImpl.java:0	2020/04/22 12:33:22	2 s	1/1	16.0 KB	

The csv file is 500+ MB ~ 5 HDFS blocks  
Only one task is used, very likely the result of  
SparkSQL optimization

Because we only need to show 5 records, one  
partition is needed and not all data need to be  
read in



# An Example: describe and show

The following cell shows simple summary statistics of column rate

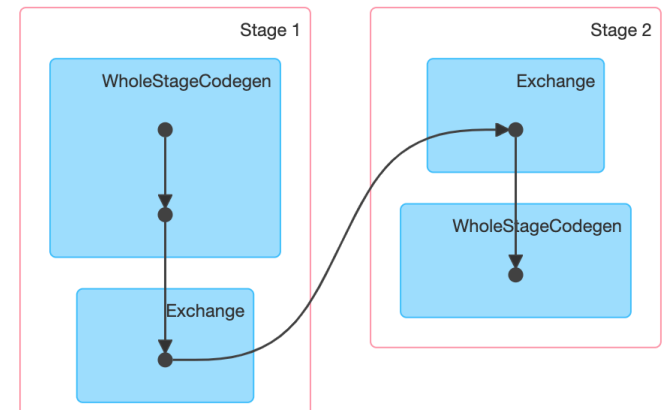
```
ratings.describe(['rate']).show()
```

Progress for describe at NativeMethodAccessorImpl.java:0 Job Progress: 6/6 Tasks Complete



Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [1]: describe at Nati...java:0	COMPLETE	5/5	35.349	
Stage [2]: describe at Nati...java:0	COMPLETE	1/1	0.237	

```
+-----+-----+
|summary|          rate|
+-----+-----+
|  count|        21063128|
|   mean|  3.5214483575279036|
|  stddev| 1.0584560524401594|
+-----+-----+
```



## Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	Job group for statement 1 <a href="#">describe at NativeMethodAccessorImpl.java:0</a> <small>+details</small>	2020/04/22 12:33:43	0.3 s	1/1			520.0 B	
1	Job group for statement 1 <a href="#">describe at NativeMethodAccessorImpl.java:0</a> <small>+details</small>	2020/04/22 12:33:25	18 s	5/5	542.0 MB			520.0 B





# An Example: filter-groupby

Stage boundary

```
ratings.filter("mid<=5").groupBy('mid').avg('rate').collect()
```

## ▼ Spark Job Progress

### ▼ Job [2]: collect at <stdin>:1

Progress for collect at <stdin>:1		Job Progress: 205/205 Tasks Com... <div></div>		
Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [3]: collect at <stdin>:1	COMPLETE	5/5 <div></div>	40.125	
Stage [4]: collect at <stdin>:1	COMPLETE	200/200 <div></div>	3.565	

```
[Row(mid='3', avg(rate)=3.181209161111478), Row(mid='5', avg(rate)=3.082546682284848), Row(mid='1', avg(rate)=3.9218118016437344), Row(mid='4', avg(rate)=2.880020597322348), Row(mid='2', avg(rate)=3.2173265592151368)]
```

File partitioned by HDFS

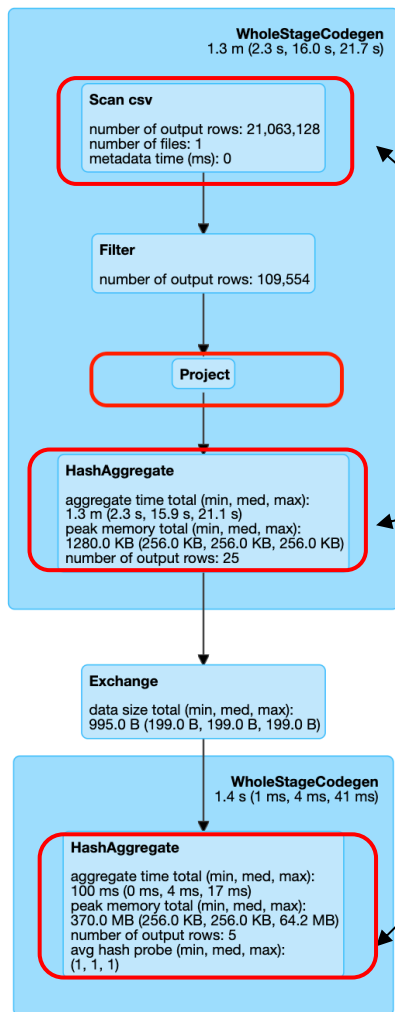
spark.sql.shuffle.partitions=200

Stage 3 takes much longer time to finish than stage 4.

# filter-groupBy physical execution plan

File I/O can be saved by caching the previous result

```
ratings = spark.read.csv(rating_data,header=False, schema=rating_schema)
ratings.describe(['rate']).show()
ratings.filter("mid<=5").groupBy('mid').avg('rate').collect()
```



Local reduce,  
similar effect  
as combiner

The first stage has 5 tasks, the first 4 running on 128M data, the last one running on smaller sized data, each task does the filtering, and local groupby-average task

Each stage 1 task produce 5 records (mid, sum, count), in total 25 records are produced

Final reduce stage

Stage 2 only has to process 25 records of (mid, sum, count) to produce the final (mid, avg), yet it has 200 tasks!

== Physical Plan ==

```
*(2) HashAggregate(keys=[mid#1], functions=[avg(cast(rate#2 as double))], output=[mid#1, avg(rate)#100])
+- Exchange hashpartitioning(mid#1, 200)
   +- *(1) HashAggregate(keys=[mid#1], functions=[partial_avg(cast(rate#2 as double))], output=[mid#1, sum#105, count#106L])
      +- *(1) Project [mid#1, rate#2]
         +- *(1) Filter (isNotNull(mid#1) && (cast(mid#1 as int) <= 5))
            +- *(1) FileScan csv [mid#1,rate#2] Batched: false, Format: CSV, Location: InMemoryFileIndex[s3://comp5349-2019/lab-data/week6/ratings.csv],
               PartitionFilters: [], PushedFilters: [IsNotNull(mid)], ReadSchema: struct<mid:string,rate:float>
```



# What do the 200 tasks end up?

- Most of them are doing nothing, in fact at most 5 tasks are needed because there are only 5 keys after applying the filter ( $\text{mid} \leq 5$ )
- The default 200 partition is set with much larger cluster and data set in mind.

164	178	0	SUCCESS	PROCESS_LOCAL	1	ip-172-30-3-56.ec2.internal stdout stderr	2020/04/23 00:55:24	3 ms		0.0 B / 0
165	179	0	SUCCESS	PROCESS_LOCAL	1	ip-172-30-3-56.ec2.internal stdout stderr	2020/04/23 00:55:24	11 ms		0.0 B / 0
166	15	0	SUCCESS	NODE_LOCAL	1	ip-172-30-3-56.ec2.internal stdout stderr	2020/04/23 00:55:22	0.3 s	5 ms	379.0 B / 5
167	180	0	SUCCESS	PROCESS_LOCAL	2	ip-172-30-3-56.ec2.internal stdout stderr	2020/04/23 00:55:24	16 ms		0.0 B / 0
168	181	0	SUCCESS	PROCESS_LOCAL	2	ip-172-30-3-56.ec2.internal stdout stderr	2020/04/23 00:55:24	8 ms		0.0 B / 0

# Effect of partition number

```
ratings.filter("mid<=5").groupBy('mid').avg('rate').collect()
```

## ▼ Spark Job Progress

Running on single node cluster

### ▼ Job [2]: collect at <stdin>:1

Progress for collect at <stdin>:1		Job Progress: 205/205 Tasks Com... <div></div>		
Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [3]: collect at <stdin>:1	COMPLETE	5/5 <div></div>	40.125	
Stage [4]: collect at <stdin>:1	COMPLETE	200/200 <div></div>	3.565	

Running on a cluster with two worker nodes

```
ratings.filter("mid<=5").groupBy('mid').avg('rate').collect()
```

=3.92181

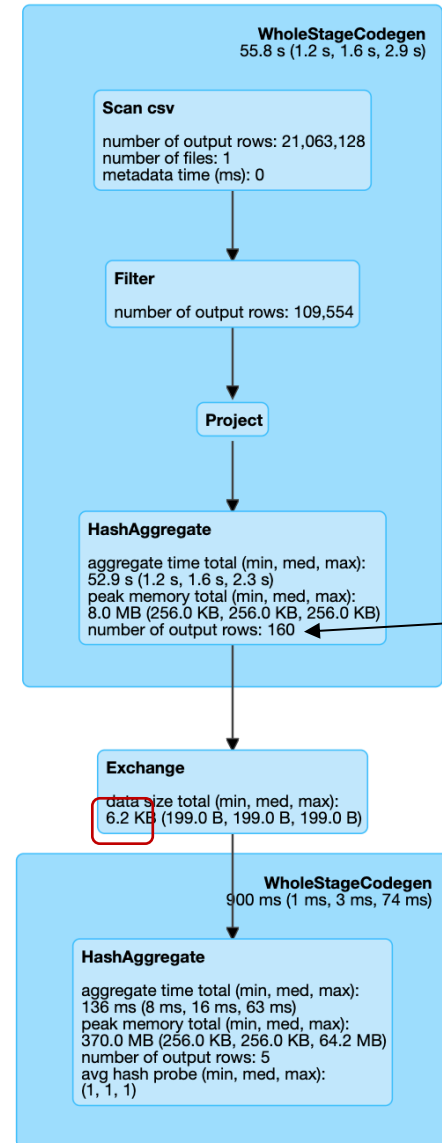
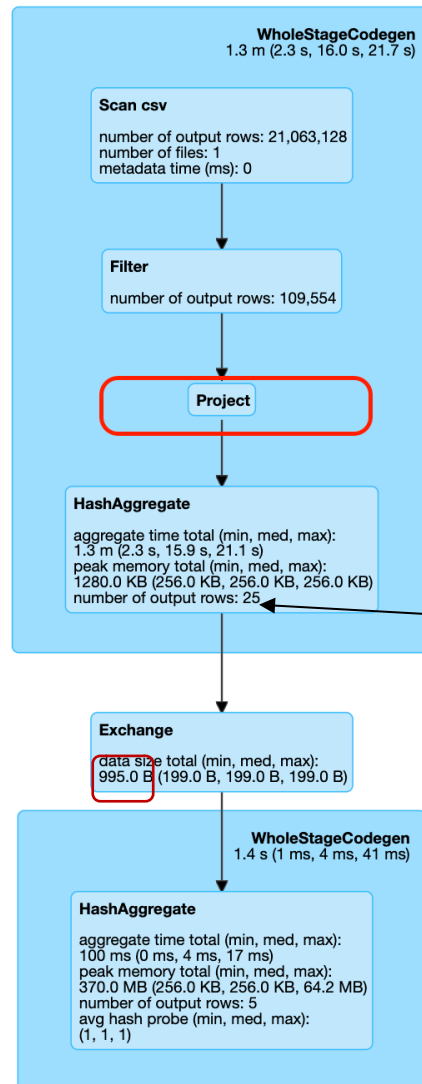
spark.default.parallelism=32

## ▼ Spark Job Progress

### ▼ Job [2]: collect at <stdin>:1

Progress for collect at <stdin>:1		Job Progress: 232/232 Tasks Com... <div></div>		
Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [3]: collect at <stdin>:1	COMPLETE	32/32 <div></div>	14.956	
Stage [4]: collect at <stdin>:1	COMPLETE	200/200 <div></div>	1.254	

# Effect of partition number



# Parallelism Properties

- Input partitions
- `Spark.default.parallelism`
  - ▶ Default number of partitions in RDDs returned by transformations like `join`, `reduceByKey`, and `parallelize` when not set by user.
  - ▶ Usually set to the number of cores on all executor nodes
  - ▶ An RDD API feature, but PySpark does use it on SparkSQL
- `Spark.sql.shuffle.partitions`
  - ▶ Configures the number of partitions to use when shuffling data for joins or aggregations.
  - ▶ Only applicable in Spark SQL
- Explicit set in various transformations
  - ▶ **`repartition(numPartitions)`**,
  - ▶ **`groupByKey([numPartitions])`**, **`join(otherDataset, [numPartitions])`**,



# Window Function

- Window functions operate on a set of rows and return a single value for each row from the underlying query
- It is particularly useful if we need to operate based on groupBy or aggregateBy results
  - ▶ E.g. remove all movies with less than or equal to 100 ratings
- It involves defining the window/frame, then apply query on the window

```
1 from pyspark.sql.window import Window
2 from pyspark.sql.functions import count
3 window = Window \
4     .partitionBy("mid") \
5     #.orderBy("ts")
6 pop_rating = ratings.withColumn("n", count("mid") \
7     .over(window)) \
8     .filter("n > 100") \
9     .drop("n")
10 pop_rating.show(5)
```



# Window Function Execution

```
from pyspark.sql.window import Window
from pyspark.sql.functions import count
window = Window \
    .partitionBy("mid") \
    #.orderBy("ts")
pop_rating = ratings.withColumn("n", count("mid") \
    .over(window)) \
    .filter("n > 100") \
    .drop("n")
pop_rating.show(5)
```

## ▼ Spark Job Progress

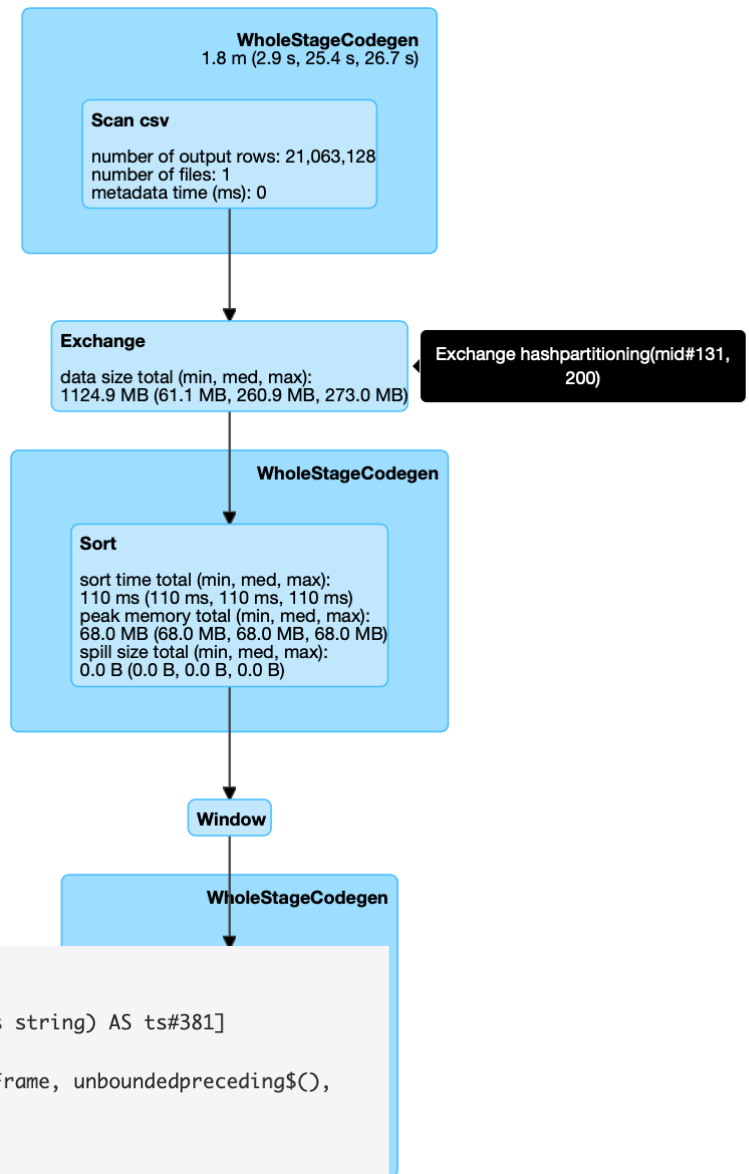
### ▼ Job [4]: showString at NativeMethodAccessorImpl.java:0

Progress for showString at NativeMethodAccessorImpl.java... Job Progress: 6/6 Tasks Complete				
Stage [ID]: name at [source]:[line]	Status	Task Progress	Elapsed Time (seconds)	Failed Task Logs
Stage [7]: showString at Na...java:0	COMPLETE	5/5	19.927	
Stage [8]: showString at Na...java:0	COMPLETE	1/1	0.572	

== Physical Plan ==

CollectLimit 6

```
+-- *(3) Project [uid#232, mid#233, cast(rate#234 as string) AS rate#380, cast(ts#235 as string) AS ts#381]
  +- *(3) Filter (n#364L > 100)
    +- Window [count(mid#233) windowSpecDefinition(mid#233, specifiedWindowFrame(RowFrame, unboundedPreceding$( ),
unboundedFollowing$( ))) AS n#364L], [mid#233]
      +- *(2) Sort [mid#233 ASC NULLS FIRST], false, 0
        +- Exchange hashpartitioning(mid#233, 200)
          +- *(1) FileScan csv [uid#232,mid#233,rate#234,ts#235] Batched: false, Format: CSV, Location:
InMemoryFileIndex[s3://comp5349-2019/lab-data/week6/ratings.csv], PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<uid:string,mid:string,rate:float,ts:int>
```





# Example of one query two jobs

- Load the movie data to find romance movie in 1939
- For each movie find out the average rating

```
1 movie_data = 's3://comp5349-data/week6/movies.csv'
2
3 movies_schema = StructType([
4     StructField("mid", StringType(), True),
5     StructField("title", StringType(), True),
6     StructField("genres", StringType(), True)])
7
8 movies = spark.read.csv(movie_data, header=False, schema=movies_schema)
9 romance_1939 = movies.filter(movies.title.contains("1939") & movies.genres.contains("Romance"))
10 romance_1939.drop('genres').join(ratings, 'mid', 'inner') \
11 .drop('ts', 'uid').groupBy('title').avg('rate').collect()
```

# Execution Summary

```
8 movies= spark.read.csv(movie_data,header=False, schema=movies_schema)
9 romance_1939=movies.filter(movies.title.contains("1939") & movies.genres.contains("Romance"))
10 romance_1939.drop('genres').join(ratings,'mid','inner') \
11 .drop('ts','uid').groupBy('title').avg('rate').collect()
```

## ▼ Spark Job Progress

Created by execution engine

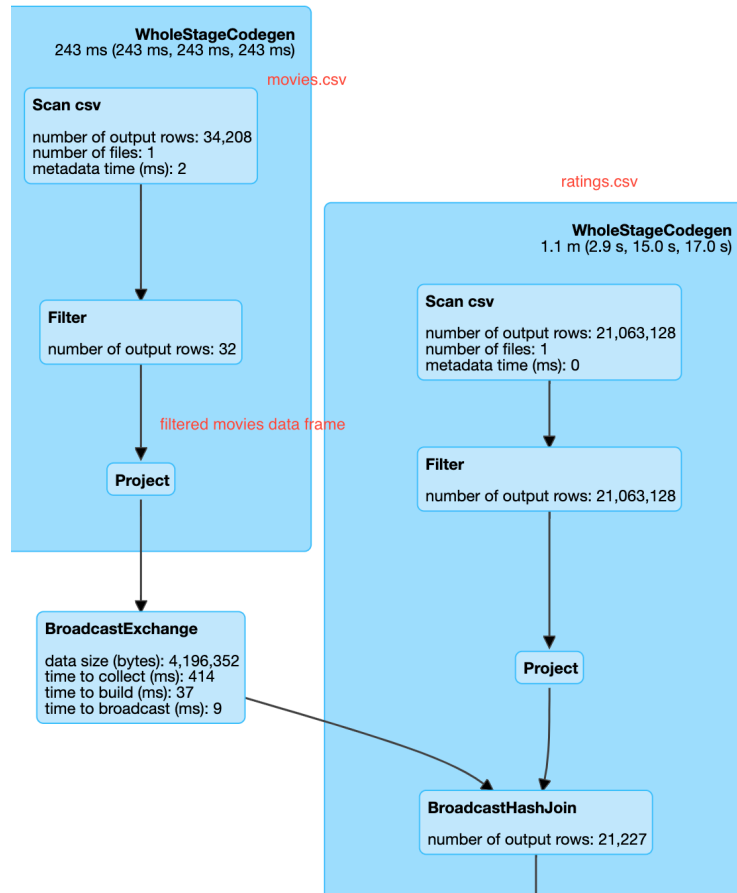
### ▼ Job [5]: collect at <stdin>:11

Progress for collect at <stdin>:11		Job Progress: 1/1 Tasks Complete		<div></div>	
Stage [ID]: name at [source]:[line]	Status	Task Progress		Elapsed Time (seconds)	Failed Task Logs
Stage [9]: collect at <stdin>:11	COMPLETE	1/1	<div></div>	0.526	

### ▼ Job [6]: collect at <stdin>:11

Progress for collect at <stdin>:11		Job Progress: 205/205 Tasks Com...		<div></div>	
Stage [ID]: name at [source]:[line]	Status	Task Progress		Elapsed Time (seconds)	Failed Task Logs
Stage [10]: collect at <stdin>:11	COMPLETE	5/5	<div></div>	17.044	
Stage [11]: collect at <stdin>:11	COMPLETE	200/200	<div></div>	2.368	

# Execution Plan of the join query



“When Spark deciding the join methods, the broadcast hash join (i.e., BHJ) is preferred”

== Physical Plan ==

```

*(2) HashAggregate(keys=[mid#1], functions=[avg(cast(rate#2 as double))], output=[mid#1, avg(rate)#100])
+- Exchange hashpartitioning(mid#1, 200)
   +- *(1) HashAggregate(keys=[mid#1], functions=[partial_avg(cast(rate#2 as double))], output=[mid#1, sum#105, count#106L])
      +- *(1) Project [mid#1, rate#2]
         +- *(1) Filter (isNotNull(mid#1) && (cast(mid#1 as int) <= 5))
            +- *(1) FileScan csv [mid#1,rate#2] Batched: false, Format: CSV, Location: InMemoryFileIndex[s3://comp5349-2019/lab-data/week6/ratings.csv],
PartitionFilters: [], PushedFilters: [IsNotNull(mid)], ReadSchema: struct<mid:string,rate:float>
  
```



# DataFrame and RDD

- DataFrame and RDD can be easily converted to each other
- To convert a DataFrame as RDD
  - ▶ Call `rdd` or `JavaRDD` on the DataFrame
  - ▶ E.g. `teenagers.rdd.map(lambda p: "Name: " + p.name)`
- To convert a RDD to DataFrame
  - ▶ `Spark.createDataFrame(RDD, optionalSchema)`
  - ▶ E.g. `schemaPeople = spark.createDataFrame(people)`



# References

- **Spark RDD programming guide**

- ▶ <https://spark.apache.org/docs/latest/rdd-programming-guide.html>

- **Spark SQL, DataFrame and DataSets Guide**

- ▶ <https://spark.apache.org/docs/latest/sql-programming-guide.html>

- **Physical Plans in Spark SQL**

- ▶ [https://databricks.com/session\\_eu19/physical-plans-in-spark-sql](https://databricks.com/session_eu19/physical-plans-in-spark-sql)