Spark Performance Tuning Using Partitioning

(30 minutes read)

(Yusuf Arif, Sr. Data Architect, Big Data - usuf@amazon.com)

Data Partitioning

Data Partitioning is critical to Spark performance, specially for large volumes of datasets that are to be joined with another dataset. Spark assigns one task for each partition and each worker thread can only process one task at a time. With too few partitions, Spark won't be able to utilize all the cores available in the cluster resulting in data-skew problem. On the other hand - too many partitions introduce overhead for Spark to manage too many tasks.

It's very important to **partition data appropriately on the columns that are in the filter clause** of the query or columns on which **data-frames are joined**.

What is a Partition in Spark?

A logical atomic chunk of data. In Spark, partitions are distributed among nodes in logical chunks. Partitions are basic unit of parallelism in Spark. **Shuffling** refers to movement of data amongst these partitions spanning across the nodes in the cluster.

Spark can run the same task concurrently for every partition of the dataset - thereby allowing **massively parallelized processing** in a distributed environment.

Why is Shuffling an issue?

Shuffling inherently involves **marshalling** and **unmarshalling** which is a pre-requisite for data **storage** and **transmission across the network**. Typically, when data must be moved between different parts of a physical or logical storage, it needs to be **serialized** (marshalled) and **deserialized** (unmarshalled), which means data must be transformed from the memory representation to a format suitable for transmission and then back to the memory representation for the storage after the transmission.

Cores and Partitions Relationship

Partitions should be a multiple of the total number of cores in the cluster. E.g. If your cluster has 100 cores, you should have at least 100 partitions to allow for 1:1 ratio of Parallelism in Spark. In practice, the number of partitions should be 3-5 x more than the number of cores. If you have 100 cores and 500 partitions of the dataset, the parallelism ratio would be 1:5, that means 100 cores (a task is running on a core, which is a Java Virtual Machine). It is important to keep in mind that each executor node typically has 2 to 4 vCores. More powerful executor machines may have 8 vCores. Each executor is a physical EC2 node with 16 GB, 32 GB or 64 GB RAM depending on the hardware appliance. A 32 GB executor node with 4 vCores means each vCore has 8 GB of RAM, running its own JVM. You should always set aside 20% of the vCore RAM for the heap size, the remaining 80% is the actual processing power of the vCore, logically available to run a task in parallel.

What is Parallelism?

Parallelism is a distributed computing paradigm that refers to each task independently and concurrently processing a logical chunk of data.

What is a task? A Stage? A Job?

A **task** is a unit of work that is sent to the executor by Spark's Driver program. Each Stage comprises of tasks, one **task** per partition.

Lazy Evaluation - In Spark, a task is not executed until an action is performed. collect(), take() and show() are examples of action.

When code is submitted to Spark, the Driver program analyzes the code and breaks it into **Jobs**. Each Job comprising of 1 or more **Stage**, each Stage comprising of 1 or more tasks - depending on the number of vCores in the Cluster.

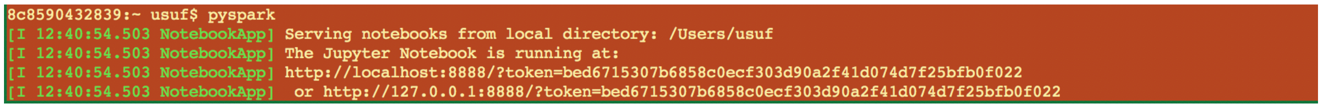
RDD

Resilient Distributed Dataset, a resilient dataset in Spark that is distributed across nodes in the cluster. RDDs get partitioned automatically without programmer's intervention. However, most of the time while dealing with massive analytical ETL or realtime workloads, you would like to adjust the size and number of partitions or the partitioning scheme to the needs of your analytics use-case.

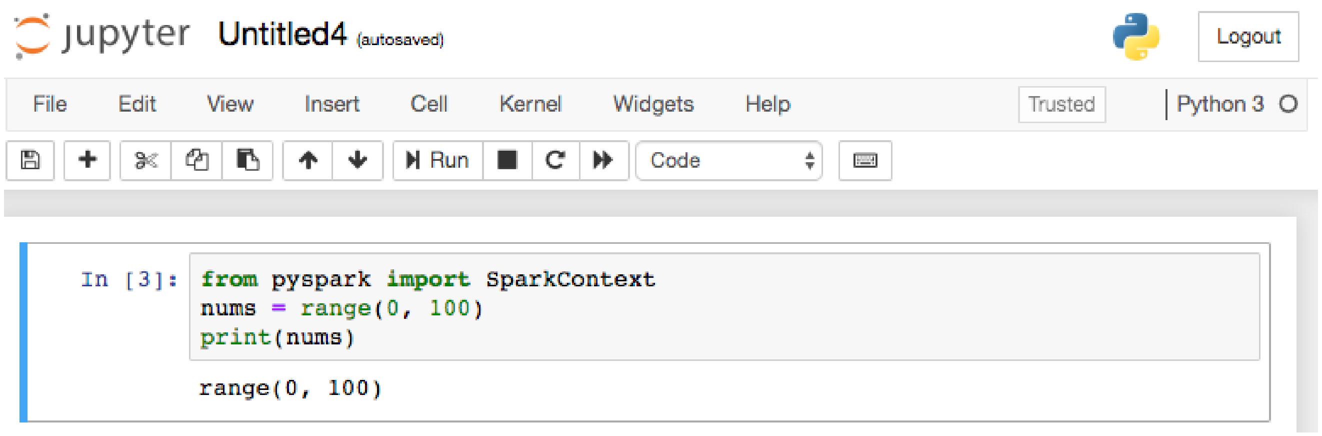
Code Examples:

Partition dataset with parallelize() API

Using Jupyter Notebook with **Spark 2.4** and **Python 3.7** to run as **PySpark** on localhost with "**local[4]**", meaning a **4 vCores cluster.**



```
8c8590432839:~ usuf$ pyspark
[I 12:40:54.503 NotebookApp] Serving notebooks from local directory: /Users/usuf
[I 12:40:54.503 NotebookApp] The Jupyter Notebook is running at:
[I 12:40:54.503 NotebookApp] http://localhost:8888/?token=bed6715307b6858c0ecf303d90a2f41d074d7f25bfb0f022
[I 12:40:54.503 NotebookApp]  or http://127.0.0.1:8888/?token=bed6715307b6858c0ecf303d90a2f41d074d7f25bfb0f022
```



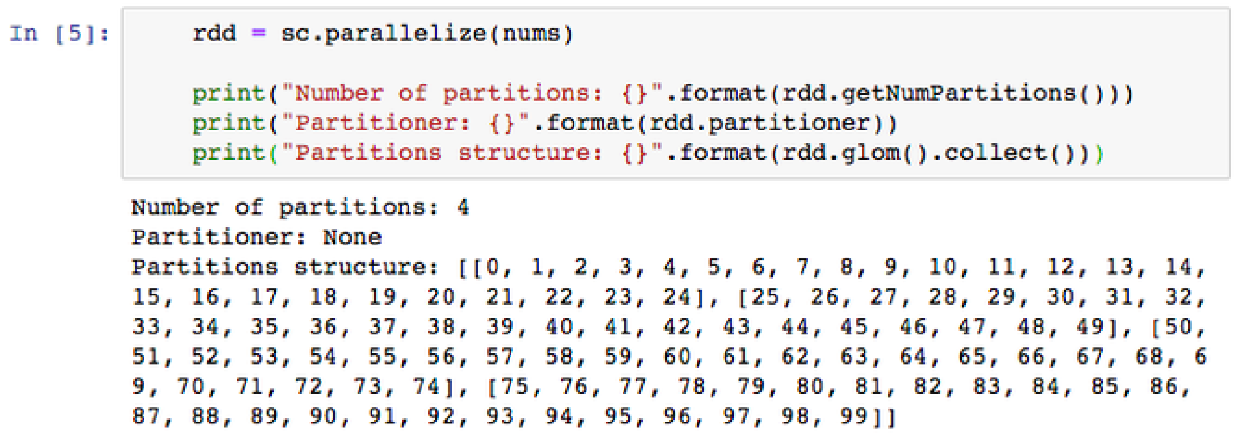
jupyter Untitled4 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [3]: from pyspark import SparkContext
        nums = range(0, 100)
        print(nums)

        range(0, 100)
```

Here, we import SparkContext from pyspark. SparkContext is the entry point, the Driver program per say, which is responsible for distributing the datasets and tasks across the cluster. The dataset is a Python array of comma separated numbers from 0 to 99.

In Jupyter Notebook, SparkContext is already initialized and available as **sc**. If you are not using Jupyter Notebook, then initialize the SparkContext like this:

with SparkContext("local[4]") as sc:



```
In [5]:     rdd = sc.parallelize(nums)

            print("Number of partitions: {}".format(rdd.getNumPartitions()))
            print("Partitioner: {}".format(rdd.partitioner))
            print("Partitions structure: {}".format(rdd.glom().collect()))

        Number of partitions: 4
        Partitioner: None
        Partitions structure: [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
        15, 16, 17, 18, 19, 20, 21, 22, 23, 24], [25, 26, 27, 28, 29, 30, 31, 32,
        33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49], [50,
        51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 6
        9, 70, 71, 72, 73, 74], [75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
        87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]]
```

Spark uses **different partitioning schemes** for various types of resilient distributed datasets (RDD) and operations. Below are the code snippets of these partitioning schemes.

parallelize(nums) - Spark evenly distributes dataset across vCores. We are parallelizing python array containing 100 comma separated values into an RDD with no partitioning scheme.

glom() - API **exposes the structure of created partitions**. It returns an RDD created by coalescing all elements within each partition into a list. Each RDD also possesses information about partitioning schema. You will observe, the dataset of 100 records - nums - is split into 4 equal partitions, each containing 25 records, as the cluster has 4 vCores available. This means we allowed the Spark Driver program to use four local vCores, each vCore is capable of running the task in parallel. This is the ideal situation for parallelism, where each vCore will be able to operate on one partition.

partitioner - inspects partitioner information used to partition the RDD.

In general, smaller, more numerous partitions allow work to be distributed among more workers (Stage comprising of more tasks).

Larger and fewer partitions allow work to be done in larger chunks, reducing shuffle between the nodes. Spark can only run 1 concurrent task for every partition of the RDD, up to the number of cores in your cluster. So, if you have a cluster with 100 cores, you want your RDDs to at least have 100 partitions, possibly even 3-4 x the number of cores.

```
In [6]:     rdd = sc.parallelize(nums, 15)

            print("Number of partitions: {}".format(rdd.getNumPartitions()))
            print("Partitioner: {}".format(rdd.partitioner))
            print("Partitions structure: {}".format(rdd.glom().collect()))
```

```
Number of partitions: 15
Partitioner: None
Partitions structure: [[0, 1, 2, 3, 4, 5], [6, 7, 8, 9, 10, 11, 12], [13,
14, 15, 16, 17, 18, 19], [20, 21, 22, 23, 24, 25], [26, 27, 28, 29, 30, 3
1, 32], [33, 34, 35, 36, 37, 38, 39], [40, 41, 42, 43, 44, 45], [46, 47,
48, 49, 50, 51, 52], [53, 54, 55, 56, 57, 58, 59], [60, 61, 62, 63, 64, 6
5], [66, 67, 68, 69, 70, 71, 72], [73, 74, 75, 76, 77, 78, 79], [80, 81,
82, 83, 84, 85], [86, 87, 88, 89, 90, 91, 92], [93, 94, 95, 96, 97, 98, 9
9]]
```

Here, we increase the number of partitions from 4 to 15. Each partition now has 6 or 7 records. Now 4 tasks can run in parallel on smaller partitions and possibly process faster. As soon as the task completes processing a partition, it can start on the next un-processed partition.

Custom partitions using partitionBy()

Allows applying custom partitioning logic over the dataset.

```
In [7]:     rdd = sc.parallelize(nums) \
                .map(lambda num: (num, num)) \
                .partitionBy(2) \
                .persist()

            print("Number of partitions: {}".format(rdd.getNumPartitions()))
            print("Partitioner: {}".format(rdd.partitioner))
            print("Partitions structure: {}".format(rdd.glom().collect()))
```

```
Number of partitions: 2
Partitioner: <pyspark.rdd.Partitioner object at 0x108fd74a8>
Partitions structure: [[(0, 0), (2, 2), (4, 4), (6, 6), (8, 8), (10, 10),
(12, 12), (14, 14), (16, 16), (18, 18), (20, 20), (22, 22), (24, 24), (2
6, 26), (28, 28), (30, 30), (32, 32), (34, 34), (36, 36), (38, 38), (40,
40), (42, 42), (44, 44), (46, 46), (48, 48), (50, 50), (52, 52), (54, 5
4), (56, 56), (58, 58), (60, 60), (62, 62), (64, 64), (66, 66), (68, 68),
(70, 70), (72, 72), (74, 74), (76, 76), (78, 78), (80, 80), (82, 82), (8
4, 84), (86, 86), (88, 88), (90, 90), (92, 92), (94, 94), (96, 96), (98,
98)], [(1, 1), (3, 3), (5, 5), (7, 7), (9, 9), (11, 11), (13, 13), (15, 1
5), (17, 17), (19, 19), (21, 21), (23, 23), (25, 25), (27, 27), (29, 29),
(31, 31), (33, 33), (35, 35), (37, 37), (39, 39), (41, 41), (43, 43), (4
5, 45), (47, 47), (49, 49), (51, 51), (53, 53), (55, 55), (57, 57), (59,
59), (61, 61), (63, 63), (65, 65), (67, 67), (69, 69), (71, 71), (73, 7
3), (75, 75), (77, 77), (79, 79), (81, 81), (83, 83), (85, 85), (87, 87),
(89, 89), (91, 91), (93, 93), (95, 95), (97, 97), (99, 99)]]
```

partitionBy() requires dataset to be in key/value (pair) objects referred to as **tuples**.

map(lambda num: (num, num)) - lambda function transforms the dataset's each element into a **key,value pair** commonly referred to as **tuple**.

In this example, since the nums dataset already contained unique values, each value is also now a key to itself. Now the PairRDD contains 100 objects of k,v pair such that it can be unpacked as k, v = kv

partitionBy(2) - Split the dataset into 2 chunks, using default hash partitioner. Elements are now distributed into 2 partitions.

To determine which records go to what partition, Spark uses partition function. The partition to which a particular record is added depends on the partition_key and the number of partitions. In the example above, all even numbered keys were added to the same partition, and the odd numbered keys were added to another partition.

partition = partitionFunc(key) % num_partitions - See the example below.

```
from pyspark.rdd import portable_hash
num_partitions = 2
for el in nums:
    print("Element: [{}]: {} % {} = partition {}".format(
        el, portable_hash(el), num_partitions, portable_hash(el) % num_partitions))
```

```
Element: [0]: 0 % 2 = partition 0
Element: [1]: 1 % 2 = partition 1
Element: [2]: 2 % 2 = partition 0
Element: [3]: 3 % 2 = partition 1
Element: [4]: 4 % 2 = partition 0
Element: [5]: 5 % 2 = partition 1
Element: [6]: 6 % 2 = partition 0
Element: [7]: 7 % 2 = partition 1
Element: [8]: 8 % 2 = partition 0
Element: [9]: 9 % 2 = partition 1
Element: [10]: 10 % 2 = partition 0
Element: [11]: 11 % 2 = partition 1
Element: [12]: 12 % 2 = partition 0
Element: [13]: 13 % 2 = partition 1
Element: [14]: 14 % 2 = partition 0
Element: [15]: 15 % 2 = partition 1
Element: [16]: 16 % 2 = partition 0
Element: [17]: 17 % 2 = partition 1
Element: [18]: 18 % 2 = partition 0
Element: [19]: 19 % 2 = partition 1
Element: [20]: 20 % 2 = partition 0
```

The code above uses hash_function % number of partitions to determine the partition to assign the records to. Since there are only 2 partitions, partition 0 and partition 1, records are assigned to either one based on the hash of the key modulus 2.

Custom Partitioner using hash function

Given a tuple key, function returns an integer - that integer % the number of partitions determines the partition to assign the records to.

```
#  Assuring that data for each state is in one partition
def state_partitioner(state):
    return hash(state)
# Validate results
num_partitions = 5
print(state_partitioner("California") % num_partitions)
print(state_partitioner("Texas") % num_partitions)
print(state_partitioner("New York") % num_partitions)
print(state_partitioner("Florida") % num_partitions)
print(state_partitioner("Illinois") % num_partitions)
```

```
4
0
1
3
0
```

With a custom partitioner, we can see what partition each state is assigned to, ensuring that records of one state are on the same node. This strategy can greatly reduce network traffic, avoiding shuffle of records belonging to the same state. In the absence of this customized hash partitioning, records of one state will be scattered all over the cluster on different nodes/partitions, causing latency during shuffle. **Before performing any joins or aggregation function in Spark, it is extremely important to ensure that records that have a common key are colocated.**

Consider the dataset below with 25 records from 5 states in JSON format.

```
cust2018 = [
{'cust_id' : '001' , 'name': 'Adam', 'age': 40, 'state': 'California', 'miles_driven': 12000, 'insurance_premium': 800},
{'cust_id' : '002' , 'name': 'Bob', 'age': 42, 'state': 'Texas', 'miles_driven': 13000, 'insurance_premium': 900},
{'cust_id' : '003' , 'name': 'Cathy', 'age': 41, 'state': 'New York', 'miles_driven': 10000, 'insurance_premium': 700},
{'cust_id' : '004' , 'name': 'Dave', 'age': 30, 'state': 'Texas', 'miles_driven': 11000, 'insurance_premium': 980},
{'cust_id' : '005' , 'name': 'Ed', 'age': 32, 'state': 'New York', 'miles_driven': 9000, 'insurance_premium': 970},
{'cust_id' : '006' , 'name': 'Frank', 'age': 35, 'state': 'California', 'miles_driven': 10000, 'insurance_premium': 650},
{'cust_id' : '007' , 'name': 'Gary', 'age': 60, 'state': 'California', 'miles_driven': 12000, 'insurance_premium': 680},
{'cust_id' : '008' , 'name': 'Julia', 'age': 55, 'state': 'Florida', 'miles_driven': 11500, 'insurance_premium': 760},
{'cust_id' : '009' , 'name': 'Mary', 'age': 53, 'state': 'Illinois', 'miles_driven': 13500, 'insurance_premium': 790}
{'cust_id' : '0010' , 'name': 'Mike', 'age': 28, 'state': 'Florida', 'miles_driven': 14000, 'insurance_premium': 800},
{'cust_id' : '0011' , 'name': 'Zac', 'age': 29, 'state': 'California', 'miles_driven': 15000, 'insurance_premium': 860},
{'cust_id' : '0012' , 'name': 'Jose', 'age': 32, 'state': 'Texas', 'miles_driven': 13000, 'insurance_premium': 900},
{'cust_id' : '0013' , 'name': 'Luis', 'age': 33, 'state': 'New York', 'miles_driven': 10000, 'insurance_premium': 700},
{'cust_id' : '0014' , 'name': 'Carlos', 'age': 26, 'state': 'Texas', 'miles_driven': 11000, 'insurance_premium': 980},
{'cust_id' : '0015' , 'name': 'Linda', 'age': 28, 'state': 'New York', 'miles_driven': 9000, 'insurance_premium': 970},
{'cust_id' : '0016' , 'name': 'Sophia', 'age': 27, 'state': 'California', 'miles_driven': 10000, 'insurance_premium': 650},
{'cust_id' : '0017' , 'name': 'Maria', 'age': 29, 'state': 'California', 'miles_driven': 12000, 'insurance_premium': 680},
{'cust_id' : '0018' , 'name': 'Sriram', 'age': 37, 'state': 'Florida', 'miles_driven': 11500, 'insurance_premium': 760},
{'cust_id' : '0019' , 'name': 'Ali', 'age': 38, 'state': 'Illinois', 'miles_driven': 13500, 'insurance_premium': 790},
{'cust_id' : '0020' , 'name': 'Usuf', 'age': 26, 'state': 'Califoria', 'miles_driven': 14000, 'insurance_premium': 800}
{'cust_id' : '0021' , 'name': 'Husain', 'age': 38, 'state': 'Wisconsin', 'miles_driven': 12900, 'insurance_premium': 790},
{'cust_id' : '0022' , 'name': 'Raj', 'age': 26, 'state': 'California', 'miles_driven': 14300, 'insurance_premium': 800},
{'cust_id' : '0023' , 'name': 'Laxman', 'age': 38, 'state': 'Texas', 'miles_driven': 11200, 'insurance_premium': 790},
{'cust_id' : '0024' , 'name': 'Sita', 'age': 26, 'state': 'Wisconsin', 'miles_driven': 7800, 'insurance_premium': 800},
{'cust_id' : '0025' , 'name': 'Geeta', 'age': 38, 'state': 'California', 'miles_driven': 7500, 'insurance_premium': 790}
]
```

```python
print(cust2018)
```

```
[{'cust-id': '001', 'name': 'Adam', 'age': 40, 'state': 'California', 'mi
les_driven': 12000, 'insurance_premium': 800}, {'cust_id': '002', 'name':
'Bob', 'age': 42, 'state': 'Texas', 'miles_driven': 13000, 'insurance_pre
mium': 900}, {'cust_id': '003', 'name': 'Cathy', 'age': 41, 'state': 'New
York', 'miles_driven': 10000, 'insurance_premium': 700}, {'cust_id': '00
4', 'name': 'Dave', 'age': 30, 'state': 'Texas', 'miles_driven': 11000,
'insurance_premium': 980}, {'cust_id': '005', 'name': 'Ed', 'age': 32, 's
tate': 'New York', 'miles_driven': 9000, 'insurance_premium': 970}, {'cus
t_id': '006', 'name': 'Frank', 'age': 35, 'state': 'California', 'miles_d
riven': 10000, 'insurance_premium': 650}, {'cust_id': '007', 'name': 'Gar
y', 'age': 60, 'state': 'California', 'miles_driven': 12000, 'insurance_p
remium': 680}, {'cust_id': '008', 'name': 'Julia', 'age': 55, 'state': 'F
lorida', 'miles_driven': 11500, 'insurance_premium': 760}, {'cust_id': '0
09', 'name': 'Mary', 'age': 53, 'state': 'Illinois', 'miles_driven': 1350
0, 'insurance_premium': 790}, {'cust_id': '0010', 'name': 'Mike', 'age':
28, 'state': 'Florida', 'miles_driven': 14000, 'insurance_premium': 800},
{'cust_id': '0011', 'name': 'Zac', 'age': 29, 'state': 'California', 'mil
es_driven': 15000, 'insurance_premium': 860}, {'cust_id': '0012', 'name':
'Jose', 'age': 32, 'state': 'Texas', 'miles_driven': 13000, 'insurance_pr
emium': 900}, {'cust_id': '0013', 'name': 'Luis', 'age': 33, 'state': 'Ne
w York', 'miles_driven': 10000, 'insurance_premium': 700}, {'cust_id': '0
014', 'name': 'Carlos', 'age': 26, 'state': 'Texas', 'miles_driven': 1100
0, 'insurance_premium': 980}, {'cust_id': '0015', 'name': 'Linda', 'age':
28, 'state': 'New York', 'miles_driven': 9000, 'insurance_premium': 970},
{'cust_id': '0016', 'name': 'Sophia', 'age': 27, 'state': 'California',
'miles_driven': 10000, 'insurance_premium': 650}, {'cust_id': '0017', 'na
me': 'Maria', 'age': 29, 'state': 'California', 'miles_driven': 12000, 'i
nsurance_premium': 680}, {'cust_id': '0018', 'name': 'Sriram', 'age': 37,
'state': 'Florida', 'miles_driven': 11500, 'insurance_premium': 760}, {'c
ust_id': '0019', 'name': 'Ali', 'age': 38, 'state': 'Illinois', 'miles_dr
iven': 13500, 'insurance_premium': 790}, {'cust_id': '0020', 'name': 'Usu
f', 'age': 26, 'state': 'Califoria', 'miles_driven': 14000, 'insurance_pr
emium': 800}, {'cust_id': '0021', 'name': 'Husain', 'age': 38, 'state':
'Wisconsin', 'miles_driven': 12900, 'insurance_premium': 790}, {'cust_i
d': '0022', 'name': 'Raj', 'age': 26, 'state': 'California', 'miles_drive
n': 14300, 'insurance_premium': 800}, {'cust_id': '0023', 'name': 'Laxma
n', 'age': 38, 'state': 'Texas', 'miles_driven': 11200, 'insurance_premiu
m': 790}, {'cust_id': '0024', 'name': 'Sita', 'age': 26, 'state': 'Wiscon
sin', 'miles_driven': 7800, 'insurance_premium': 800}, {'cust_id': '002
5', 'name': 'Geeta', 'age': 38, 'state': 'California', 'miles_driven': 75
00, 'insurance_premium': 790}]
```

Now, parallelize and distribute cust2018 dataset such that records of one state are on the same partition, using the custom state_partitioner.

```
custRDD = sc.parallelize(cust2018) \
        .map(lambda cust: (cust['state'], cust)) \
        .partitionBy(5, state_partitioner)

print("Number of partitions: {}".format(custRDD.getNumPartitions()))
print("Partitioner: {}".format(custRDD.partitioner))
print("Partitions structure: {}".format(custRDD.glom().collect()))
```

```
Number of partitions: 5
Partitioner: <pyspark.rdd.Partitioner object at 0x108fd7438>
Partitions structure: [[('Texas', {'cust_id': '002', 'name': 'Bob', 'ag
e': 42, 'state': 'Texas', 'miles_driven': 13000, 'insurance_premium': 90
0}), ('Texas', {'cust_id': '004', 'name': 'Dave', 'age': 30, 'state': 'Te
xas', 'miles_driven': 11000, 'insurance_premium': 980}), ('Illinois', {'c
ust_id': '009', 'name': 'Mary', 'age': 53, 'state': 'Illinois', 'miles_dr
iven': 13500, 'insurance_premium': 790}), ('Texas', {'cust_id': '0012',
'name': 'Jose', 'age': 32, 'state': 'Texas', 'miles_driven': 13000, 'insu
rance_premium': 900}), ('Texas', {'cust_id': '0014', 'name': 'Carlos', 'a
ge': 26, 'state': 'Texas', 'miles_driven': 11000, 'insurance_premium': 98
0}), ('Illinois', {'cust_id': '0019', 'name': 'Ali', 'age': 38, 'state':
'Illinois', 'miles_driven': 13500, 'insurance_premium': 790}), ('Texas',
{'cust_id': '0023', 'name': 'Laxman', 'age': 38, 'state': 'Texas', 'miles
_driven': 11200, 'insurance_premium': 790})], [('New York', {'cust_id':
'003', 'name': 'Cathy', 'age': 41, 'state': 'New York', 'miles_driven': 1
0000, 'insurance_premium': 700}), ('New York', {'cust_id': '005', 'name':
'Ed', 'age': 32, 'state': 'New York', 'miles_driven': 9000, 'insurance_pr
emium': 970}), ('New York', {'cust_id': '0013', 'name': 'Luis', 'age': 3
3, 'state': 'New York', 'miles_driven': 10000, 'insurance_premium': 70
0}), ('New York', {'cust_id': '0015', 'name': 'Linda', 'age': 28, 'stat
e': 'New York', 'miles_driven': 9000, 'insurance_premium': 970})], [('Cal
iforia', {'cust_id': '0020', 'name': 'Usuf', 'age': 26, 'state': 'Califor
ia', 'miles_driven': 14000, 'insurance_premium': 800})], [('Florida', {'c
ust_id': '008', 'name': 'Julia', 'age': 55, 'state': 'Florida', 'miles_dr
iven': 11500, 'insurance_premium': 760}), ('Florida', {'cust_id': '0010',
'name': 'Mike', 'age': 28, 'state': 'Florida', 'miles_driven': 14000, 'in
surance_premium': 800}), ('Florida', {'cust_id': '0018', 'name': 'Srira
m', 'age': 37, 'state': 'Florida', 'miles_driven': 11500, 'insurance_prem
ium': 760}), ('Wisconsin', {'cust_id': '0021', 'name': 'Husain', 'age': 3
8, 'state': 'Wisconsin', 'miles_driven': 12900, 'insurance_premium': 79
0}), ('Wisconsin', {'cust_id': '0024', 'name': 'Sita', 'age': 26, 'stat
e': 'Wisconsin', 'miles_driven': 7800, 'insurance_premium': 800})], [('Ca
lifornia', {'cust-id': '001', 'name': 'Adam', 'age': 40, 'state': 'Califo
rnia', 'miles_driven': 12000, 'insurance_premium': 800}), ('California',
{'cust_id': '006', 'name': 'Frank', 'age': 35, 'state': 'California', 'mi
les_driven': 10000, 'insurance_premium': 650}), ('California', {'cust_i
d': '007', 'name': 'Gary', 'age': 60, 'state': 'California', 'miles_drive
n': 12000, 'insurance_premium': 680}), ('California', {'cust_id': '0011',
'name': 'Zac', 'age': 29, 'state': 'California', 'miles_driven': 15000,
'insurance_premium': 860}), ('California', {'cust_id': '0016', 'name': 'S
ophia', 'age': 27, 'state': 'California', 'miles_driven': 10000, 'insuran
ce_premium': 650}), ('California', {'cust_id': '0017', 'name': 'Maria',
'age': 29, 'state': 'California', 'miles_driven': 12000, 'insurance_premi
um': 680}), ('California', {'cust_id': '0022', 'name': 'Raj', 'age': 26,
'state': 'California', 'miles_driven': 14300, 'insurance_premium': 800}),
('California', {'cust_id': '0025', 'name': 'Geeta', 'age': 38, 'state':
'California', 'miles_driven': 7500, 'insurance_premium': 790})]]
```

Now that all records from the state are in the same partition, it is much easier for a task to work directly on those partitions without worrying about shuffling.

mapPartitions()

Converts each partition of the source dataset into multiple elements of the result (possibly none). mapPatitions() allows **heavyweight initialization** to be done once for many elements in each partition, instead of running map() for each element in the dataset.

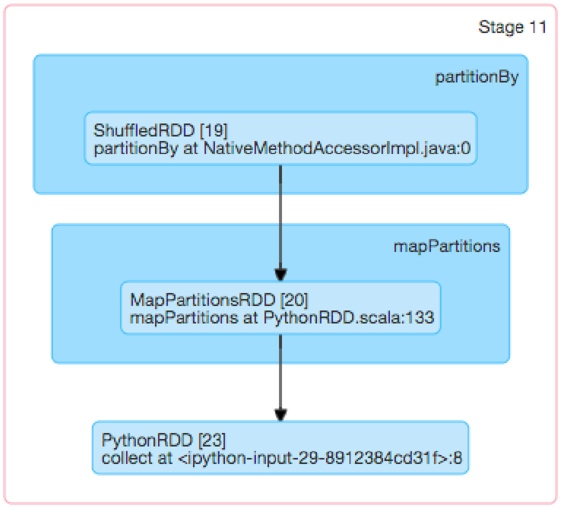
As an example, to calculate the sum of insurance premiums in each state, mapPartitions() API can be efficiently used without any shuffling of data between partitions.

```python
# Function for calculating sum of insurance_premium for each partition
# Notice that we are getting an iterator. All work is done on one node
def sum_premiums(iterator):
    yield sum(customer[1]['insurance_premium'] for customer in iterator)

# Sum sales in each partition
sum_amounts = custRDD \
    .mapPartitions(sum_premiums) \
    .collect()

print("Total Insurance_Premiums for each partition: {}".format(sum_amounts
```

```
Total Insurance_Premiums for each partition: [6130, 3340, 800, 3910, 591
0]
```

If map() was used instead of mapPartitions() in the code snippet above, map() function will be called 25 times, once for every row in the dataset. **mapPartitions() is much faster than map()** because, it operates on many rows in the same partition, thus the function is invoked only once per partition for **heavyweight initialization**. You can **achieve 300x performance increase using mapPartitions()** in group-by and aggregate functions over map(). In the web-ui at port 4040, notice minimal shuffle read in mapPartitions() operation and no garbage collection. There are 5 tasks, because we partitioned the custRDD by 5 on state key.

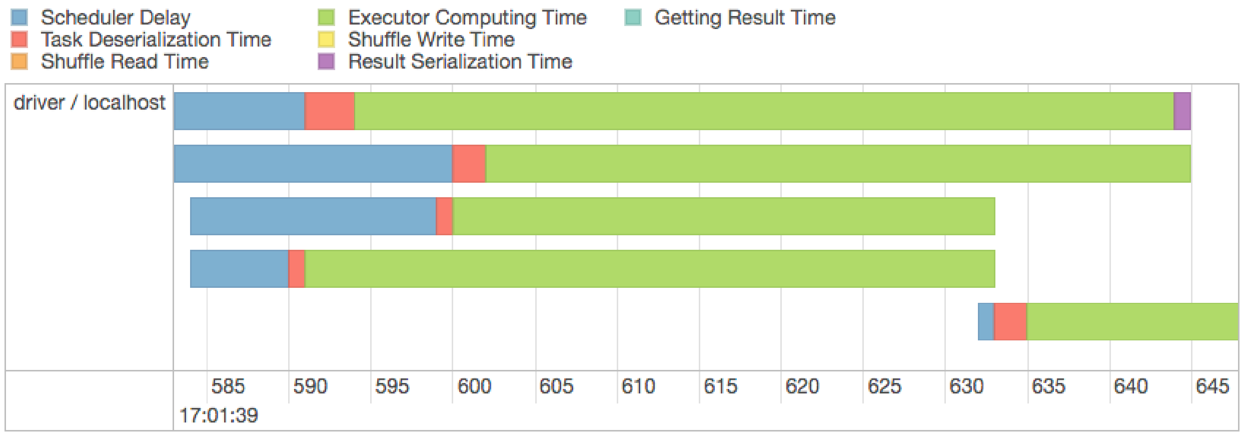
▾ DAG Visualization

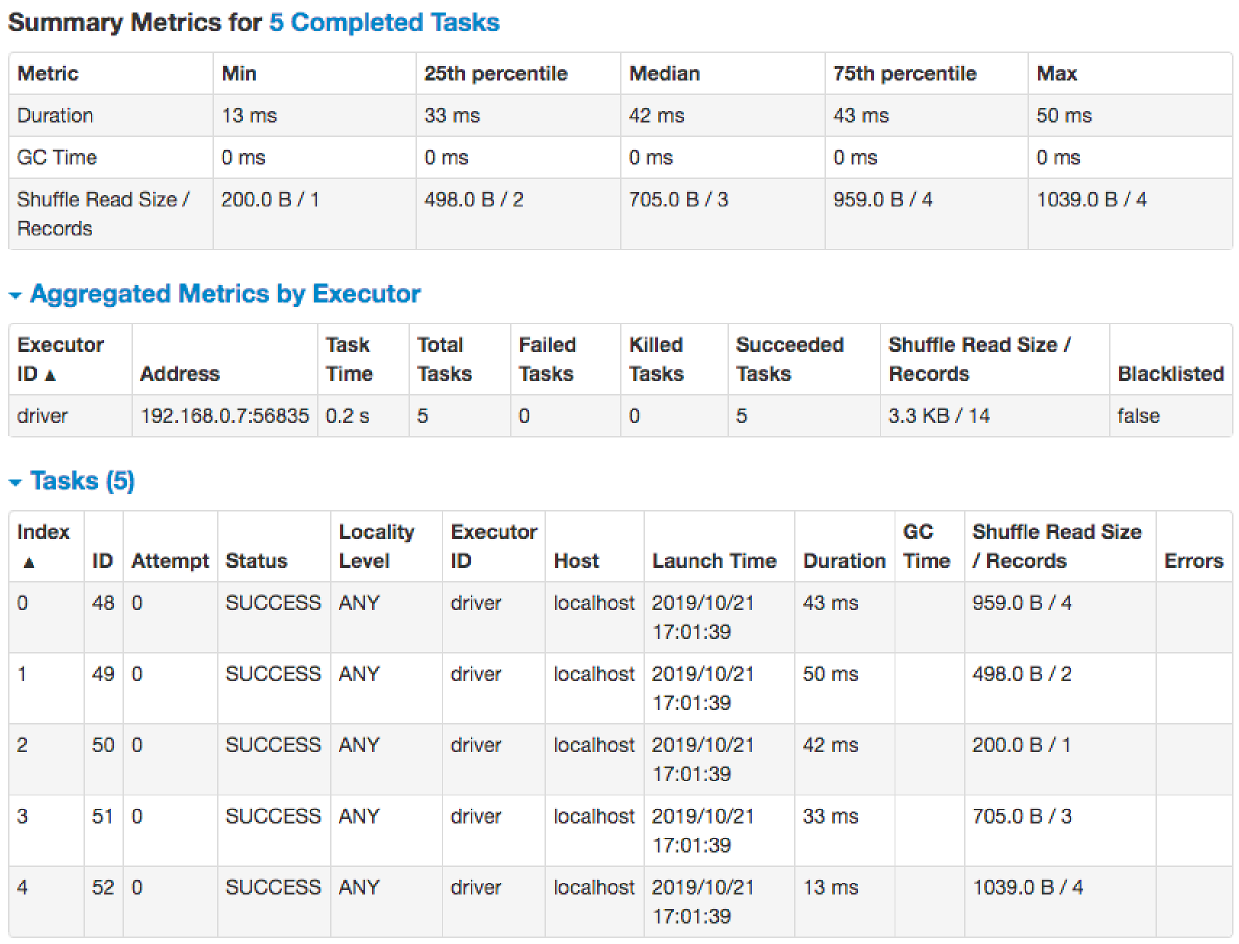

Stage 11
partitionBy
ShuffledRDD [19]
partitionBy at NativeMethodAccessorImpl.java:0
mapPartitions
MapPartitionsRDD [20]
mapPartitions at PythonRDD.scala:133
PythonRDD [23]
collect at <ipython-input-29-8912384cd31f>:8

▸ Show Additional Metrics

▾ Event Timeline

☐ Enable zooming





Summary Metrics for 5 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	13 ms	33 ms	42 ms	43 ms	50 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Size / Records	200.0 B / 1	498.0 B / 2	705.0 B / 3	959.0 B / 4	1039.0 B / 4

Aggregated Metrics by Executor

Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Shuffle Read Size / Records	Blacklisted
driver	192.168.0.7:56835	0.2 s	5	0	0	5	3.3 KB / 14	false

Tasks (5)

Index ▲	ID	Attempt	Status	Locality Level	Executor ID	Host	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Errors
0	48	0	SUCCESS	ANY	driver	localhost	2019/10/21 17:01:39	43 ms		959.0 B / 4	
1	49	0	SUCCESS	ANY	driver	localhost	2019/10/21 17:01:39	50 ms		498.0 B / 2	
2	50	0	SUCCESS	ANY	driver	localhost	2019/10/21 17:01:39	42 ms		200.0 B / 1	
3	51	0	SUCCESS	ANY	driver	localhost	2019/10/21 17:01:39	33 ms		705.0 B / 3	
4	52	0	SUCCESS	ANY	driver	localhost	2019/10/21 17:01:39	13 ms		1039.0 B / 4	

DataFrames and Datasets

Unlike RDD, **DataFrames** organize data into named columns, like a table in a relational database. Organizing structured or semi-structured data into named columns allows Spark to infer the schema of the DataFrame. Like RDD, DataFrame is also immutable, lazily evaluated and distributed collection of data. **Dataset -** available only in Scala and Java - are an extension of DataFrame API which provides **type-safe**, object-oriented programming interface. Dataset takes advantage of Spark's **Catalyst Optimizer** by exposing expressions and data fields to a **query planner**. Use Datasets wherever possible. In case of Python, use DataFrames API instead of RDD. It is very easy to create custom partitioners with DataFrames and Datasets.

repartition() - custom partitioner in DataFrame

Code below shows how to use **repartition() on a DataFrame to control parallelism** in Spark.

```python
from pyspark.sql import SparkSession, Row

spark = SparkSession(sc)

#set partitions to 5
spark.conf.set("spark.sql.shuffle.partitions", 5)

rdd = sc.parallelize(cust2018) \
        .map(lambda cust: Row(**cust))

df = spark.createDataFrame(rdd)

print("Number of partitions: {}".format(df.rdd.getNumPartitions()))
print("Partitioner: {}".format(rdd.partitioner))
print("Partitions structure: {}".format(df.rdd.glom().collect()))

# Repartition by column
df2 = df.repartition("state")

print("\nAfter 'repartition()'")
print("Number of partitions: {}".format(df2.rdd.getNumPartitions()))
print("Partitioner: {}".format(df2.rdd.partitioner))
print("Partitions structure: {}".format(df2.rdd.glom().collect()))
```

```
Number of partitions: 4
Partitioner: None
Partitions structure: [[Row(age=40, cust-id='001', insurance_premium=800, miles_driven=12000, name='Adam', state='Cal
ifornia'), Row(age=42, cust-id='002', insurance_premium=900, miles_driven=13000, name='Bob', state='Texas'), Row(age=
41, cust-id='003', insurance_premium=700, miles_driven=10000, name='Cathy', state='New York'), Row(age=30, cust-id='0
04', insurance_premium=980, miles_driven=11000, name='Dave', state='Texas'), Row(age=32, cust-id='005', insurance_pre
mium=970, miles_driven=9000, name='Ed', state='New York'), Row(age=35, cust-id='006', insurance_premium=650, miles_dr
iven=10000, name='Frank', state='California')], [Row(age=60, cust-id='007', insurance_premium=680, miles_driven=1200
0, name='Gary', state='California'), Row(age=55, cust-id='008', insurance_premium=760, miles_driven=11500, name='Juli
a', state='Florida'), Row(age=53, cust-id='009', insurance_premium=790, miles_driven=13500, name='Mary', state='Illin
ois'), Row(age=28, cust-id='0010', insurance_premium=800, miles_driven=14000, name='Mike', state='Florida'), Row(age=
29, cust-id='0011', insurance_premium=860, miles_driven=15000, name='Zac', state='California'), Row(age=32, cust-id
='0012', insurance_premium=900, miles_driven=13000, name='Jose', state='Texas')], [Row(age=33, cust-id='0013', insura
nce_premium=700, miles_driven=10000, name='Luis', state='New York'), Row(age=26, cust-id='0014', insurance_premium=98
0, miles_driven=11000, name='Carlos', state='Texas'), Row(age=28, cust-id='0015', insurance_premium=970, miles_driven
=9000, name='Linda', state='New York'), Row(age=27, cust-id='0016', insurance_premium=650, miles_driven=10000, name
='Sophia', state='California'), Row(age=29, cust-id='0017', insurance_premium=680, miles_driven=12000, name='Maria',
state='California'), Row(age=37, cust-id='0018', insurance_premium=760, miles_driven=11500, name='Sriram', state='Flo
rida')], [Row(age=38, cust-id='0019', insurance_premium=790, miles_driven=13500, name='Ali', state='Illinois'), Row(a
ge=26, cust-id='0020', insurance_premium=800, miles_driven=14000, name='Usuf', state='Califoria'), Row(age=38, cust-i
d='0021', insurance_premium=790, miles_driven=12900, name='Husain', state='Wisconsin'), Row(age=26, cust-id='0022', i
nsurance_premium=800, miles_driven=14300, name='Raj', state='California'), Row(age=38, cust-id='0023', insurance_prem
ium=790, miles_driven=11200, name='Laxman', state='Texas'), Row(age=26, cust-id='0024', insurance_premium=800, miles_
driven=7800, name='Sita', state='Wisconsin'), Row(age=38, cust-id='0025', insurance_premium=790, miles_driven=7500, n
ame='Geeta', state='California')]]


After 'repartition()'
Number of partitions: 5
Partitioner: None
Partitions structure: [[Row(age=55, cust-id='008', insurance_premium=760, miles_driven=11500, name='Julia', state='Fl
orida'), Row(age=28, cust-id='0010', insurance_premium=800, miles_driven=14000, name='Mike', state='Florida'), Row(ag
e=37, cust-id='0018', insurance_premium=760, miles_driven=11500, name='Sriram', state='Florida'), Row(age=38, cust-id
='0021', insurance_premium=790, miles_driven=12900, name='Husain', state='Wisconsin'), Row(age=26, cust-id='0024', in
surance_premium=800, miles_driven=7800, name='Sita', state='Wisconsin')], [Row(age=42, cust-id='002', insurance_premi
um=900, miles_driven=13000, name='Bob', state='Texas'), Row(age=30, cust-id='004', insurance_premium=980, miles_drive
n=11000, name='Dave', state='Texas'), Row(age=53, cust-id='009', insurance_premium=790, miles_driven=13500, name='Mar
y', state='Illinois'), Row(age=32, cust-id='0012', insurance_premium=900, miles_driven=13000, name='Jose', state='Tex
as'), Row(age=26, cust-id='0014', insurance_premium=980, miles_driven=11000, name='Carlos', state='Texas'), Row(age=3
8, cust-id='0019', insurance_premium=790, miles_driven=13500, name='Ali', state='Illinois'), Row(age=26, cust-id='002
0', insurance_premium=800, miles_driven=14000, name='Usuf', state='Califoria'), Row(age=38, cust-id='0023', insurance
_premium=790, miles_driven=11200, name='Laxman', state='Texas')], [Row(age=41, cust-id='003', insurance_premium=700,
miles_driven=10000, name='Cathy', state='New York'), Row(age=32, cust-id='005', insurance_premium=970, miles_driven=9
000, name='Ed', state='New York'), Row(age=33, cust-id='0013', insurance_premium=700, miles_driven=10000, name='Lui
s', state='New York'), Row(age=28, cust-id='0015', insurance_premium=970, miles_driven=9000, name='Linda', state='New
York')], [Row(age=40, cust-id='001', insurance_premium=800, miles_driven=12000, name='Adam', state='California'), Row
(age=35, cust-id='006', insurance_premium=650, miles_driven=10000, name='Frank', state='California'), Row(age=60, cus
t-id='007', insurance_premium=680, miles_driven=12000, name='Gary', state='California'), Row(age=29, cust-id='0011',
insurance_premium=860, miles_driven=15000, name='Zac', state='California'), Row(age=27, cust-id='0016', insurance_pre
mium=650, miles_driven=10000, name='Sophia', state='California'), Row(age=29, cust-id='0017', insurance_premium=680,
miles_driven=12000, name='Maria', state='California'), Row(age=26, cust-id='0022', insurance_premium=800, miles_drive
n=14300, name='Raj', state='California'), Row(age=38, cust-id='0025', insurance_premium=790, miles_driven=7500, name
='Geeta', state='California')], []]
```

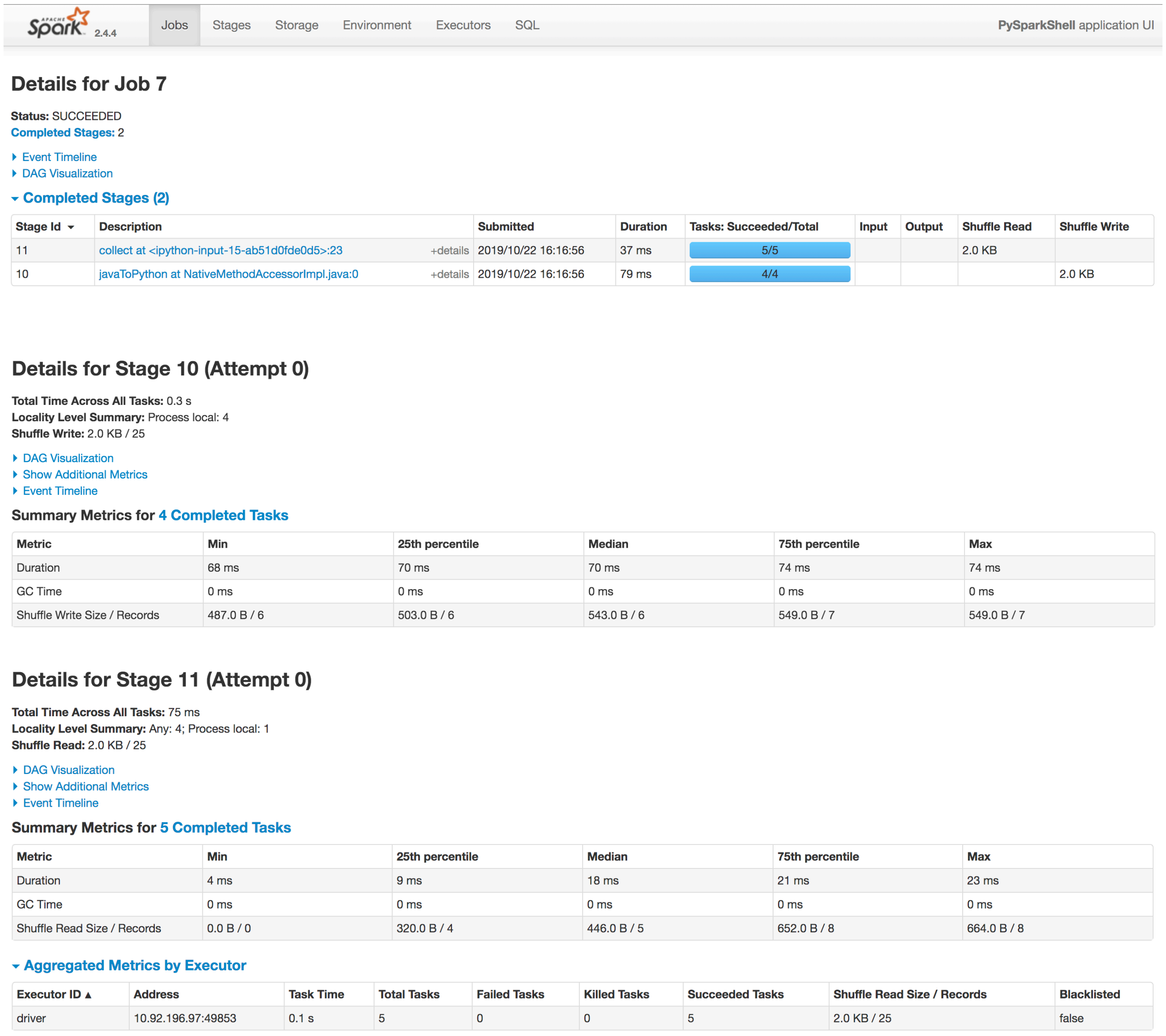
# You can also repartition with numPartitions and column_name as below.

df 2 = df.repartition(5,"state")

When you first create the DataFrame, it partitions the dataset in to 4 partitions as the available vCores are 4 in the cluster.

When the **repartition()** API is invoked with column name, it repartitions the dataset into 5 partitions - as per the parameter set in **spark.sql.shuffle.partitions.** It is much simpler to repartition the DataFrame with column names and also control the number of partitions to create for the HashPartitioner. **Default value for spark.sql.shuffle.partitions is 200**, which means when a **full shuffle** forces a new Stage, there will be 200 partitions, and thereby 200 tasks in that Stage. When you are dealing with millions or billions of records and GB or PB scale dataset, **partitioning the dataset on appropriate column(s) and the number of partitions is critically important for optimal performance.**

repartition() performs **full shuffle**, as data must be shuffled across nodes. Spark handles **full shuffle** in a new **Stage** - comprising of tasks equal to the number of partitions. When you first create the DataFrame there are 4 partitions of the dataset. As soon as you repartition the DataFrame to 5 - it forces a new Stage - now comprising of 5 tasks. This way, **you can increase or decrease parallelism in Spark by controlling the number of partitions of your dataset**. Also note, repartition() took half the time (37 ms) of original DataFrame creation (79 ms).



Details for Job 7

Status: SUCCEEDED
Completed Stages: 2

Event Timeline
DAG Visualization

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
11	collect at <ipython-input-15-ab51d0fde0d5>:23 +details	2019/10/22 16:16:56	37 ms	5/5			2.0 KB	
10	javaToPython at NativeMethodAccessorImpl.java:0 +details	2019/10/22 16:16:56	79 ms	4/4				2.0 KB

Details for Stage 10 (Attempt 0)

Total Time Across All Tasks: 0.3 s
Locality Level Summary: Process local: 4
Shuffle Write: 2.0 KB / 25

DAG Visualization
Show Additional Metrics
Event Timeline

Summary Metrics for 4 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	68 ms	70 ms	70 ms	74 ms	74 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Write Size / Records	487.0 B / 6	503.0 B / 6	543.0 B / 6	549.0 B / 7	549.0 B / 7

Details for Stage 11 (Attempt 0)

Total Time Across All Tasks: 75 ms
Locality Level Summary: Any: 4; Process local: 1
Shuffle Read: 2.0 KB / 25

DAG Visualization
Show Additional Metrics
Event Timeline

Summary Metrics for 5 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	4 ms	9 ms	18 ms	21 ms	23 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Size / Records	0.0 B / 0	320.0 B / 4	446.0 B / 5	652.0 B / 8	664.0 B / 8

Aggregated Metrics by Executor

Executor ID ▴	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Shuffle Read Size / Records	Blacklisted
driver	10.92.196.97:49853	0.1 s	5	0	0	5	2.0 KB / 25	false

Benefit of partitioning in shuffle operations

All operations performing data shuffle by key inherently benefit from partitioning. Examples are groupByKey(), groupWith(), cogroup(), reduceByKey(), combineByKey(), join(), leftOuterJoin(), rightOuterJoin(), lookup(). These operations cannot modify the key of the element in the record, so they can retain the parent DataFrame's Partitioner.

Use mapValues() instead of map()

map() transformation can possibly modify the key of each element in the record as shown in the code above, Spark cannot guarantee to produce known partitioning, so map() operation cannot retain the parent DataFrame or RDD's Partitioner.

mapValues(), flatMapValues() and filter() operations guarantee that each tuple's key remains the same and may benefit from the parent DataFrame's Partitioner.

The Code below shows transformation of the parent RDD using map() which forgoes the parent's Partitioner and the same transformation using mapValues() which retains the parent RDD's partitioner.

```
rdd = sc.parallelize(nums) \
        .map(lambda num: (num, num)) \
        .partitionBy(2) \
        .persist()

print("Number of partitions: {}".format(rdd.getNumPartitions()))
print("Partitioner: {}".format(rdd.partitioner))
print("Partitions structure: {}".format(rdd.glom().collect()))

# Transform with map() @ the cost the losing the parent's Partitioner
rdd2 = rdd.map(lambda num: (num[0],num[0]*2))

print("Number of partitions: {}".format(rdd2.getNumPartitions()))
print("Partitioner: {}".format(rdd2.partitioner))  # We have lost the partitioner
print("Partitions structure: {}".format(rdd2.glom().collect()))
```

```
Number of partitions: 2
Partitioner: <pyspark.rdd.Partitioner object at 0x11157dac8>
Partitions structure: [[(0, 0), (2, 2), (4, 4), (6, 6), (8, 8), (10, 10), (12, 12), (14, 14), (16, 16), (18, 18), (2
0, 20), (22, 22), (24, 24), (26, 26), (28, 28), (30, 30), (32, 32), (34, 34), (36, 36), (38, 38), (40, 40), (42, 42),
(44, 44), (46, 46), (48, 48), (50, 50), (52, 52), (54, 54), (56, 56), (58, 58), (60, 60), (62, 62), (64, 64), (66, 6
6), (68, 68), (70, 70), (72, 72), (74, 74), (76, 76), (78, 78), (80, 80), (82, 82), (84, 84), (86, 86), (88, 88), (9
0, 90), (92, 92), (94, 94), (96, 96), (98, 98)], [(1, 1), (3, 3), (5, 5), (7, 7), (9, 9), (11, 11), (13, 13), (15, 1
5), (17, 17), (19, 19), (21, 21), (23, 23), (25, 25), (27, 27), (29, 29), (31, 31), (33, 33), (35, 35), (37, 37), (3
9, 39), (41, 41), (43, 43), (45, 45), (47, 47), (49, 49), (51, 51), (53, 53), (55, 55), (57, 57), (59, 59), (61, 61),
(63, 63), (65, 65), (67, 67), (69, 69), (71, 71), (73, 73), (75, 75), (77, 77), (79, 79), (81, 81), (83, 83), (85, 8
5), (87, 87), (89, 89), (91, 91), (93, 93), (95, 95), (97, 97), (99, 99)]]
Number of partitions: 2
Partitioner: None
Partitions structure: [[(0, 0), (2, 4), (4, 8), (6, 12), (8, 16), (10, 20), (12, 24), (14, 28), (16, 32), (18, 36),
(20, 40), (22, 44), (24, 48), (26, 52), (28, 56), (30, 60), (32, 64), (34, 68), (36, 72), (38, 76), (40, 80), (42, 8
4), (44, 88), (46, 92), (48, 96), (50, 100), (52, 104), (54, 108), (56, 112), (58, 116), (60, 120), (62, 124), (64, 1
28), (66, 132), (68, 136), (70, 140), (72, 144), (74, 148), (76, 152), (78, 156), (80, 160), (82, 164), (84, 168), (8
6, 172), (88, 176), (90, 180), (92, 184), (94, 188), (96, 192), (98, 196)], [(1, 2), (3, 6), (5, 10), (7, 14), (9, 1
8), (11, 22), (13, 26), (15, 30), (17, 34), (19, 38), (21, 42), (23, 46), (25, 50), (27, 54), (29, 58), (31, 62), (3
3, 66), (35, 70), (37, 74), (39, 78), (41, 82), (43, 86), (45, 90), (47, 94), (49, 98), (51, 102), (53, 106), (55, 11
0), (57, 114), (59, 118), (61, 122), (63, 126), (65, 130), (67, 134), (69, 138), (71, 142), (73, 146), (75, 150), (7
7, 154), (79, 158), (81, 162), (83, 166), (85, 170), (87, 174), (89, 178), (91, 182), (93, 186), (95, 190), (97, 19
4), (99, 198)]]
```

```
rdd = sc.parallelize(nums) \
        .map(lambda num: (num, num)) \
        .partitionBy(2) \
        .persist()

print("Number of partitions: {}".format(rdd.getNumPartitions()))
print("Partitioner: {}".format(rdd.partitioner))
print("Partitions structure: {}".format(rdd.glom().collect()))

# Transform with mapValues() instead of map() such that only values are modified retaining the key. Since mapValues()
#cannot operate on the key of the tuple, Spark can retain and benefit from the parent's Partitioner.
rdd2 = rdd.mapValues(lambda x: x*2)

print("Number of partitions: {}".format(rdd2.getNumPartitions()))
print("Partitioner: {}".format(rdd2.partitioner))  # We still got the parent's partitioner
print("Partitions structure: {}".format(rdd2.glom().collect()))
```

```
Number of partitions: 2
Partitioner: <pyspark.rdd.Partitioner object at 0x111571dd8>
Partitions structure: [[(0, 0), (2, 2), (4, 4), (6, 6), (8, 8), (10, 10), (12, 12), (14, 14), (16, 16), (18, 18), (2
0, 20), (22, 22), (24, 24), (26, 26), (28, 28), (30, 30), (32, 32), (34, 34), (36, 36), (38, 38), (40, 40), (42, 42),
(44, 44), (46, 46), (48, 48), (50, 50), (52, 52), (54, 54), (56, 56), (58, 58), (60, 60), (62, 62), (64, 64), (66, 6
6), (68, 68), (70, 70), (72, 72), (74, 74), (76, 76), (78, 78), (80, 80), (82, 82), (84, 84), (86, 86), (88, 88), (9
0, 90), (92, 92), (94, 94), (96, 96), (98, 98)], [(1, 1), (3, 3), (5, 5), (7, 7), (9, 9), (11, 11), (13, 13), (15, 1
5), (17, 17), (19, 19), (21, 21), (23, 23), (25, 25), (27, 27), (29, 29), (31, 31), (33, 33), (35, 35), (37, 37), (3
9, 39), (41, 41), (43, 43), (45, 45), (47, 47), (49, 49), (51, 51), (53, 53), (55, 55), (57, 57), (59, 59), (61, 61),
(63, 63), (65, 65), (67, 67), (69, 69), (71, 71), (73, 73), (75, 75), (77, 77), (79, 79), (81, 81), (83, 83), (85, 8
5), (87, 87), (89, 89), (91, 91), (93, 93), (95, 95), (97, 97), (99, 99)]]
Number of partitions: 2
Partitioner: <pyspark.rdd.Partitioner object at 0x111571dd8>
Partitions structure: [[(0, 0), (2, 4), (4, 8), (6, 12), (8, 16), (10, 20), (12, 24), (14, 28), (16, 32), (18, 36),
(20, 40), (22, 44), (24, 48), (26, 52), (28, 56), (30, 60), (32, 64), (34, 68), (36, 72), (38, 76), (40, 80), (42, 8
4), (44, 88), (46, 92), (48, 96), (50, 100), (52, 104), (54, 108), (56, 112), (58, 116), (60, 120), (62, 124), (64, 1
28), (66, 132), (68, 136), (70, 140), (72, 144), (74, 148), (76, 152), (78, 156), (80, 160), (82, 164), (84, 168), (8
6, 172), (88, 176), (90, 180), (92, 184), (94, 188), (96, 192), (98, 196)], [(1, 2), (3, 6), (5, 10), (7, 14), (9, 1
8), (11, 22), (13, 26), (15, 30), (17, 34), (19, 38), (21, 42), (23, 46), (25, 50), (27, 54), (29, 58), (31, 62), (3
3, 66), (35, 70), (37, 74), (39, 78), (41, 82), (43, 86), (45, 90), (47, 94), (49, 98), (51, 102), (53, 106), (55, 11
0), (57, 114), (59, 118), (61, 122), (63, 126), (65, 130), (67, 134), (69, 138), (71, 142), (73, 146), (75, 150), (7
7, 154), (79, 158), (81, 162), (83, 166), (85, 170), (87, 174), (89, 178), (91, 182), (93, 186), (95, 190), (97, 19
4), (99, 198)]]
```

Data Format

RDD - Can process structured and un-structured data, but cannot infer schema of ingested data. Developer needs to specify the schema.

DataFrame - Can process structure and semi-structured data. Organizes data into named columns. Allows Spark to infer schema.

Optimization

RDD - No built-in optimization engine is available on RDDs in Spark. **RDDs cannot take advantage of Spark's Catalyst Optimizer**.

DataFrame - **Built-in optimization using Catalyst Optimizer**. DataFrames use catalyst transformation framework in 4 phases: 1) analyzing a logical plan to resolve references, 2) Logical plan optimization, 3) physical plan, 4) code generation

Aggregation

RDD - Slower to perform simple grouping and aggregation operations on RDD.

DataFrame - Easy to use and faster on large data sets.

coalesce()

Decreases the number of partitions minimizing shuffles.

Code to Follow:

Data Skew

If some partitions have a lot more records than the other partition, the task will take longer to process the partition with more data, possibly leading to data skew that results in sub-optimal use of the resources. It is important to set a limit for the number of records in every partition such that data is evenly distributed among partitions to avoid data skew. **Data Skew is a data problem and needs to be resolved at the data level**. The root cause of data skew is uneven distribution of the underlying data on heavily skewed keys. During join and group-by key operations, null values in join keys or group-by keys also result into data skew.

Identifying Data Skew

You observe all but few tasks in a Stage finishing within a reasonable time-frame. Often only 1 task takes forever to finish or never finishes.

Resolve Data Skew

If join operation is done on a skewed dataset, a quick trick to prevent data skew is to **increase the 'spark.sql.autoBroadcastJoinThreshold' value**, so the smaller table on the right side of the join is broadcast to all the nodes in the cluster. Before enabling autoBroadcastJoinThreshold, ensure there is sufficient driver and executor memory.

If there are too many null values in the dataset, **pre-process those null values with some random ids**, and handle those random ids in the application code, after the join.

Use **salting** on the join key to redistribute data in an even manner, so that that the task processing on that partition completes in a reasonable time-frame.

Salting

Is a technique where random values are added on join key of the left table. On the right table, rows are replicated to match the random keys, in such a way that if leftTable.key1 == rightTable.key1, then leftTable.key1_<salt> == leftTable.key1_<salt>.

Code Example: To be added

---

raw-code: Below is the raw code from PySpark's ipython notebook.

```
from pyspark import SparkContext
nums = range(0, 100)
print(nums)

rdd = sc.parallelize(nums)

print("Number of partitions: {}".format(rdd.getNumPartitions()))
print("Partitioner: {}".format(rdd.partitioner))
print("Partitions structure: {}".format(rdd.glom().collect()))



rdd = sc.parallelize(nums, 15)

print("Number of partitions: {}".format(rdd.getNumPartitions()))
print("Partitioner: {}".format(rdd.partitioner))
print("Partitions structure: {}".format(rdd.glom().collect()))


rdd = sc.parallelize(nums) \
        .map(lambda num: (num, num)) \
        .partitionBy(2) \
        .persist()

print("Number of partitions: {}".format(rdd.getNumPartitions()))
print("Partitioner: {}".format(rdd.partitioner))
print("Partitions structure: {}".format(rdd.glom().collect()))

from pyspark.rdd import portable_hash
num_partitions = 2
for el in nums:
    print("Element: [{}]: {} % {} = partition {}".format(
        el, portable_hash(el), num_partitions, portable_hash(el) % num_partitions))
```

```
#  Assuring that data for each state is in one partition
def state_partitioner(state):
    return hash(state)
# Validate results
num_partitions = 5
print(state_partitioner("California") % num_partitions)
print(state_partitioner("Texas") % num_partitions)
print(state_partitioner("New York") % num_partitions)
print(state_partitioner("Florida") % num_partitions)
print(state_partitioner("Illinois") % num_partitions)


cust2018 = [
    {'cust-id': '001','name': 'Adam', 'age': 40, 'state': 'California', 'miles_driven':12000, 'insurance_premium':800},
    {'cust_id' : '002' , 'name': 'Bob', 'age': 42, 'state': 'Texas', 'miles_driven': 13000, 'insurance_premium': 900},
    {'cust_id' : '003' , 'name': 'Cathy', 'age': 41, 'state': 'New York', 'miles_driven': 10000, 'insurance_premium': 700},
    {'cust_id' : '004' , 'name': 'Dave', 'age': 30, 'state': 'Texas', 'miles_driven': 11000, 'insurance_premium': 980},
    {'cust_id' : '005' , 'name': 'Ed', 'age': 32, 'state': 'New York', 'miles_driven': 9000, 'insurance_premium': 970},
    {'cust_id' : '006' , 'name': 'Frank', 'age': 35, 'state': 'California', 'miles_driven': 10000, 'insurance_premium': 650},
    {'cust_id' : '007' , 'name': 'Gary', 'age': 60, 'state': 'California', 'miles_driven': 12000, 'insurance_premium': 680},
    {'cust_id' : '008' , 'name': 'Julia', 'age': 55, 'state': 'Florida', 'miles_driven': 11500, 'insurance_premium': 760},
    {'cust_id' : '009' , 'name': 'Mary', 'age': 53, 'state': 'Illinois', 'miles_driven': 13500, 'insurance_premium': 790},
    {'cust_id' : '0010' , 'name': 'Mike', 'age': 28, 'state': 'Florida', 'miles_driven': 14000, 'insurance_premium': 800},
    {'cust_id' : '0011' , 'name': 'Zac', 'age': 29, 'state': 'California', 'miles_driven': 15000, 'insurance_premium': 860},
    {'cust_id' : '0012' , 'name': 'Jose', 'age': 32, 'state': 'Texas', 'miles_driven': 13000, 'insurance_premium': 900},
    {'cust_id' : '0013' , 'name': 'Luis', 'age': 33, 'state': 'New York', 'miles_driven': 10000, 'insurance_premium': 700},
    {'cust_id' : '0014' , 'name': 'Carlos', 'age': 26, 'state': 'Texas', 'miles_driven': 11000, 'insurance_premium': 980},
    {'cust_id' : '0015' , 'name': 'Linda', 'age': 28, 'state': 'New York', 'miles_driven': 9000, 'insurance_premium': 970},
    {'cust_id' : '0016' , 'name': 'Sophia', 'age': 27, 'state': 'California', 'miles_driven': 10000, 'insurance_premium': 650},
    {'cust_id' : '0017' , 'name': 'Maria', 'age': 29, 'state': 'California', 'miles_driven': 12000, 'insurance_premium': 680},
    {'cust_id' : '0018' , 'name': 'Sriram', 'age': 37, 'state': 'Florida', 'miles_driven': 11500, 'insurance_premium': 760},
    {'cust_id' : '0019' , 'name': 'Ali', 'age': 38, 'state': 'Illinois', 'miles_driven': 13500, 'insurance_premium': 790},
    {'cust_id' : '0020' , 'name': 'Usuf', 'age': 26, 'state': 'Califoria', 'miles_driven': 14000, 'insurance_premium': 800},
    {'cust_id' : '0021' , 'name': 'Husain', 'age': 38, 'state': 'Wisconsin', 'miles_driven': 12900, 'insurance_premium': 790},
    {'cust_id' : '0022' , 'name': 'Raj', 'age': 26, 'state': 'California', 'miles_driven': 14300, 'insurance_premium': 800},
    {'cust_id' : '0023' , 'name': 'Laxman', 'age': 38, 'state': 'Texas', 'miles_driven': 11200, 'insurance_premium': 790},
    {'cust_id' : '0024' , 'name': 'Sita', 'age': 26, 'state': 'Wisconsin', 'miles_driven': 7800, 'insurance_premium': 800},
    {'cust_id' : '0025' , 'name': 'Geeta', 'age': 38, 'state': 'California', 'miles_driven': 7500, 'insurance_premium': 790}
]


print(cust2018)
```

```
custRDD = sc.parallelize(cust2018) \
        .map(lambda cust: (cust['state'], cust)) \
        .partitionBy(5, state_partitioner)

print("Number of partitions: {}".format(custRDD.getNumPartitions()))
print("Partitioner: {}".format(custRDD.partitioner))
print("Partitions structure: {}".format(custRDD.glom().collect()))


# Function for calculating sum of insurance_premium for each partition
# Notice that we are getting an iterator. All work is done on one node
def sum_premiums(iterator):
    yield sum(customer[1]['insurance_premium'] for customer in iterator)

# Sum sales in each partition
sum_amounts = custRDD \
    .mapPartitions(sum_premiums) \
    .collect()

print("Total Insurance_Premiums for each partition: {}".format(sum_amounts))
```

```
Total Insurance_Premiums for each partition: [6130, 3340, 800, 3910, 5910]
```

```
from pyspark.sql import SparkSession, Row

spark = SparkSession(sc)

#set partitions to 5
spark.conf.set("spark.sql.shuffle.partitions", 5)

rdd = sc.parallelize(cust2018) \
        .map(lambda cust: Row(**cust))

df = spark.createDataFrame(rdd)

print("Number of partitions: {}".format(df.rdd.getNumPartitions()))
print("Partitioner: {}".format(rdd.partitioner))
print("Partitions structure: {}".format(df.rdd.glom().collect()))

# Repartition by column
df2 = df.repartition("state")

print("\nAfter 'repartition()'")
print("Number of partitions: {}".format(df2.rdd.getNumPartitions()))
print("Partitioner: {}".format(df2.rdd.partitioner))
print("Partitions structure: {}".format(df2.rdd.glom().collect()))
```

```
rdd = sc.parallelize(nums) \
        .map(lambda num: (num, num)) \
        .partitionBy(2) \
        .persist()

print("Number of partitions: {}".format(rdd.getNumPartitions()))
print("Partitioner: {}".format(rdd.partitioner))
print("Partitions structure: {}".format(rdd.glom().collect()))

# Transform with map() @ the cost the losing the parent's Partitioner
rdd2 = rdd.map(lambda num: (num[0],num[0]*2))

print("Number of partitions: {}".format(rdd2.getNumPartitions()))
print("Partitioner: {}".format(rdd2.partitioner))  # We have lost the partitioner
print("Partitions structure: {}".format(rdd2.glom().collect()))
```

```
rdd = sc.parallelize(nums) \
        .map(lambda num: (num, num)) \
        .partitionBy(2) \
        .persist()

print("Number of partitions: {}".format(rdd.getNumPartitions()))
print("Partitioner: {}".format(rdd.partitioner))
print("Partitions structure: {}".format(rdd.glom().collect()))

# Transform with mapValues() instead of map() such that only values are modified retaining the key. Since
mapValues()
#cannot operate on the key of the tuple, Spark can retain and benefit from the parent's Partitioner.
rdd2 = rdd.mapValues(lambda x: x*2)

print("Number of partitions: {}".format(rdd2.getNumPartitions()))
print("Partitioner: {}".format(rdd2.partitioner))  # We still got the parent's partitioner
print("Partitions structure: {}".format(rdd2.glom().collect()))
```