Appendix C → Questions and Answers

© Mahmoud Parsian, 2019

1. What is Apache Spark?

Apache Spark[1] is a unified analytics engine for large-scale data processing.

Spark has the following key features:

Table 1. Spark Key Features

Feature	Description
Language	Multi-language support: Python, Scala, Java, R, SQL
Speed	10 to 100 times faster than Hadoop, utilizes RAM
Data Sources	It can access diverse data sources: S3, HDFS, relational, …
Data Formats	Supports text, CSV, JSON, Parquet, …
Lazy Evaluation	Actions trigger DAG of computation
Machine Learning	Supports distributed ML algorithms
Graph Algorithms	Has built-in and external Graph libraries
Runs Everywhere	Spark runs on Hadoop, Mesos, Kubernetes, standalone, or in the cloud.

2. What is a Cluster Manager?

Spark's Cluster Manager manages the entire cluster. Spark's **cluster manager** is responsible for the scheduling and allocation of resources across the computer servers forming the cluster.

Spark currently supports several cluster managers:

[1] http://spark.apache.org

Table 2. Cluster Manager Types

Cluster Manager Type	Description
Standalone	A simple cluster manager included with Spark that makes it easy to set up a cluster.
Apache Mesos	A general cluster manager that can also run Hadoop MapReduce and service applications.
Hadoop YARN	The resource manager in Hadoop.
Kubernetes	An open-source system for automating deployment, scaling, and management of containerized applications.
Amazon EC2	Resizable compute capacity in the cloud

3. What is PySpark?

PySpark[2] is a Python API for Apache Spark.

4. What is MapReduce?

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. MapReduce is a very small subset of Spark analytics engine and is a foundation for many big data analytics engines such as:

- Spark[3]
- Hadoop[4]
- Tez[5]

To solve a big data problem by using MapReduce, you focus on writing two distributed functions:

- `map()`:

[2] http://spark.apache.org/docs/latest/api/python/index.html
[3] http://spark.apache.org
[4] https://hadoop.apache.org
[5] http://tez.apache.org

Filters and aggregates data

- `reduce()`:
  Reduces, groups, and summarizes by keys generated by `map()`

To learn MapReduce, you should visit:

- Data-Intensive Text Processing with MapReduce by Jimmy Lin and Chris Dyer[6]
- Data Algorithms by Mahmoud Parsian[7]
- What is MapReduce?[8]

Learning MapReduce will help you to master the Spark and PySpark concepts easily.

5. How is Spark different from MapReduce?

- Overall, Spark is a superset of MapReduce; in a nutshell MapReduce is a very small subset of Spark
- Spark is much faster (10 to 100 times) than MapReduce since it utilizes RAM as much as possible, but MapReduces uses disk I/O.
- Using MapReduce, typically, we do `map()` followed by `combine()` and finally followed by `reduce()`; but Spark frees us from this poor pattern and we can use any combinations of Spark transformations to solve our data problem.
- Spark tries to keep the data "in-memory" as much as possible, which yields to a better performance (if you do not have enough memory for your data, then Spark offers `StorageLevel` to accommodate combinations of memory and disk)
- Spark offers distributed Machine Learning, Graph, and Streaming libraries

6. How do you Compare Spark and Hadoop

Hadoop MapReduce and Spark are compared based on the following features:

[6] https://lintool.github.io/MapReduceAlgorithms/ed1n/MapReduce-algorithms.pdf
[7] http://shop.oreilly.com/product/0636920033950.do
[8] https://www.tutorialspoint.com/hadoop/hadoop_mapreduce

Table 3. Comparison of Spark and Hadoop

Feature	Spark	Hadoop
Speed	10 to 100 times faster than Hadoop	Decent speed
Launch time	up to 2 seconds	15 to 20 seconds
Processing	Near Real-time & Batch processing	Batch processing only
Difficulty	Easy because of high level APIs	Tough to learn, low level
Recovery	Allows recovery of partitions	Fault-tolerant
Interactivity	Has interactive modes	No interactive mode except Pig & Hive
Distributed File System	No	Yes
Machine Learning	Yes	No
Graph Algorithms	Yes	No

7. What are Spark's data abstractions?

Spark offers 3 types of data abstractions: **RDD**, **DataFrame**, and **Dataset**. This means that you can represent your data in any of these data abstractions.

- **RDD**:
  A Resilient Distributed Dataset (RDD), the basic abstraction in Spark. Represents an immutable, partitioned collection of elements that can be operated on in parallel. This class contains the basic operations available on all RDDs, such as `map()`, `filter()`, and `persist()`.
- **DataFrame**:
  DataFrame is a distributed collection of data grouped into named columns. A DataFrame is equivalent to a relational table in Spark SQL, and can be created using various functions in `SparkSession`.
- **Dataset**:

A Dataset is a strongly typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations. Each Dataset also has an untyped view called a DataFrame, which is a Dataset of `Row`. Since Python is not a strongly types language, Dataset is not supported in PySpark.

8. What is a `SparkSession`?

`SparkSession` is the entry point to programming Spark with the Dataset and DataFrame API. A `SparkSession` can be used create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files. To create a `SparkSession`, use the following builder pattern:

```
from pyspark.sql import SparkSession
...
# spark : an instance of SparkSession
spark = SparkSession.builder\
     .master("local") \
     .appName("My App Name")\
     .config("spark.some.config.option", "some-value")\
     .getOrCreate()
```

9. What are actions and transformations?

To manipulate data, Spark provides two functions:

- **Transformations**:
  create new RDD's from existing RDDs and these transformations are lazy and will not be executed until you call any action. Some example of transformations are: `map()`, `filter()`, `reduceByKey()`, `groupByKey()`, `flatMap()`, etc. Therefore, a transformation creates a new RDD.
- **Actions**:
  will return results of an RDD as a tangible data structures. Some example of actions are: `reduce()`, `count()`, `collect()`, `collectAsMap()`, etc. Therefore, an action creates a non-RDD object.

10. What is a `SparkContext`?

`SparkContext` is the main entry point for Spark functionality. A `SparkContext` represents the connection to a Spark cluster, and can be used to create RDD and broadcast variables on that cluster. Note that `SparkContext` instance is not supported to share across multiple processes out of the box, and PySpark does not guarantee multi-processing execution. Use threads instead for concurrent processing purpose. To create a `SparkContext`,use the following pattern:

```
# spark : an instance of SparkSession
# sc : an instance of SparkContext
sc = spark.sparkContext
```

11. What is an RDD?

Resilient Distributed Datasets (RDD) is a core data abstraction in Spark. This means that you can represent your data in RDD. RDD is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed in parallel on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Therefore, we can state that

- RDD is a read-only, partitioned collection of elements/records.
- RDDs can be created through deterministic operations (so called transformations) on either data on stable storage or other RDDs.
- RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are many ways to create RDDs — parallelizing an existing collection (such as arrays and lists) in your driver program, or referencing a dataset in an external storage system, such as a distributed file system, S3, HDFS, HBase, or any data source offering a Hadoop Input Format.

In PySpark, RDD has the following signature:

RDD signature is given below:

```
class pyspark.RDD(
    jrdd,
    ctx,
    jrdd_deserializer=AutoBatchedSerializer(PickleSerializer())
)
A Resilient Distributed Dataset (RDD), the basic
abstraction in  Spark.  Represents an immutable,
partitioned  collection of elements  that can be
operated on in parallel.
```

A RDD of type `T` will be denoted as `RDD[T]`: means that every element of RDD has a type `T`. For example, `RDD[String]` denotes an RDD, where each element is a String object; while `RDD[(String, Integer)]` denotes an RDD where each element is a pair of `(String, Integer)`.

12. How to create an RDD from a text file?

The simplest way to create an `RDD[String]` from a text file is presented below.

First lets create a sample text file:

```
$cat /tmp/sample.txt
record 1 is this
record 2 is here too
record 3 is the last record
```

```
# spark is an instance of SparkSession
input_path = '/tmp/sample.txt'
rdd = spark.sparkContext.textFile(input_path)
rdd.collect()
[
 'record 1 is this',
 'record 2 is here too'
 'record 3 is the last record'
]
rdd.count()
3
```

13. How to create an RDD from a Python collection?

The simplest way to create an `RDD[T]` from a collection is presented below:

```
# spark is an instance of SparkSession
strings = ['fox', 'fox is clever', 'fox jumped']
```

```
# string_rdd : RDD[String]
string_rdd = spark.sparkContext.parallelize(strings)
#
pairs = [('fox', 3), ('java', 4), ('tiger', 7), ('lion', 8)]
key_value_rdd = spark.sparkContext.parallelize(pairs)
# key_value_rdd : RDD[(String, Integer)]
key_value_rdd.collect()
[
 ('fox', 3), ('java', 4), ('tiger', 7), ('lion', 8)
]
key_value_rdd.count()
4
#
tuples = [('fox', 3, 30), ('tiger', 7, 70), ('lion', 8, 80)]
triplets_rdd = spark.sparkContext.parallelize(tuples)
# triplets_rdd : RDD[(String, Integer, Integer)]
triplets_rdd.count()
3
```

14. What are Spark's Actions?

Spark supports 2 types of operations/functions to manipulate data:

- Transformations

```
transformation : RDD --> RDD
```

- Actions

```
action: RDD --> non-RDD
```

Action is an operation which produces a non-RDD value. For example, `map()` is a transformation which accepts a source RDD and creates a new target RDD. But an action accepts an RDD but creates a non-RDD value. For example `RDD.collect()` returns a list containing all elements of a source RDD. Since `collect()` does not return an RDD, it is labeled as an "action". You may use actions to materialize a value in your Spark program. In simple words we can say that, an RDD operation that returns a value of any type but `RDD[T]` is an action.

The following table lists some of Spark's actions:

Action	Meaning
`collect()`	Return a list that contains all of the elements in this RDD.
`reduce`	Reduces the elements of this RDD using the specified commutative and associative binary operator.
`collectAsMap()`	Return the key-value pairs in this RDD to the master as a dictionary.
`saveAsTextFile()`	Save this RDD as a text file, using string representations of elements.

15. What is a Pair RDD?

A Pair RDD is an RDD where its elements are `(key, value)` pairs, where `key` and `value` can be any data type. Some of the Spark transformations require source RDD to be a Pair RDD. For example, `reduceByKey()`, `groupByKey()`, `aggregateByKey()`, and `combineByKey()` require that the source RDD to be a Pair RDD. Pair RDDs allow users to access each key in parallel. For example, `reduceByKey()` transformation merges the values for each key using an **associative** and **commutative** reduce function.

The following are examples of Pair RDDs:

```
# spark is an instance of SparkSession
pairs = [('A', 3), ('A', 4), ('B', 7), ('B', 8), ('B', 9)]
# pairs_rdd : RDD[(String, Integer)] --> a Pair RDD
#         key: String
#       value: Integer
pairs_rdd = spark.sparkContext.parallelize(pairs)
#
more_pairs = [('fox', ('jump', 3)),
              ('fox', ('run', 4)),
              ('tiger', ('run', 7)),
              ('lion', ('hunt', 8))]
pairs_rdd2 = spark.sparkContext.parallelize(more_pairs)
# pairs_rdd2 : RDD[(String, (String, Integer))] --> a Pair RDD
#         key: String
#       value: (String, Integer)
```

16. What is a classic Word Count problem?

Jon Bentley defines Word Frequency Count as: "Given a text file and an integer K, print the K most common words in the file (and the number of their occurrences) in decreasing frequency."

Assuming that words are separated by a single space, word count problem can be solved as:

```
from __future__ import print_function
import sys
from pyspark.sql import SparkSession

# define input
input_path = "/tmp/my/text/files/"

# create an instance of SparkSession
spark = SparkSession.builder.getOrCreate()

# create an RDD[String] from input
records = spark.sparkContext.textFile(input_path)

# short hand notation to find frequencies of words
counts = records.flatMap(lambda x: x.split(' ')) \
                .map(lambda x: (x, 1)) \
                .reduceByKey(lambda x,y: x+y)
frequencies = counts.collect()
for (word, count) in frequencies:
  print("%s: %i" % (word, count))

# done
spark.stop()
```

17. How to filter RDD

The `filter()` function can be used to filter (i.e., drop the non-needed) elements from the source RDD and creates a new target RDD with desired elements.

```
filter(f)
Return a new RDD containing only the
elements that satisfy a predicate.
```

What is a predicate? In mathematical logic, a predicate is commonly understood to be a Boolean-valued function:

```
P: X → {true, false}, called the predicate on X.
```

Let’s understand the `filter()` function by an example:

- `filter()` using Lambda Expression

```
numbers = [-1, 1, 2, -2, 3, 8, 0, -7, 5]
# spark : SparkSession
rdd = spark.sparkContext.parallelize(numbers)
positives = rdd.filter(lambda n: n > 0)
positives.collect()
[
 1, 2, 3, 8, 5
]
```

- `filter()` using function

```
def keep_postives(n):
  if n > 0:
    return True
  else
    return False
#end-def

numbers = [-1, 1, 2, -2, 3, 8, 0, -7, 5]
# spark : SparkSession
rdd = spark.sparkContext.parallelize(numbers)
positives = rdd.filter(keep_postives)
positives.collect()
[
 1, 2, 3, 8, 5
]
```

18. What is a DataFrame?

A DataFrame is a distributed collection of data grouped into named columns. A DataFrame is equivalent to a relational table in Spark SQL, and can be created using various functions in SparkSession. A DataFrame has the following properties:

- DataFrames are distributed in nature, which makes it a fault tolerant and highly available data structure.
- Lazy evaluation is an evaluation strategy: holds the evaluation of an expression until its value is needed.
- DataFrames are immutable in nature. By immutable, it means that they are read-only, and can not be edited/updated.

19. How to create a DataFrame from a text file?

Let's do this with a simple example: consider the following CSV file:

```
cat /tmp/sample.txt
id,name,what
1,google,search
2,illumina,genomics
3,amazon,shopping
4,apple,iphone
```

Next we will read this file and create a DataFrame:

```
input_path = '/tmp/sample.txt'
df = spark.read.format("csv")\
          .option("header","true")\
          .option("inferSchema", "true")
          .load(input_path)
>>> df.show()
+--+--------+---------+
|id|    name|     what|
+--+--------+---------+
| 1|  google|   search|
| 2|illumina| genomics|
| 3|  amazon| shopping|
| 4|   apple|   iphone|
+--+--------+---------+

>>> df.printSchema()
root
 |-- id: integer (nullable = true)
 |-- name: string (nullable = true)
 |-- what: string (nullable = true)
```

20. How to create a DataFrame from a Python collection?

You may use `SparkSession.createDataFrame()` to create a DataFrame from a collection:

```
# spark: an instance of SparkSession
pairs = [('Alice', 21), ('Bob', 34), ('Jane', 58)]
df =spark.createDataFrame(pairs, ['name', 'age'])
df.collect()
[
 Row(name='Alice', age=21)
 Row(name='Bob', age=34)
 Row(name='Jane', age=58)
]
df.show()
+------+----+
|  name| age|
+------+----+
| Alice|  21|
|   Bob|  34|
|  Jane|  58|
+------+----+
```

21. How to use `DataFrame.groupBy()`?

`DataFrame.groupBy()` has the following signature:

```
groupBy(*cols)
Groups the DataFrame using the specified
columns, so we can run aggregation on them.
```

The following example shows how to use `DataFrame.groupBy()`:

```
>>> data = [('Sunnyvale', 34), ('Sunnyvale', 26),
            ('Sunnyvale', 30), ('Ames', 44), ('Ames', 22)]
>>> df = spark.createDataFrame(data, ['city', 'votes'])
>>> df.show()
+---------+-----+
|     city|votes|
+---------+-----+
|Sunnyvale|   34|
|Sunnyvale|   26|
|Sunnyvale|   30|
|     Ames|   44|
|     Ames|   22|
+---------+-----+
```

- Group and Count

```
>>> df.groupBy(df.city).count().show()
+---------+-----+
|     city|count|
+---------+-----+
|     Ames|    2|
|Sunnyvale|    3|
+---------+-----+
```

- Group and Average

```
>>> df.groupBy(df.city).avg().show()
+---------+----------+
|     city|avg(votes)|
+---------+----------+
|     Ames|      33.0|
|Sunnyvale|      30.0|
+---------+----------+
```

- Group and Sum

```
>>> df.groupBy(df.city).sum('votes').show()
+---------+----------+
|     city|sum(votes)|
+---------+----------+
|     Ames|        66|
|Sunnyvale|        90|
+---------+----------+

>>> df.groupBy().sum('votes').show()
+----------+
|sum(votes)|
+----------+
|       156|
+----------+
```

22. What is a Parquet file format?

Parquet is a format that includes metadata about the column data types, offers file compression, and is a file format that is designed to work well with Spark. Apache Parquet[9] is a **columnar** storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.

What are the advantages of Parquet format? Parquet is a columnar file format and has the following advantages:

- Limit I/O operations
- Consumes less space (compact storage)
- Fetches only required columns (in a given query)

23. How to create a Parquet file format using PySpark?

The easiest way to create a Parquet file format is to create a DataFrame and then write/save it as a Parquet format. This is expressed by an example:

```
# spark : SparkSession
data = [('alex', 'Sunnyvale', 20),
        ('jane', 'Cupertino', 30),
        ('max', 'Sunnyvale', 27),
        ('terry', 'Stanford', 60)]
column_names = ["name", "city", "age"]
df = spark.createDataFrame(data, column_names)
df.show()
+-----+---------+---+
| name|     city|age|
+-----+---------+---+
| alex|Sunnyvale| 20|
| jane|Cupertino| 30|
```

[9] https://parquet.apache.org

```
|  max|Sunnyvale| 27|
|terry| Stanford| 60|
+-----+---------+---+

# save df as a set of Parquet file(s)
# create a set of files as a Parquet file format
df.write.parquet("/tmp/multioutput")

# save df as a single Parquet file
# create a single file as a Parquet file format
df.repartition(1).write.parquet("/tmp/singleoutput")
```

24. How to add a new column to a DataFrame?

Consider the following DataFrame:

```
# spark : SparkSession
data = [('alex', 'Sunnyvale', 20),
        ('jane', 'Cupertino', 30),
        ('max', 'Sunnyvale', 27),
        ('terry', 'Stanford', 60)]
column_names = ["name", "city", "age"]
df = spark.createDataFrame(data, column_names)
df.show()
+-----+---------+---+
| name|     city|age|
+-----+---------+---+
| alex|Sunnyvale| 20|
| jane|Cupertino| 30|
|  max|Sunnyvale| 27|
|terry| Stanford| 60|
+-----+---------+---+
```

Next we add a new column (using `DataFrame.withColumn()`), which increases the age column by `10`.

```
new_column_name = 'age10'
df.withColumn(new_column_name, df.age + 10).show()
+-----+---------+---+-----+
| name|     city|age|age10|
+-----+---------+---+-----+
| alex|Sunnyvale| 20|   30|
| jane|Cupertino| 30|   40|
|  max|Sunnyvale| 27|   37|
|terry| Stanford| 60|   70|
+-----+---------+---+-----+
```

25. How to read a Parquet file format using PySpark?

Now, let's read the `users4.parquet` (this Parquet file is available from the Chapter 7 of GitHub[10]). file by using Spark's `DataFrameReader` to create a Spark `DataFrame` (a distributed dataset):

```
# spark : SparkSession
input_path = 'users4.parquet'
users_df = spark.read.parquet(input_path)
users_df.show()
+-----+---------+---+
| name|     city|age|
+-----+---------+---+
| alex|Sunnyvale| 20|
| jane|Cupertino| 30|
|  max|Sunnyvale| 27|
|terry| Stanford| 60|
+-----+---------+---+
```

26. What are Partitions and Partitioning?

To execute tasks in parallel, Spark partitions your input (represented as RDDs and DataFrames) into chunks and then operates on these chunks in parallel. For example, Let's say that you have a data, which has `3000,000,000` records and you created an `RDD[String]` from it (now your RDD has count of `3000,000,000`). There are at least two methods to partition data into chunks. Let's say that your RDD is partitioned into `3,000` chunks (now you have `3,000` partitions and each partition has `1000,000` records/elements:

```
Number of records/elements = 3,000,000,000
Number of partitions: 3,000
Number of records/elements per partition: 1000,000
3,000 x 1000,000 = 3,000,000,000
```

[10] https://github.com/mahmoudparsian/pyspark-algorithms/raw/master/code/chap07/users4.parquet

and you want to perform a `map()` transformation on this RDD, then if you have enough resource, then `3,000` mappers can operate in parallel and each can process `1000,000` records/elements.

- Partition RDD by default
  If you have not set the number of partitions explicitly, then the Spark cluster manager will set the number of partitions for your RDD based on available resources in the cluster. RDDs get partitioned automatically without programmer intervention.
- Partition RDD by programmer
  A programmer can explicitly set the number of partitions. This can be accomplished in many different ways:

  - Method-1: Set the number of partitions when creating an RDD

```
# spark : SparkSession
input_path = '/tmp/file/with/3B_records.txt'
number_of_desired_partitions = 3000
rdd = spark.sparkContext.textfile(
  input_path,
  number_of_desired_partitions
)
rdd.getNumPartitions()
3000
```

- Method-2: Set the number of partitions after creating an RDD

```
# spark : SparkSession
input_path = '/tmp/file/with/3B_records.txt'
# rdd will have default number of partitions
rdd = spark.sparkContext.textfile(input_path)
# next we create a new RDD with the desired number of partitions
number_of_desired_partitions = 3000
# RDD.coalesce(N): return a new RDD
# that is reduced into N partitions
rdd2 = rdd.coalesce(number_of_desired_partitions)
rdd2.getNumPartitions()
3000
```

Therefore, a partition (aka split or chunk) is a logical chunk of a large distributed data set. Spark manages data using partitions that helps parallelize distributed data processing with minimal network traffic for sending data between executors.

27. How to Partition a DataFrame based on Column Value?

Partitioning is a way of dividing a table (represented as a DataFrame abstraction) into related parts based on the values of particular columns like continent, country, and city. Each table in the Hive[11], Amazon Athena[12], or Presto[13] can have one or more partition keys to identify a particular partition. Using partition it is easy to do queries on slices of the data rather than loading the entire data for analysis (of course loading the entire data will be slower than loading slices of data based on partition values such as continent and country).

I will demonstrate the Partitioning concept by an example. Consider the following data, where each record has the following format:

```
<continent><,><country><,><city><,><date-as-YYYY-MM-DD><,><avg-
temperature>
```

Let’s say that the goal is to create partitioned data by (<continent>, <country>, and <city>) for all given records. Partitioning data by these 3 columns gives lots of freedom for writing optimized queries. For example, using Amazon Athens, if you want just to analyze data for city of Sunnyvale, USA, North America, then you just load the proper slice as:

```
select ...
  from temperature
    where
          continent = 'north_america' AND
          country = 'usa' AND
          city = 'Sunnyvale';
```

Therefore after partitioning data, the partitioned data will look like:

[11] https://hive.apache.org
[12] https://aws.amazon.com/athena/
[13] https://github.com/prestodb/presto

Figure 1. Partitioned Data

How do we accomplish partitioning data in Spark. Spark offers a simple API for partitioning data. Let `df` denote the DataFrame for our example data, then we can partition data by `DataFrameWriter.partitionBy()` method as:

```
# df: a DataFrame with the following 5 columns:
#     <continent>
#     <country>
#     <city>
#     <date-as-YYYY-MM-DD>
#     <avg-temperature>
#
# partition data by "continent", "country", and "city"
output_path = '/tmp/temperature' ❶
df.write ❷
  .partitionBy("continent", "country", "city") ❸
  .parquet(output_path) ❹
```

❶ Root directory output path

❷ Get a DataFrameWriter object

❸ Partition your data by your desired columns

❹ Save each partition as a Parquet format

The main steps in this program are:

- STEP-1: Read data and create a DataFrame (as `df`) with five columns:
  - `<continent>`
  - `<country>`
  - `<city>`
  - `<date-as-YYYY-MM-DD>`
  - `<avg-temperature>`

```
# create a DataFrame, note that toDF() returns a
# new DataFrame with new specified column names
# columns = (continent,country,city,date-as-YYYY-MM-DD,avg-temperature)
df = spark.read.option("inferSchema", "true")\
  .csv(input_path)\
  .toDF('continent', 'country', 'city',
        'date-as-YYYY-MM-DD' , 'avg-temperature')
```

- STEP-2: Then use `df.write.partitionBy()` to create partitioned data.

```
# partition data
df.write.partitionBy('continent' , 'country', 'city')\
  .parquet(output_path)
```

If you want to just create a single partitioned file per partition, then we can repartition data, before partitioning:

```
# partition data
df.repartition('continent' , 'country', 'city')\
  .write.partitionBy('continent' , 'country', 'city')\
  .parquet(output_path)
```

- Sample Input

I will use the temperature.txt as an input.

```
$ cat temperature.txt
north_america,usa,Sunnyvale,2019-01-01,56
north_america,usa,Sunnyvale,2019-01-02,50
north_america,usa,Cupertino,2019-02-07,50
north_america,usa,Cupertino,2019-02-08,53
north_america,canada,Vancouver,2019-01-03,44
north_america,canada,Vancouver,2019-07-01,82
europe,france,Paris,2019-07-06,88
europe,france,Paris,2019-07-07,84
europe,france,Paris,2019-07-08,85
```

- Sample Output

After running the PySpark program, partitioned data is created under the output path:

```
$ ls -1R /tmp/temperature/
/tmp/temperature/continent=north_america/country=usa/city=Sunnyvale/
```

```
/tmp/temperature/continent=north_america/country=usa/city=Cupertino/
/tmp/temperature/continent=north_america/country=canada/city=Vancouver/
/tmp/temperature/continent=europe/country=france/city=Paris/
```

Therefore partitioning data enable us to load slice(s) of data rather than the whole data for analysis. In a nutshell, partition columns work as indexes for your data.

28. What is the function of `coalesce()`

Both RDD and DataFrame support `coalesce()`:

- `RDD.coalesce()`:

```
coalesce(numPartitions, shuffle=False)
Return a new RDD that is reduced into
numPartitions partitions.
```

- `DataFrame.coalesce()`:

```
coalesce(numPartitions)
Returns a new DataFrame that has
exactly numPartitions partitions.
```

The `coalesce()` function is used to alter the number of partitions in an RDD but it avoids full shuffle. For example, if the current number of partitions is `4,000` and you want to set to `40` partitions, then there will not be a shuffle, instead each of the `40` new partitions will claim `100` of the current partitions and this does not require a shuffle.

A simple example is provided below:

- Get default number of partitions:

```
SparkSession available as 'spark'.
>>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> rdd = spark.sparkContext.parallelize(data)
>>> rdd.getNumPartitions()
2
```

- Set desired number of partitions:

```
>>> rdd2 = spark.sparkContext.parallelize(data, 4)
>>> rdd2.getNumPartitions()
4

>>># lower the number of partitions:
>>> rdd3 = rdd2.coalesce(2)
>>> rdd3.getNumPartitions()
2

>>># increase the number of partitions
>>># will not work
>>> rdd4 = rdd2.coalesce(10)
>>> rdd4.getNumPartitions()
4
```

29. How to create an RDD with specific number of partitions?

You may specify the number of desired partitions with `textFile()` and `parallelize()` functions:

- Create RDD from a Text File:

```
#SparkSession available as 'spark'.
input_path = ...
num_of_partitions = 10
rdd = spark.sparkContext.textFile(input_path, num_of_partitions)
```

- Create RDD from a collection:

```
#SparkSession available as 'spark'.
simple_collection = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
num_of_partitions = 3
rdd = spark.sparkContext.parallelize(simple_collection,
 num_of_partitions)
```

30. How to use `reduceByKey()` Transformation

This is a distributed reduction transformation and has the following signature:

```
reduceByKey(func,
            numPartitions=None,
            partitionFunc=<function portable_hash>
)

This transformation merges the values for each key using an
associative and commutative reduce function. This will also
perform the  merging locally  on each mapper before sending
results to a reducer, similarly to a "combiner" in MapReduce.
Output will be  partitioned  with numPartitions  partitions,
or the  default  parallelism level  if numPartitions is not
specified. The default partitioner is hash-partition.
```

Given a source RDD of `RDD[(K, V)]`, it produces a target RDD of `RDD[(K, V)]` (note that the source and target RDDs have the same (key, value) data types. This restriction can be removed by using `aggregateByKey()` or `combineByKey()`.

```
# v1 : V
# v2 : V
def reducer_function(v1, v2):
  v = <compute_v_of_type_V_using_v1_and_v2>
  return v
#end-def

source_rdd : RDD[(K, V)]
target_rdd : RDD[(K, V)]
target_rdd = source_rdd.reduceByKey(reducer_function)
```

Consider the following example with 3 keys `{'A', 'B', 'C'}`.

```
pairs = [('A', 2), ('A', 3), ('B', 4), ('B', 5), ('B', 40), ('C', 7)]
# spark : and instance of SparkSession
rdd = spark.sparkContext.parallelize(pairs)
reduced = rdd.reduceByKey(lambda x, y: x+y)
reduced.collect()
[
 ('A', 5),
 ('B', 49),
 ('C', 7)
]
```

31. How to use `groupByKey()` Transformation

This is a distributed reduction transformation and has the following signature:

```
groupByKey(numPartitions=None,
           partitionFunc=<function portable_hash>
)

This transformation groups the values for each key
in the RDD into a single sequence. Hash-partitions
the  resulting  RDD with numPartitions partitions.

Note that if you  are grouping in order to perform
an  aggregation (such  as  a sum or average)  over
each key, using  `reduceByKey()` or `aggregateByKey()`
will provide much better performance.
```

Given a source RDD of `RDD[(K, V)]`, it produces a target RDD of `RDD[(K, [V])]` (note that the source and target RDDs do not have the same value data types. The source value type is V, and the target value type is `[V]` (a sequence of V).

```
source_rdd : RDD[(K, V)]
target_rdd : RDD[(K, [V])]
target_rdd = source_rdd.groupByKey()
```

Consider the following example with 3 keys `{'A', 'B', 'C'}`.

```
pairs = [('A', 2), ('A', 3), ('B', 4), ('B', 5), ('B', 40), ('C', 7)]
# spark : and instance of SparkSession
rdd = spark.sparkContext.parallelize(pairs)
grouped_by_key = rdd.groupByKey()
grouped_by_key.mapValues(lambda l: list(l)).collect()
[
 ('A', [2, 3]),
 ('C', [7]),
 ('B', [4, 5, 40])
]

# next, lets sum up values per key
sum_rdd = grouped_by_key.mapValues(lambda seq: sum(seq))
sum_rdd.collect()
[
 ('C', 7),
 ('A', 5),
 ('B', 49)
]
```

32. How to use `combineByKey()` Transformation

This is a very powerful distributed reduction transformation and has the following signature:

```
combineByKey(create_combiner,
             merge_value,
             merge_combiners,
             numPartitions=None,
             partitionFunc=<function portable_hash>
)

This is a generic reduction function to combine the
elements  for  each  key  using  a  custom set  of
aggregation functions.  Turns an RDD[(K, V)] into a
result of type RDD[(K, C)], for a "combined type" C.
```

Programmer provide three basic functions:

- `create_combiner`, which turns a V into a C (e.g., creates a one-element list)

```
create_combiner: v --> C
```

- `merge_value`, to merge a V into a C (e.g., adds it to the end of a list)

```
merge_value: C, v --> C
```

- `merge_combiners`, to combine two C's into a single one (e.g., merges the lists)

```
merge_combiners: C, C --> C
```

To avoid memory allocation, both `merge_value` and `merge_combiners` are allowed to modify and return their first argument instead of creating a new C.

In addition, users can control the partitioning of the output RDD.

Note that V and C can be different data types — for example, one might group an RDD of type `(Int, Int)` into an RDD of type `(Int, List[Int])`.

Consider the following example with 3 keys `{'A', 'B', 'C'}`.

```
pairs = [('A', 2), ('A', 3), ('A', 4), ('A', 5),
         ('B', 4), ('B', 5), ('B', 15), ('C', 7)]
# spark : and instance of SparkSession
rdd = spark.sparkContext.parallelize(pairs)
```

Let's say that we want to compute (min, max, sum, count, mean) per key: therefore, we want to generate the following output:

```
('A', (2, 5, 14, 4, 3.5)),
('C', (7, 7, 7, 1, 7.0)),
('B', (4, 15, 24, 3, 8.0))
```

From these requirements, first, using `combineByKey()`, we create (min, max, sum, count) per key, and finally, we produce (min, max, sum, count, mean) by using the `mapValues()` per key. Therefore to use `combineByKey()`, we select (min, max, sum, count) as our combined data type (denoted by `C`). Using `combineByKey()` transformation, we can not compute the mean per key since the mean function is not a monoid over a set of integer numbers (you should refer to Chapter 12 for understanding monoids).

To use `combineByKey()`, we need to define 3 functions:

- `create_combiner()` creates a new `C` (as a combined data type) from a given single integer value:

```
# v is a sigle integer value
def create_combiner(v):
  return (v, v, v, 1)
#end-def
```

- `merge_value()` creates a new `C` (as a combined data type) from a given `C` and a single integer value:

```
# C = (min,  max,  sum,  count)
# C = (C[0], C[1], C[2], C[3])
# v = single integer value
def merge_value(C, v):
  minimum = min(v, C[0])
  maximum = max(v, C[1])
  total = v + C[2]
  count = 1 + C[3]
  return (minimum, maximum, total, count)
#end-def
```

- `merge_combiners()` creates a new `C` (as a combined data type) from a given `C` and a `C` (used to merge the values of partitions):

```
# C1 = (min1, max1, sum1, count1)
# C2 = (min2, max2, sum2, count2)
def merge_combiners(C1, C2):
  minimum = min(C1[0], C2[0])
  maximum = max(C1[1], C2[1])
  total = C1[2] + C2[2]
  count = C1[3] + C2[3]
  return (minimum, maximum, total, count)
#end-def
```

Now that we have the required functions, we can write `combineByKey()` as:

```
combined_by_key = rdd.combineByKey(
     create_combiner,
     merge_value,
     merge_combiners
)
combined_by_key.collect()
('A', (2, 5, 14, 4)),
('C', (7, 7, 7, 1)),
('B', (4, 15, 24, 3))
```

Next, we use `mapValues()` to compute the mean per key:

```
result = combined_by_key
  .mapValues(lambda t: (t[0], t[1], t[2], t[3],
             float(t[2]) / float(t[3])))
result.collect()
('A', (2, 5, 14, 4, 3.5)),
('C', (7, 7, 7, 1, 7.0)),
('B', (4, 15, 24, 3, 8.0))
```

33. How to use `mapPartitions()` Transformation

This is a very powerful distributed mapper transformation and has the following singnature:

```
mapPartitions(f, preservesPartitioning=False)
Returns a new RDD by applying a function,
f(), to each partition of this RDD.
```

A typical `map()` transformation maps a single element of source RDD into a single element of the target RDD by applying a function; therefore a `map()` input is a single element of source RDD and the output is a single element of the target RDD.

Let's say that your source RDD has N partitions. Then the `mapPartitions()` transformation maps a single partition of source RDD into your desired data type of T. Therefore the target RDD will be RDD[T] of length N. This is an ideal transformation when you want to reduce (ar aggregate) each partition (comprised of a set of source RDD elements) into a condensed data structure of type T.

For example, consider a source `RDD[Integer]` with `80,000,000,000` elements and assume that your RDD is partitioned into `8,000` chunks (the number of partitions = `8,000`). Therefore, each partition will have about `10,000,000` elements. Therefore if you have enough cluster resources which can run `8,000` mappers in parallel, then each mapper will receive a partition, which is about `10,000,000` elements. Let's say that you want to find (minimum, maximum, count) for the source RDD. Then each mapper will find a local `(minimum, maximum, count)` per partition, and then eventually, you can find the final `(minimum, maximum, count)`` for all of the partitions. Here the target data type is a triplet:

```
   T = (int, int, int) = (minimum, maximum, count)
```

The `mapPartitions()` is an ideal transformation when you want to map each partition into small amount of condensed/reduced information.

The following shows the main flow of `mapPartitions()` transformation:

- First define a function, which accepts a single partition of source RDD (as `RDD[Integer]`) and returns a data type T, where

```
  T = (int, int, int) = (minimum, maximum, count)
```

Therefore given a partition `p` (where `p` in {1, 2, …, 4000}), then `mapPartitions()` will compute ($minimum_p$, $maximum_p$, $count_p$) per partition `p`.

```
def find_min_max_count(single_partition):
  # find (minimum, maximum, count) by iterating single_partition
  return (minimum, maximum, count)
#end-def
```

- Next, apply the transformation:

```
# source RDD: source_rdd = RDD[Integer]
# target RDD: min_max_count_rdd = RDD(int, int, int)
min_max_count_rdd = source_rdd.mapPartitions(find_min_max_count)
min_max_count_list = min_max_count_rdd.collect()
print(min_max_count_list)
[
 (min1, max1, count1),
 (min2, max2, count2),
 ...
 (min4000, max4000, count4000)
]
```

- Finally, we need to collect the content of `min_max_count_rdd` and find the final `(minimum, maximum, count)`:

```
# minimum = min(min1, min2, ..., min4000)
minimum = min(min_max_count_list)[0]
# maximum = max(max1, max2, ..., max4000)
maximum = max(min_max_count_list)[1]
# count = (count1+count2+...+count4000)
count = sum(min_max_count_list)[2]
```

The complete solution is given below:

```
def find_min_max_count(single_partition_iterator):
 first_time = True
 for n in single_partition_iterator:
  if (first_time == True):
   minimum = n;
   maximum = n;
   count = 1
   first_time = False
  else:
   maximum = max(n, maximum)
   minimum = min(n, minimum)
   count = count + 1
 #end-for
 return (minimum, maximum, count)
#end-def
```

Next, let's create an RDD[Integer] and then apply the mapPartitions() transformation:

```
integers = [1, 2, 3, 1, 2, 3, 70, 4, 3, 2, 1]
# spark : SparkSession
source_rdd = spark.sparkContext.parallelize(integers)
# source RDD: source_rdd = RDD[Integer]
# target RDD: min_max_count_rdd = RDD(int, int, int)
min_max_count_rdd = source_rdd.mapPartitions(find_min_max_count)
min_max_count_list = min_max_count_rdd.collect()
# compute the final values:
minimum = min(min_max_count_list)[0]
maximum = max(min_max_count_list)[1]
count = sum(min_max_count_list)[2]
```

In summary, `mapPartitions()` has the following properties:

- `mapPartitions()` can be used as an alternative to `map()` and `foreach()`
- `mapPartitions()` should be used when you want to combine many source RDD elements into a desired element of target RDD
- `mapPartitions()` can be called for each single partition while `map()` and `foreach()` is called for each element in an RDD
- Programmer can do the initialization on per-partition basis rather than each element basis

Therefore, if you have large amount of data, which should be reduced to small amount of information, then the `mapPartitions()` transformation is a possible option. For example to find Min-Max and Top-10, you may use the `mapPartitions()` transformation.

34. How to use `map()` Transformation

This is a very powerful distributed mapper transformation and has the following signature:

```
map(f, preservesPartitioning=False)
Return a new RDD by applying a function
to each element of this RDD.

source_rdd = RDD[U]
target_rdd = RDD[V]
f: U --> V
target_rdd = source_rdd.map(f)
OR
target_rdd = source_rdd.map(lambda u : f(u))
```

Note that function `f()` should be defined as:

```
# u : U
# v : V
def f(u):
  v = <use-u-to-create-data-type-V>
  return v
```

The map function iterates over every element in source RDD and creates a new target RDD. Using `map()` transformation we take in any function, and that function is applied to every element of source RDD.

In the map, we have the flexibility that the input and the return type of RDD may differ from each other. For example, we can have input RDD type as String (source element data type — U), after applying the `map()` function the target RDD can be "list of integers" (target element data type — V).

A `map()` is a 1-to-1 transformation, which maps a single element of source RDD into a single element of the target RDD by applying a programmer provided function; therefore a `map()` input is a single element (data type U) of source RDD and the output is a single element (data type V) of the target RDD.

Let’s learn the `map()` by a simple example:

```
def create_pair(record):
  return (record, len(record))
#end-def
```

Next we transform source RDD of `RDD[String]` into a target RDD of `RDD[(String, Integer)]`:

```
strings = ['fox,run', 'fox,blue', 'lion', 'tiger']
# spark : SparkSession
# rdd : source RDD : RDD[String]
rdd = spark.sparkContext.parallelize(strings)
# rdd_mapped : target RDD : RDD[(String, Integer)]
rdd_mapped = rdd.map(create_pair)
rdd_mapped.collect()
[
 ('fox,run', 7),
 ('fox,blue', 8),
 ('lion', 4),
 ('tiger', 5)
]

rdd_mapped.count()
4
```

Therefore, we can state that the `map()` function expresses a **one-to-one** transformation — it transforms each element of a source RDD into one element of the resulting target RDD. While the `flatMap()` function expresses a **one-to-many** transformation. It transforms each source element to 0, 1, 2, or more of target elements.

35. How to use `flatMap()` Transformation

This is a very powerful distributed mapper transformation and has the following signature:

```
flatMap(f, preservesPartitioning=False)
Return a new RDD by first applying a function to all
elements of this RDD, and then flattening the results.
```

A typical `map()` transformation maps a single element of source RDD into a single element of the target RDD by applying a programmer defined function; therefore a `map()` input is a single element of source RDD and the output is a single element of the target RDD. The `flatMap()` transformation maps a single element of source RDD into 0, 1, 2, or more elements of the target RDD. Note that `flatMap()` is the combination of a `map()` and a `flat` operation i.e, it applies a function to elements as well as flatten them.

Let's learn the `flatMap()` by a simple example: first, we define a function, which returns an empty list (when input has less than 2 tokens) or the entire list.

```
def tokenize_string(rec):
  tokens = rec.split(",")
  if len(tokens) < 2:
    return []
  else:
    return tokens
#end-def
```

```
strings = ['fox,run,jump', 'fox,red,gray,blue', 'lion', 'tiger']
# spark : SparkSession
rdd = spark.sparkContext.parallelize(strings)
rdd_mapped = rdd.map(tokenize_string)
rdd_mapped.collect()
[
 ['fox', 'run', 'jump'],
 ['fox', 'red', 'gray', 'blue'],
 [],
 []
]

rdd_mapped.count()
4
#
rdd_flatmapped = rdd.flatMap(tokenize_string)
rdd_flatmapped.collect()
[
 'fox',
 'run',
 'jump',
 'fox',
 'red',
 'gray',
 'blue'
]
rdd_flatmapped.count()
```

```
7
```

Using `flatMap()`, the empty lists maps into zero target RDD elements.

Therefore, we can state that the `map()` function expresses a **one-to-one** transformation — it transforms each element of a source RDD into one element of the resulting target RDD. While the `flatMap()` function expresses a **one-to-many** transformation. It transforms each element to 0, 1, 2, or more elements.

36. How to use `join()` on 2 RDDs?

The `join()` has the following signature:

```
join(other, numPartitions=None)
Return an RDD containing all pairs of elements
with  matching  keys  in  self  and  other.
```

Let's see how `join()` works:

Each pair of elements will be returned as a `(k, (v1, v2))` tuple, where `(k, v1)` is in self and `(k, v2)` is in other.

The `join()` transformation performs a hash join across the cluster.

```
# spark : SparkSession
x = spark.sparkContext.parallelize(
    [("a", 1), ("a", 5), ("b", 4), ("c", 7)])
y = spark.sparkContext.parallelize(
    [("a", 10), ("b", 40), ("b", 60), ("d", 8)])
x.collect()
[('a', 1), ('a', 5), ('b', 4), ('c', 7)]
y.collect()
[('a', 10), ('b', 40), ('b', 60), ('d', 8)]

joined = x.join(y)
joined.collect()
[
 ('b', (4, 40)),
 ('b', (4, 60)),
 ('a', (1, 10)),
 ('a', (5, 10))
]
```

37. Is there a Graph API in Spark?

GraphX is the core Spark API for graphs and graph-parallel computation. GraphX includes a set of graph algorithms and builders to simplify graph analytics tasks. GraphX (an RDD based API) is available for Java and Scala and **PySpark does not Support GraphX**.

There is an external graph library called GraphFrames[14], which is a package for Spark which provides DataFrame based Graphs. It provides high-level APIs in Scala, Java, and Python. It aims to provide both the functionality of GraphX and extended functionality taking advantage of Spark DataFrames. This extended functionality includes **motif finding**, DataFrame-based serialization, and highly expressive graph queries.

To use GraphFrames with PySpark, you have to include it as "packages" in your PySpark Application. The following shows how to include this package in your Spark Applications:

```
# define GraphFrames package with specific version
export GF_PKG="graphframes:graphframes:0.7.0-spark2.4-s_2.11"

# to use it in PySpark interactively:
$SPARK_HOME/bin/pyspark --packages $GF_PKG

# to submit a PySpark job:
$SPARK_HOME/bin/spark-submit --packages $GF_PKG <pyspark-program>
```

To create graph using GraphFrames:

```
# Create a Vertex DataFrame with unique ID column "id"
# spark : an instance of SparkSession
v = spark.createDataFrame([
  ("a", "Alice", 34),
  ("b", "Bob", 36),
  ("c", "Charlie", 30),
], ["id", "name", "age"])
# Create an Edge DataFrame with "src" and "dst" columns
```

[14] http://graphframes.github.io/graphframes/docs/_site/index.html

```
e = spark.createDataFrame([
  ("a", "b", "friend"),
  ("b", "c", "follow"),
  ("c", "b", "follow"),
], ["src", "dst", "relationship"])
# Create a GraphFrame
from graphframes import GraphFrame
graph = GraphFrame(v, e)
```

38. What is a PageRank Algorithm?

PageRank is one of the methods Google uses to determine a page's relevance or importance. It is only one part of the story when it comes to the Google search listing. In a nutshell, PageRank measures the importance of each vertex in a graph assuming an edge from u to v represents an endorsements of v's importance by u.

GraphFrames package does support PageRank algorithm and the following shows how to use it:

First build your graph:

```
# Create a Vertex DataFrame with unique ID column "id"
# spark : an instance of SparkSession
v = spark.createDataFrame(...)
# Create an Edge DataFrame with "src" and "dst" columns
e = spark.createDataFrame(...)
# Create a GraphFrame
from graphframes import GraphFrame
graph = GraphFrame(v, e)
```

Once the graph is built, then you may apply the PageRank algorithm:

```
page_rank = graph.pageRank(
    resetProbability=0.15,
    sourceId=None,
    maxIter=None,
    tol=None
)
```

For details on using PageRank algorithm, see GraphFrames[15].

[15] http://graphframes.github.io/graphframes/docs/_site/api/python/graphframes.html

39. What are Broadcast Variables?

According to Spark documentation[16]:

```
Broadcast variables allow the programmer to keep  a
read-only  variable cached  on each machine  rather
than shipping a  copy of it  with tasks.  They  can
be  used,  for  example, to  give every node a copy
of a large  input dataset  in an  efficient manner.
Spark   also  attempts   to   distribute  broadcast
variables   using  efficient  broadcast  algorithms
to reduce communication cost.
```

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. This is an example:

```
# spark : an instance of SparkSession
# create a Broadcast Variable
broadcast_var = spark.sparkContext.broadcast([1, 2, 3])
#
# Access the value of a Broadcast Variable
value_read = broadcast_var.value
[1, 2, 3]
```

Broadcast Variables are the **read-only shared variables**. Suppose, there is a set of data which you want to share among all partitions using the `mapPartitions()` transformation. This is how we can accomplish it:

```
# use the Broadcast variable inside the function
def partition_func(partition_iterator):
  # access the value of a Broadcast Variable
  value_read = broadcast_var.value
  for record in partition_iterator:
    ... use value_read here...
  #end-for
  return ...
#end-def
#----
```

[16] https://spark.apache.org/docs/latest/rdd-programming-guide.html#shared-variables

```
# spark : an instance of SparkSession
input_path = "/tmp/my/input"
rdd = spark.sparkContext.textFile(input_path)
mapped = rdd.mapPartitions(partition_func)
```

40. How is machine learning implemented in Spark?

MLlib[17] is Apache Spark’s scalable machine learning library.

Spark’s MLlib contains many algorithms and utilities:

- Classification: logistic regression, naive Bayes,…
- Regression: generalized linear regression, survival regression,…
- Decision trees, random forests, and gradient-boosted trees
- Recommendation: alternating least squares (ALS)
- Clustering: K-means, Gaussian mixtures (GMMs),…
- Topic modeling: latent Dirichlet allocation (LDA)
- Frequent itemsets, association rules, and sequential pattern mining

For details on Spark Machine Learning, refer to Machine Learning Library Guide[18].

41. What is Spark SQL?

Spark SQL is a Spark module for structured data processing. This module integrates relational processing with Spark’s functional programming API. For details on Spark SQL refer to SQL Programming Guide[19].

You may manipulate you data by SQL queries as well. This is how you do it:

- Step-1: Create a DataFrame
- Step-2: Register your created DataFrame as a Table
- Step-3: Execute SQL queries by using a registered table

[17] http://spark.apache.org/mllib/
[18] http://spark.apache.org/docs/latest/ml-guide.html
[19] http://spark.apache.org/docs/latest/sql-programming-guide.html

Simple example is provided below:

- Step-1: Create a DataFrame:

```
>>> triplets = [('alex', 33, 44000), ('jane', 44, 55000),
                ('mary', 27, 32000), ('david', 65, 88000)]
>>> column_names = ['name', 'age', 'salary']
>>> df = spark.createDataFrame(triplets, column_names)
>>> df.show()
+-----+---+------+
| name|age|salary|
+-----+---+------+
| alex| 33| 44000|
| jane| 44| 55000|
| mary| 27| 32000|
|david| 65| 88000|
+-----+---+------+

>>> df.printSchema()
root
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)
 |-- salary: long (nullable = true)
```

- Step-2: Register your created DataFrame as a Table:

```
>>> df.createOrReplaceTempView("people")
```

- Step-3: Execute SQL queries by using a registered table:

```
>>> spark.sql("select * from people").show()
+-----+---+------+
| name|age|salary|
+-----+---+------+
| alex| 33| 44000|
| jane| 44| 55000|
| mary| 27| 32000|
|david| 65| 88000|
+-----+---+------+

>>> spark.sql("select * from people where salary > 50000").show()
+-----+---+------+
| name|age|salary|
+-----+---+------+
| jane| 44| 55000|
|david| 65| 88000|
+-----+---+------+
```

42. How to Create a DataFrame from JSON

Here I assume that each record of input is a JSON object.

```
cat /tmp/sample.json
{"name" : "Michael"}
{"name" : "Rafa", "city" : "Cupertino"}
{"name" : "Andy", "city" : "Ames", "age" : 28}
{"name" : "Justin", "city" : "Sunnyvale", "age":35}
```

Next, we use a SparkSession to create a DataFrame from a JSON:

```
>>> df = spark.read.json("/tmp/sample.json")
>>> df.show()
+----+---------+-------+
| age|     city|   name|
+----+---------+-------+
|null|     null|Michael|
|null|Cupertino|   Rafa|
|  28|     Ames|   Andy|
|  35|Sunnyvale| Justin|
+----+---------+-------+

>>> df.printSchema()
root
 |-- age: long (nullable = true)
 |-- city: string (nullable = true)
 |-- name: string (nullable = true)
```

43. What is Lazy Evaluation in Spark?

In order to understand the concept of Lazy Evaluation, we do need to understand the concept of DAG (Directed Acyclic Graph). A DAG in Spark is a set of Vertices (a.k.a nodes) and Edges, where vertices represent the RDDs (data) and the edges represent the operation (function) to be applied on RDD. In Spark DAG, every edge directs from earlier to later in the sequence. When you call an action (such as `count()`, `reduce()`, `collect()`, `...`), the created DAG submits to **DAG Scheduler** which further splits the graph into the stages of the task.

- A simple example of DAG is provided below:

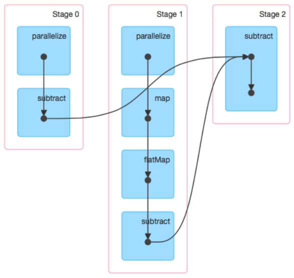


Figure 2. Simple DAG Example

- A complex example of DAG is provided below:

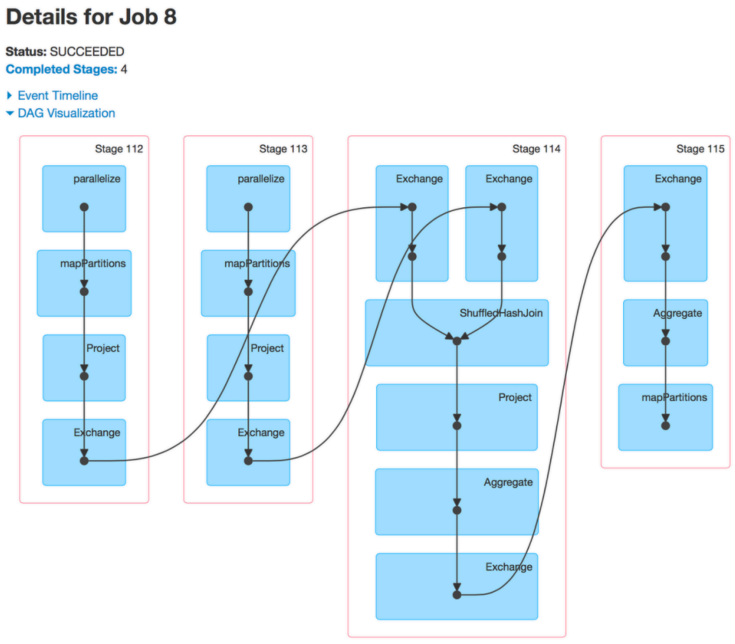


Figure 3. Complex DAG Example

Therefore, we can state that Spark uses **lazy evaluation**, so transformations are not actually executed until an action occurs. Spark's use of **lazy evaluation** can make debugging more difficult because code is not always executed immediately.

For more information on Spark's Lazy Evaluation, visit the following links:

- Why its lazy and what is the advantage?[20]
- Apache Spark Lazy Evaluation: In Spark RDD[21]

44. What are the various levels of persistence in Spark?

Spark automatically persists the intermediary data from various shuffle operations (mappers, reducers, …). A programmer can set the persistence level by the `RDD.persist()` method:

```
persist(storageLevel=StorageLevel(False, True, False, False, 1))

Set  this  RDD's storage level  to persist  its values
across operations after the first time it is computed.
This can only be used to assign a new storage level if
the RDD does not have a storage level set yet.   If no
storage  level is specified defaults to (MEMORY_ONLY).
```

PySpark supports the following storage levels:

```
DISK_ONLY             = StorageLevel(True, False, False, False, 1)
DISK_ONLY_2           = StorageLevel(True, False, False, False, 2)
MEMORY_AND_DISK       = StorageLevel(True, True, False, False, 1)
MEMORY_AND_DISK_2     = StorageLevel(True, True, False, False, 2)
MEMORY_AND_DISK_SER   = StorageLevel(True, True, False, False, 1)
MEMORY_AND_DISK_SER_2 = StorageLevel(True, True, False, False, 2)
MEMORY_ONLY           = StorageLevel(False, True, False, False, 1)
MEMORY_ONLY_2         = StorageLevel(False, True, False, False, 2)
MEMORY_ONLY_SER       = StorageLevel(False, True, False, False, 1)
MEMORY_ONLY_SER_2     = StorageLevel(False, True, False, False, 2)
OFF_HEAP              = StorageLevel(True, True, True, False, 1)
```

For details on this refer to Spark's StorageLevel[22].

[20] https://stackoverflow.com/questions/38027877/spark-transformation-why-its-lazy-and-what-is-the-advantage
[21] https://techvidvan.com/tutorials/spark-lazy-evaluation/
[22] http://spark.apache.org/docs/latest/api/python/pyspark.html?highlight=storagelevel#pyspark.StorageLevel

The following example demonstrates the concept of `StorageLevel` and `RDD.persist()`:

```
>>> data = [1, 2, 3, 4]
>>> rdd = spark.sparkContext.parallelize(data)
>>> rdd.getStorageLevel()
StorageLevel(False, False, False, False, 1)
>>> print(rdd.getStorageLevel())
Serialized 1x Replicated
>>> from pyspark import StorageLevel
>>> rdd.persist(StorageLevel.MEMORY_AND_DISK_2)
ParallelCollectionRDD[48] at parallelize at PythonRDD.scala:195
>>> rdd.getStorageLevel()
StorageLevel(True, True, False, False, 2)
>>> print(rdd.getStorageLevel())
Disk Memory Serialized 2x Replicated
```

Therefore, when you call `RDD.persist()`, you can specify that you want to store the RDD on the disk or in the memory or both.

`RDD.cache()` is like `RDD.persist()' function only, where the storage level is set to memory only.

45. What is Lineage Graph?

RDDs are created from data sources (text file, HHDFS, S3, …), collections (tuples, lists and arrays), and transformations. The RDDs in Spark, depend on one or more other RDDs. For example, here

```
rdd1 = spark.sparkContext.textFile(input_path)
rdd2 = rdd1.map(...)
rdd3 = rdd2.filter(...)
```

`rdd3` depends on `rdd2`, which depends on `rdd1`.

The representation of dependencies in between RDDs is known as the lineage graph. The lineage graph for our simple example will be

Figure 4. RDD Lineage Example-1

How do we compute each RDD on demand? The lineage graph information is used to compute each RDD on demand, so that whenever a part of persistent RDD is lost (due to server failures, network failures, …), the data that is lost can be recovered using the lineage graph information.

Therefore, RDD Lineage is a graph of all the parent RDDs of a RDD. It is built as a result of applying transformations to the RDD and creates a logical execution plan. To understand the RDD lineage, consider the following transformations:

```
#spark : an instance of SparkSession
rdd_A = spark.sparkContext.parallelize(range(1, 100))
rdd_B = spark.sparkContext.parallelize(range(1, 100, 10))
rdd_C = rdd_A.cartesian(rdd_B)
rdd_D = rdd_A.map(lambda n : (n, n+n))
rdd_E = rdd_A.zip(rdd_B)
rdd_F = rdd_B.keyBy(lambda x: x*x)
rdd_G = spark.sparkContext.union(rdd_A, rdd_B, rdd_C)
```

RDD lineage for created RDDs will look like:

Figure 5. RDD Lineage Example-2

You may use `RDD.toDebugString()` to see description of this RDD and its recursive dependencies for debugging purposes.

46. How do you monitor Spark jobs?

According to Spark documentation, there are several ways to monitor Spark applications: web UIs, metrics, and external instrumentation.

You can access Spark’s web interface by simply opening the following URL:

```
http://<driver-node>:4040
```

To understand Monitoring and Instrumentation of Spark jobs, you should refer to Monitoring and Instrumentation[23].

[23] https://spark.apache.org/docs/latest/monitoring.html

47. What is `foreach()`?

The `foreach(f)` applies a function `f()` to all elements of the source RDD. The `foreach()` action has the following properties:

- `foreach()` operation is an action.
- It do not return any value.
- It executes input function (`f()`) on each element of an RDD.

The following is an example for using `foreach()` action:

```
def f(x):
   print(x, x*x)
#end-def

# spark : and instance of SparkSession
spark.sparkContext.parallelize([1, 2, 3, 4]).foreach(f)
1 1
2 4
3 8
4 16
```

This action is an ideal for creating data to be loaded to databases and web services.

48. Explain Spark's `countByKey()` operation

`RDD.countByKey()` is an action, which counts the number of elements for each key, and return the result to the master as a dictionary. This action operates on Pair RDDs (an RDD where all elements have `` `(key, value)`` pairs). The following example shows how to uses this action:

```
# spark : an instance of SparkSession
pairs = [("a", 1), ("b", 6), ("a", 1), ("a", 7)]
rdd = spark.sparkContext.parallelize(pairs)
sorted(rdd.countByKey().items())
[('a', 3), ('b', 1)]
```

49. What is Spark Streaming?

According to Spark documentation[24]: "Spark Streaming brings Apache Spark’s language-integrated API to stream processing, letting you write streaming jobs the same way you write batch jobs. It supports Java, Scala and Python." If you have situation, where there is data flowing continuously (such as Twitter feeds, data generated by car engines, DNA sequencing machines, etc.) and you want to process the data as early as possible, in that case you can use Spark Streaming. Spark provides the Streaming API for stream processing of live data.

High-level Spark streaming is illustrated below:

Figure 6. Spark Streaming Data Flow

Where does the source data come from? Source data can flow for Kafka, Flume or from TCP sockets, Amazon Kinesis etc., and you can do complex processing on the data before you pushing them into their final data sources. Final data sources (also called destination) can be file systems (HDFS, S3) or databases or any other dashboards.

Spark Streaming echo system is presented by Figure Appendix-C-4 (source: databricks[25])

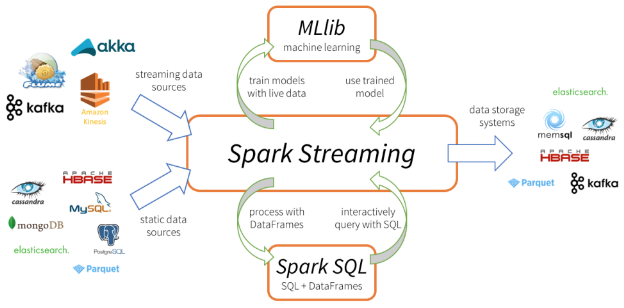


Figure 7. Spark Streaming Echo System

[24] http://spark.apache.org/streaming/
[25] https://databricks.com/glossary/what-is-spark-streaming

50. How to read LIBSVM data format?

LIBSVM[26] is a format used by machine learning algorithms. It is a text format in which each line represents a labeled sparse feature vector using the following format:

```
label index1:value1 index2:value2 ...
```

Consider the following sample data in LIBSVM format:

```
$ cat /tmp/sample_libsvm.txt
1 12:24 34:56 35:45
0 11:45 34:66 37:46
1 14:34 44:59 45:48
0 13:35 33:71 74:40
```

You may use `MLUtils.loadLibSVMFile()` to read LIBSVM data format:

```
from pyspark.mllib.util import MLUtils
input_path = '/tmp/sample_libsvm.txt'
# spark : an instance of SparkSession
>>> examples = MLUtils.loadLibSVMFile(spark.sparkContext, input_path)
>>> examples.collect()
[
 LabeledPoint(1.0, (74,[11,33,34],[24.0,56.0,45.0])),
 LabeledPoint(0.0, (74,[10,33,36],[45.0,66.0,46.0])),
 LabeledPoint(1.0, (74,[13,43,44],[34.0,59.0,48.0])),
 LabeledPoint(0.0, (74,[12,32,73],[35.0,71.0,40.0]))
]
>>>
```

51. How to generate sliding data?

Suppose you want to create an RDD from grouping items of its parent RDD in fixed size blocks by passing a sliding window over them.

First we define a function, which creates a window:

```
def create_window(xi, n):
```

[26] https://www.csie.ntu.edu.tw/~cjlin/libsvm/

```
  x, i = xi
  return [(i - offset, (i, x)) for offset in range(n)]
#end-def
```

Next, we define a sliding function:

```
def sliding(rdd, n):
  assert n > 0
  return (rdd.zipWithIndex()
    .flatMap(lambda xi: create_window(xi, n))
    .groupByKey()
    .mapValues(lambda values: [x for (i, x) in sorted(values)])
    .sortByKey()
    .values()
    .filter(lambda x: len(x) == n))
#end-def
```

Now, let's use `sliding()` to create sliding data:

```
>>> data
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> rdd = spark.sparkContext.parallelize(data)
>>> rdd.collect()
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s = sliding(rdd, 3)
>>> s.collect()
[
 [1, 2, 3],
 [2, 3, 4],
 [3, 4, 5],
 [4, 5, 6],
 [5, 6, 7],
 [6, 7, 8],
 [7, 8, 9]
]
```

52. How to Read Objects from DynamoDB?

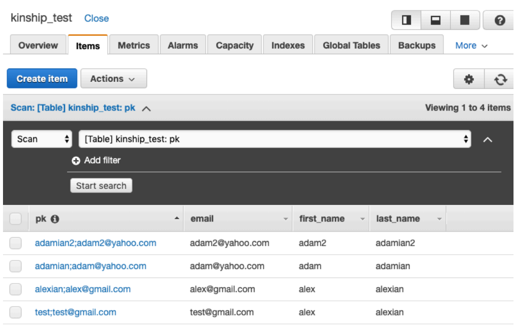
According to Amazon.com: "Amazon DynamoDB is a fully managed proprietary NoSQL database service that supports key-value and document data structures and is offered by Amazon.com as part of the Amazon Web Services portfolio." Why is DynamoDB important? DynamoDB lets you offload the administrative burdens of operating and scaling a distributed database, so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling.

To use DynamoDB, you just need to create a DynamoDB table (note that server or cluster set up is not required — DynamoDB is provided as a service) and then start writing items/objects as key-value pairs. Then you may read a single item/object from DynamoDB table (for example by providing a primary key) or you may scan all items/objects from an existing table. Your DynamoDB table must have a primary key field.

To accomplish this task, I used an external Spark library spark-dynamodb[27].

52.1. Input

Assume that the table name is "kinship_test" and has 4 items:



pk	email	first_name	last_name
adamian2;adam2@yahoo.com	adam2@yahoo.com	adam2	adamian2
adamian;adam@yahoo.com	adam@yahoo.com	adam	adamian
alexian;alex@gmail.com	alex@gmail.com	alex	alexian
test;test@gmail.com	test@gmail.com	alex	alexian

Figure 8. DynamoDB Sample Items

52.2. Read by Python

For testing and debugging purposes, I used the Python's `boto3` library to read all items for the "kinship_test" table.

- Read all items:

```
$ python
```

[27] https://github.com/audienceproject/spark-dynamodb

```
Python 3.7.2
>>> import boto3
>>> dynamodb = boto3.resource('dynamodb')
>>> table = dynamodb.Table('kinship_test')
>>> print(table)
dynamodb.Table(name='kinship_test')
>>> items = table.scan()
```

- Display all items:
  Note the pk column, which denotes a primary key field (you may use any name you wish).

```
>>> items
{
u'Count': 4,
u'Items':
[
{u'pk': u'adamian2;adam2@yahoo.com', u'first_name': u'adam2',
 u'last_name': u'adamian2', u'email': u'adam2@yahoo.com'},
{u'pk': u'alexian;alex@gmail.com', u'first_name': u'alex',
 u'last_name': u'alexian', u'email': u'alex@gmail.com'},
{u'pk': u'adamian;adam@yahoo.com', u'first_name': u'adam',
 u'last_name': u'adamian', u'email': u'adam@yahoo.com'},
{u'pk': u'test;test@gmail.com', u'first_name': u'alex',
 u'last_name': u'alexian', u'email': u'test@gmail.com'}
],
u'ScannedCount': 4,
'ResponseMetadata': {'RetryAttempts': 0, 'HTTPStatusCode': 200, ...}
}
>>>
```

52.3. Read by PySpark

I used the Spark DynamoDB[28] library to read all items for the "kinship_test" table.

- Define required libraries:

```
$ export JAR1="/home/hadoop/algorithms/lib/spark-
dynamodb_2.11-0.4.3.jar"
$ export JAR2="/home/hadoop/algorithms/lib/scala-library-2.13.0.jar"
```

[28] https://github.com/audienceproject/spark-dynamodb

```
$ export JAR3="/home/hadoop/algorithms/lib/scala-csv_2.13-1.3.6.jar"
$ pyspark --jars "${JAR1},${JAR2},${JAR3}"
Python 3.7.2
Welcome to Spark version 2.4.3
SparkSession available as 'spark'.
```

- Read DynamoDB Table as a Spark DataFrame:

```
>>> df = spark.read.format("com.audienceproject.spark.dynamodb")
              .option("tableName", "kinship_test")
              .option("region", "us-east-1").load()
>>> df.show(truncate=False)
+---------+-----------------------+----------+---------------+
|last_name|pk                     |first_name|email          |
+---------+-----------------------+----------+---------------+
|adamian2 |adamian2;adam2@yahoo.com|adam2     |adam2@yahoo.com|
|alexian  |alexian;alex@gmail.com  |alex      |alex@gmail.com |
|adamian  |adamian;adam@yahoo.com  |adam      |adam@yahoo.com |
|alexian  |test;test@gmail.com     |alex      |test@gmail.com |
+---------+-----------------------+----------+---------------+
>>>
>>> df.count()
4
```

- Apply `filter()` on created DataFrame:

```
>>> df.filter(df.first_name == 'alex').show(truncate=False)
+---------+----------------------+----------+--------------+
|last_name|pk                    |first_name|email         |
+---------+----------------------+----------+--------------+
|alexian  |alexian;alex@gmail.com|alex      |alex@gmail.com|
|alexian  |test;test@gmail.com   |alex      |test@gmail.com|
+---------+----------------------+----------+--------------+

>>> df.filter(df.first_name == 'alex')
      .filter(df.email == 'alex@gmail.com')
      .show(truncate=False)
+---------+----------------------+----------+--------------+
|last_name|pk                    |first_name|email         |
+---------+----------------------+----------+--------------+
|alexian  |alexian;alex@gmail.com|alex      |alex@gmail.com|
+---------+----------------------+----------+--------------+
```

53. How to Write Objects to DynamoDB?

You may save your Spark DataFrame in DynamoDb table, where each row of a DataFrame becomes an "item" of a DynamoDB.

To accomplish this task, I used an external Spark library spark-dynamodb[29].

I will demonstrate this with a simple example.

53.1. Create a DynamoDb Table

For this, I created a DynamoDb table with name of `test_table` with primary key of `pk` (you may name this primary key whatever you like, here I named it as `pk`).

There are so many ways to create a DynamoDB table. You may create a table by AWS command line, Java API, and AWS Web console.

Create a DynamoDB by AWS Web Console

Step-1: Open the DynamoDB console[30].

Step-2: Choose Create Table.

Step-3: In the Create DynamoDB table screen, do the following:

- On the Table name box, enter your table name (your choice)
- For the Primary key, in the Partition key box, enter a primary key name (your choice). Set the data type to Number.
- When the settings are as you want them, choose Create.

Create a DynamoDB Table from Command Line

You may create a DynamoDB Table from AWS Command Line. For details, see Amazon DynamoDB Table Creation[31].

[29] https://github.com/audienceproject/spark-dynamodb
[30] https://console.aws.amazon.com/dynamodb/
[31] https://docs.aws.amazon.com/cli/latest/reference/dynamodb/create-table.html

53.2. Define Required Libraries

```
$ export JAR1="/home/hadoop/algorithms/lib/spark-
dynamodb_2.11-0.4.3.jar"
$ export JAR2="/home/hadoop/algorithms/lib/scala-library-2.13.0.jar"
$ export JAR3="/home/hadoop/algorithms/lib/scala-csv_2.13-1.3.6.jar"
$ pyspark --jars "${JAR1},${JAR2},${JAR3}"
Python 3.7.2
Welcome to Spark version 2.4.3
SparkSession available as 'spark'.
```

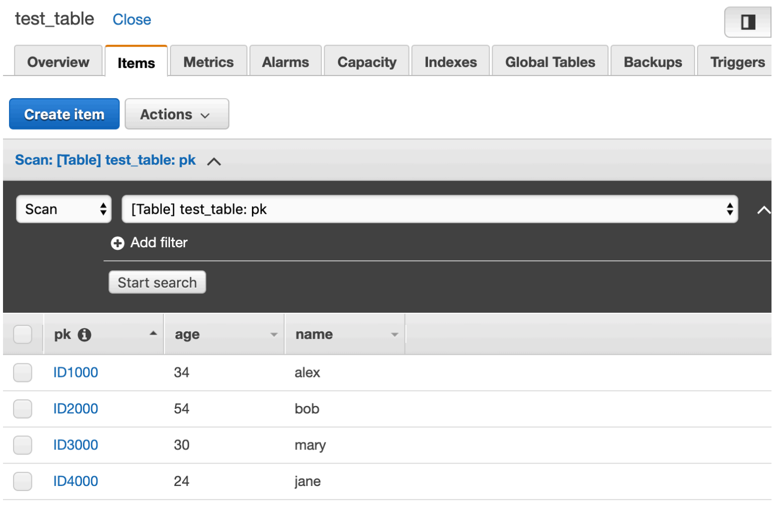
53.3. Create your Spark DataFrame

```
>>> columns = ['pk', 'name', 'age']
>>> data = [('ID1000', 'alex', 34), ('ID2000', 'bob', 54),
            ('ID3000', 'mary', 30), ('ID4000', 'jane', 24)]
>>> df = spark.createDataFrame(data, columns)
>>> df.show(truncate=False)
+------+----+---+
|pk    |name|age|
+------+----+---+
|ID1000|alex|34 |
|ID2000|bob |54 |
|ID3000|mary|30 |
|ID4000|jane|24 |
+------+----+---+
```

53.4. Write created DataFrame into a DynamoDB Table

```
>>> df.write.format("com.audienceproject.spark.dynamodb")
      .option("tableName", "test_table")
      .option("region", "us-east-1")
      .save()
```

53.5. Display created Items of a DynamoDB Table



pk	age	name
ID1000	34	alex
ID2000	54	bob
ID3000	30	mary
ID4000	24	jane

Figure 9. DynamoDB Sample Items