Sequence Files Access with Spark Core API

`sequenceFile`(*path*, *keyClass=None*, *valueClass=None*, *keyConverter=None*, *valueConverter=None*, *minSplits=None*, *batchSize=0*) [source]

Read a Hadoop SequenceFile with arbitrary key and value Writable class from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI. The mechanism is as follows:

1. A Java RDD is created from the SequenceFile or other InputFormat, and the key and value Writable classes
2. Serialization is attempted via Pyrolite pickling
3. If this fails, the fallback is to call 'toString' on each key and value
4. **`PickleSerializer`** is used to deserialize pickled objects on the Python side

Parameters:

- **path** – path to sequncefile
- **keyClass** – fully qualified classname of key Writable class (e.g. "org.apache.hadoop.io.Text")
- **valueClass** – fully qualified classname of value Writable class (e.g. "org.apache.hadoop.io.LongWritable")
- **keyConverter** –
- **valueConverter** –
- **minSplits** – minimum splits in dataset (default min(2, sc.defaultParallelism))
- **batchSize** – The number of Python objects represented as a single Java object. (default 0, choose batchSize automatically)

Sequence Files Access with Spark Core API

```python
import re
from pyspark import SparkContext

if __name__ == "__main__":
    sc = SparkContext()

    lines = sc.textFile("hdfs://devenv/user/spark/spark101/wordcount/data")

    words = lines.flatMap(lambda x: re.compile(r'\W+').split(x.lower()))

    word_counts = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)

    # Write output in Sequence Files
    word_counts.saveAsSequenceFile("hdfs://devenv/user/spark/spark101/wordcount/output_seq")

    # Read output from the saved Sequence Files
    word_counts = sc.sequenceFile("hdfs://devenv/user/spark/spark101/wordcount/output_seq")

    print(word_counts.collect())
```

Avro Files Access with Spark SQL API

The `spark-avro` module is external and not included in `spark-submit` or `spark-shell` by default.

As with any Spark applications, `spark-submit` is used to launch your application. `spark-avro_2.12` and its dependencies can be directly added to `spark-submit` using `--packages`, such as,

```
./bin/spark-submit --packages org.apache.spark:spark-avro_2.12:2.4.5 ...
```

Load and Save Functions

Since `spark-avro` module is external, there is no `.avro` API in `DataFrameReader` or `DataFrameWriter`.

To load/save data in Avro format, you need to specify the data source option `format` as `avro`(or `org.apache.spark.sql.avro`).

Scala | Java | **Python** | R

```
df = spark.read.format("avro").load("examples/src/main/resources/users.avro")
df.select("name", "favorite_color").write.format("avro").save("namesAndFavColors.avro")
```

Parquet Files Access with Spark SQL API

Parquet is a columnar format that is supported by many other data processing systems. Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data. When writing Parquet files, all columns are automatically converted to be nullable for compatibility reasons.

Loading Data Programmatically

Using the data from the above example:

Scala | Java | **Python** | R | Sql

```
peopleDF = spark.read.json("examples/src/main/resources/people.json")

# DataFrames can be saved as Parquet files, maintaining the schema information.
peopleDF.write.parquet("people.parquet")

# Read in the Parquet file created above.
# Parquet files are self-describing so the schema is preserved.
# The result of loading a parquet file is also a DataFrame.
parquetFile = spark.read.parquet("people.parquet")

# Parquet files can also be used to create a temporary view and then used in SQL statements.
parquetFile.createOrReplaceTempView("parquetFile")
teenagers = spark.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenagers.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+
```