sparkly Documentation

Release 2.4.1

Tubular

Nov 20, 2018

Contents

Sparkly is a library that makes usage of pyspark more convenient and consistent.

A brief tour on Sparkly features:

```
# The main entry point is SparklySession,
# you can think of it as of a combination of SparkSession and SparkSession.builder.
from sparkly import SparklySession


# Define dependencies in the code instead of messing with `spark-submit`.
class MySession(SparklySession):
    # Spark packages and dependencies from Maven.
    packages = [
        'datastax:spark-cassandra-connector:2.0.0-M2-s_2.11',
        'mysql:mysql-connector-java:5.1.39',
    ]

    # Jars and Hive UDFs
    jars = ['/path/to/brickhouse-0.7.1.jar'],
    udfs = {
        'collect_max': 'brickhouse.udf.collect.CollectMaxUDAF',
    }


spark = MySession()

# Operate with interchangeable URL-like data source definitions:
df = spark.read_ext.by_url('mysql://<my-sql.host>/my_database/my_database')
df.write_ext('parquet:s3://<my-bucket>/<path>/data?partition_by=<field_name1>')

# Interact with Hive Metastore via convenient python api,
# instead of verbose SQL queries:
spark.catalog_ext.has_table('my_custom_table')
spark.catalog_ext.get_table_properties('my_custom_table')

# Easy integration testing with Fixtures and base test classes.
from pyspark.sql import types as T
from sparkly.testing import SparklyTest


class TestMyShinySparkScript(SparklyTest):
    session = MySession

    fixtures = [
        MysqlFixture('<my-testing-host>', '<test-user>', '<test-pass>', '/path/to/
↪data.sql', '/path/to/clear.sql')
    ]

   def test_job_works_with_mysql(self):
      df = self.spark.read_ext.by_url('mysql://<my-testing-host>/<test-db>/<test-
↪table>?user=<test-usre>&password=<test-password>')
      res_df = my_shiny_script(df)
      self.assertRowsEqual(
         res_df.collect(),
         [T.Row(fieldA='DataA', fieldB='DataB', fieldC='DataC')],
      )
```

CHAPTER 1

Sparkly Session

`SparklySession` is the main entry point to sparkly's functionality. It's derived from `SparkSession` to provide additional features on top of the default session. The are two main differences between `SparkSession` and `SparklySession`:

1. `SparklySession` doesn't have `builder` attribute, because we prefer declarative session definition over imperative.
2. Hive support is enabled by default.

The example below shows both imperative and declarative approaches:

```
# PySpark-style (imperative)
from pyspark import SparkSession

spark = SparkSession.builder\
    .appName('My App')\
    .master('spark://')\
    .config('spark.sql.shuffle.partitions', 10)\
    .getOrCreate()

# Sparkly-style (declarative)
from sparkly import SparklySession

class MySession(SparklySession):
    options = {
        'spark.app.name': 'My App',
        'spark.master': 'spark://',
        'spark.sql.shuffle.partitions': 10,
    }

spark = MySession()

# In case you want to change default options
spark = MySession({'spark.app.name': 'My Awesome App'})

# In case you want to access the session singleton
spark = MySession.get_or_create()
```

Installing dependencies

Why: Spark forces you to specify dependencies (spark packages or maven artifacts) when a spark job is submitted (something like `spark-submit --packages=...`). We prefer a code-first approach where dependencies are actually declared as part of the job.

For example: You want to read data from Cassandra.

```
from sparkly import SparklySession


class MySession(SparklySession):
    # Define a list of spark packages or maven artifacts.
    packages = [
        'datastax:spark-cassandra-connector:2.0.0-M2-s_2.11',
    ]

# Dependencies will be fetched during the session initialisation.
spark = MySession()

# Here is how you now can access a dataset in Cassandra.
df = spark.read_ext.by_url('cassandra://<cassandra-host>/<db>/<table>?
↪consistency=QUORUM')
```

Custom Maven repositories

Why: If you have a private maven repository, this is how to point spark to it when it performs a package lookup. Order in which dependencies will be resolved is next:

- Local cache
- Custom maven repositories (if specified)
- Maven Central

For example: Let's assume your maven repository is available on: http://my.repo.net/maven, and there is some spark package published there, with identifier: *my.corp:spark-handy-util:0.0.1* You can install it to a spark session like this:

```
from sparkly import SparklySession

class MySession(SparklySession):
    repositories = ['http://my.repo.net/maven']
    packages = ['my.corp:spark-handy-util:0.0.1']

spark = MySession()
```

Tuning options

Why: You want to customise your spark session.

For example:

- `spark.sql.shuffle.partitions` to tune shuffling;
- `hive.metastore.uris` to connect to your own HiveMetastore;

- `spark.hadoop.avro.mapred.ignore.inputs.without.extension` package specific options.

```
from sparkly import SparklySession


class MySession(SparklySession):
    options = {
        # Increase the default amount of partitions for shuffling.
        'spark.sql.shuffle.partitions': 1000,
        # Setup remote Hive Metastore.
        'hive.metastore.uris': 'thrift://<host1>:9083,thrift://<host2>:9083',
        # Ignore files without `avro` extensions.
        'spark.hadoop.avro.mapred.ignore.inputs.without.extension': 'false',
    }

# You can also overwrite or add some options at initialisation time.
spark = MySession({'spark.sql.shuffle.partitions': 10})
```

Tuning options through shell environment

Why: You want to customize your spark session in a way that depends on the hardware specifications of your worker (or driver) machine(s), so you'd rather define them close to where the actual machine specs are requested / defined. Or you just want to test some new configuration without having to change your code. In both cases, you can do so by using the environmental variable `PYSPARK_SUBMIT_ARGS`. Note that any options defined this way will override any conflicting options from your Python code.

For example:

- `spark.executor.cores` to tune the cores used by each executor;
- `spark.executor.memory` to tune the memory available to each executor.

```
PYSPARK_SUBMIT_ARGS='--conf "spark.executor.cores=32" --conf "spark.executor.
↪memory=160g"' \
    ./my_spark_app.py
```

Using UDFs

Why: To start using Java UDF you have to import JAR file via SQL query like `add jar ../path/to/file` and then call `registerJavaFunction`. We think it's too many actions for such simple functionality.

For example: You want to import UDFs from brickhouse library.

```
from pyspark.sql.types import IntegerType
from sparkly import SparklySession


def my_own_udf(item):
    return len(item)


class MySession(SparklySession):
    # Import local jar files.
    jars = [
```

```
        '/path/to/brickhouse.jar'
    ]
    # Define UDFs.
    udfs = {
        'collect_max': 'brickhouse.udf.collect.CollectMaxUDAF',  # Java UDF.
        'my_udf': (my_own_udf, IntegerType()),  # Python UDF.
    }

spark = MySession()

spark.sql('SELECT collect_max(amount) FROM my_data GROUP BY ...')
spark.sql('SELECT my_udf(amount) FROM my_data')
```

Lazy access / initialization

Why: A lot of times you might need access to the sparkly session at a low-level, deeply nested function in your code. A first approach is to declare a global sparkly session instance that you access explicitly, but this usually makes testing painful because of unexpected importing side effects. A second approach is to pass the session instance explicitly as a function argument, but this makes the code ugly since you then need to propagate that argument all the way up to every caller of that function.

Other times you might want to be able to glue together and run one after the other different code segments, where each segment initializes its own sparkly session, despite the sessions being identical. This situation could occur when you are doing investigative work in a notebook.

In both cases, `SparklySession.get_or_create` is the answer, as it solves the problems mentioned above while keeping your code clean and tidy.

For example: You want to use a read function within a transformation.

```
from sparkly import SparklySession


class MySession(SparklySession):
    pass

def my_awesome_transformation():
    df = read_dataset('parquet:s3://path/to/my/data')
    df2 = read_dataset('parquet:s3://path/to/my/other/data')
    # do something with df and df2...

def read_dataset(url):
    spark = MySession.get_or_create()
    return spark.read_ext.by_url(url)
```

API documentation

class `sparkly.session.`**`SparklySession`**(*additional_options=None*)

Wrapper around SparkSession to simplify definition of options, packages, JARs and UDFs.

Example:

```
from pyspark.sql.types import IntegerType
import sparkly


class MySession(sparkly.SparklySession):
    options = {'spark.sql.shuffle.partitions': '2000'}
    repositories = ['http://packages.confluent.io/maven/']
    packages = ['com.databricks:spark-csv_2.10:1.4.0']
    jars = ['../path/to/brickhouse-0.7.1.jar']
    udfs = {
        'collect_max': 'brickhouse.udf.collect.CollectMaxUDAF',
        'my_python_udf': (lambda x: len(x), IntegerType()),
    }


spark = MySession()
spark.read_ext.cassandra(...)

# Alternatively
spark = MySession.get_or_create()
spark.read_ext.cassandra(...)
```

options
dict[str,str] – Configuration options that are passed to spark-submit. See the list of possible options. Note that any options set already through PYSPARK_SUBMIT_ARGS will override these.

repositories
list[str] – List of additional maven repositories for package lookup.

packages
list[str] – Spark packages that should be installed. See https://spark-packages.org/

jars
list[str] – Full paths to jar files that we want to include to the session. E.g. a JDBC connector or a library with UDF functions.

udfs
dict[str,str|typing.Callable] – Register UDF functions within the session. Key - a name of the function, Value - either a class name imported from a JAR file

or a tuple with python function and its return type.

classmethod get_or_create()
Access instantiated sparkly session.

If sparkly session has already been instantiated, return that instance; if not, then instantiate one and return it. Useful for lazy access to the session. Not thread-safe.

Returns SparklySession (or subclass).

has_jar(*jar_name*)
Check if the jar is available in the session.

Parameters **jar_name** (*str*) – E.g. "mysql-connector-java".

Returns bool

has_package(*package_prefix*)
Check if the package is available in the session.

Parameters **package_prefix** (*str*) – E.g. "org.elasticsearch:elasticsearch-spark".

Returns bool

classmethod `stop`()
Stop instantiated sparkly session.

CHAPTER 2

Read/write utilities for DataFrames

Sparkly isn't trying to replace any of existing storage connectors. The goal is to provide a simplified and consistent api across a wide array of storage connectors. We also added the way to work with *abstract data sources*, so you can keep your code agnostic to the storages you use.

Cassandra

Sparkly relies on the official spark cassandra connector and was successfully tested in production using version *2.0.0-M2*.

Package	https://spark-packages.org/package/datastax/spark-cassandra-connector
Configuration	https://github.com/datastax/spark-cassandra-connector/blob/v2.0.0-M2/doc/reference.md

```
from sparkly import SparklySession


class MySession(SparklySession):
    # Feel free to play with other versions
    packages = ['datastax:spark-cassandra-connector:2.0.0-M2-s_2.11']

spark = MySession()

# To read data
df = spark.read_ext.cassandra('localhost', 'my_keyspace', 'my_table')
# To write data
df.write_ext.cassandra('localhost', 'my_keyspace', 'my_table')
```

Elastic

Sparkly relies on the official elastic spark connector and was successfully tested in production using version *5.1.1*.

Package	https://spark-packages.org/package/elastic/elasticsearch-hadoop
Configuration	https://www.elastic.co/guide/en/elasticsearch/hadoop/5.1/configuration.html

```
from sparkly import SparklySession


class MySession(SparklySession):
    # Feel free to play with other versions
```

```
    packages = ['org.elasticsearch:elasticsearch-spark-20_2.11:5.1.1']

spark = MySession()

# To read data
df = spark.read_ext.elastic('localhost', 'my_index', 'my_type', query='?q=awesomeness
↪')
# To write data
df.write_ext.elastic('localhost', 'my_index', 'my_type')
```

Kafka

Sparkly's reader and writer for Kafka are built on top of the official spark package for Kafka and python library kafka-python . The first one allows us to read data efficiently, the second covers a lack of writing functionality in the official distribution.

Package	https://mvnrepository.com/artifact/org.apache.spark/spark-streaming-kafka-0-8_2.11/2.1.0
Configuration	http://spark.apache.org/docs/2.1.0/streaming-kafka-0-8-integration.html

Note:

- To interact with Kafka, `sparkly` needs the `kafka-python` library. You can get it via: ` `pip install sparkly[kafka]` `
- Sparkly was tested in production using Apache Kafka **0.10.x**.

```
import json

from sparkly import SparklySession


class MySession(SparklySession):
    packages = [
        'org.apache.spark:spark-streaming-kafka-0-8_2.11:2.1.0',
    ]

spark = MySession()

# To read JSON messaged from Kafka into a dataframe:

#   1. Define a schema of the messages you read.
df_schema = StructType([
    StructField('key', StructType([
        StructField('id', StringType(), True)
    ])),
    StructField('value', StructType([
        StructField('name', StringType(), True),
        StructField('surname', StringType(), True),
    ]))
])

#   2. Specify the schema as a reader parameter.
df = hc.read_ext.kafka(
    'kafka.host',
    topic='my.topic',
```

```
    key_deserializer=lambda item: json.loads(item.decode('utf-8')),
    value_deserializer=lambda item: json.loads(item.decode('utf-8')),
    schema=df_schema,
)

# To write a dataframe to Kafka in JSON format:
df.write_ext.kafka(
    'kafka.host',
    topic='my.topic',
    key_serializer=lambda item: json.dumps(item).encode('utf-8'),
    value_serializer=lambda item: json.dumps(item).encode('utf-8'),
)
```

MySQL

Basically, it's just a high level api on top of the native jdbc reader and jdbc writer.

Jars	https://mvnrepository.com/artifact/mysql/mysql-connector-java
Configuration	https://dev.mysql.com/doc/connector-j/5.1/en/connector-j-reference-configuration-properties.html

```
from sparkly import SparklySession
from sparkly.utils import absolute_path


class MySession(SparklySession):
    # Feel free to play with other versions.
    packages = ['mysql:mysql-connector-java:5.1.39']


spark = MySession()

# To read data
df = spark.read_ext.mysql('localhost', 'my_database', 'my_table',
                          options={'user': 'root', 'password': 'root'})
# To write data
df.write_ext.mysql('localhost', 'my_database', 'my_table', options={
    'user': 'root',
    'password': 'root',
    'rewriteBatchedStatements': 'true',  # improves write throughput dramatically
})
```

Redis

Sparkly provides a writer for Redis that is built on top of the official redis python library redis-py . It is currently capable of exporting your DataFrame as a JSON blob per row or group of rows.

Note:

- To interact with Redis, `sparkly` needs the `redis` library. You can get it via: `pip install sparkly[redis]`

```
import json

from sparkly import SparklySession


spark = SparklySession()

# Write JSON.gz data indexed by col1.col2 that will expire in a day
df.write_ext.redis(
    host='localhost',
    port=6379,
    key_by=['col1', 'col2'],
    exclude_key_columns=True,
    expire=24 * 60 * 60,
    compression='gzip',
)
```

Universal reader/writer

The *DataFrame* abstraction is really powerful when it comes to transformations. You can shape your data from various storages using exactly the same api. For instance, you can join data from Cassandra with data from Elasticsearch and write the result to MySQL.

The only problem - you have to explicitly define sources (or destinations) in order to create (or export) a *DataFrame*. But the source/destination of data doesn't really change the logic of transformations (if the schema is preserved). To solve the problem, we decided to add the universal api to read/write *DataFrames*:

```
from sparkly import SparklyContext

class MyContext(SparklyContext):
    packages = [
        'datastax:spark-cassandra-connector:1.6.1-s_2.10',
        'com.databricks:spark-csv_2.10:1.4.0',
        'org.elasticsearch:elasticsearch-spark_2.10:2.3.0',
    ]

hc = MyContext()

# To read data
df = hc.read_ext.by_url('cassandra://localhost/my_keyspace/my_table?consistency=ONE')
df = hc.read_ext.by_url('csv:s3://my-bucket/my-data?header=true')
df = hc.read_ext.by_url('elastic://localhost/my_index/my_type?q=awesomeness')
df = hc.read_ext.by_url('parquet:hdfs://my.name.node/path/on/hdfs')

# To write data
df.write_ext.by_url('cassandra://localhost/my_keyspace/my_table?consistency=QUORUM&
↪parallelism=8')
df.write_ext.by_url('csv:hdfs://my.name.node/path/on/hdfs')
df.write_ext.by_url('elastic://localhost/my_index/my_type?parallelism=4')
df.write_ext.by_url('parquet:s3://my-bucket/my-data?header=false')
```

Controlling the load

From the official documentation:

> Don't create too many partitions in parallel on a large cluster; otherwise Spark might crash your external database systems.
>
> link: <https://spark.apache.org/docs/2.0.1/api/java/org/apache/spark/sql/DataFrameReader.html>

It's a very good advice, but in practice it's hard to track the number of partitions. For instance, if you write a result of a join operation to database the number of splits might be changed implicitly via *spark.sql.shuffle.partitions*.

To prevent us from shooting to the foot, we decided to add *parallelism* option for all our readers and writers. The option is designed to control a load on a source we write to / read from. It's especially useful when you are working with data storages like Cassandra, MySQL or Elastic. However, the implementation of the throttling has some drawbacks and you should be aware of them.

The way we implemented it is pretty simple: we use *coalesce* on a dataframe to reduce an amount of tasks that will be executed in parallel. Let's say you have a dataframe with 1000 splits and you want to write no more than 10 task in parallel. In such case *coalesce* will create a dataframe that has 10 splits with 100 original tasks in each. An outcome of this: if any of these 100 tasks fails, we have to retry the whole pack in 100 tasks.

Read more about coalesce

Reader API documentation

class `sparkly.reader.`**`SparklyReader`**(*spark*)

A set of tools to create DataFrames from the external storages.

Note: This is a private class to the library. You should not use it directly. The instance of the class is available under *SparklyContext* via *read_ext* attribute.

`by_url`(*url*)

Create a dataframe using *url*.

The main idea behind the method is to unify data access interface for different formats and locations. A generic schema looks like:

```
format:[protocol:]//host[:port][/location][?configuration]
```

Supported formats:

- CSV `csv://`
- Cassandra `cassandra://`
- Elastic `elastic://`
- MySQL `mysql://`
- Parquet `parquet://`
- Hive Metastore table `table://`

Query string arguments are passed as parameters to the relevant reader.

For instance, the next data source URL:

```
cassandra://localhost:9042/my_keyspace/my_table?consistency=ONE
    &parallelism=3&spark.cassandra.connection.compression=LZ4
```

Is an equivalent for:

```
hc.read_ext.cassandra(
    host='localhost',
    port=9042,
    keyspace='my_keyspace',
    table='my_table',
    consistency='ONE',
    parallelism=3,
    options={'spark.cassandra.connection.compression': 'LZ4'},
)
```

More examples:

```
table://table_name
csv:s3://some-bucket/some_directory?header=true
csv://path/on/local/file/system?header=false
parquet:s3://some-bucket/some_directory
elastic://elasticsearch.host/es_index/es_type?parallelism=8
cassandra://cassandra.host/keyspace/table?consistency=QUORUM
mysql://mysql.host/database/table
```

Parameters **url** (*str*) – Data source URL.

Returns pyspark.sql.DataFrame

cassandra(*host*, *keyspace*, *table*, *consistency=None*, *port=None*, *parallelism=None*, *options=None*)

Create a dataframe from a Cassandra table.

Parameters

- **host** (*str*) – Cassandra server host.
- **keyspace** (*str*) –
- **table** (*str*) – Cassandra table to read from.
- **consistency** (*str*) – Read consistency level: `ONE`, `QUORUM`, `ALL`, etc.
- **port** (*int|None*) – Cassandra server port.
- **parallelism** (*int|None*) – The max number of parallel tasks that could be executed during the read stage (see *Controlling the load*).
- **options** (*dict[str,str]|None*) – Additional options for *org.apache.spark.sql.cassandra* format (see configuration for *Cassandra*).

Returns pyspark.sql.DataFrame

elastic(*host*, *es_index*, *es_type*, *query=''*, *fields=None*, *port=None*, *parallelism=None*, *options=None*)

Create a dataframe from an ElasticSearch index.

Parameters

- **host** (*str*) – Elastic server host.
- **es_index** (*str*) – Elastic index.

- **es_type** (*str*) – Elastic type.
- **query** (*str*) – Pre-filter es documents, e.g. '?q=views:>10'.
- **fields** (*list[str]|None*) – Select only specified fields.
- **port** (*int|None*) –
- **parallelism** (*int|None*) – The max number of parallel tasks that could be executed during the read stage (see *Controlling the load*).
- **options** (*dict[str,str]*) – Additional options for *org.elasticsearch.spark.sql* format (see configuration for *Elastic*).

Returns pyspark.sql.DataFrame

kafka(*host*, *topic*, *offset_ranges=None*, *key_deserializer=None*, *value_deserializer=None*, *schema=None*, *port=9092*, *parallelism=None*, *options=None*)

Creates dataframe from specified set of messages from Kafka topic.

Defining ranges:

- If *offset_ranges* is specified it defines which specific range to read.
- If *offset_ranges* is omitted it will auto-discover it's partitions.

The *schema* parameter, if specified, should contain two top level fields: *key* and *value*.

Parameters *key_deserializer* and *value_deserializer* are callables which get bytes as input and should return python structures as output.

Parameters

- **host** (*str*) – Kafka host.
- **topic** (*str|None*) – Kafka topic to read from.
- **offset_ranges** (*list[(int, int, int)*) – List of partition ranges [(partition, start_offset, end_offset)].
- **key_deserializer** (*function*) – Function used to deserialize the key.
- **value_deserializer** (*function*) – Function used to deserialize the value.
- **schema** (*pyspark.sql.types.StructType*) – Schema to apply to create a Dataframe.
- **port** (*int*) – Kafka port.
- **parallelism** (*int|None*) – The max number of parallel tasks that could be executed during the read stage (see *Controlling the load*).
- **options** (*dict|None*) – Additional kafka parameters, see KafkaUtils.createRDD docs.

Returns pyspark.sql.DataFrame

Raises `InvalidArgumentError`

mysql(*host*, *database*, *table*, *port=None*, *parallelism=None*, *options=None*)

Create a dataframe from a MySQL table.

Options should include user and password.

Parameters

- **host** (*str*) – MySQL server address.
- **database** (*str*) – Database to connect to.

- **table** (*str*) – Table to read rows from.
- **port** (*int|None*) – MySQL server port.
- **parallelism** (*int|None*) – The max number of parallel tasks that could be executed during the read stage (see *Controlling the load*).
- **options** (*dict[str,str]|None*) – Additional options for JDBC reader (see configuration for *MySQL*).

Returns pyspark.sql.DataFrame

Writer API documentation

class sparkly.writer.**SparklyWriter**(*df*)

A set of tools to write DataFrames to external storages.

Note: We don't expect you to be using the class directly. The instance of the class is available under *DataFrame* via *write_ext* attribute.

by_url(*url*)

Write a dataframe to a destination specified by *url*.

The main idea behind the method is to unify data export interface for different formats and locations. A generic schema looks like:

```
format:[protocol:]//host[:port][/location][?configuration]
```

Supported formats:

•CSV `csv://`

•Cassandra `cassandra://`

•Elastic `elastic://`

•MySQL `mysql://`

•Parquet `parquet://`

•Redis `redis://` or `rediss://`

Query string arguments are passed as parameters to the relevant writer.

For instance, the next data export URL:

```
elastic://localhost:9200/my_index/my_type?&parallelism=3&mode=overwrite
    &es.write.operation=upsert
```

Is an equivalent for:

```
hc.read_ext.elastic(
    host='localhost',
    port=9200,
    es_index='my_index',
    es_type='my_type',
    parallelism=3,
    mode='overwrite',
```

```
    options={'es.write.operation': 'upsert'},
)
```

More examples:

```
csv:s3://some-s3-bucket/some-s3-key?partitionBy=date,platform
cassandra://cassandra.host/keyspace/table?consistency=ONE&mode=append
parquet:///var/log/?partitionBy=date
elastic://elastic.host/es_index/es_type
mysql://mysql.host/database/table
redis://redis.host/db?keyBy=id
```

Parameters **url** (*str*) – Destination URL.

cassandra(*host*, *keyspace*, *table*, *consistency=None*, *port=None*, *mode=None*, *parallelism=None*, *options=None*)

Write a dataframe to a Cassandra table.

Parameters

- **host** (*str*) – Cassandra server host.
- **keyspace** (*str*) – Cassandra keyspace to write to.
- **table** (*str*) – Cassandra table to write to.
- **consistency** (*str|None*) – Write consistency level: `ONE`, `QUORUM`, `ALL`, etc.
- **port** (*int|None*) – Cassandra server port.
- **mode** (*str|None*) – Spark save mode, http://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes
- **parallelism** (*int|None*) – The max number of parallel tasks that could be executed during the write stage (see *Controlling the load*).
- **options** (*dict[str, str]*) – Additional options to *org.apache.spark.sql.cassandra* format (see configuration for *Cassandra*).

elastic(*host*, *es_index*, *es_type*, *port=None*, *mode=None*, *parallelism=None*, *options=None*)

Write a dataframe into an ElasticSearch index.

Parameters

- **host** (*str*) – Elastic server host.
- **es_index** (*str*) – Elastic index.
- **es_type** (*str*) – Elastic type.
- **port** (*int|None*) –
- **mode** (*str|None*) – Spark save mode, http://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes
- **parallelism** (*int|None*) – The max number of parallel tasks that could be executed during the write stage (see *Controlling the load*).
- **options** (*dict[str, str]*) – Additional options to *org.elasticsearch.spark.sql* format (see configuration for *Elastic*).

kafka(*host*, *topic*, *key_serializer*, *value_serializer*, *port=9092*, *parallelism=None*, *options=None*)

Writes dataframe to kafka topic.

The schema of the dataframe should conform the pattern:

```
>>> StructType([
...     StructField('key', ...),
...     StructField('value', ...),
... ])
```

Parameters *key_serializer* and *value_serializer* are callables which get's python structure as input and should return bytes of encoded data as output.

Parameters

- **host** (*str*) – Kafka host.
- **topic** (*str*) – Topic to write to.
- **key_serializer** (*function*) – Function to serialize key.
- **value_serializer** (*function*) – Function to serialize value.
- **port** (*int*) – Kafka port.
- **parallelism** (*int|None*) – The max number of parallel tasks that could be executed during the write stage (see *Controlling the load*).
- **options** (*dict|None*) – Additional options.

mysql (*host*, *database*, *table*, *port=None*, *mode=None*, *parallelism=None*, *options=None*)

Write a dataframe to a MySQL table.

Options should include user and password.

Parameters

- **host** (*str*) – MySQL server address.
- **database** (*str*) – Database to connect to.
- **table** (*str*) – Table to read rows from.
- **mode** (*str|None*) – Spark save mode, http://spark.apache.org/docs/latest/sql-programming-guide.html#save-modes
- **parallelism** (*int|None*) – The max number of parallel tasks that could be executed during the write stage (see *Controlling the load*).
- **options** (*dict*) – Additional options for JDBC writer (see configuration for *MySQL*).

redis (*key_by*, *key_prefix=None*, *key_delimiter='.'*, *group_by_key=False*, *exclude_key_columns=False*, *exclude_null_fields=False*, *expire=None*, *compression=None*, *max_pipeline_size=100*, *parallelism=None*, *mode='overwrite'*, *host=None*, *port=6379*, *db=0*, *redis_client_init=None*)

Write a dataframe to Redis as JSON.

Parameters

- **key_by** (*list[str]*) – Column names that form the redis key for each row. The columns are concatenated in the order they appear.
- **key_prefix** (*str|None*) – Common prefix to add to all keys from this DataFrame. Useful to namespace DataFrame exports.
- **key_delimiter** (*str|.*) – Characters to delimit different columns while forming the key.

- **group_by_key** (*bool|False*) – If set, group rows that share the same redis key together in an array before exporting. By default if multiple rows share the same redis key, one will overwrite the other.
- **exclude_key_columns** (*bool|False*) – If set, exclude all columns that comprise the key from the value being exported to redis.
- **exclude_null_fields** (*bool|False*) – If set, exclude all fields of a row that are null from the value being exported to redis.
- **expire** (*int|None*) – Expire the keys after this number of seconds.
- **compression** (*str|None*) – Compress each Redis entry using this protocol. Currently bzip2, gzip and zlib are supported.
- **max_pipeline_size** (*int|100*) – Number of writes to pipeline.
- **parallelism** (*int|None*) – The max number of parallel tasks that could be executed during the write stage (see *Controlling the load*).
- **mode** (*str|overwrite*) –
  - `'append'`: Append to existing data on a key by key basis.
  - `'ignore'`: Silently ignore if data already exists.
  - `'overwrite'`: Flush all existing data before writing.
- **host** (*str|None*) – Redis host. Either this or redis_client_init must be provided. See below.
- **port** (*int|6379*) – Port redis is listening to.
- **db** (*int|0*) – Redis db to write to.
- **redis_client_init** (*callable|None*) – Bypass internal redis client initialization by passing a function that does it, no arguments required. For example this could be `redis.StrictRedis.from_url` with the appropriate url and `kwargs` already set through `functools.partial`. This option overrides other conflicting arguments.

Raises

- `NotImplementedError` – if *redis-py* is not installed.
- `AssertionError` – if nor host neither `redis_client_init` are provided.
- `ValueError` – if any of the `expire`, `compression`, `max_pipeline_size` or `mode` options assume an invalid value.

`sparkly.writer.`**`attach_writer_to_dataframe`**`()`

A tiny amount of magic to attach write extensions.

CHAPTER 3

Hive Metastore Utils

About Hive Metastore

The Hive Metastore is a database with metadata for Hive tables.

To configure `SparklySession to work with external Hive Metastore, you need to set `hive.metastore.uris` option. You can do this via `hive-site.xml` file in spark config ($SPARK_HOME/conf/hive-site.xml):

```
<property>
  <name>hive.metastore.uris</name>
  <value>thrift://<n.n.n.n>:9083</value>
  <description>IP address (or fully-qualified domain name) and port of the metastore␣
↪host</description>
</property>
```

or set it dynamically via `SparklySession` options:

```
class MySession(SparklySession):
    options = {
        'hive.metastore.uris': 'thrift://<n.n.n.n>:9083',
    }
```

Tables management

Why: you need to check if tables exist, rename them, drop them, or even overwrite existing aliases in your catalog.

```
from sparkly import SparklySession


spark = SparklySession()

assert spark.catalog_ext.has_table('my_table') in {True, False}
spark.catalog_ext.rename_table('my_table', 'my_new_table')
spark.catalog_ext.create_table('my_new_table', path='s3://my/parquet/data', source=
↪'parquet', mode='overwrite')
spark.catalog_ext.drop_table('my_new_table')
```

Table properties management

Why: sometimes you want to assign custom attributes for your table, e.g. creation time, last update, purpose, data source. The only way to interact with table properties in spark - use raw SQL queries. We implemented a more convenient interface to make your code cleaner.

```
from sparkly import SparklySession


spark = SparklySession()
spark.catalog_ext.set_table_property('my_table', 'foo', 'bar')
assert spark.catalog_ext.get_table_property('my_table', 'foo') == 'bar'
assert spark.catalog_ext.get_table_properties('my_table') == {'foo': 'bar'}
```

Note properties are stored as strings. In case if you need other types, consider using a serialisation format, e.g. JSON.

Using non-default database

Why to split your warehouse into logical groups (for example by system components). In all catalog_ext.* methods you can specify full table names <db-name>.<table-name> and it should operate properly

```
from time import time
from sparkly import SparklySession

spark = SparklySession()

if spark.catalog_ext.has_database('my_database'):
    self.catalog_ext.rename_table(
        'my_database.my_badly_named_table',
        'new_shiny_name',
    )
    self.catalog_ext.set_table_property(
        'my_database.new_shiny_name',
        'last_update_at',
        time(),
    )
```

Note be careful using 'USE' statements like: spark.sql('USE my_database'), it's stateful and may lead to weird errors, if code assumes correct current database.

API documentation

class `sparkly.catalog.`**`SparklyCatalog`**(*spark*)

A set of tools to interact with HiveMetastore.

`create_table`(*table_name*, *path=None*, *source=None*, *schema=None*, ***options*)

Create table in the metastore.

Extend `SparkSession.Catalog.createExternalTable` by accepting a `mode='overwrite'` option which creates the table even if a table with the same name already exists. All other args are exactly the same.

Note: If the table exists, create two unique names, one for the new and one for the old instance, then try to swap names and drop the "old" instance. If any step fails, the metastore might be currently left at a broken state.

Parameters **mode** (*str*) – if set to `'overwrite'`, drop any table of the same name from the metastore. Given as a kwarg. Default is error out if table already exists.

Returns DataFrame associated with the created table.

Return type pyspark.sql.DataFrame

drop_table(*table_name*, *checkfirst=True*)
Drop table from the metastore.

Note: Follow the official documentation to understand *DROP TABLE* semantic. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL #LanguageManualDDL-DropTable

Parameters

- **table_name** (*str*) – A table name.
- **checkfirst** (*bool*) – Only issue DROPs for tables that are presented in the database.

get_table_properties(*table_name*)
Get table properties from the metastore.

Parameters **table_name** (*str*) – A table name.

Returns Key/value for properties.

Return type dict[str,str]

get_table_property(*table_name*, *property_name*, *to_type=None*)
Get table property value from the metastore.

Parameters

- **table_name** (*str*) – A table name. Might contain a db name. E.g. "my_table" or "default.my_table".
- **property_name** (*str*) – A property name to read value for.
- **to_type** (*function*) – Cast value to the given type. E.g. *int* or *float*.

Returns Any

has_database(*db_name*)
Check if database exists in the metastore.

Parameters **db_name** (*str*) – Database name.

Returns bool

has_table(*table_name*)
Check if table is available in the metastore.

Parameters **table_name** (*str*) – A table name.

Returns bool

rename_table(*old_table_name*, *new_table_name*)
Rename table in the metastore.

> **Note:** Follow the official documentation to understand *ALTER TABLE* semantic. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL #LanguageManualDDL-RenameTable

Parameters

- **old_table_name** (*str*) – The current table name.
- **new_table_name** (*str*) – An expected table name.

set_table_property(*table_name*, *property_name*, *value*)
Set value for table property.

Parameters

- **table_name** (*str*) – A table name.
- **property_name** (*str*) – A property name to set value for.
- **value** (*Any*) – Will be automatically casted to string.

sparkly.catalog.**get_db_name**(*table_name*)
Get database name from full table name.

sparkly.catalog.**get_table_name**(*table_name*)
Get table name from full table name.

CHAPTER 4

Testing Utils

Base TestCases

There are two main test cases available in Sparkly:

- `SparklyTest` creates a new session for each test case.
- `SparklyGlobalSessionTest` uses a single sparkly session for all test cases to boost performance.

```
from pyspark.sql import types as T

from sparkly import SparklySession
from sparkly.testing import SparklyTest, SparklyGlobalSessionTest


class MyTestCase(SparklyTest):
    session = SparklySession

    def test(self):
        df = self.spark.read_ext.by_url(...)

        # Compare all fields
        self.assertRowsEqual(
            df.collect(),
            [
                T.Row(col1='row1', col2=1),
                T.Row(col1='row2', col2=2),
            ],
        )

...

class MyTestWithReusableSession(SparklyGlobalSessionTest):
    context = SparklySession

    def test(self):
        df = self.spark.read_ext.by_url(...)

...
```

DataFrame Assertions

Asserting that the dataframe produced by your transformation is equal to some expected output can be unnecessarily complicated at times. Common issues include:

- Ignoring the order in which elements appear in an array. This could be particularly useful when that array is generated as part of a `groupBy` aggregation, and you only care about all elements being part of the end result, rather than the order in which Spark encountered them.
- Comparing floats that could be arbitrarily nested in complicated datatypes within a given tolerance; exact matching is either fragile or impossible.
- Ignoring whether a field of a complex datatype is nullable. Spark infers this based on the applied transformations, but it is oftentimes inaccurate. As a result, assertions on complex data types might fail, even though in theory they shouldn't have.
- Having rows with different field names compare equal if the values match in alphabetical order of the names (see unit tests for example).
- Unhelpful diffs in case of mismatches.

Sparkly addresses these issues by providing `assertRowsEqual`:

```
from pyspark.sql import types as T

from sparkly import SparklySession
from sparkly.test import SparklyTest


def my_transformation(spark):
    return spark.createDataFrame(
        data=[
            ('row1', {'field': 'value_1'}, [1.1, 2.2, 3.3]),
            ('row2', {'field': 'value_2'}, [4.1, 5.2, 6.3]),
        ],
        schema=T.StructType([
            T.StructField('id', T.StringType()),
            T.StructField(
                'st',
                T.StructType([
                    T.StructField('field', T.StringType()),
                ]),
            ),
            T.StructField('ar', T.ArrayType(T.FloatType())),
        ]),
    )


class MyTestCase(SparklyTest):
    session = SparklySession

    def test(self):
        df = my_transformation(self.spark)

        self.assertRowsEqual(
            df.collect(),
            [
                T.Row(id='row2', st=T.Row(field='value_2'), ar=[6.0, 5.0, 4.0]),
                T.Row(id='row1', st=T.Row(field='value_1'), ar=[2.0, 3.0, 1.0]),
```

```
            ],
            atol=0.5,
        )
```

Instant Iterative Development

The slowest part in Spark integration testing is context initialisation. `SparklyGlobalSessionTest` allows you to keep the same instance of spark context between different test cases, but it still kills the context at the end. It's especially annoying if you work in TDD fashion. On each run you have to wait 25-30 seconds till a new context is ready. We added a tool to preserve spark context between multiple test runs.

Note: In case if you change `SparklySession` definition (new options, jars or packages) you have to refresh the context via `sparkly-testing refresh`. However, you don't need to refresh context if `udfs` are changed.

Fixtures

"Fixture" is a term borrowed from Django framework. Fixtures load data to a database before the test execution.

There are several storages supported in Sparkly:

- Elastic
- Cassandra (requires `cassandra-driver`)
- Mysql (requires `PyMySql`)
- Kafka (requires `kafka-python`)

```
from sparkly.test import MysqlFixture, SparklyTest


class MyTestCase(SparklyTest):
    ...
    fixtures = [
        MysqlFixture('mysql.host',
                     'user',
                     'password',
                     '/path/to/setup_data.sql',
                     '/path/to/remove_data.sql')
    ]
    ...
```

class `sparkly.testing.`**`CassandraFixture`**(*host*, *setup_file*, *teardown_file*)

Fixture to load data into cassandra.

Notes

•Depends on cassandra-driver.

Examples

```
>>> class MyTestCase(SparklyTest):
...     fixtures = [
...         CassandraFixture(
...             'cassandra.host',
...             absolute_path(__file__, 'resources', 'setup.cql'),
...             absolute_path(__file__, 'resources', 'teardown.cql'),
...         )
...     ]
...
```

```
>>> class MyTestCase(SparklyTest):
...     data = CassandraFixture(
...         'cassandra.host',
...         absolute_path(__file__, 'resources', 'setup.cql'),
...         absolute_path(__file__, 'resources', 'teardown.cql'),
...     )
...     def setUp(self):
...         data.setup_data()
...     def tearDown(self):
...         data.teardown_data()
...
```

```
>>> def test():
...     fixture = CassandraFixture(...)
...     with fixture:
...        test_stuff()
...
```

class sparkly.testing.**ElasticFixture**(*host*, *es_index*, *es_type*, *mapping=None*, *data=None*, *port=None*)

Fixture for elastic integration tests.

Examples

```
>>> class MyTestCase(SparklyTest):
...     fixtures = [
...         ElasticFixture(
...             'elastic.host',
...             'es_index',
...             'es_type',
...             '/path/to/mapping.json',
...             '/path/to/data.json',
...         )
...     ]
...
```

class sparkly.testing.**Fixture**

Base class for fixtures.

Fixture is a term borrowed from Django tests, it's data loaded into database for integration testing.

setup_data()

Method called to load data into database.

teardown_data()
Method called to remove data from database which was loaded by *setup_data*.

class sparkly.testing.**KafkaFixture**(*host*, *port=9092*, *topic=None*, *key_serializer=None*, *value_serializer=None*, *data=None*)
Fixture for kafka integration tests.

Notes

•depends on kafka-python lib.

•json file should contain array of dicts: [{'key': ..., 'value': ...}]

Examples

```
>>> class MyTestCase(SparklySession):
...     fixtures = [
...         KafkaFixture(
...             'kafka.host', 'topic',
...             key_serializer=..., value_serializer=...,
...             data='/path/to/data.json',
...         )
...     ]
```

class sparkly.testing.**KafkaWatcher**(*spark*, *df_schema*, *key_deserializer*, *value_deserializer*, *host*, *topic*, *port=9092*)
Context manager that tracks Kafka data published to a topic

Provides access to the new items that were written to a kafka topic by code running within this context.

NOTE: This is mainly useful in integration test cases and may produce unexpected results in production environments, since there are no guarantees about who else may be publishing to a kafka topic.

Usage: my_deserializer = lambda item: json.loads(item.decode('utf-8')) kafka_watcher = KafkaWatcher(

my_sparkly_session, expected_output_dataframe_schema, my_deserializer, my_deserializer, 'my.kafkaserver.net', 'my_kafka_topic',

) with kafka_watcher:

do stuff that publishes messages to 'my_kafka_topic'

self.assertEqual(kafka_watcher.count, expected_number_of_new_messages) self.assertDataFrameEqual(kafka_watcher.df, expected_df)

class sparkly.testing.**MysqlFixture**(*host*, *user*, *password=None*, *data=None*, *teardown=None*)
Fixture for mysql integration tests.

Notes

•depends on PyMySql lib.

Examples

```
>>> class MyTestCase(SparklyTest):
...       fixtures = [
...           MysqlFixture('mysql.host', 'user', 'password', '/path/to/data.sql')
...       ]
...       def test(self):
...           pass
...
```

class sparkly.testing.**SparklyGlobalSessionTest**(*methodName='runTest'*)
Base test case that keeps a single instance for the given session class across all tests.

Integration tests are slow, especially when you have to start/stop Spark context for each test case. This class allows you to reuse Spark session across multiple test cases.

class sparkly.testing.**SparklyTest**(*methodName='runTest'*)
Base test for spark scrip tests.

Initialize and shut down Session specified in *session* attribute.

Example

```
>>> from pyspark.sql import types as T
>>> class MyTestCase(SparklyTest):
...     def test(self):
...         self.assertRowsEqual(
...             self.spark.sql('SELECT 1 as one').collect(),
...             [T.Row(one=1)],
...         )
```

assertDataFrameEqual(*actual_df*, *expected_data*, *fields=None*, *ordered=False*)
Ensure that DataFrame has the right data inside.

`assertDataFrameEqual` is being deprecated. Please use `assertRowsEqual` instead.

Parameters

- **actual_df** (*pyspark.sql.DataFrame|list[pyspark.sql.Row]*) – Dataframe to test data in.
- **expected_data** (*list[dict]*) – Expected dataframe rows defined as dicts.
- **fields** (*list[str]*) – Compare only certain fields.
- **ordered** (*bool*) – Does order of rows matter?

assertRowsEqual(*first*, *second*, *msg=None*, *ignore_order=True*, *ignore_order_depth=None*, *atol=0*, *rtol=1e-07*, *equal_nan=True*, *ignore_nullability=True*)
Assert equal on steroids.

Extend this classic function signature to work better with comparisons involving rows, datatypes, dictionaries, lists and floats by:

•ignoring the order of lists and datatypes recursively,

•comparing floats within a given tolerance,

•assuming NaNs are equal,

•ignoring the nullability requirements of datatypes (since Spark can be inaccurate when inferring it),

•providing better diffs for rows and datatypes.

Float comparisons are inspired by NumPy's `assert_allclose`. The main formula used is `| float1 -float2 | <= atol + rtol * float2`.

Parameters

- **first** – see `unittest.TestCase.assertEqual`.
- **second** – see `unittest.TestCase.assertEqual`.
- **msg** – see `unittest.TestCase.assertEqual`.
- **ignore_order** (*bool|True*) – ignore the order in lists and datatypes (rows, dicts are inherently orderless).
- **ignore_order_depth** (*int|None*) – if ignore_order is true, do ignore order up to this level of nested lists or datatypes (exclusive). Setting this to 0 or None means ignore order infinitely, 1 means ignore order only at the top level, 2 will ignore order within lists of lists and so on. Default is ignore order arbitrarily deep.
- **atol** (*int, float|0*) – Absolute tolerance in float comparisons.
- **rtol** (*int, float|1e-07*) – Relative tolerance in float comparisons.
- **equal_nan** (*bool|True*) – If set, NaNs will compare equal.
- **ignore_nullability** (*bool|True*) – If set, ignore all nullability fields in dataTypes. This includes `containsNull` in arrays, `valueContainsNull` in maps and `nullable` in struct fields.

Returns None iff the two objects are equal.

Raises AssertionError: iff the two objects are not equal. See `unittest.TestCase.assertEqual` for details.

session
alias of `SparklySession`

CHAPTER 5

Column and DataFrame Functions

A counterpart of pyspark.sql.functions providing useful shortcuts:

- a cleaner alternative to chaining together multiple when/otherwise statements.
- an easy way to join multiple dataframes at once and disambiguate fields with the same name.

API documentation

`sparkly.functions.`**`multijoin`**(*dfs*, *on=None*, *how=None*, *coalesce=None*)

Join multiple dataframes.

Parameters

- **`dfs`** (*`list[pyspark.sql.DataFrame]`*) –
- **`on`** – same as `pyspark.sql.DataFrame.join`.
- **`how`** – same as `pyspark.sql.DataFrame.join`.
- **`coalesce`** (*`list[str]`*) – column names to disambiguate by coalescing across the input dataframes. A column must be of the same type across all dataframes that define it; if different types appear coalesce will do a best-effort attempt in merging them. The selected value is the first non-null one in order of appearance of the dataframes in the input list. Default is None - don't coalesce any ambiguous columns.

Returns pyspark.sql.DataFrame or None if provided dataframe list is empty.

Example

Assume we have two DataFrames, the first is `first = [{'id': 1,'value': None},{'id': 2,'value': 2}]` and the second is `second = [{'id': 1,'value': 1},{'id': 2,'value': 22}]`

Then collecting the `DataFrame` produced by

```
multijoin([first,second],on='id',how='inner',coalesce=['value'])
```

yields `[{'id': 1,'value': 1},{'id': 2,'value': 2}]`.

`sparkly.functions.`**`switch_case`**(*switch*, *case=None*, *default=None*, *operand=<built-in function eq>*, ***additional_cases*)

Switch/case style column generation.

Parameters

- **switch** (*str, pyspark.sql.Column*) – column to "switch" on; its values are going to be compared against defined cases.
- **case** (*dict*) – case statements. When a key matches the value of the column in a specific row, the respective value will be assigned to the new column for that row. This is useful when your case condition constants are not strings.
- **default** – default value to be used when the value of the switch column doesn't match any keys.
- **operand** – function to compare the value of the switch column to the value of each case. Default is Column's eq. If user-provided, first argument will always be the switch Column; it's the user's responsibility to transform the case value to a column if they need to.
- **additional_cases** – additional "case" statements, kwargs style. Same semantics with cases above. If both are provided, cases takes precedence.

Returns pyspark.sql.Column

Example

```
switch_case('state',CA='California',NY='New York',default='Other')
```

is equivalent to

```
>>> F.when(
... F.col('state') == 'CA', 'California'
).when(
... F.col('state') == 'NY', 'New York'
).otherwise('Other')
```

If you need to "bucketize" a value

```
switch_case('age',{(13,17):  1,(18,24):  2,...},operand=lambda c,v:
c.between(*v))
```

is equivalent to

```
>>> F.when(
... F.col('age').between(13, 17), F.lit(1)
).when(
... F.col('age').between(18, 24), F.lit(2)
)
```

CHAPTER 6

Generic Utils

These are generic utils used in Sparkly.

sparkly.utils.**absolute_path**(*file_path*, **rel_path*)

Return absolute path to file.

Usage:

```
>>> absolute_path('/my/current/dir/x.txt', '..', 'x.txt')
'/my/current/x.txt'
```

```
>>> absolute_path('/my/current/dir/x.txt', 'relative', 'path')
'/my/current/dir/relative/path'
```

```
>>> import os
>>> absolute_path('x.txt', 'relative/path') == os.getcwd() + '/relative/path'
True
```

Parameters

- **file_path** (*str*) – file
- **rel_path** (*list[str]*) – path parts

Returns str

sparkly.utils.**kafka_get_topics_offsets**(*host*, *topic*, *port=9092*)

Return available partitions and their offsets for the given topic.

Parameters

- **host** (*str*) – Kafka host.
- **topic** (*str*) – Kafka topic.
- **port** (*int*) – Kafka port.

Returns [– [(partition, start_offset, end_offset)].

Return type int, int, int

class sparkly.utils.**lru_cache**(*maxsize=128*, *storage_level=StorageLevel(False*, *True*, *False*, *False*, *1)*)

LRU cache that supports DataFrames.

Enables caching of both the dataframe object and the data that df contains by persisting it according to user specs. It's the user's responsibility to make sure that the dataframe contents are not evicted from memory and/or disk should this feature get overused.

Parameters

- **maxsize** (*int|128*) – maximum number of items to cache.
- **storage_level** (*pyspark.StorageLevel|MEMORY_ONLY*) – how to cache the contents of a dataframe (only used when the cached function results in a dataframe).

sparkly.utils.**parse_schema**(*schema*)

Generate schema by its string definition.

It's basically an opposite action to *DataType.simpleString* method. Supports all atomic types (like string, int, float...) and complex types (array, map, struct) except DecimalType.

Usages:

```
>>> parse_schema('string')
StringType
>>> parse_schema('int')
IntegerType
>>> parse_schema('array<int>')
ArrayType(IntegerType,true)
>>> parse_schema('map<string,int>')
MapType(StringType,IntegerType,true)
>>> parse_schema('struct<a:int,b:string>')
StructType(List(StructField(a,IntegerType,true),StructField(b,StringType,true)))
>>> parse_schema('unsupported')
Traceback (most recent call last):
...
sparkly.exceptions.UnsupportedDataType: Cannot parse type from string:
 ↪"unsupported"
```

sparkly.utils.**schema_has**(*t*, *required_fields*)

Check whether a complex dataType has specific fields.

Parameters

- **t** (*pyspark.sql.types.ArrayType, MapType, StructType*) – type to check.
- **required_fields** (*same with t or dict[str, pyspark.sql.DataType]*) – fields that need to be present in t. For convenience, a user can define a `dict` in place of a `pyspark.sql.types.StructType`, but other than that this argument must have the same type as t.

Raises

- `AssertionError` – if t and required_fields cannot be compared because they aren't instances of the same complex dataType.
- `KeyError` – if a required field is not found in the struct.
- `TypeError` – if a required field exists but its actual type does not match the required one.

CHAPTER 7

License

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Object
      form, that is based on (or derived from) the Work and for which the
      editorial revisions, annotations, elaborations, or other modifications
```

```
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
```

```
        that You distribute, all copyright, patent, trademark, and
        attribution notices from the Source form of the Work,
        excluding those notices that do not pertain to any part of
        the Derivative Works; and

    (d) If the Work includes a "NOTICE" text file as part of its
        distribution, then any Derivative Works that You distribute must
        include a readable copy of the attribution notices contained
        within such NOTICE file, excluding those notices that do not
        pertain to any part of the Derivative Works, in at least one
        of the following places: within a NOTICE text file distributed
        as part of the Derivative Works; within the Source form or
        documentation, if provided along with the Derivative Works; or,
        within a display generated by the Derivative Works, if and
        wherever such third-party notices normally appear. The contents
        of the NOTICE file are for informational purposes only and
        do not modify the License. You may add Your own attribution
        notices within Derivative Works that You distribute, alongside
        or as an addendum to the NOTICE text from the Work, provided
        that such additional attribution notices cannot be construed
        as modifying the License.

    You may add Your own copyright statement to Your modifications and
    may provide additional or different license terms and conditions
    for use, reproduction, or distribution of Your modifications, or
    for any such Derivative Works as a whole, provided Your use,
    reproduction, and distribution of the Work otherwise complies with
    the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
    any Contribution intentionally submitted for inclusion in the Work
    by You to the Licensor shall be under the terms and conditions of
    this License, without any additional terms or conditions.
    Notwithstanding the above, nothing herein shall supersede or modify
    the terms of any separate license agreement you may have executed
    with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
    names, trademarks, service marks, or product names of the Licensor,
    except as required for reasonable and customary use in describing the
    origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
    agreed to in writing, Licensor provides the Work (and each
    Contributor provides its Contributions) on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied, including, without limitation, any warranties or conditions
    of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
    PARTICULAR PURPOSE. You are solely responsible for determining the
    appropriateness of using or redistributing the Work and assume any
    risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
    whether in tort (including negligence), contract, or otherwise,
    unless required by applicable law (such as deliberate and grossly
    negligent acts) or agreed to in writing, shall any Contributor be
    liable to You for damages, including any direct, indirect, special,
    incidental, or consequential damages of any character arising as a
```

```
      result of this License or out of the use or inability to use the
      Work (including but not limited to damages for loss of goodwill,
      work stoppage, computer failure or malfunction, or any and all
      other commercial damages or losses), even if such Contributor
      has been advised of the possibility of such damages.

   9. Accepting Warranty or Additional Liability. While redistributing
      the Work or Derivative Works thereof, You may choose to offer,
      and charge a fee for, acceptance of support, warranty, indemnity,
      or other liability obligations and/or rights consistent with this
      License. However, in accepting such obligations, You may act only
      on Your own behalf and on Your sole responsibility, not on behalf
      of any other Contributor, and only if You agree to indemnify,
      defend, and hold each Contributor harmless for any liability
      incurred by, or claims asserted against, such Contributor by reason
      of your accepting any such warranty or additional liability.

   END OF TERMS AND CONDITIONS

   APPENDIX: How to apply the Apache License to your work.

      To apply the Apache License to your work, attach the following
      boilerplate notice, with the fields enclosed by brackets "[]"
      replaced with your own identifying information. (Don't include
      the brackets!)  The text should be enclosed in the appropriate
      comment syntax for the file format. We also recommend that a
      file or class name and description of purpose be included on the
      same "printed page" as the copyright notice for easier
      identification within third-party archives.

   Copyright 2017 Tubular Labs, Inc.

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.


========================================================================
Sparkly Subcomponents:

The Sparkly project contains subcomponents with separate copyright
notices and license terms. Your use of the source code for the these
subcomponents is subject to the terms and conditions of the following
licenses.

========================================================================
Apache licenses
========================================================================

The following dependencies are provided under a Apache license. See project link for␣
↪details.
```

```
(Apache License 2.0) Spark (https://github.com/apache/spark)
(Apache License 2.0) cassandra-driver (https://github.com/datastax/python-driver)

========================================================================
BSD-style licenses
========================================================================

The following dependencies are provided under a BSD-style license. See project link␣
↪for details.

(BSD License) mock (https://github.com/testing-cabal/mock)
(PSF License) Sphinx (https://github.com/sphinx-doc/sphinx)

========================================================================
MIT licenses
========================================================================

The following dependencies are provided under the MIT License. See project link for␣
↪details.

(MIT License) sphinx_rtd_theme (https://github.com/snide/sphinx_rtd_theme)
(MIT License) pytest (https://github.com/pytest-dev/pytest)
(MIT License) pytest-cov (https://github.com/pytest-dev/pytest-cov)
(MIT License) PyMySQL (https://github.com/PyMySQL/PyMySQL)
```

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

S

S

T

U