Spark SQL Concepts

CMPT 732, Fall 2017

Working With Data

You have probably noticed a few things about how you work with Spark RDDs:

- You are often using tuples (or other data structures) to store some "fields" in each element.
- There is a fixed schema for that RDD's data structure, known only to you.
- You spend a lot of effort building the right key/value pairs, because there are so many "by key" operations.
- The actual operations you are trying to do are SQL-like.

> "As various NoSQL databases matured, a curious thing happened to their APIs: they started looking more like SQL. This is because SQL is a pretty direct implementation of relational set theory, and math is hard to fool." - *[Carlos Bueno, Cache is the new RAM]*

Spark SQL

If we are going to express SQL-like things, why not admit it and have an API that lets us?

Spark SQL is essentially the result of thinking: Spark RDDs are a good way to do distributed data manipulation, but (for some use-cases) we need a more tabular data layout and richer query/manipulation operations.

Data Frames

The basic data structure in Spark SQL is the *DataFrame*. Inspired by Pandas' DataFrames.

It is inherently tabular: has a fixed schema (≈ set of columns) with types, like a database table.

It is implemented with RDDs: each "row" is an element in some underlying RDD.

DataFrames can be created by a `SparkSession` object.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('example').getOrCreate()
df = spark.createDataFrame(some_rdd_or_list)
df = spark.read.….load(input_dir)
```

Think of a DataFrame being implemented with an RDD of `Row` objects. Nicest way to create `Row`s: create a custom subclass for your data:

```
NameAge = Row('fname', 'lname', 'age') # build a Row subclass
data_rows = [
    NameAge('John', 'Smith', 47),
    NameAge('Jane', 'Smith', 22),
    NameAge('Frank', 'Jones', 28),
]


# create a DataFrame from an RDD of Rows
sc = spark.sparkContext
data_rdd = sc.parallelize(data_rows)
data = spark.createDataFrame(data_rdd)
```

```
data = spark.createDataFrame(data_rdd)

# ... or from a list (equivalent for small data)
data = spark.createDataFrame(data_rows)
```

Data Frames are table-like and have a fixed schema:

```
>>> data.show()
+-----+-----+---+
|fname|lname|age|
+-----+-----+---+
| John|Smith| 47|
| Jane|Smith| 22|
|Frank|Jones| 28|
+-----+-----+---+
>>> data.printSchema()
root
 |-- fname: string (nullable = true)
 |-- lname: string (nullable = true)
 |-- age: long (nullable = true)
```

`.show()` is a convenient debugging/testing output method. You can also specify a schema explicitly: later.

Methods on Data Frames feel very SQL-like:

```
>>> data.where(data['age'] < 30) \
    .select([data['fname'], data['age']]).show()
+-----+---+
|fname|age|
+-----+---+
| Jane| 22|
|Frank| 28|
+-----+---+
>>> data.groupby(data['lname']).max().show()
+-----+--------+
|lname|max(age)|
+-----+--------+
|Jones|      28|
|Smith|      47|
+-----+--------+
```

Their arguments are slightly-odd expressions, **not Python functions**.

Column Expressions

These arguments to the DataFrame methods are *column expressions*:

```
data['fname']
data['age'] < 30
from pyspark.sql import functions
functions.log10(data['age'])
```

These are **not** Python calculations: they are a way to express an operation like “take the ‘age’ column from ‘data’ and compare it to the integer 30”.

The actual calculation is done by the Spark SQL engine (in Scala code). We have to build the expression in Python with this (somtimes odd) syntax.

There are many Spark SQL functions that can be used in column expressions, as well as basic Python operators that are overloaded to imply a column operation.

There are many places you have to refer to a column, and there are three different ways to do it. These are **equivalent**:

```
data.groupby(data['lname'])  # as a getitem on the DF
data.groupby('lname')        # by column name only
data.groupby(data.lname)     # as a property on the DF
```

Mixing these can be confusing. Suggestion: stick to `data['lname']` style: it always works an is unambiguous.

... or the `'lname'` style where it's more clear.

The various representations fail weirdly:

```
data.where(data['age'] < 25) # works
data.where(data.age < 25)    # works
data.where('age' < 25)       # fails: TypeError
```

... because `'age' < 25` is a bool, not a column expression.

```
maxage = data.groupby(data['lname']).max()
maxage.select(maxage['max(age)']) # works
maxage.select('max(age)')         # works
maxage.select(maxage.max(age))    # fails: AttributeError
```

... because `maxage` doesn't have a `max` attribute, and if it did, `maxage.max(age)` is a Python function call, not what you expect.

Because you can refer to a column with its name in a string, this is ambiguous:

```
data.where(data['fname'] == 'John')
data.where(data['fname'] == 'lname')
```

Are `'John'` and `'lname'` column names or string literals? Be explicit.

```
data.where(data['fname'] == functions.lit('John'))
data.where(data['fname'] == data['lname'])
```

The pyspark.sql.functions module has functions for lots of useful calculations in column expressions: use/combine when possible.

In core-Spark, we wrote Python functions so could have any logic. Spark SQL has only column expressions: we're restricted to what they can do.

UDFs

But we can register a *user defined function* (*UDF*) from Python.

```
def my_awesome_logic(name):
    ⋮
awesome = functions.udf(my_awesome_logic, types.IntegerType())
df = df.select(df['name'], awesome(df['name']))
```

... but there's a speed penalty getting the data to/from Python.

SQL Syntax

There is also a `spark.sql` function where you can do the same things with SQL query syntax. These are equivalent:

```
maxage = data.groupby(data['lname']).max()
ages = maxage.select(maxage['max(age)'])
```

```
data.createOrReplaceTempView('data')
ages = spark.sql("""
    SELECT MAX(age) FROM data GROUP BY lname
""")
```

My experience: simple logic looks simpler in SQL syntax; difficult logic looks simpler in the Python-method-call syntax.

Limitations

The big limitation of Spark SQL: you're limited to SQL-like operations, but you can convert RDDs and DataFrames.

Common pattern:

```
# build a DataFrame from an RDD
data_rows = sc.textFile(input).map(lines_to_rows)
data = spark.createDataFrame(data_rows, schema=schema)
# work in Spark SQL
intermediate_data = data.…
final_results = intermediate_data.…
# take out the DataFrame of Rows and output
result_rows = final_results.rdd
result_lines = result_rows.map(row_to_output)
result_lines.saveAsTextFile(output)
```

The Optimizer

Because we are expressing things at a higher level, there's more opportunity for an optimizer to do good work.

Like most database tools, Spark SQL will explain a query:

```
>>> comments = spark.read.json(inputs)
>>> averages = comments.groupby('subreddit').avg('score')
>>> averages.explain()
== Physical Plan ==
*HashAggregate(keys=[subreddit#26], functions=[avg(score#24L)])
+- Exchange hashpartitioning(subreddit#26, 200)
   +- *HashAggregate(keys=[subreddit#26], functions=[partial_avg(score#24L)])
      +- *FileScan json [score#24L,subreddit#26] Batched: false, Format: JSON,
         Location: InMemoryFileIndex[…/reddit-1], PartitionFilters: [], PushedFi
         ReadSchema: struct<score:bigint,subreddit:string>
```

Notes: only required columns read; aggregation done locally for sum and count (like a combiner), then shuffled and finished.

Compare the execution plan for a `.sort()`:

```
>>> averages.sort('avg(score)').explain()
== Physical Plan ==
*Sort [avg(score)#74 ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(avg(score)#74 ASC NULLS FIRST, 200)
   +- *HashAggregate(keys=[subreddit#26], functions=[avg(score#24L)])
      +- Exchange hashpartitioning(subreddit#26, 200)
```

```
         +- *HashAggregate(keys=[subreddit#26], functions=[partial_avg(score#24L
            +- *FileScan json [score#24L,subreddit#26] …
```

Note: repartitions by "range" then sorts partitions.

By looking at execution plans, I realized there were different kinds of repartitioning:

```
>>> comments.repartition(10).groupby('subreddit').avg('score').explain()
== Physical Plan ==
*HashAggregate(keys=[subreddit#26], functions=[avg(score#24L)])
+- Exchange hashpartitioning(subreddit#26, 200)
   +- *HashAggregate(keys=[subreddit#26], functions=[partial_avg(score#24L)])
      +- Exchange RoundRobinPartitioning(10)
         +- *FileScan json [score#24L,subreddit#26] …
>>> comments.repartition(10, 'subreddit').groupby('subreddit').avg('score').expl
== Physical Plan ==
*HashAggregate(keys=[subreddit#26], functions=[avg(score#24L)])
+- *HashAggregate(keys=[subreddit#26], functions=[partial_avg(score#24L)])
   +- Exchange hashpartitioning(subreddit#26, 10)
      +- *FileScan json [score#24L,subreddit#26] …
```

Note: plan for `.groupby()` changes with different input partitioning.

The DataFrames optimizer seems to be where future performance improvements are going to come from in Spark. Some links:

- `DataFrame.join()` automatically does a broadcast join if appropriate and can be given a hint if it guesses wrong.
- Core DataFrames: Catalyst Optimizer, Project Tungsten.
- The new Cost Based Optimizer.

Implication: you should probably think of DataFrame operations less like an imperative series of program steps, and more like a declarative SQL query.

Describe the results you want as clearly as possible. Let the optimizer figure it out. Explore the execution plan and fix as needed.

Python ↔ JVM

The elements of a Python RDD were always opaque to the underlying Scala/JVM code: they were just **serialized Python objects**, and all work on them was done by *passing the data* back to Python code.

DataFrames contain JVM (Scala) objects: all **manipulation is done in the JVM**. Our Python code *passes descriptions of the calculations* to the JVM.

Implications:

- Spark SQL can be faster, since no significant logic is happening in Python (which is generally slower).
- Converting to a DataFrame (`spark.createDataFrame(rdd)`) or RDD (`df.rdd`) isn't free: data must be converted between representations.
- Same for a UDF: requires JVM → Python → JVM.

Partitioning

When saving a DataFrame, you can partition by the value of a field (or several):

```
comments.write.partitionBy('subreddit').parquet('output')
```

This creates a directory structure like:

- output
  - subreddit=canada
    - part-00000.gz.parquet
  - subreddit=django
    - part-00000.gz.parquet
  - subreddit=xkcd
    - part-00000.gz.parquet

With a partitioned file, you can read only parts:

```
spark.read.parquet('output/subreddit=canada')
```

And Spark will know the partitioning, so this *should* be fast:

```
spark.read.parquet('output').… \
    .where('subreddit' == lit('canada'))
```

Also, the files in `subreddit=canada` **do not store** a subreddit field: it's implied by the directory name.

Other Input/Output Formats

We have seen JSON and Parquet.

Spark SQL can also natively read/write JDBC connections, Hive ORC files.

There are also Spark Packages that add other input/output formats including e.g. MongoDB, Cassandra, ElasticSearch. (Other packages include other functionality: ML algos, streaming sources, reading/writing RDDs, ...)

Parquet

Parquet is an efficient columnar format usable by many big data tools (including Spark).

Columnar format: the data for each column is stored together (as opposed to each row). Allows efficient reading/writing of only some columns.

Parquet *contains a schema* for the data: no need to give it explicitly yourself.

Spark SQL can append to Parquet files (and also JSON and others).

```
data1.write.parquet('output-directory', mode='overwrite')
data2.write.parquet('output-directory', mode='append')
```

The two DataFrames here probably should have similar schemas. Creates files like:

- output-directory
  - part-r-00000-10540a49-4828.gz.parquet
  - part-r-00000-a2e195a1-ccf8.gz.parquet
  - part-r-00001-10540a49-4828.gz.parquet
  - part-r-00001-a2e195a1-ccf8.gz.parquet