Spark DataFrames Concepts

CMPT 732, Fall 2022

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, 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]*

The way RDDs store data force you to have a row-oriented organization: *each row is stored together* in memory.

But computers have memory cache and vector instructions (SSE, AVX). For those, column-oriented data makes much more sense: *keep columns together*.

Spark DataFrames

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

Spark DataFrames are essentially the result of thinking: Spark RDDs are a good way to do distributed data manipulation, but (usually) we need a more tabular data layout and richer query/manipulation operations.

DataFrames

The basic data structure we'll be using here is a ***DataFrame***. Inspired by Pandas' DataFrames.

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

Think of a DataFrame as a table where each “row” is an element in some underlying RDD (*).

(*) It's not implemented as an RDD, but close enough for now.

DataFrames can be created by a `SparkSession` object.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('example').getOrCreate()
cities = spark.read.csv('cities', header=True, inferSchema=True)
```

The `SparkSession` does for DataFrames what the `SparkContext` does for RDDs: gives us an entry point to all of the functionality.

[A small example CSV file]

In pyspark, the `spark` object is already created:

```
>>> cities = spark.read.csv('cities', header=True, inferSchema=True)
```

We have asked that the first line of the CSV file(s?) be used for column names, and that the data types be inferred. You can (and probably should) specify a schema explicitly: later.

`.show()` is a convenient debugging/testing output method.

```
>>> cities.show()
+---------+----------+-------+
|     city|population|   area|
+---------+----------+-------+
|Vancouver|   2463431|2878.52|
|  Calgary|   1392609|5110.21|
|  Toronto|   5928040|5905.71|
| Montreal|   4098927|4604.26|
|  Halifax|    403390|5496.31|
+---------+----------+-------+
```

It shows only the first few rows.

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

```
>>> cities.printSchema()
root
 |-- city: string (nullable = true)
 |-- population: integer (nullable = true)
 |-- area: double (nullable = true)
```

Methods on DataFrames feel very SQL-like:

```
>>> small_cities = cities.where(cities['area'] < 5000)
>>> small_cities.show()
+---------+----------+-------+
|     city|population|   area|
+---------+----------+-------+
|Vancouver|   2463431|2878.52|
| Montreal|   4098927|4604.26|
+---------+----------+-------+
>>> cities.select(cities['population'] * 2).show()
+----------------+
|(population * 2)|
+----------------+
|         4926862|
|         2785218|
|        11856080|
|         8197854|
|          806780|
+----------------+
```

The arguments to these functions are slightly-odd expressions, **not Python functions**.

```
cities.where(cities['area'] < 5000)
cities.select(cities['population'] * 2)
```

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 (sometimes 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.

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'])
```

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.

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 can explain a plan:

```
>>> comments = spark.read.json(inputs, schema=comments_schema)
>>> averages = comments.groupby('subreddit').avg('score')
>>> averages.explain()
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[subreddit#18], functions=[avg(score#16L)])
   +- Exchange hashpartitioning(subreddit#18, 200), ENSURE_REQUIREMENTS, [id=#11]
      +- HashAggregate(keys=[subreddit#18], functions=[partial_avg(score#16L)])
         +- FileScan json [score#16L,subreddit#18] Batched: false, DataFilters: [], Form
```

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 ==
AdaptiveSparkPlan isFinalPlan=false
+- Sort [avg(score)#64 ASC NULLS FIRST], true, 0
   +- Exchange rangepartitioning(avg(score)#64 ASC NULLS FIRST, 200), ENSURE_REQUIREMENT
      +- HashAggregate(keys=[subreddit#18], functions=[avg(score#16L)])
         +- Exchange hashpartitioning(subreddit#18, 200), ENSURE_REQUIREMENTS, [id=#28]
            +- HashAggregate(keys=[subreddit#18], functions=[partial_avg(score#16L)])
               +- FileScan json [score#16L,subreddit#18] …
```

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 ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[subreddit#18], functions=[avg(score#16L)])
   +- Exchange hashpartitioning(subreddit#18, 200), ENSURE_REQUIREMENTS, [id=#48]
      +- HashAggregate(keys=[subreddit#18], functions=[partial_avg(score#16L)])
         +- Exchange RoundRobinPartitioning(10), REPARTITION_BY_NUM, [id=#44]
            +- FileScan json [score#16L,subreddit#18] …
>>> comments.repartition(10, 'subreddit').groupby('subreddit').avg('score').explain()
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[subreddit#18], functions=[avg(score#16L)])
   +- HashAggregate(keys=[subreddit#18], functions=[partial_avg(score#16L)])
      +- Exchange hashpartitioning(subreddit#18, 10), REPARTITION_BY_NUM, [id=#61]
         +- FileScan json [score#16L,subreddit#18] …
```

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

We saw on assignment 5: the `'s3selectCSV'` input format, where Spark can push the filter all the way down to the input, and never have to process filtered-out records itself.

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 Cost Based Optimizer and Adaptive Query Execution.

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.

Input/Output

Reading and writing DataFrames is done with `spark.read` and `df.write`.

These provide access to many formats: CSV, newline-separated JSON, JDBC database connections, text files (line-by-line).

Compression and existing directories are handled easily:

```
df.write.json('output', compression='gzip', mode='overwrite')
df.write.csv('output', compression='lz4', mode='append')
```

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

Aside: compression. Why have we been keeping all of the input files compressed on disk?

Option 1: keep the files compressed on disk. On each run, read the files and uncompress.

Option 2: keep the files uncompressed. Read the files and use it directly.

#1 can often be faster: processors are fast and disks are slow. Our files have been gzip-compressed. A faster algorithm like LZ4 or Snappy might have been better.

Parquet

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

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.

Depending on your data, it might make sense to do an ***ETL*** (extract-transform-load) step where you:

1. Read the original data format you got.
2. Do some basic transforms/cleanup to make the data more reasonable. Maybe repartition.
3. Write to Parquet files.

… and then start your analysis from there. Working with the cleaned Parquet files should be easier and faster.

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

But most simply, consider this as a ETL one-liner for your (project?) data:

```
spark.read.….repartition(1000).write.parquet('nicer-data')
```

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.

Limitations

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

With RDDs, we wrote Python functions so could have any logic.

The methods on DataFrames & columns, and column functions are usually enough to do the analysis you need. But what about when they aren't?

It's possible to convert a DataFrame to an RDD (and back).

It's not free: the Scala-based representation of the DataFrame must be converted to a Python representation for the RDD. The result is an RDD of `Row` objects.

Or take an RDD of `Row` objects (or similar) and build a DataFrame from it.

A common pattern if you have less-structured input:

```
def lines_to_rows(line):
    ⋮ # deal with funny input structure
    return Row(length=l, width=w, name=name)

# build a DataFrame from an RDD
data_rows = sc.textFile(inputs).map(lines_to_rows)
data = spark.createDataFrame(data_rows, schema=schema)
# work with the DataFrame
⋮
```

Or if you want an output format that isn't one provided by DataFrames' `.write`, you can do something like:

```
final_results = …
# 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)
```

UDFs

Another option: we can register a ***user defined function*** (***UDF***) from Python.

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

There's a significant time penalty for a Python UDF: send value from Scala to Python process, converting the format; call the Python function; send the value back and convert. A UDF should be a last resort to get something working.

As of Spark 2.3, you can use a Vectorized UDFs (or `pandas_udf`) where you get a **Pandas DataFrame of a partition** at a time, which can be created efficiently because of Apache Arrow. You do Python work and return the new partition.

Much faster than Python UDFs. Probably still slower than Spark DataFrame logic.

How will they compare? Let's try a simple example.

Remember the first option: **do it in Spark DataFrame calculations** and never run any Python logic:

```
res = df.select(
    (df['a'] + 2*df['b']*functions.log2(df['a'])).alias('res')
)
```

But if the computation was much easier to implement with NumPy or Pandas DataFrame operations, we could:

```
@functions.pandas_udf(returnType=types.DoubleType())
def pandas_logic(a: pd.Series, b: pd.Series) -> pd.Series:
    return a + 2*b*np.log2(a)
res = df.select(pandas_logic(df['a'], df['b']).alias('res'))
```

Or with pure Python operations if we must:

```
@functions.udf(returnType=types.DoubleType())
def python_logic(a: float, b: float) -> float:
    return a + 2*b*math.log2(a)
res = df.select(python_logic(df['a'], df['b']).alias('res'))
```

How long do they take? (n = 3×10^9 on cluster; 2×10^8 locally)

Implementation	Cluster Time	Local Time
Spark DataFrame ops	10 s	2.2 s
Pandas UDF	113 s	18.2 s
Python UDF	437 s	54.7 s

Others' examples suggest that the differences can be much larger than this. It depends on the calculation.

Or Option 4: find a Java/Scala library to do work you need, or write the UDF there.

The Pandas UDFs are called once for each partition of the DataFrame. You still don't operate on the whole collection of data, but on (hopefully) nicely-sized subsets at a time.

Aside: This can be done efficiently because DataFrames can be stored in the Apache Arrow representation. Then, no conversion is necessary between the JVM and Python calls.

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.