

SQL on Spark

Taryn Heilman
March 14, 2018



- Review Spark, RDDs and review SQL
- Introduce spark dataframes and spark SQL
- Be able to use python API and/or SQL method to operate on spark DataFrames
- Understand partitioning and how to query efficiently
- Introduce SQL functions

Python is an imperative language. What kind of language is SQL? What is the practical difference?

Put the following operations in the order they should appear in a SQL query: (GROUP BY, ORDER BY, SELECT, WHERE)

Put the following operations in the order in which they are EVALUATED in a SQL query: (GROUP BY, ORDER BY, SELECT, WHERE)

What does it mean that Spark is a lazy evaluator?

Why is Spark faster than Hadoop MapReduce?

What piece of code creates an RDD out of another object?

Are RDDs mutable? What are the practical implications of this?

This is the order that your queries should take!

SELECT and FROM are the only ones that are required

```
SELECT (DISTINCT, AGG*) <table1.col1, ... ,  
table1.colm, table2.col1, ... ,table2.coln>  
FROM <table1>  
JOIN <table2>  
ON <table1.colj> = <table2.colk>  
WHERE <table1.col1 = some_val> AND <table1.col1 =  
some_val>  
GROUP BY <table1.col>  
HAVING <AGG*(table1.col) = some_val>  
ORDER BY <table1.col> ASC <table2.col> DESC
```

1. **FROM + JOIN**: first the product of all tables is formed
2. **WHERE**: the where clause filters rows that do not meet the search condition
3. **GROUP BY + (COUNT, SUM, etc)**: the rows are grouped using the columns in the group by clause and the aggregation functions are applied on the grouping
4. **HAVING**: like the WHERE clause, but can be applied after aggregation
5. **SELECT**: the targeted list of columns are evaluated and returned
6. **DISTINCT**: duplicate rows are eliminated
7. **ORDER BY**: the resulting rows are sorted

WHERE clause: eliminate rows you don't want, and if data is smartly partitioned...
eliminate entire files! EFFICIENCY!

WHERE and **GROUP BY** are evaluated *before* **SELECT** statement. If you do an aggregation/ name change/other manipulation, you will need to use the original column name here because your new alias won't be recognized.

ORDER BY is evaluated *after* the **SELECT** statement. Use new aliases

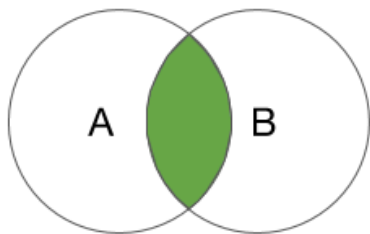
With smartly partitioned data, you can narrow down the number of files to open when querying out of your file system (in addition to simply eliminating rows to look at)

```
hdfs -ls -lh /data/front_end_records/data_center=DEN/year=2016/month=9
  jlnsf135hjv34jhv3q4tjhv.json      Sept  1 17:20      64M
  13gh134gvh1234ghv134fcg.json      Sept  1 17:21      64M
  ...
```

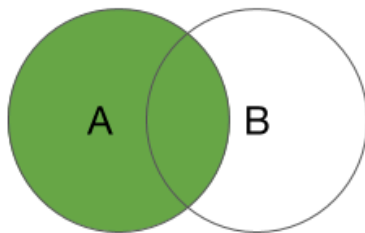
In practical terms....

- **(INNER) JOINS:** no null/nan values (only keeps rows that exist in both tables)
- **LEFT (RIGHT) JOIN:** Keeps all rows from the left (right) table. Expect some null values for rows that don't exist in the right (left) table.*
- **FULL JOIN:** Keeps all the rows! Lots of null values

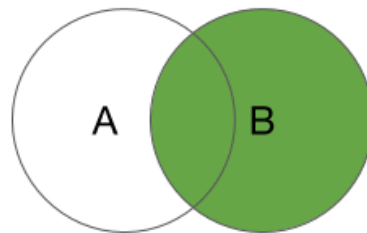
* in practice, there is no reason to use a right join.



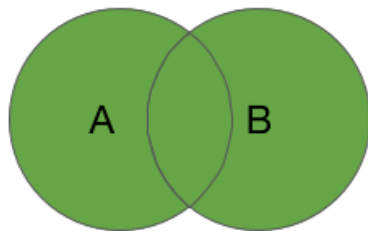
INNER JOIN



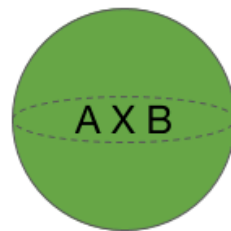
LEFT OUTER JOIN



RIGHT OUTER JOIN



FULL OUTER JOIN



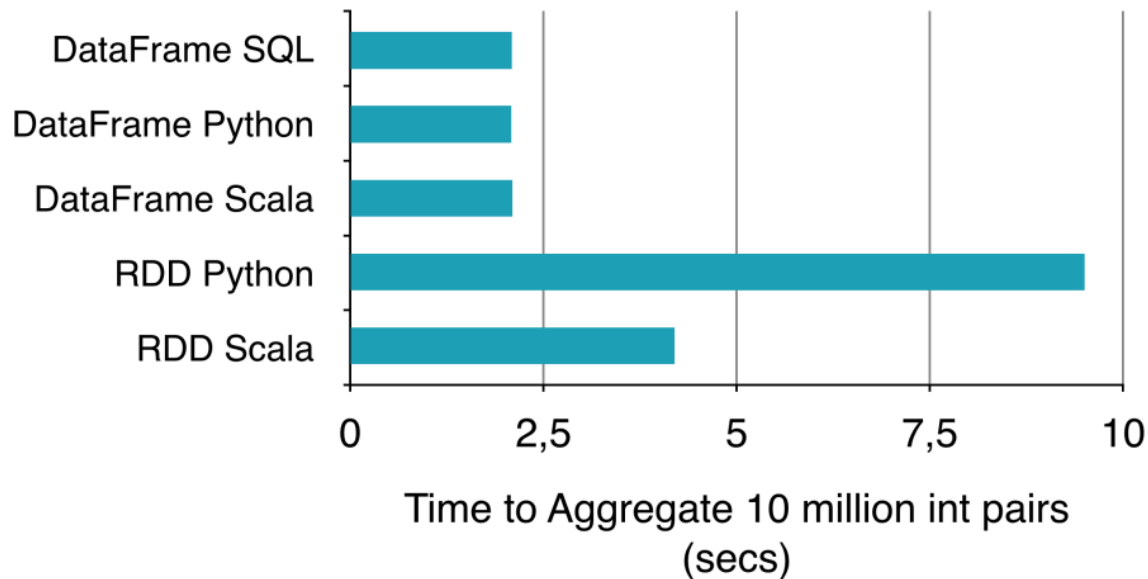
CARTESIAN
(CROSS) JOIN

*LEFT SEMI JOINS are also a thing. We won't get into them today

Spark DataFrames



Physical Execution: Unified Across Languages



- Primary abstraction in Spark SQL - have a defined schema, unlike RDDs
- Look and behave (mostly) like pandas Dataframes, R dataframes, sql data tables and other tabular data objects
- SQL functionality is great - as of Spark 2.0, Hive, Pig, and SQL functionality are all integrated within the sparkSQL paradigm.
- Can operate on DataFrames with DataFrame methods, SQL functions, or query out of them
- `spark.read.csv` (or `.json` or `.parquet`, etc.) automatically gives you a DataFrame
- immutable, like RDDs

You can create a data frame by applying a schema to an RDD, or by inferring one as you read in the file (set the argument `inferSchema=True`)

What is a schema?

- Schemas are metadata about your data.
- Schemas enable using SQL and DataFrame syntax to query your RDDs, instead of using column positions.
- Schema = Table Names + Column Names + Column Types

What are the benefits of having a schema?

- Schemas enable using column names instead of column positions
- Schemas enable queries using SQL and DataFrame syntax
- Schemas make your data more structured (all columns same data type, etc.)

Creating a Schema from RDD example



```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType, FloatType

schema = StructType([StructField('id', IntegerType(), True),
                      StructField('date', StringType(), True),
                      StructField('store', IntegerType(), True),
                      StructField('state', StringType(), True),
                      StructField('product', IntegerType(), True),
                      StructField('amount', FloatType(), True)])

# spark is a SparkSession
df = spark.createDataFrame(rdd_sales, schema)

df.show()

df.printSchema()
```

```
+---+-----+-----+-----+-----+-----+
| id|      date|store|state|product|amount|
+---+-----+-----+-----+-----+-----+
|101|11/13/2014| 100|  WA|    331| 300.0|
|104|11/18/2014| 700|  OR|    329| 450.0|
|102|11/15/2014| 203|  CA|    321| 200.0|
|106|11/19/2014| 202|  CA|    331| 330.0|
|103|11/17/2014| 101|  WA|    373| 750.0|
|105|11/19/2014| 202|  CA|    321| 200.0|
+---+-----+-----+-----+-----+-----+
```

```
root
 |-- id: integer (nullable = true)
 |-- date: string (nullable = true)
 |-- store: integer (nullable = true)
 |-- state: string (nullable = true)
 |-- product: integer (nullable = true)
 |-- amount: float (nullable = true)
```

Everything you are used to from RDDs, plus lots more. (not a complete list!)

Actions

- `.show(n)` or `.head(n)` to get the first `n` rows
- `.printSchema()` gives you the schema of the table (columns and datatypes, like `df.info()` in pandas)
- `.collect()` works the same as it does for RDDs, but is ugly. Use `show` instead!
- Aggregations (`.sum()`, `.count()`, `.min()`, `.max()`, etc.)

Transformations

- `.describe()` computes statistics for numeric and string columns
- `.sample()` and `.sampleBy()` give you subsets of the data for easier development

Spark SQL

galvanize

Documentation....the spark ml documentation is a little weak (e.g. examples have data frames with two rows and two columns....because that's totally a situation where we should use distributed computing)

This is NOT so for sparkSQL, this documentation is actually very helpful

<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html>

Two ways to do the same operation on a DataFrame...

```
# python API way
```

```
new_df = df.filter('col1 = some_val').groupBy('col2')\
    .agg({'col3': 'avg', 'col4': 'max'})
```

```
# SQL way
```

```
df.registerTempTable('df')
new_df = spark.sql('''
    SELECT AVG(col3), MAX(col4)
    FROM df
    WHERE col1 = some_val
    GROUP BY col2
''')
```

Two ways to use:

- Within SQL query - can use all functions except user-defined functions (udf) without importing
- As operation on dataframe, must import to do this

```
# DataFrame API way
import pyspark.sql.functions as F

new_df = df.select('col1', F.abs('col2').alias('abs_col2'))

# SQL query way
df.registerTempTable('df')
spark.sql('''
    SELECT
    col1, ABS(col2) AS abs_col2
    FROM df
    ''')
```

- Mathematical (round, floor/ceil, trig functions, exponents, log, factorial, etc.)
- Aggregations (count, average, min, max, first, last, collect_set, collect_list, etc.)
- Datetime manipulations (change timezone, change string/datetime/unix time)
- Hashing functions
- String manipulations (concatenations, slicing)
- Datatype manipulation (array certain columns together, cast to change datatype, etc.)

You aren't limited to only the functions available in spark SQL...you can make your own custom function to apply

```
from pyspark.sql.functions import udf
from pyspark.sql.types import *

def foo(args):
    """
    func foo, returns a string
    """
    return new_str

udf_foo = udf(lambda x: foo(x, other_args), StringType())
df = df.withCol('new_col', udf_foo(df.old_col))
```

<http://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.functions>

- Especially useful with time series or ordered data
- Rolling mean, exponentially weighted time series models, cumulative sums

```
from pyspark.sql.window import Window
import pyspark.sql.functions as f
windowSpec = Window.partitionBy().orderBy().rangeBetween() # fill these in

df = df.withCol('new_col', f.max(df.old_col.over(windowSpec)))
```

```
SELECT
  foo,
  func(old_col) OVER (PARTITION BY partition_col ORDER BY order_col DESC)
```

- Review Spark, RDDs and review SQL
- Introduce spark dataframes and spark SQL
- Be able to use python API and/or SQL method to operate on spark DataFrames
- Understand partitioning and how to query efficiently
- Introduce SQL functions

Questions?