Intro to DataFrames and Spark SQL

Spark SQL

- 对结构化数据，能够执行 SQL 命令
- Spark SQL 包括优化器，列存储和代码生成，可快速回答查询
- 可扩展到数千个工作节点

Spark SQL

- Part of the core distribution since 1.0 (April 2014)
- Runs SQL / HiveQL queries, optionally alongside or replacing existing Hive deployments

```
SELECT COUNT(*)
FROM hiveTable
WHERE hive_udf(data)
```

Improved multi-version support in 1.4

例：Spark SQL 文件输入

- 数据文件

```
Date, Time, desired temp, actual temp, buildingID
3/23/2016, 11:45, 67, 54, headquarters
3/23/2016, 11:51, 67, 77, lab1
3/23/2016, 11:20, 67, 33, coldroom
```

例：Spark SQL 文件输入

- Spark SQL 文件读入
  - 将文件加载到 Spark 中，通过 textFile 应用于 Spark 上下文对象，创建文本文件 RDD
  - 过滤掉标题行，将其余行映射到类型化的元组，将文本文件 RDD 转换为元组 RDD

```python
from pyspark.sql.types import *
hvacText = sc.textFile("/pathto/file/hvac.csv")
hvac = hvacText.map(lambda s: s.split(",")) \
               .filter(lambda s: s[0] != "Date") \
               .map(lambda s:(str(s[0]), str(s[1]),
                           int(s[2]), int(s[3]), str(s[4]) ))
sqlCtx = SQLContext(sc)
hvacSchema = StructType([StructField("date", StringType(), False),
                   StructField("time", StringType(), False),
                   StructField("targettemp", IntegerType(), False),
                   StructField("actualtemp", IntegerType(), False),
                   StructField("buildingID", StringType(), False)])
hvacDF = sqlCtx.createDataFrame(hvac, hvacSchema)
```

例：Spark SQL

- 使用 sql()方法执行 SQL 命令
- 结果返回为 DataFrame

```
x = sqlCtx.sql('SELECT buildingID from hvac')
```

例：Spark SQL

- 魔术运算符 %% sql_show
  - Jupyter 和 IPython 魔术运算符定义语言小型扩展
  - 能以自然方式输入 SQL 命令，结果打印为表格

```
%%sql_show
SELECT buildingID ,
      (targettemp - actualtemp) AS temp_diff ,
      date FROM hvac
WHERE date = "3/23/2016"
```

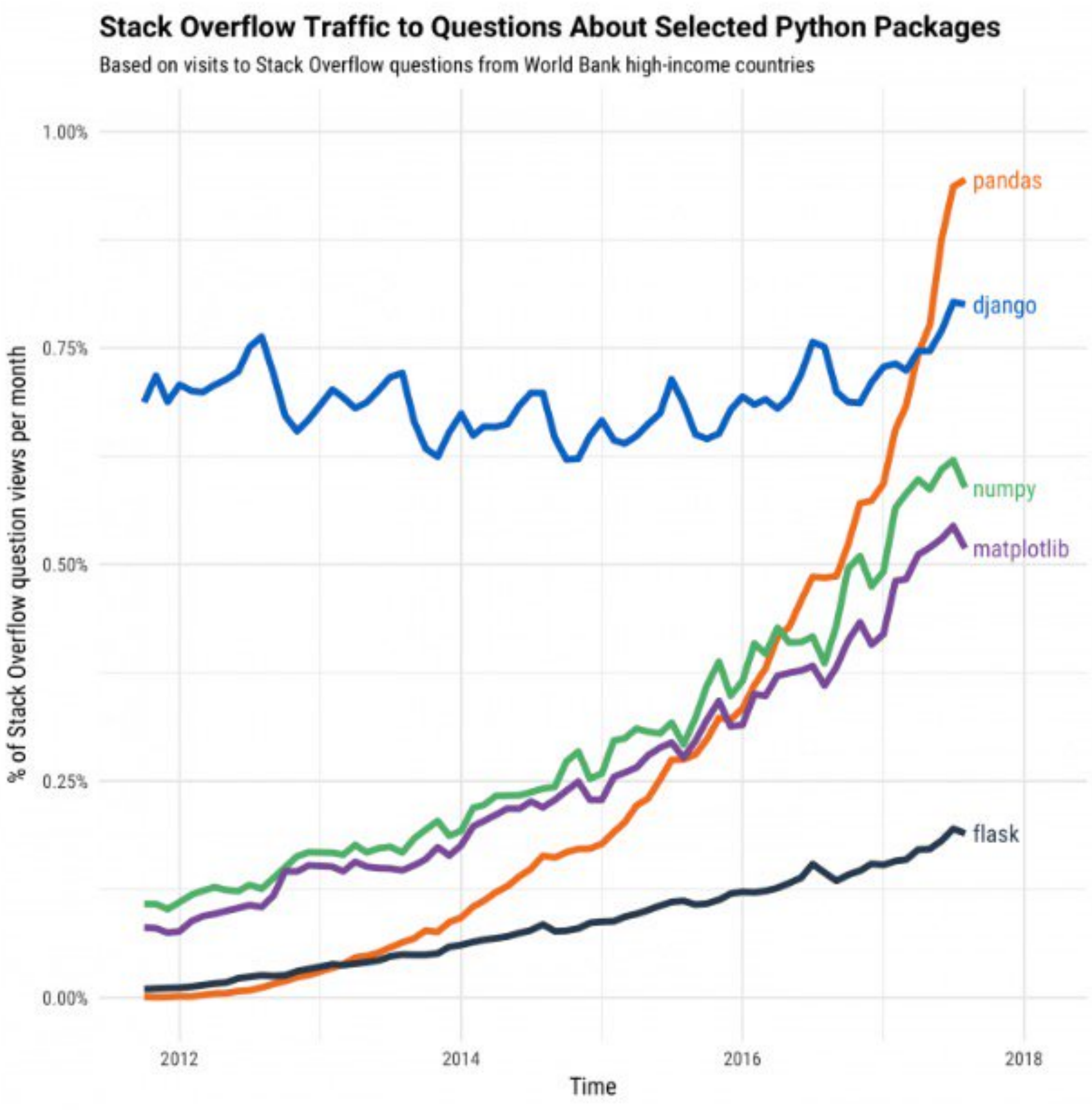
```
+------------+---------+---------+
|  buildingID|temp_diff|     date|
+------------+---------+---------+
|headquarters|       13|3/23/2016|
|        lab1|      -10|3/23/2016|
|    coldroom|       34|3/23/2016|
+------------+---------+---------+
```

DataFrames API

- Enable wider audiences beyond "Big Data" engineers to leverage the power of distributed processing
- Inspired by data frames in R and Python (Pandas)
- Designed from the ground-up to support modern big data and data science applications
- Extension to the existing RDD API

See

- https://spark.apache.org/docs/latest/sql-programming-guide.html
- databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html


Stack Overflow Traffic to Questions About Selected Python Packages
Based on visits to Stack Overflow questions from World Bank high-income countries
1.00%
0.75%
0.50%
0.25%
0.00%
% of Stack Overflow question views per month
pandas
django
numpy
matplotlib
flask
2012
2014
2016
2018
Time

Data Sources supported by DataFrames

built-in

Parquet

JDBC

{ JSON }

MySQL

PostgreSQL

HIVE

HDFS

amazon web services S3

external

AVRO

CSV

dBase

APACHE HBASE

elasticsearch.

cassandra

amazon web services Amazon Redshift

and more …

Write Less Code: High-Level Operations

Solve common problems concisely with DataFrame functions:

- selecting columns and filtering
- joining different data sources
- aggregation (count, sum, average, etc.)
- plotting results (e.g., with Pandas)

Write Less Code: Compute an Average

```
private IntWritable one = new IntWritable(1);
private IntWritable output =new IntWritable();
protected void map(LongWritable key,
                   Text value,
                   Context context) {
    String[] fields = value.split("\t");
    output.set(Integer.parseInt(fields[1]));
    context.write(one, output);
}

-----------------------------------------------------------------------------

IntWritable one = new IntWritable(1)
DoubleWritable average = new DoubleWritable();

protected void reduce(IntWritable key,
                      Iterable<IntWritable> values,
                      Context context) {
    int sum = 0;
    int count = 0;
    for (IntWritable value: values) {
        sum += value.get();
        count++;
    }
    average.set(sum / (double) count);
    context.write(key, average);
}
```

```
rdd = sc.textFile(...).map(_.split(" "))
rdd.map { x => (x(0), (x(1).toFloat, 1)) }.
    reduceByKey { case ((num1, count1), (num2, count2)) =>
      (num1 + num2, count1 + count2)
    }.
    map { case (key, (num, count)) => (key, num / count) }.
    collect()
```

```
rdd = sc.textFile(...).map(lambda s: s.split())
rdd.map(lambda x: (x[0], (float(x[1]), 1))).\
    reduceByKey(lambda t1, t2: (t1[0] + t2[0], t1[1] + t2[1])).\
    map(lambda t: (t[0], t[1][0] / t[1][1])).\
    collect()
```

Write Less Code: Compute an Average

Using RDDs

```
rdd = sc.textFile(...).map(_.split(" "))
rdd.map { x => (x(0), (x(1).toFloat, 1)) }.
    reduceByKey { case ((num1, count1), (num2, count2)) =>
      (num1 + num2, count1 + count2)
    }.
    map { case (key, (num, count)) => (key, num / count) }.
    collect()
```

Using DataFrames

```
import org.apache.spark.sql.functions._

val df = rdd.map(a => (a(0), a(1))).toDF("key", "value")
df.groupBy("key")
  .agg(avg("value"))
  .collect()
```

Full API Docs

- Scala
- Java
- Python
- R

Use DataFrames (Python)

```python
# Create a new DataFrame that contains only "young" users
young = users.filter(users["age"] < 21)

# Alternatively, using a Pandas-like syntax
young = users[users.age < 21]

# Increment everybody's age by 1
young.select(young["name"], young["age"] + 1)

# Count the number of young users by gender
young.groupBy("gender").count()

# Join young users with another DataFrame, logs
young.join(log, logs["userId"] == users["userId"], "left_outer")
```

DataFrames and Spark SQL

```
young.registerTempTable("young")

sqlContext.sql("SELECT count(*) FROM young")
```

Spark SQL

- You issue SQL queries through a `SQLContext` or `HiveContext`, using the `sql()` method.
- The `sql()` method returns a `DataFrame`.
- You can mix DataFrame methods and SQL queries in the same code.
- To use SQL, you *must* either:
  - query a persisted Hive table, or
  - make a *table alias* for a DataFrame, using `registerTempTable()`

Transformations, Actions, Laziness

Like RDDs, DataFrames are *lazy*. *Transformations* contribute to the query plan, but they don't execute anything.

Actions cause the execution of the query.

Transformation examples

- filter
- select
- drop
- intersect
- join

Action examples

- count
- collect
- show
- head
- take

DataFrame 示例

- 文本搜索
- 创建一个只有一个名为“line”的列的 DataFrame

```
textFile=sc.textFile(“hdfs:// …”)
df=textFile.map(lambda r: Row(r)).toDF([“line”])
```

- 计数所有错误

```
err=df.filter(col(“line”).like(“%ERROR%”))
err.count()
```

DataFrame 示例

- 计数提及 MySQL 的错误

```
err.filter(col(“line”).like(“%MySQL%”)).count()
```

- 以字符串数组的形式获取 MySQL 错误

```
err.filter(col(“line”)
        .like(“%MySQL%”)).collect()
```

printSchema()

You can have Spark tell you what it thinks the data schema is, by calling the `printSchema()` method. (This is mostly useful in the shell.)

```
> df.printSchema()
root
 |-- firstName: string (nullable = true)
 |-- lastName: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- age: integer (nullable = false)
```

Schema Inference Example

- Suppose you have a (text) file that looks like this:

```
Erin,Shannon,F,42
Norman,Lockwood,M,81
Miguel,Ruiz,M,64
Rosalita,Ramirez,F,14
Ally,Garcia,F,39
Claire,McBride,F,23
Abigail,Cottrell,F,75
José,Rivera,M,59
Ravi,Dasgupta,M,25
…
```

The file has no schema, but it's obvious there *is* one:

First name:	*string*
Last name:	*string*
Gender:	*string*
Age:	*integer*

Let's see how to get Spark to infer the schema.

Columns

<table>
<tr><th>Input Source Format</th><th>Data Frame Variable Name</th><th>Data</th></tr>
<tr><td>JSON</td><td><code>dataFrame1</code></td><td><pre>[{"first": "Amy",
   "last": "Bello",
   "age":   29 },
  {"first": "Ravi",
   "last": "Agarwal",
   "age":   33 },
  …
]</pre></td></tr>
<tr><td>CSV</td><td><code>dataFrame2</code></td><td><pre>first,last,age
Fred,Hoover,91
Joaquin,Hernandez,24
…</pre></td></tr>
<tr><td>SQL Table</td><td><code>dataFrame3</code></td><td><table><tr><th>first</th><th>last</th><th>age</th></tr><tr><td>Joe</td><td>Smith</td><td>42</td></tr><tr><td>Jill</td><td>Jones</td><td>33</td></tr></table></td></tr>
</table>

Let's see how DataFrame columns map onto some common data sources.

Columns

Assume we have a DataFrame, `df`, that reads a data source that has "first", "last", and "age" columns.

Python	Java	Scala	R
`df["first"]` `df.first`†	`df.col("first")`	`df("first")` `$"first"`‡	`df$first`

†In Python, it's possible to access a DataFrame's columns either by attribute (`df.age`) or by indexing (`df['age']`). While the former is convenient for interactive data exploration, you should *use the index form.* It's future proof and won't break with column names that are also attributes on the DataFrame class.

‡The $ syntax can be ambiguous, if there are multiple DataFrames in the lineage.

show()

```
> df.show()
+---------+--------+------+---+
|firstName|lastName|gender|age|
+---------+--------+------+---+
|     Erin| Shannon|     F| 42|
|   Claire| McBride|     F| 23|
|   Norman|Lockwood|     M| 81|
|   Miguel|    Ruiz|     M| 64|
| Rosalita| Ramirez|     F| 14|
|     Ally|  Garcia|     F| 39|
|  Abigail|Cottrell|     F| 75|
|     José|  Rivera|     M| 59|
+---------+--------+------+---+
```

select()

```
In[1]: df.select(df['first_name'], df['age'], df['age'] > 49).show(5)
+----------+---+----------+
|first_name|age|(age > 49)|
+----------+---+----------+
|      Erin| 42|     false|
|    Claire| 23|     false|
|    Norman| 81|      true|
|    Miguel| 64|      true|
|  Rosalita| 14|     false|
+----------+---+----------+
```

as() or alias()

```
In [7]: df.select(df['first_name'],\
                  df['age'],\
                  (df['age'] < 30).alias('young')).show(5)
+----------+---+-----+
|first_name|age|young|
+----------+---+-----+
|      Erin| 42|false|
|    Claire| 23| true|
|    Norman| 81|false|
|    Miguel| 64|false|
|  Rosalita| 14| true|
+----------+---+-----+
```


Note: In Python, you *must* use `alias`, because `as` is a keyword.

as()

And, of course, SQL:

```
sqlContext.sql("SELECT firstName, age, age < 30 AS young FROM names")

+---------+---+-----+
|firstName|age|young|
+---------+---+-----+
|     Erin| 42|false|
|   Claire| 23| true|
|   Norman| 81|false|
|   Miguel| 64|false|
| Rosalita| 14| true|
+---------+---+-----+
```

Joins

We can load that into a second DataFrame and join it with our first one.

```
In [1]: df2 = sqlContext.read.json("artists.json")
# Schema inferred as DataFrame[firstName: string, lastName: string, medium: string]
In [2]: df.join(
            df2,
            df.first_name == df2.firstName and df.last_name == df2.lastName
        ).show()
+----------+---------+------+---+---------+--------+-----------------+
|first_name|last_name|gender|age|firstName|lastName|           medium|
+----------+---------+------+---+---------+--------+-----------------+
|    Norman| Lockwood|     M| 81|   Norman|Lockwood|metal (sculpture)|
|      Erin|  Shannon|     F| 42|     Erin| Shannon|    oil on canvas|
|  Rosalita|  Ramirez|     F| 14| Rosalita| Ramirez|         charcoal|
|    Miguel|     Ruiz|     M| 64|   Miguel|    Ruiz|    oil on canvas|
+----------+---------+------+---+---------+--------+-----------------+
```

User Defined Functions

And… in Python

```
In [8]: from pyspark.sql.functions import udf
In [9]: from datetime import datetime
In [10]: month_name = udf(lambda d: datetime.strftime(d, "%b"))
In [11]: df.select(month_name(df['birth_date'])).show(5)
+----------------------------+
|PythonUDF#<lambda>(birth_date)|
+----------------------------+
|                         Jan|
|                         Feb|
|                         Dec|
|                         Aug|
|                         Aug|
+----------------------------+
```

`alias()` would "fix" this generated column name.

DataFrames can be *significantly* faster than RDDs. And they perform the same, regardless of language.