SPARK Tutorial Fed4FIRE 2016

Dieter De Witte
Big Data Scientist
iMinds Data Science Lab
drdwitte@gmail.com

Outline

I. Introducing Apache Spark

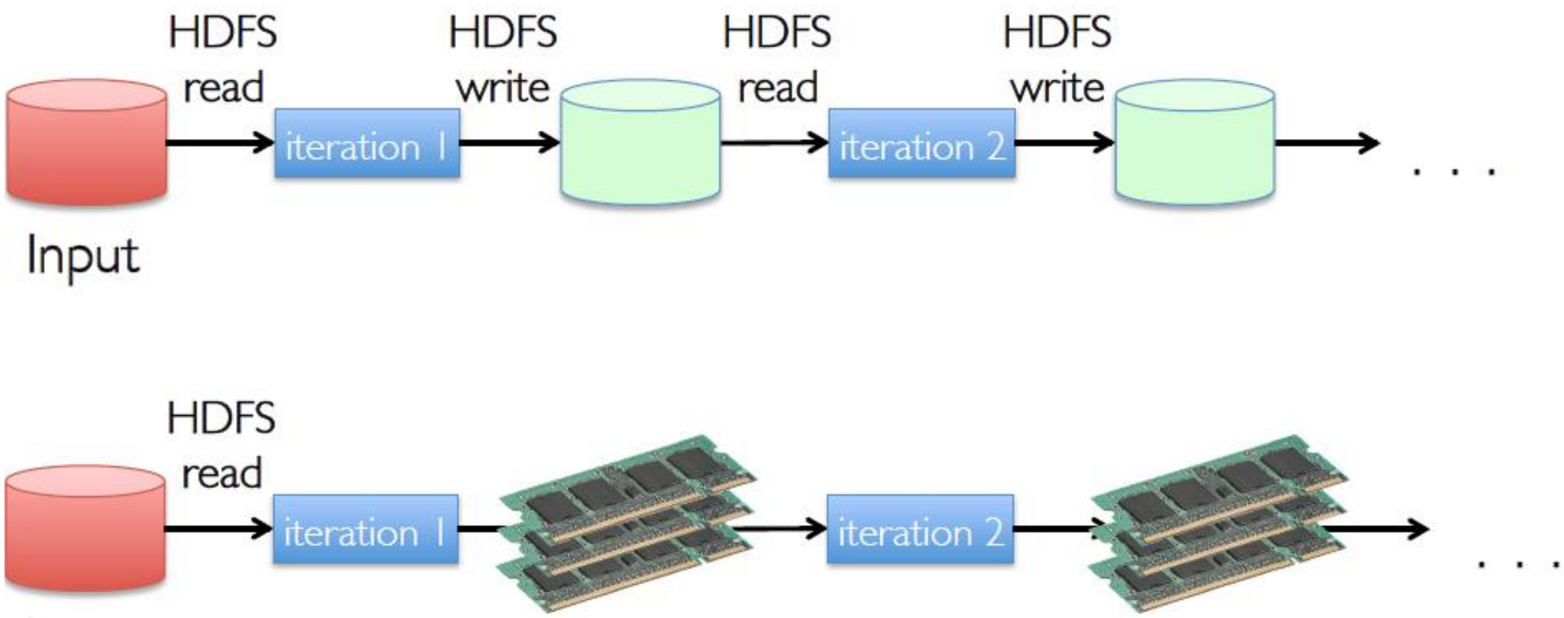
II. A spark playground for everyone!

III. Dataframes API

Spark's Design Goals

- **Low latency** (sub-second)
  while maintaining MR's
  Fault-tolerance
  & Scalability
- **Generality & Simplicity**
  - Support a wider range of workloads than MR
    => batch processing +
    machine learning & interactive querying
    & stream processing
  - Support for a more general set of data transformation operators
  - Less LOC
  - Support for a broader set of data science languages

Adding memory to the hardware stack



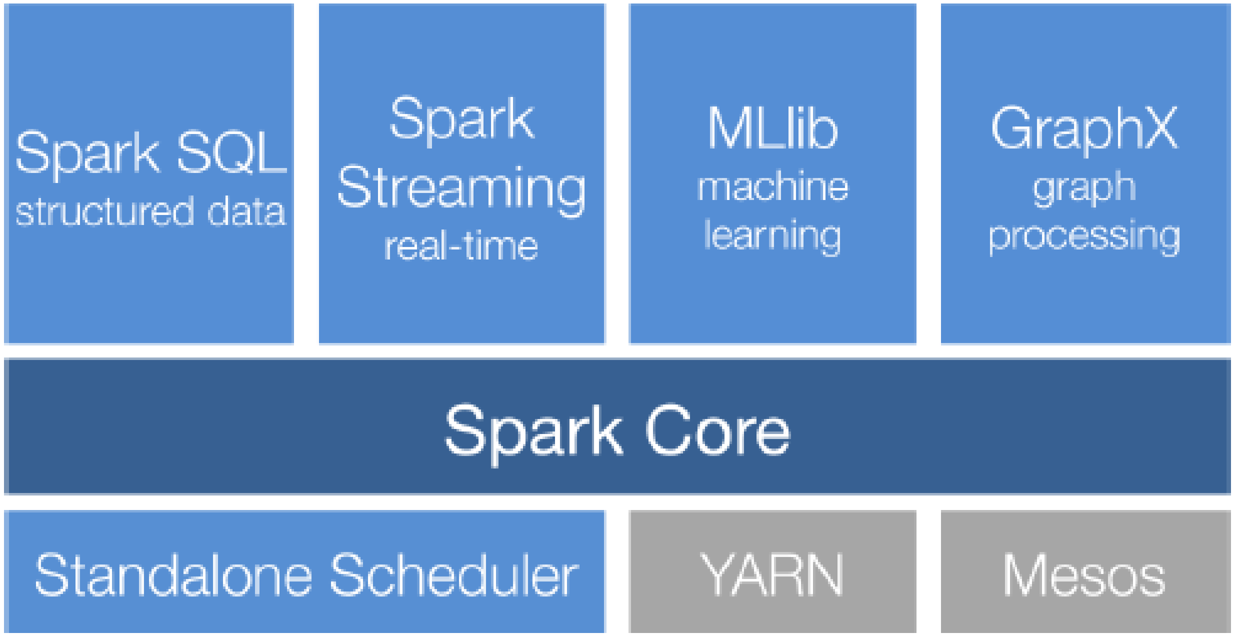
Improves iterative algorithms!

Same holds for iterative querying

Fault-tolerance without I/O overhead

- **Lazy transformations** on **distributed collections** (similar to transformations on Scala collections albeit distributed)
- **Failures** result in re-running **lineage graph**: data partitions ‘know’ their parent partitions
- Note: Lazy evaluation was already introduced in Apache Pig

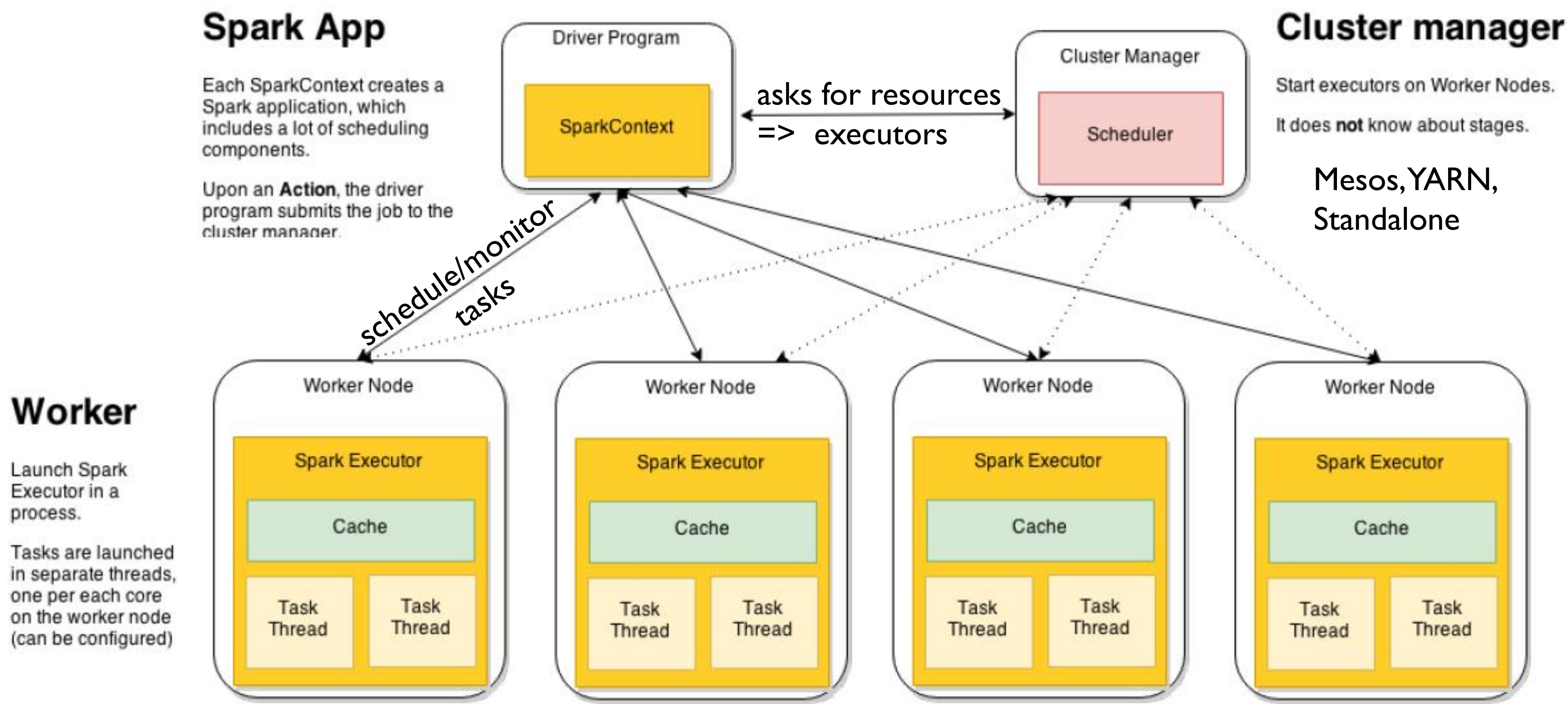
Spark Stack for diverse workloads



- Spark Stack:
  - Spark Streaming (stream processing)
  - GraphX (graph processing algorithms)
  - MLlib (parallel machine learning algorithms)
  - SparkSQL (SQL queries on Spark)

Spark Overview

Spark Components



Executors are long-lived JVMs with >=0 threads, 1 thread per task

MR: each task one JVM => overhead!

Lifecycle

- Create a Spark Context
- Creating RDDs & Persisting RDDs
- (Lazy) Transformations on RDDs
- (Lazy) Transformations on Pair RDDs
- Actions

Create a Spark Context

- A spark program requires a **SparkContext** object

sc = pyspark.SparkContext('local[*]')

=> How and where to access cluster?

=> master parameter determines cluster type:

- local (1machine 1 thread)
- local[K] (1 machine K threads), **local[*]**
- spark://HOST:PORT spark standalone cluster
- mesos://HOST:PORT connect to mesos cluster
- yarn-client, yarn-cluster

Creating RDDs

- Create an RDD from a local collection:

specify #partitions

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

```
sc.parallelize(data, 10))
```

- Create an RDD from external storage

```
distFile = sc.textFile("data.txt")
```

- Create an RDD from another file system
  - Amazon S3: "s3n://path/to/file.txt"
  - HDFS: "hdfs://namenode:port/path/to/file.txt"
- Custom Hadoop Input Formats:
  - old (hadoopRDD, hadoopFile), new(newAPIHadoopRDD,...)

arguments: HadoopInputFormat.class, Key.class, Value.class
"org.apache.hadoop.mapred.TextInputFormat", "org.apache.hadoop.io.Text", ...

- Note:
  - More partitions = more parallellism (rule of thumb 2*numCores)
  - *sc.wholeTextFiles(...)* for nonsplittable files

Persisting RDDs

Level	Space Used	CPU time	In memory	On Disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Store serialized representation in memory!

```
val result = input.map(x => x*x)
result.persist(MEMORY_ONLY)
```

NOTE: .cache() = .persist(**MEMORY_ONLY**)

Running out of cache? **LRU caching** = Least Recently Used

Experimental: **OFF_HEAP** => store in Tachyon (in-memory filesystem)

(Lazy) Transformations on RDDs

Function Name	Purpose	Example	Result
map	Apply a function to each element in the RDD and return an RDD of the result	`rdd.map(x => x + 1)`	{2, 3, 4, 4}
flatMap	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	{1, 2, 3, 2, 3, 3, 3}
filter	Return an RDD consisting of only elements which pass the condition passed to filter	`rdd.filter(x => x != 1)`	{2, 3, 3}
distinct	Remove duplicates	`rdd.distinct()`	{1, 2, 3}
sample(withReplacement, fraction, [seed])	Sample an RDD	`rdd.sample(false, 0.5)`	non-deterministic

Arguments are (lambda) functions

```
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> rdd.map(lambda x: x * 2)
RDD: [1, 2, 3, 4] → [2, 4, 6, 8]

>>> rdd.filter(lambda x: x % 2 == 0)
RDD: [1, 2, 3, 4] → [2, 4]
```

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.map(lambda x: [x, x+5])
RDD: [1, 2, 3] → [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(lambda x: [x, x+5])
RDD: [1, 2, 3] → [1, 6, 2, 7, 3, 8]
```

(Lazy) Transformations on Pair RDDs

Same API PLUS KV transformations

`groupByKey()`	Group together values with the same key	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`reduceByKey(func)`	Combine values with the same key together	`rdd.reduceByKey( (x, y) => x + y)`	`{(1, 2), (3, 10)}`
`mapValues(func)`	Apply a function to each value of a Pair RDD without changing the key	`rdd.mapValues(x => x+1)`	`{(1, 3), (3, 5), (3, 7)}`
`keys()`	Return an RDD of just the keys	`rdd.keys()`	`{1, 3, 3}`
`values()`	Return an RDD of just the values	`rdd.values()`	`{2, 4, 6}`
`sortByKey()`	Returns an RDD sorted by the key	`rdd.sortByKey()`	`{(1, 2), (3, 4), (3, 6)}`

```
>>> rdd.reduceByKey(lambda a, b: a + b)
RDD: [(1,2), (3,4), (3,6)] → [(1,2), (3,10)]
```

Actions trigger execution!

Function Name	Purpose	Example	Result
collect()	Return all elements from the RDD	`rdd.collect()`	{1, 2, 3, 3}
count()	Number of elements in the RDD	`rdd.count()`	4
take(num)	Return num elements from the RDD	`rdd.take(2)`	{1, 2}
takeOrdered(num)(ordering)	Return num elements based on providing ordering	`rdd.takeOrdered(2)(myOrdering)`	{3, 3}
takeSample(withReplacement, num, [seed])	Return num elements at random	`rdd.takeSample(false, 1)`	non-deterministic
reduce(func)	Combine the elements of the RDD together in parallel (e.g. sum)	`rdd.reduce((x, y) => x + y)`	9

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.reduce(lambda a, b: a * b)
Value: 6
```

Shared variables

- Function closures containing global variables

  => have to be resent for every job!

  - for example: large lookup table!

pySpark Shared Variables

- Broadcast Variables
  - » Efficiently send large, *read-only* value to all workers
  - » Saved at workers for use in one or more Spark operations
  - » Like sending a large, read-only lookup table to all the nodes

- Accumulators
  - » Aggregate values from workers back to driver
  - » Only driver can access value of accumulator
  - » For tasks, accumulators are write-only
  - » Use to count errors seen in RDD across workers

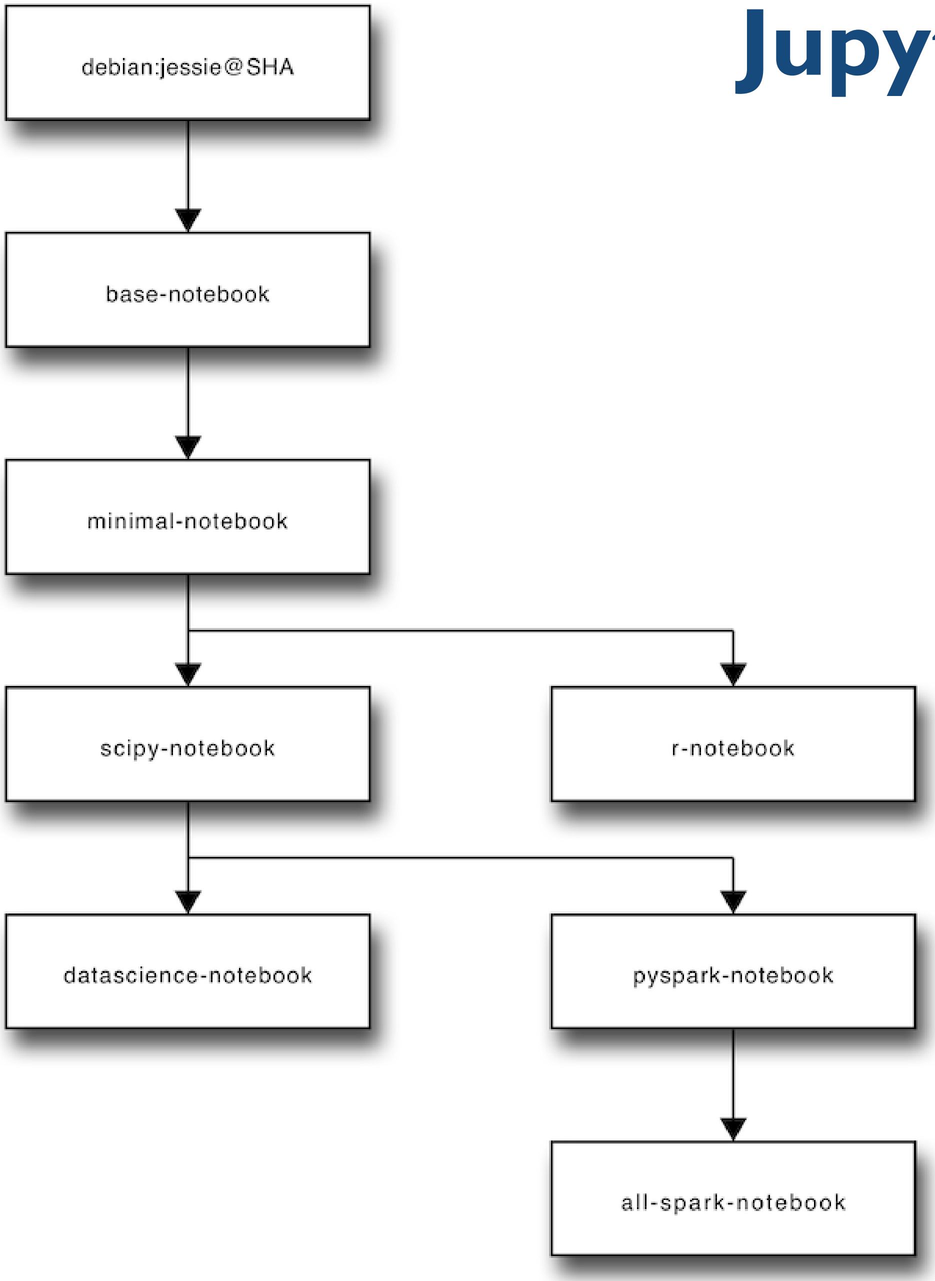
Broadcast variables & Accumulators

```
At the driver:
>>> broadcastVar = sc.broadcast([1, 2, 3])

At a worker (in code passed via a closure)
>>> broadcastVar.value
[1, 2, 3]

>>> accum = sc.accumulator(0)
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> def f(x):
>>>   global accum
>>>   accum += x

>>> rdd.foreach(f)
>>> accum.value
Value: 10
```

A Spark Playground for everyone?

Jupyter docker stacks!

https://github.com/jupyter/docker-stacks



- requires docker (OS independent)
- single command installation
- works both local or with a cluster
- has all relevant python libraries installed

New to python? No problem!

- Cheat sheet with the basics to get you started:
- https://www.google.be/url?sa=t&rct=j&q=&esrc=s&source=web&cd=10&ved=0ahUKEwia0MWn3u3NAhWCJsAKHU8BAyMQFghnMAk&url=http%3A%2F%2Fwww.cogsci.rpi.edu%2F~destem%2Figd%2Fpython_cheat_sheet.pdf&usg=AFQjCNFN9vxq3S7TXrRu6JJIjzfZtc4qiQ&sig2=ZTs-TyniW7QBDbPzXw0TDQ&cad=rja

  - Have a look at control structures: if, for
  - Have a look at function definitions
  - Have a look at lists, dictionaries
  - Other things are mostly provided in the notebooks

Dataframes API

Definition

DataFrame

noun – [dey-tuh-freym]

1. A distributed collection of rows organized into named columns.
2. An abstraction for selecting, filtering, aggregating and plotting structured data (*cf. R, Pandas*).
3. Archaic: Previously SchemaRDD (*cf. Spark < 1.3*).

Python data frames

Motivation

- Less lines of code
- Higher level operations on tuples
- Unified interface on different data sources
- Interoperability with pandas_df and RDDs
- Language-agnostic performance
- Tungsten query plan optimizer

Less LOC

RDD API

```
pdata.map(lambda x: (x.dept, [x.age, 1])) \
  .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]])
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()
```

Guess what happens if we have 10 columns!?

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	43	B Jones
Chem	61	M Kennedy

DataFrame API

```
data.groupBy("dept").avg("age")
```

Higher level operations on tuples

- Filtering rows, selecting certain columns ...

```
c.filter(c.authorEmail.like("rxin%")).select(c.authorDate, c.subject)
```

- Aggregations on tuples (count, average, sum,...)

```
c.groupBy("authorName").count().sort(desc("count"))
```

- Joining, sorting, custom aggregations, ...

*number of authors per file in Github

```
f.join(c, f.commitHash == c.commitHash)
 .groupBy("path").agg(col("path"), countDistinct("authorName").alias("numAuthors"))
 .sort(desc("numAuthors"))
```

- User defined functions

```
toDate = udf(lambda x: datetime.utcfromtimestamp(float(x)), DateType())
c.select(toDate(c.authorDate))
```

Unified interface on different data sources

built-in

Parquet
JDBC
{ JSON }
PostgreSQL
HIVE
MySQL
HDFS
amazon web services S3
H2

external

AVRO
CSV
dBase
APACHE HBASE
elasticsearch.
cassandra
amazon web services Amazon Redshift
and more …

Unified interface for reading/writing

```
df = sqlContext.read \
  .format("json") \
  .option("samplingRatio", "0.1") \
  .load("/home/michael/data.json")

df.write \
  .format("parquet") \
  .mode("append") \
  .partitionBy("year") \
  .saveAsTable("fasterData")
```

Builder methods specify:

- Format
- Partitioning
- Handling of existing data

Automatic schema inference

```
# If the underlying format has self-describing schema, DataFrames will use that
# For JSON, it will automatically infer the schema based on the data.
tweets = sqlContext.load("/home/rxin/tweets-demo.json", "json")
```

Command took 2.22s

```
tweets.printSchema()
```

```
root
 |-- coordinates: struct (nullable = true)
 |    |-- coordinates: array (nullable = true)
 |    |    |-- element: double (containsNull = false)
 |    |-- type: string (nullable = true)
 |-- created_at: string (nullable = true)
```

Compatibility with Pandas

- Easy to convert between Pandas and pySpark
  » *Note: pandas DataFrame must fit in driver*

```
# Convert Spark DataFrame to Pandas
pandas_df = spark_df.toPandas()

# Create a Spark DataFrame from Pandas
spark_df = context.createDataFrame(pandas_df)
```

Compatibility with RDDs

```
rdd = sc.parallelize(range(10)).map(lambda x: (str(x), x))
kvdf = rdd.toDF(["key", "value"])
```

Language-agnostic performance!

Performance is language independent!!!