# SparkSQL and DataFrames
## Simple and Fast Analytics on Structured Data

**Nagiza F. Samatova**, samatova@csc.ncsu.edu
Professor, Department of Computer Science
North Carolina State University

Senior Scientist, Computer Science & Mathematics Division
Oak Ridge National Laboratory

# Structured Data & Spark SQL

**Structured data is any data that has a schema—that is, a known set of fields for each record. When you have this type of data, Spark SQL makes it both easier and more efficient to load and query. In particular, Spark SQL provides three main capabilities:**

1. Load data from a variety of structured sources (e.g., JSON, Hive, and Parquet).
2. Query the data using SQL, both inside a Spark program and from external tools that connect to Spark SQL through standard database connectors (JDBC/ODBC), such as business intelligence tools like Tableau.
3. When used within a Spark program, Spark SQL provides rich integration between SQL and regular Python/Java/Scala code, including the ability to join RDDs and SQL tables, expose custom functions in SQL, and more. Many jobs are easier to write using this combination.

# DataFrames: Unified Data Abstraction

SparkSQL provides unified data abstraction through DataFrames (previously known as SchemaRDD).

# Why SparkSQL?

- Rich language bindings in Scala, Python, Java, R
- DataFrames – best data abstraction for *selecting*, *filtering*, *aggregating*, and *plotting* structured data.
- Write Less code, Read less data, Optimizer does all the hard work!
- No performance overhead in using Python, R or Java APIs - All of them use the same SparkSQL engine underneath.
- Interoperable - Spark RDDs and Data Frames
- Hive Support – Run HiveQL queries, access Hive UDFs, UDAFs, SerDes
- Supports various data formats - JSON, Parquet, JDBC, mySQL, HDFS, S3, H2, Hive, etc.

# Write Less Code: Compute Average( )

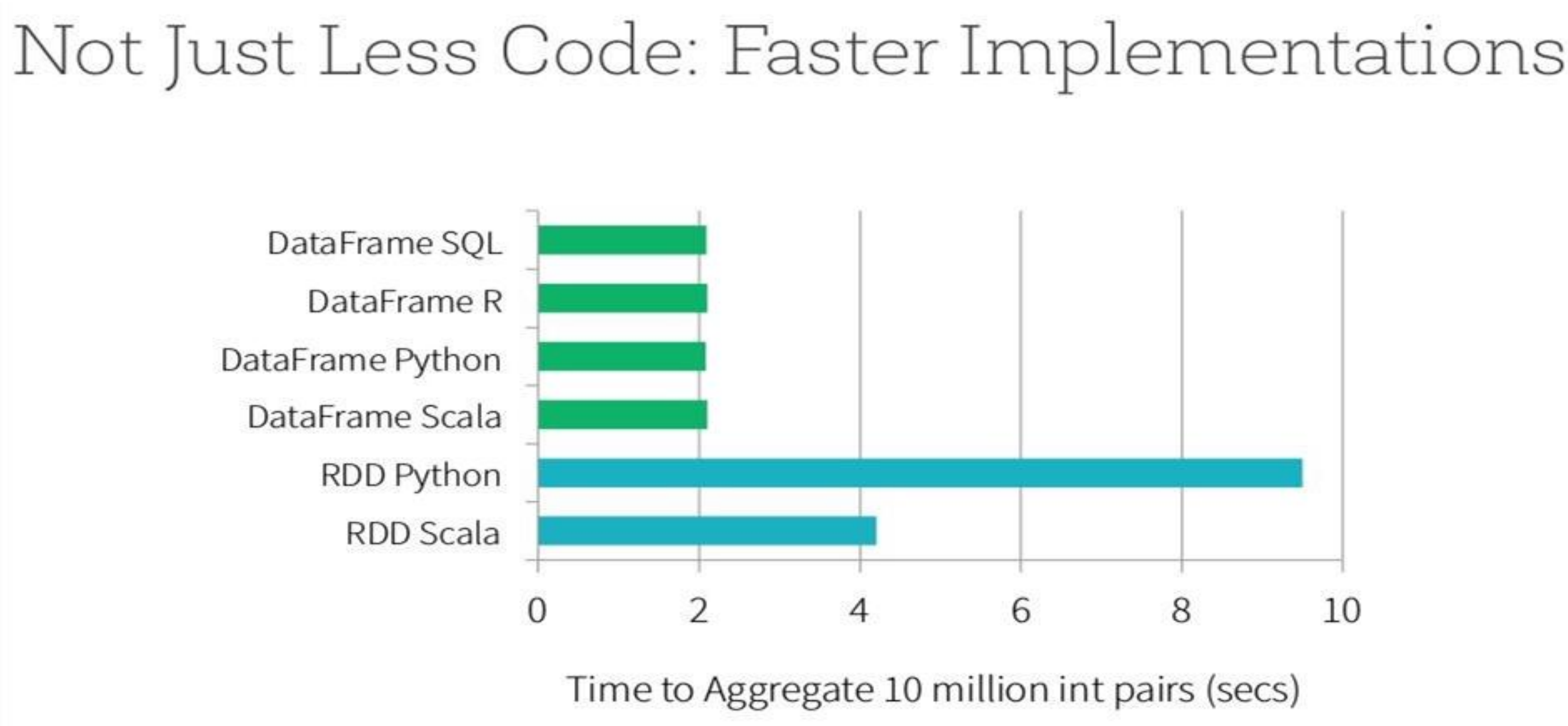
Using RDDs

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 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()
```

Using DataFrames

```
sqlCtx.table("people") \
   .groupBy("name") \
   .agg("name", avg("age")) \
   .collect()
```

# Better Performance!



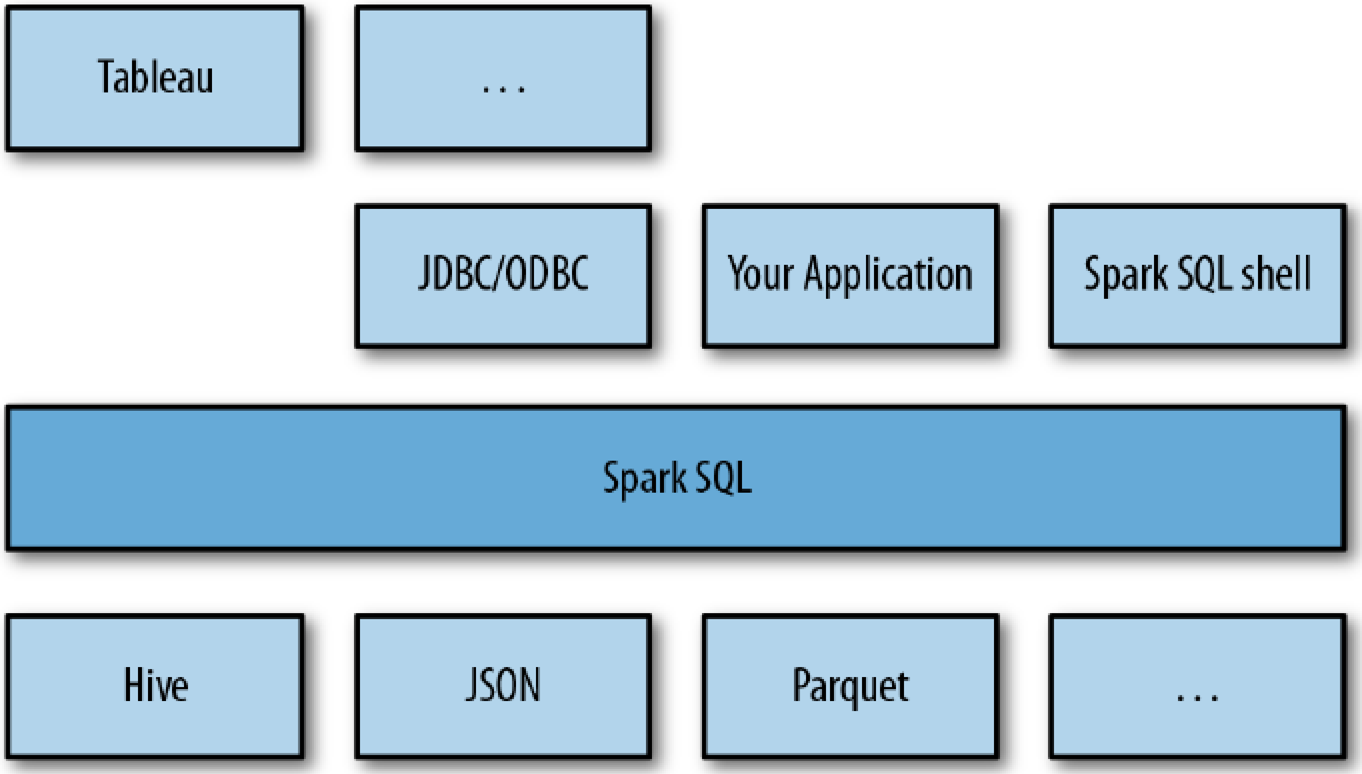
# Using SparkSQL in Applications

- **The most powerful way to use Spark SQL is inside a Spark application:**
  - This gives us the power to easily **load data** and **query it with SQL** while simultaneously **combining it with "regular" program code** in Python, Java, or Scala.

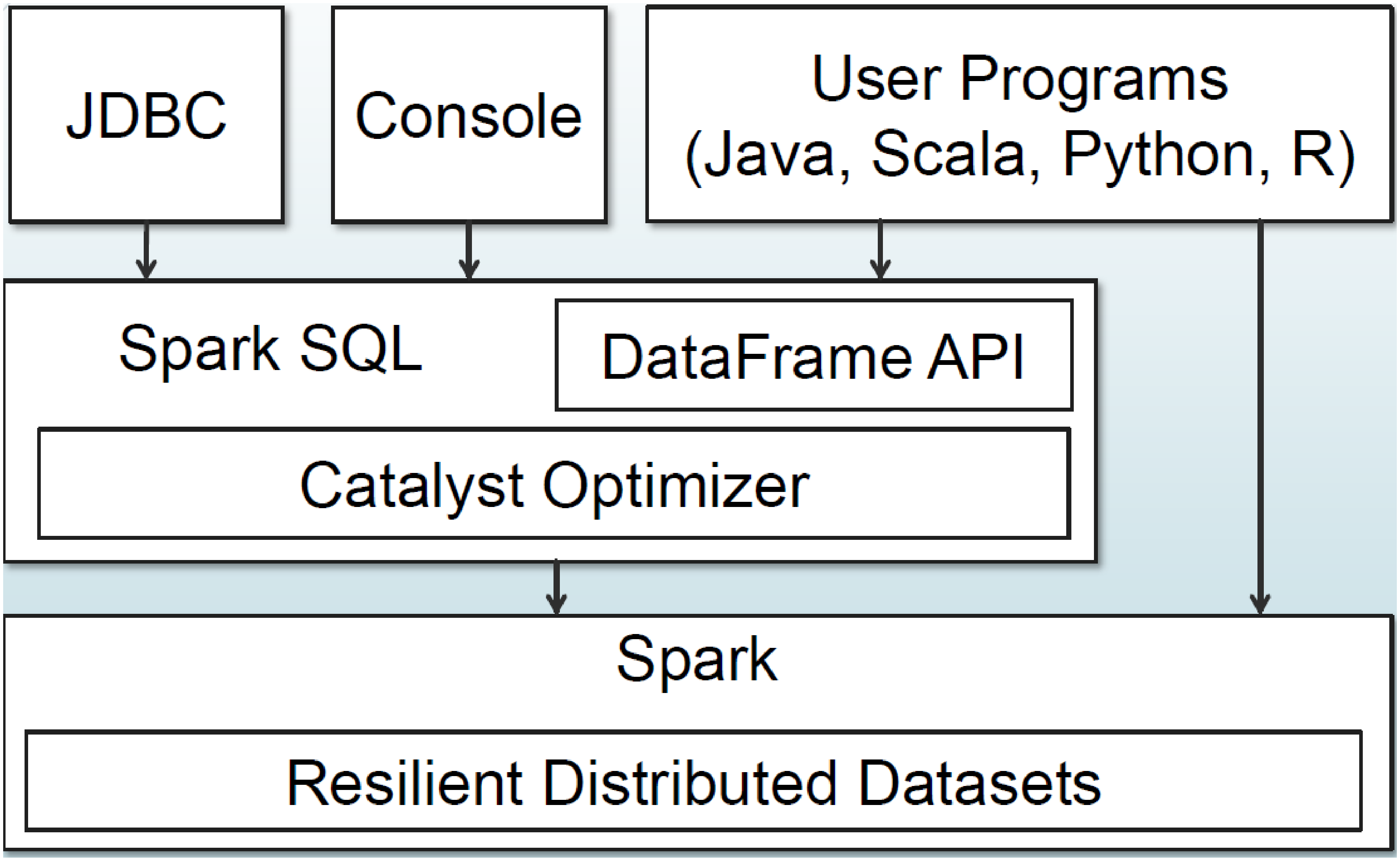
- **To use SparkSQL, construct a HiveContext (or SQLContext for a stripped-down version) based on the SparkContext.**
  - This context provides additional functions for querying and interacting with Spark SQL data.

- **Using the HiveContext, one can build DataFrames, which represent the structured data, and operate on them with SQL or with normal RDD operations like map().**

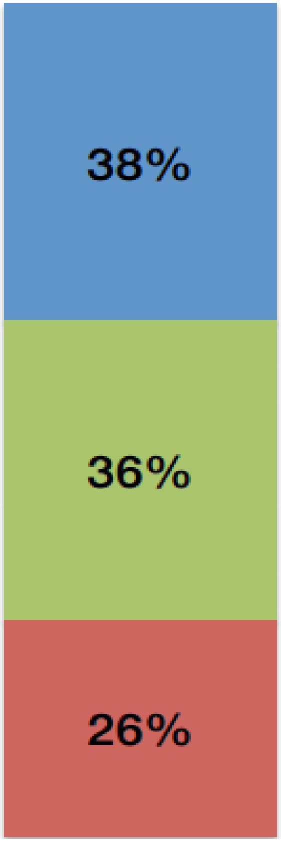
# SparkSQL within the Spark Software Stack



# SparkSQL Components

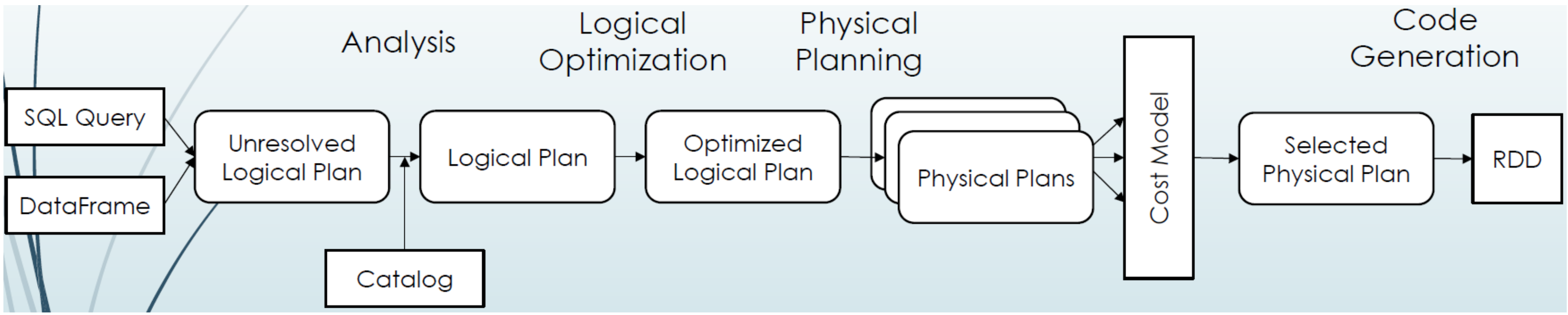


# SparkSQL Components




- Catalyst Optimizer
  - Relational algebra + expressions
  - Query optimization
- Spark SQL Core
  - Execution of queries as RDDs
  - Reading in Parquet, JSON …
- Hive Support
  - HQL, MetaStore, SerDes, UDFs

# Catalyst Optimizer



# SparkSQL Features Support

- **Data types:**
  - Boolean, Integer, Double, Decimal, String, Date, Timestamp
  - Complex data types: Struct, Array, Map, Union
- **In-memory caching**
- **User-defined functions**

# How to Setup SparkSQL?

**Create a basic SQLContext, all you need is a SparkContext:**

– ./bin/pyspark will create a SparkContext named sc

```
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext (sc)
```

**Create a HiveContext to provide additional functionalities:**

- write queries using the more complete HiveQL parser,
- access to Hive UDFs, and
- the ability to read data from Hive tables.

**Hive support is enabled by adding the -Phive and -Phive-thriftserver flags while building Spark. Configuration of Hive is done by placing your hive-site.xml file in conf/ folder.**

```
from pyspark.sql import HiveContext, Row
sqlContext = HiveContext (sc)
```

# SpakSQL Basic API

**example.json:**

```
{'x': 0, 'y': 1}
{'x': 1, 'y': 2}
```

```
points = sqlCtx.read.json('example.json')
# show table:
points.show()
# print schema:
points.printSchema()
# filter:
points.filter('x = 1')
# two different select types:
points.select('x', points['y'] + 1)
# both:
points.filter('x = 1').select('x')
# other:
dir(points)
```

# Basic Query Example

- **To make a query against a table, call the sql() method on the HiveContext or SQLContext:**
  - The first thing to do is to tell Spark SQL about some data to query.
  - Ex: Load some Twitter data from JSON, and give it a name by registering it as a "temporary table" so we can query it with SQL.
- **Then select the top tweets by retweetCount**

```
input = hiveCtx.jsonFile(inputFile)
# Register the input schema RDD
input.registerTempTable("tweets")
# Select tweets based on the retweetCount
topTweets = hiveCtx.sql("""SELECT text, retweetCount  FROM
  tweets ORDER BY retweetCount LIMIT 10""")
```

# Table Registration with registerTempTable()

```
points.registerTempTable('points') # finally!
sqlCtx.sql("SELECT * FROM points").show() # awesome!
sqlCtx.sql("SELECT y FROM points WHERE x = 1").show()
```

# DataFrame

- **Under the hood, SparkSQL is based on an extension of the RDD model called a DataFrame.**
- **A DataFrame contains an RDD of Row objects:**
    - Each Row object represents a record.
    - Row objects are just wrappers around arrays of basic types (e.g., integers and strings)
    - DataFrame knows the schema (i.e., data fields) of its columns
- **DataFrames provide a way to access the RDD (by calling rdd()):**
    - so you can operate on the using existing RDD transformations such as map() and filter()
- **DataFrames store data in a more efficient manner than native RDDs, taking advantage of their schema.**
- **DataFrames provide new operations not available on RDDs, such as the ability to run SQL queries.**
- **DataFrames can be created from:**
    - external data sources, by loading the data as a DataFrame
    - the results of queries, by executing queries
    - regular RDDs, by converting RDDs into DataFrames

# DataFrames ~ Tables in DBMS

- DataFrames are similar to tables in a traditional database
- You can register any DataFrame as a temporary table to query it via HiveContext.sql or SQLContext.sql
- You do so by using the DataFrame's registerTempTable() method

# Transformations on DataFrames

- **DataFrames provide a number of operations that operate directly on the DataFrame, without having to register as a temporary table or manipulate the underlying RDD.**
- **show() brings DataFrame back to the local machine and displays the results**
  - Similar to collect() on RDDs
- **The other functions behave much like their SQL counterparts**
  - **e.g. df.select ( columns ) is like SELECT in SQL**
- **For more info, consult API:**
- http://spark.apache.org/docs/1.3.0/api/python/pyspark.sql.html#pyspark.sql.DataFrame

# Basic DataFrame Operations

**DataFrame with Name, Age { Row(“Bear”, None), Row (“Databricks”, 1) }**

| Function Name | Purpose | Example |
|---|---|---|
| show() | Show the contents of the DataFrame | df.show() |
| select() | Select the specified fields/functions | df.select(“name”, df(“age”)+1 |
| filter() | Select only the rows meeting the criteria | df.filter (df(“age”) > 19) |
| groupBy() | Group together on a column, needs to be followed by an aggregation like min(), max(), mean() or agg(). | df.groupBy(df(“name”)) .min() |

# Types Stored by DataFrames

| SparkSQL/ HiveQL type | Python | SparkSQL HiveQL type | Python |
|---|---|---|---|
| TINYINT | int/long(in range of -128 to 127 | SMALLINT | int/long (in range of -32768 to 32767) |
| INT | int or long | BIGINT | long |
| FLOAT | float | DOUBLE | float |
| DECIMAL | decimal.Decimal | STRING | string |
| BINARY | bytearray | BOOLEAN | bool |
| TIMESTAMP | datetime.datetime | ARRAY<DATA_TYPE> | list, tuple, or array |
| MAP<KEY_TYPE, VAL_TYPE> | dict | STRUCT<COL1:COL1_TYPE, ...> | Row |

- **The structure is simply represented as other Rows in SparkSQL**
- **All of these types can also be nested within each other; you can have arrays of structs, or maps that contain structs**

# DataFrames API: Reading Input Data

```
# SQLContext wraps around a SparkContext
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

# Create the DataFrame
# Note: the schema is automatically inferred from the data !!!
df = sqlContext.read.json (
    "examples/src/main/resources/people.json" )

# Show the content of the DataFrame
df.show()

# Print the schema in a tree format
df.printSchema()
```

# Dataframes: Select, Filter, Aggregate

```
# Select only the "name" column
df.select ( "name" ).show()

# Select everybody, but increment the age by 1
df.select ( df['name'], df['age'] + 1 ).show()


# Select people older than 21
df.filter ( df['age'] > 21).show()

# Count people by age
df.groupBy ( "age" ).count().show()
```

# Converting between DataFrames and RDDs

- **Row objects represent records inside DataFrames, and are simply fixed-length arrays of fields**
  - In Python, Row objects are a bit different since we don't have explicit typing
  - We just access the $i^{\text{th}}$ element using row [i]
- **Python Rows support named access to their fields, of the form `row.column_name`**

```
topTweetText = topTweets.rdd().map( lambda row: row.text )
```

# DataFrames from RDDs

**Create RDD of Row objects and then call inferSchema()**

```
happyPeopleRDD = sc.parellize ( [Row(name=“holden”,
                                 favoriteBeverage = “coffee”)] )

happyPeopleDataFrame = hiveCtx.inferSchema ( happyPeopleRDD )

happyPeopleDataFrame.registerTempTable (“happy_people”)
```

# From RDDs to DataFrames

**Two ways of converting RDDs to DataFrames:**

1. Inferring schema using Reflection
2. Programmatically Specifying Schema

## 1. Inferring Schema using Reflection:

a. Used when the schema is already known while writing the Spark application.
b. Types of the columns are inferred by looking at the contents of the first row. (So, the first row columns cannot be NULL -- a limitation!)

## 2. Programmatically Specifying Schema:

a. This conversion method is used when the ***schema is not known until runtime*** (e.g.: the structure of records is encoded in a string)

# Example: Inferring Schema using Reflection

```
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext (sc)

# Load a text file
lines = sc.textFile( "examples/src/main/resources/people.txt" )

# Split each line by comma and convert it to a Row
parts = lines.map ( lambda l: l.split(",") )
people = parts.map ( lambda p: Row(name=p[0], age=int(p[1])))  #<--
    Schema was already known

# Infer the schema, and register the DataFrame as a table
schemaPeople = sqlContext.createDataFrame ( people )
schemaPeople.registerTempTable ( "people" )
```

# Ex (cont.): Inferring Schema using Reflection

```
# SQL can be run over DataFrames that have been registered as a table
teenagers = sqlContext.sql ( "SELECT name
                              FROM people
                              WHERE age >= 13 AND age <= 19" )

# The results of SQL queries are RDDs and support all the normal RDD
    operations
teenNames = teenagers.map ( lambda p: "Name: " + p.name )
for teenName in teenNames.collect ():
    print ( teenName )
```

# Ex: Programmatically Specifying Schema

```
# Import SQLContext and data types
from pyspark.sql import SQLContext
from pyspark.sql.types import *

# sc is an existing SparkContext
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a tuple
lines = sc.textFile ( "examples/src/main/resources/people.txt" )
parts = lines.map ( lambda l: l.split(",") )
people = parts.map ( lambda p: (p[0], p[1].strip()) )

# The schema is encoded in a string
schemaString = "name age"
```

# Ex. (cont.): Programmatically Specifying Schema

```
# Schema: Name(string), Age(string)
# Age is of type string as the type could not be known before
fields = [StructField( field_name, StringType(), True ) for field_name in
   schemaString.split()]
schema = StructType ( fields )

# Apply the schema to the RDD
schemaPeople = sqlContext.createDataFrame ( people, schema )

# Register the DataFrame as a table
schemaPeople.registerTempTable ( "people")

# SQL can be run over DataFrames that have been registered as a table
results = sqlContext.sql ( "SELECT name FROM people" )

# SQL query results are RDDs and support all the normal RDD operations
names = results.map ( lambda p: "Name: " + p.name )
for name in names.collect():
   print ( name )
```

# Example: From RDD to Data Frames

```
lines = sc.textFile('example.txt')
points_raw = lines.map(lambda l: l.split(','))
points_raw_int = points_raw.map(lambda p: (int(p[0]), int(p[1])))

from pyspark.sql.types import IntegerType, StructField, StructType

fields = [StructField('x', IntegerType()), StructField('y', IntegerType())]
schema = StructType(fields)

points = sqlCtx.createDataFrame(points_raw_int, schema)
```

# Support for Multiple Data Formats


#Reading from a file in JSON format
df = sqlContext.read.load("examples/src/main/resources/people.json", format="json")

#Writing to a file in Parquet format
df.select("name", "age").write.save("namesAndAges.parquet", format="parquet", mode="overwrite")

# Loading and Saving Data

- SparkSQL supports a number of structured data sources out of the box, letting you get Row objects from them without any complicated loading process.
- These resources include Hive tables, JSON, and Parquet files.
- In addition, if you query these sources using SQL and select only a subset of the fields, SparkSQL can smartly scan only the subset of the data for those fields, instead of scanning all the data like a native Spark Context.hadoopFile might

# JSON

- If you have a JSON file with records fitting the same schema, SparkSQL can infer the schema by scanning the file and letting you access fields by name

```
{"name": "Holden"}
{"name":"Sparky The Bear", "lovesPandas":true, "knows":{"friends": ["holden"]}}
```

- If you have ever found yourself staring at a huge directory of JSON records, SparkSQL's schema inference is a very effective way to start working with the data without writing any special loading code
- To load JSON data, simply call the jsonFile() function on the hiveCtx

```
input = hiveCtx.jsonFile(inputFile)
```

# User Defined Functions

```
def foo(x): ...
def bar(x): ...

sqlCtx.udf.register('foo', foo, IntegerType())
sqlCtx.udf.register('bar', bar, BooleanType())
sqlCtx.sql("SELECT foo(x) FROM points WHERE bar(y)").show()
```

# Acknowledgements & References


- Anu Krishna Rajamohan, NCSU
- Anatoli Melechko, NCSU



- Learning Spark by Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia (http://shop.oreilly.com/product/0636920028512.do)
- Apache Spark Documentation
- SparkSQL and Dataframes 2015
- SparkSQL Training 2014