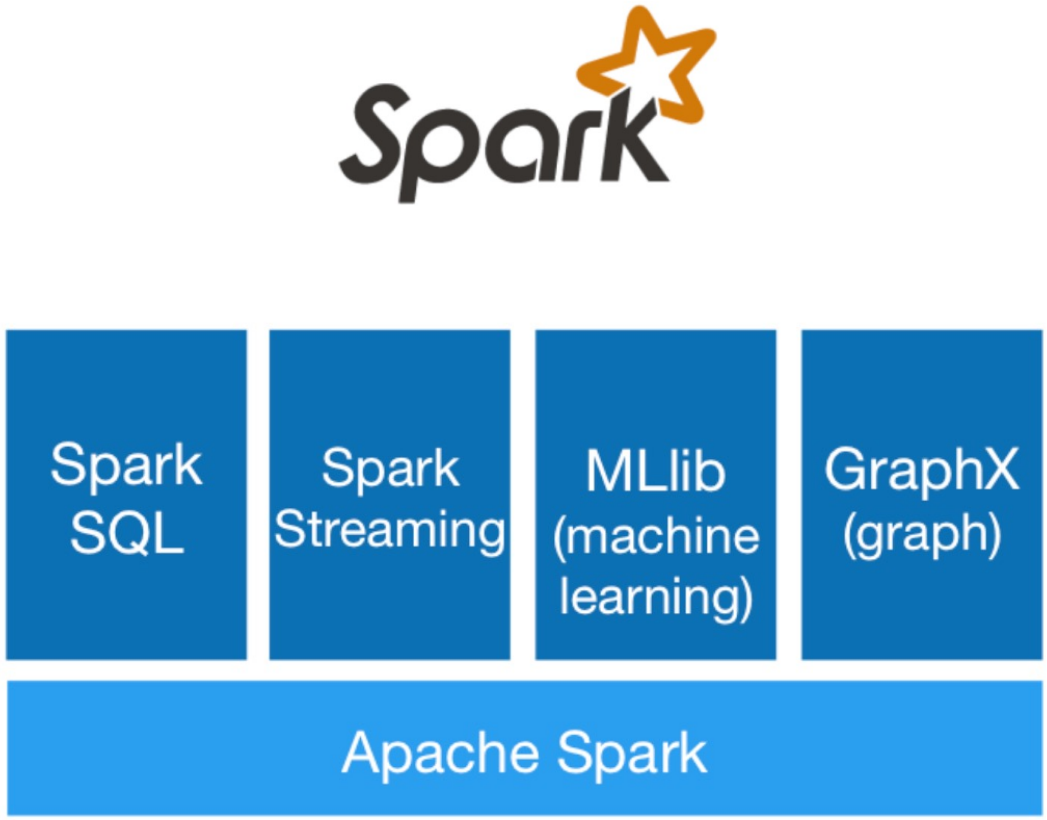
Spark SQL Introduction

Spark SQL is Spark's module for working with structured data.



Spark SQL is Spark's module for working with structured data.

Integrated

Seamlessly mix SQL queries with Spark programs.

Spark SQL lets you query structured data inside Spark programs, using either SQL or a familiar DataFrame API. Usable in Java, Scala, Python and R.

```
results = spark.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Apply functions to results of SQL queries.

Uniform Data Access

Connect to any data source the same way.

DataFrames and SQL provide a common way to access a variety of data sources, including Hive, Avro, Parquet, ORC, JSON, and JDBC. You can even join data across these sources.

```
spark.read.json("s3n://...")
  .registerTempTable("json")
results = spark.sql(
  """SELECT *
     FROM people
     JOIN json ...""")
```

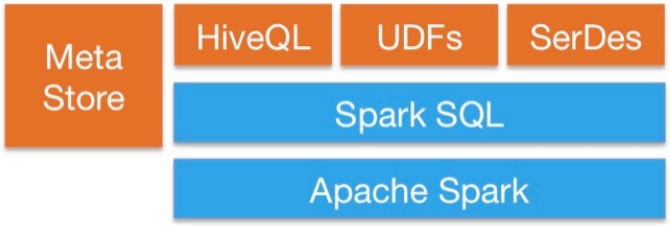
Query and join different data sources.

Spark SQL is Spark's module for working with structured data.

Hive Integration

Run SQL or HiveQL queries on existing warehouses.

Spark SQL supports the HiveQL syntax as well as Hive SerDes and UDFs, allowing you to access existing Hive warehouses.

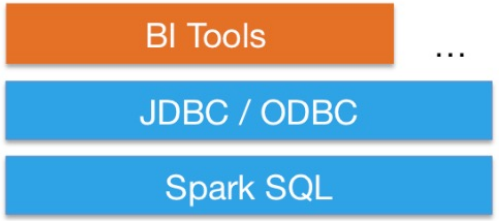


Spark SQL can use existing Hive metastores, SerDes, and UDFs.

Standard Connectivity

Connect through JDBC or ODBC.

A server mode provides industry standard JDBC and ODBC connectivity for business intelligence tools.



Use your existing BI tools to query big data.

Spark SQL API Programming Model

Spark SQL API for Python: **pyspark.sql**

Create DataFrame

DataFrame Analysis using SQL or DataFrame transformation Methods

Output DataFrame)

i **Structural/Semi-Structural Files**

1. CSV
2. JSON files
3. Parquet/ORC/RC files
4. Avro files
...

HDFS/S3/...

```
people = spark.read.csv(…) or
people = spark.read.format("csv")
                .load(…)
```

ii **RDB Table**

```
people = spark.read.jdbc(…)
```

RDB

iii **Python list or Pandas DataFrame**

```
[(1, "John", 20), (2, "May", 17), …]
```

(type: list)

(Pandas DataFrame)

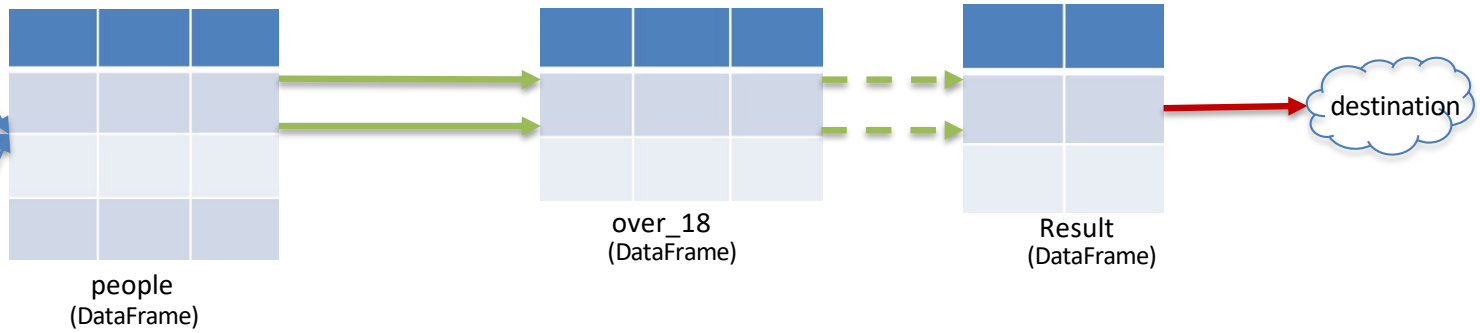
... There are more like creating DataFrame from Hive Metastore and converting RDD to DataFrame

i **Make queries with spark.sql(...)**

```
people.createOrReplaceTempView("people")
under_18 = spark.sql("select … from people where age >= 18")
```

i **DataFrame Action Methods**

```
result.show()
```



ii **DataFrame Transformation Methods**

```
under_18 = people.where("age >= 18")
```

More example DataFrame transformation methods:

```
.select(…)
.where()
.groupBy()
.orderBy()
.join()
.distinct()
…
```

More example DataFrame Action methods:

```
result.wirte.csv(…)
result.write.json(…)
result.write.parquet(…)
result.write.jdbc(…)
result.collect()
result.take(N)
result.toPandas()
…
```

Spark SQL API Programming Model

Spark SQL API for Python: **pyspark.sql**

Create DataFrame

DataFrame Analysis using SQL or DataFrame transformations

DataFrame Actions (outputs)

Make queries with spark.sql(...")

```
1  from pyspark.sql import SparkSession
2  from udf_lib import *
3
4  spark = SparkSession \
5      .builder \
6      .appName("GeoData") \
7      .getOrCreate()
8
9
10 report_input = spark.read.parquet("/data/")
11
12 report_input.createOrReplaceTempView("report_input")
13 result = spark.sql("""
14                     SELECT ad_id, region,SUM(imp),COUNT(DISTINCT imp_imei),SUM(click),COUNT(DISTINCT click_imei)
15                     FROM
16                     (
17                         SELECT
18                         ad_id,
19                         region,
20                         CASE WHEN log_type=1 THEN 1 ELSE 0 END AS imp,
21                         CASE WHEN log_type=1 THEN imei ELSE null END AS imp_imei,
22                         CASE WHEN log_type=2 THEN 1 ELSE 0 END AS click,
23                         CASE WHEN log_type=2 THEN imei ELSE null END AS click_imei,
24                         explode(get_geo_locations(ip_quadkey)) AS region
25                         FROM report_input
26                     ) src
27                     GROUP BY ad_id, region
28                     ORDER BY ad_id, region""")
29
30 result.write.csv("/output")
```

Spark SQL API Programming Model

Spark SQL API for Python: **pyspark.sql**

Create DataFrame

DataFrame Analysis using SQL or DataFrame transformations

DataFrame Actions (outputs)

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

spark = SparkSession \
    .builder \
    .appName("Crime Data") \
    .getOrCreate()

data = spark.read.csv(header=True, path="/data/")

crime_2015_6 = data.filter("year >= 2015")

boraobh_convictions = crime_2015_6.groupBy("borough") \
                                  .agg({"value": "sum"})

boraobh_convictions = boraobh_convictions.withColumnRenamed("sum(value)", "convictions")

total = boraobh_convictions.agg({"convictions": "sum"}) \
                           .collect()[0][0]

convictions_percent = boraobh_convictions.select("*", \
                      format_number(boraobh_convictions["convictions"] / total * 100, 2))

convictions_percent.show()
```

Use DataFrame Transformation Methods

Spark Program Run-time Excution Flow (In Spark SQL API)

Spark SQL API for Python:
pyspark.sql

```
spark-submit –master …  wordcount.py
```

Job

Tasks are internal instructions according to your codes

5

DataFrame's action method trigger real execution by submitting a job with tasks to the executors

The executors folk threads to run the tasks accordingly until all tasks are done

1-3

```
spark = SparkSession \
    .builder \
    .appName("GeoData") \
    .getOrCreate()

report_input = spark.read.parquet("/data/")

report_input.createOrReplaceTempView("report_input")
result = spark.sql("""
                SELECT ad_id, region,SUM(imp),COUNT(DISTINCT imp_imei),SUM(click),
                FROM
                (
                    SELECT
                    ad_id,
                    region,
                    CASE WHEN log_type=1 THEN 1 ELSE 0 END AS imp,
                    CASE WHEN log_type=1 THEN imei ELSE null END AS imp_imei,
                    CASE WHEN log_type=2 THEN 1 ELSE 0 END AS click,
                    CASE WHEN log_type=2 THEN imei ELSE null END AS click_imei,
                    explode(get_geo_locations(ip_quadkey)) AS region
                    FROM report_input
                ) src
                GROUP BY ad_id, region
                ORDER BY ad_id, region""")

result.write.csv("/output")
```

4

Driver

1 Ask for resources that meat this Spark program

M

2 Allocated node

S

Executor 3 Create spark "Executor"

T

(Row)
(Row)
(Row)

Standalone (/YARN/Mesos)

/data/ 1

/output/ 1 2 3

HDFS (/S3/…)

A Dive-in Example

```
from pyspark.sql import SparkSession

if __name__ == "__main__":
    # Create SparkSession
    spark = SparkSession \
        .builder \
        .getOrCreate()

    # Create a DataFrame from a JSON file
    stocks_df = spark.read.json("hdfs://devenv/user/spark/spark_sql_101/data/stocks.json")

    # use SQL to query teh DataFrame with spark.sql() methods
    stocks_df.createOrReplaceTempView("stocks")

    result_df = spark.sql("""select symbol, avg(open) as avg_open, avg(close) as avg_close, count(1) as rec_count
                             FROM stocks
                             GROUP BY symbol""")

    # Output result to console (Action)
    result_df.show()
```

```
In [27]: stocks_df.printSchema()
root
 |-- adj_close: double (nullable = true)
 |-- close: double (nullable = true)
 |-- date: string (nullable = true)
 |-- high: double (nullable = true)
 |-- low: double (nullable = true)
 |-- open: double (nullable = true)
 |-- symbol: string (nullable = true)
 |-- volume: long (nullable = true)
```

SparkSession

- SparkSession is the entry point to programming, which can be used to create DataFrames and execute SQL over tables, etc.
- An example of creating a SparkSession object

```
# Create a SparkSession object
spark = SparkSession \
        .builder \
        .appName("My Analysis") \
        .getOrCreate()
```

- The SparkSession is built on top of SparkContext with more functionality for Spark SQL library

```
# Get the SparkContext object from the SparkSession object
sc = spark.sparkContext
```

DataFrame

- A DataFrame object is a conceptually equivalent to a table.
  - The DataFrame is built on top of RDD with more functionality for Spark SQL
- The DataFrame object can be created from a wide range of sources. For examples:
  - Structured data files
    - JSON, CSV, … files
    - Parquet, ORC, RC, .. files
  - RDB tables
  - Local lists or Pandas DataFrames from the driver program

Data Types of DataFrame Columns (Primitive Types)

- Numeric types
  - `ByteType`: Represents 1-byte signed integer numbers. The range of numbers is from `-128` to `127`.
  - `ShortType`: Represents 2-byte signed integer numbers. The range of numbers is from `-32768` to `32767`.
  - `IntegerType`: Represents 4-byte signed integer numbers. The range of numbers is from `-2147483648` to `2147483647`.
  - `LongType`: Represents 8-byte signed integer numbers. The range of numbers is from `-9223372036854775808` to `9223372036854775807`.
  - `FloatType`: Represents 4-byte single-precision floating point numbers.
  - `DoubleType`: Represents 8-byte double-precision floating point numbers.
  - `DecimalType`: Represents arbitrary-precision signed decimal numbers. Backed internally by `java.math.BigDecimal`. A `BigDecimal` consists of an arbitrary precision integer unscaled value and a 32-bit integer scale.
- String type
  - `StringType`: Represents character string values.
- Binary type
  - `BinaryType`: Represents byte sequence values.
- Boolean type
  - `BooleanType`: Represents boolean values.
- Datetime type
  - `TimestampType`: Represents values comprising values of fields year, month, day, hour, minute, and second.
  - `DateType`: Represents values comprising values of fields year, month, day.

Data Types of DataFrame Columns (Complex Types)

- Complex types
  - `ArrayType(elementType, containsNull)`: Represents values comprising a sequence of elements with the type of `elementType`. `containsNull` is used to indicate if elements in a `ArrayType` value can have `null` values.
  - `MapType(keyType, valueType, valueContainsNull)`: Represents values comprising a set of key-value pairs. The data type of keys are described by `keyType` and the data type of values are described by `valueType`. For a `MapType` value, keys are not allowed to have `null` values. `valueContainsNull` is used to indicate if values of a `MapType` value can have `null` values.
  - `StructType(fields)`: Represents values with the structure described by a sequence of `StructFields` (`fields`).
    - `StructField(name, dataType, nullable)`: Represents a field in a `StructType`. The name of a field is indicated by `name`. The data type of a field is indicated by `dataType`. `nullable` is used to indicate if values of this fields can have `null` values.

DataFrame Methods - Transformation

- Use SQL or DataFrame methods to transform the table for data analysis.
  1. Use SQL

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

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

  2. Use DataFrame Transformations methods to transform the DataFrame step by step.

```
# Select everybody, but increment the age by 1
df.select(df['name'], df['age'] + 1).show()
# +-------+---------+
# |   name|(age + 1)|
# +-------+---------+
# |Michael|     null|
# |   Andy|       31|
# | Justin|       20|
# +-------+---------+
```

```
# Select people older than 21
df.filter(df['age'] > 21).show()
# +---+----+
# |age|name|
# +---+----+
# | 30|Andy|
# +---+----+
```

DataFrame Methods - Action

- To trigger the job to start in the cluster and get a result, use one of the actions. For example
  - show()
  - collect(), take(N)
  - toPandas()
  - The “write” property of DataFrame is of DataFrameWriter type
    - DataFrameWriter supports methods to output DataFrame data
    - csv(…), json(…)
    - parquet(…), orc(…), rc(…)
    - jdbc()
    - …

Other DataFrame methods

- DataFrame also has other utility methods
  - printSchema()
  - persist(), cache()
  - createOrReplaceTempView(…)
  - …

Functions

- Defined in pyspark.sql.functions module
  - A collections of built-in functions
  - There are normal, aggregation and table generating functions
  - Usages cover math functions, string functions, date functions, collection functions, etc.
- Spark SQL support user-defined functions
  - Normal UDF (Python, Scala, Java)
  - Aggregation UDFS (Scala, Java)

https://spark.apache.org/docs/2.4.5/api/python/pyspark.sql.html#module-pyspark.sql.functions

Create DataFrames from CSV Files

- Use **spark.read.csv()** method to load a CSV file or a folder that contain the CVS files and return a DataFrame

csv(*path, schema=None, sep=None, encoding=None, quote=None, escape=None, comment=None, header=None, inferSchema=None, ignoreLeadingWhiteSpace=None, ignoreTrailingWhiteSpace=None, nullValue=None, nanValue=None, positiveInf=None, negativeInf=None, dateFormat=None, timestampFormat=None, maxColumns=None, maxCharsPerColumn=None, maxMalformedLogPerPartition=None, mode=None, columnNameOfCorruptRecord=None, multiLine=None, charToEscapeQuoteEscaping=None, samplingRatio=None, enforceSchema=None, emptyValue=None*) [source]

https://spark.apache.org/docs/2.4.5/api/python/pyspark.sql.html#pyspark.sql.DataFrameReader

Commonly used arguments:

- header
- schema
- inferSchema
- sep
- mode

Demo

- Create data from CSV files and introduce commonly used arguments

Create DataFrames from JSON Files

- Use spark.read.json() method to load a JSON file or a folder that contain the JSON files and return a DataFrame

json(*path, schema=None, primitivesAsString=None, prefersDecimal=None, allowComments=None, allowUnquotedFieldNames=None, allowSingleQuotes=None, allowNumericLeadingZero=None, allowBackslashEscapingAnyCharacter=None, mode=None, columnNameOfCorruptRecord=None, dateFormat=None, timestampFormat=None, multiLine=None, allowUnquotedControlChars=None, lineSep=None, samplingRatio=None, dropFieldIfAllNull=None, encoding=None*) ¶ [source]

https://spark.apache.org/docs/2.4.5/api/python/pyspark.sql.html#pyspark.sql.DataFrameReader

Each line must contain a separate, self-contained valid JSON object.


```
{"symbol":"AAPL","date":"2008-01-02","open":199.27,"high":200.26,"low":192.55,"close":194.84,"volume":38542100,"adj_close":194.84}
{"symbol":"AAPL","date":"2007-01-03","open":86.29,"high":86.58,"low":81.9,"close":83.8,"volume":44225700,"adj_close":83.8}
{"symbol":"AAPL","date":"2006-01-03","open":72.38,"high":74.75,"low":72.25,"close":74.75,"volume":28829800,"adj_close":74.75}
{"symbol":"AAPL","date":"2005-01-03","open":64.78,"high":65.11,"low":62.6,"close":63.29,"volume":24714000,"adj_close":31.65}
{"symbol":"AAPL","date":"2004-01-02","open":21.55,"high":21.75,"low":21.18,"close":21.28,"volume":5165800,"adj_close":10.64}
{"symbol":"AAPL","date":"2003-01-02","open":14.36,"high":14.92,"low":14.35,"close":14.8,"volume":6479600,"adj_close":7.4}
{"symbol":"AAPL","date":"2002-01-02","open":22.05,"high":23.3,"low":21.96,"close":23.3,"volume":18910600,"adj_close":11.65}
```


Commonly used arguments:

- schema
- mode

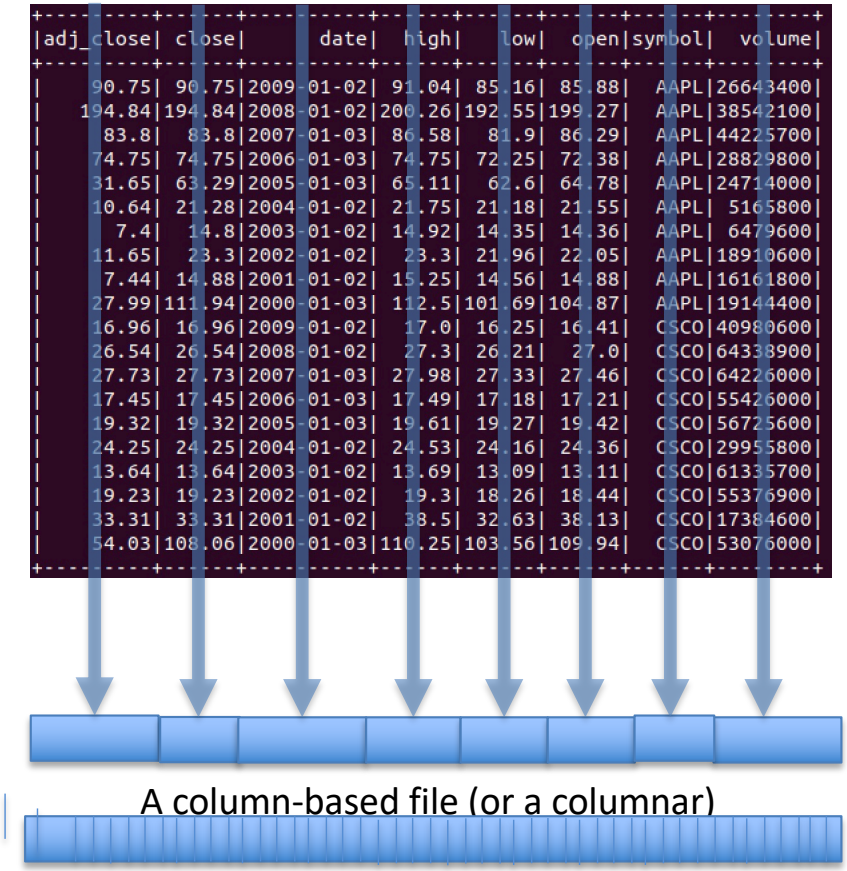
Demo

- Create DataFrame from JSON files

Create DataFrames from Parquet Files

- Parquet is a popular open source file format for Hadoop and several other big data tools
- Stores data in a **column-based** file format
  - Only a subset of the used columns are loaded while reading the data
- **Compression** support
  - Support various compression codecs
  - Column-based file format allows for better compression, as data is more homogeneous

The concept of how a column-based file is stored



*A columnar is especially **efficient when the data access pattern tends to be retrieving relatively fewer columns** among all

Demo

- Create DataFrame from Parquet files
- Convert other file formats to Parquet files for reducing data size to load by only loading needed columns

Create DataFrames from RDDs

- Spark SQL supports two different methods for converting existing RDDs into Datasets.
  1. The first method uses reflection to infer the schema of an RDD that contains specific types of objects.
  2. The second method for creating Datasets is through a programmatic interface that allows you to construct a schema and then apply it to an existing RDD.

Create DataFrames from RDDs

```python
from pyspark.sql import SparkSession
from pyspark.sql.types import *

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .getOrCreate()

    sc = spark.sparkContext

    # Load a text file and convert each line to a tuple.
    lines = sc.textFile("hdfs://devenv/user/spark/spark_sql_101/data/people.txt")
    parts = lines.map(lambda l: l.split(","))
    people = parts.map(lambda p: (p[0], int(p[1].strip())))

    # 1 The schema is specified using a StructType object
    schema = StructType([
        StructField("name",  StringType(), True),
        StructField("age", IntegerType(), True),
    ])

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

    schemaPeople.printSchema()

    schemaPeople.show()

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

    teenagers.show()

    # collect() is a DataFrame action that return a list of Rows
    for teenName in teenagers.collect():
        print(teenName)

    # 2 The schema is specified using a schema string
    schemaPeople2 = spark.createDataFrame(people, "name string, age int")
    schemaPeople2.show()
```

Create DataFrames from RDDs

```python
from pyspark.sql import SparkSession
from pyspark.sql.types import *

if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .getOrCreate()

    sc = spark.sparkContext

    # Load a text file and convert each line to a tuple.
    lines = sc.textFile("hdfs://devenv/user/spark/spark_sql_101/data/people.txt")
    parts = lines.map(lambda l: l.split(","))
    people = parts.map(lambda p: (p[0], int(p[1].strip())))

    # 1 The schema is specified using a StructType object
    schema = StructType([
        StructField("name",  StringType(), True),
        StructField("age", IntegerType(), True),
    ])

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

    schemaPeople.printSchema()

    schemaPeople.show()

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

    teenagers.show()

    # collect() is a DataFrame action that return a list of Rows
    for teenName in teenagers.collect():
        print(teenName)

    # 2 The schema is specified using a schema string
    schemaPeople2 = spark.createDataFrame(people, "name string, age int")
    schemaPeople2.show()
```

Demo

- Create DataFrame from RDD

Working on DataFrames with DataFrame Transformation Methods

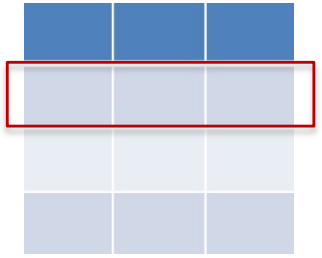
1. Use SQL language
    1. Register a table name for the DataFrame (or Dataset)
    2. Use Spark Session's sql() methods

       e.g. spark.sql(”select * from stocks”)
2. DataFrame Transformation methods
    – Transformations are the ones that produce a new DataFrame

Row

- A row in DataFrame.
  - The fields in it can be accessed:

```
>>> row = Row(name="Alice", age=11)
>>> row
Row(age=11, name='Alice')
>>> row['name'], row['age']
('Alice', 11)
>>> row.name, row.age
('Alice', 11)
>>> 'name' in row
True
>>> 'wrong_key' in row
False
```



https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.Row

Column

- The Column object is used to represent a specific column's information

```
# 1. Select a column out of a DataFrame

df.colName
df["colName"]
col("colName")
# 2. Create from an expression
df.colName + 1
1 / df.colName
```

https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.Column

Common DataFrame Transformation Methods

- Get familiar with DataFrame transformation methods by examples

Common DataFrame Transformation Methods

select(**cols*)

Projects a set of expressions and returns a new **DataFrame**.

Parameters: **cols** – list of column names (string) or expressions (**Column**). If one of the column names is '*', that column is expanded to include all columns in the current DataFrame.

```
>>> df.select('*').collect()
[Row(age=2, name=u'Alice'), Row(age=5, name=u'Bob')]
>>> df.select('name', 'age').collect()
[Row(name=u'Alice', age=2), Row(name=u'Bob', age=5)]
>>> df.select(df.name, (df.age + 10).alias('age')).collect()
[Row(name=u'Alice', age=12), Row(name=u'Bob', age=15)]
```

Common DataFrame Transformation Methods

filter(*condition*)

Filters rows using the given condition.

where() is an alias for **filter()**.

Parameters: **condition** – a **Column** of **types.BooleanType** or a string of SQL expression.

```
>>> df.filter(df.age > 3).collect()
[Row(age=5, name=u'Bob')]
>>> df.where(df.age == 2).collect()
[Row(age=2, name=u'Alice')]
```

```
>>> df.filter("age > 3").collect()
[Row(age=5, name=u'Bob')]
>>> df.where("age = 2").collect()
[Row(age=2, name=u'Alice')]
```

where(*condition*)

where() is an alias for **filter()**.

New in version 1.3.

Common DataFrame Transformation Methods

orderBy(**cols*, ***kwargs*) ¶

Returns a new **DataFrame** sorted by the specified column(s).

Parameters:
- **cols** – list of **Column** or column names to sort by.
- **ascending** – boolean or list of boolean (default True). Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, length of the list must equal length of the *cols*.

```
>>> df.sort(df.age.desc()).collect()
[Row(age=5, name=u'Bob'), Row(age=2, name=u'Alice')]
>>> df.sort("age", ascending=False).collect()
[Row(age=5, name=u'Bob'), Row(age=2, name=u'Alice')]
>>> df.orderBy(df.age.desc()).collect()
[Row(age=5, name=u'Bob'), Row(age=2, name=u'Alice')]
>>> from pyspark.sql.functions import *
>>> df.sort(asc("age")).collect()
[Row(age=2, name=u'Alice'), Row(age=5, name=u'Bob')]
>>> df.orderBy(desc("age"), "name").collect()
[Row(age=5, name=u'Bob'), Row(age=2, name=u'Alice')]
>>> df.orderBy(["age", "name"], ascending=[0, 1]).collect()
[Row(age=5, name=u'Bob'), Row(age=2, name=u'Alice')]
```

sort(**cols*, ***kwargs*) ¶

Returns a new **DataFrame** sorted by the specified column(s).

Common DataFrame Transformation Methods

join(*other*, *on=None*, *how=None*)

Joins with another **DataFrame**, using the given join expression.

Parameters:
- **other** – Right side of the join
- **on** – a string for the join column name, a list of column names, a join expression (Column), or a list of Columns. If *on* is a string or a list of strings indicating the name of the join column(s), the column(s) must exist on both sides, and this performs an equi-join.
- **how** – str, default 'inner'. One of *inner*, *outer*, *left_outer*, *right_outer*, *leftsemi*.

The following performs a full outer join between `df1` and `df2`.

```
>>> df.join(df2, df.name == df2.name, 'outer').select(df.name, df2.height).collect()
[Row(name=None, height=80), Row(name=u'Bob', height=85), Row(name=u'Alice', height=None)]
```

```
>>> df.join(df2, 'name', 'outer').select('name', 'height').collect()
[Row(name=u'Tom', height=80), Row(name=u'Bob', height=85), Row(name=u'Alice', height=None)]
```

```
>>> cond = [df.name == df3.name, df.age == df3.age]
>>> df.join(df3, cond, 'outer').select(df.name, df3.age).collect()
[Row(name=u'Alice', age=2), Row(name=u'Bob', age=5)]
```

```
>>> df.join(df2, 'name').select(df.name, df2.height).collect()
[Row(name=u'Bob', height=85)]
```

```
>>> df.join(df4, ['name', 'age']).select(df.name, df.age).collect()
[Row(name=u'Bob', age=5)]
```

Common DataFrame Transformation Methods

groupBy(**cols*)

Groups the **DataFrame** using the specified columns, so we can run aggregation on them. See **GroupedData** for all the available aggregate functions.

groupby() is an alias for **groupBy()**.

Parameters: **cols** – list of columns to group by. Each element should be a column name (string) or an expression (**Column**).

```
>>> df.groupBy().avg().collect()
[Row(avg(age)=3.5)]
>>> sorted(df.groupBy('name').agg({'age': 'mean'}).collect())
[Row(name=u'Alice', avg(age)=2.0), Row(name=u'Bob', avg(age)=5.0)]
>>> sorted(df.groupBy(df.name).avg().collect())
[Row(name=u'Alice', avg(age)=2.0), Row(name=u'Bob', avg(age)=5.0)]
>>> sorted(df.groupBy(['name', df.age]).count().collect())
[Row(name=u'Alice', age=2, count=1), Row(name=u'Bob', age=5, count=1)]
```

Common DataFrame Transformation Methods

agg(**exprs*)

Aggregate on the entire **DataFrame** without groups (shorthand for `df.groupBy.agg()`).

```
>>> df.agg({"age": "max"}).collect()
[Row(max(age)=5)]
>>> from pyspark.sql import functions as F
>>> df.agg(F.min(df.age)).collect()
[Row(min(age)=2)]
```

Common DataFrame Transformation Methods

limit(*num*)

Limits the result count to the number specified.

```
>>> df.limit(1).collect()
[Row(age=2, name=u'Alice')]
>>> df.limit(0).collect()
[]
```

Common DataFrame Transformation Methods

distinct()

Returns a new **DataFrame** containing the distinct rows in this **DataFrame**.

```
>>> df.distinct().count()
2
```

Common DataFrame Transformation Methods

withColumn(*colName*, *col*)

Returns a new **DataFrame** by adding a column or replacing the existing column that has the same name.

Parameters:
- **colName** – string, name of the new column.
- **col** – a **Column** expression for the new column.

```
>>> df.withColumn('age2', df.age + 2).collect()
[Row(age=2, name=u'Alice', age2=4), Row(age=5, name=u'Bob', age2=7)]
```

withColumnRenamed(*existing*, *new*)

Returns a new **DataFrame** by renaming an existing column. This is a no-op if schema doesn't contain the given column name.

Parameters:
- **existing** – string, name of the existing column to rename.
- **col** – string, new name of the column.

```
>>> df.withColumnRenamed('age', 'age2').collect()
[Row(age2=2, name=u'Alice'), Row(age2=5, name=u'Bob')]
```

Common DataFrame Transformation Methods

drop(**cols*)

Returns a new **DataFrame** that drops the specified column. This is a no-op if schema doesn't contain the given column name(s).

Parameters: **cols** – a string name of the column to drop, or a **Column** to drop, or a list of string name of the columns to drop.

```
>>> df.drop('age').collect()
[Row(name=u'Alice'), Row(name=u'Bob')]
```

```
>>> df.drop(df.age).collect()
[Row(name=u'Alice'), Row(name=u'Bob')]
```

```
>>> df.join(df2, df.name == df2.name, 'inner').drop(df.name).collect()
[Row(age=5, height=85, name=u'Bob')]
```

```
>>> df.join(df2, df.name == df2.name, 'inner').drop(df2.name).collect()
[Row(age=5, name=u'Bob', height=85)]
```

```
>>> df.join(df2, 'name', 'inner').drop('age', 'height').collect()
[Row(name=u'Bob')]
```

Common DataFrame Transformation Methods

dropna(*how='any'*, *thresh=None*, *subset=None*)

Returns a new **DataFrame** omitting rows with null values. **DataFrame.dropna()** and **DataFrameNaFunctions.drop()** are aliases of each other.

Parameters:

- **how** – 'any' or 'all'. If 'any', drop a row if it contains any nulls. If 'all', drop a row only if all its values are null.
- **thresh** – int, default None If specified, drop rows that have less than *thresh* non-null values. This overwrites the *how* parameter.
- **subset** – optional list of column names to consider.

```
>>> df4.na.drop().show()
+---+------+-----+
|age|height| name|
+---+------+-----+
| 10|    80|Alice|
+---+------+-----+
```

Common DataFrame Transformation Methods

fillna(*value*, *subset=None*)

Replace null values, alias for `na.fill()`. **`DataFrame.fillna()`** and **`DataFrameNaFunctions.fill()`** are aliases of each other.

Parameters:
- **value** – int, long, float, string, or dict. Value to replace null values with. If the value is a dict, then *subset* is ignored and *value* must be a mapping from column name (string) to replacement value. The replacement value must be an int, long, float, or string.
- **subset** – optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if *value* is a string, and subset contains a non-string column, then the non-string column is simply ignored.

```
>>> df4.na.fill(50).show()
+---+------+-----+
|age|height| name|
+---+------+-----+
| 10|    80|Alice|
|  5|    50|  Bob|
| 50|    50|  Tom|
| 50|    50| null|
+---+------+-----+
```

```
>>> df4.na.fill({'age': 50, 'name': 'unknown'}).show()
+---+------+-------+
|age|height|   name|
+---+------+-------+
| 10|    80|  Alice|
|  5|  null|    Bob|
| 50|  null|    Tom|
| 50|  null|unknown|
+---+------+-------+
```

Common DataFrame Transformation Methods

describe(**cols*)

Computes statistics for numeric and string columns.

This include count, mean, stddev, min, and max. If no columns are given, this function computes statistics for all numerical or string columns.

> **Note:** This function is meant for exploratory data analysis, as we make no guarantee about the backward compatibility of the schema of the resulting DataFrame.

```
>>> df.describe(['age']).show()
+-------+------------------+
|summary|               age|
+-------+------------------+
|  count|                 2|
|   mean|               3.5|
| stddev|2.1213203435596424|
|    min|                 2|
|    max|                 5|
+-------+------------------+
>>> df.describe().show()
+-------+------------------+-----+
|summary|               age| name|
+-------+------------------+-----+
|  count|                 2|    2|
|   mean|               3.5| null|
| stddev|2.1213203435596424| null|
|    min|                 2|Alice|
|    max|                 5|  Bob|
+-------+------------------+-----+
```

Demo

- Crime data analysis with DataFrame transformation API

Common DataFrame Actions

show(*n=20, truncate=True*)

Prints the first `n` rows to the console.

Parameters:
- **n** – Number of rows to show.
- **truncate** – If set to True, truncate strings longer than 20 chars by default. If set to a number greater than one, truncates long strings to length `truncate` and align cells right.

```
>>> df
DataFrame[age: int, name: string]
>>> df.show()
+---+-----+
|age| name|
+---+-----+
|  2|Alice|
|  5|  Bob|
+---+-----+
>>> df.show(truncate=3)
+---+----+
|age|name|
+---+----+
|  2| Ali|
|  5| Bob|
+---+----+
```

Common DataFrame Actions

`write`

Interface for saving the content of the non-streaming **`DataFrame`** out into external storage.

Returns: **`DataFrameWriter`**

DataFrameWriter.csv()

csv(*path*, *mode=None*, *compression=None*, *sep=None*, *quote=None*, *escape=None*, *header=None*, *nullValue=None*, *escapeQuotes=None*, *quoteAll=None*, *dateFormat=None*, *timestampFormat=None*)

Saves the content of the **DataFrame** in CSV format at the specified path.

Parameters:

- **path** – the path in any Hadoop supported file system
- **mode** –
  specifies the behavior of the save operation when data already exists.
  - `append`: Append contents of this **DataFrame** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` (default case): Throw an exception if data already exists.
- **compression** – compression codec to use when saving to file. This can be one of the known case-insensitive shorten names (none, bzip2, gzip, lz4, snappy and deflate).
- **sep** – sets the single character as a separator for each field and value. If None is set, it uses the default value, `,`.
- **quote** – sets the single character used for escaping quoted values where the separator can be part of the value. If None is set, it uses the default value, `"`. If you would like to turn off quotations, you need to set an empty string.
- **escape** – sets the single character used for escaping quotes inside an already quoted value. If None is set, it uses the default value, `\`
- **escapeQuotes** – A flag indicating whether values containing quotes should always be enclosed in quotes. If None is set, it uses the default value `true`, escaping all values containing a quote character.
- **quoteAll** – A flag indicating whether all values should always be enclosed in quotes. If None is set, it uses the default value `false`, only escaping values containing a quote character.
- **header** – writes the names of columns as the first line. If None is set, it uses the default value, `false`.

DataFrameWriter.json()

json(*path*, *mode=None*, *compression=None*, *dateFormat=None*, *timestampFormat=None*)

Saves the content of the **DataFrame** in JSON format at the specified path.

Parameters:

- **path** – the path in any Hadoop supported file system
- **mode** –
  specifies the behavior of the save operation when data already exists.
  - `append`: Append contents of this **DataFrame** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` (default case): Throw an exception if data already exists.
- **compression** – compression codec to use when saving to file. This can be one of the known case-insensitive shorten names (none, bzip2, gzip, lz4, snappy and deflate).
- **dateFormat** – sets the string that indicates a date format. Custom date formats follow the formats at `java.text.SimpleDateFormat`. This applies to date type. If None is set, it uses the default value value, `yyyy-MM-dd`.
- **timestampFormat** – sets the string that indicates a timestamp format. Custom date formats follow the formats at `java.text.SimpleDateFormat`. This applies to timestamp type. If None is set, it uses the default value value, `yyyy-MM-dd'T'HH:mm:ss.SSSZZ`.

```
>>> df.write.json(os.path.join(tempfile.mkdtemp(), 'data'))
```

DataFrameWriter.parquet()

parquet(*path*, *mode=None*, *partitionBy=None*, *compression=None*)

Saves the content of the **DataFrame** in Parquet format at the specified path.

Parameters:

- **path** – the path in any Hadoop supported file system
- **mode** –
  specifies the behavior of the save operation when data already exists.
  - `append`: Append contents of this **DataFrame** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` (default case): Throw an exception if data already exists.
- **partitionBy** – names of partitioning columns
- **compression** – compression codec to use when saving to file. This can be one of the known case-insensitive shorten names (none, snappy, gzip, and lzo). This will override `spark.sql.parquet.compression.codec`. If None is set, it uses the value specified in `spark.sql.parquet.compression.codec`.

```
>>> df.write.parquet(os.path.join(tempfile.mkdtemp(), 'data'))
```

DataFrameWriter.parquet()

parquet(*path*, *mode=None*, *partitionBy=None*, *compression=None*)

Saves the content of the **DataFrame** in Parquet format at the specified path.

Parameters:

- **path** – the path in any Hadoop supported file system
- **mode** –
  specifies the behavior of the save operation when data already exists.
  - `append`: Append contents of this **DataFrame** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` (default case): Throw an exception if data already exists.
- **partitionBy** – names of partitioning columns
- **compression** – compression codec to use when saving to file. This can be one of the known case-insensitive shorten names (none, snappy, gzip, and lzo). This will override `spark.sql.parquet.compression.codec`. If None is set, it uses the value specified in `spark.sql.parquet.compression.codec`.

```
>>> df.write.parquet(os.path.join(tempfile.mkdtemp(), 'data'))
```

DataFrameWriter.jdbc()

jdbc(*url*, *table*, *mode=None*, *properties=None*) ¶

Saves the content of the **DataFrame** to an external database table via JDBC.

> **Note:** Don't create too many partitions in parallel on a large cluster; otherwise Spark might crash your external database systems.

Parameters:

- **url** – a JDBC URL of the form `jdbc:subprotocol:subname`
- **table** – Name of the table in the external database.
- **mode** –
  specifies the behavior of the save operation when data already exists.
  - `append`: Append contents of this **DataFrame** to existing data.
  - `overwrite`: Overwrite existing data.
  - `ignore`: Silently ignore this operation if data already exists.
  - `error` (default case): Throw an exception if data already exists.
- **properties** – a dictionary of JDBC database connection arguments. Normally at least properties "user" and "password" with their corresponding values. For example { 'user' : 'SYSTEM', 'password' : 'mypassword' }

DataFrameWriter.mode()

mode(*saveMode*)

Specifies the behavior when data or table already exists.

Options include:

- *append*: Append contents of this **DataFrame** to existing data.
- *overwrite*: Overwrite existing data.
- *error*: Throw an exception if data already exists.
- *ignore*: Silently ignore this operation if data already exists.

```
>>> df.write.mode('append').parquet(os.path.join(tempfile.mkdtemp(), 'data'))
```

Write to RDB

- To get started you will need to include the JDBC driver for your database

Create a DF from a RDB table

```
jdbcDF2 = spark.read \
    .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
          properties={"user": "username", "password": "password"})
```

Write a DF to a RDB table

```
jdbcDF2.write \
    .jdbc("jdbc:postgresql:dbserver", "schema.tablename",
          properties={"user": "username", "password": "password"})
```

Demo

- Output DataFrame to to MySQL table

Function Types

- Types of functions
  - UDF (User-Defined Function):
    - UDF function applies to values of each involved row at a time
    - *e.g. format number(c, 2), substr(c), concat(c1,c2,),..*
  - UDAF (User-Defined Aggregation Function)
    - UDAF function applies to values of ALL involved rows at once for getting an aggregated result
    - *e.g. max(c), avg(c), stddev(c),…*

Create UDF Type Function

- You will also need to learn how to define your own functions to your domain-specific requirements.
- Currently, only UDF can be defined in Python but it is quite enough in most cases.
- To define a UDF is simple. There are 2 steps:

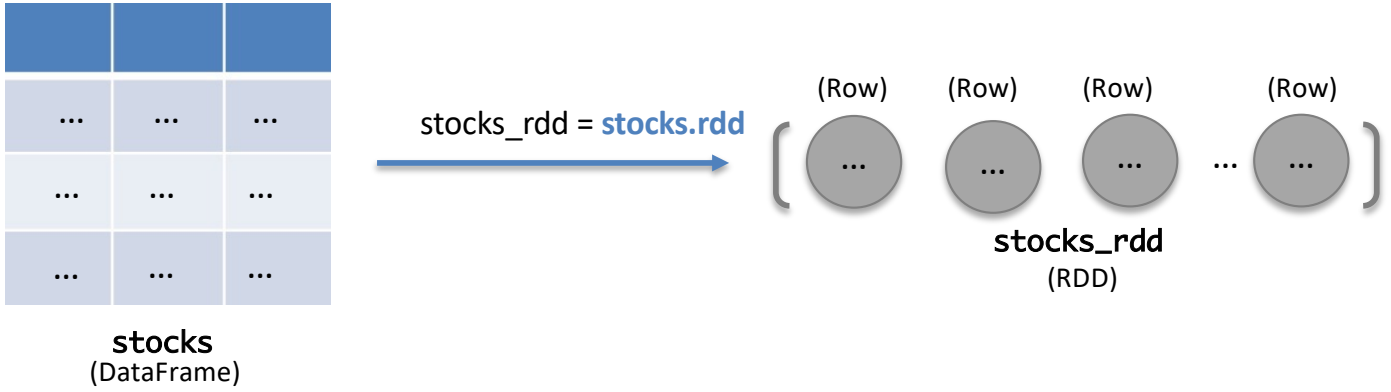
```
from pyspark.sql.functions import *
from pyspark.sql.types import *

# (1) Define a normal Python function and
# match arguments to your UDF (i.e. number of arguments and their types)
def slen_py(s):
    return len(s)

# (2) Register UDF function
spark.udf.register("slen", slen_py, IntegerType()) # for SQL
slen = udf(slen_py, IntegerType()) # for DataFrame APi
```

Demo: More examples

- Create a few UDFs to be used with SQL queries or DataFrame API

DataFrame to RDD

- A DataFrame object internally contains a RDD object member



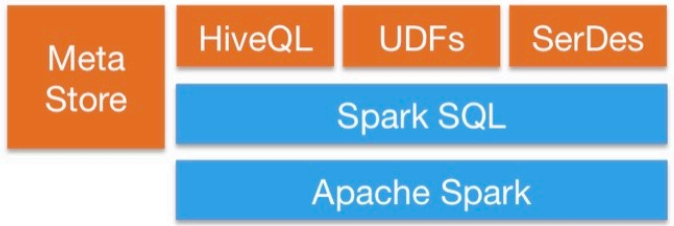

```
In [21]: stocks.show(5)
+---------+------+----------+------+------+------+------+--------+
|adj_close| close|      date|  high|   low|  open|symbol|  volume|
+---------+------+----------+------+------+------+------+--------+
|    90.75| 90.75|2009-01-02| 91.04| 85.16| 85.88|  AAPL|26643400|
|   194.84|194.84|2008-01-02|200.26|192.55|199.27|  AAPL|38542100|
|     83.8|  83.8|2007-01-03| 86.58|  81.9| 86.29|  AAPL|44225700|
|    74.75| 74.75|2006-01-03| 74.75| 72.25| 72.38|  AAPL|28829800|
|    31.65| 63.29|2005-01-03| 65.11|  62.6| 64.78|  AAPL|24714000|
+---------+------+----------+------+------+------+------+--------+
only showing top 5 rows


In [22]: stocks_rdd = stocks.rdd

In [23]: stocks_rdd.take(5)
Out[23]:
[Row(adj_close=90.75, close=90.75, date=datetime.date(2009, 1, 2), high=91.04000091552734, low=85.16000366210938, open=85.87999725341797, symbol='AAPL', volume=26643400),
 Row(adj_close=194.83999633789062, close=194.83999633789062, date=datetime.date(2008, 1, 2), high=200.25999450683594, low=192.5500030517578, open=199.27000427246094, symbol='AAPL', volume=38542100),
 Row(adj_close=83.80000305175781, close=83.80000305175781, date=datetime.date(2007, 1, 3), high=86.58000183105469, low=81.9000015258789, open=86.29000091552734, symbol='AAPL', volume=44225700),
 Row(adj_close=74.75, close=74.75, date=datetime.date(2006, 1, 3), high=74.75, low=72.25, open=72.37999725341797, symbol='AAPL', volume=28829800),
 Row(adj_close=31.649999618530273, close=63.290000915527344, date=datetime.date(2005, 1, 3), high=65.11000061035156, low=62.599998474121094, open=64.77999877929688, symbol='AAPL', volume=24714000)]
```

Spark SQL is Hive-Compatible

Hive Integration

Run SQL or HiveQL queries on existing warehouses.

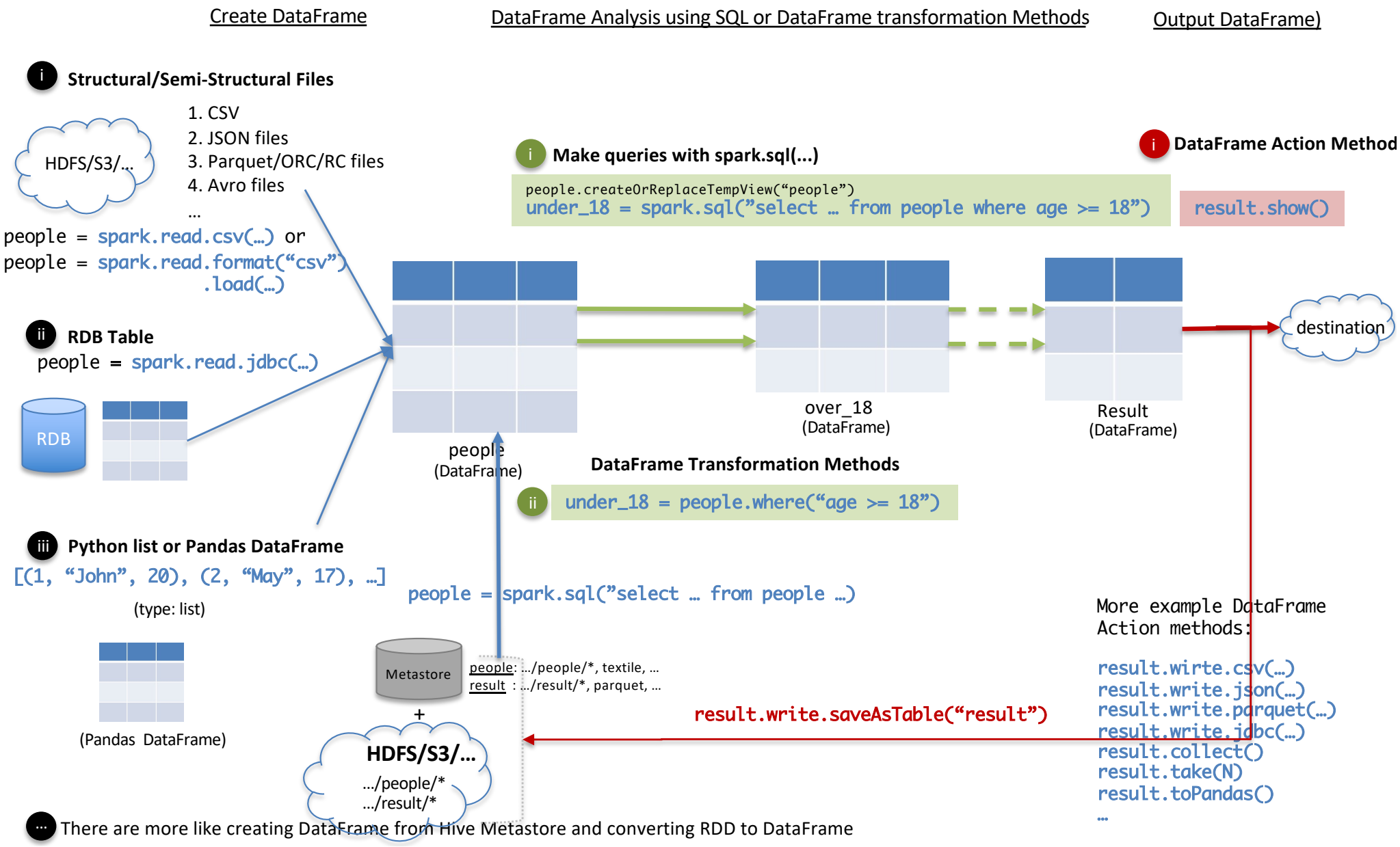
Spark SQL supports the HiveQL syntax as well as Hive SerDes and UDFs, allowing you to access existing Hive warehouses.



Spark SQL can use existing Hive metastores, SerDes, and UDFs.

Spark SQL API Programming Model

Spark SQL API for Python: **pyspark.sql**



Create DataFrame from Hive Tables

```
from pyspark.sql import SparkSession


if __name__ == "__main__":
# Make sure that SparkSession is created with hive support

    spark = SparkSession \
            .builder \
            .enableHiveSupport() \
            .getOrCreate()


    # Just make quries on tables defined in MetaStore with spark.sql("...")

    df = spark.sql("select symbol, count(*) from stocks group by symbol")
    df.show()


    # Hive commands are supported
    spark.sql("show databases").show()
    spark.sql("show tables").show()
    spark.sql("describe stocks").show()
    spark.sql("describe formatted stocks").show(100, truncate=False)


    # Hive UDFs are supported

    spark.sql("add jar /home/spark/IdeaProjects/proj_hive_udf/hive_udf.jar")
    spark.sql("create temporary function revs as 'com.iii.udf.MyReverse'")

    df2 = spark.sql("select symbol, revs(symbol) from stocks")
    df2.show()
```

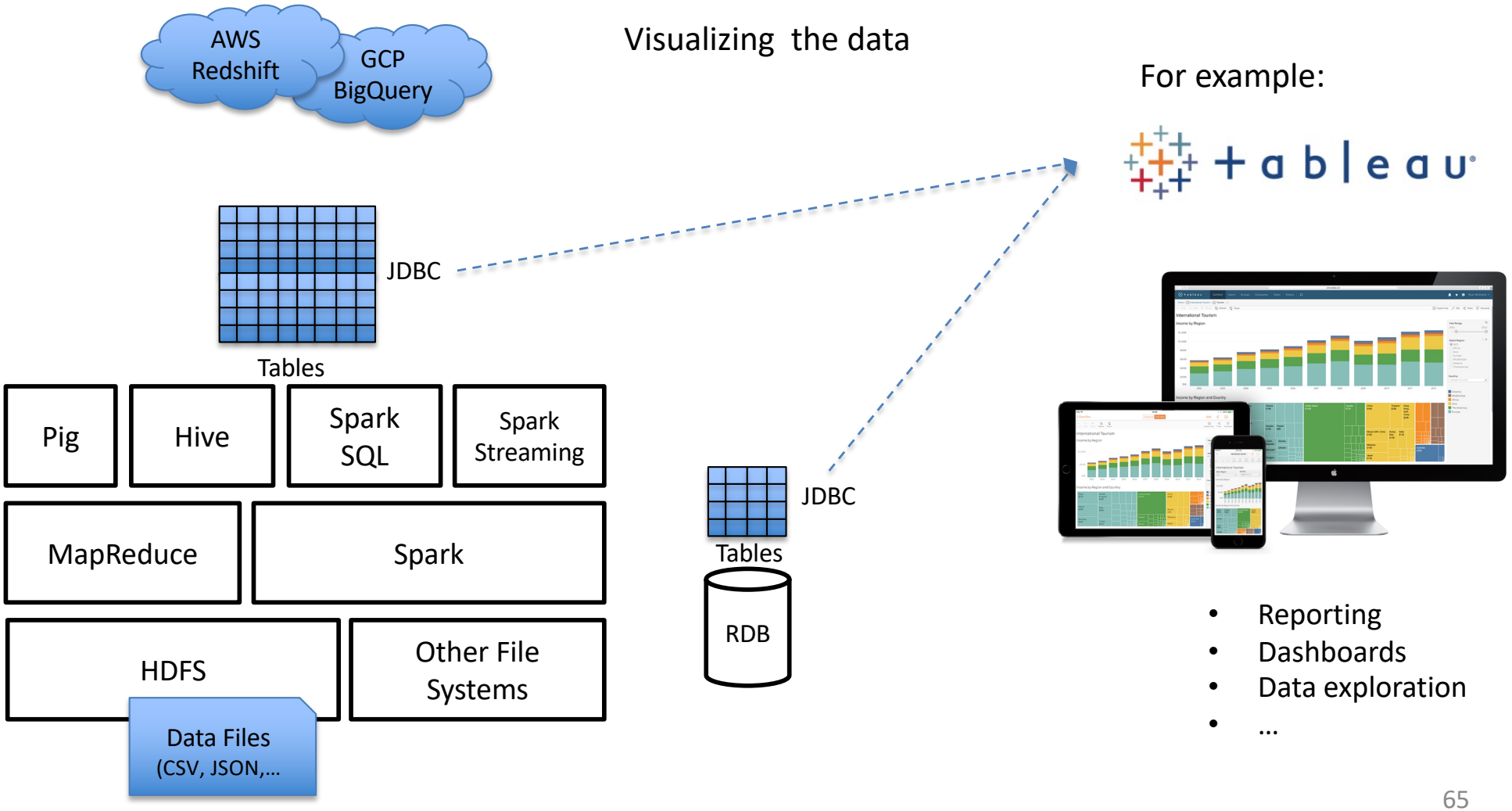
Create DataFrame from Hive Tables

- Setup
  - Make sure that the Spark and Hive versions are matched, or the other steps may be required.
  - Place your hive-site.xml in $SPARK_HOME/conf/ to allow Spark to access Metastore

Setup Spark SQL as a SQL engine

- Spark SQL can also act as a distributed query engine using its JDBC/ODBC or command-line interface
- In this mode, end-users or applications can interact with Spark SQL directly to run SQL queries, without the need to write any code.

Access data in Spark SQL and Hive via JDBC



Setup Spark SQL as a SQL engine

- Running the Thrift JDBC/ODBC server

```
export HIVE_SERVER2_THRIFT_PORT=<listening-port>
export HIVE_SERVER2_THRIFT_BIND_HOST=<listening-host>
./sbin/start-thriftserver.sh \
  --master <master-uri> \
  ...
```

or system properties:

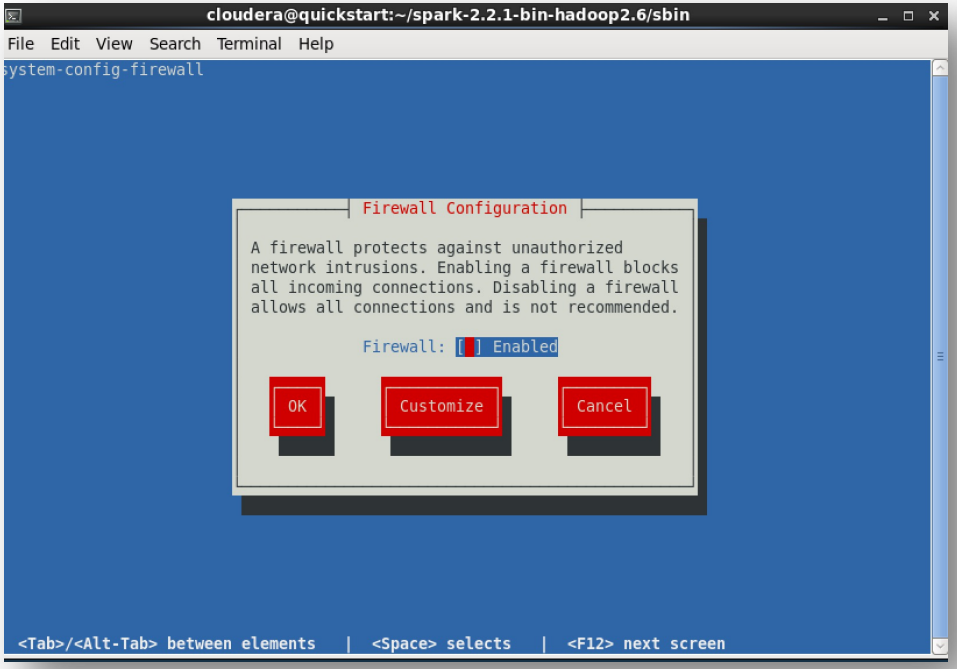
```
./sbin/start-thriftserver.sh \
  --hiveconf hive.server2.thrift.port=<listening-port> \
  --hiveconf hive.server2.thrift.bind.host=<listening-host> \
  --master <master-uri>
  ...
```

Setup Spark SQL as a SQL engine

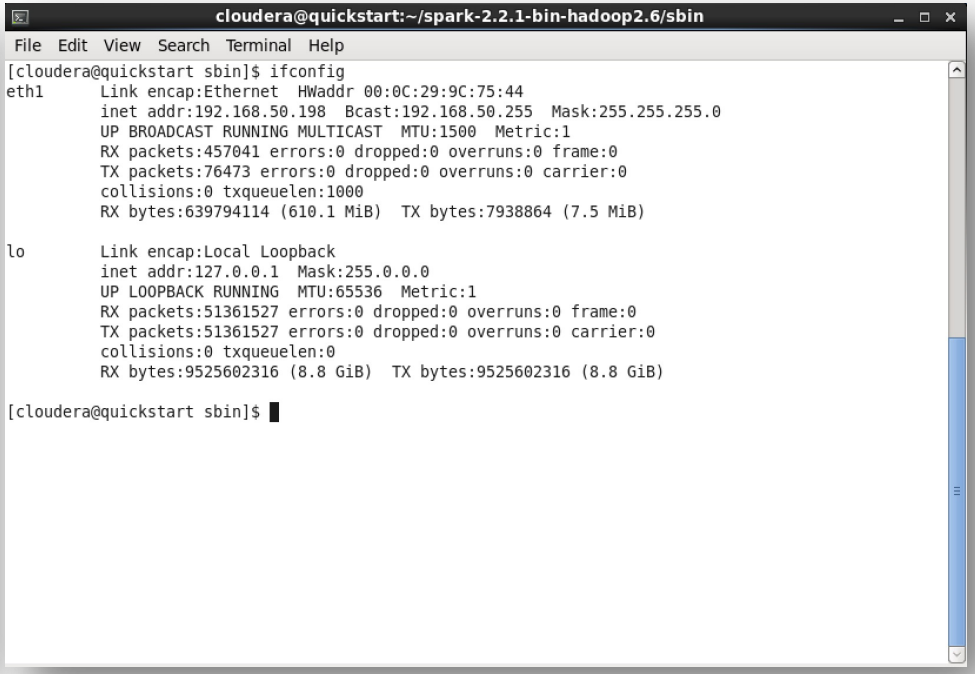
Start the Thrift Server

```
./start-thriftserver.sh --master spark://devenv:7077
```

Make sure 10001 is open to access outside



Check the IP address



Common File Types in Big Data Eco-system

(reference)

Topics

- Compression overview
- Object serialization
- Major big data file formats

Compression

- Reduces the space
- Speed up data transmission

Compression

- Hadoop use compression for
  - Store compressed files to reduce files size and transfer efficiency
  - Record or block compression in SequenceFile, Avro file
  - Compress workers output to reduce data size

Comparison of compression codecs

Codec	Extension	Licensing	Splittable	Java-only compression support	Native compression support
Deflate	.deflate	zlib	No	Yes	Yes
gzip	.gz	GNU GPL	No	Yes	Yes
bzip2	.gz	BSD	Yes [a]	Yes	No
lzo	.lzo_deflate	GNU GPL	No	No	Yes
lzop	.lzo	GNU GPL	Yes [b]	No	Yes
Snappy	.gz	New BSD	No	No	Yes

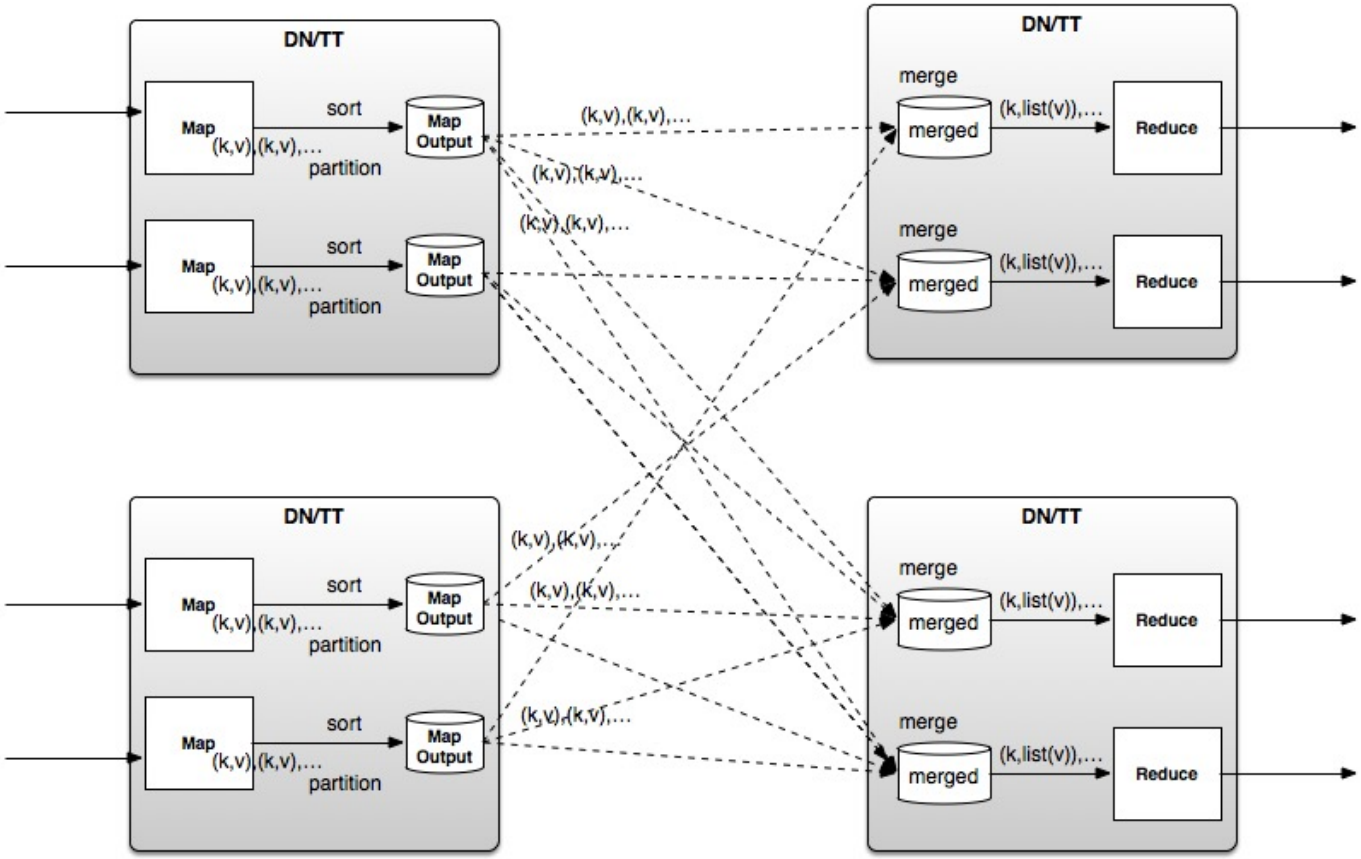
Figures and tables from Hadoop the definitive guide

Performance comparison of compression codecs on a 128 MB text file

Codec	Compression time (secs)	Decompression time (secs)	Compressed file size	Compressed percentage
Deflate	6.88	6.80	24,866,259	18.53%
gzip	6.68	6.88	24,866,271	18.53%
bzip2	3,012.34	24.31	19,270,217	14.36%
lzo	1.69	7.00	40,946,704	30.51%
lzop	1.70	5.62	40,946,746	30.51%
Snappy	1.31	6.66	46,108,189	34.45%

Figures and tables from Hadoop the definitive guide

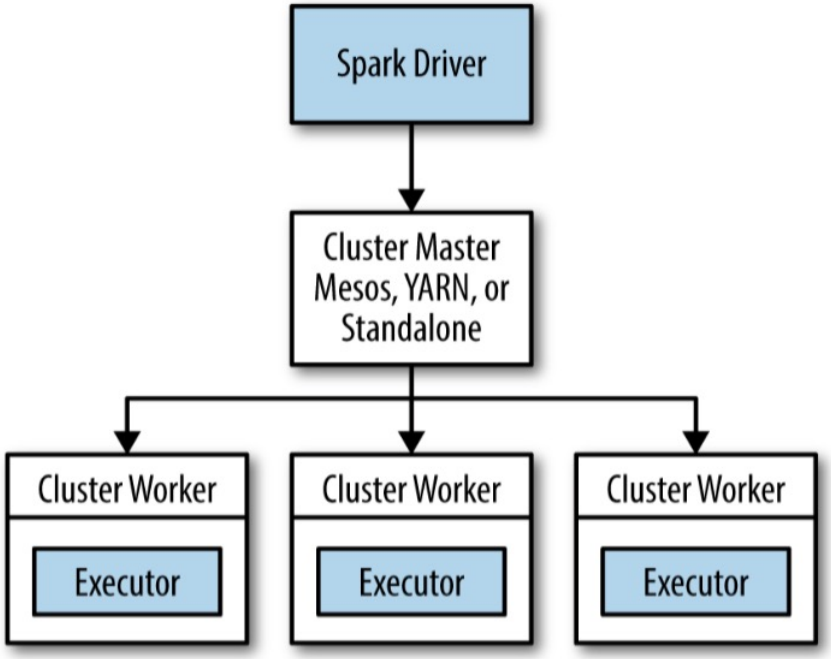
Compress Mapper output




```
mapreduce.map.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec
```

Compress executors output

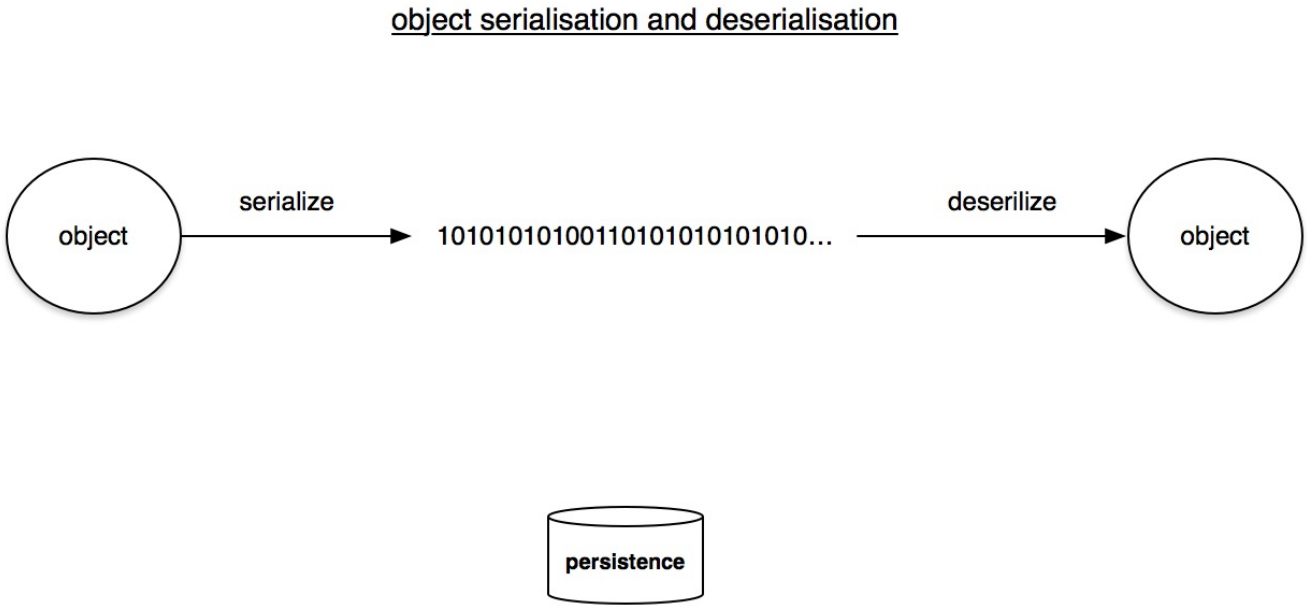
- compress internal data such as RDD partitions, broadcast variables and shuffle outputs



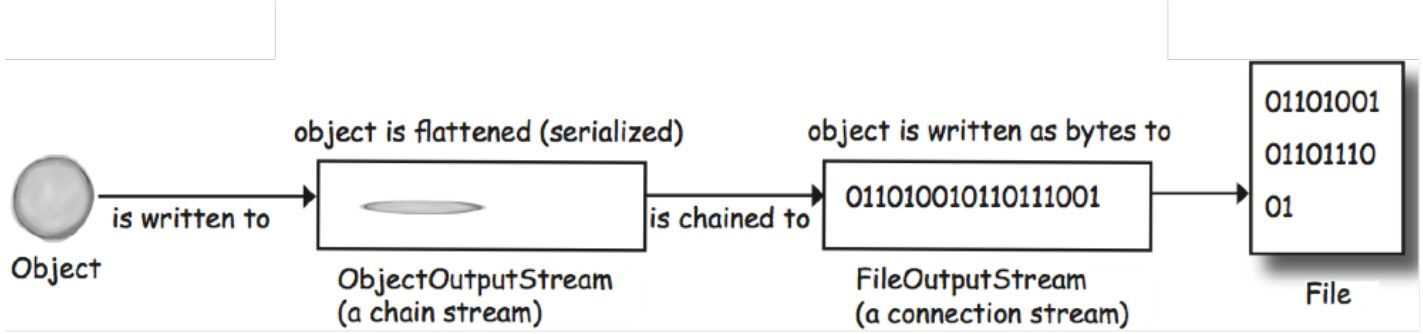
`spark.io.compression.codec=` org.apache.spark.io.SnappyCompressionCodec

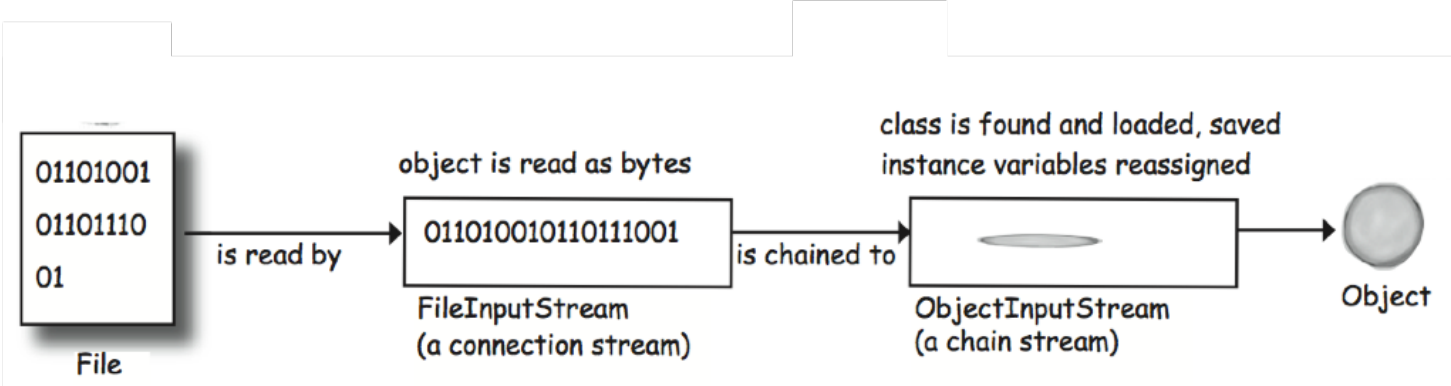
Figures and tables from Hadoop the definitive guide

Data Serialization overview



Serialization in Java





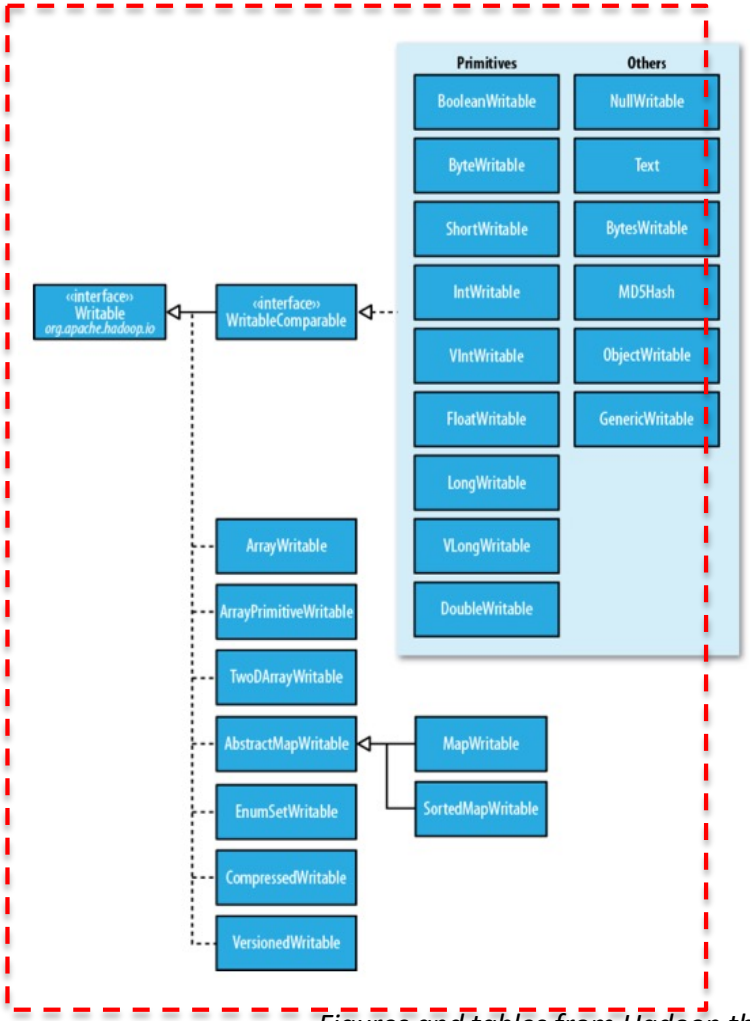
Writables in Hadoop

- Built-in serialization from Hadoop MapReduce

org.apache.hadoop.io

Interface Writable

Method Summary	
void	**readFields**(DataInput in) Deserialize the fields of this object from in.
void	**write**(DataOutput out) Serialize the fields of this object to out.



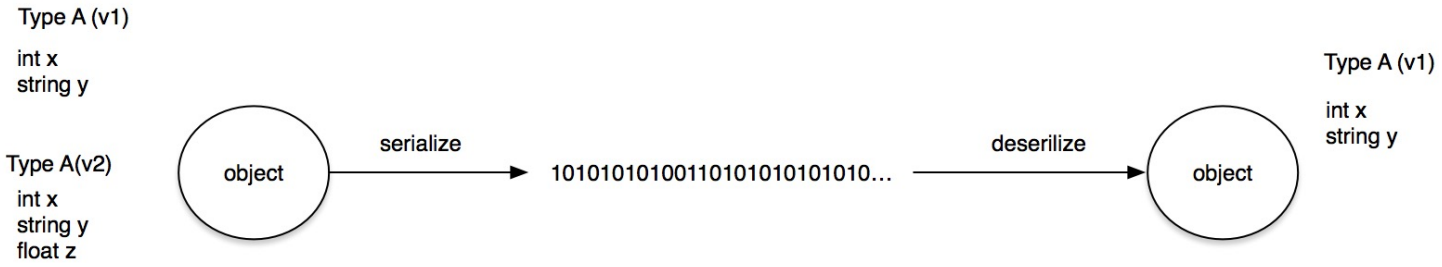
Figures and tables from Hadoop the definitive guide

Example

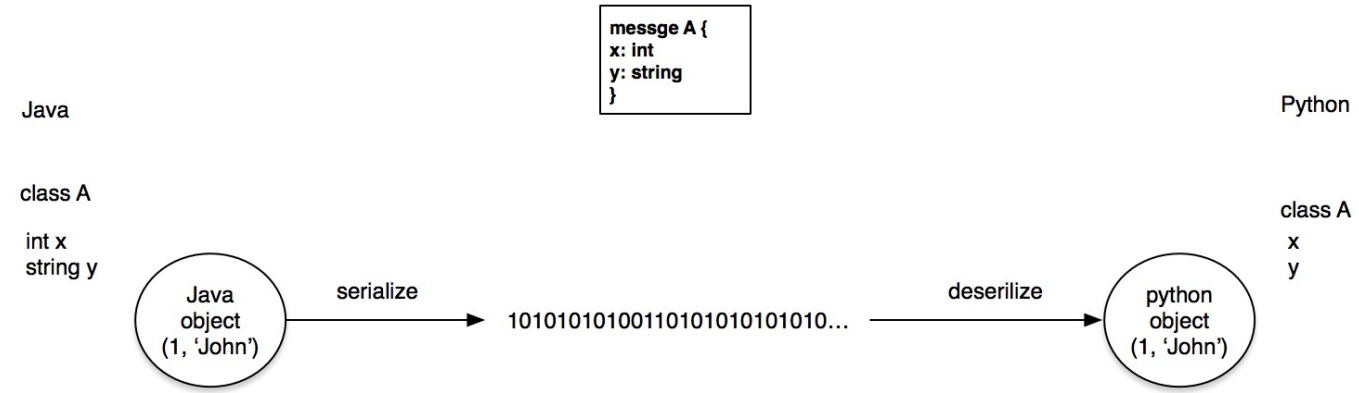
```java
package iii.mr101;

import java.io.*;

import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

public class AvgTempMapper extends MapReduceBase
implements Mapper<LongWritable, Text, Text, IntWritable>  {

    public void map(LongWritable key, Text value,
            OutputCollector<Text, IntWritable> output, Reporter reporter)
    throws IOException {

        String[] line = value.toString().split(",");

        String dataPart = line[1];
        String temp = line[10];

        if (StringUtils.isNumeric(temp) && !temp.equals("") ){
            output.collect(new Text(dataPart), new IntWritable(Integer.parseInt(temp)));
        }
    }
}
```

Modern Data Serialization

Schema Evolution



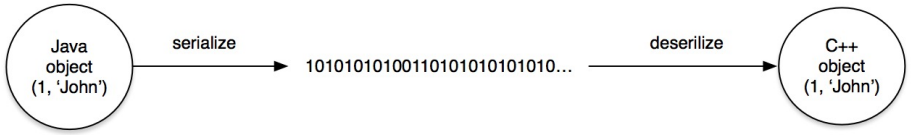
Interoperiablity



Compact and compressed

Google Protocol buffer / Thrift / Avro (JSON/XML)

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```




```
Person john =
  Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .addPhone(
      Person.PhoneNumber.newBuilder()
        .setNumber("555-4321")
        .setType(Person.PhoneType.HOME))
    .build();
```

```
Person john;
fstream input(argv[1],
    ios::in | ios::binary);
john.ParseFromIstream(&input);
id = john.id();
name = john.name();
email = john.email();
```

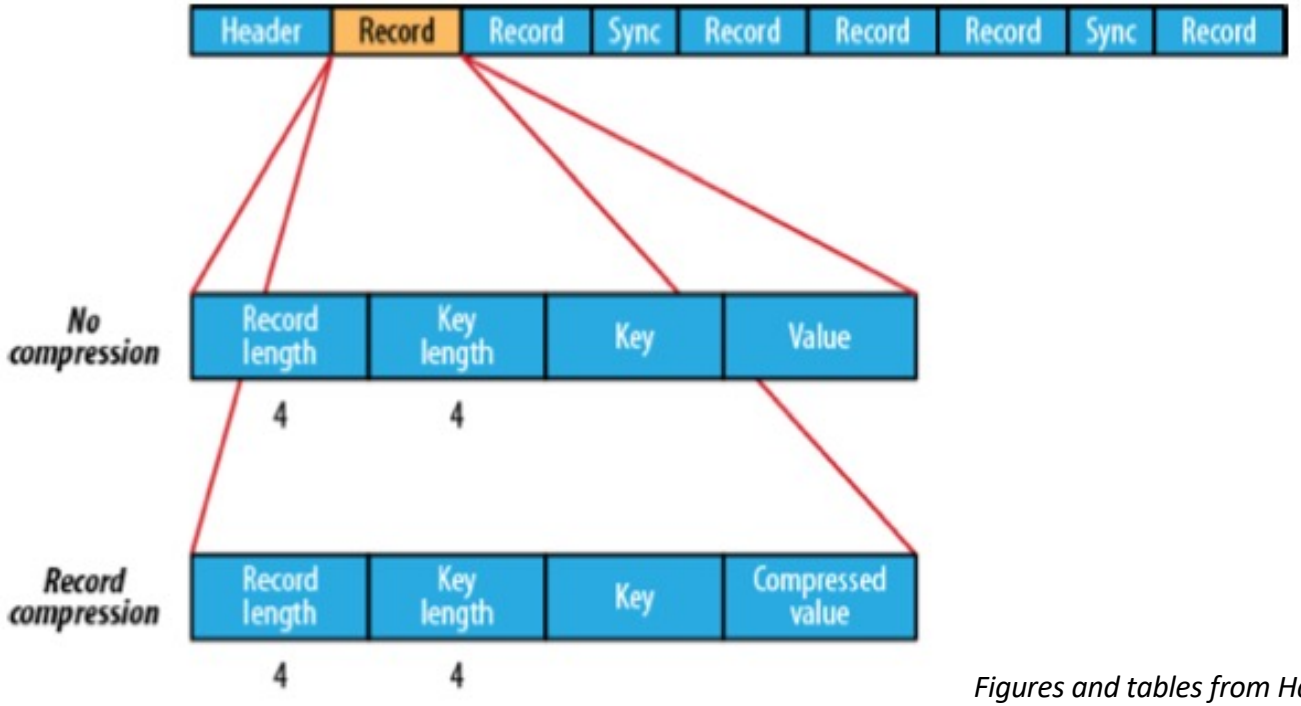
File formats on Hadoop

Library	Code generation	Schema evolution	Language support	Transparent compression	Splittable	Native support in MapReduce	Pig and Hive support	Spark support
Sequence-File	No	No	Java, Python	Yes	Yes	Yes	Yes	Yes
Avro	Yes (optional)	Yes	C, C++, Java, Python, Ruby, C#	Yes	Yes	Yes	Yes	Yes
Parquet	No	Yes	Java, Python (C++ planned in 2.0)	Yes	Yes	Yes	Yes	Yes

Figures and tables from Hadoop in practice

SequenceFile

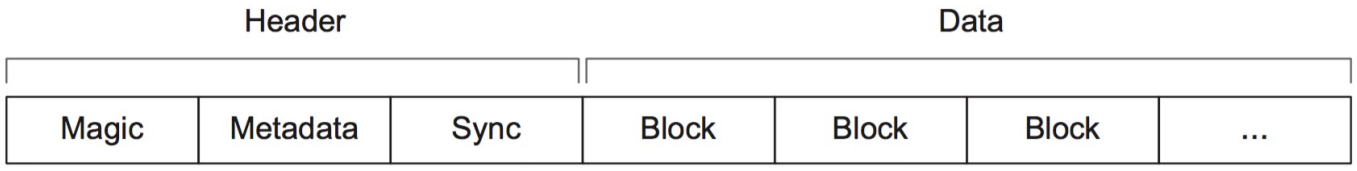
- Writable is the default record format
- Support record and block-level compression
- Default serialization in MapReduce



Figures and tables from Hadoop the definitive guide

Avro

- Avro utilizes a compact binary data format
- Doug Cutting created Avro, a data serialization and RPC library, to help improve data interchange, interoperability, and versioning in MapReduce.



Avro container file format

Figures and tables from Hadoop in practice

Avro

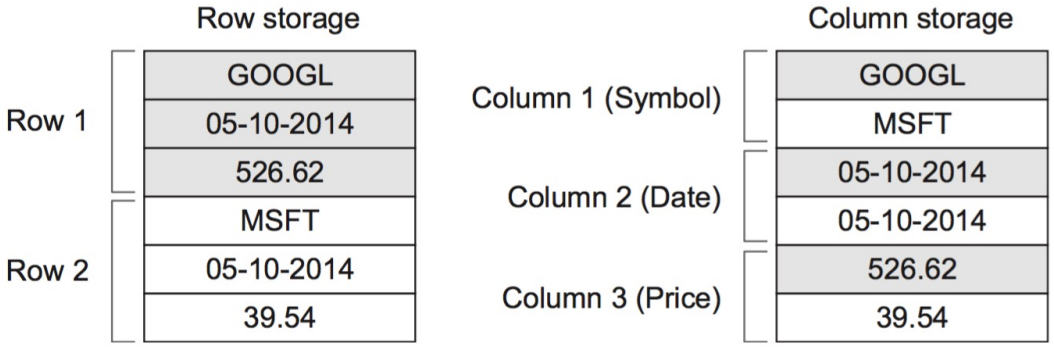
- Schema in JSON form, and use Avro tools to generate rich APIs to interact with your data.

Parquet

One of the most advanced column-based file formats

Sample records

Symbol	Date	Price
GOOGL	05-10-2014	526.62
MSFT	05-10-2014	39.54



Figures and tables from Hadoop in practice

Parquet

- Parquet is to maximize support throughout the Hadoop ecosystem.
- It currently data objects from a variety of object types from several major tools such as Writables, Google Protocol Buffer, Avro and Thrift.

Figures and tables are modified from from Hadoop in practice