COMP9313: Big Data Management

Spark SQL

Why Spark SQL?

- Table is one of the most commonly used ways to present data
  - Easy to scan, analyze, filter, sort, etc.
  - Widely used in communication, research, and data analysis
- Table has (relatively) stable data structure
  - 2 Dimension: row and column
  - Pre-defined attribute types
- In general, customized/specialized is better!

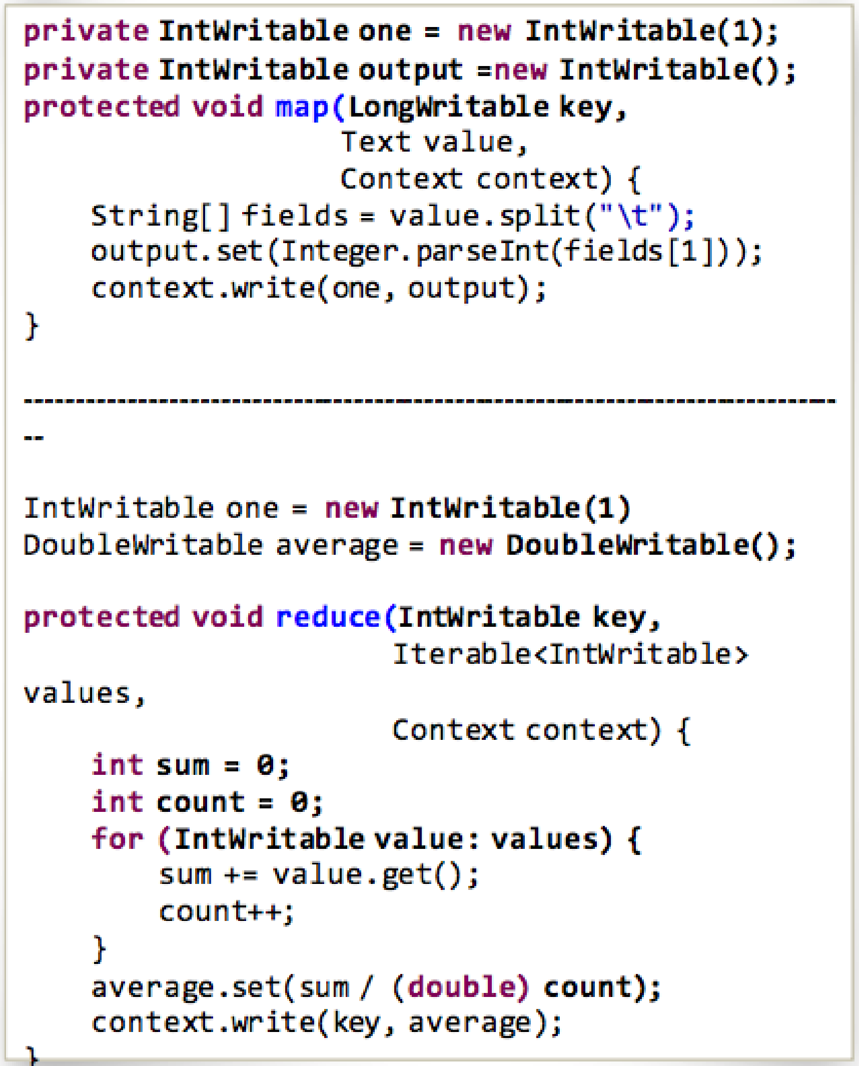
What is Spark SQL?

- Part of the core distribution since Spark 1.0 (April 2014)
- Tightly integrated way to work with structured data
  - tables with rows/columns
- Transform RDDs using SQL
- Data source integration: Hive, Parquet, JSON, csv, etc.

- Spark SQL is **not** about SQL
  - Aims to Create and Run Spark Programs Faster

Compute Average

source



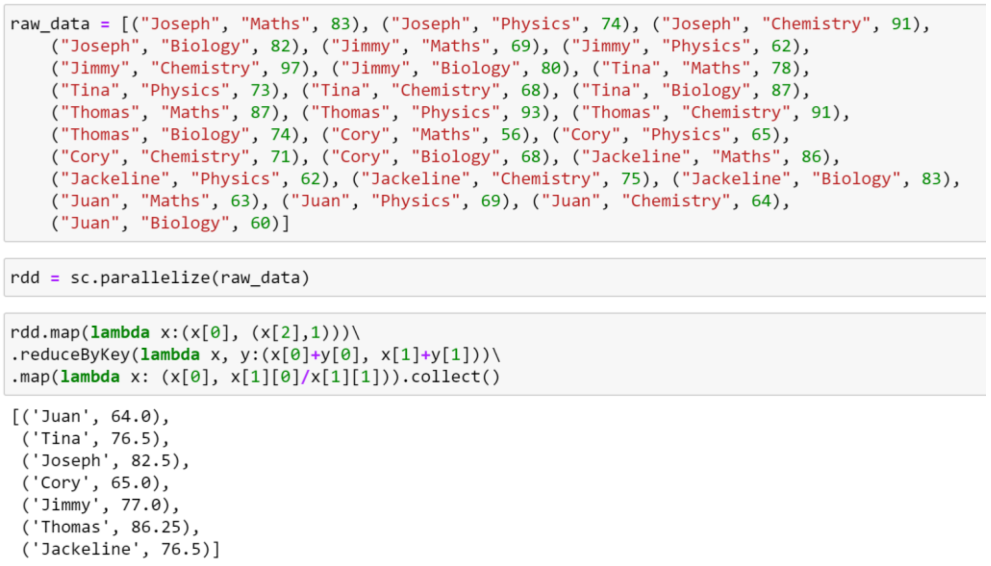
```
private IntWritable one = new IntWritable(1);
private IntWritable output =new IntWritable();
protected void map(LongWritable key,
                   Text value,
                   Context context) {
    String[] fields = value.split("\t");
    output.set(Integer.parseInt(fields[1]));
    context.write(one,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

Use RDD



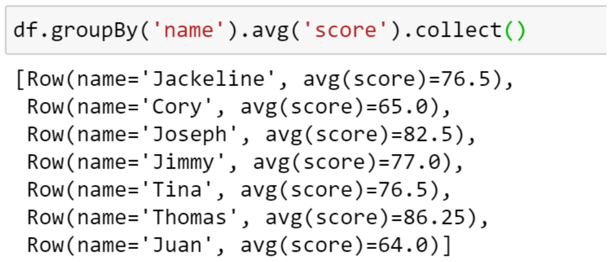
```
raw_data = [("Joseph", "Maths", 83), ("Joseph", "Physics", 74), ("Joseph", "Chemistry", 91),
    ("Joseph", "Biology", 82), ("Jimmy", "Maths", 69), ("Jimmy", "Physics", 62),
    ("Jimmy", "Chemistry", 97), ("Jimmy", "Biology", 80), ("Tina", "Maths", 78),
    ("Tina", "Physics", 73), ("Tina", "Chemistry", 68), ("Tina", "Biology", 87),
    ("Thomas", "Maths", 87), ("Thomas", "Physics", 93), ("Thomas", "Chemistry", 91),
    ("Thomas", "Biology", 74), ("Cory", "Maths", 56), ("Cory", "Physics", 65),
    ("Cory", "Chemistry", 71), ("Cory", "Biology", 68), ("Jackeline", "Maths", 86),
    ("Jackeline", "Physics", 62), ("Jackeline", "Chemistry", 75), ("Jackeline", "Biology", 83),
    ("Juan", "Maths", 63), ("Juan", "Physics", 69), ("Juan", "Chemistry", 64),
    ("Juan", "Biology", 60)]
```

```
rdd = sc.parallelize(raw_data)
```

```
rdd.map(lambda x:(x[0], (x[2],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()
```

```
[('Juan', 64.0),
 ('Tina', 76.5),
 ('Joseph', 82.5),
 ('Cory', 65.0),
 ('Jimmy', 77.0),
 ('Thomas', 86.25),
 ('Jackeline', 76.5)]
```

Use Spark SQL



```
df.groupBy('name').avg('score').collect()
```

```
[Row(name='Jackeline', avg(score)=76.5),
 Row(name='Cory', avg(score)=65.0),
 Row(name='Joseph', avg(score)=82.5),
 Row(name='Jimmy', avg(score)=77.0),
 Row(name='Tina', avg(score)=76.5),
 Row(name='Thomas', avg(score)=86.25),
 Row(name='Juan', avg(score)=64.0)]
```

DataFrame

- A DataFrame is a distributed collection of data organized into named columns.
- The DataFrames API is:
    - intended to 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
    - an extension to the existing RDD API

DataFrame

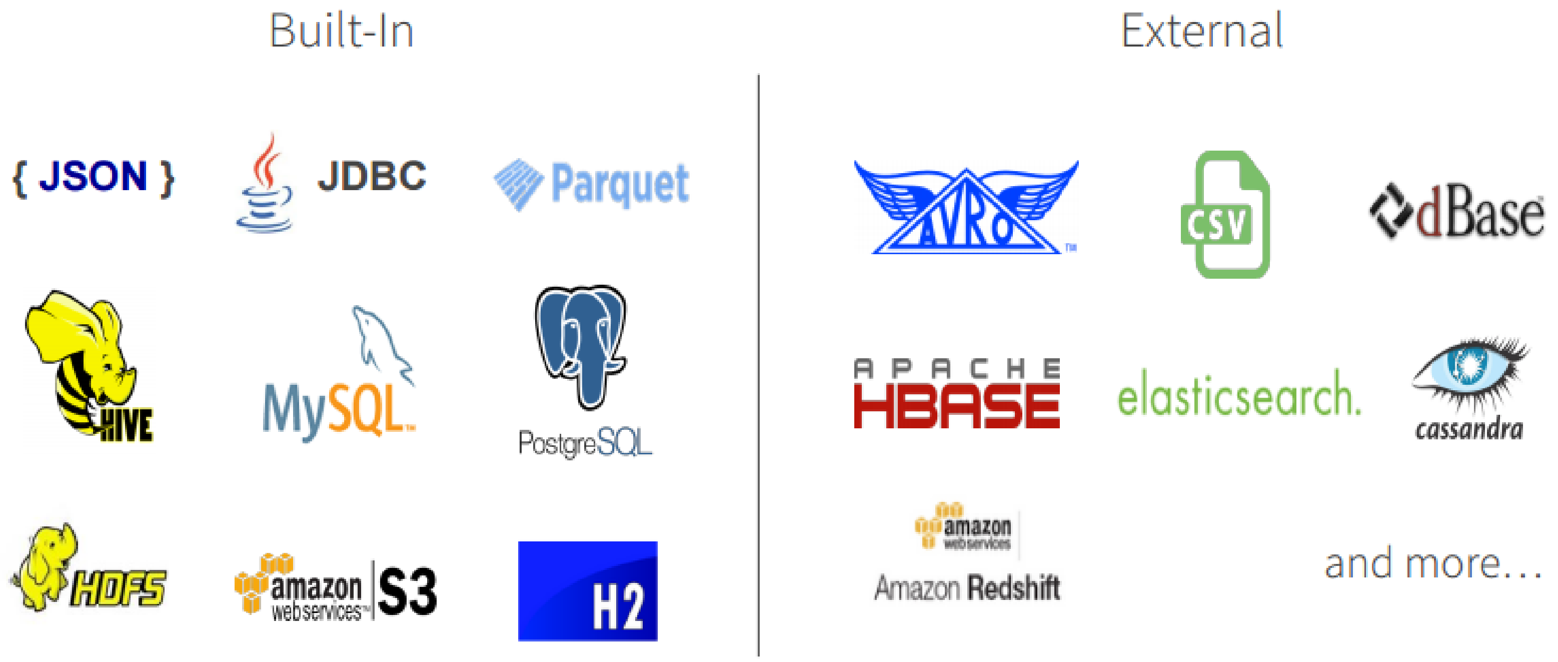
- DataFrames have the following features:
  - Ability to scale from kilobytes of data on a single laptop to petabytes on a large cluster
  - Support for a wide array of data formats and storage systems
  - State-of-the-art optimization and code generation through the Spark SQL Catalyst optimizer
  - Seamless integration with all big data tooling and infrastructure via Spark
  - APIs for Python, Java, Scala, and R

DataFrame

- For new users familiar with data frames in other programming languages, this API should make them feel at home.
- For existing Spark users, the API will make Spark easier to program.
- For both sets of users, DataFrames will improve performance through intelligent optimizations and code-generation.

DataFrame Data Sources

- Spark SQL's Data Source API can read and write DataFrames using a variety of formats.
  - E.g., structured data files, tables in Hive, external databases, or existing RDDs



Creating DataFrames

- SparkSession is the entry point to programming Spark with the Dataset and DataFrame API.
    - Creates a DataFrame based on the content of a JSON file:

```
spark = SparkSession.builder.config(conf=conf).getOrCreate()
df = spark.read.format(”json”).load(”example.json")

# Displays the content of the DataFrame to stdout
df.show()

// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

DataFrame Operations

```
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

# Select only the "name" column
df.select("name").show()
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+

# 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|
// +-------+---------+
```

DataFrame Operations

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

# Count people by age
df.groupBy("age").count().show()
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

DataFrames and Spark SQL

- DataFrames are fundamentally tied to Spark SQL.
  - The DataFrames API provides a programmatic interface—really, a domain-specific language (DSL)—for interacting with your data.
  - Spark SQL provides a SQL-like interface.
  - What you can do in Spark SQL, you can do in DataFrames
  - … and vice versa

Spark SQL

- Spark SQL allows you to manipulate distributed data with SQL queries.
- You issue SQL queries through a SparkSession, using the sql() method.
- sql() enables applications to run SQL queries programmatically and returns the result as a DataFrame.
- You can mix DataFrame methods and SQL queries in the same code.

Spark SQL

- To use SQL, you need make a table aliasfor a DataFrame, using registerTempTable()

```
# Register the DataFrame as a SQL temporary view
df.registerTempTable(” people ")

spark.sql("SELECT * FROM people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

More on DataFrames

- DataFrames are lazy.
- Transformations contribute to the query plan, but they don't execute anything.
- Actions cause the execution of the query

Transformation examples

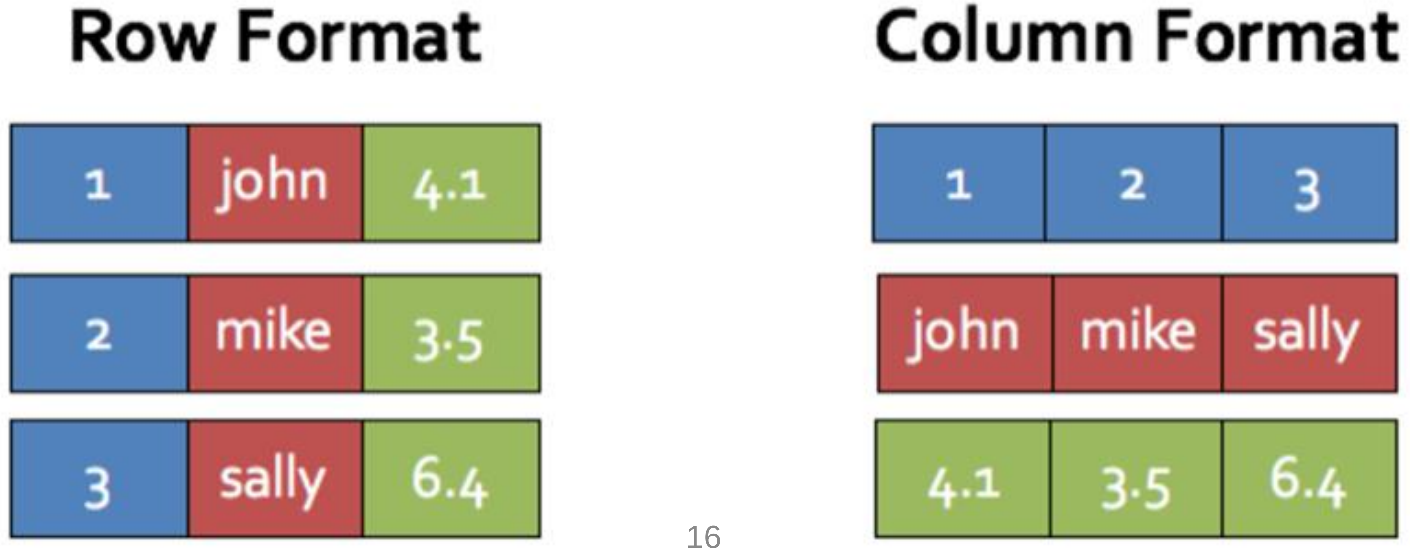
filter
select
drop
intersect
join

Action examples

count
collect
show
head
take

More on DataFrames – Columnar Storage

- DataFrames use columnar storage
  - Transpose of row-based storage
  - Data in each column (with the same type) are packed together



More on DataFrames – Columnar Storage

- Data access is more regular
- Denser storage
- Compatibility and zero serialization
- More Extensions to GPU/TPU
- Efficient query processing

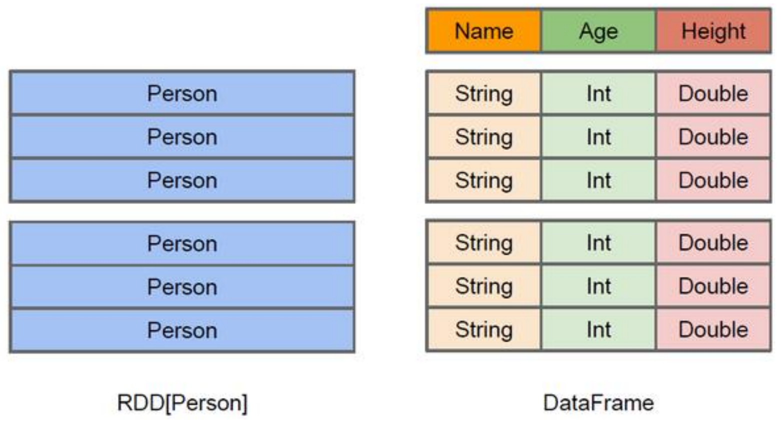
- Not good for transactions
  - But we don’t expect many transactions in Spark!

DataFrame and RDD

- DataFrames are built on top of the Spark RDD API
  - You can use normal RDD operations on DataFrames
- Stick with the DataFrame API if possible
  - Using RDD operations will often give you back an RDD, not a DataFrame
  - The DataFrame API is likely to be more efficient, because it can optimize the underlying operations with Catalyst

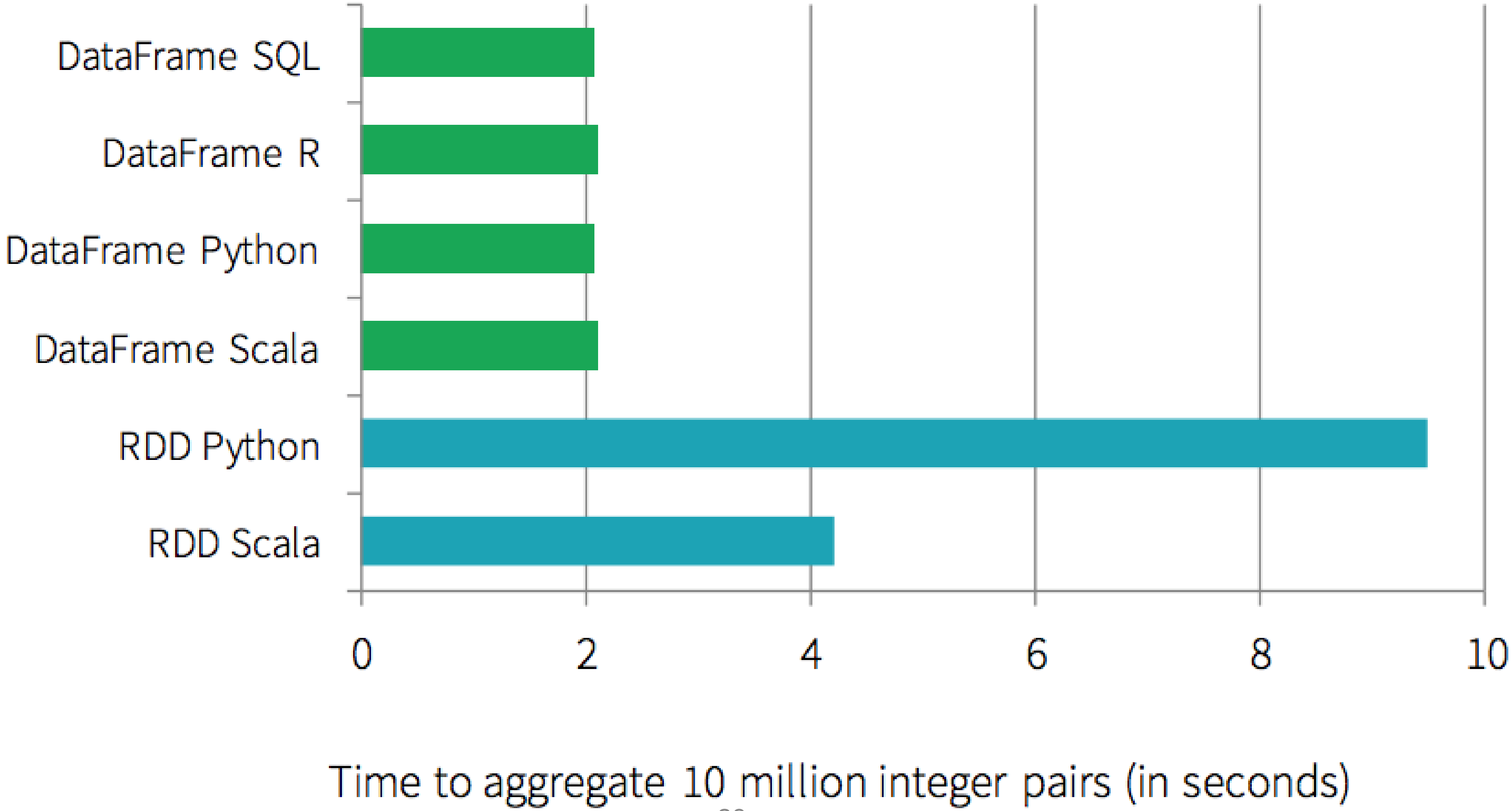
DataFrame and RDD

- DataFrame more like a traditional database of two-dimensional form, in addition to data, but also to grasp the structural information of the data, that is, schema



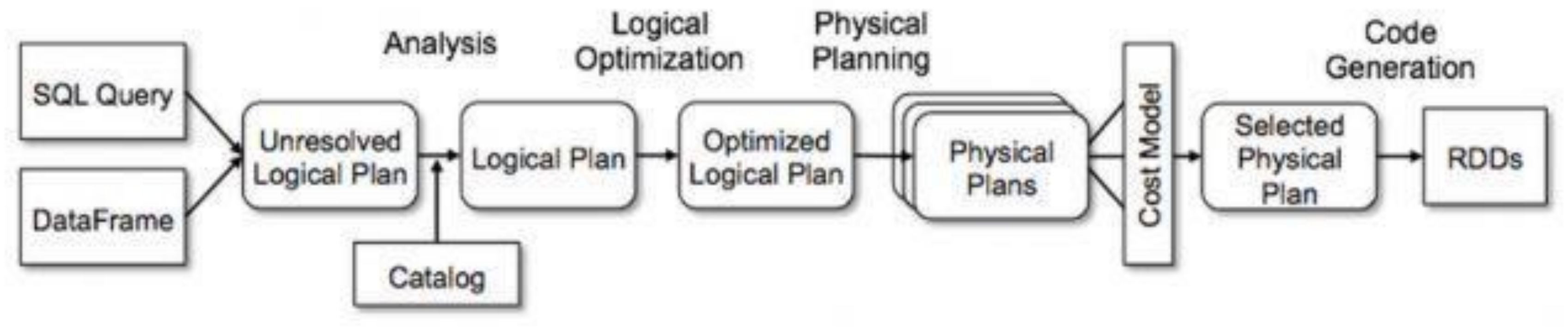
  - RDD[Person] although with Person for type parameters, but the Spark framework itself does not understand internal structure of Person class
  - DataFrame has provided a detailed structural information, making Spark SQL can clearly know what columns are included in the dataset, and what is the name and type of each column. Thus Spark SQL query optimizer can target optimization

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



Spark SQL – Plan Optimization & Execution

- Logical plan and physical plans
- Execution is lazy, allowing it to be optimized



DataFrames and SQL share the same optimization/execution pipeline

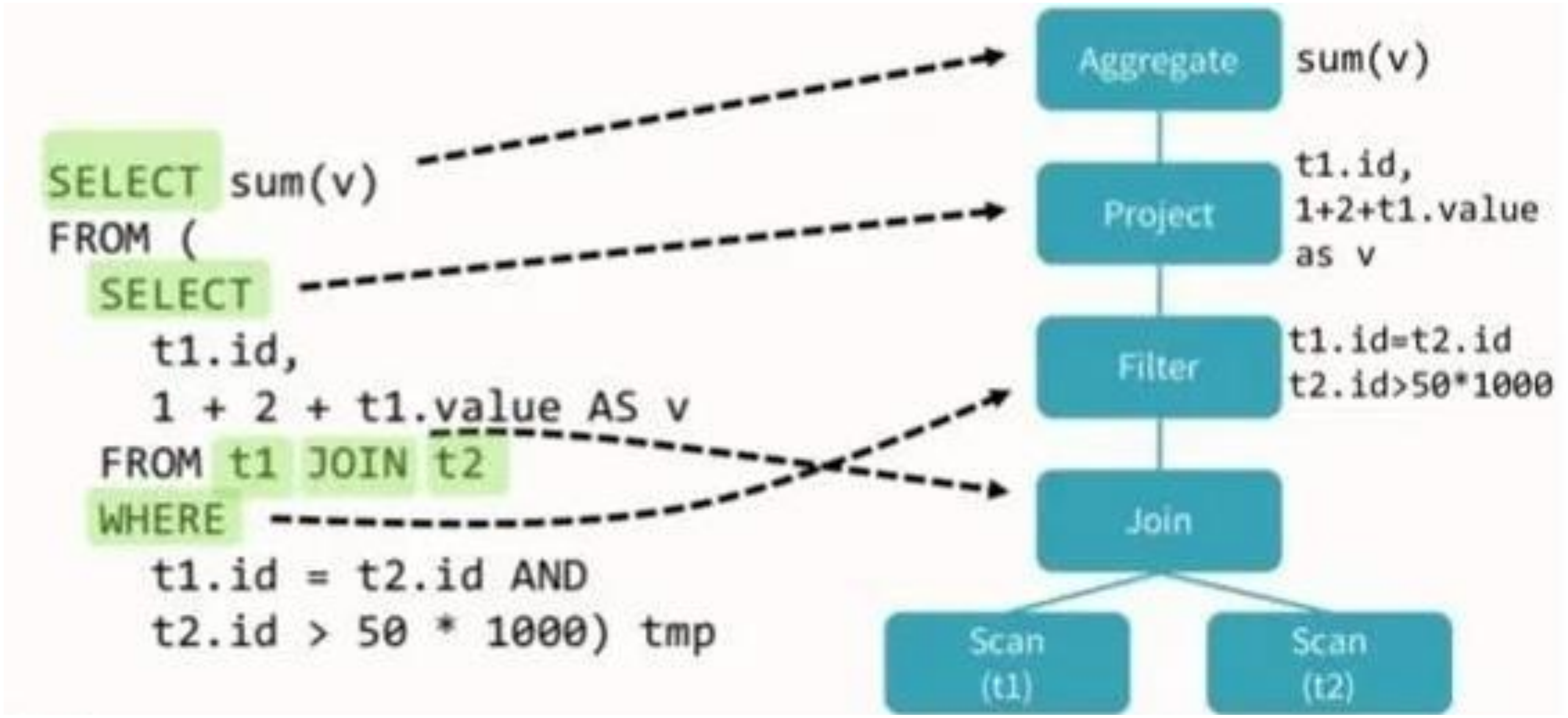
Logical Plan

- Logical Plan is an abstract of all transformation steps that need to be performed
  - It does not refer anything about the Driver (Master Node) or Executor (Worker Node)
  - SparkContext is responsible for generating and storing it

- Logical Plan is divided into three parts:
  - Unresolved Logical Plan
  - Resolved Logical Plan
  - Optimized Logical Plan

SQL to Resolved Logical Plan

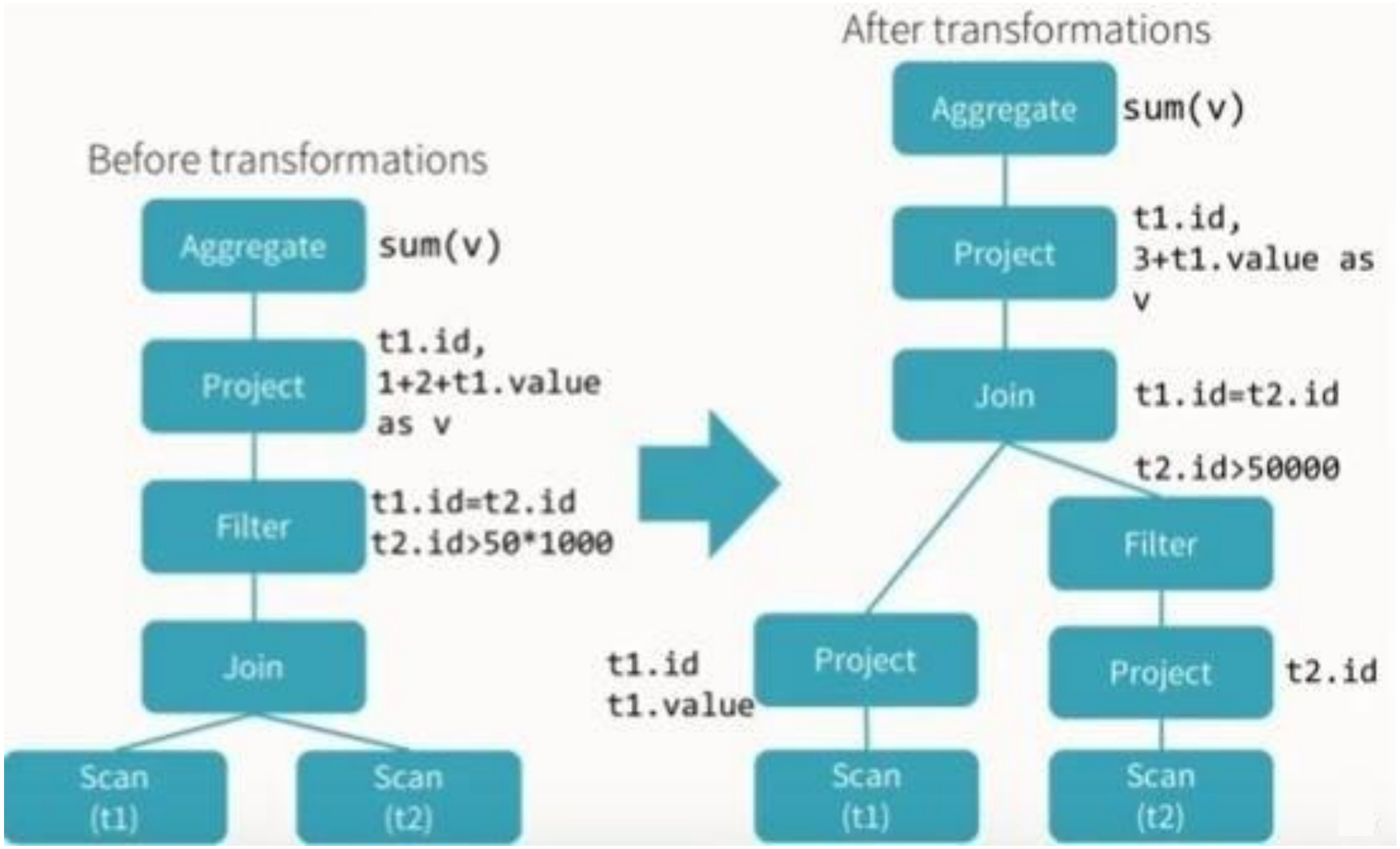
- If the SQL query is unresolvable, then it will be rejected
  - Otherwise a resolved logical plan is created



```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```

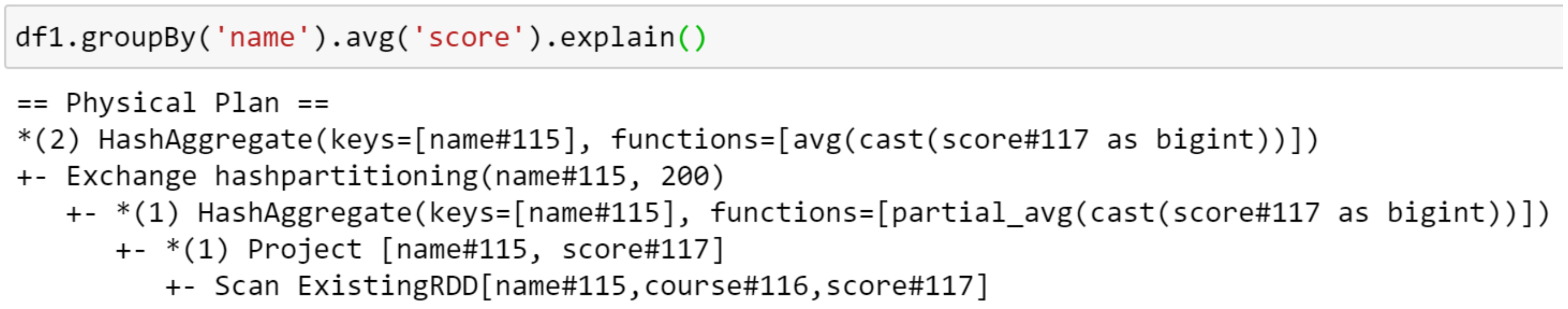
Resolved logical plan to Optimized Logical Plan

- Catalyst optimizer will try to optimize the plan



Physical Plan

- A physical plan describes computation on datasets with specific definitions on how to conduct the computation
- A physical plan is executable



```
df1.groupBy('name').avg('score').explain()
```

```
== Physical Plan ==
*(2) HashAggregate(keys=[name#115], functions=[avg(cast(score#117 as bigint))])
+- Exchange hashpartitioning(name#115, 200)
   +- *(1) HashAggregate(keys=[name#115], functions=[partial_avg(cast(score#117 as bigint))])
      +- *(1) Project [name#115, score#117]
         +- Scan ExistingRDD[name#115,course#116,score#117]
```

Schemas in DataFrame

- A schema is the description of the structure of your data
  - column names and types
- DataFrames Have Schemas
  - A Parquet table has a schema that Spark can use
  - Spark can infer a Schema from a JSON file

```
[ {”name": "Amy",
   ”course": ”Maths",
   ”score": 75 },
  {”name": "Ravi",
   ”course": ”Biology",
   ”score": 93 },
 … ]
```

  - What if the data file have no schema?
    - Create an RDD of a particular type and let Spark infer the schema from that type
    - **Use the API to specify the schema programmatically**

Specify the Schema

- E.g., Create DataFrame from RDD or list
- The schema must match the real data, or an exception will be thrown at runtime.

```
from pyspark.sql.types import *

schema = StructType([StructField(“name”, StringType(), False),
                     StructField(”course", StringType(), False),
                     StructField(”score", IntegerType(), False)])

df = spark.createDataFrame(data, schema)
```

- If *schema* is
  - A list of column names, type will be inferred from data
  - None, will **try** to infer column names and types
- You can have Spark tell you what it thinks the data schema is, by calling the printSchema()

Show()

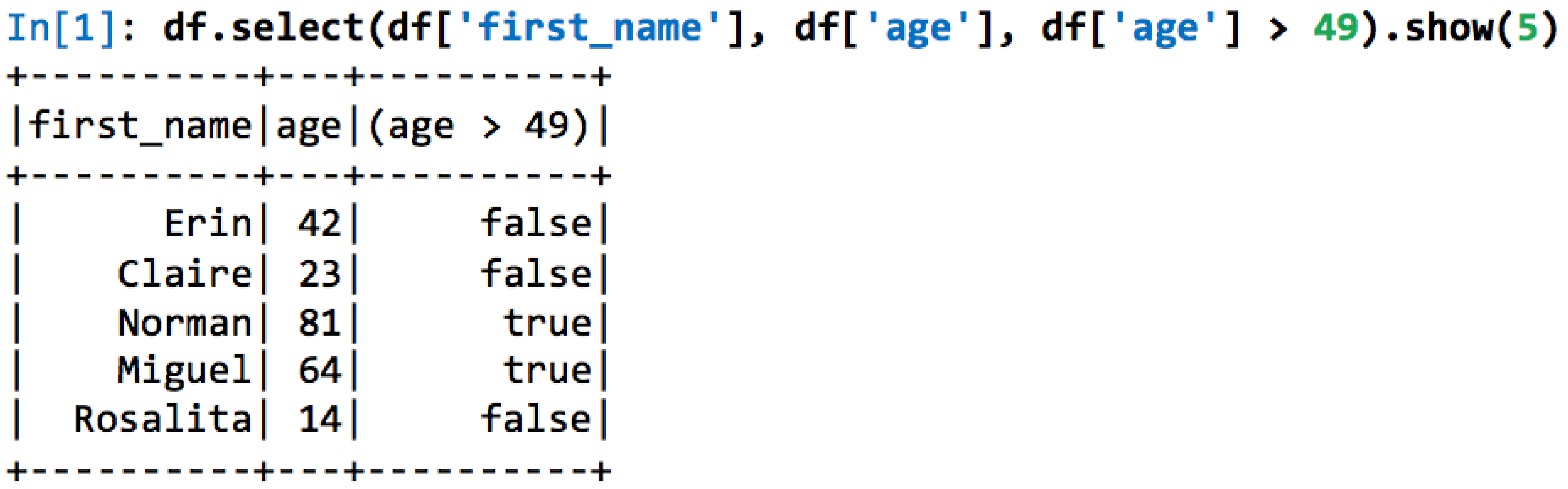
- You can look at the first n elements in a DataFrame with the show() method. If not specified, n defaults to 20. This method is an action, it
  - reads (or re-reads) the input source
  - executes the RDD DAG across the cluster
  - pulls the n elements back to the driver
  - displays those elements in a tabular form

cache()

- Spark can cache a DataFrame, using an in-memory columnar format, by calling df.cache()
- Spark will scan only those columns used by the DataFrame and will automatically tune compression to minimize memory usage and GC pressure.
- You can call the unpersist() method to remove the cached data from memory.

select()

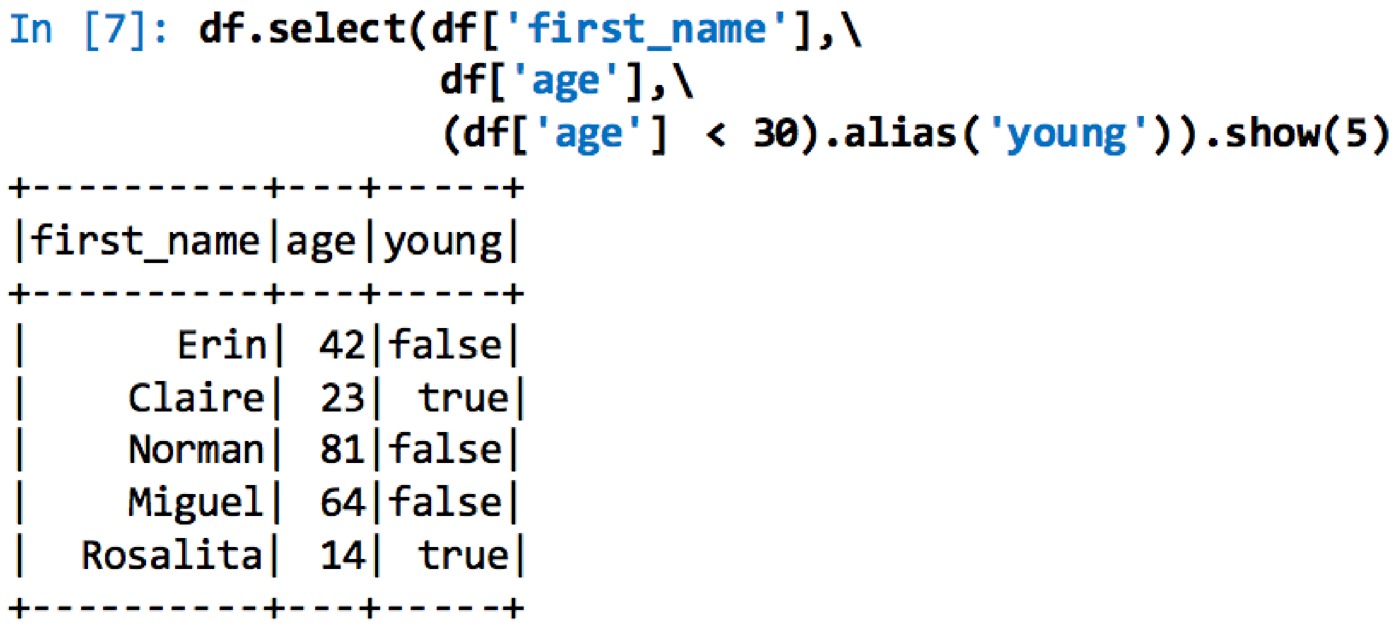
- select() is like a SQL SELECT, allowing you to limit the results to specific columns
- you can create on-the-fly derived columns



```
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|
+----------+---+----------+
```

alias()

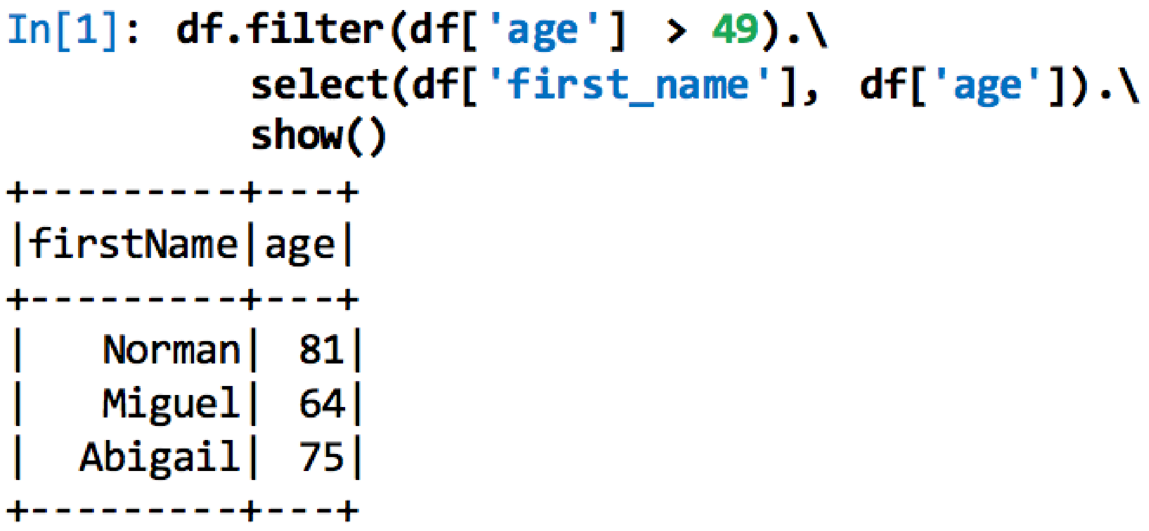
- alias() allows you to rename a column.
  - Especially useful for generated columns



```
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|
+----------+---+-----+
```

filter()

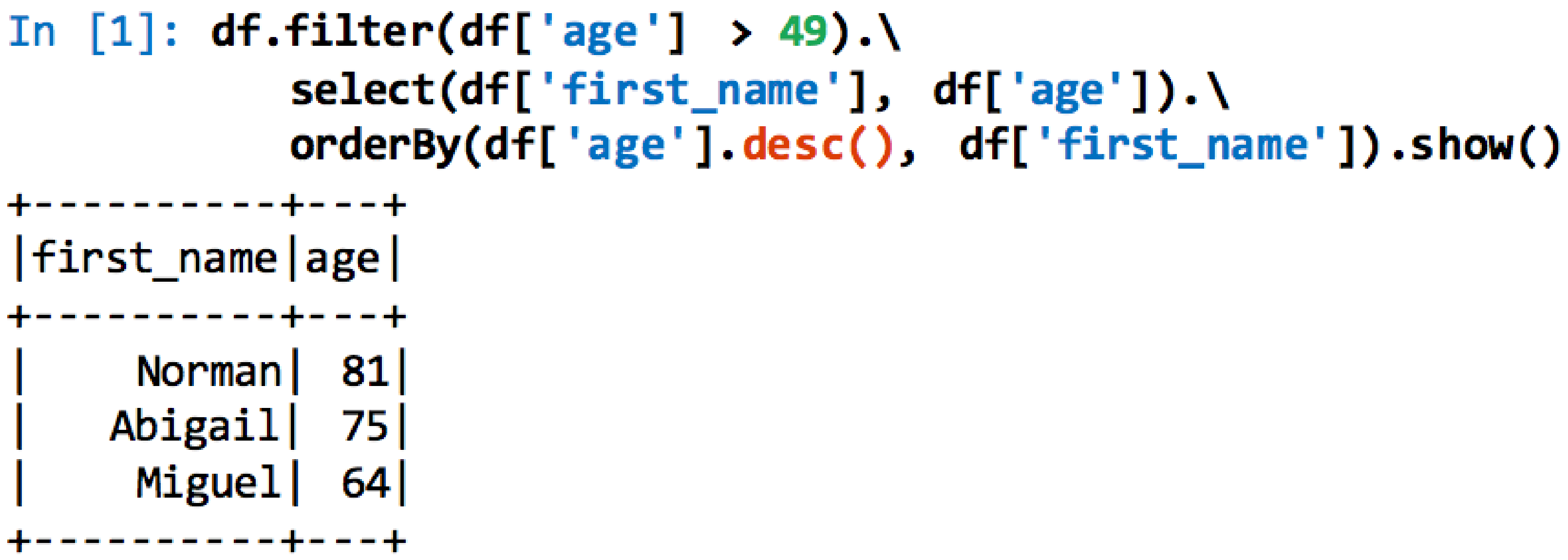
- The filter() method allows you to filter rows out of your results
  - WHERE in SQL



```
In[1]: df.filter(df['age'] > 49).\
          select(df['first_name'], df['age']).\
          show()
+---------+---+
|firstName|age|
+---------+---+
|   Norman| 81|
|   Miguel| 64|
|  Abigail| 75|
+---------+---+
```

orderBy()

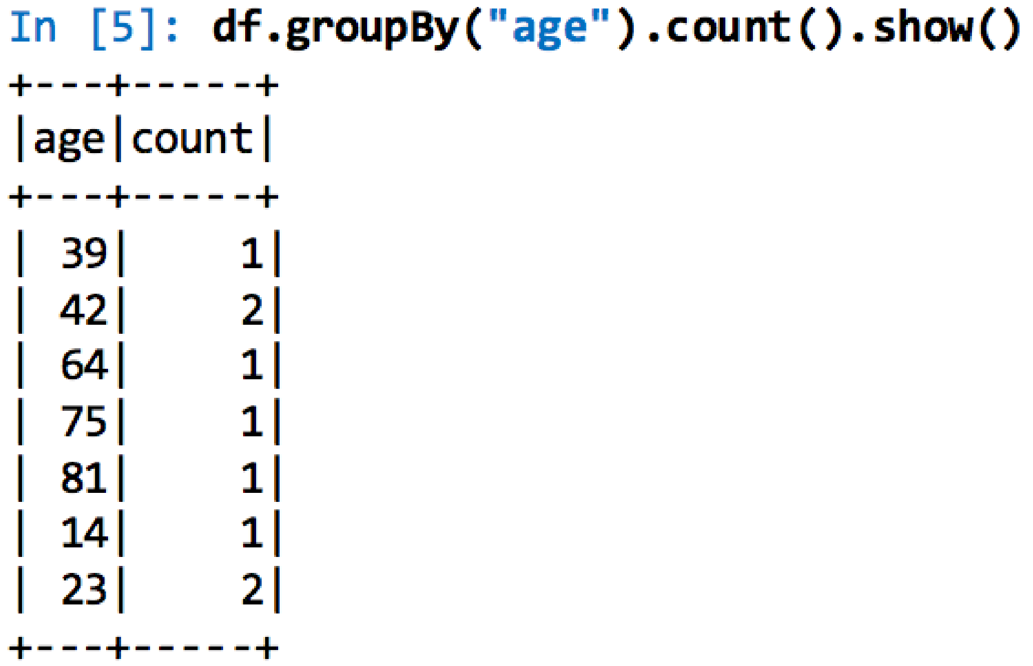
- The orderBy() method allows you to sort the results
    - ORDER BY age DESC in SQL



```
In [1]: df.filter(df['age'] > 49).\
            select(df['first_name'], df['age']).\
            orderBy(df['age'].desc(), df['first_name']).show()
+----------+---+
|first_name|age|
+----------+---+
|    Norman| 81|
|   Abigail| 75|
|    Miguel| 64|
+----------+---+
```

groupBy()

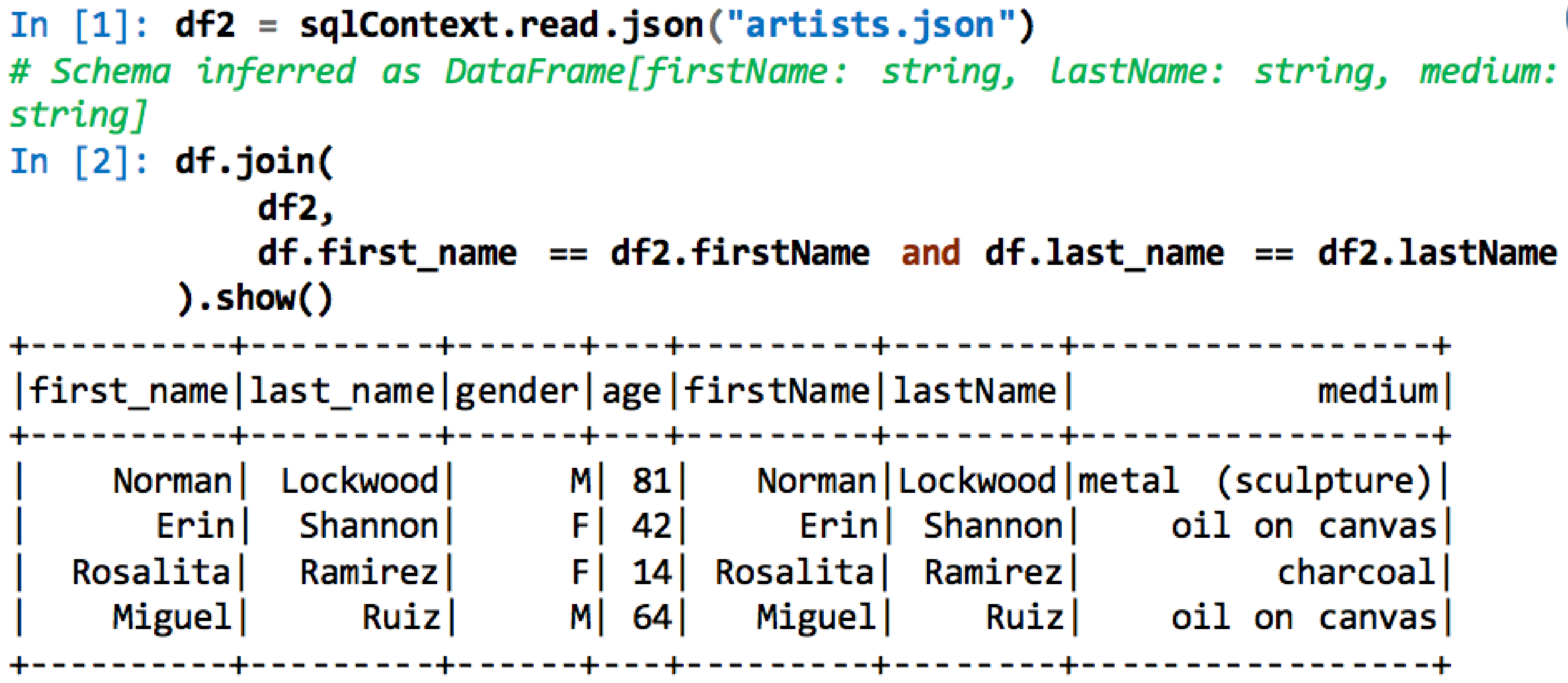
- Often followed by aggregation methods (e.g., sum(), count(), …), groupBy() groups data items by a specific column value.
  - GROUP BY in SQL



```
In [5]: df.groupBy("age").count().show()
+---+-----+
|age|count|
+---+-----+
| 39|    1|
| 42|    2|
| 64|    1|
| 75|    1|
| 81|    1|
| 14|    1|
| 23|    2|
+---+-----+
```

join()

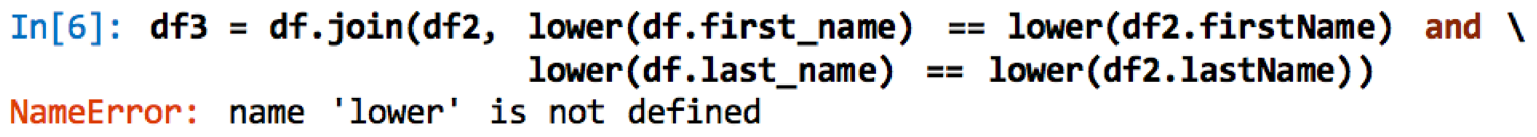
- We can load that into a second DataFrame and join it with our first one



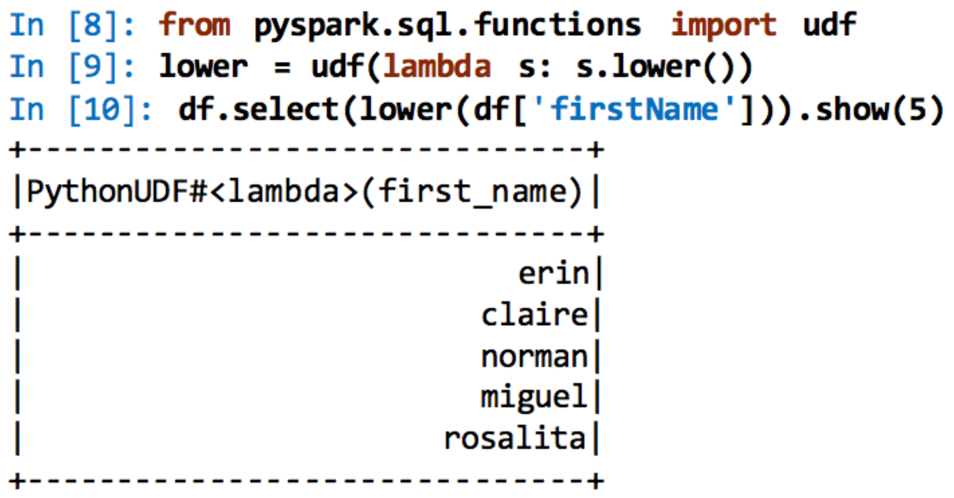
```
In [1]: df2 = sqlContext.read.json("artists.json")
# Schema inferred as DataFrame[firstName: string, lastName: string, medium:
string]
In [2]: df.join(
            df2,
            df.first_name == df2.firstName and df.last_name == df2.lastName
        ).show()
+----------+---------+------+---+---------+--------+------------------+
|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

- If we want to force all names to lower case before joining



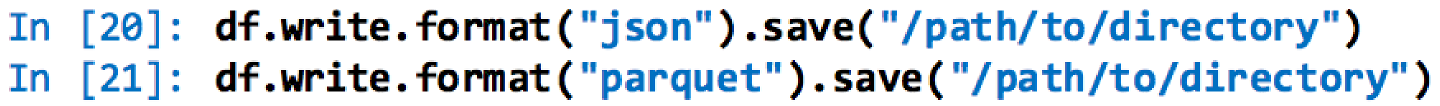
```
In[6]: df3 = df.join(df2, lower(df.first_name) == lower(df2.firstName) and \
                          lower(df.last_name) == lower(df2.lastName))
NameError: name 'lower' is not defined
```



```
In [8]: from pyspark.sql.functions import udf
In [9]: lower = udf(lambda s: s.lower())
In [10]: df.select(lower(df['firstName'])).show(5)
+------------------------------+
|PythonUDF#<lambda>(first_name)|
+------------------------------+
|                          erin|
|                        claire|
|                        norman|
|                        miguel|
|                      rosalita|
+------------------------------+
```

Writing/save DataFrames

- In most cases, if you can read a data format, you can write that data format
- If you're writing to a text file format (e.g., JSON), you'll typically get multiple output files



```
In [20]: df.write.format("json").save("/path/to/directory")
In [21]: df.write.format("parquet").save("/path/to/directory")
```