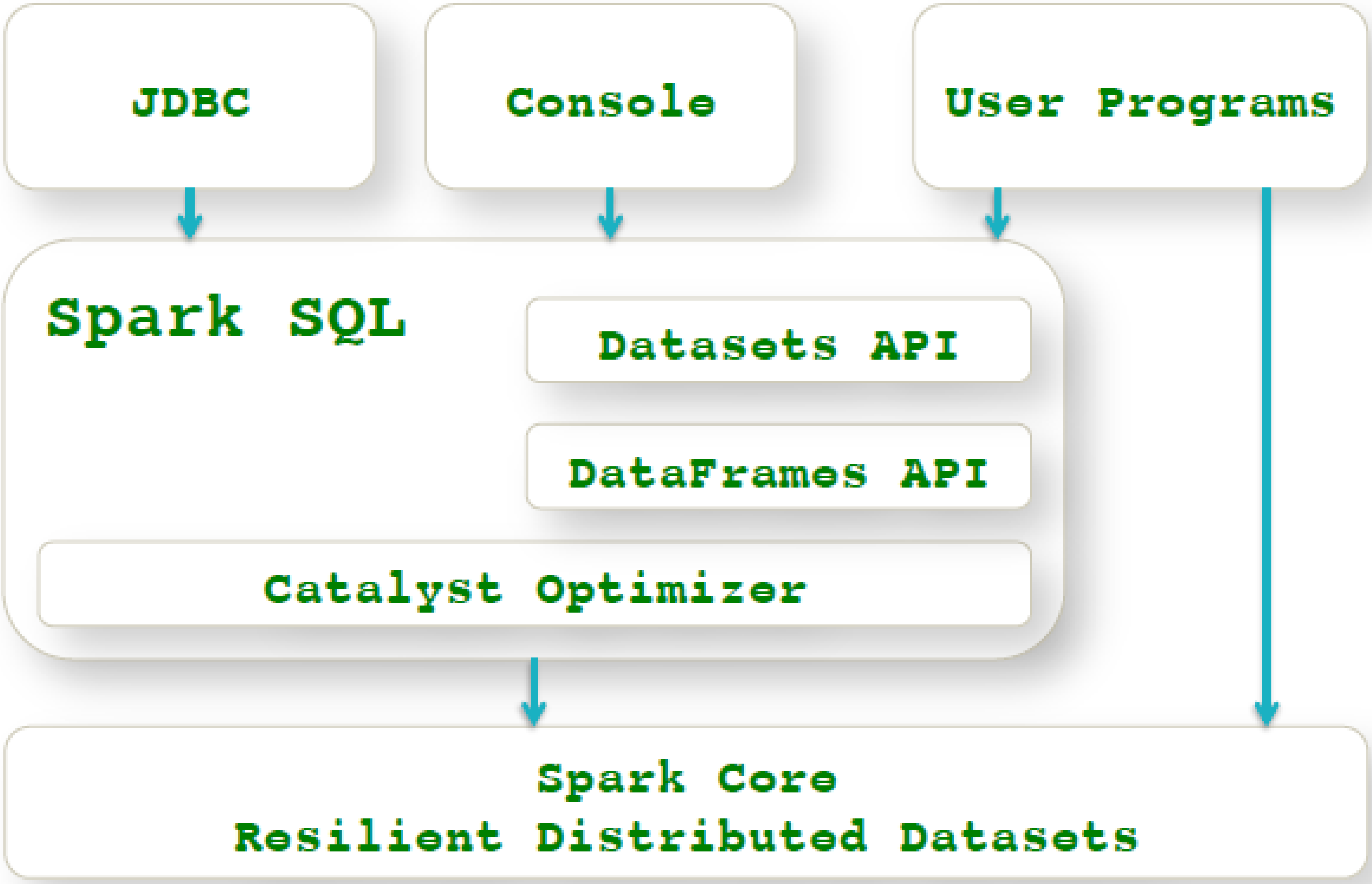
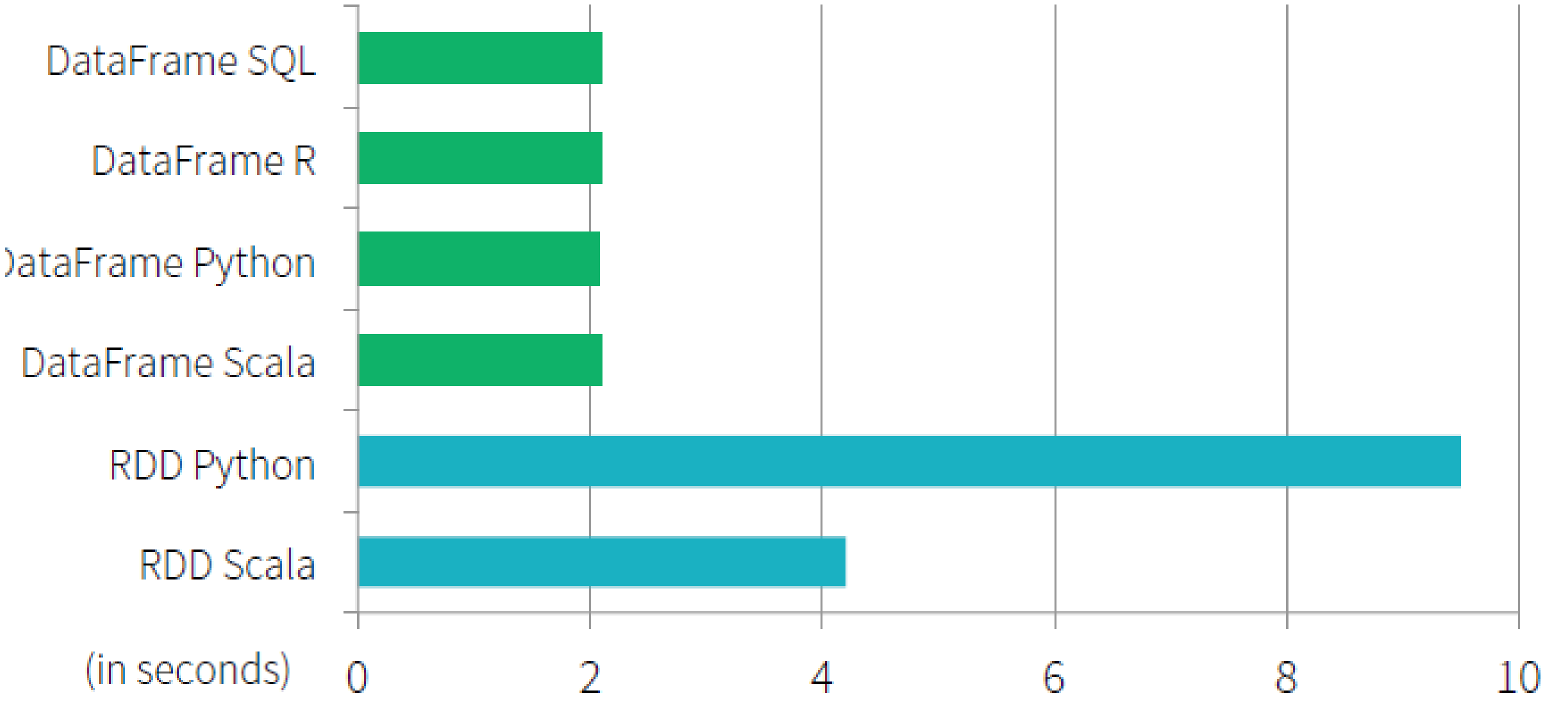
Apache Spark SQL

Spark SQL - Architecture



Performance Comparison

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



Code Size - Compute an Average

hadoop

```
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);
}
```

Spark

Using RDDs

```
var data = sc.textFile(...).split("\t")
data.map { x => (x(0), (x(1), 1))) }
    .reduceByKey { case (x, y) =>
      (x._1 + y._1, x._2 + y._2) }
    .map { x => (x._1, x._2(0) / x._2(1)) }
    .collect()
```

Using DataFrames

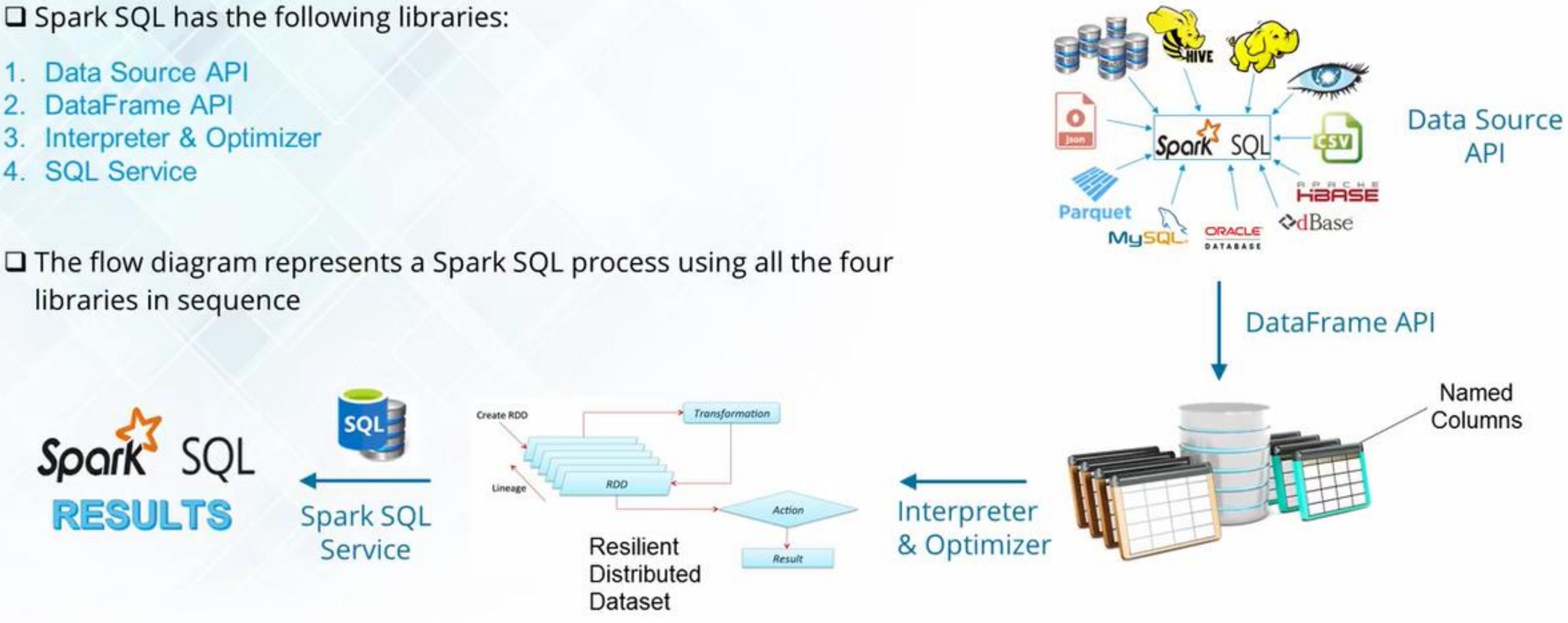
```
sqlContext.table("people")
          .groupBy("name")
          .agg("name", avg("age"))
          .collect()
```

Spark SQL Flow Diagram

- Spark SQL has the following libraries:

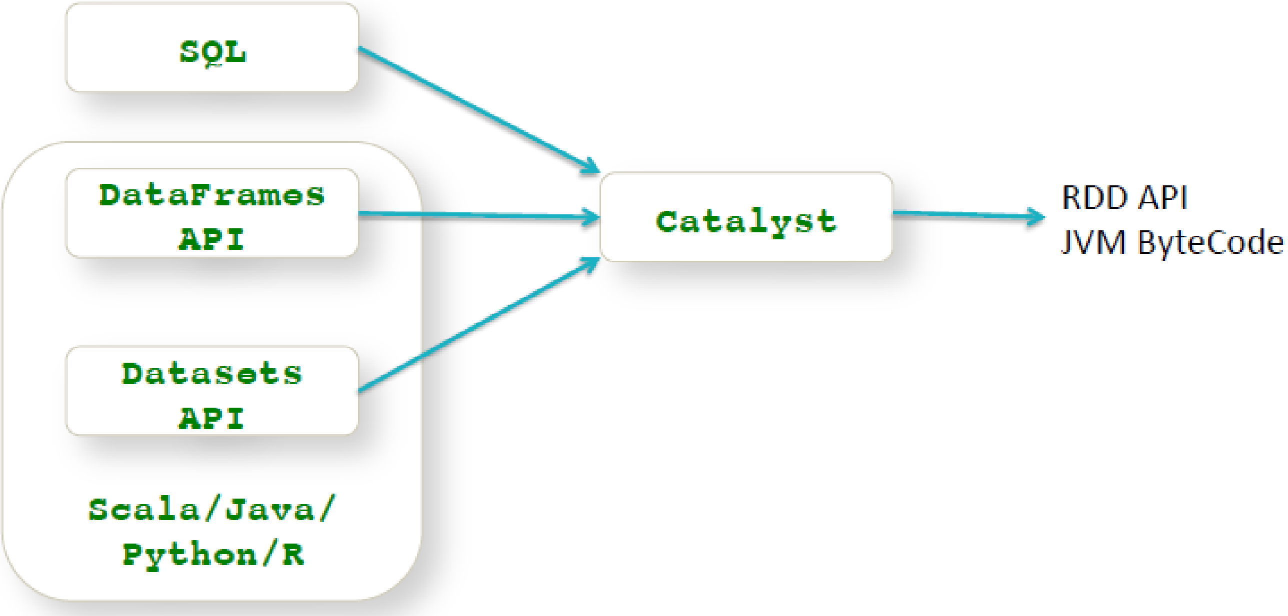
1. Data Source API
2. DataFrame API
3. Interpreter & Optimizer
4. SQL Service

- The flow diagram represents a Spark SQL process using all the four libraries in sequence



Catalyst Optimizer

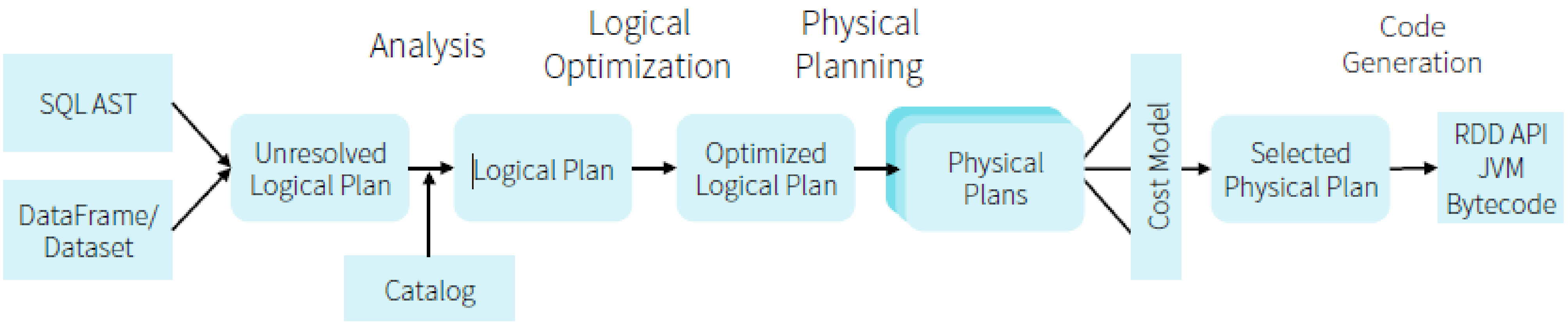
- Generates logical plan
- Optimize the plan
- Generate physical plan
- Generate byte code
- Send it to Spark Core for execution



- DataFrames, Datasets and SQL share the same optimization/execution pipeline

Catalyst Optimizer…

- It gets the catalog information (In – memory Catalog or Hive Metastore) and converts the unresolved logical plan to logical plan
- It then evaluates logical plan based on the rules built in catalyst optimizer and comes up with optimized logical plan
- It then comes with multiple physical plans
- Uses cost model to select least cost physical plan
- Then it generates RDD JVM bytecode



SQLContext and HiveContext

- Both are Entry points for creating DataFrame objects

SQL Context:

- Supports only the “sql” dialect
- val sqlContext = new org.apache.spark.sql.SQLContext(sc)

where sc is Spark Context

Hive Context:

- Uses the “hiveql” dialect by default but supports the “sql” dialect as well
- HiveContext extends SQLContext and provides additional features
- Requires Spark to be built with Hive support
- Brings in additional JARs (Hive dependencies)
- val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)

Features of Hive Context:

- Write queries using the more complete HiveQL query language
- Access to Hive user-defined functions
- Read/write data from/to Hive tables
- Interact with an existing Hive metastore or create a new one

SparkSession

- From Spark 2.0.0 onwards, SparkSession provides a single point of entry to interact with underlying Spark functionality
- In order to use APIs of SQL, HIVE, and Streaming, no need to create separate contexts as sparkSession includes all the APIs.

- Creating Spark session:

```
val spark = SparkSession
.builder
.appName(“WorldBankIndex”)
.getOrCreate()
```

- `val df = spark.read.json("path/to/file.json")`

What is a DataFrame

- Inspired by data frames in R and Python (Pandas)
- DataFrame is a distributed collection of data organized into rows and named columns
- When you create a DataFrame, it ends up as a pair of RDD[Row] and StructType (Schema)
- A schema describes the columns and for each column it defines the name, the type and whether or not it accepts empty values.

- Dataframe is similar to RDD and only difference is it carries schema associated with the data to be processed
- Dataframe – RDD with structure of the data

- It is conceptually equivalent to a table in a relational database
- Evaluation is lazy, just like RDDs

Data Sources

DataFrames can be constructed from the below datasources:

- Structured data files(JSON files, Parquet files)
- Tables in Hive
- Existing RDDs
- It can read from local file systems, distributed file systems (HDFS), cloud storage (S3), and external relational database systems via JDBC.

DataFrame API vs RDD API

- DataFrames are built on top of the Spark RDD API
- It is a Declarative API
- You can access the RDD corresponding to a DataFrame
- You can create a DataFrame out of an RDD

RDD API vs DataFrame API:

- Use DataFrame API, wherever possible.
- DataFrame API is likely to be more efficient, because it can optimize the underlying operations with Catalyst
- If there is a part of your problem that cannot be expressed in DataFrame API, express it using RDD API

DataFrame API - Operations

- Operations on a DataFrame are divided into

Transformations:

- Return a new DataFrame
- Build up an operator DAG/Lineage
- Evaluated lazily

Actions:

- Return a value to user
- Store a result to stable storage
- Force the operator DAG/Lineage to be evaluated

RDD interop:

- Create DataFrame from RDD or vice versa
- Run RDD operations on DataFrame

Transformation examples	Action examples
• filter	• count
• select	• collect
• drop	• show
• intersect	• head
• join	• take

Data Pipeline

- We can express the pipeline in two ways

1. You can incorporate SQL/HiveQL while working with DataFrames

Register the DataFrame as a temporary table and run SQL

createOrReplaceTempView(viewName) – Has replaced registerTemptable

```
df.createOrReplaceTempView("people")
adultsDF = sqlContext \
  .sql("select name, age from people where age > 21")
```

2. You can use DataFrame API methods

```
df.select("name", "age") \
  .filter(df['age'] > 21) \
  .groupBy("age").count() \
  .show()
```

- A single pipeline can mix and match both methods

Schema Inference

- What if your data file doesn’t have a schema?
- CSV file or a plain text file

- There are 2 types of schema inference

1. Automatic schema inference:

- Start by creating an RDD from the dataset
- Map the elements to a data type from which Spark can infer schema
  - Python – namedtuple, dict, pyspark.sql.Row
  - Scala – Tuple, case class, org.apache.spark.sql.Row

2. Manual schema specification

- Use the DataFrameReader.schema(dfSchema) method
- Use the SQLContext.createDataFrame API

```
val schema = StructType(
    StructField("lat", DoubleType, false)
    StructField("long", DoubleType, false)
    StructField("key", StringType, false)

val df = sqlContext.createDataFrame(rdd, schema)
```

Schema Inference Example

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

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

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

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

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

```scala
case class Person(firstName: String,
                  lastName:  String,
                  gender:    String,
                  age:       Int)

val rdd = sc.textFile("people.csv")
val peopleRDD = rdd.map { line =>
  val cols = line.split(",")
  Person(cols(0), cols(1), cols(2), cols(3).toInt)
}
val df = peopleRDD.toDF("firstName", "lastName",
                        "gender", "age")

// df: DataFrame = [firstName: string, lastName: string,
gender: string, age: int]
```

- To check if spark can support any datasource, refer the below link

http://spark-packages.org/

Save Operation – Persistent Table

- Saving data in a DataFrame to Hive tables

```
df.write \
   .format("parquet") \
   .mode("append") \
   .partitionBy("year") \
   .saveAsTable("faster-stuff")
```

Scala/Java	Python	Meaning
`SaveMode.ErrorIfExists` (default)	`"error"`	If output data or table already exists, an exception is expected to be thrown.
`SaveMode.Append`	`"append"`	If output data or table already exists, append contents of the DataFrame to existing data.
`SaveMode.Overwrite`	`"overwrite"`	If output data or table already exists, replace existing data with contents of DataFrame.
`SaveMode.Ignore`	`"ignore"`	If output data or table already exists, do not write DataFrame at all.

- Saving data in a DataFrame to files

```
# Default data source is "parquet"
df.write.save("faster-stuff.parquet")

# Specify "format" for other data sources
df.write.save("/path/to/data.json", format="json")
```

Caching

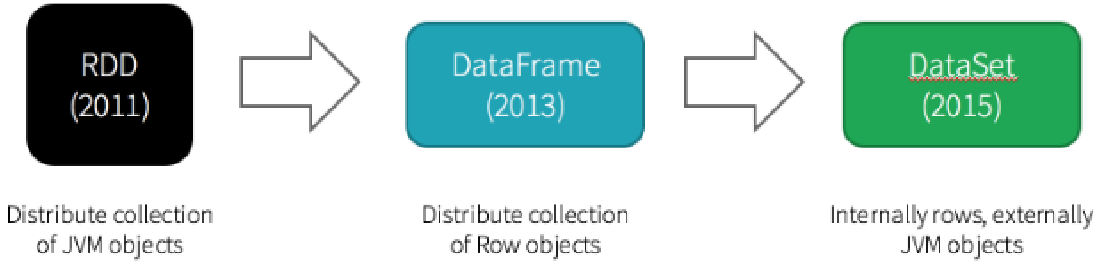
- Spark can cache a DataFrame, using an in-memory columnar format
- Spark will scan only those columns used by the DataFrame

```
df.cache()
# --OR—
sqlContext.cacheTable("table")
```

- Cache() calls df.persist(MEMORY_ONLY)
- unpersist() method is used to remove the cached data from memory

Dataset

- Dataset tries to provide the benefits of RDDs with the benefits of Spark SQL's optimized execution engine.
- Type Safety – Compile time type safety
- Dataset in Spark provides Optimized query using Catalyst Query Optimizer and Tungsten



	RDD	DataFrame	Dataset
Immutability	✓	✓	✓
Schéma	X	✓	✓
Performance optimization	X	✓	✓
Typed	✓	X	✓
Syntax Error	Compile time	Compile time	Compile time
Analysis Error	Compile time	Runtime	Compile time

Dataset Preserve schema when converted back to RDD.

Dataset – Type Safety

```scala
case class Person(name : String , age : Int)
val personRDD = List(Person("Paul",10),Person("Anna",23))
val persondf = personRDD.toDF
persondf.filter("age > 21").show()
persondf.filter("salary > 10000").show()
```

```
+----+---+
|name|age|
+----+---+
|Anna| 23|
+----+---+
```

```
⊞org.apache.spark.sql.AnalysisException: cannot resolve '`salary`' given input columns: [name, age]; line 1 pos 0;
```

```scala
case class Person(name : String , age : Int)
val personRDD = List(Person("Paul",10),Person("Anna",23))

val personds = personRDD.toDS()
personds.filter(_.age > 21).show()
personds.filter(_.salary > 10000).show()
```

```
notebook:6: error: value salary is not a member of Person
personds.filter(_.salary > 10000).show()
                  ^
```

Dataset - Interoperability

- To convert an RDD into a Dataset, call rdd.toDS().
- Call df.as[SomeCaseClass] to convert the DataFrame to a Dataset.

```
val rdd = sc.parallelize(Seq((1, "Spark"), (2, "Databricks")))
val integerDS = rdd.toDS()
integerDS.show()
```

```
case class Company(name: String, foundingYear: Int, numEmployees: Int)
val inputSeq = Seq(Company("ABC", 1998, 310), Company("XYZ", 1983, 904), Company("NOP", 2005, 83))
val df = sc.parallelize(inputSeq).toDF()

val companyDS = df.as[Company]
companyDS.show()
```

DataFrame Operations

Operations	Description	Example

printSchema()	• Prints schema in tree ascii art format
	• Useful for data exploration
		`scala> df.printSchema()`
		`root`
		` |-- firstName: string (nullable = true)`
		` |-- lastName: string (nullable = true)`
		` |-- gender: string (nullable = true)`
		` |-- age: integer (nullable = false)`

show()	• Reads the input source
	• Executes the RDD DAG across the cluster
	• Pulls the n elements back to the driver JVM
	• Displays those elements in a tabular form
		`scala> df.show()`
		`+---------+--------+------+---+`
		`|firstName|lastName|gender|age|`
		`+---------+--------+------+---+`
		`|     Erin| Shannon|     F| 42|`
		`|   Claire| McBride|     F| 23|`
		`|   Norman|Lockwood|     M| 81|`

select()	• Like a SQL SELECT
	• Create on-the-fly derived columns.
		`scala> df.select($"firstName",`
		`                 $"age",`
		`                 $"age" > 49,`
		`                 $"age" + 10).show(5)`
		`+---------+---+----------+----------+`
		`|firstName|age|(age > 49)|(age + 10)|`
		`+---------+---+----------+----------+`
		`|     Erin| 42|     false|        52|`
		`|   Claire| 23|     false|        33|`

DataFrame Operations…

Operations	Description	Example

filter()	• Filter rows out of the results.
	`scala> df.filter($"age" > 49).select($"firstName", $"age").show()`
		`+---------+---+`
		`|firstName|age|`
		`+---------+---+`
		`|   Norman| 81|`
		`|   Miguel| 64|`
		`|  Abigail| 75|`
		`+---------+---+`

orderBy()	• To sort the results
	`scala> df.filter($"age" > 49).`
	`select($"firstName", $"age").`
	`orderBy($"age".desc, $"firstName").`
	`show()`
		`+----------+---+`
		`|first_name|age|`
		`+----------+---+`
		`|    Norman| 81|`
		`|   Abigail| 75|`
		`|    Arnold| 75|`
		`|    Miguel| 64|`
		`+----------+---+`

groupBy()	• Groups data items by a specific column value.
	`df.groupBy("age").count().show()`
		`+---+-----+`
		`|age|count|`
		`+---+-----+`
		`| 39|    1|`
		`| 42|    2|`
		`| 64|    1|`

as() or alias()	• rename a column
	`scala> df.select($"firstName", $"age", ($"age" < 30).as("young")).`
	`show()`
		`+----------+---+-----+`
		`|first_name|age|young|`
		`+----------+---+-----+`
		`|      Erin| 42|false|`
		`|    Claire| 23| true|`

DataFrame Operations…

Operations	Description
limit(n)	• Limit the results to n rows.
distinct()	• Returns a new DataFrame containing only the unique rows
drop(column)	• Returns a new DataFrame with a column dropped
explain()	• Gives the physical plan of the query • Pass true to get a more detailed query plan

join(dataframe)	• Join one DataFrame with another
	`young = users.filter(users.age < 21)`
	`# Join young users with another DataFrame called logs`
	`young.join(logs, logs.userId == users.userId, "left_outer")`

Explode()

- Used in nested objects
- A single column value will be exploded into multiple values, one per row

```
> select name, number_of_employees, offices from companies
```

(1) Spark Jobs

name	number_of_employees	offices
Wetpaint	47	[{"address1":"710 - 2nd Avenue","address2":"Suite 1100","city":"Seattle","country_code":"USA","description":"","latitude":47.603122,"longitude":-122.333253,"state_code":"WA","zip_code":"98104"},{"address1":"270 Lafayette Street","address2":"Suite 505","city":"New York","country_code":"USA","description":"","latitude":40.7237306,"longitude":-73.9964312,"state_code":"NY","zip_code":"10012"}]
AdventNet	600	[{"address1":"4900 Hopyard Rd.","address2":"Suite 310","city":"Pleasanton","country_code":"USA","description":"Headquarters","latitude":37.692934,"longitude":-121.904945,"state_code":"CA","zip_code":"94588"}]

```
> select name, number_of_employees, offices.city from companies
```

(1) Spark Jobs

name	number_of_employees	city
Wetpaint	47	["Seattle","New York"]
AdventNet	600	["Pleasanton"]
Zoho	1600	["Pleasanton"]

```
> select name, number_of_employees, explode(offices.city) as city from companies
```

(1) Spark Jobs

name	number_of_employees	city
Wetpaint	47	Seattle
Wetpaint	47	New York
AdventNet	600	Pleasanton
Zoho	1600	Pleasanton

Explain Plan

```
scala> df.join(df2, lower(df("firstName")) === lower(df2("firstName"))).explain(true)
== Parsed Logical Plan ==
Join Inner, Some((Lower(firstName#1) = Lower(firstName#13)))
 Relation[birthDate#0,firstName#1,gender#2,lastName#3,middleName#4,salary#5L,ssn#6]
org.apache.spark.sql.json.JSONRelation@7cbb370e
 Relation[firstName#13,lastName#14,medium#15] org.apache.spark.sql.json.JSONRelation@e5203d2c

== Analyzed Logical Plan ==
birthDate: string, firstName: string, gender: string, lastName: string, middleName: string, salary: bigint, ssn: string,
firstName: string, lastName: string, medium: string
Join Inner, Some((Lower(firstName#1) = Lower(firstName#13)))
 Relation[birthDate#0,firstName#1,gender#2,lastName#3,middleName#4,salary#5L,ssn#6]
org.apache.spark.sql.json.JSONRelation@7cbb370e
 Relation[firstName#13,lastName#14,medium#15] org.apache.spark.sql.json.JSONRelation@e5203d2c

== Optimized Logical Plan ==
Join Inner, Some((Lower(firstName#1) = Lower(firstName#13)))
 Relation[birthDate#0,firstName#1,gender#2,lastName#3,middleName#4,salary#5L,ssn#6]
org.apache.spark.sql.json.JSONRelation@7cbb370e
 Relation[firstName#13,lastName#14,medium#15] org.apache.spark.sql.json.JSONRelation@e5203d2c

== Physical Plan ==
ShuffledHashJoin [Lower(firstName#1)], [Lower(firstName#13)], BuildRight
 Exchange (HashPartitioning 200)
  PhysicalRDD [birthDate#0,firstName#1,gender#2,lastName#3,middleName#4,salary#5L,ssn#6], MapPartitionsRDD[40] at explain at
<console>:25
 Exchange (HashPartitioning 200)
  PhysicalRDD [firstName#13,lastName#14,medium#15], MapPartitionsRDD[43] at explain at <console>:25

Code Generation: false
== RDD ==
```

User Defined Functions

- Define and register UDF and then use it on a Spark DataFrame.

```
scala> val double = sqlContext.udf.register("double",
                                            (i: Int) => i.toDouble)
```

```
scala> df.select(double($("total"))).show(5)
+---------------+
|scalaUDF(total)|
+---------------+
|         7065.0|
|         2604.0|
|         2003.0|
|         1939.0|
|         1746.0|
+---------------+
```

Spark Shared Variables

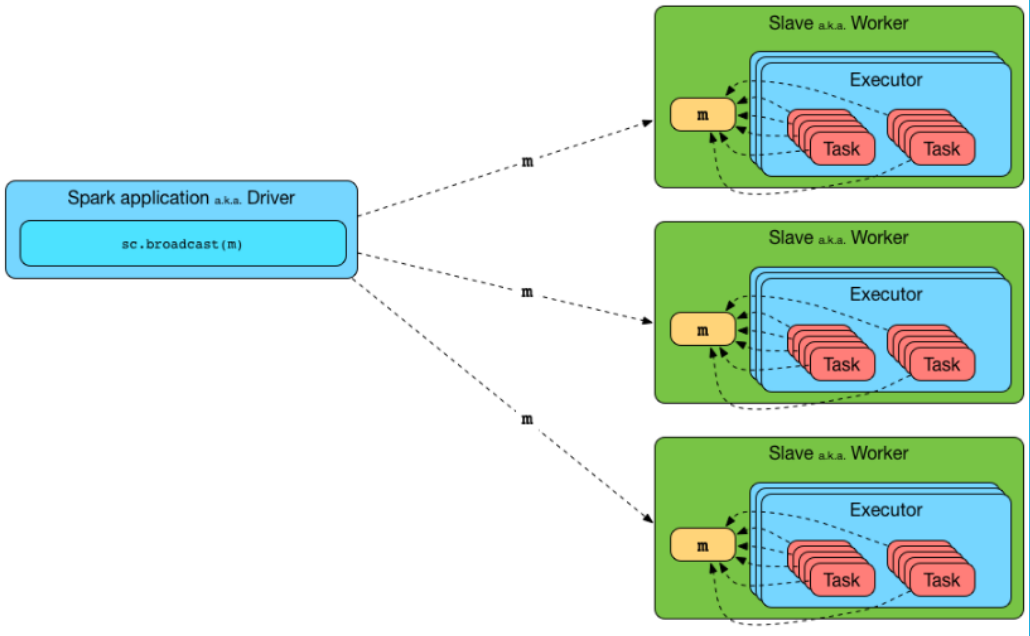
- In spark, while doing any functions/spark operation, it works on different variables used in that function.
- Generally, multiple copies of same variables copied to each worker node and the update to this variable never return to driver program
- This is an inefficient way as the data transfer rate will be very high here
- Spark Shared Variables help in reducing data transfer

There two types of shared variables

- Broadcast variable
- Accumulator

Broadcast Variable

- Copy the data to each node at one time and share the same data for each task in that node.
- Spark automatically broadcasts the common data needed by tasks within each stage.
- The data broadcasted this way is cached in a serialized form and deserialized before running each task.



- First, we need to create a broadcast variable using SparkContext.broadcast and then broadcast the same to all nodes from driver program.
- Value method – to access the shared value
- Unpersist method - removes it from the executors
- Destroy method - remove the broadcast variable from both executors and driver

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Broadcast variables can be considered:

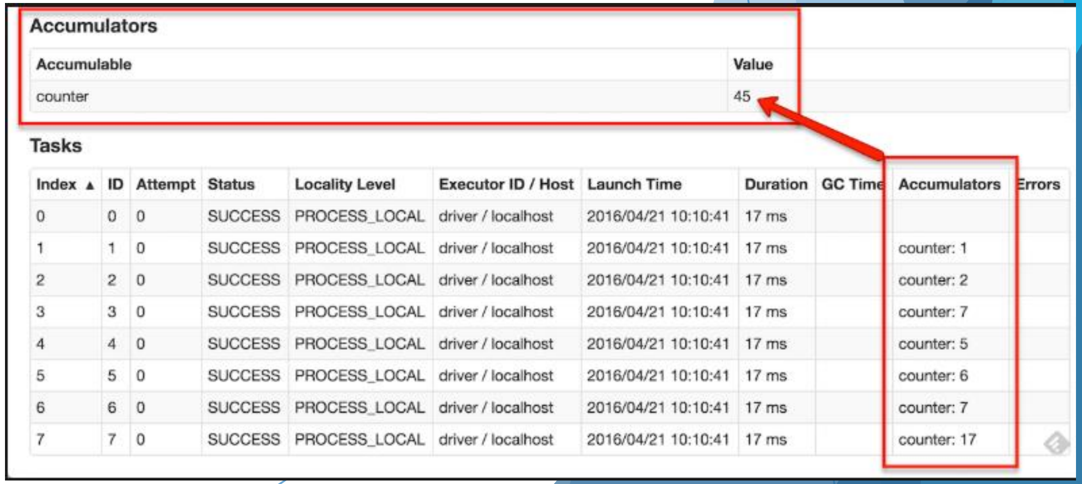
- There is read-only reference data that does not change throughout the life of your Spark application.
- The data is used across multiple stages of application execution and would benefit from being locally cached on the worker nodes.
- The data is small enough to fit in memory on your worker nodes, but large enough that the overhead of serializing and deserializing it multiple times is impacting your performance.

Accumulator

- Accumulators are variables that are used for aggregating information across the executors.
- They can be used to implement counters or sums
- An accumulator can have an optional name that you can specify when creating an accumulator.

```scala
// Accumulator Example
val counter = sc.longAccumulator("counter")
sc.parallelize(1 to 9).foreach(x => counter.add(x))
counter.value
```

In Spark Web UI



Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Broadcast Join - Example

```
movie_metadata = spark.read\
                          .format("csv")\
                          .option("sep", ",")\
                          .option("header", "true")\
                          .load("../datasets/movies_metadata.csv")
```

```
movie_ratings = spark.read\
                .format("csv")\
                .option("sep", ",")\
                .option("header", "true")\
                .load("../datasets/ratings.csv")
```

```
movies_joined = movie_ratings.select('movieId',
                                     'rating')\
                             .join(broadcast(movie_metadata),
                                   movie_ratings.movieId == movie_metadata.id, 'inner')
```

Accumulator Example

```
terrible_count = spark.sparkContext.accumulator(0)
subpar_count = spark.sparkContext.accumulator(0)
average_count = spark.sparkContext.accumulator(0)
good_count = spark.sparkContext.accumulator(0)
```

```
movies_joined.select("rating").describe().show()
```

```
def count_movie_by_rating(row):
    rating = float(row.rating)

    if (rating <= 2.0 ):
        terrible_count.add(1)
    elif (rating <= 3.0 and rating > 2.0 ):
        subpar_count.add(1)
    elif (rating <= 4.0 and rating > 3.0 ):
        average_count.add(1)
    elif (rating > 4.0) :
        good_count.add(1)
```

```
count_movie_by_rating(x)
```

```
print("Terrible movies: ", terrible_count.value)
print("Sub-par movies: ", subpar_count.value)
print("Average movies: ", average_count.value)
print("Good movies: ", good_count.value)
```

```
Terrible movies:  51857
Sub-par movies:  107724
Average movies:  137547
Good movies:  80021
```

References

- DataFrame API Documentation:

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrame

- Spark SQL Functions:

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$

- Spark SQL Query Support:

http://spark.apache.org/docs/latest/sql-programming-guide.html#reference

Thank You

Keerthiga Barathan