APACHE Spark SQL

is not about SQL

is about more than SQL

Spark SQL: The whole story

Is About Creating and Running Spark Programs Faster:

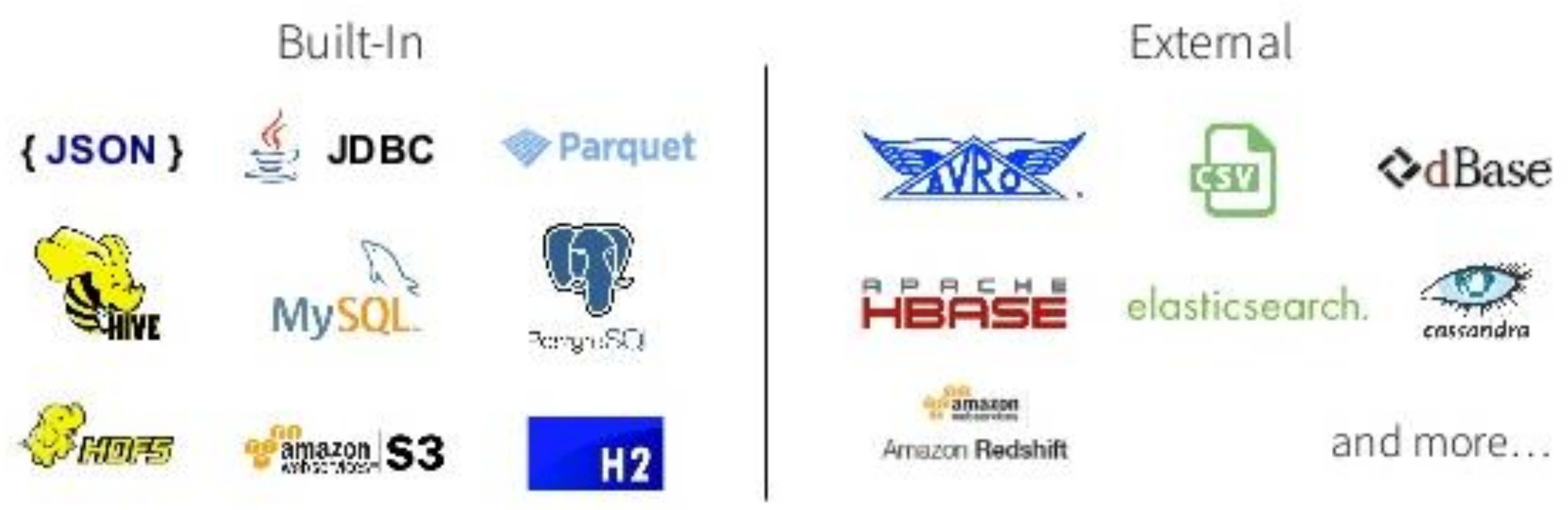
- Write less code
- Read less data
- Do less work
  - optimizer do the hard work

Why Spark SQL

- Coding Complexity is less
- Scalability
- Let optimizer do the hard work
- Integrated
  - Seamlessly mix SQL queries with Spark programs.
- Unified Data Access
  - Connect to any data source the same way.
- Hive Compatibility
  - Run unmodified Hive queries on existing data.
- Standard Connectivity
  - Connect through JDBC or ODBC.

Data Source API

Read and write with a variety of formats

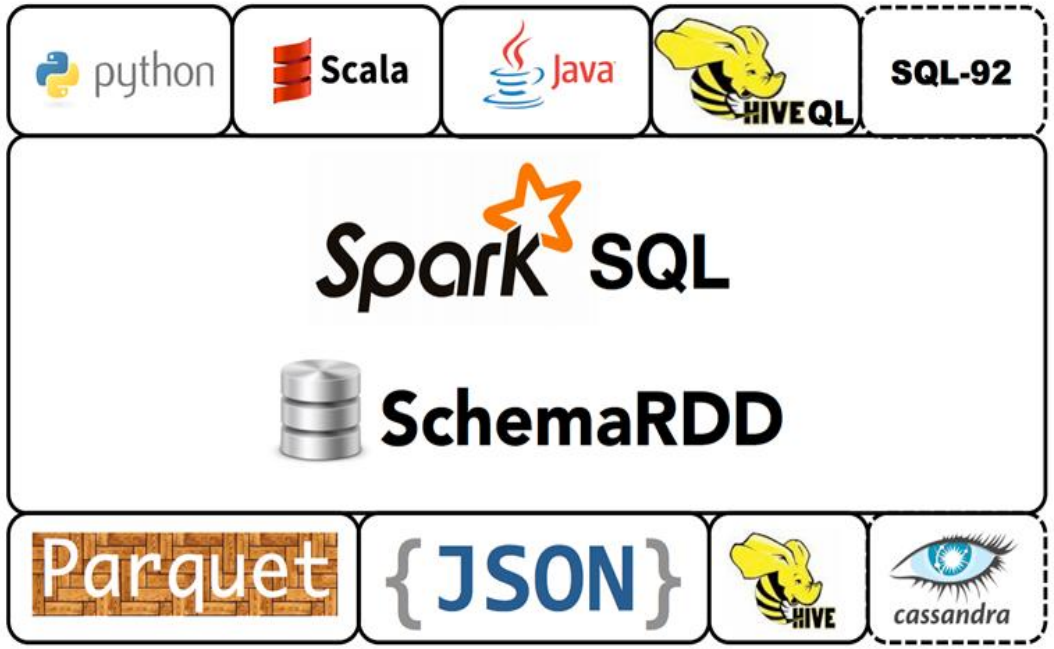


Data Source API

Unified interface to reading/writing data in a variety of formats

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

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



Spark SQL APIs

- **SQLContext**
  - It’s the entry point for working along structured data (rows and columns) in Spark. Allows the creation of Data Frame objects as well as the execution of SQL queries.
- **Hive Context**
  - Working with Hive tables, a descendant of SQLContext.
  - Hive Context is optimized and provides a richer functionality than SQLContext accesses unified metastore.
- **Datasets**
  - A Dataset provides the benefits of RDDs like strongly-typed, immutable collection of objects that are mapped to a relational schema are already available (i.e. you can use field of a row by name naturally row.columnName).
  - Datasets extends benefit of compile-time type safety – produced applications can be analyzed for errors before they are run.
- **JDBC Datasource**
  - JDBC data source can be used to read data from relational databases using JDBC API. This is preferred over using the RDD because the data source returns the results as a DataFrame can be handled in Spark SQL or joined beside other data sources.

DataFrames

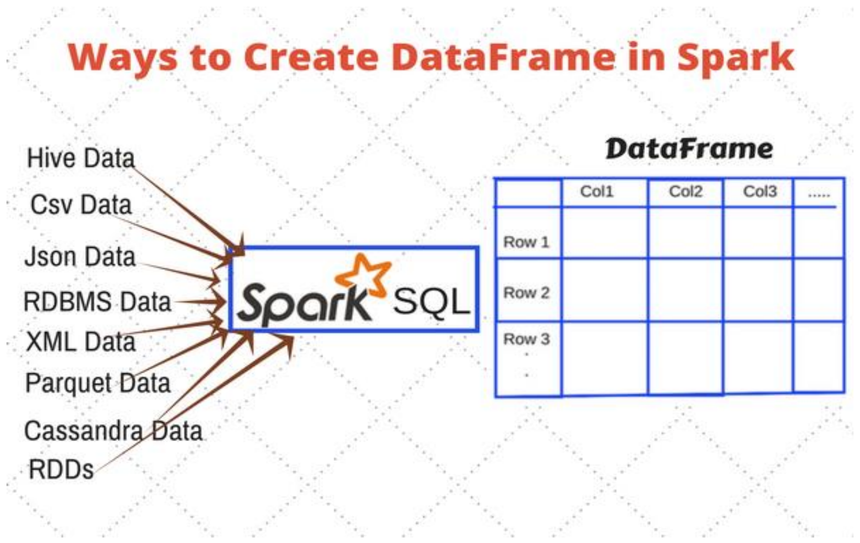
- Spark SQL uses a programming abstraction called DataFrame. It is a distributed collection of data,organized in named columns.
- DataFrame is equivalent to a database table but provides a much finer level of optimization.
- Spark Dataframe are lazily evaluated like Transformations in Apache Spark.
- A Data Frame is similar to a table in a relational database with richer optimization.
- It is a data abstraction and domain specific language (DSL) applicable on structured and semi structured data.
- DataFrame contains rows with Schema. The schema is the illustration of the structure of data.
- For accessing data frames either SQL Context or Hive Context is needed.
- DataFrames can be used to easily filter data, select column, count, average, and join data together from different sources.

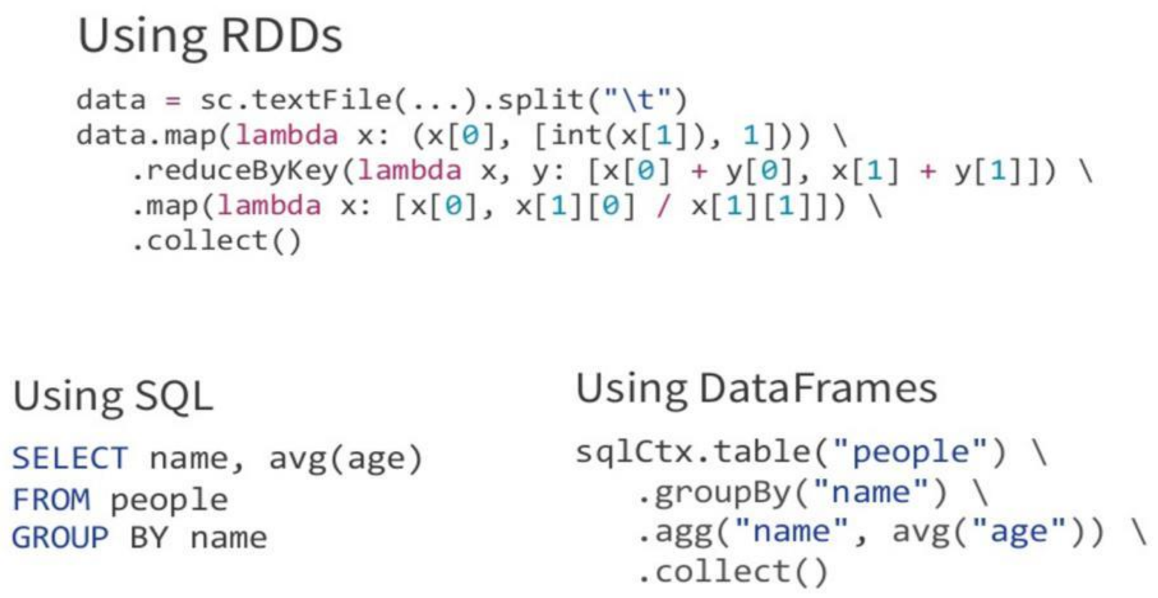
DataFrames

DataFrame can be created using

- ✓ Raw Data
- ✓ SQL over JDBC
- ✓ Hive tables

Writing less code





Using RDDs

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

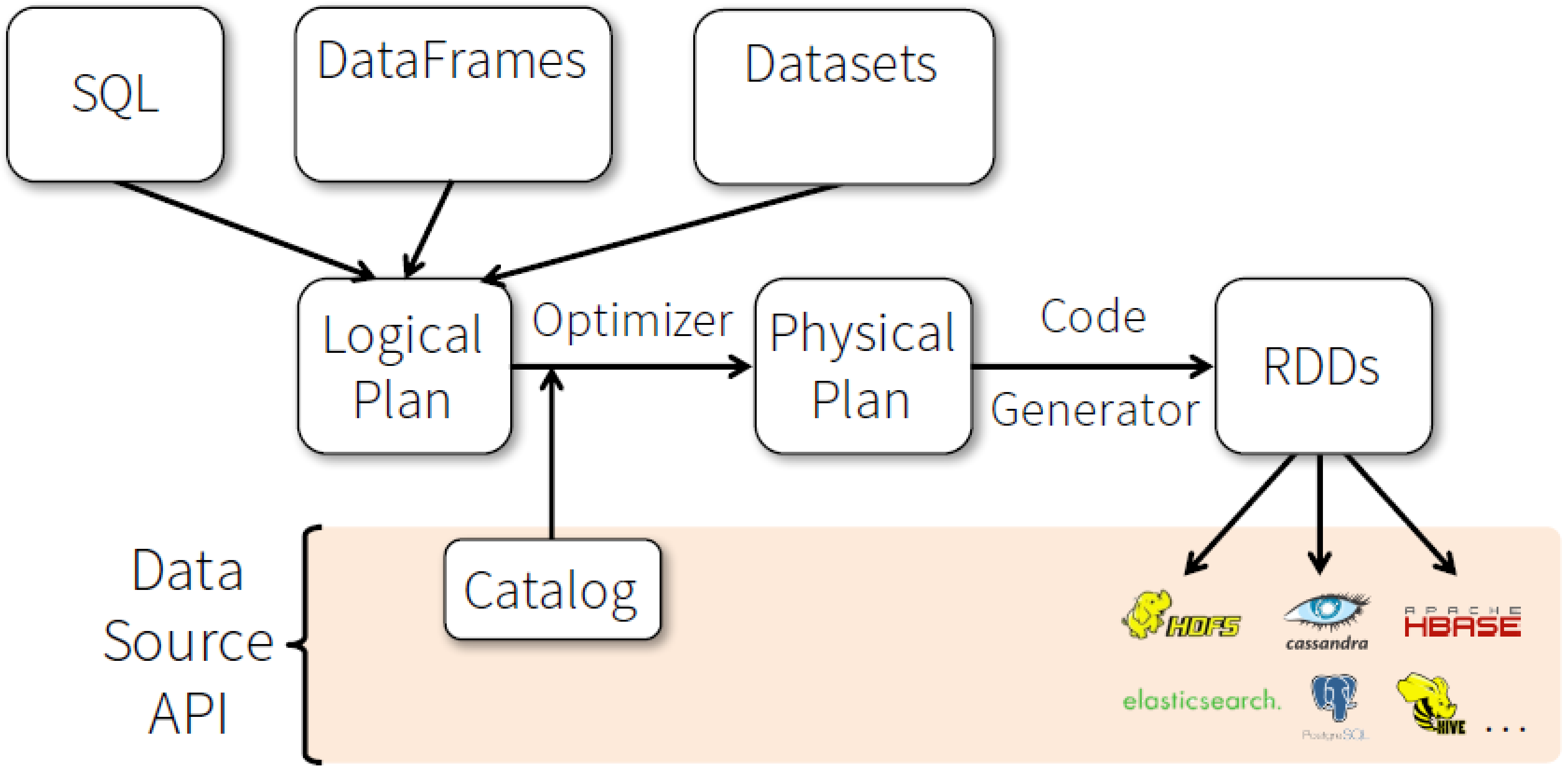
Using SQL

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

Using DataFrames

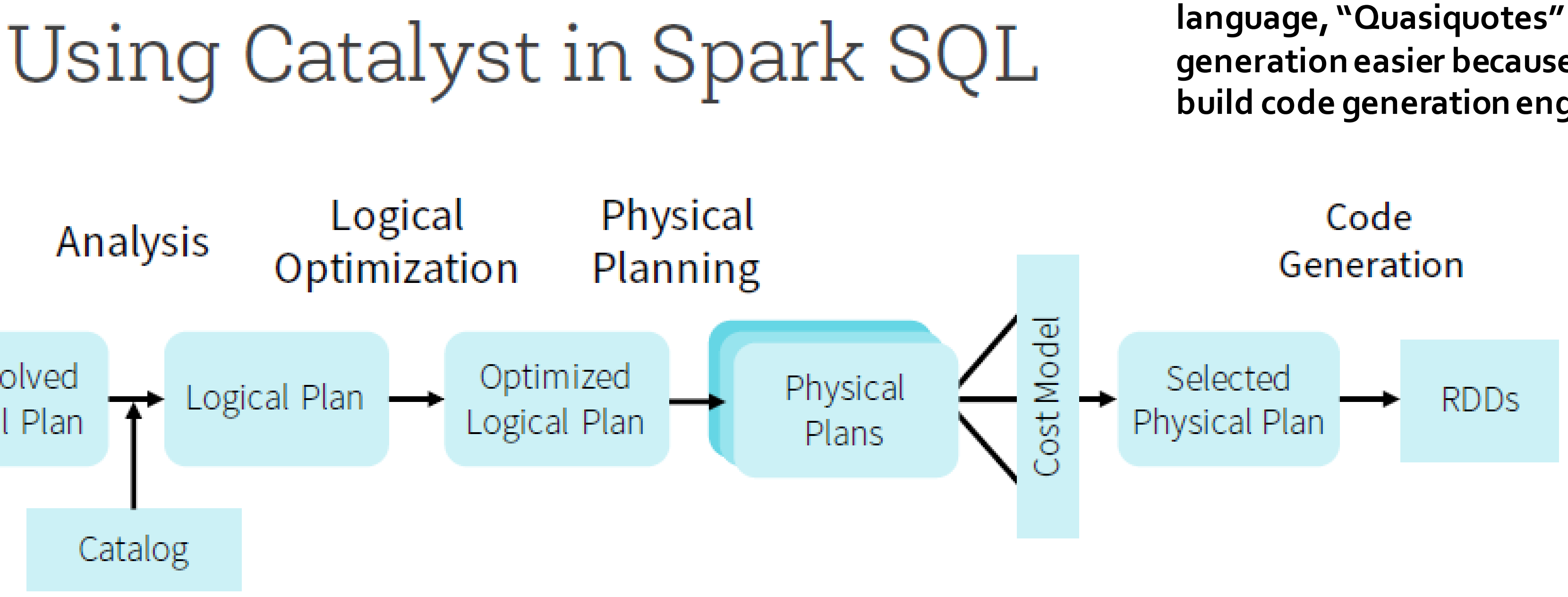
```
sqlCtx.table("people") \
   .groupBy("name") \
   .agg("name", avg("age")) \
   .collect()
```

Spark SQL Architecture



Spark SQL Architecture

Catalyst uses the special feature of Scala language, "Quasiquotes" to make code generation easier because it is very tough to build code generation engines



Analysis: analyzing a logical plan to resolve references

Logical Optimization: logical plan optimization

Physical Planning: Physical planning

Code Generation: Compile parts of the query to Java bytecode

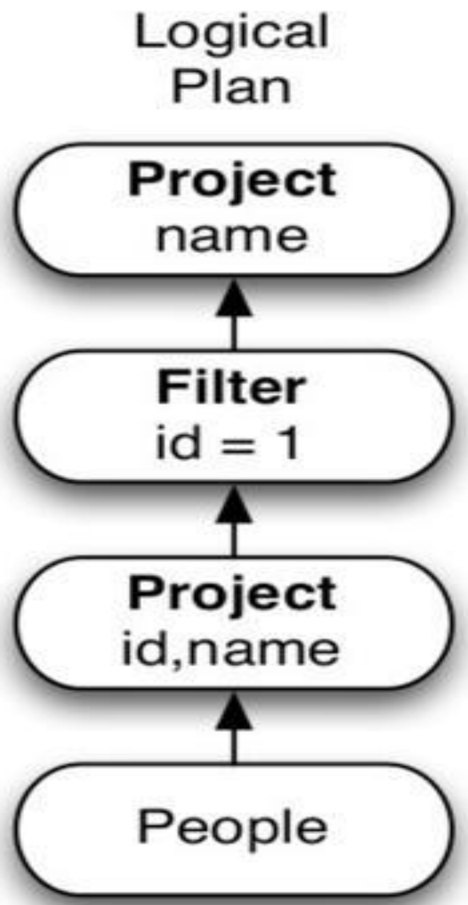
CATALYST OPTIMIZER

- ❖ Goal: convert logical plan to physical plans
- ❖ Process:
  - ✓ Logical plan is a tree representing data and schema
  - ✓ The tree is manipulated and optimized by catalyst rules
- ❖ Optimizing rules
  - ✓ Constant folding
  - ✓ Push Predicate through filter
  - ✓ Project collapsing

CATALYST OPTIMIZER

An example query:

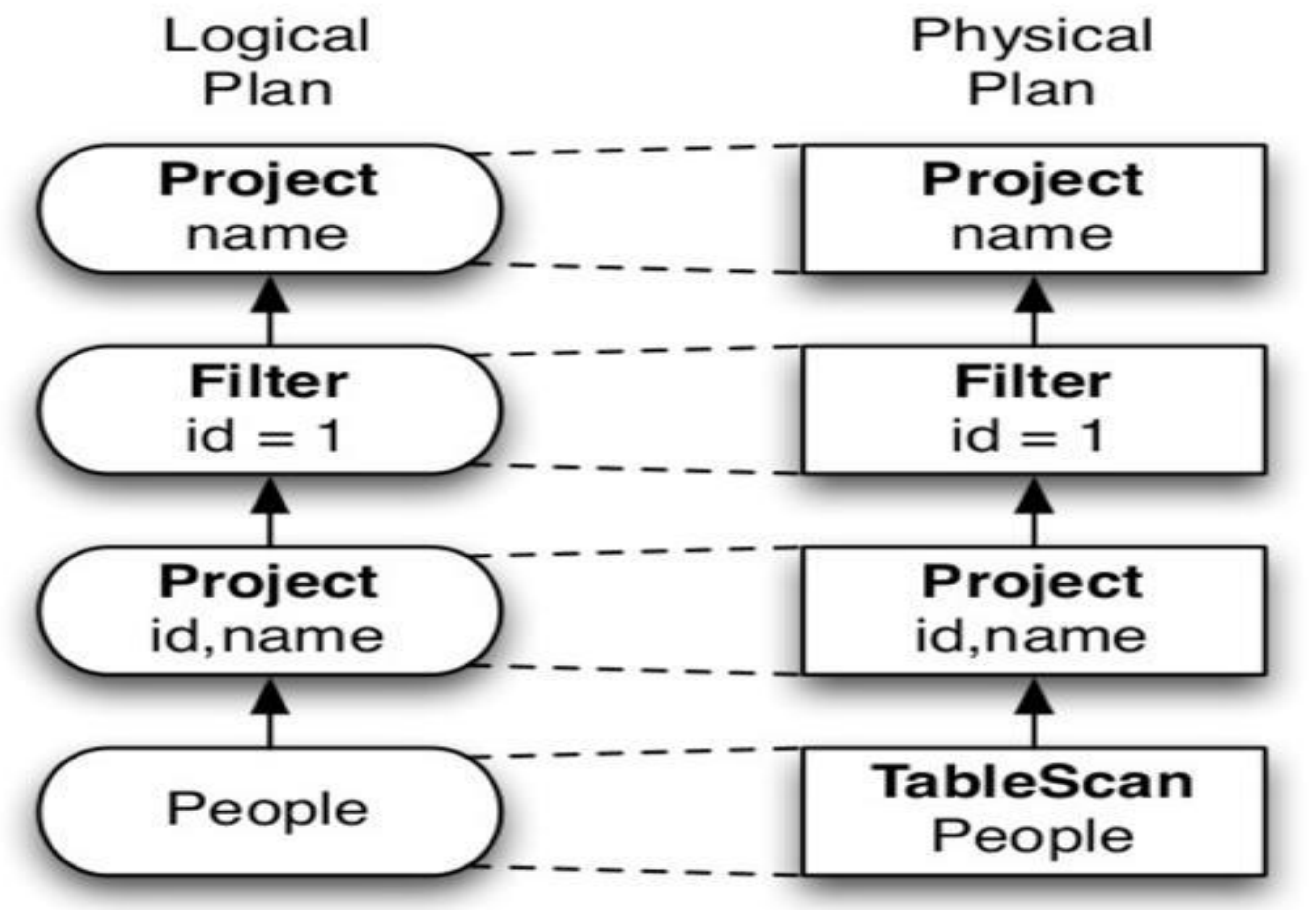
```
SELECT name
FROM (
    SELECT id, name
    FROM People) p
WHERE p.id = 1
```



CATALYST OPTIMIZER

Native Query Planning:

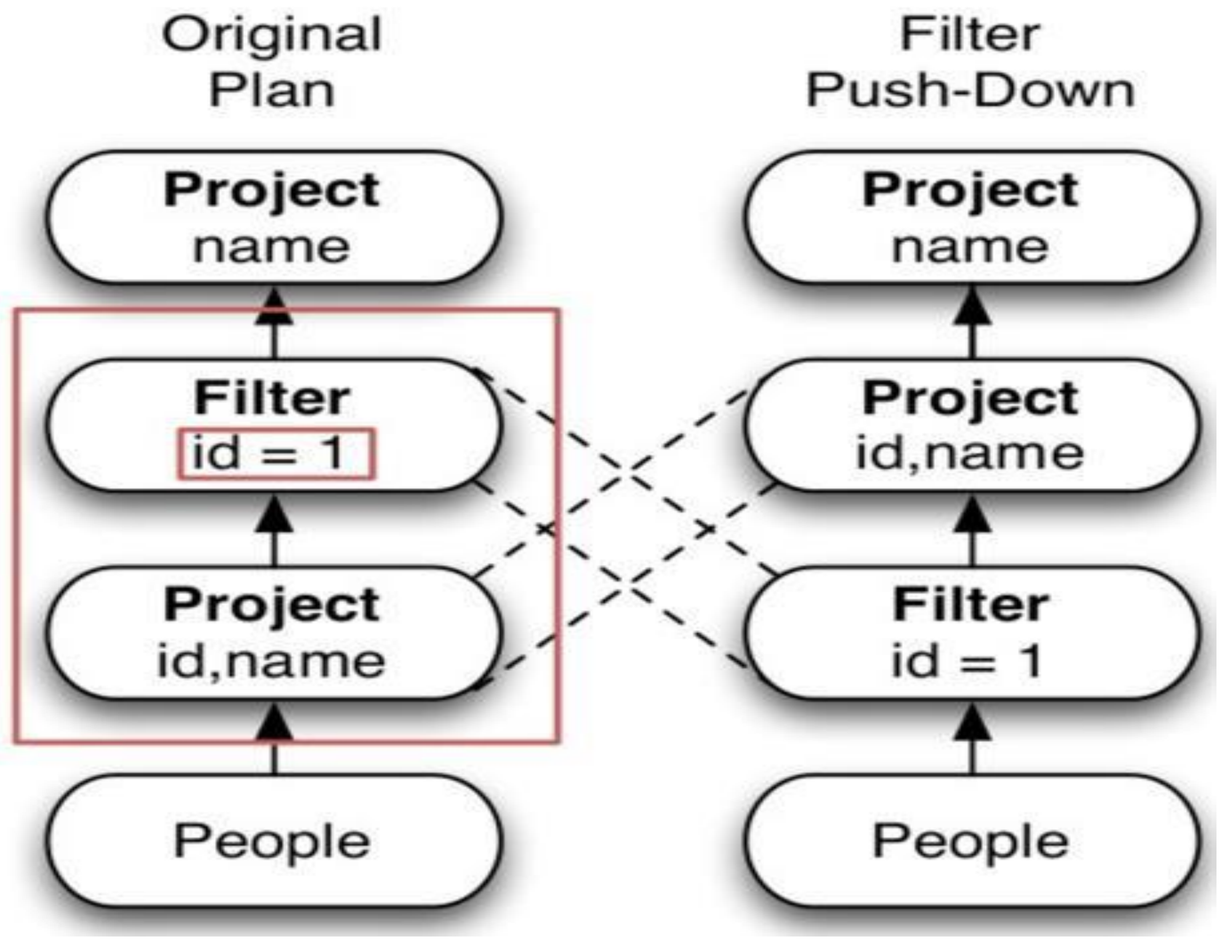
```
SELECT name
FROM (
    SELECT id, name
    FROM People) p
WHERE p.id = 1
```



CATALYST OPTIMIZER

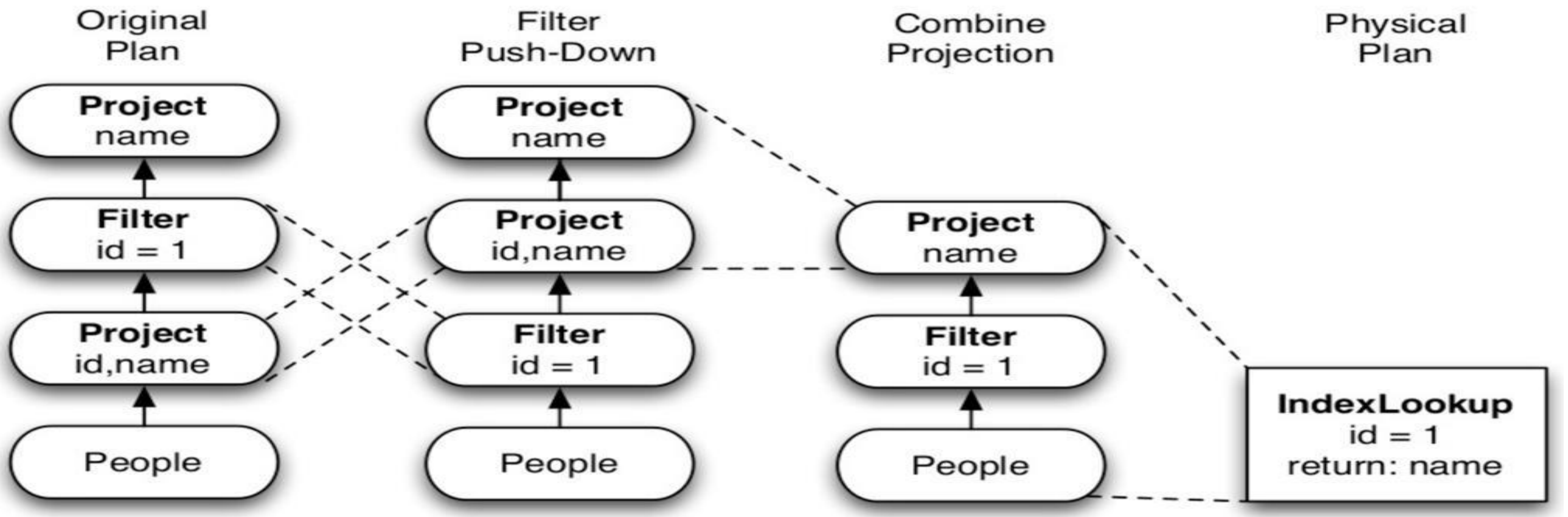
Optimization Rule Example:

1. Find filters on top of projections.
2. Check that the filter can be evaluated without the result of the project.
3. If so, switch the operators.

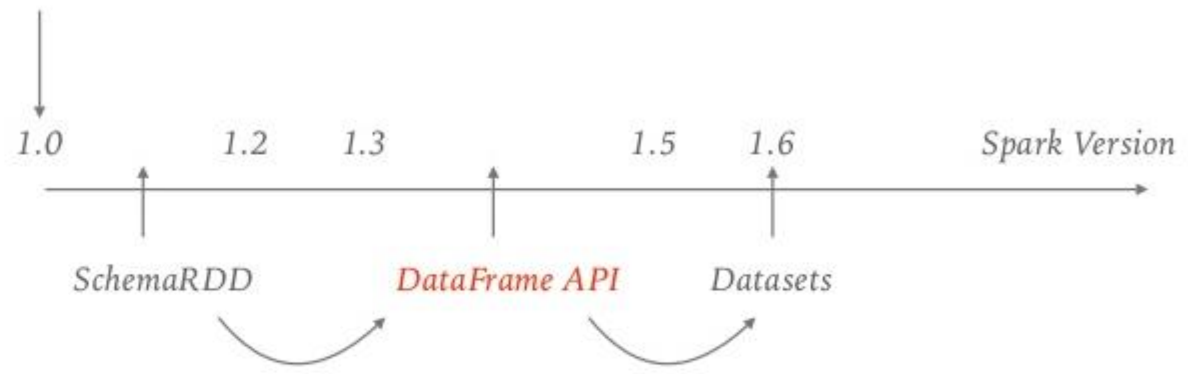


CATALYST OPTIMIZER

Optimization Rule Example:

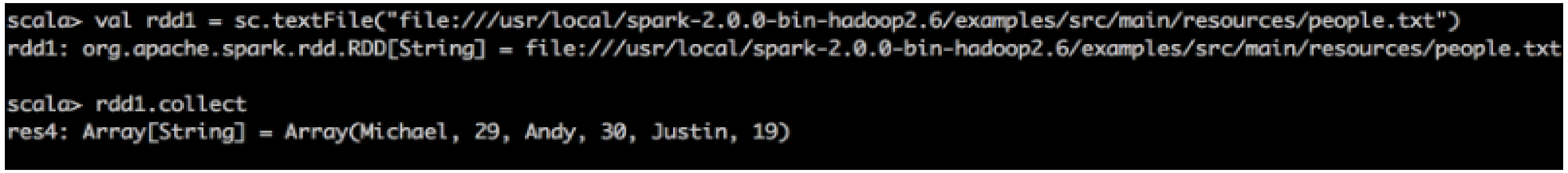


API History & Comparison



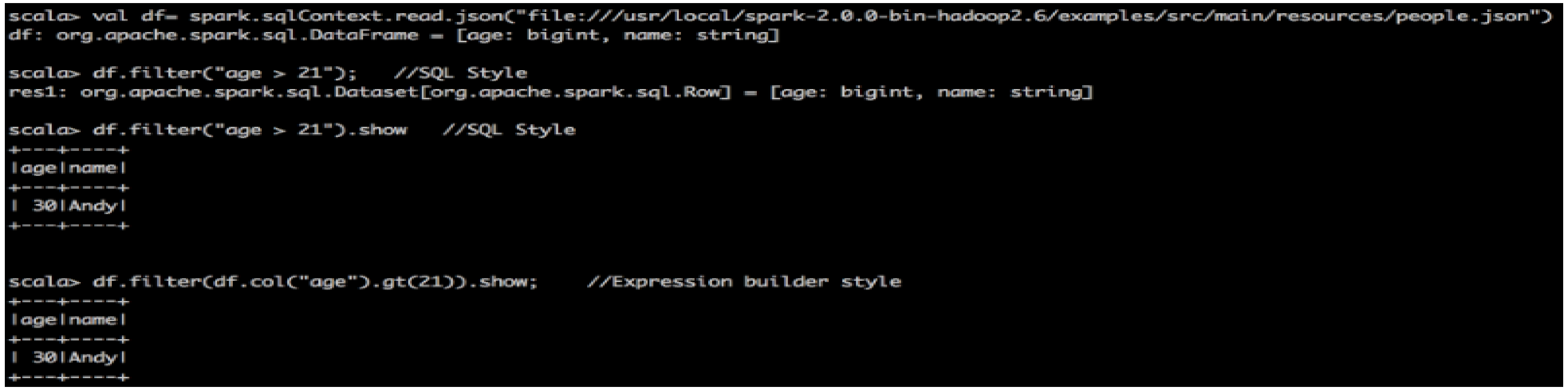
	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

RDDs vs DataFrames vs DataSets



```
scala> val rdd1 = sc.textFile("file:///usr/local/spark-2.0.0-bin-hadoop2.6/examples/src/main/resources/people.txt")
rdd1: org.apache.spark.rdd.RDD[String] = file:///usr/local/spark-2.0.0-bin-hadoop2.6/examples/src/main/resources/people.txt

scala> rdd1.collect
res4: Array[String] = Array(Michael, 29, Andy, 30, Justin, 19)
```

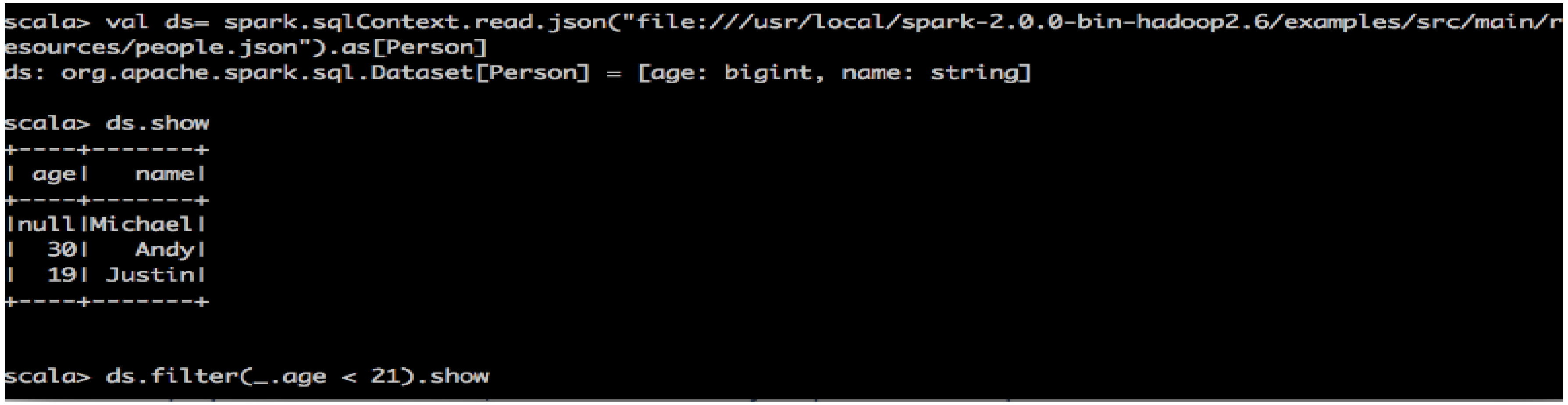


```
scala> val df= spark.sqlContext.read.json("file:///usr/local/spark-2.0.0-bin-hadoop2.6/examples/src/main/resources/people.json")
df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]

scala> df.filter("age > 21");   //SQL Style
res1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [age: bigint, name: string]

scala> df.filter("age > 21").show   //SQL Style
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+


scala> df.filter(df.col("age").gt(21)).show;    //Expression builder style
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+
```

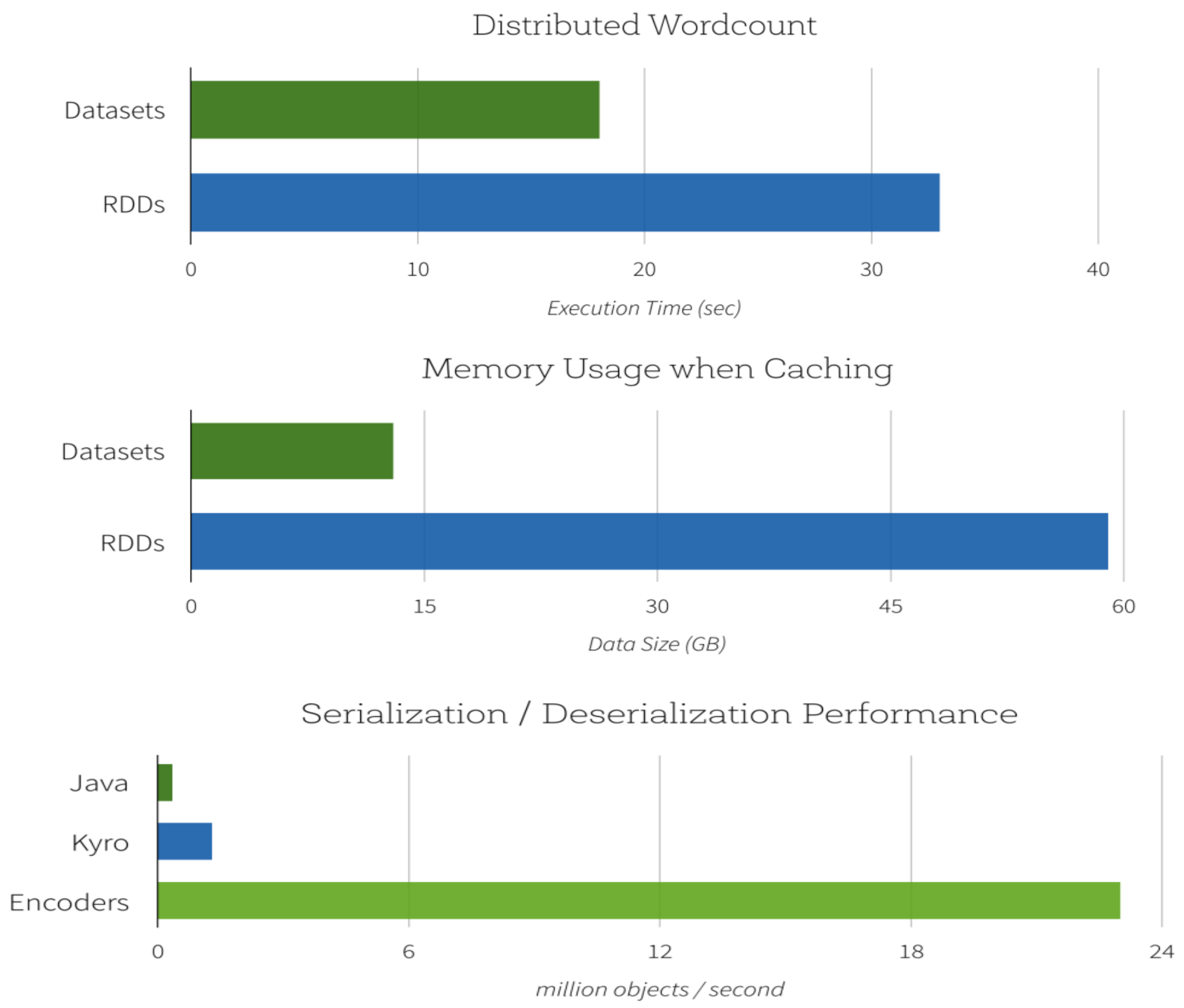


```
scala> val ds= spark.sqlContext.read.json("file:///usr/local/spark-2.0.0-bin-hadoop2.6/examples/src/main/r
esources/people.json").as[Person]
ds: org.apache.spark.sql.Dataset[Person] = [age: bigint, name: string]

scala> ds.show
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+


scala> ds.filter(_.age < 21).show
```

RDDs and DataSets Performance



RDDs

```
val lines = sc.textFile("/wikipedia")
val words = lines
.flatMap(_.split(" "))
.filter(_ != "")
```

Datasets

```
val lines = sqlContext.read.text("/wikipedia").as[String]
val words = lines
.flatMap(_.split(" "))
.filter(_ != "")
```

RDDs vs DataFrames vs DataSets

RDDs:

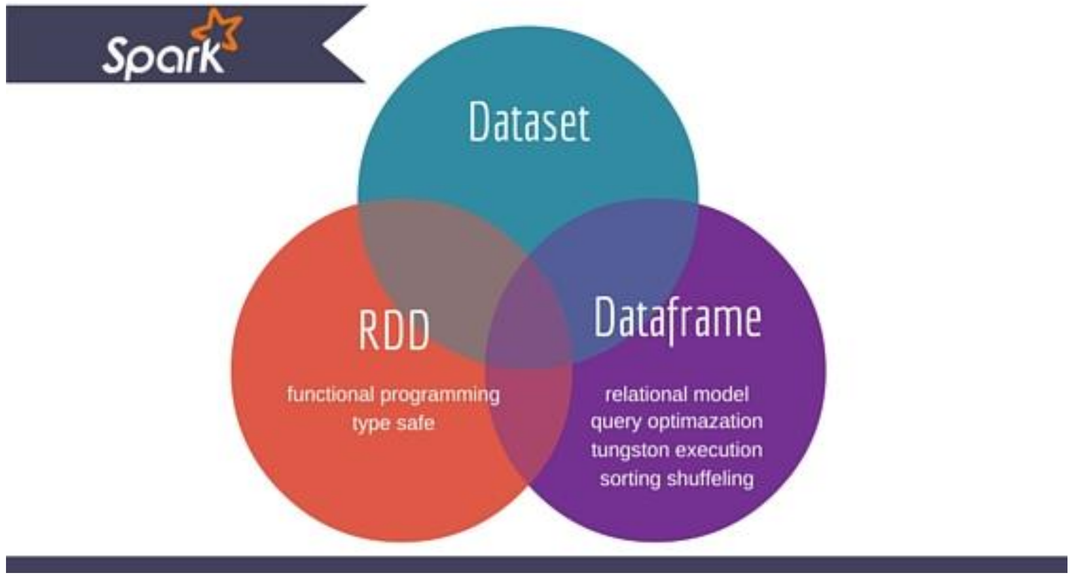
- Final Computation on RDDs
- Compile Time Safety
- Downside of RDD: performance limitations

DataFrame:

- Relational Model
- offers huge performance improvement
  - Custom Memory management (aka Project Tungsten)
  - Optimized Execution Plans (aka Catalyst Optimizer)
- Downside of Dataframe : Lack of Type Safety

DataSet :

- Latest Abstraction - extension of DataFrame API
- Developer friendly compile time safety
- It uses Encoders
- Convert DataSet to DataFrame at any point of time objects.DataFrame=Dataset[Row]



When to Use and What

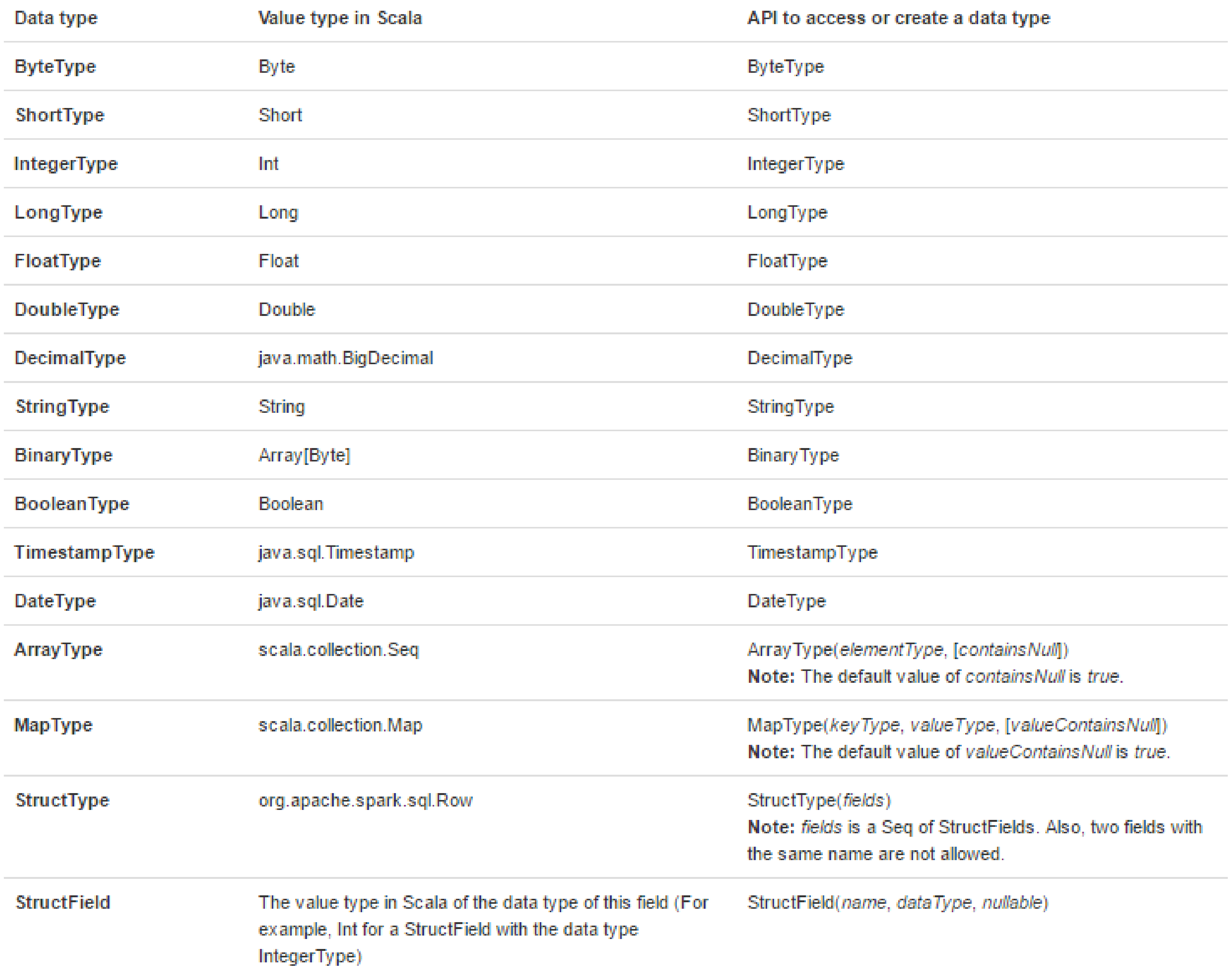
RDD:

- Data is unstructured, such as media streams or streams of text
- Manipulate data with functional programming constructs than domain specific expressions
- Low-level transformation and actions and control on dataset

DataFrame and DataSet:

- High-level abstractions, and domain specific APIs, use DataFrame or Dataset.
- High-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access and use of lambda functions on semi-structured data, use DataFrame or Dataset.
- Higher degree of type-safety at compile time, want typed JVM objects, take advantage of Catalyst optimization, and benefit from Tungsten's efficient code generation, use Dataset.
- Unification and simplification of APIs across Spark Libraries

Spark SQL Datatypes



Data type	Value type in Scala	API to access or create a data type
ByteType	Byte	ByteType
ShortType	Short	ShortType
IntegerType	Int	IntegerType
LongType	Long	LongType
FloatType	Float	FloatType
DoubleType	Double	DoubleType
DecimalType	java.math.BigDecimal	DecimalType
StringType	String	StringType
BinaryType	Array[Byte]	BinaryType
BooleanType	Boolean	BooleanType
TimestampType	java.sql.Timestamp	TimestampType
DateType	java.sql.Date	DateType
ArrayType	scala.collection.Seq	ArrayType(*elementType*, [*containsNull*]) **Note:** The default value of *containsNull* is *true*.
MapType	scala.collection.Map	MapType(*keyType*, *valueType*, [*valueContainsNull*]) **Note:** The default value of *valueContainsNull* is *true*.
StructType	org.apache.spark.sql.Row	StructType(*fields*) **Note:** *fields* is a Seq of StructFields. Also, two fields with the same name are not allowed.
StructField	The value type in Scala of the data type of this field (For example, Int for a StructField with the data type IntegerType)	StructField(*name*, *dataType*, *nullable*)

SPARK SQL Artifacts - MAVEN

```
<dependency>
   <groupId>org.apache.spark</groupId>
   <artifactId>spark-sql_2.11</artifactId>
   <version>2.0.1</version>
   <scope>provided</scope>
</dependency>

<dependency>
   <groupId>org.apache.spark</groupId>
   <artifactId>spark-hive_2.11</artifactId>
   <version>2.0.1</version>
   <scope>provided</scope>
</dependency>
```

SQL Performance – Caching

Caching Tables In-Memory

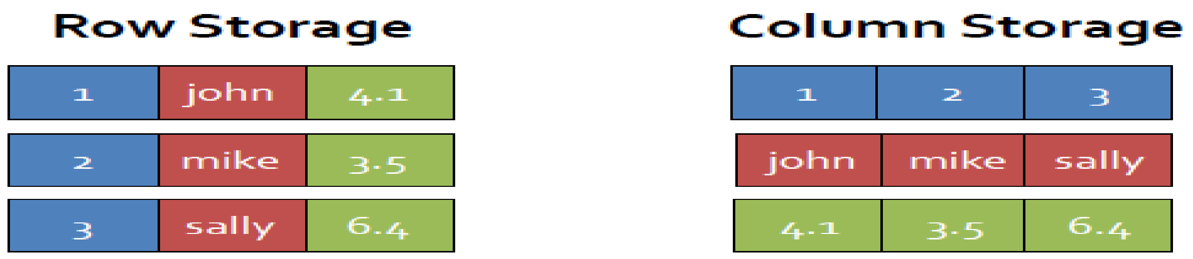
Spark SQL can cache tables using an in-memory columnar format:

- Scan only required columns
- Fewer allocated objects (less GC)
- Automatically selects best compression

```
cacheTable("people")
```

Simply caching Hive records as Java objects is inefficient due to high per-object overhead

Instead, Spark SQL employs column-oriented storage using **arrays of primitive types**



HiveContext

- HiveContext is an alternative entry point into the Spark SQL library
- HiveContext can do all that SQLContext can do and, on top of that, it can read from Hive meta store and tables, and can access Hive user-defined functions as well.
- The only requirement to use HiveContext is obviously that there should be an already existing Hive setup readily available. In this way, Spark SQL can easily co-exist with Hive.
- Spark SQL can process the data from Hive tables faster than Hive using its Hive Query Language.
- Spark SQL from different versions of Hive, which is a great feature enabling data source consolidation for data processing.

```
scala> import org.apache.spark.sql.hive._
scala> val hc = new HiveContext(sc)
```

Hive Features - Supported and Un Supported

- **Spark SQL supports the vast majority of Hive features, such as:**
  - **Hive query statements, including:**
  - SELECT, GROUP BY, ORDER BY, CLUSTER BY, SORT BY
  - **All Hive operators, including:**
  - Relational operators, Arithmetic operators, Logical operators, Mathematical functions , String functions
  - **User defined functions (UDF)**
  - **User defined aggregation functions (UDAF)**
  - **User defined serialization formats (SerDes)**
  - **Window functions**
  - **Joins**
  - JOIN, {LEFT|RIGHT|FULL} OUTER JOIN, LEFT SEMI JOIN, CROSS JOIN
  - **Unions, Sub-queries**
  - **Partitioned tables including dynamic partition insertion**
  - **All Hive DDL Functions, including:**
- **Unsupported Hive Functionality**
  - Tables with buckets: bucket is the hash partitioning within a Hive table partition. Spark SQL doesn't support buckets yet.
  - Column statistics collecting: Spark SQL does not scans to collect column statistics at the moment.
  - File format for CLI: For results showing back to the CLI, Spark SQL only supports TextOutputFormat.
- **Unsupported Hive Optimizations**
  - A handful of Hive optimizations are not yet included in Spark.
  - Some of these (such as indexes) are less important due to Spark SQL's in-memory computational model.
  - Does not automatically determine the number of reducers for joins and groupbys: Currently in Spark SQL - you need to control the degree of parallelism post-shuffle using "SET spark.sql.shuffle.partitions=[num_tasks];".
  - STREAMTABLE hint in join: Spark SQL does not follow the STREAMTABLE hint.
  - Merge multiple small files for query results: if the result output contains multiple small files, Hive can optionally merge the small files into fewer large files to avoid overflowing the HDFS metadata. Spark SQL does not support that.

SQL Workouts