Shark
SparkSQL
DataFrame

Shark

Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, Ion Stoica:
Shark: SQL and rich analytics at scale.
SIGMOD Conference 2013: 13-24

http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-214.pdf

Introduction

- A data analysis system that marries query processing with complex analytics (iterative ML) on large clusters.
- Fine-grained fault tolerance (
- Column-oriented in-memory storage and dynamic mid-query replanning.
- “Run SQL queries up to 100x faster than Apache Hive”.
- Shark = “Spark on Hive”

Hive

- A standard for SQL queries over petabytes of data in Hadoop.
- Provides SQL-like (HQL) access for data in HDFS.
- Data model provides a high-level, table-like structure ontop of HDFS.
- Data structures: tables, partitions, and buckets, where tables correspond to HDFS directories and can bedivided into partitions, which in turn can be divided into buckets.

Hive (2)

- The Hive metadata store(aka metastore) can use either embedded, local, or remote databases.
- Hive servers are built on Apache Thrift Server technology.
- Hive supports adhoc querying data on HDFS
- Hive has a well-defined architecture for metadata management, authentication, and queryoptimizations.

Motivation

- MapReduce for SQL queries (Google Tenzing, Hadapt) have a minimum latency of 10 seconds.
- Systems like Google Dremel, Impala use a coarse-grained recovery model (in case of failure, the whole query is restarted) => works well some small queries only.
- These systems do not interact efficiently with iterative ML-like operations.

Design (1)

- Shark builds on RDD, so:
  - In-memory
  - Fault-tolerance through lineage.
- With some extensions:
  - in-memory columnar storage and columnar compression
  - SQL query optimization
  - Support for all HQL dialect and UDFs

Design (2)

- With some extensions:
  - augments SQL with complex analytics functions written in Spark, using Spark's Java, Scala or Python APIs. These functions can be combined with
  - SQL in a single execution plan, providing in-memory data sharing
  - and fast recovery across both types of processing.

Architecture

- Consists of a single master node and a number of slave nodes, with the warehouse metadata stored in an external transactional database.
- Master receives queries, Shark compiles them into operator tree represented as RDDs. These RDDs are then translated by Spark into a graph of tasks to execute on the slave nodes.

Execution model

- 3 standard steps:
  - Query parsing
    - Using Hive query compiler to parse a query and generate an Abstract Syntax Tree (AST)
  - Logical plan generation
    - ASTs are turned into logical plans
  - Physical plan generation
    - Logical plans become physical plans consisting of transformations on RDDs.

Deprecated

- Since the emergence of SparkSQL, Shark is deprecated.
- SparkSQL becomes the standard for querying SQL queries on top of Spark

Spark SQL

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia:

Spark SQL: Relational Data Processing in Spark.
SIGMOD Conference 2015: 1383-1394

https://web.eecs.umich.edu/~prabal/teaching/resources/eecs582/armbrust15sparksql.pdf

Introduction

- Builds on the experience of designing Shark.
- With a tighter integration between relational and procedural processing, through a declarative **DataFrame** API that integrates with procedural Spark code.
- Includes a highly extensible optimizer, **Catalyst**

Motivation

- Problems with Shark:
  - could only be used to query external data stored in the Hive catalog.
  - Only way to call Shark from Spark was to write a SQL string.
  - The Hive optimizer was to MapReduce specific and difficult to extend.

Goals for Spark SQL

- 4 goals for Spark SQL:

1) Support relational processing both within Spark programs and external data sources.
2) Provide high performance using established DBMS techniques.
3) Easily support new data sources (semi-structured, external DB to query federation).
4) Enable extension with advanced analytics algorithms such as graph processing and ML.

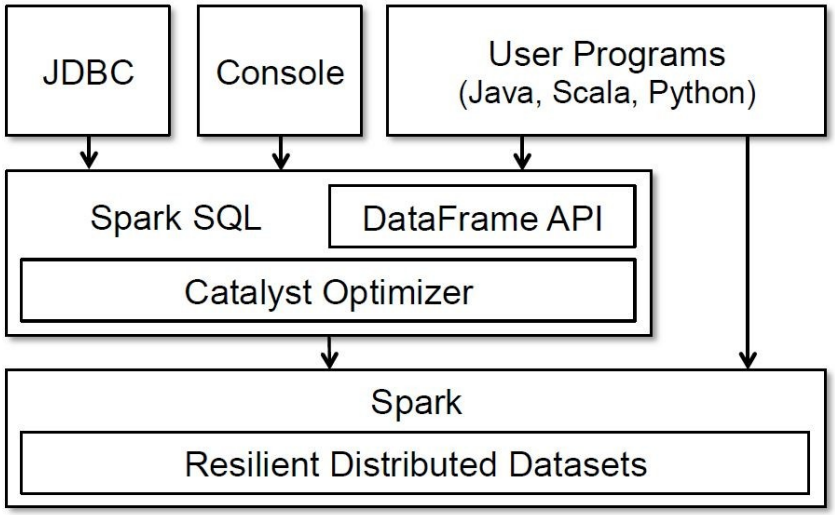


Figure 1: Interfaces to Spark SQL, and interaction with Spark.

Data model

- Spark SQL uses a nested data model based on Hive for tables and DataFrames.
- Supports all major SQL data types and some complex ones: structs, arrays, maps and unions.
- Complex data types can be nested together.
- Supports UDF.

Spark SQL - example

```
import sqlContext._
import org.apache.spark.sql.types._
import
org.apache.spark.sql.catalyst.expressions._

import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

import sqlContext.implicits._
import org.apache.spark.sql._
```

Spark SQL - example

```
val sqlContext = new
org.apache.spark.sql.SQLContext(sc)
val schemaTriple =  StructType(
  StructField("s", LongType, false)::
  StructField("p", LongType, false)::
  StructField("o", LongType, false)::
  Nil)

val triples =
sc.textFile("/user/olivier/dbpedia/infobox_proper
ties").map(x=>x.split(" ")).map(t=>(t(0).toLong,
(t(1).toLong,t(2).toLong)))
```

Spark SQL - example

```
val rowTriples = triples.map{case(s,
(p,o))=>Row(s,p,o)}

val tripleSchemaRDD =
sqlContext.applySchema(rowTriples, schemaTriple)
tripleSchemaRDD.registerTempTable("triple")

val res = sqlContext.sql("SELECT t1.s, t3.o FROM
triple t1, triple t2, triple t3 WHERE
t1.p=1446244352 AND t2.p =1446133760 AND t3.p
=5755 AND t1.o=t2.s AND t2.o=t3.s")

res.count
```

DataFrame

- Similar to the data frame concept in R.
- But evaluates operations lazily to perform some optimizations.
- Collections of structured records that can be manipulated with Spark's procedural API or relational APIs for richer optimizations.
- stores data in a columnar format that is significantly more compact than Java/Python objects.

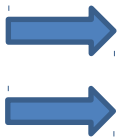
DataFrame

- Each DataFrame can be viewed as a RDD of Row objects.
- Relational operations can be performed using a DSL similar to R data frames and Python Pandas:
  - select, where, join, groupBy.

DataFrame API

- DataFrame is the main abstraction in Spark SQL's API.
- DataFrame keep track of their schema that leads more optimized execution.
- Each DataFrame can be viewed as an RDD of Row objects.
- DataFrame is lazy.
- Each DataFrame represents a logical plan.

```
ctx = new HiveContext()
users = ctx.table("users")
young = users.where(users("age") < 21)
println(young.count())
```



Physical plan

DataFrame

```
val df = triples.map{case(s,
(p,o))=>(s,p,o)}.toDF("s","p","o")

val r1 =
df.where(df("p")===1446244352).select("o").withCo
lumnRenamed("o","x")
val r2 =
df.where(df("p")===1446133760).select("s","o").wi
thColumnRenamed("s", "x").withColumnRenamed("o",
"y")
val r3 =
df.where(df("p")===5755).select("s","o").withColu
mnRenamed("o", "z").withColumnRenamed("s", "y")

val res =  r1.join(r2, Seq("x")).join(r3,
Seq("y"))

res.count
```

DataFrame Operation

- DataFrame supports:

1) All common relational operators: select, where, join, groupBy, etc.

2) Comparison and arithmetic operators: ===, >, <, +, -, etc.

- All above-mentioned operators build up an **Abstract Syntax Tree (AST)** of the expression, then passed to Catalyst.
- The DataFrame registered in the catalog are still unmaterialized views, so the optimization can happen across SAL and the original DataFrame expression.

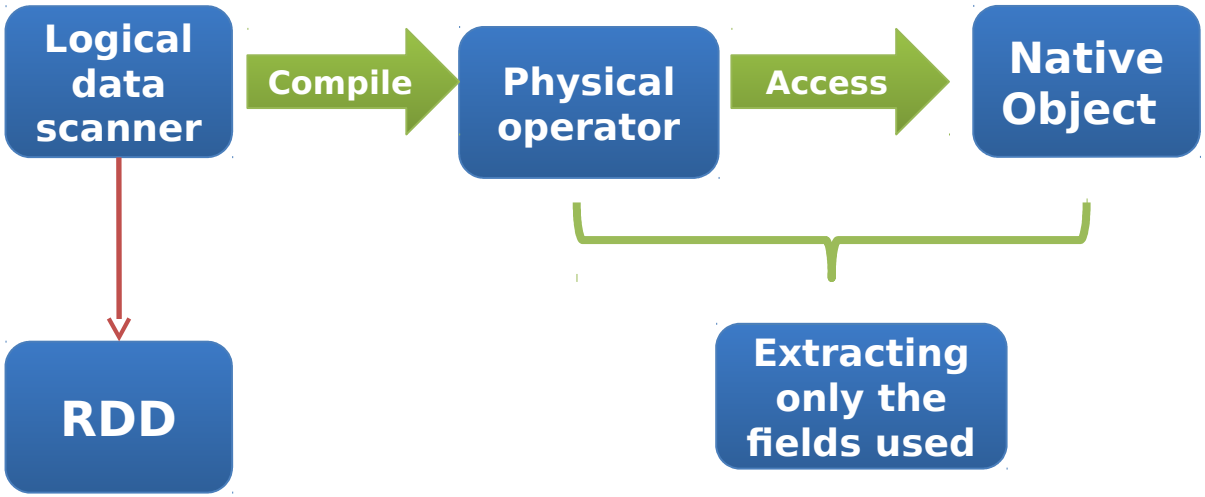
DataFrame VS Relational Query Languages

- An API analyze the correctness of logical plan eagerly:
1)Column name used in expression.
2)Data types.

Spark reports the errors as soon as the users type invalid code instead of waiting until execution.

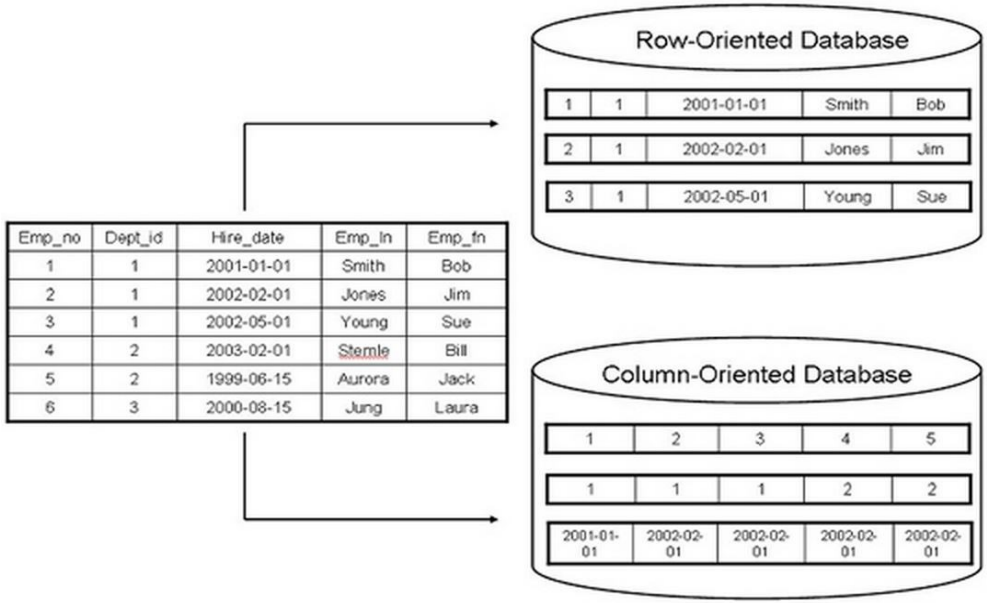
Querying Native Datasets

- Spark SQL allows users to construct DataFrame directly against RDDs of objects native to the programming language, e.g. : JavaBean, Scala case.



In-Memory Caching

- Cache hot data by:
Columnar cache cache(), reduce memory footprint by applying compression schema such as: dictionary encoding and run-length encoding.



Columnar Storage

- Advantages of columnar storage:

1)Skip unsatisfied data, reduce the IO operations.

2)Efficiently apply compression encoding.

3)Read the required column, support vector computing, better scanning performance.

Catalyst Optimizer

- **Two goals of Catalyst:**

1)Make it easy to add new optimization technique and features to Spark SQL.

2)Enable external developers to extend the optimizer (e.g. adding data source specific rules, support new data types).

- At the core of Catalyst, it contains a general library for representing **trees** and applying **rule** to manipulate them. For example:

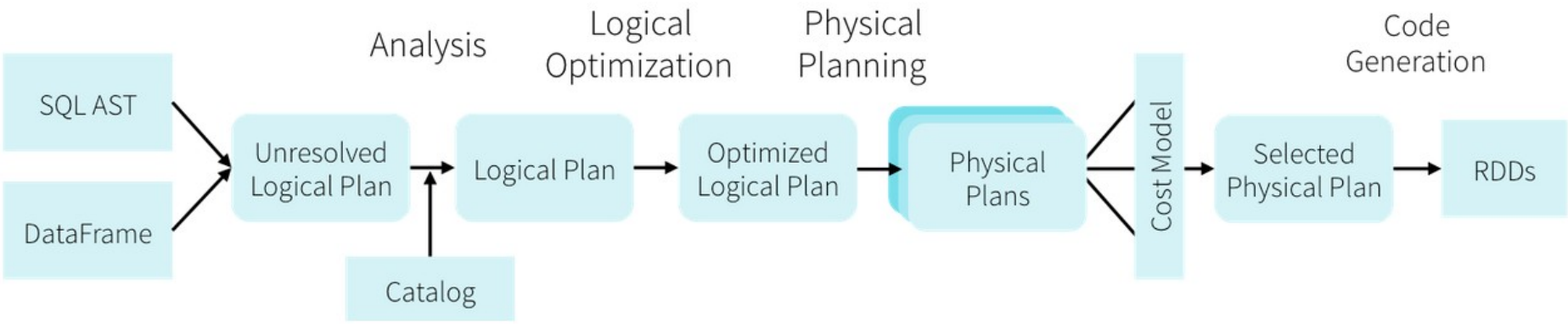
Cost-based optimization is performed by generating multiple plans using rules, and then computing their costs.

Catalyst

- Optimizer for SQL and DataFrames.
- Makes it easy to add data sources, (e.g., JSON), optimization rules and data types.

  Uses features of Scala (pattern-matching).

Catalyst


Analysis
Logical Optimization
Physical Planning
Code Generation
SQL AST
DataFrame
Unresolved Logical Plan
Catalog
Logical Plan
Optimized Logical Plan
Physical Plans
Cost Model
Selected Physical Plan
RDDs

Catalyst Optimizer

Tree:

- The **main data type in Catalyst is tree**, composed of **node** and **objects**.
- New node types are defined in Scala as subclasses of the **TreeNode**.

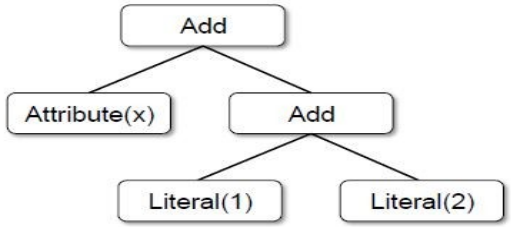


Figure 2: Catalyst tree for the expression `x+(1+2)`.

Catalyst Optimizer

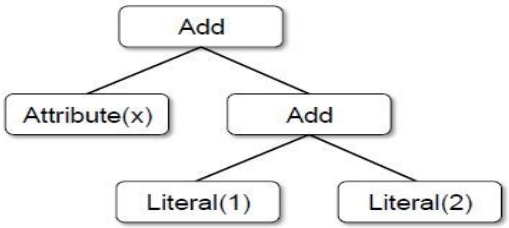


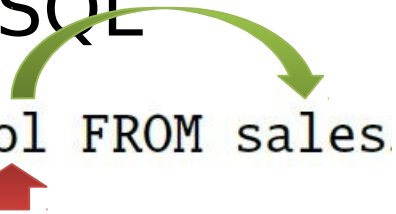
Figure 2: Catalyst tree for the expression x+(1+2).

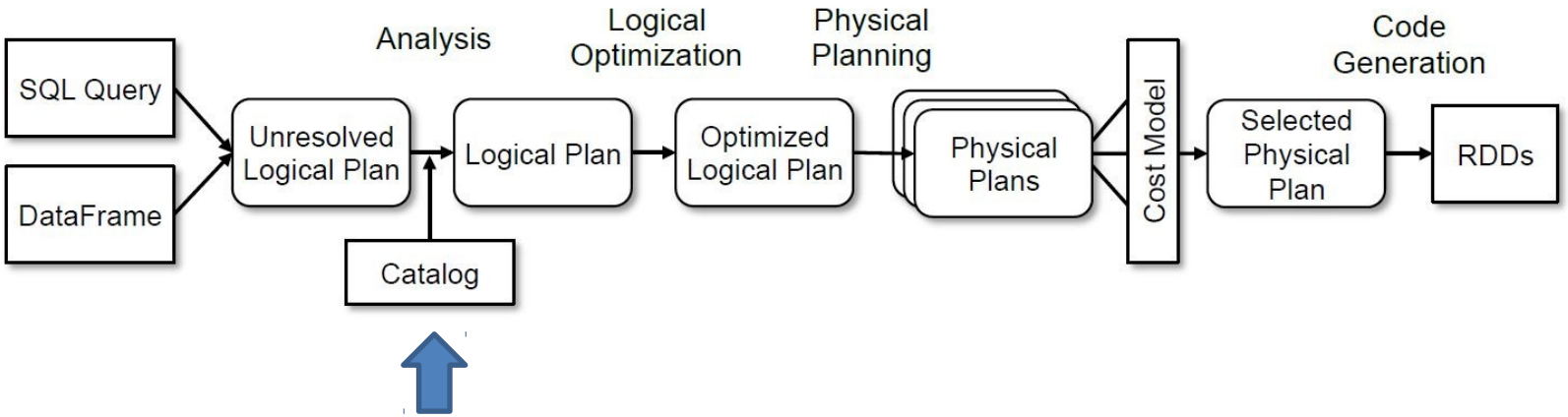
Rules:

- Rules: Tree_1 => Tree_2 = Rules(Tree_2)
- The most common approach is using **pattern matching** (**transform method** in Catalyst).
- Transform is a partial function, i.e. only needs to match a part of input tree. => The rest optimization only needs to be applied on the matched part.
- **In practice, rules may need to execute multiple times to fully transform an input treee. Until tree stops changing after applying its rules.**

Catalyst Optimizer – Using Catalyst in Spark SQL

- Unresolved logical pl

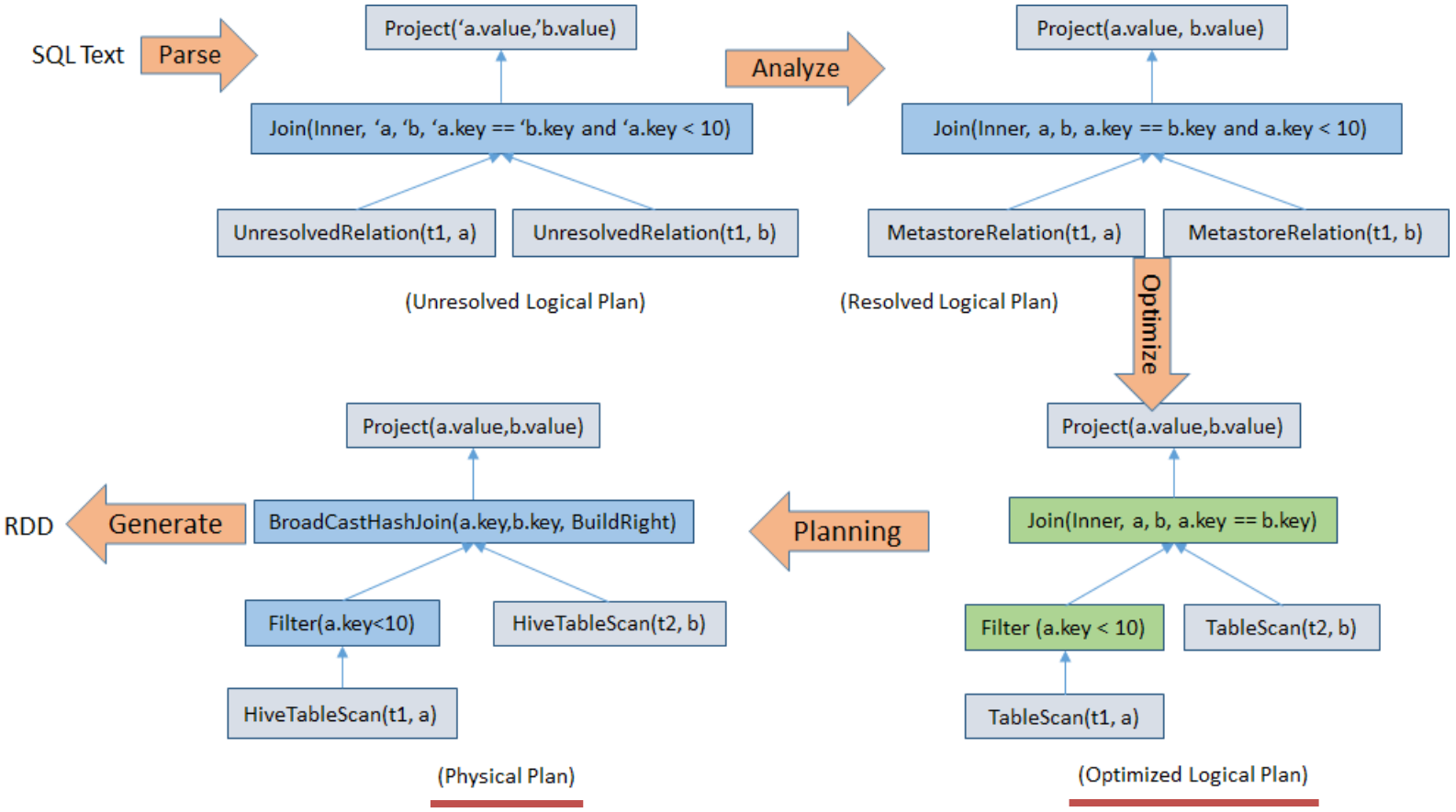




Spark SQL uses Catalyst rules and Catalog objects to track the tables in all data sources to resolve attributes.

Catalyst Optimizer – Using Catalyst in Spark SQL

- Example:



Catalyst Optimizer – Logical Optimization

- Applies standard **rule-based optimization** to the logical plan, such as constant folding, predicate pushdown, projection pruning, null propagation, Boolean expression simplification, etc.

Example: decimal control

```
object DecimalAggregates extends Rule[LogicalPlan] {
  /** Maximum number of decimal digits in a Long */
  val MAX_LONG_DIGITS = 18

  def apply(plan: LogicalPlan): LogicalPlan = {
    plan transformAllExpressions {
      case Sum(e @ DecimalType.Expression(prec, scale))
              if prec + 10 <= MAX_LONG_DIGITS =>
        MakeDecimal(Sum(UnscaledValue(e)), prec + 10, scale)
    }
}
```

Catalyst Optimizer – Physical Planning

- For a logical plan, Spark SQL generates one or more physical plans, selects a plan using a cost model: **select join algo.**

Catalyst Optimizer – Code Generation

- Final phase of query optimization, generates Java byte-code to run on each machine: quasiquotes on scala.

- **For precedent example (x+y)+1**

Without code generation: interpreted for each row of data, by walking down a tree of Add, Attribute and Literal nodes.
-> Slow down execution.

Catalyst Optimizer – Code Generation

With code generation: write a function to translate specific expression tree to scala

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case Add(left, right) =>
    q"${compile(left)} + ${compile(right)}"
}
```

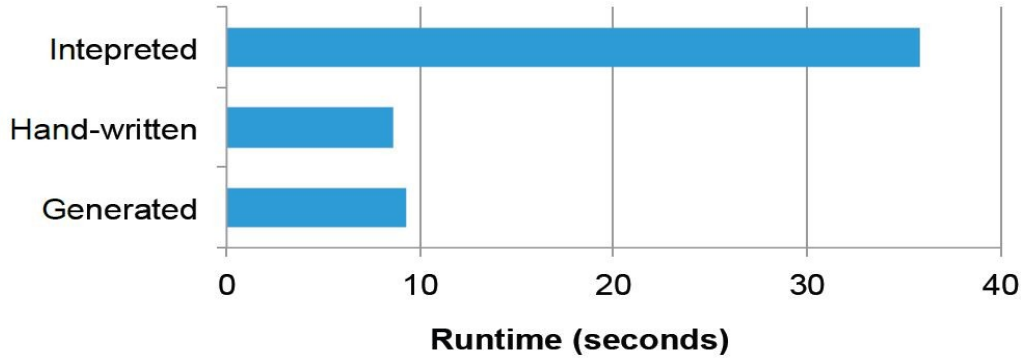


Figure 4: A comparision of the performance evaluating the expresion `x+x+x`, where `x` is an integer, 1 billion times.

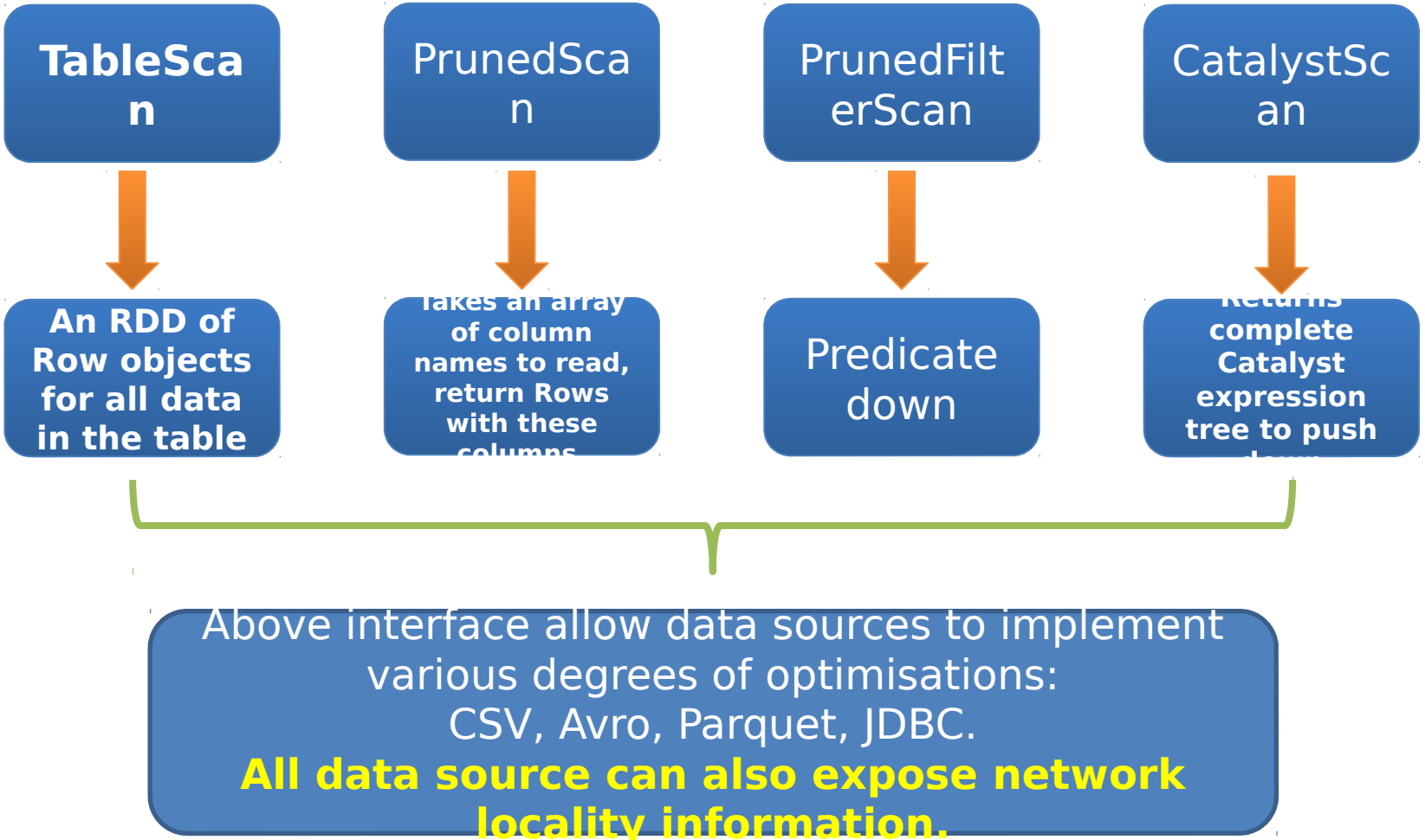
Catalyst Optimizer – Extension Point

- To simpler add some types of extension without understanding Catalyst rules, Spark SQL defines two public extension points:

1) Data Sources.
2) UDTs.

Catalyst Optimizer – Data Sources

- **Data sources**: implements a **createRelation** function (K,V) returns a **BaseRelation** object for the relation.



Catalyst Optimizer – Data Srouces

- **UDTs: User-defined types**
- The built-in data types are stored in a columnar, compressed format for in-memory caching.
- Use built-in types **<->** Catalyst built-in types (Java Beans, Scala case):

```
class PointUDT extends UserDefinedType[Point] {
  def dataType = StructType(Seq(   // Our native structure
    StructField("x", DoubleType),
    StructField("y", DoubleType)
  ))
  def serialize(p: Point) = Row(p.x, p.y)
  def deserialize(r: Row) =
    Point(r.getDouble(0), r.getDouble(1))
}
```

Advanced Analytics Features

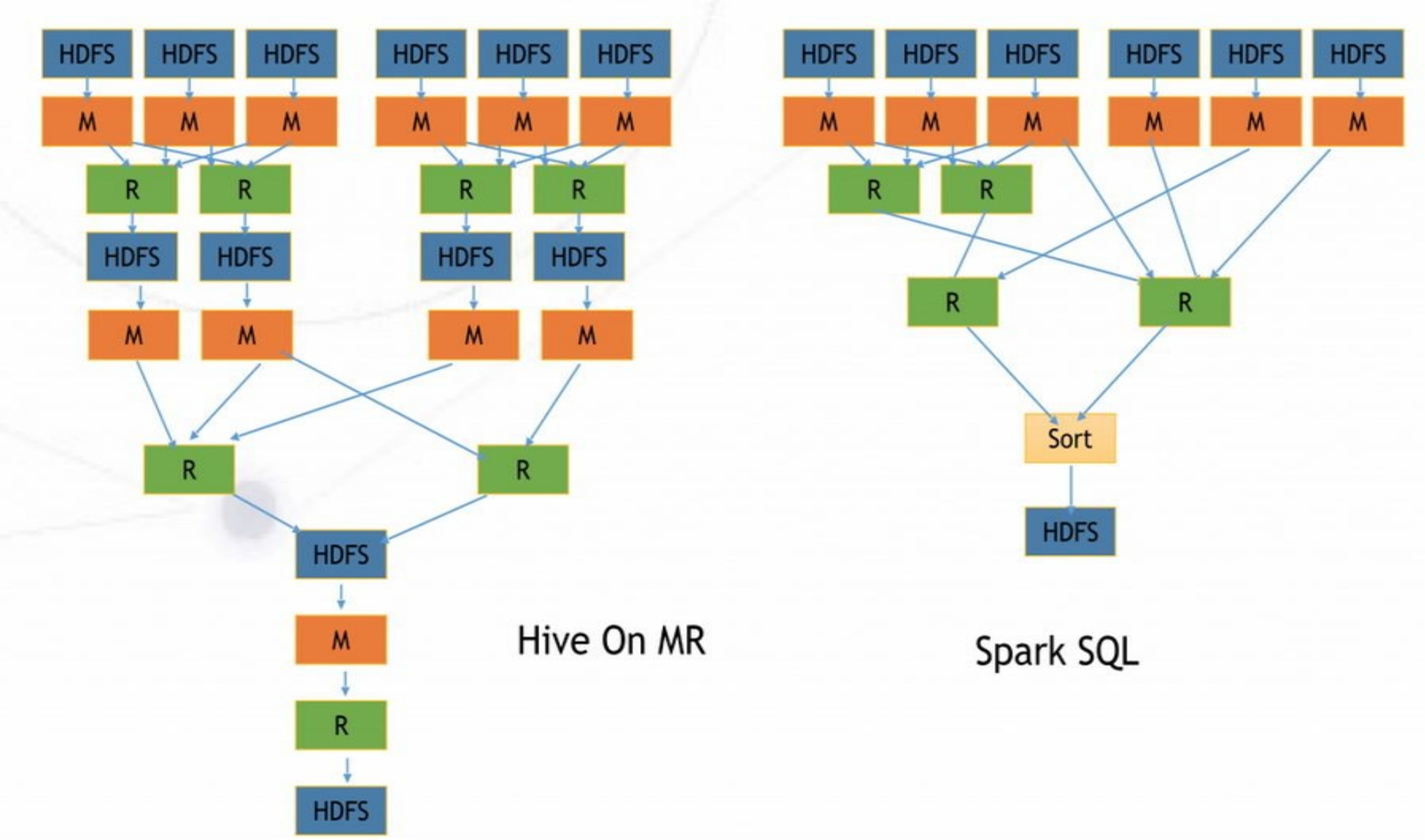
- Schema inference for SemiStructured Data.

- Integration with Spark's Mlib.
- Query federation to external database:

Example:

```
CREATE TEMPORARY TABLE users USING jdbc
OPTIONS(driver "mysql" url "jdbc:mysql://userDB/users")

CREATE TEMPORARY TABLE logs
USING json OPTIONS (path "logs.json")

SELECT users.id, users.name, logs.message
FROM users JOIN logs WHERE users.id = logs.userId
AND users.registrationDate > "2015-01-01"
```

Evaluation

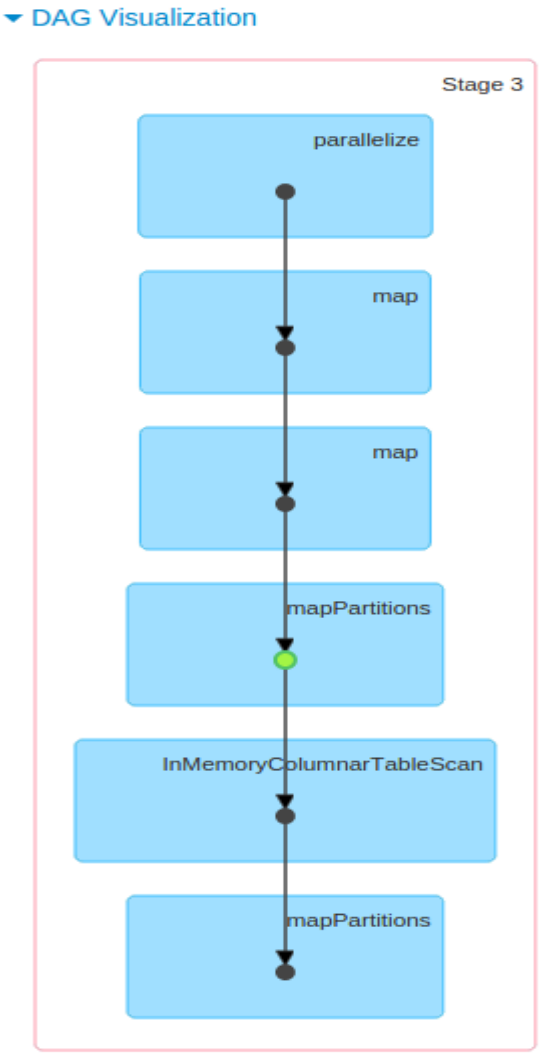


Visualization / Explain

- We can visualize the executed query plan on the HTTP Spark interface.

- We can also visualize using the method 'explain'.

```
scala> res.explain
== Physical Plan ==
InMemoryColumnarTableScan
[s#6L,p#7L,o#8L], (InMemoryRelation
[s#6L,p#7L,o#8L], true, 10000,
StorageLevel(true, true, false,
true, 1), (Scan
PhysicalRDD[s#6L,p#7L,o#8L]), None)
```



Datasets

- The Spark community appreciates:
  - In DataFrame: the compression aspect and ability to associate a schema.
  - Type-safety, object-oriented programming interface nature of RDDs
- Databricks proposes an extension of the DataFrame API that provides a type-safe, object-oriented programming interface which is named **Datasets**.
- A Dataset is a strongly-typed, immutable collection of objects that are mapped to a relational schema.
- At the core of the Dataset API is a new concept called an encoder, which is responsible for converting between JVM objects and tabular representation.
- The tabular representation is stored using Spark's internal Tungsten binary format, allowing for operations on serialized data and improved memory utilization. Spark 1.6 comes with support for automatically generating encoders for a wide variety of types, including primitive types (e.g. String, Integer, Long), Scala case classes, and Java Beans.
- Users of RDDs will find the Dataset API quite familiar, as it provides many of the same functional transformations (e.g. map, flatMap, filter).

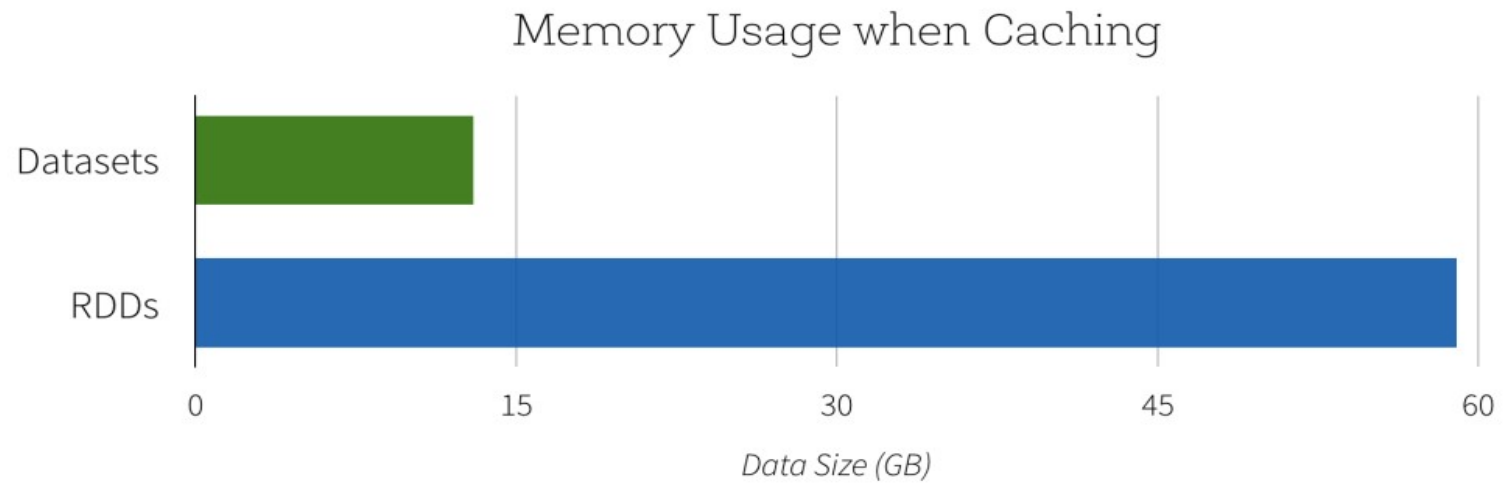
RDD

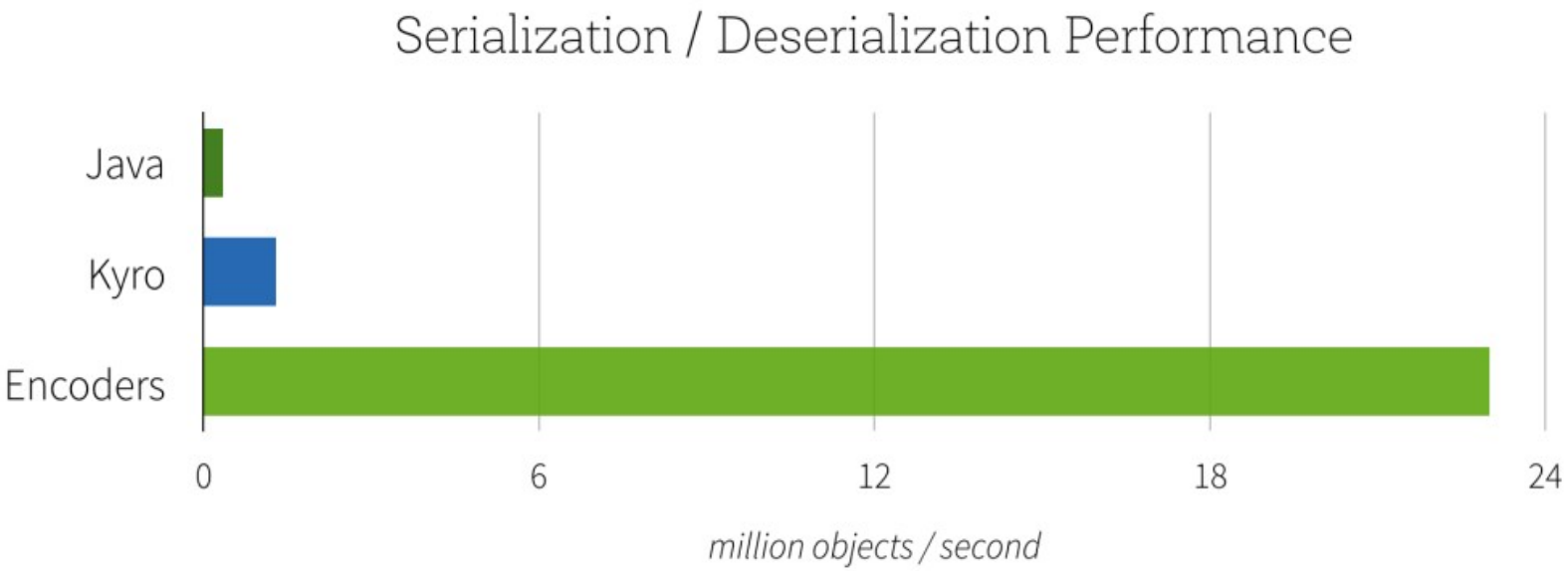
```
val lines = sc.textFile("/wikipedia")

val words = lines  .flatMap(_.split(" ")).filter(_ != "")
```

Datasets

```
val lines = sqlContext.read.text("/wikipedia").as[String]

val words = lines.flatMap(_.split(" ")).filter(_ != "")
```

Datasets


Memory Usage when Caching
Datasets
RDDs
0
15
30
45
60
Data Size (GB)


Serialization / Deserialization Performance
Java
Kyro
Encoders
0
6
12
18
24
million objects / second

Datasets

- In Spark 2.0, improvements to Datasets, specifically:

- Performance optimizations – In many cases, the current implementation of the Dataset API does not yet leverage the additional information it has and can be slower than RDDs. Over the next several releases, we will be working on improving the performance of this new API.
- Custom encoders – while we currently autogenerate encoders for a wide variety of types, we'd like to open up an API for custom objects.
- Python Support.
- Unification of DataFrames with Datasets – due to compatibility guarantees, DataFrames and Datasets currently cannot share a common parent class. With Spark 2.0, we will be able to unify these abstractions with minor changes to the API, making it easy to build libraries that work with both.