Dataset in Spark

Wenchen Fan
@cloud-fan
Spark Summit China 2016

About Me: 范文臣

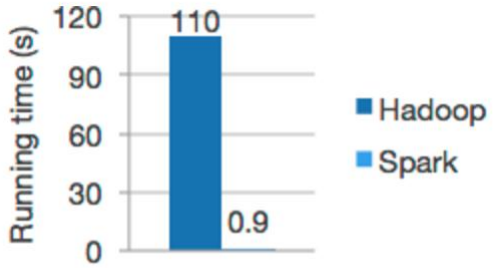
- Joined Spark Community at 2014
- Joined Databricks at 2015 (work remotely at Hangzhou)
- Became Spark Committer at 2016
- Interested in query engine and distributed system

Github: @cloud-fan Weibo: @cloud__fan

Why you love Spark?

Why you love Spark?

- **Efficient**: general execution graphs, in memory storage



Logistic regression in Hadoop and Spark

Why you love Spark?

- **Ease of Use**: collection(RDD) based API

```
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

Word count in Spark's Python API

Background: What is in an RDD?

Dependencies
Partitions (with optional locality info)
Compute function: Partition => Iterator[T]

Background: What is in an RDD?

Dependencies

Partitions (with optional locality info)

Compute function: Partition => Iterator[T]

Opaque Computation

Background: What is in an RDD?

Dependencies

Partitions (with optional locality info)

Compute function: Partition => Iterator[T]

Opaque Data

RDD API is not expressive enough

```
pdata.map(lambda x: (x.dept, [x.age, 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()
```

```
SELECT dept, AVG(age) FROM pdata GROUP BY dept
```

Structure

By definition, structure will *limit* what can be expressed.

In practice, we can accommodate the vast majority of computations.

Limiting the space of what can be expressed enables optimizations.

Spark SQL

Process structured data faster and easier

Spark SQL user interface

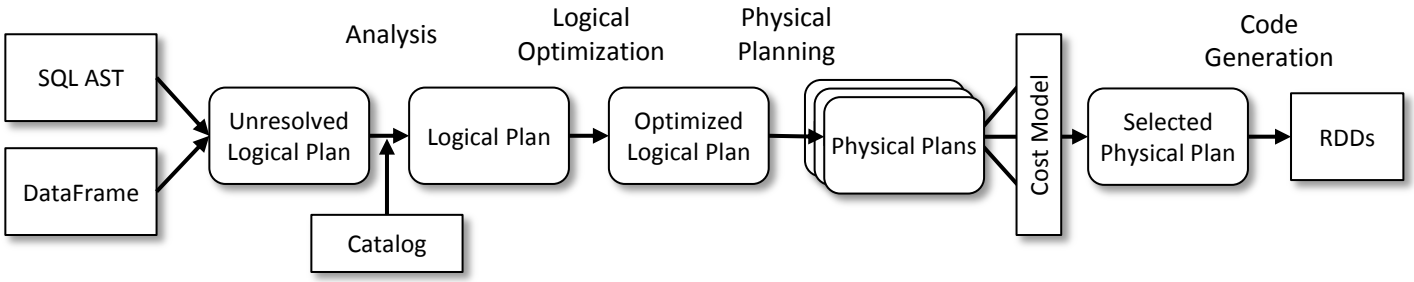
- SQL:

```
SELECT dept, AVG(age) FROM pdata GROUP BY dept
```

- DataFrame (added in 1.3): a DSL for structural operations

```
pdata.groupBy("dept").avg("age")
```

Plan Optimization & Execution



DataFrames and SQL share the same optimization/execution p

Are we good now?

Are we good now?

	SQL	DataFrames
Syntax Errors	Runtime	Compile Time
Analysis Errors	Runtime	Runtime

Are we good now?

	SQL	DataFrames
Syntax Errors	`SELETC name FROM person` `// compile pass ✗`	Compile Time
Analysis Errors	Runtime	Runtime

Are we good now?

	SQL	DataFrames
Syntax Errors	Runtime	`df.seletc("name")` `// compile fails` ✅
Analysis Errors	Runtime	Runtime

Are we good now?

	SQL	DataFrames
Syntax Errors	Runtime	Compile Time
Analysis Errors	Runtime	`df.select("naname")` `// compile pass ✗`

Now we are good!

	SQL	DataFrames	Dataset(since 1.6)
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

Now we are good!

	SQL	DataFrames	Dataset(since 1.6)
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	`ds.map(_.naname)` `// compile fails` ✅

Dataset

A type-safe version of DataFrame

Dataset examples

```scala
val df = spark.read.json("person.json") // {email, age, name}

case class Person(email: String, age: Int, name: String)

val ds: Dataset[Person] = df.as[Person]

val names: Array[String] = ds.map(_.name).collect()
// df.select("name").collect()

val adults: Dataset[Person] = ds.filter(_.age > 18)
// df.filter($"age" > 18)

val agged: Dataset[(String, Int)] =
  ds.groupByKey(_.name.take(3)).reduceGroups(_ + _)
```

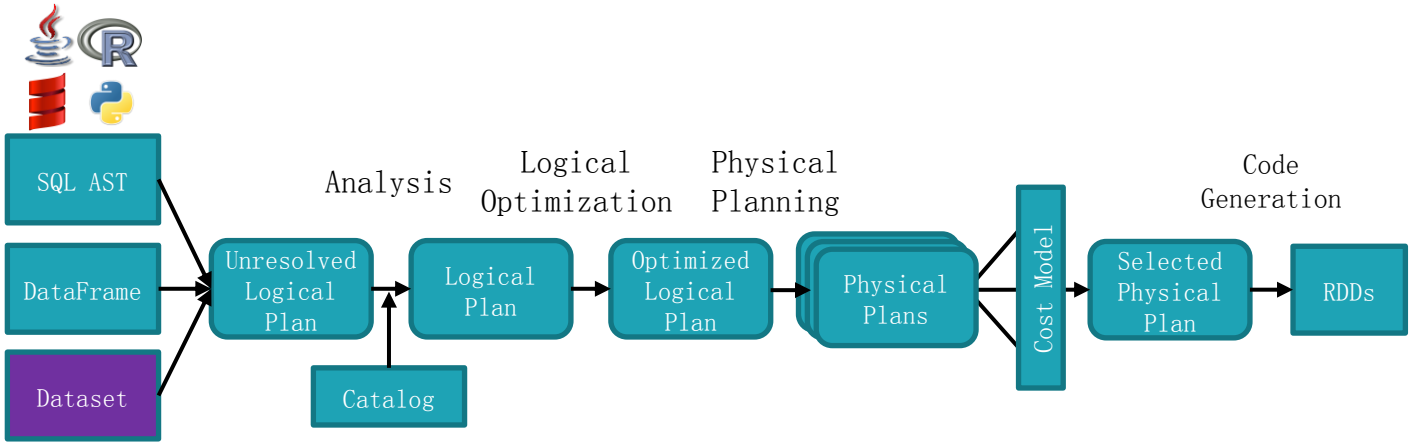
Dataset Aggregator

```
abstract class Aggregator[-IN, BUF, OUT] extends Serializable {

  def zero: BUF

  def reduce(b: BUF, a: IN): BUF

  def merge(b1: BUF, b2: BUF): BUF

  def finish(reduction: BUF): OUT
}
```

Dataset Aggregator

```
object TypedSumLong extends Aggregator[Long, Long, Long] {

  override def zero: Long = 0L

  override def reduce(b: Long, a: Long): Long = b + a

  override def merge(b1: Long, b2: Long): Long = b1 + b2

  override def finish(reduction: Long): Long = reduction
}

val result: Dataset[(Long, Long)] = (ds: Dataset[Long])
  .groupByKey(_ % 7)
  .agg(TypedSumLong.toColumn)
```

Shared Optimization & Execution



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

DataFrame and Dataset are unified at 2.0

- Dataset[Row] = DataFrame
- Stringly-typed methods will downcast to generic Row

```
val result: Dataset[Row] = (ds: Dataset[Person]).select("name")
```

- Ask Spark SQL to enforce types on generic rows using as

```
val result: Dataset[Person] = (df: Dataset[Row]).as[Person]
```

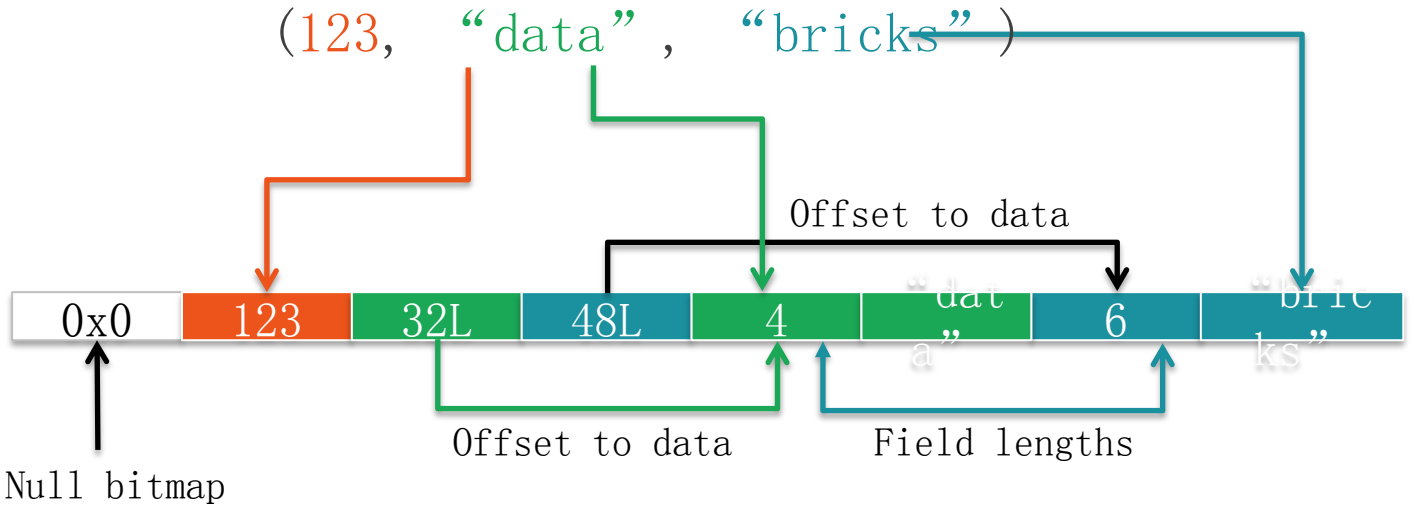
Unfortunately, type-safe is not free…

Spark SQL execution model

Each operator is an "iterator" that consumes records from its input operator

```
class Filter {
  def next(): Boolean = {
    var found = false
    while (!found && child.next()) {
      found = predicate(child.fetch())
    }
    return found
  }

  def fetch(): InternalRow = {
    child.fetch()
  }
  …
}
```

Tungsten's Compact Encoding

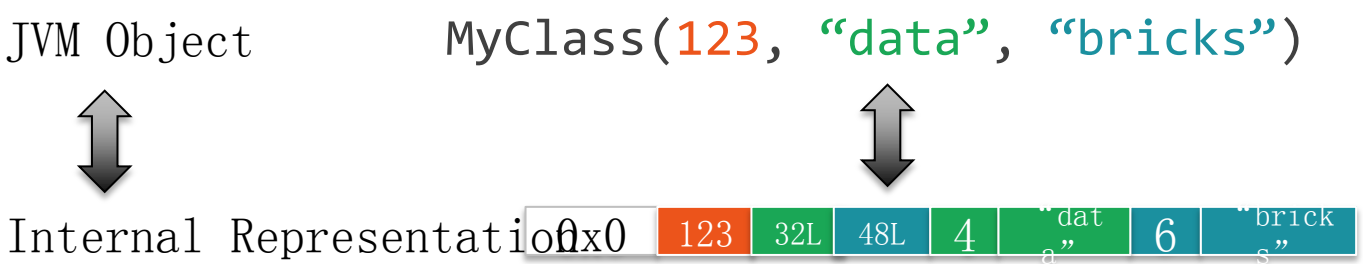


Type-safe is not free

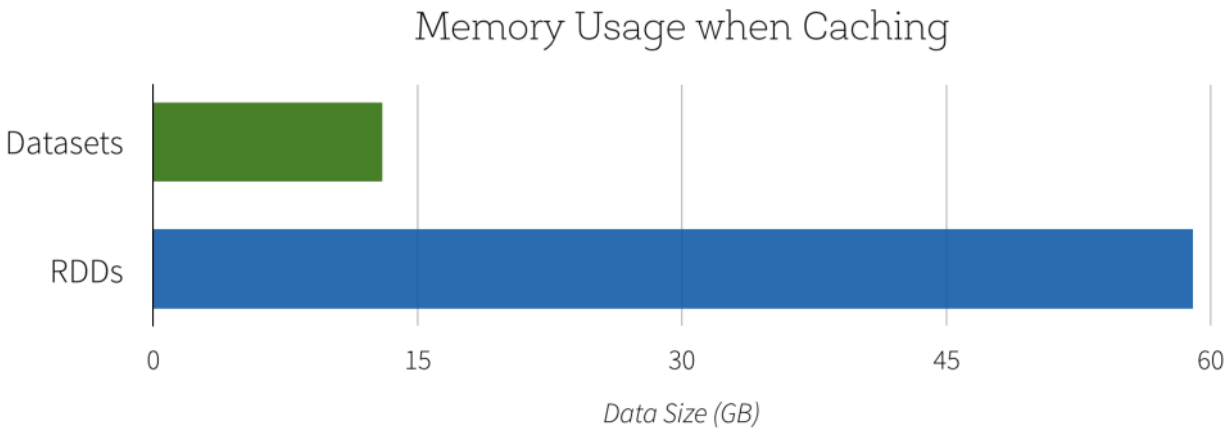
- Dataset deserialize binary data to domain object before apply lambda function.
- Dataset serialize domain object to Tungsten binary format before execute normal SQL operator.

Encoders

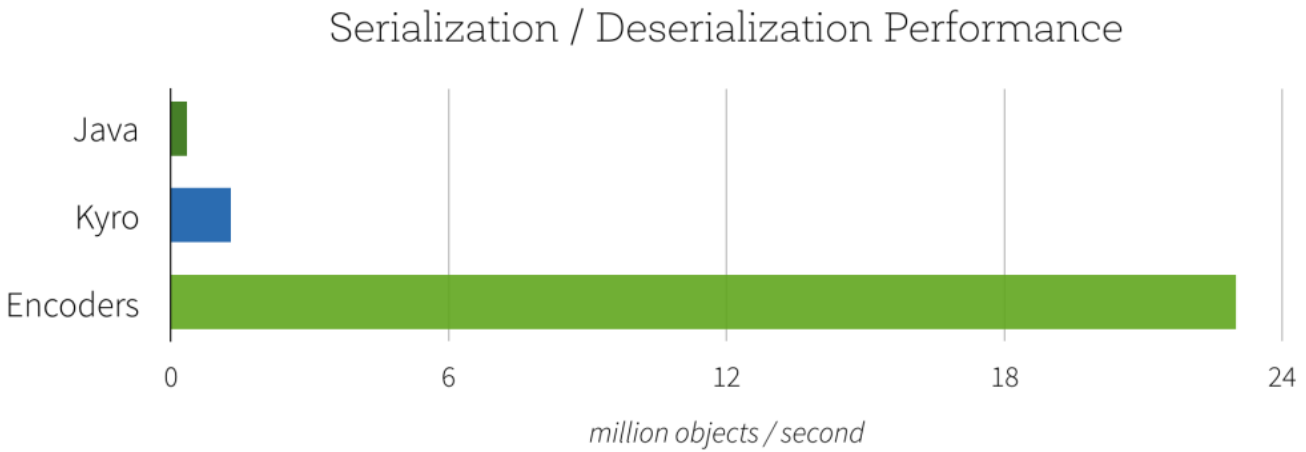
Encoders translate between domain objects and Spark's internal format



Space Efficiency



Serialization performance



Higher is better

Other efforts

- Eliminate back-to-back serialization

```
ds.map(...).filter(...).map(...)
```

Only has one serialization

- Whole-stage code generation support

```
ds.map(...).filter(...).map(...)
```

Compile to one java method

Other efforts

- Eliminate back-to-back serialization

```
ds.map(...).filter(...).map(...)
```

Only has one serialization

- W[illegible]pport

```
ds.map(...).filter(...).map(...)
```

Compile to one java method

RDD is faster than Dataset on very simple operations

Future Plans

More supported types in Encoder

- **Supported basic types**: boolean, int, double, String, Timestamp, BigDecimal, BigInteger, etc.
- **Supported composable types**: Array, Seq, Map, scala Product, java bean

- **Going to support**: custom collection(List, MySeq) and more composable type.

Support Whole-Stage codegen in more operations

- Now only support `map`, `filter` and aggregate

- Need to support: `mapPartitions`, `flatMap`, `mapGroups`, `cogroup`, etc.

JVM bytecode analysis

```
df.map(_.name)              =>      df.select(“name”)
df.filter(_.age > 18)       =>      df.filter(GreaterThan(col(“age”), 18))
df.map(_.id + 1)            =>      df.select(Add(col(“id”), 1))
```

No serialization overhead anymore!

Thank you!