Structuring Apache Spark

SQL, DataFrames, Datasets, and Streaming

Michael Armbrust - @michaelarmbrust

Spark Summit 2016

databricks™

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: $\underbrace{\text{Partition => Iterator[T]}}_{\text{Opaque Computation}}$

Background: What is in an RDD?

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

Opaque Data

Struc·ture

['strək(t)SHər]

verb

1. construct or arrange according to a plan; give a pattern or organization to.

Why 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.

Structured APIs In Spark

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

Analysis errors reported before a distributed job starts

Datasets API

Type-safe: operate on domain objects with compiled lambda functions

```scala
val df = spark.read.json("people.json")

// Convert data to domain objects.
case class Person(name: String, age: Int)
val ds: Dataset[Person] = df.as[Person]
ds.filter(_.age > 30)

// Compute histogram of age by name.
val hist = ds.groupBy(_.name).mapGroups {
  case (name, people: Iter[Person]) =>
    val buckets = new Array[Int](10)
    people.map(_.age).foreach { a =>
      buckets(a / 10) += 1
    }
    (name, buckets)
}
```

DataFrame = Dataset[Row]

- Spark 2.0 unifies these APIs
- Stringly-typed methods will downcast to generic **Row** objects
- Ask Spark SQL to enforce types on generic rows using `df.as[MyClass]`

What about python?

Some of the goals of the Dataset API have always been available!

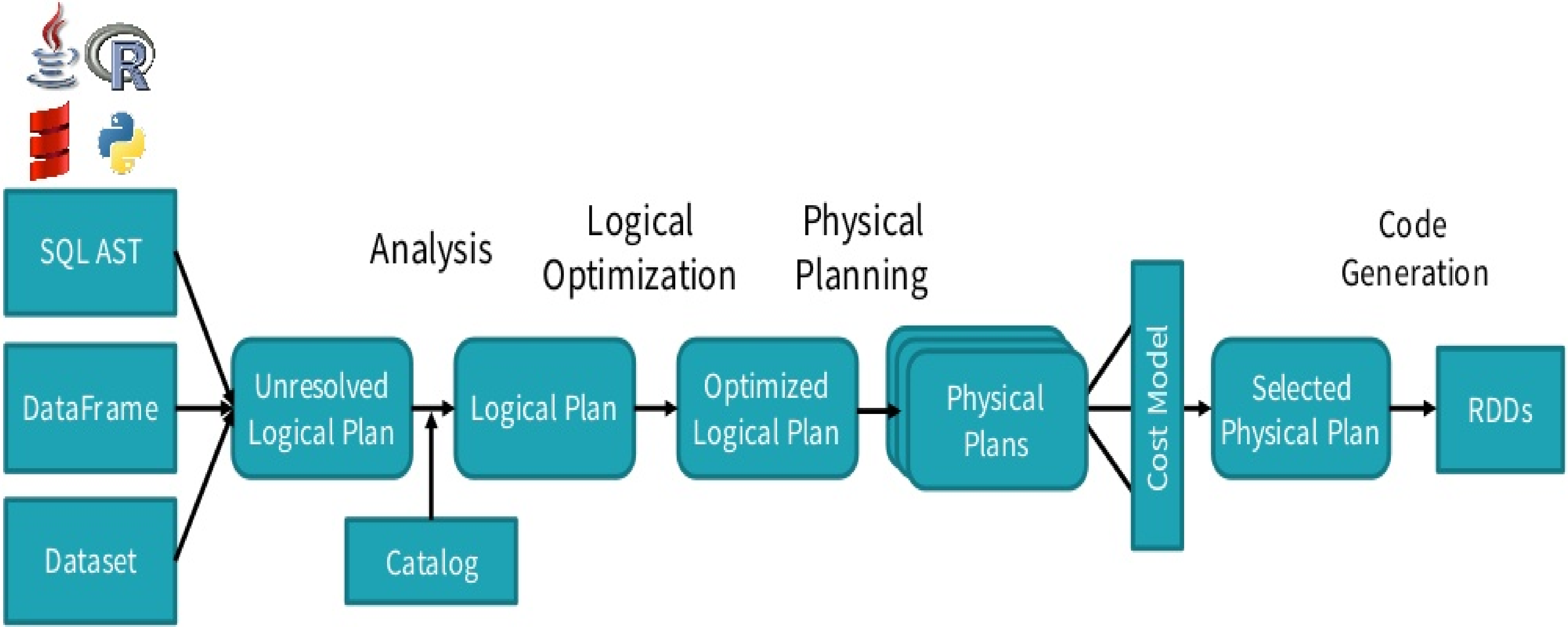
Scala

```
df.map(x => x(0).asInstanceOf[String])
```

python

```
df.map(lambda x: x.name)
```

Shared Optimization & Execution



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

Structuring Computation

```
WITH customer_total_return AS  (SELECT
sr_customer_sk AS ctr_customer_sk,
sr_store_sk AS ctr_store_sk,
sum(sr_return_amt) AS ctr_total_return
FROM store_returns, date_dim  WHERE
sr_returned_date_sk = d_date_sk AND d_year
= 2000  GROUP BY sr_customer_sk,
sr_store_sk) SELECT c_customer_id  FROM
customer_t
```

Columns

New value, computed based on input values.

DSL

```
col("x") === 1
df("x") === 1
expr("x = 1")
```

SQL Parser

```
sql("SELECT … WHERE x = 1")
```

Complex Columns With Functions

- 100+ native functions with optimized codegen implementations
  - String manipulation – `concat`, `format_string`, `lower`, `lpad`
  - Data/Time – `current_timestamp`, `date_format`, `date_add`, …
  - Math – `sqrt`, `randn`, …
  - Other – `monotonicallyIncreasingId`, `sparkPartitionId`, …

python

```python
from pyspark.sql.functions import *
yesterday = date_sub(current_date(), 1)
df2 = df.filter(df.created_at > yesterday)
```

Scala

```scala
import org.apache.spark.sql.functions._
val yesterday = date_sub(current_date(), 1)
val df2 = df.filter(df("created_at") > yesterday)
```

	Functions	Columns
You Type	`(x: Int) => x == 1`	`col("x") === 1`
Spark Sees	`class $anonfun$1{ def apply(Int): Boolean }`	`EqualTo(x, Lit(1))`

Columns: Predicate pushdown

You Write

```
spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "people")
  .load()
  .where($"name" === "michael")
```

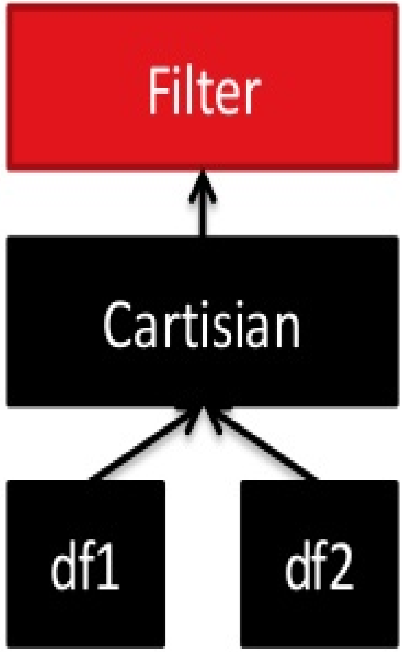
Spark Translates For Postgres

```
SELECT * FROM people WHERE name = 'michael'
```

Columns: Efficient Joins

```
myUDF = udf(lambda x, y: x == y)
df1.join(df2, myUDF(col("x"), col("y")))
```

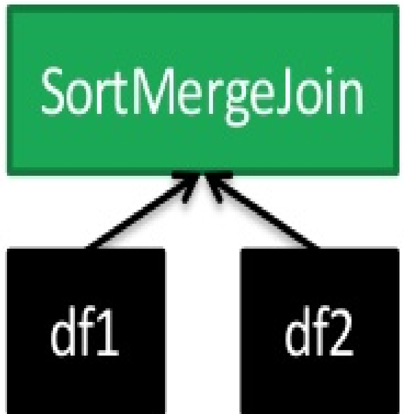
n^2



Equal values sort to the same place

```
df1.join(df2, col("x") == col("y"))
```

n log n

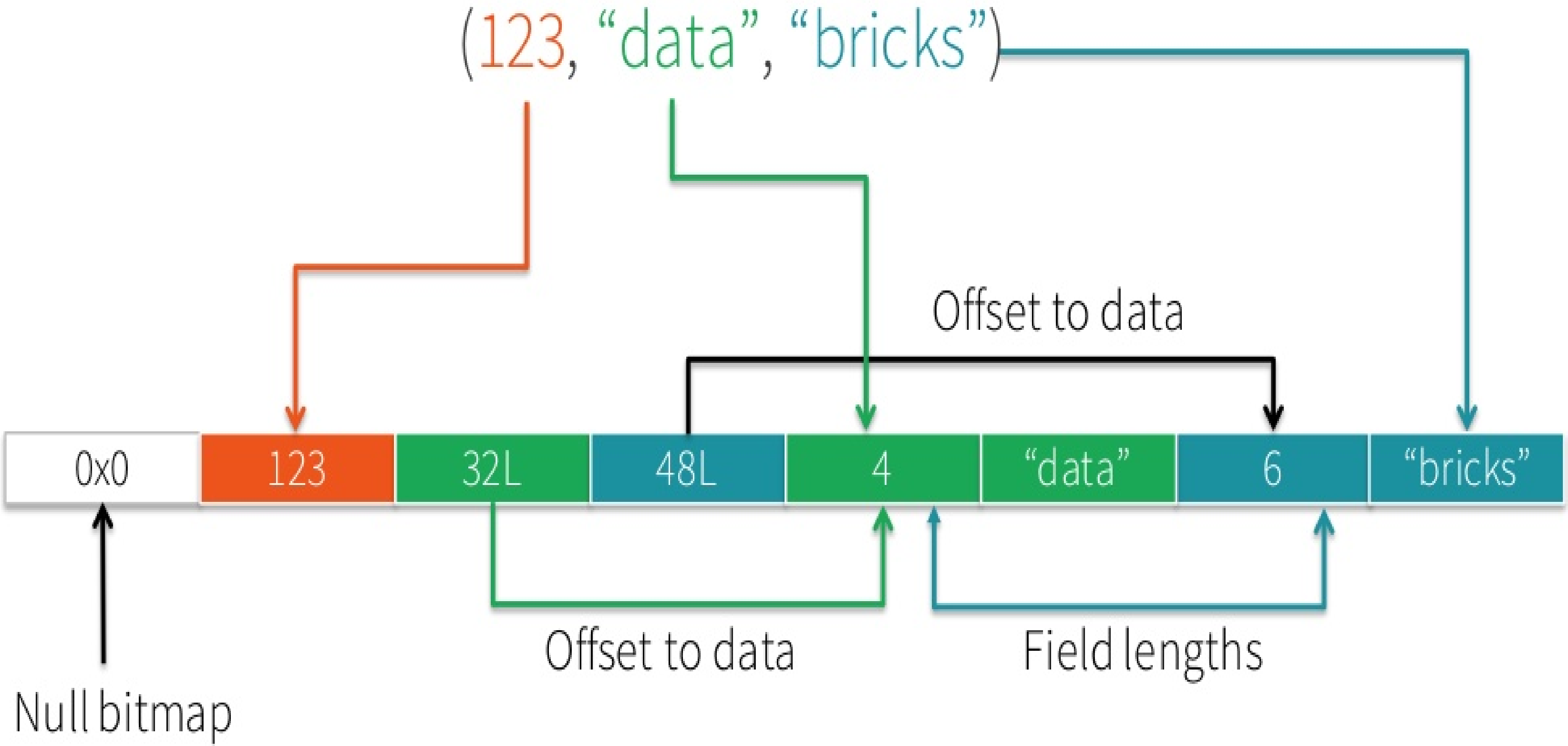


Structuring Data

Spark's Structured Data Model

- **Primitives**: Byte, Short, Integer, Long, Float, Double, Decimal, String, Binary, Boolean, Timestamp, Date
- **Array[Type]**: variable length collection
- **Struct**: fixed # of nested columns with fixed types
- **Map[Type, Type]**: variable length association

Tungsten's Compact Encoding



Encoders

Encoders translate between domain objects and Spark's internal format

JVM Object

`MyClass(123, “data”, “bricks”)`

Internal Representation

0x0	123	32L	48L	4	“data”	6	“bricks”

Bridge Objects with Data Sources

Encoders map columns to fields by name

{ JSON }　JDBC　Parquet

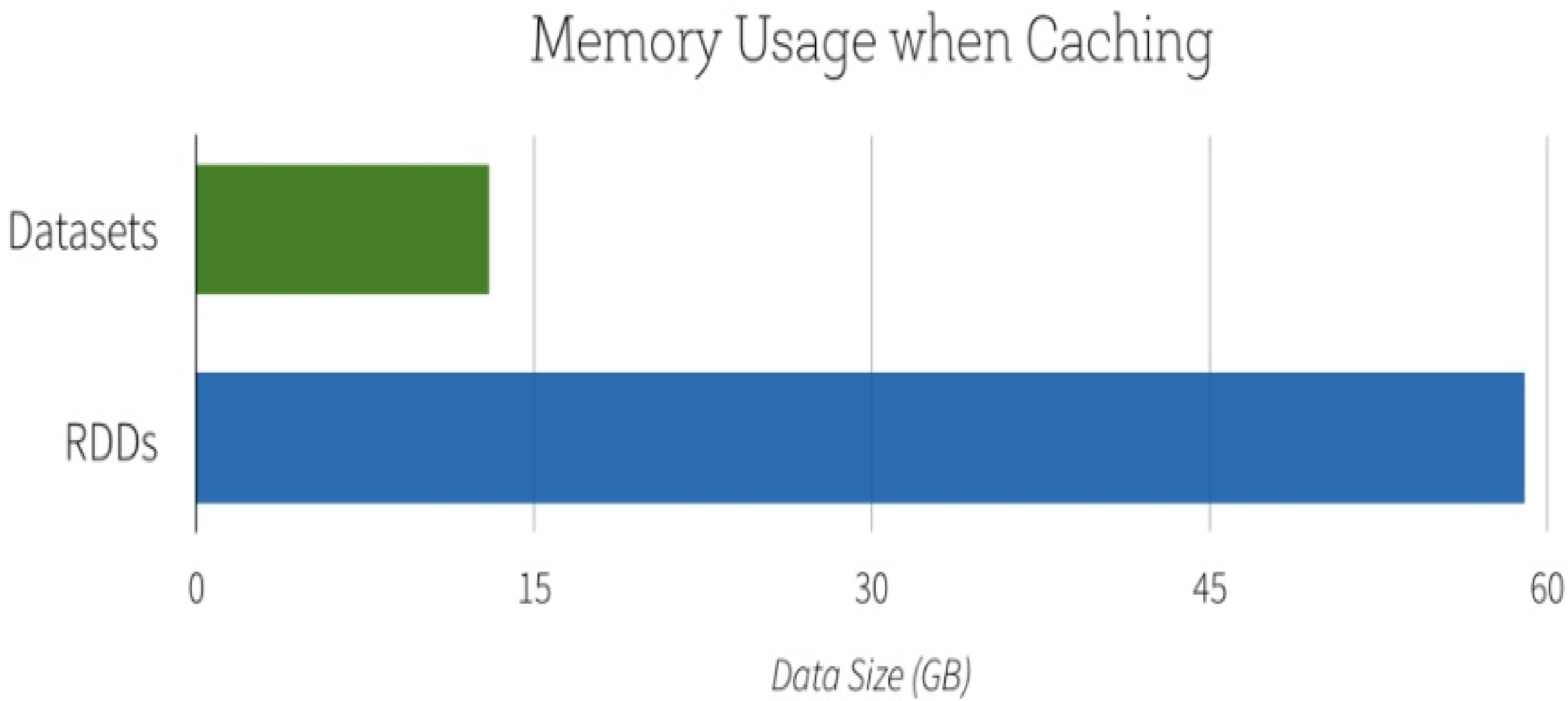
AVRO　CSV　cassandra

elasticsearch.　amazon webservices Amazon Redshift　memsql

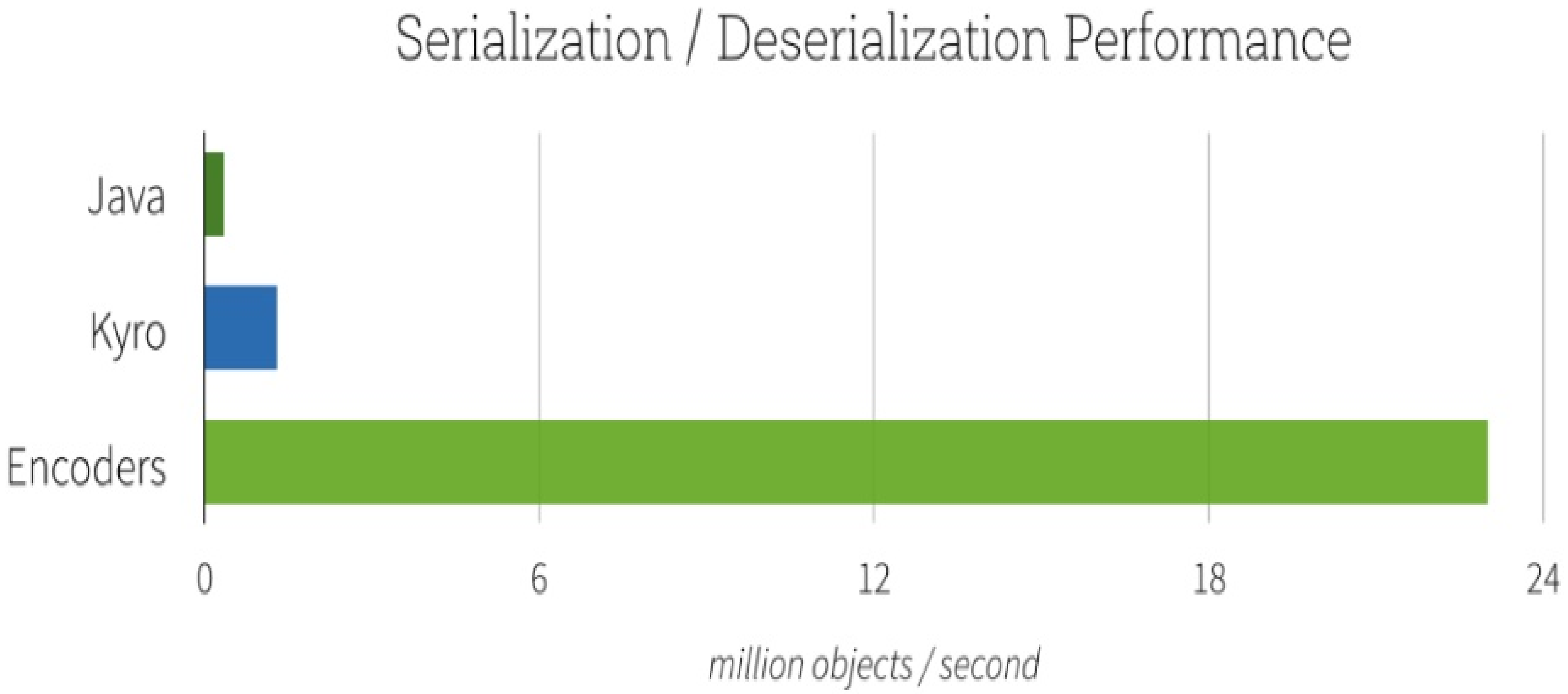
```
{
  "name": "Michael",
  "zip": "94709"
  "languages": ["scala"]
}
```

```
case class Person(
  name: String,
  languages: Seq[String],
  zip: Int)
```

Space Efficiency



Serialization performance



Operate Directly On Serialized Data

DataFrame Code / SQL

```
df.where(df("year") > 2015)
```

Catalyst Expressions

```
GreaterThan(year#234, Literal(2015))
```

Low-level bytecode

```
bool filter(Object baseObject) {
    int offset = baseOffset + bitSetWidthInBytes + 3*8L;
    int value = Platform.getInt(baseObject, offset);
    return value34 > 2015;
}
```

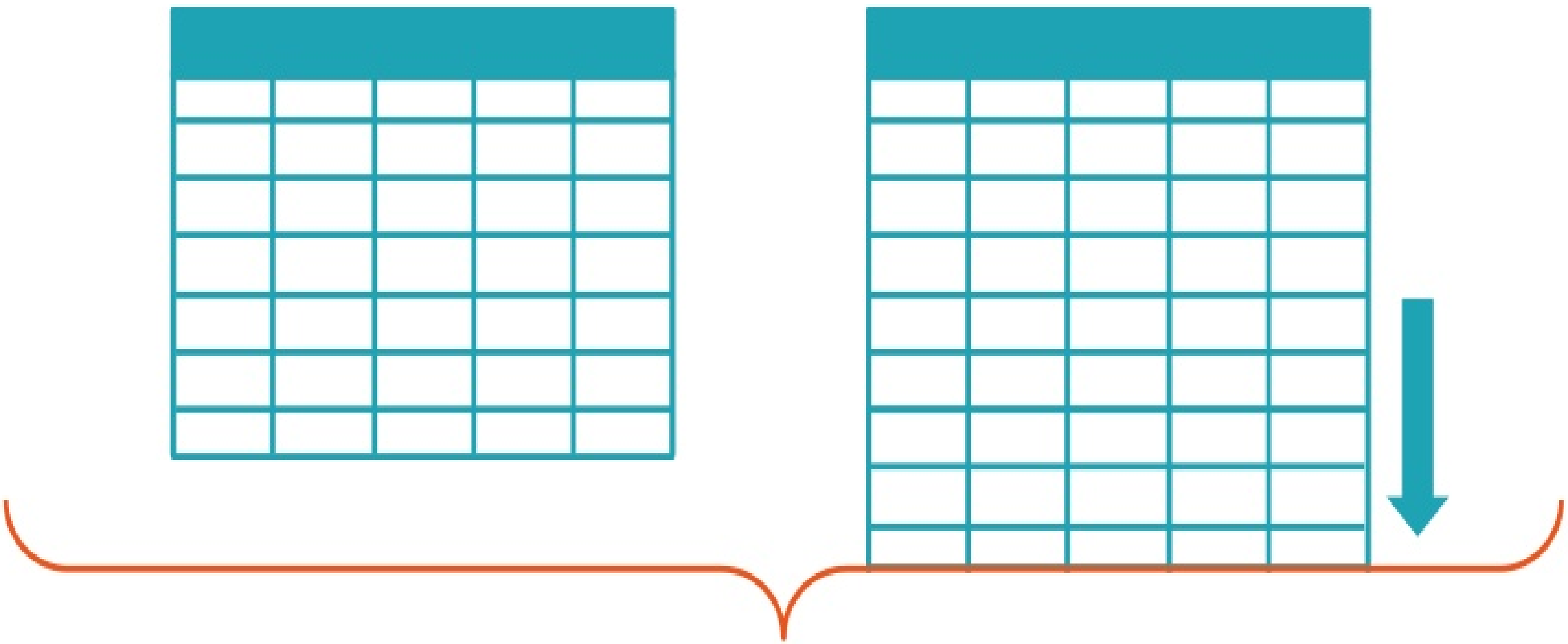
JVM ***intrinsic*** JIT-ed to pointer arithmetic

Structured Streaming

The simplest way to perform streaming analytics is not having to ***reason*** about streaming.

Apache Spark 1.3
Static DataFrames

Apache Spark 2.0
Continuous DataFrames



Single API !

Structured Streaming

- **High-level streaming API built on Apache Spark SQL engine**
  - Runs the same queries on DataFrames
  - Event time, windowing, sessions, sources & sinks

- **Unifies streaming, interactive and batch queries**
  - Aggregate data in a stream, then serve using JDBC
  - Change queries at runtime
  - Build and apply ML models

Example: Batch Aggregation

```
logs = spark.read.format("json").open("s3://logs")

logs.groupBy(logs.user_id).agg(sum(logs.time))
    .write.format("jdbc")
    .save("jdbc:mysql//...")
```

Example: Continuous Aggregation

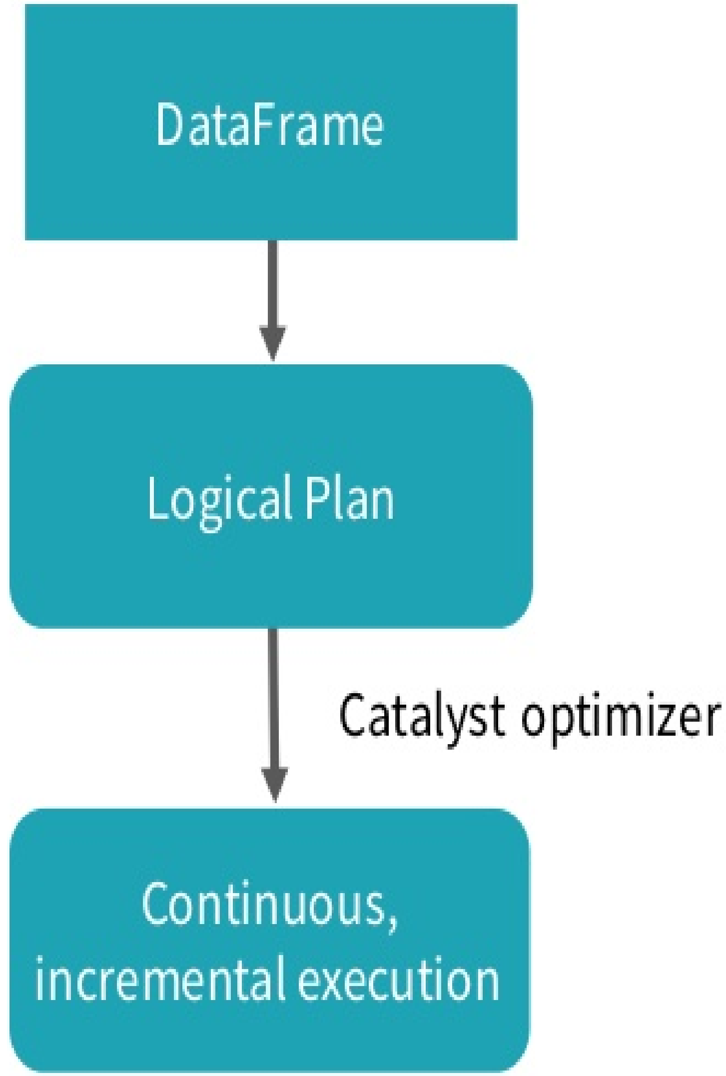
```
logs  = spark.read.format("json").stream("s3://logs")

logs.groupBy(logs.user_id).agg(sum(logs.time))
    .write.format("jdbc")
    .stream("jdbc:mysql//...")
```

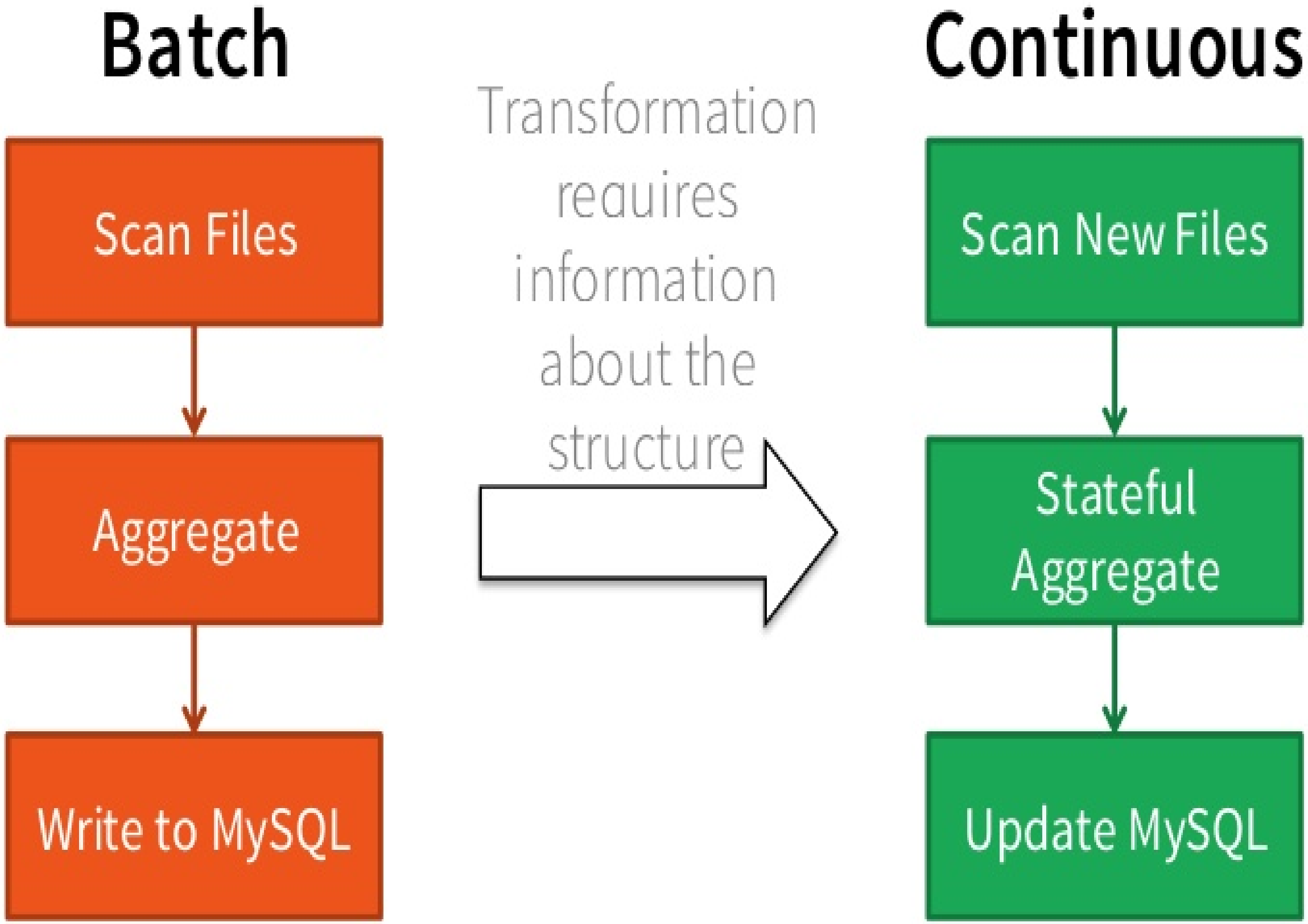
Execution

Logically:
DataFrame operations on static data
(i.e. as easy to understand as batch)

Physically:
Spark automatically runs the query in streaming fashion
(i.e. incrementally and continuously)



Incrementalized By Spark



What's Coming?

- Apache Spark 2.0
  - Unification of the DataFrame/Dataset & *Context APIs
  - Basic streaming API
  - Event-time aggregations
- Apache Spark 2.1+
  - Other streaming sources / sinks
  - Machine learning
  - Watermarks
- Structure in other libraries: MLlib, GraphFrames

Questions?

@michaelarmbrust

databricks™