Real-time Spark

From interactive queries to streaming

Michael Armbrust - @michaelarmbrust

Strata + Hadoop World 2016

Real-time Analytics

Goal: *freshest* answer, as *fast* as possible

Challenges

- Implementing the analysis
- Making sure it runs efficiently
- Keeping the answer is up to date

Develop Productively

with powerful, simple APIs in 

Write Less Code: Compute an Average


```
private IntWritable one =
  new IntWritable(1)
private IntWritable output =
  new IntWritable()
proctected 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)
}
```


```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [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()
```

Write Less Code: Compute an Average

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")) \
    .map(lambda …) \
    .collect()
```

Full API Docs

- Python
- Scala
- Java
- R

Dataset

noun – [dey-tuh-set]

1. A distributed collection of data with a known schema.
2. A high-level abstraction for selecting, filtering, mapping, reducing, aggregating and plotting structured data (*cf. Hadoop, RDDs, R, Pandas*).
3. ***Related:*** DataFrame – a distributed collection of generic row objects (i.e. the result of a SQL query)

Datasets with SQL

Standards based: Supports most popular constructs in HiveQL, ANSI SQL

Compatible: Use JDBC to connect with popular BI tools


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

Dynamic Datasets (DataFrames)

Concise: great for ad-hoc interactive analysis

Interoperable: based on and R / pandas, easy to go back and forth

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

Static Datasets

No boilerplate: automatically convert to/from domain objects

Safe: compile time checks for correctness

```
val df = ctx.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)
}
```

Unified, Structured APIs in Spark 2.0

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

Unified: Input & Output

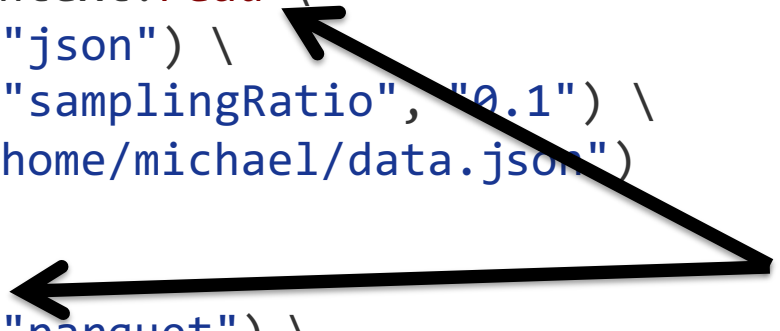
Unified interface to reading/writing data in a variety of formats:

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

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

  .saveAsTable("fasterData")
```

Unified: Input & Output

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")
```



`read` and `write` functions create new builders for doing I/O

Unified: Input & Output

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")
```

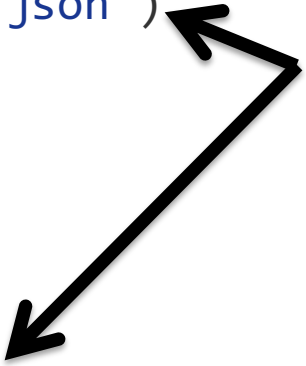
Builder methods specify:

- Format
- Partitioning
- Handling of existing data

Unified: Input & Output

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")
```



`load(…)`, `save(…)` or `saveAsTable(…)` to finish the I/O

Unified: Data Source API

Spark SQL's Data Source API can read and write DataFrames using a variety of formats.

Built-In

plain text*


ORC

External

Amazon Redshift

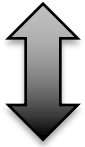
and more…
memsql

Find more sources at http://spark-packages.org/

Bridge Objects with Data Sources

Automatically map columns to fields by name

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

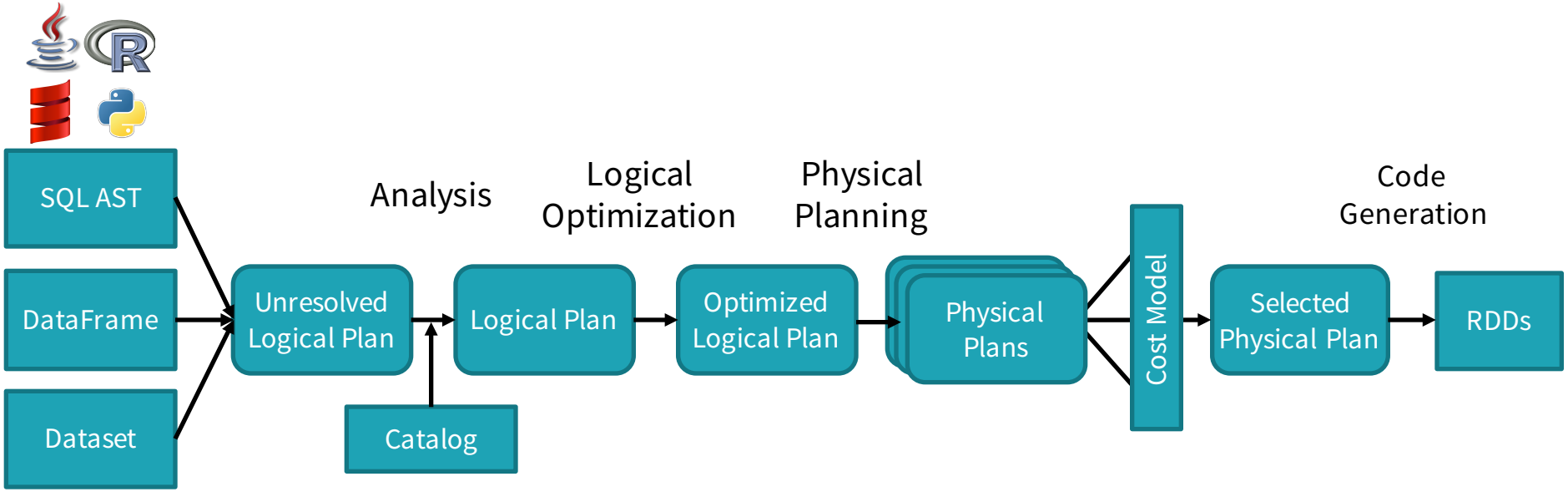


```
case class Person(
  name: String,
  languages: Seq[String],
  zip: Int)
```

Execute Efficiently

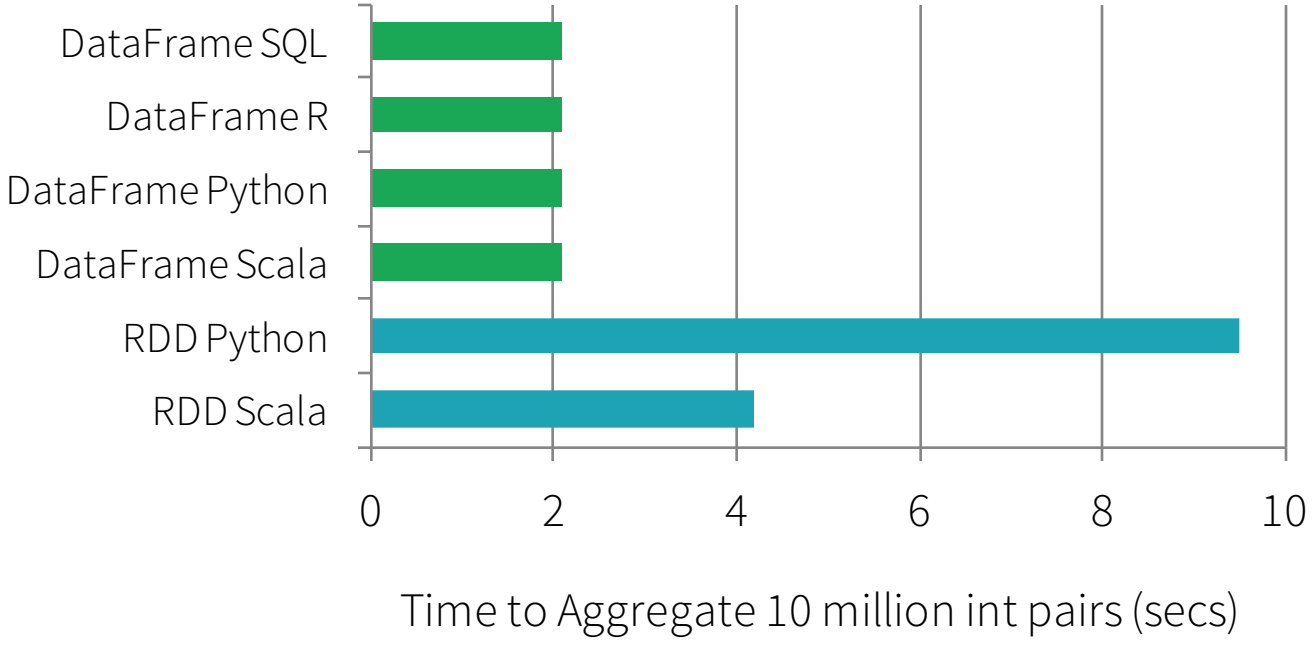
using the catalyst optimizer & tungsten engine in 

Shared Optimization & Execution



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

Not Just Less Code, Faster Too!



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


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

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;
}
```

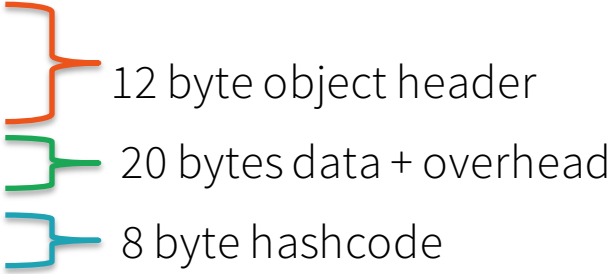
JVM ***intrinsic*** JIT-ed to pointer arithmetic

The overheads of JVM objects

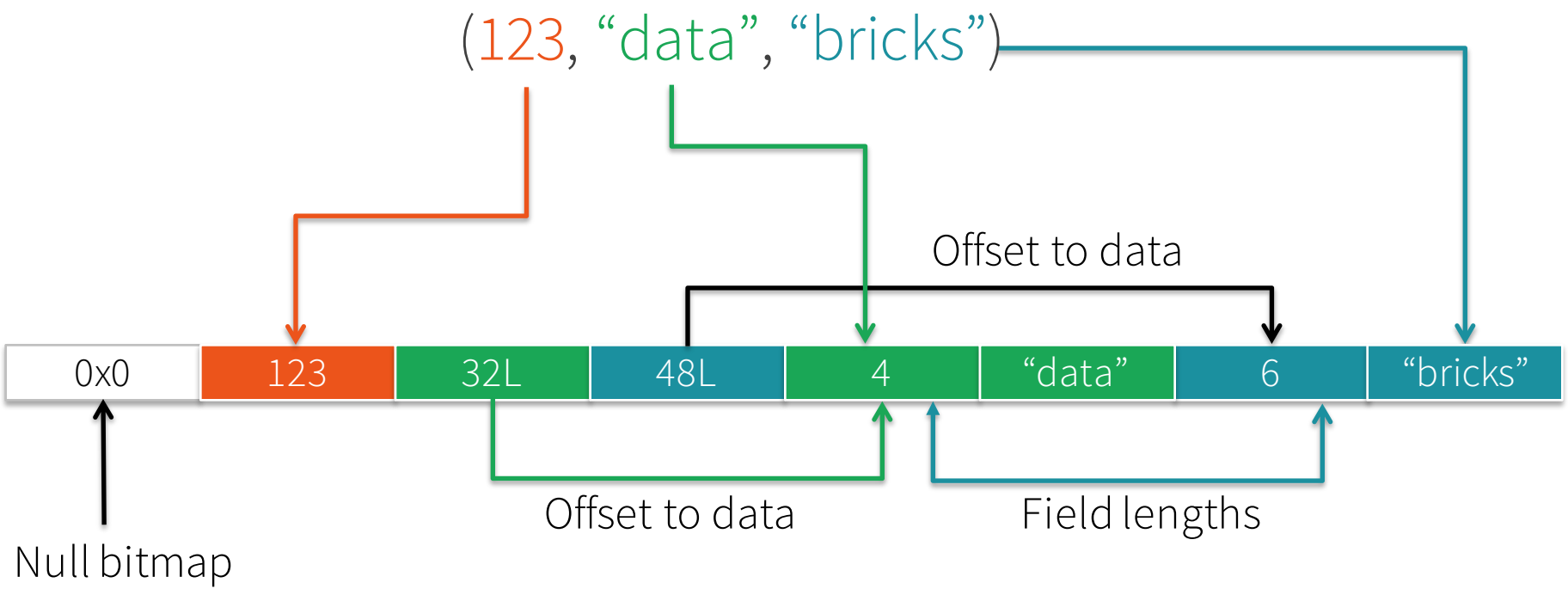
"abcd"

- **Native:** 4 bytes with UTF-8 encoding
- Java: 48 bytes

```
java.lang.String object internals:
OFFSET  SIZE   TYPE DESCRIPTION                    VALUE
     0     4        (object header)                ...
     4     4        (object header)                ...
     8     4        (object header)                ...
    12     4 char[] String.value                   []
    16     4    int String.hash                    0
    20     4    int String.hash32                  0
Instance size: 24 bytes (reported by Instrumentation API)
```



Tungsten's Compact Encoding

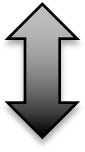


Encoders

Encoders translate between domain objects and Spark's internal format

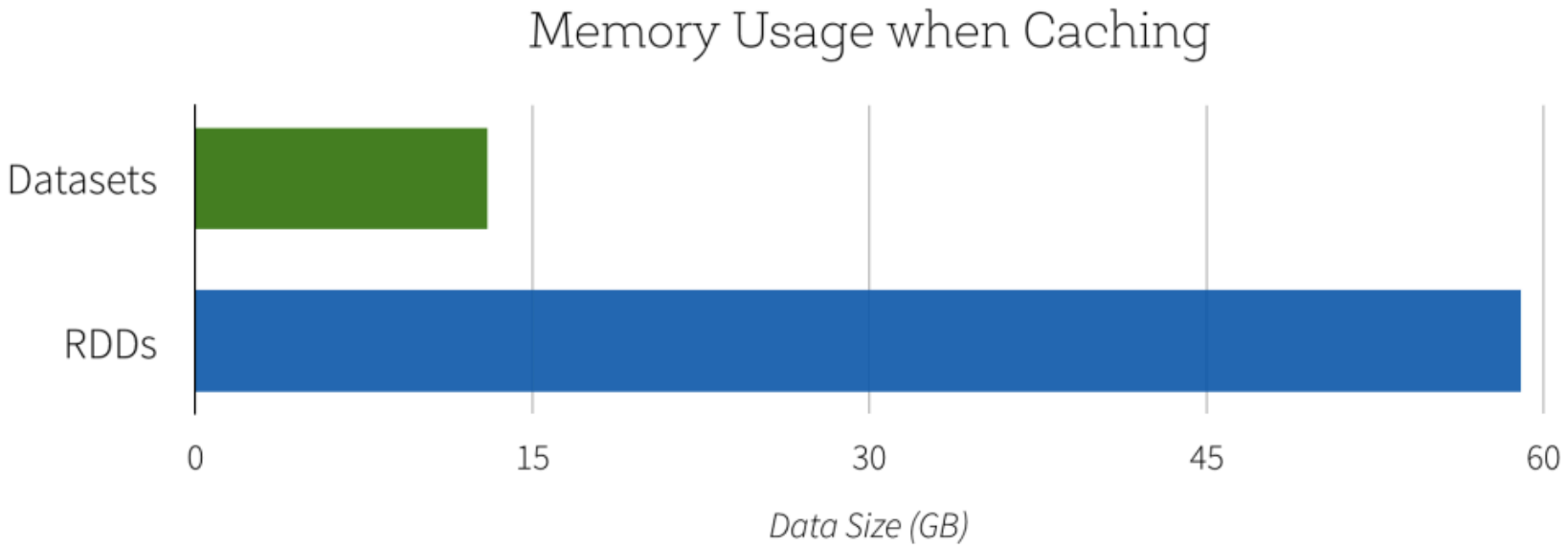
JVM Object

```
MyClass(123, “data”, “bricks”)
```

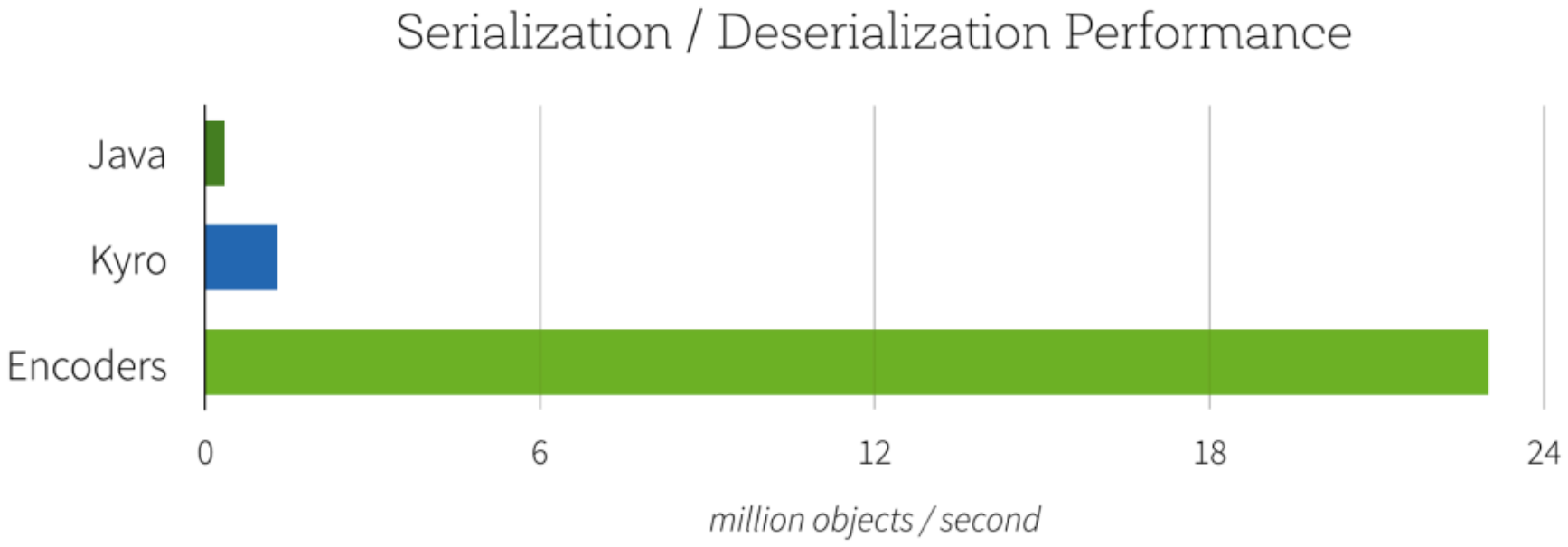


Internal Representation

Space Efficiency



Serialization performance



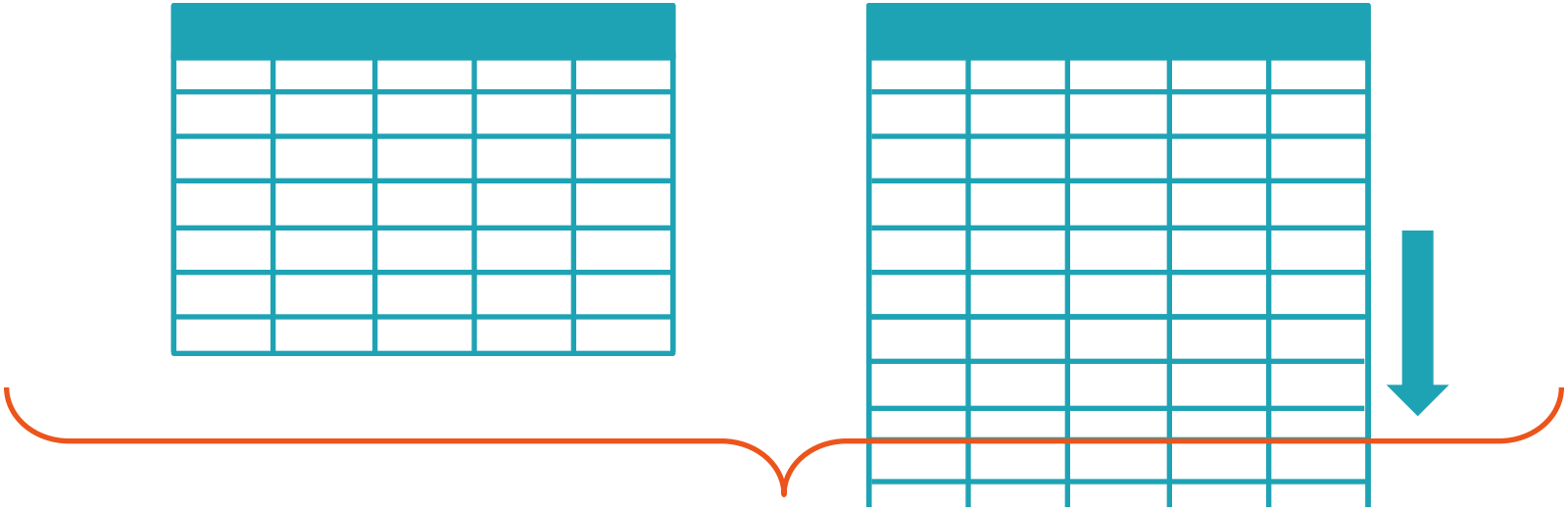
Update Automatically

using Structured Streaming in 

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

Spark 1.3
Static DataFrames

Spark 2.0
Infinite DataFrames



Single API

Example: Batch Aggregation

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

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

Example: Continuous Aggregation

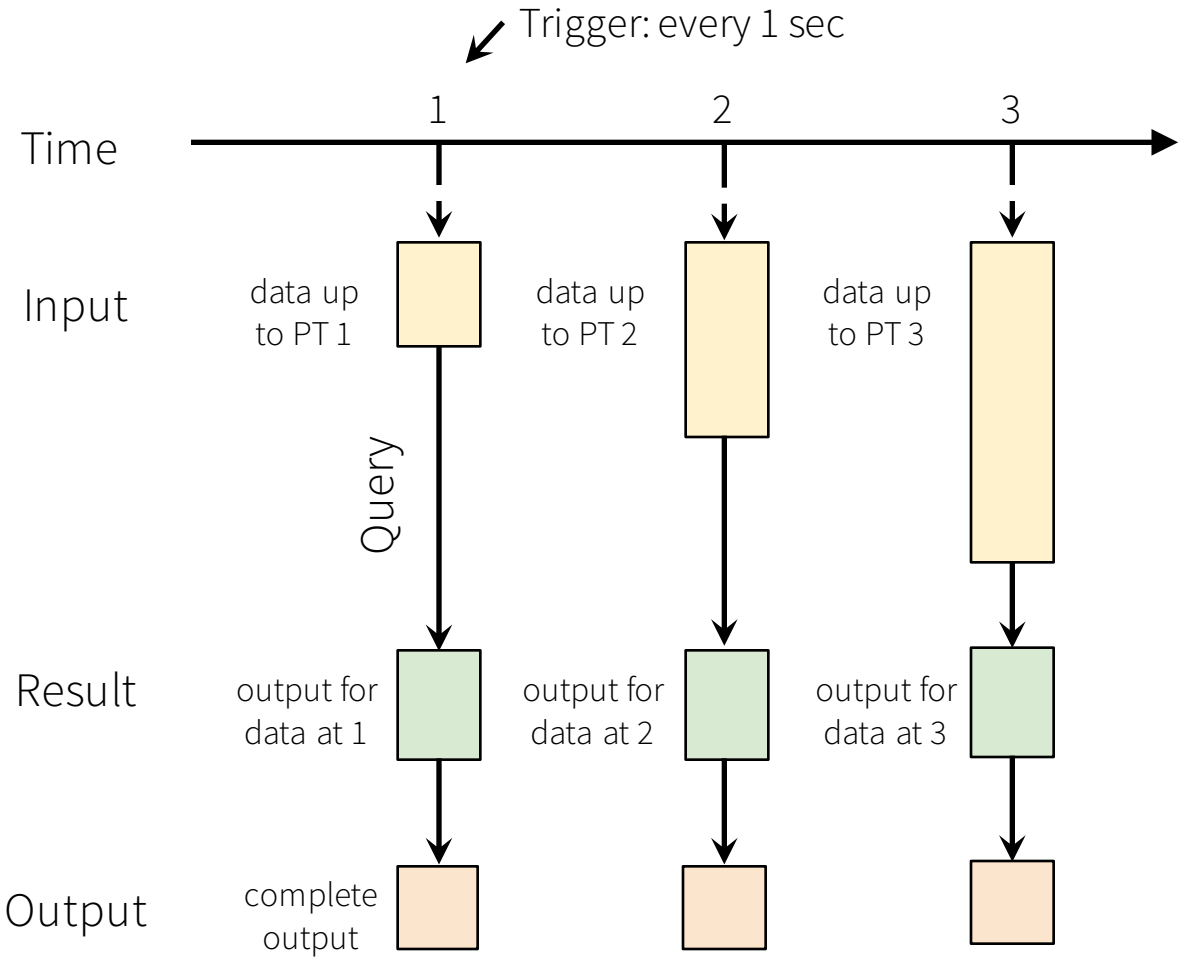
```
logs  = ctx.read.format("json").stream("s3://logs")

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

Structured Streaming in Spark

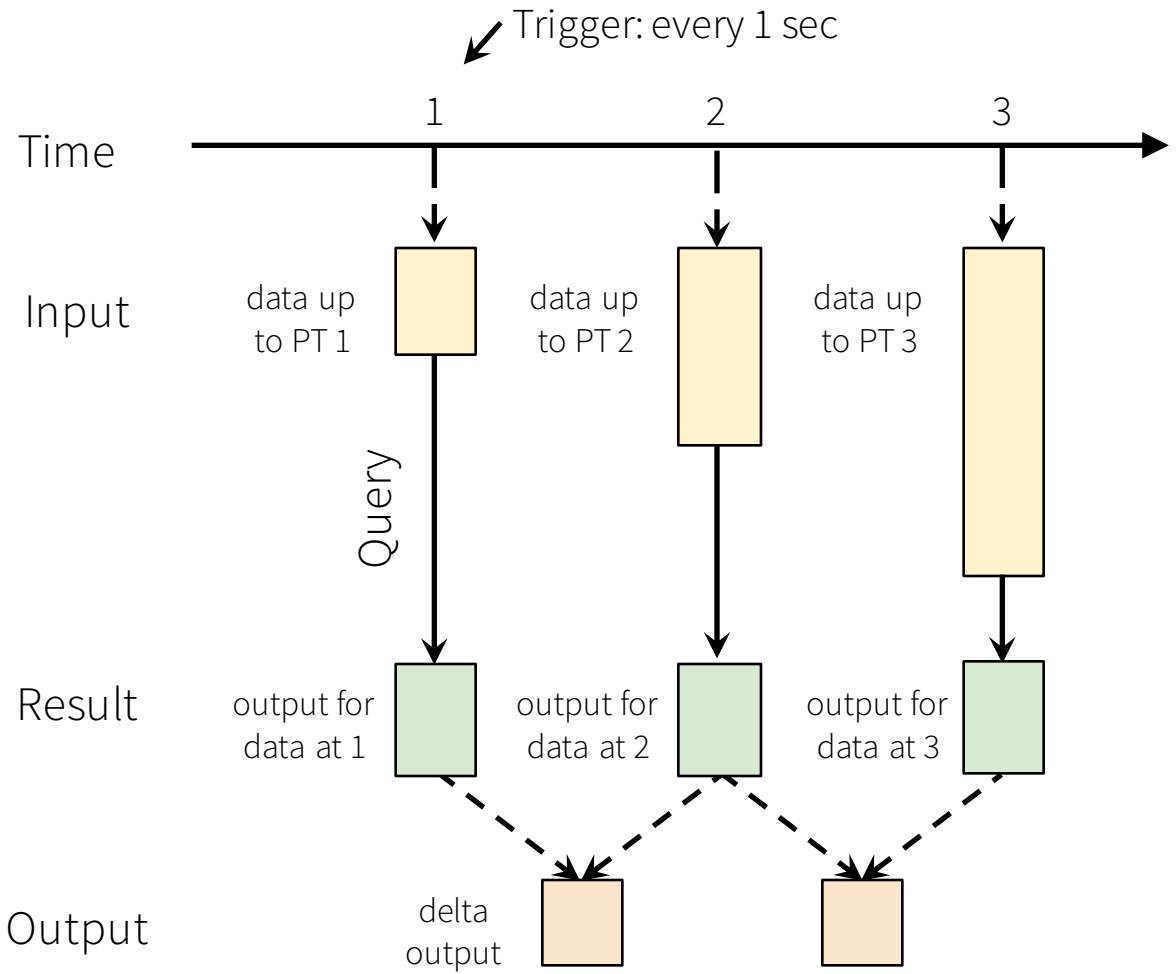
- **High-level streaming API built on Spark SQL engine**
  - Runs the same queries on DataFrames
  - Event time, windowing, sessions, sources & sinks

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

Model


Trigger: every 1 sec
Time
1
2
3
Input
data up to PT 1
data up to PT 2
data up to PT 3
Query
Result
output for data at 1
output for data at 2
output for data at 3
Output
complete output

Model


Trigger: every 1 sec
Time
1
2
3
Input
data up to PT 1
data up to PT 2
data up to PT 3
Query
Result
output for data at 1
output for data at 2
output for data at 3
Output
delta output

Integration End-to-End

Stream

(home.html, 10:08)

(product.html, 10:09)

(home.html, 10:10)

. . .

Streaming engine

MySQL

Page	Minute	Visits
home	10:09	21
pricing	10:10	30
…	…	…

What could go wrong?

- Late events
- Partial outputs to MySQL
- State recovery on failure
- Distributed reads/writes
- …

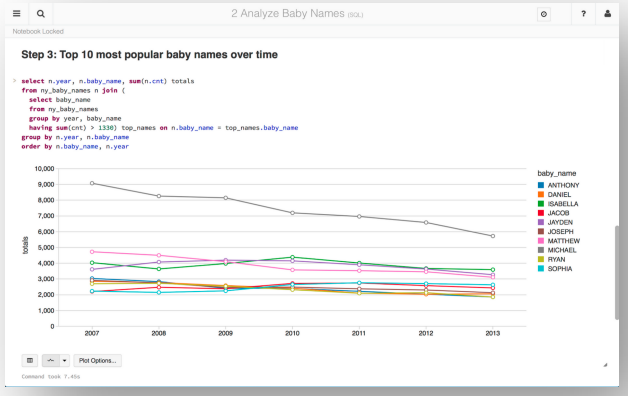
Rest of Spark will follow

- Interactive queries should just work
- Spark's data source API will be updated to support seamless
  streaming integration
  - Exactly once semantics **end-to-end**
  - Different output modes (complete, delta, update-in-place)
- ML algorithms will be updated too

What can we do with this that's hard with other engines?

- Ad-hoc, interactive queries
- Dynamic changing queries
- Benefits of Spark: elastic scaling, straggler mitigation, etc

Demo



Running in databricks™

- Hosted Spark in the cloud
- Notebooks with integrated visualization
- Scheduled production jobs

Community Edition is *Free* for everyone!

http://go.databricks.com/databricks-community-edition-beta-waitlist

Learn More

Up Next TD

Today, 1:50-2:30 AMA

@michaelarmbrust

Simple and fast real-time analytics

- Develop Productively
- Execute Efficiently
- Update Automatically

Questions?