Writing Continuous Applications with PySpark

https://dbricks.co/pybaysf

Jules S. Damji

PyBay, SF
August 15, 2019

databricks

I have used Apache Spark 2.x Before…

I know the difference between DataFrame and RDDs…

Apache Spark Developer Advocate and Author @ Databricks

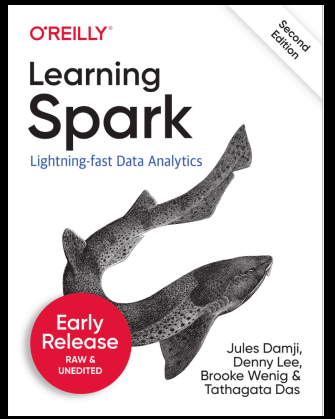
Developer Advocate @ Hortonworks

Software engineering @ Sun Microsystems, Netscape, @Home, VeriSign, Scalix, Centrify, LoudCloud/Opsware, ProQuest

Program Chair Spark + AI Summit

https://www.linkedin.com/in/dmatrix

@2twitme



databricks

VISION

Accelerate innovation by unifying data science, engineering and business

---

SOLUTION

Unified Analytics Platform

WHO WE ARE

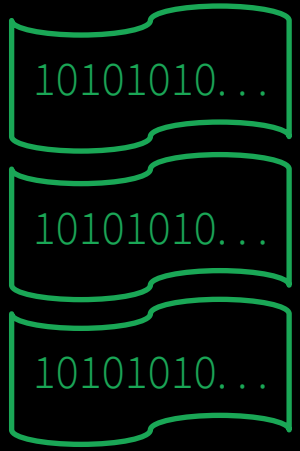
- Original creators of 

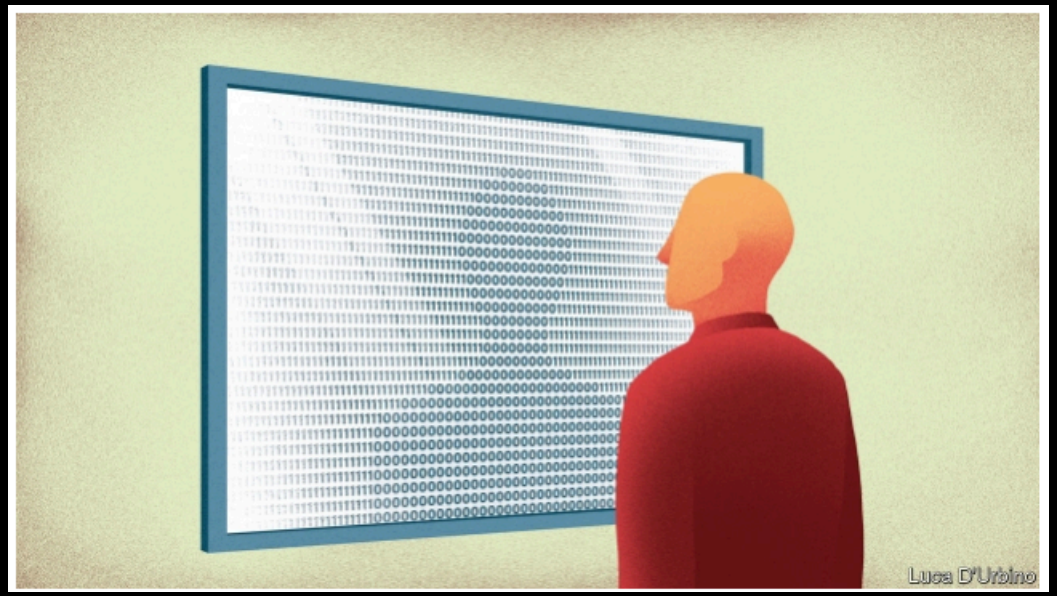
- 2000+ global companies use our platform across big data & machine learning lifecycle

Agenda for Today's Talk

- What & Why Apache Spark
- Why Streaming Applications are Difficult
- What's Structured Streaming
  - Anatomy of a Continunous Application
  - Tutorials
- Q & A

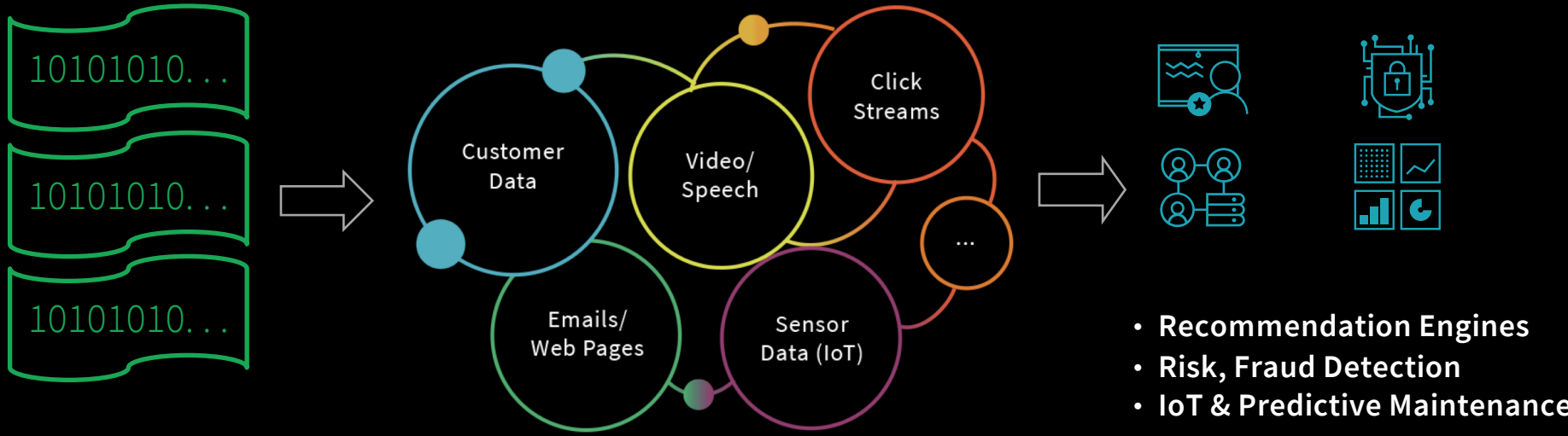
How to think about data in 2019 - 2020





"Digital Data Zeitgeist"

How to think about data in 2019 - 2020



- **Recommendation Engines**
- **Risk, Fraud Detection**
- **IoT & Predictive Maintenance**

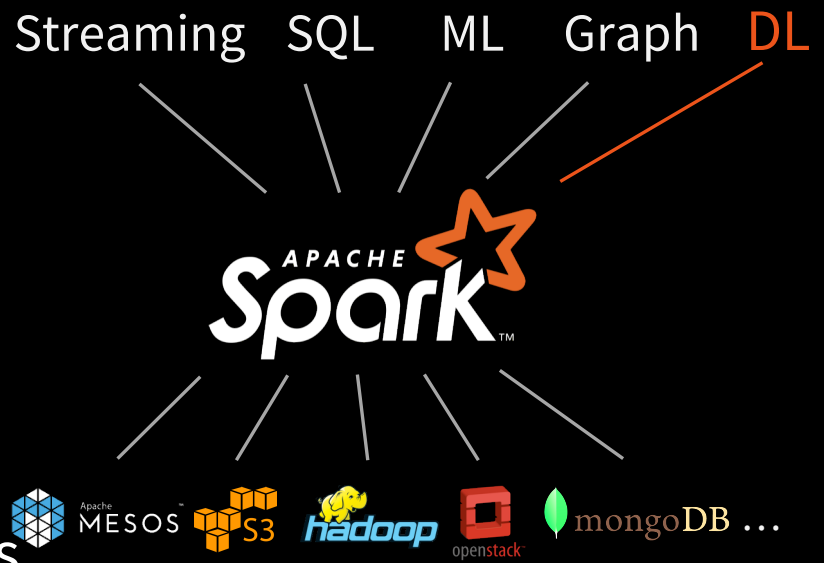
"Digital Data Zeitgeist"

Why Apache Spark?

databricks

What is Apache Spark?

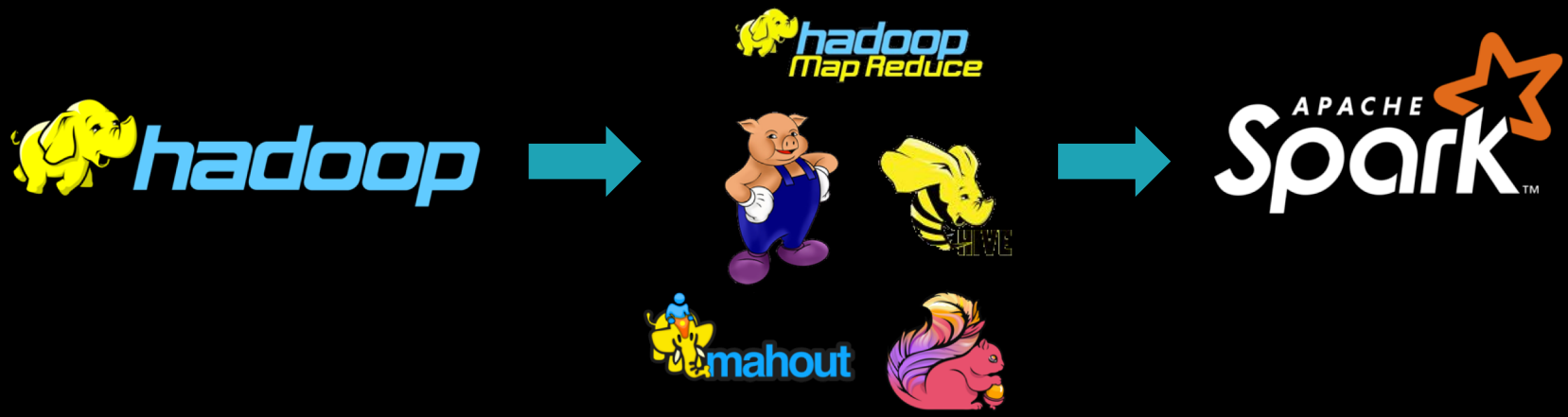
- General cluster computing engine that extends MapReduce
- Rich set of APIs and libraries
- Unified Engine
- Large community: 1000+ orgs, clusters up to 8000 nodes
- Supports Deep Learning Frameworks



Unique Thing about Spark

- Unification: same engine and same API for diverse use cases
  - Streaming, batch, or interactive
  - ETL, SQL, machine learning, or graph
  - Deep Learning Frameworks w/Horovod
    - TensorFlow
    - Keras
    - PyTorch

Faster, Easier to Use, Unified



First Distributed Processing Engine

Specialized Data Processing Engines

Unified Data Processing Engine

Benefits of Unification

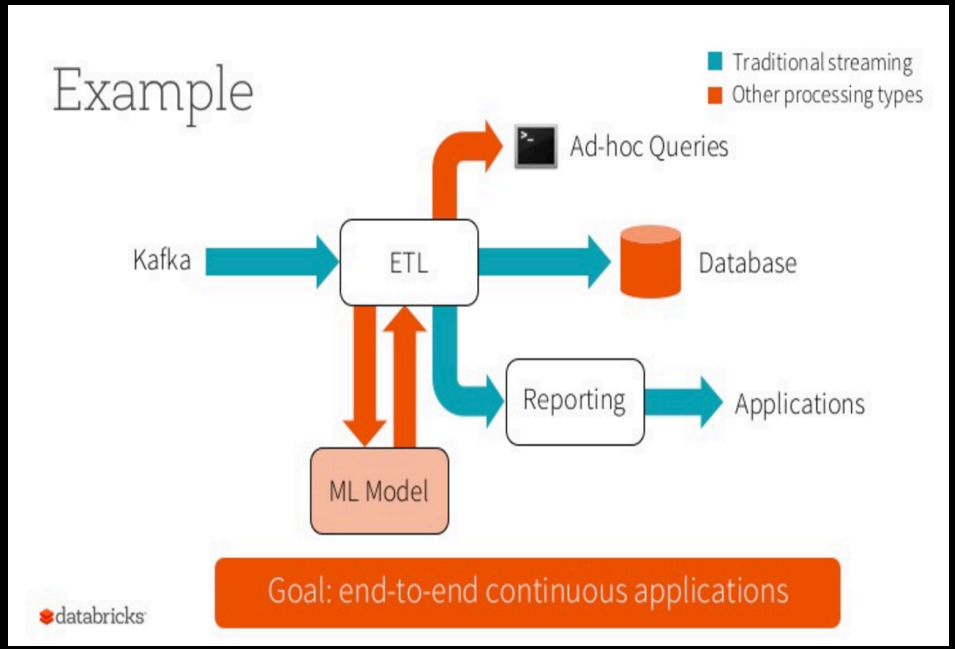
1. Simpler to use and operate

2. Code reuse: e.g. only write monitoring, FT, etc once

3. New apps that span processing types: e.g. interactive queries on a stream, online machine learning

Why Streaming Applications are Inherently Difficult?

building robust stream processing apps is hard



Complexities in stream processing

COMPLEX DATA

Diverse data formats
(json, avro, txt, csv, parquet, orc, binary, …)

Data can be dirty,
And tardy (out-of-order)

COMPLEX WORKLOADS

Combining streaming with interactive queries

Machine learning

COMPLEX SYSTEMS

Diverse storage systems
(Kafka, S3, Kinesis, RDBMS, …)

System failures

Structured Streaming

stream processing on Spark SQL engine

fast, scalable, fault-tolerant

rich, unified, high level APIs

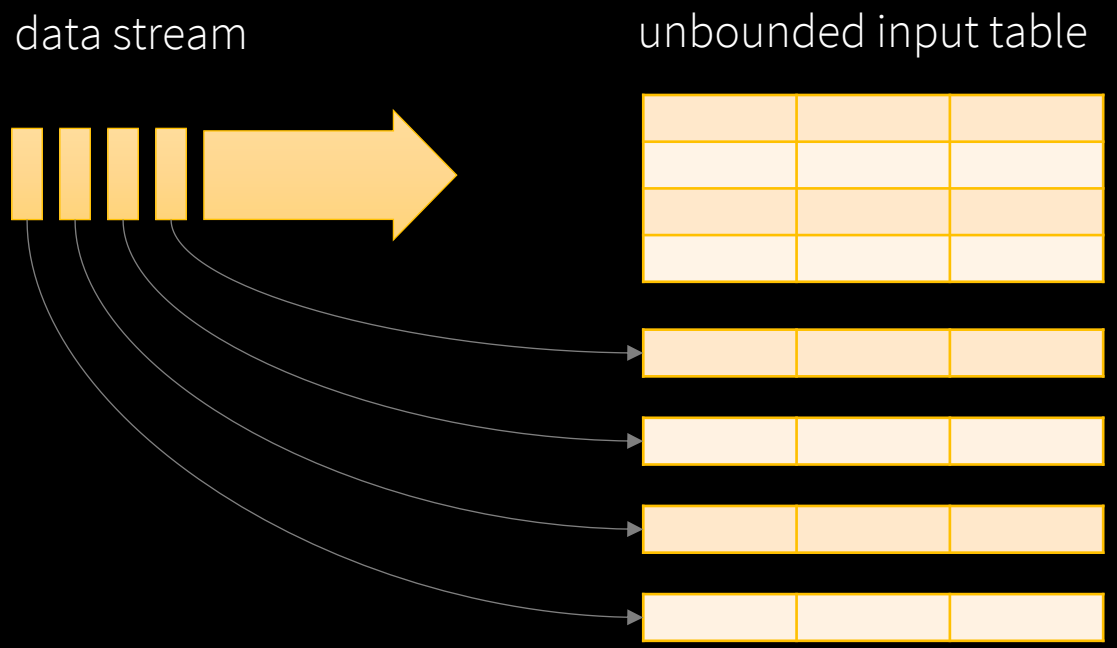
deal with *complex data* and *complex workloads*

rich ecosystem of data sources

integrate with many *storage systems*

you
should not have to
reason about streaming

Treat Streams as Unbounded Tables



new data in the
data stream
=
new rows appended
to a unbounded table

you

should write queries

&

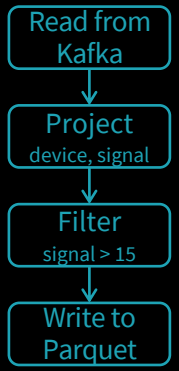
Apache Spark

should continuously update the answer

Apache Spark automatically *streamifies*!

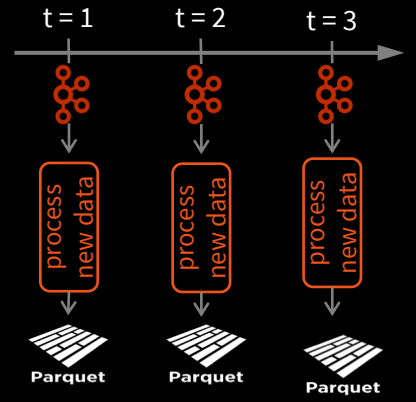
```
input = spark.readStream
  .format("kafka")
  .option("subscribe", "topic")
  .load()

result = input
  .select("device", "signal")
  .where("signal > 15")

result.writeStream
  .format("parquet")
  .start("dest-path")
```

DataFrames, Datasets, SQL



Logical Plan

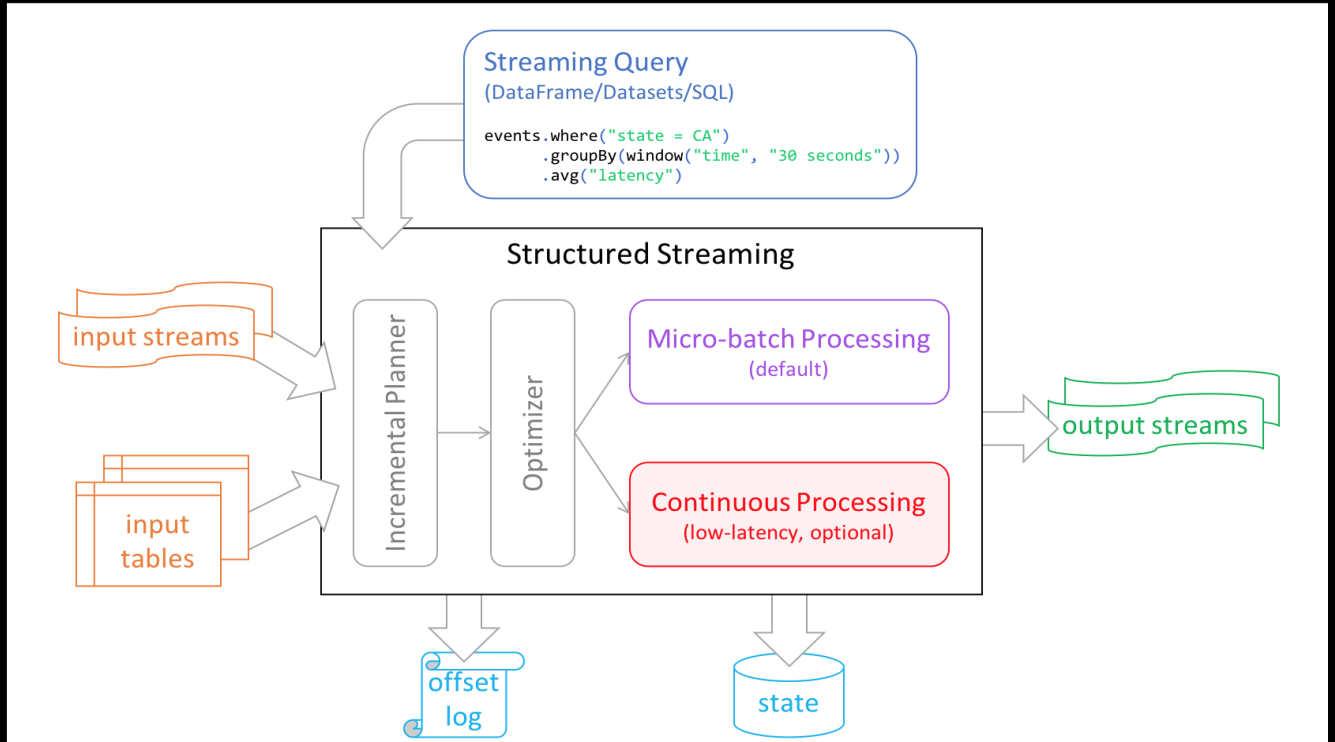
Optimized Physical Plan



Series of Incremental Execution Plans

Spark SQL converts batch-like query to a series of incremental execution plans operating on new batches of data

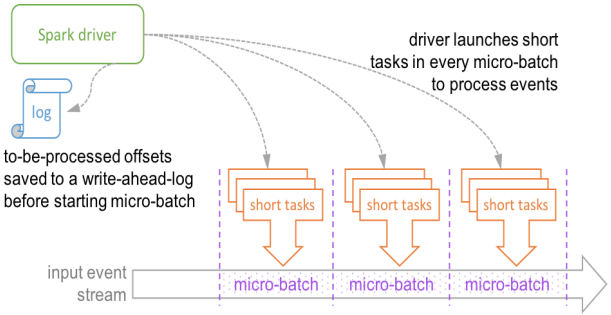
Structured Streaming – Processing Modes



Structured Streaming processing modes: Micro-batch and Continuous Processing

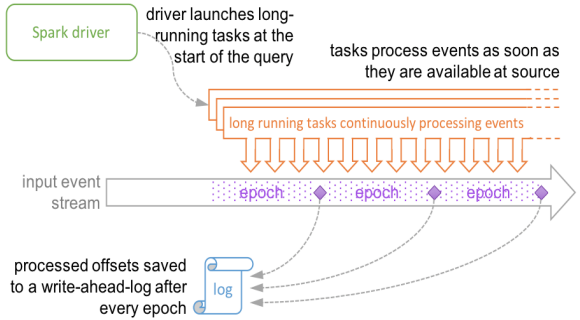
Structured Streaming Processing Modes

Batch Mode



Micro-batch Processing uses periodic tasks to process events

Continuous Mode



Continuous Processing uses long-running tasks to continuously process events

Anatomy of a Continunous Application

Anatomy of a Streaming Query

Streaming word count

Simple Streaming ETL

Anatomy of a Streaming Query: Step 1

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
```

Source

- Specify one or more locations to read data from
- Built in support for Files/Kafka/Socket, pluggable.

Anatomy of a Streaming Query: Step 2

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy(“value.cast("string") as key”)
  .agg(count("*") as “value”)
```

Transformation

- Using DataFrames, Datasets and/or SQL.
- Internal processing always exactly-once.

Anatomy of a Streaming Query: Step 3

```
spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy(“value.cast("string") as key”)
  .agg(count("*") as “value”)
  .writeStream()
  .format("kafka")
  .option("topic", "output")

  .outputMode(OutputMode.Complete())
  .option("checkpointLocation", "…")
  .start()
```

Sink

- Accepts the output of each batch.
- When supported sinks are transactional and exactly once (Files).

Anatomy of a Streaming Query: Output Modes

```
from pyspark.sql import Trigger

spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy(“value.cast("string") as key”)
  .agg(count("*") as 'value’)
  .writeStream()
  .format("kafka")
  .option("topic", "output")
  .trigger("1 minute")
  .outputMode("update")
  .option("checkpointLocation", "…")
  .start()
```

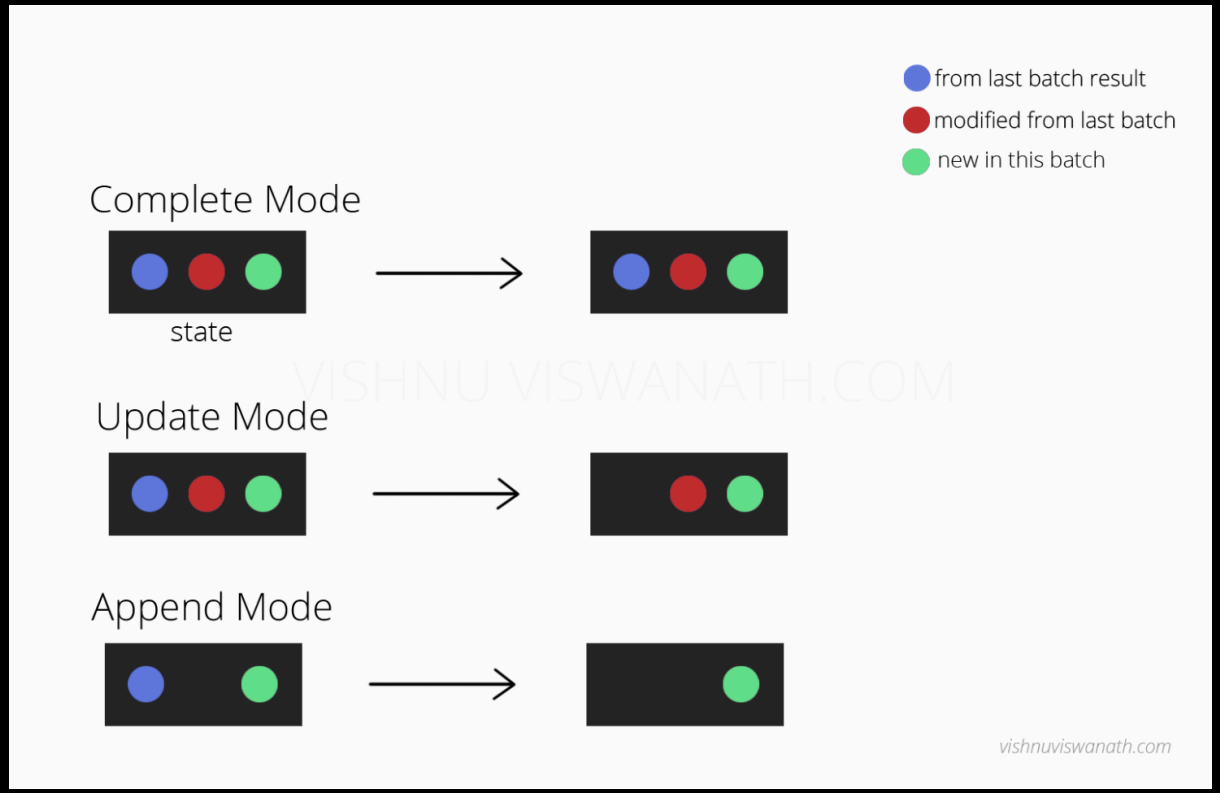
Output mode – What's output

- Complete – Output the whole answer every time
- Update – Output changed rows
- Append – Output new rows only

Trigger – When to output

- Specified as a time, eventually supports data size
- No trigger means as fast as possible

Streaming Query: Output Modes



Output mode – What's output

- Complete – Output the whole answer every time
- Update – Output changed rows
- Append – Output new rows only

Anatomy of a Streaming Query: Checkpoint

```
from pyspark.sql import Trigger

spark.readStream
  .format("kafka")
  .option("subscribe", "input")
  .load()
  .groupBy(“value.cast("string") as key”)
  .agg(count("*") as 'value)
  .writeStream()
  .format("kafka")
  .option("topic", "output")
  .trigger("1 minute")
  .outputMode("update")
  .option("checkpointLocation", "…")
  .withWatermark(“timestamp” “2 minutes”)
  .start()
```

Checkpoint & Watermark

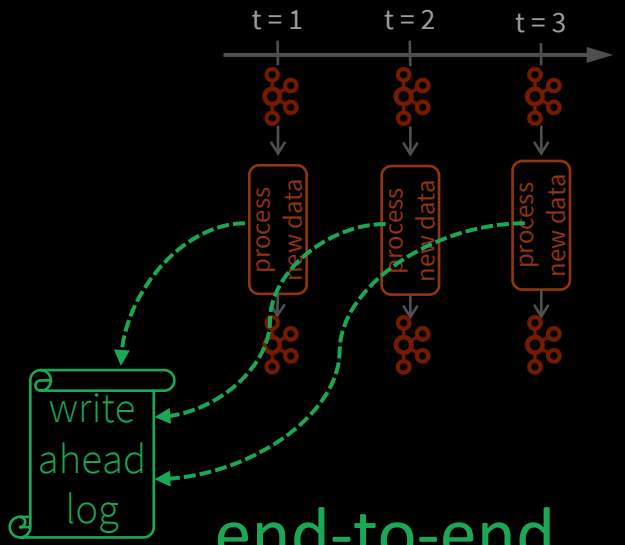
- Tracks the progress of a query in persistent storage
- Can be used to restart the query if there is a failure.
- *trigger(Trigger. Continunous(“ 1 second”))*

Set checkpoint location & watermark to drop very late events

Fault-tolerance with Checkpointing

Checkpointing – tracks progress (offsets) of consuming data from the source and intermediate state.

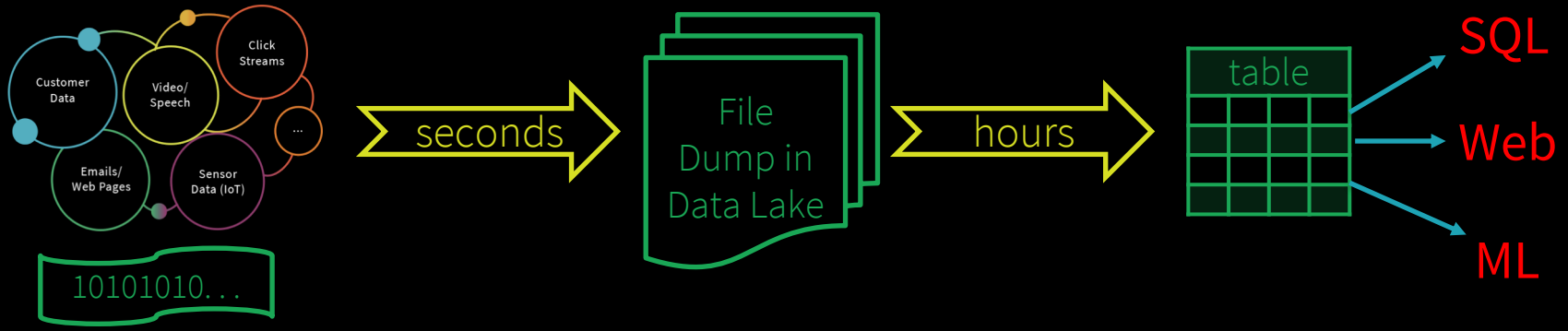
- Offsets and metadata saved as JSON
- Can resume after changing your streaming transformations



end-to-end exactly-once guarantees

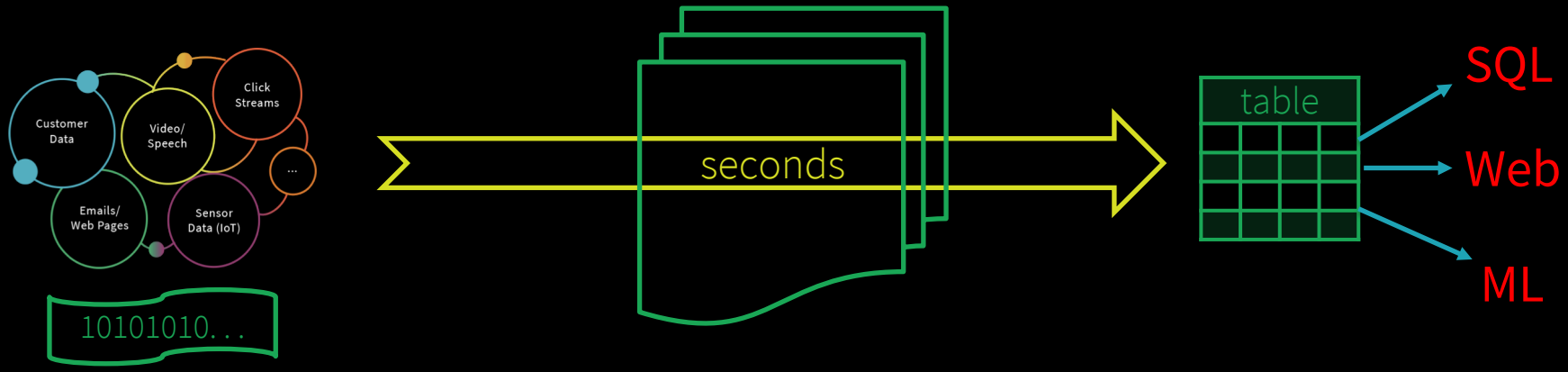
Complex Streaming ETL

Traditional ETL



- Raw, dirty, un/semi-structured is data dumped as files
- Periodic jobs run every few hours to convert raw data to structured data ready for further analytics
- Problem:
  - Hours of delay before taking decisions on latest data
  - Unacceptable when time is of essence
    - [intrusion , anomaly or fraud detection, monitoring IoT devices, etc.]

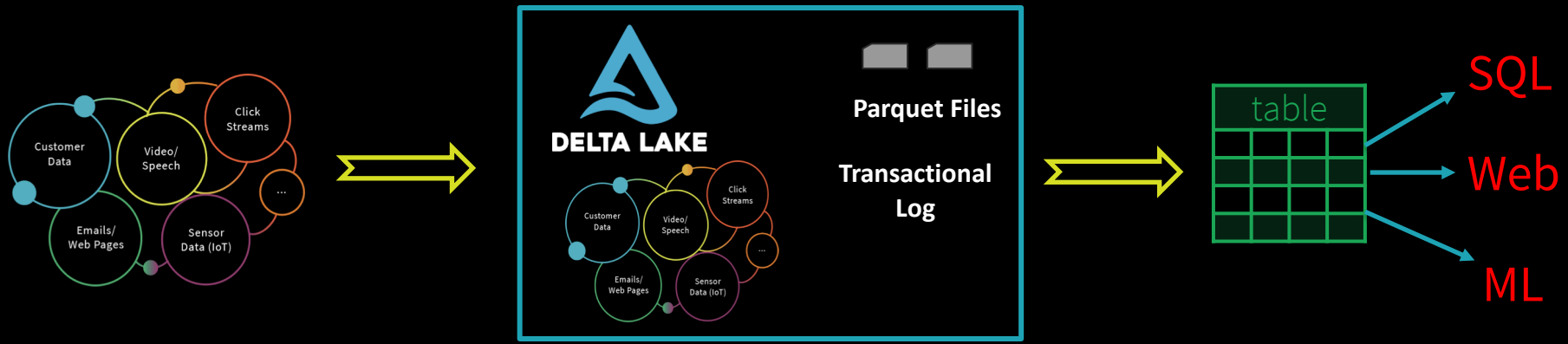
1. Streaming ETL w/ Structured Streaming



Structured Streaming Changes the Equation:

- eliminates latencies
- adds immediacy
- transforms data continuously

2. ETL w/ Structured Streaming & Delta Lake



Delta Lake ensures data reliability

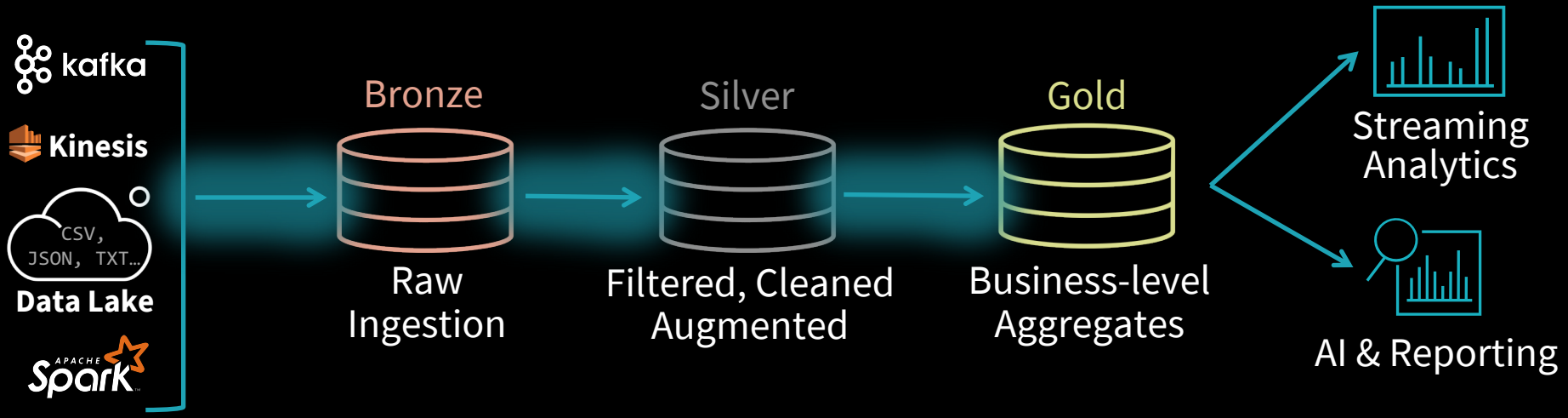
High Quality & Reliable Data
always ready for analytics

https://delta.io

Key Features

- ACID Transactions
- Schema Enforcement
- Unified Batch & Streaming
- Time Travel/Data Snapshots

The DELTA LAKE



Streams move data through the Delta Lake

- Low-latency or manually triggered
- Eliminates management of schedules and jobs

Streaming ETL w/ Structured Streaming

Example

1. Json data being received in Kafka
2. Parse nested json and flatten it
3. Store in structured Parquet table
4. Get end-to-end failure guarantees

```
from pyspark.sql import Trigger

rawData = spark.readStream
  .format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()

parsedData = rawData
  .selectExpr("cast (value as string) as json"))
  .select(from_json("json", schema).as("data"))
  .select("data.*") # do your ETL/Transformation

query = parsedData.writeStream
  .option("checkpointLocation", "/checkpoint")
  .partitionBy("date")
  .format("parquet")                    →   .format(”delta")
  .trigger(processingTime='5 seconds')
  .start("/parquetTable")               →   .start("/deltaTable”)
```

Reading from Kafka

rawData dataframe has the following columns

```
raw_data_df = spark.readStream
  .format("kafka")
  .option("kafka.boostrap.servers",...)
  .option("subscribe", "topic")
  .load()
```

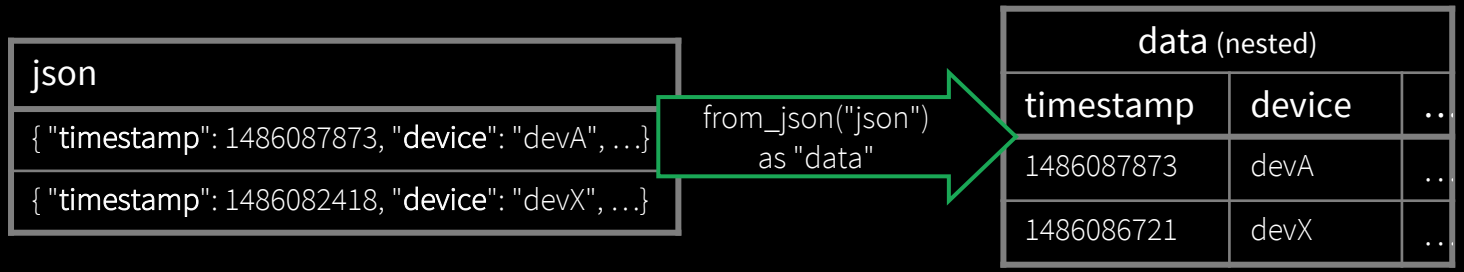
key	value	topic	partition	offset	timestamp
[binary]	*[binary]*	"topicA"	0	345	1486087873
[binary]	*[binary]*	"topicB"	3	2890	1486086721

Transforming Data

Cast binary *value* to string
Name it column *json*

```
parsedData = rawData
      .selectExpr("cast (value as string) as json")
      .select(from_json("json", schema).as("data"))
      .select("data.*")
```

Parse *json* string and expand into nested columns, name it *data*



Transforming Data

Cast binary *value* to string
Name it column *json*

Parse *json* string and expand into nested columns, name it data

Flatten the nested columns

```
parsedData = rawData
      .selectExpr("cast (value as string) as json")
      .select(from_json("json", schema).as("data"))
      .select("data.*")
```

powerful built-in Python APIs to perform complex data transformations

from_json, to_json, explode, ...
100s of functions

(see our blog post & tutorial)

Writing to  

Save parsed data as Parquet table or Delta Lake table in the given path

Partition files by date so that future queries on time slices of data is fast

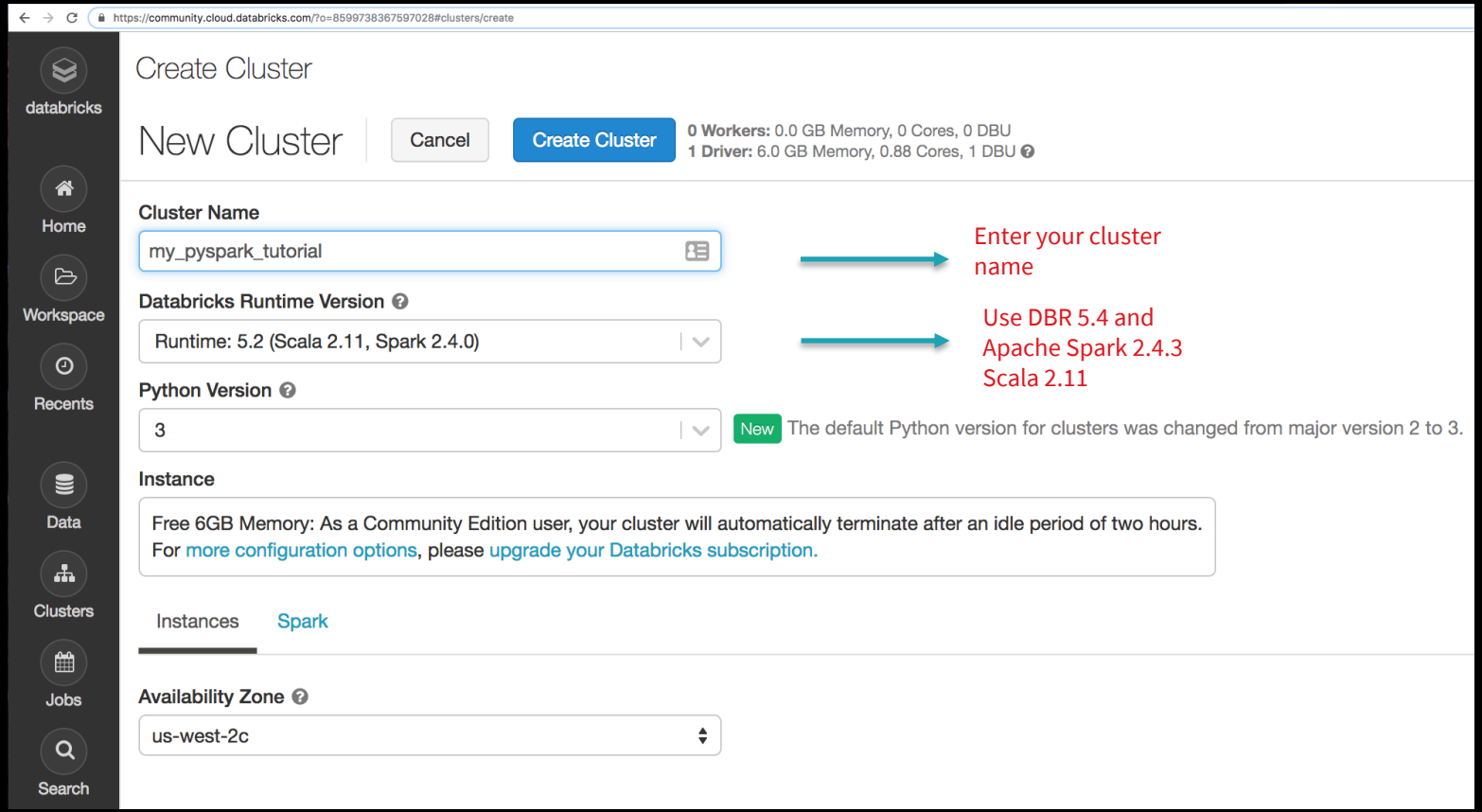
e.g. query on last 48 hours of data

```
queryP = parsedData.writeStream
     .option("checkpointLocation", ...)
     .partitionBy("date")
     .format("parquet")
     .start("/parquetTable") #pathname

queryD = parsedData.writeStream
     .option("checkpointLocation", ...)
     .partitionBy("date")
     .format("delta")
     .start("/deltaTable") #pathname
```

Tutorials

databricks

https://dbricks.co/pybaysf



Summary

- Apache Spark best suited for unified analytics & processing at scale
- Structured Streaming APIs Enables Continunous Applications
  - Populate in Parquet tables or Delta Lake
- Demonstrated Continunous Application

Resources

- Getting Started Guide with Apache Spark on Databricks
- docs.databricks.com
- Spark Programming Guide
- Structured Streaming Programming Guide
- Anthology of Technical Assets for Structured Streaming
- Databricks Engineering Blogs
- https://databricks.com/training/instructor-led-training
- https://delta.io

Thank You ☺

jules@databricks.com
@2twitme
https://www.linkedin.com/in/dmatrix/

Appendix

databricks

Jump Start on Apache® Spark™ 2.2 with Databricks

Jules S. Damji
Apache Spark Community Evangelist

Spark Saturday DC , Nov 11, 2017
Big Data Madison @ Madison College

I know the difference between DataFrame and RDDs…

Why Apache Spark?

Big Data Systems of Yesterday…

MapReduce/Hadoop

General batch processing

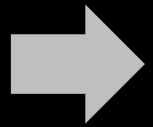
Pregel
Giraph
Dremel
Mahout
Storm
Impala
Drill
…

Specialized systems for new workloads

Hard to *combine* in pipelines

Big Data Systems Today

MapReduce

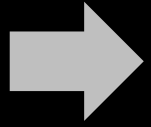


Pregel Giraph

Dremel Millwheel

Storm Impala

Drill S4 . . .



General batch processing

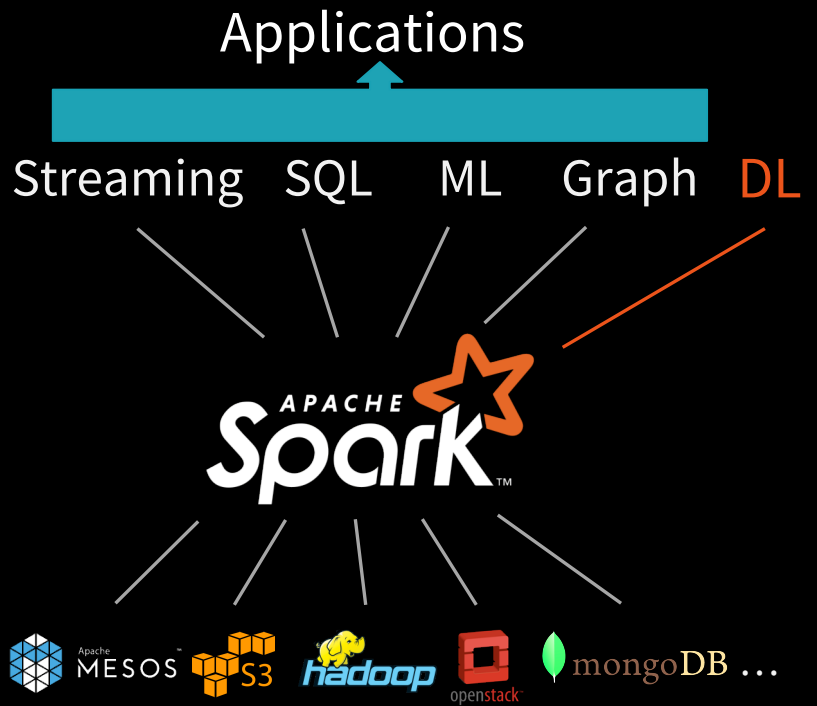
Specialized systems for new workloads

Unified engine

Apache Spark Philosophy

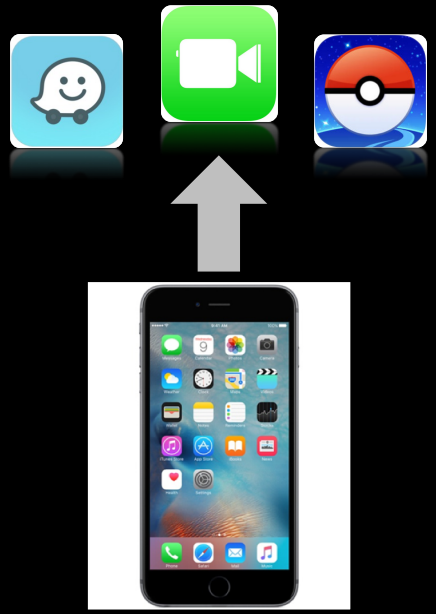
Unified engine for complete data applications

High-level user-friendly APIs



An Analogy ….

New applications



Unified engine across diverse workloads & environments

CONTRIBUTED ARTICLES

Apache Spark: A Unified Engine for Big Data Processing

By Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, Ion Stoica

Communications of the ACM, Vol. 59 No. 11, Pages 56-65

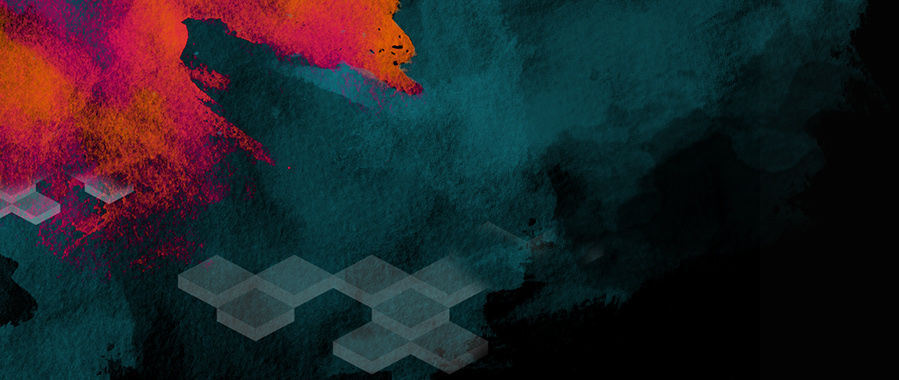
10.1145/2934664

Comments

SIGN IN for Full Access

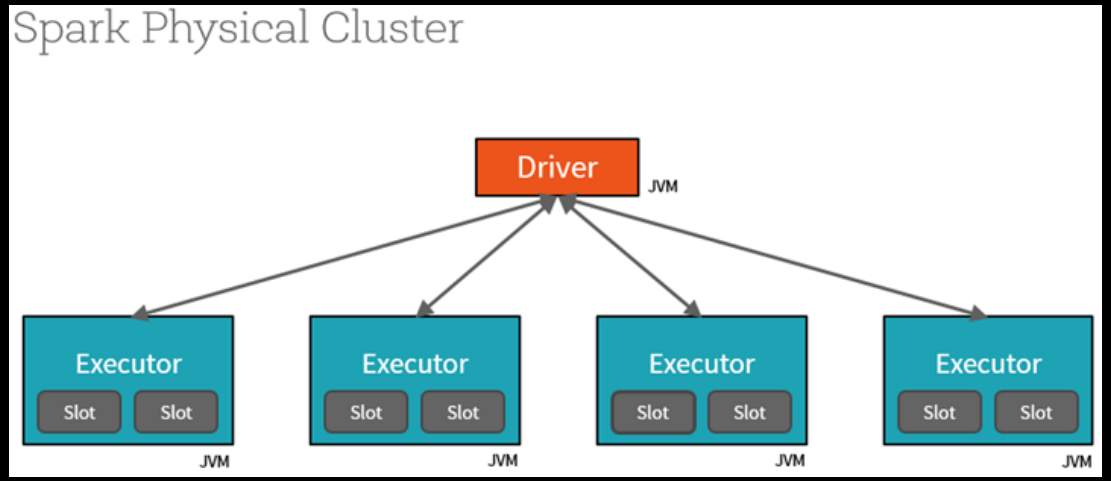
User Name

Apache Spark Architecture



databricks

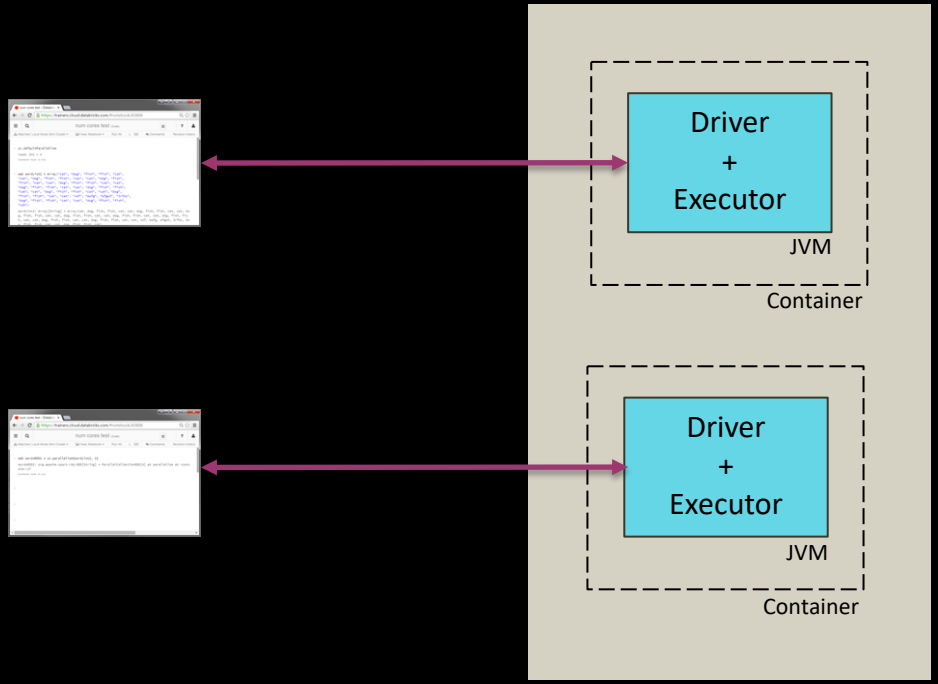
Apache Spark Architecture



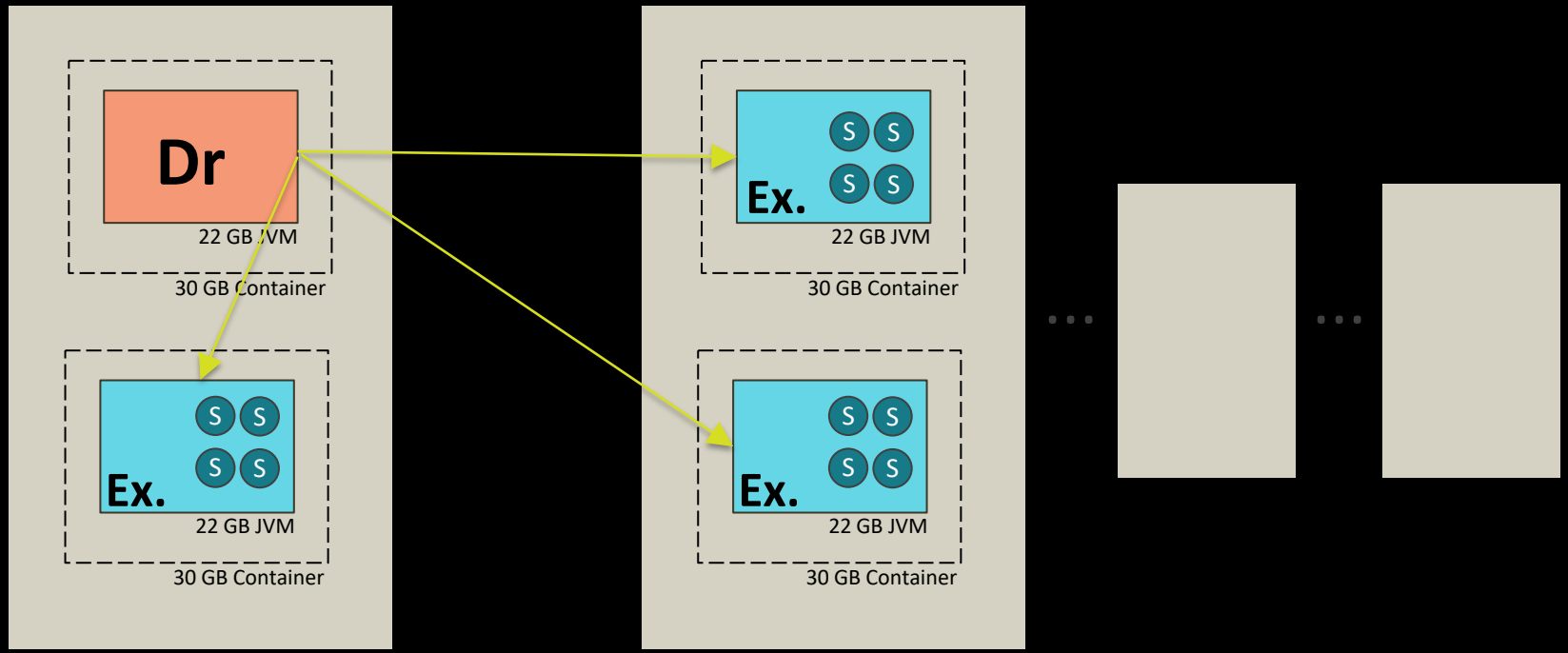
Deployments Modes

- Local
- Standalone
- YARN
- Mesos

Local Mode in Databricks



Standalone Mode



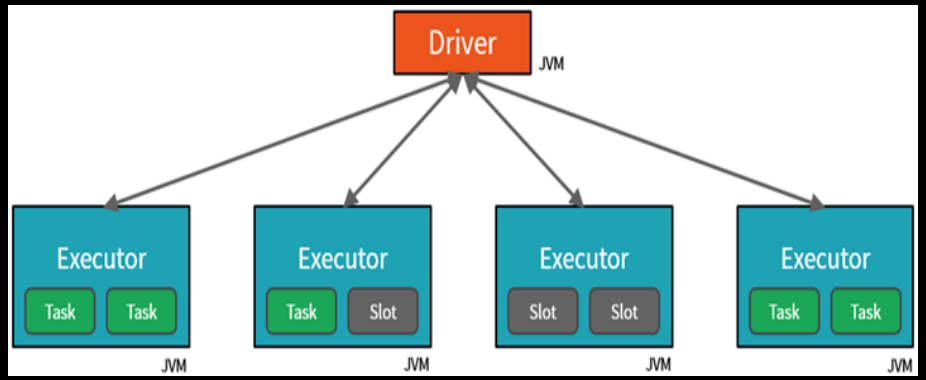
Spark Deployment Modes

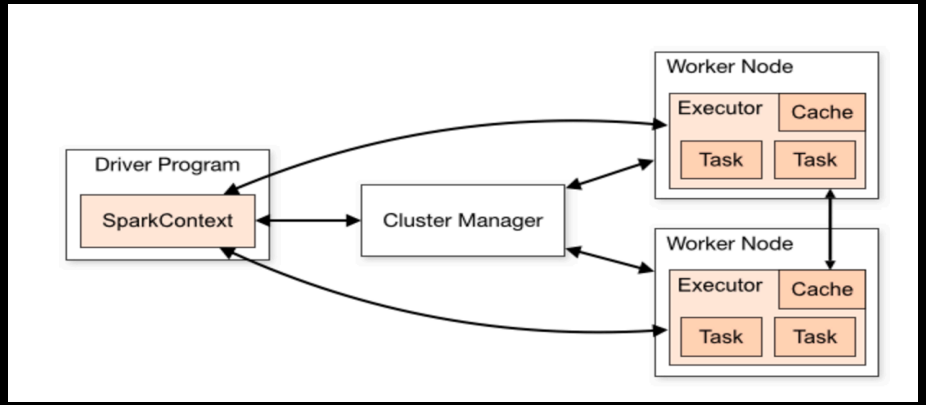
MODE	DRIVER	WORKER	EXECUTOR	MASTER
Local	Runs on a single JVM	Runs on the same JVM as the driver	Runs on the same JVM as the driver	Runs on a single host
Standalone	Can run on any node in the cluster	Runs on its own JVM on each node	Each worker in the cluster will launch its own JVM	Can be allocated arbitrarily where the master is started
Yarn (client)	On a client, not part of the cluster	YARN NodeManager	YARN's NodeManager's Container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for Executors.
YARN (cluster)	Runs within the YARN's Application Master	Same as YARN client mode	Same as YARN client mode	Same as YARN client mode
Mesos (client)	Runs on a client machine, not part of Mesos cluster	Runs on Mesos Slave	Container within Mesos Slave	Mesos' master
Mesos (cluster)	Runs within one of Mesos' master	Same as client mode	Same as client mode	Mesos' master

Table 1. Cheat Sheet Depicting Deployment Modes And Where Each Spark Component Runs

Apache Spark Architecture

An Anatomy of an Application

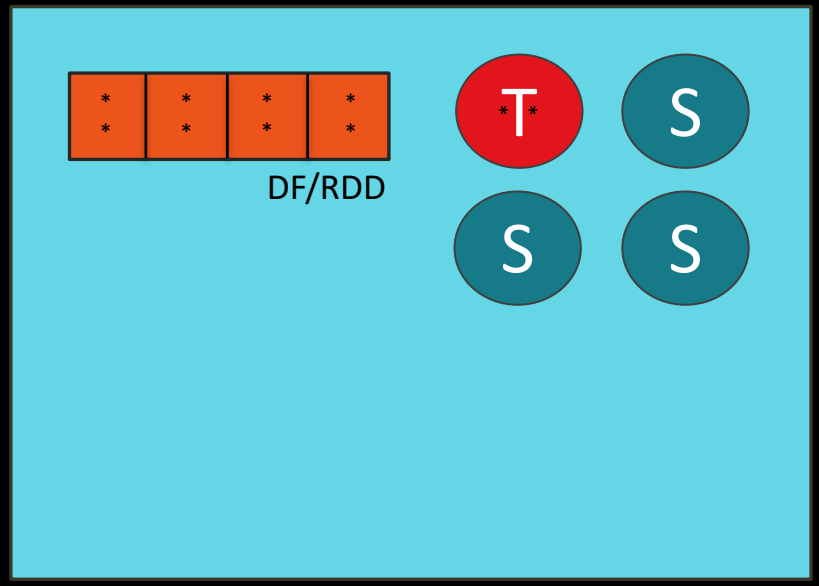




Spark Application

- Jobs
- Stages
- Tasks

A Spark Executor

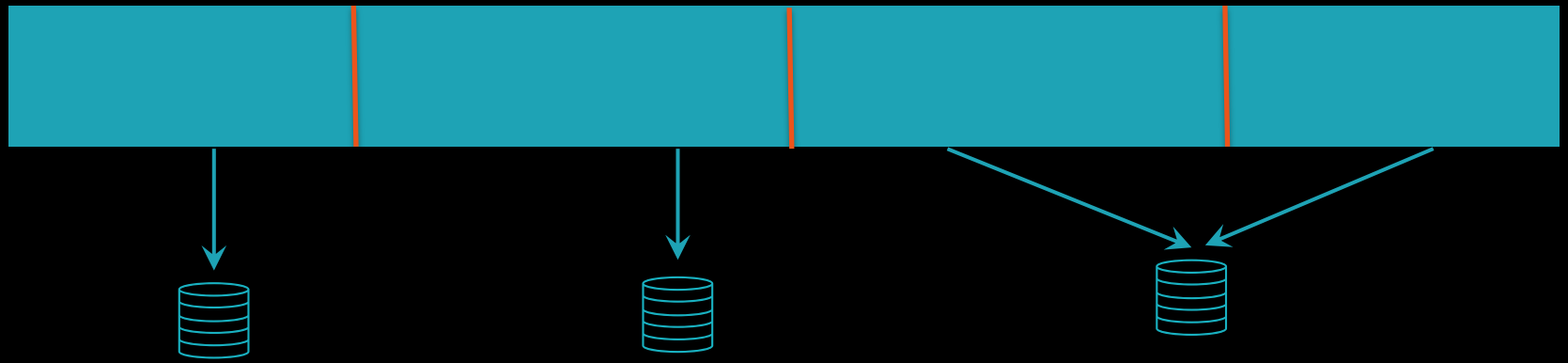


Resilient Distributed Dataset (RDD)

What are RDDs?

1. Distributed Data Abstraction

Logical Model Across Distributed Storage



S3 or HDFS

2. Resilient & Immutable

RDD → T→ RDD -> RDD

T = Transformation

3. Compile-time Type-safe

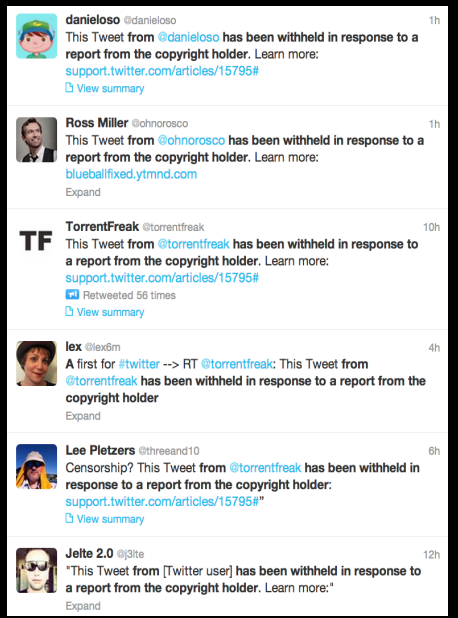
Integer RDD

String or Text RDD

Double or Binary RDD

4. Unstructured/Structured Data: Text (logs, tweets, articles, social)

```
jkreps-mn:~ jkreps$ tail -f -n 20 /var/log/apache2/access_log
::1 - - [23/Mar/2014:15:07:00 -0700] "GET /images/apache_feather.gif HTTP/1.1" 200 4128
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/producer_consumer.png HTTP/1.1" 200 86
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/log_anatomy.png HTTP/1.1" 200 19579
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/consumer-groups.png HTTP/1.1" 200 2682
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/log_compaction.png HTTP/1.1" 200 41414
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /documentation.html HTTP/1.1" 200 189893
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/log_cleaner_anatomy.png HTTP/1.1" 200
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/kafka_log.png HTTP/1.1" 200 134321
::1 - - [23/Mar/2014:15:07:04 -0700] "GET /images/mirror-maker.png HTTP/1.1" 200 17054
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /documentation.html HTTP/1.1" 200 189937
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /styles.css HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/kafka_logo.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/producer_consumer.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/log_anatomy.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/consumer-groups.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/log_cleaner_anatomy.png HTTP/1.1" 304
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/log_compaction.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/kafka_log.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:08:07 -0700] "GET /images/mirror-maker.png HTTP/1.1" 304 -
::1 - - [23/Mar/2014:15:09:55 -0700] "GET /documentation.html HTTP/1.1" 200 195264
```



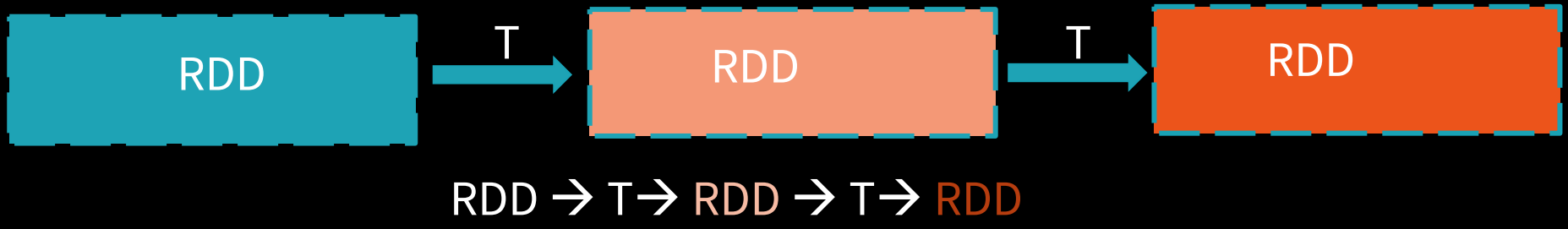
Structured Tabular data..

```
> %sql select ipaddress, datetime, method, endpoint, protocol, responsecode, agent from accesslog limit 10;
```

▸ (1) Spark Jobs

ipaddress	datetime	method	endpoint	protocol	responsecode	agent
10.223.144.123	04/Nov/2015:08:15:00 +0000	GET	/company/contact	HTTP/1.1	200	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_ AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.80 Safari/537.36
10.223.144.123	04/Nov/2015:08:15:37 +0000	GET	/blog	HTTP/1.1	200	Mozilla/5.0 (Windows NT 6.3; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0
10.223.96.51	04/Nov/2015:08:16:38 +0000	GET	/pantheon_healthcheck	HTTP/1.1	200	Pingdom.com_bot_version_1.4_(http://www.pi
10.223.144.123	04/Nov/2015:08:18:15 +0000	GET	/blog/2014/04/14/spark-with-java-8.html	HTTP/1.1	200	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/46.0.2490.80 Safari/537.36

5. Lazy



RDD → T→ RDD → T→ RDD

T = Transformation

A = Action

TRANSFORMATIONS

Apache Spark Operations =

+

ACTIONS

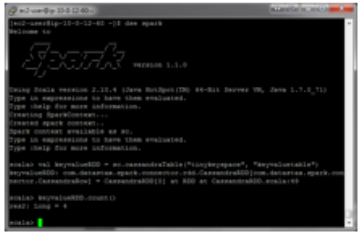
Transformations *(lazy)*	Actions
`orderBy`	`show`
`filter`	`count`
`groupBy`	`take`
`select`	`collect`
`drop`	`save`
`join`	

Transformations contribute to a query plan,
but nothing is executed until an action is called

distributed

occurs across the cluster

vs



driver

result must fit in driver JVM

 = easy  = medium

Essential Core & Intermediate Spark Operations

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

ACTIONS

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

- takeOrdered

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

 = easy  = medium

Essential Core & Intermediate Spark Operations

TRANSFORMATIONS

General

- flatMapValues
- groupByKey
- reduceByKey
- reduceByKeyLocally
- foldByKey
- aggregateByKey
- sortByKey
- combineByKey
- keys
- values

Math / Statistical

- sampleByKey

Set Theory / Relational

- cogroup (=groupWith)
- join
- subtractByKey
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

Data Structure / I/O

- partitionBy

ACTIONS

- countByKey
- countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact

Types of RDDs:

HadoopRDD: core functionality for reading data in HDFS using older MR API

MapPartitionsRDD: a result of calling actions like `map`, `flatMap`, `filter`, `mapPartitions`, etc

PairRDD: holds key/value pairs, a result of `groupByKey` and `join` operations

PipedRDD: result of piping elements to a forked external process

ShuffledRDD: a result after shuffling (`repartition` or `coalesce`)

7 Steps to Mastering Apache Spark 2.0

Sep 2016 Silver nuggets Blog

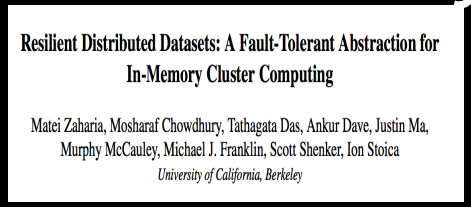
Looking for a comprehensive guide on going from zero to Apache Spark hero in steps? Look no further! Written by our friends at Databricks, this exclusive guide provides a solid foundation for those looking to master Apache Spark 2.0.

By Jules S. Damji & Sameer Farooqui, Databricks.

Not a week goes by without a mention of Apache Spark in a blog, news article, or webinar on Spark's impact in the big data landscape. Not a meetup or conference on big data or advanced analytics is without a speaker that expounds on aspects of Spark—touting of its rapid adoption; speaking of its developments; explaining of its uses cases, in enterprises across industries.

How did we get here…?
Where we going…?

A Brief History


Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica

University of California, Berkeley

2010

2012 — Started @ UC Berkeley

2013 — Databricks started & donated to ASF

Spark 1.0 & libraries (SQL, ML, GraphX)

2014

2015 — DataFrames/Datasets Tungsten Catalyst Optimizer ML Pipelines

Apache Spark 2.0, 2.1,2.2,2.3,2.4
Structured Streaming
Cost Based Optimizer
Deep Learning Pipelines

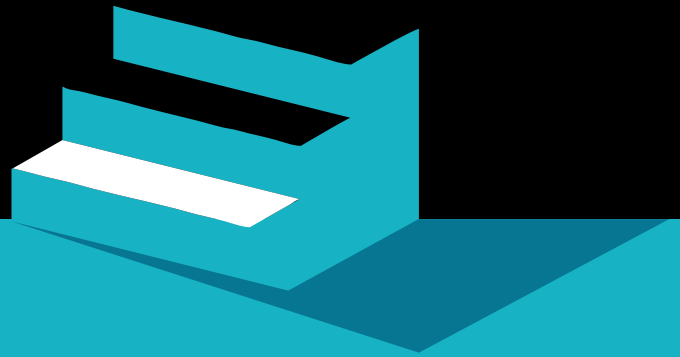
2016-19

Easier
Smarter
Faster

databricks

Apache Spark 2.X

- Steps to Bigger & Better Things….



Builds on all we learned in past 2 years

Major Themes in Apache Spark 2.x

Unifying Datasets and DataFrames & SparkSessions

Easier

Tungsten Phase 2 speedups of 5-10x & Catalyst Optimizer

Faster

Structured Streaming real-time engine on SQL / DataFrames

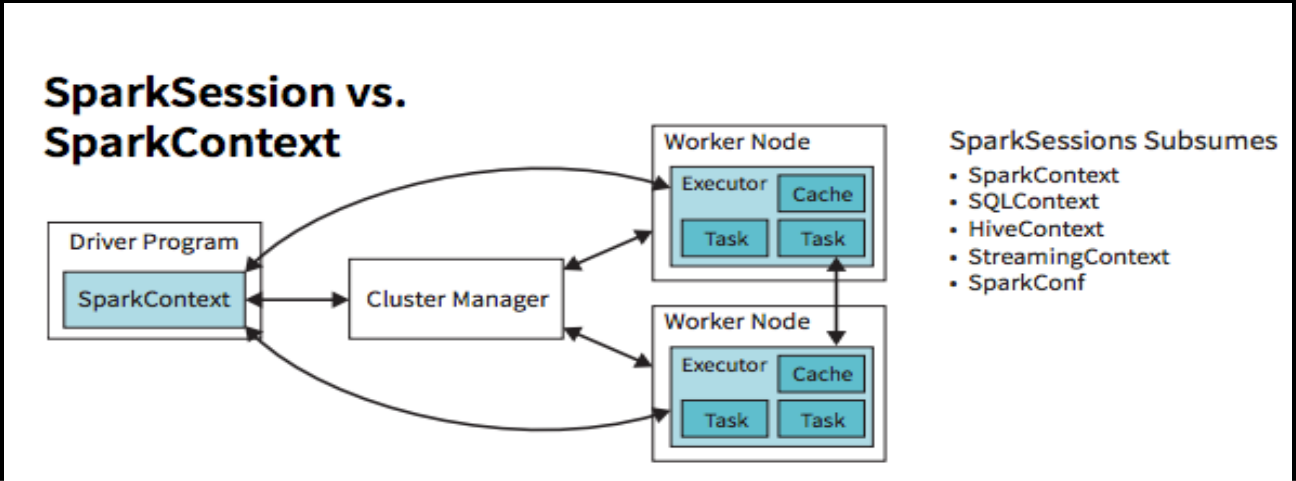
Smarter

Unified API Foundation for the Future: SparkSessions, DataFrame, Dataset, MLlib, Structured Streaming

SparkSession – A Unified entry point to Spark

- Conduit to Spark
  - Creates Datasets/DataFrames
  - Reads/writes data
  - Works with metadata
  - Sets/gets Spark Configuration
  - Driver uses for Cluster resource management

SparkSession vs SparkContext




```
val warehouseLocation = "file:${system:user.dir}/spark-warehouse"

val spark = SparkSession
 .builder()
 .appName("SparkSessionZipsExample")
 .config("spark.sql.warehouse.dir", warehouseLocation)
 .enableHiveSupport()
 .getOrCreate()
```

SparkSession – A Unified entry point to Spark

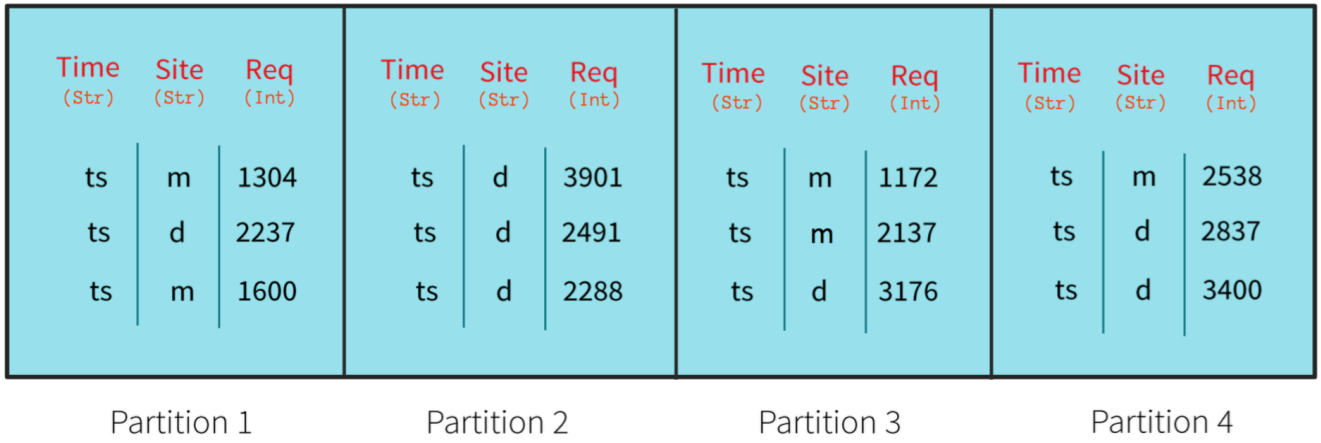
How to use SparkSession in Apache Spark 2.0

A unified entry point for manipulating data with Spark

by Jules Damji
Posted in **ENGINEERING BLOG** | August 15, 2016

Try this notebook in Databricks

DataFrame & Dataset Structure

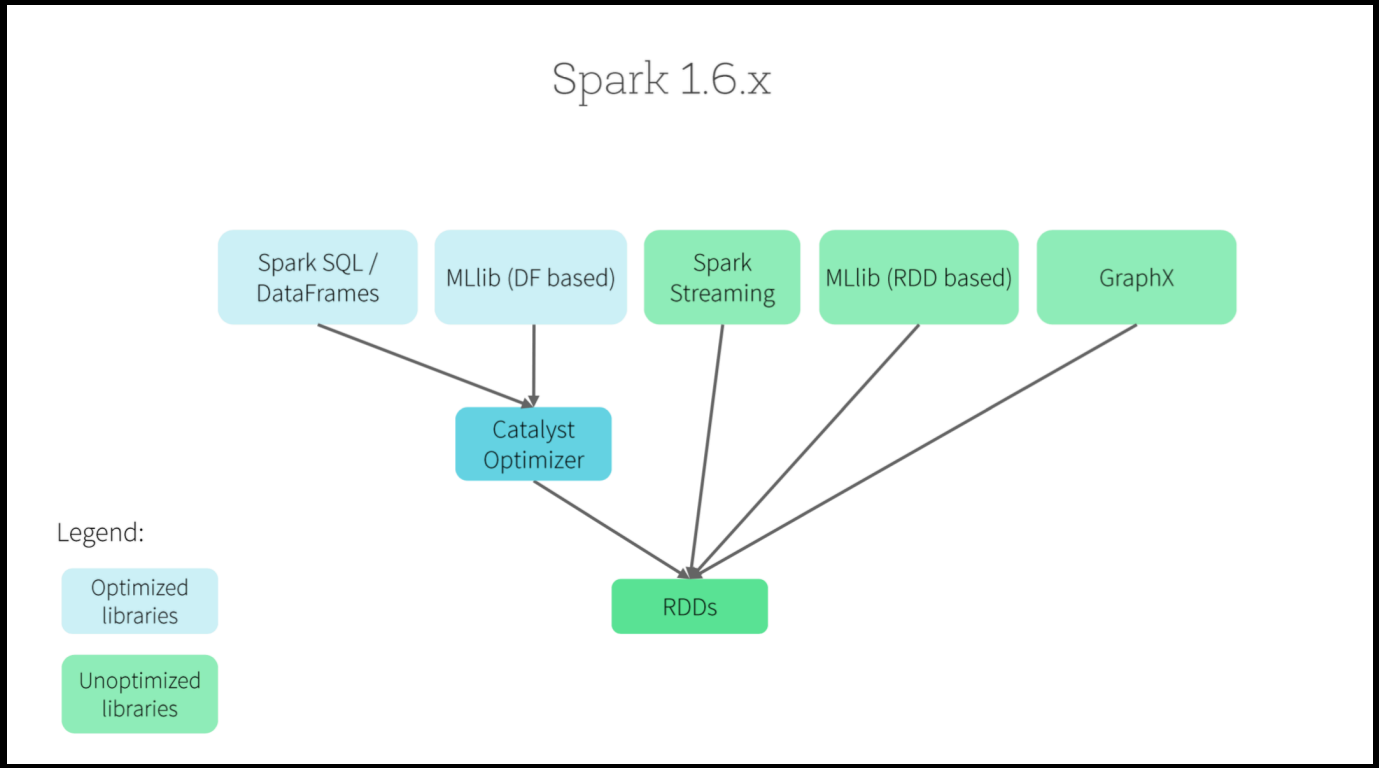



```
df.rdd.partitions.size  =  4
```

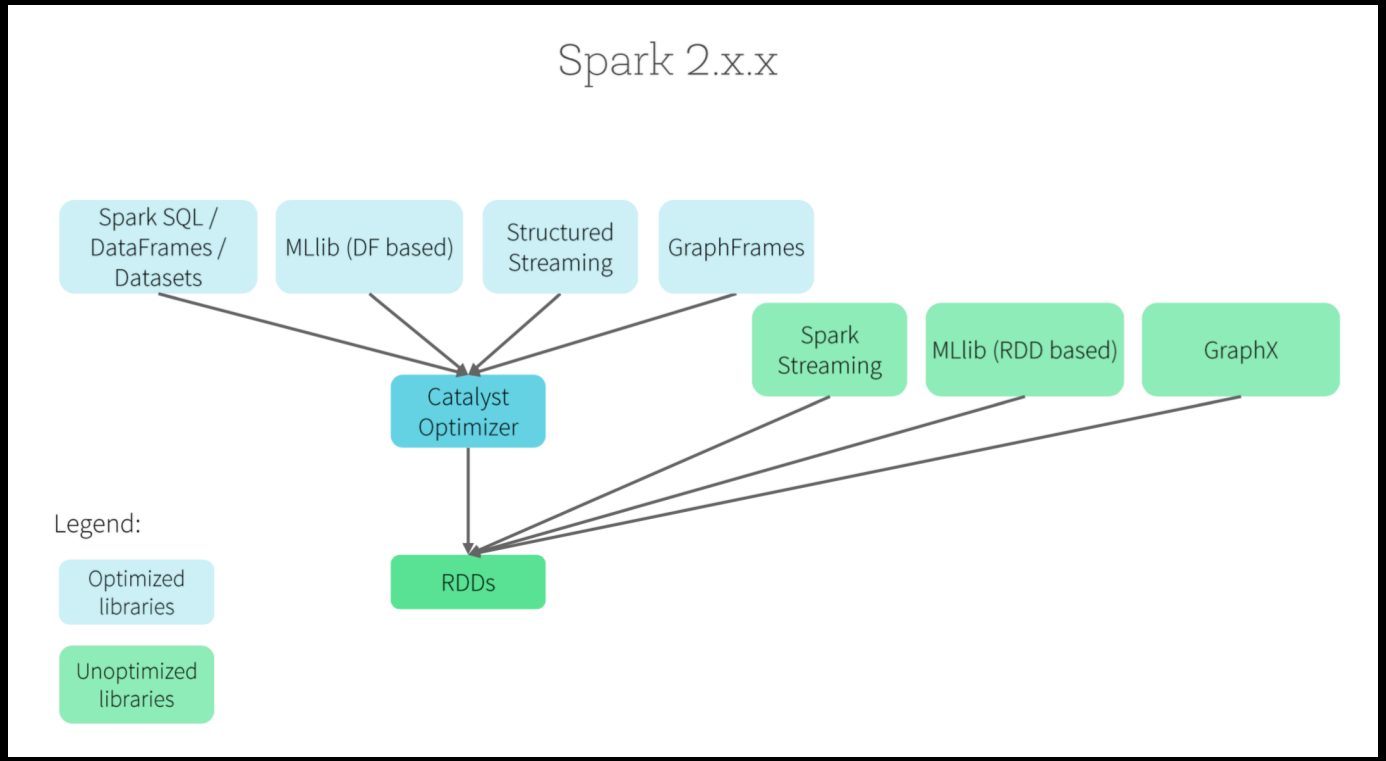
Long Term

- RDD as the low-level API in Spark
  - For control and certain type-safety in Java/Scala
- Datasets & DataFrames give richer semantics & optimizations
  - For semi-structured data and DSL like operations
  - New libraries will increasingly use these as interchange format
  - Examples: Structured Streaming, MLlib, GraphFrames, and Deep Learning Pipelines

Spark 1.6 vs Spark 2.x



Spark 1.6 vs Spark 2.x



Other notable API improvements

- DataFrame-based ML pipeline API becoming the main MLlib API
- ML model & pipeline persistence with almost complete coverage
  - In all programming languages: Scala, Java, Python, R
- Improved R support
  - (Parallelizable) User-defined functions in R
  - Generalized Linear Models (GLMs), Naïve Bayes, Survival Regression, K-Means
- Structured Streaming Features & Production Readiness
- https://databricks.com/blog/2017/07/11/introducing-apache-spark-2-2.html

DataFrames/Dataset, Spark SQL & Catalyst Optimizer

The not so secret truth…

SQL

is not about SQL

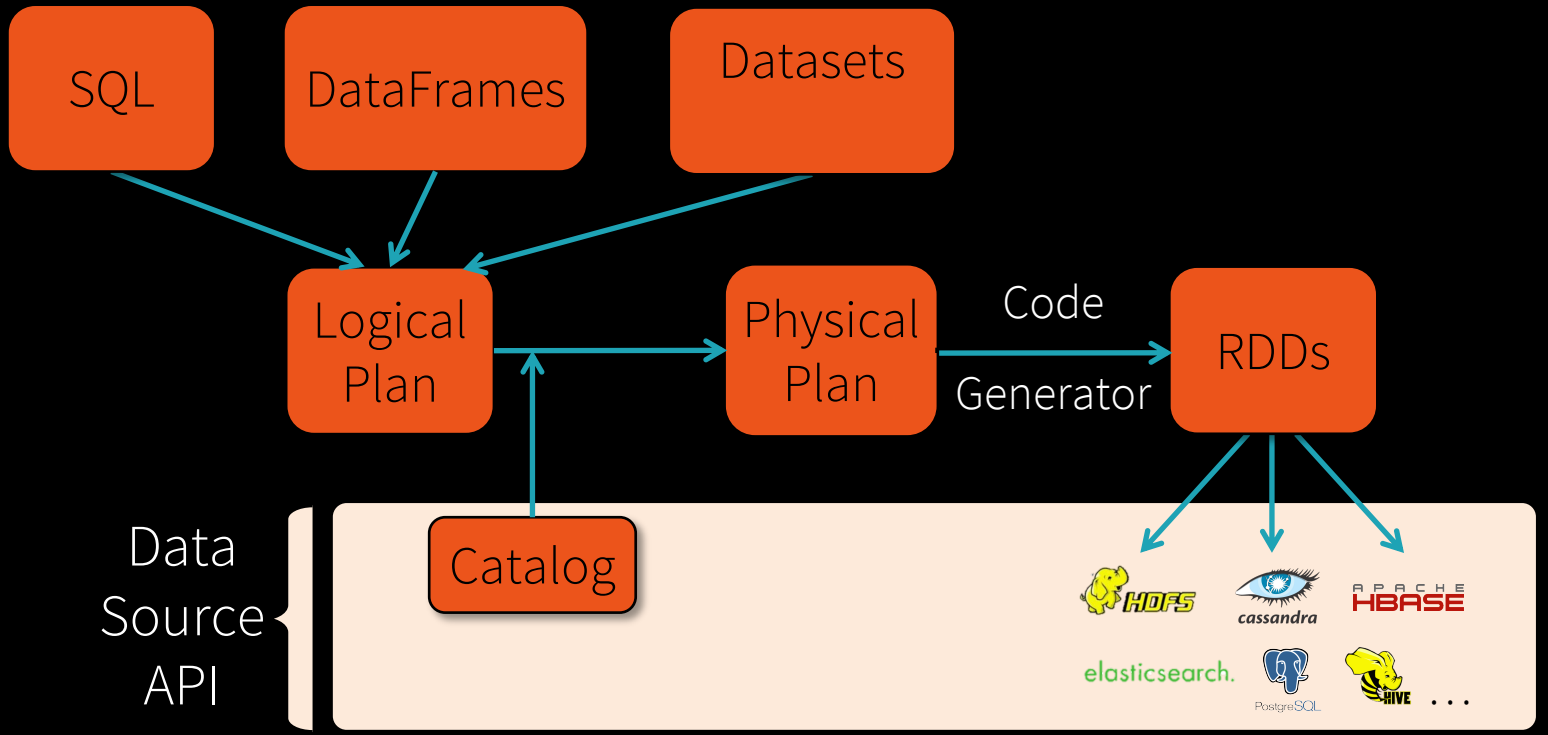
is about more than SQL

Spark SQL: The whole story

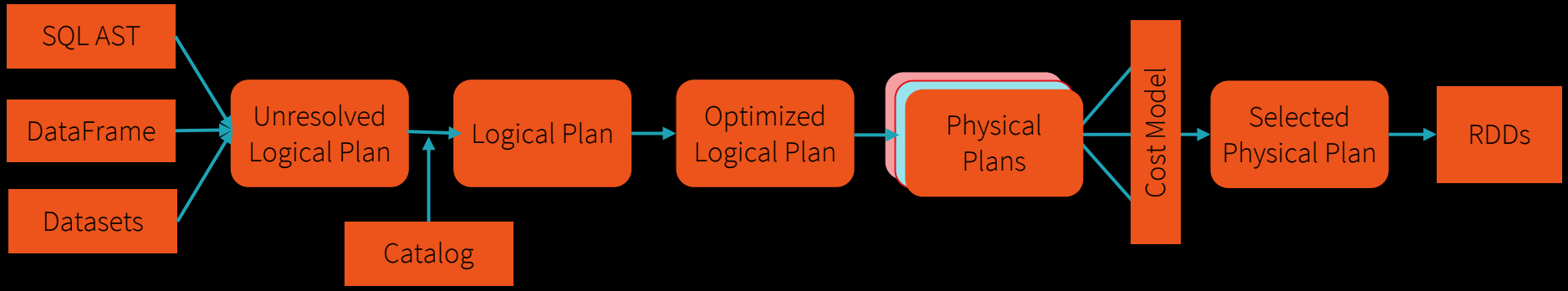
Is About Creating and Running Spark Programs Faster:

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

Spark SQL Architecture



Using Catalyst in Spark SQL



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 Optimizations

LOGICAL OPTIMIZATIONS

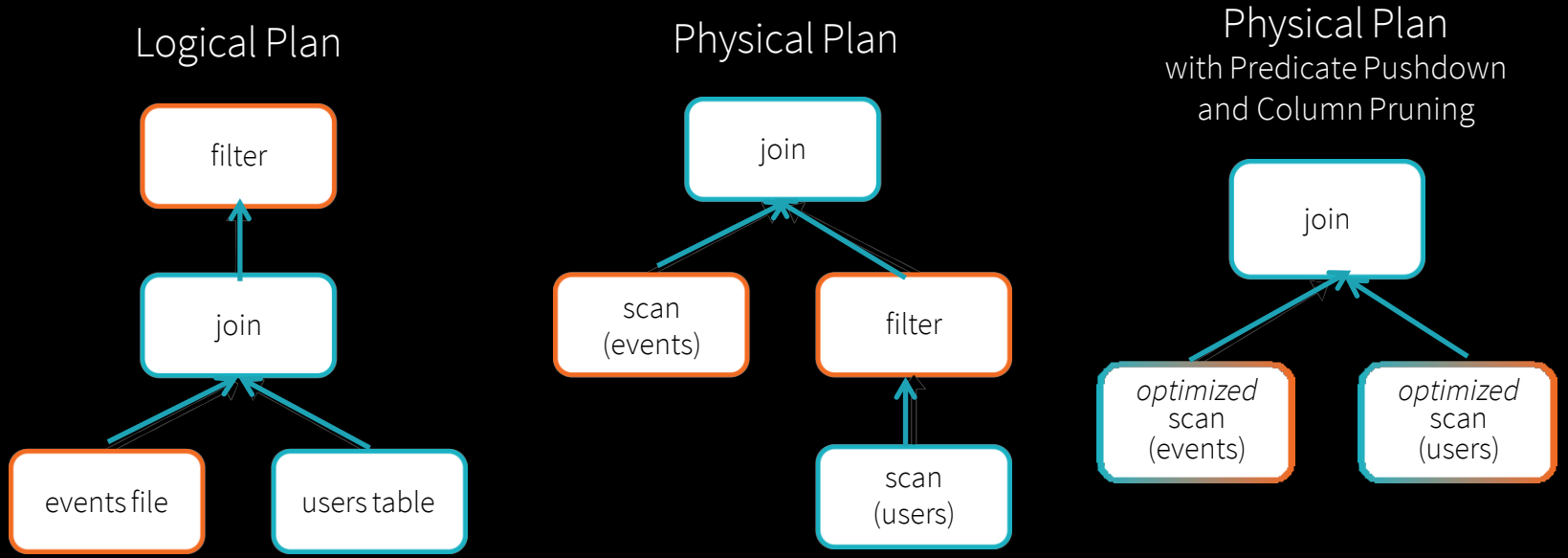
- Push filter predicate down to data source, so irrelevant data can be skipped
- **Parquet:** skip entire blocks, turn comparisons into cheaper integer comparisons via dictionary coding
- RDMS: reduce amount of data traffic by pushing down predicates

PHYSICAL OPTIMIZATIONS

- Catalyst compiles operations into physical plan for execution and generates JVM byte code
- Intelligently choose between broadcast joins and shuffle joins to reduce network traffic
- **Lower level optimizations:** eliminate expensive object allocations and reduce virtual functions calls

DataFrame Optimization

```
users.join(events, users("id") === events("uid")) .

  filter(events("date") > "2015-01-01")
```



Columns: Predicate pushdown

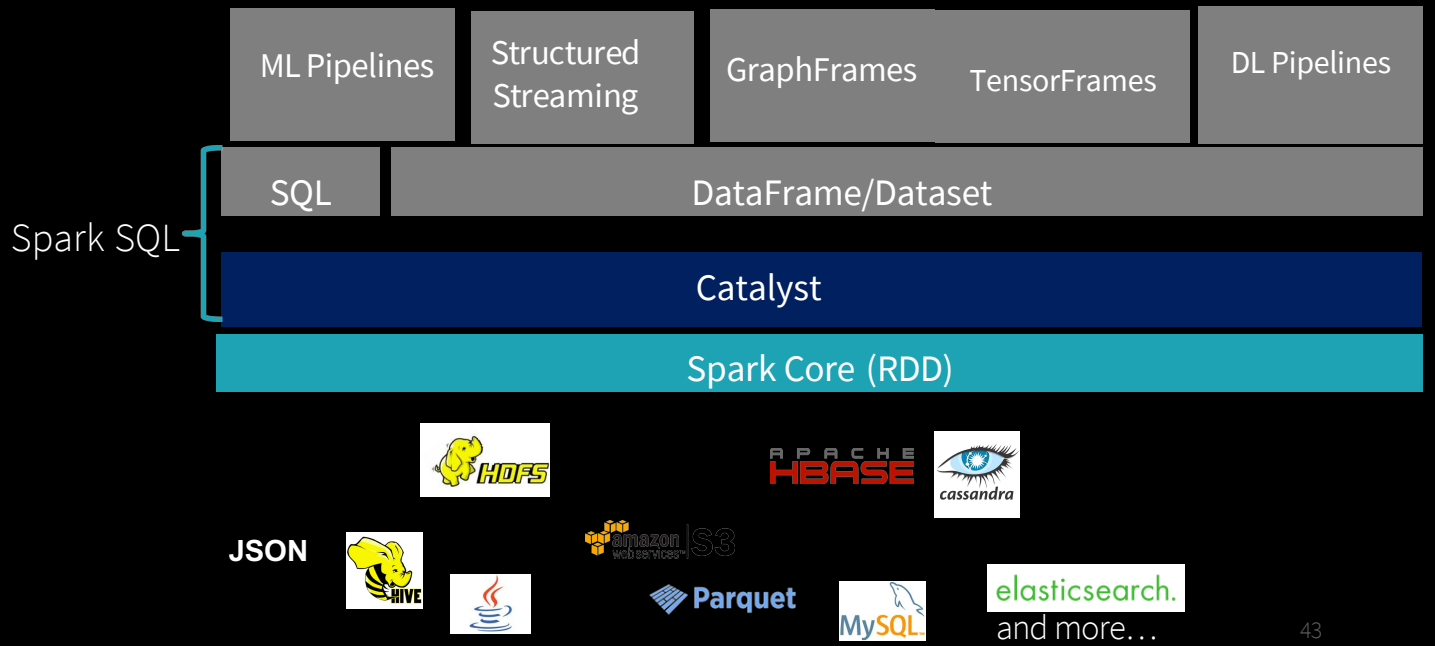
You Write

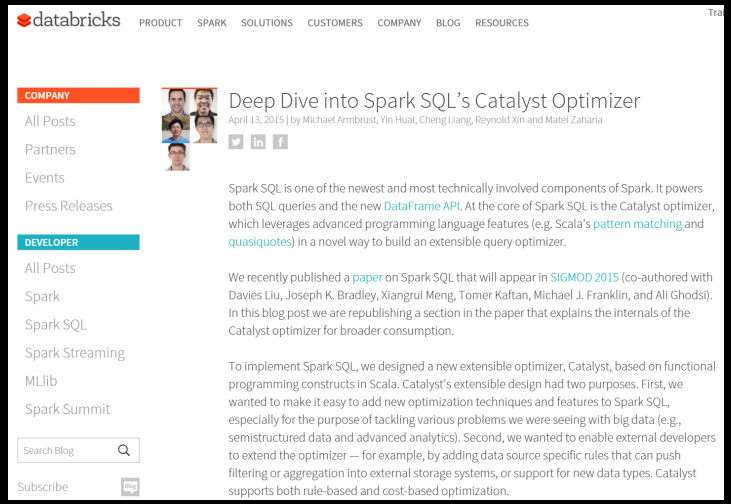
```
spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "people")
  .load()
  .where($"name" === "michael")
```

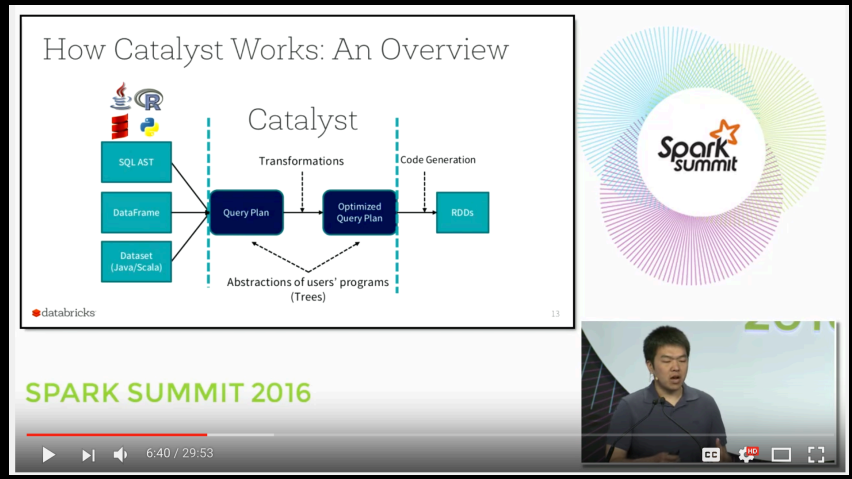
Spark Translates For Postgres

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

Foundational Spark 2.x Components







http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf

DataFrames & Datasets
Spark 2.x APIs

Background: What is in an RDD?

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

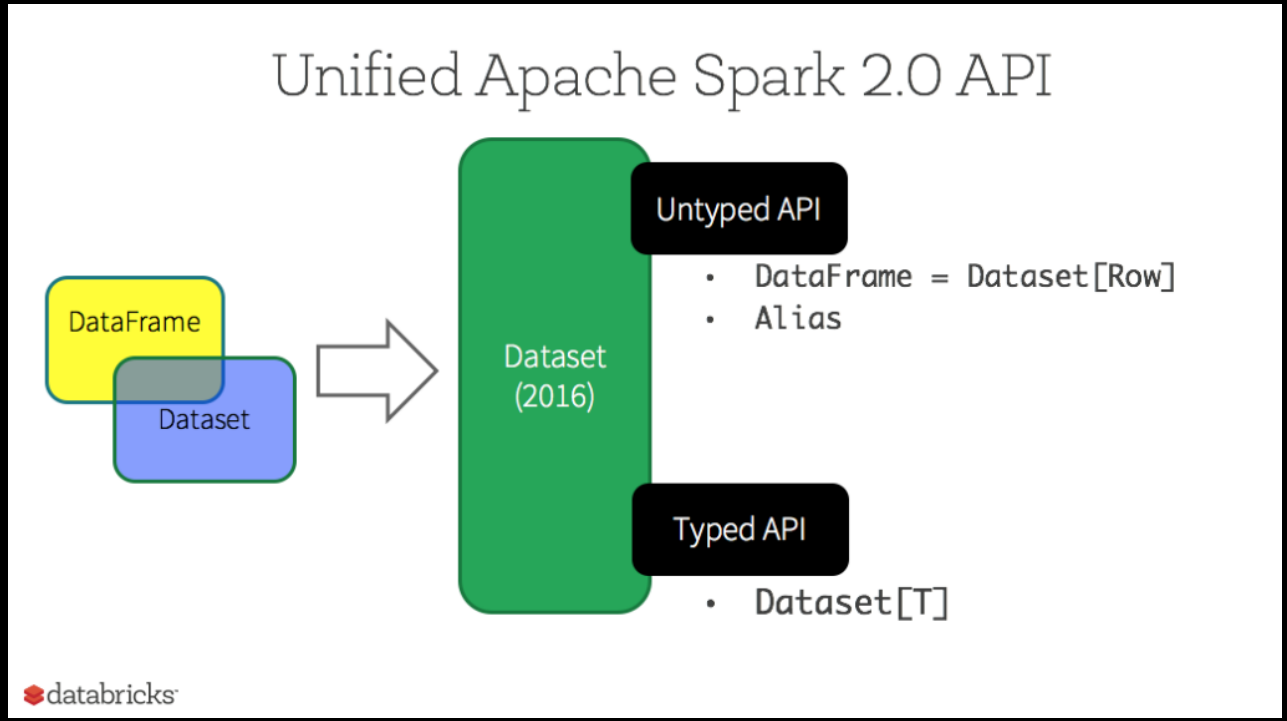
Opaque Computation
& Opaque Data

Structured APIs In Spark

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

Analysis errors are reported before a distributed job starts

Unification of APIs in Spark 2.0



DataFrame API code.

```
// convert RDD -> DF with column names
val df = parsedRDD.toDF("project", "page", "numRequests")
//filter, groupBy, sum, and then agg()
df.filter($"project" === "en").
      groupBy($"page").
      agg(sum($"numRequests").as("count")).
      limit(100).
      show(100)
```

project	page	numRequests
en	23	45
en	24	200

Take DataFrame → SQL Table → Query

```
df. createOrReplaceTempView ("edits")

val results = spark.sql("""SELECT page, sum(numRequests)
AS count FROM edits WHERE project = 'en' GROUP BY page
LIMIT 100""")

results.show(100)
```

project	page	numRequests
en	23	45
en	24	200

Easy to write code… Believe it!

```
from pyspark.sql.functions import avg

dataRDD = sc.parallelize([("Jim", 20), ("Anne", 31), ("Jim", 30)])
dataDF = dataRDD.toDF(["name", "age"])

# Using RDD code to compute aggregate average
(dataRDD.map(lambda (x,y): (x, (y,1))) .reduceByKey(lambda x,y: (x[0] +y[0], x[1]
+y[1])) .map(lambda (x, (y, z)): (x, y / z)))

# Using DataFrame
dataDF.groupBy("name").agg(avg("age"))
```

name	age
Jim	20
Ann	31
Jim	30

Why structure APIs?

DataFrame

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

SQL

```
select dept, avg(age) from data group by 1
```

RDD

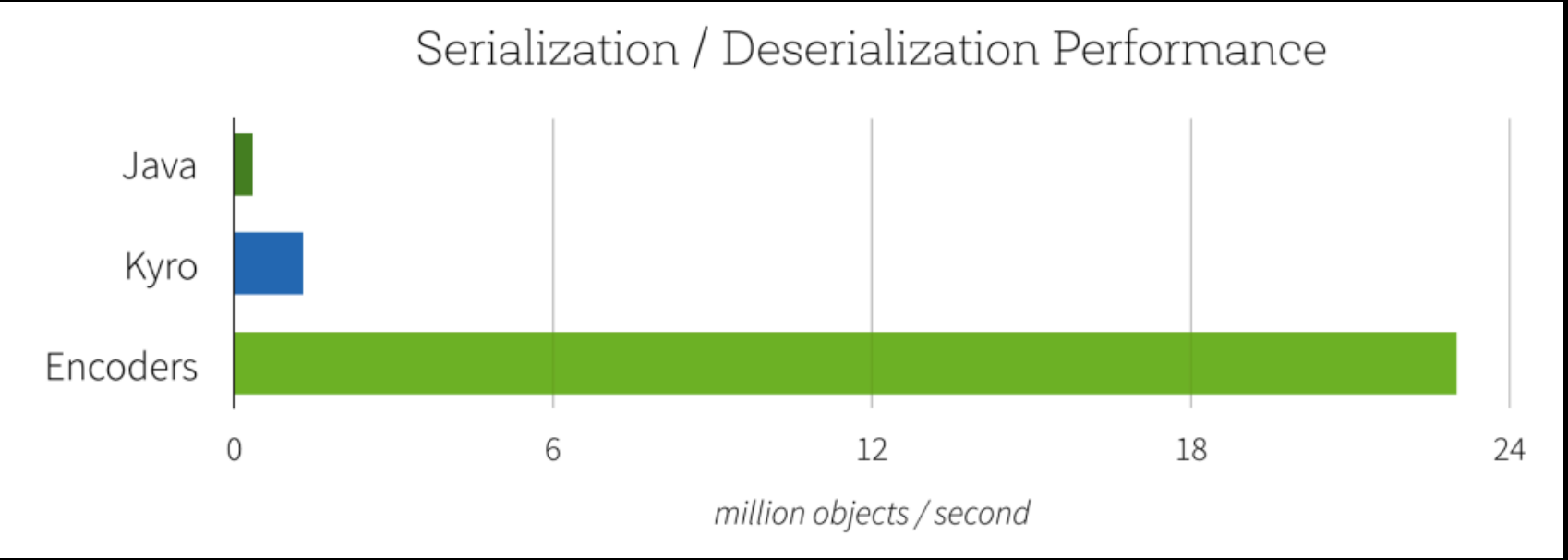
```
data.map { case (dept, age) => dept -> (age, 1) }
    .reduceByKey { case ((a1, c1), (a2, c2)) => (a1 + a2, c1 + c2)}
    .map { case (dept, (age, c)) => dept -> age / c }
```

Dataset API in Spark 2.x

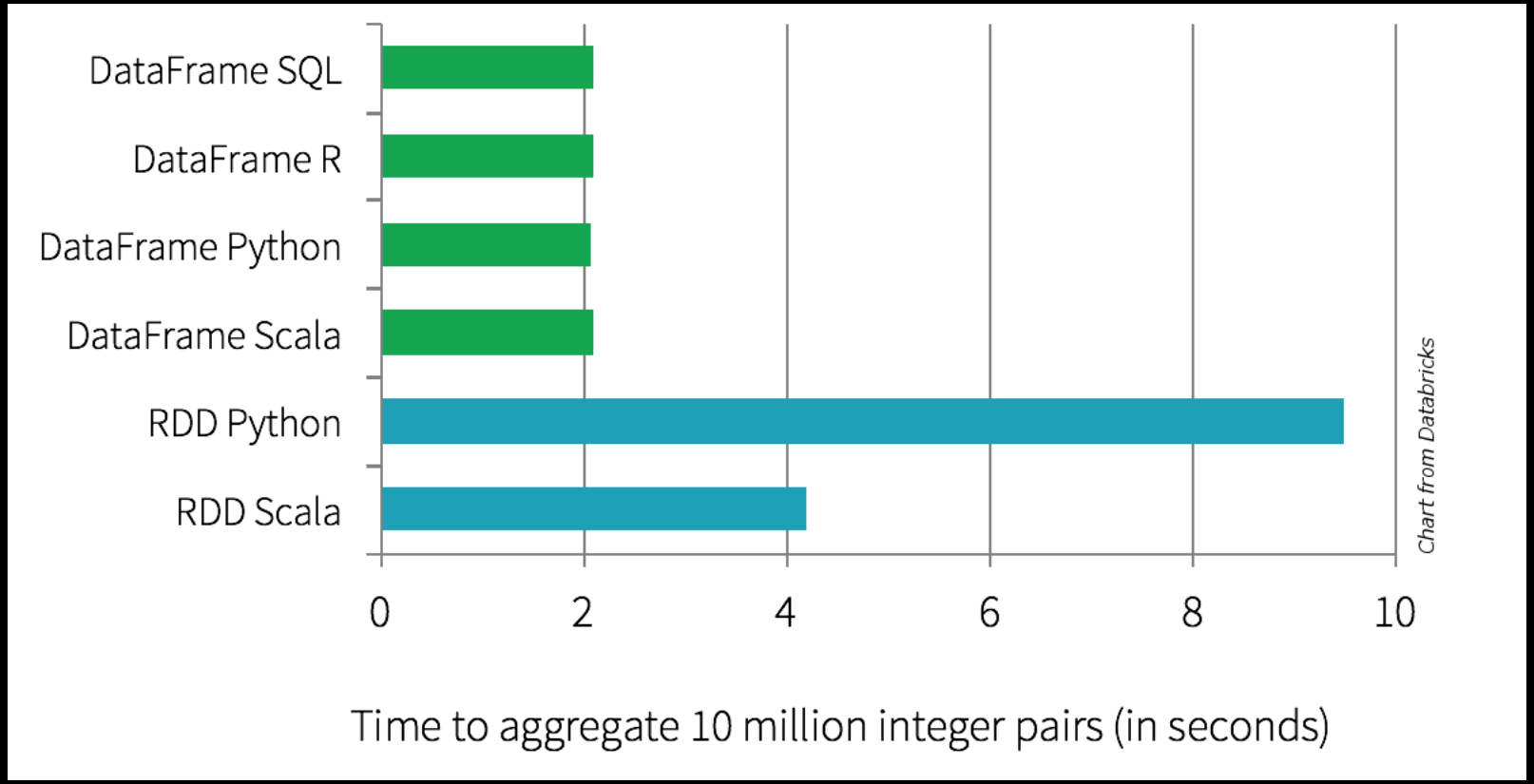
Type-safe: operate on domain objects with compiled lambda functions

```
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]
val filterDS = ds.filter(p=>p.age > 30)
```

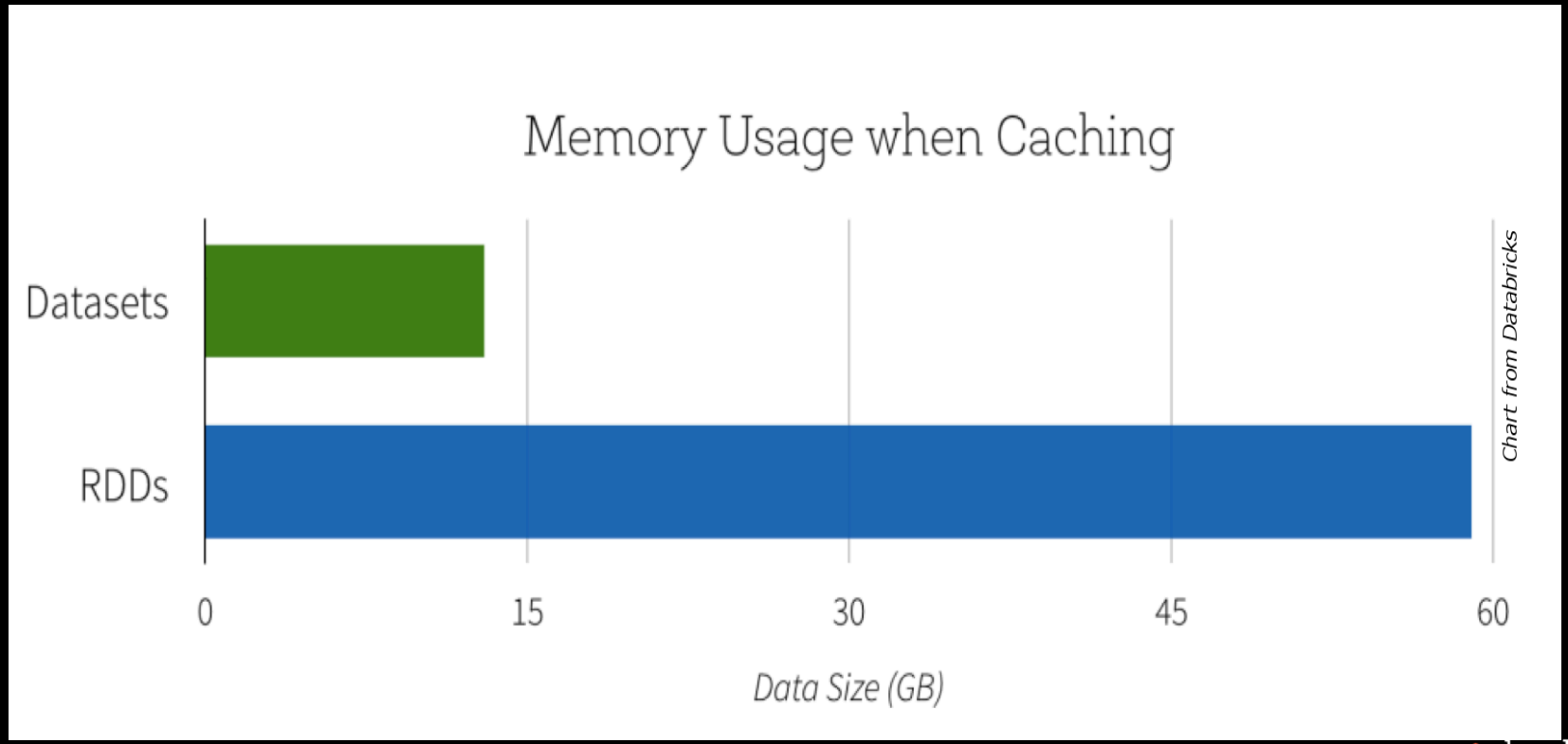
Datasets: Lightning-fast Serialization with Encoders



DataFrames are Faster than RDDs



Datasets < Memory RDDs



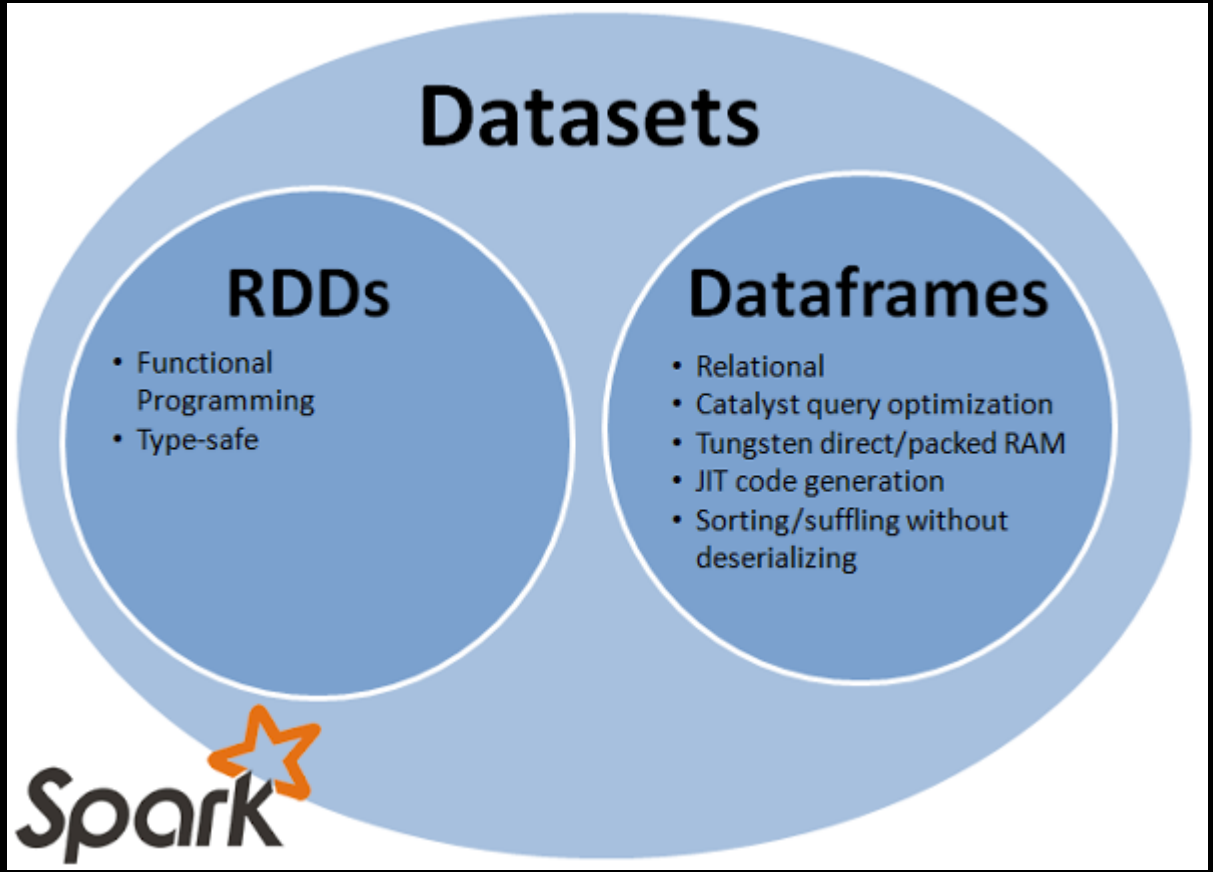
DataFrames & Datasets

Why

- High-level APIs and DSL
- Strong Type-safety
- Ease-of-use & Readability
- ***What-to-do***

When

- Structured Data schema
- Code optimization & performance
- Space efficiency with Tungsten



Source: michaelmalak

A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets

When to use them and why

by Jules Damji
Posted in **ENGINEERING BLOG** | July 14, 2016

BLOG: http://dbricks.co/3-apis

Spark Summit Talk: http://dbricks.co/summit-3aps

Project Tungsten II

Project Tungsten

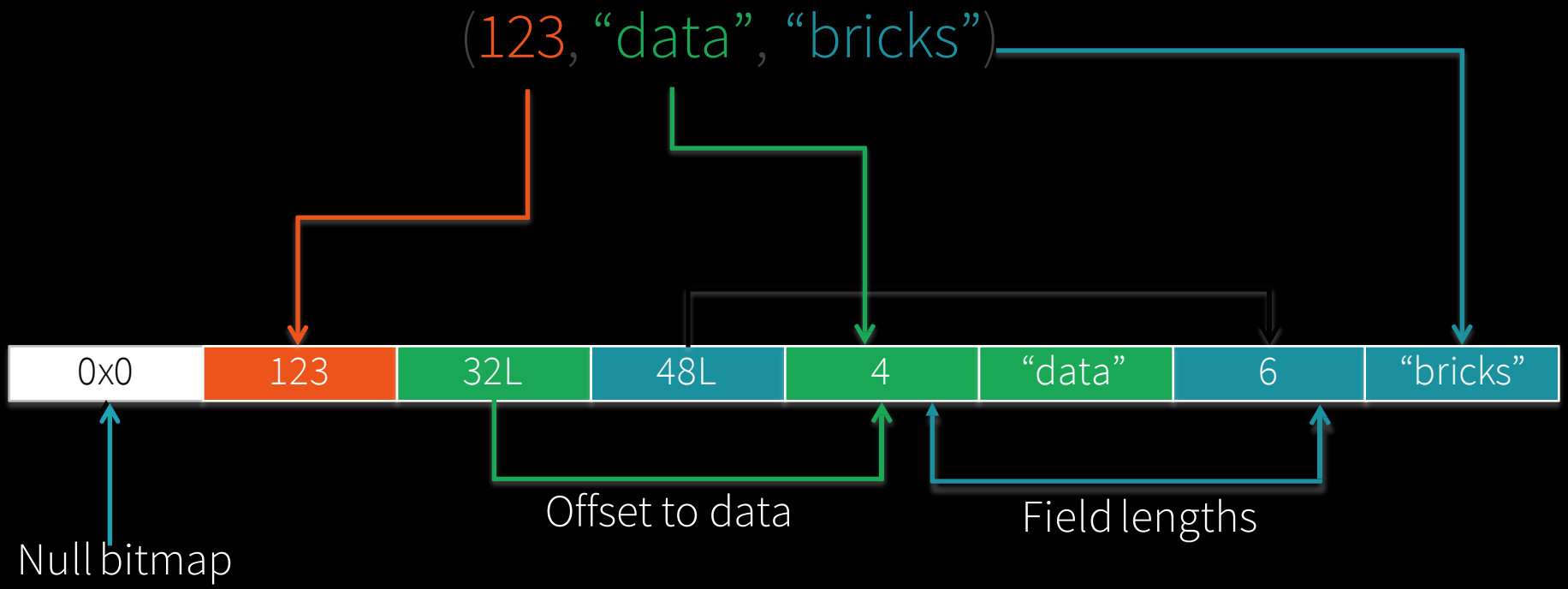
- Substantially speed up execution by optimizing CPU efficiency, via: SPARK-12795

(1) Runtime code generation

(2) Exploiting cache locality

(3) Off-heap memory management

Tungsten's Compact Row Format



Encoders

Encoders translate between domain objects and Spark's internal format

JVM Object

MyClass(123, “data”, “bricks”)

Internal Representation

Project Tungsten: Bringing Apache Spark Closer to Bare Metal

by Reynold Xin and Josh Rosen
Posted in **ENGINEERING BLOG** | April 28, 2015

In a previous blog post, we looked back and surveyed performance improvements made to Apache Spark in the past year. In this post, we look forward and share with you the next chapter, which we are calling *Project Tungsten*. 2014 witnessed Spark setting the world record in large-scale sorting and saw major improvements across the entire engine from Python to SQL to machine learning. Performance optimization, however, is a never ending process.

Project Tungsten will be the largest change to Spark's execution engine since the project's inception. It focuses on substantially improving the efficiency of *memory and CPU* for Spark applications, to push performance closer to the limits of modern hardware. This effort includes three initiatives:

1. *Memory Management and Binary Processing:* leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection
2. *Cache-aware computation*: algorithms and data structures to exploit memory hierarchy
3. *Code generation*: using code generation to exploit modern compilers and CPUs