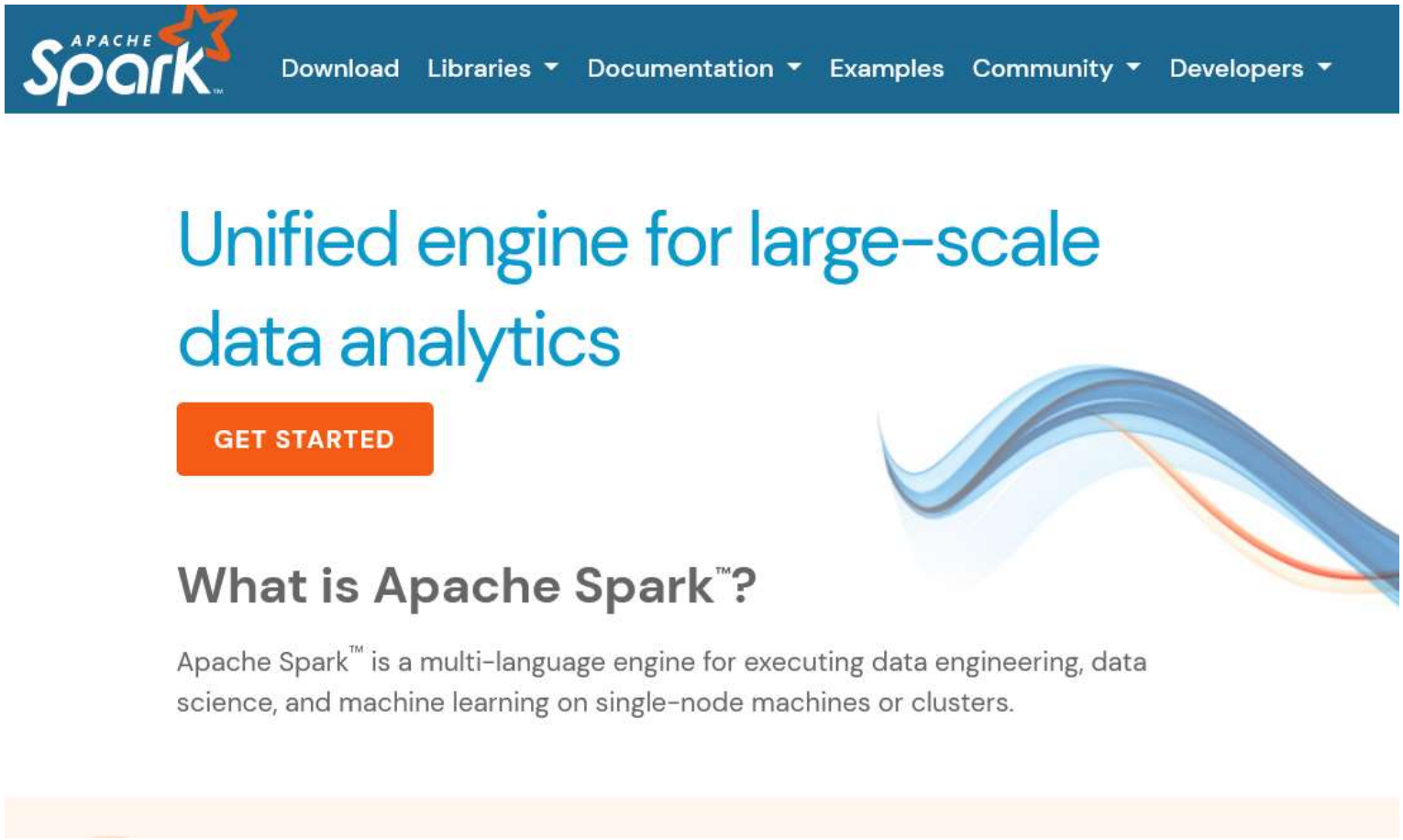
جلسه نهم

آپاچی اسپارک

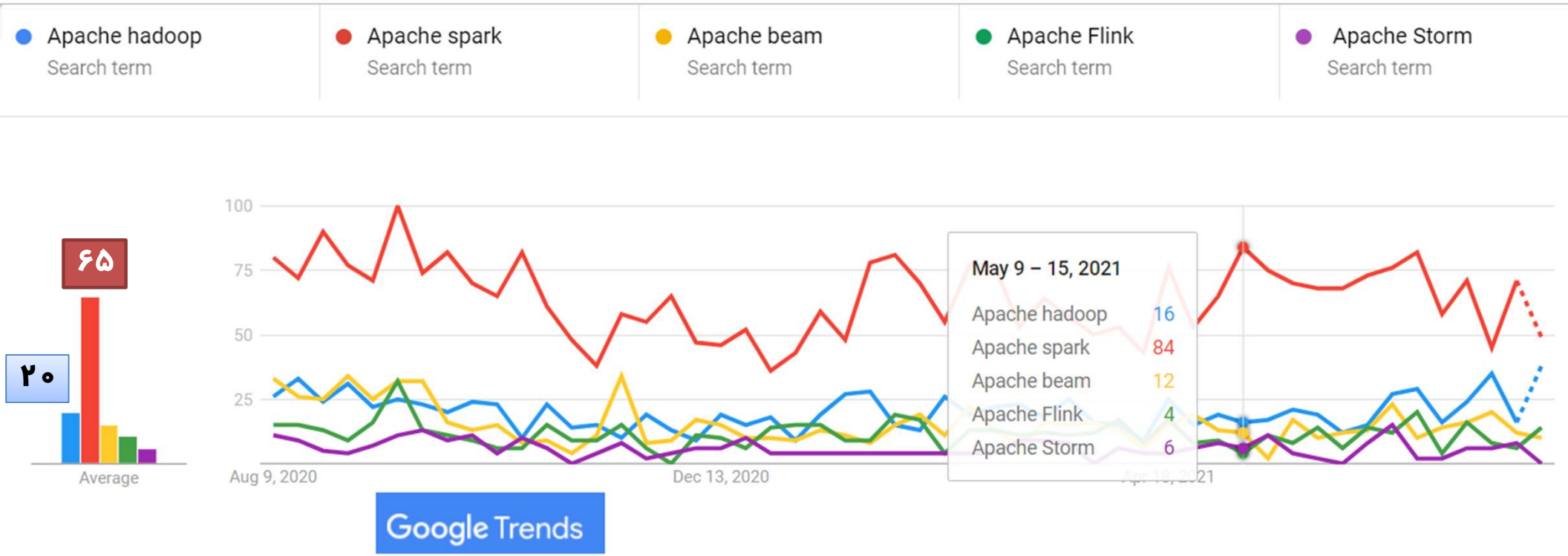
آنچه خواهیم دید

- اسپارک و جایگاه آن
- معماری اسپارک
- مفاهیم پایه و نحوه اجرای برنامه‌ها
- RDD, DataFrame, SparkSQL
- بررسی چند برنامه ساده با پای‌اسپارک

اسپارک – تعریف رسمی بنیاد آپاچی

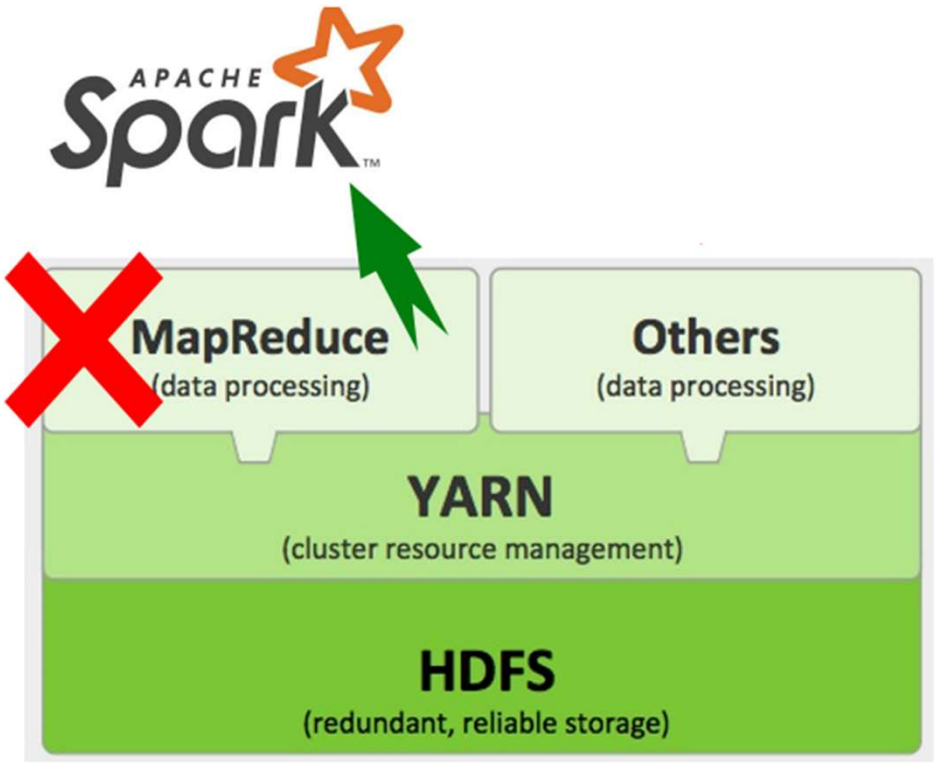


اسپارک در مقایسه با رقبا

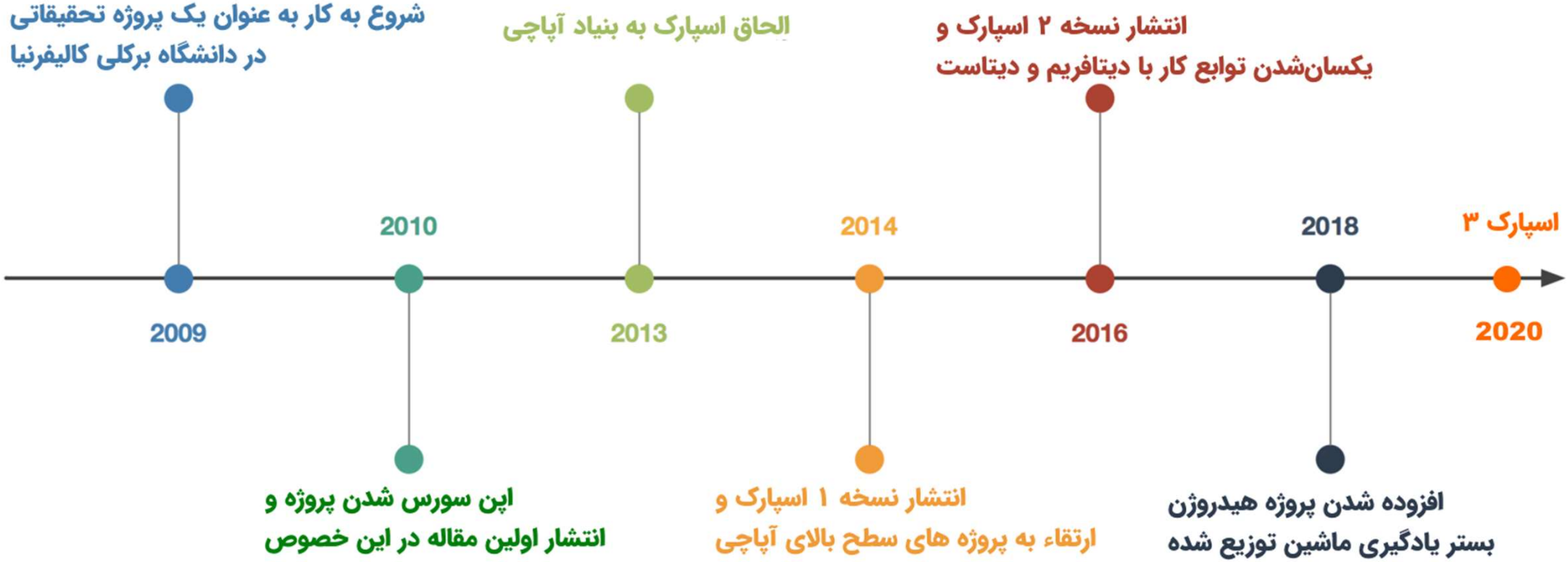


اسپارک و هدوپ

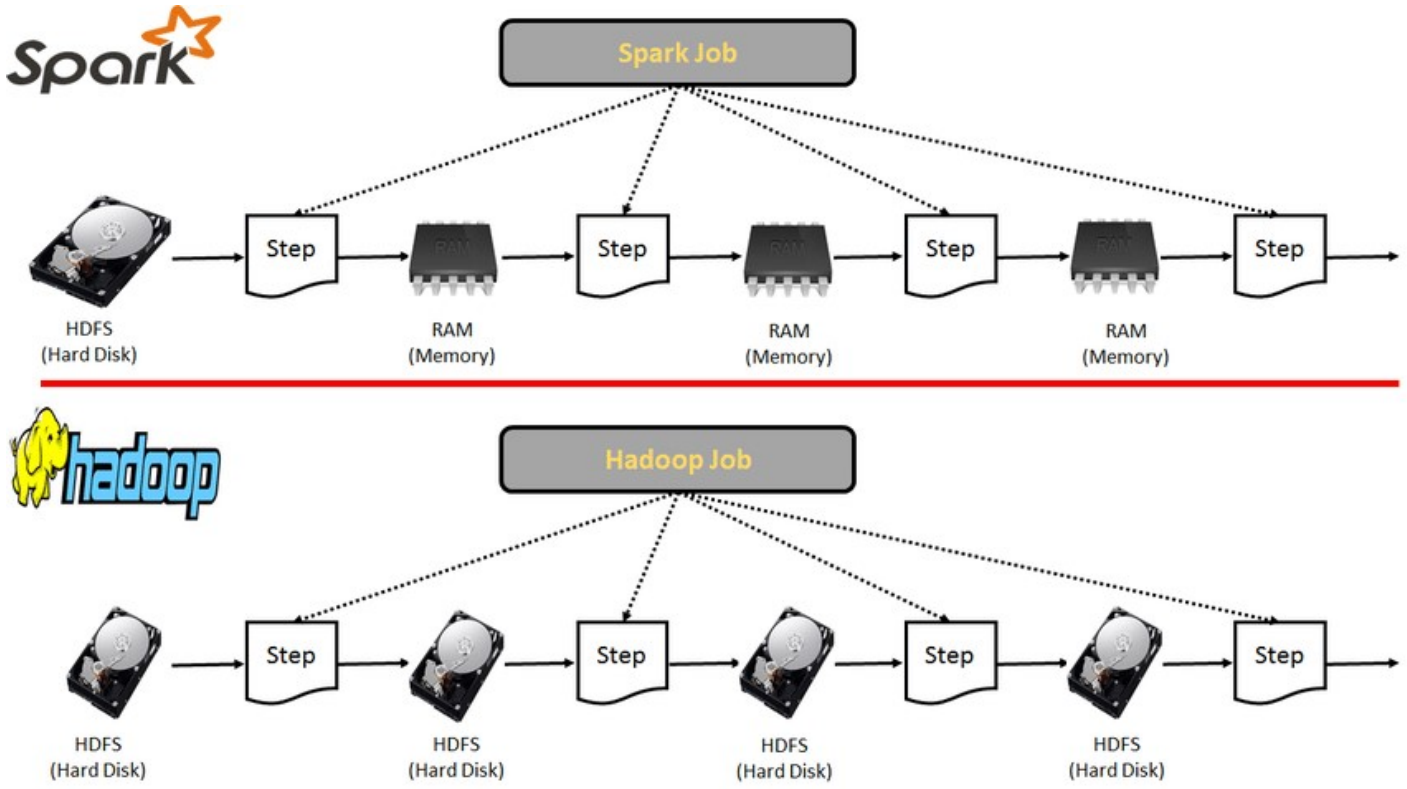
- **تنها جایگزین بخش پردازشی هدوپ**
- **نیازمند مکانیزمی برای خواندن/نوشتن داده**
  - HDFS
  - Cassandra
  - Mini.io
  - …..
- **نیازمند ابزارهای مدیریت منابع**
  - Yarn
  - Apache Mesos
  - Kubernetes
  - ….



تاریخچه مختصر اسپارک

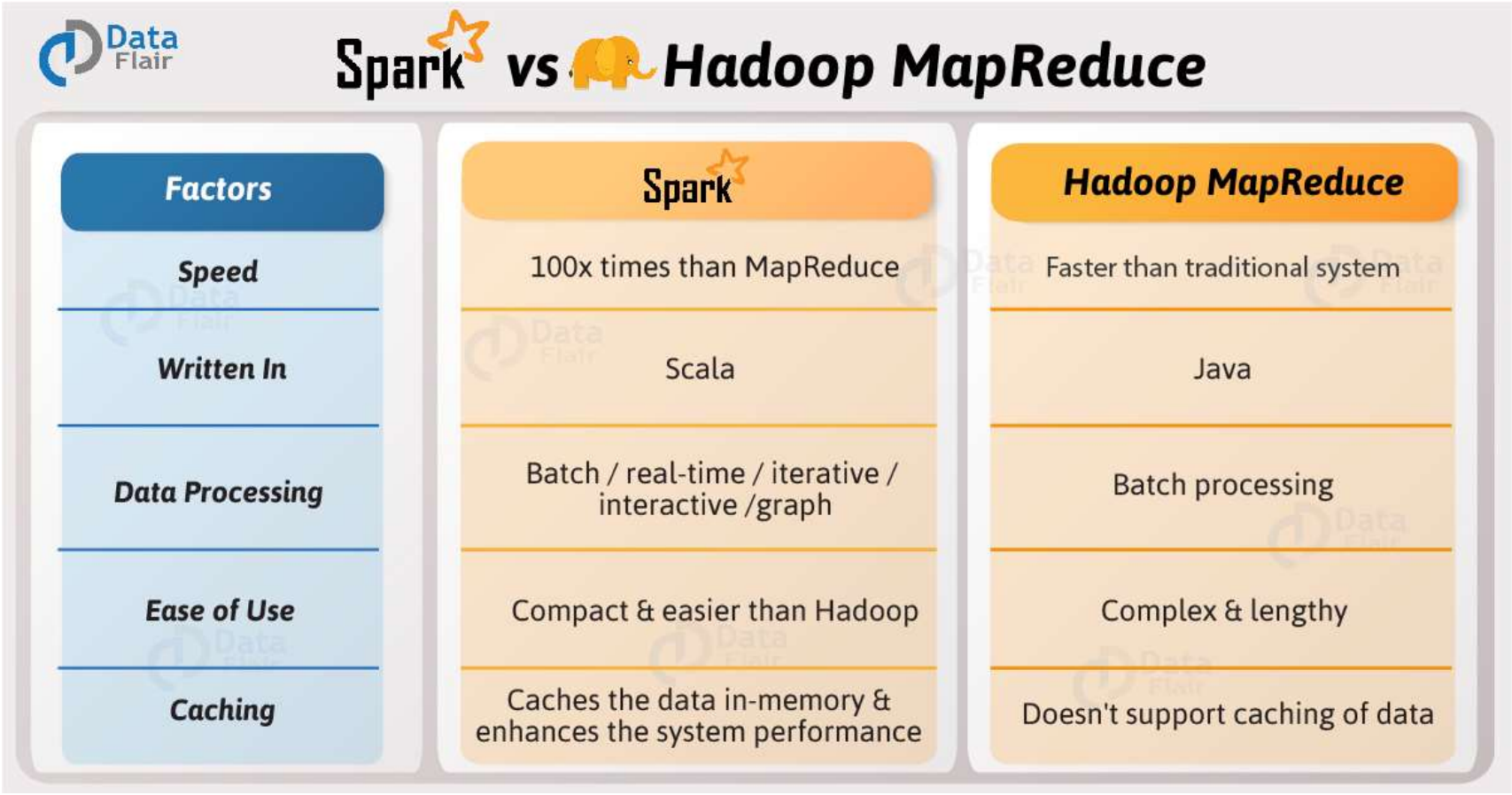


چرا اسپارک محبوبتر از هدوپ است؟



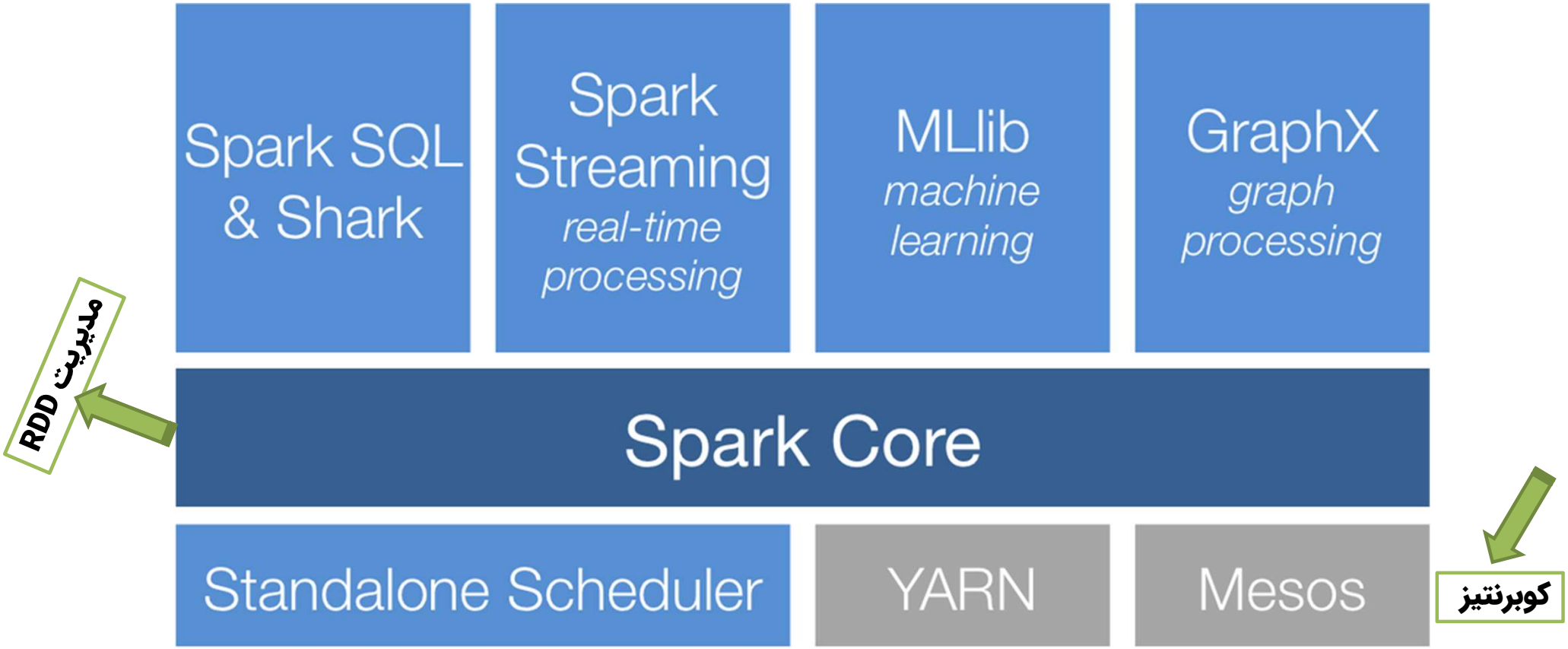
مهم ترین دلیل : سرعت

چرا اسپارک محبوبتر از هدوپ است؟

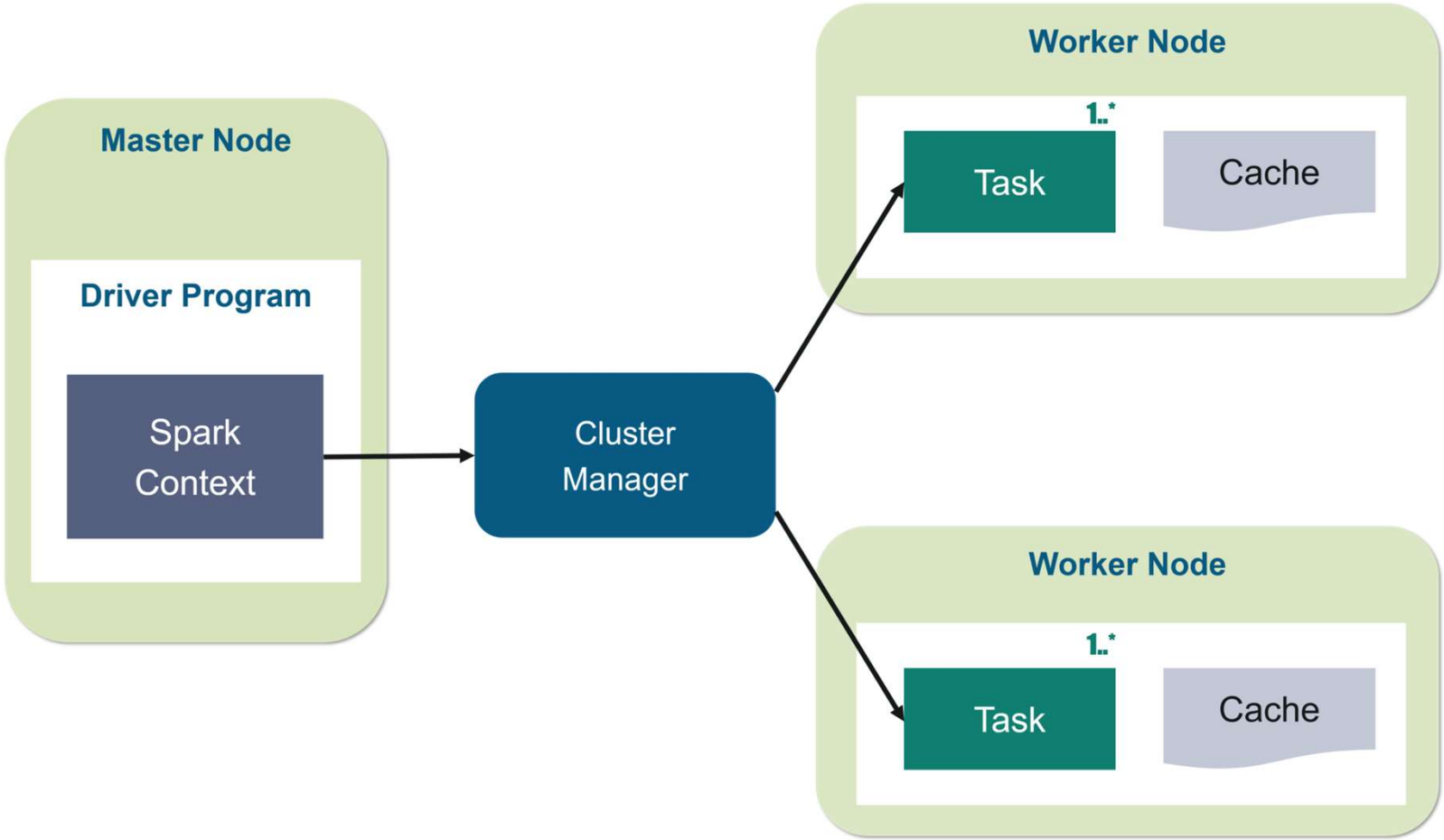


مزایای اصلی اسپارک

اجزای اصلی اسپارک



معماری اسپارک : اجزای اصلی یک کلاستر(؟)

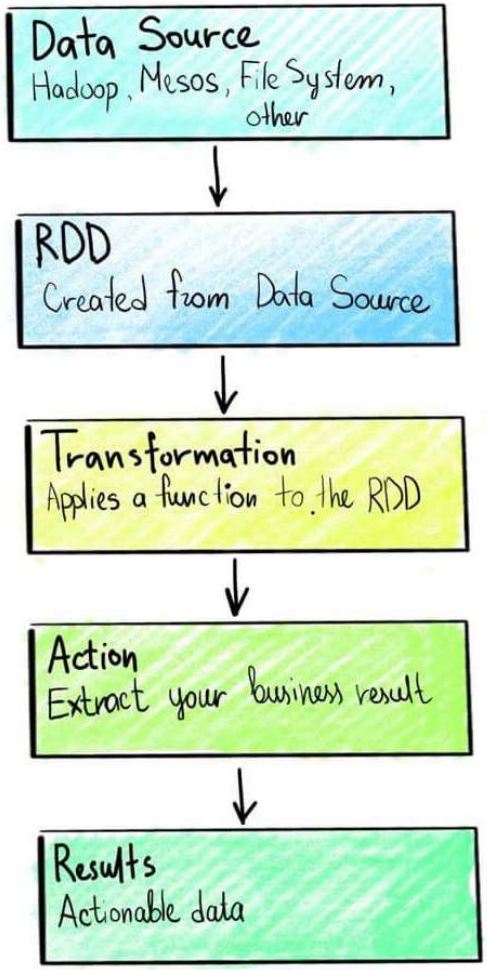


معماری اسپارک : اجزای اصلی یک کلاستر

- **Spark Driver** : برنامه شروع کننده و اجرا کننده دستورات اسپارک (Main Program) که وظیفه اجرای برنامه‌های اسپارک را برعهده دارد و یک برنامه اسپارک را به مجموعه‌ای از تسک‌ها تقسیم کرده، با ارتباط با مدیر کلاستر، این تسک‌ها را در کلاستر اسپارک برای اجرا توزیع می‌کند.
- **Spark ~~Context~~/Session** : دسترسی به امکانات اسپارک از طریق جلسات اسپارک مدیریت می‌شود. به ازای هر برنامه یک جلسه اسپارک ایجاد شده و از طریق این کتابخانه، اسپارک درایور به اجرای دستورات اسپارک می‌پردازد.
- **Cluster Manager** : مدیریت منابع در کلاستر را برعهده دارد.
  - Standalone
  - Mesos
  - Yarn
  - Kubernetes
- **Worker Node** : وظیفه اجرای تسک‌های ارسال شده از طریق درایور / ارسال وضعیت تسک‌ها به صورت مداوم به درایور

Prior to Spark 2.0.0 sparkContext was used as a channel to access all spark functionality. In order to use APIs of SQL, HIVE, and Streaming, separate contexts need to be created.
SPARK 2.0.0 onwards, SparkSession provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with DataFrame and Dataset APIs.

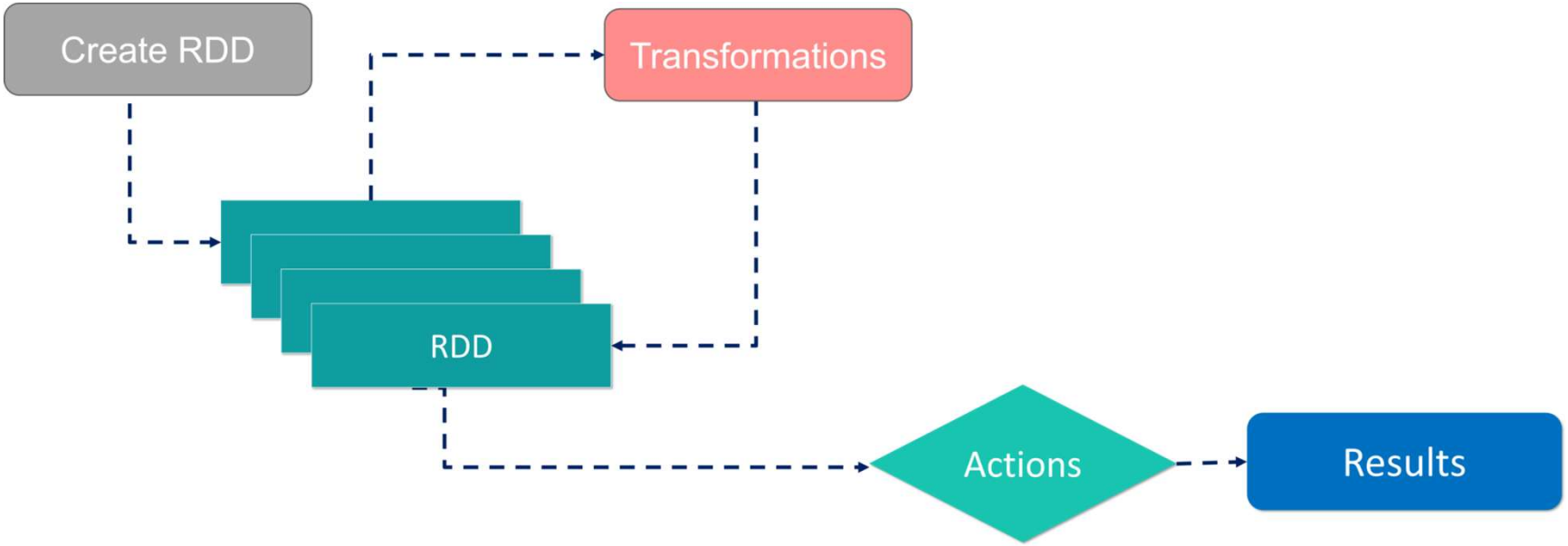
مفاهیم پایه اسپارک : RDD



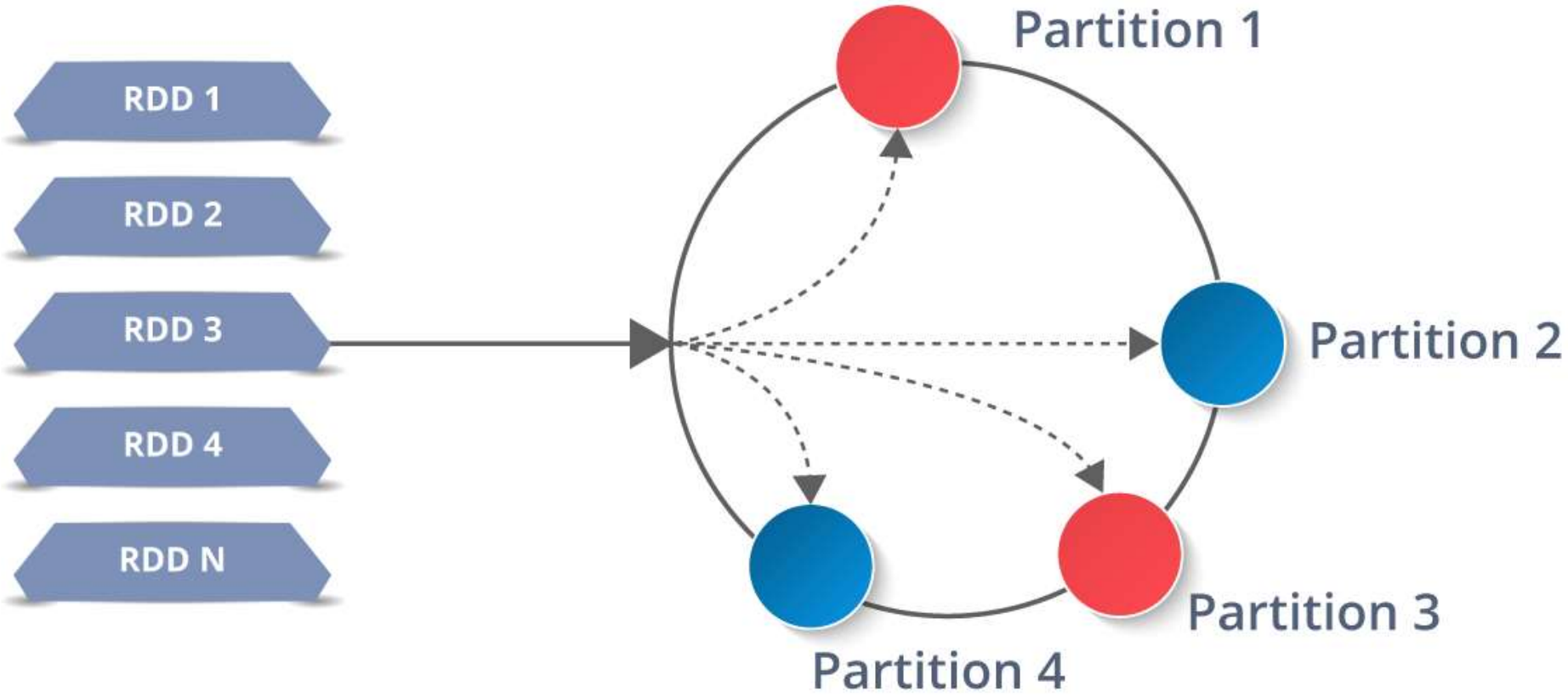
- **RDD(Resilient, Distributed, Dataset)**: **ساختمان داده اصلی پردازشی در اسپارک**
- تمام داده‌ها برای پردازش، به RDD تبدیل شده و در کلاستر اسپارک توزیع می‌شوند. (Partitioning)
- **هر RDD یا از داده‌های اولیه ایجاد می‌شود یا نتیجه تغییری بر روی RDD های قبلی است.**
- هر RDD فقط خواندنی است (immutable).
- **ساختمان داده توزیع شده برگشت‌پذیر(کشسانی)** : با رخداد هر خطا در شبکه، RDD ها از روی تاریخچه تغییرات قابل بازیابی هستند.

```
val rdd = sc.textFile("spam.txt")
val filtered = rdd.filter(line => line.contains("money"))
filtered.count()
```

مفاهیم پایه اسپارک : Transformation/Action

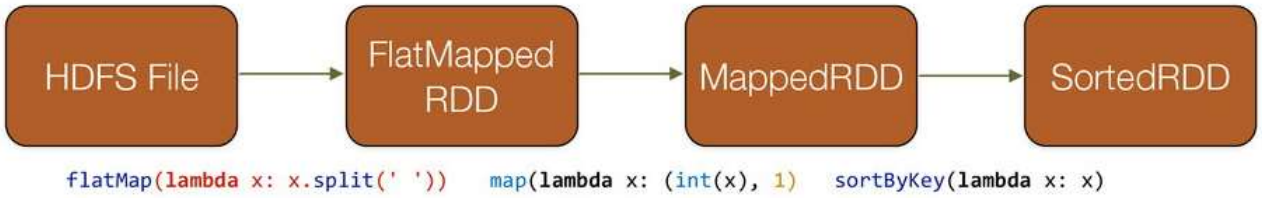


مفاهیم پایه اسپارک : توزیع RDD در کلاستر



مفاهیم پایه اسپارک : RDD Lineage

```
lines = sc.textFile("hdfs://...")
sortedCount = lines.flatMap(lambda x: x.split(' ')) \
                   .map(lambda x: (int(x), 1)) \
                   .sortByKey(lambda x: x)
```



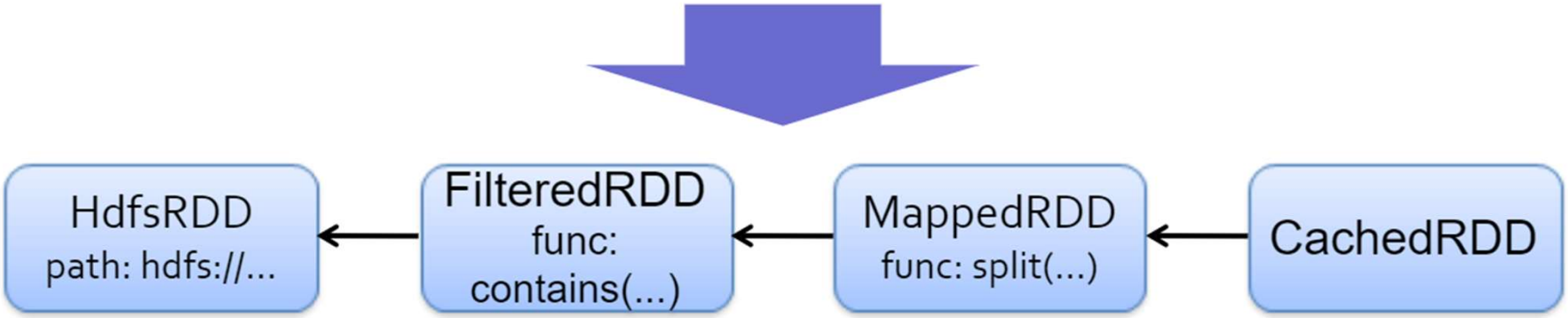
به ازای هر **RDD**، تمام تغییرات انجام گرفته بر روی **RDD** پایه تا رسیدن به آن، در درایور ذخیره می‌گردد تا در صورت رخداد خطا در شبکه و از بین رفتن **RDD**، بتوان آنرا به سرعت بازیابی کرد.

مفاهیم پایه اسپارک : RDD Lineage و تحمل‌پذیری خطا

- RDDs maintain *lineage* information that can be used to reconstruct lost partitions

- e.g.:

```
cachedMsgs = textFile(...).filter(_.contains("error"))
                          .map(_.split('\t')(2))
                          .cache()
```



مفاهیم پایه اسپارک : RDD Lineage و تابع toDebugString

```
scala> val wordCount = sc.textFile("README.md").flatMap(_.split("\\s+")).map((_, 1)).reduceByKey(_ + _)
wordCount: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[21] at reduceByKey at <console>:24

scala> wordCount.toDebugString
res13: String =
(2) ShuffledRDD[21] at reduceByKey at <console>:24 []
 +-(2) MapPartitionsRDD[20] at map at <console>:24 []
    |  MapPartitionsRDD[19] at flatMap at <console>:24 []
    |  README.md MapPartitionsRDD[18] at textFile at <console>:24 []
    |  README.md HadoopRDD[17] at textFile at <console>:24 []

------------------------------------------------------------------------
toDebugString uses indentations to indicate a shuffle boundary.

The numbers in round brackets show the level of parallelism at each stage, e.g. (2) in the above output.
```

Common

Transformations & Actions

ایجاد RDD ها

1

 # Using Parallelized Collections

```
>>> rdd = spark.parallelize([('Jim',24),('Hope', 25),('Sue', 26)])
>>> num_rdd = spark.parallelize(range(1,5000))
```

2

 # From other RDDs

```
>>> new_rdd = rdd.groupByKey()
>>> new_rdd = rdd.map(lambda x: (x,1))
```

 # From a text File

```
>>> tfile_rdd = Spark.textFile("/path/of_file/*.txt")
```

3

 # Reading directory of Text Files

```
>>> tfile_rdd = spark.wholeTextFiles("/path/of_directory/")
```

 # External Sources

HDFS / Databases / Kafka / …

Map/flatMap

map

Return a new RDD by applying a function to each element of this RDD

```
>>> rdd = spark.parallelize(["b", "a", "c"])
>>> rdd.map(lambda x: (x, 1))
```

[('a', 1), ('b', 1), ('c', 1)]

flatMap

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

```
>>> rdd = spark.parallelize([2, 3, 4])
>>> rdd.flatMap(lambda x: range(1, x))
```

[1, 1, 1, 2, 2, 3]

filter/mapPartition

mapPartitions

Return a new RDD by applying a function to each partition of this RDD.

```
>>> rdd = spark.parallelize([1, 2, 3, 4], 2)
>>> def f(iterator): yield sum(iterator)
>>> rdd.mapPartitions(f).collect()
[3, 7]
```

filter

Return a new RDD containing only the elements that satisfy a predicate.

```
>>> rdd = spark.parallelize([1, 2, 3, 4, 5])
>>> rdd.filter(lambda x: x % 2 == 0).collect()
[2, 4]
```

sortBy/sortByKey

Transformation

sortBy

Sorts this RDD by the given keyfunc

```
>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
>>> sc.parallelize(tmp).sortBy(lambda x: x[0]).collect()
```

[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

sortByKey

Sorts this RDD, which is assumed to consist of (key, value) pairs.

```
>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
>>> sc.parallelize(tmp).sortByKey(True, 1).collect()
```

[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

```
>> spark.sparkContext.parallelize(tmp).sortByKey(True,
2).glom().collect()
```

[[('1', 3), ('2', 5), ('a', 1)], [('b', 2), ('d', 4)]]

[(0, [2, 8]), (1, [1, 3, 1, 5])]

Transformation

groupBy/groupByKey

groupBy

Return an RDD of grouped items.

```
>>rdd = spark.sparkContext.parallelize([1, 1, 2, 3, 5, 8])
>>result = rdd.groupBy(lambda x: x % 2).mapValues(list).collect()
```

[(0, [2, 8]), (1, [1, 1, 3, 5])]

groupByKey

Group the values for each key in the RDD into a single sequence.

```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
rdd.groupByKey().mapValues(list).collect()
```

[('a', [1, 1]), ('b', [1])]

```
rdd.groupByKey().mapValues(len).collect()
```

[('a', 2), ('b', 1)]

collect/count/take/first

collect

Is the common and simplest operation that returns our entire RDDs content to driver program

count

Return the number of elements in this RDD.

```
>>> sc.parallelize([2, 3, 4]).count()
3
```

countByValue

Return the count of each unique value in this RDD as a dictionary of (value, count) pairs.

```
>>> sc.parallelize([1, 2, 1, 2, 2], 2).countByValue()).collect()
[(1, 2), (2, 3)]
```

first

Return the first element in this RDD.

```
>>> sc.parallelize([2, 3, 4]).first()
2
```

take

Take the first "n" num elements of the RDD.

```
>>> sc.parallelize([2, 3, 4, 5, 6]).cache().take(2)
[2, 3]
```

Actions

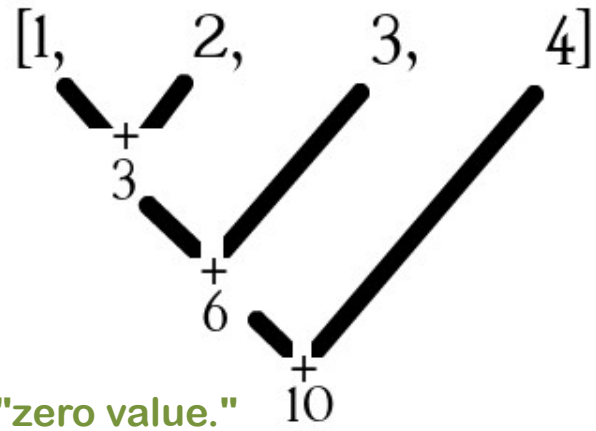
reduce/fold

reduce

Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function.

```
>>> from operator import add
>>> sc.parallelize([1, 2, 3, 4, 5]).reduce(add)
```

15



fold

Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral "zero value."

```
>>> from operator import add
>>> sc.parallelize([1, 2, 3, 4, 5]).fold(0, add)
```

15

Saving RDDs

saveAsTextFile

Save this RDD as a text file, using string representations of elements.

```
>>> rdd.saveAsTextFile("rdd.txt")
```

saveAsSequenceFile

Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system.

#saveAsObjectFile

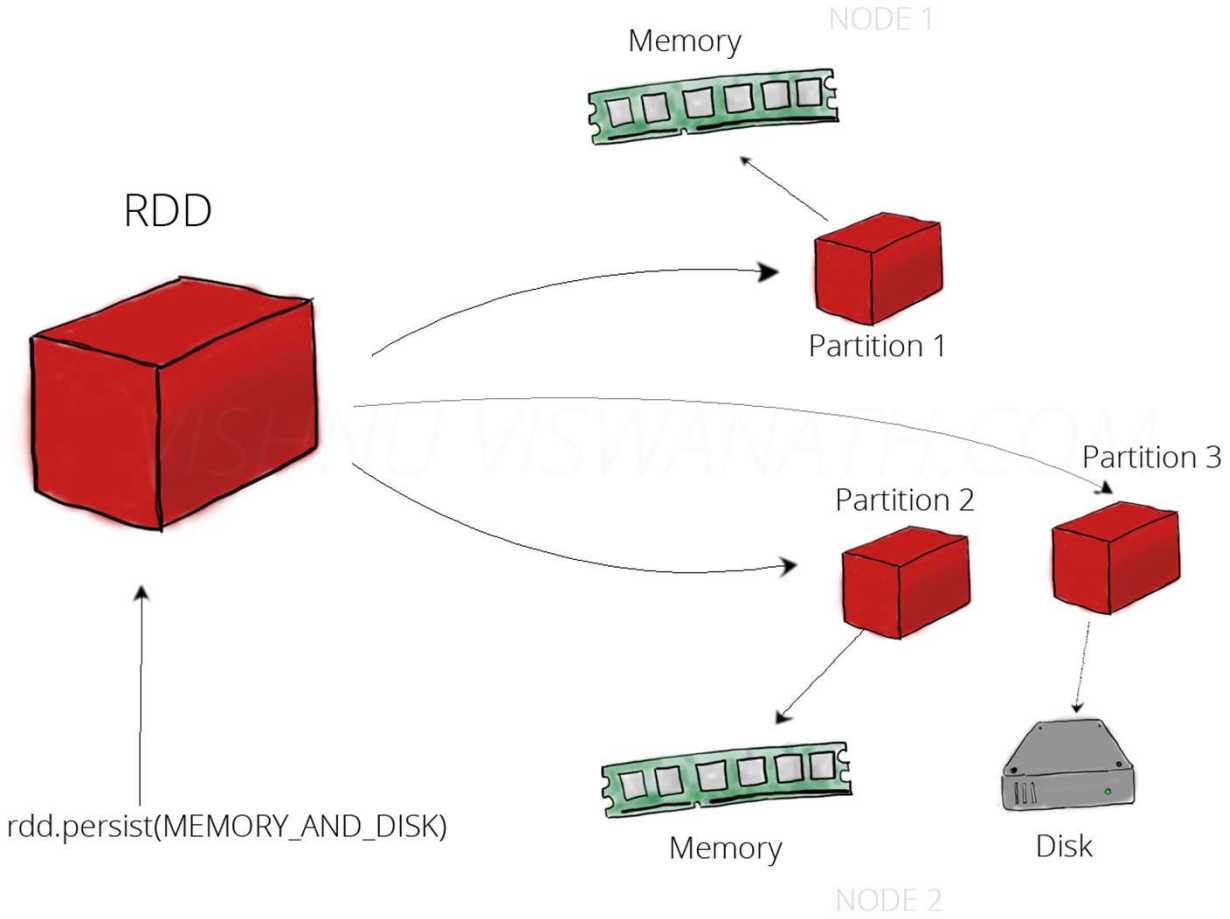
Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().

Persist/Cache

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

Persist/Cache

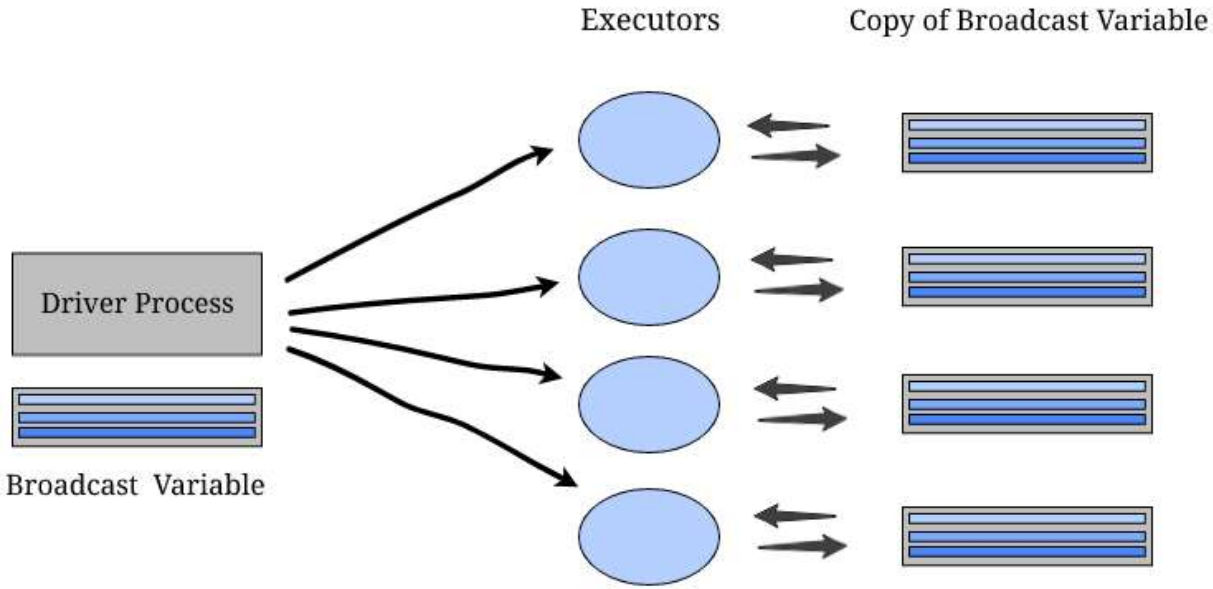
```
Persist Types
-----------------
MEMORY_ONLY
MEMORY_AND_DISK
MEMORY_ONLY_SER
MEMORY_AND_DISK_SER
DISK_ONLY
MEMORY_ONLY_2
DISK_ONLY_2
…
OFF_HEAP
```



متغیرهای سراسری در اسپارک

Broadcast Variables

برای اشتراک متغیرها در کل کلاستر از متغیرهای برودکست استفاده می‌کنیم.
این متغیرها فقط خواندنی هستند و توسط درایور در بین اجرا کننده‌ها توزیع می‌شوند




```
>>>bVar=spark.broadcast([1, 2, 3])
<pyspark.broadcast.Broadcast
object at 0x102789f10>

>>> bVar.value
[1, 2, 3]
```

Accumulators : متغیرهای سراسری برای انجام عملیات عددی

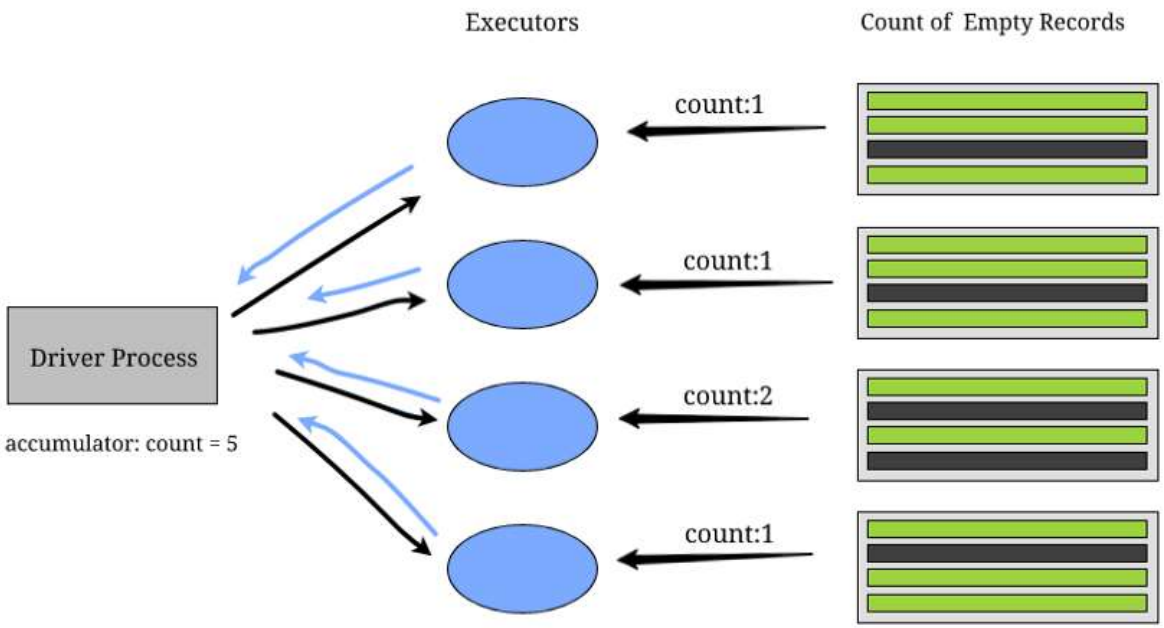
Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Accumulators : متغیرهای سراسری برای انجام عملیات عددی




```
>>> accum = spark.accumulator(0)
>>> accum
Accumulator<id=0, value=0>

>>> sc.parallelize([1, 2, 3, 4])
.foreach(lambda x: accum.add(x))
...

>>> accum.value
10
```

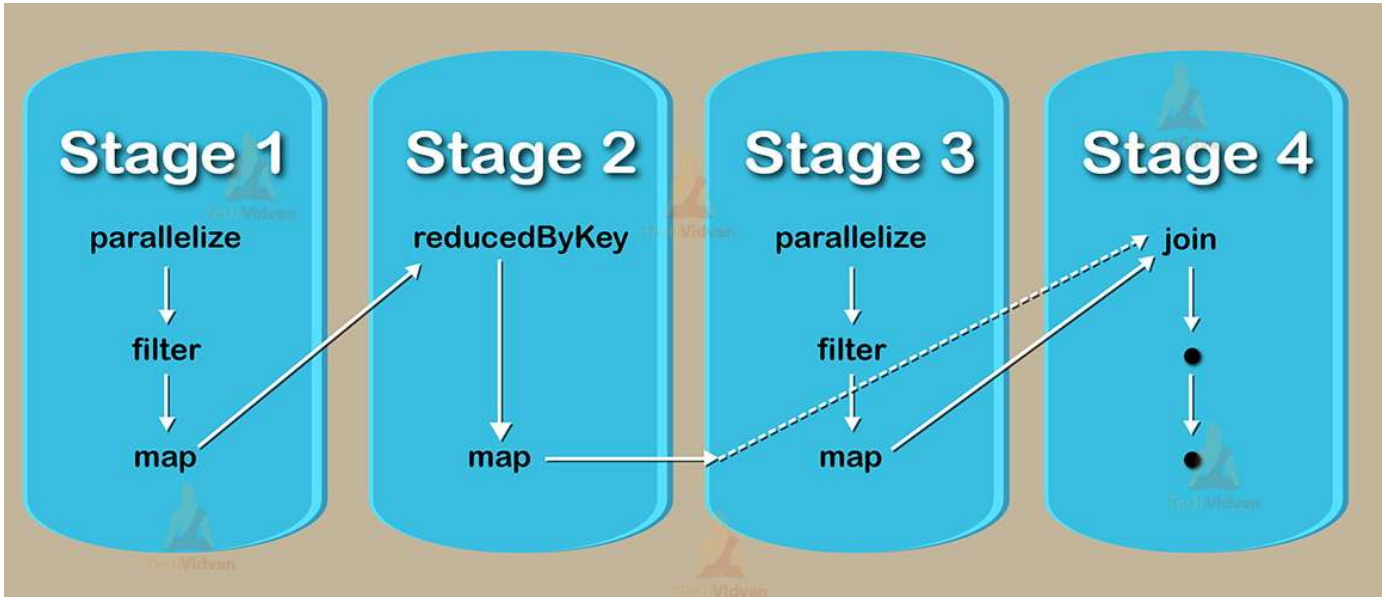
Lazy Evaluation

Spark will not start the execution of the process until an ACTION is called.

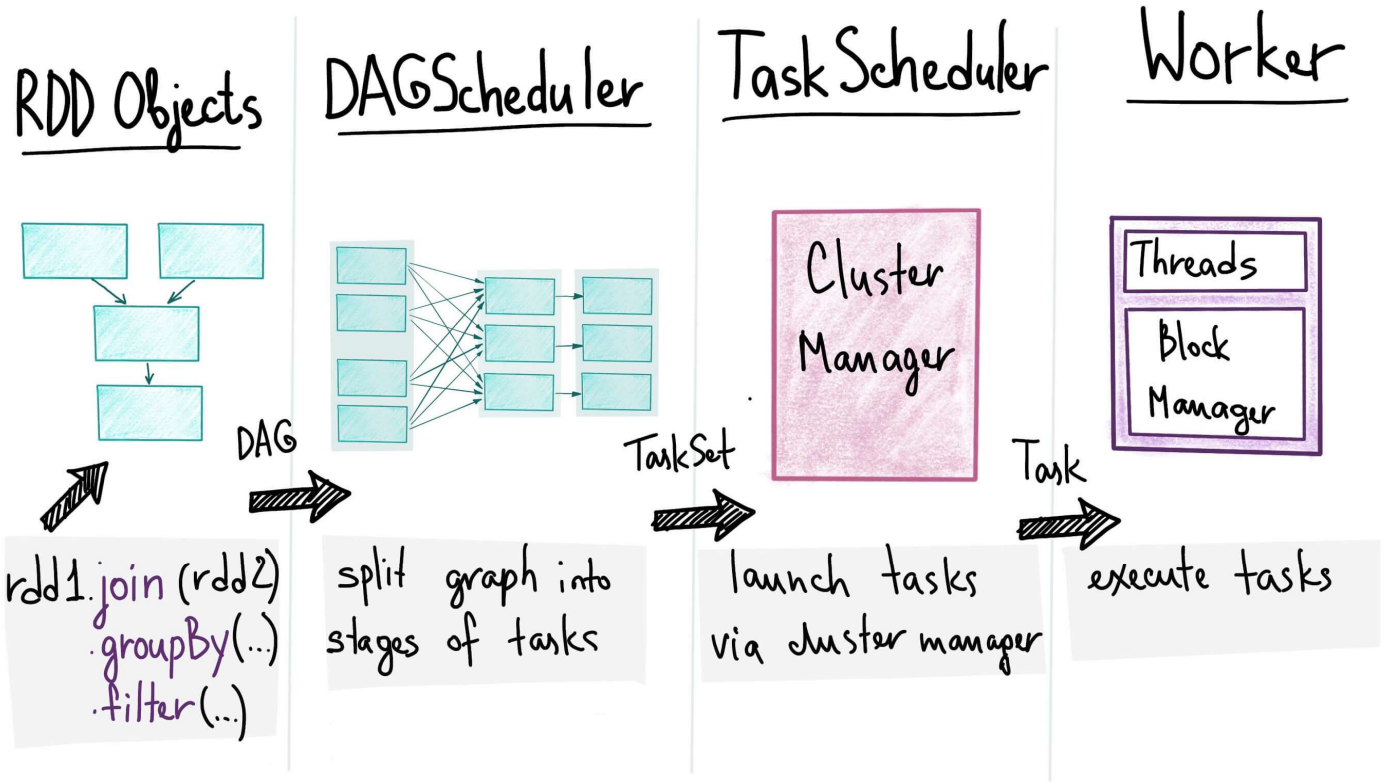
- Evaluation of RDDs is completely lazy.
- Spark does not begin computing the partitions until an action is called.
- Actions trigger the scheduler, which builds a *directed acyclic graph* (called the DAG), based on the dependencies between RDD transformations.

DAG : گراف پردازش کارها در اسپارک

به ازای هر اکشن، گراف پردازش داده از اولین RDD تا حصول نتیجه، ایجاد شده و به ازای هر بخشی از گراف پیش‌نیازی وظایف (DAG) که امکان موازی سازی و توزیع در شبکه را داشته و منتظر نتایج مراحل قبل نیست، یک Stage تعریف میشود.



پردازش تسک‌ها : نگاهی دقیق‌تر

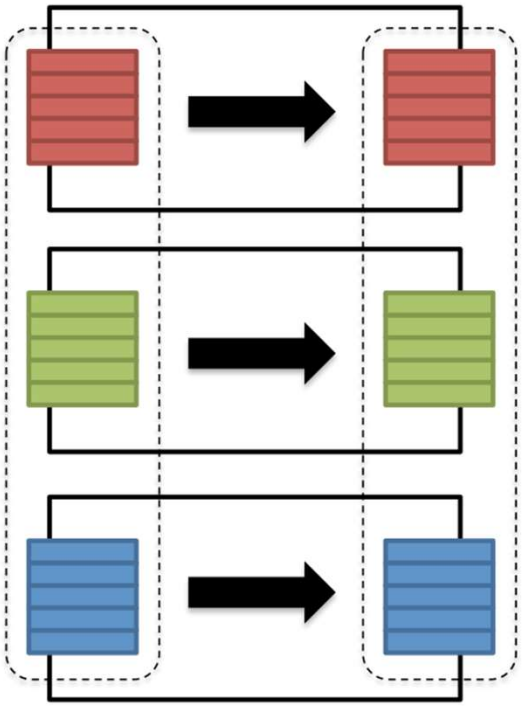


Narrow/Wide Transformation

- Map
- Filter

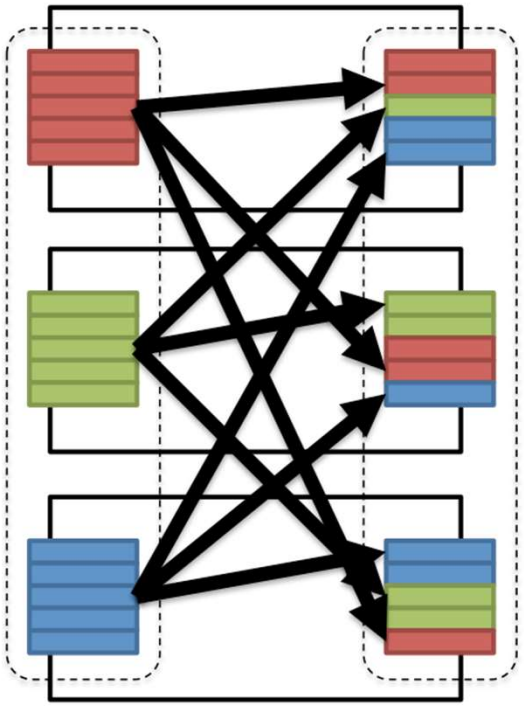
Narrow transformation

- Input and output stays in same partition
- No data movement is needed



Wide transformation

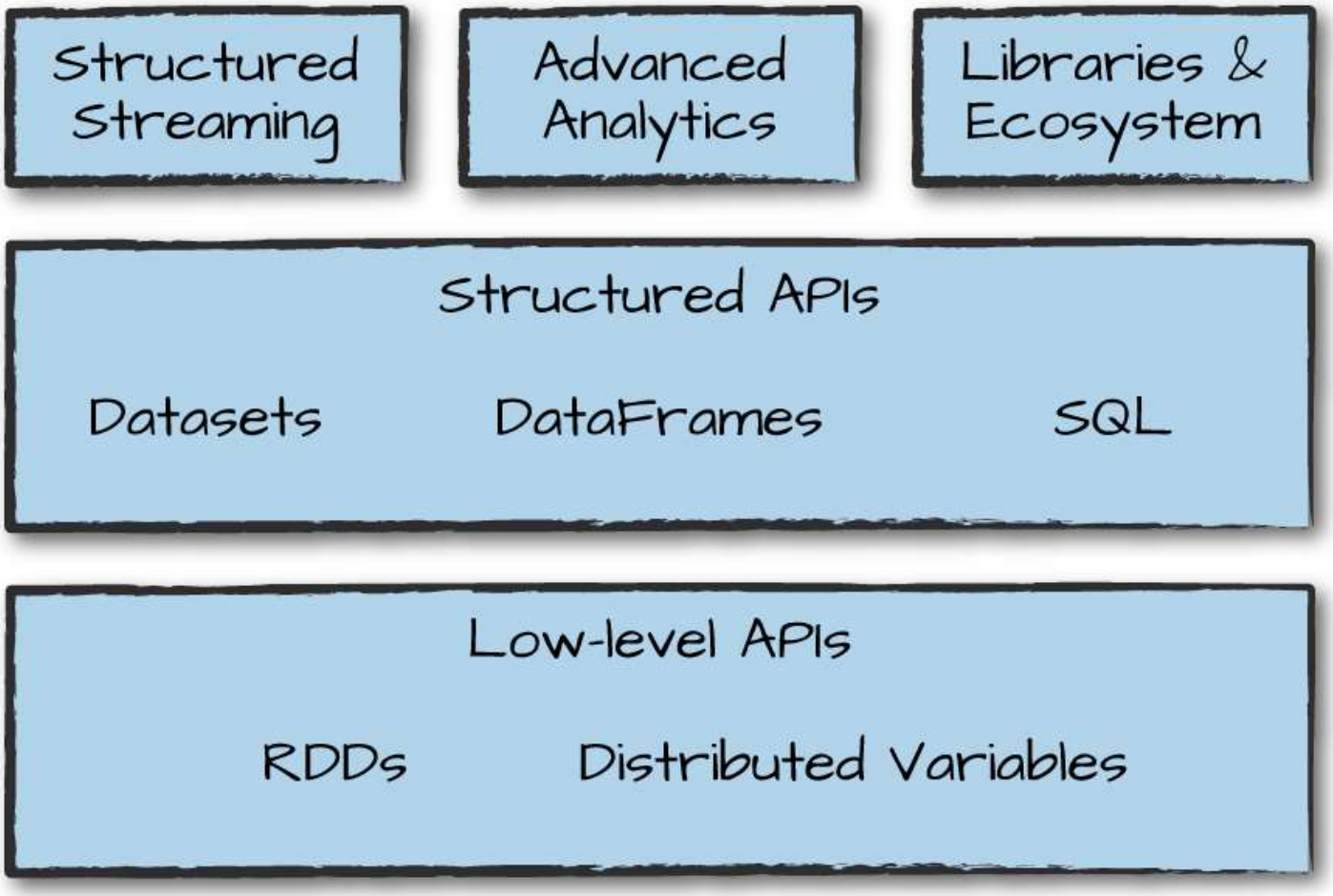
- Input from other partitions are required
- Data shuffling is needed before processing



- groupByKey
- reduceByKey

عملیات شافل یا توزیع داده‌ها بین نودها در شبکه

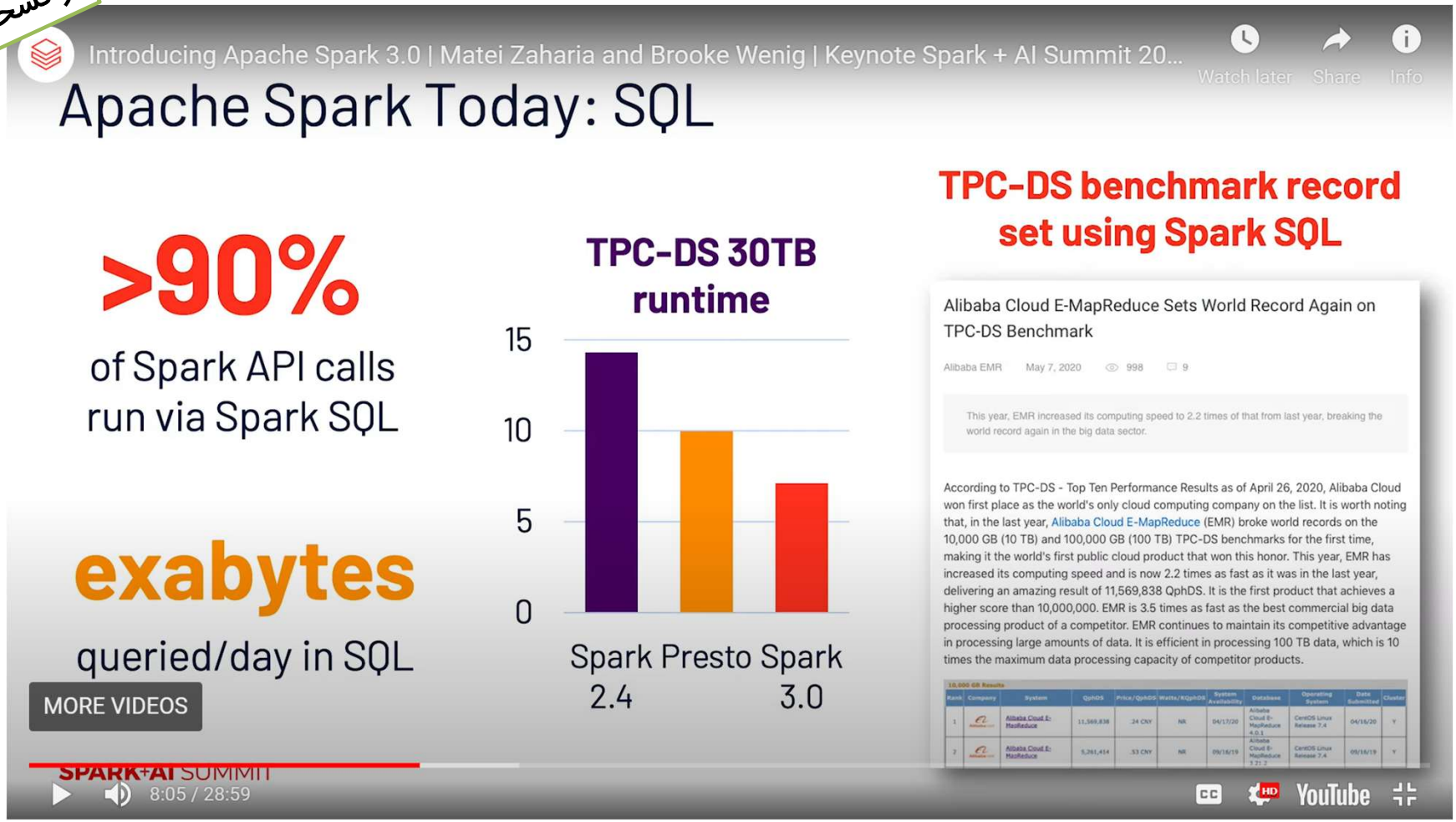
نگاهی مجدد به اکوسیستم توسعه اسپارک



آنچه تاکنون صحبت کردیم

Sark SQL: ۹۰ درصد استفاده از اسپارک

از نسخه ۱ اسپارک



تاریخچه توابع و ساختارهای اسپارک

Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

Fast/efficient internal representations

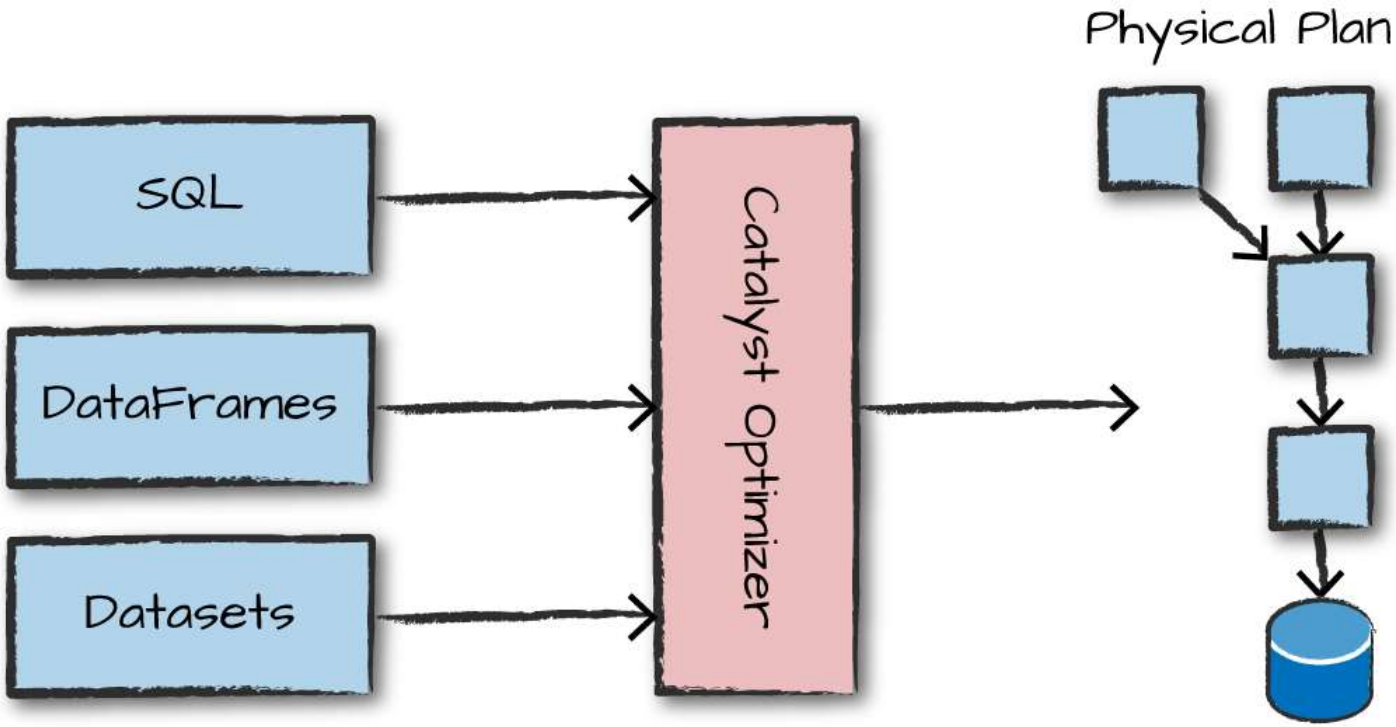
Internally rows, externally JVM objects

Almost the "Best of both worlds": **type safe + fast**

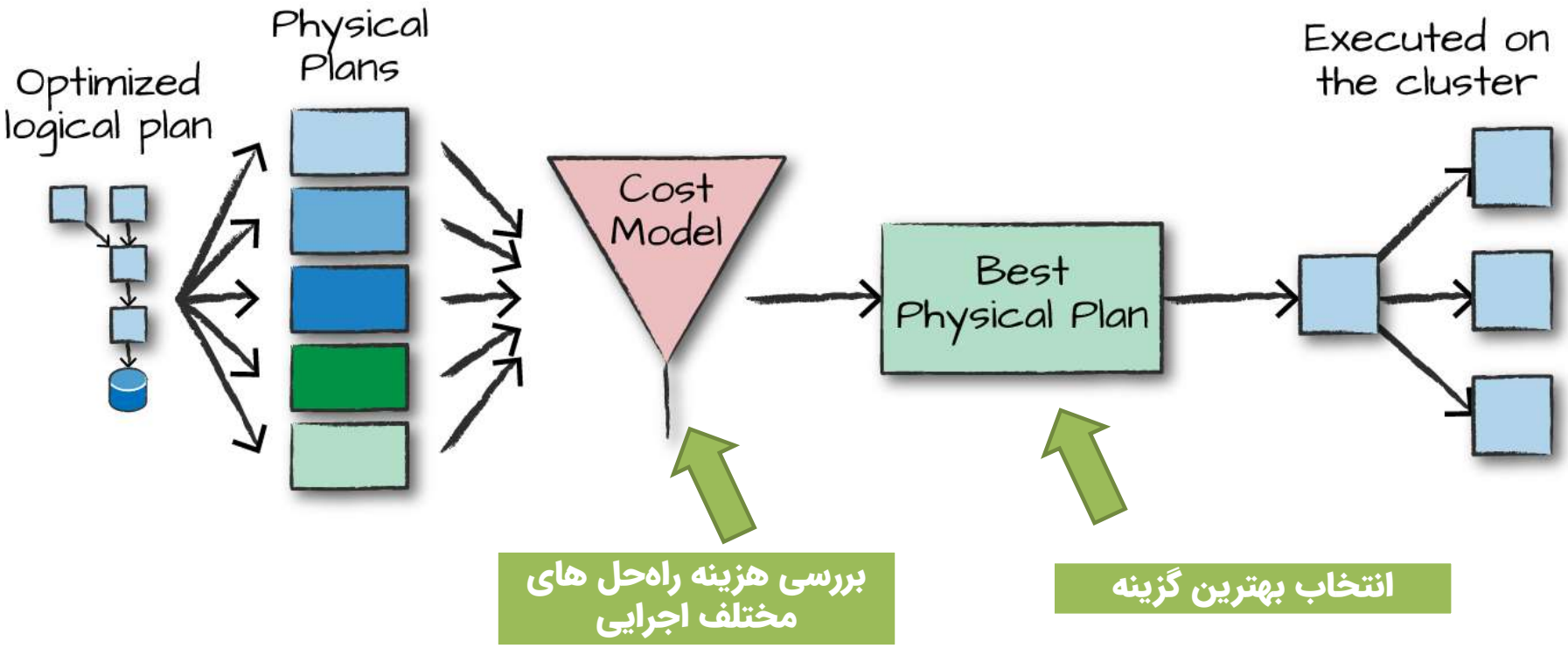
But slower than DF
Not as good for interactive analysis, especially Python

Structured API

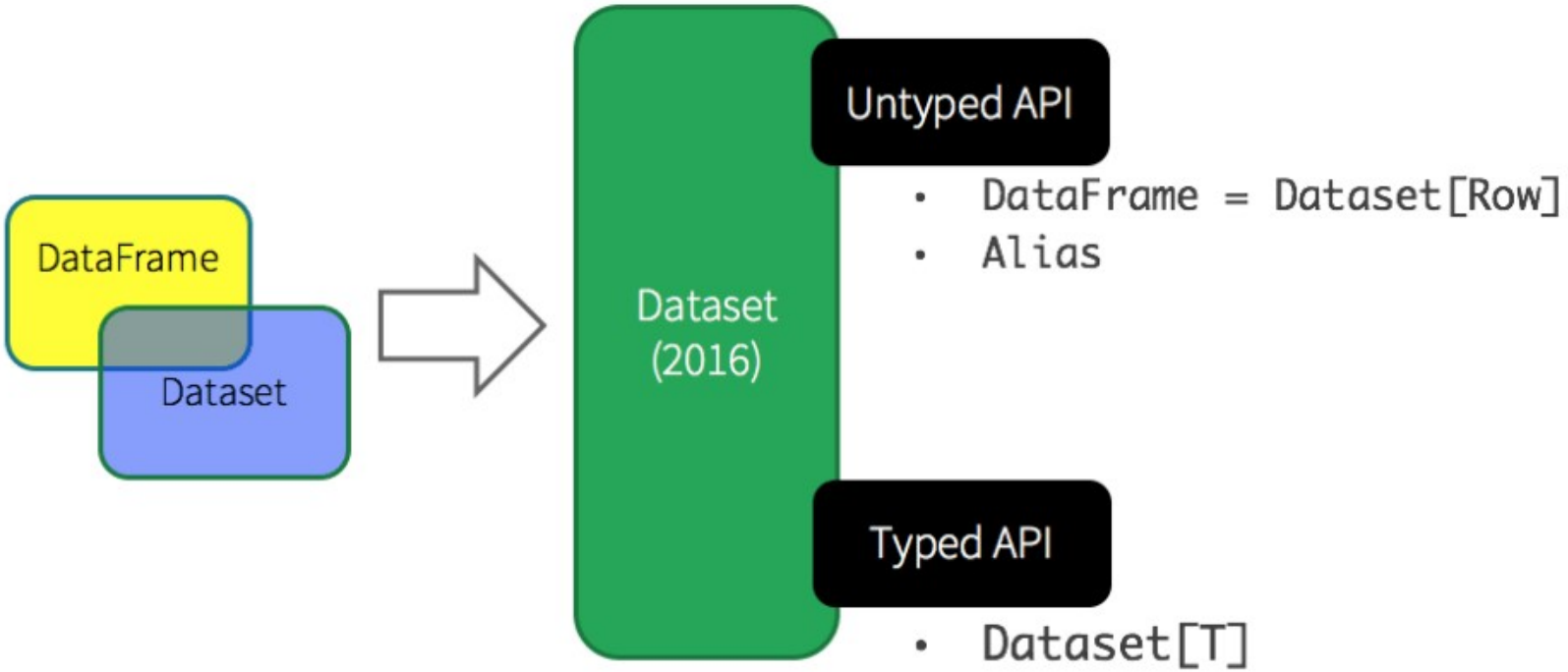
- **در اسپارک ساختیافته، داده‌ها به صورت جداولی متشکل از سطر و ستون در نظر گرفته میشوند.**
- در این جا، داده ها دارای شِما یا ساختار هستند. (Spark SQL / Dataframe / Dataset)



Structured API : فراهم شدن امکان بهینه‌سازی



Unified Apache Spark 2.0 API

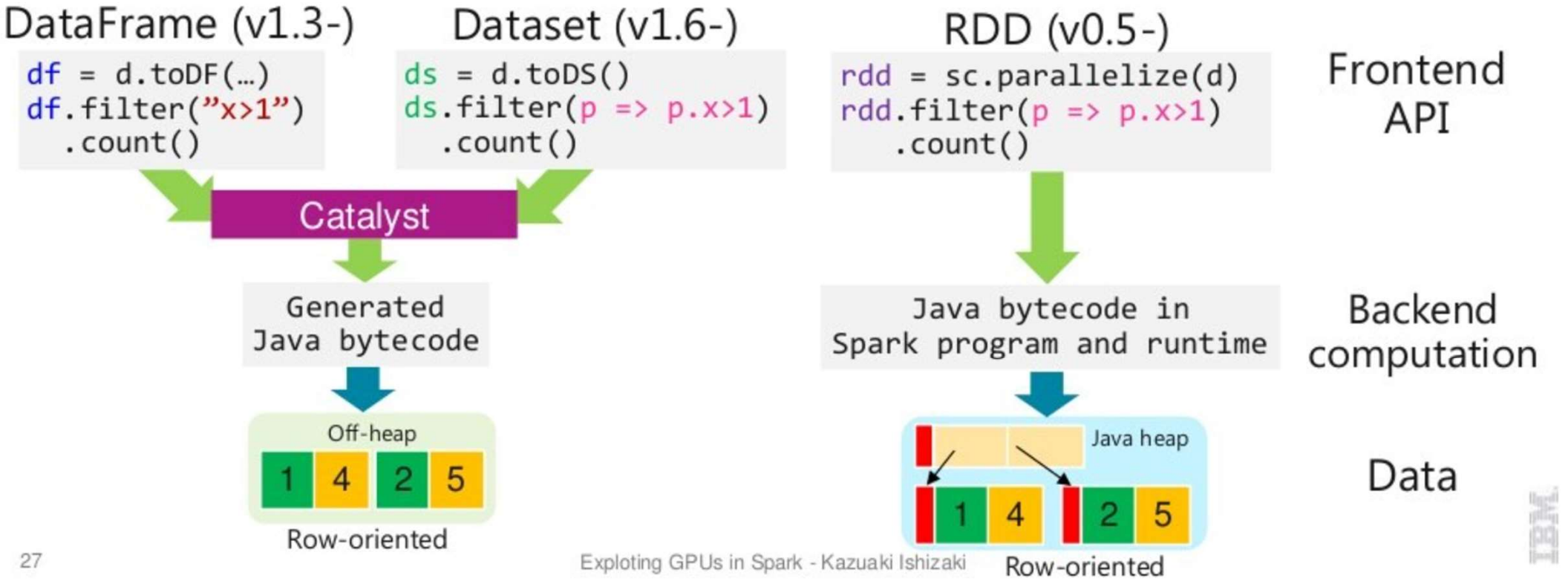


databricks

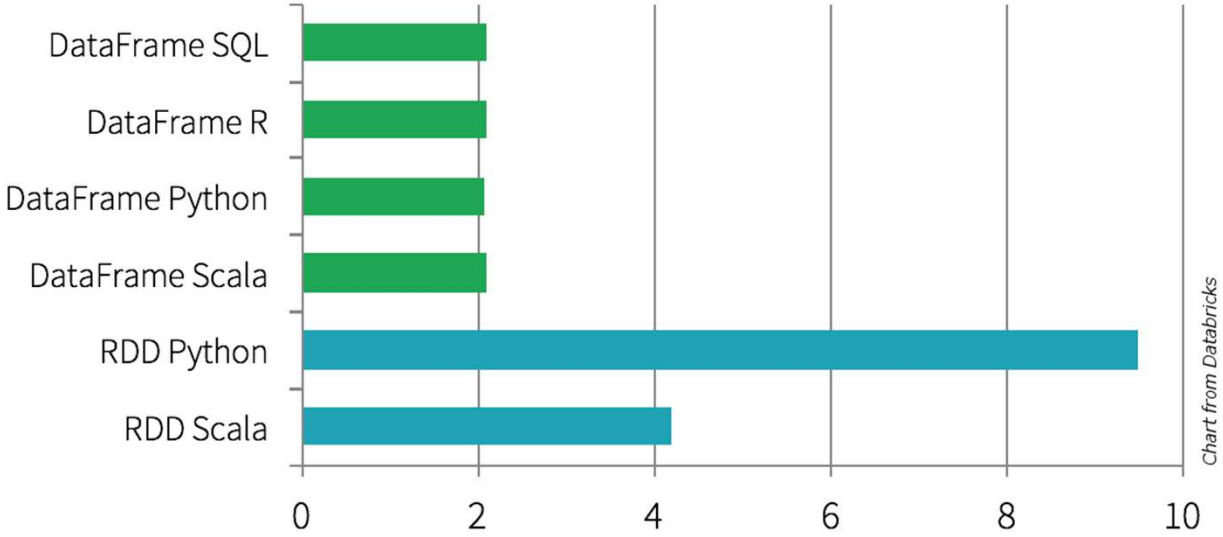
Dataset vs DataFrame vs RDD

- DataFrame (with relational operations) and Dataset (with lambda functions) use Catalyst and row-oriented data representation on off-heap

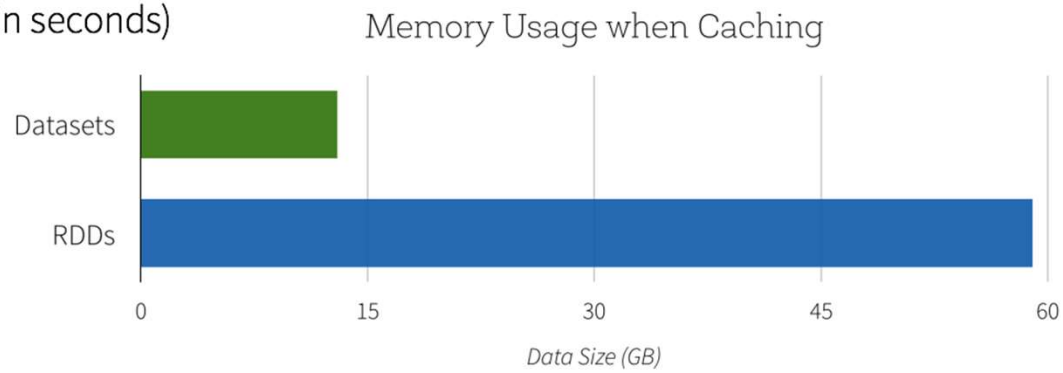
```
case class Pt(x: Int, y: Int)
d = Array(Pt(1, 4), Pt(2, 5))
```



DataFrame vs RDD



Time to aggregate 10 million integer pairs (in seconds)



یک مثال ساده : Dataframe

```
textFile = sc.textFile("hdfs://...")

# Creates a DataFrame having a single column named "line"
df = textFile.map(lambda r: Row(r)).toDF(["line"])
errors = df.filter(col("line").like("%ERROR%"))
# Counts all the errors
errors.count()
# Counts errors mentioning MySQL
errors.filter(col("line").like("%MySQL%")).count()
# Fetches the MySQL errors as an array of strings
errors.filter(col("line").like("%MySQL%")).collect()
```

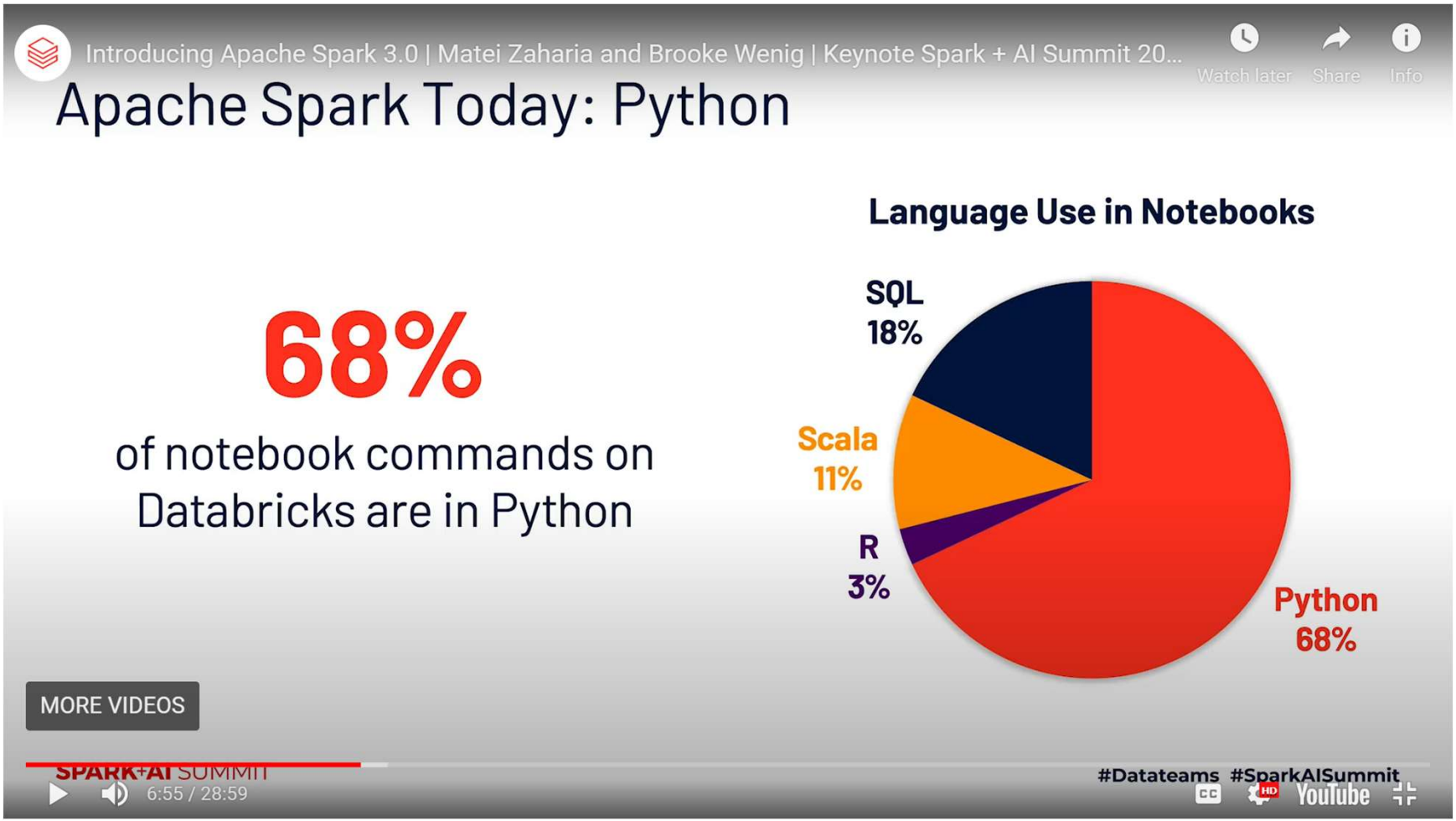
یک مثال ساده : Spark SQL

```
# Load data
df = spark.read.csv('cars.csv', header=True, sep=";")
# Register Temporary Table
df.createOrReplaceTempView("temp")
# Select all data from temp table
spark.sql("select * from temp limit 5").show()
# Select count of data in table
spark.sql("select count(*) as total_count from temp").show()
```

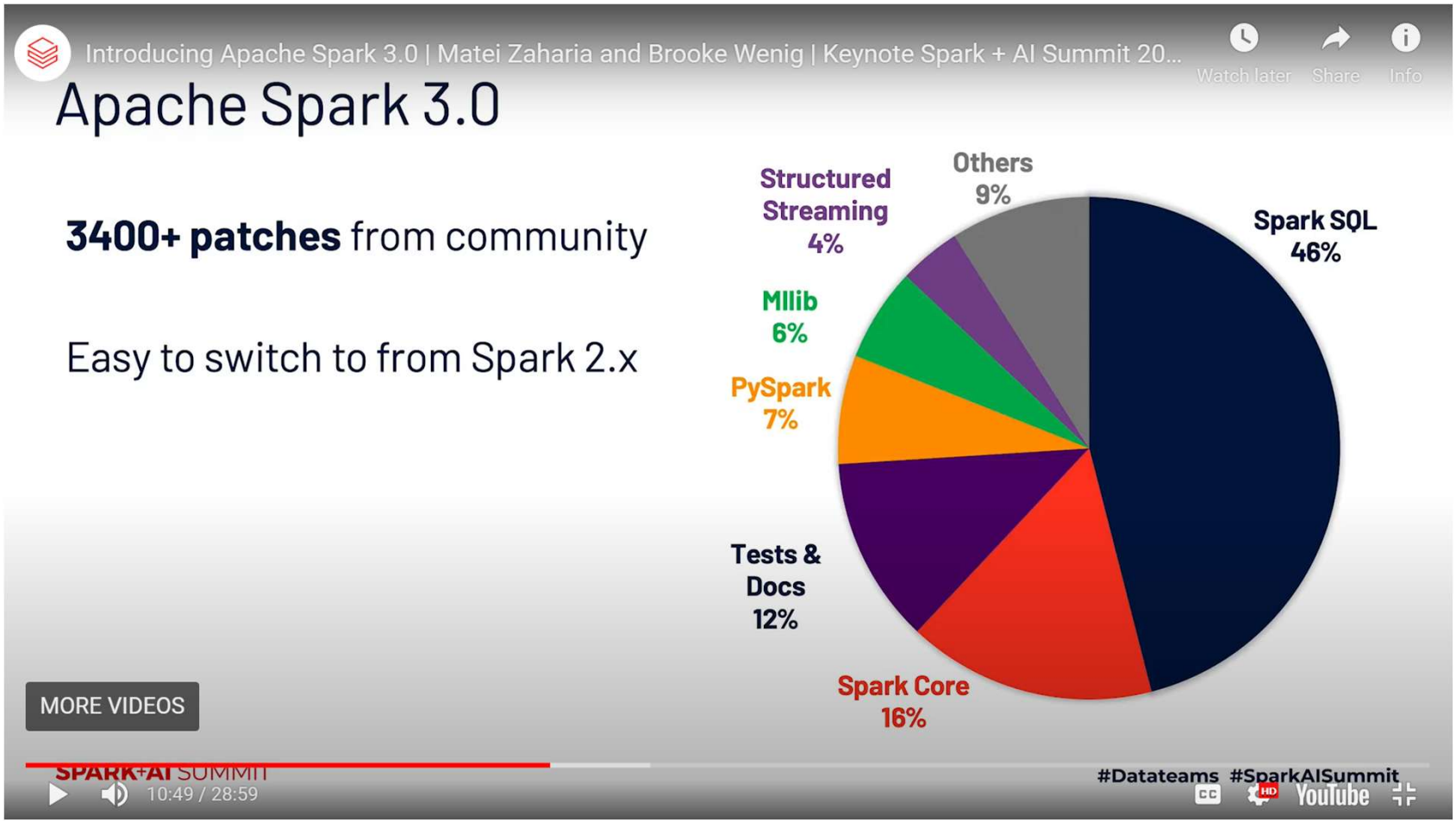
```
+--------------------+----+---------+------------+----------+------+------------+-----+------+
|                 Car| MPG|Cylinders|Displacement|Horsepower|Weight|Acceleration|Model|Origin|
+--------------------+----+---------+------------+----------+------+------------+-----+------+
|Chevrolet Chevell...|18.0|        8|       307.0|     130.0| 3504.|        12.0|   70|    US|
|   Buick Skylark 320|15.0|        8|       350.0|     165.0| 3693.|        11.5|   70|    US|
|  Plymouth Satellite|18.0|        8|       318.0|     150.0| 3436.|        11.0|   70|    US|
|       AMC Rebel SST|16.0|        8|       304.0|     150.0| 3433.|        12.0|   70|    US|
|         Ford Torino|17.0|        8|       302.0|     140.0| 3449.|        10.5|   70|    US|
+--------------------+----+---------+------------+----------+------+------------+-----+------+
```

```
+-----------+
|total_count|
+-----------+
|        406|
+-----------+
```

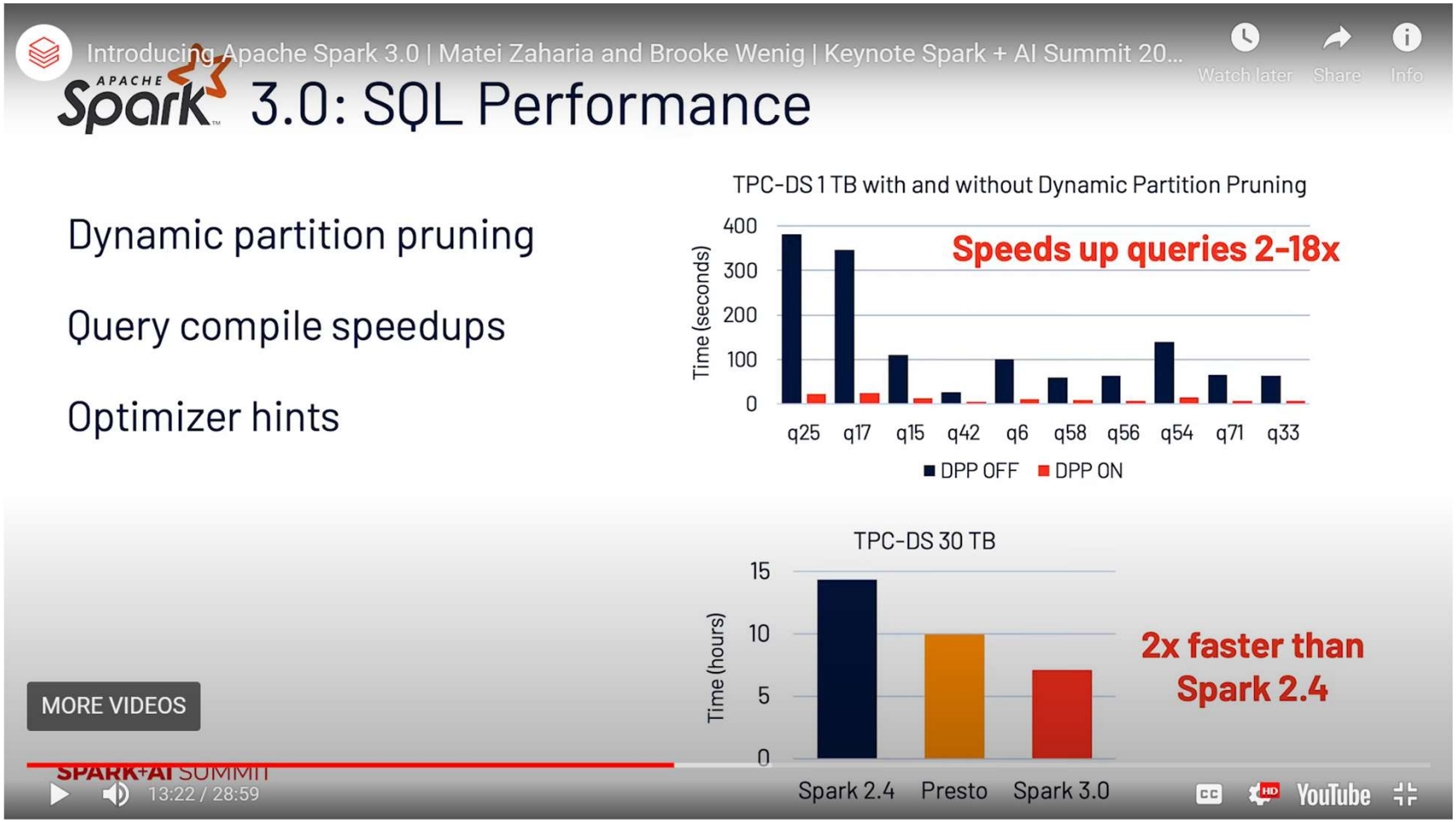
کدام زبان محبوب تر است ؟



اسپارک ۳ : تغییرات انجام شده



اسپارک ۳ : تاثیر بهبودهای انجام شده بر SQL



اسپارک ۳ : سایر بهبودها

Introducing Apache Spark 3.0 | Matei Zaharia and Brooke Wenig | Keynote Spark + AI Summit 20...

Watch later Share Info

Apache Spark 3.0: Other Features

- Structured Streaming UI
- Observable stream metrics
- SQL reference guide
- Data Source v2 API
- GPU scheduling support


```scala
val stream = spark.readStream...

stream.observe("data_quality", count($"error") / count(lit(1))).writeStream...

spark.streams.addListener(new StreamingQueryListener() {
  override def onQueryProgress(event: QueryProgressEvent): Unit = {
    event.progress.observedMetrics.get("data_quality").foreach {
      case Row(pct_parse_errors: Double) if pct_parse_errors > 0.05 => // Trigger alert
      case _ => // OK
    }
  }
})
```

MORE VIDEOS

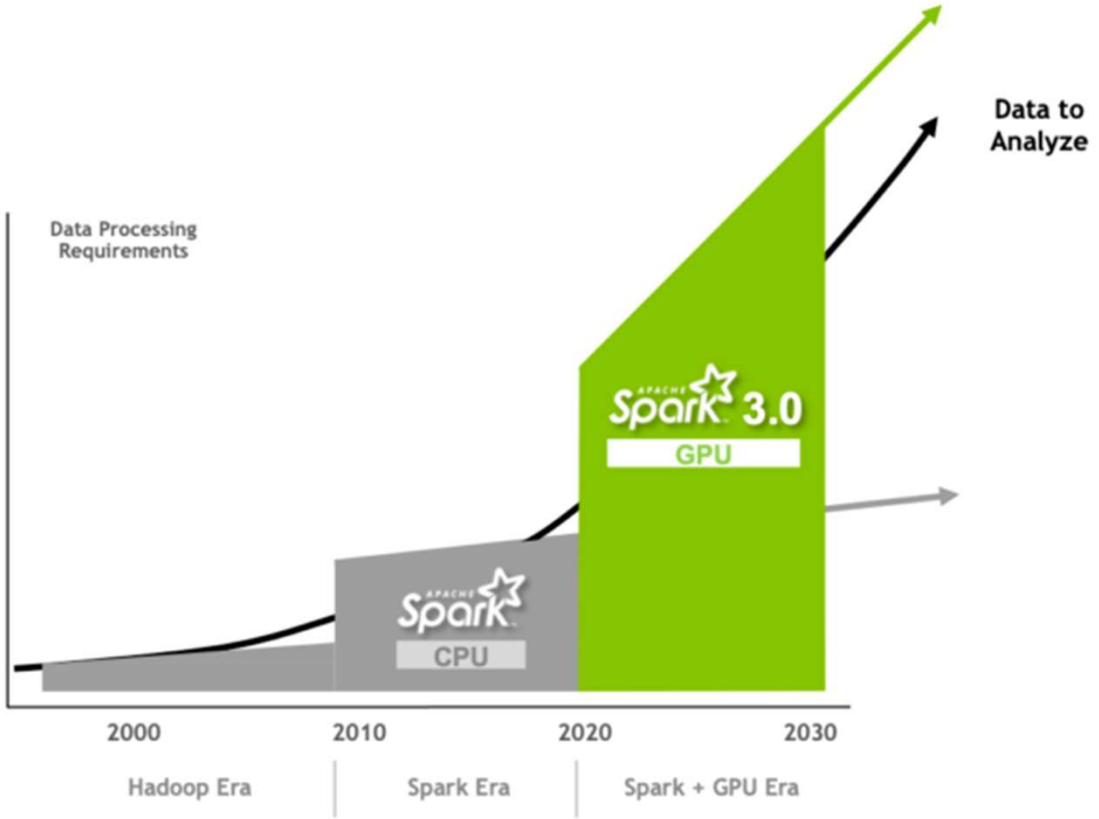
SPARK+AI SUMMIT

15:31 / 28:59

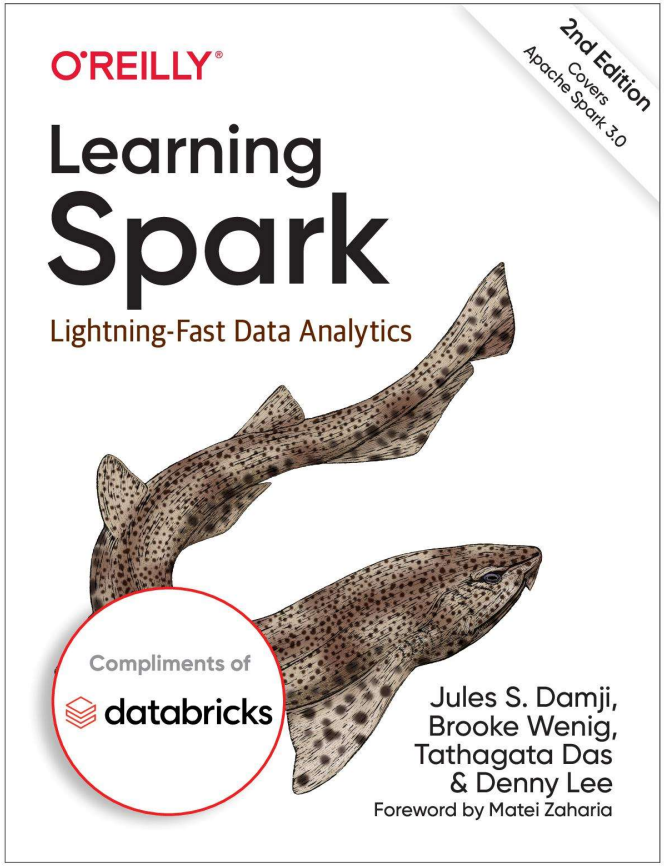
#Datateams #SparkAISummit

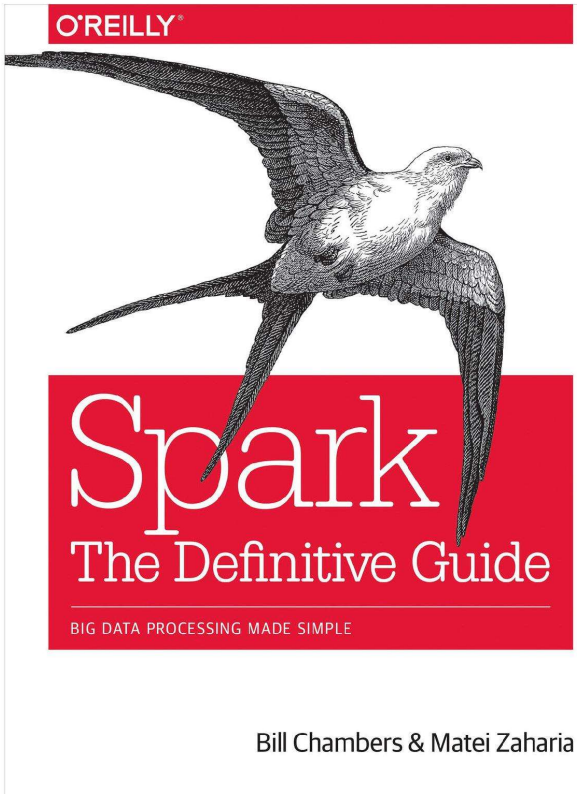
CC HD YouTube

اسپارک ۳ : استفاده از GPU



منابع


O'REILLY
2nd Edition
Covers Apache Spark 3.0
Learning
Spark
Lightning-Fast Data Analytics
Compliments of
databricks
Jules S. Damji,
Brooke Wenig,
Tathagata Das
& Denny Lee
Foreword by Matei Zaharia


O'REILLY
Spark
The Definitive Guide
BIG DATA PROCESSING MADE SIMPLE
Bill Chambers & Matei Zaharia