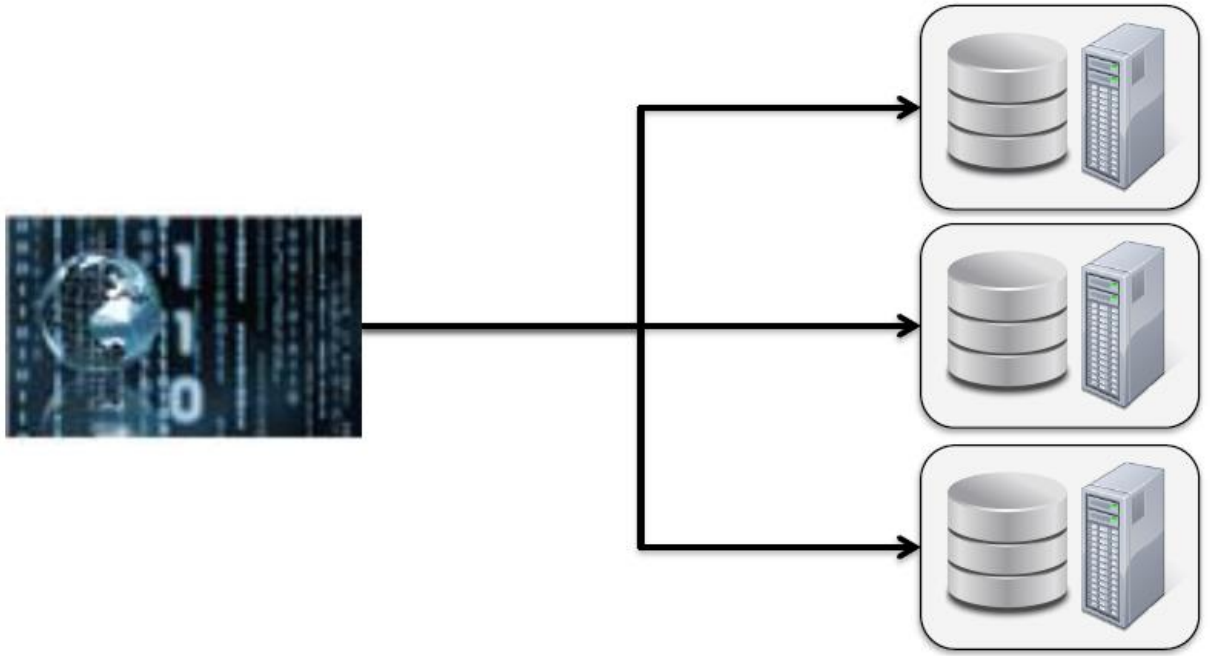
APACHE SPARK

Big Data Processing

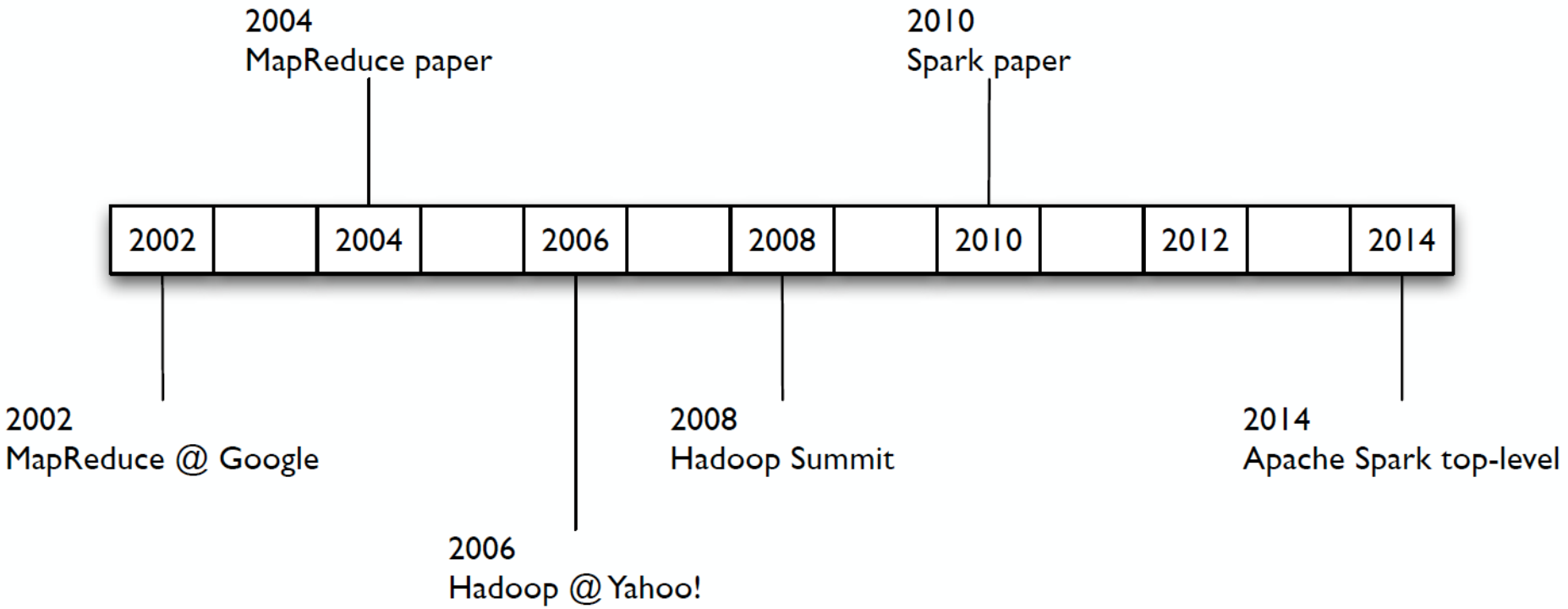
- **Hadoop introduced a radical new approach based on two key concepts**
  - Distribute the data when it is stored
  - Run computation where the data is
- **Spark takes this new approach to the next level**
  - Data is distributed in memory



Introducing Apache Spark

- **Apache Spark** is a fast, general engine for large-scale data processing on a cluster
- **Originally developed at AMPLab at UC Berkeley**
  - Started as a research project in 2009
- **Open source Apache project**
  - Committers from Cloudera, Yahoo, Databricks, UC Berkeley, Intel, Groupon, …
  - One of the most active and fastest-growing Apache projects
  - Cloudera provides enterprise-level support for Spark

Brief History



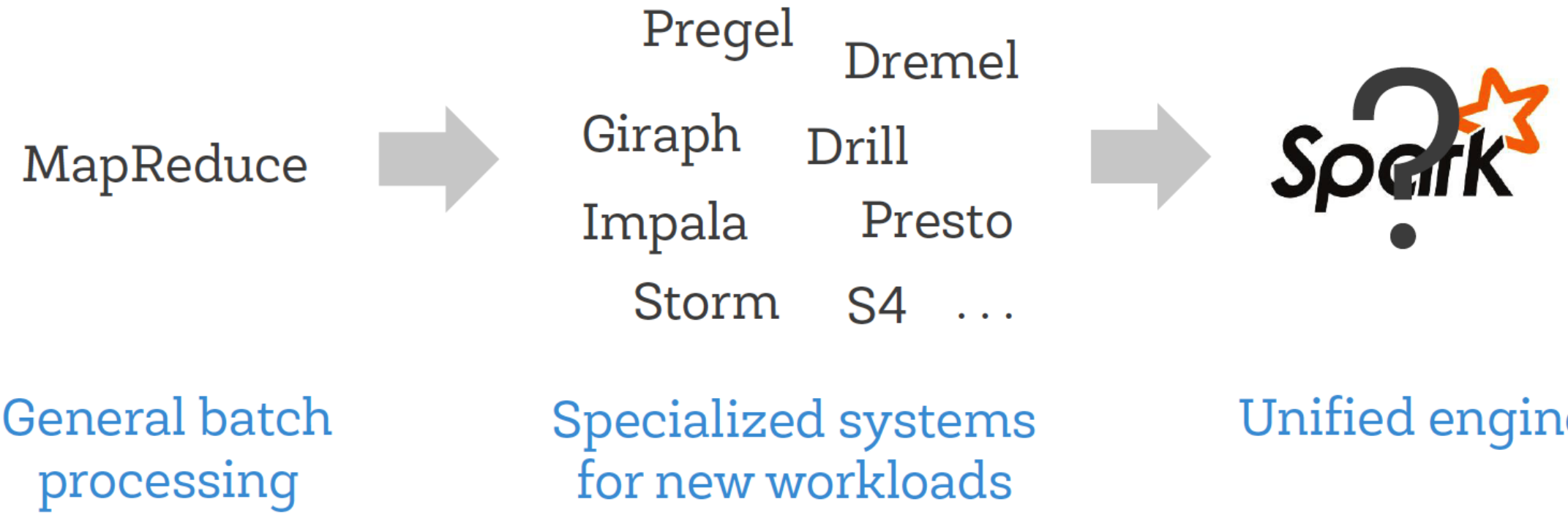
Beyond MapReduce

MapReduce was great for batch processing, but users quickly needed to do more:

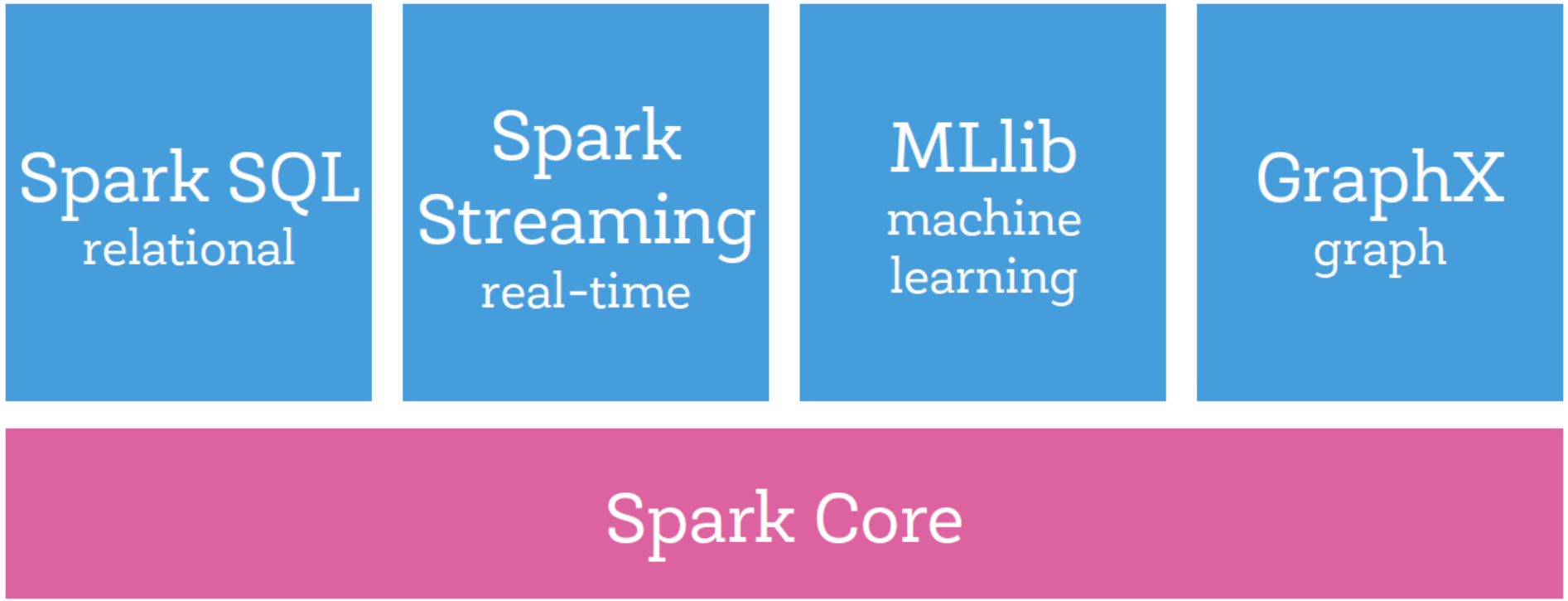
- More complex, multi-pass algorithms
- More interactive ad-hoc queries
- More real-time stream processing

Result: many *specialized* systems for these workloads

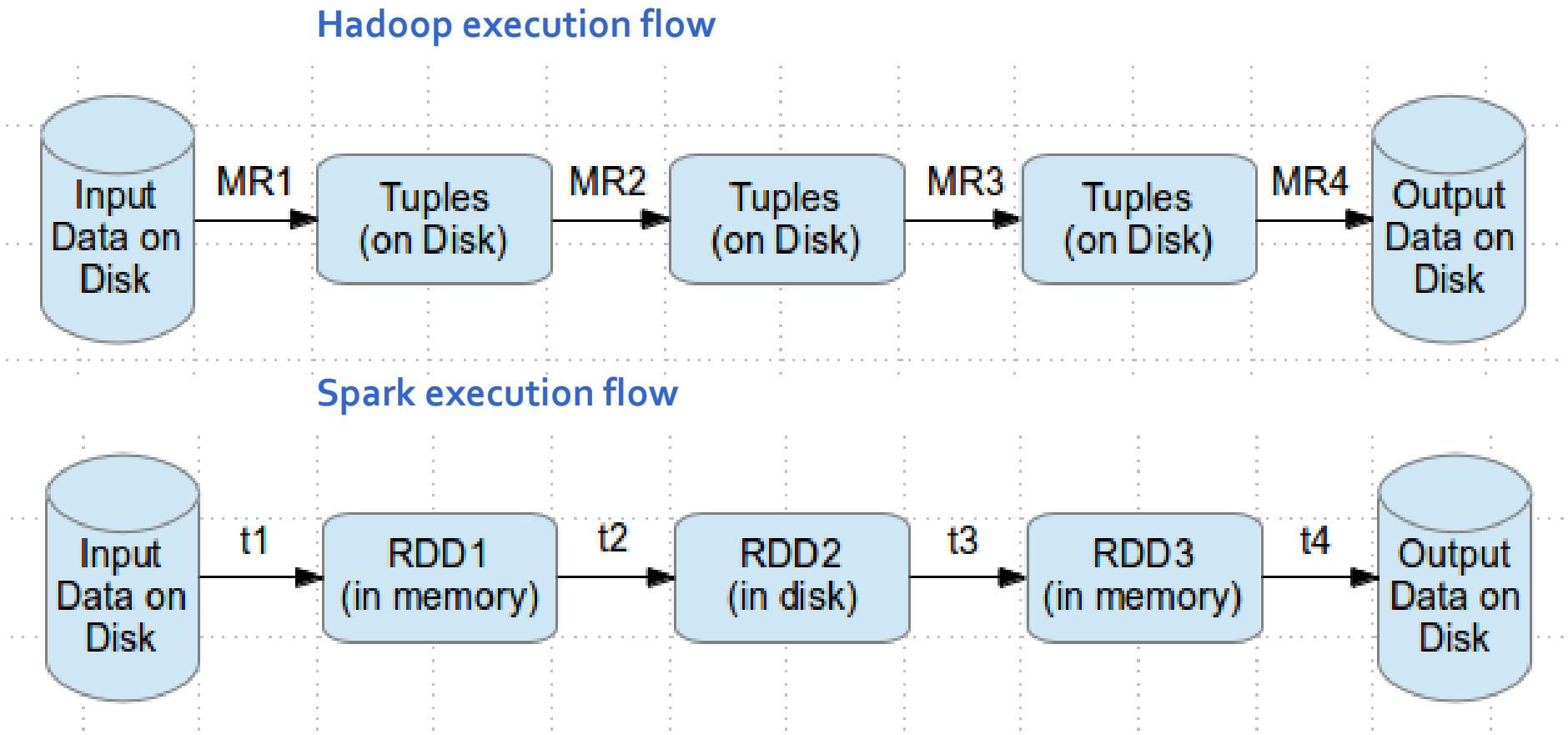
BigData Systems Today



Libraries Built on Spark



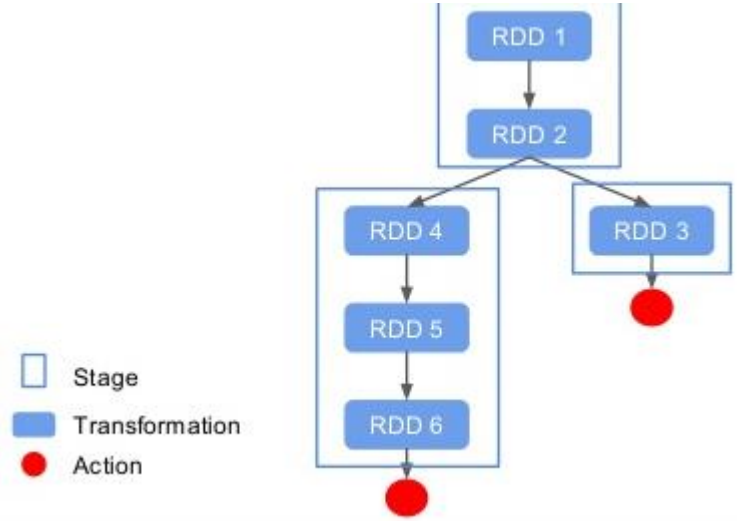
Why Spark


Hadoop execution flow
Input Data on Disk
MR1
Tuples (on Disk)
MR2
Tuples (on Disk)
MR3
Tuples (on Disk)
MR4
Output Data on Disk
Spark execution flow
Input Data on Disk
t1
RDD1 (in memory)
t2
RDD2 (in disk)
t3
RDD3 (in memory)
t4
Output Data on Disk

Pillars of Spark

Direct Acyclic Graph – sequence of computations performed on data

- *Node* – RDD partition
- *Edge* – transformation on top of data
- *Acyclic* – graph cannot return to the older partition
- *Direct* – transformation is an action that transitions data partition state (from A to B)



RDD (Resilient Distributed Dataset)

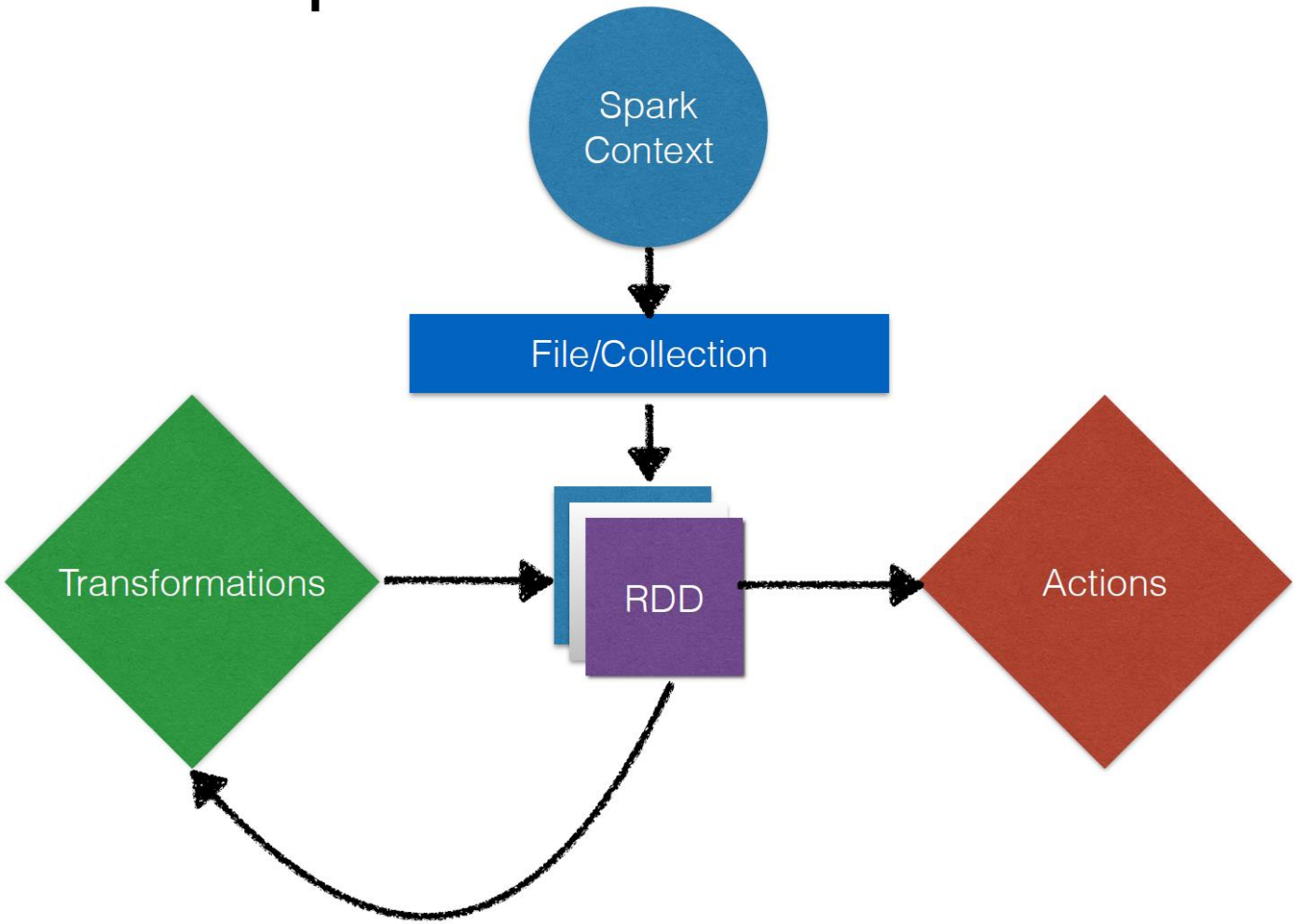
- **RDD (Resilient Distributed Dataset)**
  - Resilient – if data in memory is lost, it can be recreated
  - Distributed – stored in memory across the cluster
  - Dataset – initial data can come from a file or be created programmatically
- **RDDs are the fundamental unit of data in Spark**
- **Most Spark programming consists of performing operations on RDDs**

Spark Internals

Now what?

Lazy Evaluation

Spark Internals

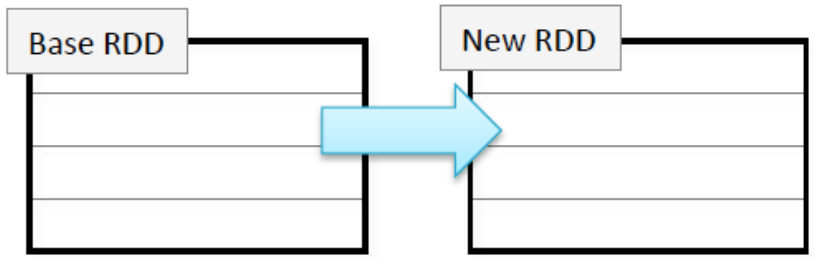


Creating an RDD

- **Three ways to create an RDD**
  - From a file or set of files
  - From data in memory
  - From another RDD

RDD Operations: Transformations

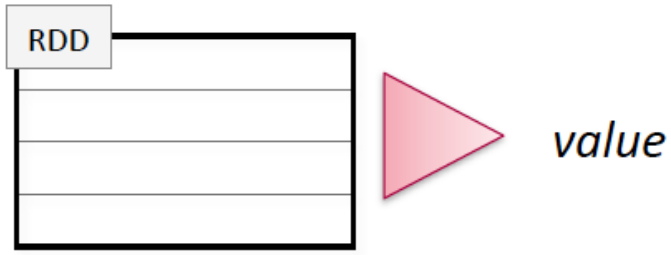
- **Transformations create a new RDD from an existing one**
- **RDDs are immutable**
  - Data in an RDD is never changed
  - Transform in sequence to modify the data as needed



- **Some common transformations**
  - **`map(`***`function`***`)`** – creates a new RDD by performing a function on each record in the base RDD
  - **`filter(`***`function`***`)`** – creates a new RDD by including or excluding each record in the base RDD according to a boolean function

RDD Operations: Actions

- **Some common actions**
  - **`count()`** – return the number of elements
  - **`take(`***`n`***`)`** – return an array of the first *n* elements
  - **`collect()`** – return an array of all elements
  - **`saveAsTextFile(`***`filename`***`)`** – save to text file(s)




```
> mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for line in mydata.take(2):
    print line
I've never seen a purple cow.
I never hope to see one;
```

```
> val mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for (line <- mydata.take(2))
    println(line)
I've never seen a purple cow.
I never hope to see one;
```

Example Application

```
val sc = new SparkContext(...)

val file = sc.textFile("hdfs://...")

val errors = file.filter(_.contains("ERROR"))

errors.cache()

errors.count()
```

Resilient distributed datasets (RDDs)

Action

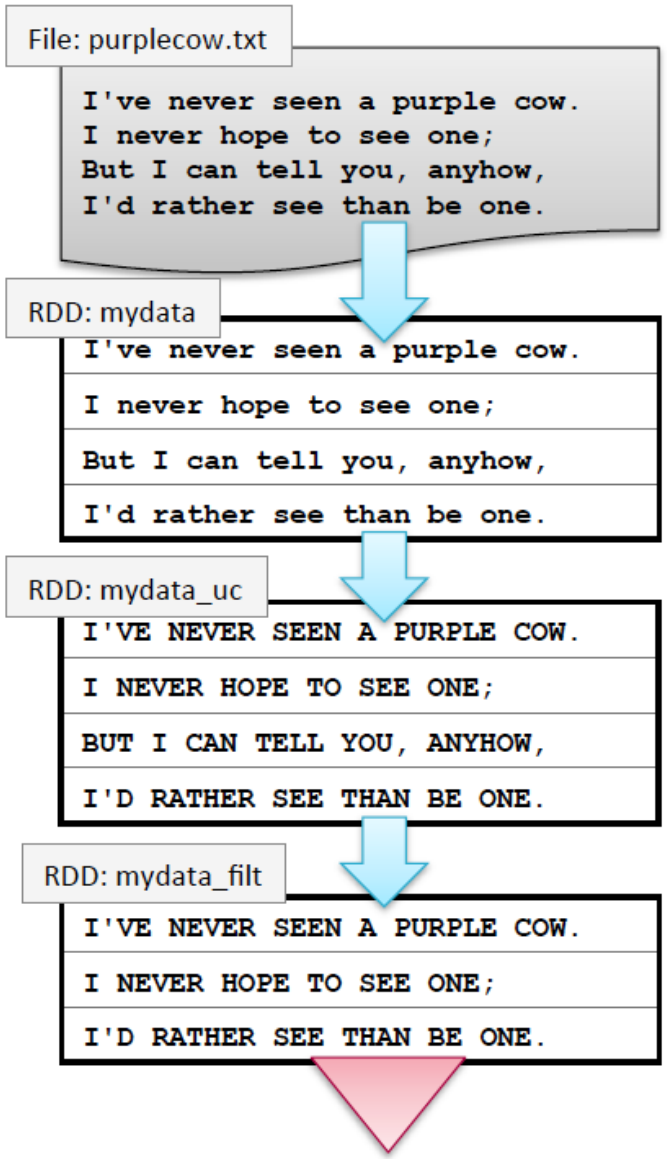
File-Based RDDs

- **For file-based RDDS, use `SparkContext.textFile`**
  - Accepts a single file, a wildcard list of files, or a comma-separated list of files
  - Examples
    - `sc.textFile("myfile.txt")`
    - `sc.textFile("mydata/*.log")`
    - `sc.textFile("myfile1.txt,myfile2.txt")`
  - Each line in the file(s) is a separate record in the RDD
- **Files are referenced by absolute or relative URI**
  - Absolute URI: `file:/home/training/myfile.txt`
  - Relative URI (uses default file system): `myfile.txt`

Lazy Execution

- **Data in RDDs is not processed until an *action* is performed**
  - RDD is materialized in memory upon the first action that uses it

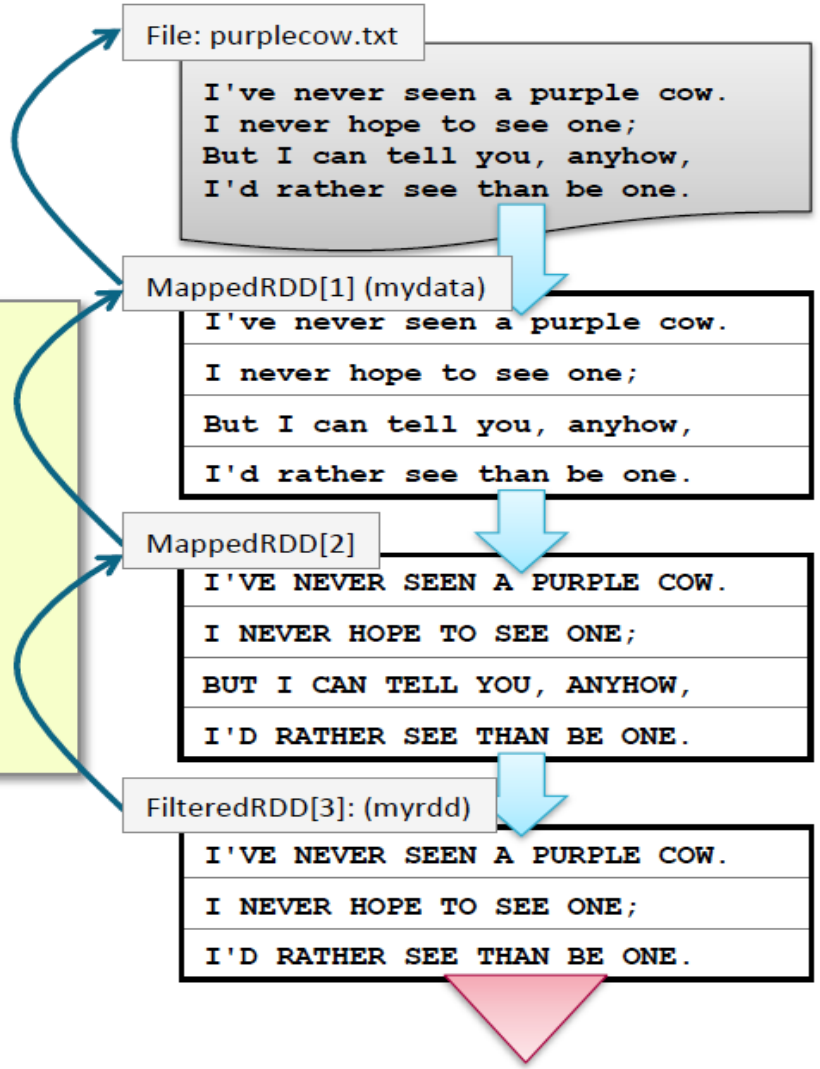
```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line:
  line.upper())
> mydata_filt = \
     mydata_uc.filter(lambda line: \
     line.startswith('I'))
> mydata_filt.count()
3
```



Lineage Example (7)

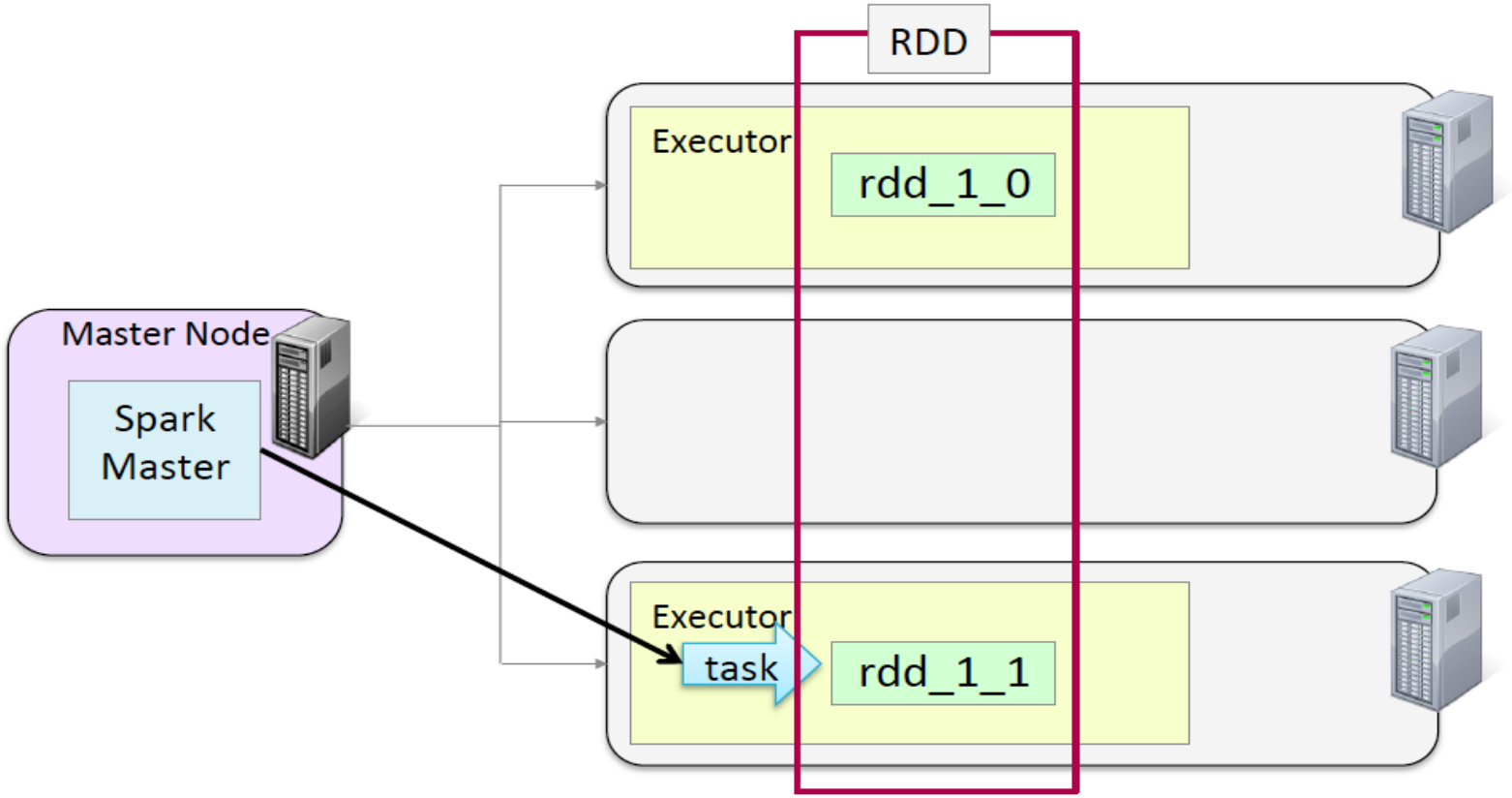
- **Each action re-executes the lineage transformations starting with the base**
  - By default

```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper())\
  .filter(lambda s:s.startswith('I'))
> myrdd.count()
3
> myrdd.count()
3
```



RDD Fault-Tolerance (2)

- **The SparkMaster starts a new task to recompute the partition on a different node**



Chaining Transformations

- **Transformations may be chained together**

```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda line: line.upper())
> mydata_filt = mydata_uc.filter(lambda line: line.startswith('I'))
> mydata_filt.count()
3
```

is exactly equivalent to

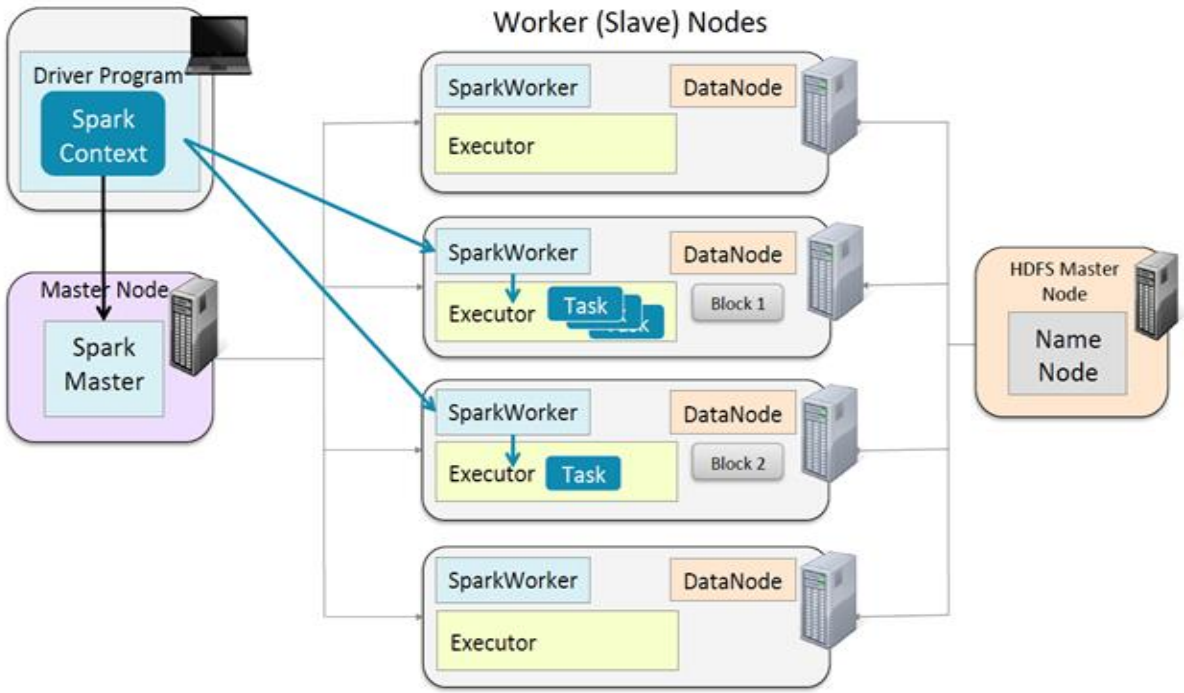
```
> sc.textFile("purplecow.txt").map(lambda line: line.upper()) \
  .filter(lambda line: line.startswith('I')).count()
3
```

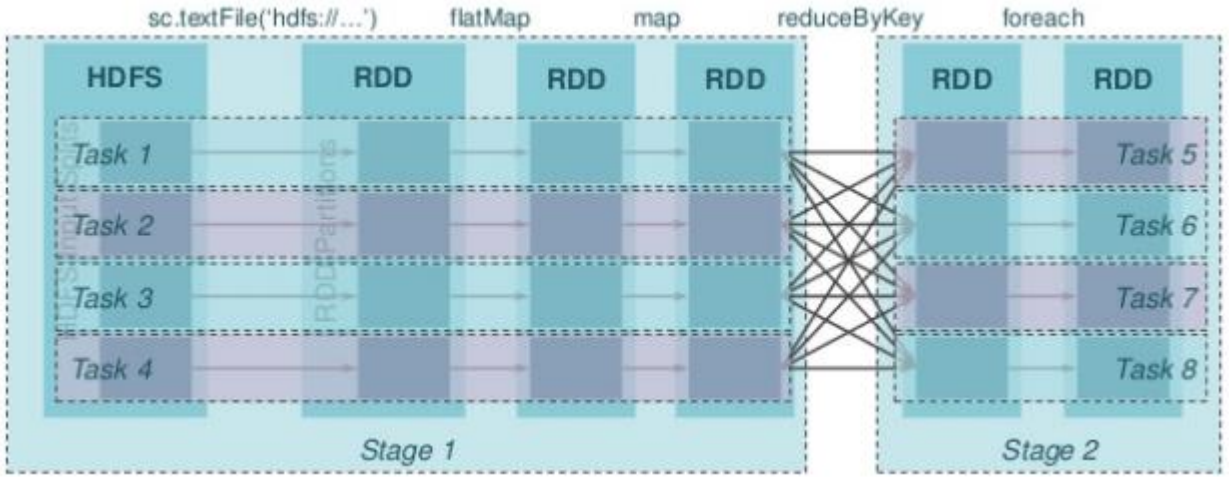
Terminologies

- **Application Jar**
  - User Program and its dependencies except Hadoop & Spark Jars bundled into a Jar file
- **Driver Program**
  - The process to start the execution (main() function)
- **Cluster Manager**
  - An external service to manage resources on the cluster (standalone manager, YARN, Apache Mesos)
- **Deploy Mode**
  - **cluster** : Driver inside the cluster
  - **client** : Driver outside of Cluster

Terminology (Contd.)

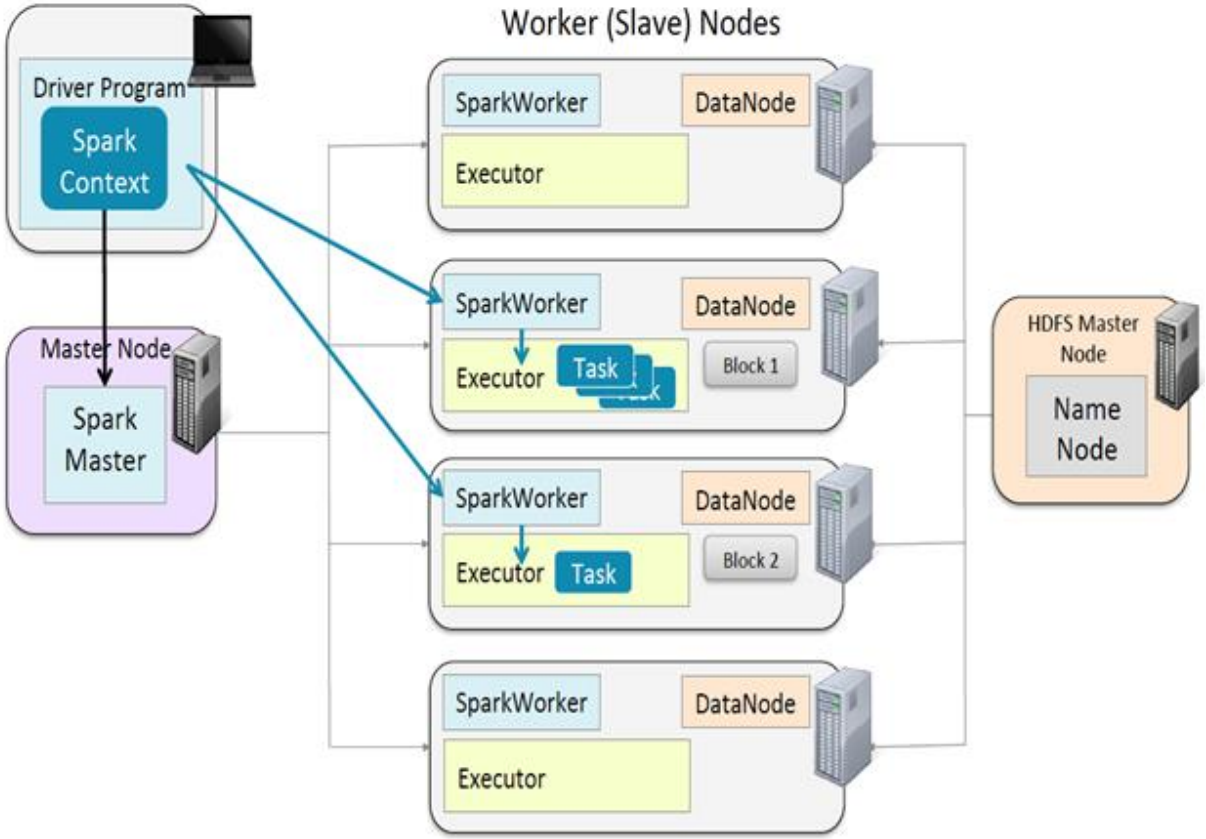
- **Worker Node :** Node that run the application program in cluster
- **Executor**
  - Process launched on a worker node, that runs the Tasks
  - Keep data in memory or disk storage
  - Cache Memory & Swap storage for RDD lineage
- **Job**
  - Consists multiple tasks, Created based on a Action
- **Stage :** Each Job is divided into a smaller set of tasks called Stages that is sequential and depend on each other
- **Task :** A unit of work that will be sent to executor.
- **Partitions:** Data unit that will be handled parallel, Same as Blocks in HDFS.



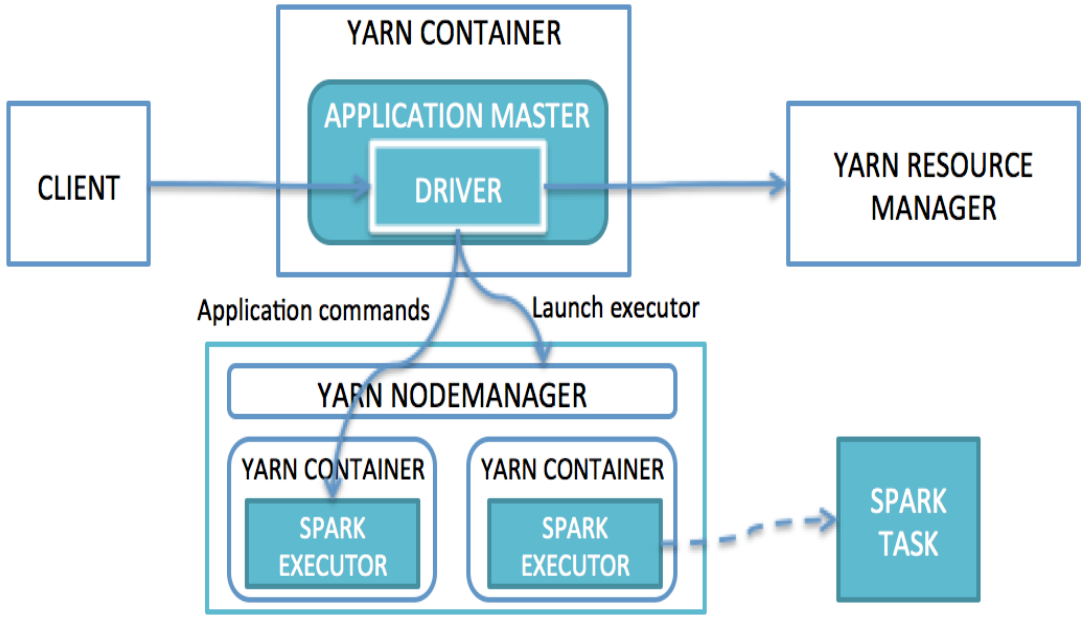


Spark Cluster Deployment

Standalone Spark



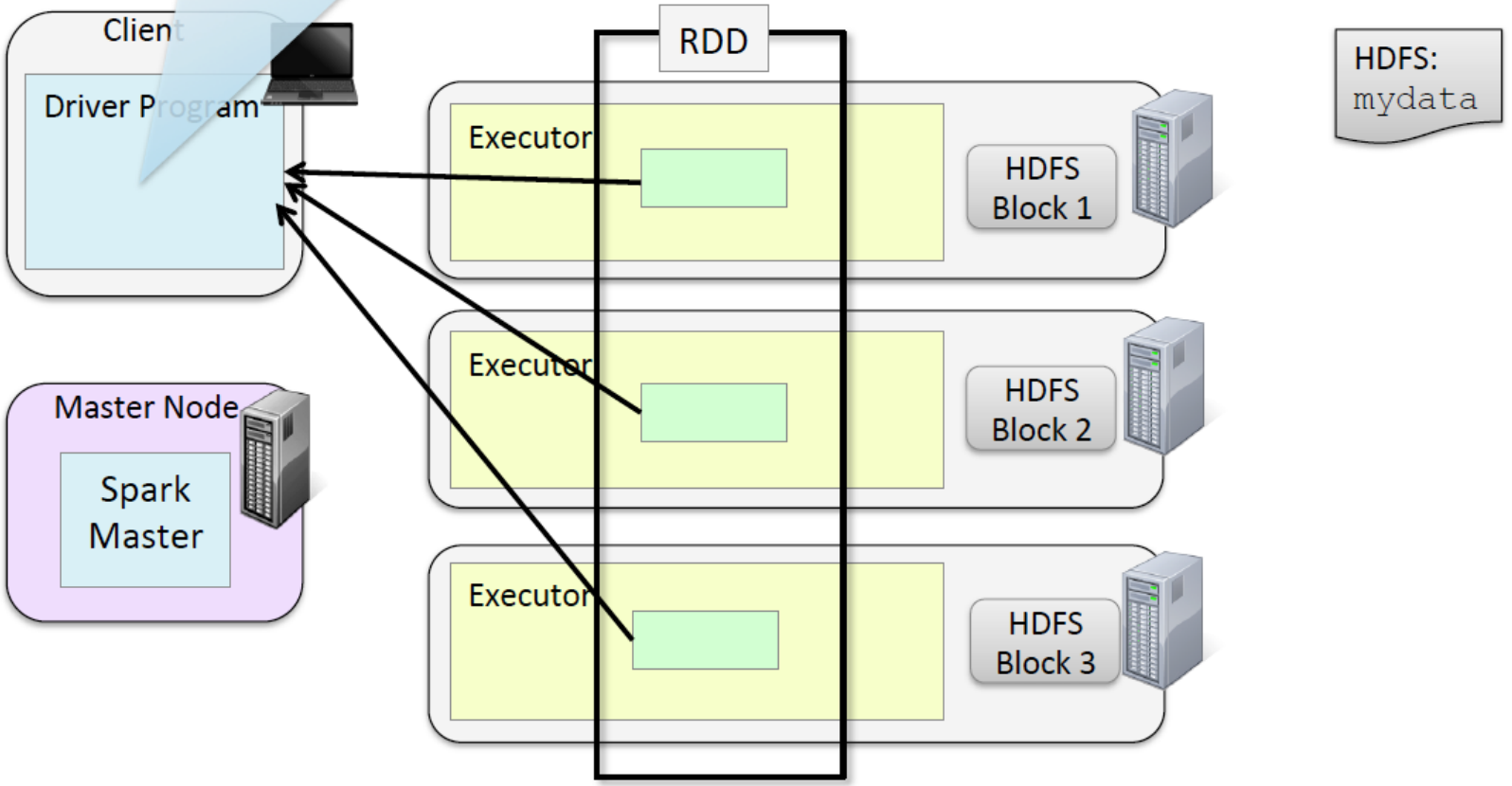
Spark on YARN



HDFS and Data Locality

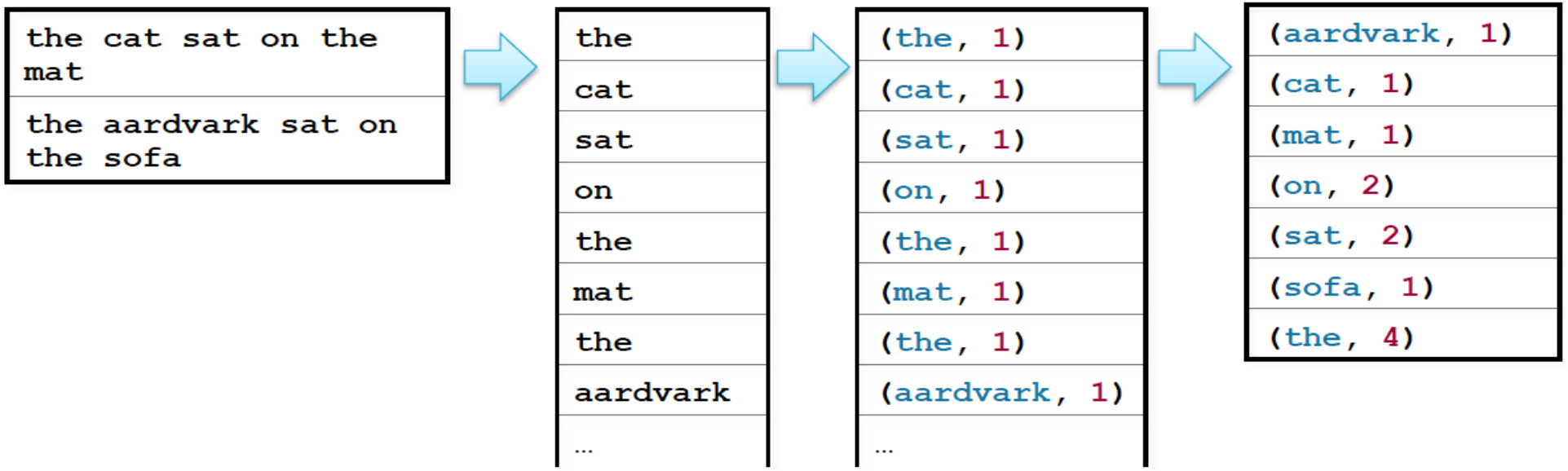
```
sc.textFile("hdfs://…mydata…").collect()
```

Data is distributed across executors until an action returns a value to the driver



Example: Word Count (4)

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```



Spark SQL

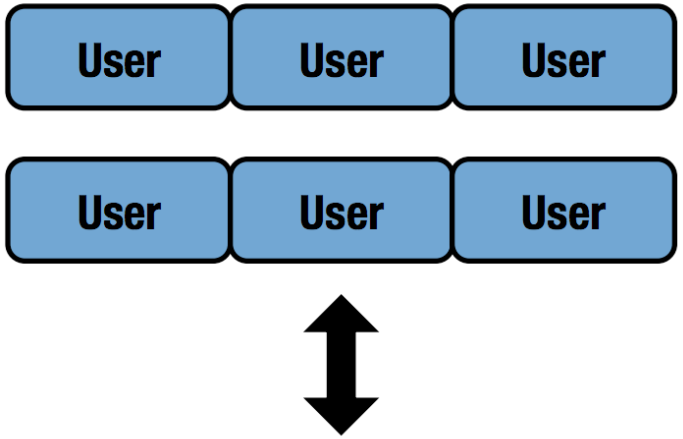
Represents tables as RDDs

Tables = Schema + Data

Adding Schema to RDD

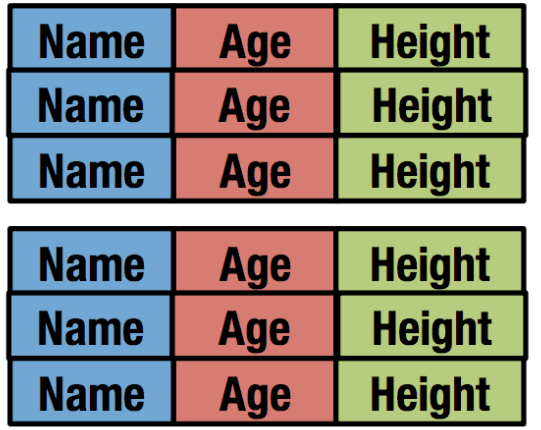
Spark + RDDs

Functional transformations on partitioned collections of **opaque** objects.



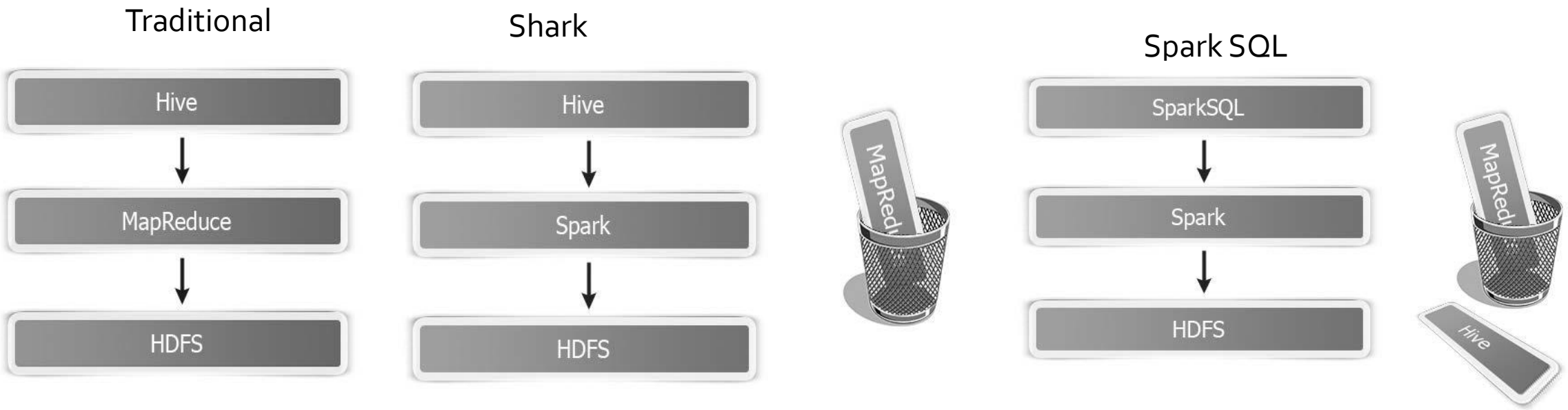
SQL + SchemaRDDs

Declarative transformations on partitioned collections of **tuples.**



Unified Data Abstraction

Evolution of Spark SQL



Using Spark SQL

`SQLContext`

- Entry point for all SQL functionality
- Wraps/extends existing spark context

```
from pyspark.sql import SQLContext
sqlCtx = SQLContext(sc)
```

Sample data set

A text file filled with people's names and ages:

```
Michael, 30

Andy, 31

Justin Bieber, 19

…
```

RDDs into Relations (Python)

```
# Load a text file and convert each line to a dictionary.
lines = sc.textFile("examples/…/people.txt")

parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p:{"name": p[0],"age": int(p[1])})

# Infer the schema, and register the SchemaRDD as a table
peopleTable = sqlCtx.inferSchema(people)
peopleTable.registerAsTable("people")
```

Querying using SQL

```
# SQL can be run over SchemaRDDs that have been registered
# as a table.
teenagers = sqlCtx.sql("""
  SELECT name FROM people WHERE age >= 13 AND age <= 19""")

# The results of SQL queries are RDDs and support all the normal
# RDD operations.
teenNames = teenagers.map(lambda p: "Name: " + p.name)
```

Reading Data stored in Hive

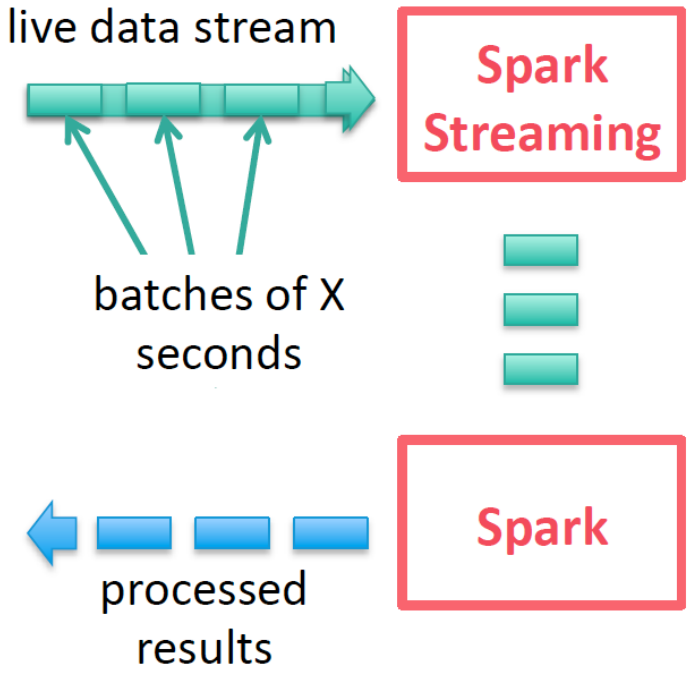
```
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)

hiveCtx.hql("""
  CREATE TABLE IF NOT EXISTS src (key INT, value STRING)""")

hiveCtx.hql("""
  LOAD DATA LOCAL INPATH 'examples/…/kv1.txt' INTO TABLE src""")

# Queries can be expressed in HiveQL.
results = hiveCtx.hql("FROM src SELECT key, value").collect()
```

SPARK STREAMING

Spark Streaming

Run a streaming computation as a series of very small, deterministic batch jobs

- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



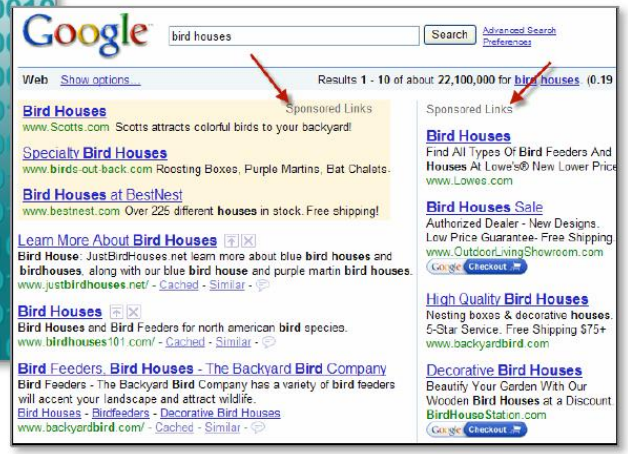
Why Spark Streaming?

Many big-data applications need to process large data streams in realtime

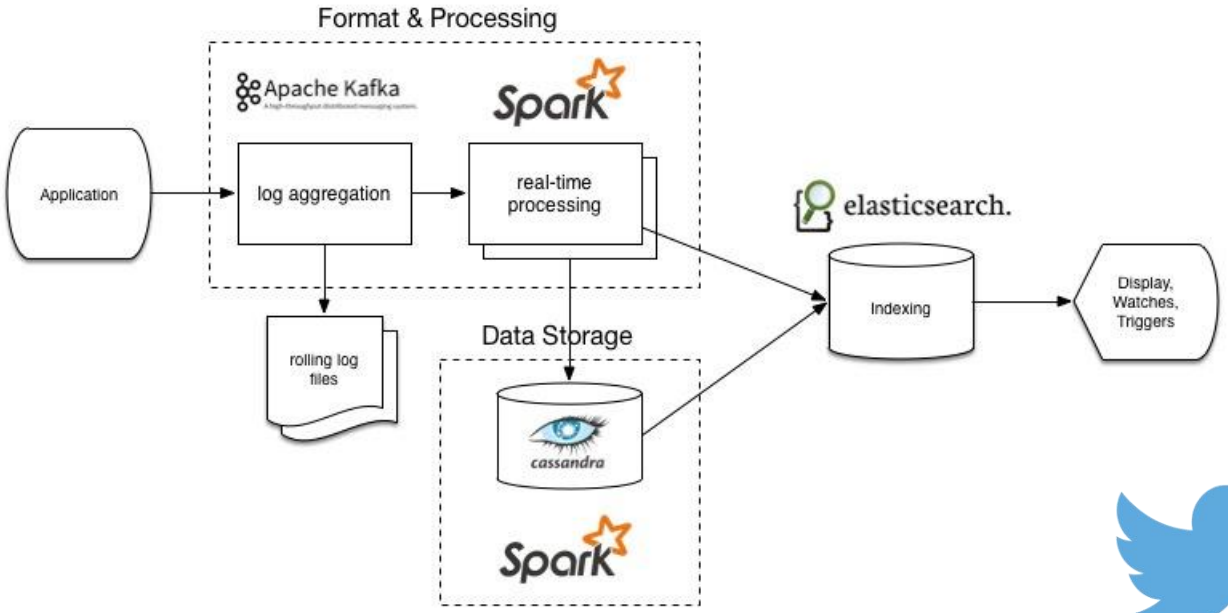
Website monitoring

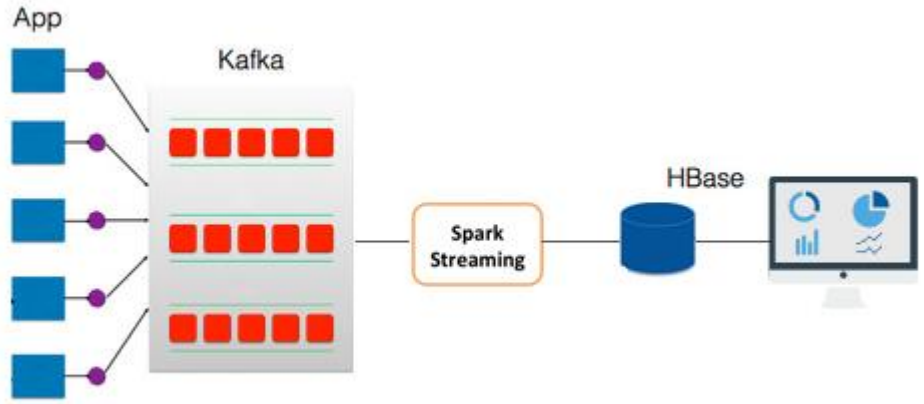
Fraud detection

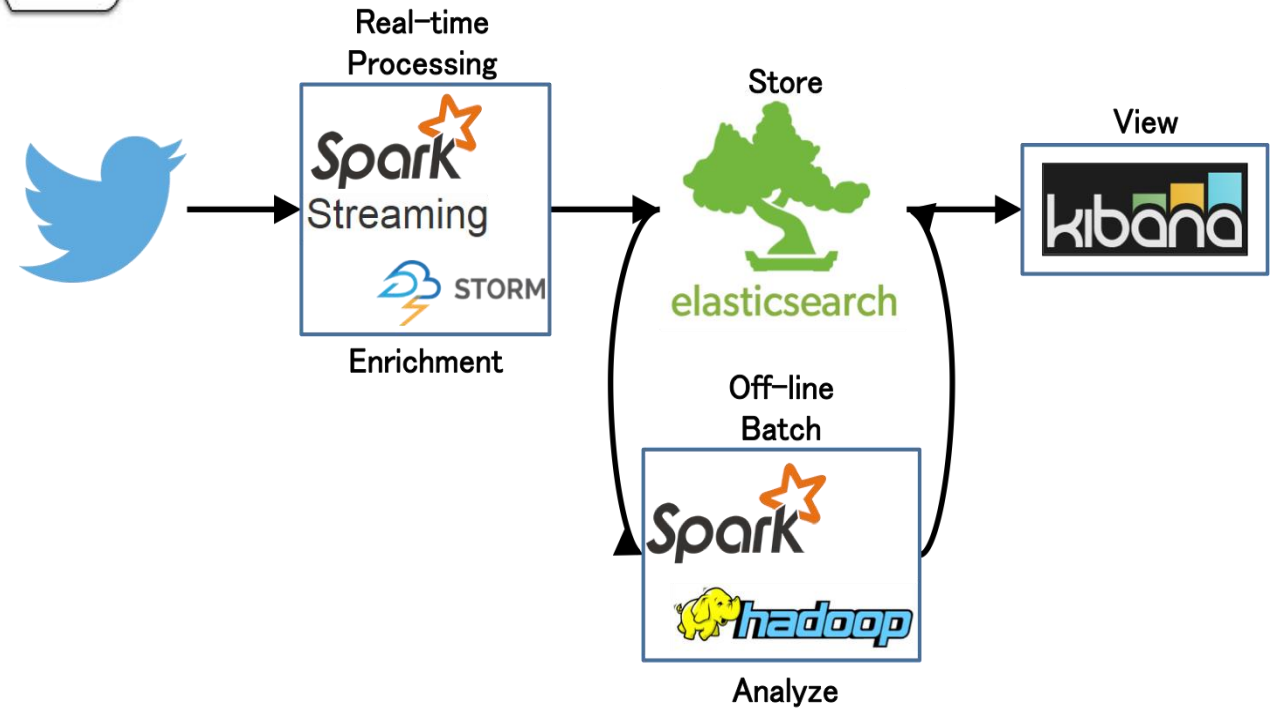
Ad monetization



Realtime Streaming Usecases







WORKOUTS