Distributed computing for Bigdata

Saeed Sharifian

Big Data for IoT

Telephony

- CDR processing
- Social analysis
- Churn prediction
- Geomapping

Transportation

- Intelligent traffic management
- Automotive telematics

Energy and utilities

- Transactive control
- Phasor measurement unit
- Downhole sensor monitoring

Health and life sciences

- ICU monitoring
- Epidemic early warning system
- Remote healthcare monitoring

Natural systems

- Wildlife management
- Water management

Law enforcement, defense and cybersecurity

- Real-time multimodal surveillance
- Situational awareness
- Cybersecurity detection

Stock market

- Impact of weather on securities prices
- Market data analysis at ultra-low latencies
- Momentum calculator

Fraud prevention

- Multi-party fraud detection
- Real-time fraud prevention

eScience

- Space weather prediction
- Transient event detection
- Synchrotron atomic research
- Genomic research

Other

- Manufacturing
- Text analysis
- ERP for commodities

Large scale computing

- **Large-scale computing** for **data mining** problems on **commodity hardware**
- **Challenges:**
  - **How do you distribute computation?**
  - **How can we make it easy to write distributed programs?**
  - **Machines fail:**
    - One server may stay up 3 years (1,000 days)
    - If you have 1,000 servers, expect to lose 1/day
    - With 1M machines 1,000 machines fail every day!

An idea and solution

- **Issue:**

  Copying data over a network takes time
- **Idea:**
  - Bring computation to data
  - Store files multiple times for reliability
- **Spark/Hadoop** address these problems
  - **Storage Infrastructure – File system**
    - Google: GFS. Hadoop: HDFS
  - **Programming model**
    - MapReduce
    - Spark

Storage infrastructure

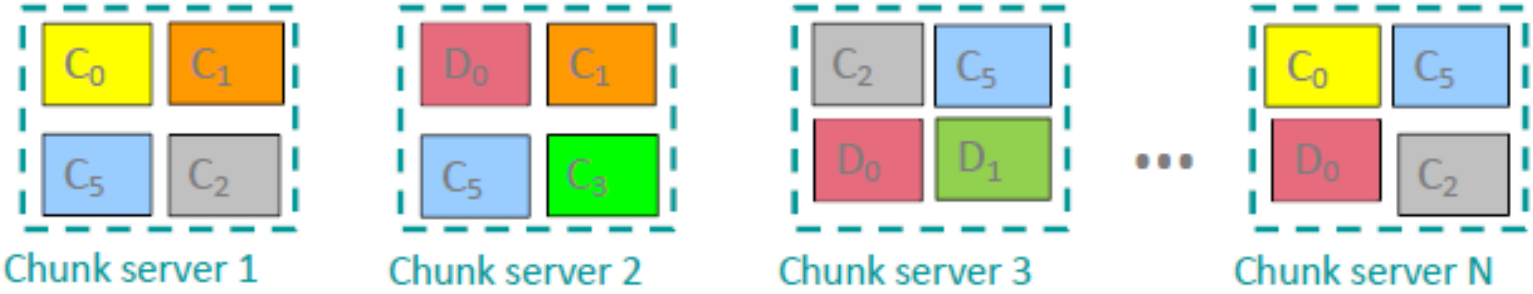
- **Problem:**
  - If nodes fail, how to store data persistently?
- **Answer:**
  - **Distributed File System**
    - Provides global file namespace
- **Typical usage pattern:**
  - Huge files (100s of GB to TB)
  - Data is rarely updated in place
  - Reads and appends are common

Distributed file system

- **Chunk servers**
  - File is split into contiguous chunks
  - Typically each chunk is 16-64MB
  - Each chunk replicated (usually 2x or 3x)
  - Try to keep replicas in different racks
- **Master node**
  - a.k.a. Name Node in Hadoop's HDFS
  - Stores metadata about where files are stored
  - Might be replicated
- **Client library for file access**
  - Talks to master to find chunk servers
  - Connects directly to chunk servers to access data

Distributed file system

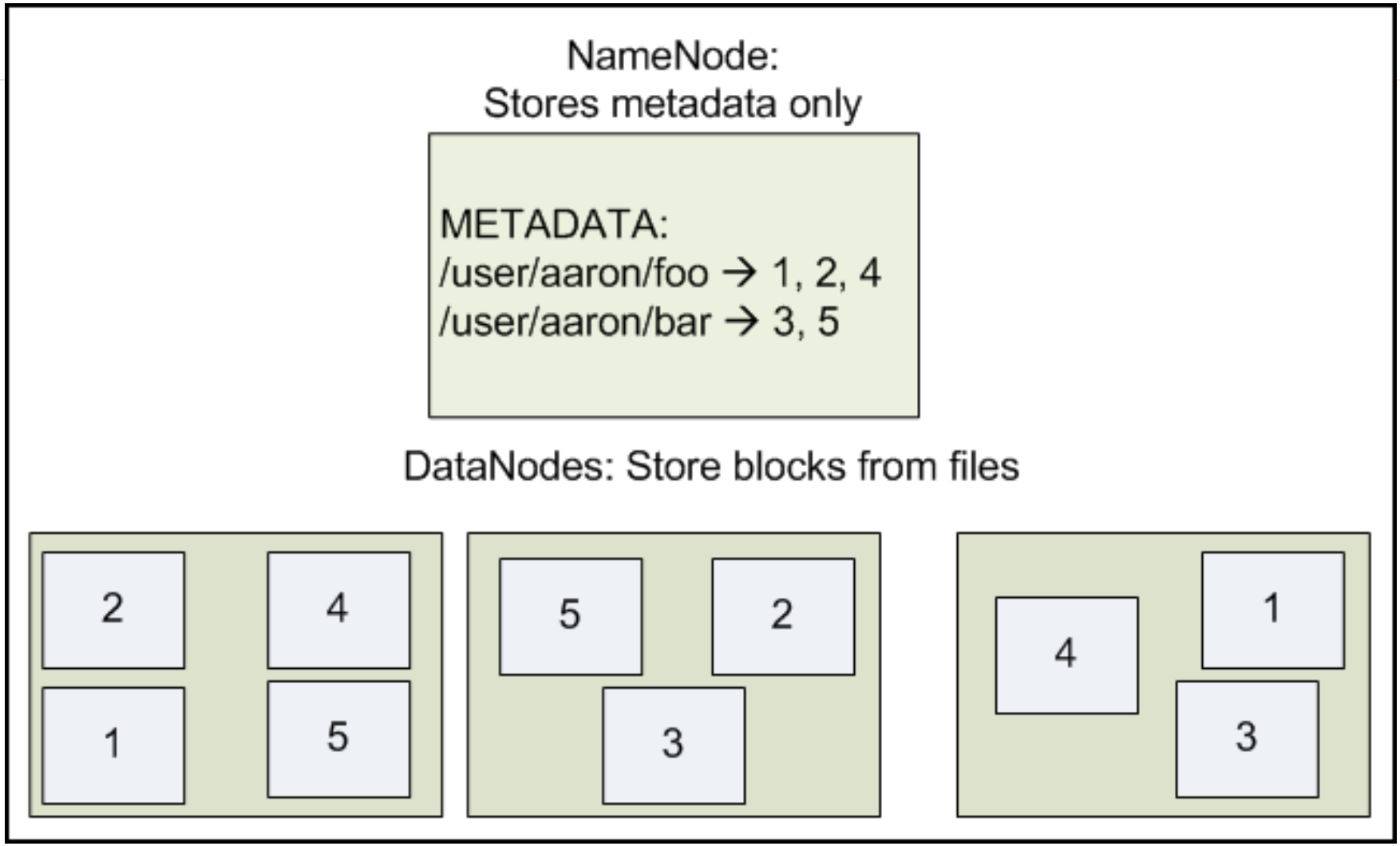
- **Reliable distributed file system**
- Data kept in “chunks” spread across machines
- Each chunk **replicated** on different machines
  - Seamless recovery from disk or machine failure



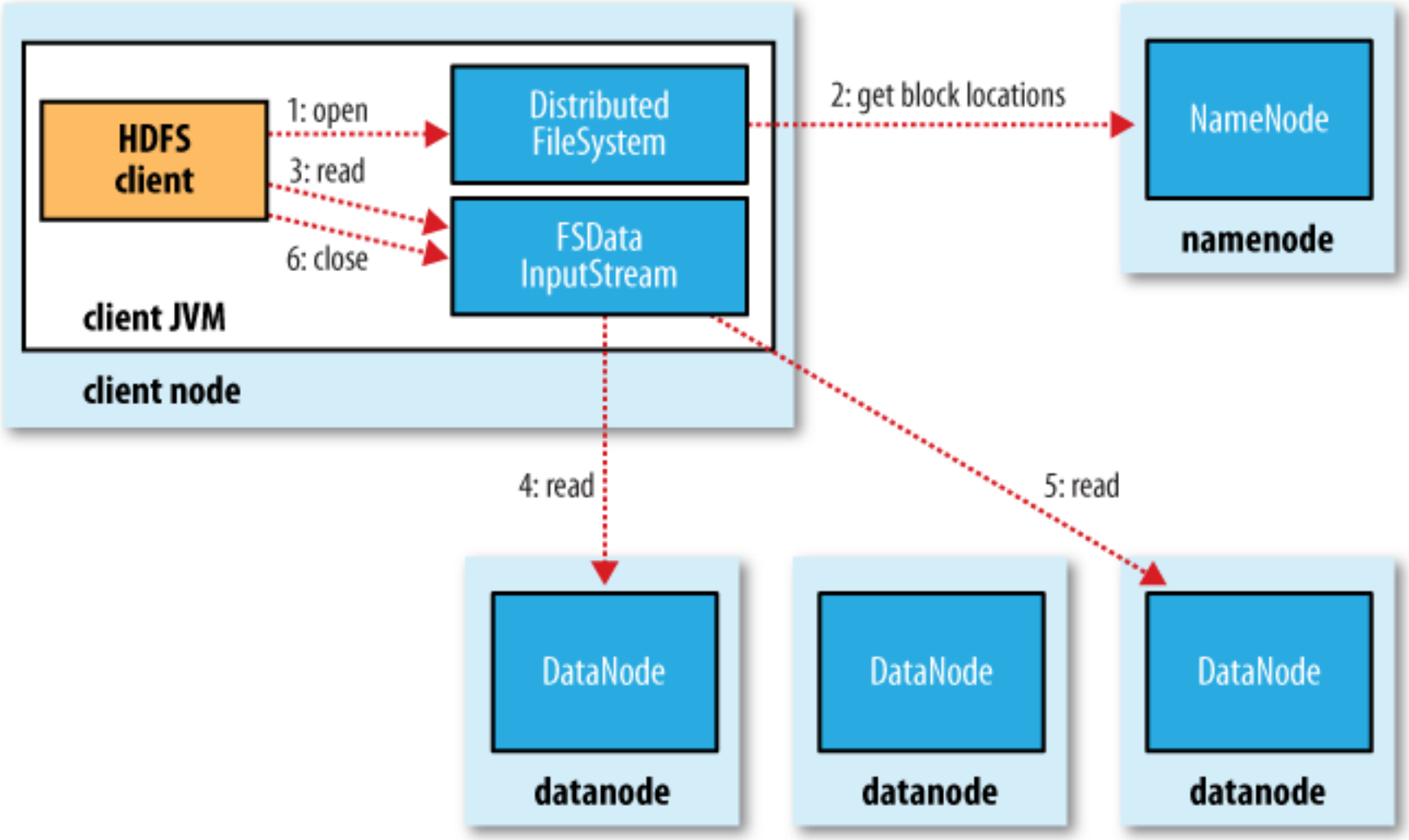
Bring computation directly to the data!

Chunk servers also serve as compute servers

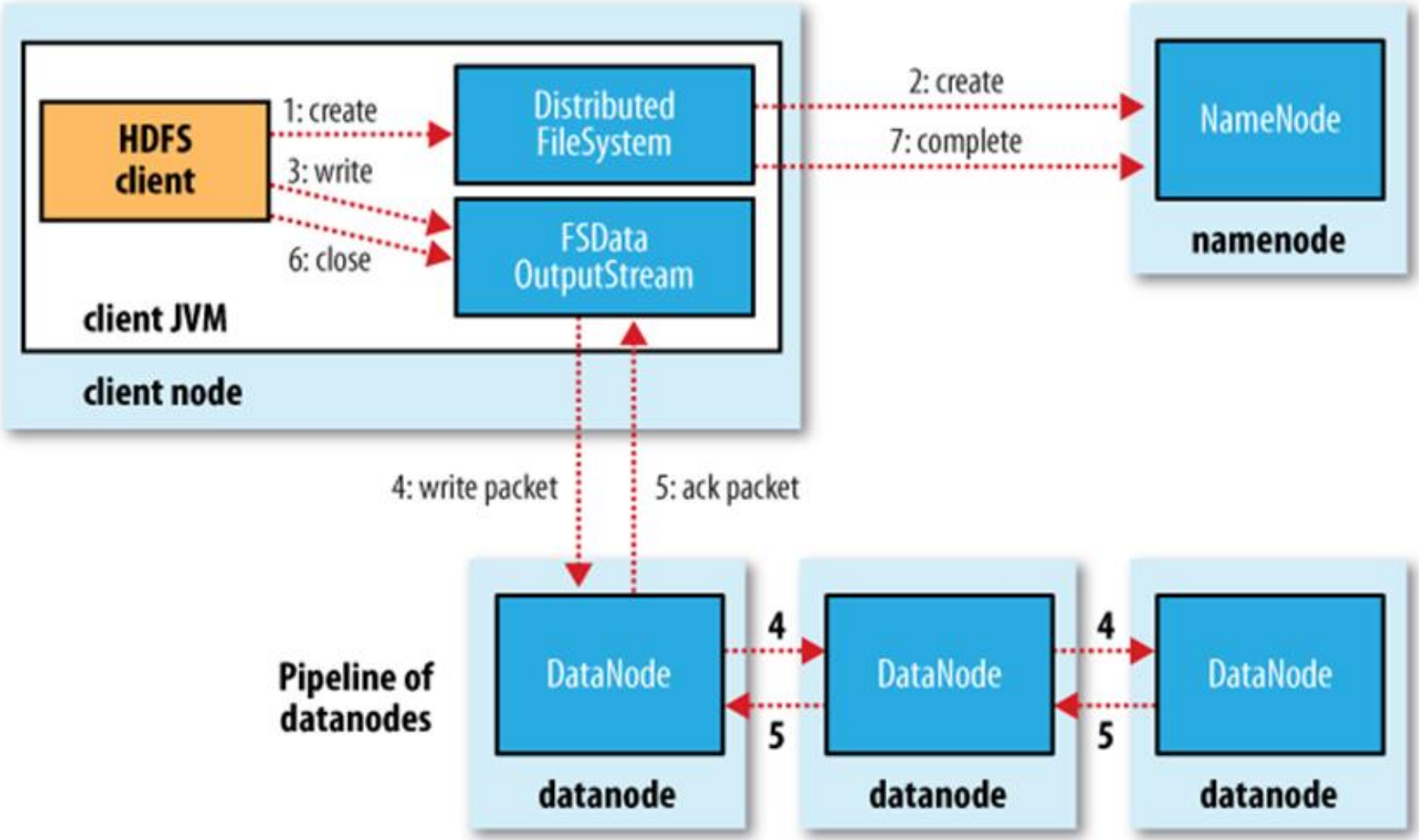
Hadoop HDFS


NameNode:
Stores metadata only
METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5
DataNodes: Store blocks from files
2
4
1
5
5
2
3
4
1
3

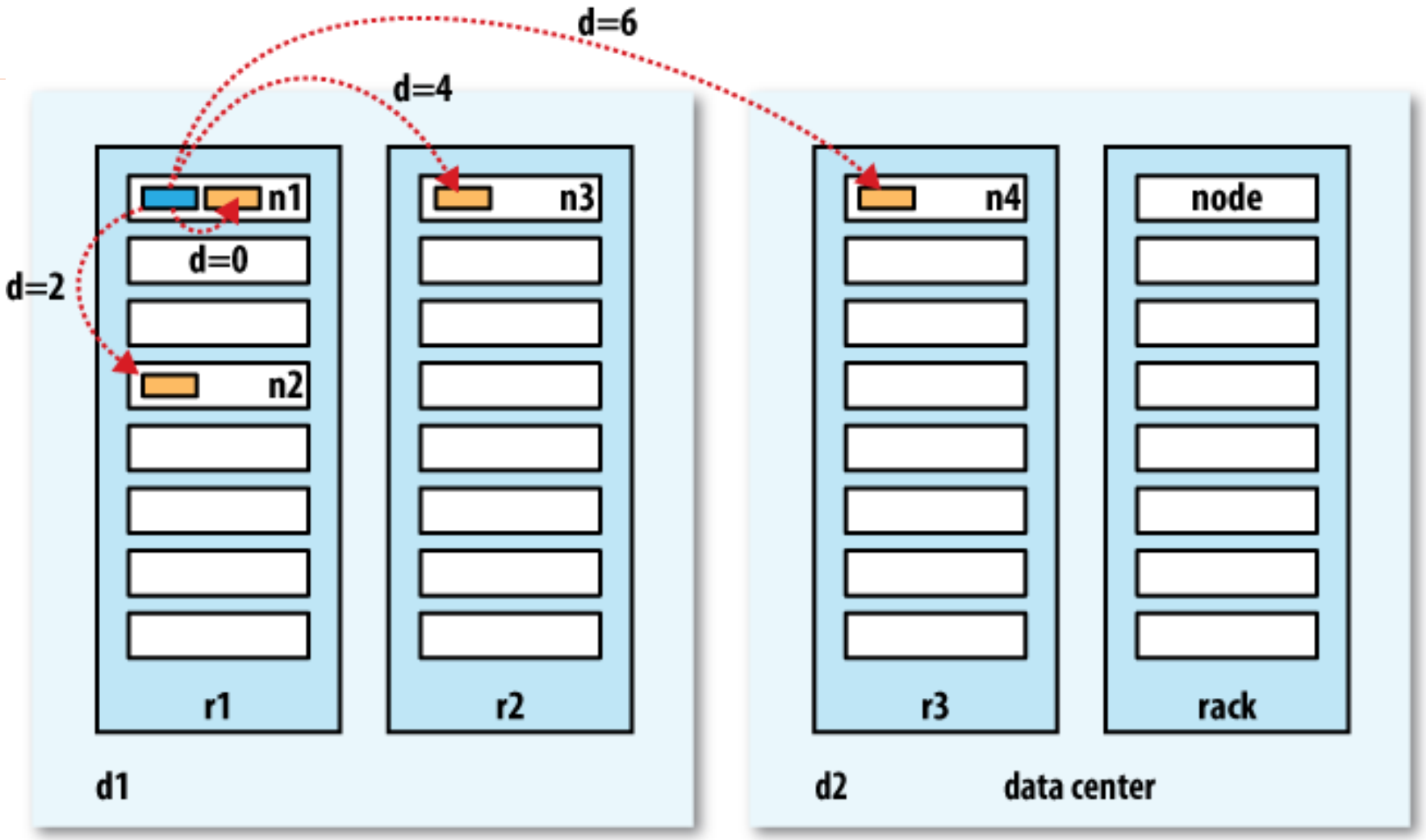
A client reading data from HDFS



A client writing data to HDFS



Hadoop Cluster Rack Awareness



Programming model

- **MapReduce is a style of programming designed for:**
  1. Easy parallel programming
  2. Invisible management of hardware and software failures
  3. Easy management of very-large-scale data

- It has several implementations, including Hadoop, Spark (used in this class), Flink, and the original Google implementation just called "MapReduce"

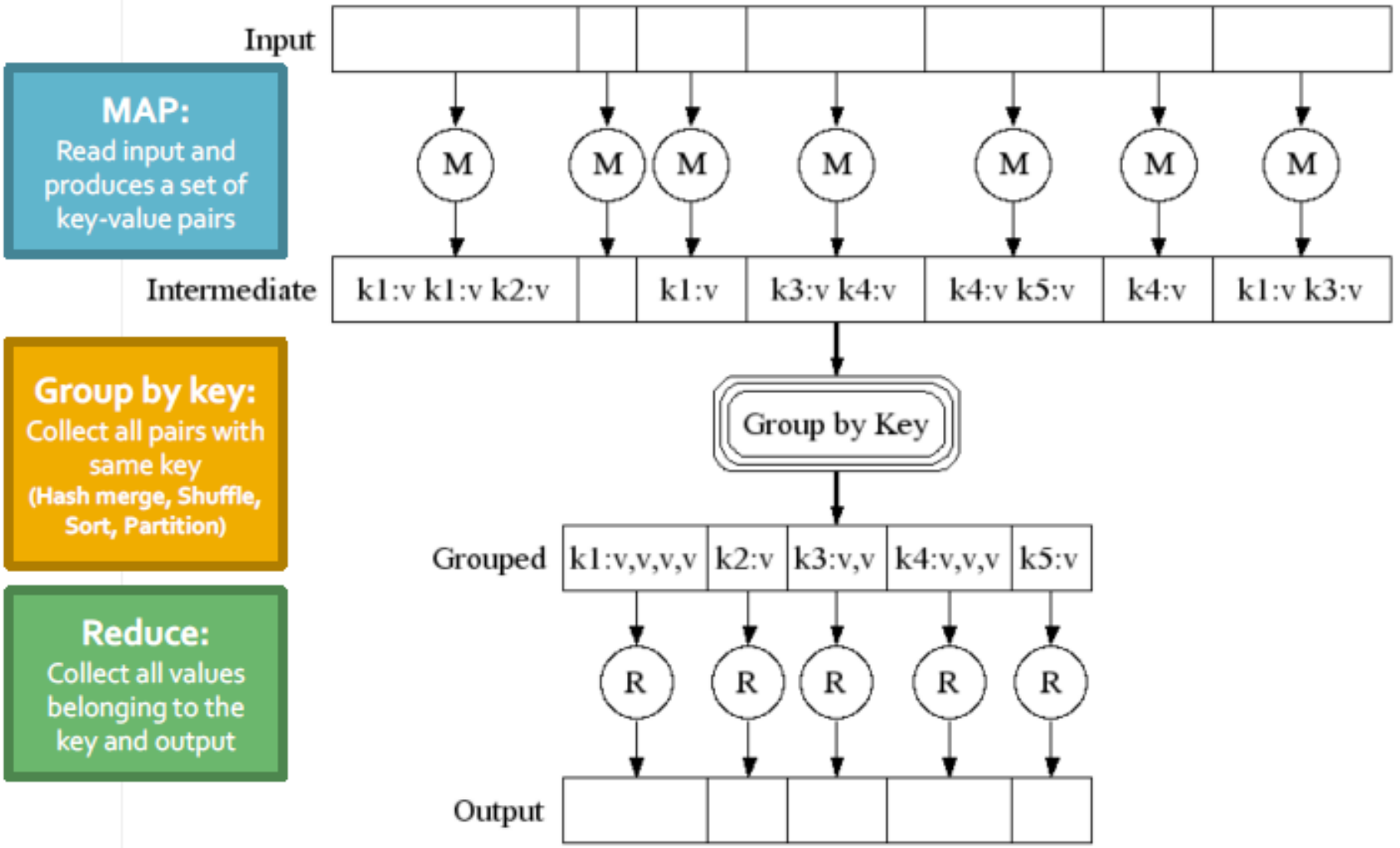
Map-reduce overview

3 steps of MapReduce

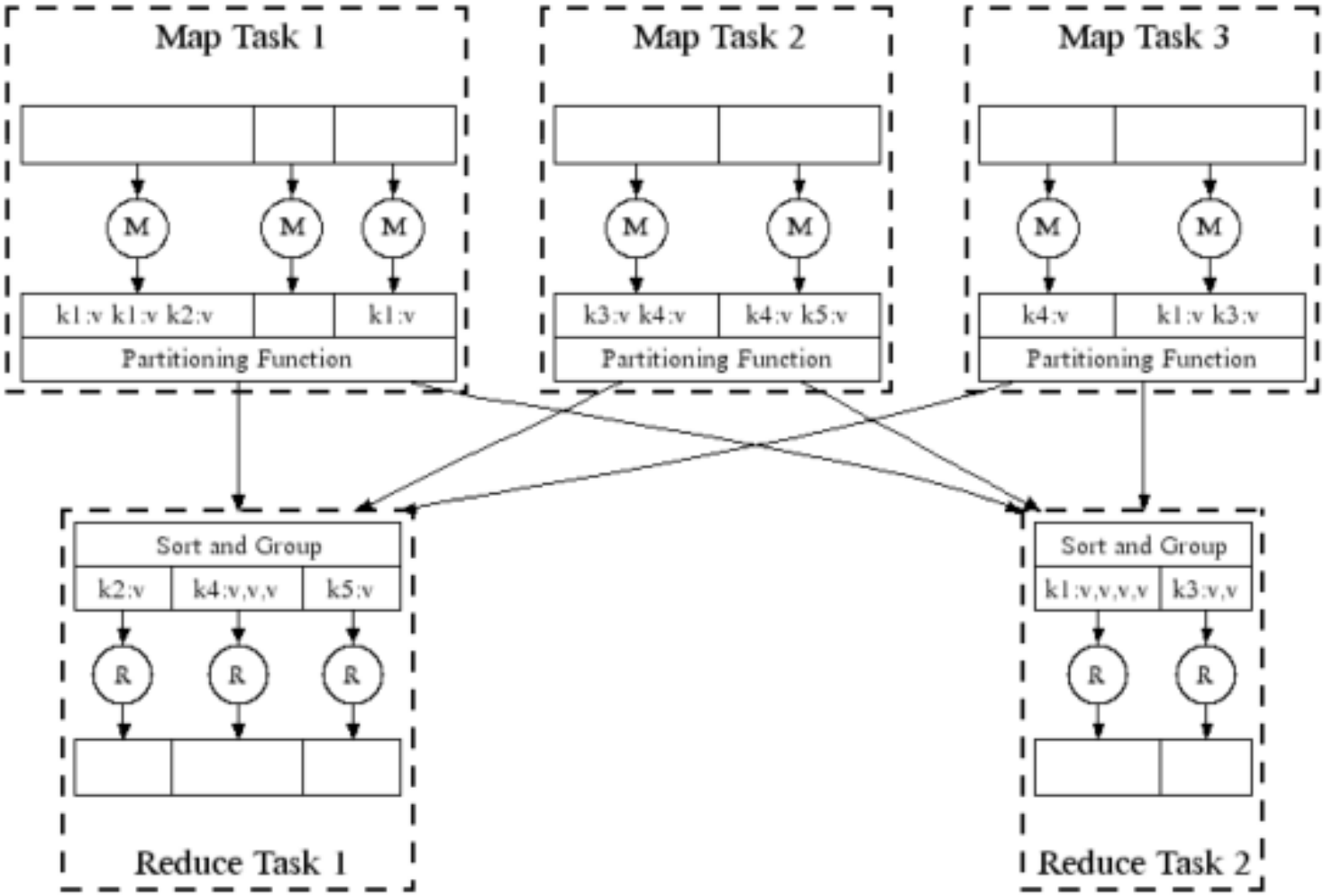
- **Map:**
  - Apply a user-written *Map function* to each input element
    - *Mapper* applies the Map function to a single element
      - Many mappers grouped in a *Map task* (the unit of parallelism)
  - The output of the Map function is a set of 0, 1, or more *key-value pairs*.
- **Group by key:** Sort and shuffle
  - System sorts all the key-value pairs by key, and outputs key-(list of values) pairs
- **Reduce:**
  - User-written *Reduce function* is applied to each key-(list of values)

Outline stays the same, **Map** and **Reduce** change to fit the problem

Map-reduce a diagram

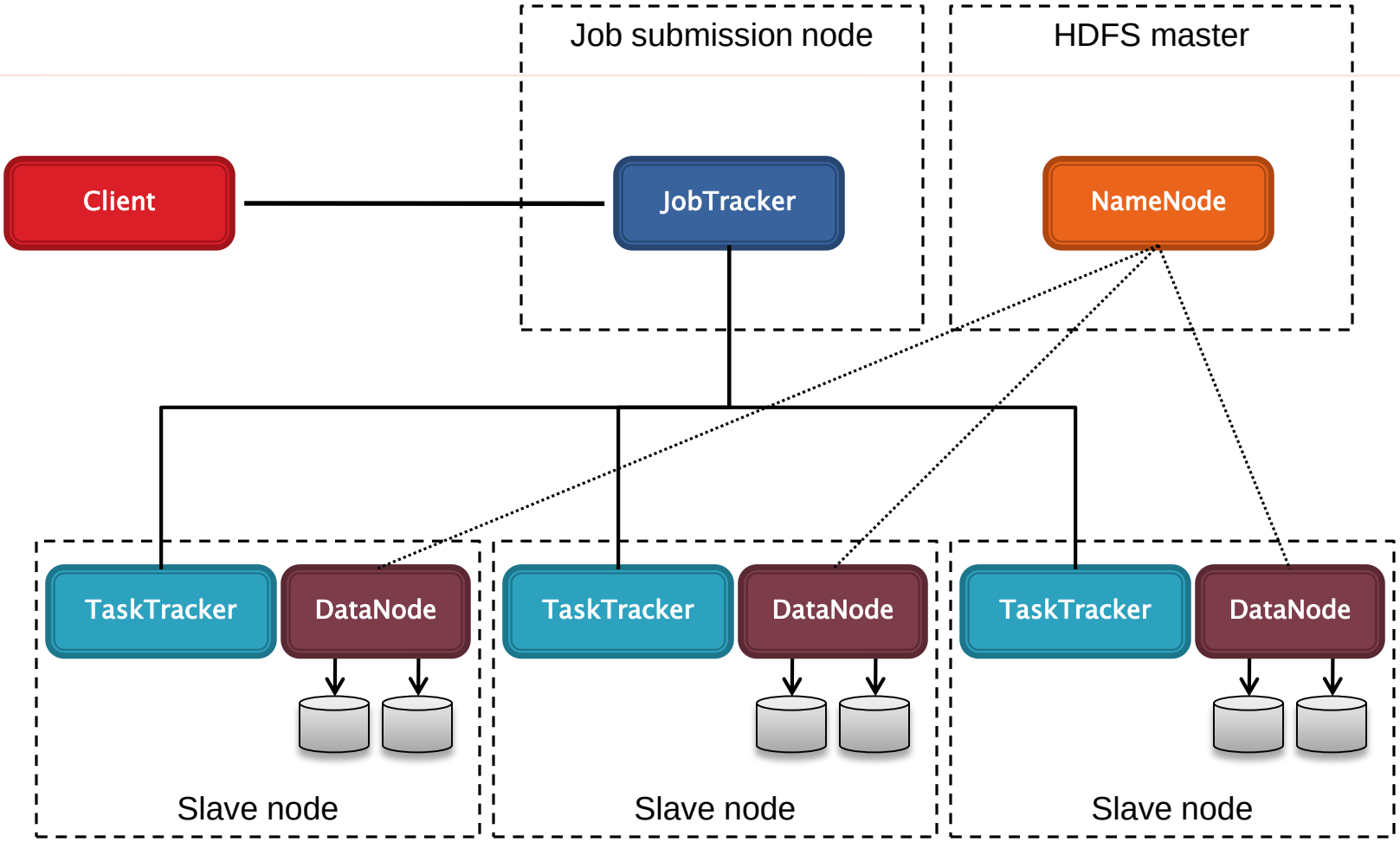


Map-reduce in parallel

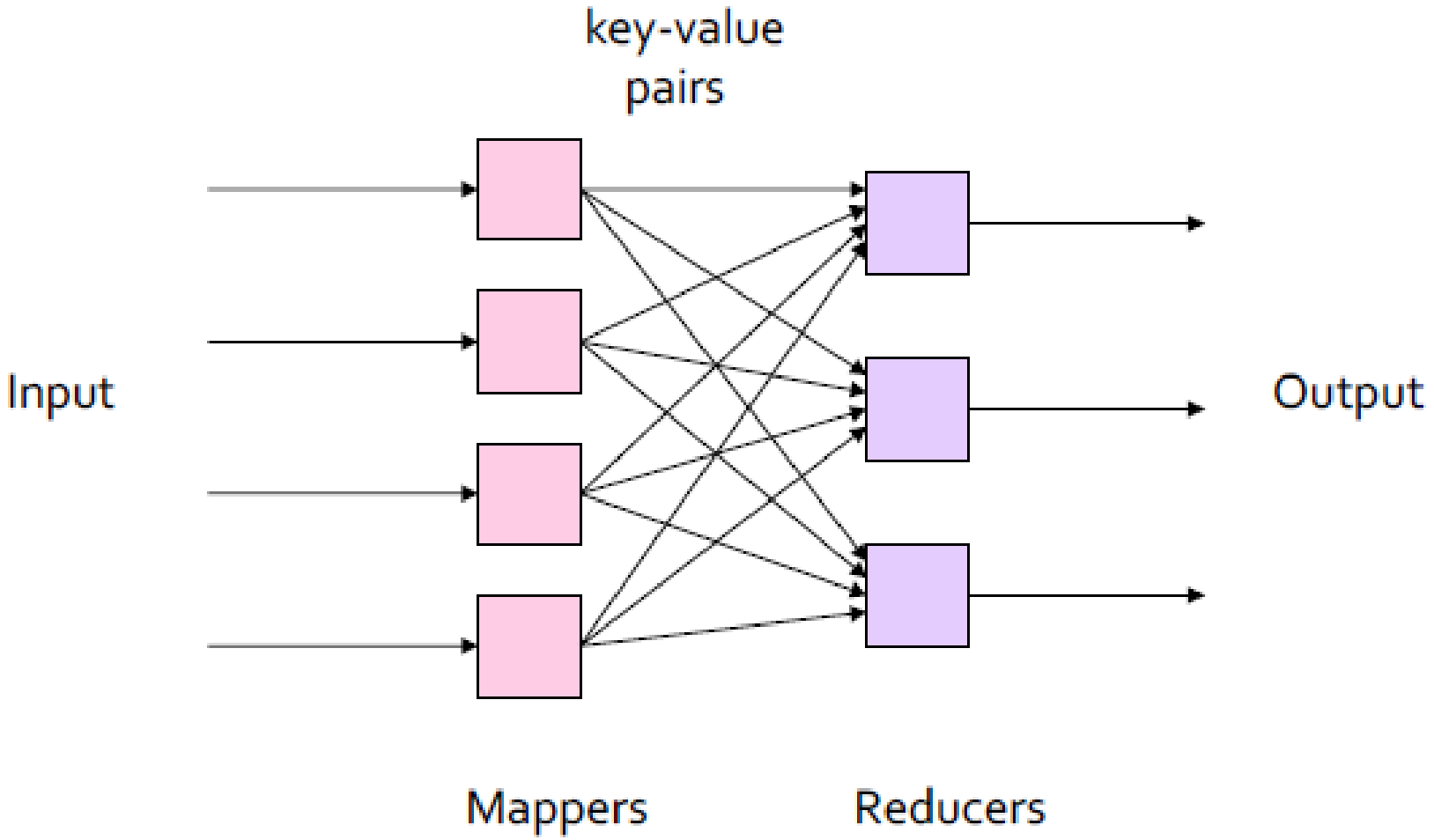


All phases are distributed with many tasks doing the work

Hadoop Cluster Architecture



Map reduce pattern



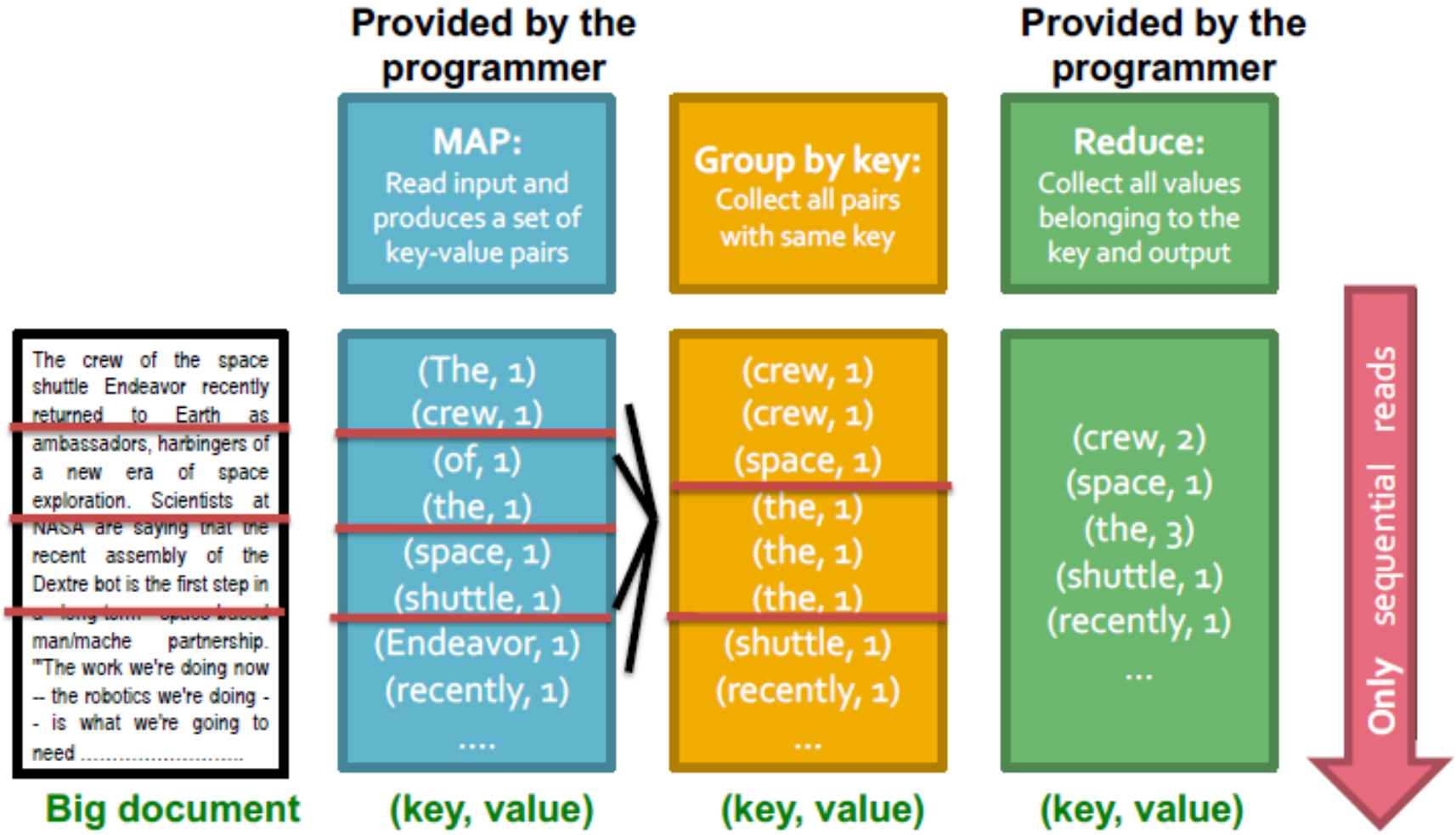
Example : word count

Example MapReduce task:

- We have a huge text document
- Count the number of times each distinct word appears in the file

- **Many applications of this:**
  - Analyze web server logs to find popular URLs
  - Statistical machine translation:
    - Need to count number of times every 5-word sequence occurs in a large corpus of documents

Example : word count



Example : word count

```
map(key, value):
# key: document name; value: text of the document
   for each word w in value:
      emit(w, 1)
```

```
reduce(key, values):
# key: a word; value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(key, result)
```

Map reduce environment

MapReduce environment takes care of:

- **Partitioning** the input data
- **Scheduling** the program's execution across a set of machines
- Performing the **group by key** step
  - In practice this is is the bottleneck
- Handling machine **failures**
- Managing required inter-machine **communication**

Dealing with failure

- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle and rescheduled
  - Reduce workers are notified when map task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle and the reduce task is restarted

Problems with MapReduce

- **Two major limitations of MapReduce:**
  - Difficulty of programming directly in MR
    - Many problems aren't easily described as map-reduce
  - Performance bottlenecks, or batch not fitting the use cases
    - Persistence to disk typically slower than in-memory work
- **In short, MR doesn't compose well for large applications**
  - Many times one needs to chain multiple map-reduce steps

Dataflow systems

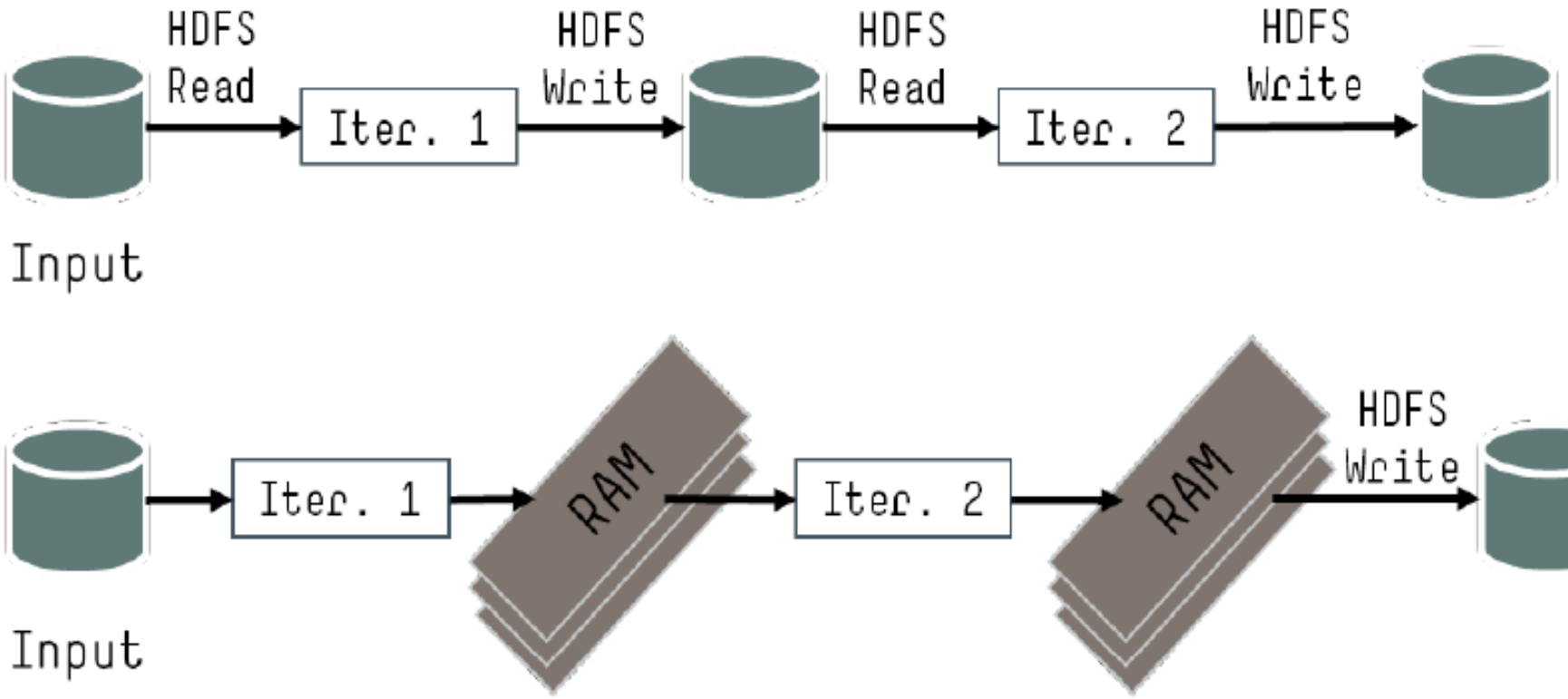
- **MapReduce uses two "ranks" of tasks:**
  One for Map the second for Reduce
  - Data flows from the first rank to the second

- **Data-Flow Systems generalize this in two ways:**
  1. Allow any number of tasks/ranks
  2. Allow functions other than Map and Reduce
  - As long as data flow is in one direction only, we can have the blocking property and allow recovery of tasks rather than whole jobs

Spark : most popular data-flow system

- **Expressive computing system, not limited to the map-reduce model**
- **Additions to MapReduce model:**
  - Fast data sharing
    - Avoids saving intermediate results to disk
    - Caches data for repetitive queries (e.g. for machine learning)
  - General execution graphs (DAGs)
  - Richer functions than just map and reduce
- Compatible with Hadoop

MapReduce versus Spark



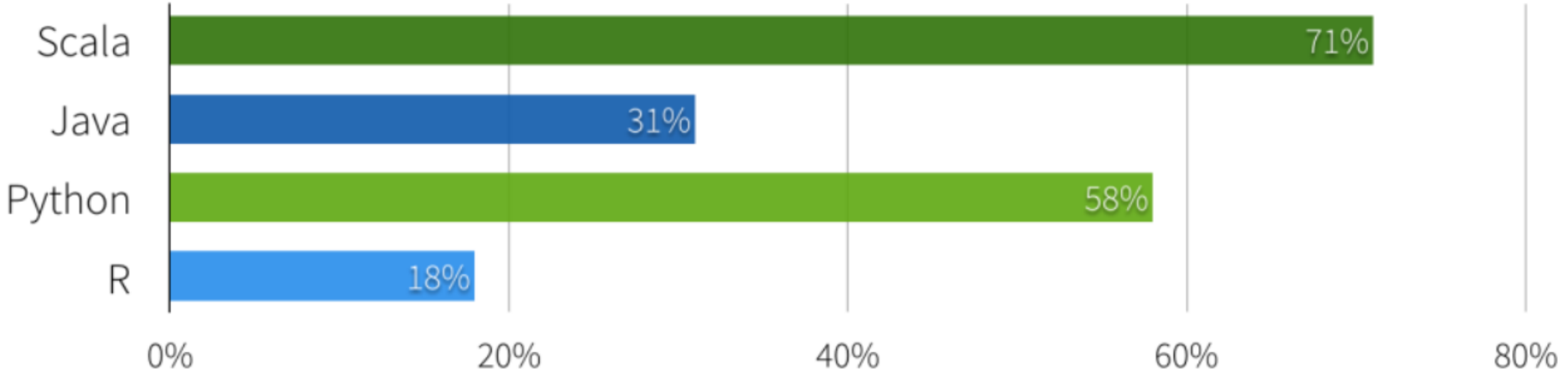
Spark : overview

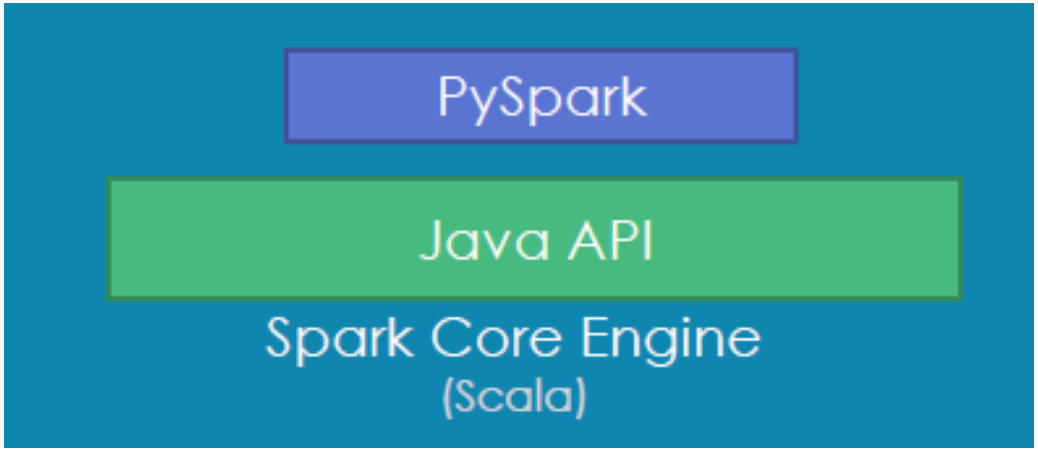
- Open source software (Apache Foundation)
- Supports **Java, Scala and Python**

- **Key construct/idea:** Resilient Distributed Dataset (RDD)

- **Higher-level APIs:** DataFrames & DataSets
  - Introduced in more recent versions of Spark
  - Different APIs for aggregate data, which allowed to introduce SQL support

Programming





Spark :RDD

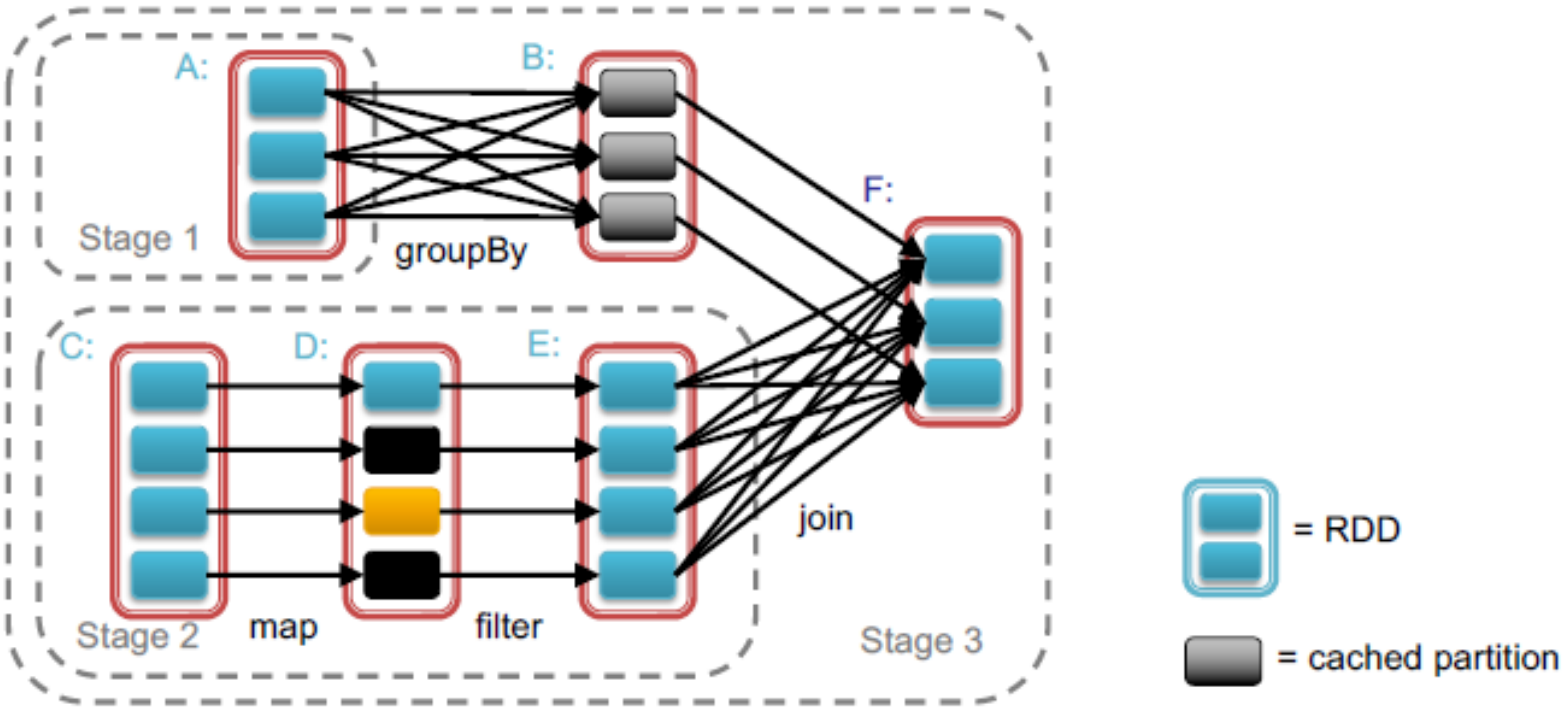
Key concept *Resilient Distributed Dataset* (RDD)

- Partitioned collection of records
  - Generalizes (key-value) pairs
- Spread across the cluster, Read-only
- Caching dataset in memory
  - Different storage levels available
  - Fallback to disk possible
- RDDs can be created from Hadoop, or by transforming other RDDs (you can stack RDDs)
- RDDs are best suited for applications that apply the same operation to all elements of a dataset

Spark RDD operations

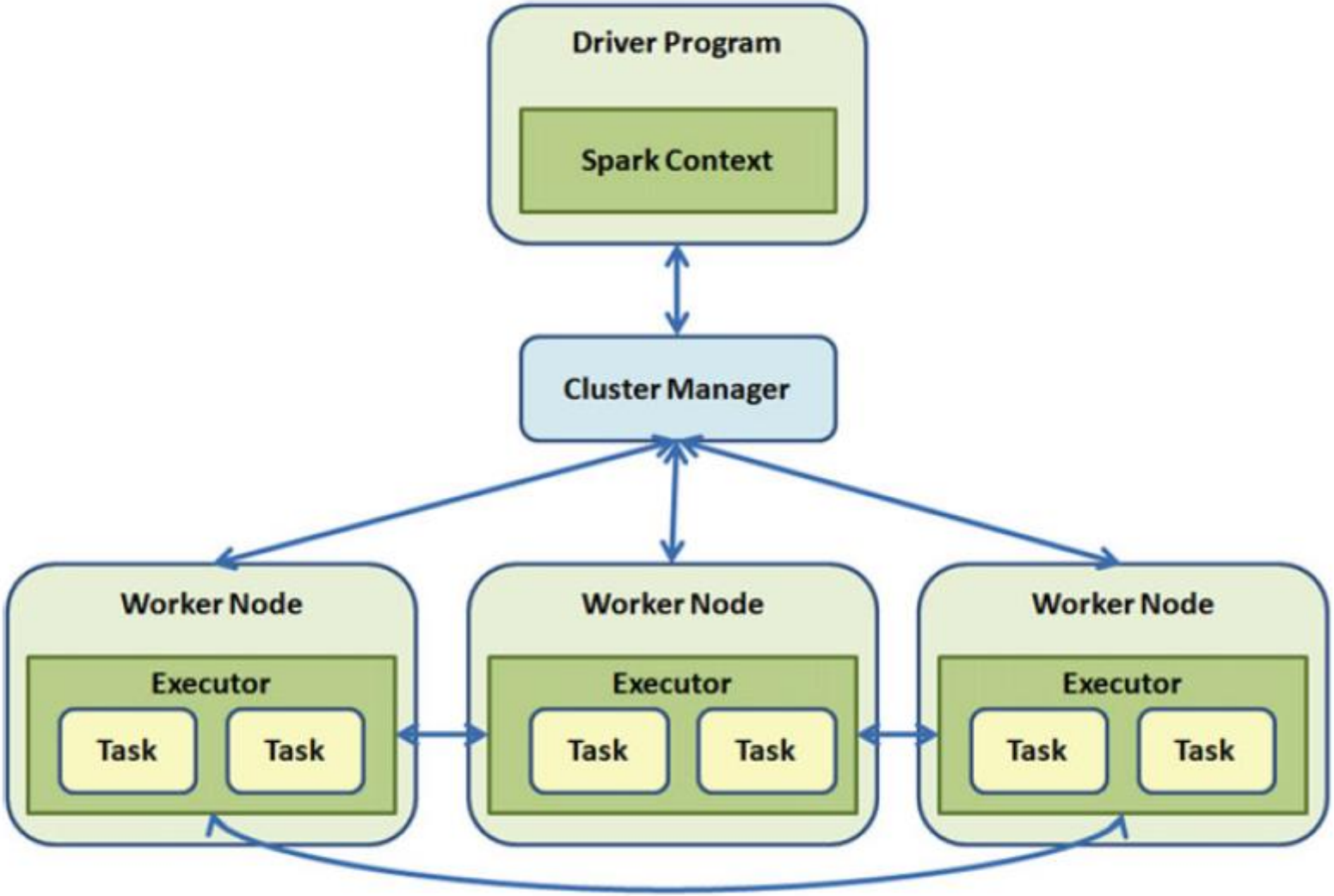
- **Transformations** build RDDs through deterministic operations on other RDDs:
  - Transformations include *map*, *filter*, *join*, *union*, *intersection*, *distinct*
  - **Lazy evaluation:** Nothing computed until an action requires it
- **Actions** to return value or export data
  - Actions include *count*, *collect*, *reduce*, *save*
  - Actions can be applied to RDDs; actions force calculations and return values

Task scheduler : general DAGs



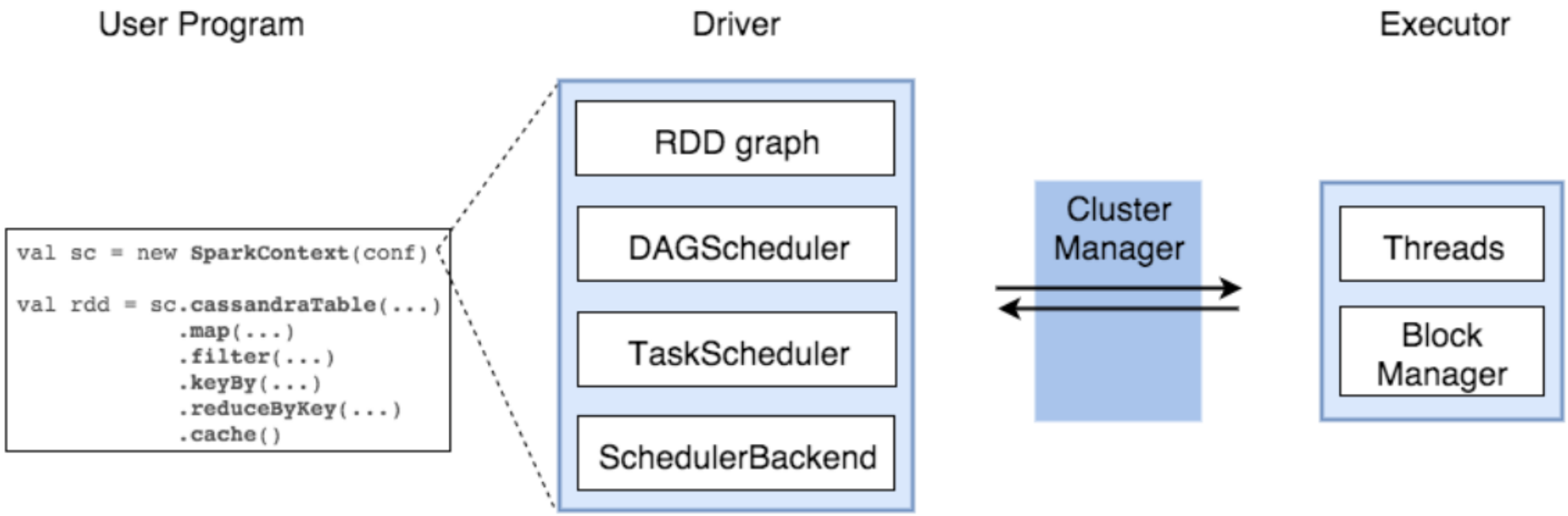
- Supports general task graphs
- Pipelines functions where possible
- Cache-aware data reuse & locality
- Partitioning-aware to avoid shuffles

Spark cluster

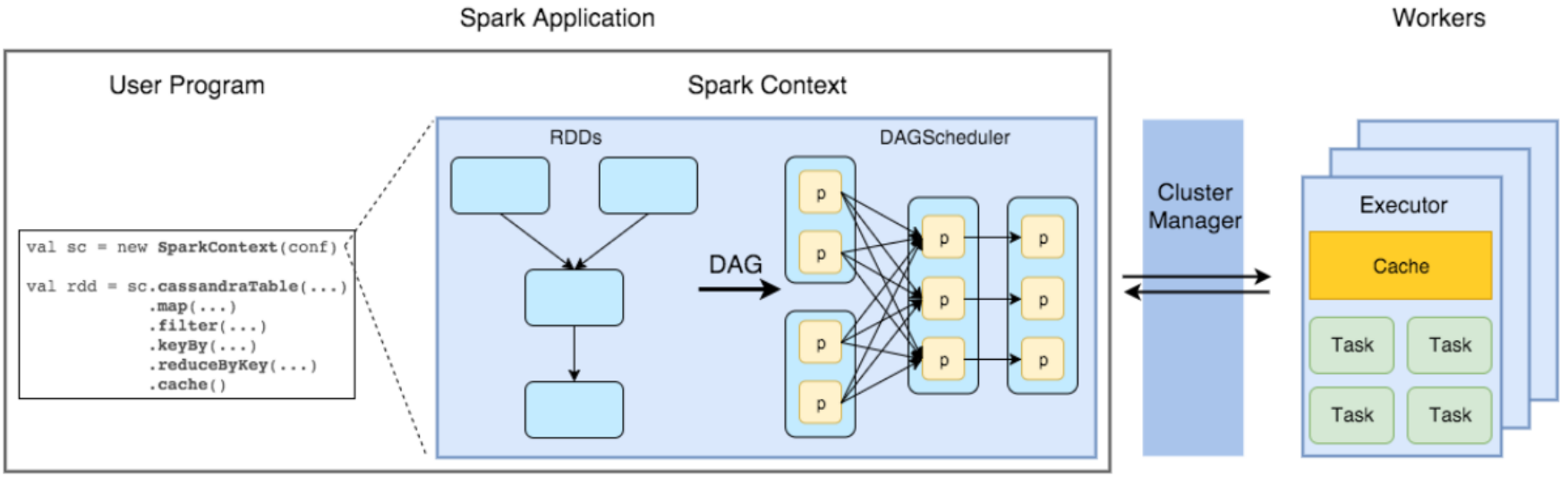


Spark Driver

contains components responsible for translation of user code into actual jobs executed on cluster



Spark Execution workflow

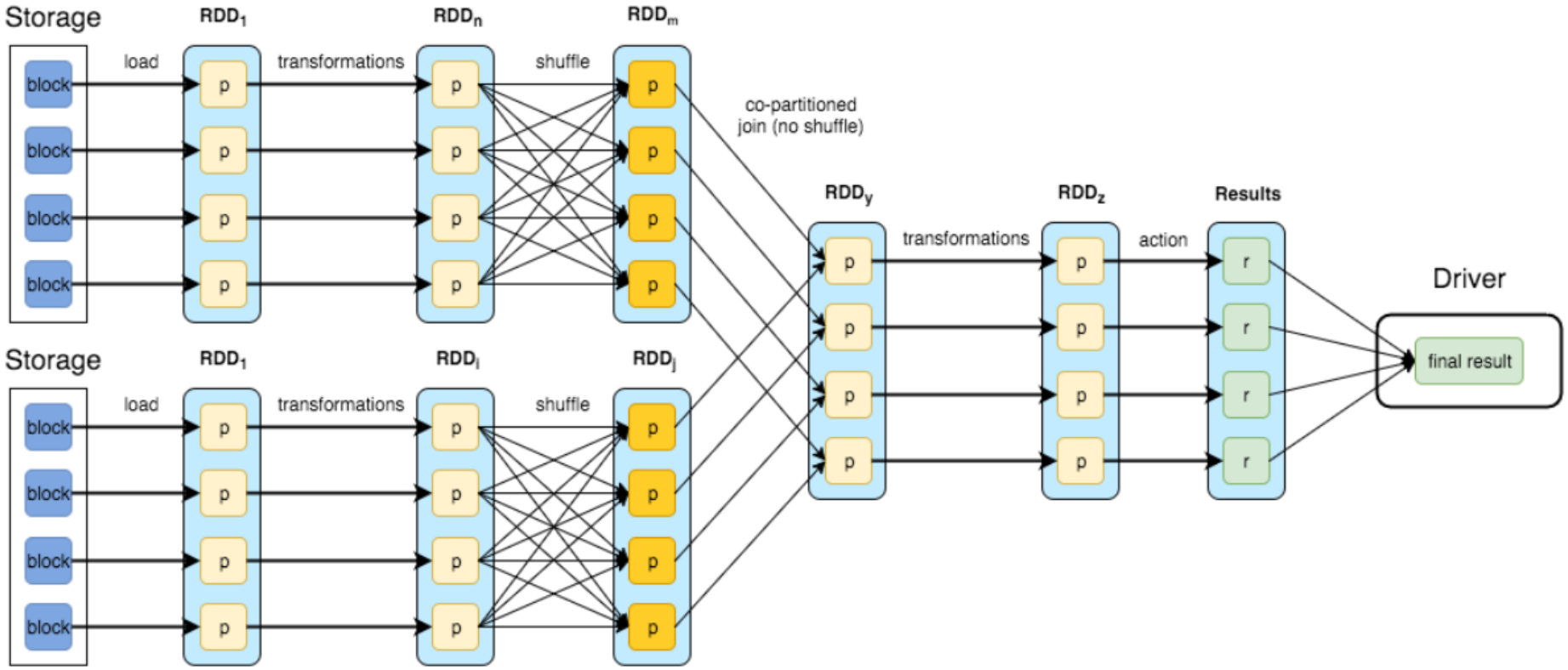


user code containing RDD transformations forms Direct Acyclic Graph which is then split into stages of tasks by DAGScheduler. Stages combine tasks which don't require shuffling/repartitioning if the data. Tasks run on workers and results then return to client.

Spark

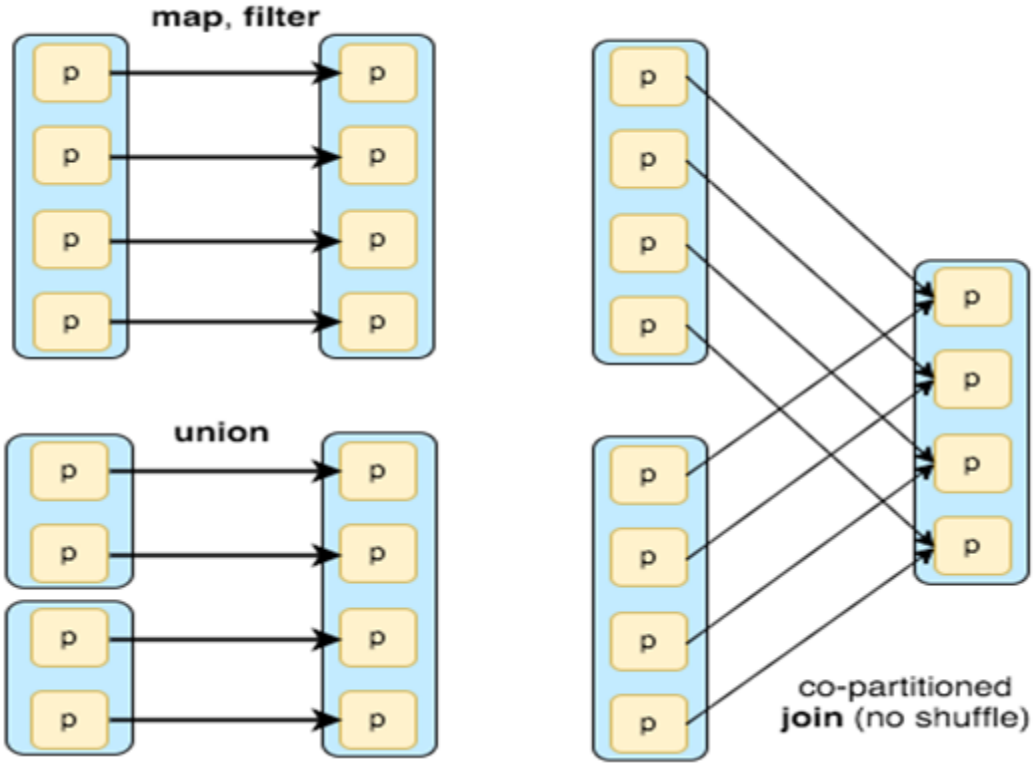
- **SparkContext**
  - represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster
- **DAGScheduler**
  - computes a DAG of stages for each job and submits them to TaskScheduler
  - determines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs
- **TaskScheduler**
  - responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers
- **SchedulerBackend**
  - backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local)
- **BlockManage**r
  - provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap)

Spark DAG


Storage
RDD1
RDDn
RDDm
load
transformations
shuffle
block
p
co-partitioned
join (no shuffle)
RDDy
RDDz
Results
transformations
action
r
Driver
final result
Storage
RDD1
RDDi
RDDj
load
transformations
shuffle

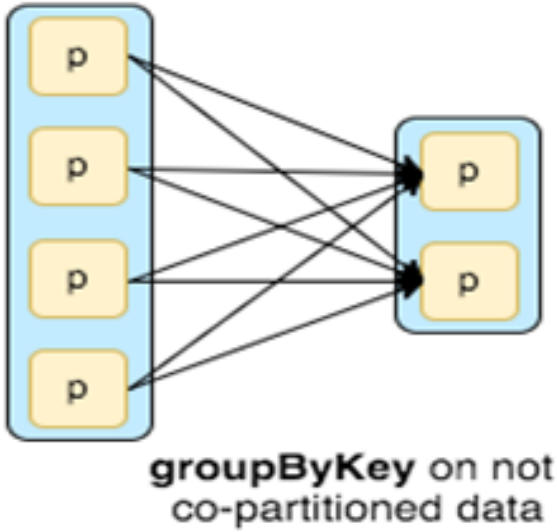
Narrow dependencies between RDDs (pipeline able)

- each partition of the parent RDD is used by at most one partition of the child RDD
- allow for pipelined execution on one cluster node
- failure recovery is more efficient as only lost parent partitions need to be recomputed

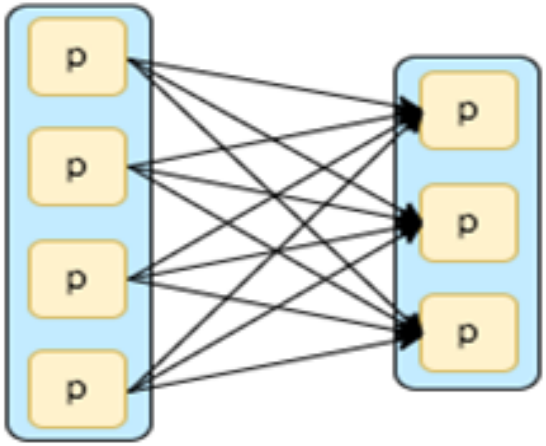


Wide dependencies between RDDs (shuffle)

- multiple child partitions may depend on one parent partition
- require data from all parent partitions to be available and to be shuffled across the nodes
- if some partition is lost from all the ancestors a complete recomputation is needed

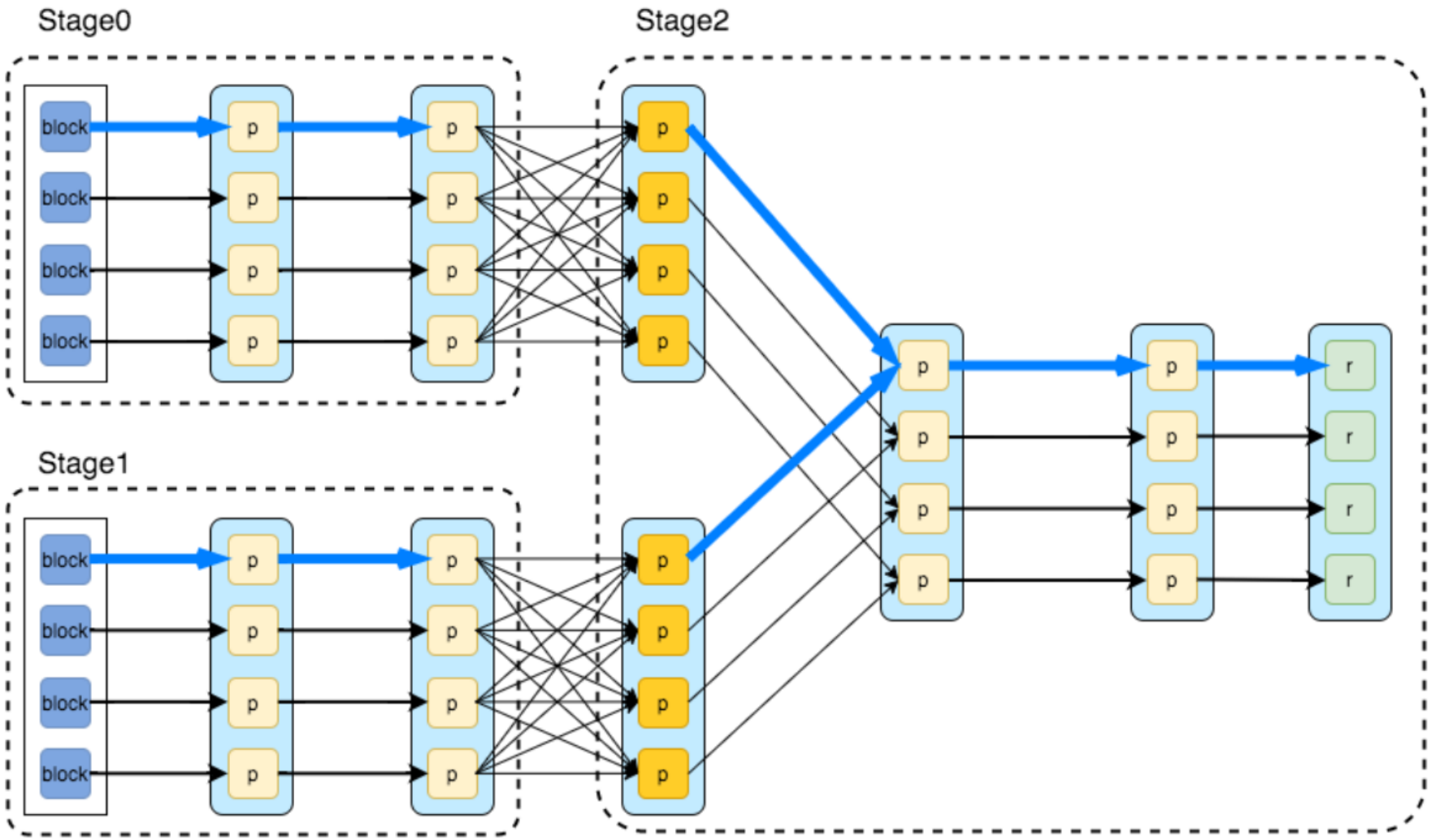


groupByKey on not co-partitioned data



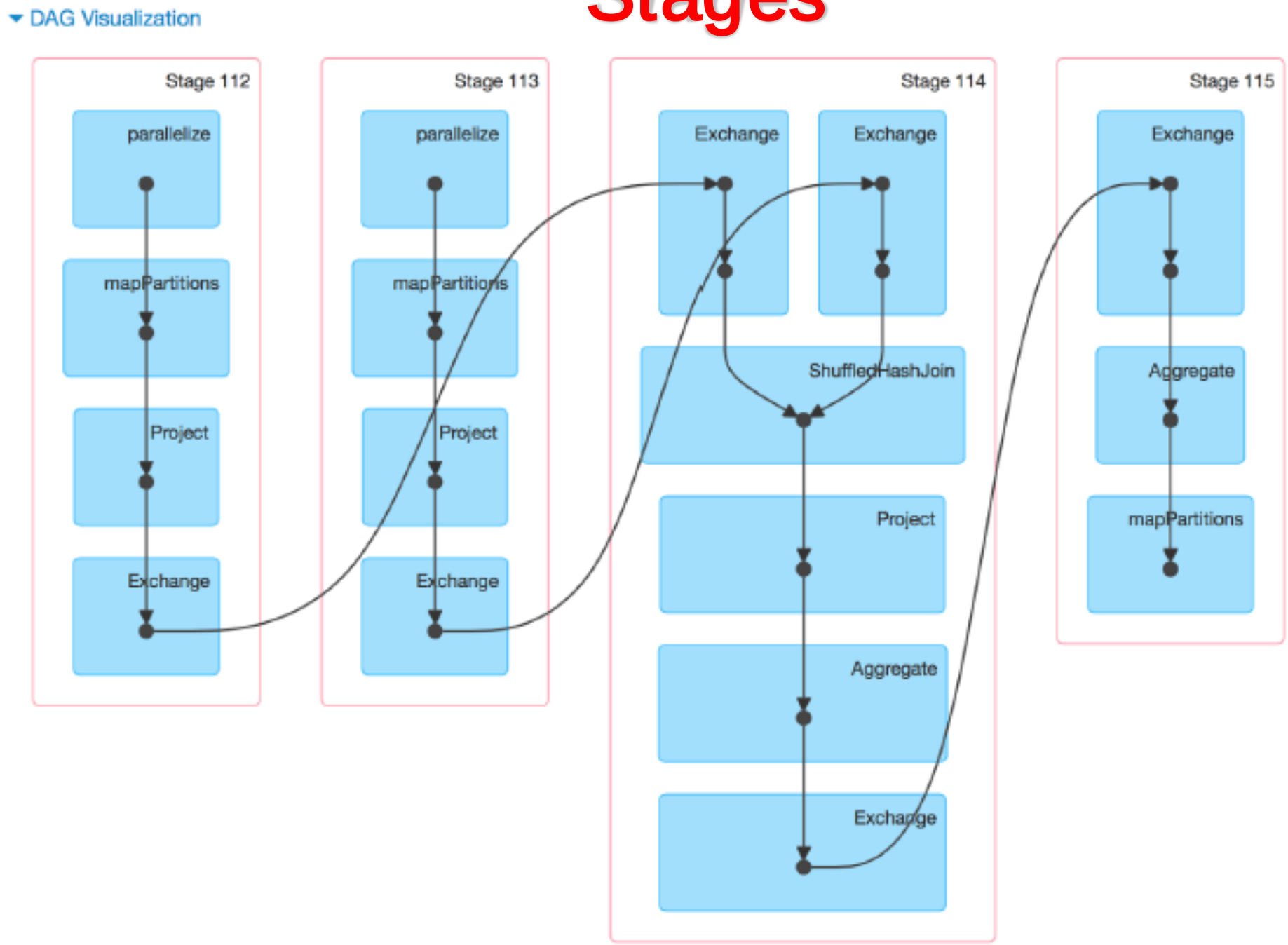
join with inputs not co-partitioned

Splitting DAG into Stages



Spark stages are created by breaking the RDD graph at shuffle boundaries

Stages


DAG Visualization
Stage 112
parallelize
mapPartitions
Project
Exchange
Stage 113
parallelize
mapPartitions
Project
Exchange
Stage 114
Exchange
Exchange
ShuffledHashJoin
Project
Aggregate
Exchange
Stage 115
Exchange
Aggregate
mapPartitions

Data frame & Dataset

- DataFrame:
  - Unlike an RDD, data organized into named columns, e.g. a **table in a relational database**.
  - Imposes a structure onto a distributed collection of data, allowing higher-level abstraction
- Dataset:
  - Extension of DataFrame API which provides **type-safe, object-oriented programming interface** (compile-time error detection)

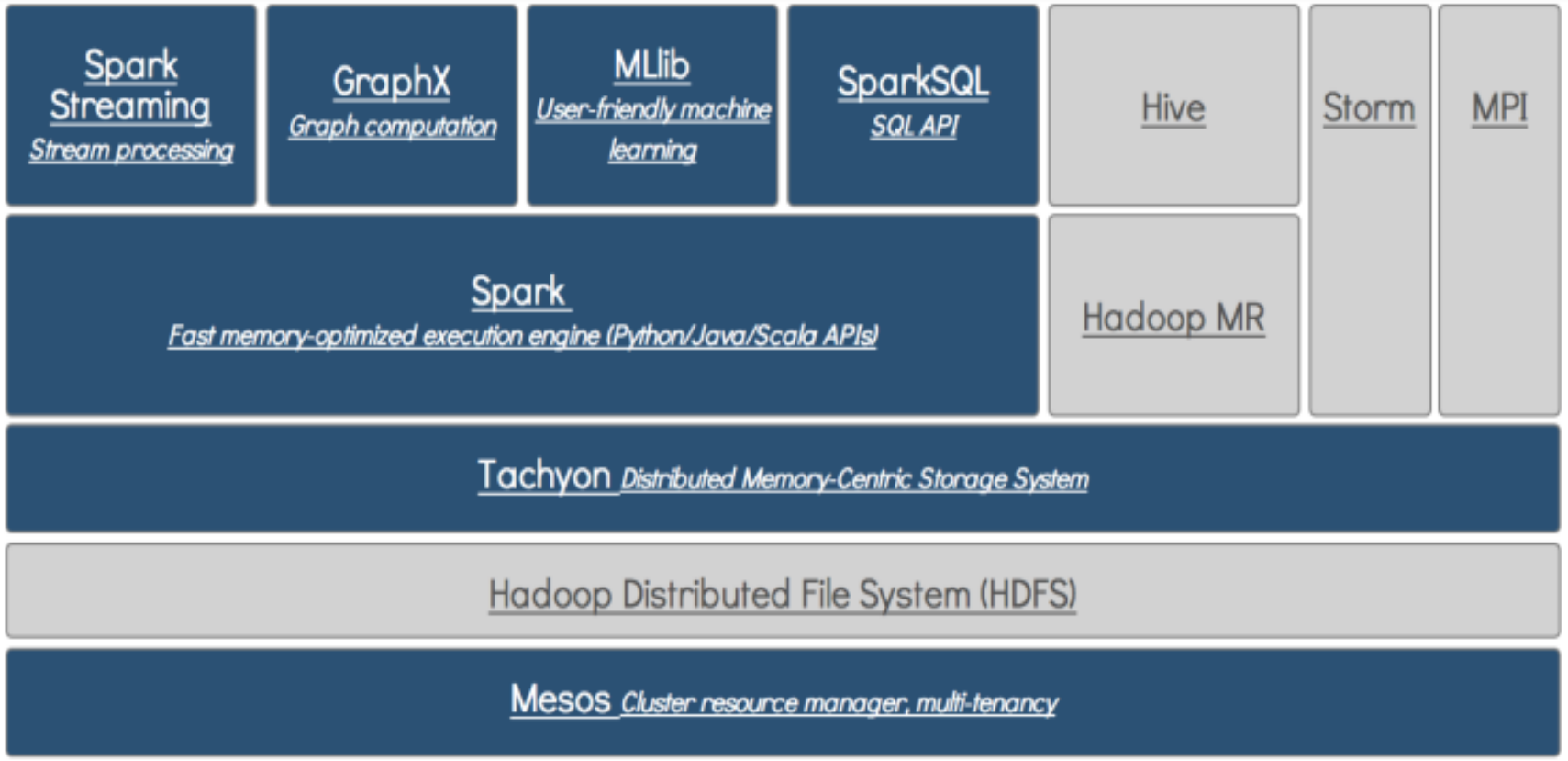
Both built on Spark SQL engine. both can be converted back to an RDD

Useful libraries for spark

- Spark SQL

- Spark Streaming – **stream processing of live datastreams**
- MLlib – **scalable machine learning**
- GraphX – **graph manipulation**
  - extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge

Data analytics software stack



Spark vs Hadoop MapReduce

- Performance: Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it **often needs lots of memory to perform well**; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- Ease of use: **Spark is easier to program** (higher-level APIs)
- Data processing: **Spark more general**

VS

YARN

Mesos

Tachyon

SQL

MLlib

Streaming

Problems suited for MapReduce

- **Suppose we have a large web corpus**
- Look at the metadata file
  - Lines of the form: (URL, size, date, …)
- **For each host, find the total number of bytes**
  - That is, the sum of the page sizes for all URLs from that particular host

- **Other examples:**
  - Link analysis and graph processing
  - Machine Learning algorithms

Problems suited for MapReduce

- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents

- **Very easy with MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts

Problems suited for MapReduce

- **Compute the natural join *R(A,B)* ⋈ *S(B,C)***
- *R* and *S* are each stored in files
- Tuples are pairs *(a,b)* or *(b,c)*

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

⋈

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Problems suited for MapReduce

- **Use a hash function *h* from B-values to *1...k***
- **A Map process turns:**
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*
- **Map processes** send each key-value pair with key *b* to Reduce process *h(b)*
  - Hadoop does this automatically; just tell it what *k* is.
- Each **Reduce process** matches all the pairs *(b,(a,R))* with all *(b,(c,S))* and outputs *(a,b,c)*.

Problems NOT suited for MapReduce

- **MapReduce is great for:**
  - Problems that require sequential data access
  - Large batch jobs (**not** interactive, real-time)
- **MapReduce is inefficient for problems where random (or irregular) access to data required:**
  - **Graphs**
  - Interdependent data
    - Machine learning
    - Comparisons of many pairs of items

Cost measures for the algorithm

- **In MapReduce we quantify the cost of an algorithm using**

1. *Communication cost* = total I/O of all processes
2. *Elapsed communication cost* = max of I/O along any path
3. (*Elapsed*) *computation cost* analogous, but count only running time of processes

Note that here the big-O notation is not the most useful (adding more machines is always an option)

Cost measures for the algorithm

- **For a map-reduce algorithm:**
  - **Communication cost =** input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process

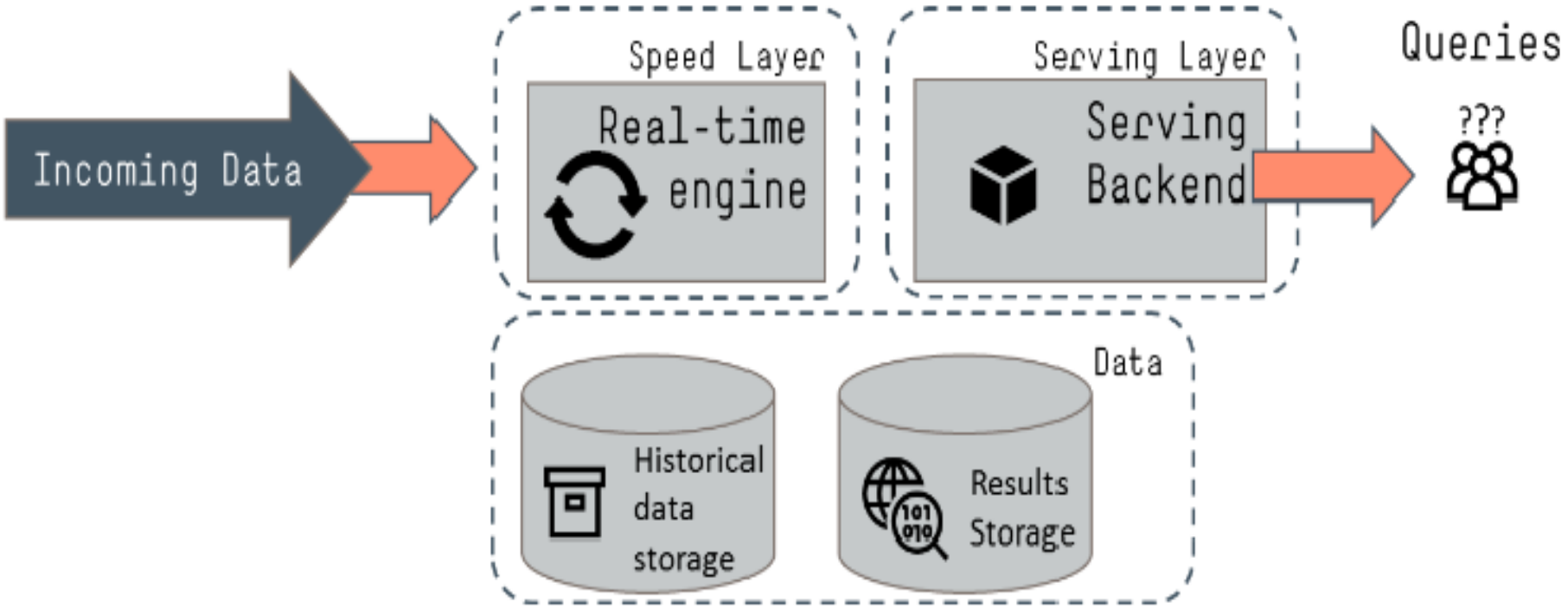
What cost measures mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other
- Total cost tells what you pay in rent from your friendly neighborhood cloud
- Elapsed cost is wall-clock time using parallelism

Cost of MapReduce join

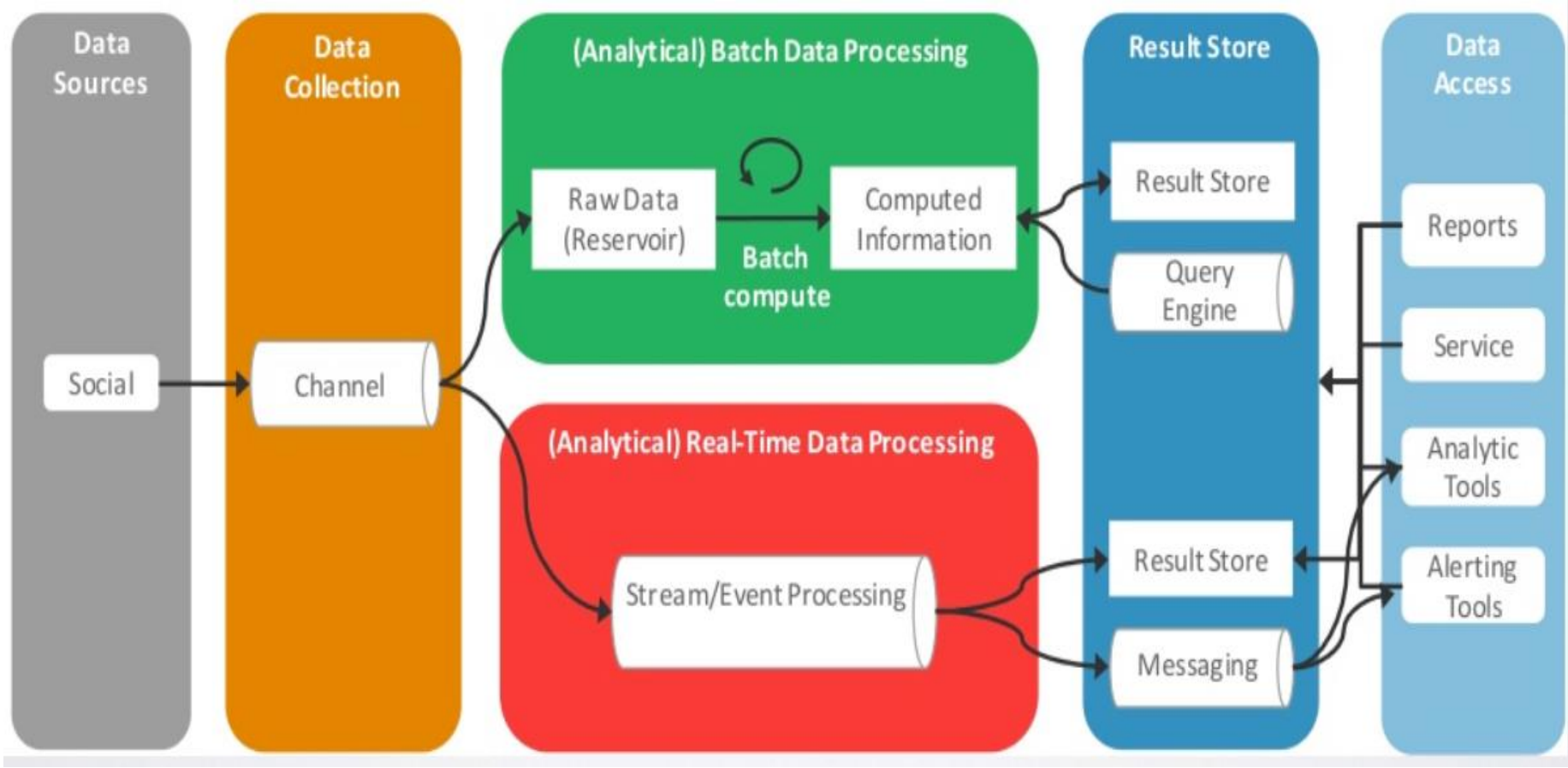
- **Total communication cost**
  $= O(|R|+|S|+|R \bowtie S|)$
- **Elapsed communication cost** = O(s)
  - We're going to pick ***k*** and the number of Map processes so that the I/O limit ***s*** is respected
  - We put a limit ***s*** on the amount of input or output that any one process can have. ***s* could be:**
    - What fits in main memory
    - What fits on local disk
- With proper indexes, computation cost is linear in the input + output size
  - So computation cost is like comm. cost

Kappa architectures

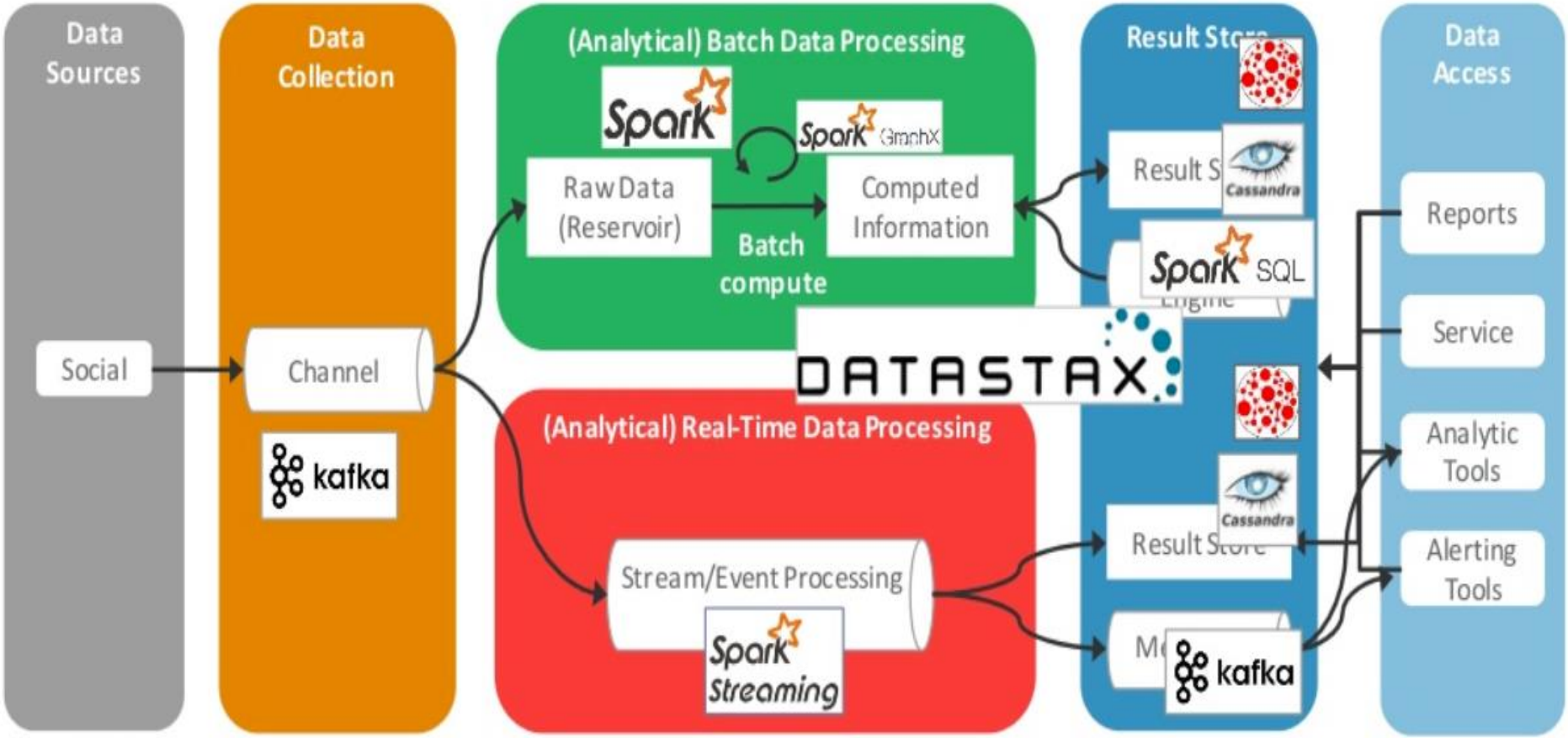


KAPPA

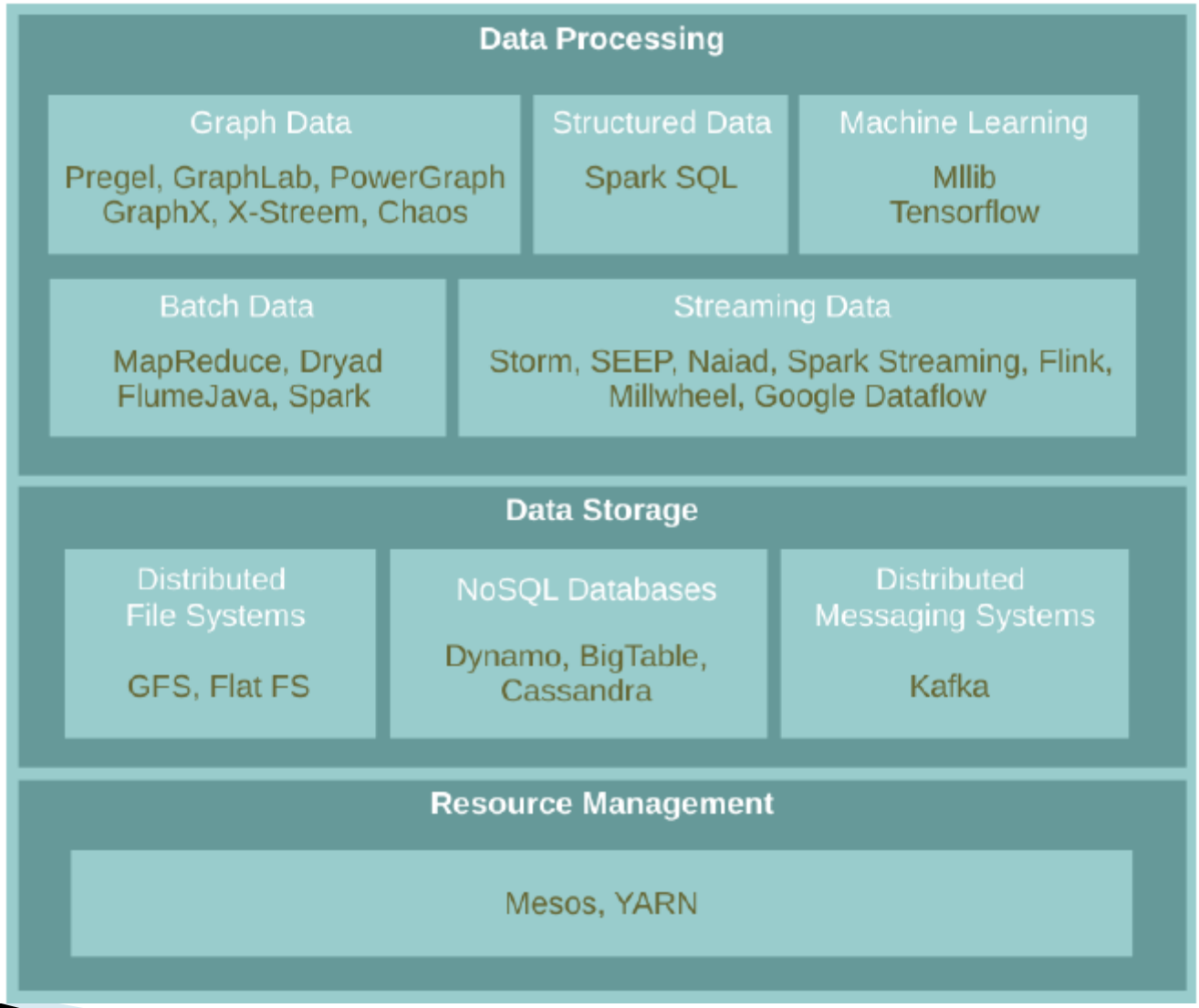
Lambda architectures



Lambda architecture with Spark

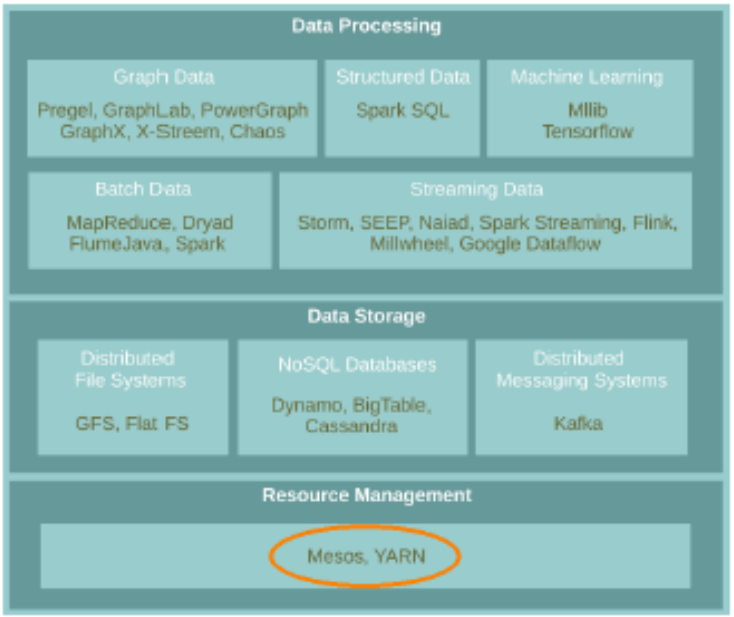


Big Data Stack



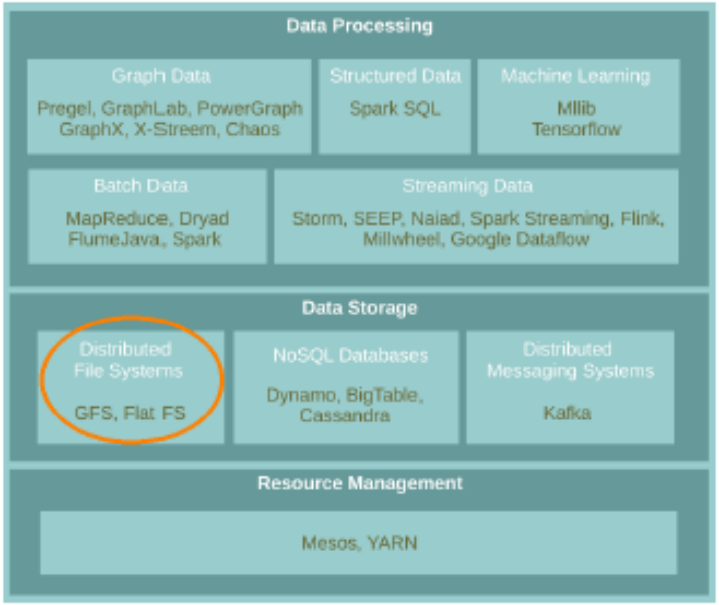
Resource Management

- Manage resources of a cluster
- Share them among the platforms
- Mesos, YARN, Borg, ...



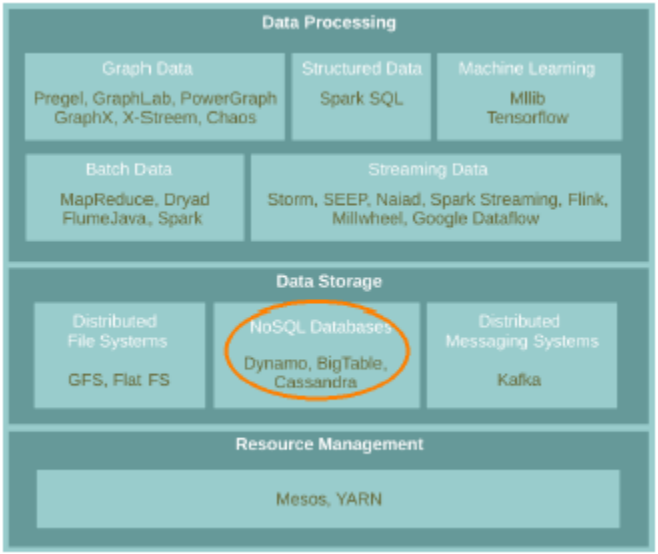
Data Storage - Distributed File Systems

- Store and retrieve files on/from distributed disks
- GFS, HDFS, FlatFS, ...



Data Storage - NoSQL Databases

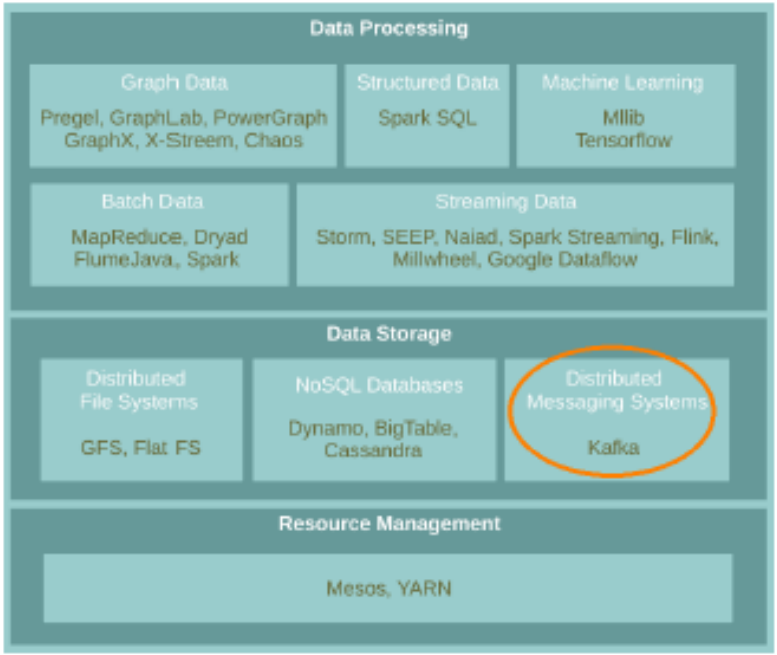
- BASE instead of ACID
- BigTable, Dyanamo, Cassandra, ...



Data Storage - Messaging Systems

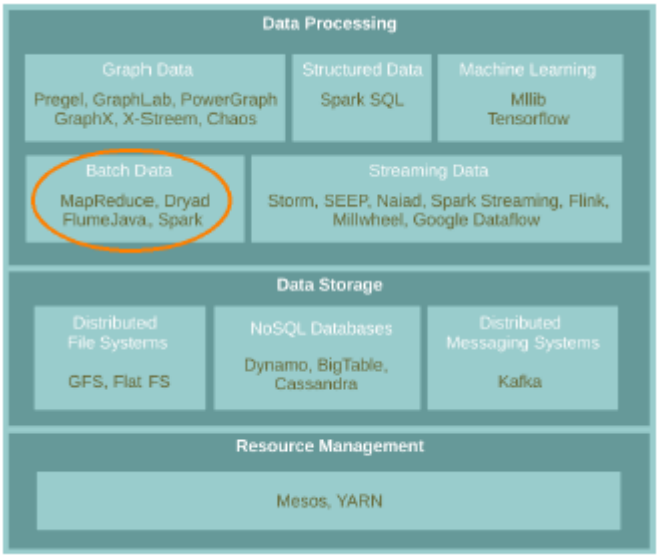
- Store streaming data
- Kafka, Flume, ActiveMQ, ...

kafka



Data Processing - Batch Data

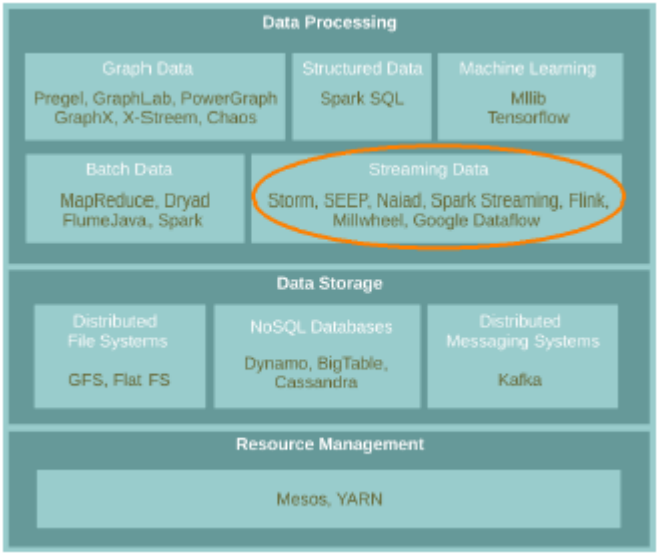
- Process data-at-rest
- Data-parallel processing model
- MapReduce, FlumeJava, Spark, ...



Data Processing - Streaming Data

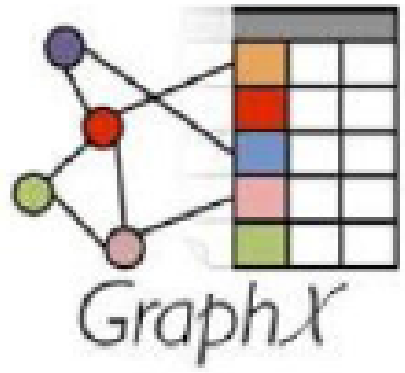
- Process data-in-motion
- Storm, Flink, Spark Streaming, ...

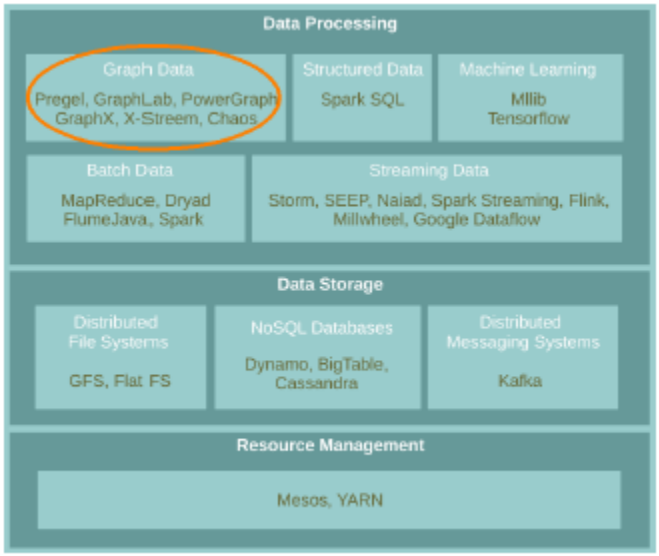
Storm



Data Processing - Linked Data (Graph)

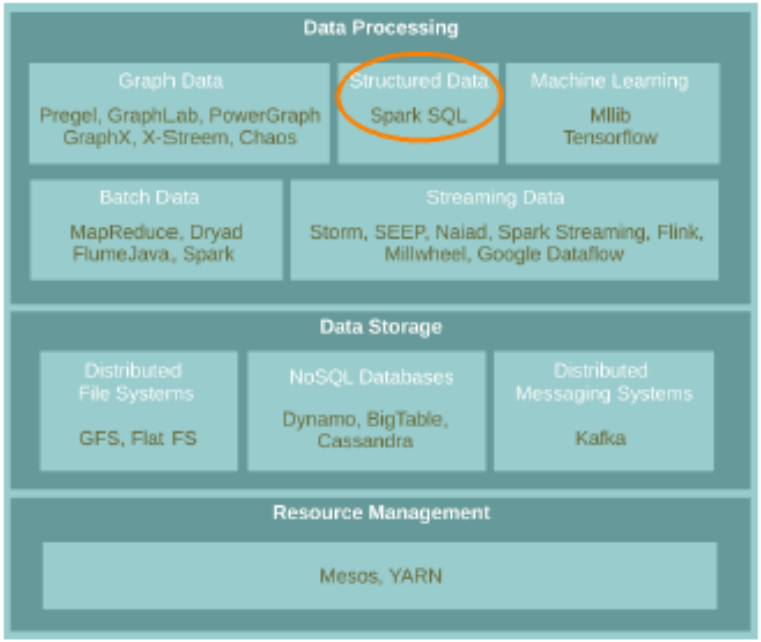
- Graph-parallel processing model
- Vertex-centric and Edge-centric programming model
- Pregel, GraphLab, GraphX, ...





Data Processing - Structured Data

- Take advantage of schemas in data to process
- Hive, Spark SQL, ...



Data Processing - Machine Learning

- Data analysis, e.g., supervised and unsupervised learning
- Mahout, Tensorflow, MLlib, ...