Big Data & MapReduce

Thomas Woo
Bell Labs
thomas.woo@nokia-bell-labs..com

Outline

- Big Data Overview
- MapReduce Framework
- Hadoop
- HDFS
- Building on Hadoop

Data are Plentiful

- There are lots of sources of large scale data
  - Weather data is collected on a daily basis by the U.S. National Oceanic and Atmospheric Administration (NOAA) to aide in climate, ecosystem, weather and commercial research. NOAA collects 80 TB of scientific data daily, and a ten-fold increase is expected by 2020.
  - U.S. National Aeronautics and Space Administration (NASA)
  - Energy companies have amassed huge amounts of geophysical data for geophysical analysis
  - Pharmaceutical companies have enormous amounts of drug testing data
  - Disney has customer data across its stores, theme parks, and Web properties
  - Internet – web pages, blogs, …

Use of Big Data

- Core business
  - Google search
  - Weather
  - Drug companies
- Competitive advantage
  - Business intelligence - organizations are turning to large repositories of corporate and external data to uncover trends, statistics, and other actionable information to help decide on their next move
  - Understand behavior – customer, NSA
- Compliance
  - Email
  - Supplier record

What is Big Data?

- Those data sets, along with their associated tools, platforms, and analytics, are often referred to as "Big Data”
- The biggest challenge facing those pursuing Big Data is getting a platform that can store and access all the current and future information and make it available online for analysis cost-effectively. That means a highly scalable platform.
- Such platforms consist of storage technologies, query languages, analytics tools, content analysis tools, and transport infrastructures.

Map & Reduce

Computing over Big Data

- Why is this hard
  - Just use bigger machines with faster CPUs
- Two problems
  - Single machine CPUs are hitting the upper limit
    - Moore's Law
    - Moving to more cores rather than faster clocks speeds
  - Failures
    - Job failure after multiple days
    - Cf. downloading big file
- Solution is distributed computing

Parallelization is hard

```
global_max = none
Def set_max(list):
        global global_max
        local_max = max(list)
        if local_max > global_max:
                global_max = local_max
```

- set_max([2, 34, 2, 56])
- set_max([98,6,7,2])
- Race condition, shared state

New Programming Paradigm

- Key: Avoid mutable state and side effects to make concurrency easier
- Imperative programming => functional programming
- Functional programming does this automatically
  - avoids mutating state and side-effects
  - purely functional programs have no side effects and are thus trivially parallelizable
- Advantages: More natural concurrency (without locks)
- Disadvantages: Little harder to understand (initially)
- The terms Map and Reduce come from Lisp and functional programming.

Functional Programming: Map

- map(): Take a function and apply it to a list of data.
- Example
  - times_2 = lambda x: 2*x
  - map(times_2, [1, 2, 3, 4, 5]) yields [2, 4, 6, 8, 10]
- Does it matter which direction map() is run?
- Will the results change if we split the list first, and run it on two different machines?
- No, because there are no side-effects!

Functional Programming: Reduce

- reduce(): Take a list and "fold" the elements together with some function
- Example
  - reduce(lambda x,y: x+y, [1, 2, 3, 4, 5]) yields 15
  - ((((1 + 2) + 3) + 4) + 5)
- Does it matter which direction reduce() is run?
- Will this also work if we run it over split data?

Functional Programming: Reduce

- Would reduce() give the same results if the function were
  - lambda x,y: x-y
- reduce(lambda x,y: x-y, [1, 2, 3, 4, 5]) yields -13
- reduce(lambda x,y: x-y, [5, 4, 3, 2, 1]) yields -5
- Reduce functions must be
  - Associative: (a + b) + c = a + (b + c)
  - Commutative: a + b = b + a

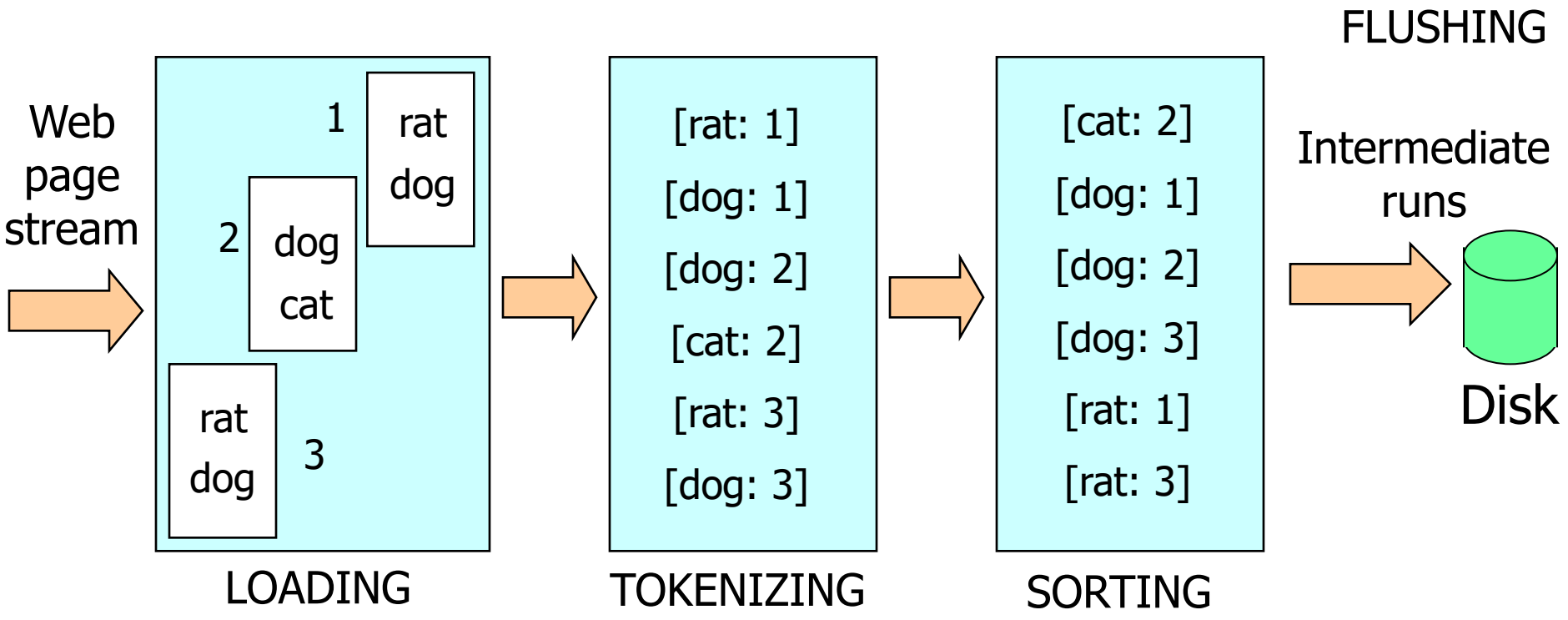
MapReduce

- The MapReduce framework is broken down into two functional areas:
  - Map, a function that parcels out work to different nodes in the distributed cluster
  - Reduce, a function that collates the work and resolves the results into a single value
- The essence of map/reduce and parallelization is the idea of dividing a problem, and conquering each part of it, and then combining the results

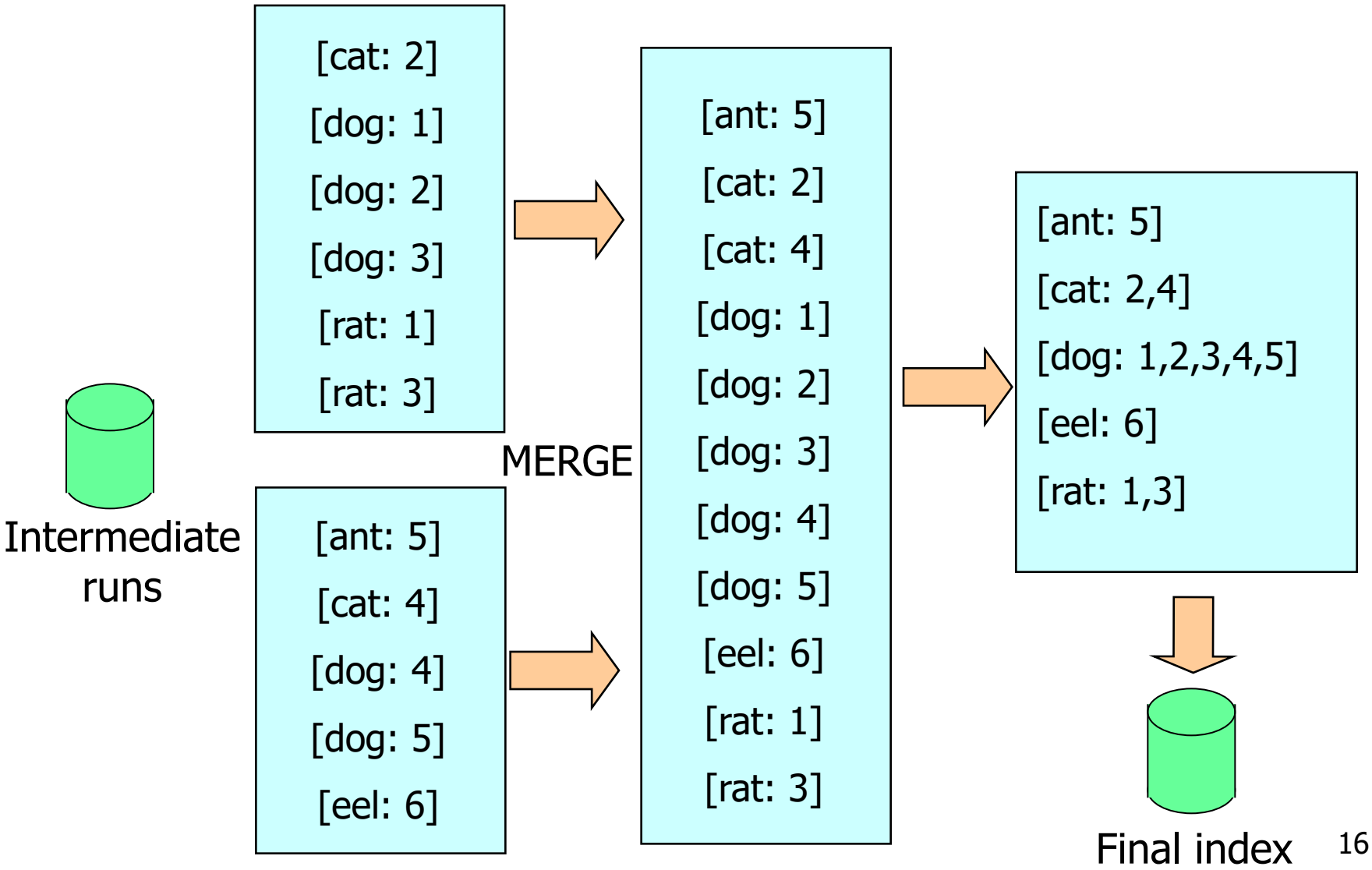
Fault Tolerance

- One of MapReduce's primary advantages is that it is fault-tolerant, which it accomplishes by monitoring each node in the cluster; each node is expected to report back periodically with completed work and status updates. If a node remains silent for longer than the expected interval, a master node makes note and reassigns the work to other nodes.

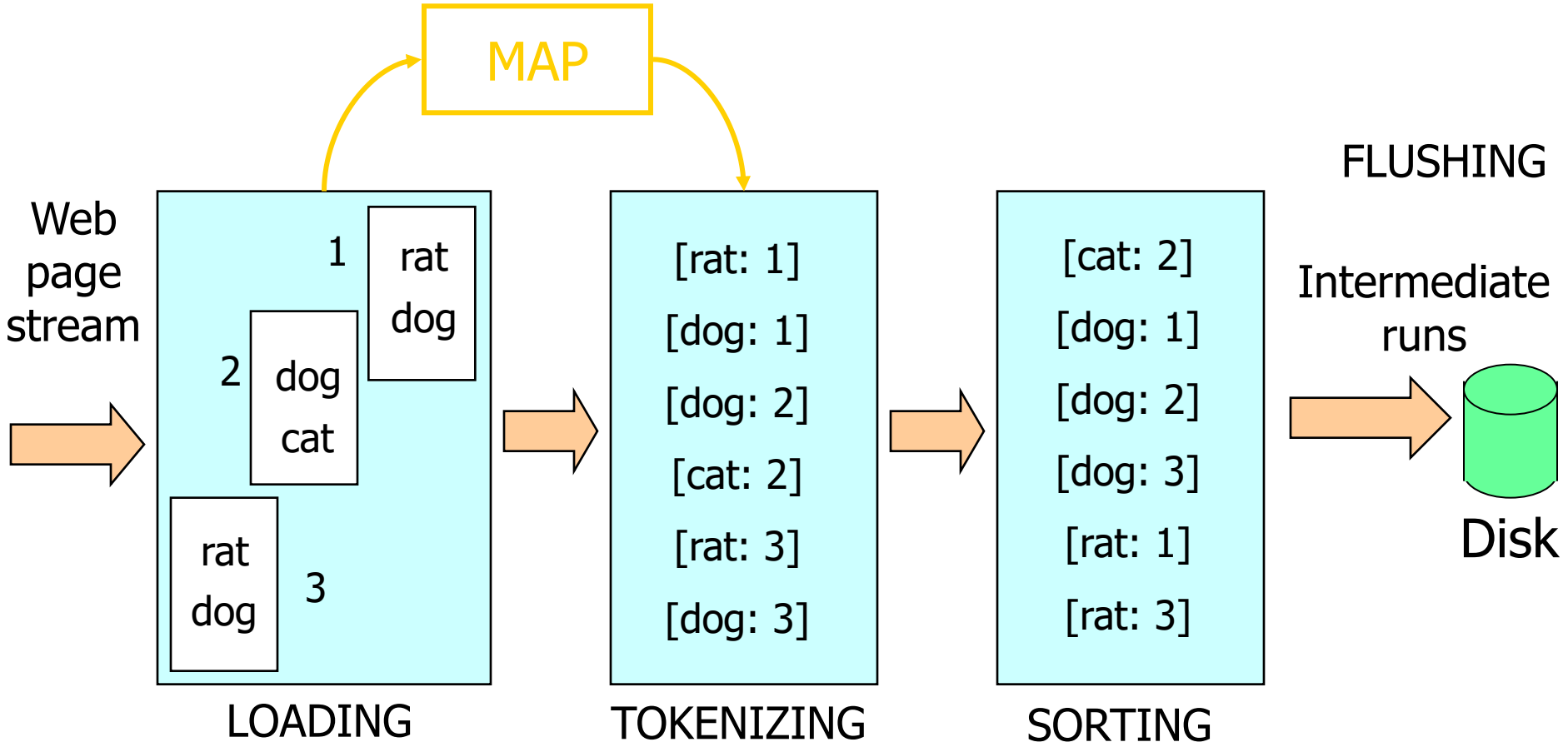
Example: Building a Text Index



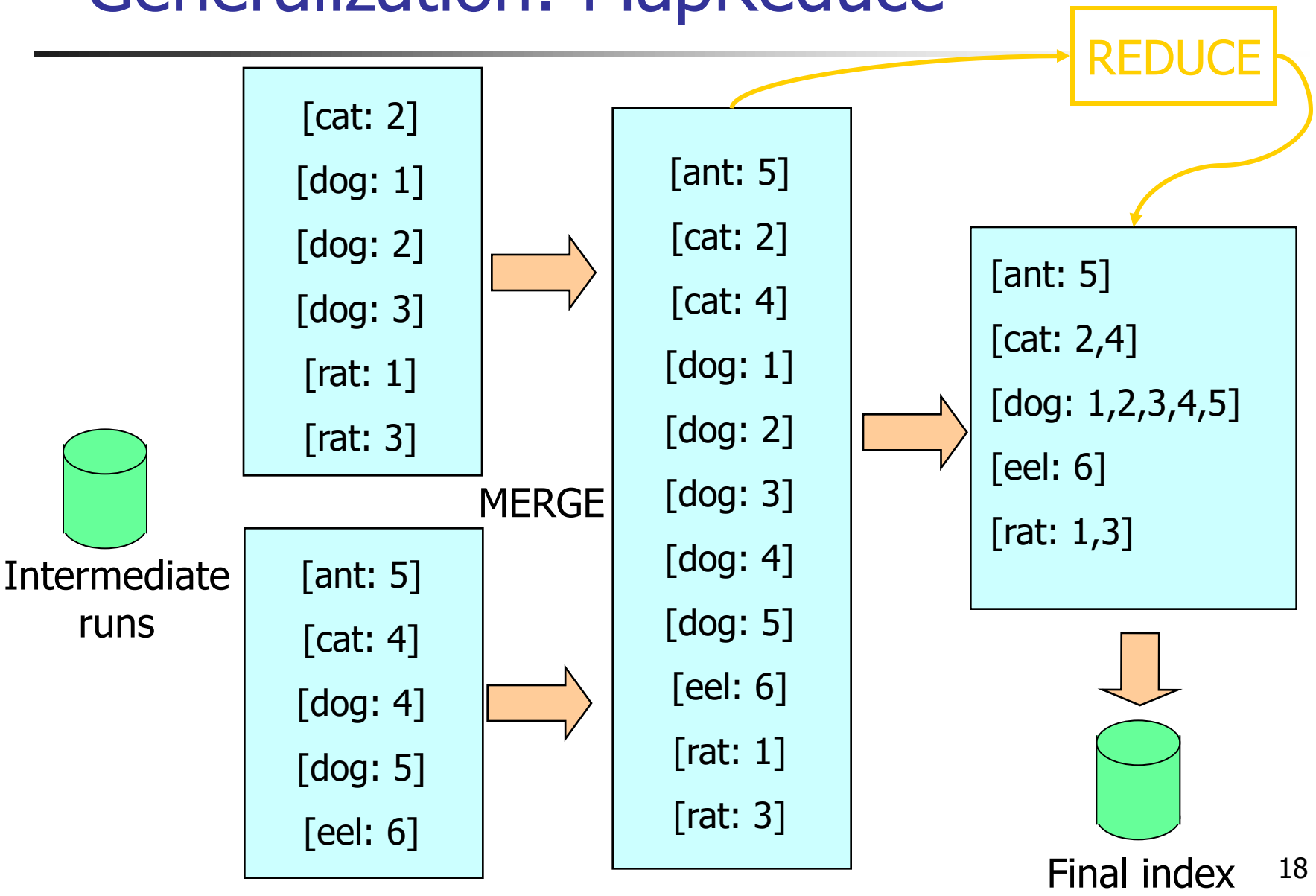
Example: Building a Text Index



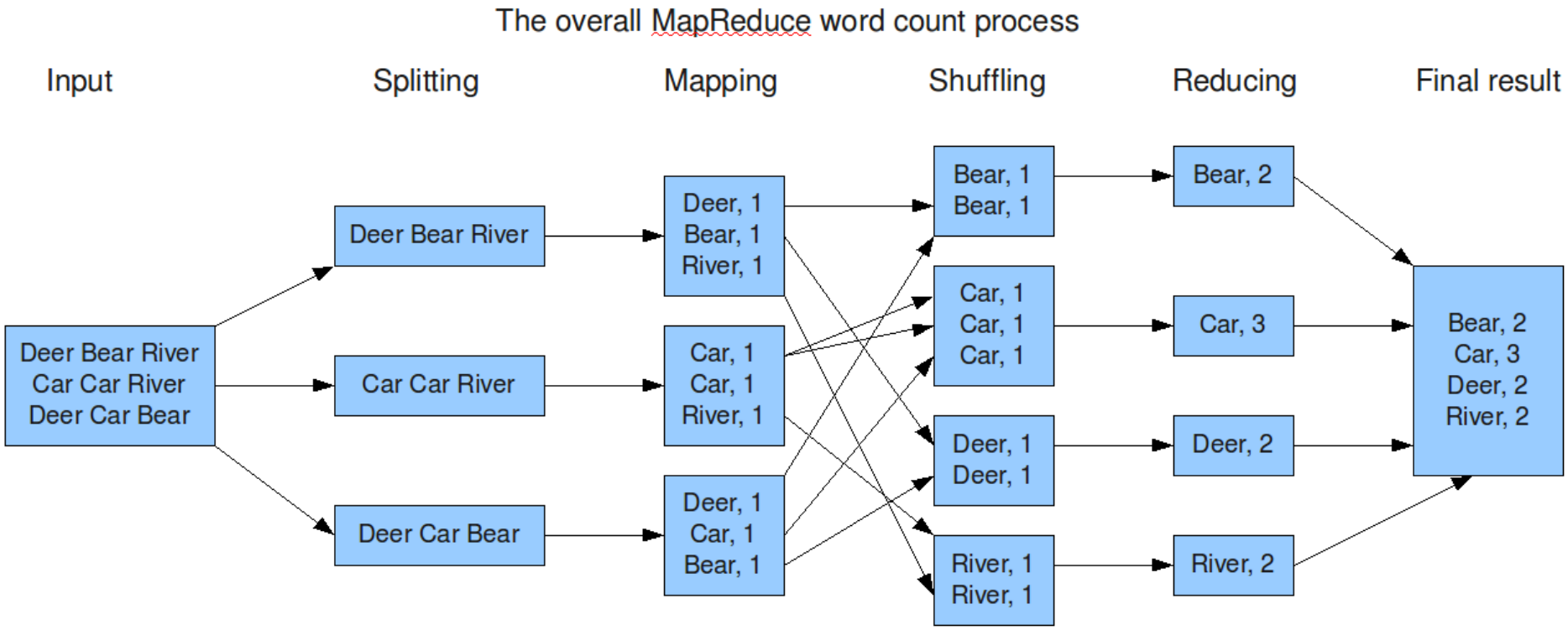
Generalization: MapReduce



Generalization: MapReduce



Exposing all the Stages



MapReduce

- Input
  - R = { r_1, r_2, …, r_n }
  - Functions **MAP**, **REDUCE**
  - **MAP** (r_i) → { [k_1: v_1], [k_2: v_2], … }
  - **REDUCE** (k_i, value bag) → "new" value for k_i
- Let
  - S = { [k: v] | [k: v] ∈ **MAP** (r) for some r ∈ R }
  - K = {{ k | [k: v] ∈ S, for any v }}
  - **V** (k) = { v | [k: v] ∈ S }
- Output
  - O = {{ [k: t] | k ∈ K, t = **REDUCE** (k, **V** (k)) }}

{{ … }} = set
{ … } = bag

Example: Inverted Index

- Map(string *key*, string *value*):

  // *key* is the document ID

  // *value* is the document body

  for each word *w* in *value* :

  EmitIntermediate(*w*, *key*)

- Example:
  - Map("w", "cat dog cat bat dog") emits
    [cat: w], [dog: w], [cat: w], [bat: w], [dog: w]

Example: Inverted Index

- Reduce(string *key*, string iterator *values*):

  // *key* is a word

  // *values* is a list of counts

  set *occurrences* = {}

  for each value *v* in *values* :

  occurrences += *v*

  EmitFinal(*key*, *occurrences*)

- Example:
  - Reduce("britney", ["w", "spears.com", "w", "w"]) emits "britney", ["w","spears.com"]

Example: Counting Word Occurrences

- Map(string *key*, string *value*):

  // *key* is the document ID

  // *value* is the document body

  for each word *w* in *value* :

  EmitIntermediate(*w*, '1')

- Example:
  - Map('29875', 'cat dog cat bat dog') emits
    ['cat': '1'], ['dog': '1'], ['cat': '1'], ['bat': '1'], ['dog': '1']

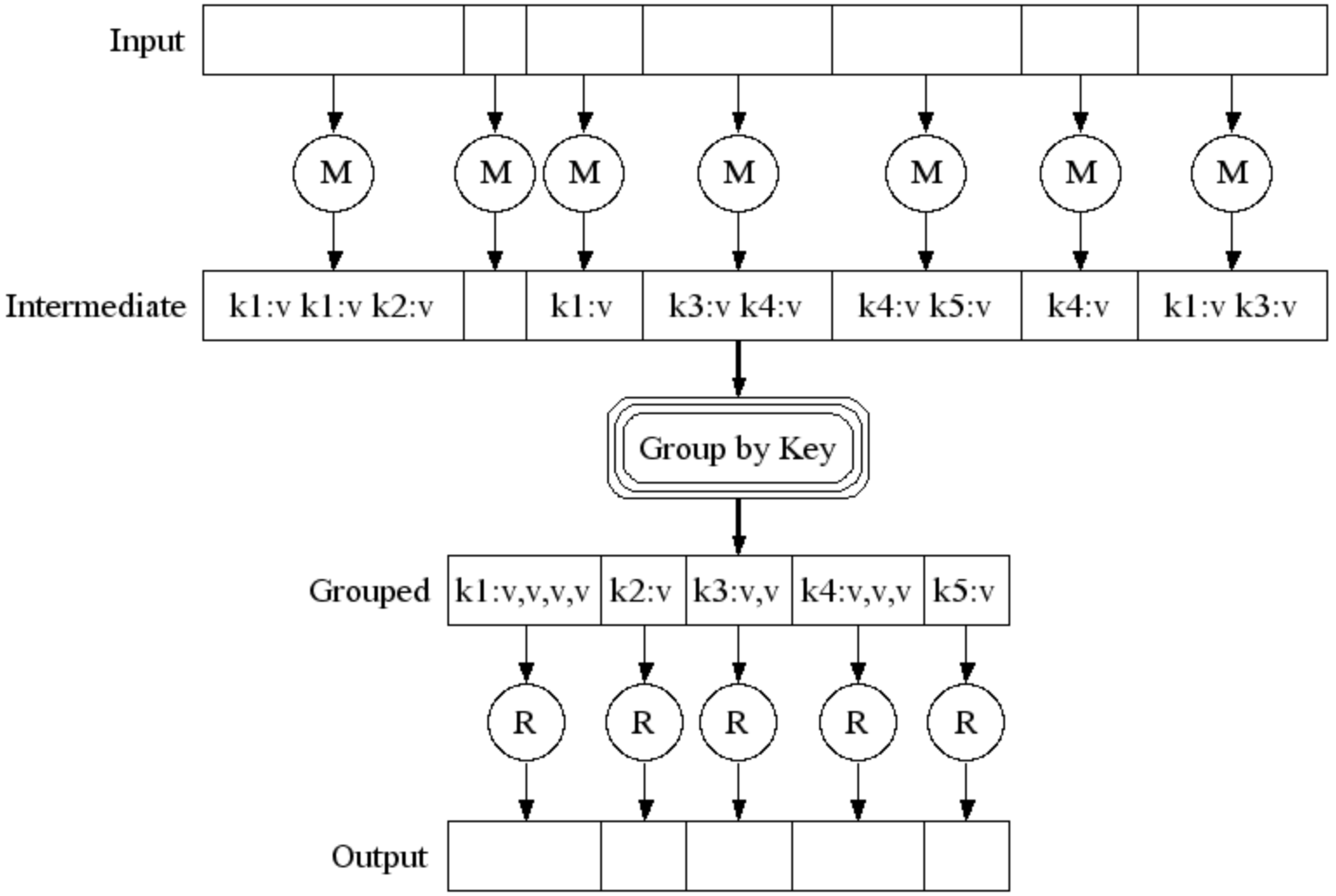
Example: Counting Word Occurrences

- Reduce(string *key*, string iterator *values*):

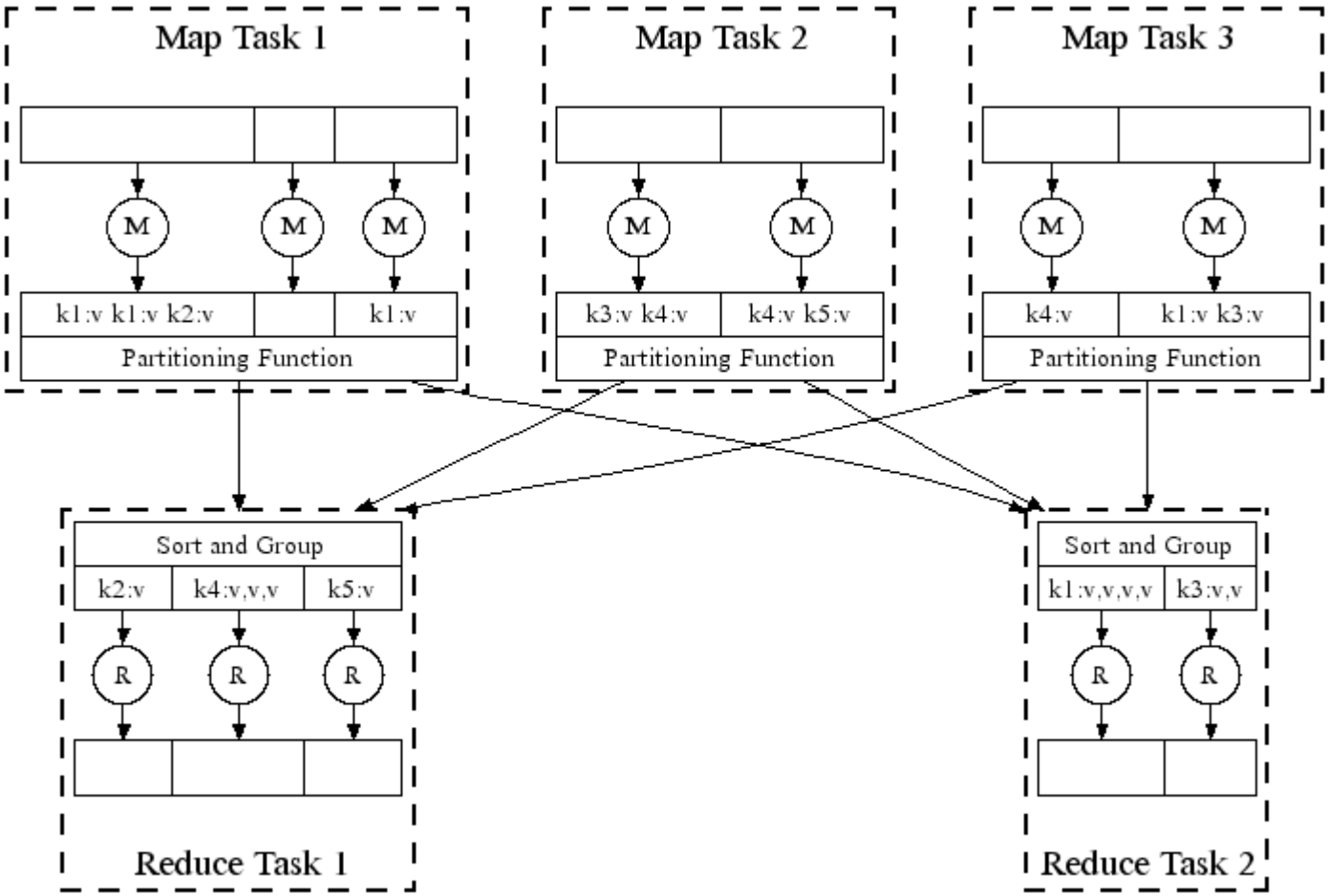
```
// key is a word
// values is a list of counts
int result = 0
for each value v in values :
    result += ParseInteger(v)
EmitFinal(ToString(result))
```

- Example:
  - Reduce('dog', { '1', '1', '1', '1' }) emits '4'

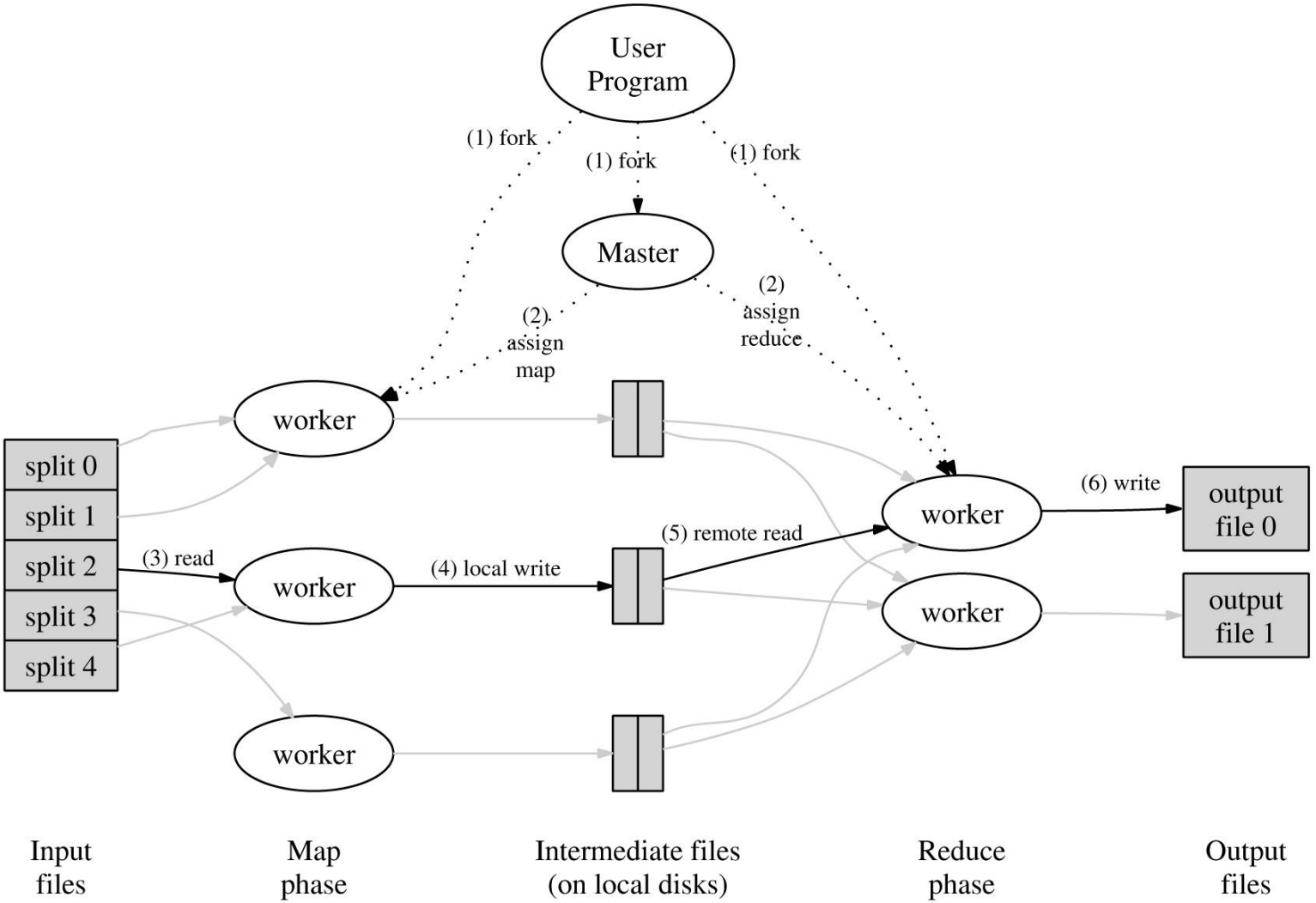
Logical Execution



Parallel Execution



Google MapReduce Architecture



Implementation Issues

- File system
- Data partitioning
- Combine functions
- Failure handling
- Backup tasks

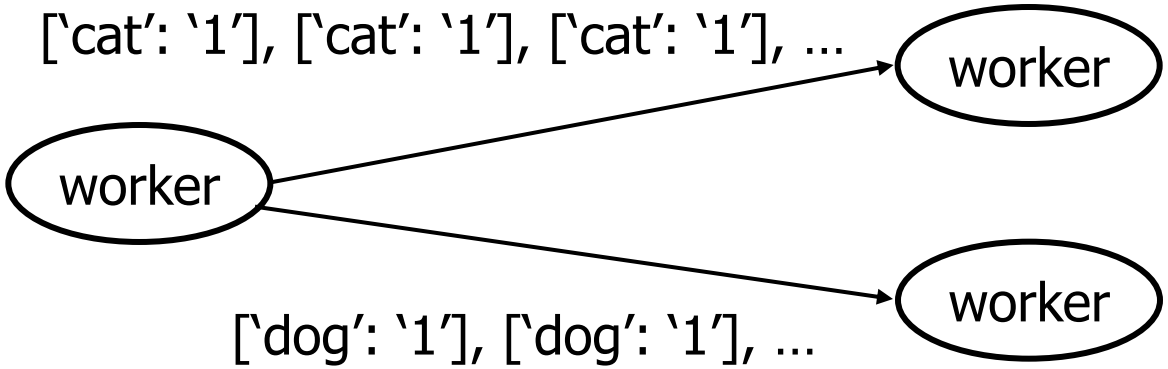
File system

- All data transfer between workers occurs through distributed file system
  - Support for split files
  - Workers perform local writes
  - Each map worker performs local or remote read of one or more input splits
  - Each reduce worker performs remote read of multiple intermediate splits
  - Output is left in as many splits as reduce workers

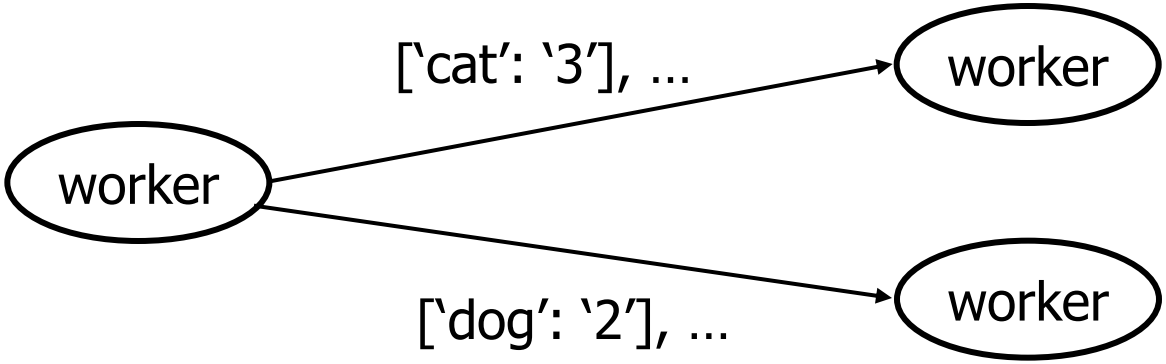
Data partitioning

- Data partitioned (split) by hash on key
- Each worker responsible for certain hash bucket(s)
- How many workers/splits?
  - Best to have multiple splits per worker
    - Improves load balance
    - If worker fails, splits could be re-distributed across multiple other workers
  - Best to assign splits to "nearby" workers
  - Rules apply to both map and reduce workers

Combine functions



Combine is like a local reduce applied (at map worker) before storing/distributing intermediate results:



Failure handling

- Worker failure
  - Detected by master through periodic pings
  - Handled via re-execution
    - Redo in-progress or completed map tasks
    - Redo in-progress reduce tasks
    - Map/reduce tasks committed through master
- Master failure
  - Not covered in original implementation
  - Could be detected by user program or monitor
  - Could recover persistent state from disk

Backup tasks

- Straggler: worker that takes unusually long to finish task
  - Possible causes include bad disks, network issues, overloaded machines
- Near the end of the map/reduce phase, master spawns backup copies of remaining tasks
  - Use workers that completed their task already
  - Whichever finishes first "wins"

Other Issues

- Handling bad records
  - Best is to debug and fix data/code
  - If master detects at least 2 task failures for a particular input record, skips record during 3rd attempt
- Debugging
  - Tricky in a distributed environment
  - Done through log messages and counters

MapReduce Advantages

- Easy to use
- General enough for expressing many practical problems
- Hides parallelization and fault recovery details
- Scales well, way beyond thousands of machines and terabytes of data

MapReduce Disadvantages

- One-input two-phase data flow rigid, hard to adapt
  - Does not allow for stateful multiple-step processing of records
- Procedural programming model requires (often repetitive) code for even the simplest operations (e.g., projection, filtering)
- Opaque nature of the map and reduce functions impedes optimization

Hadoop

Overview I

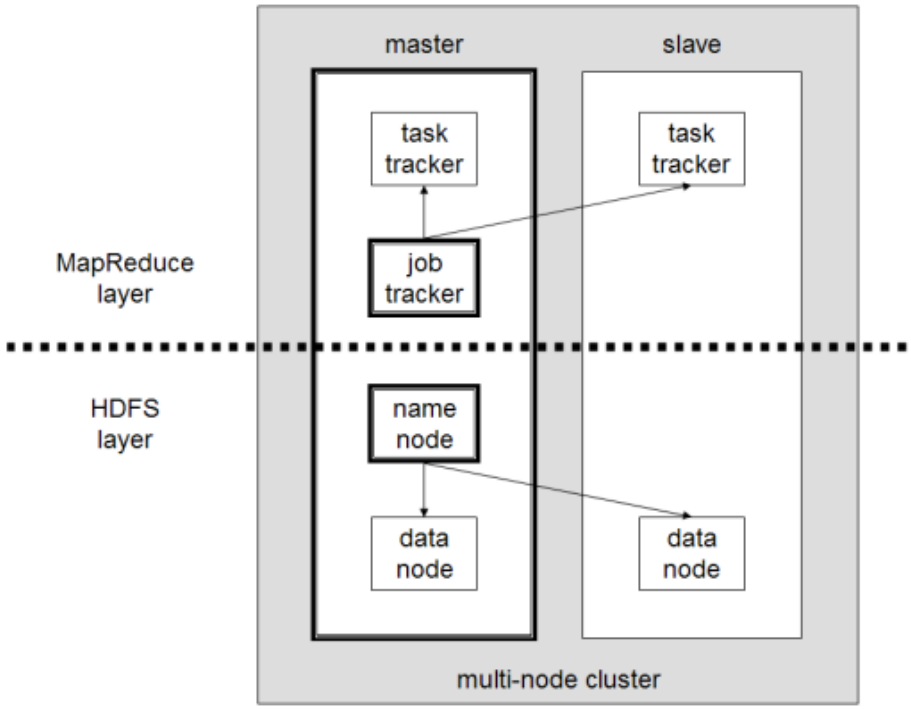
- Hadoop is a project administered by the Apache Software Foundation that consists of Google-derived technologies for building a platform to consolidate, combine, and understand data.
- Technically, Hadoop consists of two key services:
  - reliable data storage using the Hadoop Distributed File System (HDFS), and
  - high-performance parallel data processing using a technique called MapReduce.
- The goal of those services is to provide a foundation where the fast, reliable analysis of both structured and complex data becomes a reality

Overview II

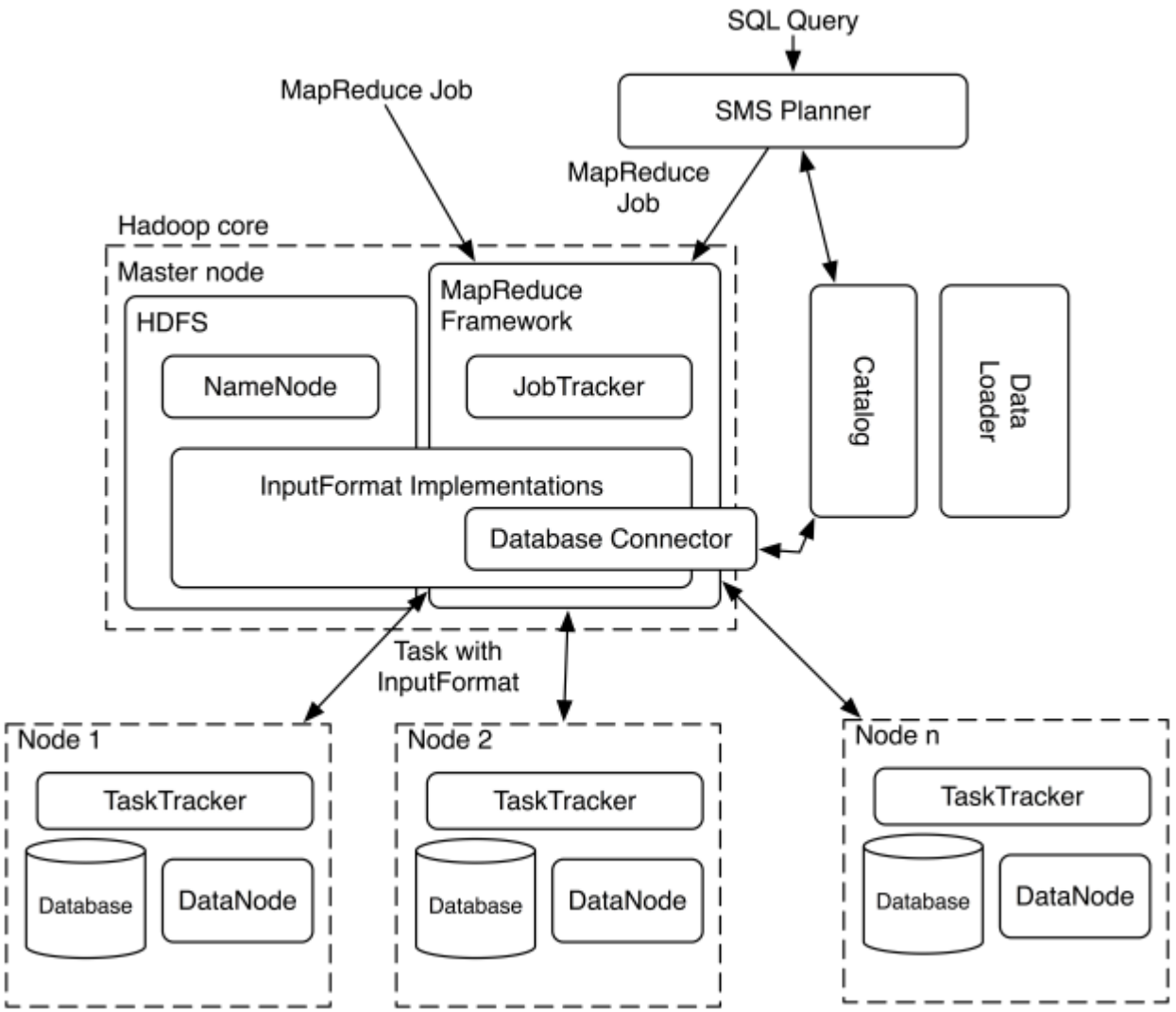
- Hadoop runs on a collection of commodity, shared-nothing servers. You can add or remove servers in a Hadoop cluster at will; the system detects and compensates for hardware or system problems on any server.
- Hadoop, in other words, is self-healing. It can deliver data - and run large-scale, high-performance processing jobs - in spite of system changes or failures.

A Hadoop Cluster

- A Hadoop cluster will include a single master and multiple worker nodes.
- The master node consists of a JobTracker, TaskTracker, NameNode, and DataNode.
- A slave or worker node acts as both a DataNode and TaskTracker, though it is possible to have data-only worker nodes, and compute-only worker nodes; these are normally only used in non-standard applications.



Key Hadoop Components



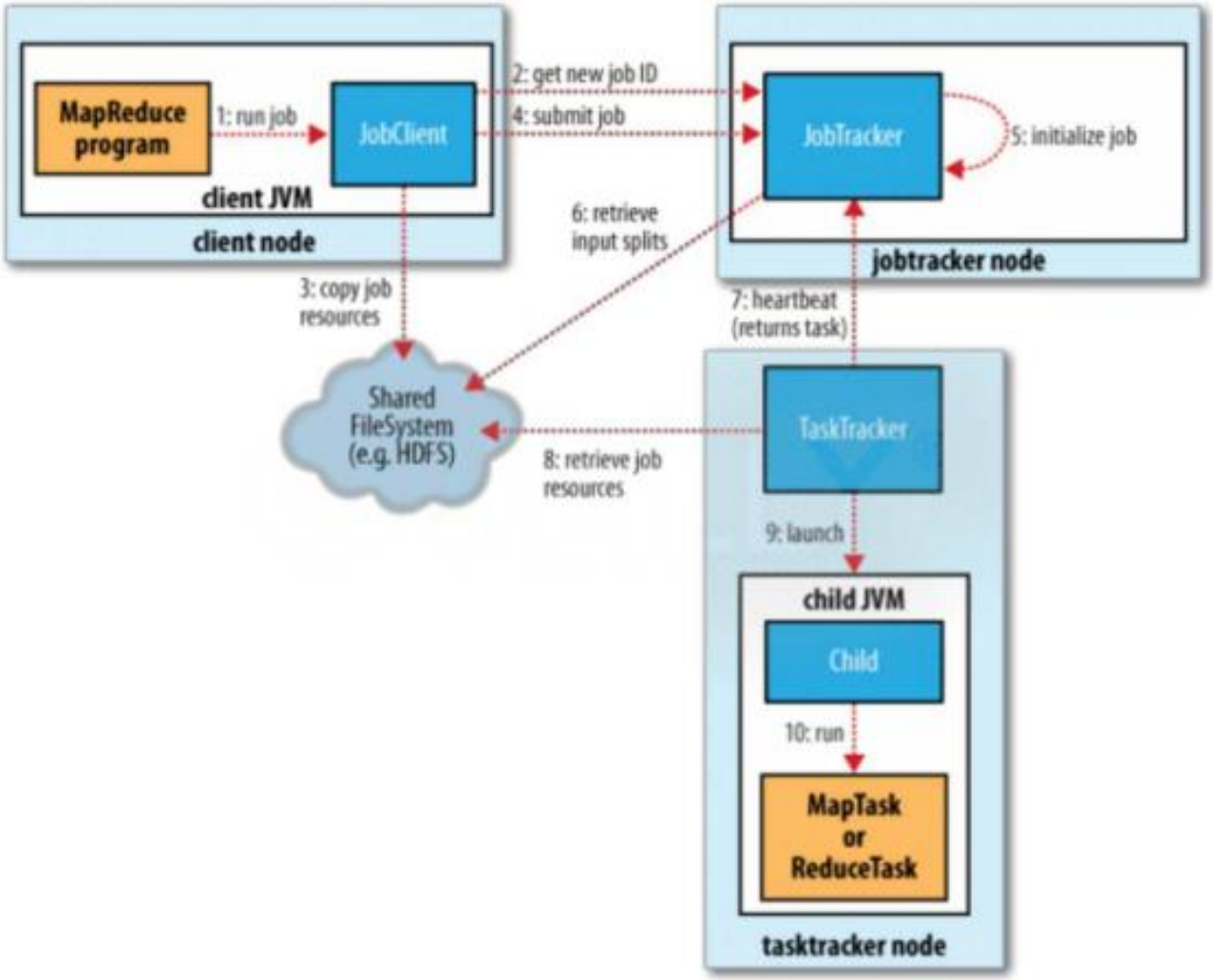
Hadoop JobTracker

- The JobTracker pushes work out to available TaskTracker nodes in the cluster, striving to keep the work as close to the data as possible.
- With a rack-aware filesystem, the JobTracker knows which node contains the data, and which other machines are nearby. If the work cannot be hosted on the actual node where the data resides, priority is given to nodes in the same rack. This reduces network traffic on the main backbone network.

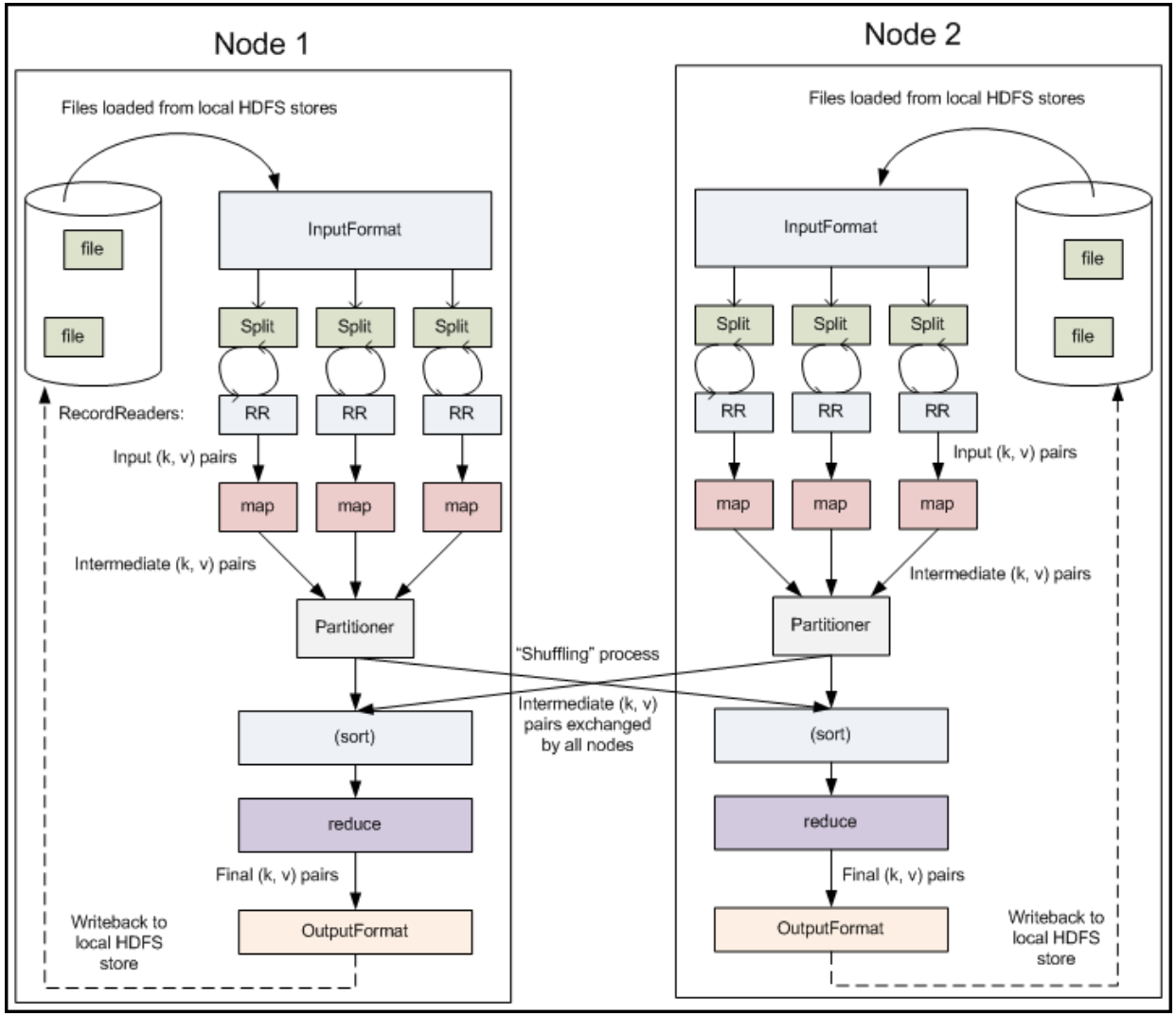
Hadoop TaskTracker

- The TaskTracker on each node spawns off a separate Java Virtual Machine process to prevent the TaskTracker itself from failing if the running job crashes the JVM.
- If a TaskTracker fails or times out, that part of the job is rescheduled.
- A heartbeat is sent from the TaskTracker to the JobTracker every few minutes to check its status.

How Hadoop Runs a Job



MapReduce in Hadoop



HDFS

HDFS I

- HDFS is fault tolerant and provides high-throughput access to large data sets
- HDFS's write-once-read-many model that relaxes concurrency control requirements, simplifies data coherency, and enables high-throughput access.
- HDFS assumes the viewpoint that it is usually better to locate processing logic near the data rather than moving the data to the application space.

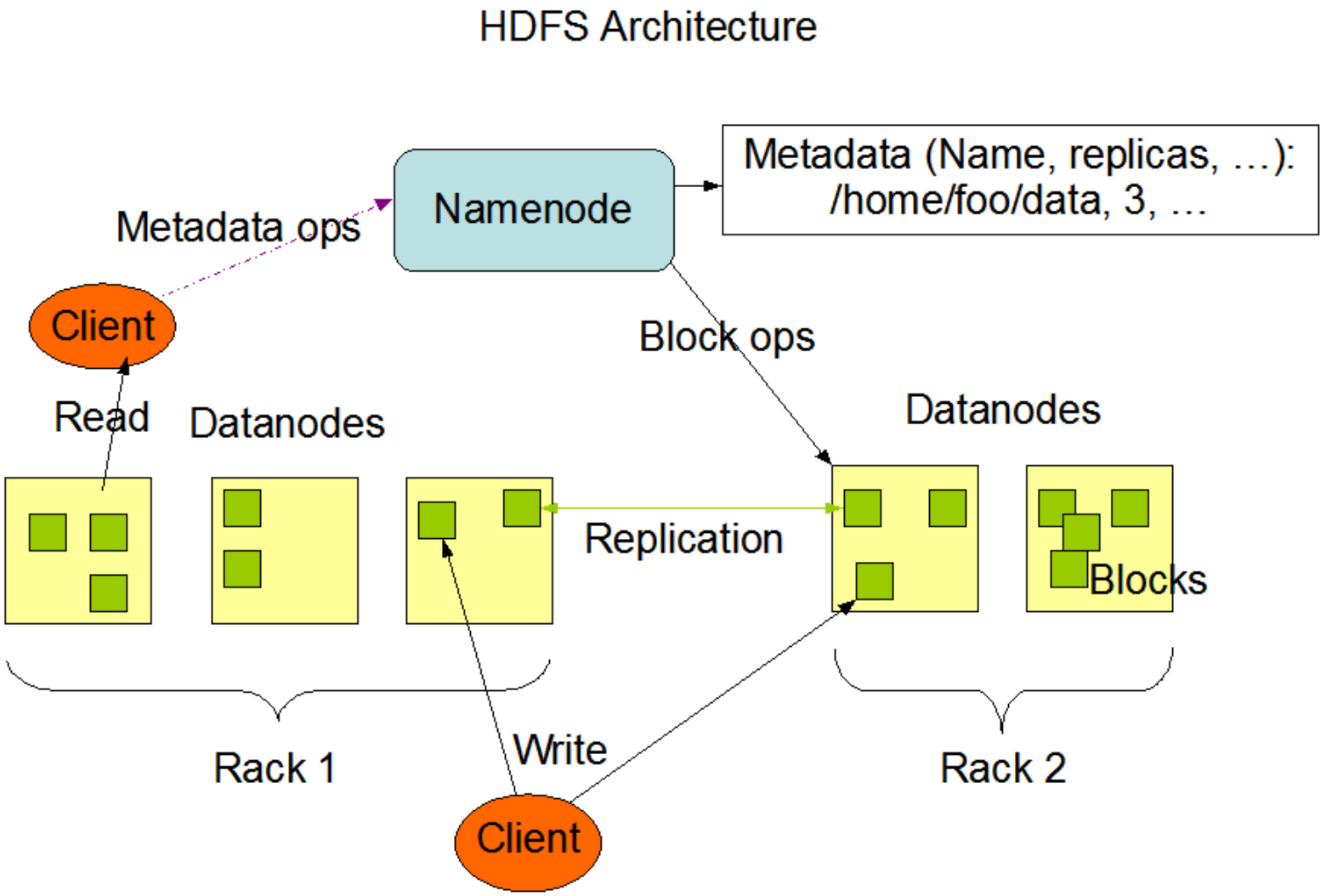
HDFS II

- HDFS is comprised of interconnected clusters of nodes where files and directories reside.
- An HDFS cluster consists of a single node, known as a NameNode, that manages the file system namespace and regulates client access to files. In addition, data nodes (DataNodes) store data as blocks within files.
- Data nodes continuously loop, asking the name node for instructions. A name node can't connect directly to a data node; it simply returns values from functions invoked by a data node.

HDFS III

- Each data node maintains an open server socket so that client code or other data nodes can read or write data. The host or port for this server socket is known by the name node, which provides the information to interested clients or other data nodes
- All HDFS communication protocols build on the TCP/IP protocol. HDFS clients connect to a Transmission Control Protocol (TCP) port opened on the name node, and then communicate with the name node using a proprietary Remote Procedure Call (RPC)-based protocol.

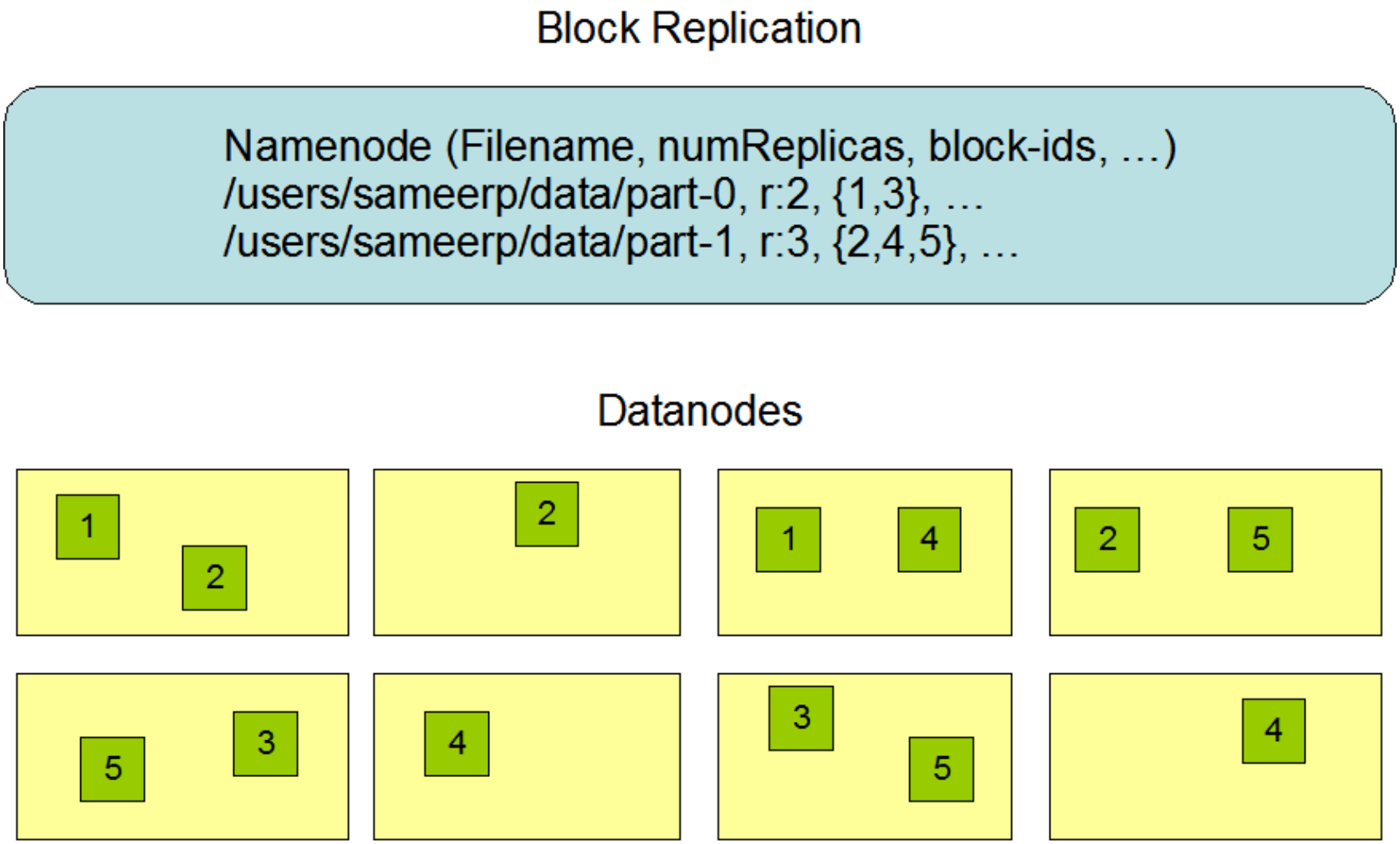
HDFS Architecture



HDFS
Data Replication

- HDFS replicates file blocks for fault tolerance. An application can specify the number of replicas of a file at the time it is created, and this number can be changed any time after that.
- The name node makes all decisions concerning block replication.
- HDFS uses an intelligent replica placement model for reliability and performance. Optimizing replica placement makes HDFS unique from most other distributed filesystems, and is facilitated by a rack-aware replica placement policy that uses network bandwidth efficiently.

HDFS
Block Replication



HDFS
Rack Awareness

- Typically, large HDFS clusters are arranged across multiple installations (racks). Network traffic between different nodes within the same installation is more efficient than network traffic across installations.
- A name node tries to place replicas of a block on multiple installations for improved fault tolerance.
- However, HDFS allows administrators to decide on which installation a node belongs. Therefore, each node knows its rack ID, making it rack-aware.

HDFS
Data Organization

- One of the main goals of HDFS is to support large files. The size of a typical HDFS block is 64MB. Therefore, each HDFS file consists of one or more 64MB blocks.
- HDFS tries to place each block on separate data nodes.

HDFS Heartbeats

- Each data node sends periodic heartbeat messages to its name node, so the latter can detect loss of connectivity if it stops receiving them. The name node marks as dead data nodes not responding to heartbeats and refrains from sending further requests to them. Data stored on a dead node is no longer available to an HDFS client from that node, which is effectively removed from the system.
- If the death of a node causes the replication factor of data blocks to drop below their minimum value, the name node initiates additional replication to bring the replication factor back to a normalized state.

HDFS
Data Integrity

- HDFS goes to great lengths to ensure the integrity of data across clusters. It uses checksum validation on the contents of HDFS files by storing computed checksums in separate, hidden files in the same namespace as the actual data.
- When a client retrieves file data, it can verify that the data received matches the checksum stored in the associated file.

Building On Hadoop

Many Add-ons

- HBase: A scalable, distributed database that supports structured data storage for large tables.
- Hive: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- Pig: A high-level data-flow language and execution framework for parallel computation.
- Chukwa: A data collection system for managing large distributed systems.
- ZooKeeper: A high-performance coordination service for distributed applications.

Pig & Pig Latin

- Layer on top of MapReduce (Hadoop)
  - Pig is the system
  - Pig Latin is the language, a hybrid between
    - A high-level declarative query language, such as SQL
    - A low-level procedural language, such as C++/Java/Python typically used to define Map() and Reduce()

Pig on Top of MapReduce

- Pig Latin program can be "compiled" into a sequence of mapreductions
- Load, for each, filter: can be implemented as map functions
- Group, store: can be implemented as reduce functions (given proper intermediate data)
- Cogroup and join: special map functions that handle multiple inputs split using the same hash function
- Depending on sequence of operations, include identity mapper and reducer phases as needed

Hive & HiveQL

- Data warehouse on top of Hadoop
  - Hive is the system
  - HiveQL is the language
    - Fully declarative, SQL-like
    - Most SQL features present
    - Supports custom mapreduce scripts
  - MetaStore system catalog
    - Table schemas and statistics
    - Also for keeping track of underlying distributed file structure

Apache Spark

Overview

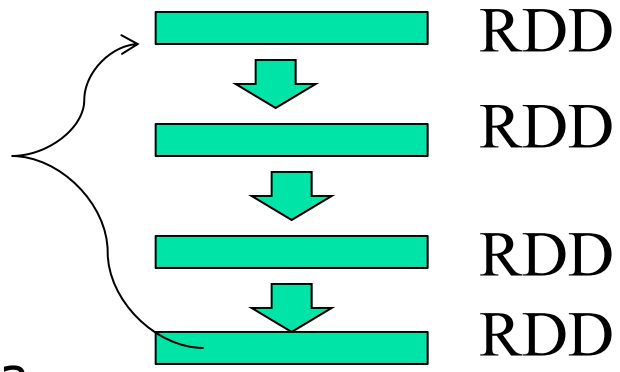
- Spark: Berkeley re-design of Mapreduce programming
- Focus on in-memory operation
- Support more general operations than just Map/Reduce

Key Concept

- Key abstraction: Resilient Distributed Datasets (RDD)

RDD: Resilient Distributed Datasets

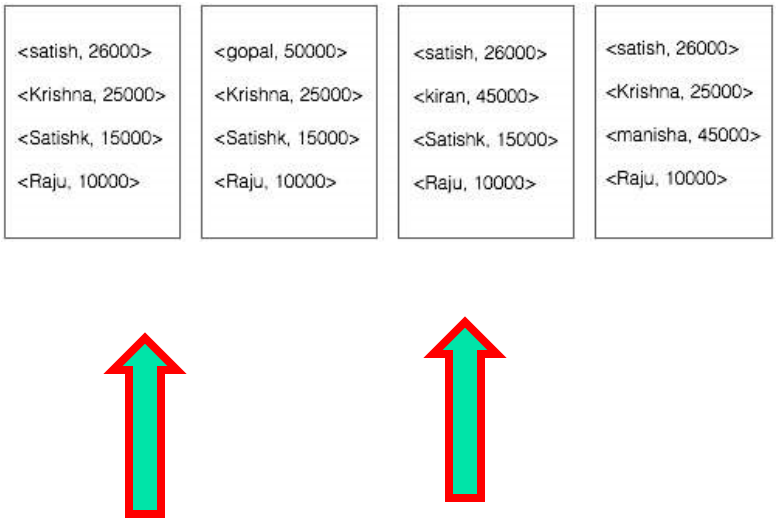
- Like a big list:
  - Collections of objects spread across a cluster, stored in RAM or on Disk
- Built through parallel transformations
- Automatically rebuilt on failure



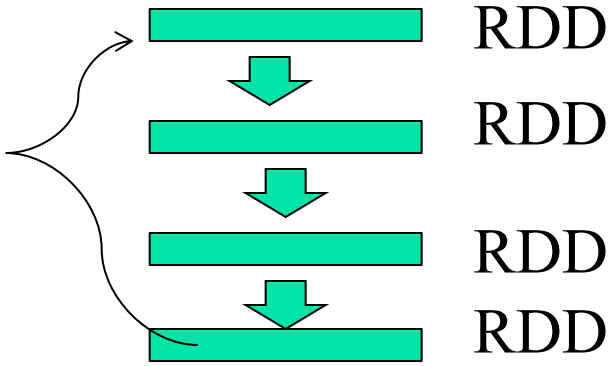
Operations

- Transformations (e.g. map, filter, groupBy)
- Make sure input/output match

MapReduce vs Spark

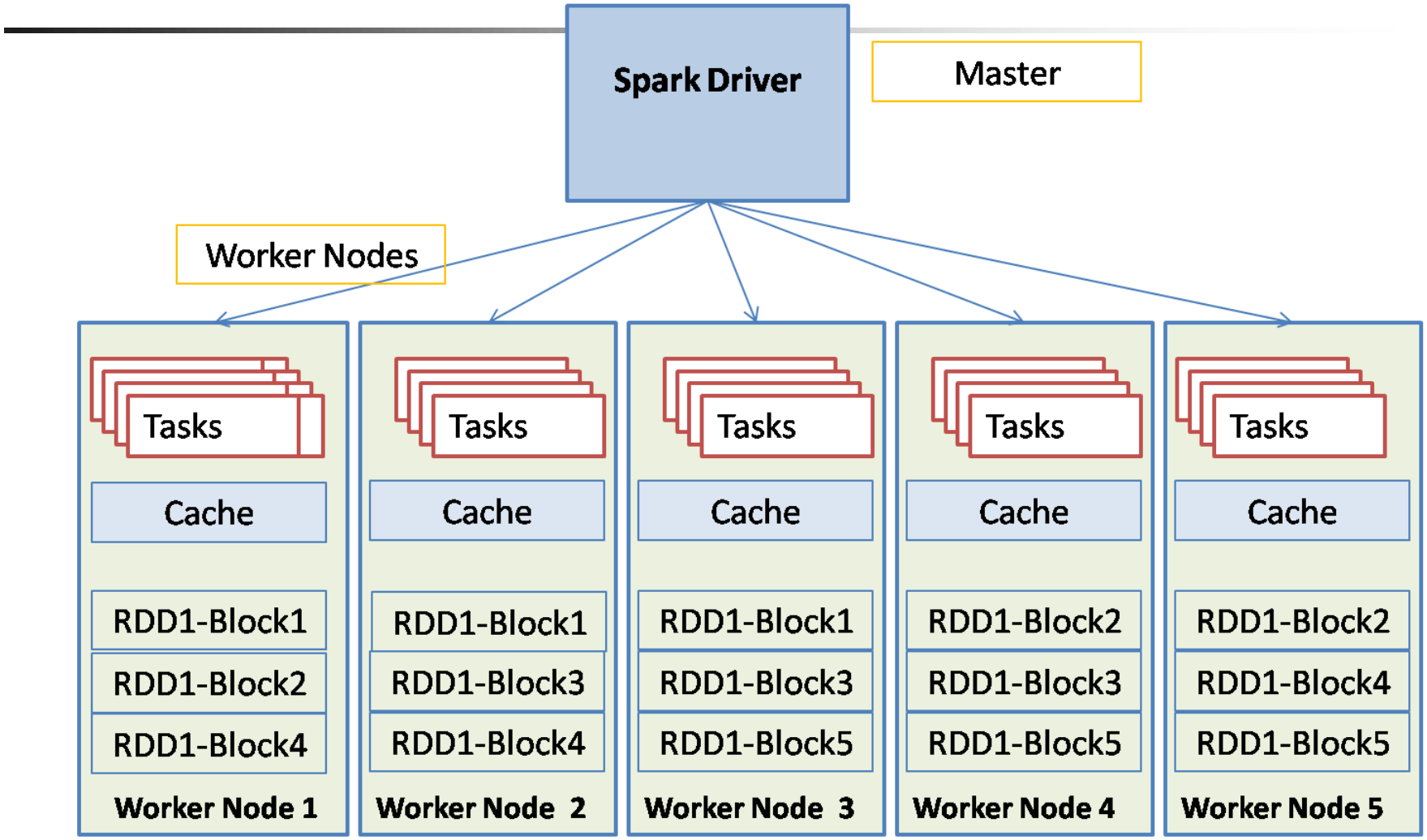


Map and reduce tasks operate on key-value pairs



Spark operates on RDD

Spark Components



Basic Transformations

```
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x)   // {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) // {4}
```

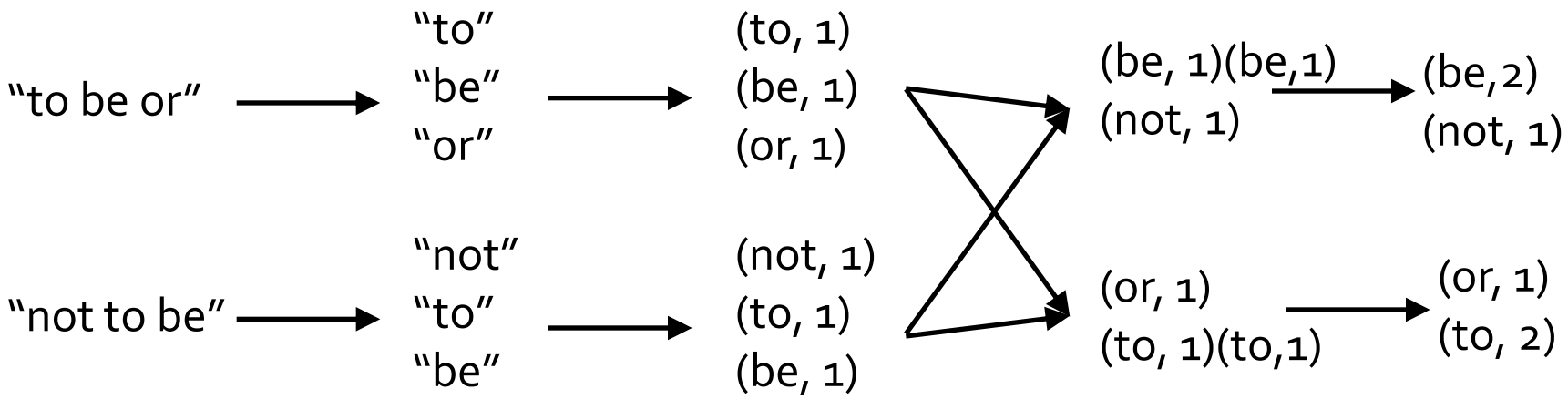
```
#read a text file and count number of lines
containing error

lines = sc.textFile("file.log")
lines.filter(lambda s: "ERROR" in s).count()
```

Basic Actions

```
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2)   # => [1, 2]

# Count number of elements
nums.count()   # => 3

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```

Example: Word Count

```
lines = sc.textFile("hamlet.txt")
counts = lines.flatMap(lambda line: line.split(" "))
              .map(lambda word: (word, 1))
              .reduceByKey(lambda x, y: x + y)
```



References

- Some slides adapted from Stanford CS347 lecture notes on Distributed Data Processing Using MapReduce
- Original Google MapReduce paper is: "MapReduce: Simplified Data Processing on Large Clusters", Jeffrey Dean and Sanjay Ghemawat, OSDI 2004
- "An introduction to the Hadoop Distributed File System", J. Jeffrey Hanson, IBM developerWorks
- "Anatomy of a MapReduce Job Run with Hadoop", at http://answers.oreilly.com/topic/459-anatomy-of-a-mapreduce-job-run-with-hadoop/

Hadoop MapReduce processes & data flow

Mathijs Homminga, Knowlogy 2007