Doc. RNDr. Irena Holubová, Ph.D. & PROFINIT

DATA SCIENCE

NDBI048

Big Data and Data Science

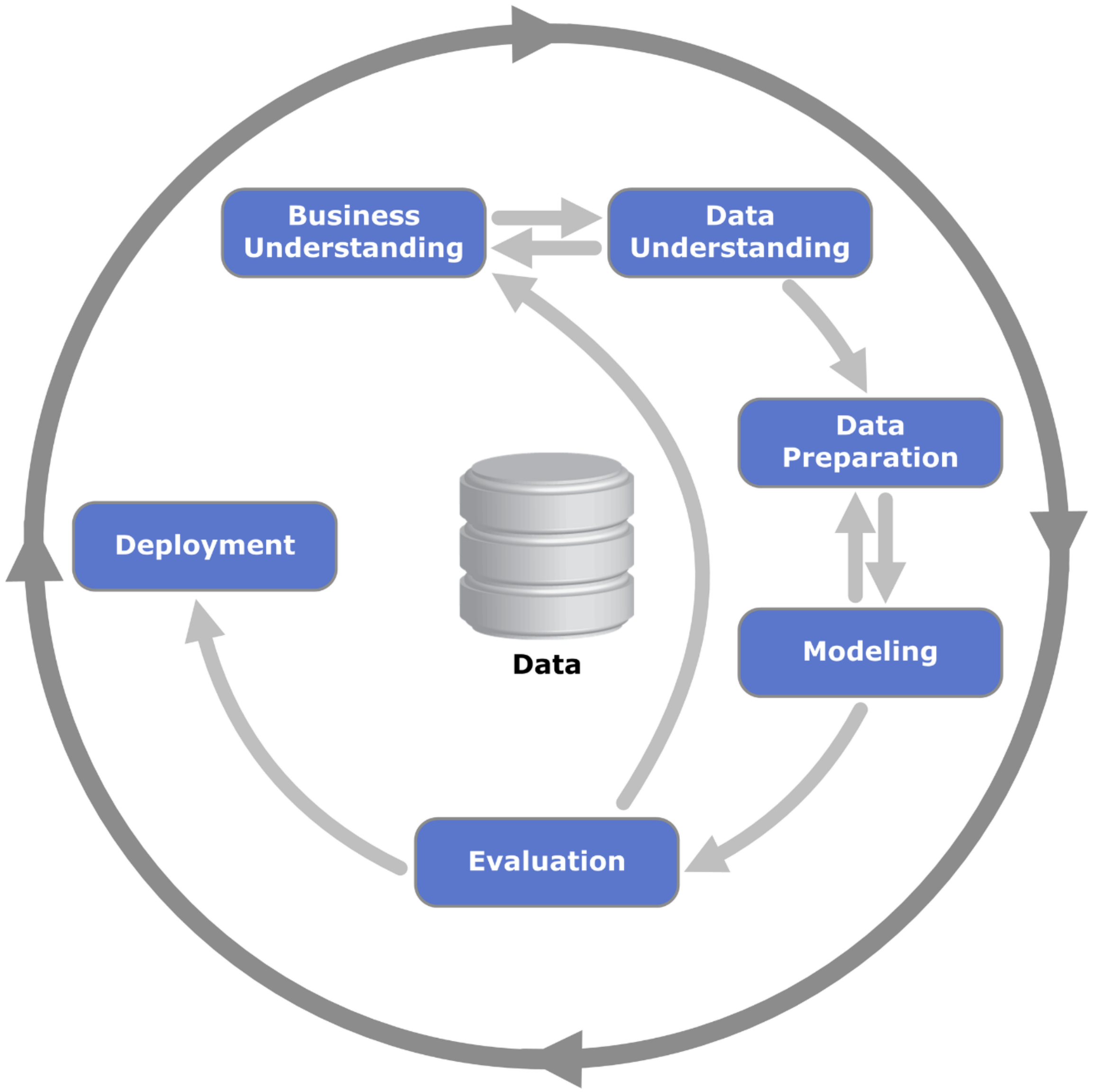
https://www.ksi.mff.cuni.cz/~holubova/NDBI048/

OUTLINE

- Big Data
- MapReduce
- Apache Spark

CRISP-DM PHASES

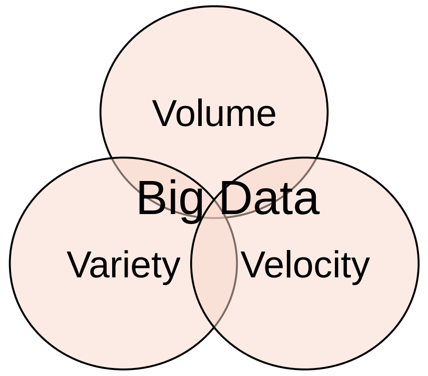
I. Business Understanding
II. Data Understanding
III. Data Preparation
IV. Modeling
V. Evaluation
VI. Deployment



https://www.datascience-pm.com/crisp-dm-2/

BIG DATA

WHAT IS BIG DATA?



- No standard definition
- First occurrence of the term: High Performance Computing (HPC)

Gartner: *"**Big Data**" is high **v**olume, high **v**elocity, and/or high **v**ariety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization.*

3 (4, 5) Vs

http://www.gartner.com/

WHO IS Gartner ?

- Information technology research and advisory company
- Founded in 1979 by Gideon Gartner
- HQ in Stanford, Connecticut, USA
  - > 5,300 employees
  - > 12,400 client organizations
- Provides: competitive analysis reports, industry overviews, market trend data, product evaluation reports, …

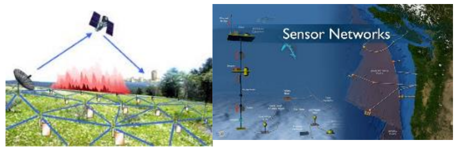
http://www.ibmbigdatahub.com/

WHAT IS BIG DATA?

Mobile devices
(tracking all objects all the time)

Social media and networks
(all of us are generating data)

Scientific instruments
(collecting all sorts of data)



Sensor technology and networks
(measuring all kinds of data)

IBM: *Depending on the industry and organization,* ***Big Data*** *encompasses information from internal and external sources such as transactions, social media, enterprise content, sensors, and mobile devices.*
Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.

FACEBOOK BY THE NUMBERS: STATS, DEMOGRAPHICS & FUN FACTS (LAST UPDATE: APRIL 2020)

- 2.5 billion monthly active users
- 5 billion comments are left on Facebook pages monthly
- 55 million status updates are made every day
- Every 60 seconds
  - 317,000 status updates
  - 147,000 photos uploaded
  - 54,000 shared links

https://www.omnicoreagency.com/facebook-statistics/

63 FACEBOOK STATISTICS YOU NEED TO KNOW IN 2022 (LAST UPDATE: JANUARY 4, 2022)

- 2.91 billion monthly active users
- Facebook has over 10 million advertisers
  - A Facebook user clicks on 12 ads on average every month
- On average, users spend 34 minutes on Facebook every day
- There were over 3.5 billion live feeds on Facebook towards the end of 2018
- 500 million people use Facebook Stories daily

https://www.omnicoreagency.com/facebook-statistics/

WHAT TO DO WITH BIG DATA?

- Association rule learning – discovering interesting relationships, i.e., "association rules," among variables in large databases
  - e.g., market basket analysis
- Classification – to identify the categories in which new data points belong, based on a training set containing data points that have already been categorized
  - Supervised learning
  - e.g., buying decisions
- Cluster analysis – classifying objects that split a diverse group into smaller groups of similar objects
  - Unsupervised learning
- Data fusion and data integration
- Signal processing

WHAT TO DO WITH BIG DATA?

- Crowdsourcing - collecting data submitted by a large group of people or community
- Data mining - extract patterns from large datasets
  - Involves association rule learning, cluster analysis, classification, regression, …
- Time series analysis and forecasting
  - e.g., hourly value of a stock market index
- Sentiment analysis - identifying the feature/aspect/product about which a sentiment is being expressed,
  - Determining the type (i.e., positive, negative, or neutral)
  - Determining the degree and strength of the sentiment
- Visualization
- …

MAIN PROBLEM: SCALABILITY

Vertical Scaling (scaling up)

- Traditional choice has been in favour of strong consistency
  - System architects have in the past gone in favour of scaling up (vertical scaling)
    - Involves larger and more powerful machines
- Works in many cases but…
- Vendor lock-in
  - Not everyone makes large and powerful machines
    - Who do, often use proprietary formats
  - Makes a customer dependent on a vendor for products and services
    - Unable to use another vendor

Horizontal Scaling (scaling out)

- Systems are distributed across multiple machines/nodes (horizontal scaling)
  - Commodity machines (cost effective)
  - Often surpasses scalability of vertical approach
- But…
- Fallacies of distributed computing:
  - The network is reliable
  - Latency is zero
  - Bandwidth is infinite
  - The network is secure
  - Topology does not change
  - There is one administrator
  - Transport cost is zero
  - The network is homogeneous

2012

Big Data Landscape

Vertical Apps

PREDICTIVE POLICING

bloomreach

MYRRIX

Log Data Apps

splunk>

loggly

sumologic

Ad/Media Apps

rocketfuel

collective[i]

bluefin

Recorded Future

LuckySort

Media Science

TURN

DataXu

Data As A Service

kaggle

factual.

knoema beta

GNIP

DATASIFT

Windows Azure Marketplace

INRIX

LexisNexis

SPACE CURVE

LOQATE Everything Location

Business Intelligence

ORACLE | Hyperion

SAP

Business Objects

RJMetrics

Microsoft | Business Intelligence

IBM

COGNOS

birst

MicroStrategy

Autonomy

bime

DOMO

QlikView

Chart.io

GoodData

Analytics and Visualization

tableau

Palantir

OPERA

metaLayer

METAMARKETS

dataspora

TERADATA ASTER

centrifuge

sas

TIBCO

KARMASPHERE

panopticon Real-Time Visual Data Analysis

pentaho

Datameer

ClearStory

platfora

CIRRO

alteryx

visual.ly

AYATA

Analytics Infrastructure

Hortonworks

VERTICA An HP Company

MAPR

INFOBRIGHT

cloudera

PARACCEL

EMC²

GREENPLUM.

NETEZZA

kognitio

DATASTAX

EXASOL

calpont

Operational Infrastructure

COUCHBASE

10gen | MongoDB company

TERADATA

HADAPT

TERRACOTTA

VoltDB

MarkLogic

INFORMATICA

Infrastructure As A Service

amazon web services

Windows Azure

infochimps

Google BigQuery

Structured Databases

ORACLE

MySQL

Microsoft SQL Server

PostgreSQL

IBM DB2.

SYBASE

memsql

Technologies

hadoop

hadoop MapReduce

mahout

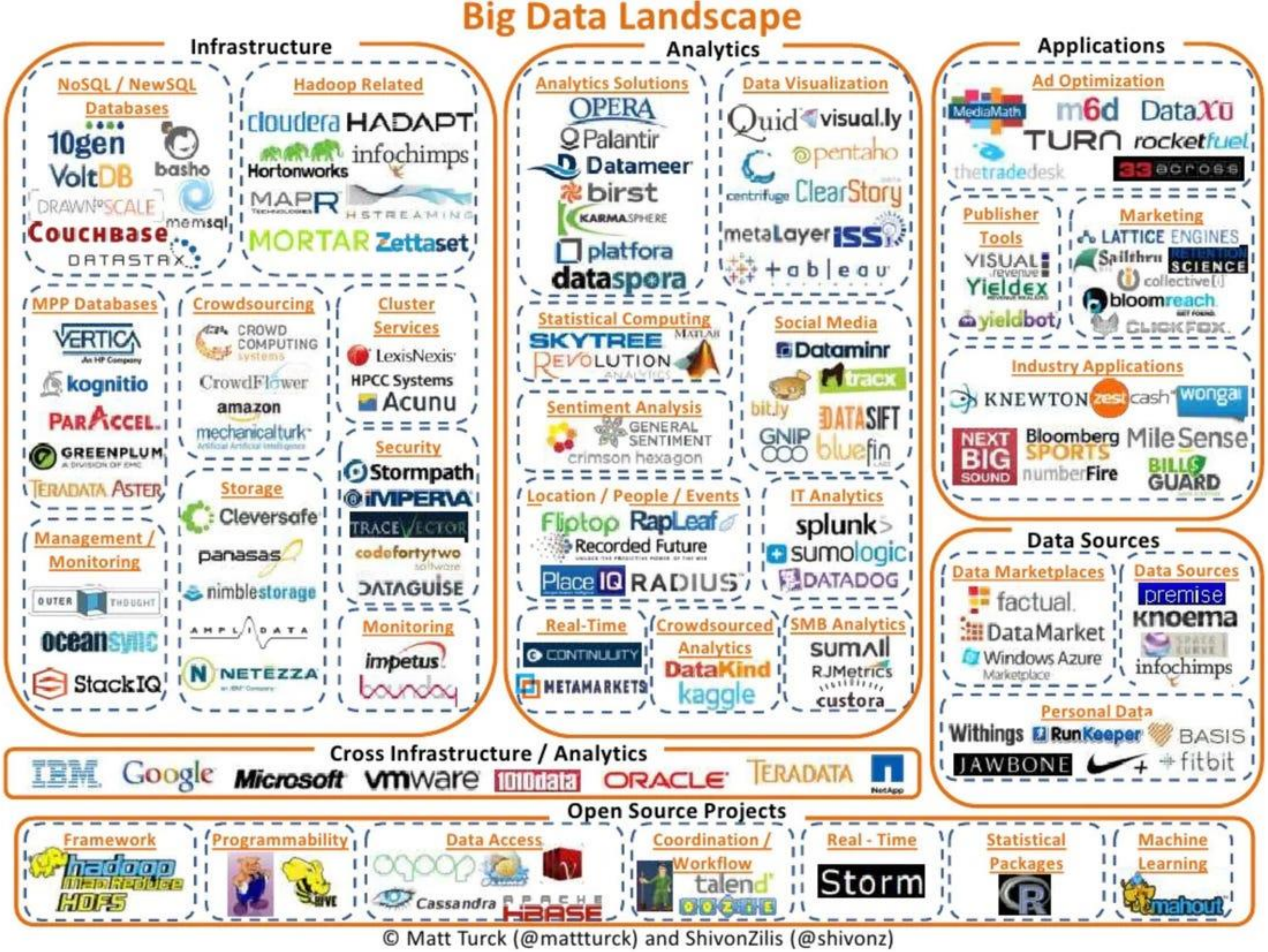
APACHE HBASE

Cassandra

dave@vcdave.com

blogs.forbes.com/davefeinleib

2021


Big Data Landscape
Infrastructure
NoSQL / NewSQL Databases
10gen
basho
VoltDB
DRAWNtoSCALE
memsql
Couchbase
DATASTAX
Hadoop Related
cloudera
HADAPT
Hortonworks
infochimps
MAPR
HSTREAMING
MORTAR
Zettaset
MPP Databases
VERTICA
kognitio
ParAccel
GREENPLUM
TERADATA ASTER
Crowdsourcing
CROWD COMPUTING systems
CrowdFlower
amazon mechanicalturk
Cluster Services
LexisNexis
HPCC Systems
Acunu
Management / Monitoring
OUTER THOUGHT
oceansync
StackIQ
Storage
Cleversafe
panasas
nimblestorage
AMPLIDATA
NETEZZA
Security
Stormpath
IMPERVA
TRACEVECTOR
codefortytwo software
DATAGUISE
Monitoring
impetus
boundary
Analytics
Analytics Solutions
OPERA
Palantir
Datameer
birst
KARMASPHERE
platfora
dataspora
Data Visualization
Quid
visual.ly
pentaho
centrifuge
ClearStory
metaLayer
ISS
tableau
Statistical Computing
SKYTREE
REVOLUTION ANALYTICS
MATLAB
Social Media
Dataminr
bit.ly
tracx
DATASIFT
GNIP
bluefin
Sentiment Analysis
GENERAL SENTIMENT
crimson hexagon
Location / People / Events
Fliptop
RapLeaf
Recorded Future
Place IQ
RADIUS
IT Analytics
splunk
sumologic
DATADOG
Real-Time
CONTINUITY
METAMARKETS
Crowdsourced Analytics
DataKind
kaggle
SMB Analytics
sumAll
RJMetrics
custora
Applications
Ad Optimization
MediaMath
m6d
DataXu
TURN
rocketfuel
thetradedesk
33across
Publisher Tools
VISUAL revenue
Yieldex
yieldbot
Marketing
LATTICE ENGINES
Sailthru
RETENTION SCIENCE
collective[i]
bloomreach
CLICKFOX
Industry Applications
KNEWTON
zest cash
wonga
NEXT BIG SOUND
Bloomberg SPORTS
MileSense
numberFire
BILLGUARD
Data Sources
Data Marketplaces
factual
DataMarket
Windows Azure Marketplace
Data Sources
premise
knoema
SPACE CURVE
infochimps
Personal Data
Withings
RunKeeper
BASIS
JAWBONE
+
fitbit
Cross Infrastructure / Analytics
IBM
Google
Microsoft
vmware
1010data
ORACLE
TERADATA
NetApp
Open Source Projects
Framework
hadoop
MapReduce
HDFS
Programmability
HIVE
Data Access
sqoop
Cassandra
APACHE HBASE
Coordination / Workflow
talend
oozie
Real - Time
Storm
Statistical Packages
R
Machine Learning
mahout
© Matt Turck (@mattturck) and ShivonZilis (@shivonz)

MAPREDUCE

MAPREDUCE FRAMEWORK

- A programming model + implementation
- Developed by Google in 2008
  - To replace old, centralized index structure
- Distributed, parallel computing on large data

Google: "A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs."

Example: Von Neumann's model = sequence of instructions

- Programming model in general:
  - Mental model a programmer has about execution of application
  - Purpose: improve programmer's productivity
  - Evaluation: expressiveness, simplicity, performance

MAPREDUCE FRAMEWORK

- Divide-and-conquer paradigm
  - Map breaks down a problem into sub-problems
    - Processes a key/value pair to generate a set of intermediate key/value pairs
  - Reduce receives and combines the sub-solutions to solve the problem
    - Processes intermediate values associated with the same intermediate key
- Many real-world tasks can be expressed this way
  - Programmer focuses on map/reduce code
  - Framework cares about data partitioning, scheduling execution across machines, handling machine failures, managing inter-machine communication, …

MAPREDUCE

A BIT MORE FORMALLY

- Map
  - Input: a key/value pair
  - Output: a set of intermediate key/value pairs
    - Usually different domain
  - $(k_1,v_1) \rightarrow \text{list}(k_2,v_2)$
- Reduce
  - Input: an intermediate key and a set of all values <u>for that key</u>
  - Output: a possibly smaller set of values
    - The same domain
  - $(k_2,\text{list}(v_2)) \rightarrow (k_2,\text{possibly smaller list}(v_2))$

MAPREDUCE

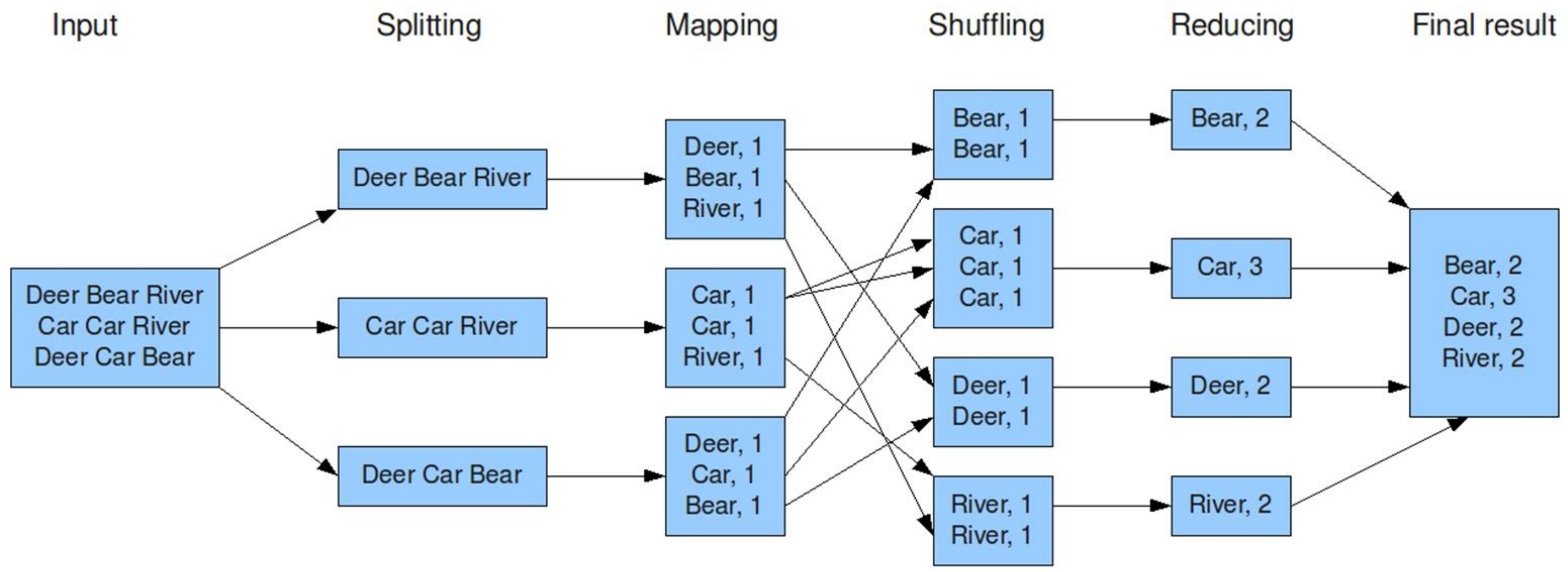
EXAMPLE: WORD FREQUENCY

```
map(String key, String value):
  // key: document name
  // value: document contents
for each word w in value:
  EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
int result = 0;
for each v in values:
  result += ParseInt(v);
Emit(key, AsString(result));
```

MAPREDUCE

EXAMPLE: WORD FREQUENCY



MAPREDUCE
MORE EXAMPLES

- distributed grep
  - Map: emits <word, line number> if it matches a supplied pattern
  - Reduce: identity
- URL access frequency
  - Map: processes web logs, emits <URL, 1>
  - Reduce: sums values and emits <URL, sum>
- reverse web-link graph
  - Map: <target, source> for each link to a target URL found in a page named source
  - Reduce: concatenates the list of all source URLs associated with a given target URL <target, list(source)>

MAPREDUCE
MORE EXAMPLES

- term vector per host
  - "Term vector" summarizes the most important words that occur in a document or a set of documents
  - Map: emits <hostname, term vector> for each input document
    - The hostname is extracted from the URL of the document
  - Reduce: adds the term vectors together, throws away infrequent terms
- inverted index
  - Map: parses each document, emits <word, document ID>
  - Reduce: sorts the corresponding document IDs, emits <word, list(document ID)>
- distributed sort
  - Map: extracts the key from each record, and emits <key, record>
  - Reduce: emits all pairs unchanged

MAPREDUCE

APPLICATION PARTS

- Input reader
  - Divides the input into appropriate size 'splits'
    - Each assigned to a single Map function
  - Reads data from stable storage
    - e.g., a distributed file system
  - Generates key/value pairs
- Map function
  - User-specified processing of key/value pairs
- Partition function
  - Map function output is allocated to a reducer
  - Partition function is given the key (output of Map) and the number of reducers and returns the index of the desired reducer
    - Default is to hash the key and use the hash value modulo the number of reducers

MAPREDUCE

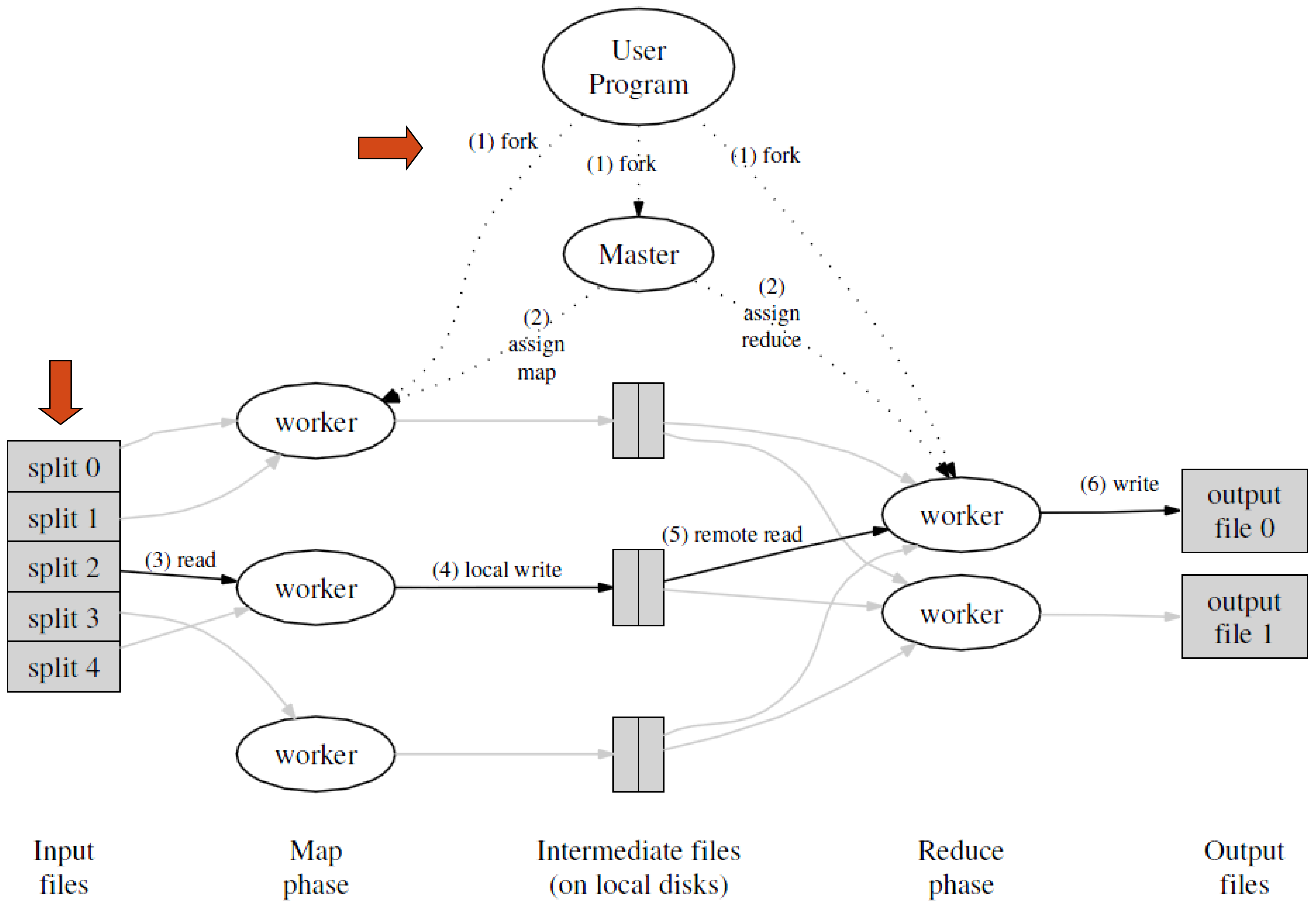
APPLICATION PARTS

- Compare function
  - Sorts the input for the Reduce function
- Reduce function
  - User-specified processing of key/values
- Output writer
  - Writes the output of the Reduce function to stable storage
    - e.g., a distributed file system

MAPREDUCE

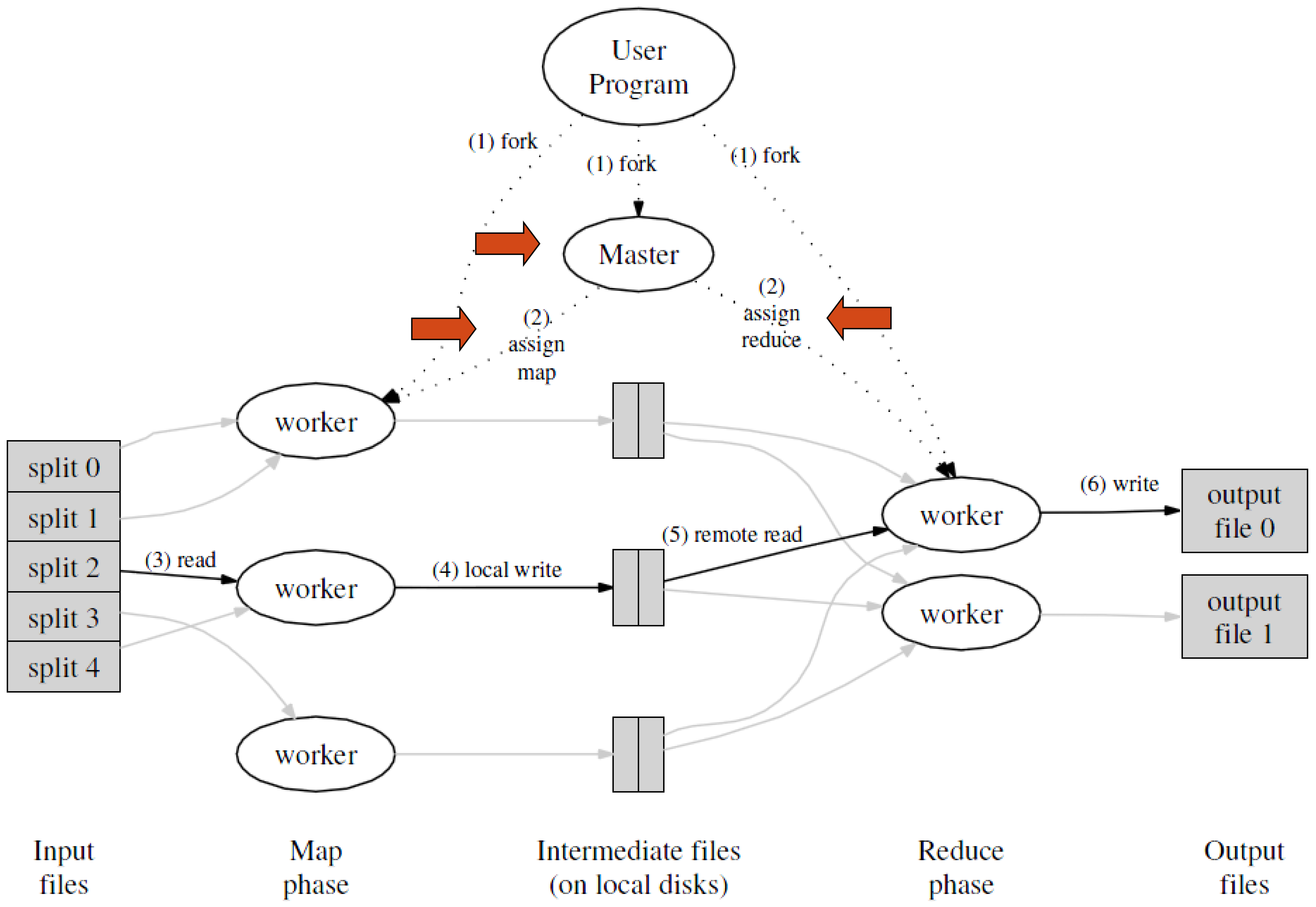
EXECUTION (GOOGLE) – STEP 1

1. MapReduce library in the user program splits the input files into *M* pieces
   - Typically 16 – 64 MB per piece
   - Controllable by the user via optional parameter
2. It starts copies of the program on a cluster of machines


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

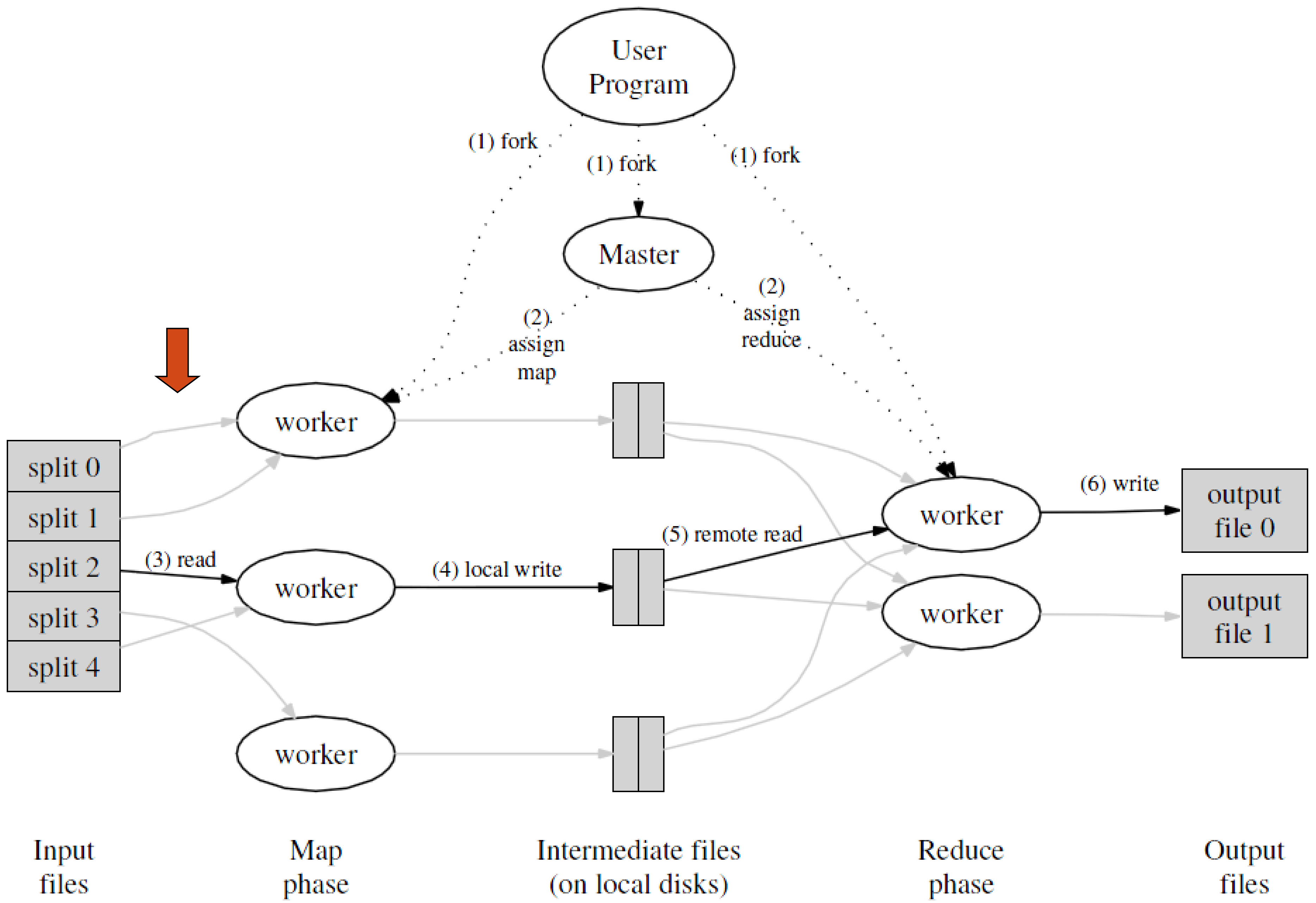
MAPREDUCE
EXECUTION – STEP 2

- Master = a special copy of the program
- Workers = other copies that are assigned work by master
- *M* Map tasks and *R* Reduce tasks to assign
- Master picks idle workers and assigns each one a Map task (or a Reduce task)


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

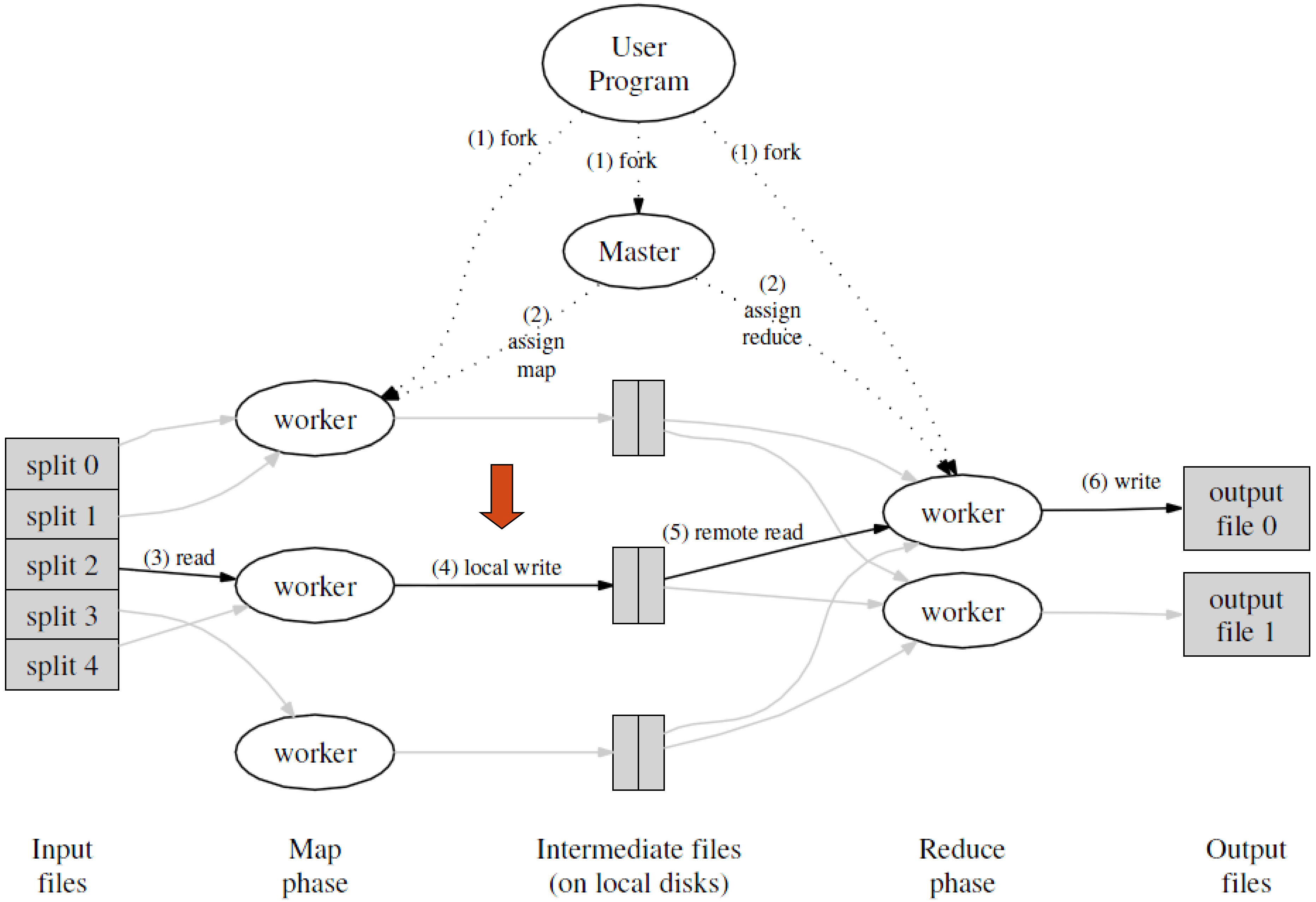
MAPREDUCE
EXECUTION – STEP 3

- A worker who is assigned a Map task:
  - Reads the contents of the corresponding input split
  - Parses key/value pairs out of the input data
  - Passes each pair to the user-defined Map function
  - Intermediate key/value pairs produced by the Map function are buffered in memory


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

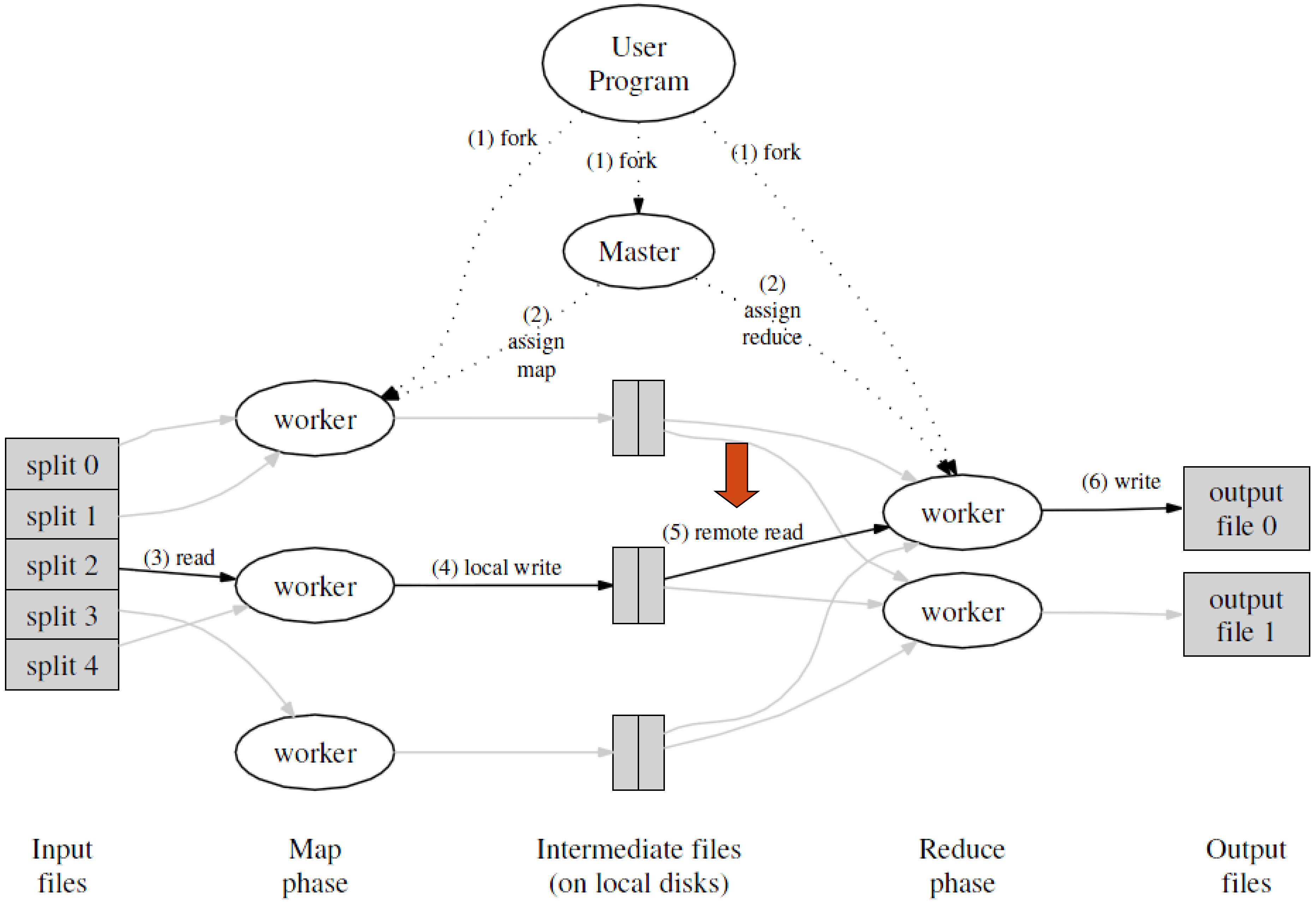
MAPREDUCE
EXECUTION – STEP 4

- Periodically, the buffered pairs are written to local disk
  - Partitioned into *R* regions by the partitioning function
- Locations of the buffered pairs on the local disk are passed back to the master
  - It is responsible for forwarding the locations to the Reduce workers


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

MAPREDUCE
EXECUTION – STEP 5

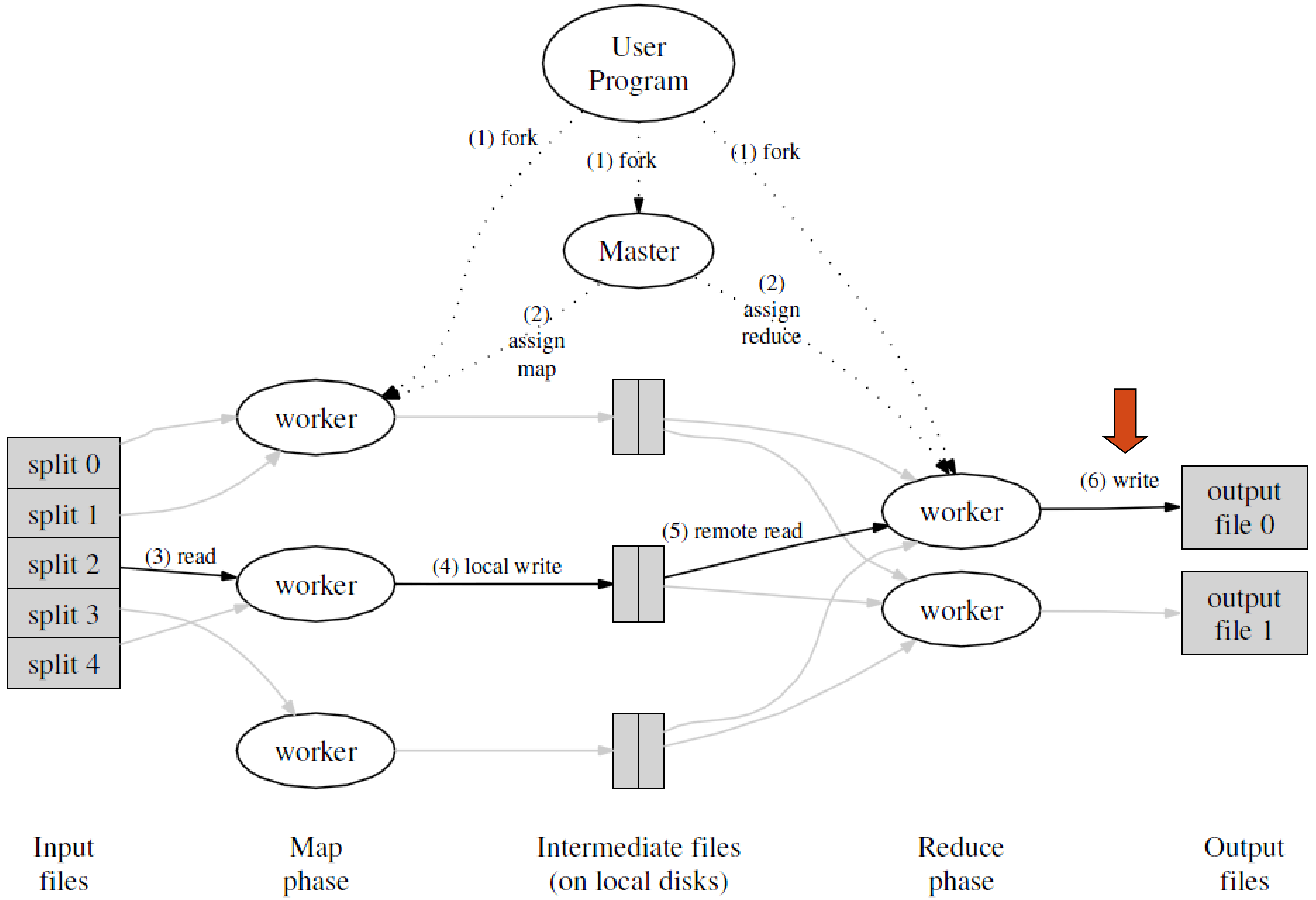
- Reduce worker is notified by the master about data locations
- It uses remote procedure calls to read the buffered data from local disks of the Map workers
- When it has read all intermediate data, it sorts it by the intermediate keys
  - Typically many different keys map to the same Reduce task
  - If the amount of intermediate data is too large, an external sort is used


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

MAPREDUCE

EXECUTION – STEP 6

- A Reduce worker iterates over the sorted intermediate data
- For each intermediate key encountered:
  - It passes the key and the corresponding set of intermediate values to the user's Reduce function
  - The output is appended to a final output file for this Reduce partition


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
worker
worker
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

MAPREDUCE

FUNCTION COMBINE

- After a map phase, the mapper transmits over the network the entire intermediate data file to the reducer
- Sometimes this file is highly compressible
- User can specify function combine
  - Like a reduce function
  - It is run by the mapper before passing the job to the reducer
    - Over local data

MAPREDUCE

COUNTERS

- Can be associated with any action that a mapper or a reducer does
  - In addition to default counters
    - e.g., the number of input and output key/value pairs processed
- User can watch the counters in real time to see the progress of a job

MAPREDUCE

FAULT TOLERANCE

- A large number of machines process a large number of data → fault tolerance is necessary
- Worker failure
  - Master pings every worker periodically
  - If no response is received in a certain amount of time, master marks the worker as failed
  - <u>All</u> its tasks are reset back to their initial <u>idle</u> state → become eligible for scheduling on other workers

MAPREDUCE

FAULT TOLERANCE

- Master failure
  - Strategy A:
    - Master writes periodic checkpoints of the master data structures
    - If it dies, a new copy can be started from the last checkpointed state
  - Strategy B:
    - There is only a single master → its failure is unlikely
    - MapReduce computation is simply aborted if the master fails
    - Clients can check for this condition and retry the MapReduce operation if they desire

MAPREDUCE
STRAGGLERS

- Straggler = a machine that takes an unusually long time to complete one of the map/reduce tasks in the computation
  - Example: a machine with a bad disk
- Solution:
  - When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks
  - A task is marked as completed whenever either the primary or the backup execution completes

MAPREDUCE
TASK GRANULARITY

- *M* pieces of Map phase and *R* pieces of Reduce phase
  - Ideally both much larger than the number of worker machines
  - How to set them?
- Master makes *O(M + R)* scheduling decisions
- Master keeps *O(M * R)* status information in memory
  - For each Map/Reduce task: state (idle/in-progress/completed)
  - For each non-idle task: identity of worker machine
  - For each completed Map task: locations and sizes of the *R* intermediate file regions
- *R* is often constrained by users
  - The output of each Reduce task ends up in a separate output file
- Practical recommendation (Google):
  - Choose *M* so that each individual task is roughly 16 – 64 MB of input data
  - Make *R* a small multiple of the number of worker machines we expect to use

MAPREDUCE CRITICISM

DAVID DEWITT AND MICHAEL STONEBRAKER -- 2008

1. MapReduce is a step backwards in database access based on
   - Schema describing data structure
   - Separating schema from the application
   - Advanced query languages
2. MapReduce is a poor implementation
   - Instead of indices it uses brute force
3. MapReduce is not novel (ideas more than 20 years old and overcome)
4. MapReduce is missing features common in DBMSs
   - Indices, transactions, integrity constraints, views, …
5. MapReduce is incompatible with applications implemented over DBMSs
   - Data mining, business intelligence, …

NOTE: WHO IS MICHAEL STONEBRAKER?

- *1943
- Computer scientist – database researcher
- Academic prototypes form the core of various databases
  - Ingres, Postgres, C-store (Vertica), H-store (VoltDB), SciDB, …
- 2015 – Turing award (ACM)
  - "Nobel Prize of computing"
  - For concepts and practices underlying modern database systems
  - 2016 – Tim Berners Lee
    - For inventing the WWW

HADOOP MAPREDUCE

- MapReduce requires:
  - Distributed file system
    - HDFS = Hadoop distributed file system
  - Engine that can distribute, coordinate, monitor and gather the results
- Hadoop: HDFS + JobTracker + TaskTracker
  - JobTracker (master) = scheduler
  - TaskTracker (slave per node) – is assigned a Map or Reduce (or other operations)
    - Map or Reduce run on a node → so does the TaskTracker
    - Each task is run on its own JVM

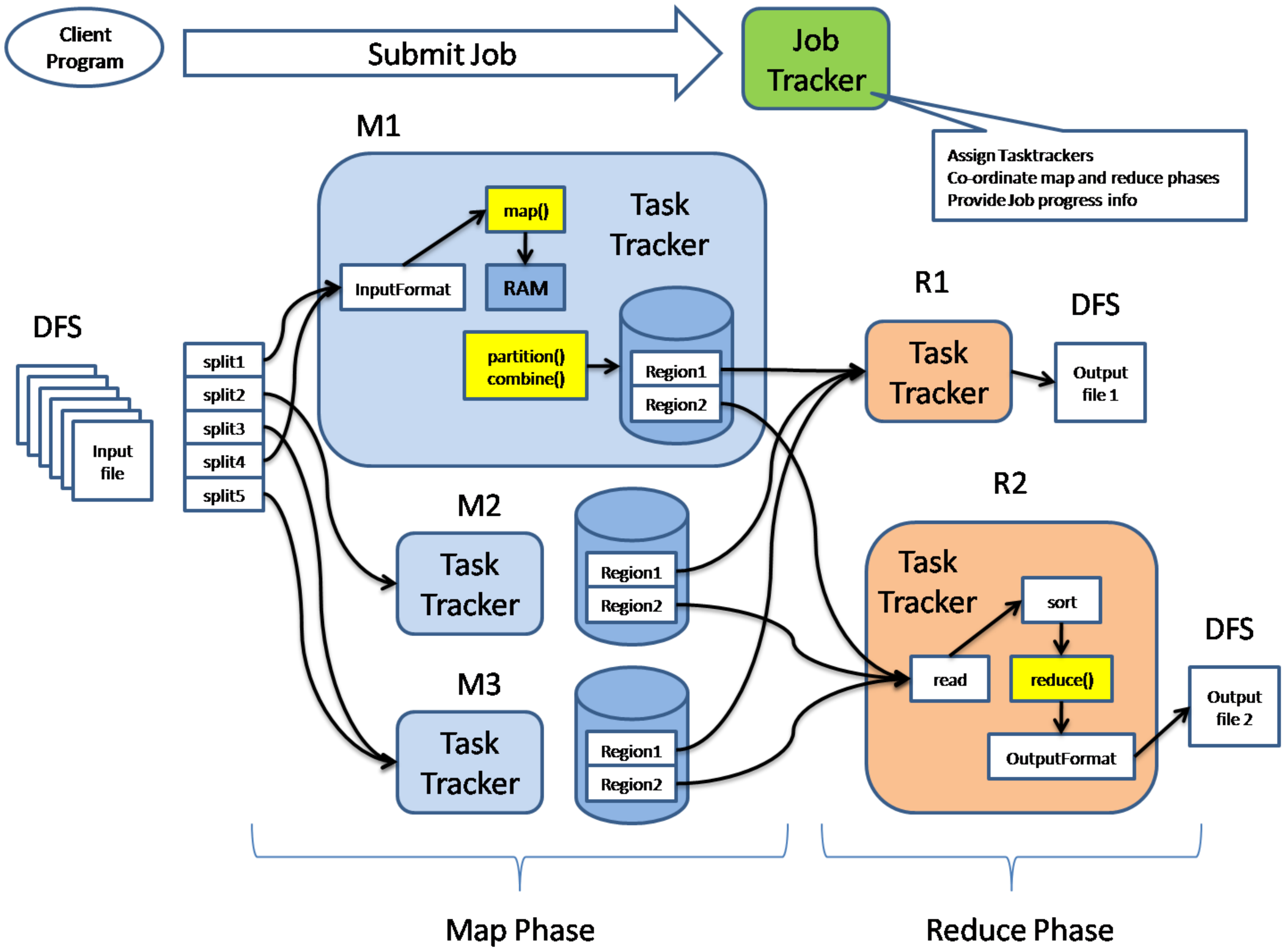
MAPREDUCE
JOBTRACKER (MASTER)

- Like a scheduler:
  1. A client application is sent to the JobTracker
  2. It “talks” to the NameNode (= HDFS master) and locates the TaskTracker (Hadoop client) near the data
  3. It moves the work to the chosen TaskTracker node

MAPREDUCE

TASKTRACKER (CLIENT)

- Accepts tasks from JobTracker
  - Map, Reduce, Combine, …
  - Input, output paths
- Has a number of slots for the tasks
  - Execution slots available on the machine (or machines on the same rack)
- Spawns a separate JVM for execution of a task
- Indicates the number of available slots through the hearbeat message to the JobTracker
  - A failed task is re-executed by the JobTracker


Client Program
Submit Job
Job Tracker
Assign Tasktrackers
Co-ordinate map and reduce phases
Provide Job progress info
M1
Task Tracker
map()
InputFormat
RAM
partition()
combine()
Region1
Region2
DFS
Input file
split1
split2
split3
split4
split5
M2
Task Tracker
Region1
Region2
M3
Task Tracker
Region1
Region2
R1
Task Tracker
DFS
Output file 1
R2
Task Tracker
sort
read
reduce()
OutputFormat
DFS
Output file 2
Map Phase
Reduce Phase

JOB LAUNCHING

JOB CONFIGURATION

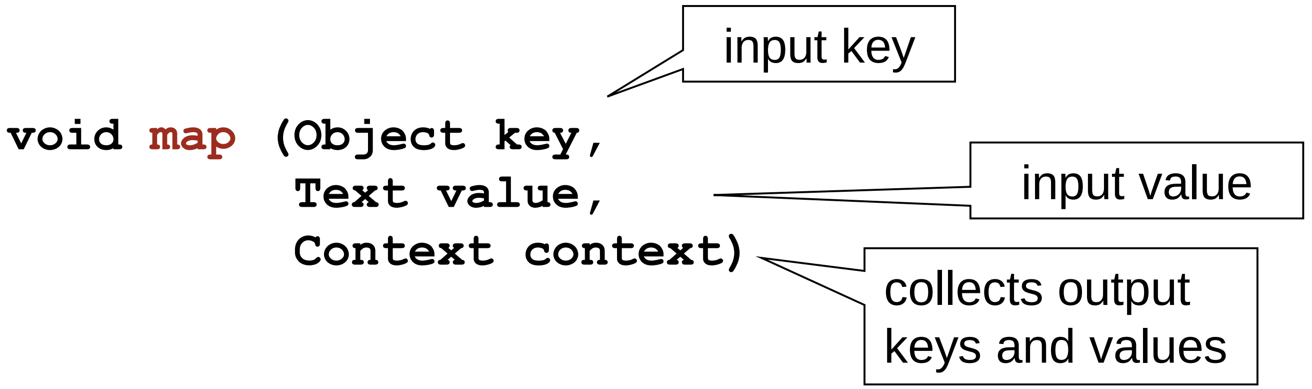
- For launching program:
  1. Create a Job to define a job
     - Using class Configuration
  2. Submit Job to the cluster and wait for completion
- Job involves:
  - Classes implementing Mapper and Reducer interfaces
    - `Job.`**`setMapperClass`**`()`
    - `Job.`**`setReducerClass`**`()`
  - Job outputs
    - `Job.`**`setOutputKeyClass`**`()`
    - `Job.`**`setOutputValueClass`**`()`
  - Other options:
    - `Job.`**`setNumReduceTasks`**`()`
    - …

JOB LAUNCHING
JOB

- `waitForCompletion()` – waits (blocks) until the job finishes
- `submit()` – does not block
- `monitorAndPrintJob()` – monitor a job and print status in real-time as progress is made and tasks fail

MAPPER

- The user provides an instance of Mapper
  - Implements interface `Mapper`
    - Overrides function `map`
  - Emits (k_2,v_2) using `context.write(k2, v2)`
- Exists in separate process from all other instances of Mapper
  - No data sharing



```
void map (Object key,
          Text value,
          Context context)
```

```
public static class TokenizerMapper
      extends Mapper<Object, Text, Text, IntWritable>{

   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map (Object key, Text value, Context context)
        throws IOException, InterruptedException {
     StringTokenizer itr
        = new StringTokenizer(value.toString());
     while (itr.hasMoreTokens()) {
       word.set(itr.nextToken());
       context.write(word, one);
     }
   }
 }
```

REDUCER

```
reduce(Text key,
       Iterable<IntWritable> values,
       Context context)
```

- Keys & values sent to one partition all go to the same reduce task
- Calls are sorted by key

```
public static class IntSumReducer
      extends Reducer<Text,IntWritable,Text,IntWritable> {
   private IntWritable result = new IntWritable();

   public void reduce (Text key,
                       Iterable<IntWritable> values,
                       Context context
                      )
        throws IOException, InterruptedException {
     int sum = 0;
     for (IntWritable val : values) {
       sum += val.get();
     }
     result.set(sum);
     context.write(key, result);
   }
 }
```

HDFS (HADOOP DISTRIBUTED FILE SYSTEM)
BASIC FEATURES

- Free and open source
- Crossplatform
  - Pure Java
  - Has bindings for non-Java programming languages
- Fault-tolerant
- Highly scalable
- Hierarchical file system
  - Directories and files
- Operations: Create, remove, move, rename, …

HDFS
FAULT TOLERANCE

- Idea: “failure is the norm rather than exception”
  - A HDFS instance may consist of thousands of machines
    - Each storing a part of the file system’s data
  - Each component has non-trivial probability of failure

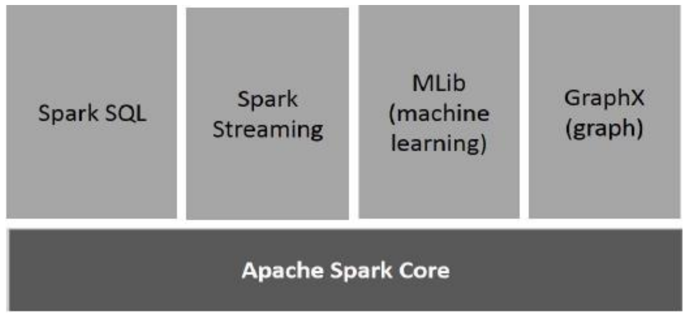
→ Assumption: “There is always some component that is non-functional.”

  - Detection of faults
  - Quick, automatic recovery

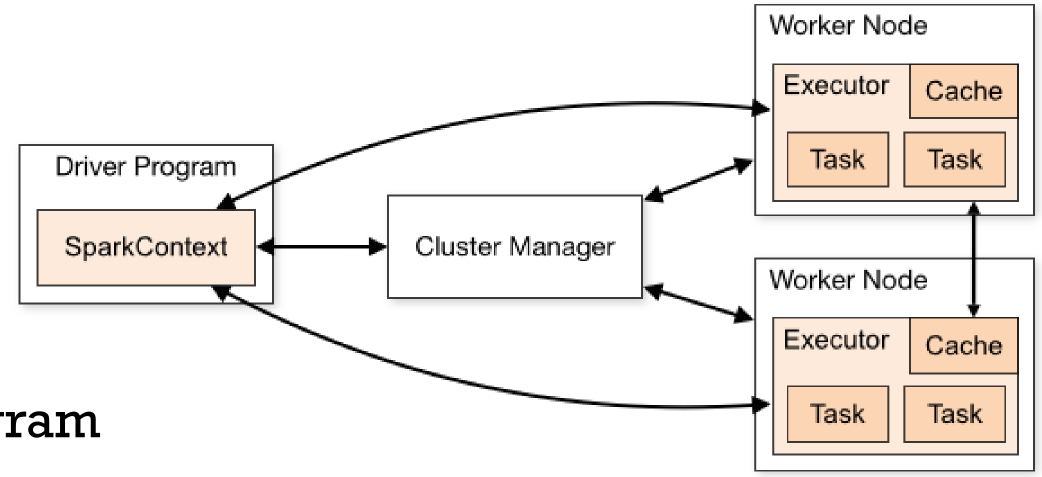
APACHE SPARK

APACHE SPARK

- Initial release : 2014
- Unified analytics engine for large-scale data processing
  - Runs on a cluster of nodes
- Contains:
  - High-level APIs in Java, Scala, Python and R
  - Optimized engine that supports general execution graphs (DAGs)
    - MapReduce has only 2 levels
  - Higher-level tools
    - Spark SQL (SQL and structured data processing)
    - MLlib (machine learning)
    - GraphX (graph processing)
    - Spark Streaming



SPARK APPLICATION



- Spark application = driver program
  - Runs the user's main function
  - Executes parallel operations on a cluster
    - Independent set of processes
    - Coordinated by SparkContext object in the driver program
- SparkContext can connect to several types of cluster managers
  - They allocate resources across applications
- When connected:
  1. Spark acquires executors on nodes in the cluster
     - Processes that run computations and store data for the application
  2. Sends the application code to the executors
     - Defined by JAR or Python files passed to SparkContext
  3. Sends tasks to the executors to run

SPARK APPLICATION

- Each application gets its own executor processes which run tasks in multiple threads
    - Pros: isolating of applications
        - Scheduling + executing
    - Cons: data cannot be shared across different Spark applications (instances of SparkContext) without writing it to an external storage system
- Driver program
    - Must listen for and accept incoming connections from its executors throughout its lifetime
    - Should be run close to the worker nodes
        - Preferably on the same local area network
    - Has a web UI
        - Displays information about running tasks, executors, and storage usage

INITIALIZING SPARK

1. Build a SparkConf object
   - Contains information about application
   - appName = application name to show on the cluster UI
   - master = Spark/Mesos/YARN cluster URL or string "local" to run in local mode
2. Create a JavaSparkContext object
   - Tells Spark how to access a cluster

```
SparkConf conf =
  new SparkConf().setAppName(appName).setMaster(master);
JavaSparkContext sc =
  new JavaSparkContext(conf);
```

RESILIENT DISTRIBUTED DATASET (RDD)

- Immutable collection of elements partitioned across the nodes of the cluster
  - Can be operated on in parallel
  - Can be persisted in memory
  - Automatically recover from node failures
- Ways to create RDDs:
  1. Parallelizing an existing collection in a driver program
  2. Referencing a dataset in an external storage system
     - e.g., HDFS, HBase, …
     - In general: any offering a Hadoop InputFormat

RESILIENT DISTRIBUTED DATASET (RDD)
PARALLELIZED COLLECTIONS

- Parallelized collections are created by calling SparkContext's parallelize method
  - Elements of the collection are copied to form a distributed dataset
  - The distributed dataset (distData) can be operated on in parallel
    - See later

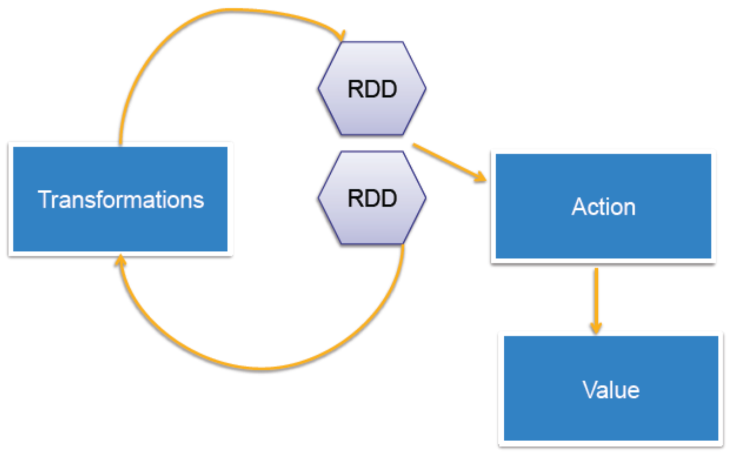
```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

RESILIENT DISTRIBUTED DATASET (RDD)
EXTERNAL DATASETS

- Spark can create distributed datasets from any storage source supported by Hadoop
  - Local file system, HDFS, Cassandra, HBase, …
- Supports text files, SequenceFiles, and any other Hadoop InputFormat
- Example:
  - Text file RDDs can be created using SparkContext's textFile method
    - Takes an URI for the file (local, HDFS, …)
    - Reads it as a collection of lines
    - Optional argument: number of partitions of the file
      - Default: one partition for each block of the file (128MB by default in HDFS)
  - Once created, distFile can be acted on by dataset operations

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```

RDD OPERATIONS



1. Transformations = create (lazily)

   a new dataset from an existing one
   - e.g., map = passes each dataset element through a function and returns a new RDD representing the results
2. Actions = return a value to the driver program after running a computation on the dataset
   - e.g., reduce = aggregates all the elements of the RDD using some function and returns the final result to the driver program

- By default: each transformed RDD may be recomputed each time we run an action on it
  - We may also persist an RDD in memory using the persist (or cache) method
    - Much faster access the next time we query it
  - There is also support for persisting RDDs on disk or replicated across multiple nodes

TRANSFORMATIONS

- **map**(func) Returns a new distributed dataset formed by passing each element of the source through a function func.
- **union**(otherDataset) Returns a new dataset that contains the union of the elements in the source dataset and the argument.
  - **intersection, distinct**
- **filter**(func) Returns a new dataset formed by selecting those elements of the source on which func returns true.
- **reduceByKey**(func, [numPartitions]) When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. The number of reduce tasks is configurable through an optional second argument.
- **sortByKey**([ascending], [numPartitions]) When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument.
- …

ACTIONS

- **reduce**(func) Aggregates the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
- **count**() Returns the number of elements in the dataset.
- **first**() Returns the first element of the dataset.
- **take**(n) Returns an array with the first n elements of the dataset.
- **takeOrdered**(n, [ordering]) Returns the first n elements of the RDD using either their natural order or a custom comparator.
- …

SIMPLE SPARK EXAMPLE

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

1. Defines a base RDD from an external file
   - Not loaded in memory or otherwise acted on, due to laziness
   - lines is merely a pointer to the file
2. Defines lineLengths as the result of a map transformation
   - Not immediately computed, due to laziness
3. Runs reduce = action
   - Spark breaks the computation into tasks to run on separate machines
   - Each machine runs both its part of the map and a local reduction, returning its answer to the driver program

SPARK SQL

Spark SQL

- Spark module for structured data processing
- More information about the structure of both the data and the computation being performed
  - Internally, Spark SQL uses this extra information to perform extra optimizations
- Interact with Spark SQL: SQL, Dataset API, …

RDD VS. DATAFRAME VS. DATASET

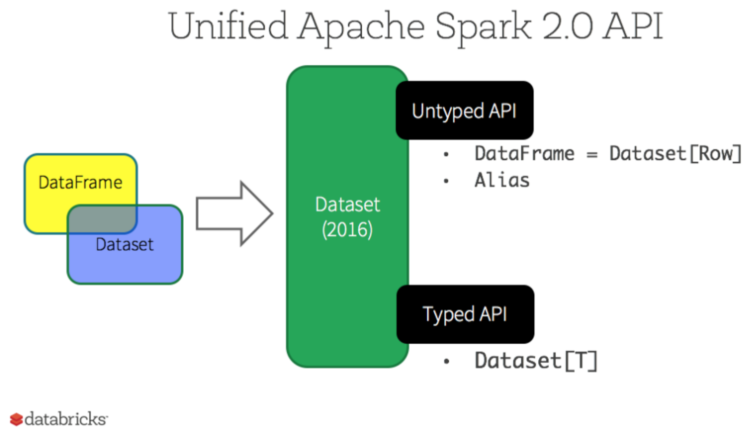
- RDD = primary API in Spark since its inception
  - Since Spark 1.0
  - Internally each final computation is still done on RDDs
- DataFrame = data organized into <u>named columns</u>
  - Since Spark 1.3
  - Distributed collection of data, which is organized into named columns
    - Designed to make data processing easier
      - Higher level of abstraction
      - Similar to a table in a relational database or a data frame in R/Python
  - Can be constructed from: structured data files, tables in Hive, external databases, existing RDDs , …
  - API: Scala, Java, Python, R

RDD VS. DATAFRAME VS. DATASET

- Dataset = a distributed collection of data
  - Since Spark 1.6
  - Provides the benefits of
    - RDDs - strong typing, ability to use powerful lambda functions
    - Spark SQL - optimized execution engine
      - i.e. DataFrame processing
  - Can be constructed from: JVM objects
  - API: Scala, Java

RDD VS. DATAFRAME VS. DATASET

- Since Spark 2.0: unification of DataFrame and Dataset
- Two distinct APIs:
  - Untyped API
    - Conceptually: DataFrame ~ collection of generic objects `Dataset<Row>`, where a `Row` is a generic untyped JVM object
  - Strongly-typed API
    - Conceptually: Dataset ~ collection `Dataset<T>` of strongly-typed JVM objects, dictated by a case class `T`
      - Defined in Scala or a class in Java



BASIC EXAMPLES

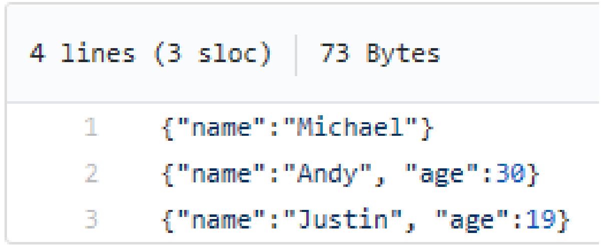
```
SparkSession spark = SparkSession.builder().
  appName("Java Spark SQL basic example").
  config("spark.some.config.option", "some-value").getOrCreate();

Dataset<Row> df =
  spark.read().json("examples/src/main/resources/people.json");
```

DataFrame - untyped

```
df.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

df.printSchema();
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)
```



4 lines (3 sloc) | 73 Bytes

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

```
// Select only the "name" column
df.select("name").show();
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+

// Select everybody, but increment the age by 1
df.select(col("name"), col("age").plus(1)).show();
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter(col("age").gt(21)).show();
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+
// Count people by age
df.groupBy("age").count().show();
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+


// Register the DataFrame as an SQL temporary view
df.createOrReplaceTempView("people");

Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
sqlDF.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

- Temporary views are session-scoped
  - Disappear if the session that creates it terminates
- Global temporary view = a temporary view shared among all sessions
  - Keeps alive until the Spark application terminates
  - Tied to a system preserved database `global_temp`
    - `df.createGlobalTempView("people");`
    - Must use the qualified name to refer it
      - e.g. `SELECT * FROM global_temp.people`

REFERENCES

- Jeffrey Dean and Sanjay Ghemawat: MapReduce: **Simplified Data Processing on Large Clusters**, Google, Inc.
  - http://labs.google.com/papers/mapreduce.html
- Google Code: **Introduction to Parallel Programming and MapReduce**
  - code.google.com/edu/parallel/mapreduce-tutorial.html
- **Apache Hadoop:** http://hadoop.apache.org/
- **Hadoop Map/Reduce Tutorial**
  - http://hadoop.apache.org/docs/r0.20.2/mapred_tutorial.html
- **Open Source MapReduce**
  - http://lucene.apache.org/hadoop/
- Hadoop: **The Definitive Guide**, by Tom White, 2nd edition, Oreilly's, 2010
- David DeWitt and Michael Stonebraker: **Relational Database Experts Jump The MapReduce Shark**

REFERENCES

- Spark Overview https://spark.apache.org/docs/latest/index.html
- Apache Spark Examples https://spark.apache.org/examples.html
- Mastering Apache Spark 2.3.2 https://jaceklaskowski.gitbooks.io/mastering-apache-spark/
- A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html