MapReduce and Spark: Overview

2015

Professor Sasu Tarkoma

Overview

Computing Environment

- Scaling up
  - More powerful servers
- Scaling out
  - More servers
- Clusters provide computing resources
  - Space requirements, power, cooling
  - Most power converted into heat
- Datacenters
  - Massive computing units
  - **Warehouse-sized computer** with hundreds or thousands of racks
- Networks of datacenters

Cluster Computing Environment

- Big Data compute and storage nodes are stored on racks based on common off the shelf components
- Typically many racks in a cluster or datacenter
- The compute nodes are connected by a high speed network (typically 10 Gbit/s Ethernet)
  - Different datacenter network topologies
- Intra-rack and inter-rack communication have differing latencies
- Nodes can fail
  - Redundancy for stored file (replication)
  - Computation is task based
- **Software ensures fault-tolerance and availability**

Typical Hardware

CSC Pouta Cluster running on the Taito supercluster in Kajaani

The nodes are HP ProLiant SL230s servers with two Intel Xeon 2.6 GHz E5-2670 CPUs

16 cores per server

Most with 64 GB of RAM per server

Taito extension in 2014: 17 000 cores

The nodes are connected using a fast FDR InfiniBand fabric

Cloud Computing

Definition by NIST:

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

IaaS, PaaS, SaaS, XaaS

Big Data Frameworks are typically run in the cloud

Big Data Environment

Typically common-of-the-shelf servers

- Compute nodes, storage nodes, …

Virtualized resources running on a cloud platform

Heterogeneous hardware, choice of OS

Contrasts traditional High Performance Computing (HPC)

History of Cluster Frameworks

2003: Google GFS

2004: Google Map-Reduce

2005: Hadoop development starts

2008: Apache Hadoop (in production)

2008: Yahoo! Pig language

2009: Facebook Hive for data warehouses

2010: Cloudera Flume (message interceptor/filtering model)

2010: Cloudera S4 (continuous stream processing)

2011: LinkedIn Kafka (topic-based commit log service)

2011: Storm (Nathan Marz)

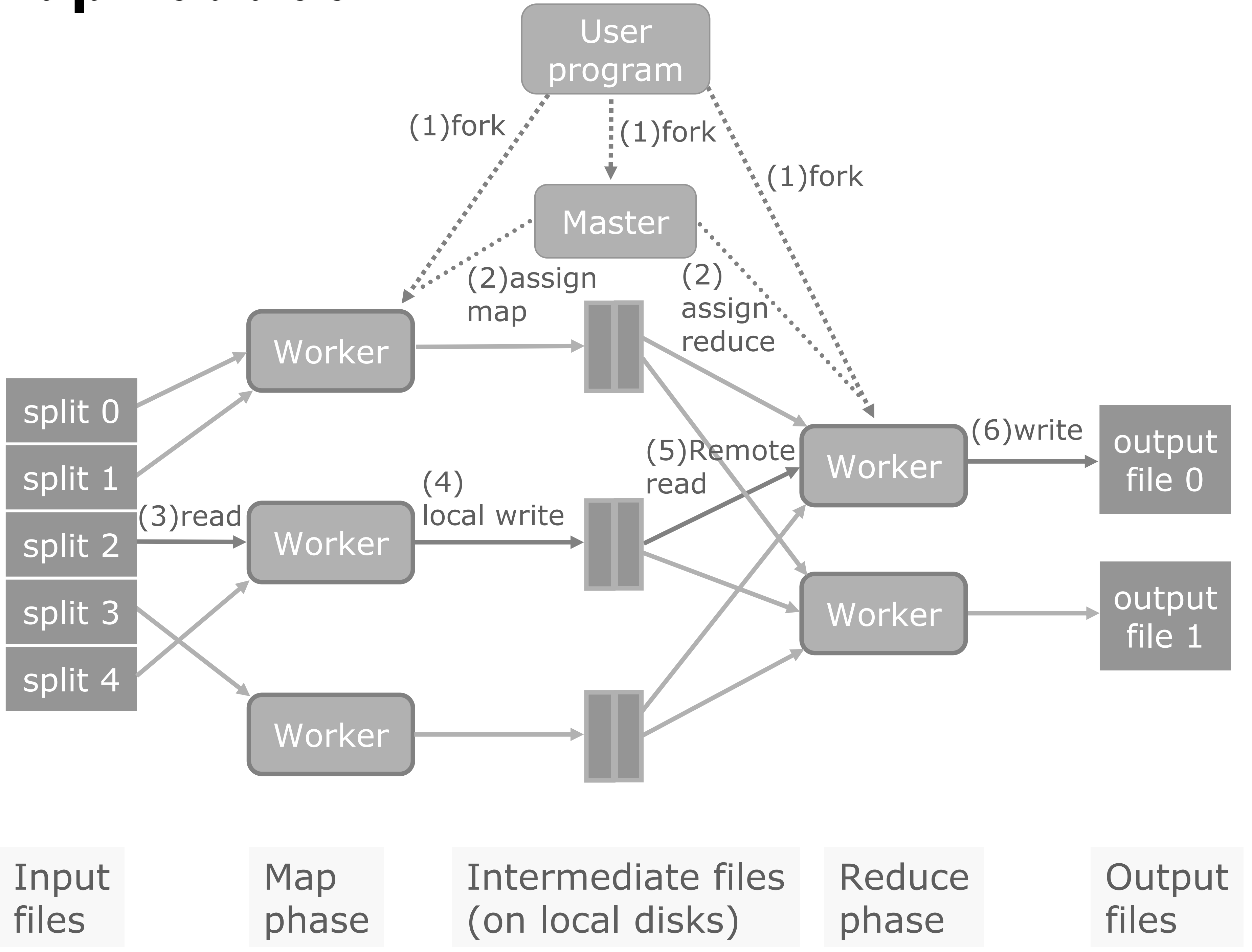
2011: Apache Mesos cluster management framework

2012: Lambda Architecture (Nathan Marz)

2012: Spark for iterative cluster programming

2013: Shark for SQL data warehouses

MapReduce


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

Major trends

Apache Hadoop

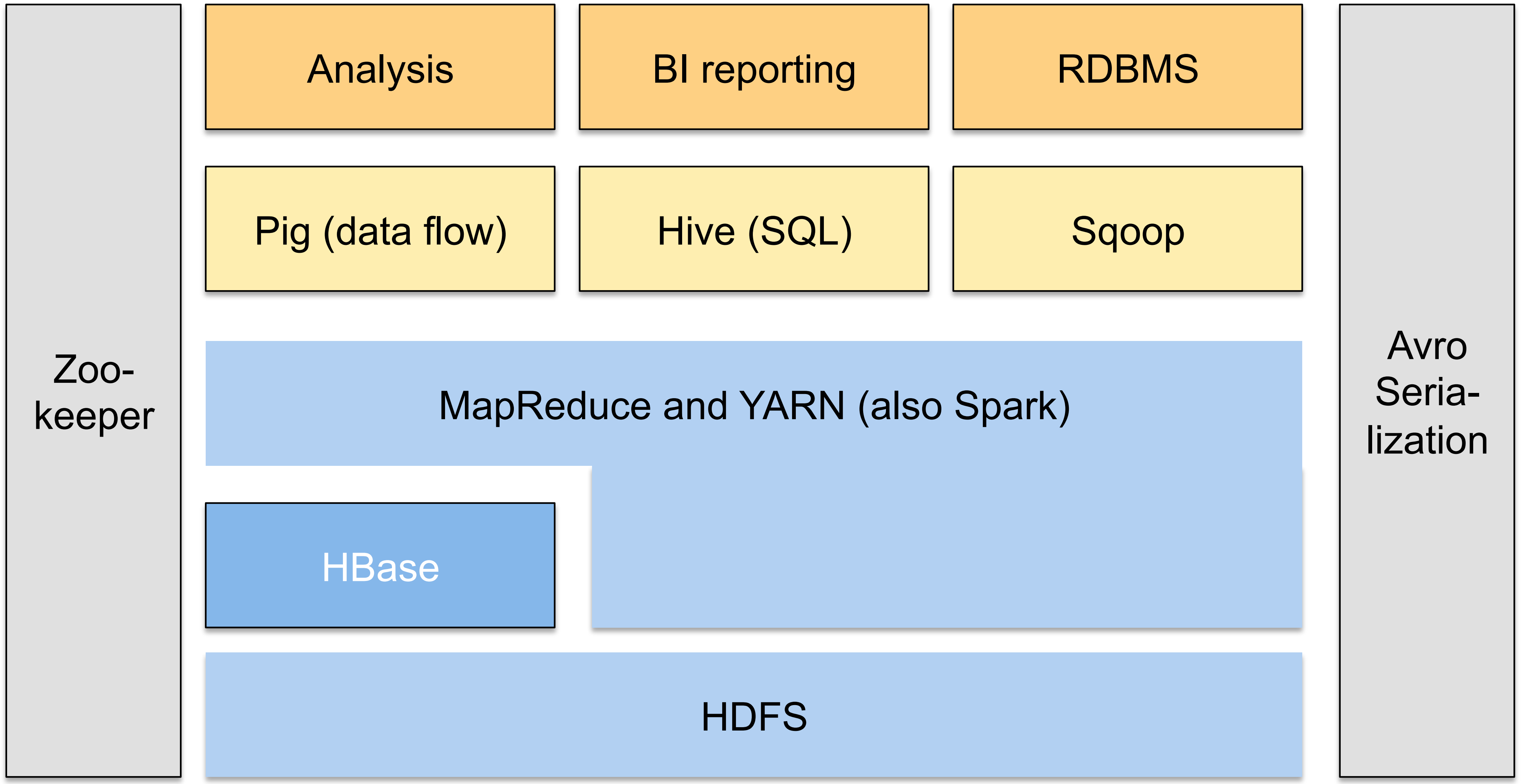
Hive, R, and others

Berkeley Data Analytics Stack (BDAS)

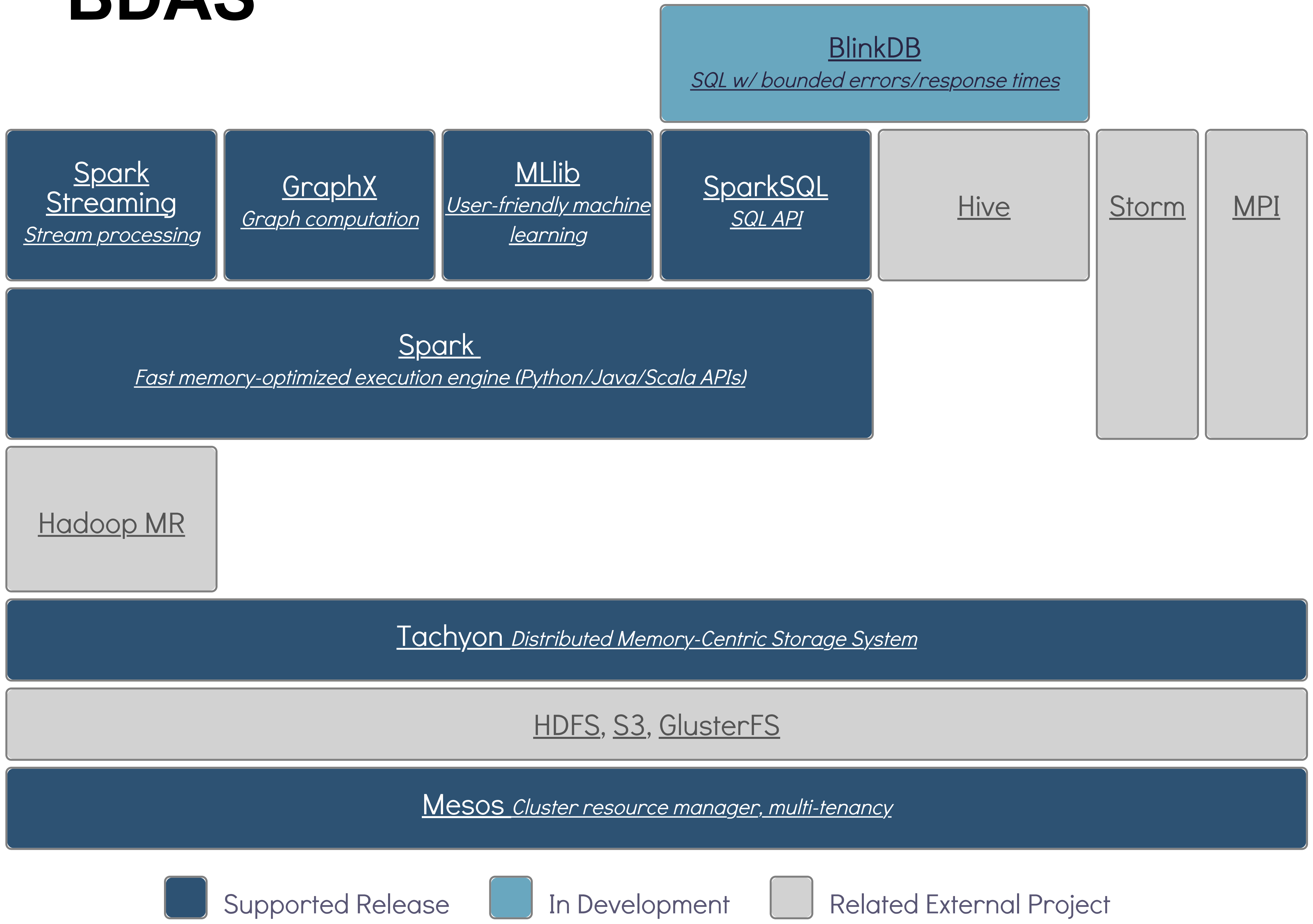
Mesos, Spark, Mlib, GraphX, Shark, …

Apache Spark is part of Apache Hadoop

Apache Hadoop Ecosystem



BDAS



BlinkDB
SQL w/ bounded errors/response times
Spark Streaming
Stream processing
GraphX
Graph computation
MLlib
User-friendly machine learning
SparkSQL
SQL API
Hive
Storm
MPI
Spark
Fast memory-optimized execution engine (Python/Java/Scala APIs)
Hadoop MR
Tachyon Distributed Memory-Centric Storage System
HDFS, S3, GlusterFS
Mesos Cluster resource manager, multi-tenancy
Supported Release
In Development
Related External Project

https://amplab.cs.berkeley.edu/software/

Key idea in Spark

Resilient distributed datasets (RDDs)

- Immutable collections of objects across a cluster
- Built with parallel transformations (map, filter, …)
- Automatically rebuilt when failure is detected
- Allow persistence to be controlled (in-memory operation)

Transformations on RDDs

- Lazy operations to build RDDs from other RDDs
- Always creates a new RDD

Actions on RDDs

- Count, collect, save

Overview of State of the Art

- Data storage
- Data storage for real-time
- Data analysis
- Real-time data analysis
- Statistics and machine learning

State of the Art: Data Storage

GFS (Google File System) and HDFS (Hadoop Distributed File System)

- Data replicated across nodes
- HDFS: rack-aware placement (replicas in different racks)
- Take data locality into account when assigning tasks
- Do not support job locality (distance between map and reduce workers)

Hbase

- Modeled after Google’s BigTable for sparse data
- Non-relational distributed column-oriented database
- Rows stored in sorted order

Sqoop

- Tool for transferring data between HDFS/Hbase and structural datastores
- Connectors for MySQL, Oracle, … and Java API

Example: HDFS Architecture

HDFS has a master/slave architecture

NameNode is the master server for metadata

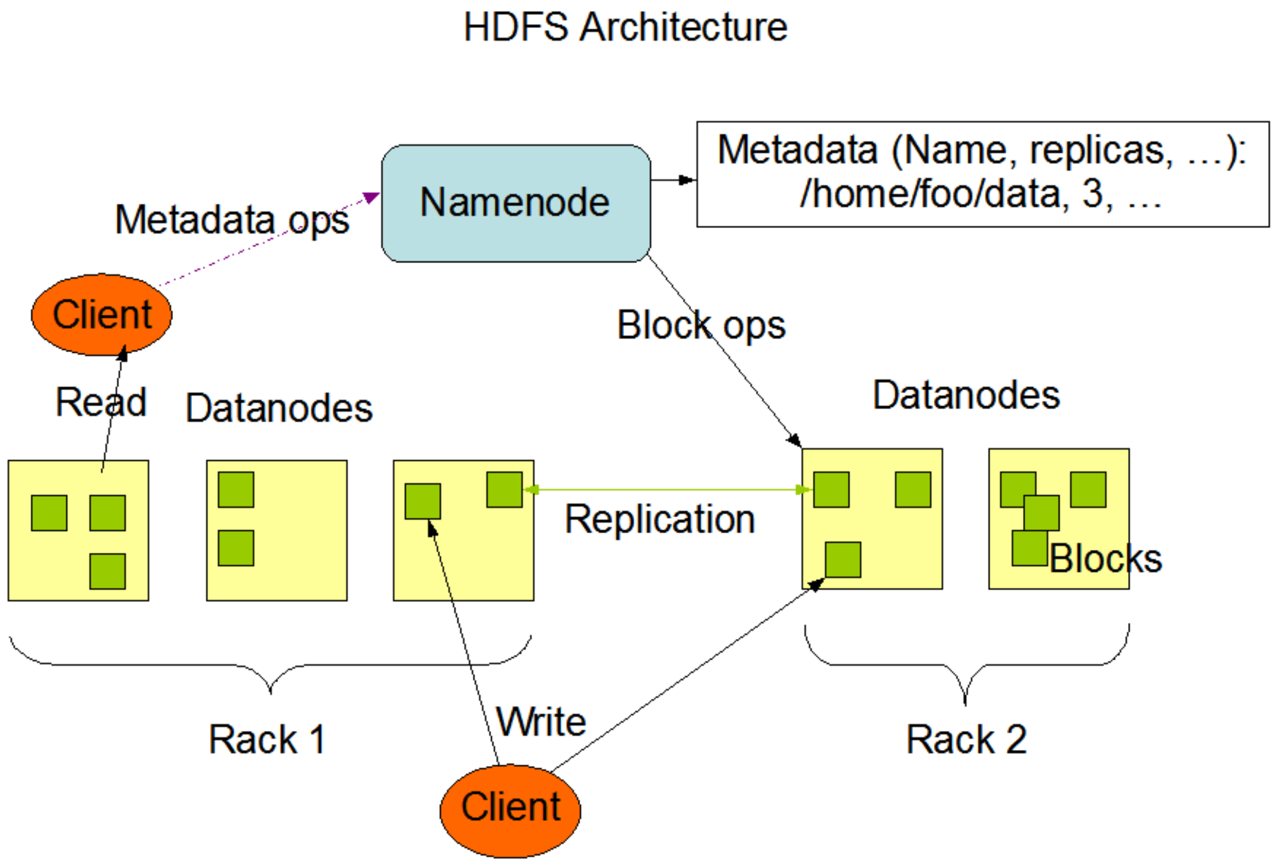
DataNodes manage storage

A file is stored as a sequence of blocks

The blocks are replicated fo fault-tolerance

Common replication scheme: factor of 3, one replica local, two in a remote rack

Rack-aware replica placement



Namenode provides information for retrieving blocks
Nearest replica is used to retrieve a block

http://hadoop.apache.org/docs/r1.2.1/images/hdfsarchitecture.gif

State of the Art: Data Storage for Real-time

Kafka

- Distributed, partitioned, replicated commit log service
- Keeps messages in categories
- Topic based system
- Coordination through Zookeeper (through distributed consensus)

Kestrel

- Distributed message queue (server has a set of queues)
- A server maintains queues (FIFO)
- Does not support ordered consumption
- Simpler than Kafka

State of the Art: Data Analysis I/II

MapReduce

- Map and reduce tasks for processing large datasets in parallel

Hive

- A data warehouse system for Hadoop
- Data summarization, ad-hoc queries, analysis for large sets
- SQL-like language called HiveQL

Pig

- Data analysis platform
- High-level language for defining data analysis programs, Pig Latin, procedural language

Cascading

- Data processing API and query planner for workflows
- Supports complex Hadoop Map-Reduce workflows

Apache Drill

- SQL query engine for Hadoop and noSQL

State of the Art: Data Analysis II

Spark

- Cluster computing for data analytics
- In-memory storage for iterative processing

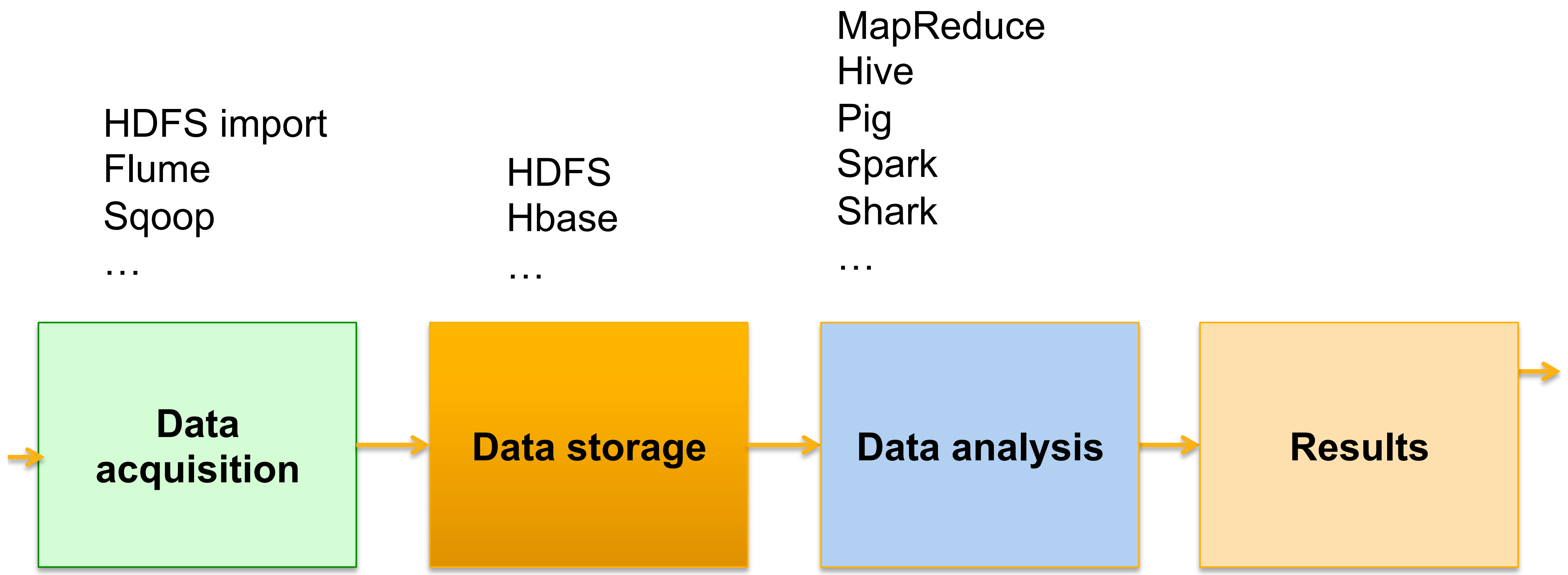
Shark

- Data warehouse system (SQL) for Spark
- Up to 100x faster than Hive

Spark/Shark is a distinct ecosystem from Hadoop

- Faster than Hadoop
- Support for Scala, Java, Python
- Can be problematic if reducer data does not fit into memory

Summary of batch systems



State of the Art: Real-time Data Analysis I/II

Flume

- Interceptor model that modifies/drops messages based on filters
- Chaining of interceptors
- Combine with Kafka

Storm

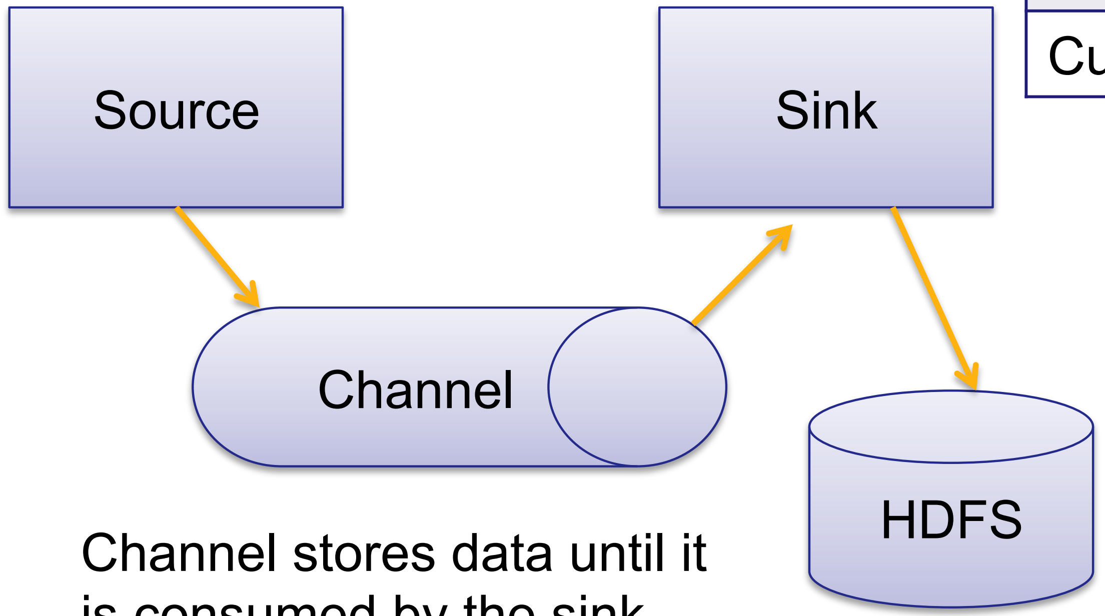
- Distributed realtime computation framework
- “Hadoop for realtime”
- Based on processing graph, links between nodes are streams

Trident

- Abstraction on top of Storm
- Operations: joins, filters, projections, aggregations, ..
- Exactly once-semantics (replay tuples for fault tolerance, stores additional state information)
- https://storm.apache.org/documentation/Trident-state

Flume example

Source	Channel	Sink
Avro	Memory	HDFS
Thrift	JDBC	Logger
Exec	File	Avro
HTTP		Null
JMS		Thrift
Syslog TCP/IP		File roll
		Hbase
Custom		Custom



Channel stores data until it is consumed by the sink.

Storm

Developed around 2008-2009 at BackType, open sourced in 2011

Spout: is a flow of tuples

Bolt: accepts tuples and operates on those

Topologies: spouts → bolts → spouts

Example:

Tweet spout → parse Tweet bolt → count hashtags Bolt

Tweet spout → store in a file

State of the Art: Real-time Data Analysis II

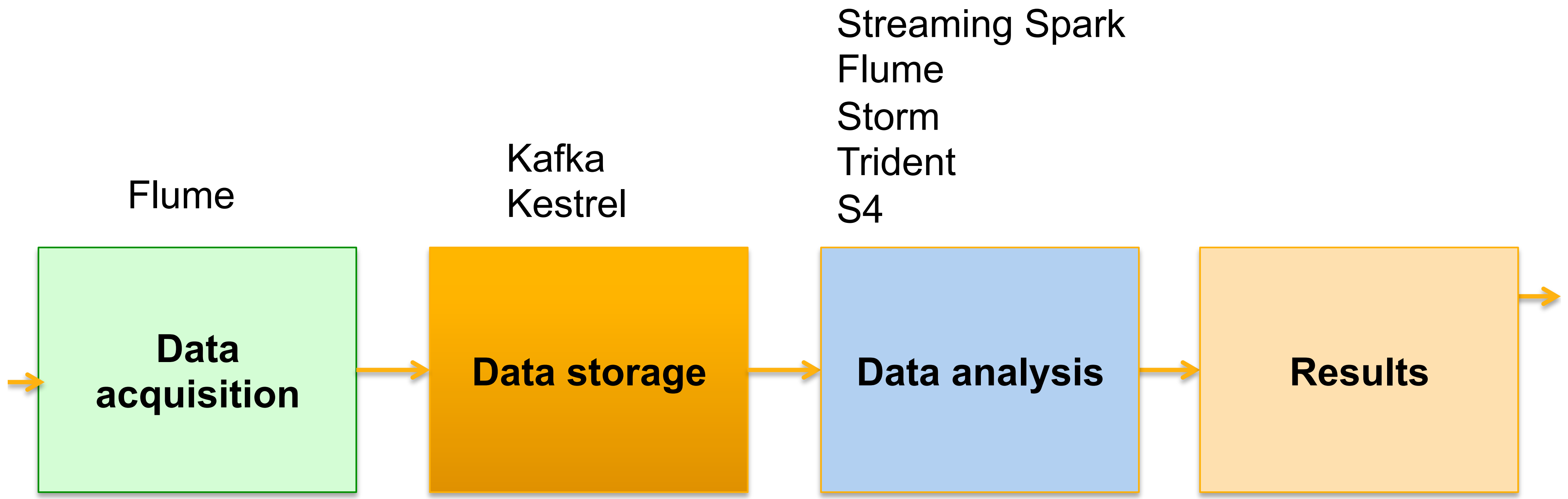
Simple Scalable Streaming System (S4)

- Platform for continuous processing of unbounded streams
- Based on processing elements (PE) that act on events (key, attributes)

Spark streaming

- Spark for real-time streams
- Computation with a series of short batch jobs (windows)
- State is kept in memory
- API similar to Spark

Summary of real-time processing



State of the art: Hybrid models

Lambda architecture combined batch and stream processing

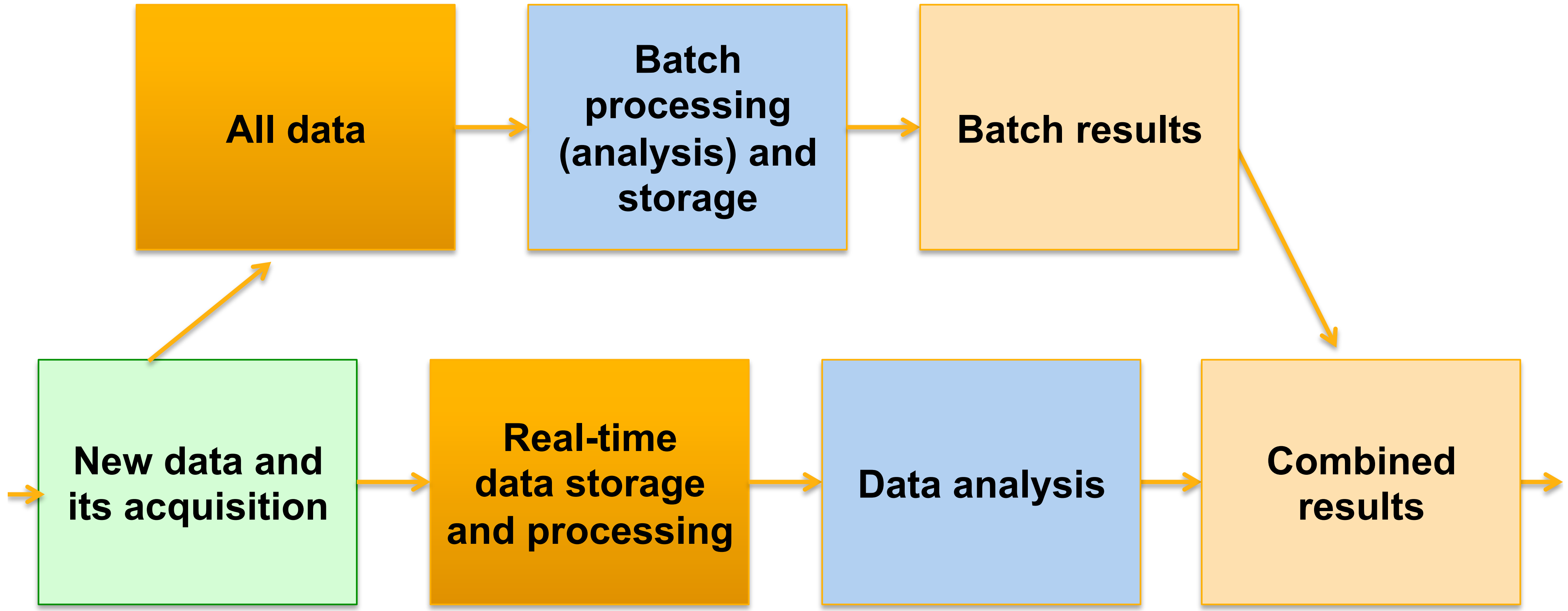
Supports volume (batch) + velocity (streaming)

Hybrid models

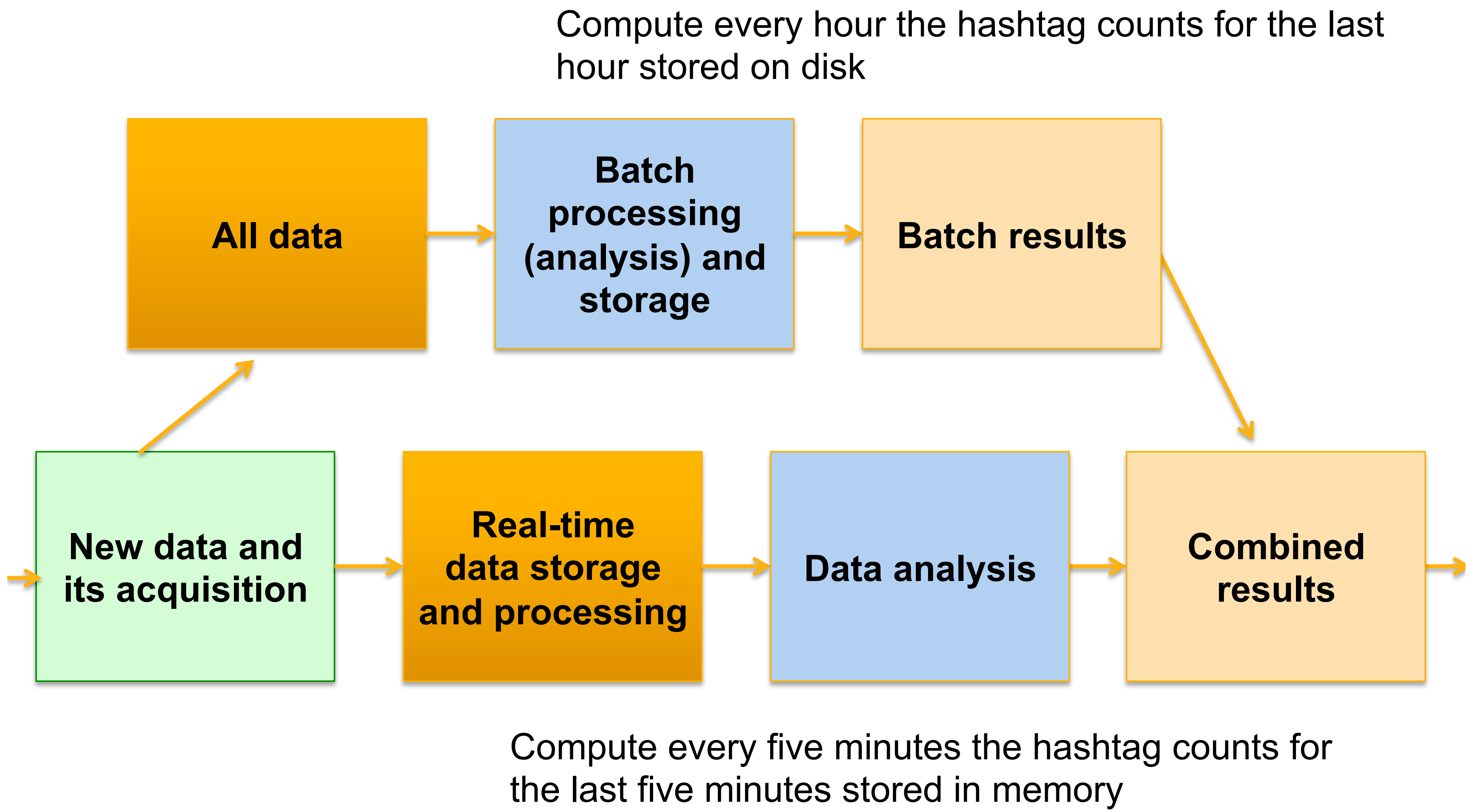
- **SummingBird** (Hadoop + Storm)
  - MapReduce like process with Scala syntax

- **Lambdoop** (abstraction over Hadoop, HBase, Sqoop, Flume, Kafka, Storm, Trident)
  - Common patterns provided by platform
  - No MapReduce like process

Lambda Architecture


All data
Batch processing (analysis) and storage
Batch results
New data and its acquisition
Real-time data storage and processing
Data analysis
Combined results

Lambda Architecture: Twitter hashtags



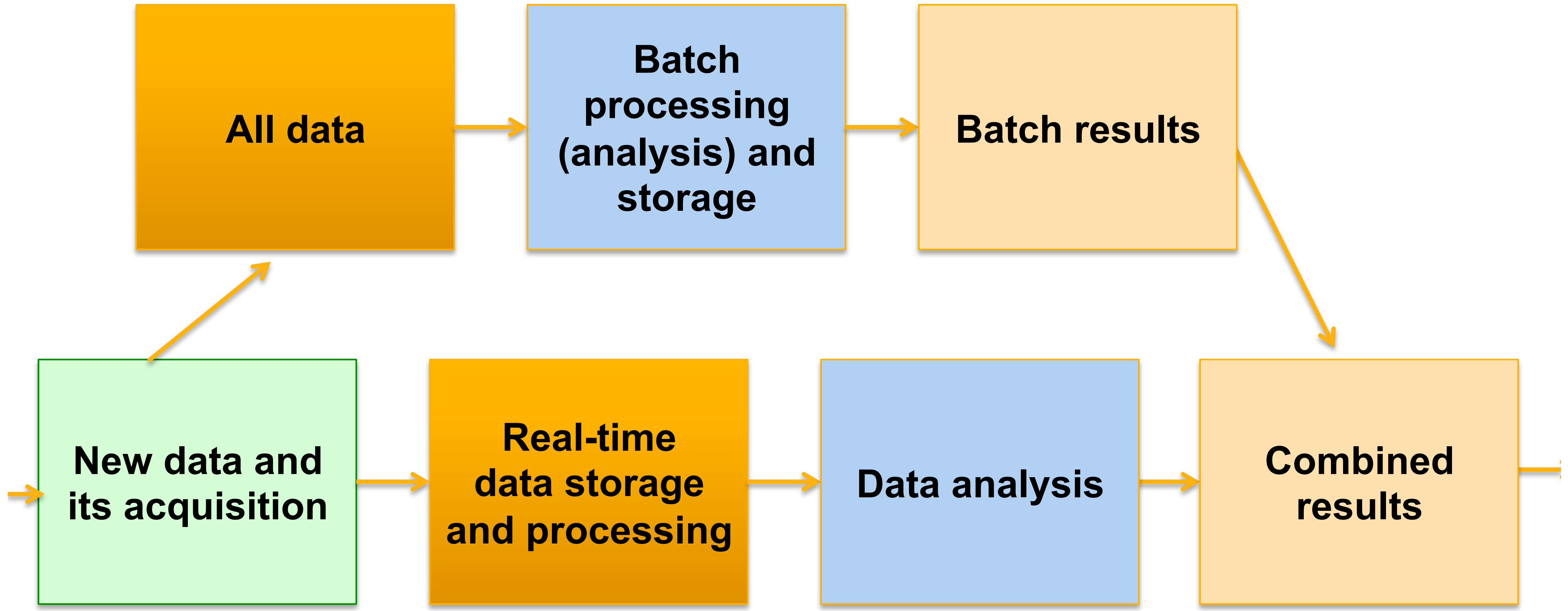
Compute every hour the hashtag counts for the last hour stored on disk
All data
Batch processing (analysis) and storage
Batch results
New data and its acquisition
Real-time data storage and processing
Data analysis
Combined results
Compute every five minutes the hashtag counts for the last five minutes stored in memory

Inspiration: http://www.slideshare.net/Dataiku/dataiku-devoxx-lambda-architecture-choose-your-tools

Lambda Architecture: Recommendations

User item views
Compute/maintain item-item similarity matrix on disk
Fallback when online part is offline

Users and items



All data
Batch processing (analysis) and storage
Batch results
New data and its acquisition
Real-time data storage and processing
Data analysis
Combined results

User item views
Update the online version

Inspiration: http://www.slideshare.net/Dataiku/dataiku-devoxx-lambda-architecture-choose-your-tools

Challenges for the platform

- Exactly-once semantics
  - Requires costly synchronization
- High velocity: how to go to thousands of messages per second
- Changes to structures and schemas
  - Data versioning in a production system

State of the Art: Statistics and Machine Learning

R for Hadoop

- Distributed R for the cluster environment

R for Spark

Mahout

- Currently Hadoop, next Spark

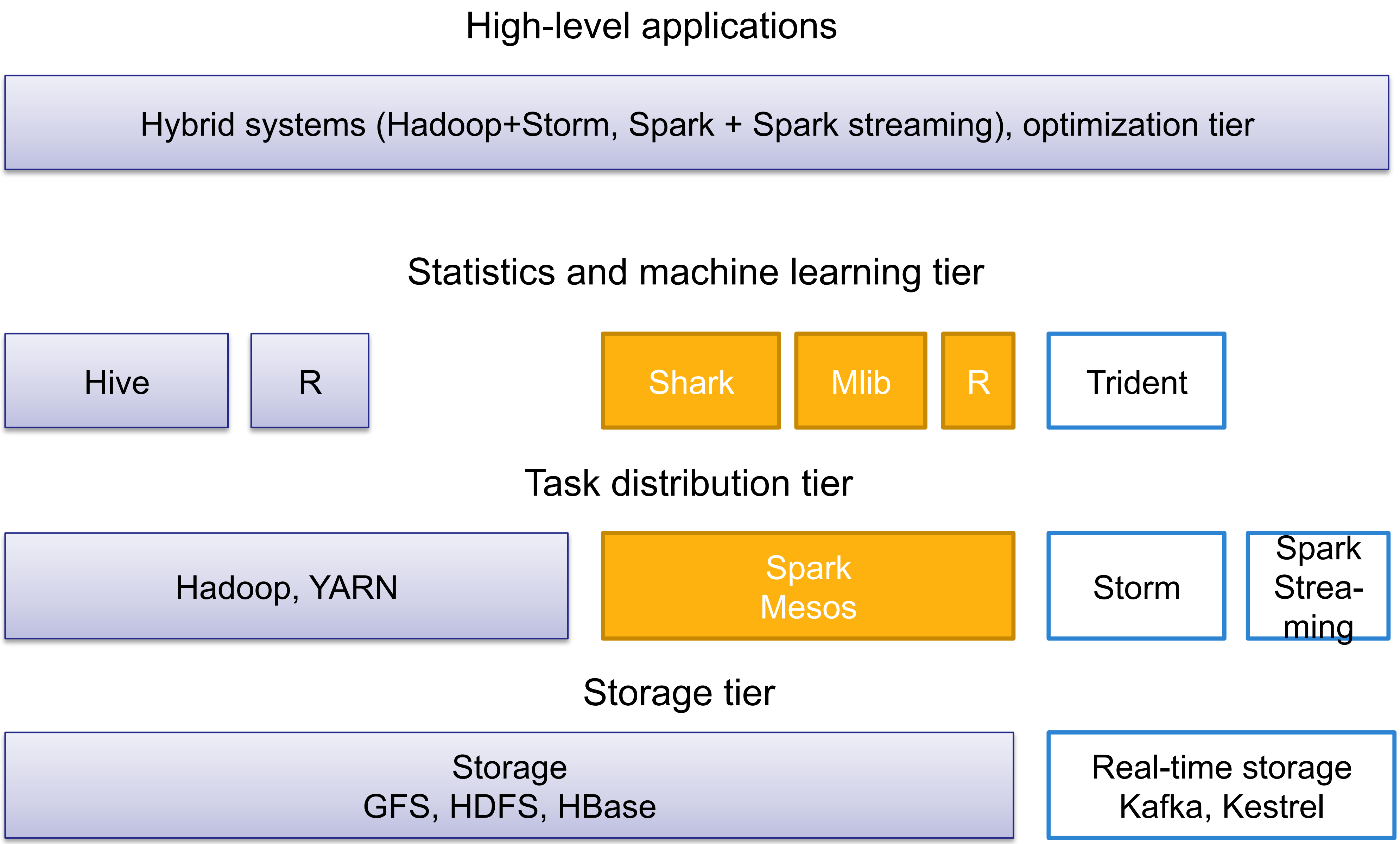
Weka

- State of the art machine learning library
- Does not focus on the distributed case
- Hadoop support, Spark wrappers

MLLib

- Machine learning for Spark

State of the Art Distributed Toolbox



MapReduce

2015

Professor Sasu Tarkoma

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

www.cs.helsinki.fi

MapReduce Model

Google MapReduce introduced in 2004

Jeffrey Dean et al. MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004.

Apache Hadoop since 2005

http://hadoop.apache.org/

Apache Hadoop 2.0 introduced in 2012

Vinod Kumar Vavilapalli et al. Apache Hadoop YARN: Yet Another Resource Negotiator, SOCC 2013.

New cluster resource management layer (YARN)

Implicit Parallelism

The map function has implicit parallelism

This is because the order of the application of the function f to elements in a list is commutative

We can parallelize or reorder the execution

MapReduce builds on this parallelism

MapReduce

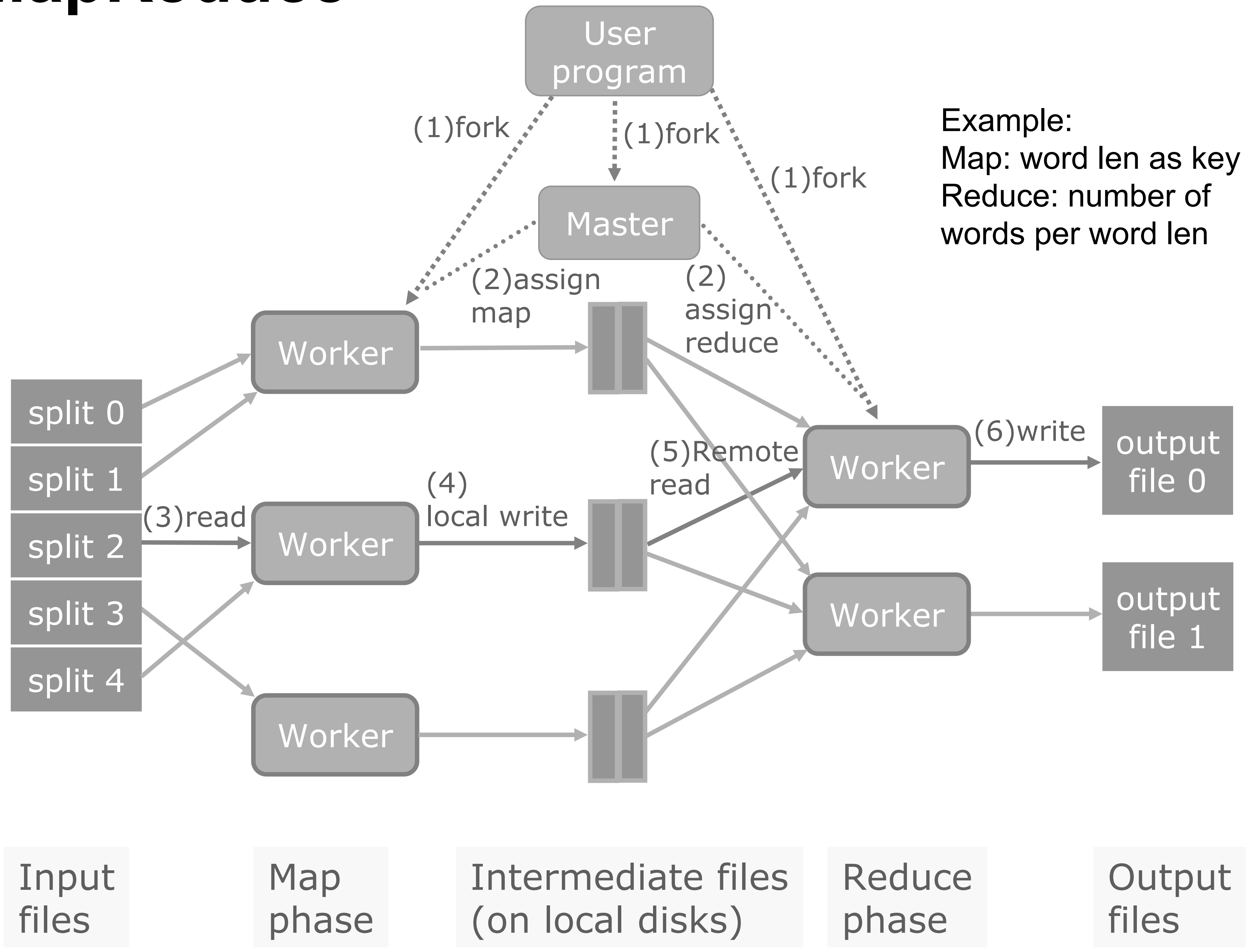
Automatic distribution and parallelization

Fault-tolerance

Cluster management tools

Abstraction for programmers

MapReduce



Example:
Map: word len as key
Reduce: number of words per word len

MapReduce terminology

Job is a full program that consists of a **Mapper** and **Reducer** for a dataset

Task is an execution of a Mapper / Reducer on some data

Task-in-Progress (TIP)

Task Attempt is an instance of an attempt to run a task on a node

Task Attempts

A given task is attempted at least once and possibly many times if it crashes

If a task crashes consistently, it will be abandoned eventually

Multiple attempts pertaining to a task may happen in parallel with the **speculative execution** feature

MapReduce example: counting words per word length

1. Input data

Word1
Word2
Word3
….
Word N

2. MAP Phase
(len of word, word) lists
Interim data files

Map 1
3: word 1
3: word 2
4: word 3

Map …

Map k

3. REDUCE Phase
(len of word, num of word) lists
Interim data files

Reduce 1

Reduce …

Reduce j

4- Final Phase
Combined reduce phase output

MapReduce Programming Model

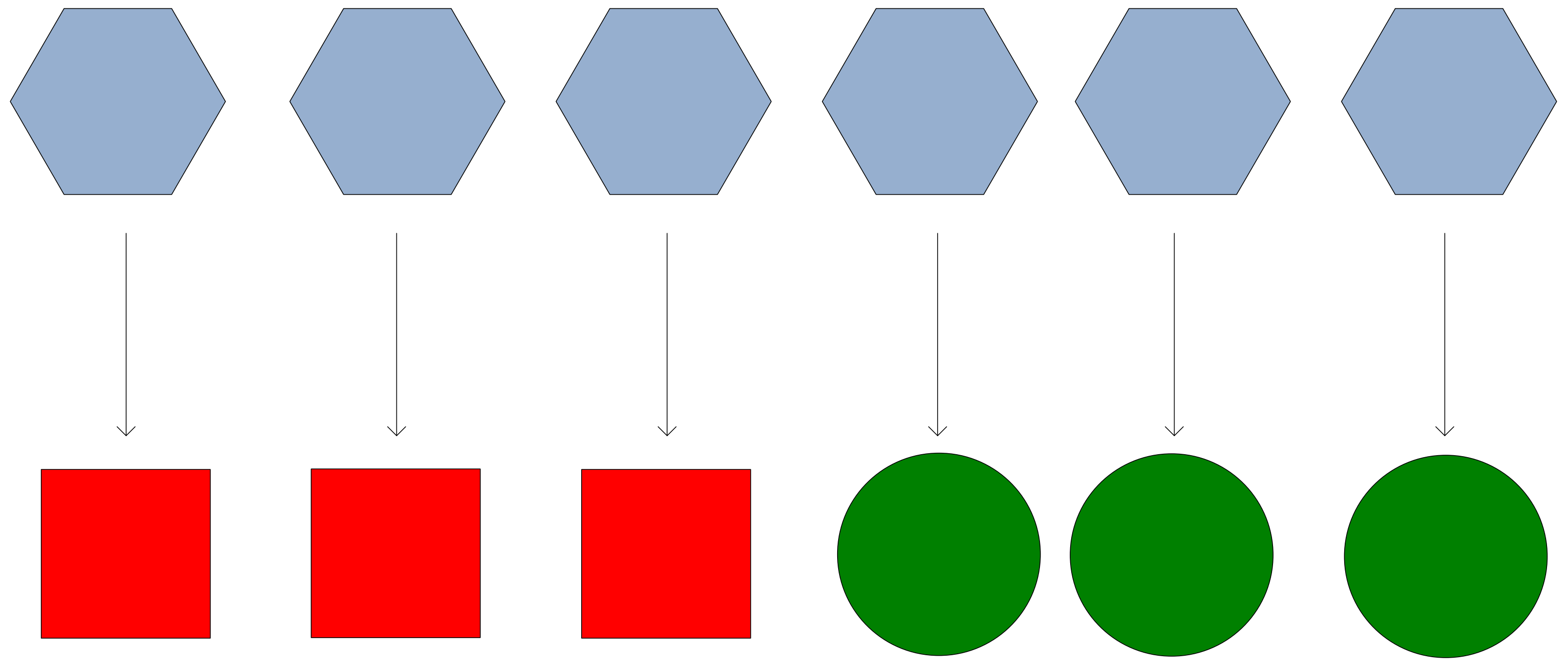
Inspired by functional programming

Two key functions that need to be implemented:

- **map** (in_key, in_value) → (out_key, intermediate_value) list
  - Data source records are fed as key,value pairs
  - map() produces one or more intermediate values with an output key
- **reduce** (out_key, intermediate_value list) → out_value list
  - Intermediate values for given key are combined into a list
  - Reduce() combines those values into one or more final values for the same output key
  - Optional and not needed by all applications

map

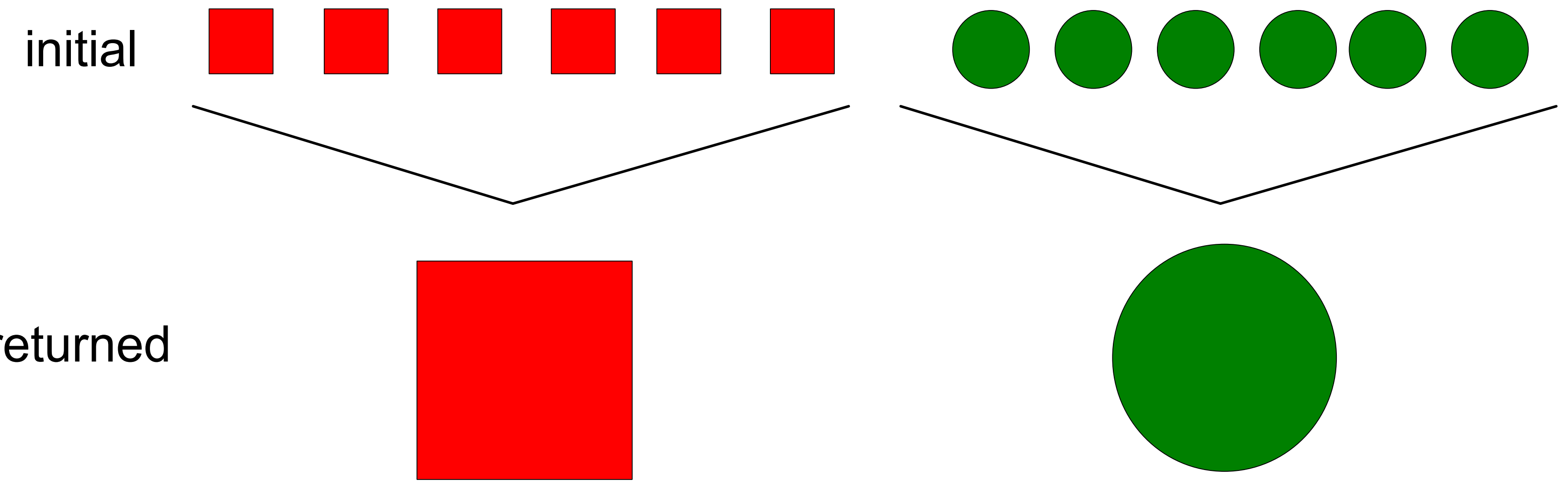
```
map  (in_key, in_value) ->
    (out_key, intermediate_value) list
```



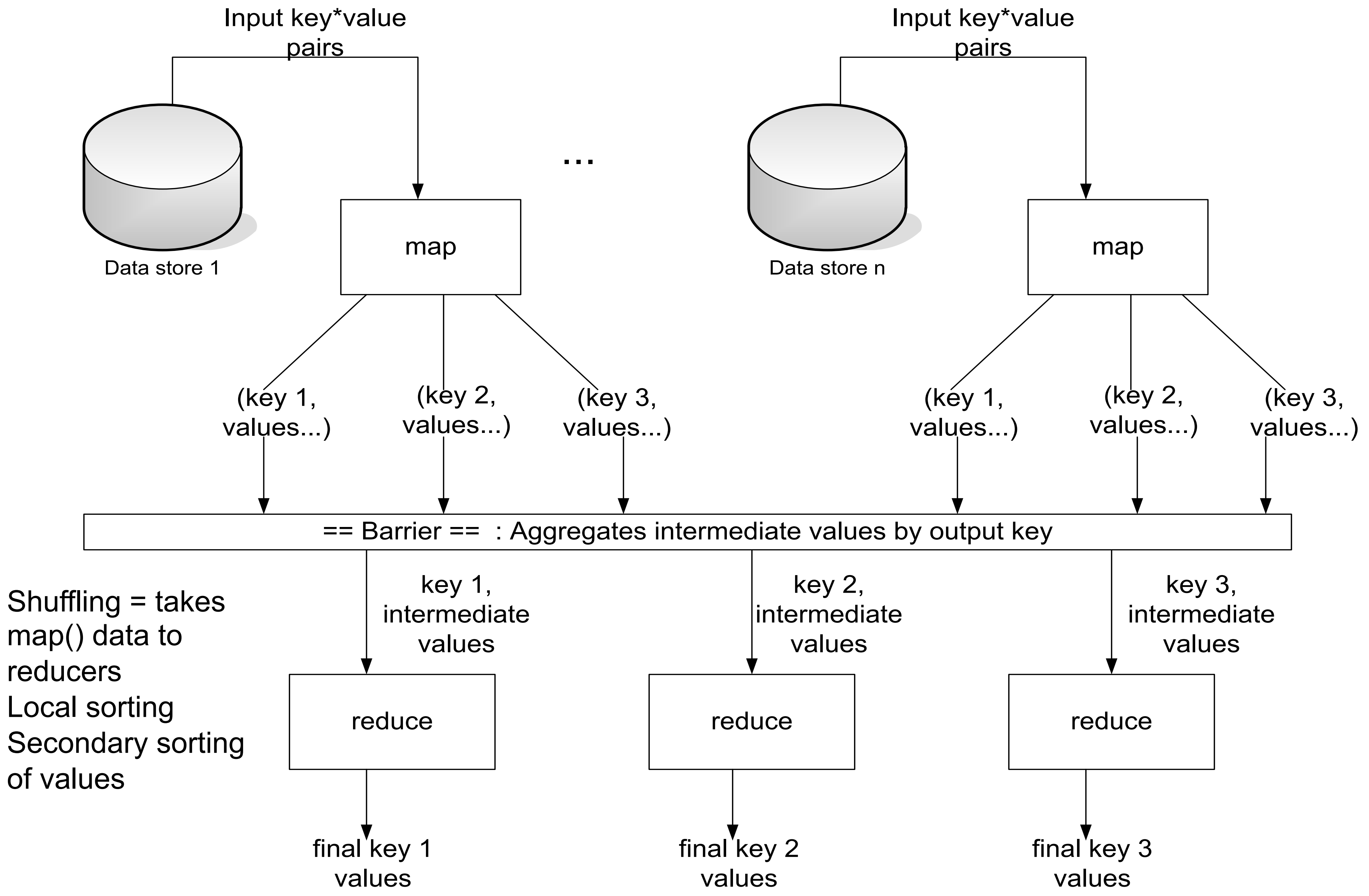
Source: https://courses.cs.washington.edu/courses/cse490h/08au/lectures/mapred.pdf

Reduce

```
reduce (out_key, intermediate_value list) ->
          out_value list
```



Source: https://courses.cs.washington.edu/courses/cse490h/08au/lectures/mapred.pdf



Source: https://courses.cs.washington.edu/courses/cse490h/08au/lectures/mapred.pdf

Reduce Details

- Three key phases
  1. Reducer copies sorted output (based on key) from each Mapper using HTTP
  2. The framework sorts Reducer inputs by keys, because different Mappers have emitted the same key. The shuffle and sort phases happen simultaneously. SecondarySort can be used to sort values.
  3. Reduce is applied for each key in the sorted inputs

Reducer output is not sorted.

MapReduce Parallelism

- Both map() and reduce() operations are executed in parallel
- Map() creates intermediate values from input
- Reduce() functions operate on different output keys
- Values are independently processed

- Note that reduce cannot start before map is finished
- Secondary sort on values: the application can extend key with a secondary key and define a grouping comparator

Locality

The map() task input data is divided into GFS/HDFS blocks

Master program distributes tasks based on the location of data

Map() tasks should be run on the same machine as the physical data file (or the same rack)

Fault Tolerance

Fault tolerance is realized by periodic heartbeats

Master pings worker nodes to detect node failures

Master must re-execute tasks that have failed

Master can notices if particular input key/values cause problems in map and can skip those

Node failures

In the worst-case scenario the master compute node fails causing the job to be restarted

Other failures can be managed by the master

Failure of a worker node requires that the task is assigned to another worker

Master reschedules failed tasks when workers become available

Optimizations

Reduce tasks cannot start before the whole map phase is complete

Thus single slow machine can slow down the whole process

Master can execute many redundant map tasks and then use the results of the first task to complete

Optimizations: Combining Phase

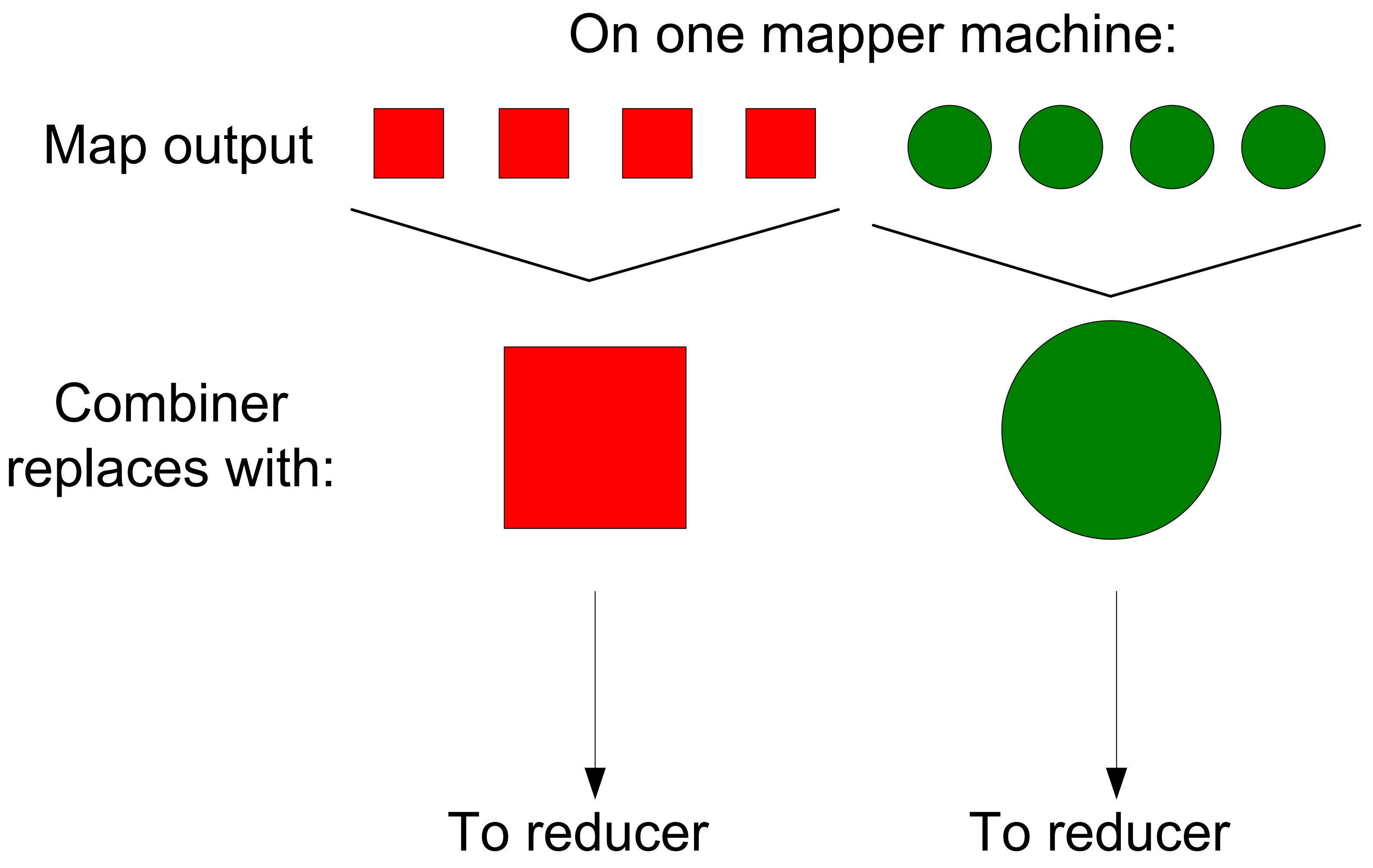
Performance can be increased by running a mini reduce phase on local map output

Executed on mapper nodes after map phase

Saves bandwidth before sending data to a full reducer

Reducer can be a combiner if it is commutative and associative

Combiner, graphically



Source: https://courses.cs.washington.edu/courses/cse490h/08au/lectures/mapred.pdf

Partitioner

Partitioner divides the **intermediate key space**

Assigns intermediate key-value pairs to reducers

Thus **n partitions results in n reducers**

Between map and reduce phases:

- data is shuffled: parallel-sorted and exchanged
- data is moved to the correct shard for reducing
- partition function accepts the key and the number of reducers and then returns the index of the reducer

Supports load balancing

MapReduce Summary

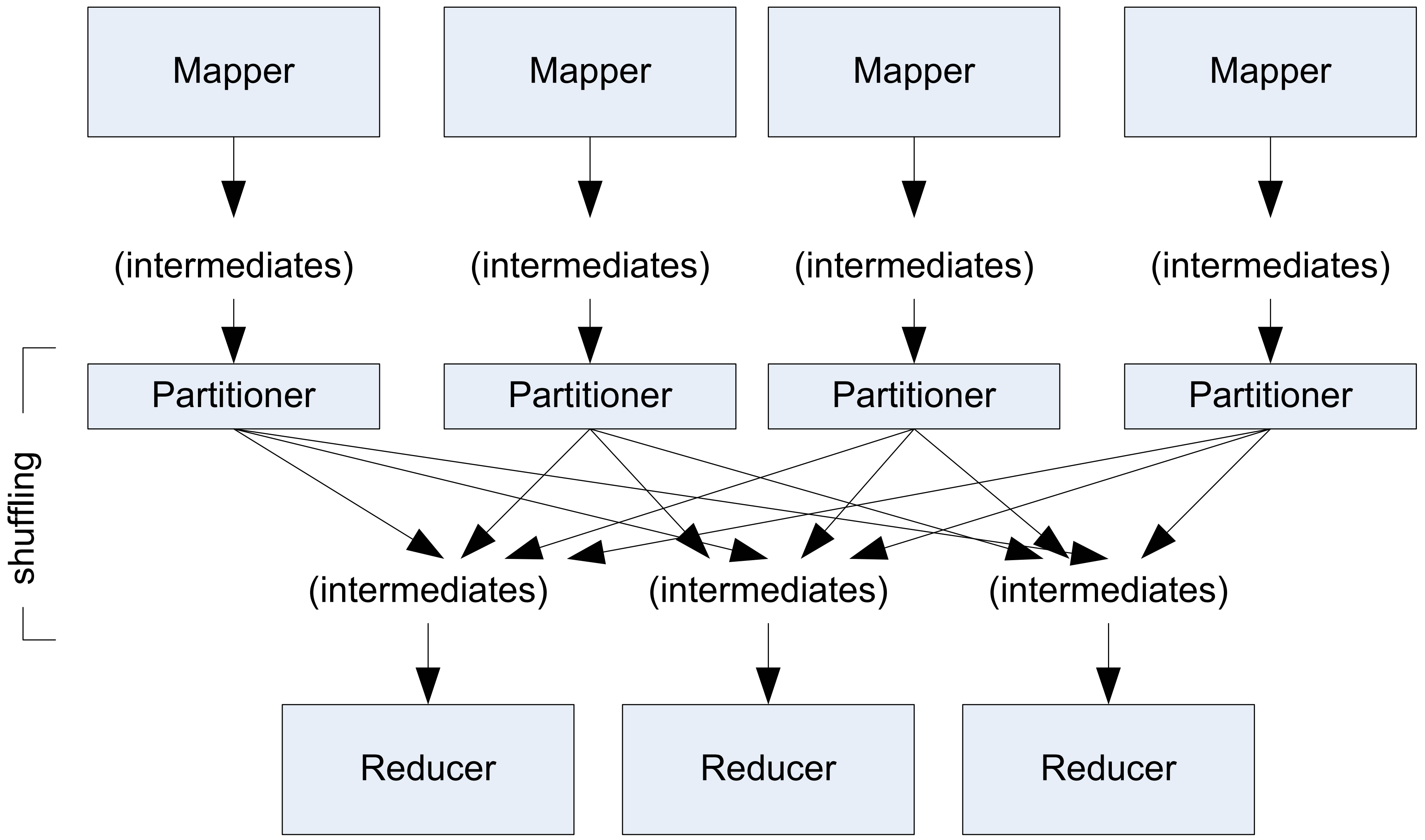
Two key functions that need to be implemented:

- **map** (in_key, in_value) → (out_key, intermediate_value) list
- **reduce** (out_key, intermediate_value list) → out_value list

With two optimizations:

- **combine** (key, intermediate_value list) → intermediate_out_value list
- **partition** (key, number of partitions) → partition for key

Partition and Shuffle



Data management as part of MapReduce

Purlieus and CAM

Data management as part of MapReduce

Need to know task types (map/reduce intensive)

Map intensive jobs benefit from locality awareness

Up to 80% reduction in execution time

Key algorithms for MapReduce

- Inverted Index
- Sorting
- PageRank
- Sorting
- Searching
- Statistics
    - Average, SD, count
- Advanced algorithms

Filtering Algorithms

Finding files or items with specific characteristics

Searching for patterns in web logs or files

Filtering is mostly done in the map phase

Reduce can be simply the identity

Aggregation Algorithms

Computing the minimum, maximum, sum, average, ... of the given values

Count the number of Tweets per day

Map is simple or the identity, reduce is doing most of the work

K-Means Clustering Algorithm

Iterative algorithm that is run until it converges

1. K initial points (centers) are chosen at random.
2. K clusters are formed by associating every data point (observation) with the nearest center.
3. For each cluster, recompute the centers (determine centroid)
4. Repeat from 2 until convergence.

Source: Riccardo Torlone. Analytics on Big Data. Universita Roma Tre.

K-Means for MapReduce

- Map phase
  - Each map reads the K centroids and a block from the input dataset
  - Each point is assigned to the closest centroid
  - Output: <centroid, point>
- Reduce phase
  - Obtain all points for a given centroid
  - Recompute the new centroid
  - Output: <new centroid>
- Iteration:
  - Compare the old and new set of K centroids
  - If they are similar then Stop
  - Else Start another iteration unless maximum of iterations has been reached.

Optimizing K-Means for MapReduce

Combiners can be used to optimize the distributed algorithm

- Compute for each centroid the local sums of the points
- Send to the reducer: <centroid, partial sums>

Use of a single reducer

- Data to reducers is very small
- Single reducer can tell immediately if the computation has converged
- Creation of a single output file

Limitation: iterative algorithms

MapReduce tasks must be written as acyclic dataflow programs

- Stateless mappers and reducers
- Batch model

Difficult to implement iterative processing of datasets

- Machine learning typically requires iterative operation: the dataset is visited multiple times by the algorithm