- Yogesh Simmhan
- simmhan@iisc.ac.in

- Department of Computational and Data Sciences
- Indian Institute of Science, Bangalore

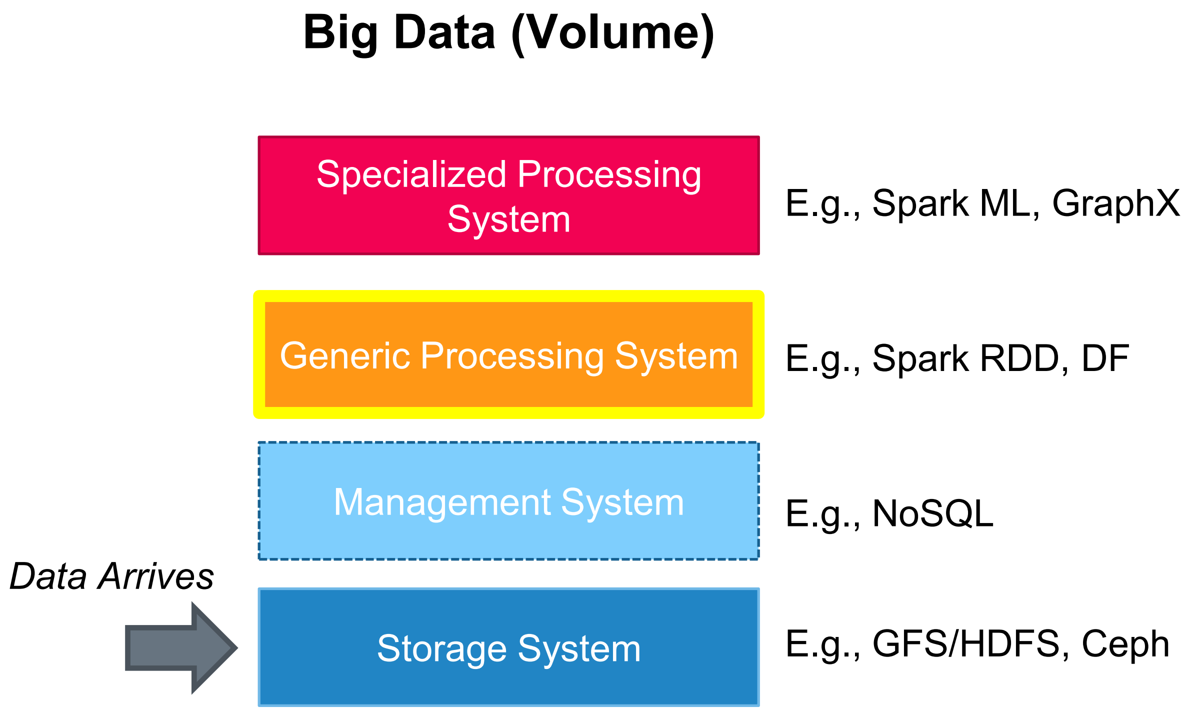
DS256 (3:1)

Scalable Systems for Data Science

Module 2

Processing Large Volumes of Big Data

Role of Big Data Processing System for Large Data Volumes



Required Reading

- ***MapReduce: Simplified Data Processing on Large Clusters***, *Dean and Ghemawat, USENIX OSDI, 2004*
- ***Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing***, *Matei Zaharia, et al., USENIX NSDI, 2012*
- ***Spark SQL: Relational Data Processing in Spark***, *Michael Armbrust, et al., ACM SIGMOD 2015*
- *Select chapters from **Learning Spark**, Holden Karau, et al., 1st Editions and **Learning Spark**, Jules S. Damji, Brooke Wenig, Tathagata Das, Denny Lee, 2nd Edition*
- **Optional: Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics**, Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao Chen Wang, Berthold Reinwald, and Fatma Ozcan, *VLDB* 2015

MapReduce

MapReduce: Simplified Data Processing on Large Clusters, Dean and Ghemawat, *USENIX OSDI*, 2004

MapReduce

*"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**."*

Dean and Ghermawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI, 2004

Google's Cluster Model (2004)

- Machines have 2 x86 CPU, running Linux, with 2-4 GB of memory
  - 2020: Raspberry Pi 4B has 4-core ARM, 2-4GB RAM, Rs.5000 ☺
- Commodity networking – 100 Mbps or 1 Gbps per machine, low bisection bandwidth.
- A cluster has 100's – 1000's of machines, failures are common
- Storage on inexpensive IDE disks attached to servers
  - GFS manages the data
- Users submit jobs to a scheduling system
  - Each job is a set of tasks
  - Mapped by the scheduler to available machines in cluster

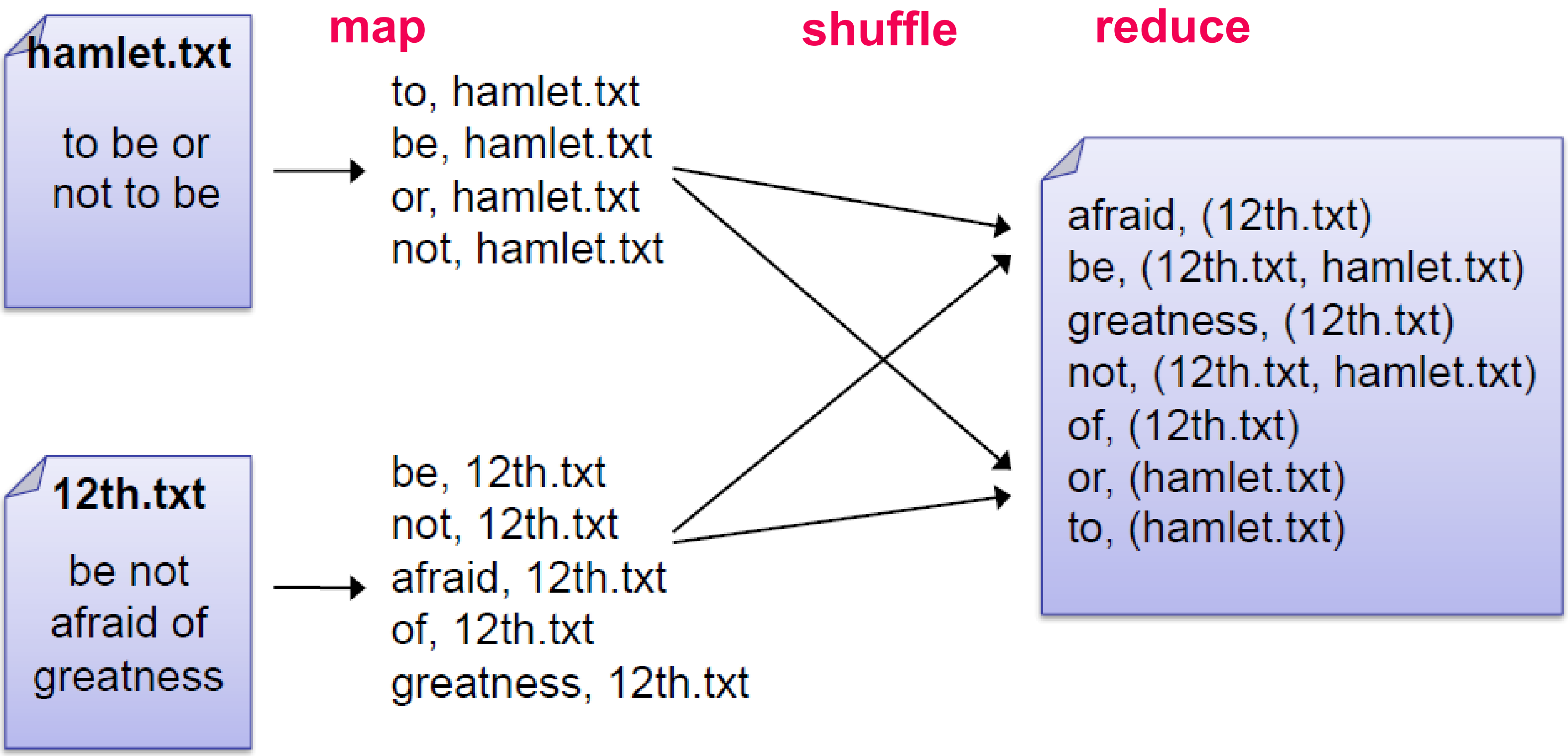
MapReduce Design Pattern

- Programming model for distributed applications
  - Clean abstraction for programmers
  - Automatic parallelization & distribution
- Fault-tolerance
- Batch data processing system
  - Large inputs sizes
- Simple data-intensive applications
  - Distributed Grep: Document list → Occurrence of search term
  - URL Access Frequency: URL access list → <URL, freq>
  - Reverse Web-Link Graph: <target,src> → <src, target[]>
  - Term-Vector per Host: <host,word[]> → <host,<word,freq>[]>

MapReduce: Data-parallel Programming Model

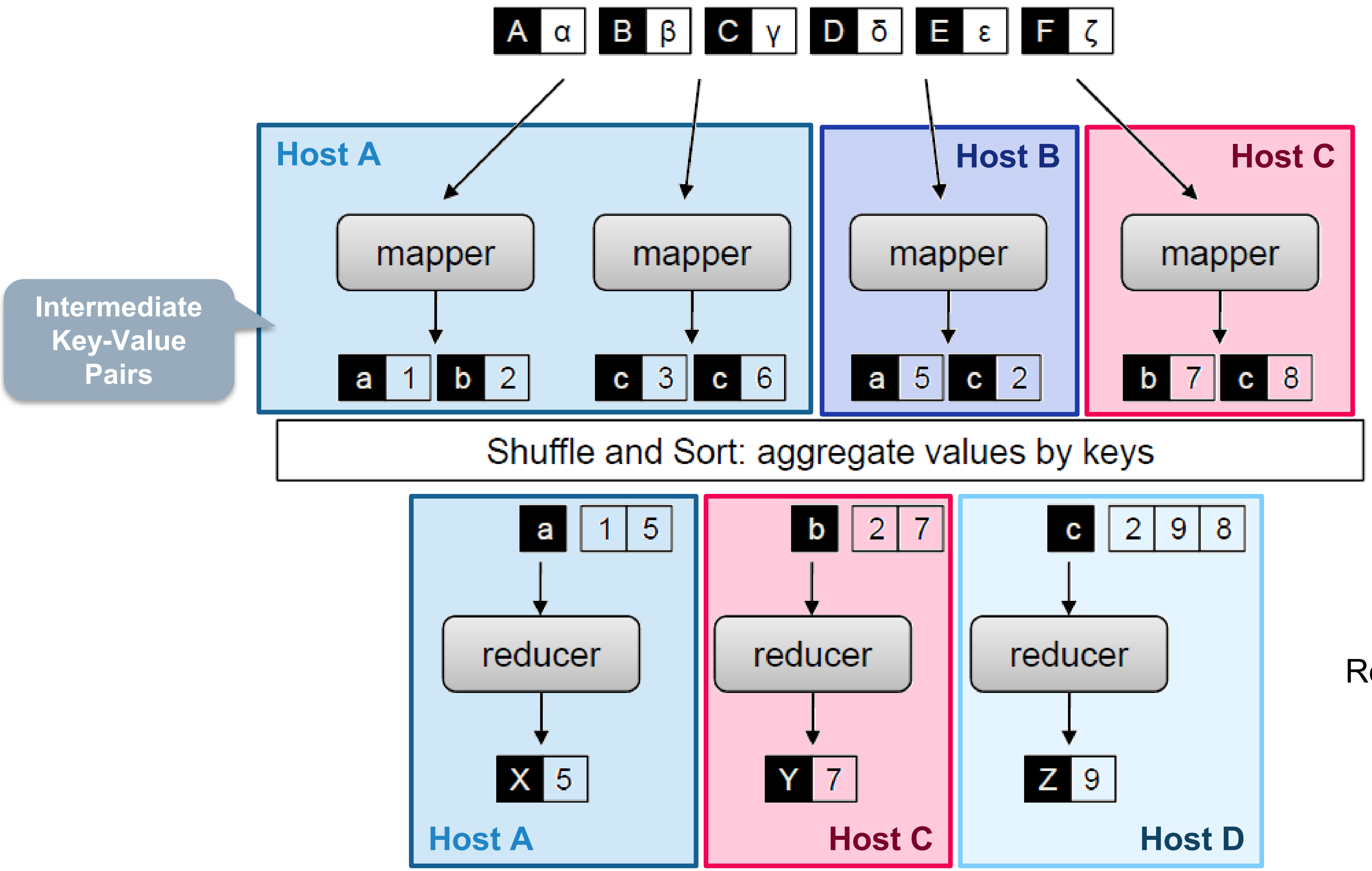
- Process data using **map** & **reduce** user-defined functions
- `map(`k_i`, `v_i`)` → `List<`k_m`, `v_m`>`
  - ***map*** is called once on every input item
  - Emits a series of intermediate key/value pairs
- **`shuffle & sort phase`**
  - All map output values (**v_m**) with a given key (**k_m**) are *grouped* together, keys *sorted* within a group
  - Happens internally within the framework
- `reduce(`k_m`, List<`v_m`>)` → `List<`k_r`, `v_r`>`
  - ***reduce*** is called once on *every unique key & all its values*
  - Emits a value that is added to the output

Inverted Index using MR



- map(url, line)
  - *foreach*(word *in* line.split()) *emit*(word, url)
- reduce(word, url[])
  - *save*(word, findDistinct(url[]))

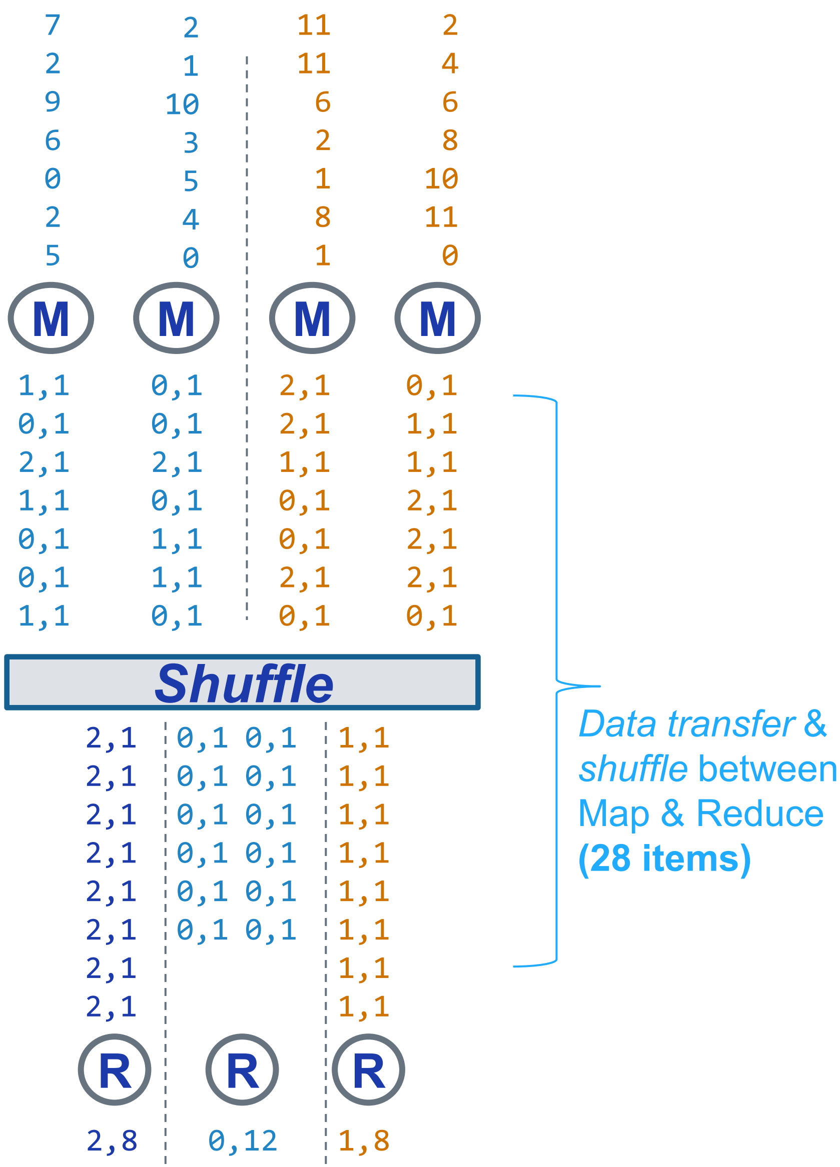
Map-*Shuffle-Sort*-Reduce



Combiner: Sum

Reduce: Max

Histogram using MR

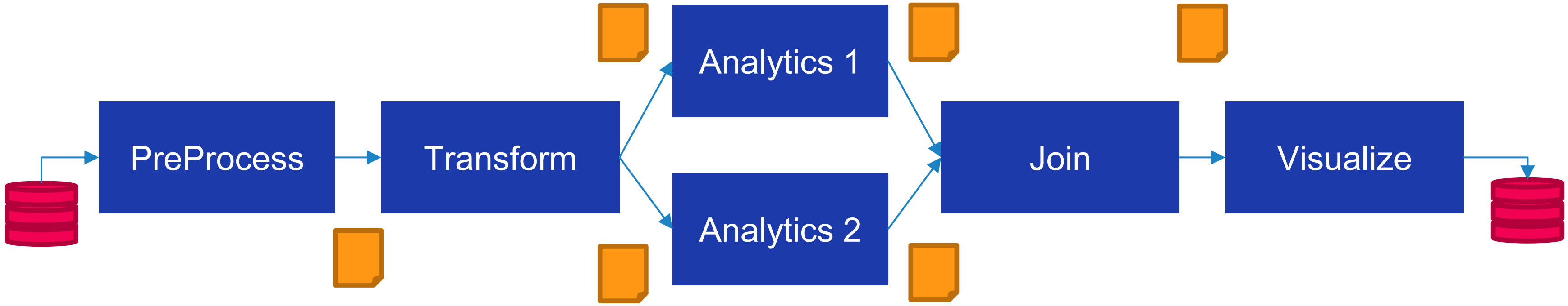



```
int bucketWidth = 4 // input

Map(k, v) {
   emit(floor(v/bucketWidth), 1)
   // <bucketID, 1>
}



// one reduce per bucketID
Reduce(k, v[]){
   sum=0;
   foreach(n in v[])  sum+=n;
   emit(k, sum)
   // <bucketID, frequency>
}
```

Limitations of MapReduce

Limitations in **Expressivity** of MapReduce

- Multi-stage computing not simple
  - Many different jobs
- Complex code for simple transformations
  - Repetitive, not *data centric*

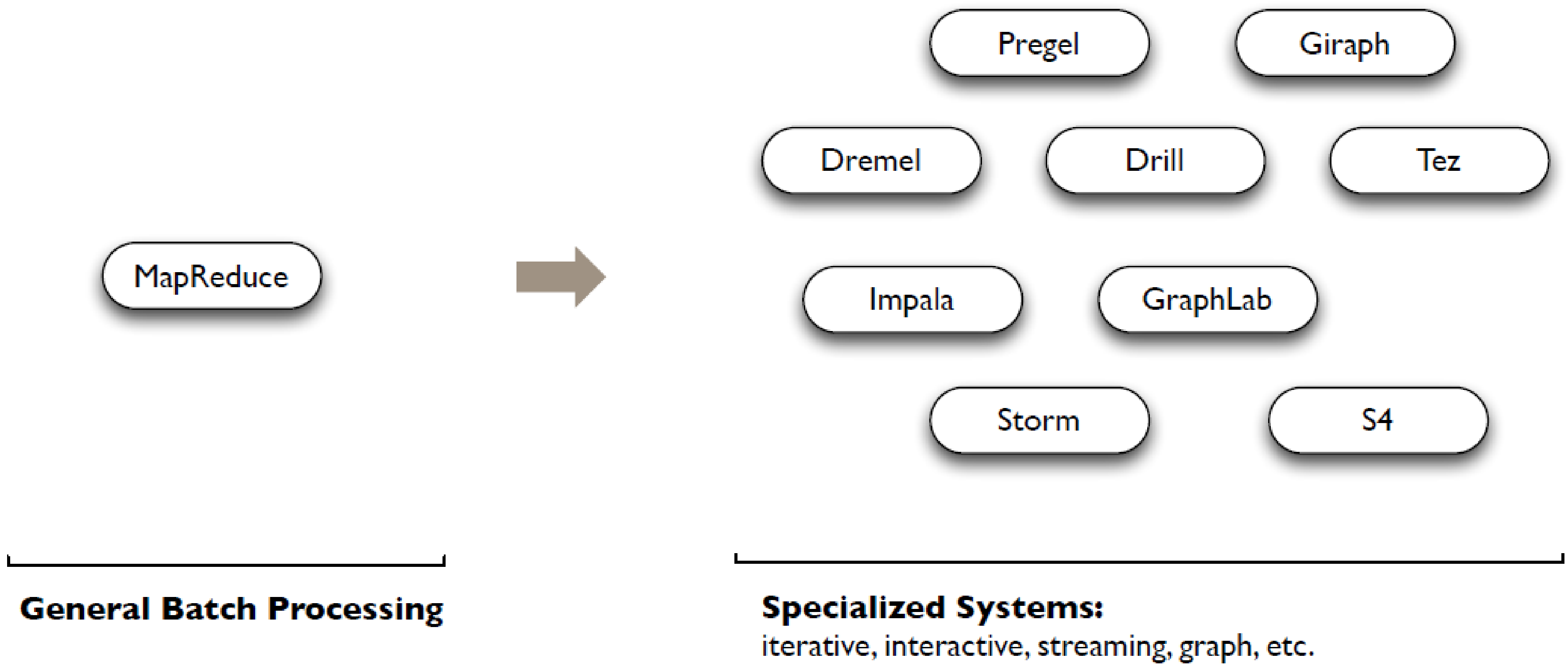


https://stanford.edu/~rezab/sparkclass/slides/itas_workshop.pdf

Limitations of MapReduce

Limitations in **Expressivity** of MapReduce

- Limited support for non-text, Non-static data

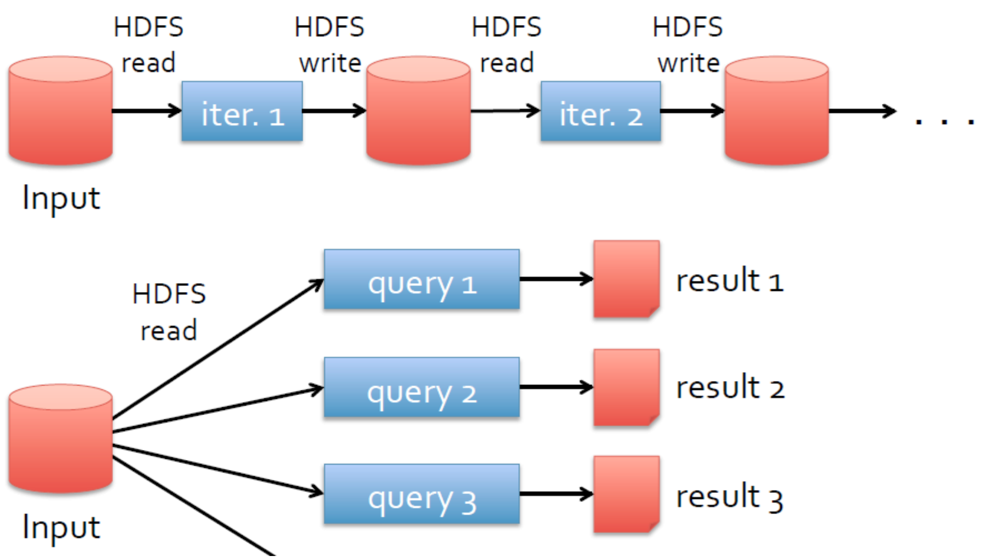


https://stanford.edu/~rezab/sparkclass/slides/itas_workshop.pdf

Limitations of MapReduce

Limitations in **Performance** of MapReduce

- Poor performance for:
  - Complex, multi--stage applications (e.g. iterative machine learning & graph processing)
  - Interactive *ad hoc* queries



Latency & Bandwidth

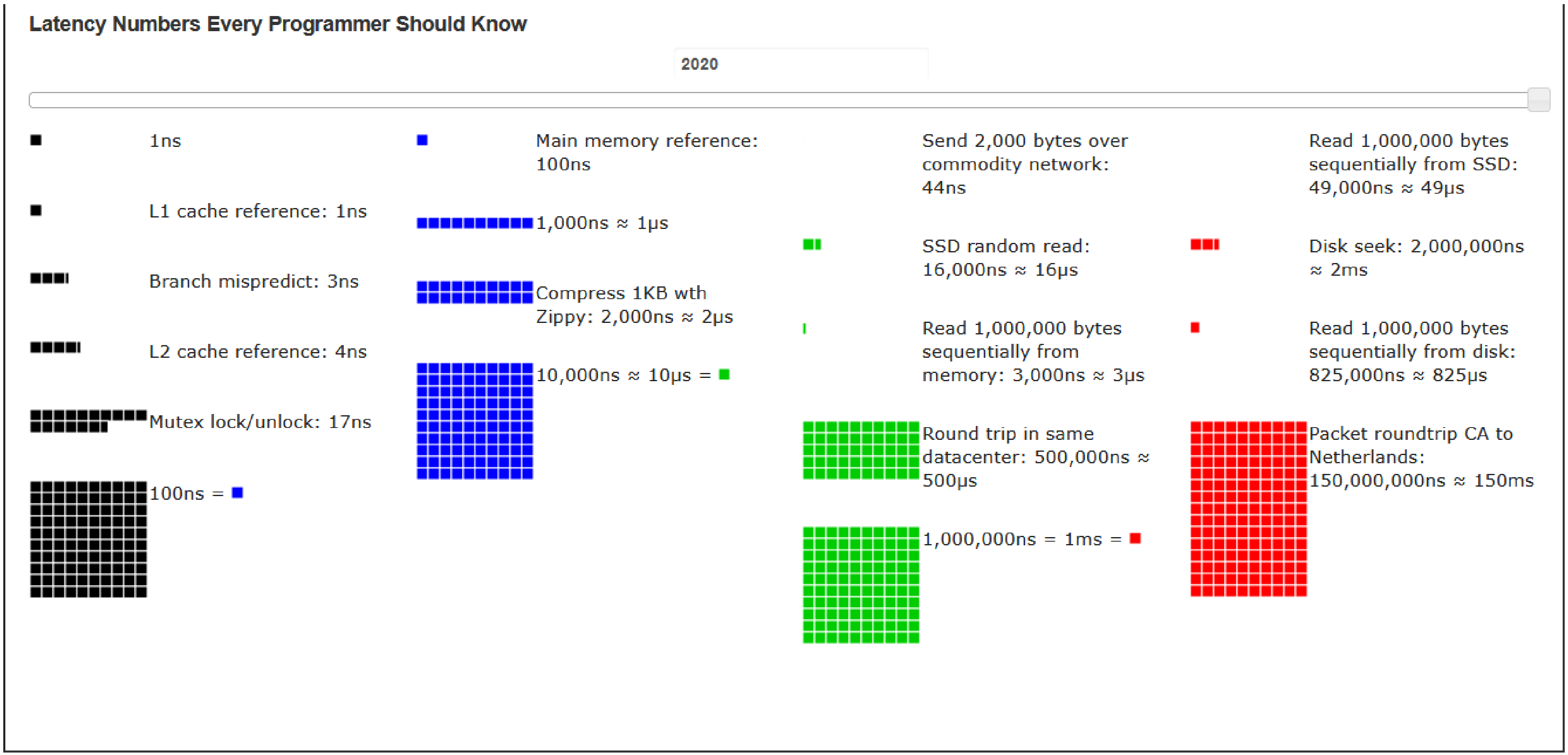
- L1 cache reference
- L2 cache reference
- **Main memory reference**
- **Send 1K bytes over 1 Gbps network**
- **Read 4K randomly from SSD***
- **Read 1MB sequentially from memory**
- **Round trip within same datacenter**
- **Read 1MB sequentially from SSD***
- **Send 1MB over 1 Gbps network**
- **Disk seek**
- **Read 1MB sequentially from disk**
- **Send packet CA->NL->CA**

https://gist.github.com/jboner/2841832

Latency & Bandwidth

L1 cache reference	0.5	ns				
L2 cache reference	7	ns				
Main memory reference	**100**	**ns**				
Send 1K bytes over 1 Gbps network	**10,000**	**ns**	**10**	**µs**		
Read 4K randomly from SSD*	**150,000**	**ns**	**150**	**µs**		
Read 1MB sequentially from memory	**250,000**	**ns**	**250**	**µs**		
Round trip within same datacenter	**500,000**	**ns**	**500**	**µs**		
Read 1MB sequentially from SSD*	**1,000,000**	**ns**	**1,000**	**µs**	**1**	**ms**
Send 1MB over 1 Gbps network	**8,250,000**	**ns**	**8,250**	**µs**	**8**	**ms**
Disk seek	**10,000,000**	**ns**	**10,000**	**µs**	**10**	**ms**
Read 1MB sequentially from disk	**20,000,000**	**ns**	**20,000**	**µs**	**20**	**ms**
Send packet CA->NL->CA	**150,000,000**	**ns**	**150,000**	**µs**	**150**	**ms**

https://gist.github.com/jboner/2841832

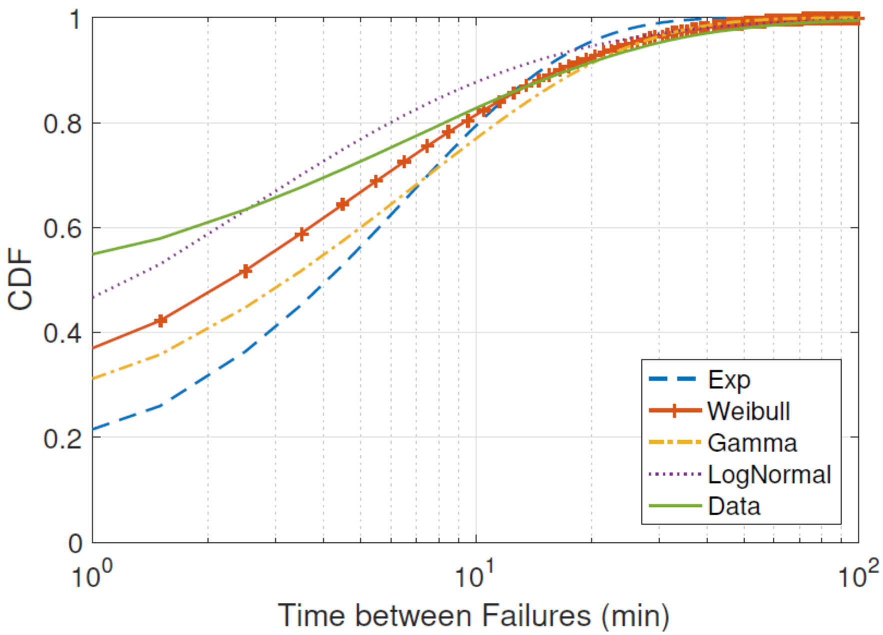


https://colin-scott.github.io/personal_website/research/interactive_latency.html

“

Bandwidth of Memory $\gg$ Network or Disk

MTTF in Data Center



"The MTBF (mean time between failures) across all data centers we investigate (with hundreds of thousands of servers) is only 6.8 minutes, while the MTBF in different data centers varies between 32 minutes and 390 minutes."

→ MTBF with 1000 servers is 680mins

→ MTBF with 100 servers is 6800mins (4.7 days)

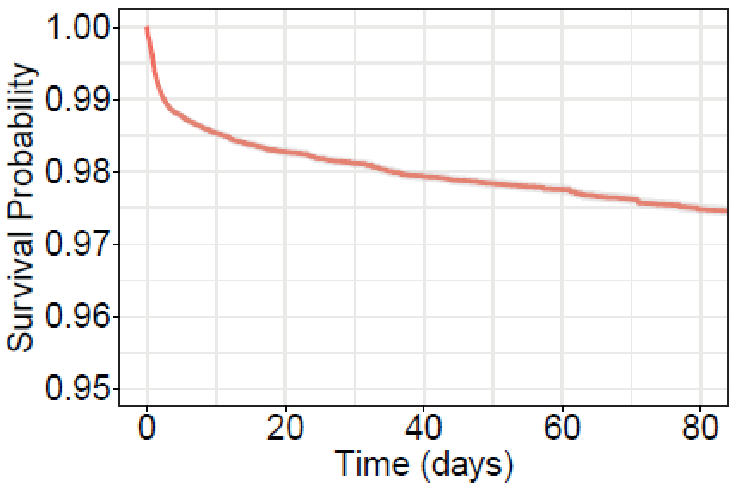


Figure 4: Kaplan Meier survival estimate of datacenter switches, shaded region shows 95% confidence intervals.

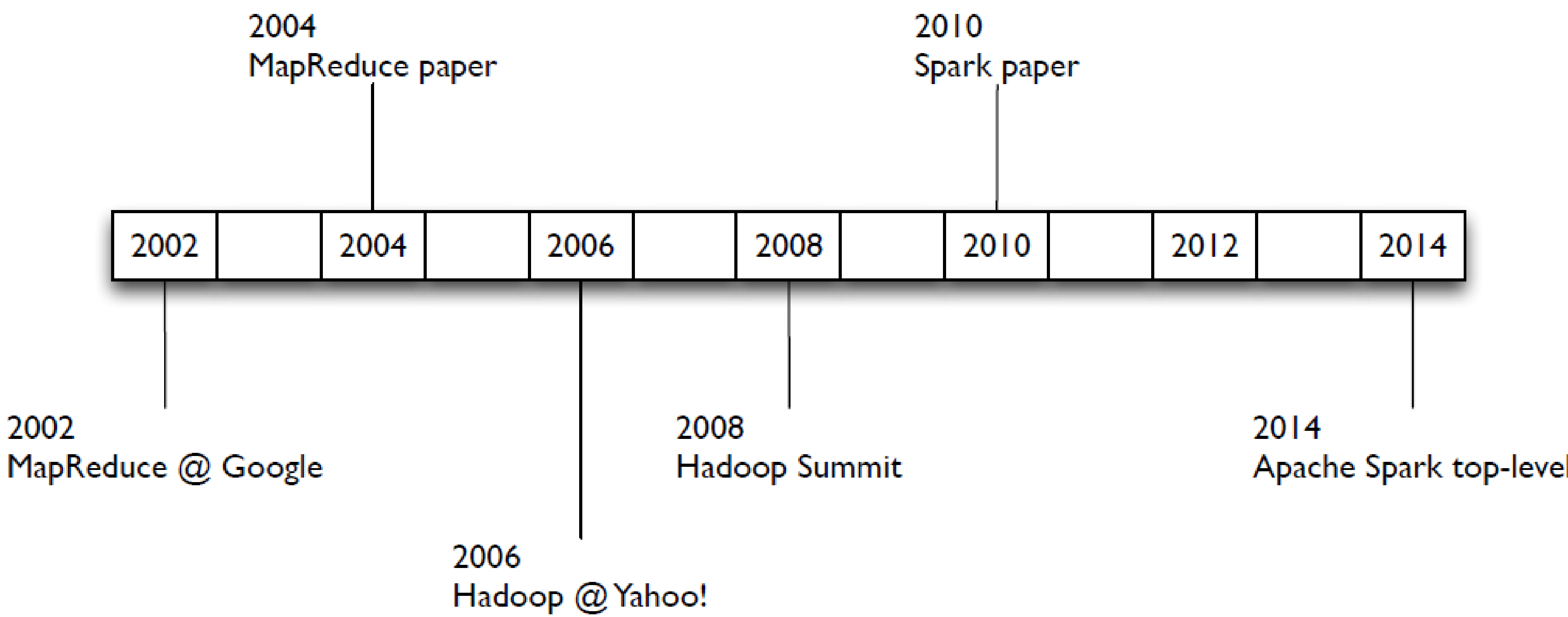
Surviving switch failures in cloud datacenters, ACM SIGCOMM Computer Communication Review, Volume 51Issue 2April 2021
What Can We Learn from Four Years of Data Center Hardware Failures?, DSN, 2017

“

Failures may be infrequent during the lifetime of an application execution

From MapReduce to Spark

- Google's MapReduce
  - Programming Model
  - Apache Hadoop runtime environment



https://stanford.edu/~rezab/sparkclass/slides/itas_workshop.pdf

Apache Spark

Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, Zaharia, Chowdhury, Das, Dave, Ma, McCauly, Franklin, Shenker and Stoica, *USENIX NSDI*, 2012

Learning Spark
Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia, O'Reilly, First and Second Editions

https://databricks.com/p/ebook/learning-spark-from-oreilly

The Spark Ecosystem

- Core Spark Engine
  - RDDs, Transformations, Actions, batch processing
- Higher level abstractions
  - Data frames, SQL-like queries
  - Discretized streams, semi-realtime data
  - Machine learning libraries, **Mllib**
  - Linked data analytics, **GraphX**

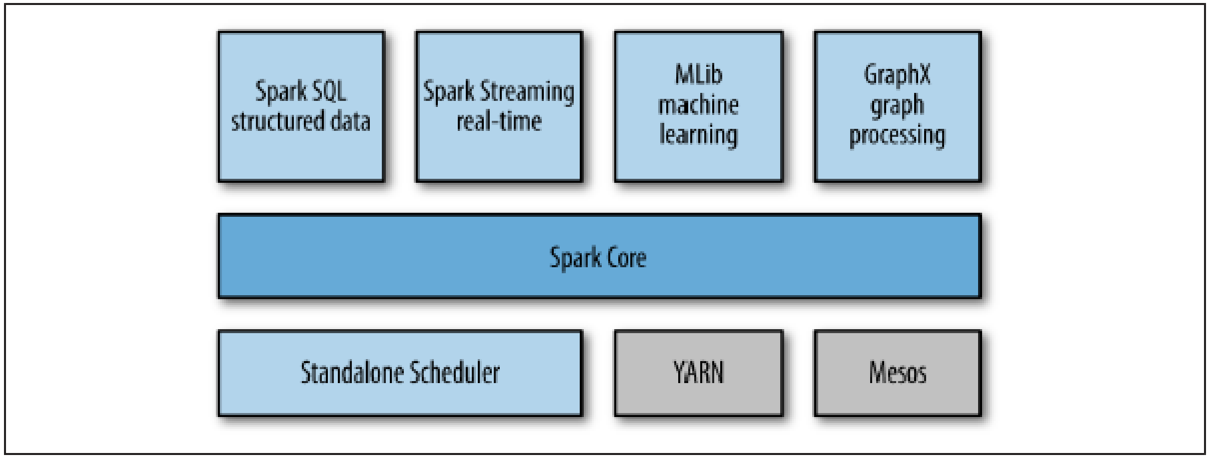
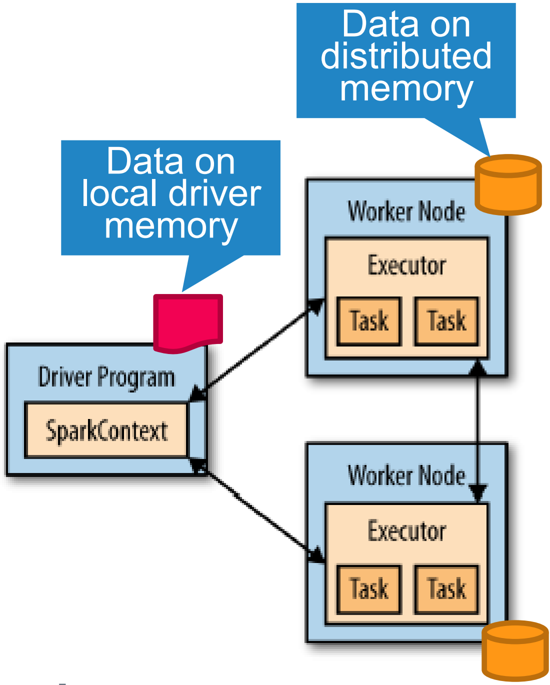


Figure 1-1. The Spark stack

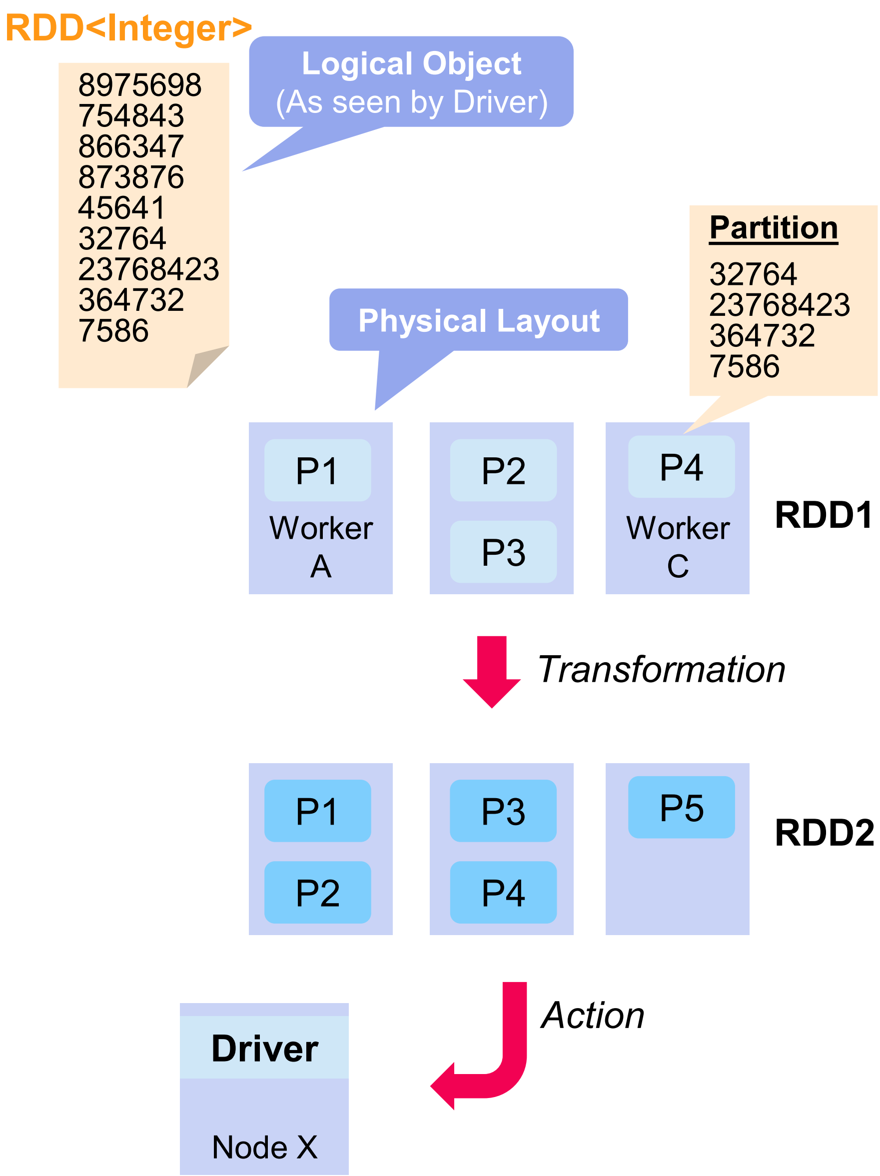
Spark: A Distributed Execution Engine

- **Driver**: User program for application, uses Spark Context, local variables
- **Spark Context**: Gives access to distributed computing environment
- **Worker**: Machines on which actual heavy-lift happens
- **Executor**: Spark execution environment in a worker, Process, exclusive to an application
- **Task**: Single operation on data, thread



Spark RDD
Resilient Distributed Dataset

- Collection of homo-geneous objects
  - Order is not preserved*
- **Distributed** on workers
  - 1 or more **Partitions**
- **Read-only**, immutable
- Can be **rebuilt**
- Can be **cached**
- MR like data-parallel operations
  - **Execute** on workers



Creating and Operating on an RDD

- Users can provide driver code in multiple languages: *Scala*, *Java*, ***Python*** ***(PySpark)***
  - Spark offers equivalent transformations and actions in each language
- Actual Spark execution environment is in Scala
- Create RDD by loading data
  - HDFS, local vars, filesystem, NoSQL DB, etc.
  - Data is loaded on partitions on different workers
- RDD Object offers a logical view of the dataset
  - Can perform operations on the object

Example 3-1. Creating an RDD of strings with textFile() in Python

```
>>> lines = sc.textFile("README.md")
```

Creating and Operating on an RDD

- Users can provide driver code in multiple languages: *Scala*, *Java*, ***Python*** ***(PySpark)***
  - Spark offers equivalent transformations and actions in each language
- Actual Spark execution environment is in Scala
- Create RDD by loading data
  - HDFS, local vars, filesystem, NoSQL DB, etc.
  - Data is loaded on partitions on different workers
- RDD Object offers a logical view of the dataset
  - Can perform operations on the object

Example 3-1. Creating an RDD of strings with textFile() in Python

```
>>> lines = sc.textFile("README.md")
```

Language Bindings

- Users can provide driver code in multiple languages
  - Scala, Java, Python
- Spark offers equivalent transformations and actions in each language
- Logic within transformations and actions can also be in these languages
- Actual Spark execution environment is in Scala
  - Standard data structure mapping
  - Python code is pickled (de/serialize) and shipped remotely

https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals

Passing Functions to Spark Operations

- Lambda syntax
  - **Functions** are *input parameters* to other functions
  - Pass short functions concisely, inline

```python
pythonLines = lines.filter(lambda line: "Python" in line)
```

⬇

```python
def hasPython(line):
    return "Python" in line

pythonLines = lines.filter(hasPython)
```

Programming with RDDs

Learning Spark

Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia, O'Reilly, First Edition

Chapter 3

Basics of Transformations

- Returns a new RDD, computed lazily
- Transforms tend to be **element-wise** operations
  - **Iterate** through each item, apply the operation, e.g. *Filter*
- **Filter** on *inputRDD* does not affect *inputRDD*
  - Returns a new RDD, *warningsRDD*
- **Union** operates on two RDDs
  - One of them is an input parameter

```python
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

Basics of Actions

- *Actually* triggers operations, returns a final result to driver
  - Force any required transformations to be executed
  - Count, Take, Collect
- Result of action must fit in memory of driver
  - Else, can write RDD to HDFS, saveAsTextFile
- RDDs are computed from scratch when actions are called…See *persist/cache*

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
    print line
```

Lazy Evaluation

- Transformations are lazily evaluated
  - Calling a transform does NOT immediately execute it
- *Action* **triggers** execution of *dependent transformations*
- E.g., load().map().count()
  - Load & Map do not execute till we see Count
- Allows Spark to reduce the number of passes through the data
  - Materializes RDD only when required
  - Reused RDDs that have been materialized earlier
  - Immutability!

Lineage Graph

- Keeps track of operations used to derive an RDD
- Helps *lazily materializ* RDD
- Helps *recover* RDD or their partitions that are lost