Big Data Processing Pipelines

John Klein

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Copyright 2021 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material is distributed by the Software Engineering Institute (SEI) only to course attendees for their own individual study.

Except for any U.S. government purposes described herein, this material SHALL NOT be reproduced or used in any other manner without requesting formal permission from the Software Engineering Institute at permission@sei.cmu.edu.

Although the rights granted by contract do not require course attendance to use this material for U.S. Government purposes, the SEI recommends attendance to ensure proper understanding.

DM21-0385

Who am I?

jklein@sei.cmu.edu https://www.linkedin.com/in/johnrklein

https://resources.sei.cmu.edu/library/author.cfm?authorID=3271

By Alexrk2 - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=7123857

By Paul Keleher from Mass, US - fishermans memorial, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=7320920

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Distributed Data Processing Frameworks

In this module, we will discuss

- Motivations and needs for aggregation in big data systems
- An overview of Hadoop and HDFS
- Hadoop and NoSQL databases
- Distributed computations with Spark
- Stream processing and Storm

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Aggregation Frameworks

NoSQL technologies offer simple querying approaches:

- Key-based
- Limited/no support for aggregation
  - Simple key-based queries
  - Note: Graph DBs and document stores offer more in this area

Example – wine store:

- For all Canadian wine in stock, return total of bottles per vintage and value

Requires scanning all objects

- If (country = “CA”) …
- Potentially large, sharded data store

Id
Wine Varietal
Name
Variety
No. in stock
Region
Price per unit
Country
Owners
Address
Vintage

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Aggregation Frameworks: Motivations

Big data sets are slow to access sequentially on a single disk:

- 1-TB disk at 100 MB/s takes 2.5+ hours to read
- Even slower to write
- Seek times improving more slowly than transfer rates

Parallel access speeds things up

- Partition 1 TB over 100 disks
- ~2 minutes to read

Requires replication to provide high reliability

- Studies show ~8% of hard disks in a data center fail annually

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Aggregation Frameworks

A lot of big data is

- Written once
- Read a lot
- Often analyzed in its entirety

This makes it suitable for high-speed streaming reads

- Have minimal seek times
- Exploit disk transfer rates
- Analyze locally to select data that matches a give query

Enter MapReduce …

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

MapReduce (and Hadoop)

MapReduce: programming model (Google 2004)

Designed for non-interactive processing

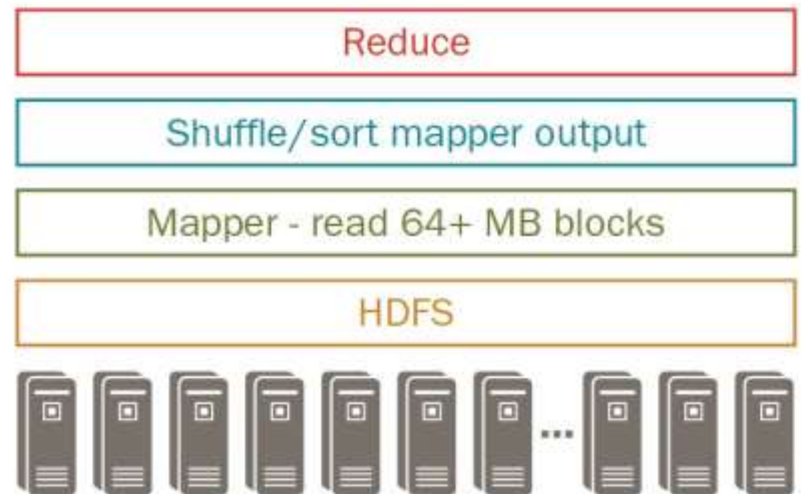
Hadoop: open source Apache implementation (2006)

Data stored in Hadoop Distributed File System

- Distributed across multiple nodes
- Replicated for fault tolerance

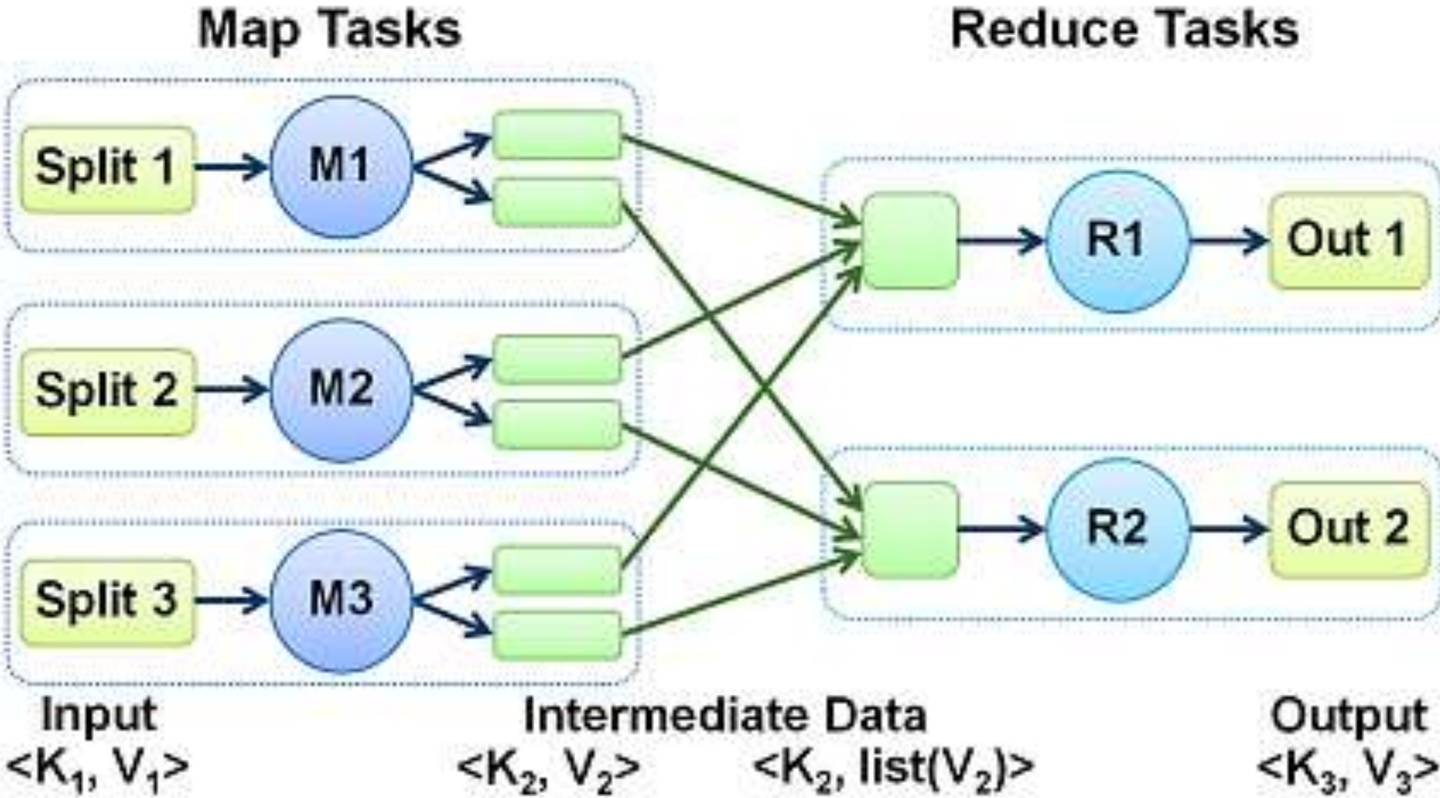
Two main phases:

- Map: process local data to select values relevant for a query
- Reduce: combine and analyze results emitted from the map phase



© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

MapReduce: Acyclic Dataflow Model



© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

MapReduce Example–Find the highest rated wine worldwide for any given vintage

Input: Wine ratings

- {Vintage, Wine, Rating, Description}

```
2011, Quinta do Noval Port, 100, Yum!!
2012, Penfold’s Grange, 99, the best
2012, Old Vines Swiller, 76, ughhh!!
2012, Leonetti Cabernet, 99, yes more!
2011, Lake Erie Red, 81, drinkable ...
```

map

```
(2011 {100, Quinta do Noval Port})
(2012 {99, Penfold’s Grange})
(2012 {76, Old Vines Swiller})
(2012 {99, Leonette Cabernet})
(2011 {81, Lake Erie Red})
```

shuffle

```
(2012,   [{99, Penfold’s Grange},
          {76, Old Vines Swiller},
          {99, Leonette Cabernet}] )
(2011,   [{100, Quinta do Noval Port},
          {81, Lake Erie Red}] )
```

reduce

Output: Sorted List

- 2012
  - 99 Penfold’s Grange
  - 99 Leonetti Cabernet
- 2011
  - 100 Quinta do Noval Port
- 2010
  - 98 ...

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Hadoop Example: Word Count

```
public class WordCount {
 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

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

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

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

https://cwiki.apache.org/confluence/display/HADOOP2/wordcount

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

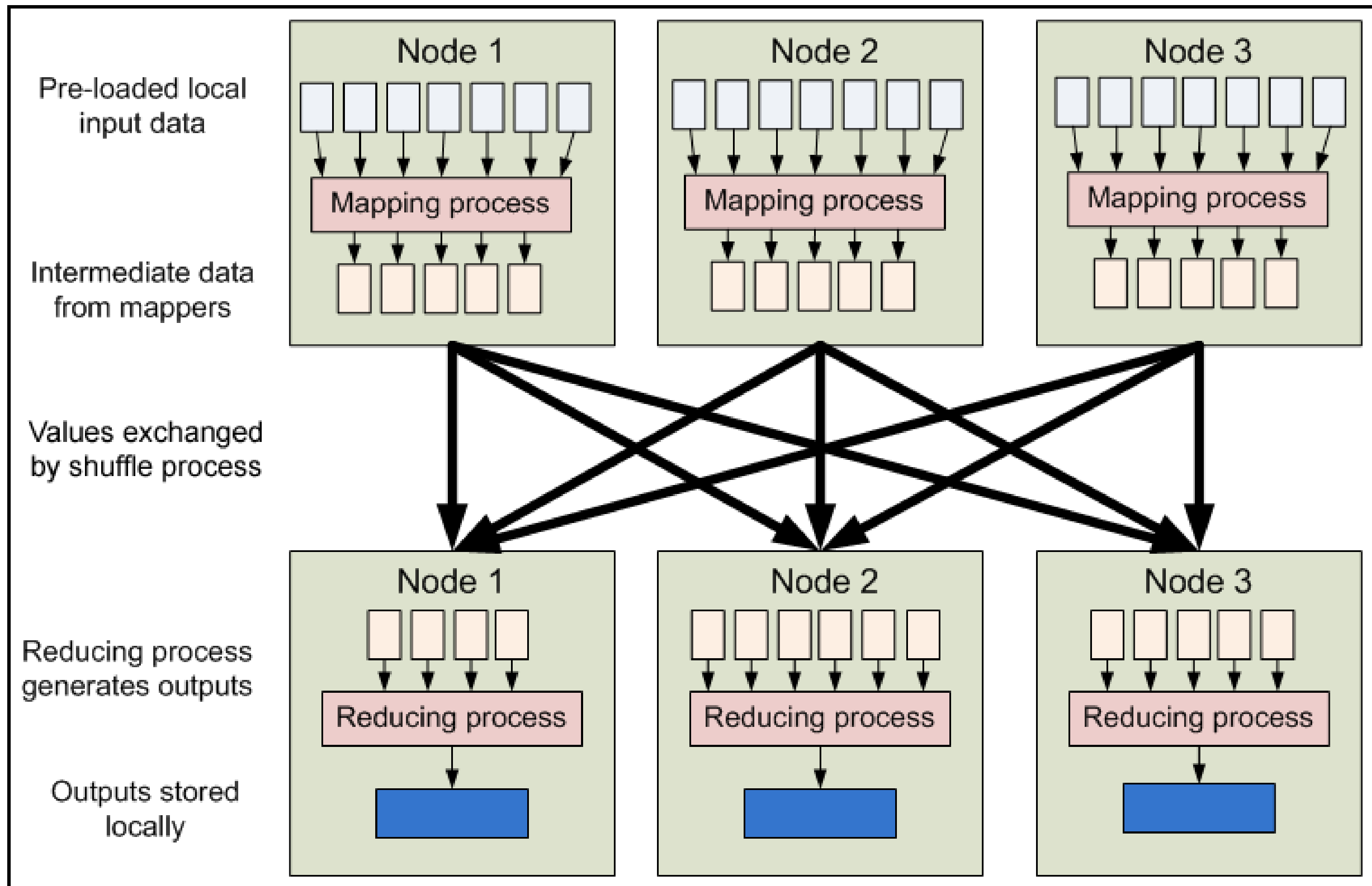
Hadoop Example: Word Count

```
public static void main(String[] args) throws Exception {
   Configuration conf = new Configuration();

   Job job = new Job(conf, "wordcount");

   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);

   job.setMapperClass(Map.class);
   job.setReducerClass(Reduce.class);

   job.setInputFormatClass(TextInputFormat.class);
   job.setOutputFormatClass(TextOutputFormat.class);

   FileInputFormat.addInputPath(job, new Path(args[0]));
   FileOutputFormat.setOutputPath(job, new Path(args[1]));

   job.waitForCompletion(true);
 }

}
```

https://cwiki.apache.org/confluence/display/HADOOP2/wordcount

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

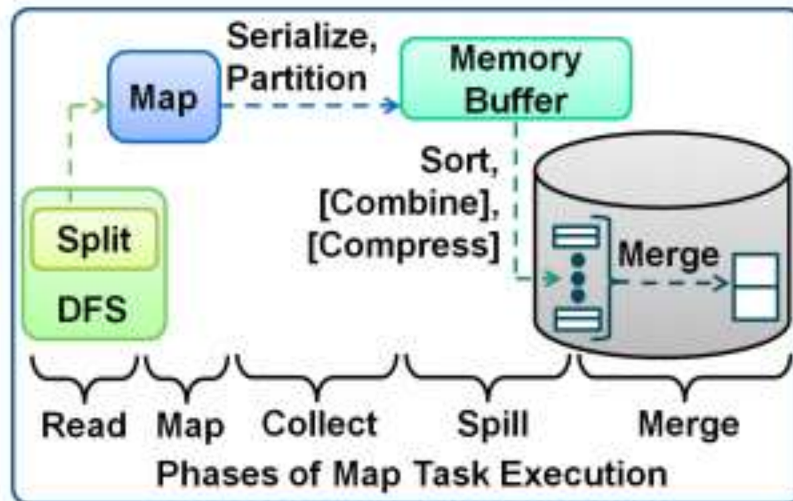
Hadoop – How It Works

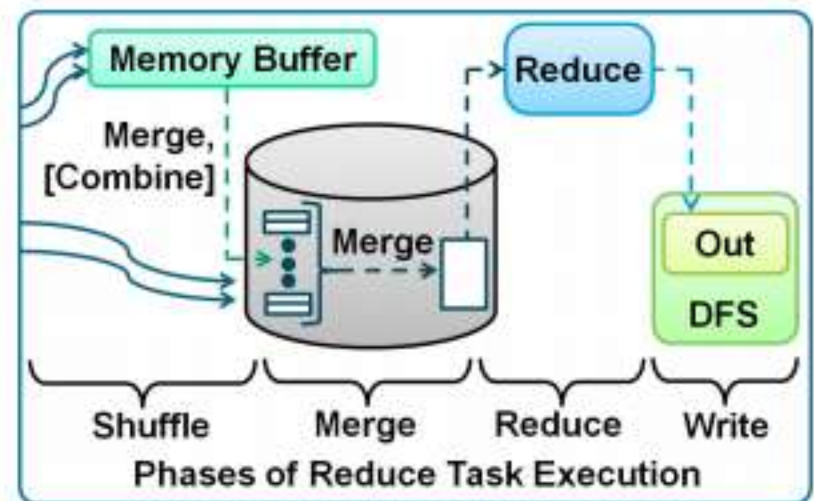


Source:
https://developer.yahoo.com/hadoop/tutorial/module1.html

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Hadoop – Map and Reduce Phases





Source: https://www.cs.duke.edu/starfish/files/hadoop-models.pdf

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Optional Optimization – The Combiner

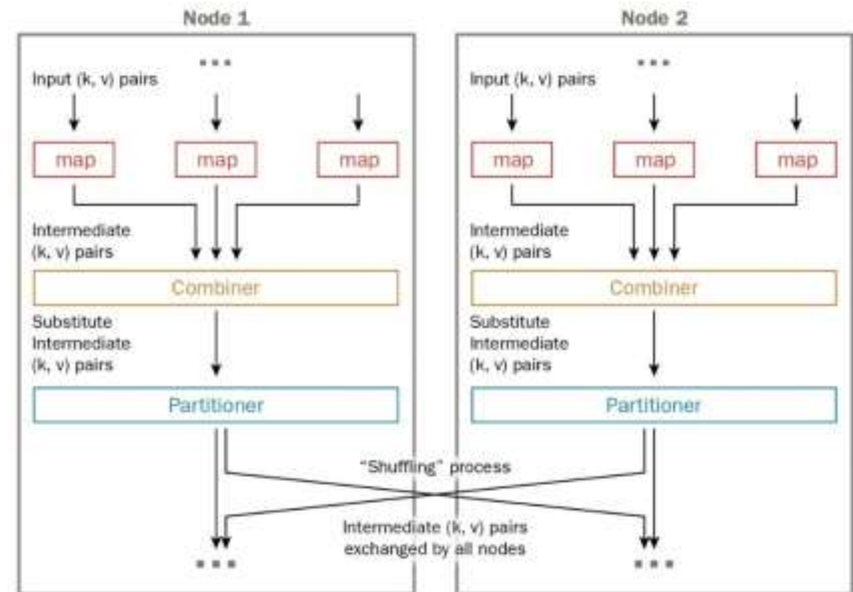
Reduces bandwidth required to shuffle key-value pairs between mappers and reducers

Combiner runs on each mapper node

- Performs a “mini-reduce” step on local outputs
- Outputs from combiner sent to reducers

If Reduce function is both commutative and associative, then it can be used as a Combiner

- conf.setCombinerClass (MyReduce.class)



Source: https://developer.yahoo.com/hadoop/tutorial/module4.html#closer

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Hadoop Distributed File System

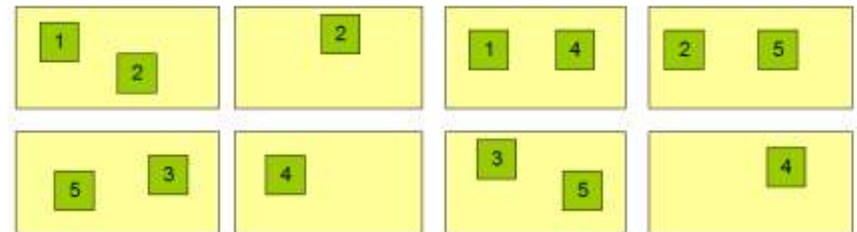
Distributes and replicates data across many nodes

Designed to support long streaming reads from disk

- Transfer rate optimized over seek times
- No local caching of data

Files broken up into fixed-sized blocks (default 64 MB)

- Blocks stored randomly across multiple DataNodes
- NameNode stores file metadata
- Balancer utility to distribute blocks across new nodes added to cluster



Source: http://hadoop.apache.org

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

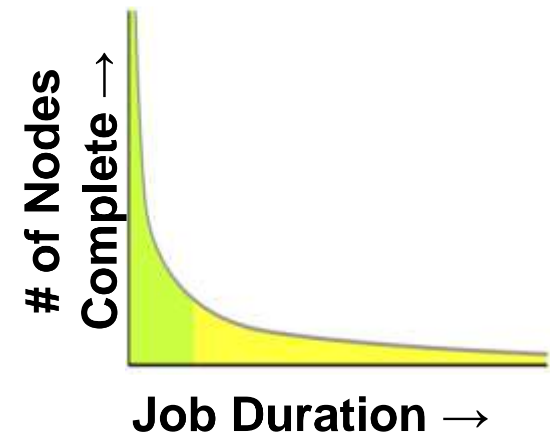
Hadoop Performance Considerations

Running many jobs simultaneously can dramatically lower performance

- Contention for disk access
- HDFS not optimized for seeks

Execution time for map phase determined by slowest mapper

- Many jobs exhibit long tail distributions
- Stragglers
- Need to carefully design data partitions to attempt to evenly distribute work across mappers



Highly configurable behavior

- Over 200 configuration parameters
- ~30 can greatly effect performance

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Aggregation Frameworks

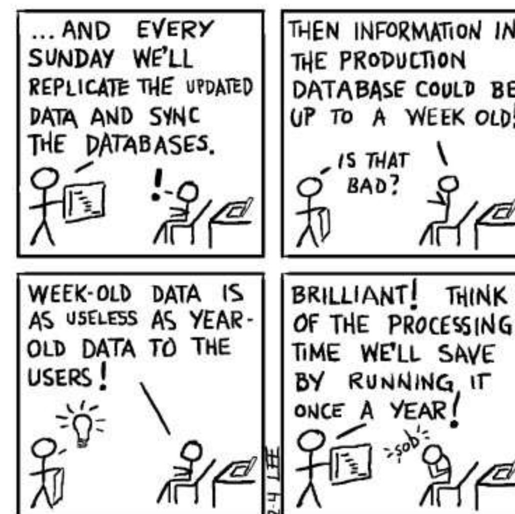
Hadoop is rarely suitable for low-latency queries

If results from aggregate queries are commonly accessed,

- Results can be persisted in the database
- AKA materialized views

Can be updated

- Eagerly
  - When underlying database is updated
  - More reads of view than writes to database
- Periodically
  - Run batch job every N seconds
  - Clients can handle stale results
- Incrementally
  - Incorporate only changed data
  - Problem dependent



© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Further Reading: MapReduce

MongoDB

- Custom aggregation framework
- Limited JavaScript MapReduce support

Riak

- MapReduce HTTP API
- Map phases execute in parallel with data locality
- Reduce phases execute in parallel on the node where the job was

Apache Pig

- High-level procedural language – Pig Latin – for Hadoop

Apache Hive

- Data warehouse functionality for Hadoop
- Stores metadata in an RDBMS
- SQL-like HiveQL for data summation/query/analysis
  - Translated to directed acyclic graph of MapReduce jobs

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Apache Spark (https://spark.apache.org)

General-purpose cluster computing framework

Developed in the AMPLab at UC Berkeley

Apache top-level project in Feb 2014

Java, Scala, Python APIs

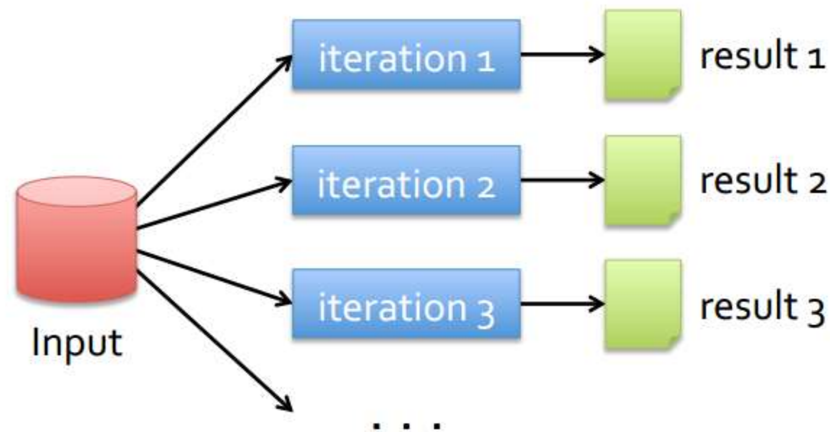
Large (e.g., 10–100x) performance gains over Hadoop for certain types of applications

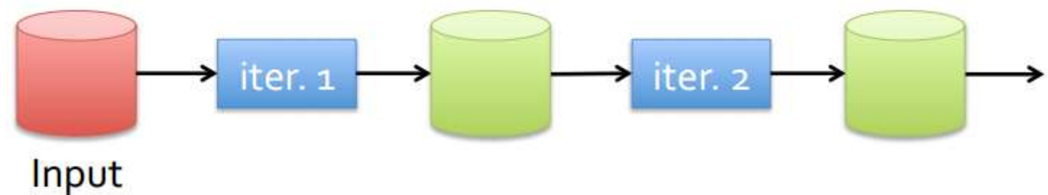
- Repeatedly reuse/share data across a set of operations
  - Machine learning, graph processing
- Interactive data mining

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Iterative Algorithms (e.g., Machine Learning Training)

In Hadoop, large overheads incurred due to reading/writing data to stable storage between iterations





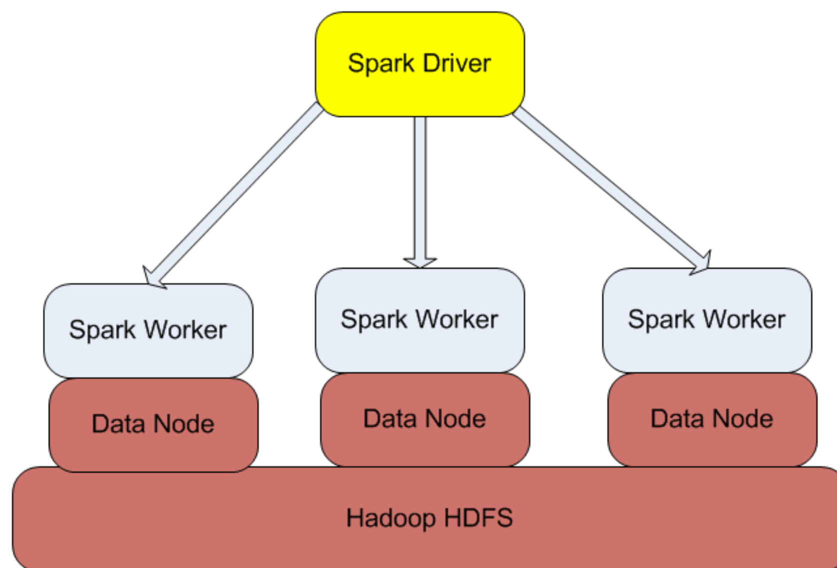
© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Spark Basics

Eliminate overheads of disk accesses by keeping these shared data sets in memory

Spark provides

- Programming model – Resilient Distributed Datasets (RDDs) – that supports coarse-grained operations
- Fault-tolerance model so data loss can be addressed by recomputation of lost data



© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Resilient Distributed Datasets

A data-parallel programming model

for fault-tolerant distributed data sets

- Partitioned collections with controllable caching
- Transformations (define new RDDs), actions (compute results)
- Restricted shared variables (broadcast, accumulators)

Spark distributes the data across slices in a cluster:

- Runs 1 task per slice
- Chooses value automatically (typically 2–4 slices per CPU)
- Or sets in parallelize function

```
JavaSparkContext sc;

//create a parallel data set
//with 5 slices
List<Integer> data =
Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData =
sc.parallelize(data, 5);

//create a text file RDD with
//a slice per HDFS block
JavaRDD<String> distFile =
sc.textFile("data.txt");
```


© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

RDD Operations

Transformations

- Create a new RDD from an existing one
- map, filter, distinct, union, join, repartition, sortByKey, etc.
- Can cache a new RDD for later use

Actions

- Return a result to the driver program
- reduce(func), foreach(func), count, takeSample, etc.

All operations are "lazy"

- Only executed when an action is called to compute a result
- Support optimization of Spark execution

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


© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

RDD Persistence

RDDs can be selectively cached between operations

Various options:

- Leave in memory for fastest option
- Serialize to save space
- Spill to disk only if the data is expensive to compute
- Fault tolerance allows immediate partition recovery for computation
  - All data is fault tolerant in that it can always be recomputed, but with latency costs

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Shared Variables

Functions passed to Spark operate on local copies of their variables

Two specific use cases for sharing variables across functions are supported:

- Broadcast read-only variables to all nodes
- Distribute a large read-only data set to every node

Accumulators can only be “added” to through an associative operation:

- Added in tasks (not read)
- Read in driver

```
Broadcast<int[]> broadcastVar =
sc.broadcast(new int[] {1, 2, 3});
broadcastVar.value();
// returns [1, 2, 3]

Accumulator<Integer> accum =
sc.accumulator(0);
sc.parallelize(Arrays.asList(1, 2,
3, 4)).foreach(x -> accum.add(x));

// wait for tasks to end
accum.value(); // returns 10
```


© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Storm

Distributed computation framework for unbounded streams of data

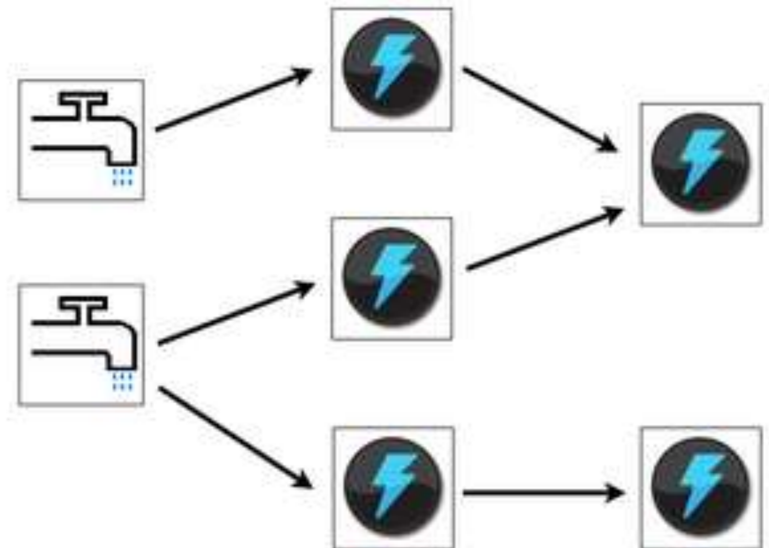
Defines "spouts" and "bolts" for information sources and manipulations that can be organized into arbitrary topologies

Example uses:

- Real-time analytics
- ETL
- Machine learning

Open sourced by Twitter

Apache incubator in 2013



© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Storm Basics

Storm topology runs many workers across many processors

- Workers implement a subset of a topology

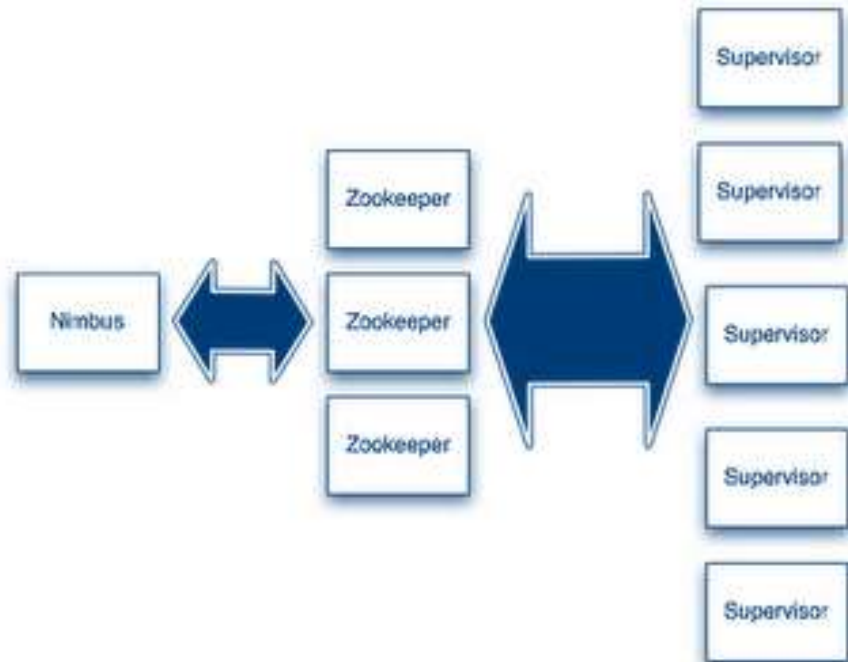
A single Nimbus master daemon assigns tasks to workers, distributes code, and monitors for failures

Supervisors schedule workers when work is assigned to them

ZooKeeper is used to store configuration and connect Nimbus and supervisors

- Storm daemons are fail fast
- State stored on local disk or in ZooKeeper

ZooKeeper is a distributed configuration and synchronization service, and a naming registry for large distributed system



© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Spouts and Bolts

Topologies comprise spouts and bolts

- Implement defined interfaces to run application-specific logic.

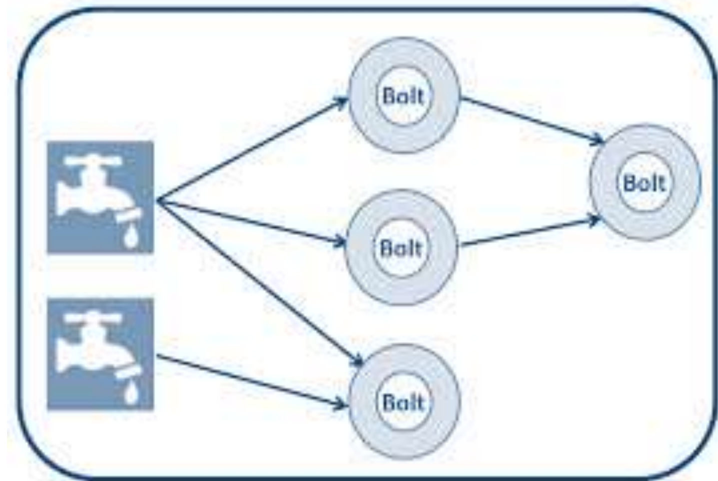
A spout is a source of streams:

- Sensor messages
- Log files
- Databases

A bolt consumes streams, does computation, and may emit a new stream:

- Filtering
- Transformations
- Aggregations

Bolts can be pipelined to perform complex computations



© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

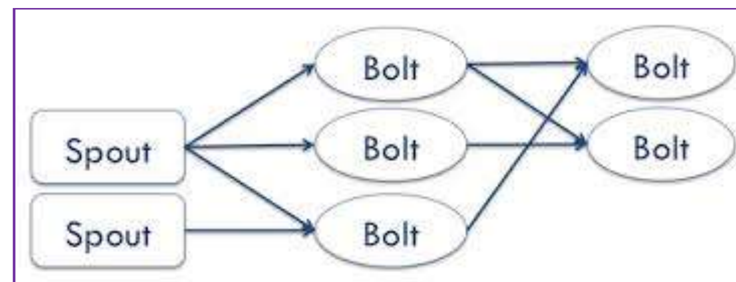
Topologies

A topology is a graph where every node is a spout or bolt

- Data model based on tuples
  - Named list of values
  - Field in a tuple can be an object of any type
- Edges represent a subscription to a stream
- All subscribers receive emitted tuples

In a topology,

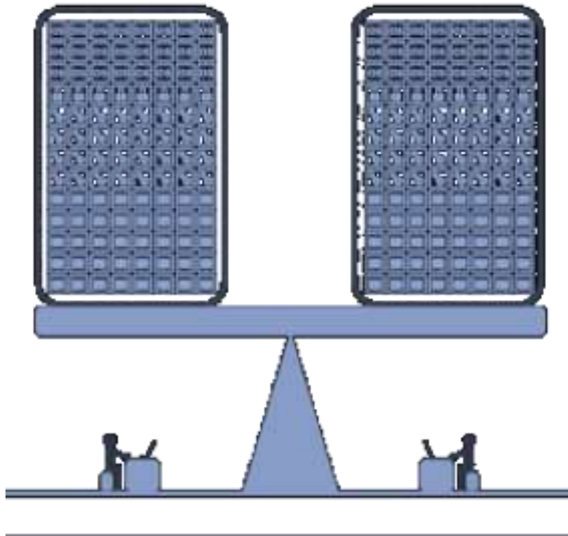
- All nodes run in parallel
- Parallelism is configurable for every node
- A topology never ends
- No data is lost




```
// Bolt code fragment …
public void execute(Tuple input) {
    int val = input.getInteger(0);
    _collector.emit(input, new
Values(val*2, val*3));
    _collector.ack(input);
}
// declare the output fields for
// the tuple
public void
declareOutputFields(OutputFieldsDeclare
r declarer) {
    declarer.declare(new
Fields("double", "triple"));
}
```


© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Scalability – Dimensions



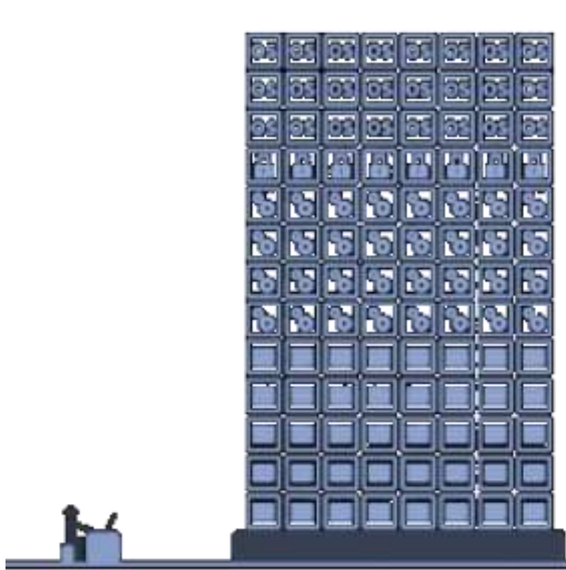
Workload
- # of concurrent sessions and operations
- Operation mix (create, read, update, delete)
- Read – query mix
- Generally, each system use case represents a distinct workload

Data Sets – Volume, Velocity, Variety
- Number of records
- Record size
- Record structure (e.g., sparse records)
- Homogeneity/heterogeneity of structure/schema

Elasticity
- Runtime peaks and valleys – how frequently, how quickly, how much

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

Characterizing Scalability



Who doesn’t want a scalable system? But what does that mean?

Scalability = Dependability at Scale

- Criteria is often “cost increases linearly as <X> increases”

Scalability is a composite quality that includes

- Throughput
- Latency
- Availability
- Consistency
- Security, and more

Need to define scalability for each system

- The challenge is that the qualities are not independent – requirements must account for tradeoffs as system scales
- Quality Attribute Scenarios are one useful tool

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

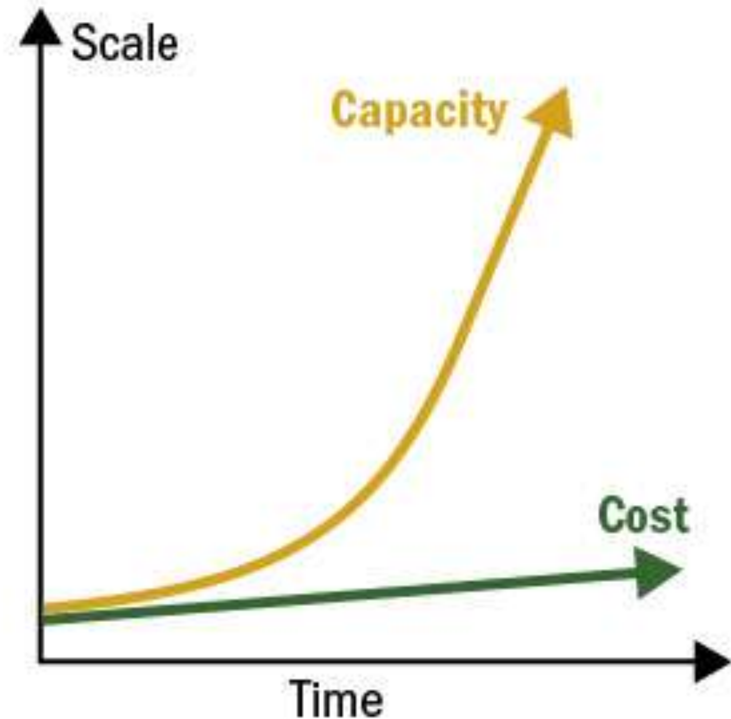
Software Engineering at Scale

Key concept:

- System capacity must scale faster than cost/effort
  - Adopt approaches so that capacity scales faster than the effort needed to support that capacity
  - Scalable systems at predictable costs

Approaches:

- Scalable software architectures
- Scalable software technologies
- Scalable execution platforms



© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.

The “Long Tail” as we add resources

Single Server, p99* latency = 1 sec

- 1 request in 100 takes > 1 sec

Aggregate responses across 100 servers

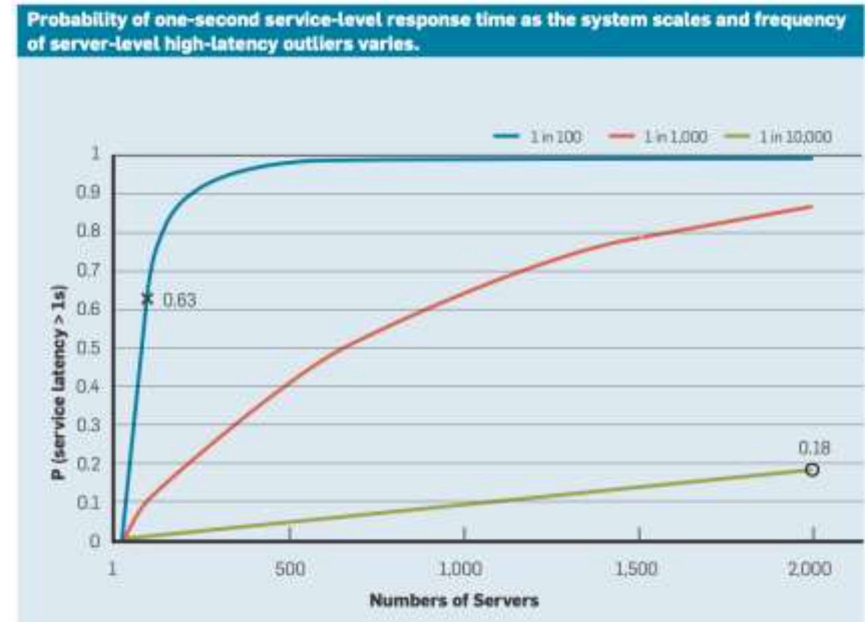
- 63 requests in 100 take > 1 sec

p9999 = 1 sec

- 1 request in 10,000 takes > 1 sec

Aggregate responses across 2,000 servers

- ~1 request in 5 takes > 1 sec



*p99 → 99th percentile

J. Dean and L. A. Barroso, “The Tail at Scale,” *Comm. ACM,* vol. 56, no. 2, pp. 74-80, February 2013. doi: 10.1145/2408776.2408794

© 2021 Carnegie Mellon University
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution.