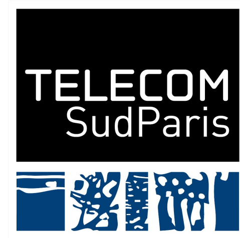


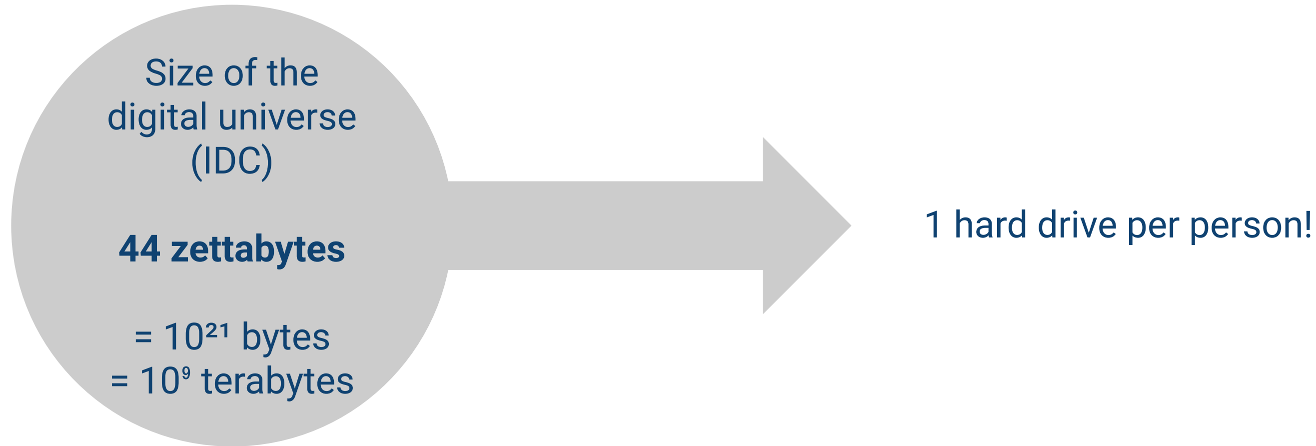
Hadoop And MapReduce

Hadoop Fundamentals

Hadoop Fundamentals

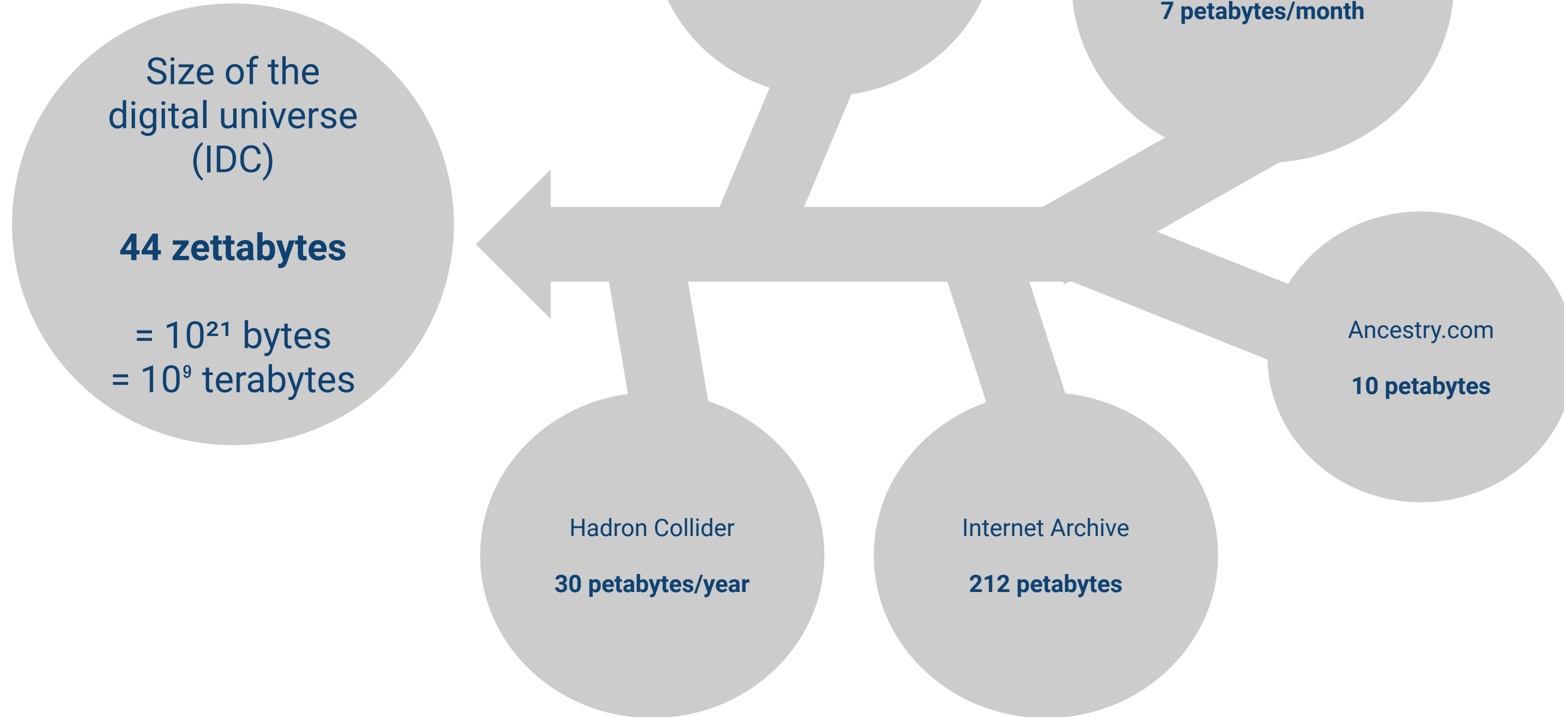
Meet Hadoop

Data!

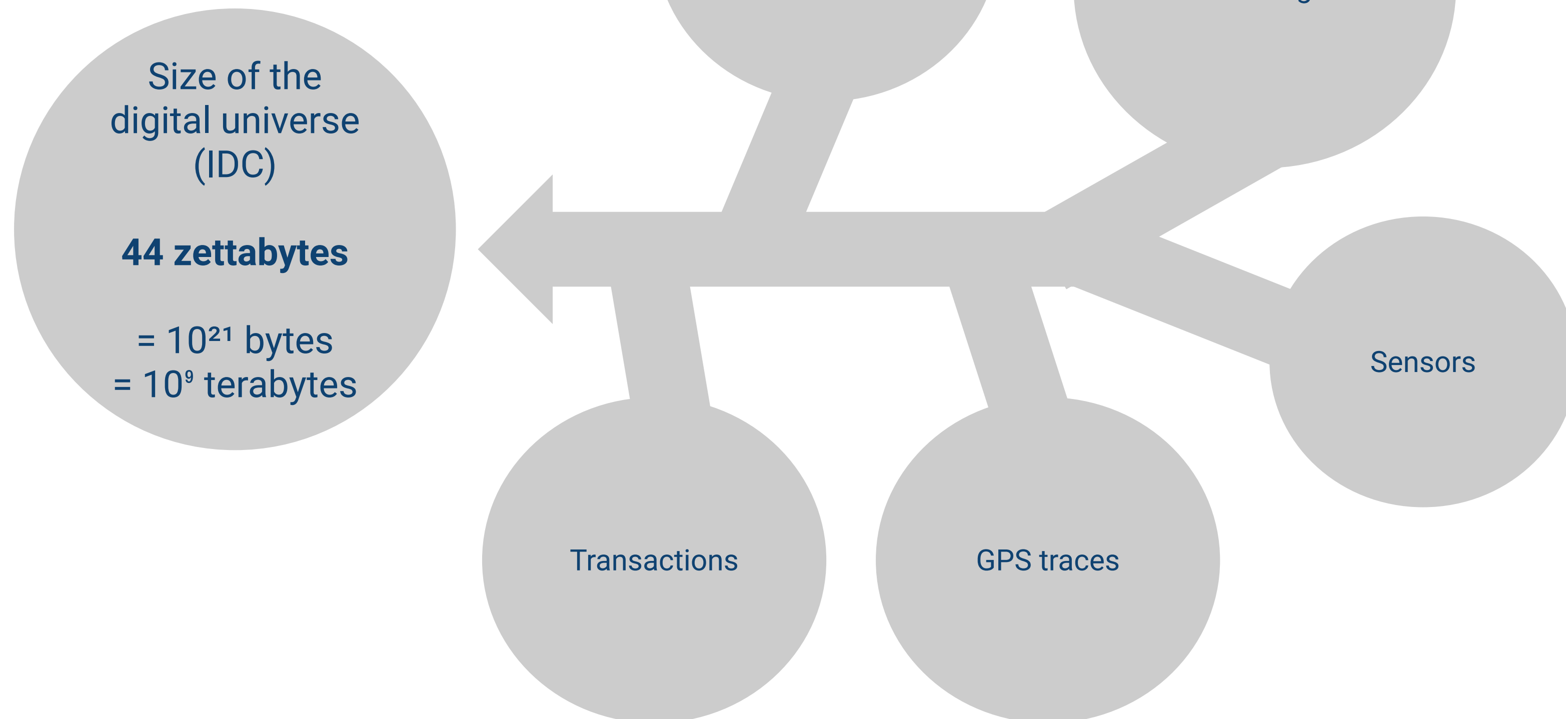


1 hard drive per person!

Data! Big Numbers!



Data! Internet Of Things



Open Data!

yelp.com/dataset

Kaggle

ncdc.noaa.gov

opendata.aws

datasetsearch.research.google.com

data.unicef.org

OPENDATA

opendata.paris.fr

data.gouv.fr

data.worldbank.org

data.gov

data.fivethirtyeight.com

who.int/data/gho

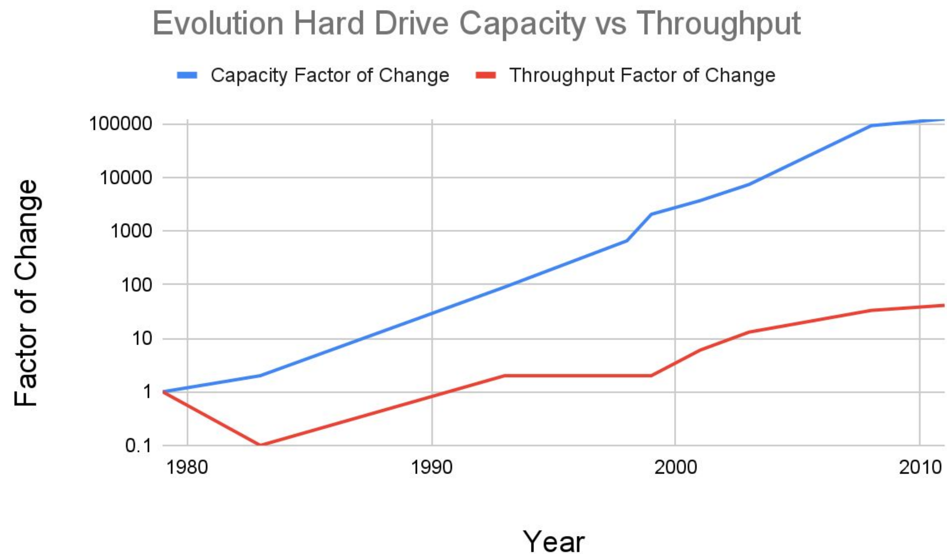
data.europa.eu

“More data usually beats better algorithms”

How to store and exploit this large amount of data?

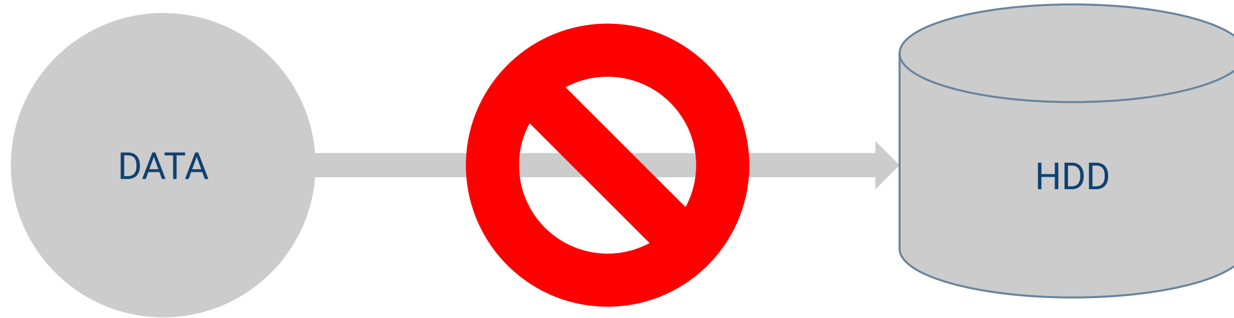
- We need potentially several hundreds of hard drives
- We need to write programs that can be run in parallel, on several computers
- We need to limit data transfers on the network
- The data will be unstructured

Problem With Hard Drives - The Throughput Does Not Change Fast Enough

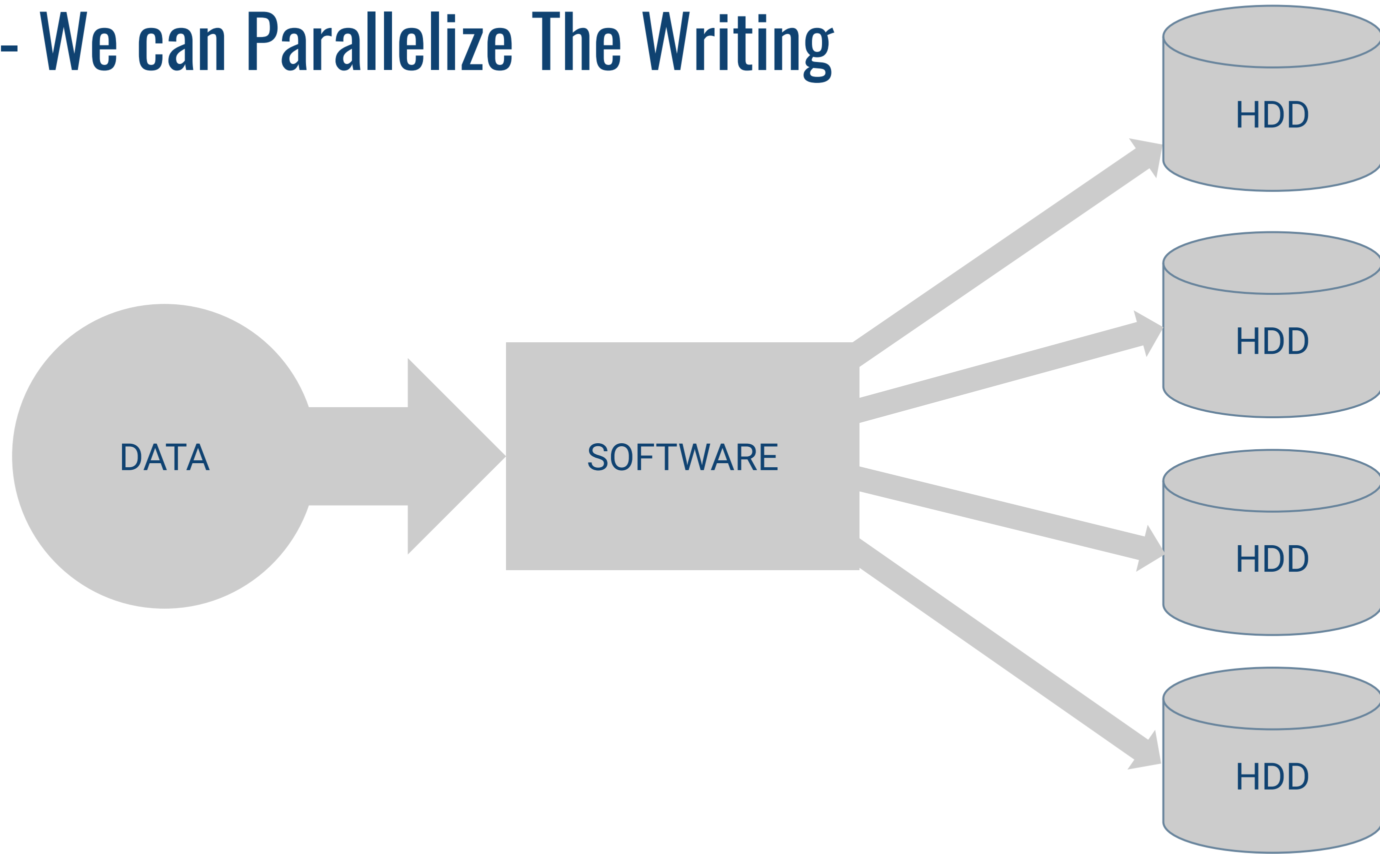


Source: https://tylermuth.wordpress.com/2011/11/02/a-little-hard-drive-history-and-the-big-data-problem/

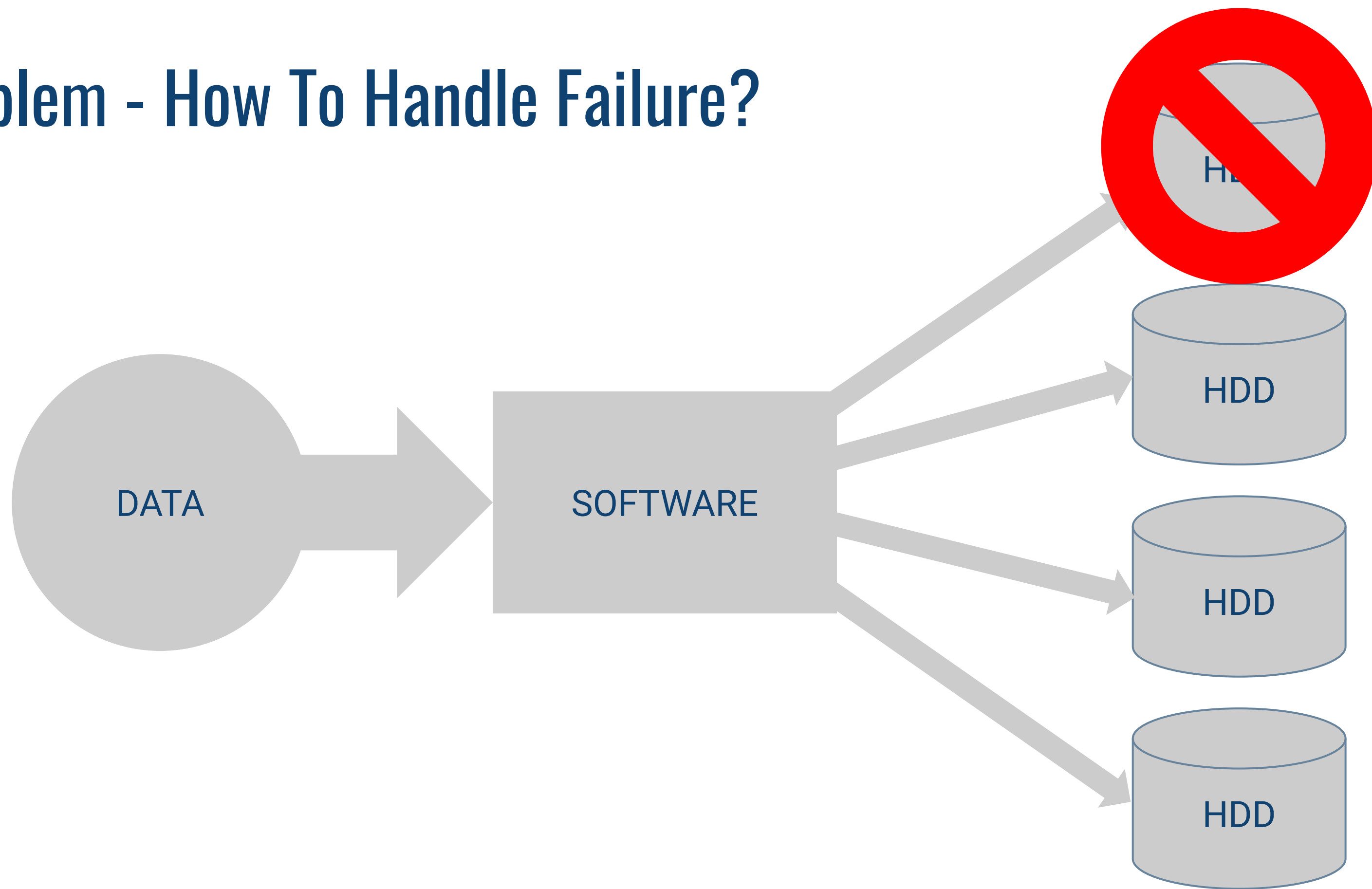
Consequence: Moving The Data Is Expensive



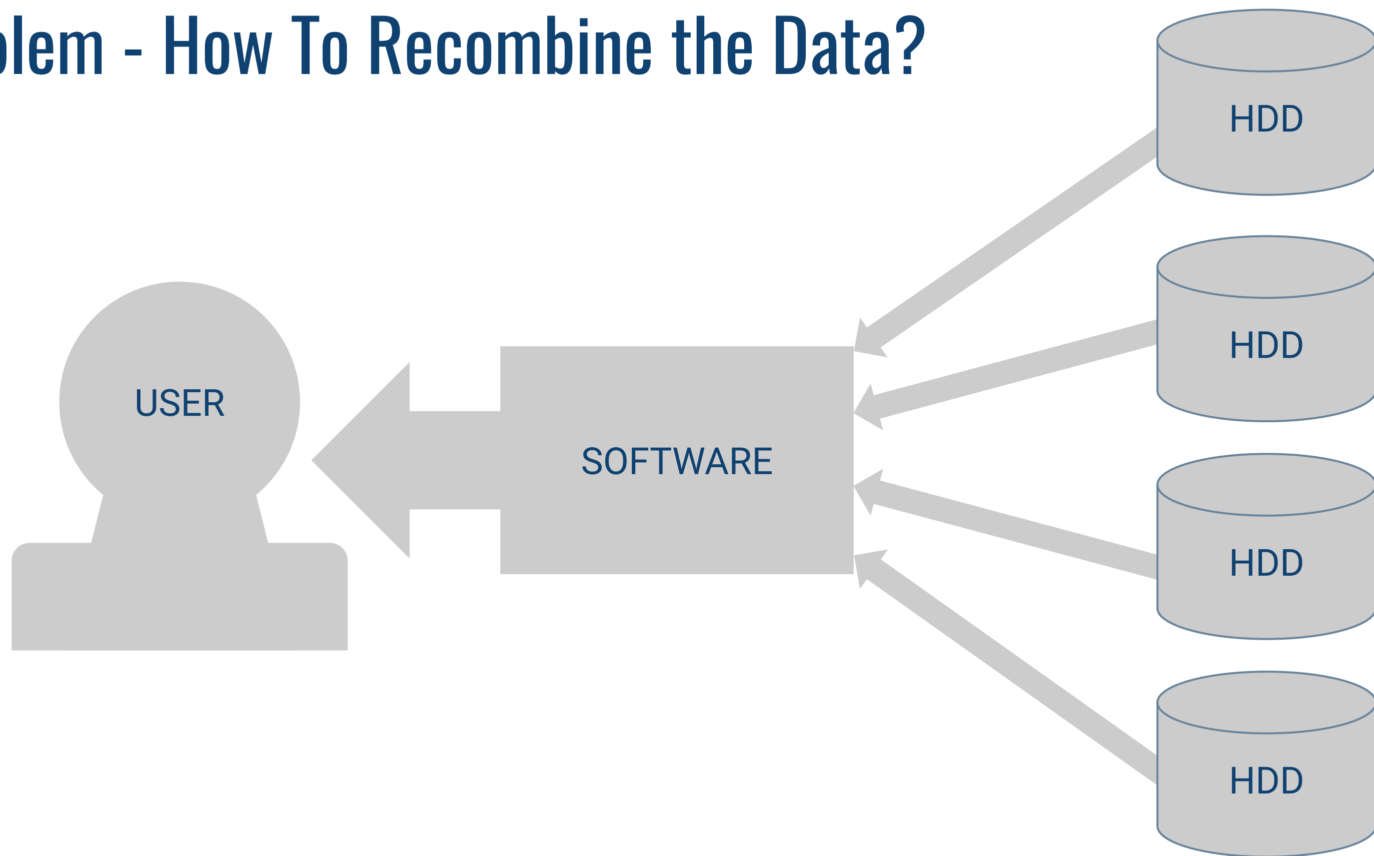
Solution - We can Parallelize The Writing



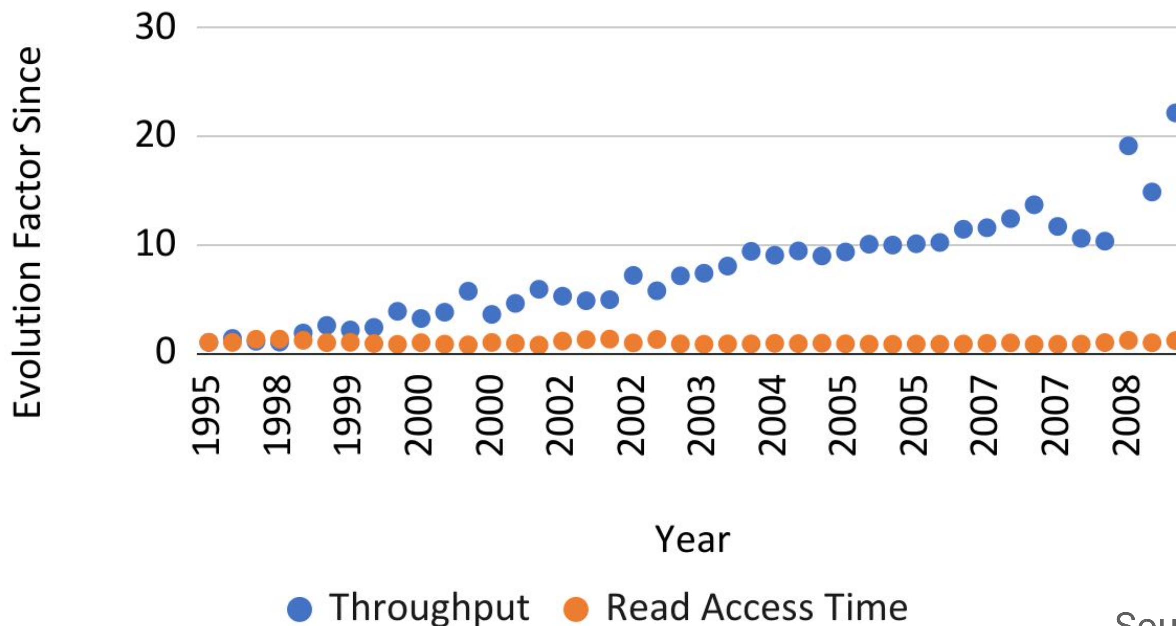
New Problem - How To Handle Failure?



New Problem - How To Recombine the Data?



The "Seek" Problem - Accessing Random Data Is Expensive

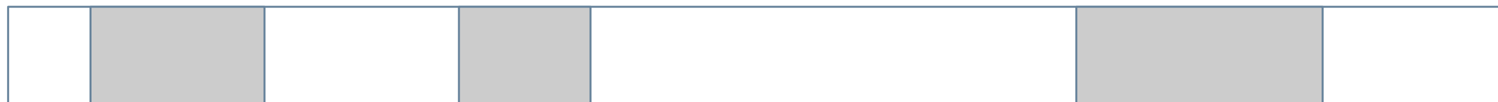


Source: goughlui.com

Reminder - Blocks

- A file is not stored continuously in the memory, it is split into several *blocks*
  - It makes the storage more flexible: We might not be able to store several Gb continuously, but we might have a lot of “holes” in the memory

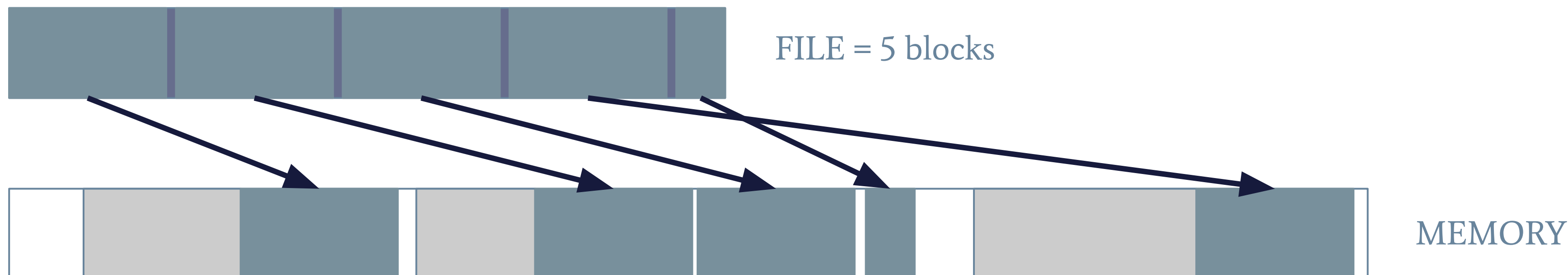
FILE



MEMORY

Reminder - Blocks

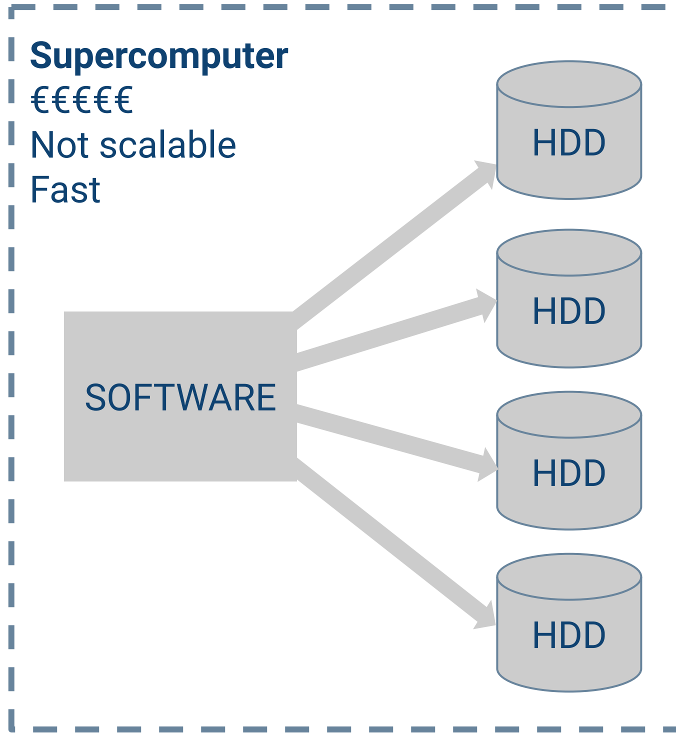
- A file is not stored continuously in the memory, it is split into several *blocks*
    - It makes the storage more flexible: We might not be able to store several Gb continuously, but we might have a lot of “holes” in the memory



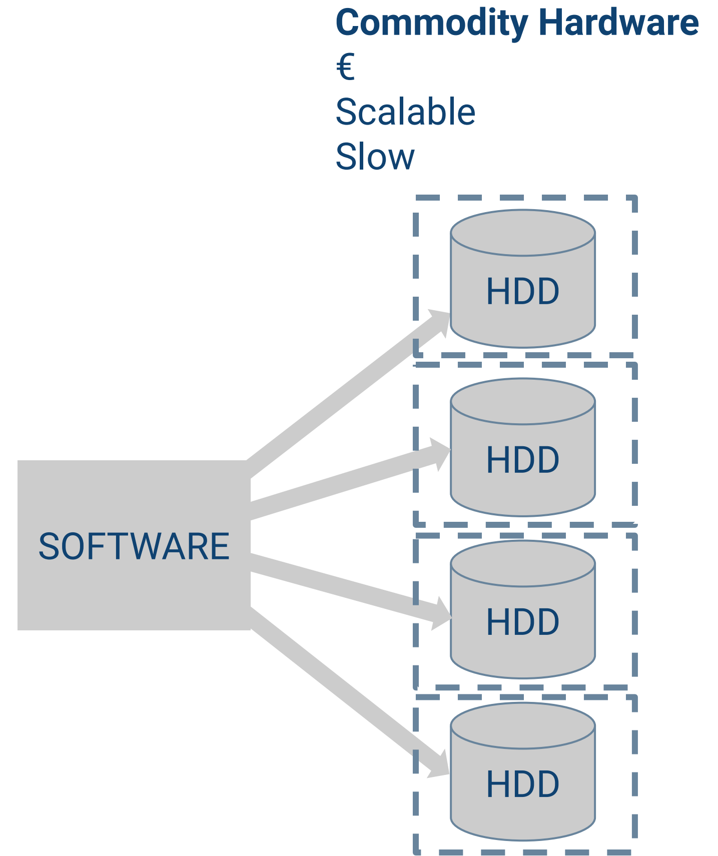
The “Seek” Problem - Too Many Small Blocks Causes Overhead

- Unix blocks are generally of a few Kb
  - With a throughput of 100Mb/s = 100kb/ms, transferring a block of 4kb takes 0.04ms. In comparison, a typical seek time is 10ms.
  - For a file of 1Tb, a block size of 4Kb, and a seek time of 10ms, the worst total seek time for reading the file is the number of blocks ($3*10^8$) time the seek time = 1 month.
  - In practice, this time is much smaller as the OS tries not to fragment a file too much by storing blocks continuously in the memory. This is particularly true for large files.
- A system to store big files must have a large block size

What Kind of Computers Do We Need?



OR



Different Applications: Grid Computing/HPC vs Hadoop

HPC	Hadoop
Supercomputer = Compute-intensive jobs	Commodity hardware = simple computations
Data accessed through a shared file system, hosted by a Storage Area Network	Data locality = data with compute nodes Network bandwidth is critical
More control (data flow, low level C) More complex (failure handling, recovery)	High level Easy to use

Different Processing Methods: Batch vs Online Processing

Online/Interactive Processing	Batch Processing
Process one datapoint at the time	Process all data at once
Relatively simple computation on few data	Large data and complex computation
Latency in seconds or less	Latency in minutes or more (priority = throughput)
Complex and expensive hardware	Simple to implement and inexpensive hardware
E.g.: Fraud detection, bank ATM, customer service, radar systems, …	E.g.: Payroll and billing systems, log analysis, Web exploration, …

Hadoop Fundamentals

Map Reduce

MapReduce - Motivation

- In August 2022, Common crawl collected 2.6 billion pages, for a total of 300 TB
  - We need 300 hard drives per month to store it (less if we compress the data)
- With a hard drive reading at 80Mb/s, it takes 1.5 months to read the data sequentially
- With a connection at 10Gbps, it requires 3 days to transfer everything
- So we need:
  - To store the data on several computers to parallelize the reading time
  - To reduce the data transfers by making the computation where the data is

What Is MapReduce?

MapReduce is a programming model in which the programmer must implement two functions:

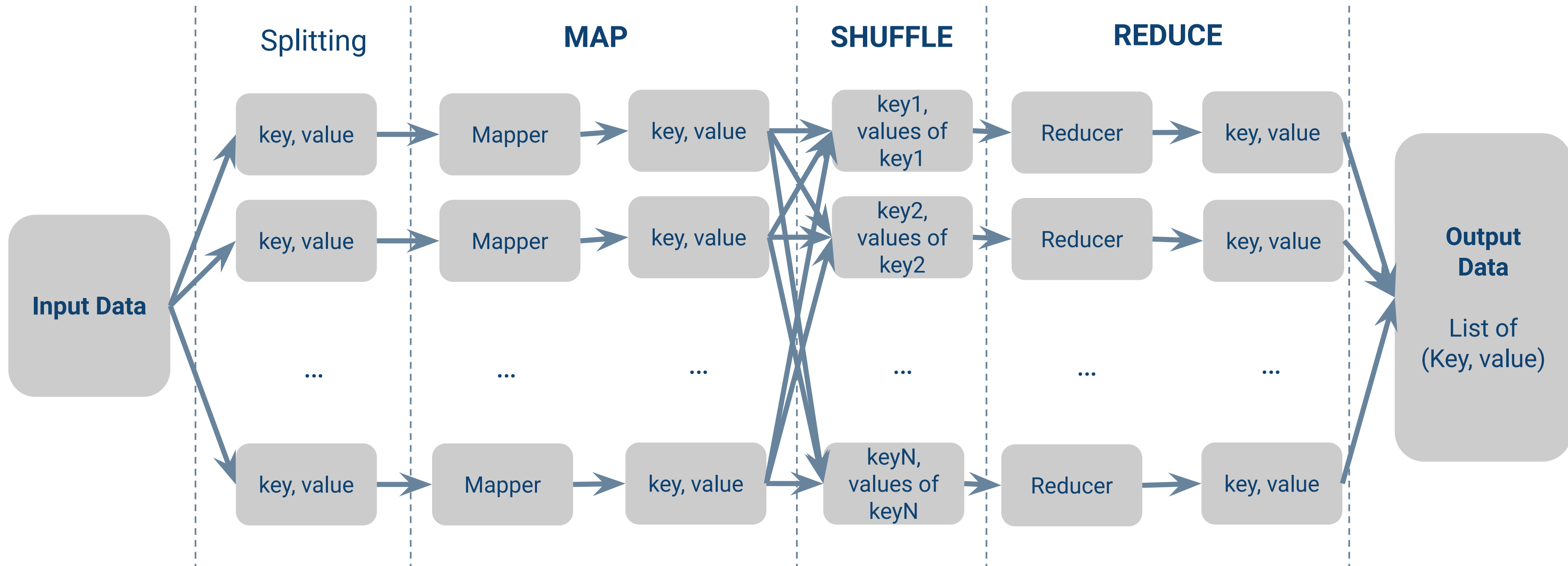
- A map function that transforms a single data point
- A reduce function that summarizes a group of similar data points

The mapper and the reducer will be executed on several machines.

Writing a MapReduce program forces to rethink our algorithm and use a more functional approach (c.f. Scala)

Example of MapReduce program: WordCount. It computes the frequency of each work in a file;

MapReduce - Overview



MapReduce - Splitting

- The input of MapReduce is a list of (key, value)
  - In general, at the beginning, we just have a meaningless identifier and a value
- E.g.: Word count: (id, sentence)
- The step is already implemented by the frameworks

MapReduce - Map

- The Map is a function that transforms the input.
  - Input: a (key, value) pair
  - Output: zero, one or several (key, value) pair(s) (not necessary the same)
- E.g.: Word count
  - Input: (0, “the cat eats the food”)
  - Output: (“the”, 2), (“cat”, 1), (“eats”, 1), (“food”, 2)
- **WARNING:** All the map functions are the same and do not know about other data except the one given as input.

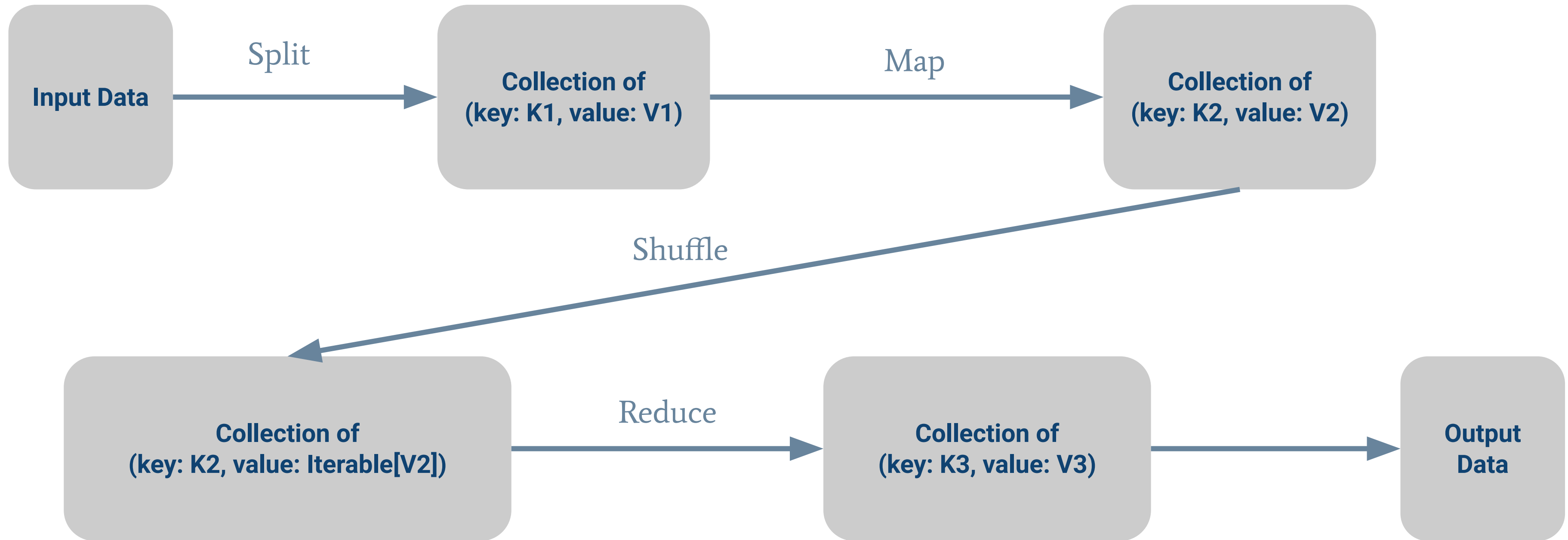
MapReduce - Shuffle

- The shuffle performs a group by key
  - Input: List of (key, value)
  - Output: List of (key, list of values) where the keys are all different
- It is already implemented by most frameworks
- E.g. Word count:
  - Input: (“the”, 2), (“cat”, 1), (“eats”, 1), (“food”, 1), (“the”, 1), (“cat”, 1), (“sleeps”, 1)
  - Output: (“the”, [2, 1]), (“cat”, [1, 2]), (“eats”, [1]), (“food”, [1]), (“sleeps”, [1])

MapReduce - Reduce

- A reduce function transforms the values associated to a key into a single value
  - Input: (key, list of values)
  - Output: (key, value)
- E.g.: Word count
  - Input (“the”, [2, 1])
  - Output (“the”, 3)

MapReduce Summary



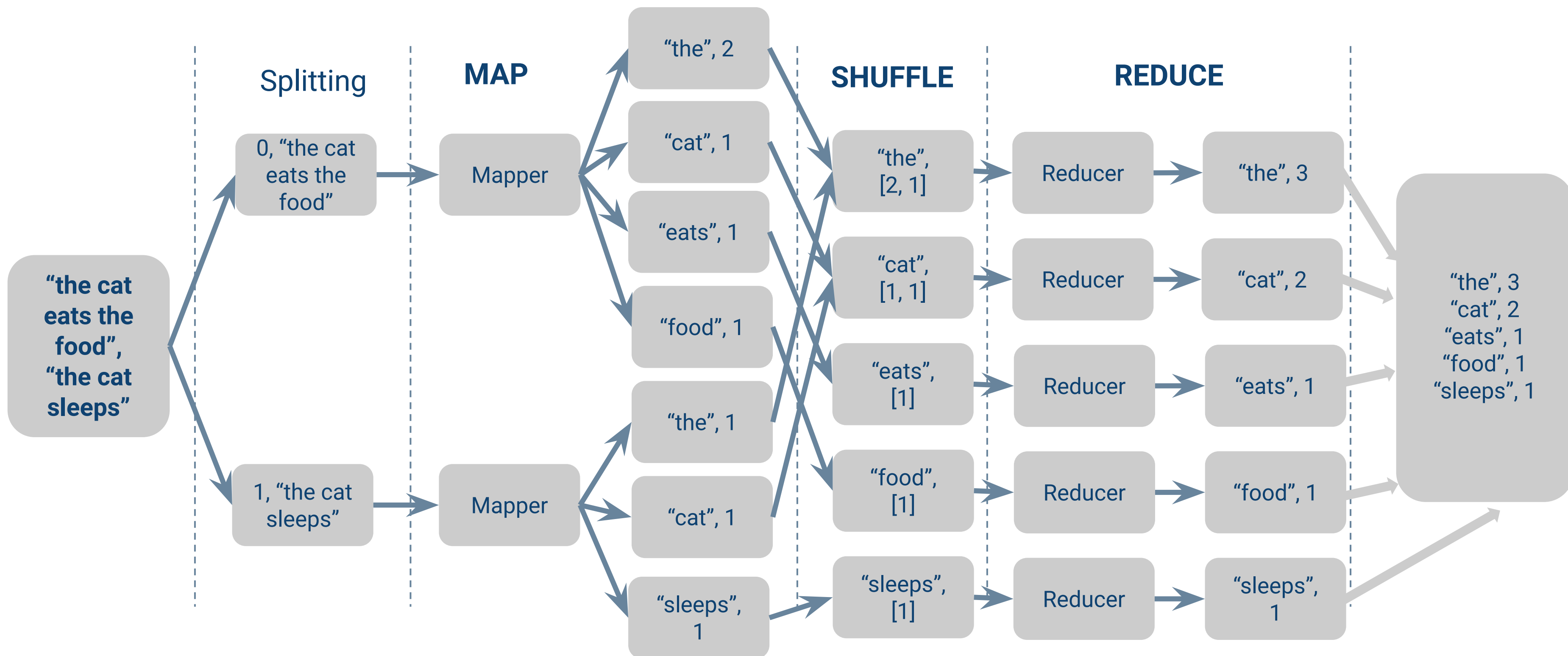
MapReduce Summary - Equivalent In Scala - WordCount

```
val inputData = List((0, "The cat eats the grass"), (1, "The cat sleeps on
the bed"))

inputData.map((x, y) => y.split(" ").map( z => (z, 1))) // Mapper
  .flatten.groupBy( x => x._1).map((x, y) => (x, y.map(z => z._2))) //
Shuffle
  .map((x, y) => (x, y.reduce(_ + _))) // Reducer


val res: Map[String, Int] = HashMap(grass -> 1, bed -> 1, The -> 2, on -> 1,
cat -> 2, sleeps -> 1, eats -> 1, the -> 2)
```

MapReduce - Word Count



MapReduce - Word Count Pseudocode

- The programmer needs to provide two functions: **map** and **reduce**

```
function map(int id, String text):
    List[String] words = text.split(" ")
    for word in words:
        emit(word, 1)
```

```
function reduce(String word, List[int] counts):
    emit(word, sum(counts))
```

MapReduce - Exercice 1

- Given a file with integers, find the **maximum** value
- We suppose the splitter splits the document into list of integers

MapReduce - Exercice 1

- Given a file with integers, find the **maximum** value
- We suppose the splitter splits the document into list of integers

```
function map(int id, List[int] integers):
        emit(1, max(integers))
```

```
function reduce(int id, List[int] maxima):
    emit(id, max(maxima))
```

MapReduce - Exercice 2

- Given a file with integers, find the **mean** value
- We suppose the splitter splits the document into list of integers

MapReduce - Exercice 2

- Given a file with integers, find the **mean** value
- We suppose the splitter splits the document into list of integers

```
function map(int id, List[int] integers):
    emit(1, (len(integers), sum(integers)))
```

```
function reduce(int id, List[(int, int)] values):
  res = 0
  sum_weights = 0
  for weight, total in values:
    res += total
    sum_weights += weight
  emit(id, res / sum_weights)
```

MapReduce - Exercice 3

- Given a file with integers, find the **set of unique integers**
- We suppose the splitter splits the document into list of integers

MapReduce - Exercice 3

- Given a file with integers, find the **set of unique integers**
- We suppose the splitter splits the document into list of integers

```
function map(int id, List[int] integers):
    for integer in integers:
        emit(integer, 1)
```

```
function reduce(int id, List[int] values):
    emit(id, 1)
```

MapReduce - Exercice 4

- Given a file with integers, find the **number of unique integers**
- We suppose the splitter splits the document into list of integers

MapReduce - Exercice 4

- Given a file with integers, find the **number of unique integers**
- We suppose the splitter splits the document into list of integers

```
function map1(int id, List[int] integers):
    for integer in integers:
        emit(integer, 1)
```

```
function reduce1(int id, List[int] values):
    emit(1, 1)
```

```
function map2(int id, int integer):
    emit(id, integer)
```

```
function reduce2(int id, List[int] values):
    emit(1, sum(values))
```

MapReduce - Exercice 5

- Given a matrix **M** of size n*n and a vector **v** for size n, compute Mv (n is large!)
- We suppose that **M** is stored as ((i, j), **M**[i,j]) and that **v** fits in the memory and is a global variable.

$$x_i = \sum_{j=1}^{n} M[i,j] \cdot v_j$$

MapReduce - Exercice 5

- Given a matrix **M** of size n*n and a vector **v** for size n, compute Mv (n is large!)
- We suppose that **M** is stored as ((i, j), **M**[i,j]) and that **v** fits in the memory and is a global variable.

$$x_i = \sum_{j=1}^{n} M[i,j] \cdot v_j$$

```
function map((int, int) pos, float value):
    emit(pos[0], value * v[pos[1]])
```

```
function reduce(int i, List[float] values):
    emit(i, sum(values))
```

MapReduce - Exercice 6

- Given a matrix **M** of size l*m and a matrix **N** of size m*n, compute **MN**
- We suppose that **M** is stored as ((“M”, i, j), **M**[i,j]) and **N** as ((“N”, i, j) **N**[i, j])

$$p[i,\, k] = \sum_{j=1}^{m} M[i,\, j] \cdot N[j,\, k]$$

MapReduce - Exercice 6

- Given a matrix **M** of size l*m and a matrix **N** of size m*n, compute **MN**
- We suppose that **M** is stored as ((“M”, i, j), **M**[i,j]) and **N** as ((“N”, i, j) **N**[i, j])

$$p[i,\, k] = \sum_{j=1}^{m} M[i, j] \cdot N[j,\, k]$$

```
function map1((String, int, int) pos, float value):
  if pos[0] == “M”:
    emit(pos[2], (pos[0], pos[1], value))
  else:
    emit(pos[1], (pos[0], pos[2], value))
```

```
function map2((int, int) pos, float value):
  emit(pos, value)
```

```
function reduce1(int j, List[(String, int, float)] values):
  M_values = [(value[1], value[2]) for value in values if value[0] == “M”]
  N_values = [(value[1], value[2]) for value in values if value[0] == “N”]
  for (i, m) in M_values:
    for (k, n) in N_values:
      emit((i, k), m*n)
```

```
function reduce2((int, int) pos, List[float] values):
  emit(pos, sum(values))
```

When To Use MapReduce

- With true “big” data (>terabytes)
  - Not so frequent: Microsoft/Yahoo! median job size is under 14G, 90% of Facebook jobs are < 100G
  - https://adamdrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html
- Do not require fast response time
  - Often used for precomputing
- Application compatible with batches
  - No human interaction, process all data, no real-time data (sensors), no random access
- Can express data with key/value pairs
  - E.g.: No graph data
- Algorithms do not require interdependence between data points
  - E.g.: Comparisons, graph algorithms, some machine learning algorithms

RDBMS vs MapReduce

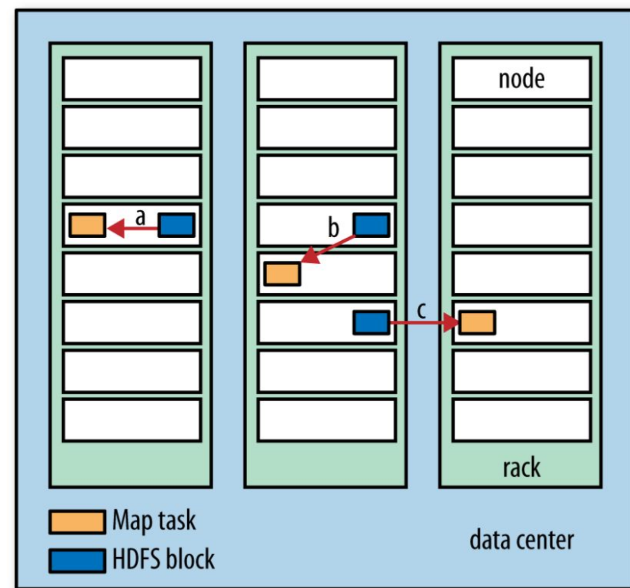
	RDBMS	MapReduce
Data Size	Gigabytes	Petabytes
Access	Interactive/Batch	Batch
Update	Read/Write many times	Write only, Read many times
Transactions	ACID	None
Structure	Schema-on-write	Schema-on-read
Integrity	High	Low
Scaling	Nonlinear	Linear

MapReduce In Hadoop

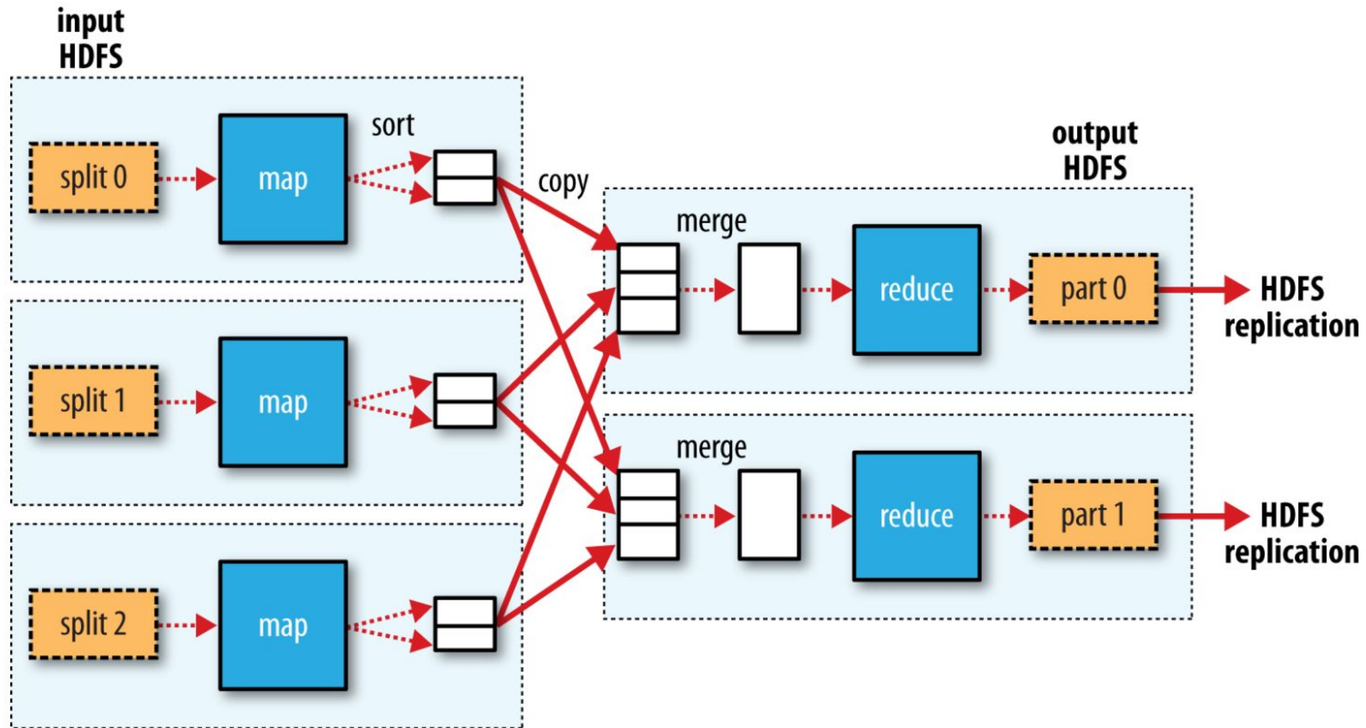
- The unit of work is called a *job*
- A job is decomposed into *tasks*: *map tasks* and *reduce tasks*
- The tasks are scheduled by *YARN* on the nodes of the cluster
- The input file is splitted in *input splits* (or just *splits*)
  - One map task for each split
  - Apply the map function on each *record*
- Tradeoff for the splits sizes:
  - Small = more jobs, overhead of managing
  - Big = less parallelism
  - In general, HDFS block, 128 MB (maximum size guaranteed on a single node)

MapReduce In Hadoop - Data Locality Optimization

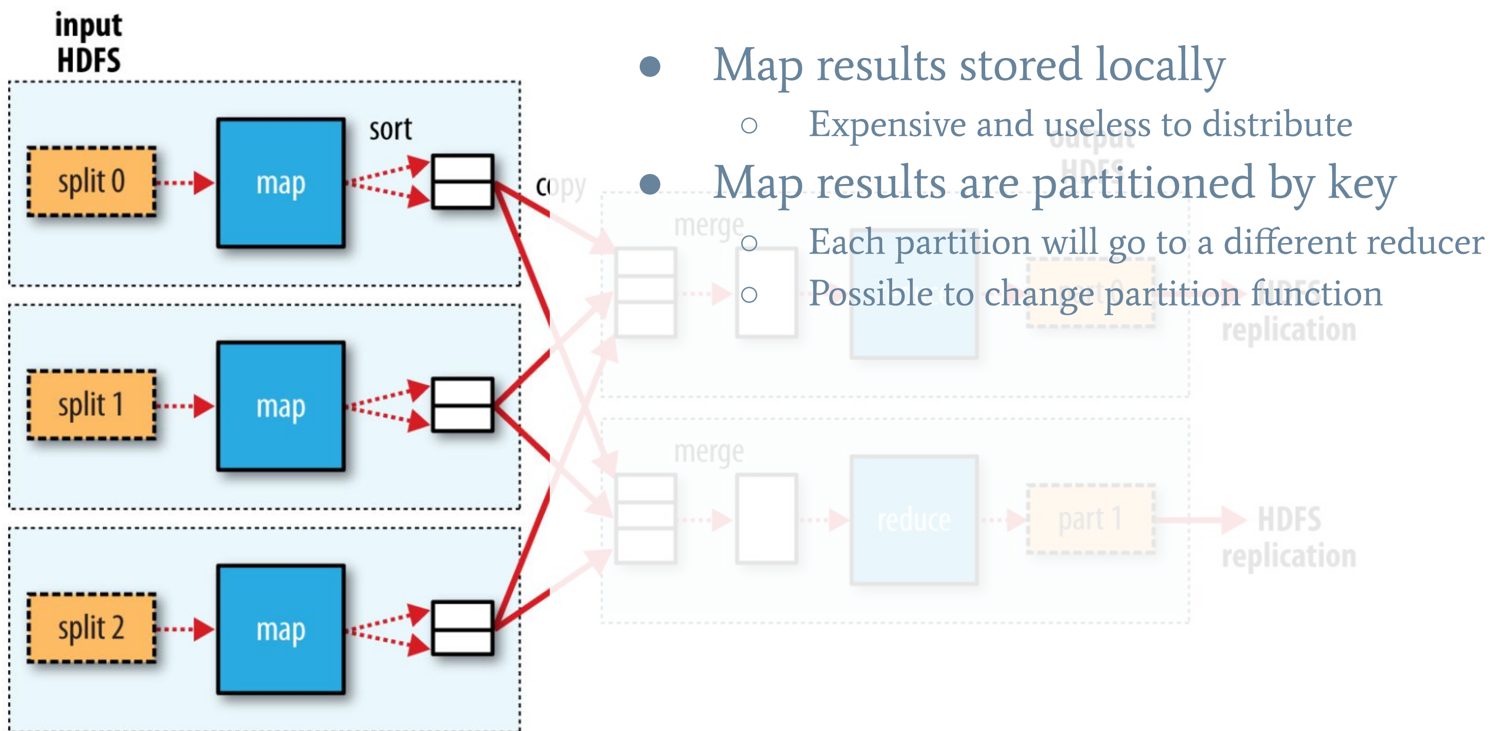
- Try to run the map task on the node where the data is.
  - If not possible, run it on a node on the same rack
  - If not possible, run it where you can



MapReduce In Hadoop - Data Flow



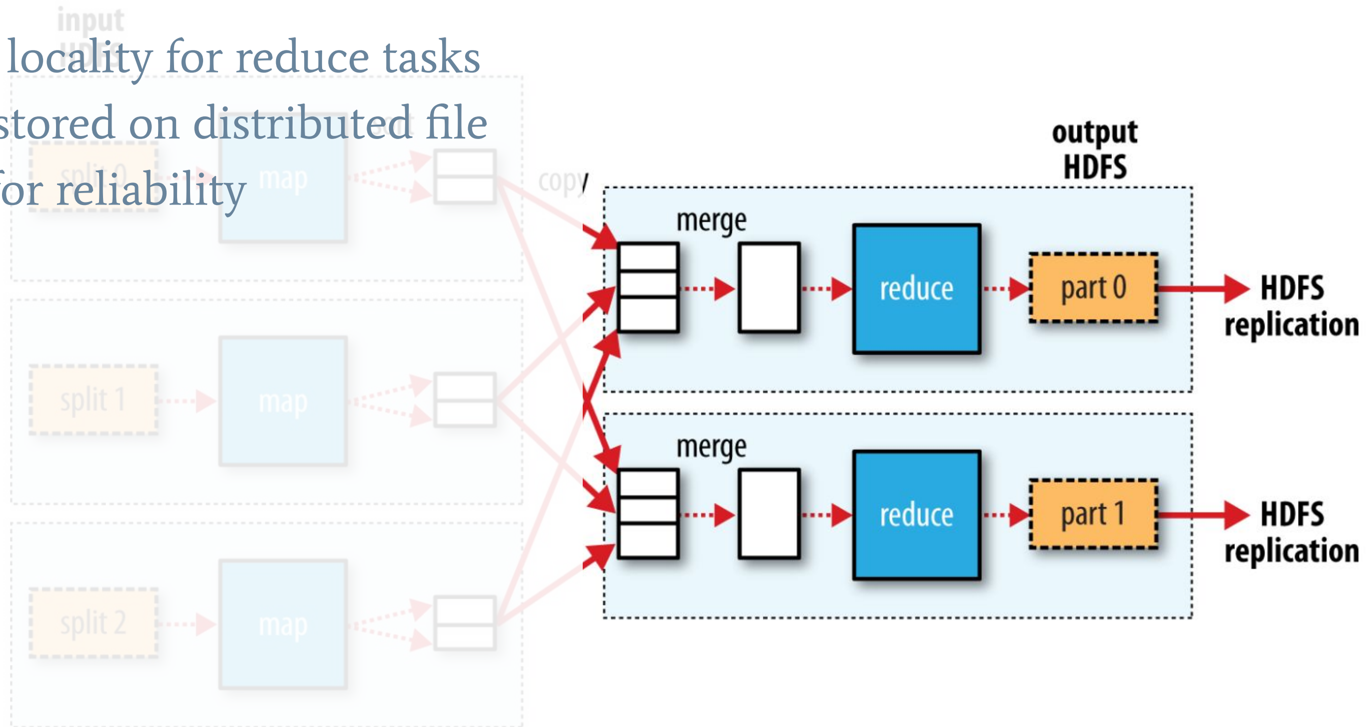
MapReduce In Hadoop - Data Flow



- Map results stored locally
  - Expensive and useless to distribute
- Map results are partitioned by key
  - Each partition will go to a different reducer
  - Possible to change partition function

MapReduce In Hadoop - Data Flow

- No data locality for reduce tasks
- Results stored on distributed file system for reliability



MapReduce In Hadoop - The Combiner Function Optimization

- If the reduce function is associative and commutative, it can be executed after the map task and before the split
  - We reduce the data to transfer (throughput is critical)
  - Ok: Max, Min, sum
  - Not Ok: Mean, median
- No guarantee that the combiner will be executed
  - The program should work the same!

Hadoop Fundamentals

Hadoop Distributed File System

Hadoop Distributed File System

The HDFS is a distributed file system designed for:

- Managing very large files (several gigabytes)
- Streaming data access: files are written once, read several time, often entirely
- Running on commodity hardware: failure tolerant

It does not work well for:

- Low-latency data access
- Lot of small files
- Multiple writer
- File modification (except append)

Hadoop Concepts - Blocks

Blocks: Minimum of data that can be read or written.

- Similar to Linux blocks (CSC3102) but:
  - Much bigger (128MB vs a few KB)
  - If data smaller that a block, does not take entire block
- Block size = tradeoff seek time/transfer time
  - If seek time = 10ms and transfer rate = 100MB/s, to have seek time = 1% total time we need a block size of 100MB
- Advantages:
  - Can split a file on several disks
  - Simple for the filesystem
  - Good for replication

Hadoop Concepts - Namenodes/Datanodes

- Master-slave pattern:
  - Namenode manages the filesystem (master)
  - Datanodes store the blocks
- How to make the namenode resilient to failure?
  - Back up the files (e.g., NFS)
  - Secondary namenode that follows the namenode (faster but lags behind)

Hadoop Concepts - Others

- **Block caching**: take advantage of blocks already in RAM
- **HDFS federation**: if too many files, the filesystem might not fit into a single namenode. We can split it using an HDFS federation.

Hadoop Command Line - Local/Hadoop Filesystem

Hadoop commands are similar to Linux ones, but they start with **`hadoop fs -`**

Copy a file from local filesystem to Hadoop filesystem

```
hadoop fs -copyFromLocal src target
```

Copy a file from Hadoop filesystem to local filesystem

```
hadoop fs -copyToLocal src target
```

To make the difference between the local filesystem and Hadoop filesystem, we can add: `file:///` and `hdfs://`

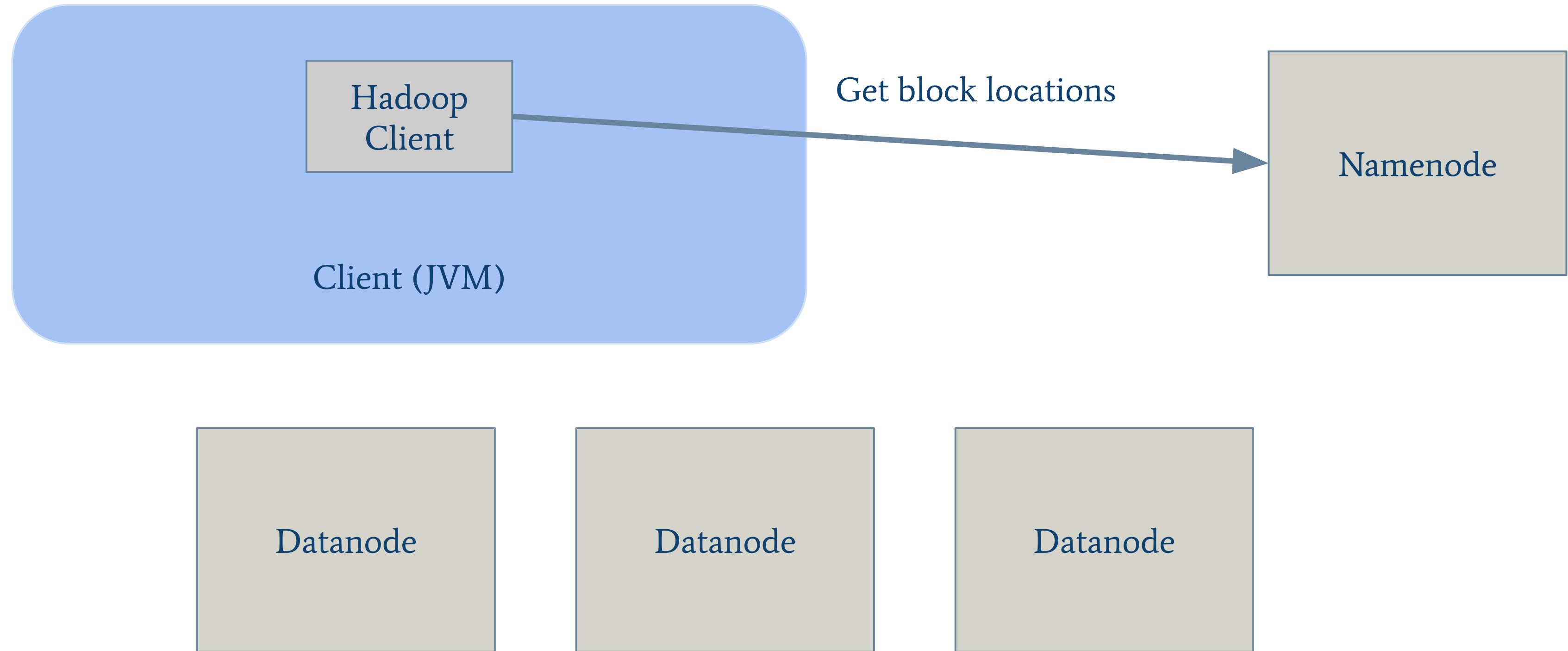
Hadoop Command Line - Traditional Commands

- Create directory: `hadoop fs -mkdir`
- List files: `hadoop fs -ls path`
- Change permissions: `hadoop fs -chmod`
- ...

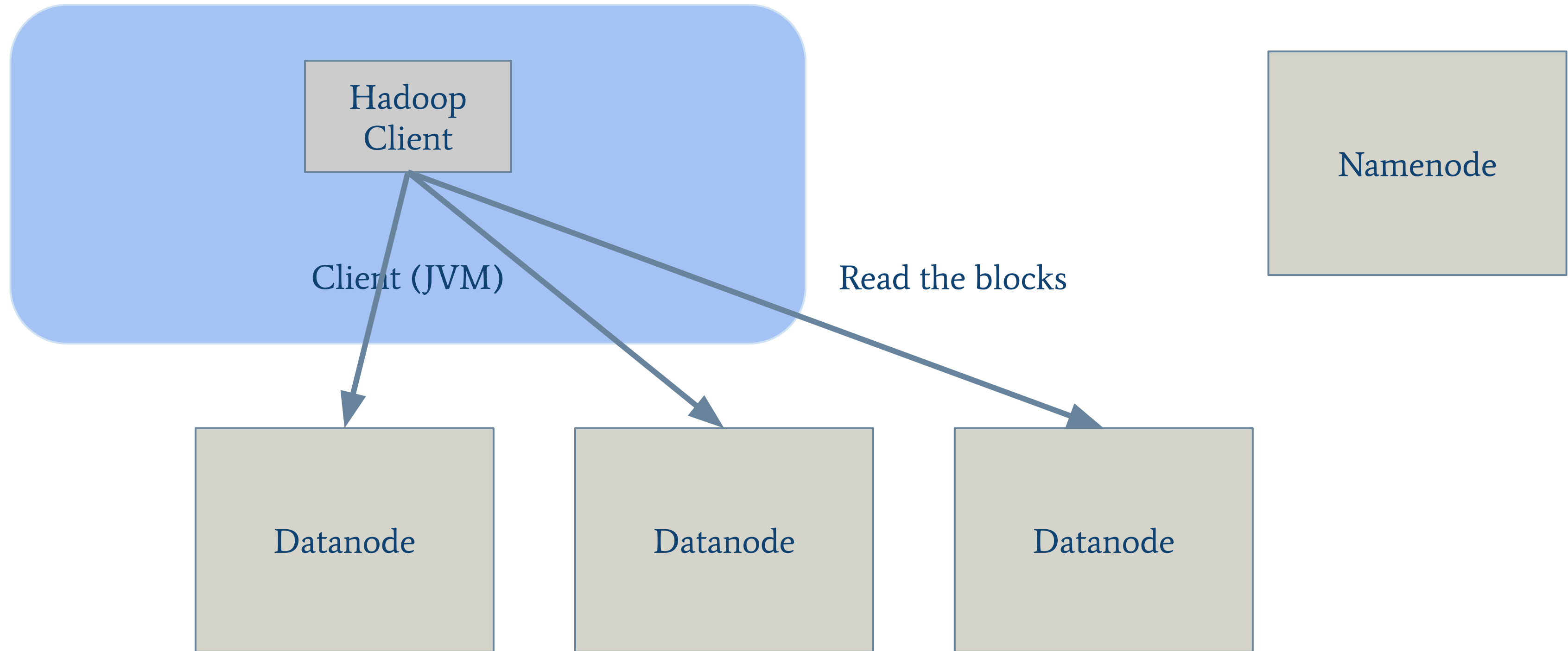
(`cd` does not make sense here)

- **`Beware!`** `hadoop fs -cp src target` copies a file that is already on the HDFS to a file also on the HDFS! It cannot go from the local filesystem to HDFS.

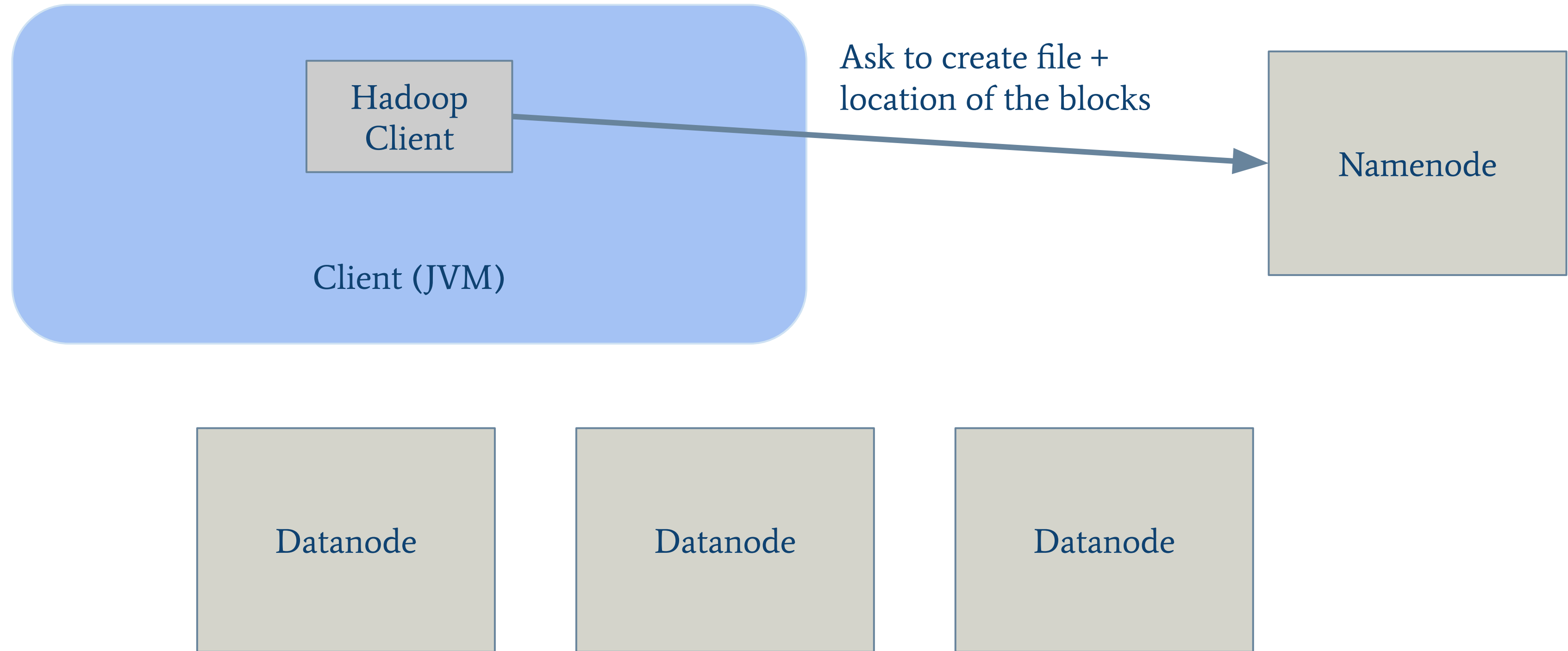
Dataflow - Read



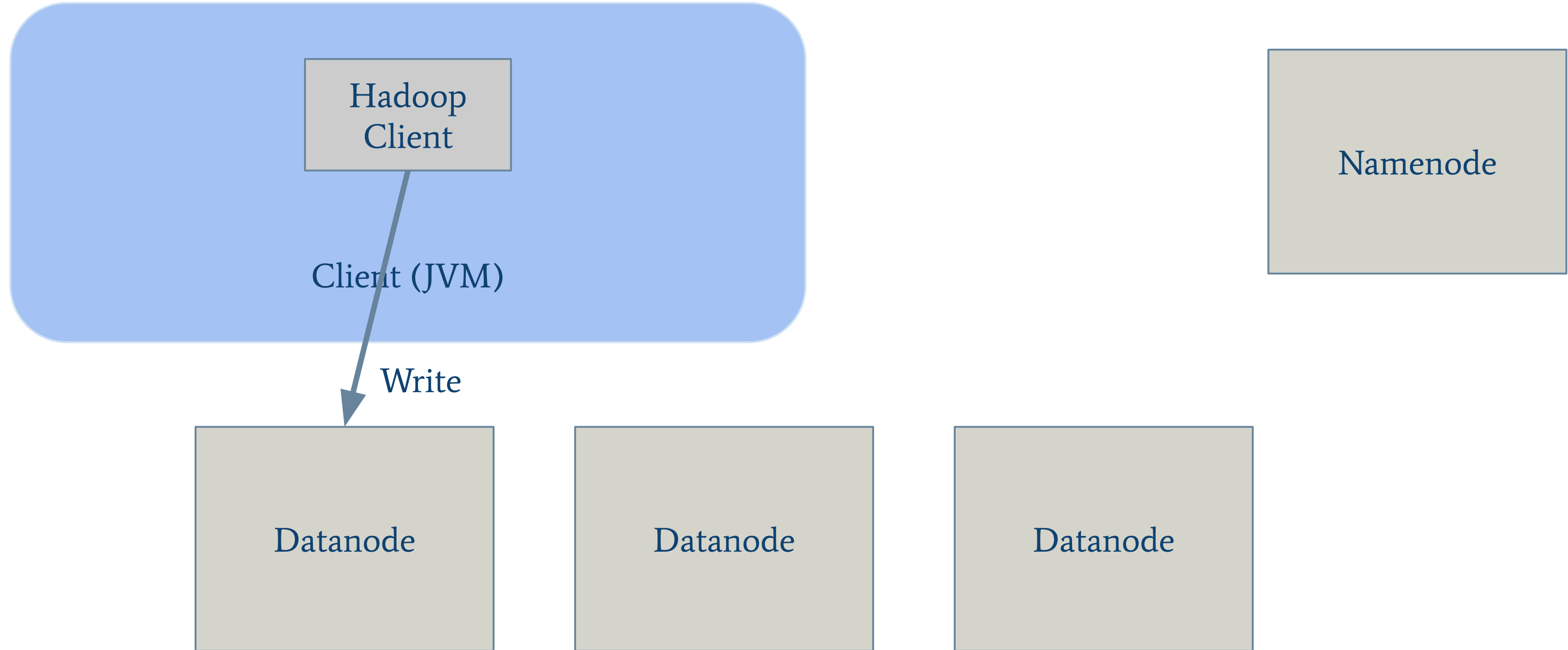
Dataflow - Read



Dataflow - Write



Dataflow - Write



Dataflow - Write