Distributed Computing

Common Big Data Challenges

- Volume:
  - ***Too much data to store/process*** — reduce processing time and/or scale out computation/storage
    - Performing OCR on 1000s of articles simultaneously
  - ***Too big to fit in RAM*** (esp. when no cluster resource available)
    - Given 30GB of taxi trip records → *how to plot the trend?*
- Velocity — ***data comes in real-time*** → too much to store → process **on the fly**!
  - Detecting network failures from inspecting billions of packets per hour

Common Big Data Challenges

- Volume:
  - ***Too much data to store/process*** — reduce processing time and/or scale out computation/storage
    - Performing OCR on 1000s of articles simultaneously
  - ***Too big to fit in RAM*** (esp. when no cluster resource available)
    - Given 30GB of taxi trip records → *how to plot the trend?*
- Velocity - **Streaming Computations on the fly**!
    - Detecting network failures from inspecting billions of packets per hour

Common Big Data Challenges

- Volume:
  - ***Too much data to*** **Distributed and** g time and/or scale out computation/st**Parallel Computing**
    - Performing OCR on 1000s of articles simultaneously
  - ***Too big to fit in RAM*** (esp. when no cluster resource available)
    - Given 30GB of taxi trip records → *how to plot the trend?*
- Velocity - **Streaming Computation**s **on the fly**!
  - Detecting network failures from inspecting billions of packets per hour

Parallel vs. Concurrent vs. Distributed Computing

Parallel vs. Concurrent vs. Distributed Computing

- Parallel Computing: taking full advantage of parallel architecture to speed up computation — *all about performance*
  - Data can be distributed to leverage additional processing power
- Concurrent Computing: enabling processes/tasks to progress without waiting for each other — *all about dependencies*
  - Parallel Computing in task graphs requires concurrency
- Distributed Computing: executing computations on distributed systems properly — *all about uncertainty*
  - Data are distributed by nature (e.g. because of their volume)

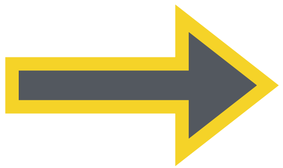
Big Data Computing

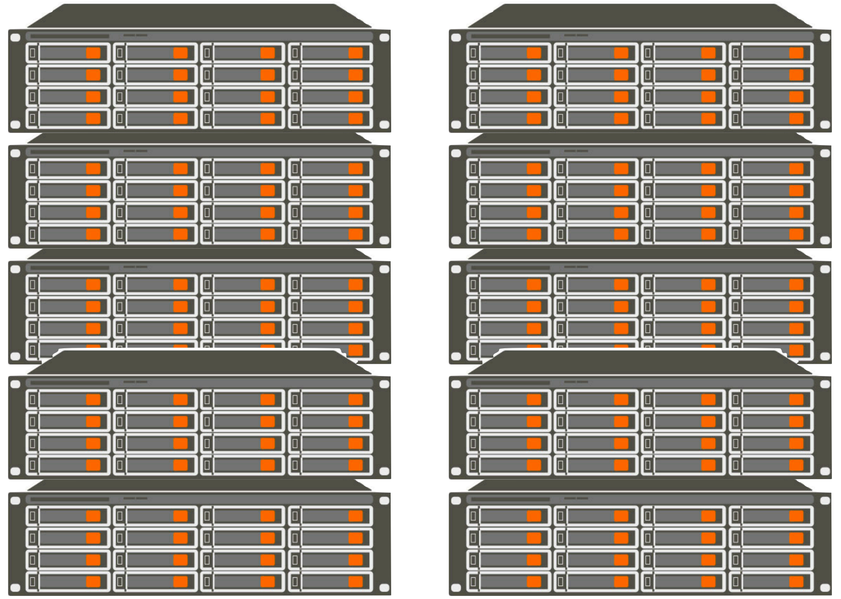
- Big Data requires distributed computing
  - part of the problem
  - to scale out storage and processing
- Big Data needs parallelism (thus, also concurrent) computing
  - focus on data parallelism for its scalability
- Big Data Platform usually
  - expose parallelism through a custom programming model
  - but handle most distributed computing issues behind the scene

Big Data Computing *needs* Scale Out!

- However, Scale Up is more efficient than Scale out
  - Minimal network communication (slowest in the bandwidth hierarchy)
  - Can fit even several TBs of RAM (compute@cusp has 1TB!)
  - Fast access for spatial (pre-fetch) and temporal (caching) locality
- Leverage multi-core architectures in distributed environment
  - Need to exhaust each machine resources before going across nodes
  - Hierarchical (non-uniform) memory access matters!

Scale Out Storage





1 machine
10 I/O Channels ~1 GB/s
Read 1 TB: ~17 minutes

10 machines
100 I/O Channels ~10 GB/s
Read 1 TB: ~2 minutes

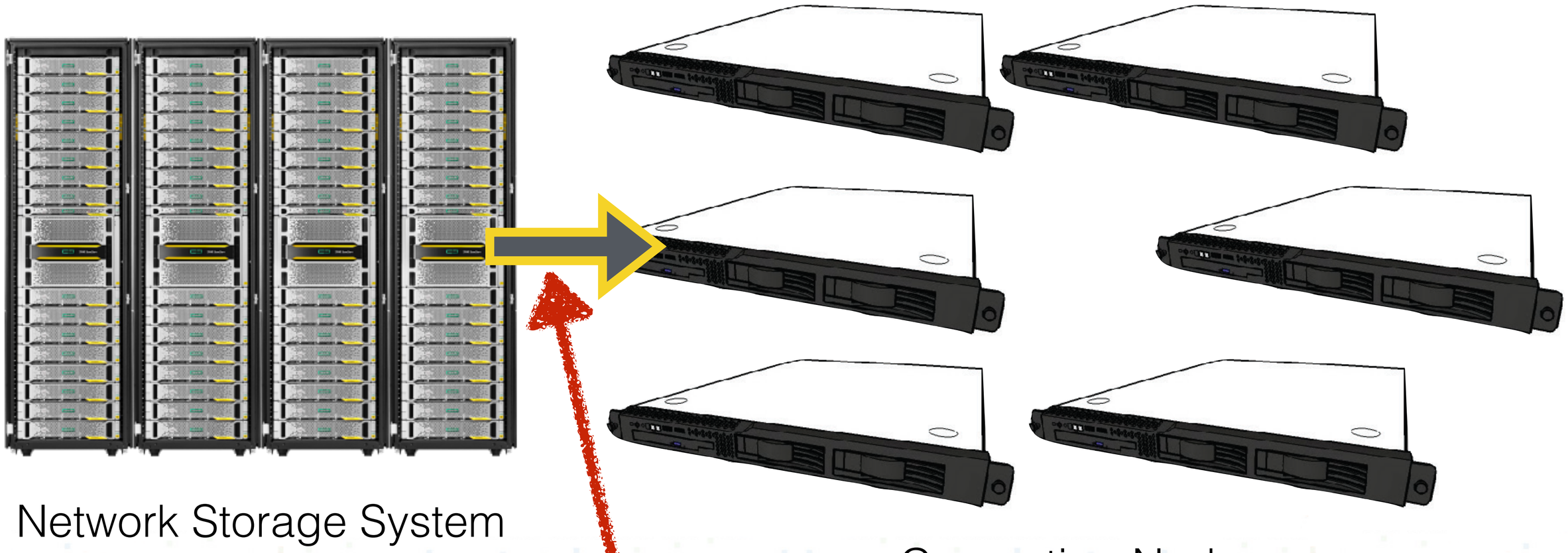
Traditional Parallel Computing Model (HPC)

Network Storage System

Computing Nodes

data over the network

Traditional Parallel Computing Model (HPC)

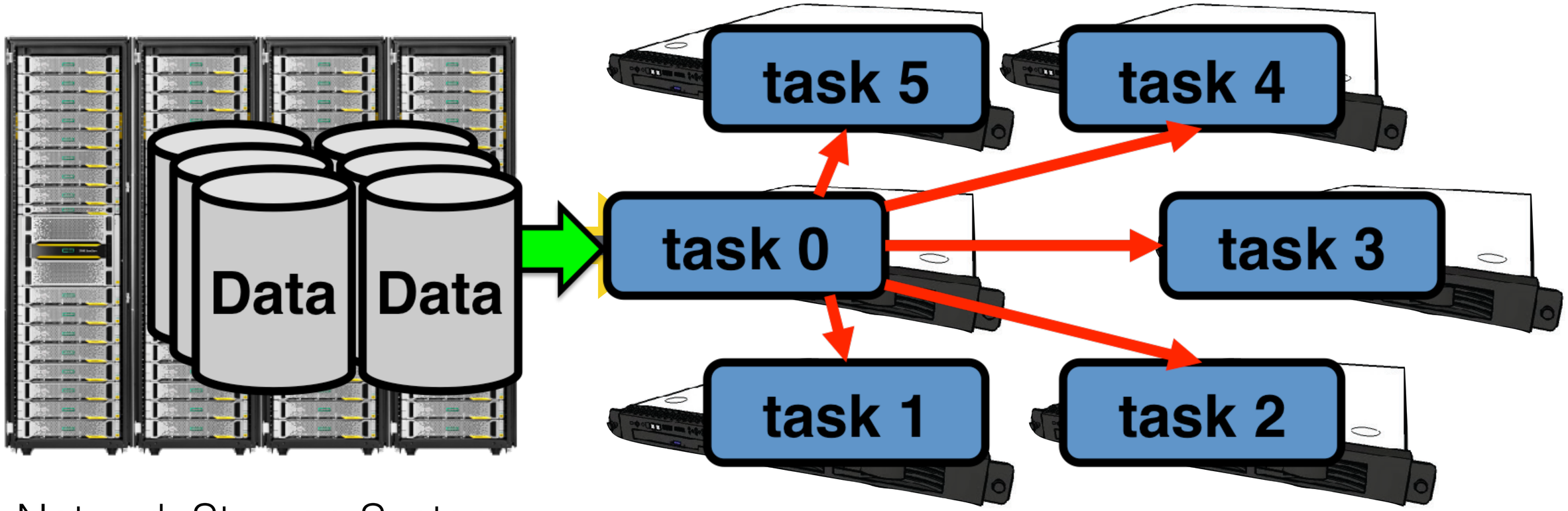


Network Storage System

Computing Nodes

data over the network

Traditional Parallel Computing Model (HPC)

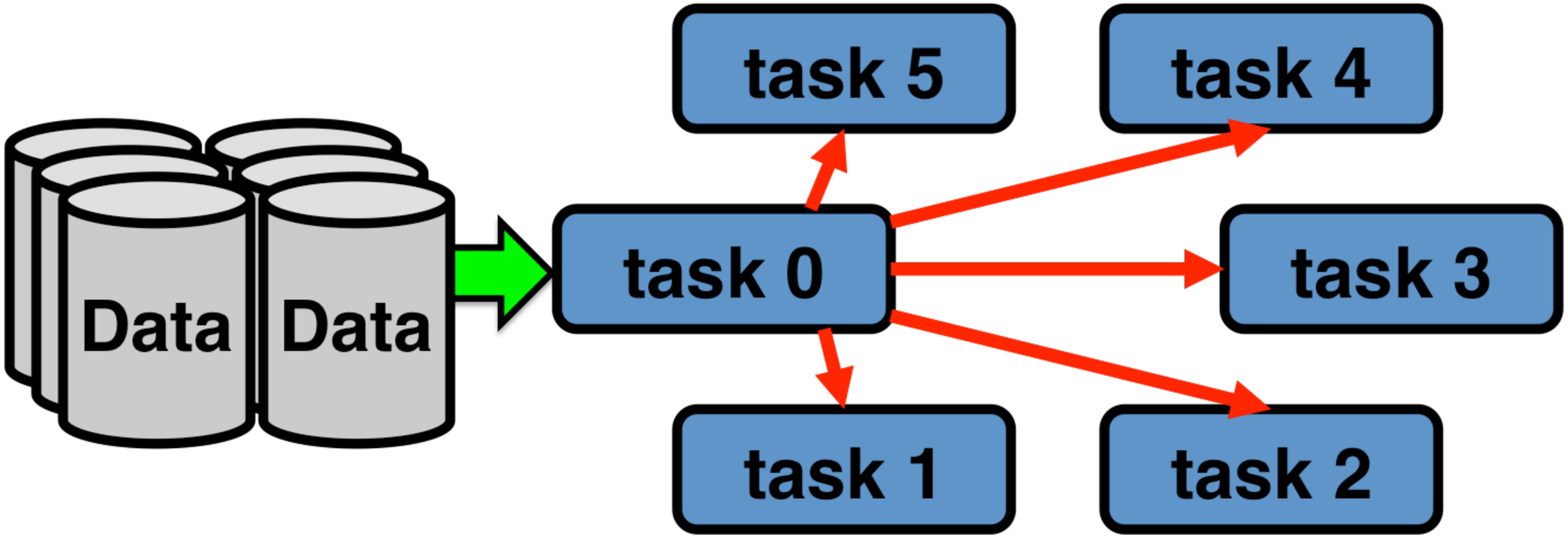


Network Storage System

Computing Nodes

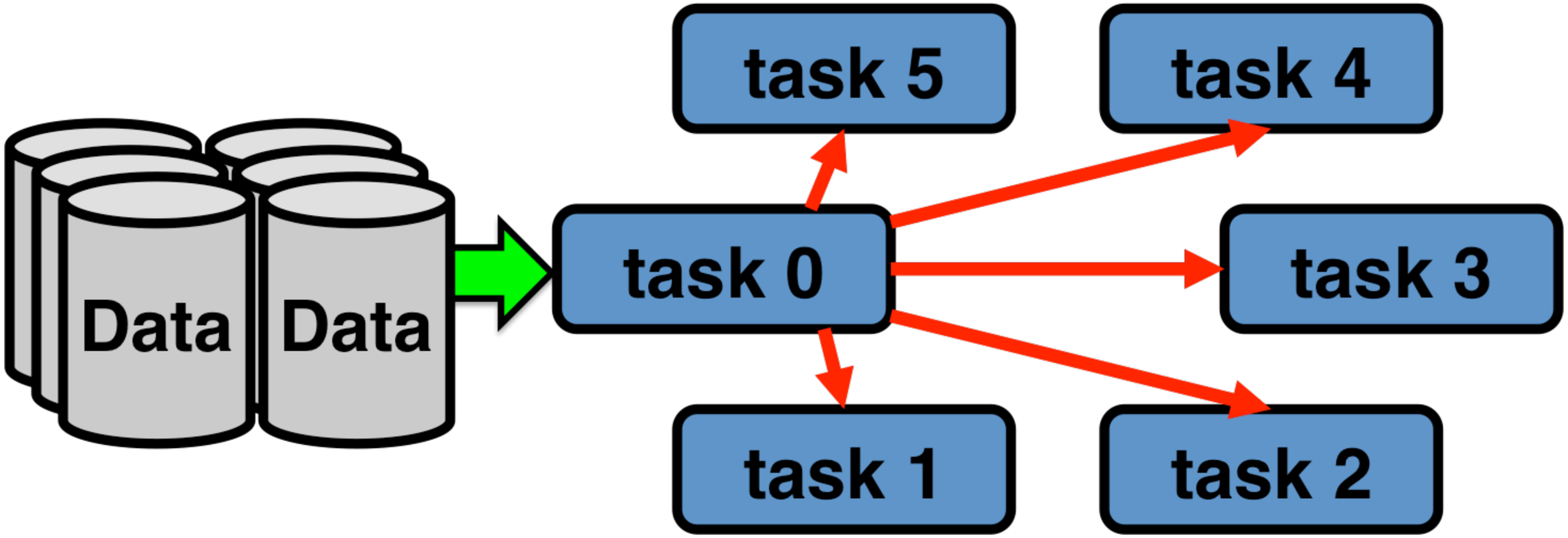
[Glenn K. Lockwood. Conceptual overview of map-reduce and hadoop]

Traditional Parallel Computing Model (HPC)



[Glenn K. Lockwood. Conceptual overview of map-reduce and hadoop]

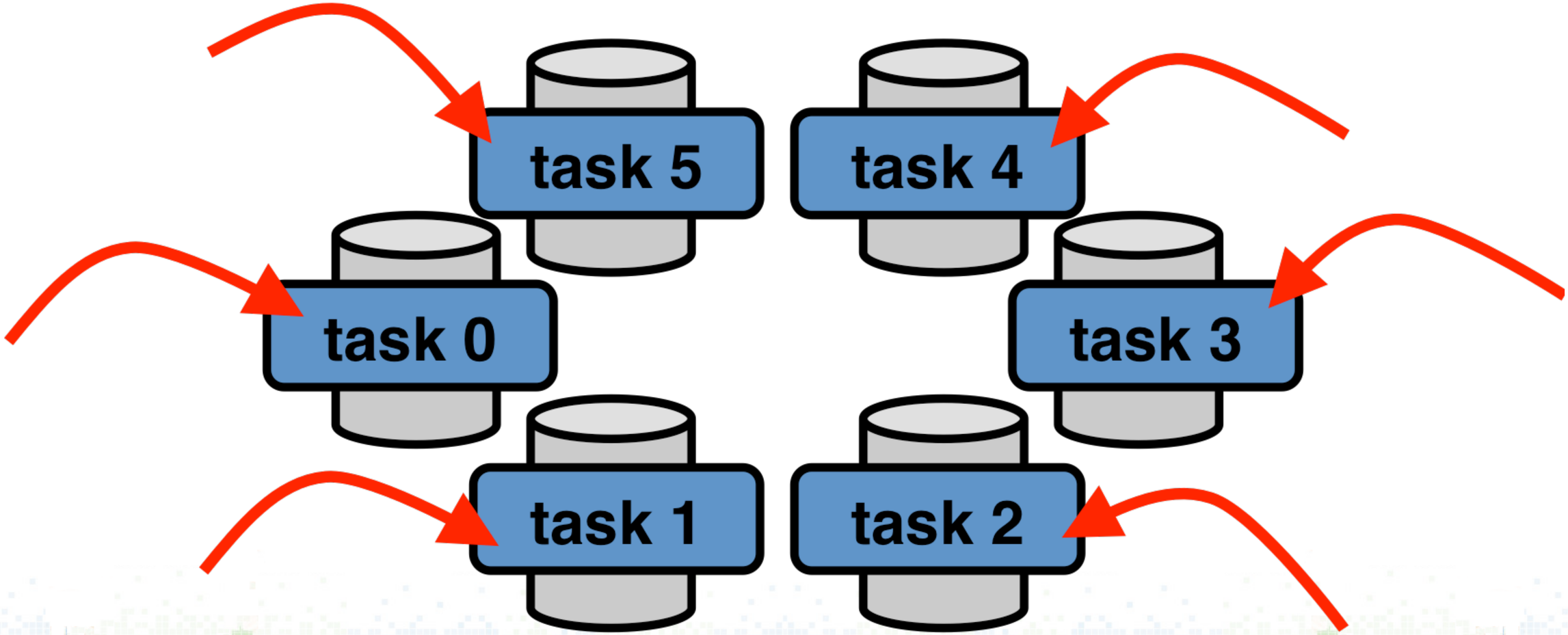
Traditional Parallel Computing Model (HPC)



bring data to compute — *limit by connection bandwidth*

[Glenn K. Lockwood. Conceptual overview of map-reduce and hadoop]

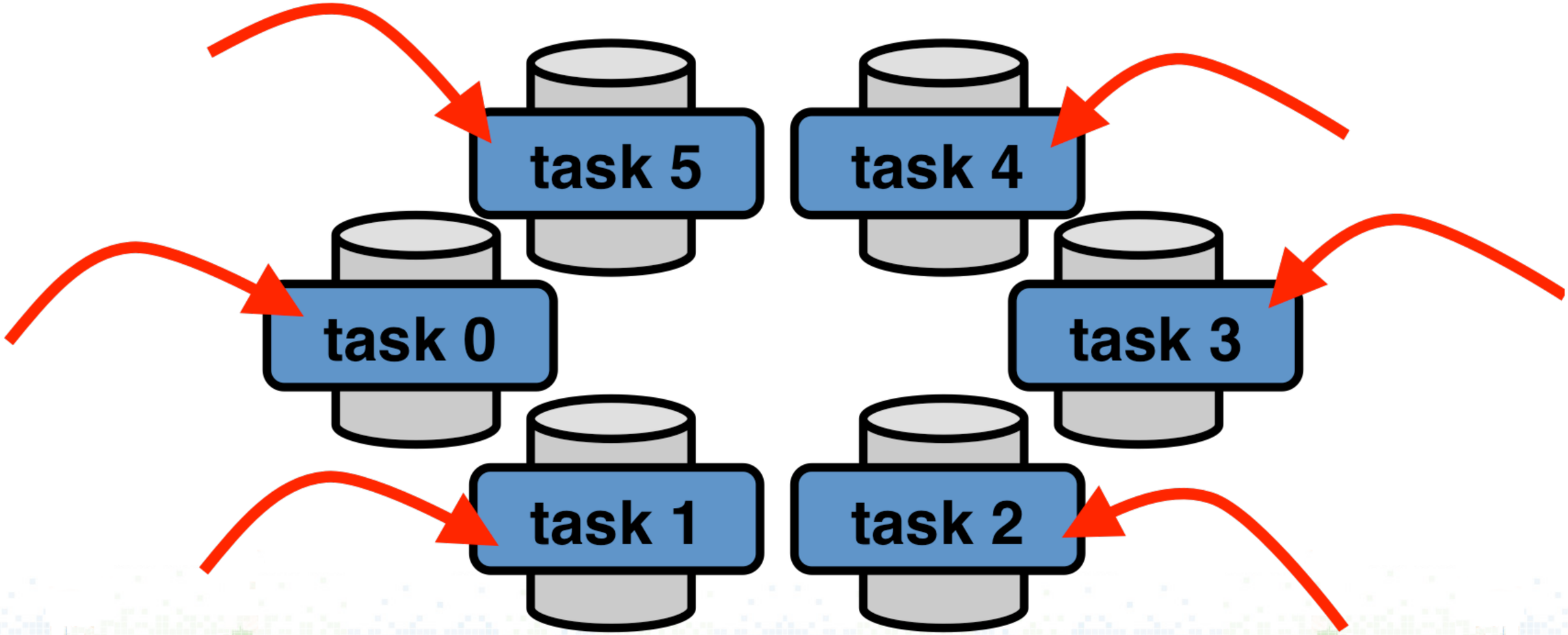
Big Data Computing Model



[Glenn K. Lockwood. Conceptual overview of map-reduce and hadoop]

Big Data Computing Model

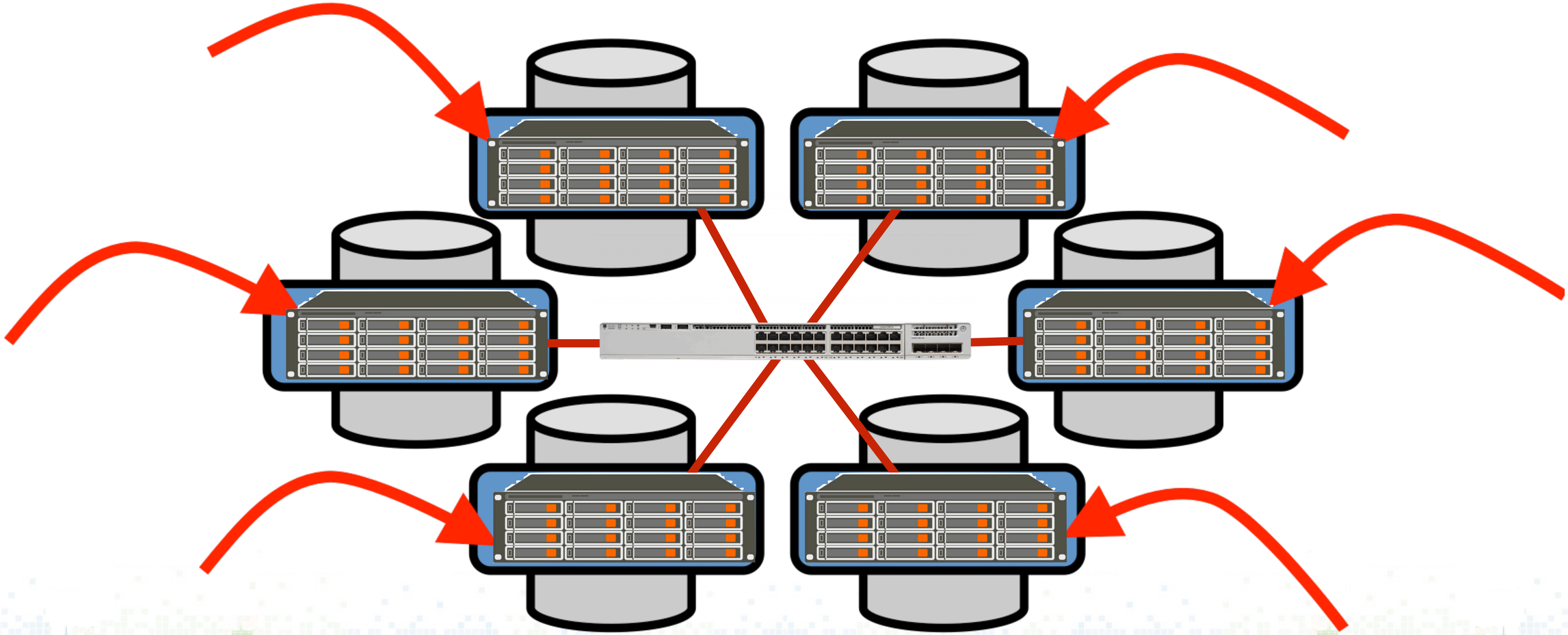
bring compute to data — *avoid data movement*



[Glenn K. Lockwood. Conceptual overview of map-reduce and hadoop]

Big Data Computing Model

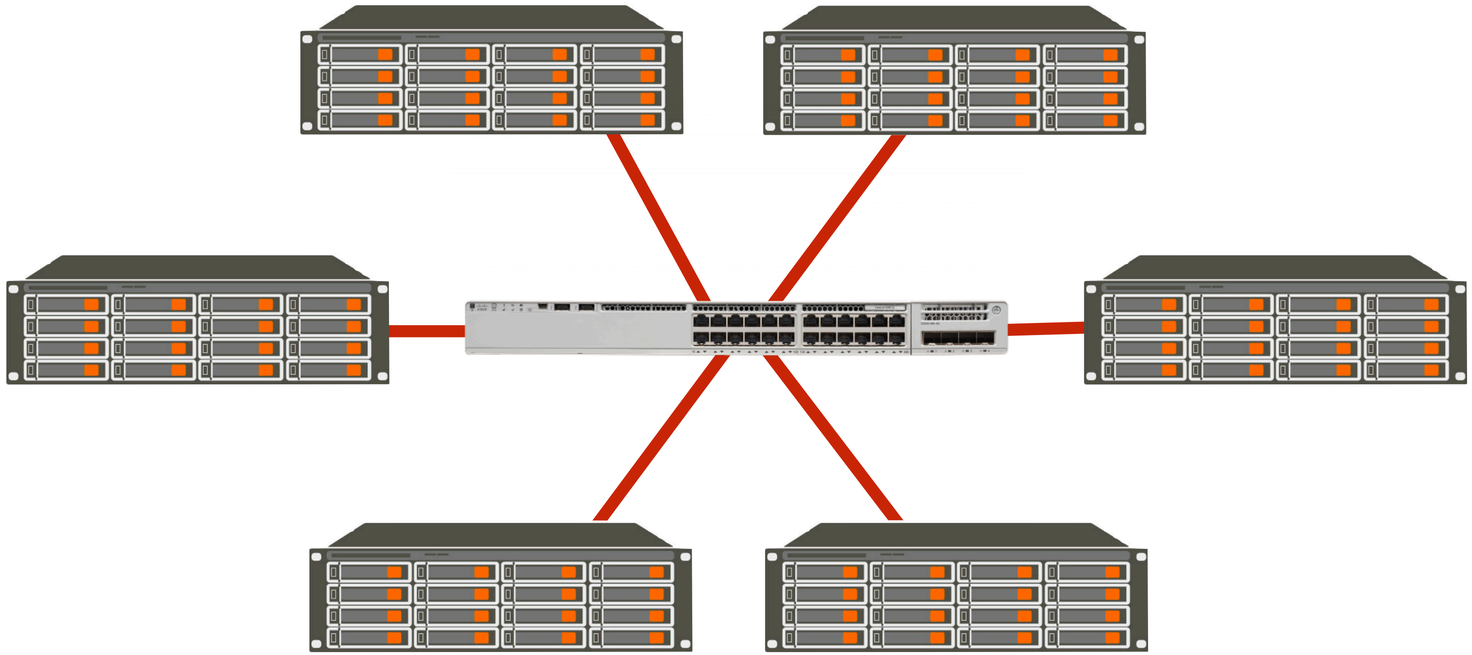
bring compute to data — *avoid data movement*



[Glenn K. Lockwood. Conceptual overview of map-reduce and hadoop]

Big Data Computing Model

bring compute to data — *avoid data movement*



Challenges in Analyzing Big Data

- How to specify *data computation* so that the system can:
  - bring compute to the data
  - minimize moving or making changes to data
  - operate on collection of data
  - allow users to dictate *what* to be done to the data and LEAVING *how* it will be done to the platform
  - leveraging both scaling up and scaling out capabilities

Challenges in Analyzing Big Data

- How to specify *data computation* so that the system can:
  - bring compute to the data
  - minimize moving or making changes to data
  - operate on collection of data
  - allow users to dictate *what* to be done to the data and LEAVING *how* it will be done to the platform
  - leveraging both scaling up and scaling out capabilities

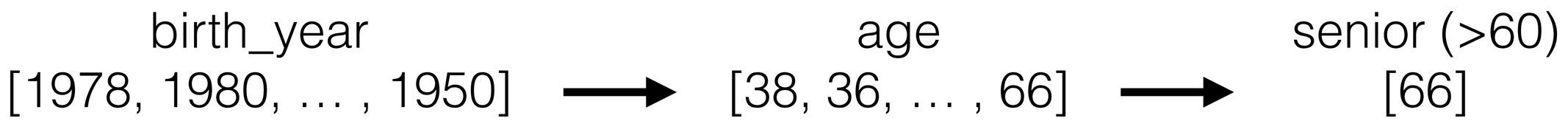
a declarative and/or functional language for data processing

Higher-Order Functions (Functional)

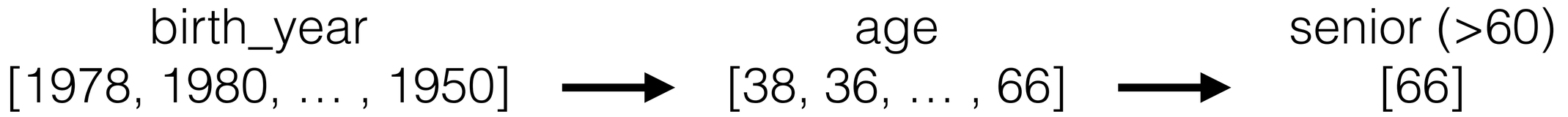
- Higher-Order functions (functional) are functions that:
  - take functions as arguments
  - **AND/OR** return functions as its results
- HOF is used to abstract common iteration operations
  - *focus on the* **what** *instead of the* **how**
- Using a predefined set of Higher-Order Functions, we can apply data transformation to data using our own **function** (aka transformation)
  - similar to streaming data elements through our function

Review: General Data Computing Model

- Given a collection of data records (e.g. a generator or list of records)
- Apply a series of *data transformations* to the collection
- Collect the final transformation



Example: How to do it in Python?



Example: How to do it in Python?

birth_year		age		senior (>60)
[1978, 1980, … , 1950]	→	[38, 36, … , 66]	→	[66]

```
age = []
for y in birth_year:
    age.append(2016-y)

senior = []
for y in age:
    if y>60:
        senior.append(y)
```

Example: How to do it in Python?

birth_year
[1978, 1980, … , 1950] → age
[38, 36, … , 66] → senior (>60)
[66]

```
age = []
for y in birth_year:
    age.append(2016-y)

senior = []
for y in age:
    if y>60:
        senior.append(y)
```

```
birth_year = pd.Series(birth_year)

age = 2016-birth_year

senior = age[age>60]
```

Example: How to do it in Python?

name,birth_year
[(‘A’,1978), … , (‘C’,1950)] → name,age
[(’A’,38), … , (’C’,66)] → name,senior
(>60 with name ‘C’)
[(’C’,66)]

Example: How to do it in Python?

name,birth_year
[(‘A’,1978), … , (‘C’,1950)] → name,age
[(’A’,38), … , (’C’,66)] → name,senior
(>60 with name ‘C’)
[(’C’,66)]

```
age = []
for y in birth_year:
    age.append((y[0], 2016-y[1]))

senior = []
for y in age:
    if y[0]=='C' and y[1]>60:
        senior.append(y)
```

Example: How to do it in Python?

name,birth_year
[('A',1978), … , ('C',1950)] → name,age
[('A',38), … , ('C',66)] → name,senior
(>60 with name 'C')
[('C',66)]

```
age = []
for y in birth_year:
    age.append((y[0], 2016-y[1]))

senior = []
for y in age:
    if y[0]=='C' and y[1]>60:
        senior.append(y)
```

```
birth_year = pd.Series(birth_year)

age = 2016-birth_year

senior = age[age>60]
```

?

Example: How to do it i using HOF?

```
age = []
for y in birth_year:
    age.append(2016-y)

senior = []
for y in age:
    if y>60:
        senior.append(y)
```

Example: How to do it i using HOF?

```
def AGE_TRANSFORM(y):
    return 2016-y

def AGE_FILTER(y):
    return y>60


age = []
for y in birth_year:
    age.append(AGE_TRANSFORM(y))

senior = []
for y in age:
    if AGE_FILTER(y):
        senior.append(y)
```

Example: How to do it i using HOF?

```
def AGE_TRANSFORM(y):
    return (y[0], 2016-y[1])

def AGE_FILTER(y):
    return y[0]=='C' and y[1]>60


age = []
for y in birth_year:
    age.append(AGE_TRANSFORM(y))

senior = []
for y in age:
    if AGE_FILTER(y):
        senior.append(y)
```

Example: How to do it i using HOF?

```
def AGE_TRANSFORM(y):
    return (y[0], 2016-y[1])

def AGE_FILTER(y):
    return y[0]=='C' and y[1]>60
```

```
age = map(AGE_TRANSFORM, birth_year)
```

```
senior = filter(AGE_FILTER, age)
```

Python's Core HOF

- map(): applies a function over an iterable to produce a new iterable

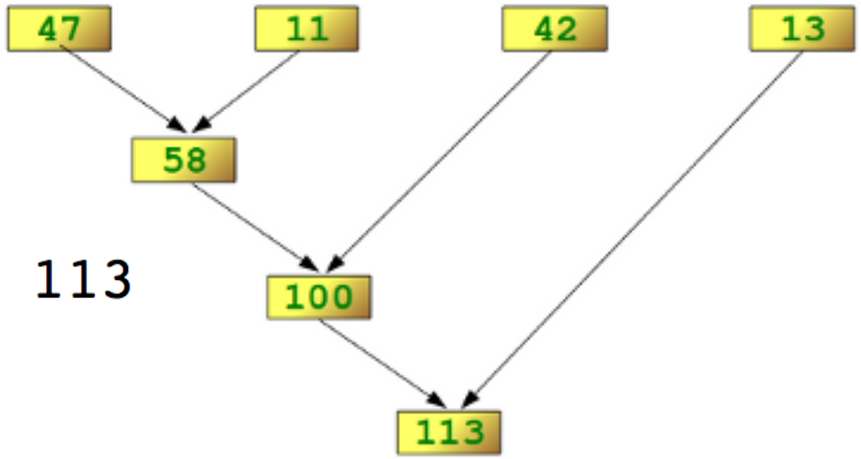
```
map(int, ['0', '1']) -> [0, 1]
```

- filter(): return only values that satisfy a predicate in the new iterable

```
filter(bool, [0, 1]) -> [1]
```

- reduce(): accumulate a sequence of values from left

```
reduce(operator.add, [47,11,42,13]) -> 113
```



- sorted(): return a new sorted list using a comparator
-

Anonymous function: *lambda*

- What if we only use a function once or would like to define it in-place?
- Python's anonymous lambda function:

```
lambda VAR1,VAR2,...,VARN: (expression on VAR1..N)
```

  - No name, no return statement, only an expression to evaluate
  - Come in handy in defining functions in HOF

```
x2 = lambda x: x*x
x2(10) # 100

x_y = lambda x,y: x+y
x_y(1,2) # 3
```

-

Python's Core HOF

- map(): applies a function over an iterable to produce a new iterable

```
map(lambda x: int(x)+1, ['0', '1']) -> [1, 2]
```

- filter(): return only values that satisfy a predicate in the new iterable

```
filter(lambda x: x<1, [0, 1]) -> [0]
```

- reduce(): accumulate a sequence of values from left to right

```
reduce(lambda x,y: x+y, [47,11,42,13]) -> 113
```

- sorted(): return a new sorted list using a comparator function

```
sorted(xrange(5))                          -> [0,1,2,3,4]
sorted(xrange(5), cmp=lambda x,y:y-x) -> [4,3,2,1,0]
```

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

```
age = []
for y in birth_year:
    age.append(2016-y)

senior = []
for y in age:
    if y>60:
        senior.append(y)
```

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

```
age = [2016-y for y in birth_year]
```

```
senior = []
for y in age:
    if y>60:
        senior.append(y)
```

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

```
age = [2016-y for y in birth_year]
```

```
senior = [y for y in age if y>60]
```

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

```
age = map(lambda y: 2016-y, birth_year)
```

```
senior = [y for y in age if y>60]
```

List Comprehension

- Syntactic sugar for Higher-Order Functions (w/ select…from…where)
  - to construct lists in a mathematically beautiful way (no more append!)

```
age = map(lambda y: 2016-y, birth_year)
```

```
senior = filter(lambda y: y>60, age)
```

Why HOF matters?

- Big Data Platforms including Apache Spark (and the principals of Hadoop) are built on functional programming languages
  - has a smaller set of data processing constructs, thus, easier for the platforms to optimize parallelism
  - less prone to data states and copies
  - keep track of provenance (data + transformation)
- As a user, it keeps us more conscious about concurrency and parallelism
  - algorithm design at a high level (mathematically)

reduce() in Python 3

- Its usage is “discouraged”
  - “it is 99% better to use a for loop instead”
  - It is available in the “functools” module

```
from functools import reduce
```

- But for loop is not always possible for big data platforms
  - Still need an abstract way (specifying “what”, not “how”) for aggregation

Questions?

- Big Data Platforms need to abstract data transformation -> Higher-Order functions
  - Think Big Data -> think Functional!
- Resources on HOF:
  - https://wizardforcel.gitbooks.io/sicp-in-python/content/6.html — Higher-Order Functions
  - https://github.com/joelgrus/stupid-itertools-tricks-pydata — Functional Python for Learning Data Science
    - Slides included

Next Class

- *Hadoop: The Definitive Guide — Storage and Analysis at Internet Scale*
  - Please read Chapter 2 and 3 (MapReduce and HDFS)
- Google File System (papers on Class resources)