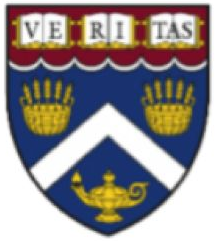
CSCI E-88A Introduction to Functional and Stream Processing for Big Data Systems

Harvard University Extension, Spring 2020

Marina Popova, Edward Sumitra



Lecture 1 - Introduction

Agenda

- Class goals and Admin info
- What is Functional Programming?
- how is it relevant for true distributed programming?
- intro to data processing scaling
- how FP fits into distributed processing - example
- Class RoadMap and logistics

Teaching Staff - Instructors:

Marina Popova

- MS in Mathematics and Astronomy, St. Petersburg State University, Russia
- ALM in IT, Harvard University Extension
- Engineer, NetApp, Inc.

Edward Sumitra

- Master of Science, Boston University
- Bachelor of Technology - Indian Institute of Technology (IIT) - Kharagpur
- Software Development Manager, Curriculum Associates

Teaching Staff - Teaching Assistants

Pavithra Venkatachalam

- MS in Computer Science, University of Massachusetts, Boston
- BE in Electronics & Communication, Visvesvaraya Technological University, India

Misha Smirnov

- MS in Computer Science, St. Petersburg State University, Russia
- Manager Software Engineering, Endurance

@Marina Popova, Edward Sumitra

What are we going to do in this class?

- What and how are we going to learn?
  - It's a breadth-first type of class
  - this is "Introduction" - not a deep-dive into advanced FP
  - we will learn basics of FP and:
  - where it is used - the big picture
  - how it is used in real-life scenarios/ applications/ frameworks
  - get hands-on experience writing FP applications for some use cases
- By the end of the class you should be able to understand the value and applicability of FP, and know where to go from there
- this is not a class to learn Java or Scala in details - there are many classes for that and with a narrow focus of mastering different specific areas of FP

Lets Start !!

What Is Functional Programming?

Ref: https://medium.com/javascript-scene/can-you-avoid-functional-programming-as-a-policy-7bd0570bcfb2 by Eric Elliot

"Functional programming is a programming paradigm using pure functions as the atomic unit of composition, avoiding shared mutable state and side-effects.

A **pure function** is a function which:

Given the same input, always return the same output

Has no side effects

The essence of FP really boils down to:

Program with functions

Avoid shared mutable state & side effects "

The next important concept to understand is the difference between Imperative vs. Functional style of programming

What Is Functional Programming - cont

Ref: https://medium.com/front-end-weekly/imperative-versus-declarative-code-whats-the-difference-adc7dd6c8380

Imperative Paradigm:

- Procedural and OO programming: C, C++, Java, PHP
- programs consists of statements that change program's state (registers, memory slots)
- programs instruct computer how to do things, often by using:
  - conditional statements
  - loops
  - class inheritance

Declarative Paradigm:

- Functional, Logic and Domain-language-specific languages: HTML, XML, SQL, Prolog, Haskel, Scala
- programs declare the logic but not define the flow of execution
- Example of declarative code: HTML:
- In Functional PRogramming:
  - program creates expressions instead of statements and evaluates functions
  - avoids program state, side-effects and mutable data
  - given the same input, functions return the same result

why do we focus on FP ? → **foundation of true distributed programming !**

Introduction to Scaling

Why is "distributed" programming so important? especially in the context of Big Data?

Because applications processing Big Data cannot fulfil required SLAs on a "fixed resources" hardware

why ??

It's a Million $$$$ question!

Lets see how applications scale when processing more and more data ...

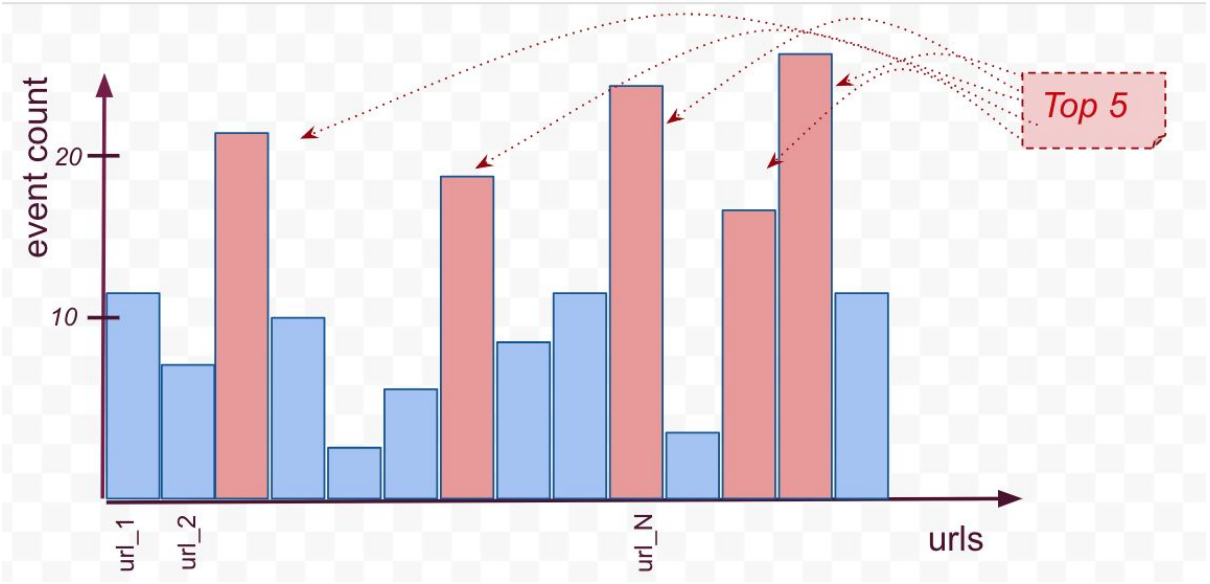
The Journey: Small → Big

Example Application:

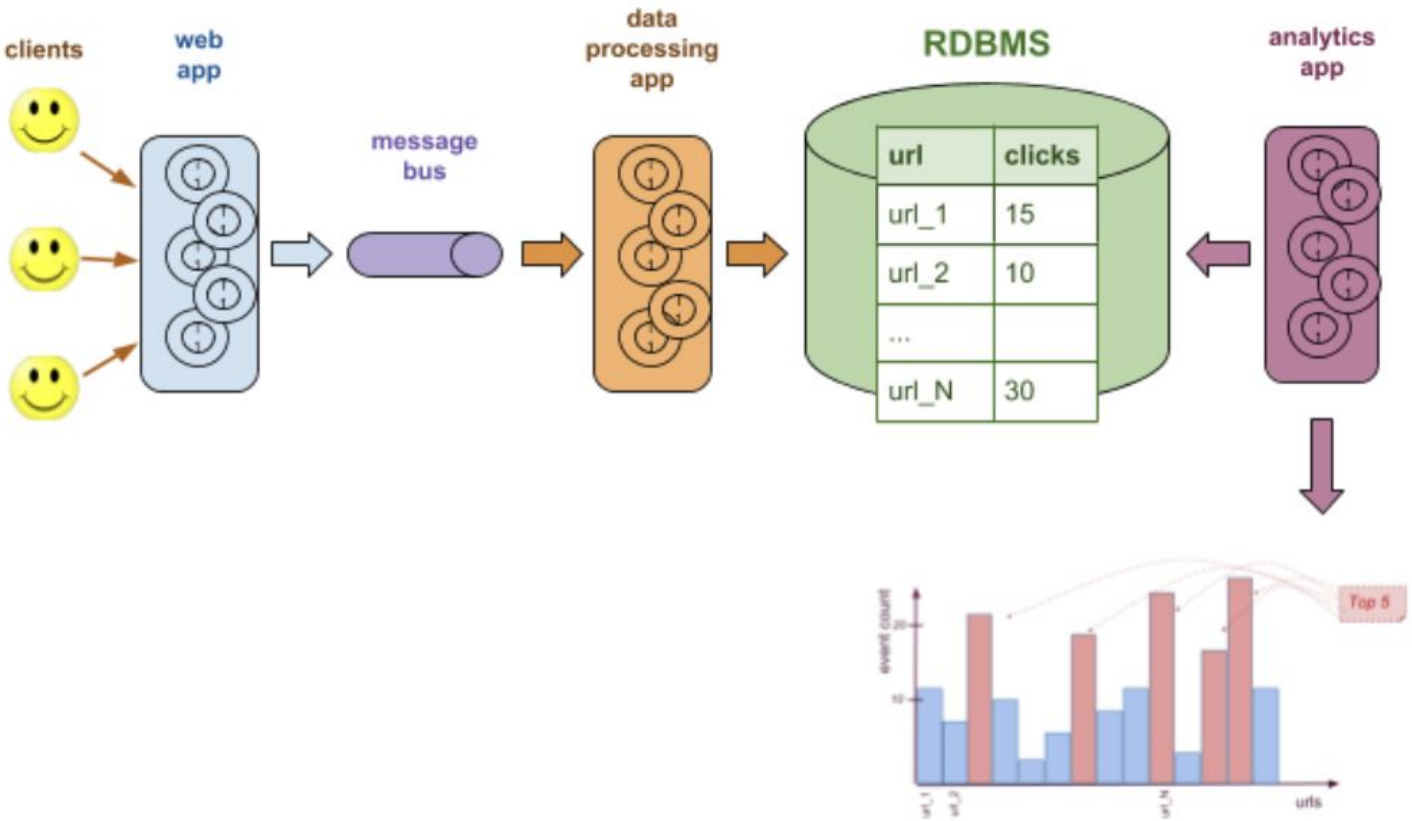
Online Store Page Analytics

Problem Statement/ Questions:

- how many clicks are done per each page URL ?
- what are Top 5 URLs based on their click numbers ?

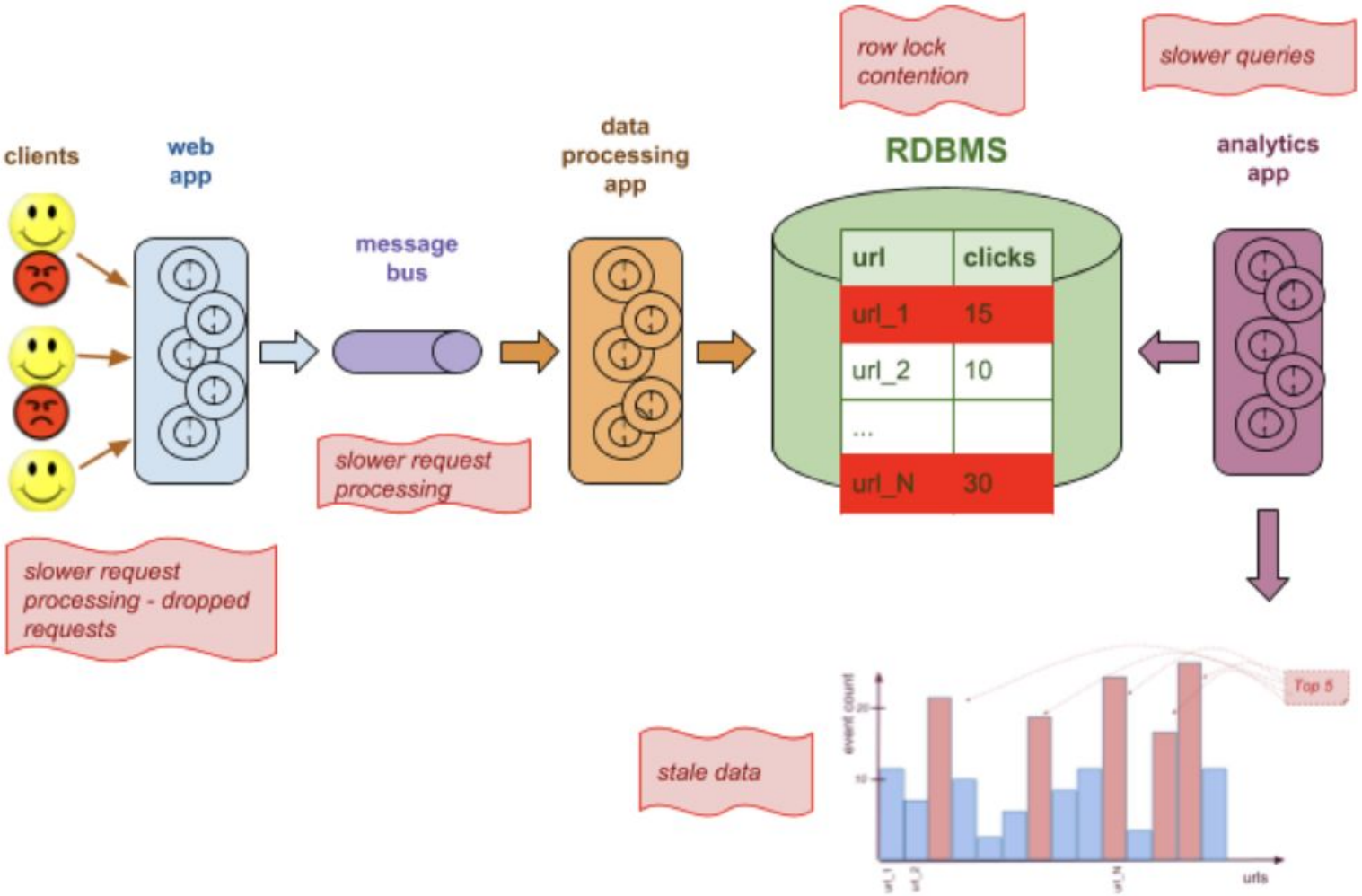


Typical architecture for this application:



What problems do we start seeing as traffic increases?

The Journey: Small → Big



So what do we do ?

Introduction to Scaling

We need to scale!

Usually, the first step in scaling is to scale vertically - get "more powerful" hardware

What does it mean - "more powerful" hardware?

Lets briefly see what hardware is made up of and how it can be "scaled"

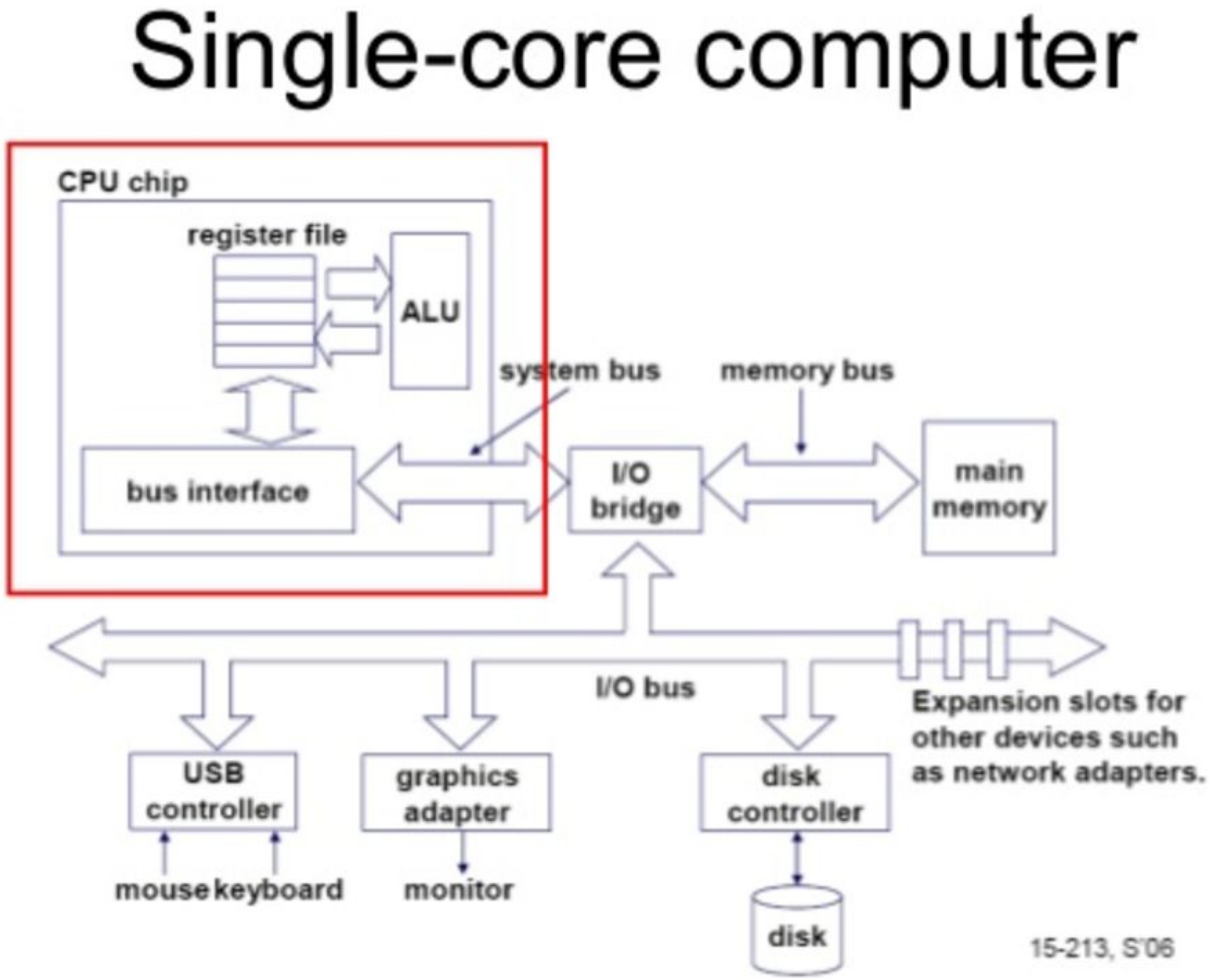
Scaling on Hardware Level

A CPU, or Central Processing Unit, is what is typically referred to as a processor.

- it contains many discrete parts within it, such as:
- one or more memory caches for instructions and data
- instruction decoders
- various types of execution units for performing arithmetic or logical operations.

A Core is the physical element that executes the code (like ALUs == Arithmetic Logic Unit).

- usually, each core has all necessary elements to perform computations, register files, interrupt lines etc.



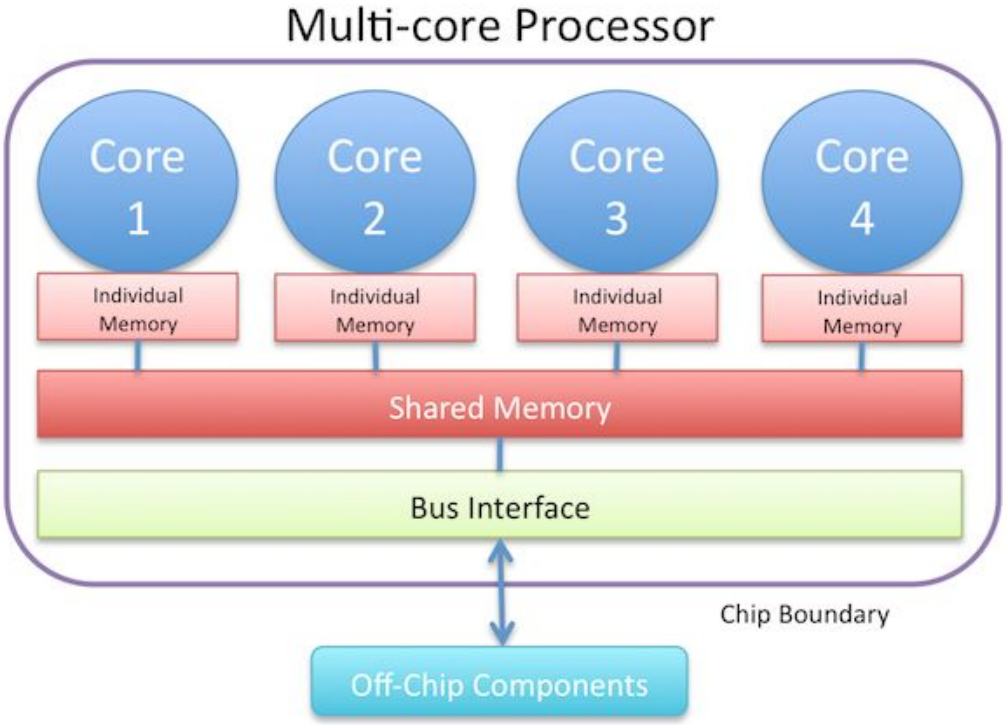
Ref: https://www.slideshare.net/piyushmittalin/multi-corearchitecture

Scaling on Hardware Level

A multi*core* CPU has multiple execution cores on one CPU.

It basically means that **a certain subset of the CPU's components is duplicated**, so that multiple "cores" can work in parallel on separate operations.

This is called CMP, Chip-level Multiprocessing.



For example, a multicore processor may have a separate L1 cache and execution unit for each core, while it has a shared L2 cache for the entire processor.

That means that while the processor has one big pool of slower cache, it has separate fast memory and arithmetic/logic units for each of several cores. This would allow each core to perform operations at the same time as the others

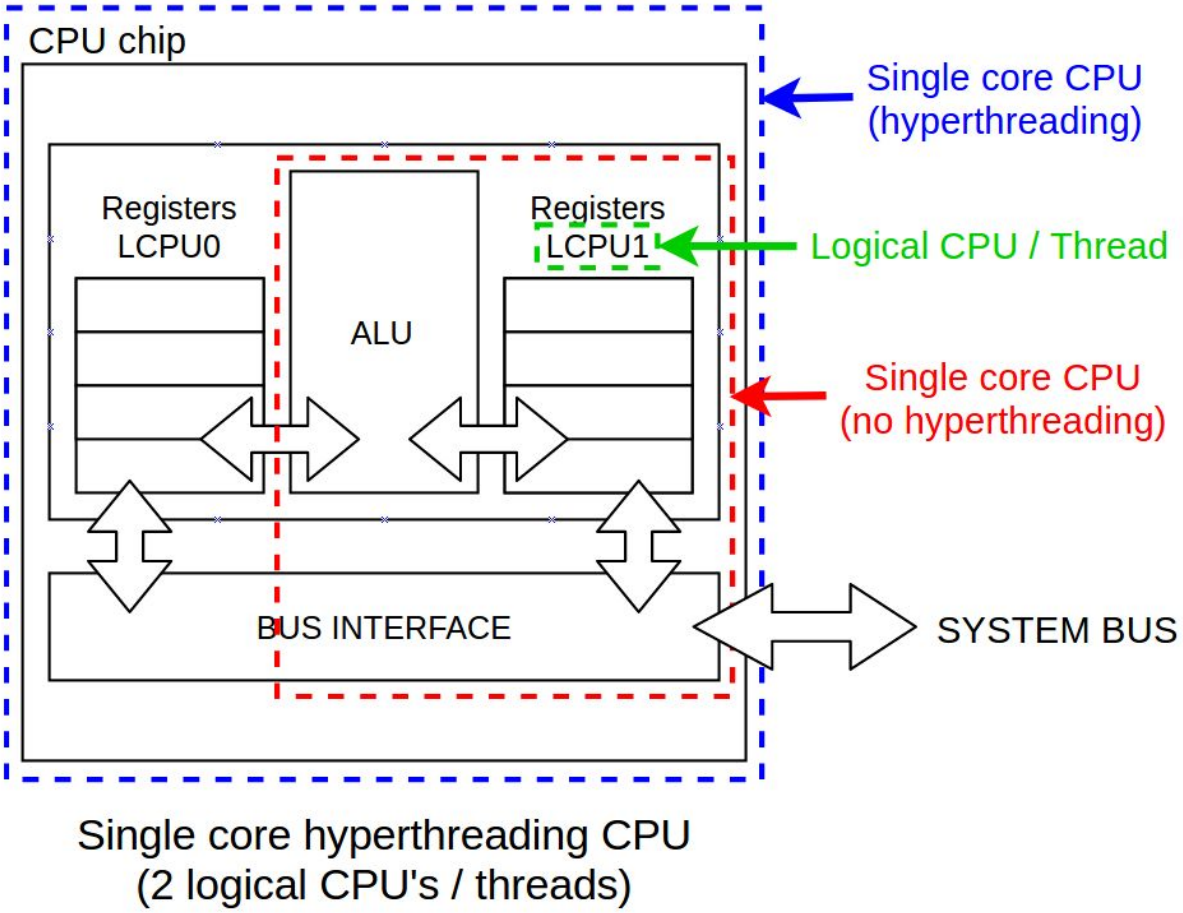
Scaling on Hardware Level - Hyperthreading

Hyper-Threading Technology is a form of simultaneous multithreading technology introduced by Intel

A processor with HT consists of **two logical processors[cores] per physical CPU**

each CORE has its own processor architectural state: Registers, local thread caches

But both cores share the execution resources: execution engine [ALU], caches, and system bus interface

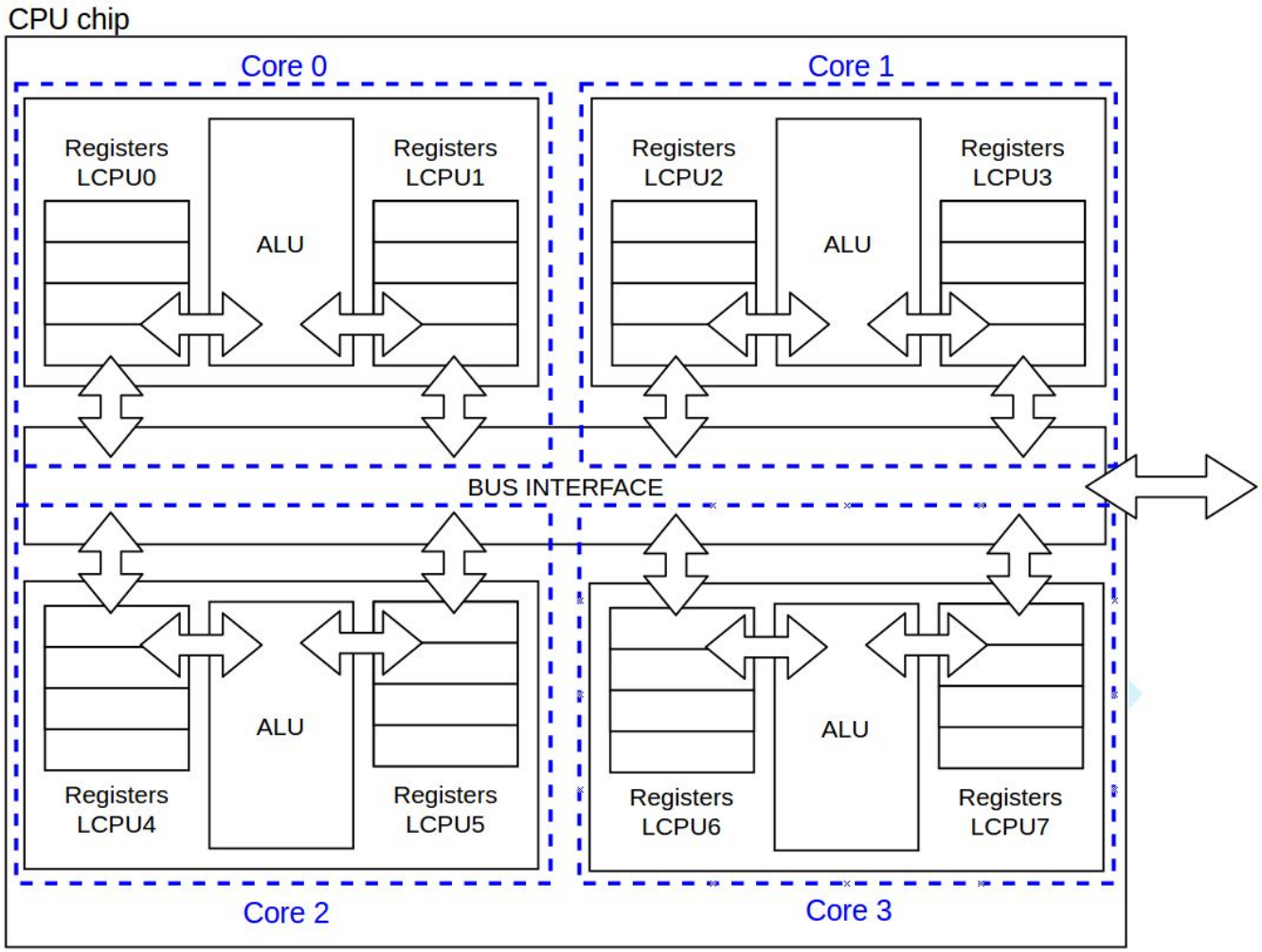


Hyperthreading is far less efficient then adding a full core, since if two threads need access to the same execution resource at the same time, one of them must wait for the other to finish.

Scaling on Hardware - Multi-Core with HT

Multi-core system with HT:

Cores in such system have faster (than physical CPUs) communications between them by means of a shared internal bus

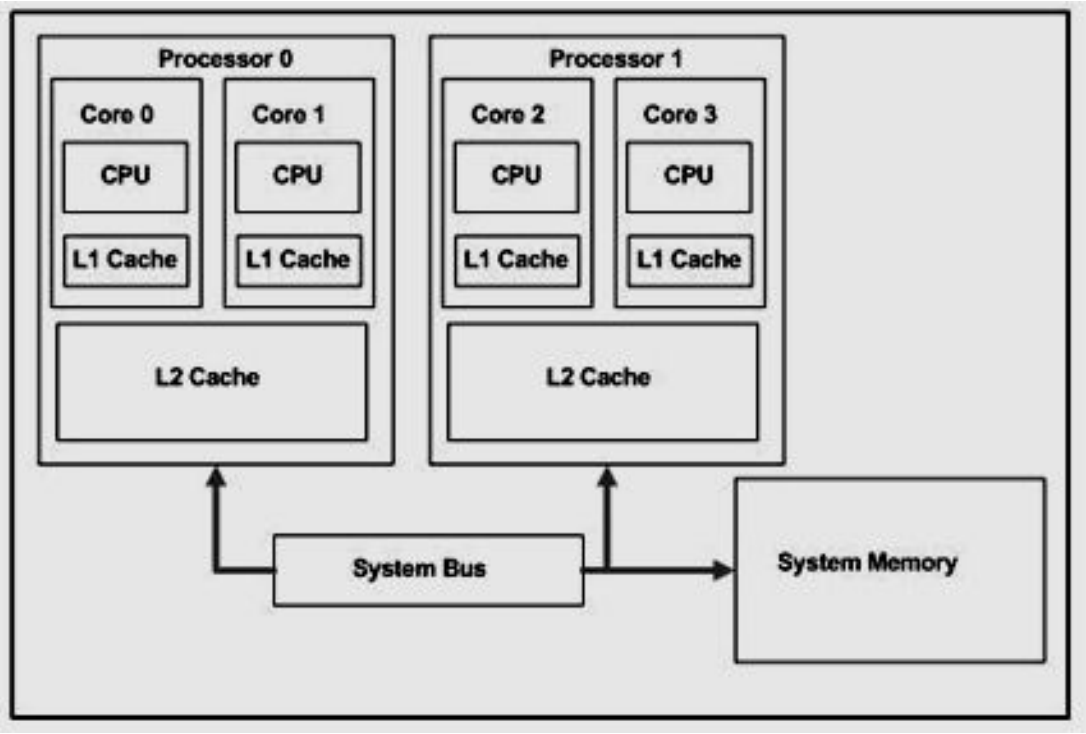


Quad-core hyperthreading CPU

Scaling on Hardware Level - Multi-Processor

Multi-processor vs multi-core architectures

Multi-processor system is a system with multiple physical processors, that communicate via System bus
However, the system must have support for a multiprocessor to work
A processor can have multiple cores - in which case It is a multi-core processor



Introduction to Scaling

We saw what hardware resources are and how they can be scaled

However, in order to take advantage of the available hardware resources - we have to design and build the software layer accordingly

How?

How can our applications take full advantage of the available cores/processing resources of the hardware?

By using "threads" and "processes" ...

Scaling on Software Level

Processes and Threads

- Most operating systems represent applications as **processes**. This means that the application has its own address space (== view of memory), where the OS makes sure that this view and its content are isolated from other applications.
- A process consists of one or more **threads**, which carry out the real work of an application by executing machine code on a CPU. The operating system determines, which thread executes on which CPU
- threads and processes can run concurrently on multi-core CPUs. **A single process or thread only runs on a single core at a time.** If there are more threads requesting CPU time than available cores (generally the case), the operating system scheduler will move threads on and off cores as needed.

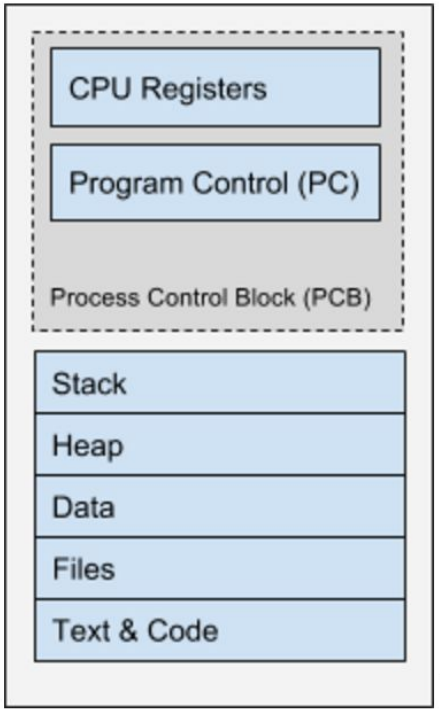
- OS doesn't much care which process the threads are from. It will usually schedule threads to processors / cores regardless of which process the thread is from. This could lead to four threads from one process running at the same time, as easily as one thread from four processes running at the same time.

Lets look at both Processes and Threads in details

Scaling on Software Level

What is a Process?
A process is an executing instance of a computer program. There may be multiple copies of the same program running simultaneously.
Process layout when loaded into memory for execution:

- **Text section** contains compiled code of the program logic.
- **Data** section stores global and static variables.
- **Heap** section contains dynamically allocated memory (ex. when you use malloc or new in C or C++).
- **Stack** section stores local variables and function return values
- **PCB:** the OS maintains a special table called **process control block (PCB)** to keep track of process state. PCB table contains various information about the process, the main two items are: **program counter (PC) and CPU registers.**
- PC points to the next instruction to execute
- CPU registers hold temporary execution information such as instruction arguments.

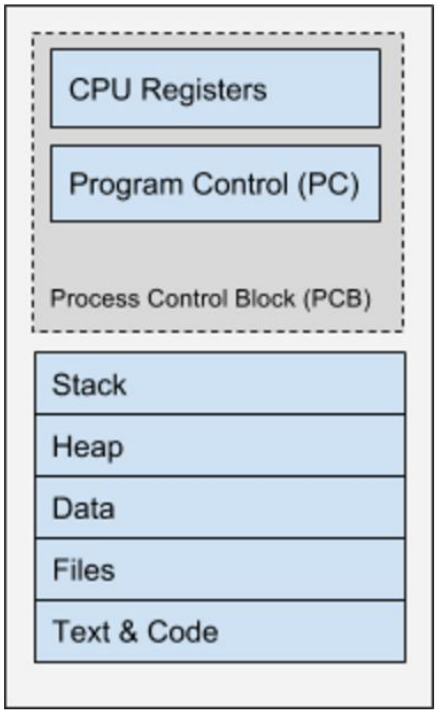


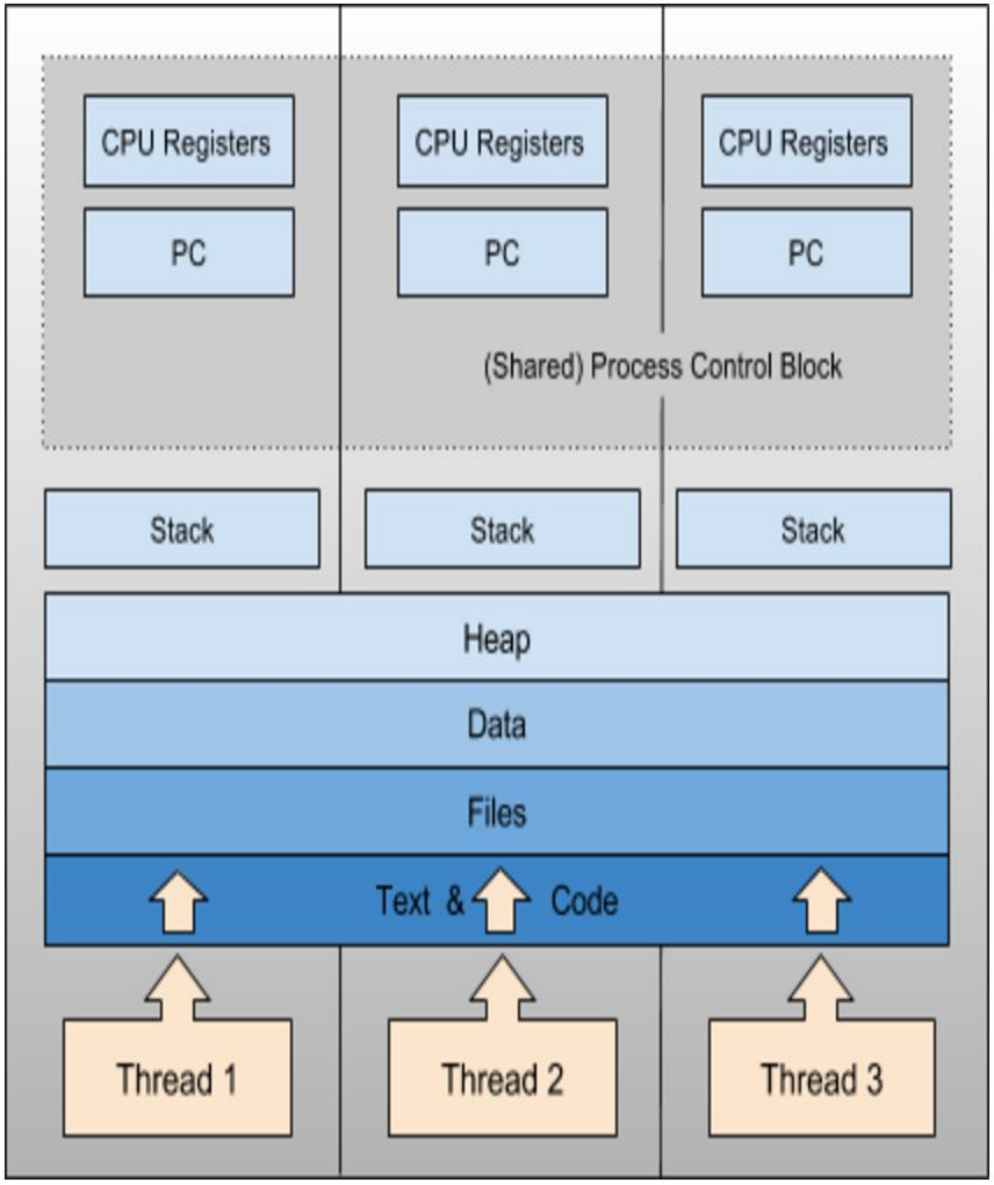
Scaling on Software Level

What is a Thread?

- A thread is a flow of execution through the process code
- From an OS perspective, just like a process, a thread has a private stack, program counter and a set of CPU registers.
- A thread is also called a **lightweight process** because it shares code, data, heap and open files with the parent process
- Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control.

One process, one thread memory layout vs. one process, three threads





@Marina Popova, Edward Sumitra

Scaling on Software Level

Context Switching

In a single processor system, there is no real multitasking. CPU time is shared among running processes. When the time slice for a running process expires, a new process has to be loaded for execution. **Switching from one process or thread to another is called context switch**.

Process context switch involves saving and restoring process state information including program counter, CPU registers and process control block which is a relatively expensive (in terms of CPU time) operation.

Similarly, **thread context switch** involves pushing all thread CPU registers and program counter to the thread private stack and saving the stack pointer. Thread context switch compared to process context switch is relatively cheap and fast as it only involves saving and restoring CPU registers.

Scaling on Software Level

There are different types of threads, which is defined by the thread management type:

User level threads are managed by a user level library and
Kernel (CPU core) level threads are managed by the operating system kernel code.
To better understand the difference between user level code and kernel level code, refer to the following article:
http://www.8bitavenue.com/2015/07/difference-between-system-call-procedure-call-and-function-call/

User Level Threads
Even though user level threads are managed by a user level library, they still require a kernel system call to operate.
The kernel does not have to and usually does not know anything about the user thread management. It only takes care of the execution part.

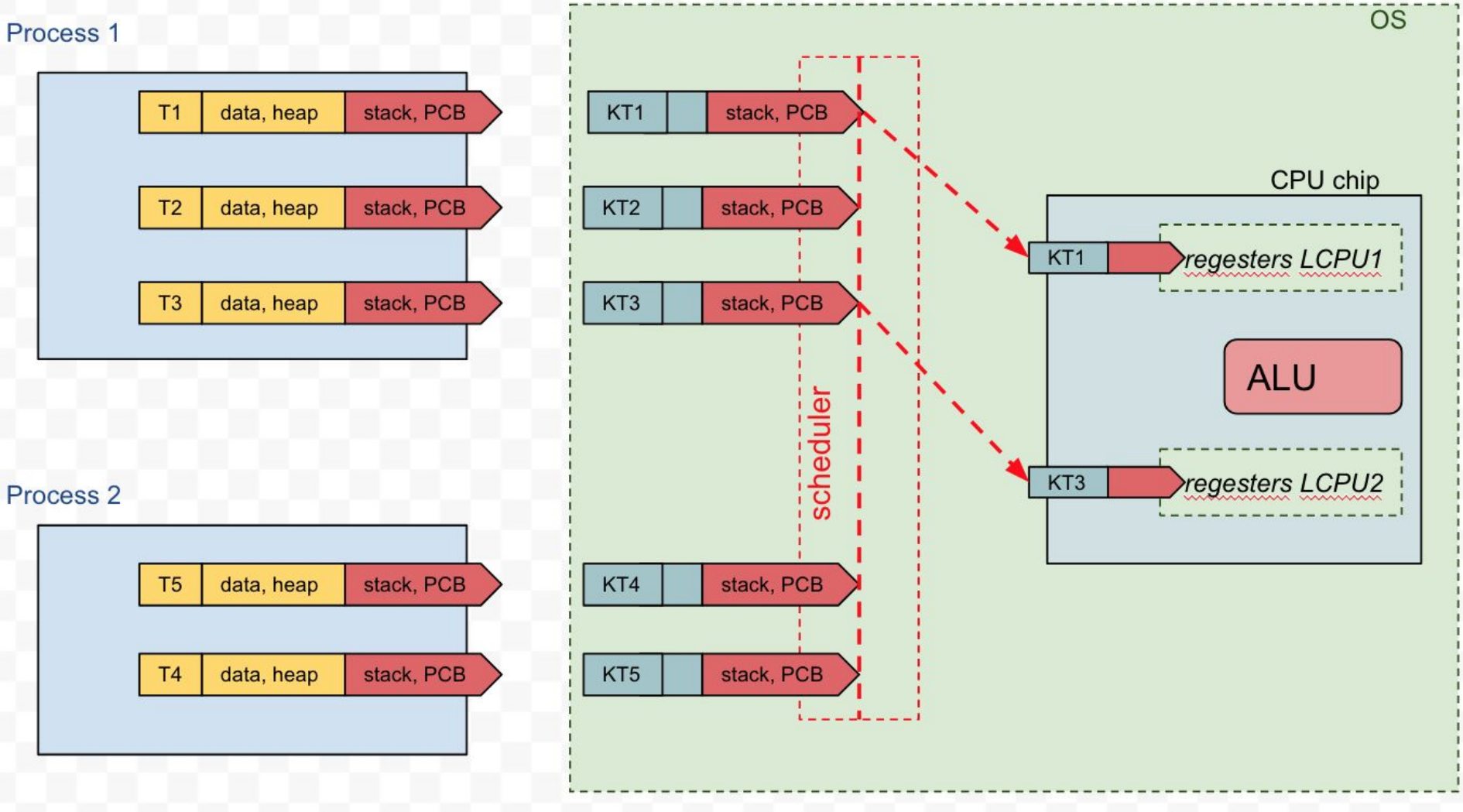
User level threads are typically fast. Creating threads, switching between threads and synchronizing threads only needs a procedure call.

Scaling on Software Level

Kernel Level Threads

- Kernel level threads are managed by the OS, therefore, thread operations (ex. Scheduling) are implemented in the kernel code. This means kernel level threads may favor thread heavy processes.
- **they can utilize multiprocessor systems by splitting threads on different processors or cores.**
- They are a good choice for processes that block frequently. If one thread blocks it does not cause the entire process to block.
- Kernel level threads are **slower** than user level threads due to the management overhead. Kernel level context switch involves more steps than just saving some registers.
- Finally, they are **not portable** because the implementation is operating system dependent.

Threads-to-Core Journey



@Marina Popova, Edward Sumitra

User Thread Management

- user-level thread management is done in your application code
- by using specific language libraries
- thread management usually involves management of access to shared data
- as well as management of the control flow between threads
- we will review how this is done in Java and Scala
- in this class - we will learn how FP style of programming makes these tasks much easier to do

Scaling Beyond One Server

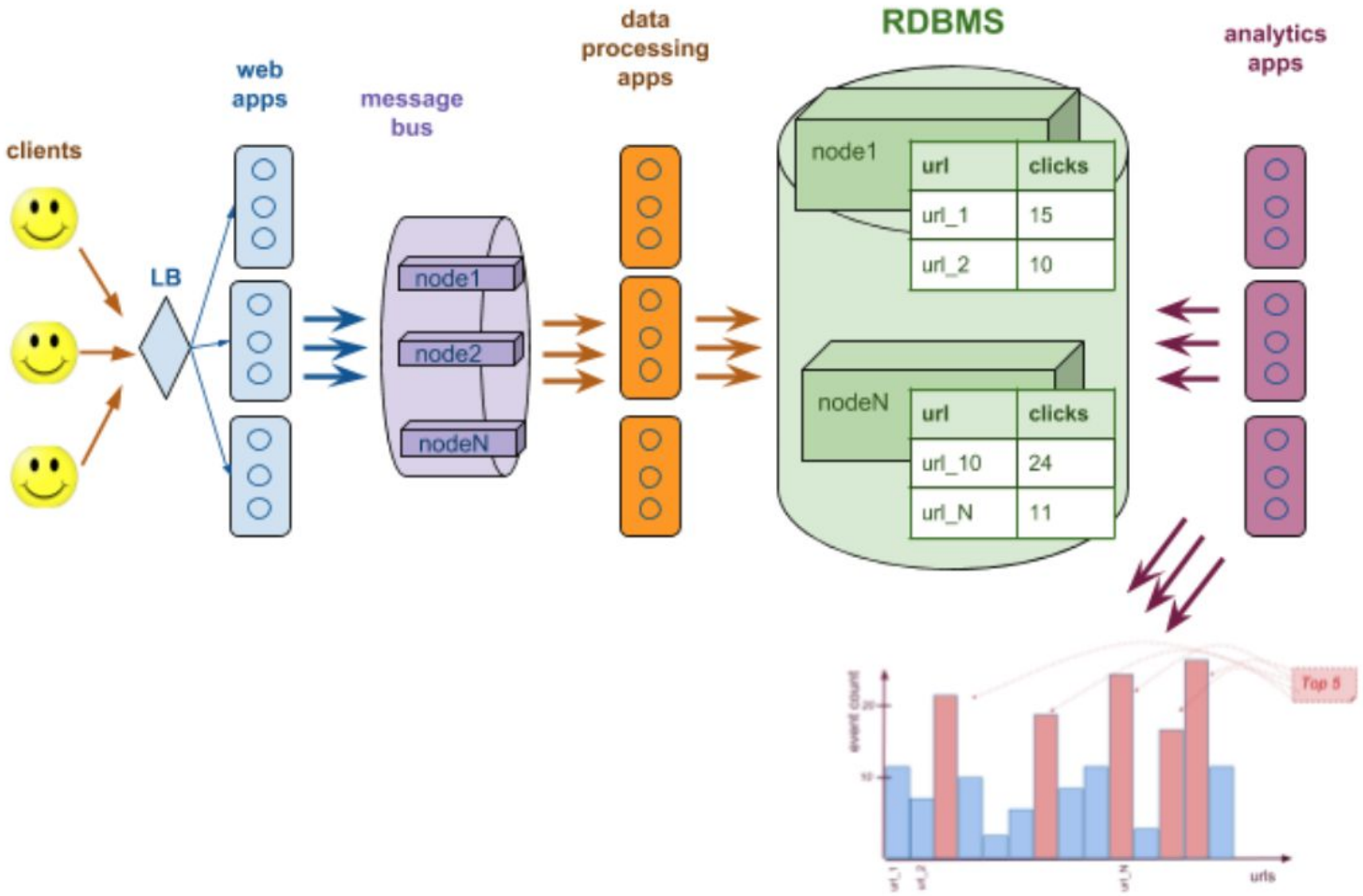
At some point, your application will not be able to scale any further vertically on software (by adding more threads) and hardware (adding more resources) levels ... Why??

Because of the single-server scaling limitations:

- hardware limits are reached
  - CPU/Cores are saturated
  - IOps saturated
- no more operations can be performed on the single physical server

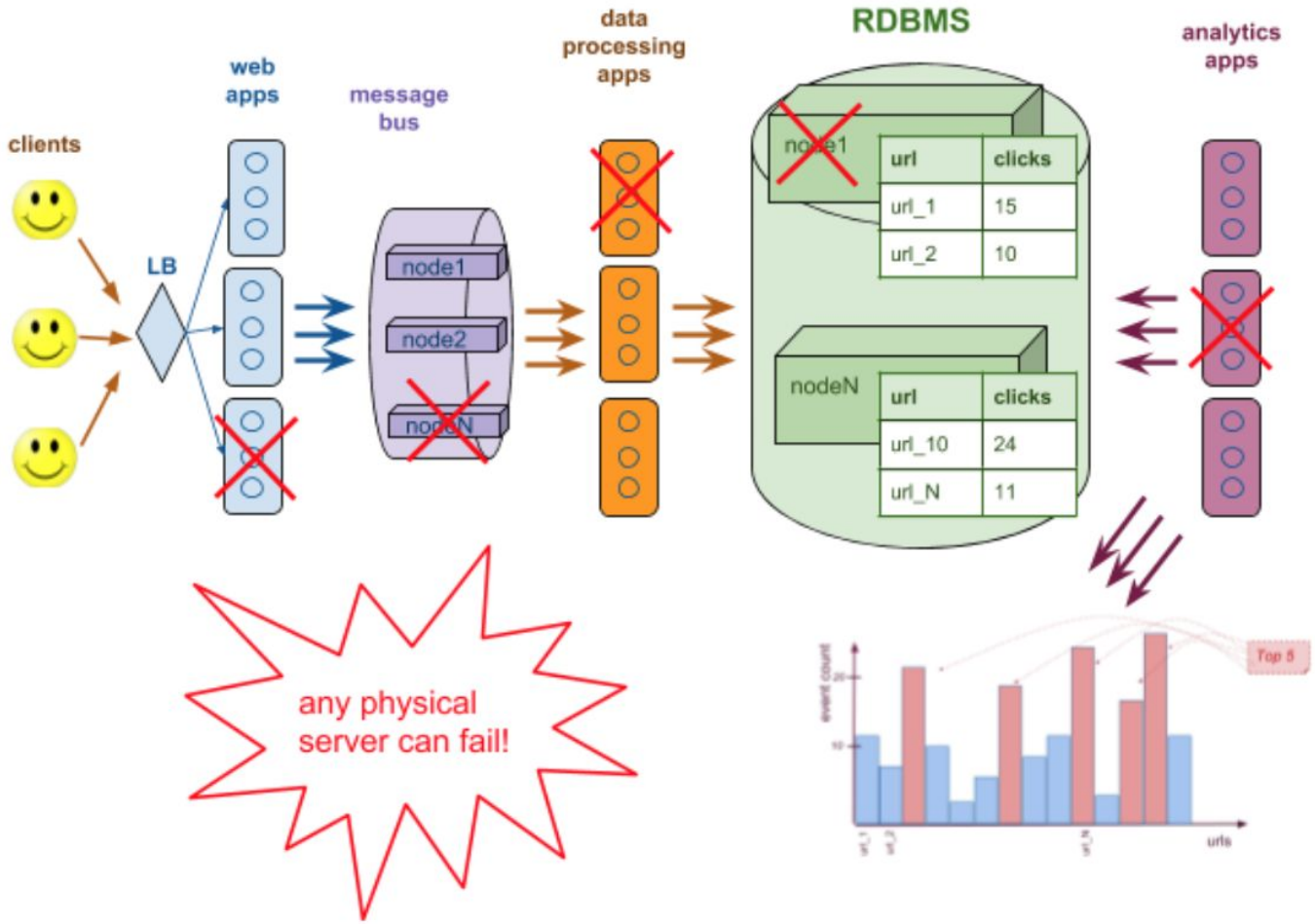
What do we do NEXT ?

... we move to a Distributed Architecture!



Moving to a distributed architecture is not a piece of cake ...

WHY?



This means that additional requirements have to be considered and met ...

Scaling For Real

Main Challenges and Goals building Distributed systems:

1. **Scalability** - how to distribute processing and data over more and more physical servers?
2. **Fault Tolerance/ Availability** - how to survive failure of individual nodes
3. **Data Consistency** - is my data correct, distributed over multiple nodes? and when some nodes die?
4. **Data Latency** (staleness) - do I collect/store/process incoming data as soon as it is generated?
5. **Query response timings** - how long does it take to query [and visualize] my data?

How do we achieve all this?

we have to address Scaling of both Data Processing and Data Storage for real

The focus of this class is on scaling the Data Processing part

what does it mean to scale to a multi-server setup? and how does it relate to Small vs Big data processing ?

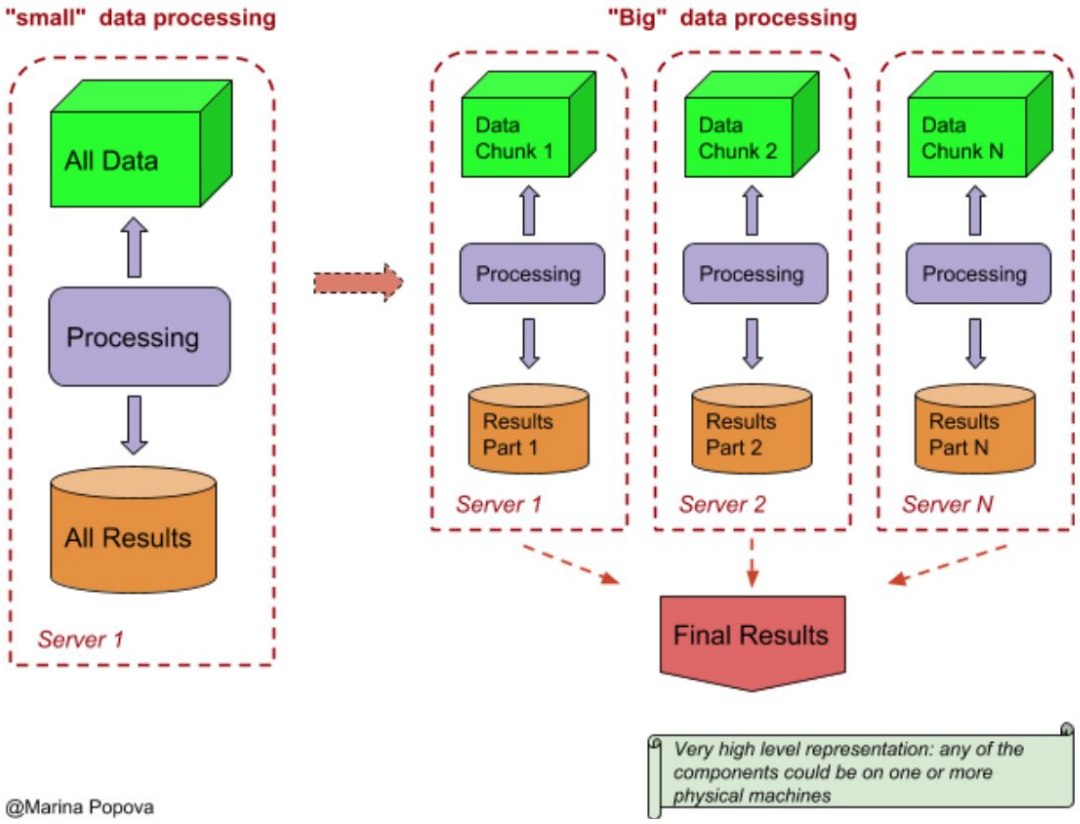
Scaling For Real

The main difference between processing "Big Data" vs "small data" is that it is no longer feasible to do this using a single physical server's CPU and RAM, for many reasons.

This means, **we have to split the data, computations and results between multiple physical servers** and potentially get the final results by combining the partial results from those servers.

And at the same time meet requirements for:

- Scalability
- Fault Tolerance
- Data Consistency
- Data Latency
- Query Performance



Scaling For Real

What we saw in the last example is the next step in scaling - called XYZ scaling:

It is a variation of horizontal / vertical scaling where we introduce data splitting, so certain types of data or certain operations are located across systems (horizontally) which are optimized for those types of loads (vertically)

X-axis scaling is pretty much traditional horizontal scaling, which distributes the total load across a given number of nodes.

Y-axis and Z-axis scaling, however, are **two entirely different approaches** by focusing on different things that can be scaled.

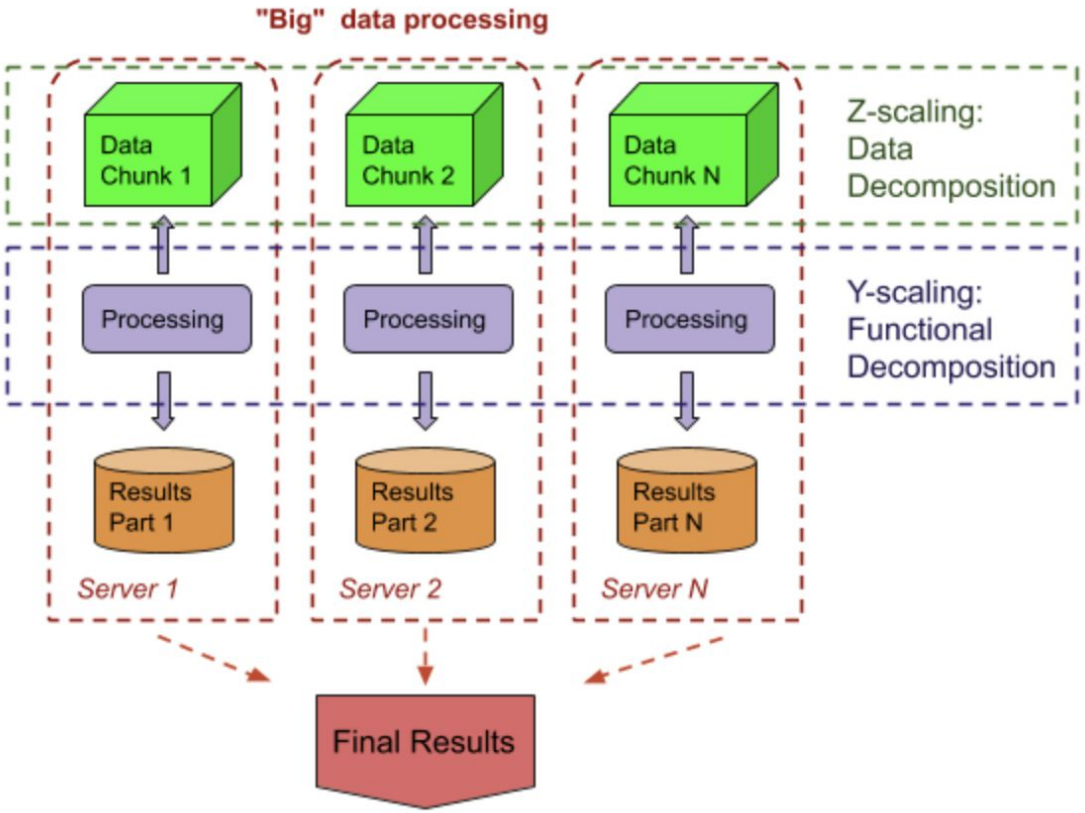
Scaling: X, Y, Z scaling

Y-axis scaling refers to breaking out and distributing services. This is called *Functional Decomposition*, and it is a design approach which is reflected in ***service-oriented and microservices architectures***.

Z-axis scaling refers to data partitioning and is also referred to as *Data Decomposition*. It is a way of distributing data among many nodes or blocks as **a way of improving performance**. The underlying data sets contain the same type of information, but any given partition only contains part of the information.Multiple threads can have the same code (instructions) being executed but they are using different sets of [localised] data

Both functional decomposition and data decomposition are core principles used in the **map-reduce type parallelization** frameworks and, as you will see later, are the underlying core concepts enforced and enabled by the Functional style of programming

XYZ Scaling in Action



enables Big data processing

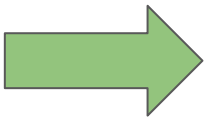
enables parallel processing

FP Core Principles and YZ-scaling

How does YZ-Scaling relate to FP? As it turns out - directly !!!

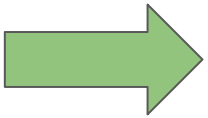
The core principles of FP make it possible to implement applications that can be YZ-scaled!

pure functions:
- no side effects - deterministic and idempotent
- same input -> same output
- no external state is used



Y-Scaling:
Functional Decomposition:
- operations can be distributed and parallelized

Immutable data
- no assignments
- no variables



Z-Scaling:
Data Decomposition:
- data can be partitioned and replicated

FP control flow style:
- no loops
- no **imperative** instructions on how to do something
- instead, **declare** behavior via functions

So, how do we do it?

Before we go further into theoretical concepts, lets consider an example of scaling a very simple operation:

~~Herding~~ counting cats!

Our task:
How many black cats with green eyes do we have?

Cat Aggregator

CatAggregator - count cats by color:

Model class: Cat.java

Imperative way:

```
long numOfCats = 0l;
for (Cat cat: allCats) {
    if (cat.getBodyColor().equalsIgnoreCase(bodyColorToMatch) &&
        cat.getEyeColor().equalsIgnoreCase(eyeColorToMatch)) {
        numOfCats++;
    }
}
```

```
public class Cat {

    private String name;
    private String eyeColor;
    private String bodyColor;
```

Functional way:

```
long numOfCats = allCats.stream().parallel()
    .filter(cat -> cat.getBodyColor().equalsIgnoreCase(bodyColorToMatch))
    .filter(cat -> cat.getEyeColor().equalsIgnoreCase(eyeColorToMatch))
    .mapToLong(cat -> 1l)
    .reduce(0, (partialSum1, partialSum2) -> partialSum1 + partialSum2);
```

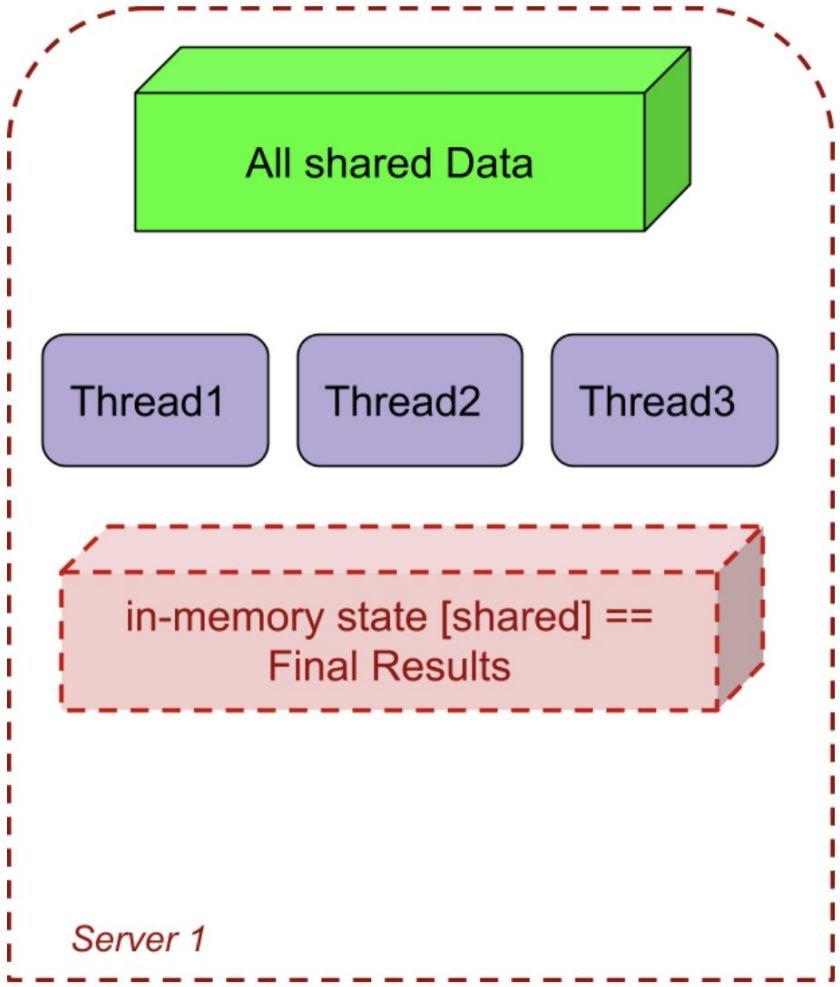
Scaling with FP - single server, shared mutable state (for imperative code)

Imperative way:

```
Collection<Cat> allCats
```

```
for (Cat cat: allCats) {
    if (cat.getBodyColor().e
        cat.getEyeColor().eq
        numOfCats++;
    }
}
```

long numOfCats




```
Collection<Cat> allCats
```

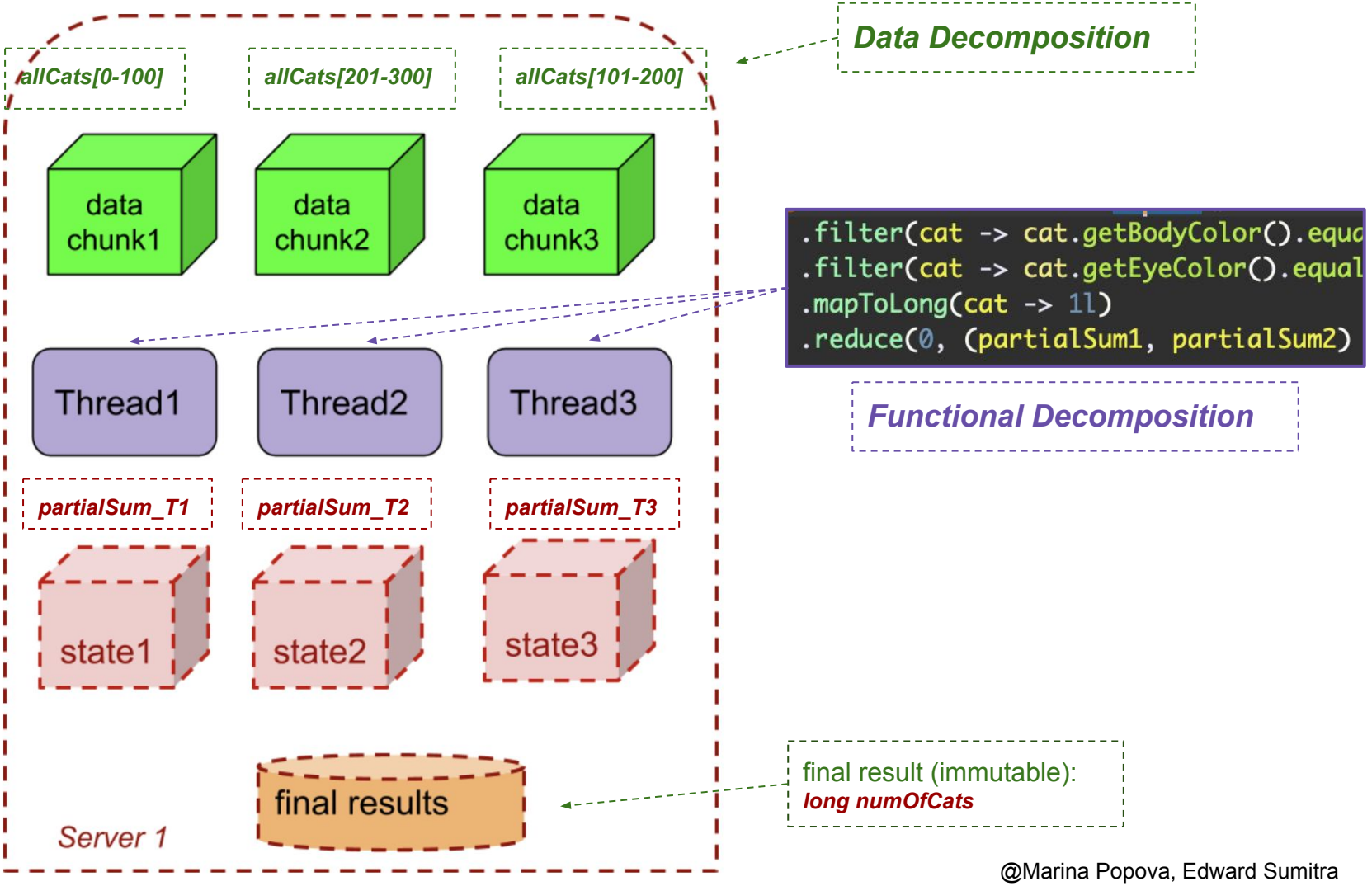
```
long numOfCats = allCats.stream()
    .filter(cat -> cat.getBodyColor()
    .filter(cat -> cat.getEyeColor()
    .mapToLong(cat -> 1l)
    .reduce(0, (partialSum1, partialS
```

Functional way:
not really applicable here - somewhat like when the stream execution is sequential, but there is still **no shared mutable data**

Scaling with FP - single server immutable per-thread state

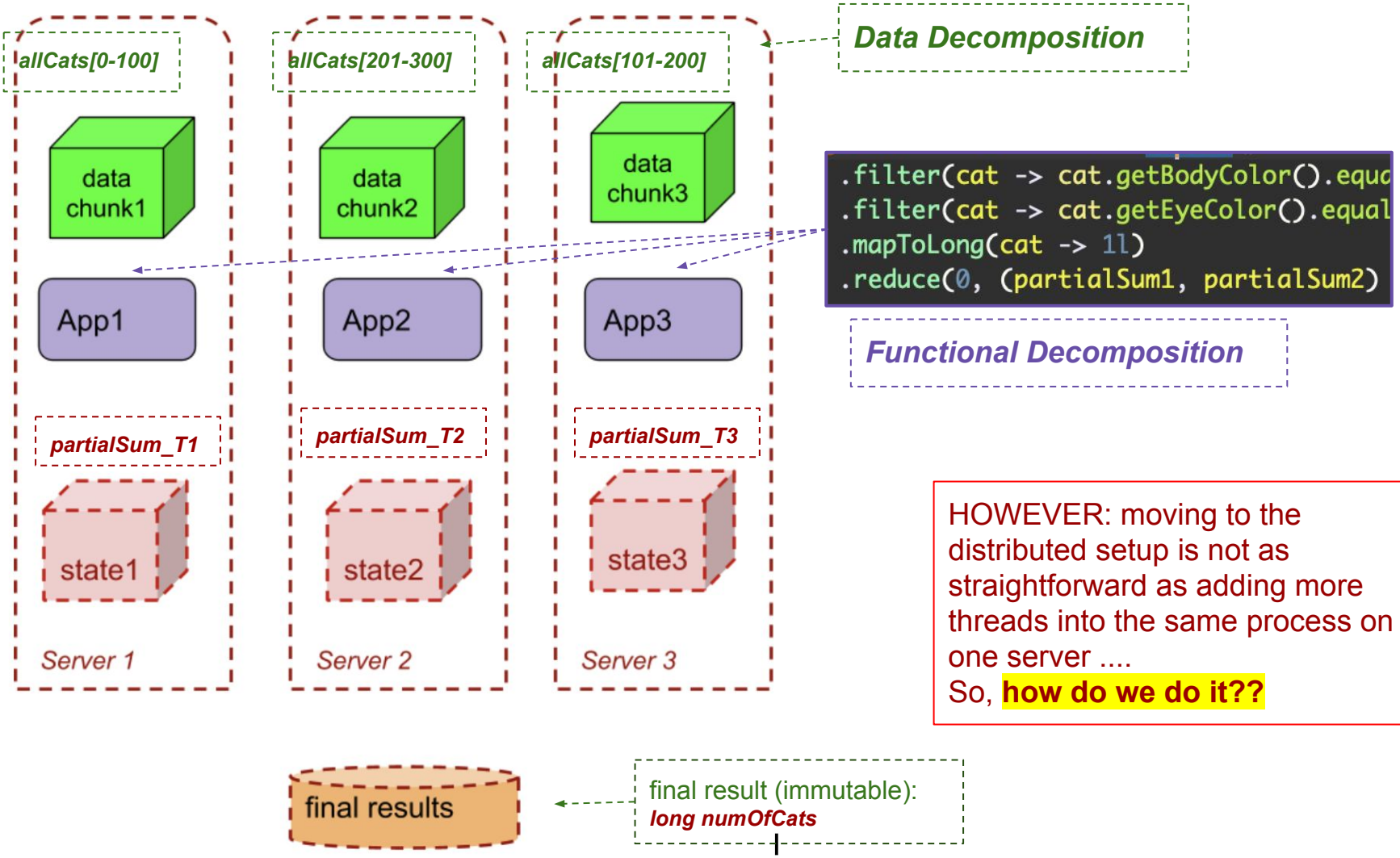
Imperative way - will NOT work here!

Functional way:



Scaling with FP - multi-server immutable per-process state

Functional way: the same code can be moved from single-server to multi-server setup!



HOWEVER: moving to the distributed setup is not as straightforward as adding more threads into the same process on one server ....
So, **how do we do it??**

How do we implement this type of scaling?

First, why is this so hard?

Because your system has to take care of many things:

- **split all data into chunks and move to separate physical servers**
- **distribute your "functionality" == application to the same (or not) servers**
- **initiate and coordinate execution of your application (== "jobs") on each server**
- **wait for all of them to be done to "construct" the final result**

  and: any server can die at any moment!!

- **monitor for any kind of crash/issue - and ensure data and jobs are recovered**

@Marina Popova, Edward Sumitra

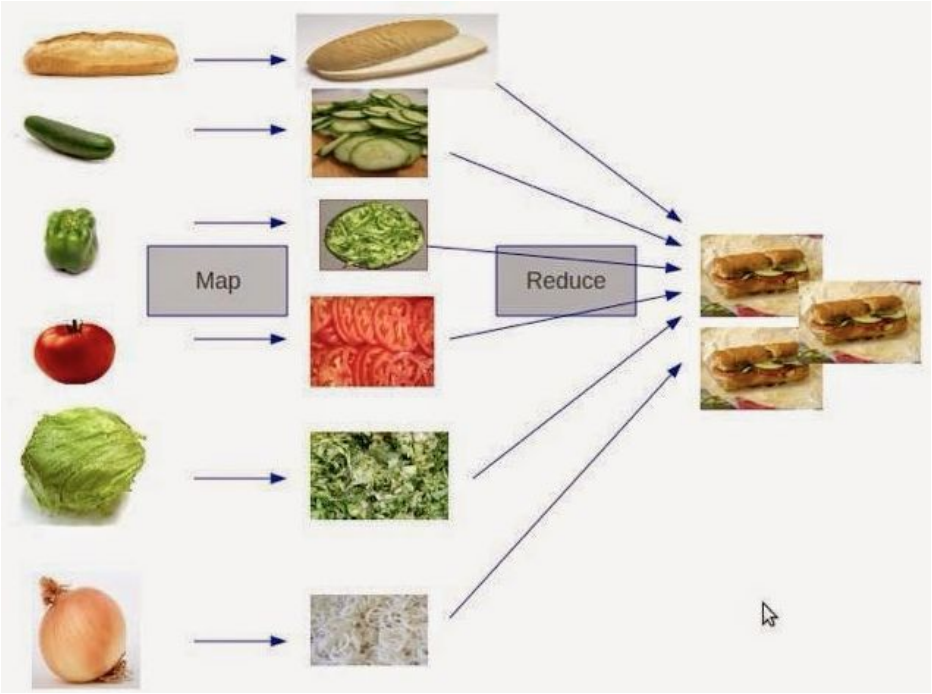
Good News!!!
It's a well known problem with a well known solution...

... and the solution is: MR + special Frameworks

Map-Reduce paradigm:
A way to process large volumes of data in parallel by dividing the work into a set of independent tasks.

In essence, it is a set of **Map** and **Reduce** tasks that are combined to get final results:

- Map function transforms the piece of data into **key-value pairs** and then the keys are sorted
- Reduce function is applied to **merge the values** based on the key into a single output

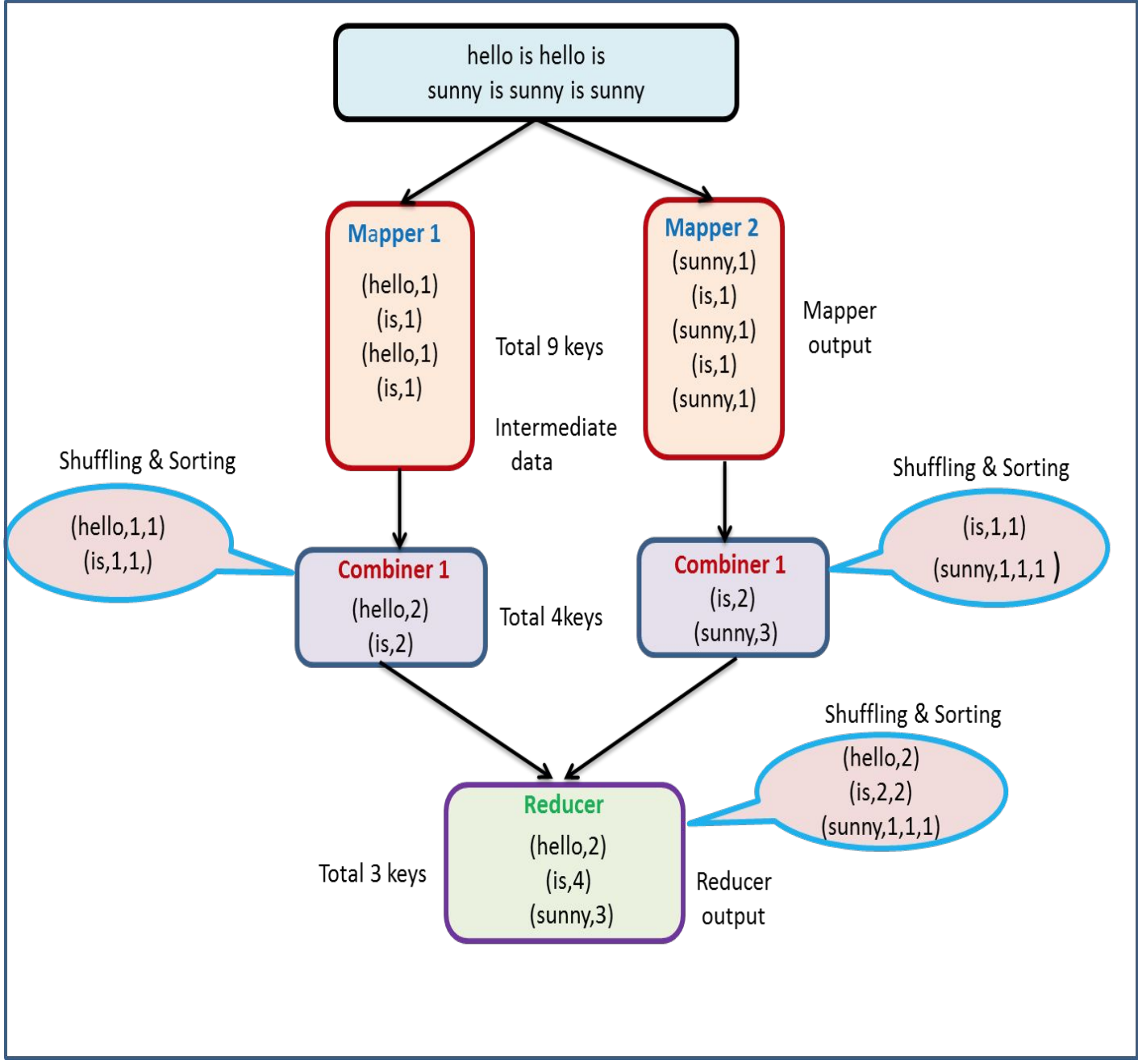


Picture source:
https://www.datasciencecentral.com/forum/topics/what-is-map-reduce

MR Execution Flow:

1. Input splitting
2. Task allocation
3. Map phase
4. Combiner phase
5. Partition phase
6. Shuffle / Sort phase
7. Reduce phase

we will not go deep into the details of each stage - but we will come back to the MR concepts later on



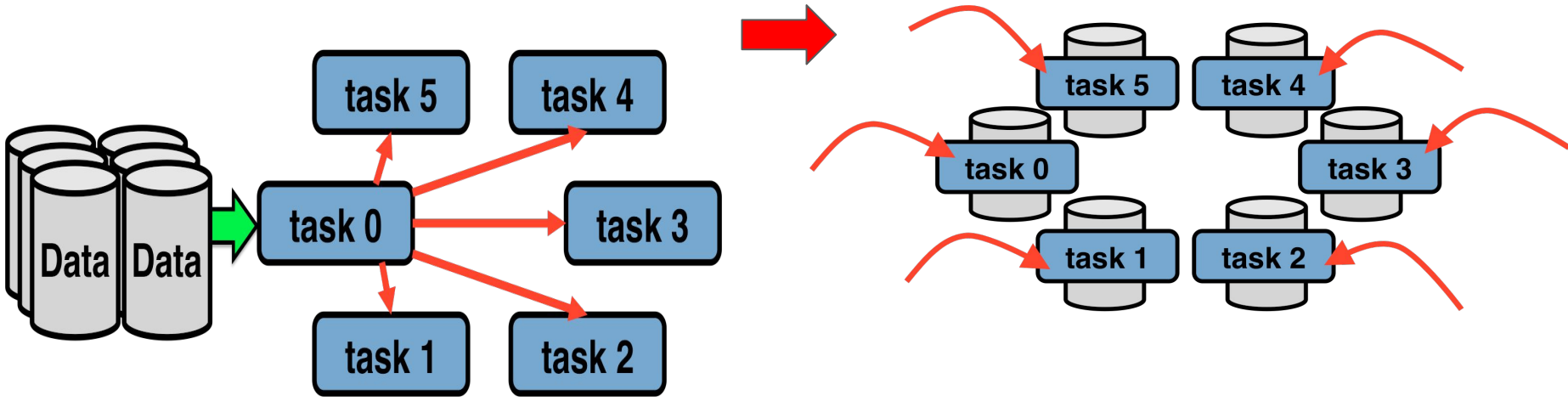
References: http://data-flair.training/blogs/hadoop-mapreduce-introduction-tutorial-comprehensive-guide/

Big Data Processing - scaling out

What we observed is a well-known distinction between **Traditional Parallel CPU-intensive** applications and **Data Intensive** applications that use a different way to parallelize computing: Map-Reduce paradigm

CPU-intensive
Data is stored in large storage
Data is brought to Compute

Data-intensive
Data is stored in chunks on compute nodes
Compute is brought to Data



Ref: https://www.glennklockwood.com/data-intensive/hadoop/overview.html

MapReduce Implementation - examples

Even using the MR concepts - it is not easy to write your own applications using it!

This is why there are many libraries and frameworks that are developed to help with that:

- Java: (single server)
  - ForkJoinPool
  - Java 8 streams
- Spark
- Hadoop
- AWS Athena
- many others ....

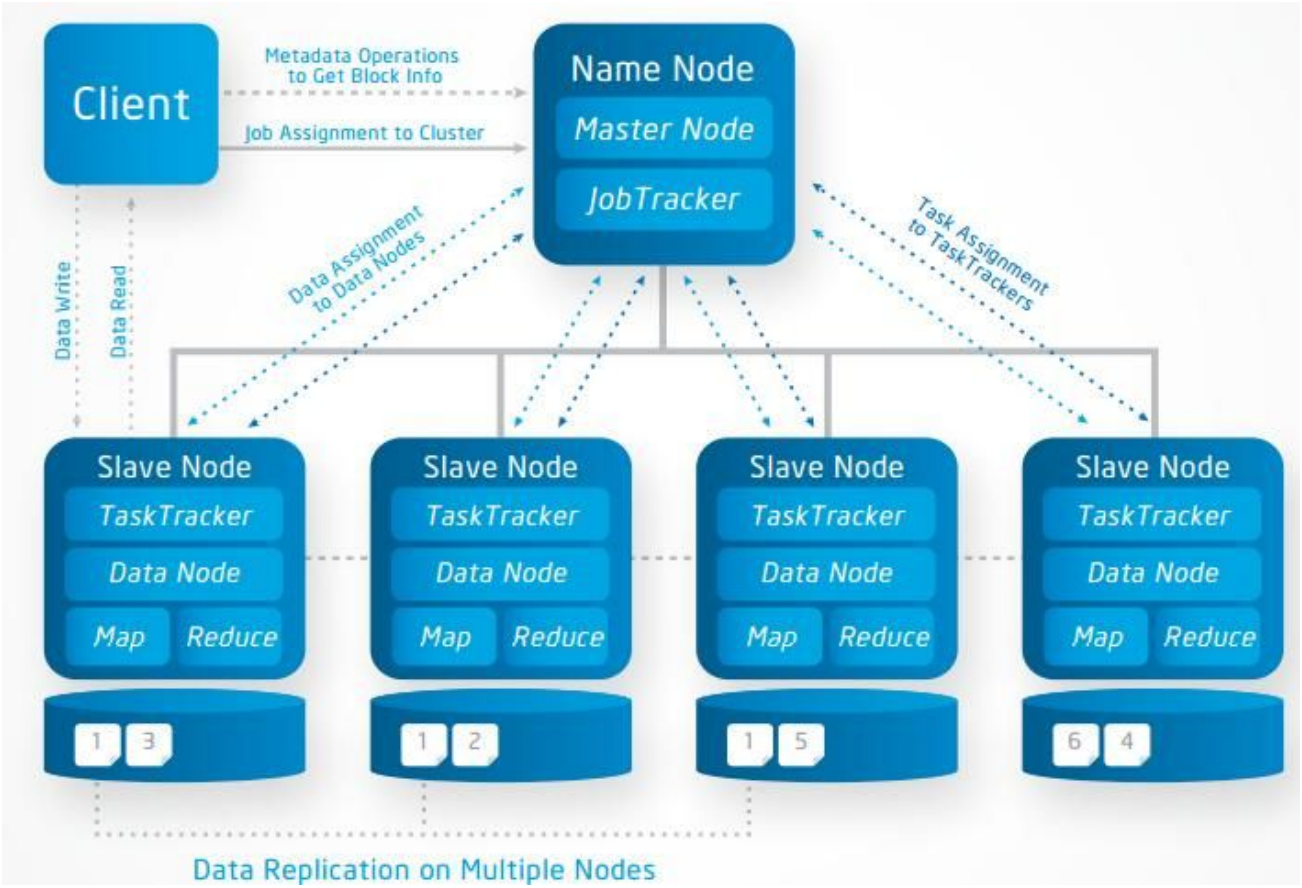
We will take a quick look at how it works in the Hadoop MR framework.

MapReduce Implementation - Hadoop MR

The MapReduce framework consists of:

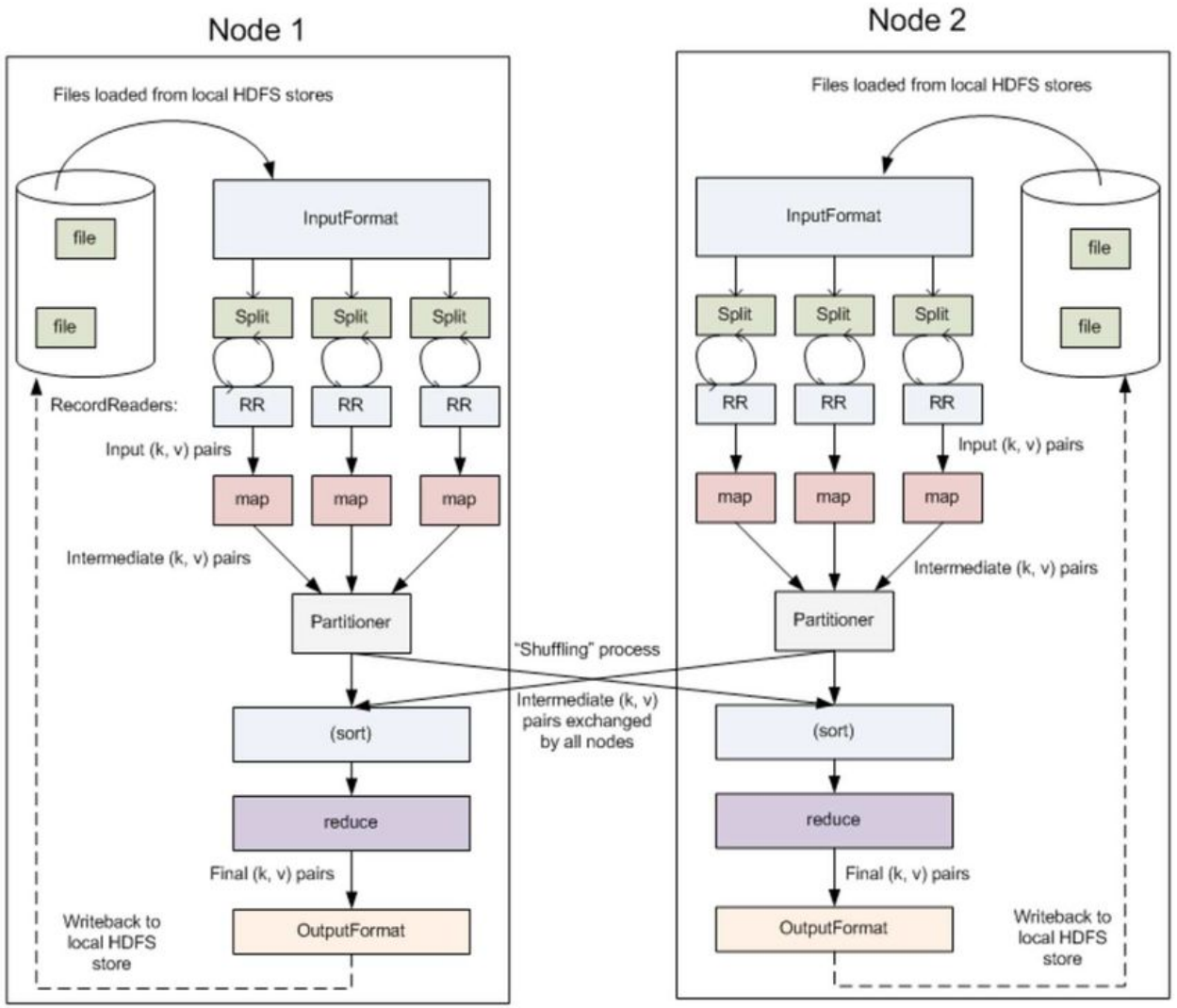
- a single master JobTracker (or ResourceManager in v2)
- one slave TaskTracker (NodeManager in v2) per cluster-node
- MRAppMaster per application

- input data chunks are stored on Data Nodes (HDFS)
- jobs (functionality) are run on the same node as the chunks of data assigned to them



Lets see a bit more detailed view of a MR job execution in Hadooop:

Ref: https://www.researchgate.net/figure/Detailed-Hadoop-MapReduce-Data-Flow-14_fig1_224255467



It's applicable to pretty much any MR-implementation, it is just a conceptual view

So, how are Functional Programming and Big Data / MR Processing related?

They go hand in hand ....

@Marina Popova, Edward Sumitra

Scaling of Data Processing using FP

both Functional decomposition and Data decomposition are core principles used in the **map-reduce type parallelization** frameworks - which is the foundation and the only way any truly distributed (both batch and stream) processing can be realized

Also, both Functional and Data decompositions are the underlying core concepts enforced and enabled by the Functional style of programming

This is why it is very important to use functional programming concepts and frameworks to implement distributed processing

Great posts that highlight different areas of FP applicability to the Big Data World::
https://medium.com/@maximebeauchemin/functional-data-engineering-a-modern-paradigm-for-batch-data-processing-2327ec32c42a
https://medium.com/@WomenWhoCode/3-design-principles-for-engineering-data-9d03dcb1711f

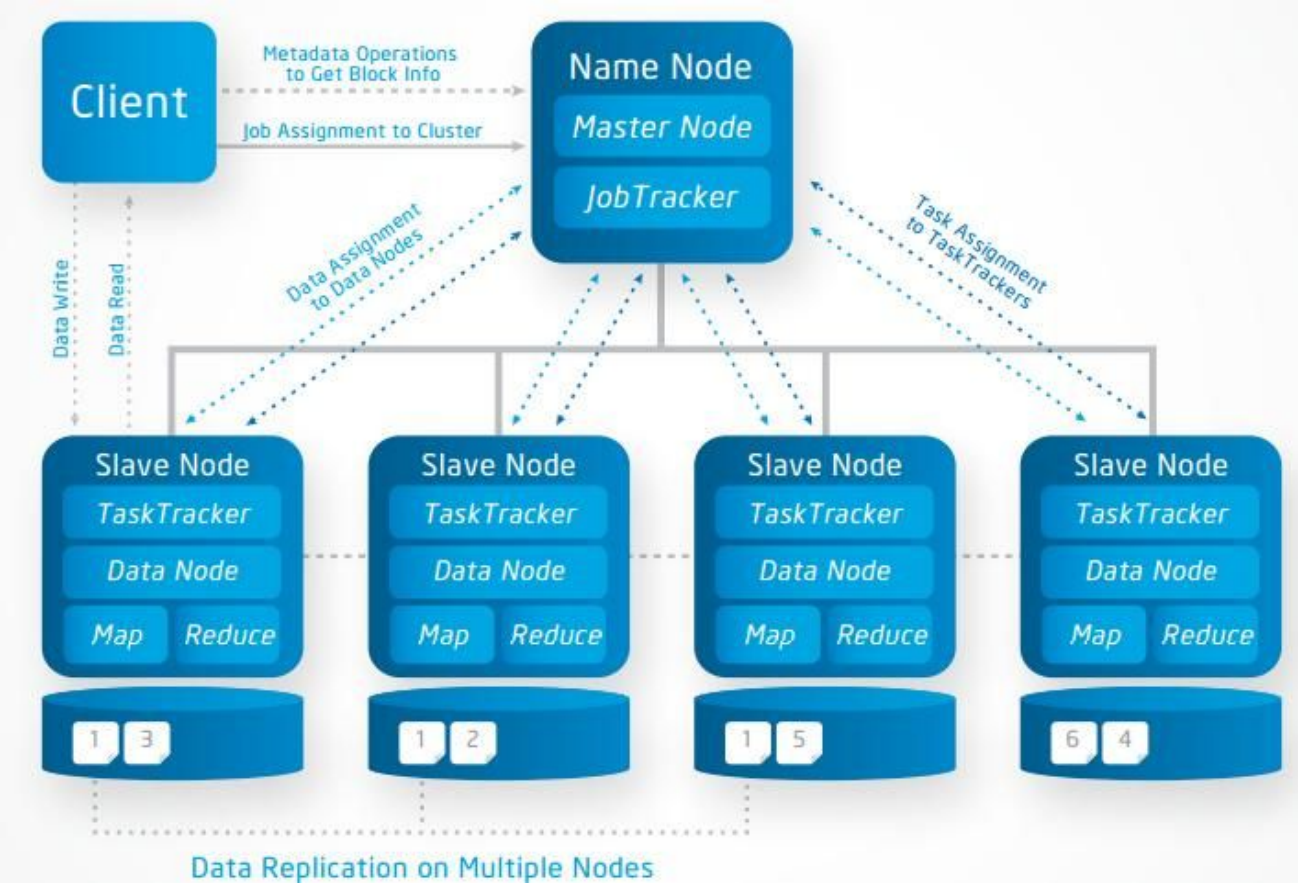
Lets see how our earlier Cat counting example applies here ...

Imperative way:

```
long numOfCats = 0l;
for (Cat cat: allCats) {
    if (cat.getBodyColor().equalsIgnoreCase(bodyColorToMatch) &&
        cat.getEyeColor().equalsIgnoreCase(eyeColorToMatch)) {
        numOfCats++;
    }
}
```

Will not work as a MR job!
[without additional changes]

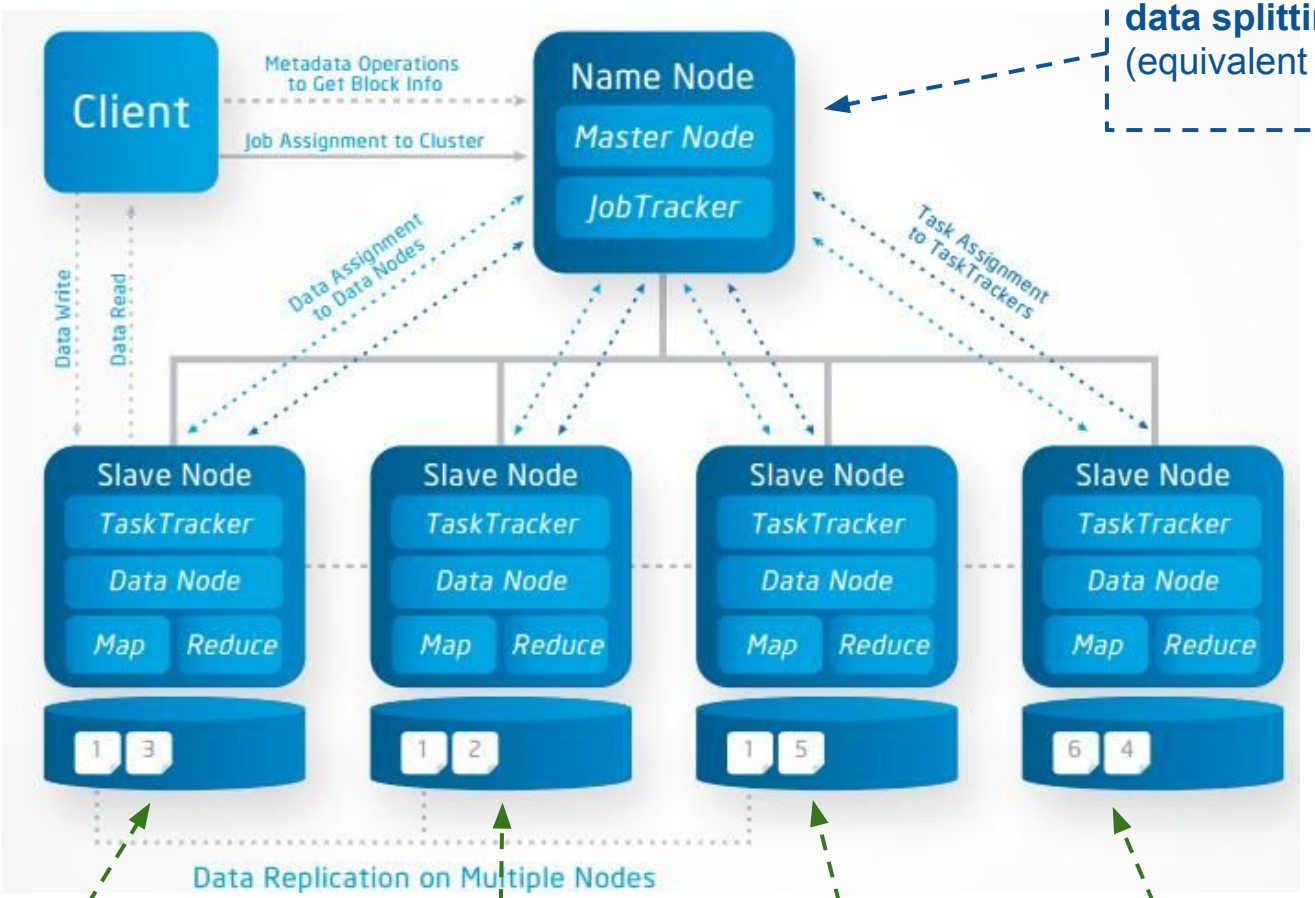
- input cannot be split
- there is shared external state
- operations are not "pure functions" - not idempotent --> cannot be parallelized



Functional way:

```
long numOfCats = allCats.stream().parallel()
    .filter(cat -> cat.getBodyColor().equalsIgnoreCase(bodyColorToMatch))
    .filter(cat -> cat.getEyeColor().equalsIgnoreCase(eyeColorToMatch))
    .mapToLong(cat -> 1l)
    .reduce(0, (partialSum1, partialSum2) -> partialSum1 + partialSum2);
```

Note: the exact syntax would change slightly to use the specific MR frameworks APIs ...



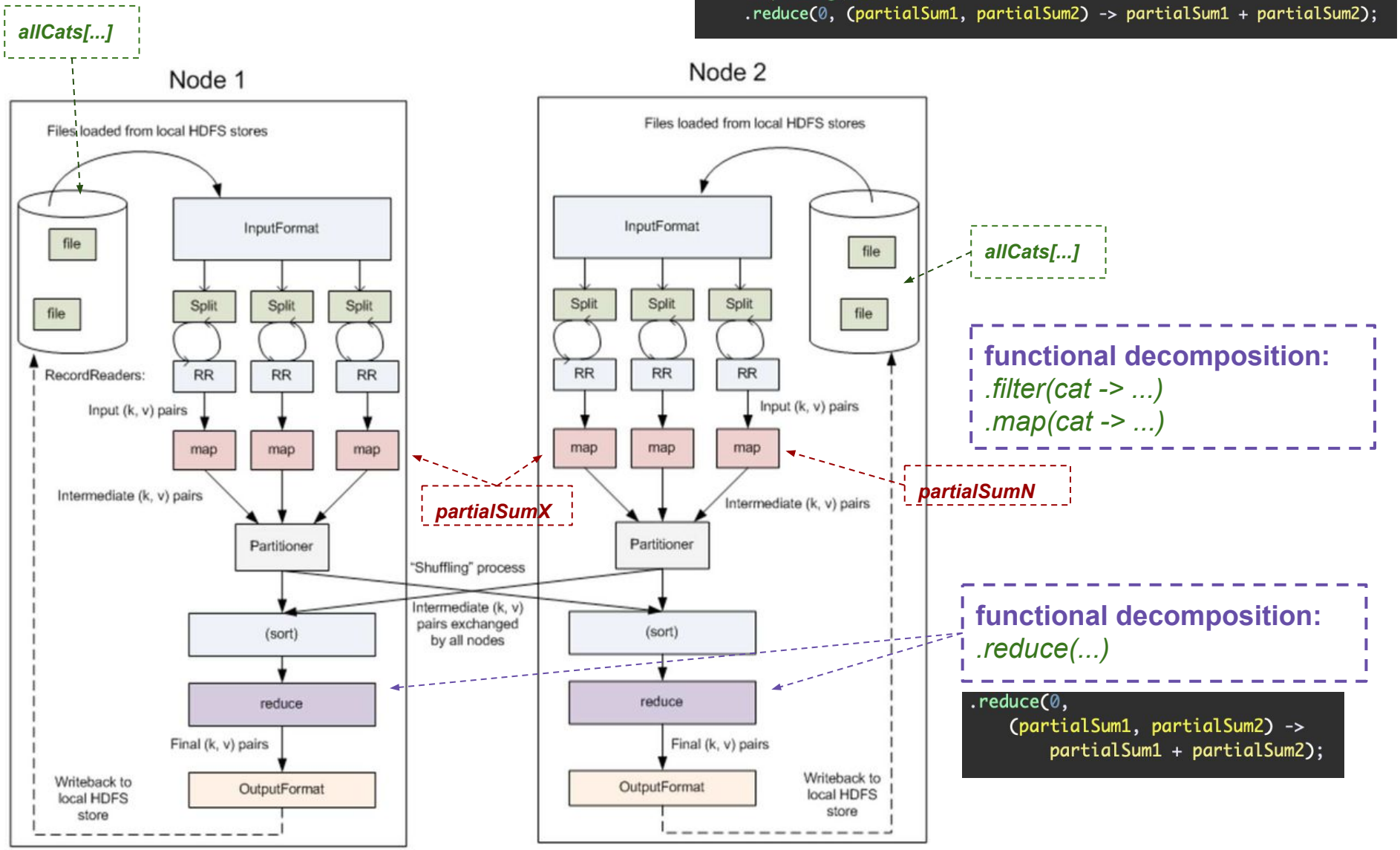
data splitting and task distribution
(equivalent of *allCats.stream().parallel()*)

functional decomposition:
.filter(cat -> ...)
.map(cat -> ...)
.reduce(...)

data decomposition

Functional Decomposition in MR - details

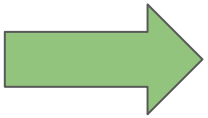
```
long numOfCats = allCats.stream().parallel()
    .filter(cat -> cat.getBodyColor().equalsIgnoreCase(bodyColorToMatch))
    .filter(cat -> cat.getEyeColor().equalsIgnoreCase(eyeColorToMatch))
    .mapToLong(cat -> 1l)
    .reduce(0, (partialSum1, partialSum2) -> partialSum1 + partialSum2);
```



Scaling of Data Processing using FP - Finale

So, what, specifically, makes FP concepts so fitting for BDP?

FP principles:
- immutable data
- pure functions with no side effects - deterministic and idempotent



BDP concepts - via MR
- operations parallelization
- idempotent operations
- data partitioning and replication

BDP requirements:
- scalability
- fault tolerance
- data consistency
- data latency
- query performance (maybe)

BDP properties:
- ability to re-process
- ability to reproduce behavior/ issues
- single unit of work == single unit of output (partition)

Class Goals - continue

Before you start panicking ....

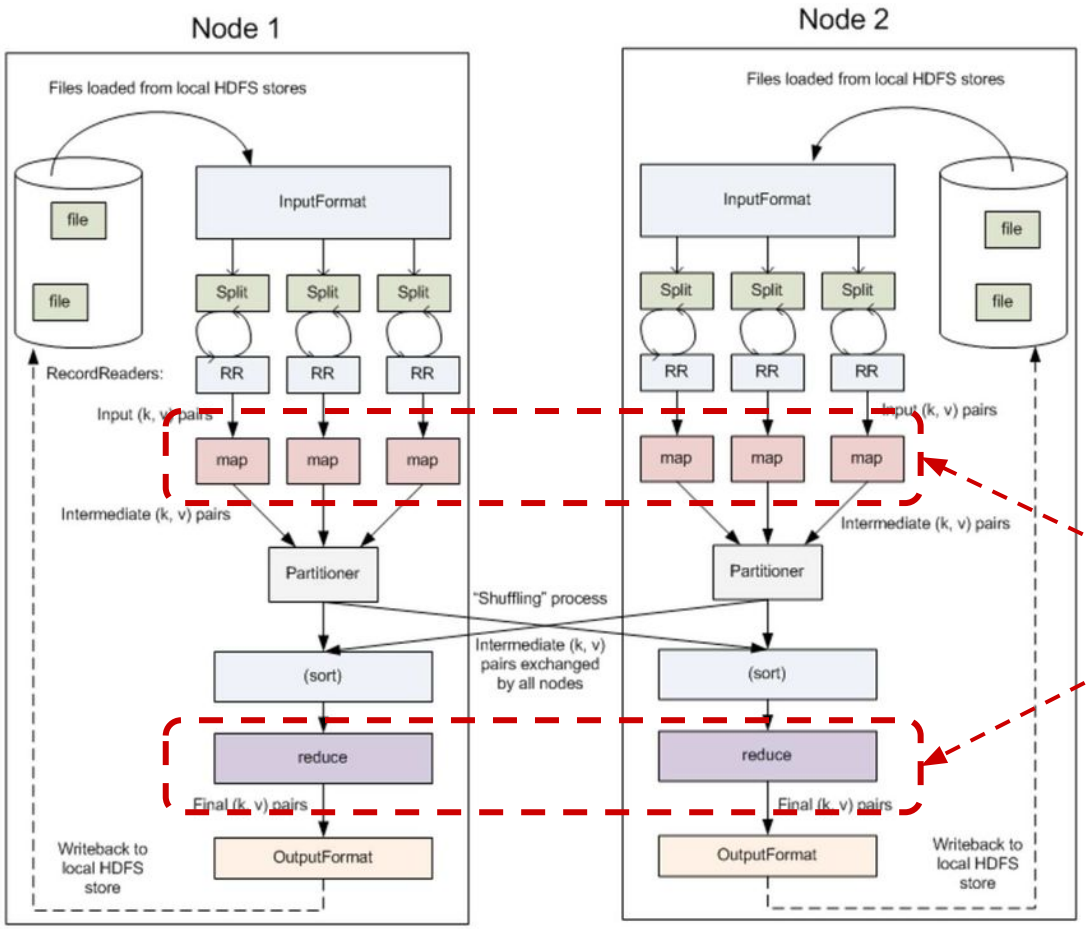
we will not be learning all of the above in this class!

what, and how, will we learn then?

Class Goals - continue

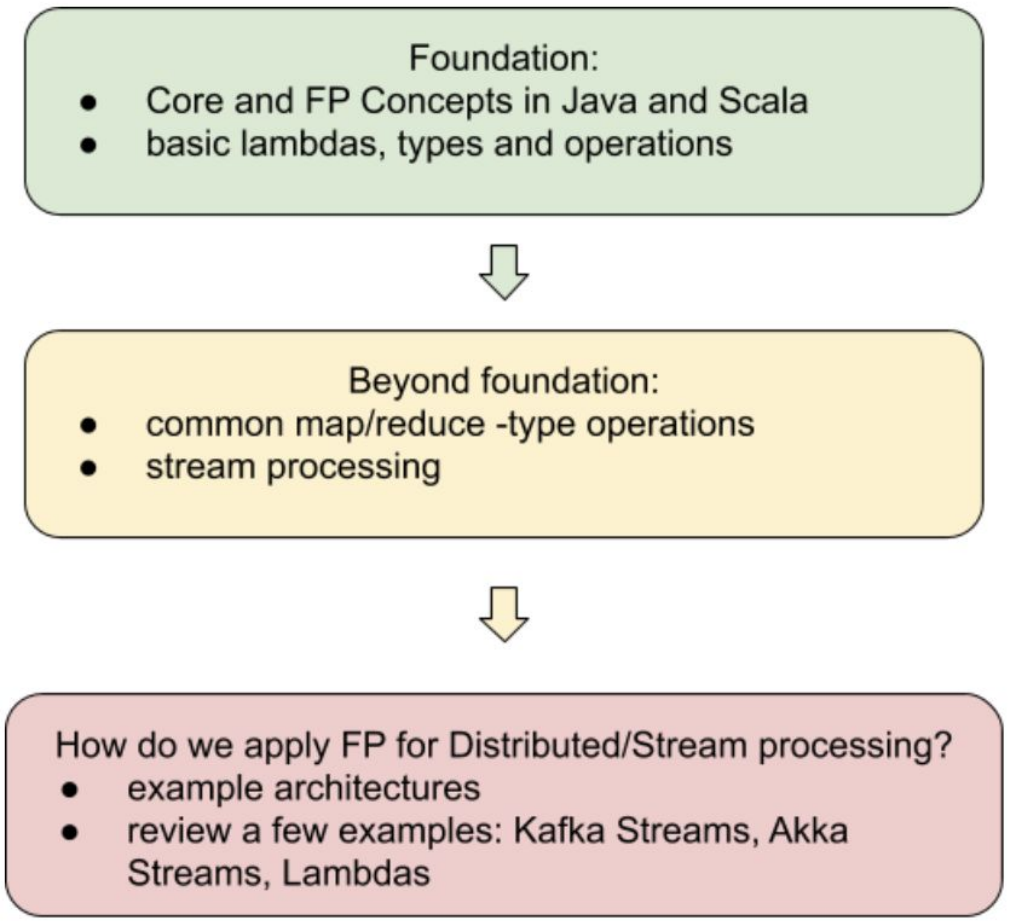
We will be learning the foundation!

- how to write common operations using FP style
- focus on the concepts that are most relevant to distributed processing - which are falling under map() and reduce() - type of operations



this means mostly these areas

Class Roadmap



Class Logistics - Homeworks

- mostly weekly
- focus on mastering/exercising specific concepts of interest for that week
- effort goal: 2-6 hrs max per week
- lost of starter code provided

we are not here to torture you! :-)

- late policies: 2 days, with 15% penalty each
- bonus problems - for extra challenge
    - voided after Due Date

Class Logistics - Dev and Test setup

- provided Gradle scripts/builds for Java
- sbt for Scala
- Class code in GitHub: https://github.com/cscie88-ta/CSCIE88AFP-2020
- Assignment verification: Unit tests and execution on Docker containers
- Assignment submission:
- later in the class: will use AWS - with credits through AWS Educate program
- Labs: will demo a lot of this setup and examples

Class Logistics - continue

- Piazza!
- Polls and surveys for feedback
- Labs: ask questions!
  - mostly Sat, 8:30-9AM
  - Zoom - via the "Web Conferencing" link on Canvas
- Canvas: all class materials will be posted here

And remember - this is the first time this class is taught!
Be patient with bumps and hiccups and tmp obstacles ! :-)