



Introduction to Map/Reduce: From Hadoop to SPARK

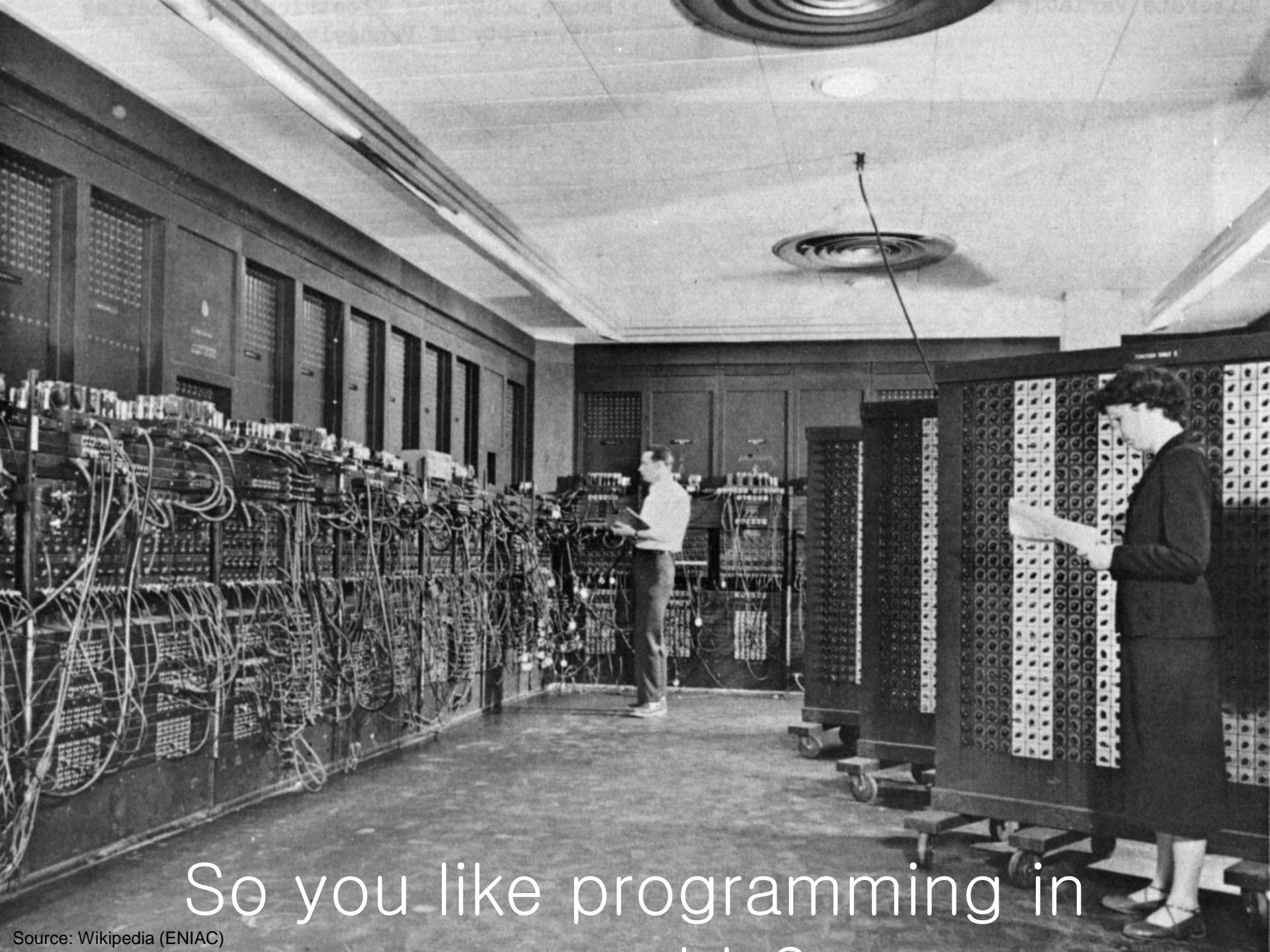
Serafeim Mustakas
Computer Science Department
University of Crete, Greece

What we will cover...

- Dataflow Languages for Cluster Computing
- What is MapReduce?
- How does it work?
- A simple word count example
 - (the “Hello World!” of MapReduce)
- From MapReduce to Spark

An aerial photograph of a large industrial or data center facility during sunset. The sun is low on the horizon, casting a warm orange glow over the scene. The facility consists of several large, white, rectangular buildings with flat roofs, arranged in a grid-like pattern. A prominent building in the lower right corner has a complex structure with many small, white, rectangular units on its roof, likely server racks. The facility is surrounded by green fields and some smaller buildings in the distance. The sky is a mix of orange, yellow, and blue.

The datacenter *is* the computer!
What's the instruction
set?



So you like programming in



Traditional Network Programming

Message-passing between nodes (e.g. MPI)

Very difficult to do at scale:

- How to split problem across nodes?
 - Must consider network & data locality

How to deal with failures? (inevitable at scale)

Even worse: stragglers (node not failed, but slow)

Ethernet networking not fast

- Have to write programs for each machine



Data Flow Models

Restrict the programming interface so that the system can do more automatically

Express jobs as graphs of high-level operators »System picks how to split each operator into tasks and where to run each task

- Run parts twice fault recovery

Biggest example: [MapReduce](#)



Why Use a Data Flow Engine?

Ease of programming

- High-level functions instead of message passing

Wide deployment

- More common than MPI, especially “near” data

Scalability to very largest clusters

- Even HPC world is now concerned about resilience

Examples: Pig, Hive, Scalding, Storm, Spark

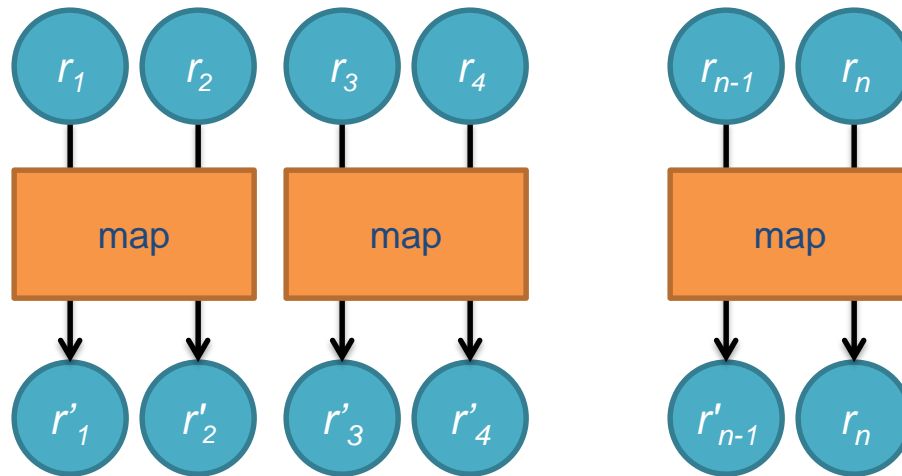


Data-Parallel Dataflow Languages

We have a collection of **records**,
want to apply a bunch of operations
to compute some result

Assumption: static collection of records
(what's the limitation here?)

We Need Per-record Processing



Remarks: Easy to parallelize maps,
record to “mapper” assignment is an implementation
detail

What is MapReduce?

A programming model for processing large datasets in parallel on a cluster, by dividing the work into a set of independent tasks (introduced by Google in 2005)

All we have to do is provide the implementation of two methods:

- map()
- reduce()

...*but we can do much more...*

← *even that, is optional!*

How does it work?

keys and values

- everything is expressed as (*key*, *value*) pairs
 - e.g. the information that the word “hello” appears 4 times in a text, could be expressed as: (“hello”, 4)

Each *map* method receives a list of (*key*, *value*) pairs and emits a list of (*key*, *value*) pairs

- the intermediate output of the program

Each *reduce* method receives, for each unique intermediate key *k*, a **list** of all intermediate values that were emitted for *k*.

Then, it emits a list of (*key*, *value*) pairs

- the final output of the program

MapReduce – Input Data

e1

e2

e3

e4

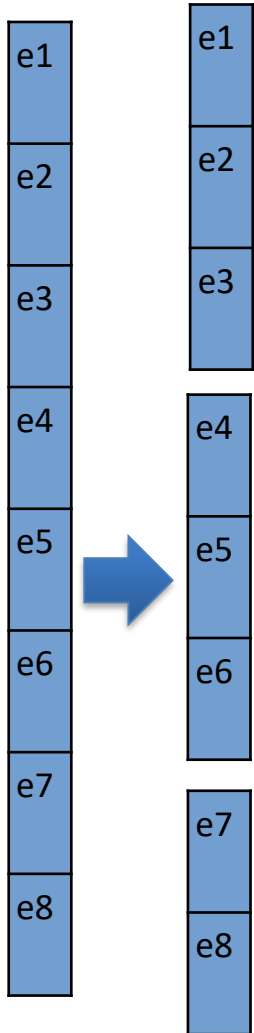
e5

e6

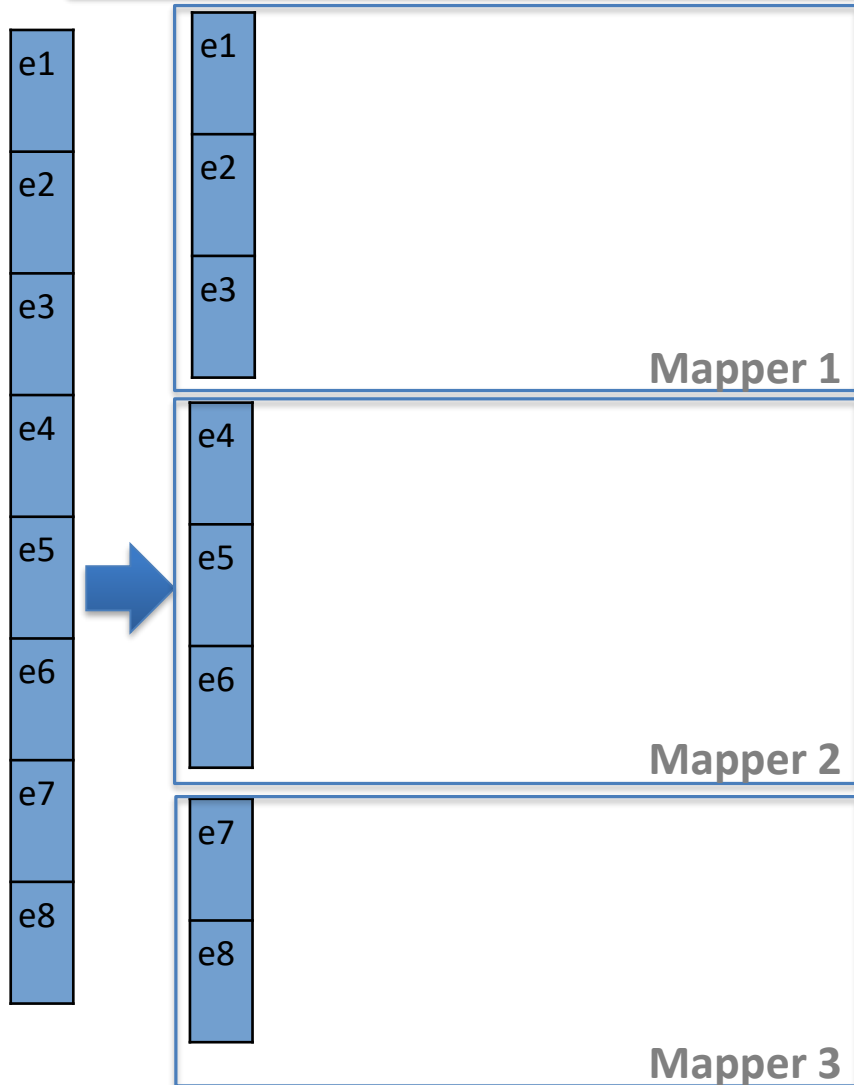
e7

e8

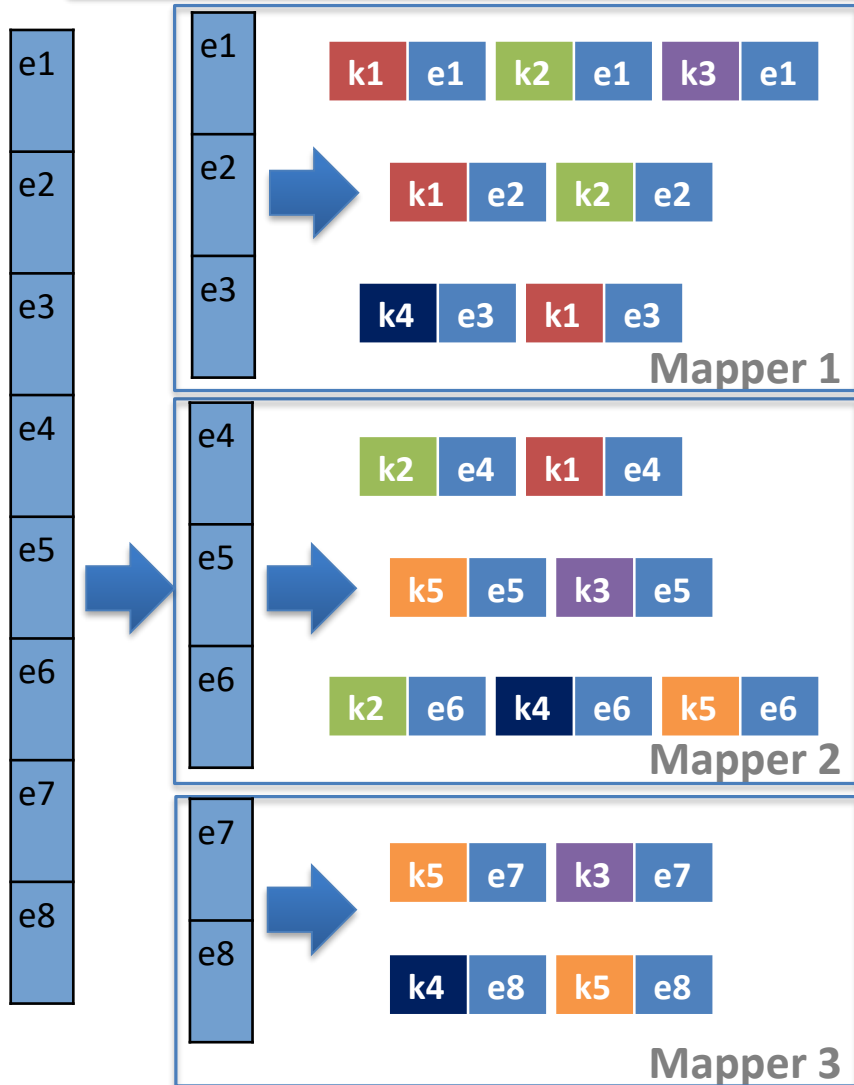
MapReduce – Input Data Splitting



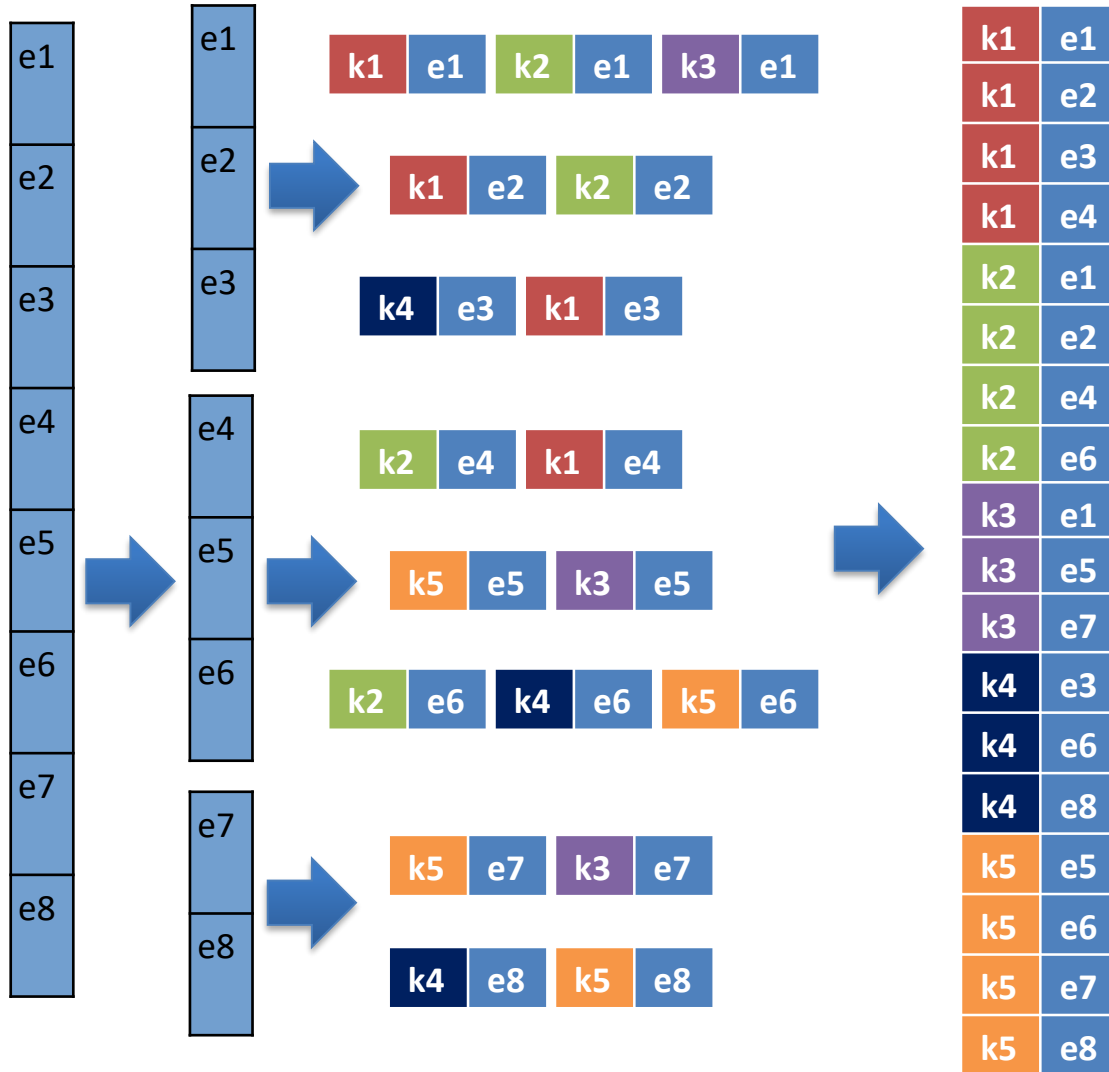
MapReduce – Mapper Input



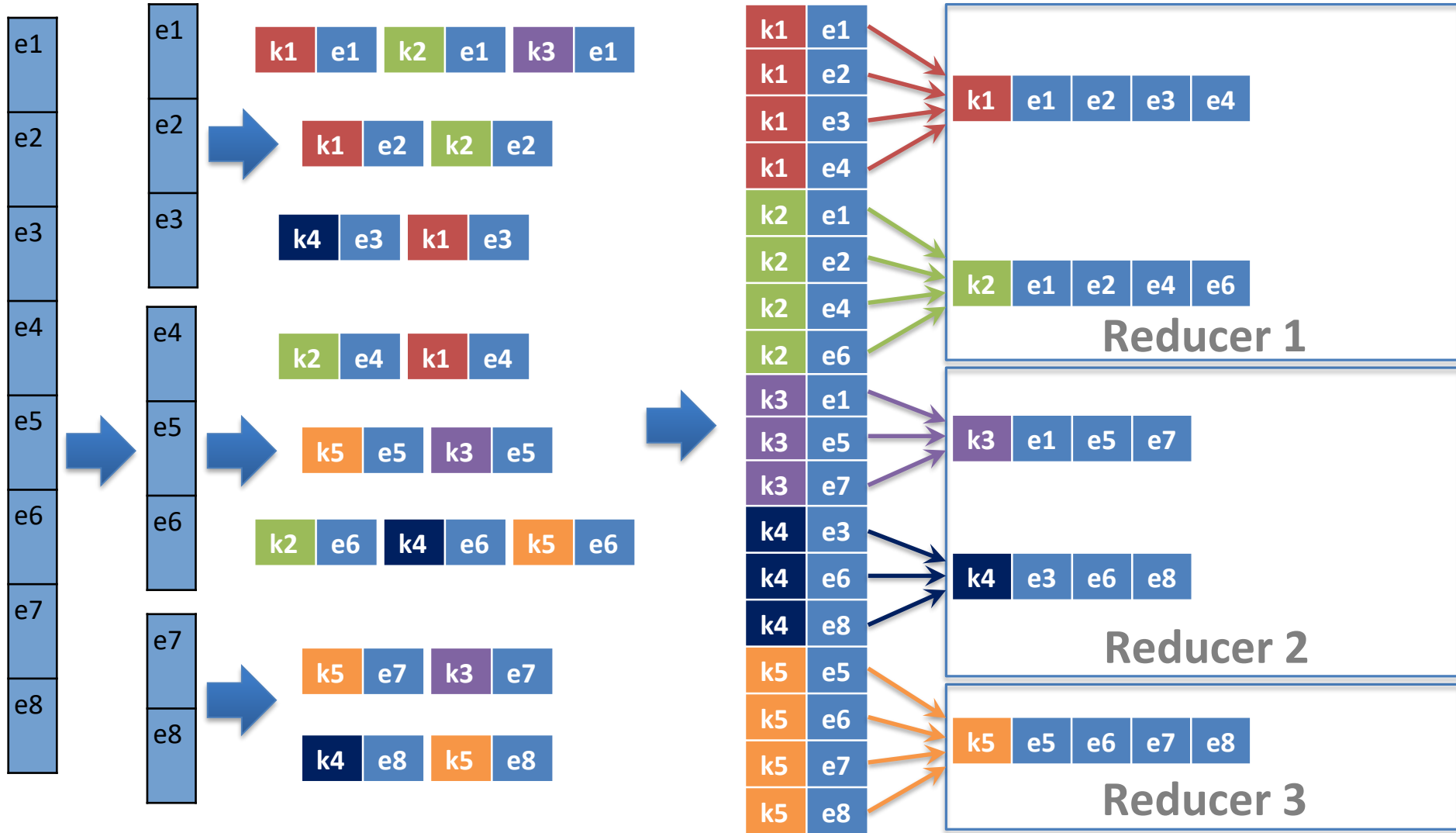
MapReduce – Mapper Output



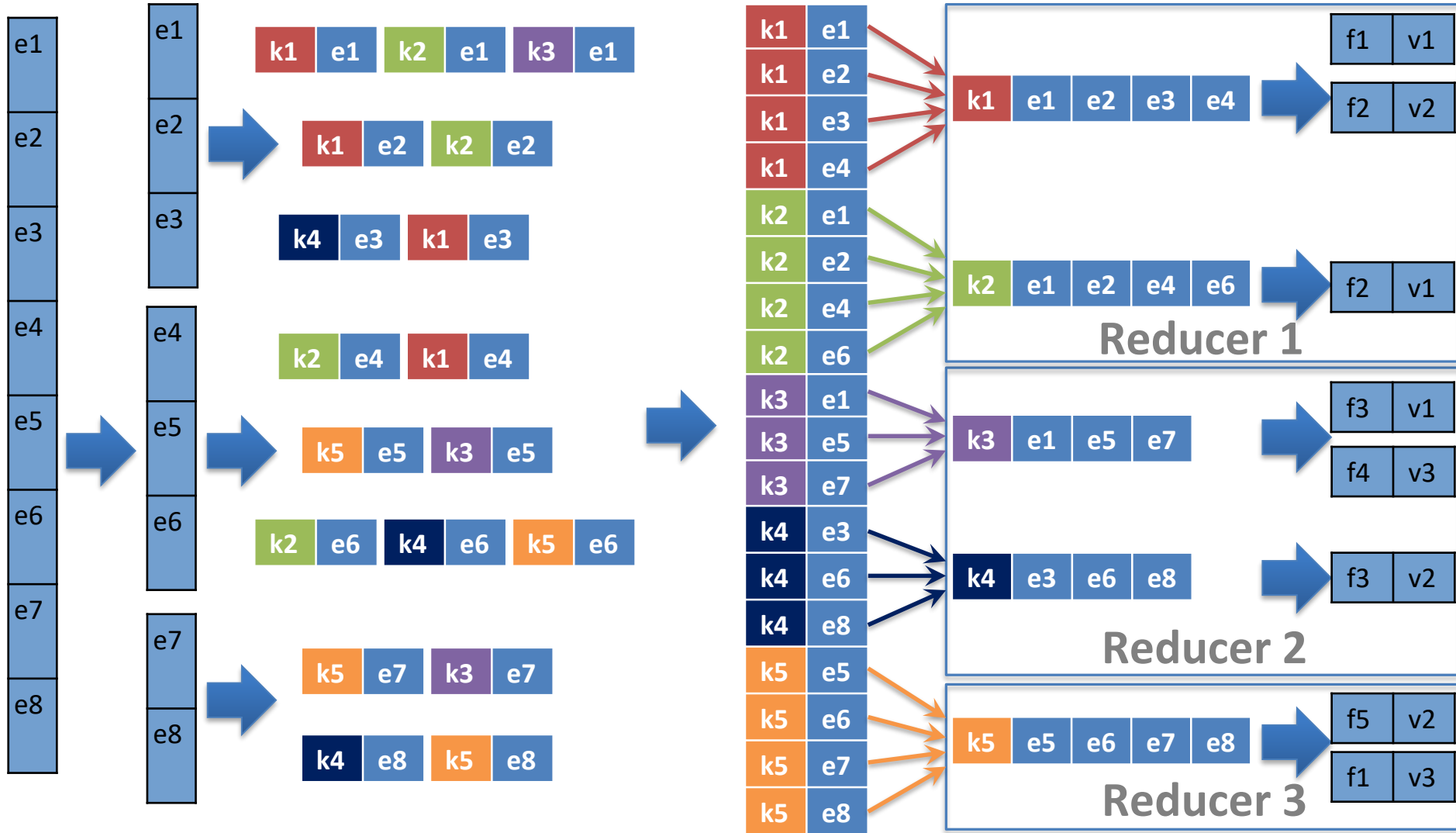
MapReduce – Shuffling & Sorting (simplified)



MapReduce – Reducing



MapReduce – Reducing



Example: WordCount

- **Input:** A list of (file-name, line) pairs
- **Output:** A list of (word, frequency) pairs for each unique word appearing in the input

Idea:

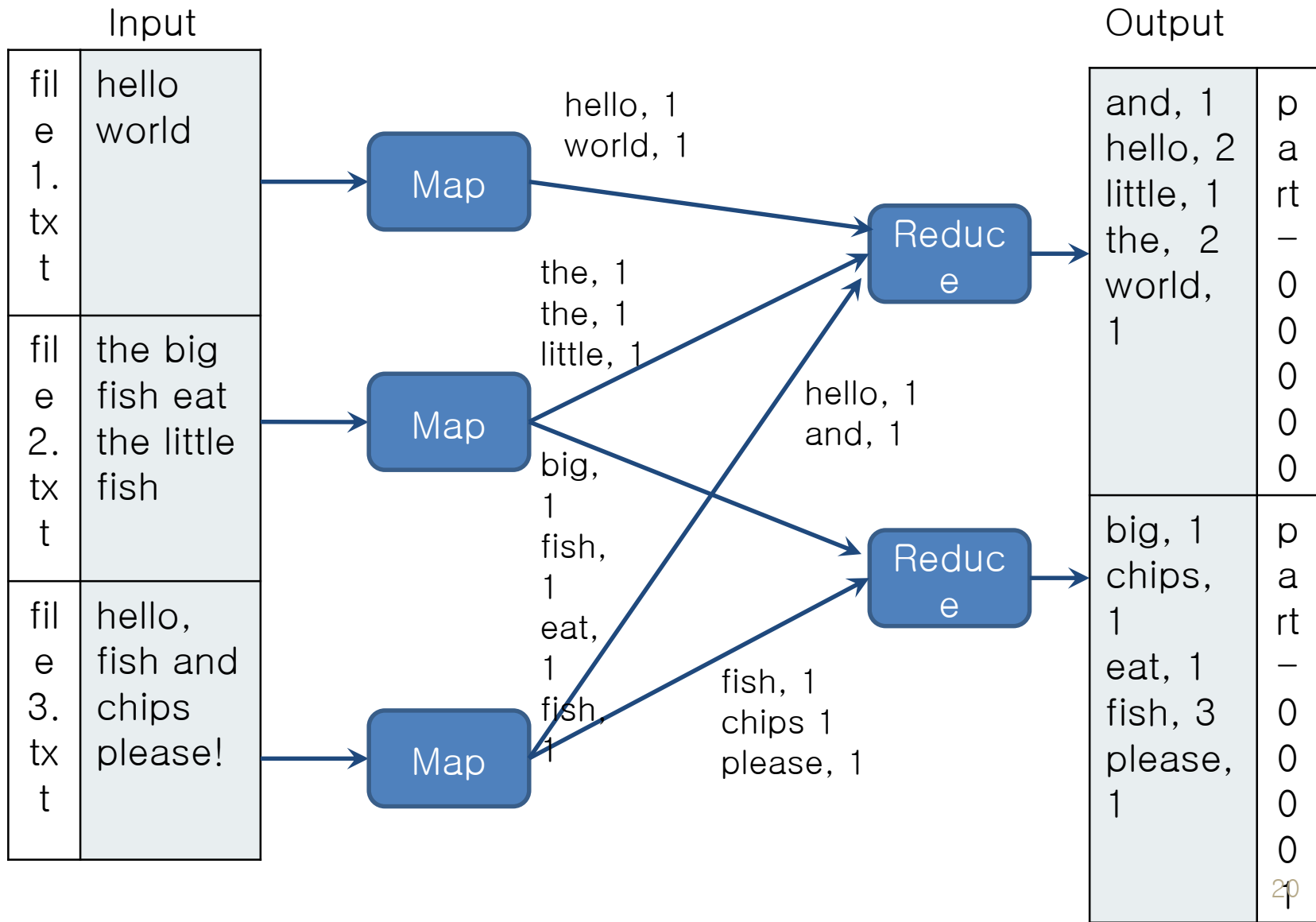
Map:

for each word w , emit a $(w, 1)$ pair

Reduce:

for each $(w, \text{list}(1,1,\dots,1))$, sum up the 1's and emit a $(w, 1+1+\dots+1)$ pair

Example: WordCount



WordCount Mapper

```
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);
        }
    }
}
```

WordCount Reducer

```
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));
    }
}
```

Combiner: a local, mini-reducer

- An optional class that works like a reducer, run locally
 - for the output of each mapper
- Goal:
 - reduce the network traffic from mappers to reducers
 - could be a bottleneck
 - reduce the workload of the reducers

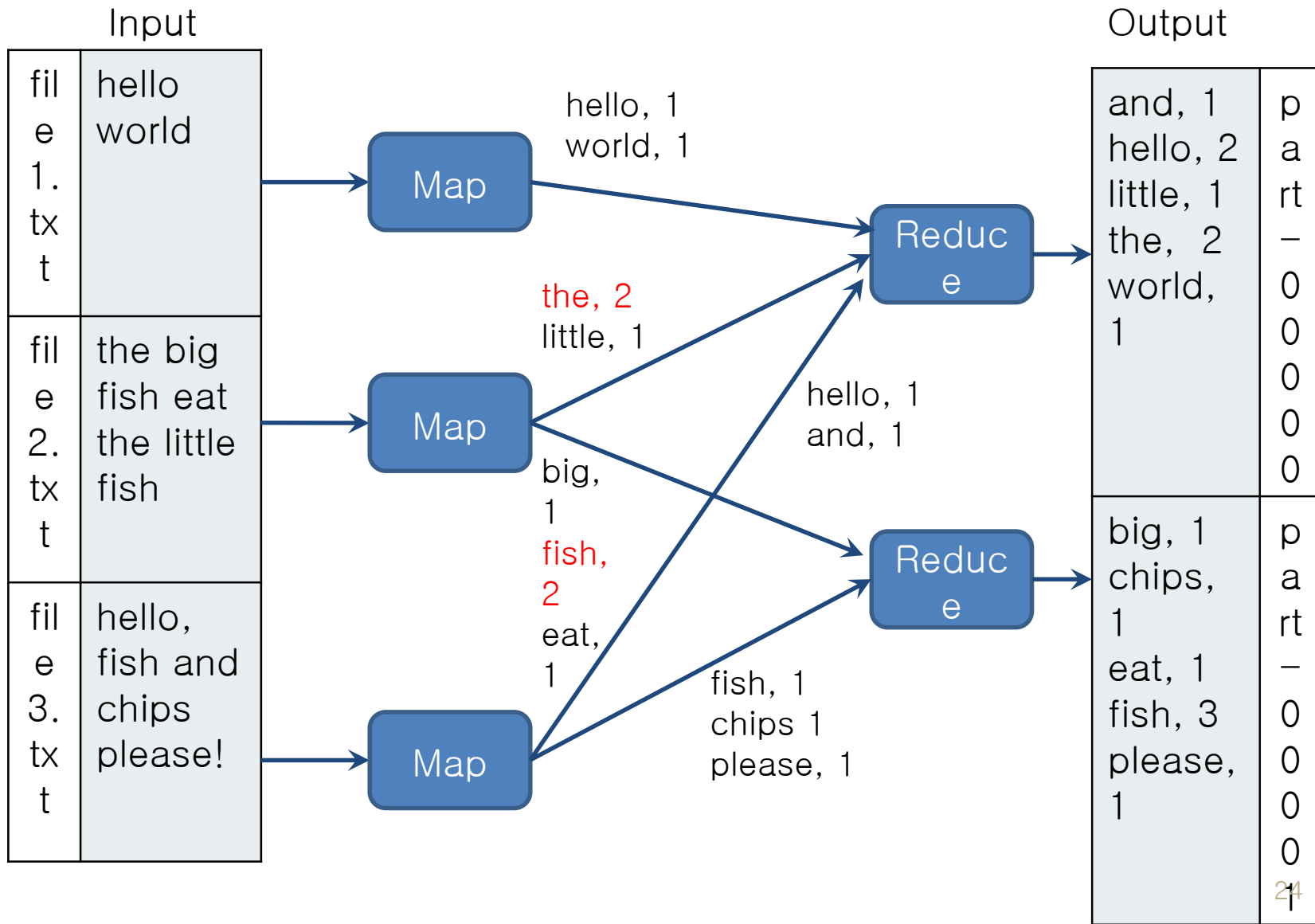
WordCount Example:

We could sum up the local 1's corresponding to the same key and

emit a temporary word count to the reducer

- fewer pairs are sent to the network
- the reducers save some operations

WordCount with Combiner



Map Alone Isn't Enough!

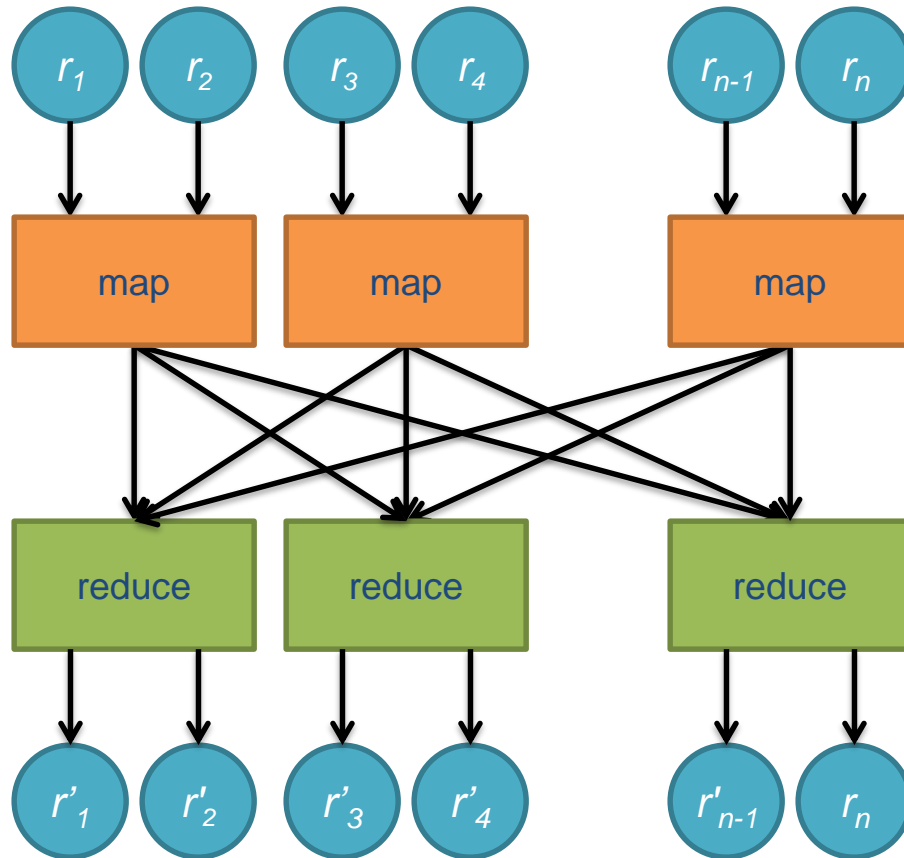
(If w

Where do intermediate results go?
We need an addressing mechanism!
What's the semantics of the group by?

Once we resolve the addressing, apply another
computation

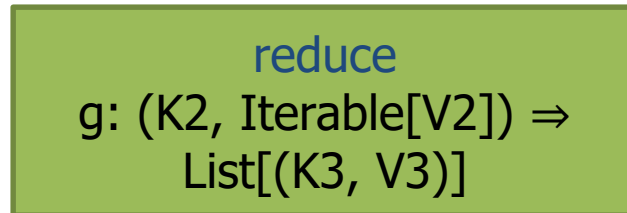
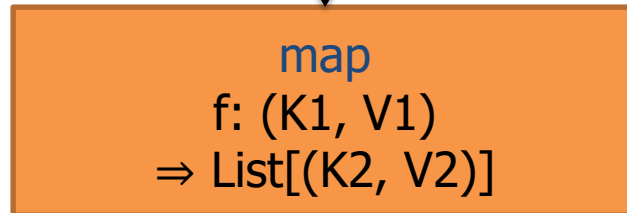
That's what we call reduce!
(What's with the sorting then?)

MapReduce



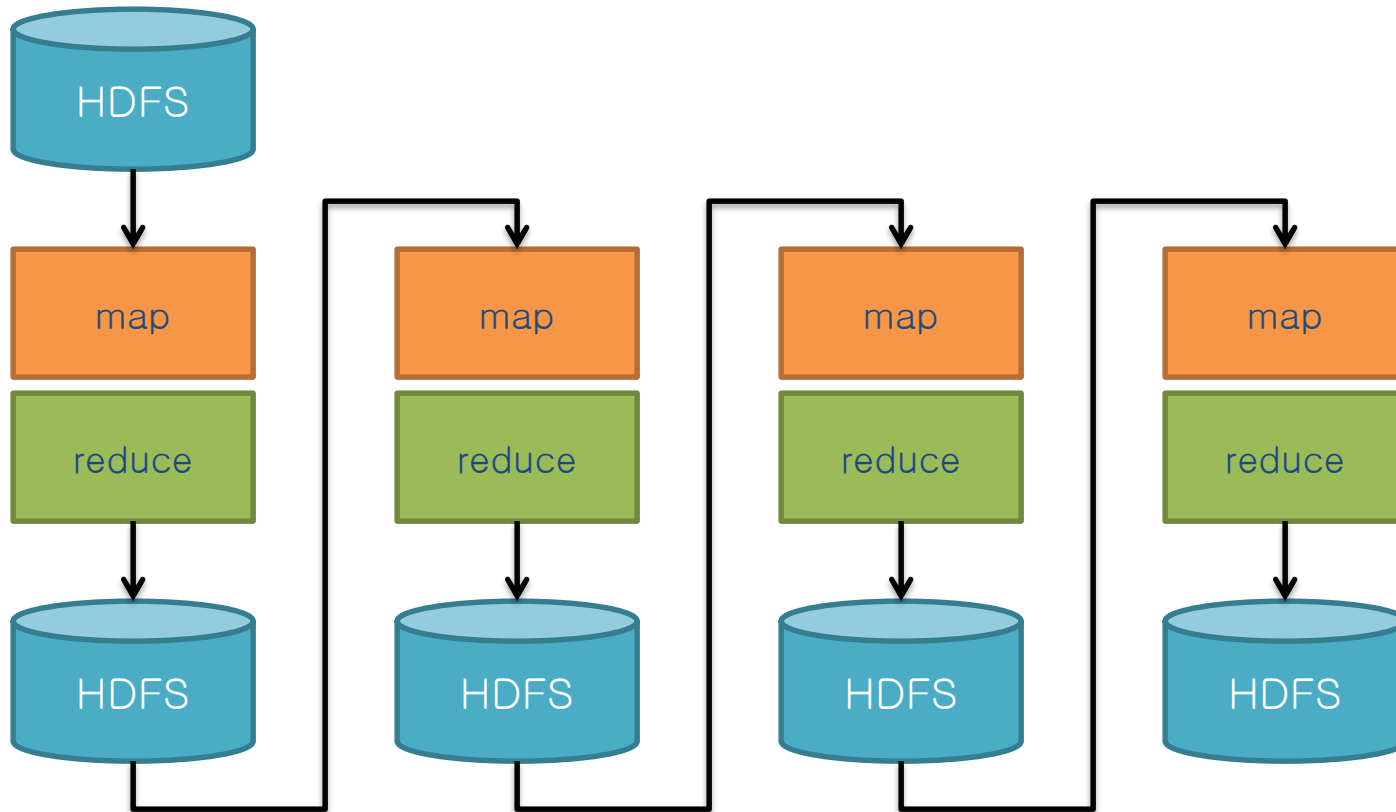
MapReduce

List[(K1,V1)]



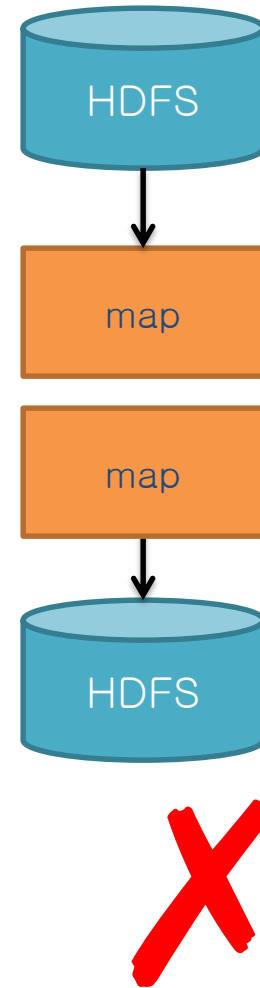
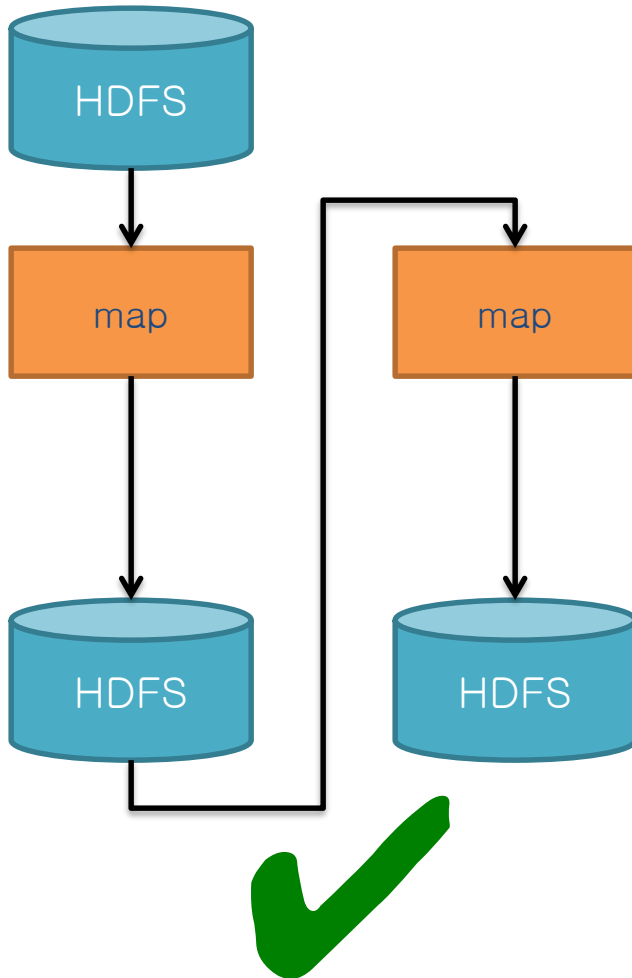
List[K3,V3])

MapReduce Workflows

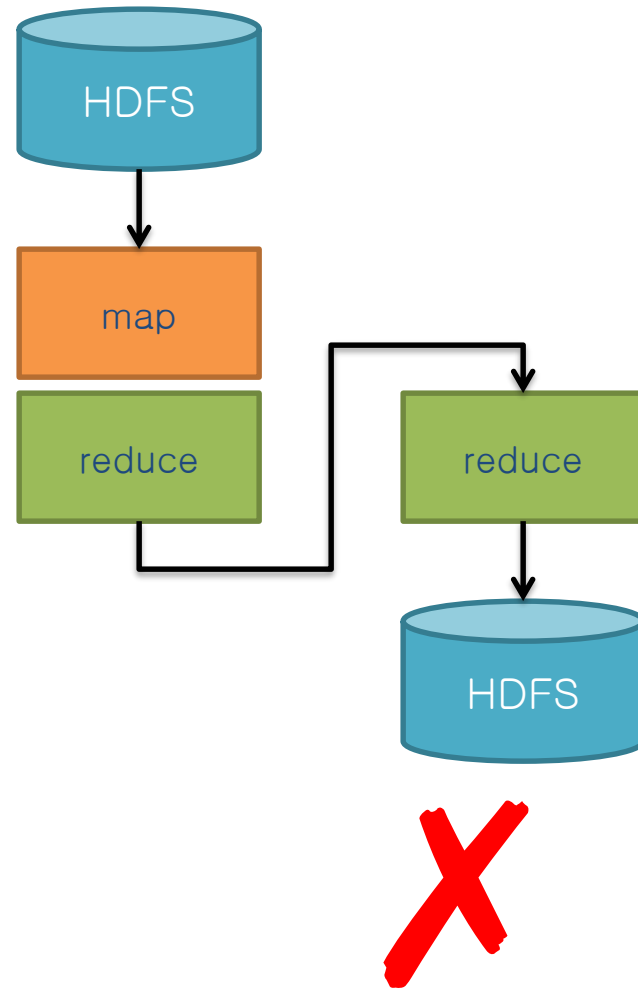
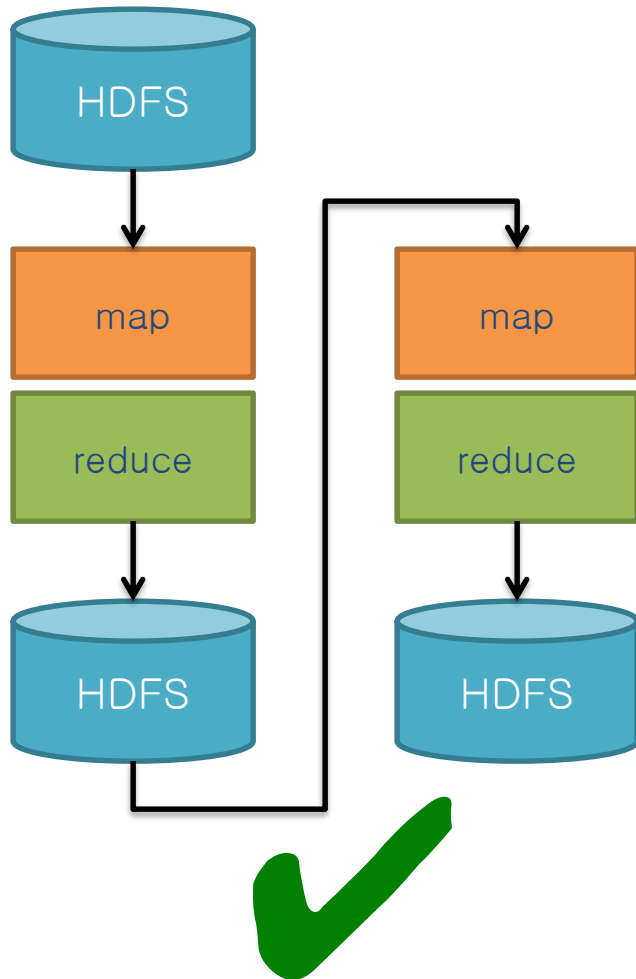


What's wrong?

Want MM?



Want MRR?





Spark

Answer to “What’s beyond MapReduce?”

Brief history:

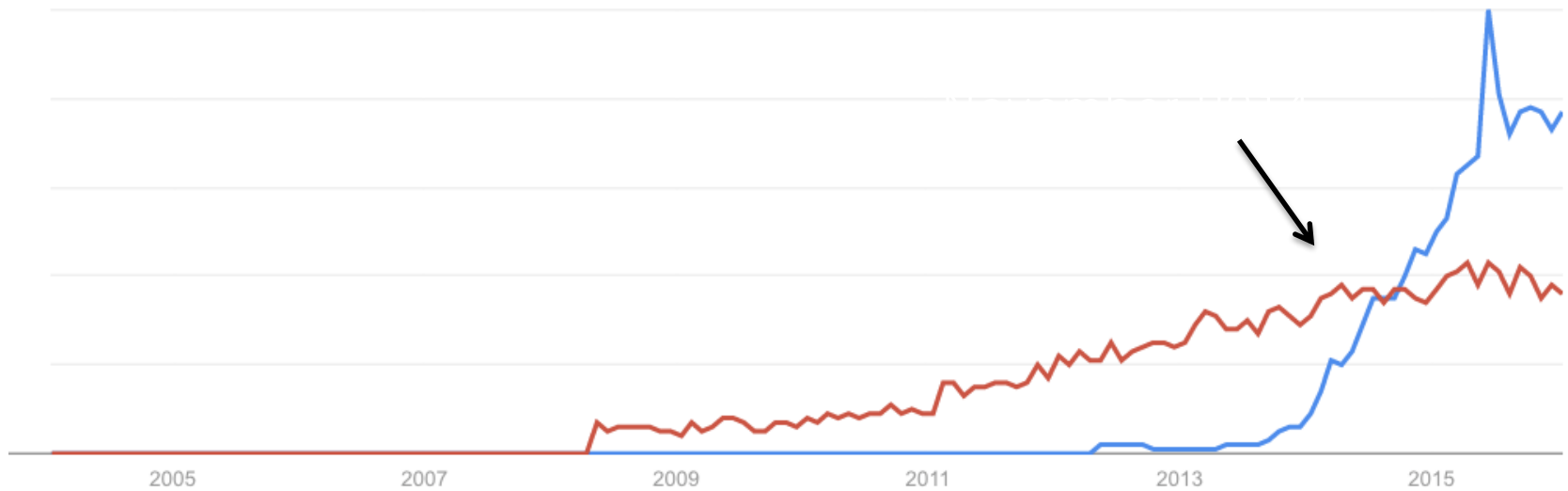
Developed at UC Berkeley AMPLab in 2009

Open-sourced in 2010

Became top-level Apache project in February 2014

Commercial support provided by DataBricks

Spark vs. Hadoop Popularity



MapReduce

List[(K1,V1)]



map

f: (K1, V1)
⇒ List[(K2, V2)]

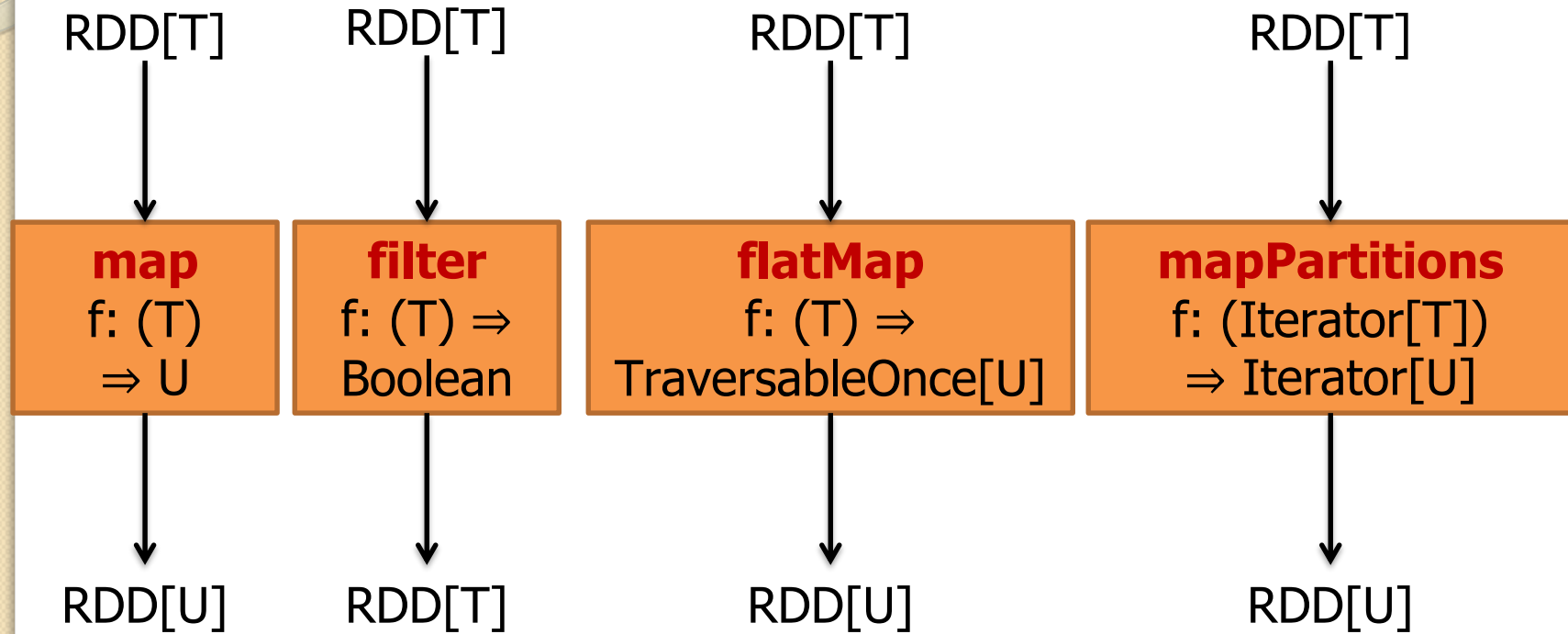
reduce

g: (K2, Iterable[V2]) ⇒
List[(K3, V3)]



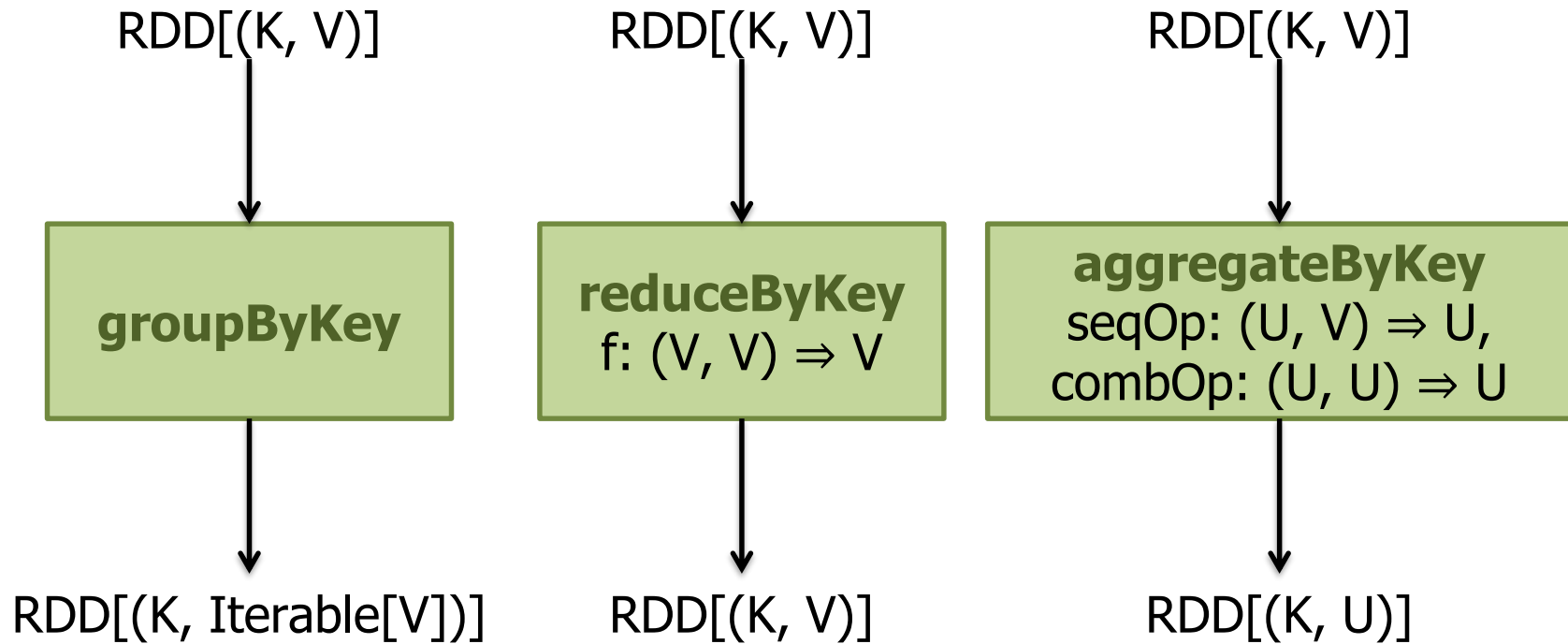
List[(K3,V3)]

Map-like Operations



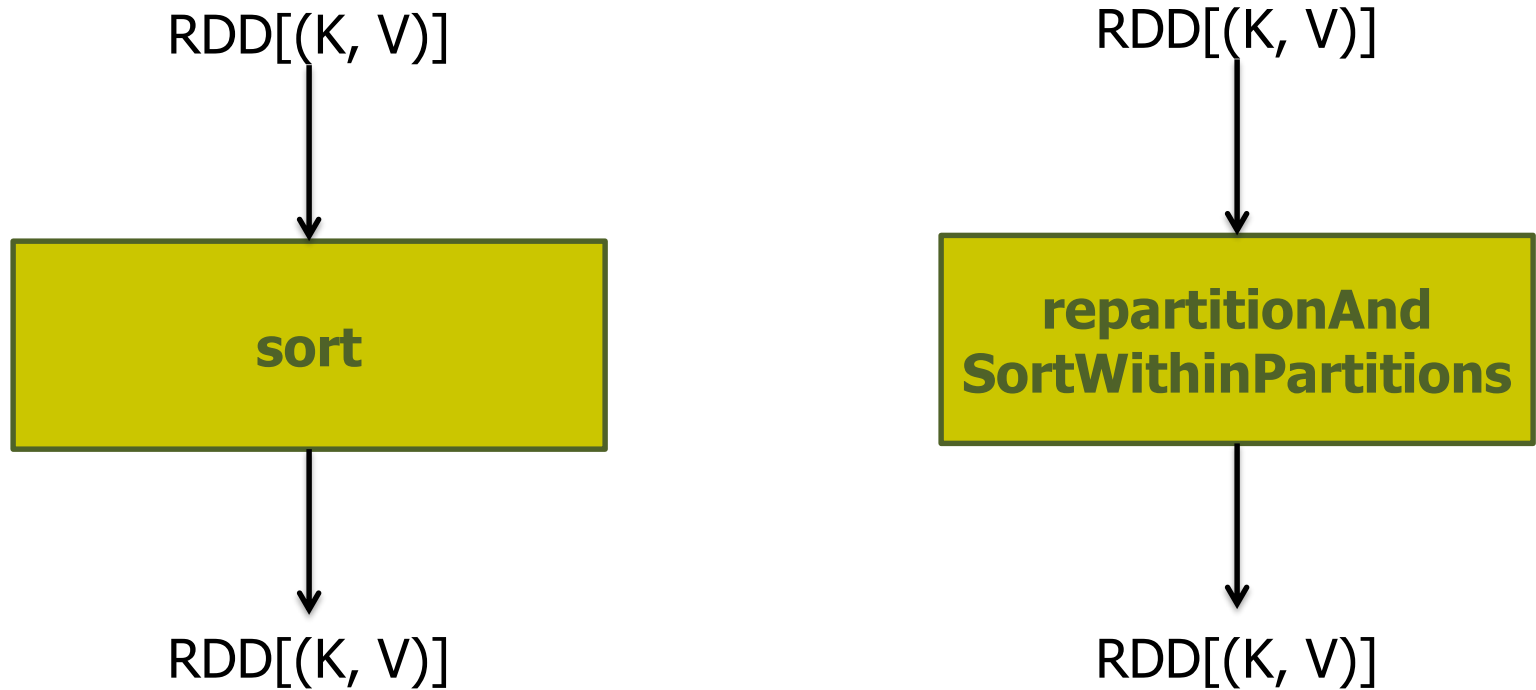
(Not meant to be exhaustive)

Reduce-like Operations



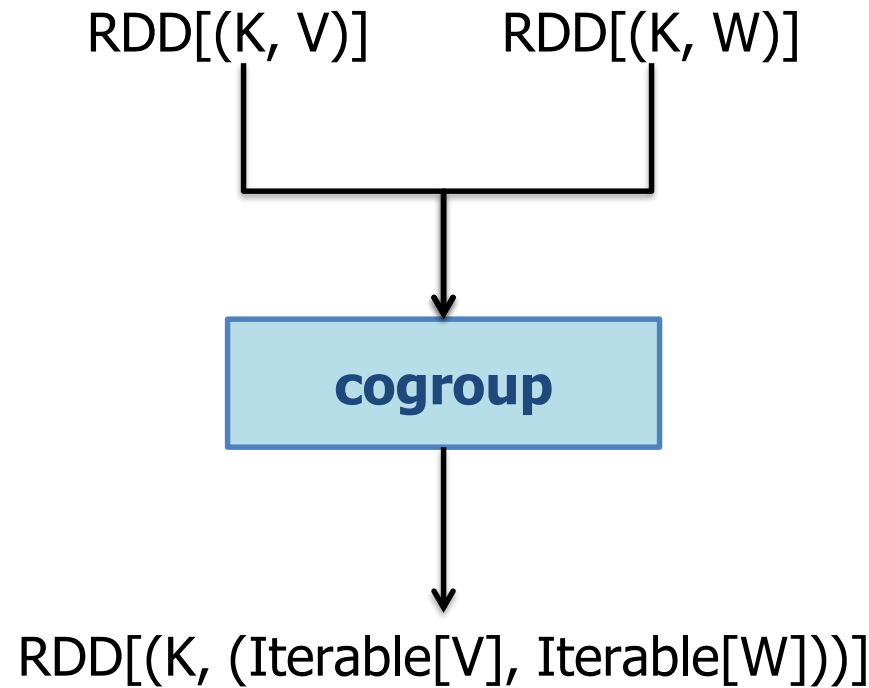
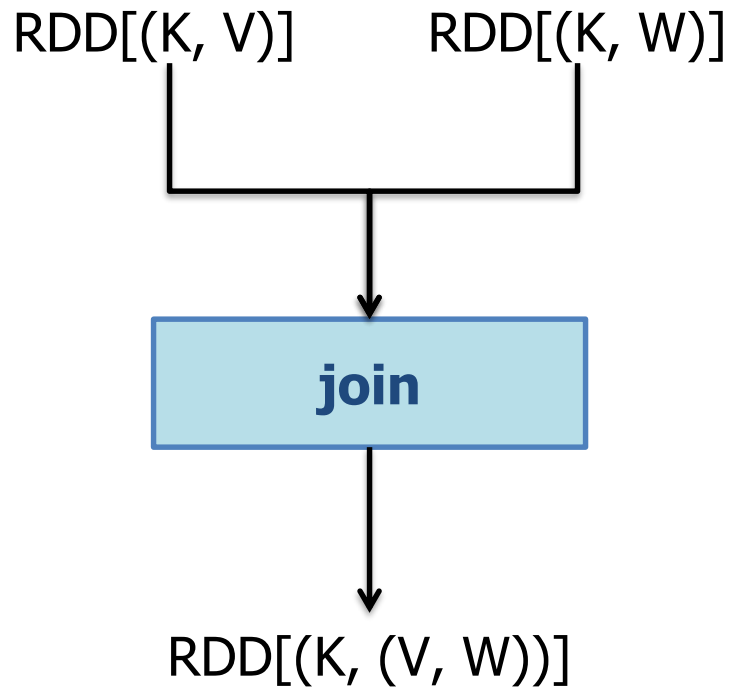
(Not meant to be exhaustive)

Sort Operations



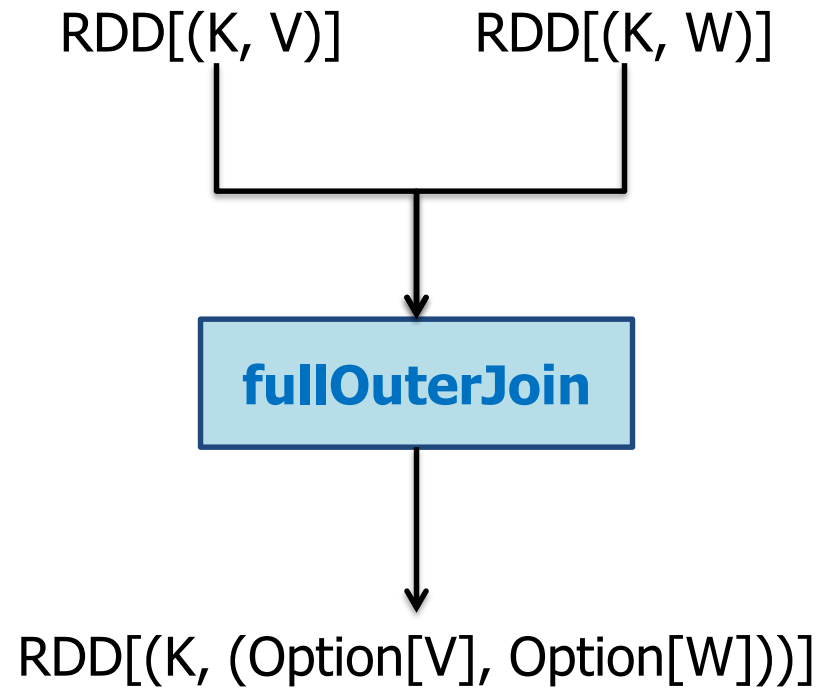
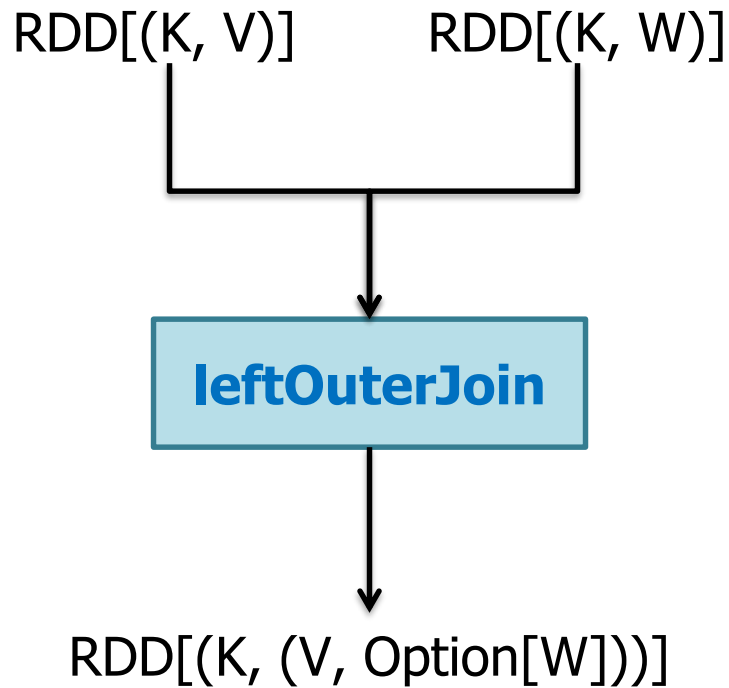
(Not meant to be exhaustive)

Join-like Operations



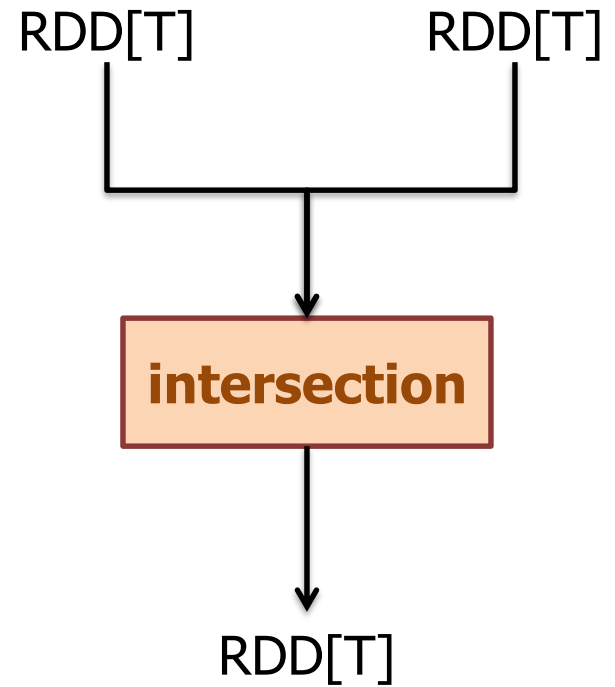
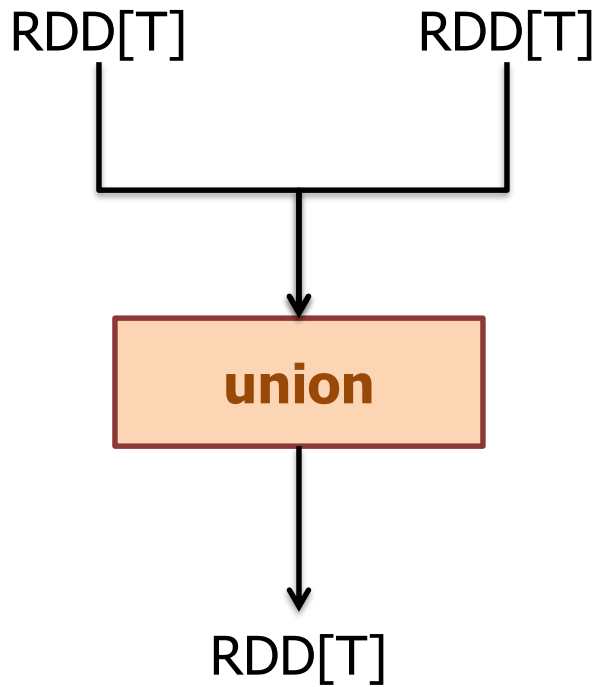
(Not meant to be exhaustive)

Join-like Operations



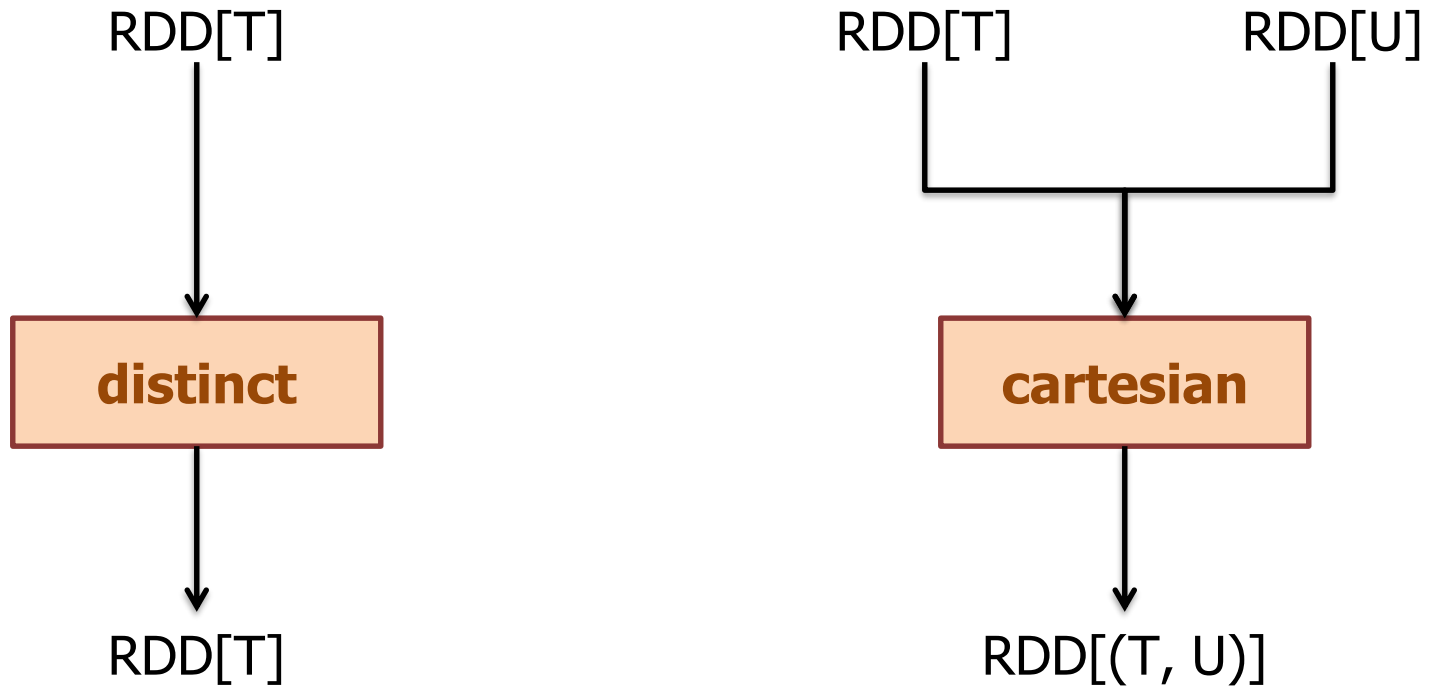
(Not meant to be exhaustive)

Set-ish Operations



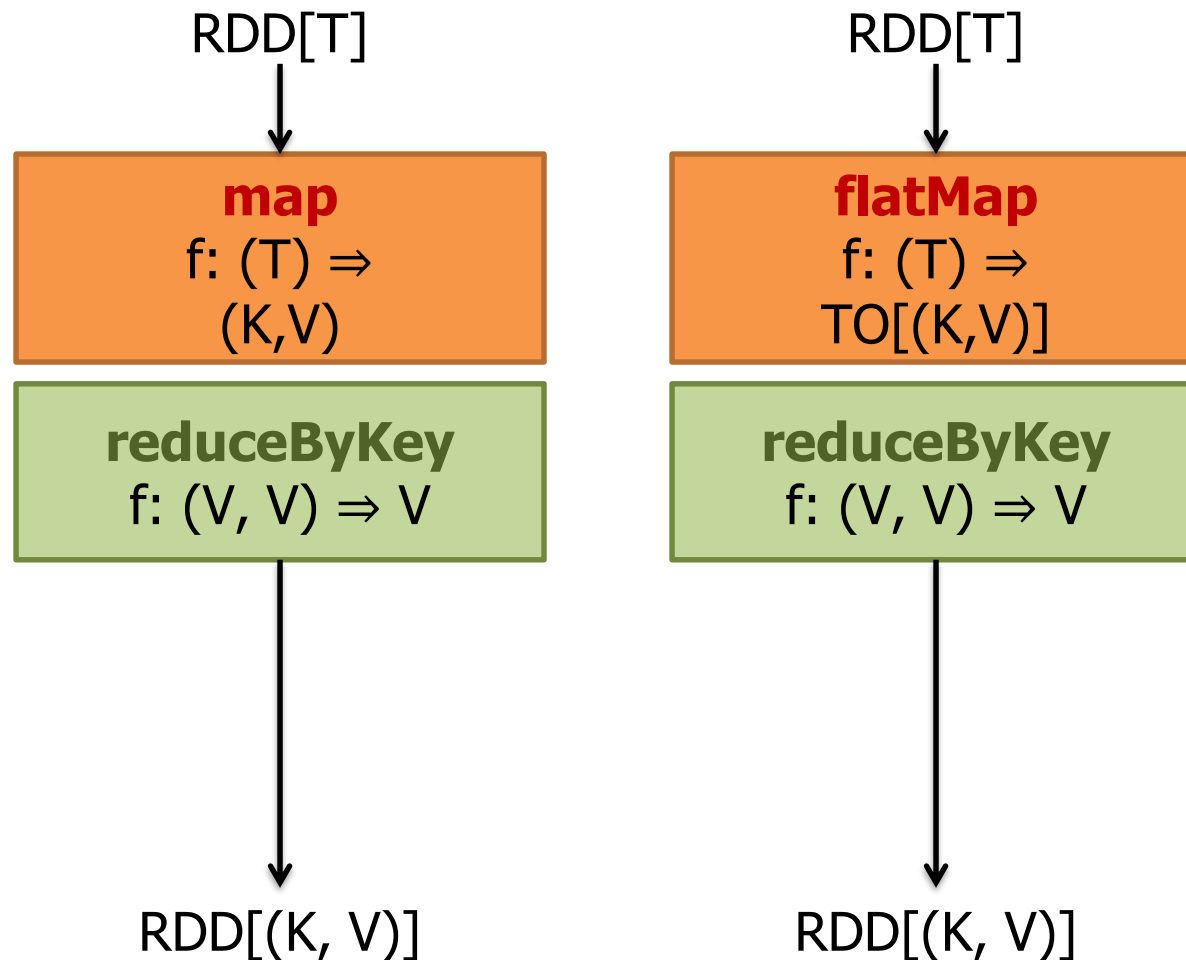
(Not meant to be exhaustive)

Set-ish Operations



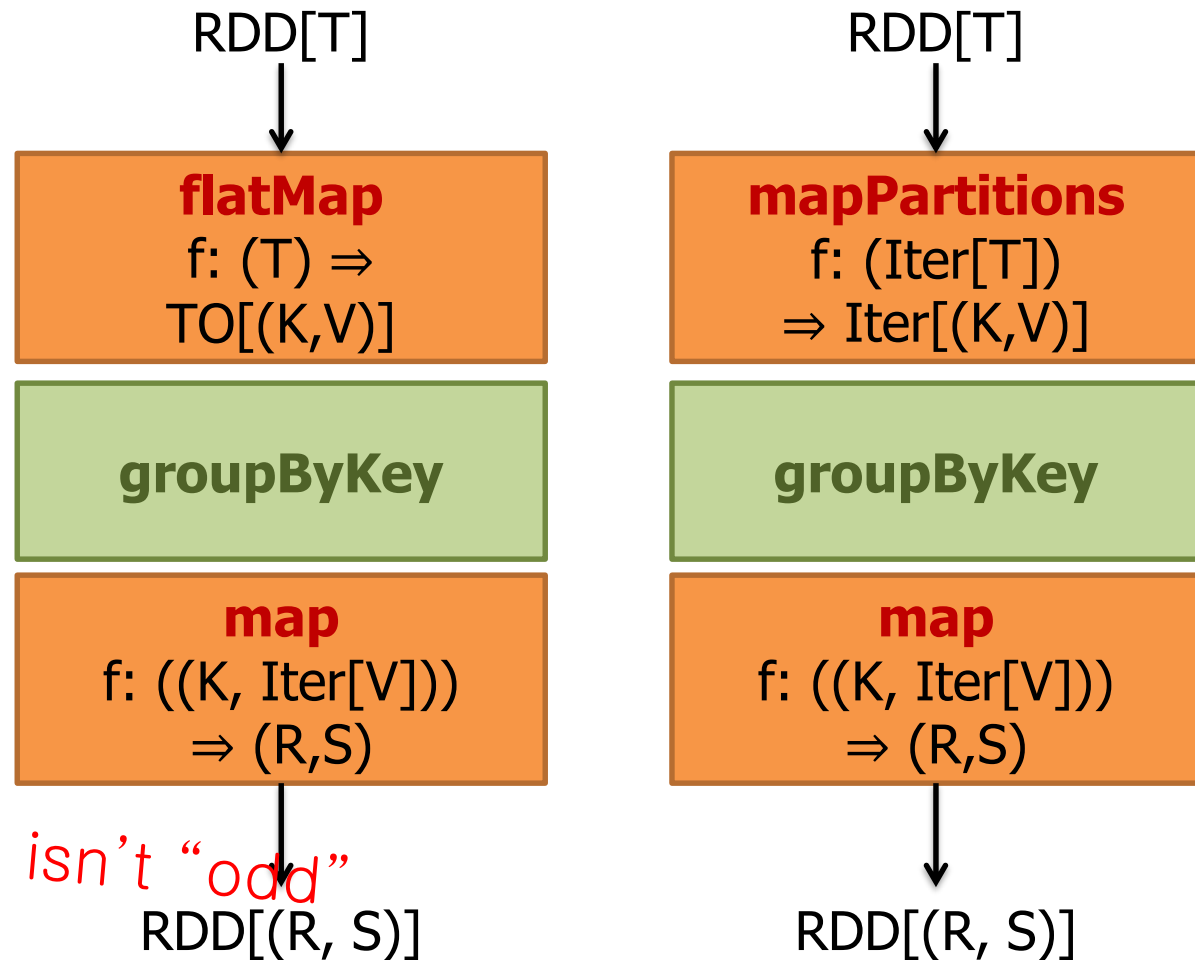
(Not meant to be exhaustive)

MapReduce in Spark?



Not quite...

MapReduce in Spark?



Nope, this isn't "odd"

Still not quite...

Spark Word Count

```
val textFile = sc.textFile(args.input())
```

```
textFile
```

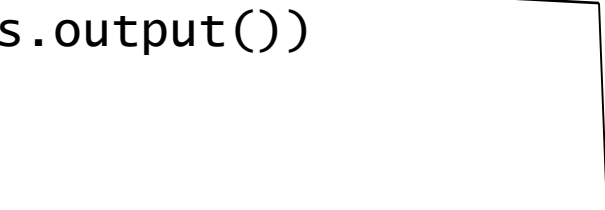
```
.flatMap(line => tokenize(line))
```

```
.map(word => (word, 1))
```

```
.reduceByKey(_ + _)←
```

```
.saveAsTextFile(args.output())
```

(x, y) => x + y



Aside: Scala tuple access notation, e.g., a._1

Don't focus on Java verbosity!

```
val textFile = sc.textFile(args.input())

textFile
  .map(object mapper {
    def map(key: Long, value: Text) =
      tokenize(value).foreach(word => write(word, 1))
  })
  .reduce(object reducer {
    def reduce(key: Text, values: Iterable[Int]) = {
      var sum = 0
      for (value <- values) sum += value
      write(key, sum)
    }
  })
  .saveAsTextFile(args.output())
```

Install Spark

Let's get started using Apache Spark, in just four easy steps...

Step 1: Install Java JDK 6/7 on [MacOSX](#) or [Windows](#)

oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html

follow the license agreement instructions

then click the download for your OS

need JDK instead of JRE (for Maven, etc.)

this is much simpler on Linux: `sudo apt-get -y install openjdk-7-jdk`

Step 2: Download the latest Spark version 2.4.4

open spark.apache.org/downloads.html with a browser

double click the archive file to open it

connect into the newly created directory

Install Spark

Step 3: Run Spark Shell

we'll run Spark's interactive shell...

```
./bin/spark-shell
```

then from the "scala>" REPL prompt,
let's create some data...

```
val data = 1 to 10000
```

Step 4: Create an RDD

create an RDD based on that data...

```
val distData = sc.parallelize(data)
```

then use a filter to select values less than 10...

```
distData.filter(_ < 10).collect()
```

Check your output :

gist.github.com/ceteri/f2c3486062c9610eac1d#file-01-repl-txt

Optional Downloads


Python:

For Python 2.7, check out Anaconda by Continuum Analytics for a full-featured platform:
store.continuum.io/cshop/anaconda/

Maven

Java builds later also require Maven, which you can download at:
maven.apache.org/download.cgi

Resources

- Jimmy Lin. CS 489/698 Big Data Infrastructure, Winter 2017. David R. Cheriton School of Computer Science, University of Waterloo <http://lintool.github.io/bigdata-2017w/> This work is licensed under a Creative Commons Attribution–Noncommercial–Share Alike 3.0 United States 
- First part of this tutorial was adapted from <https://developer.yahoo.com/hadoop/tutorial/index.html>, under a [Creative Commons Attribution 3.0 Unported License](https://creativecommons.org/licenses/by/3.0/).