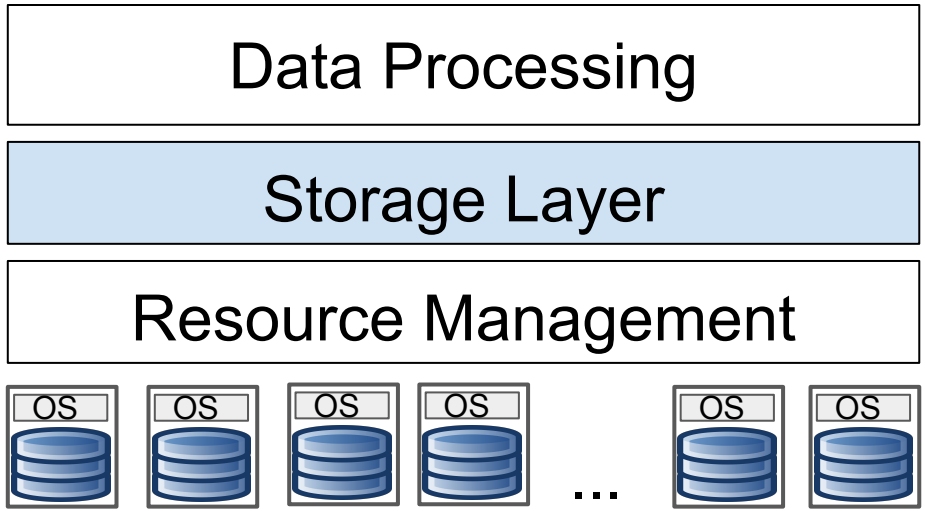
Databases and Big Data

Hadoop 2

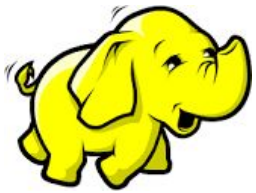
Recap

- HDFS implements Google File System
  - Storage layer for cloud computing
  - Append-only files
  - Fault tolerance: replicas
  - Scalability: slave nodes



Recap

Data is only useful if we can query it



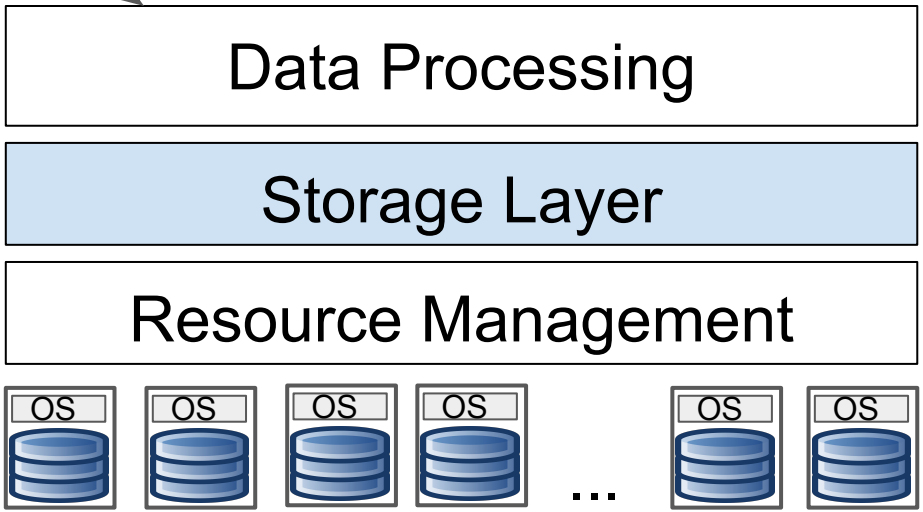
Today

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.



Agenda

- MapReduce Model
- Example
- MapReduce in Hadoop
- APIs

1) Divide Books Across Individuals

2) Compute Counts Locally

Word	Count
Apple	2
Bird	7
…	

Word	Count
Apple	0
Bird	1
…	

Map phase

Compute number of occurrences of each word in all books

1) Divide Books Across Individuals

2) Compute Counts Locally

Word	Count
Apple	2
Bird	7
…	

Word	Count
Apple	0
Bird	1
…	

Word	Count
Apple	2
Bird	8
…	

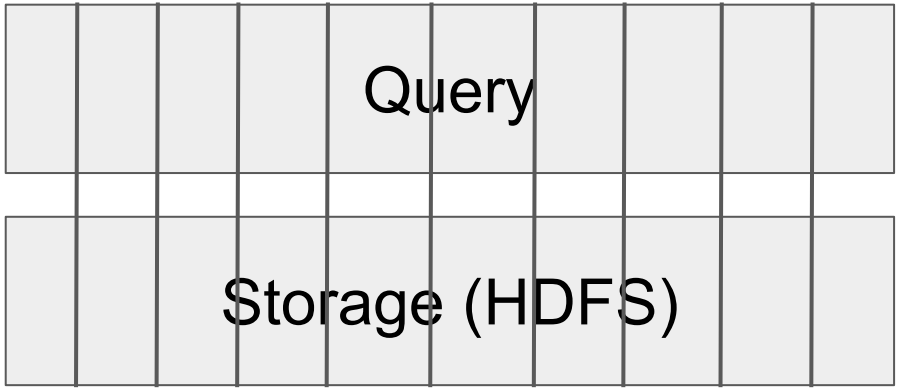
Map phase

Reduce phase

Data Processing

- Embarrassingly parallel data processing
  - Query cover the entire datasets
  - Query broken up into sub-queries
  - Sub-queries run in parallel
- Early 2000s
  - Many jobs look like this

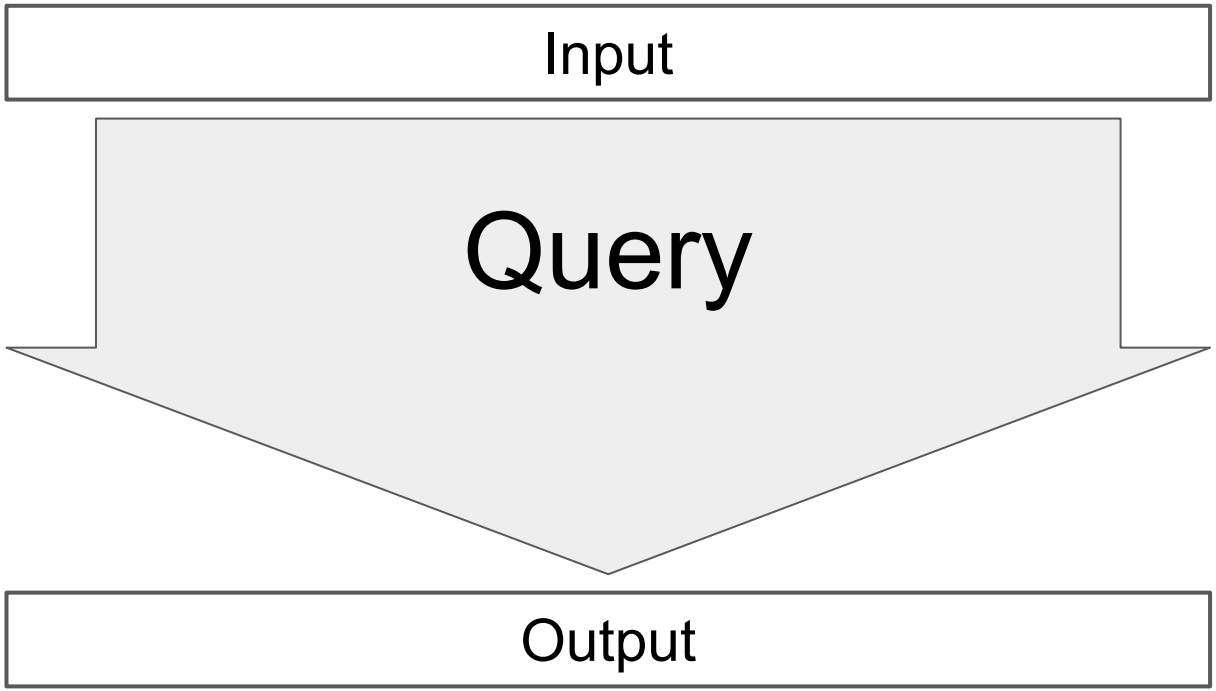
Early 2000s, you can split and everything can be done in parallel (cause smoll enough)



Examples:
Extract-Transform-Load pipeline
Build inverted index on Web

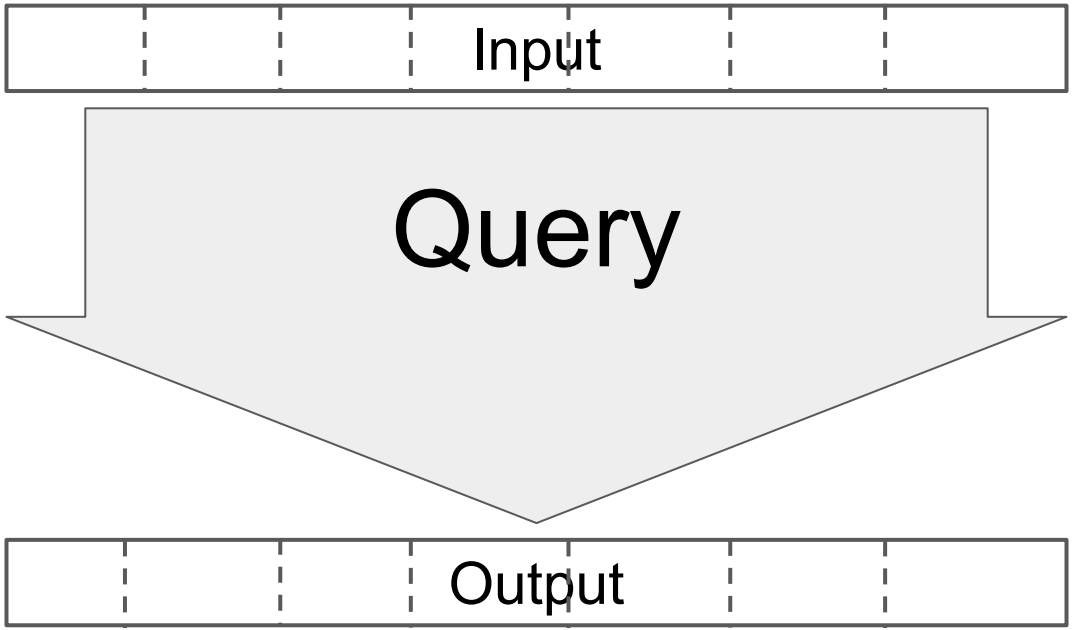
Data Processing

- What user wants



Data Processing

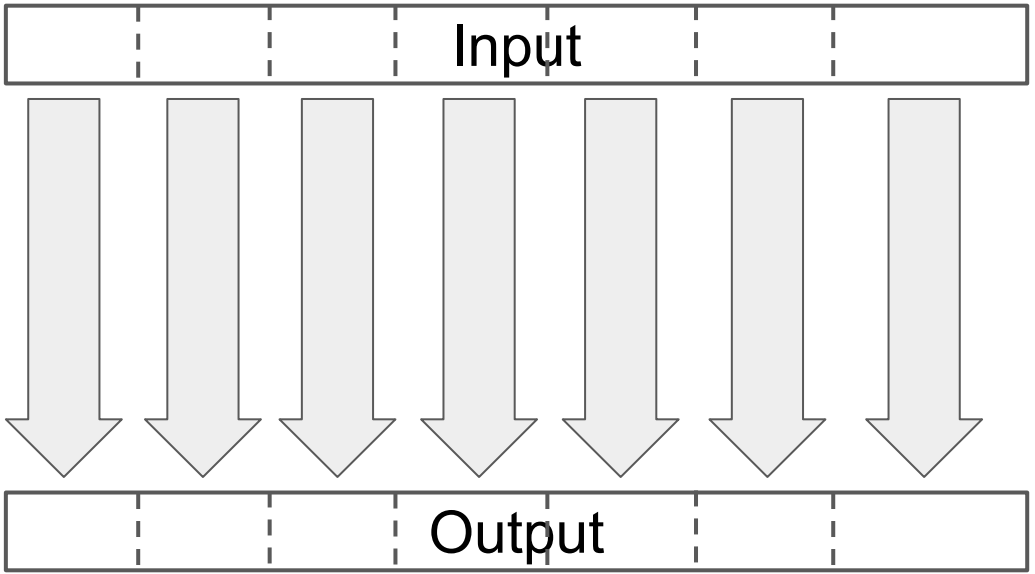
- Data comes in chunks



Data Processing

- Ideal case

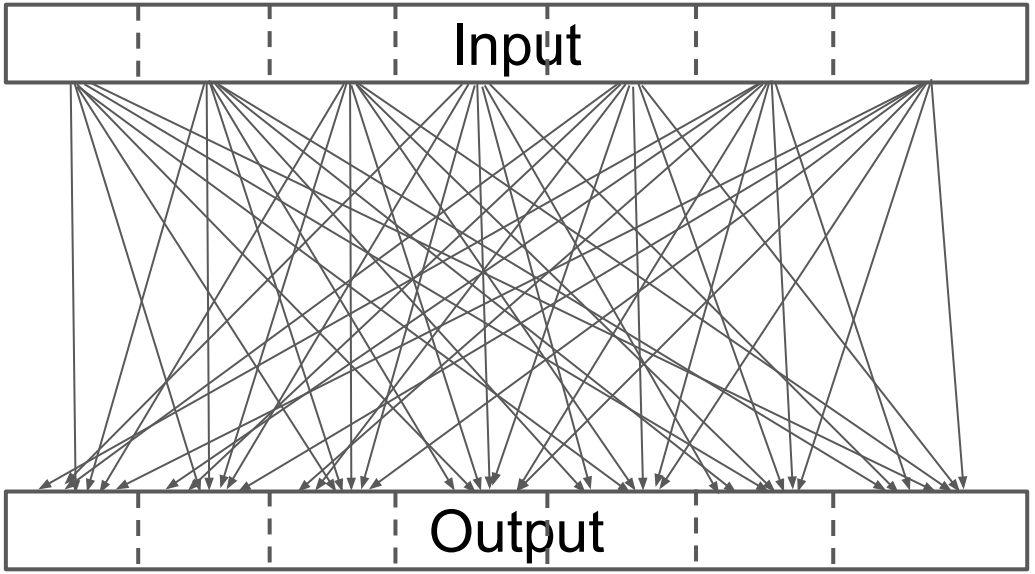
Map



Data Processing

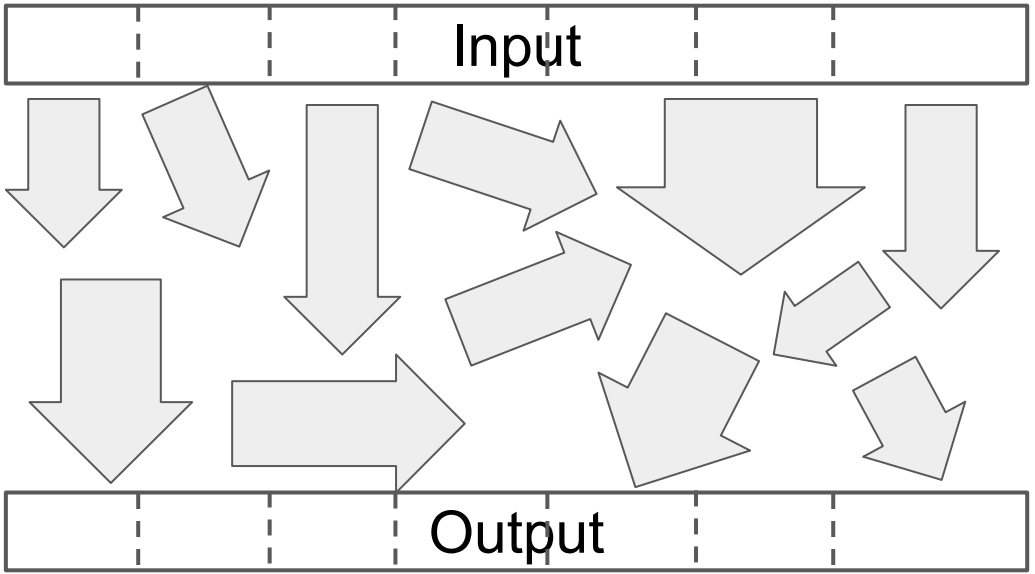
- Worst case

Shuffle



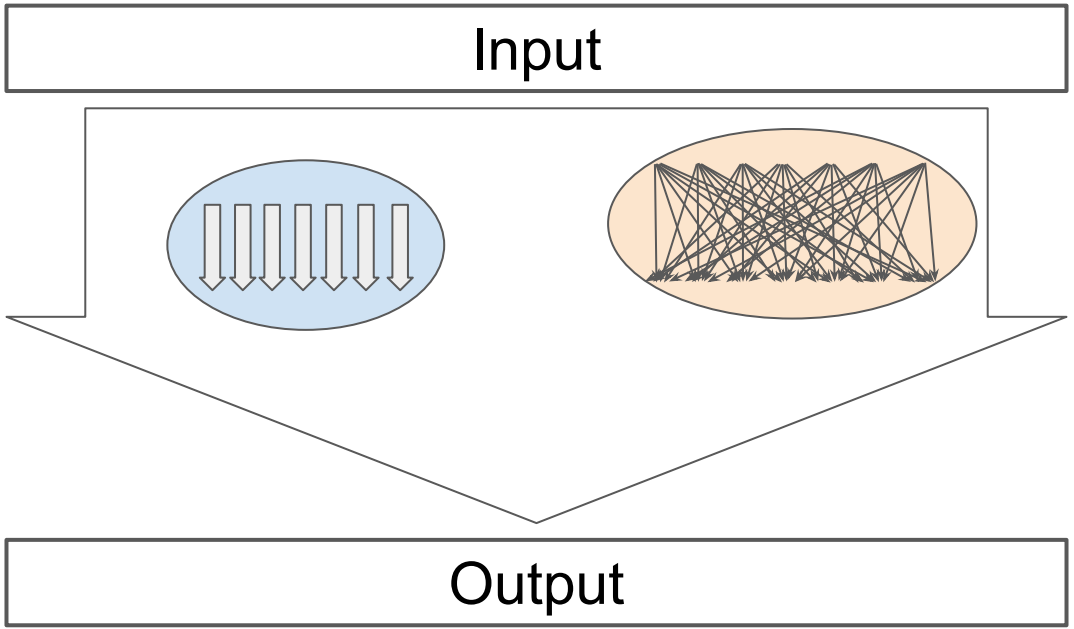
Data Processing

- In practice



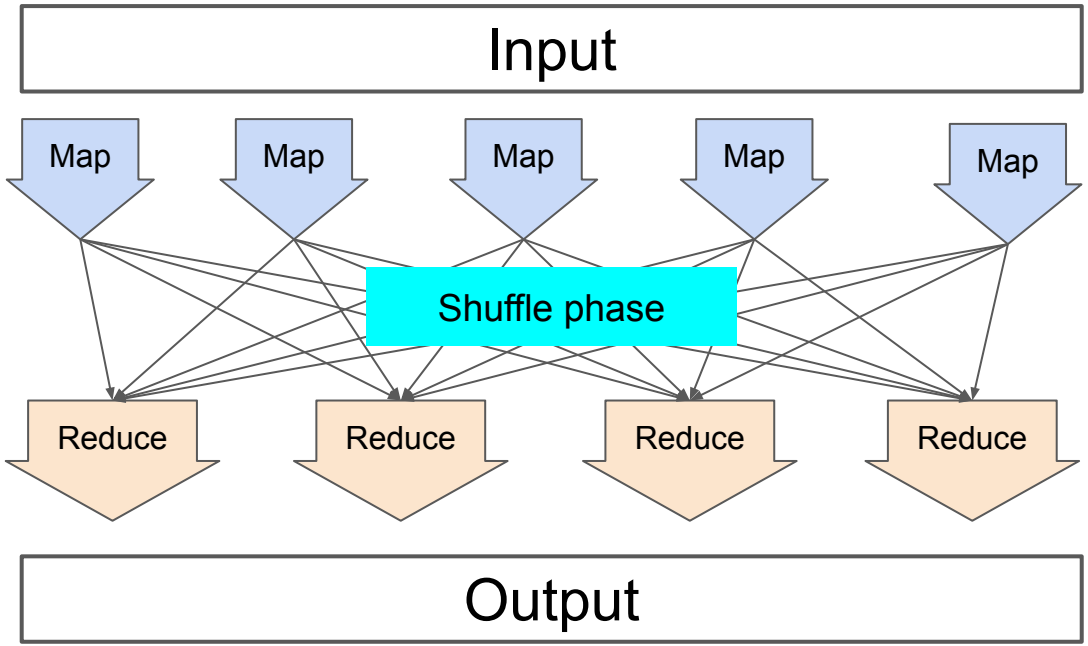
Data Processing

- Map here…
- Shuffle there...



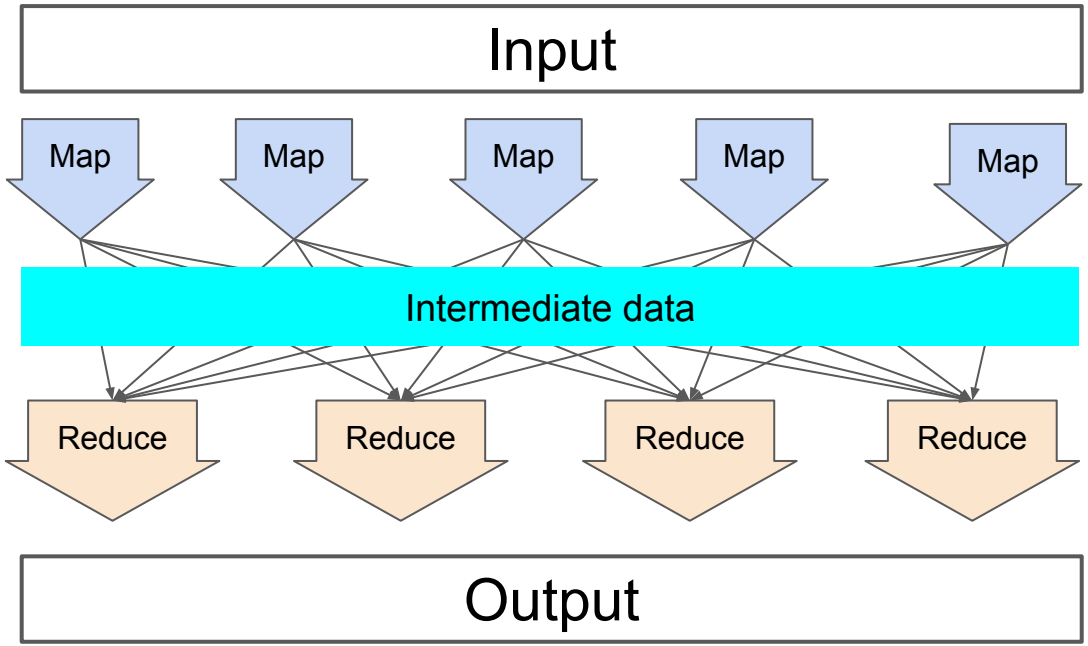
Data Processing

- MapReduce: unified map and shuffle phase



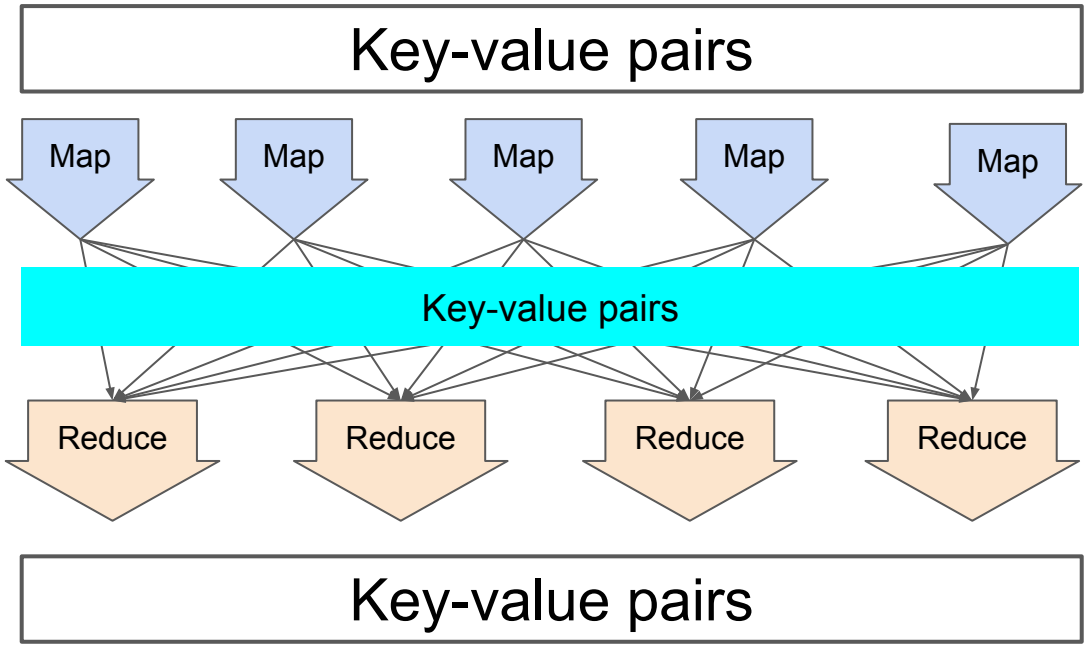
Data Processing

- MapReduce: unified map and shuffle phase



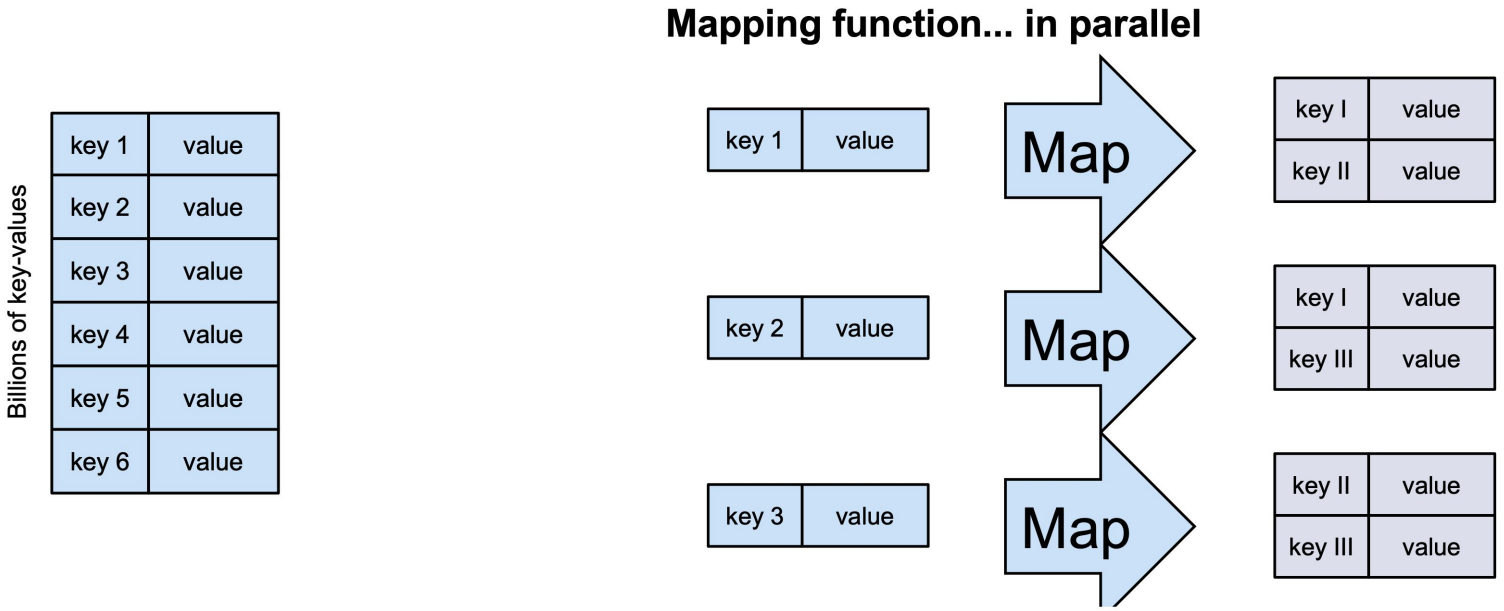
Data Processing

- MapReduce: data types



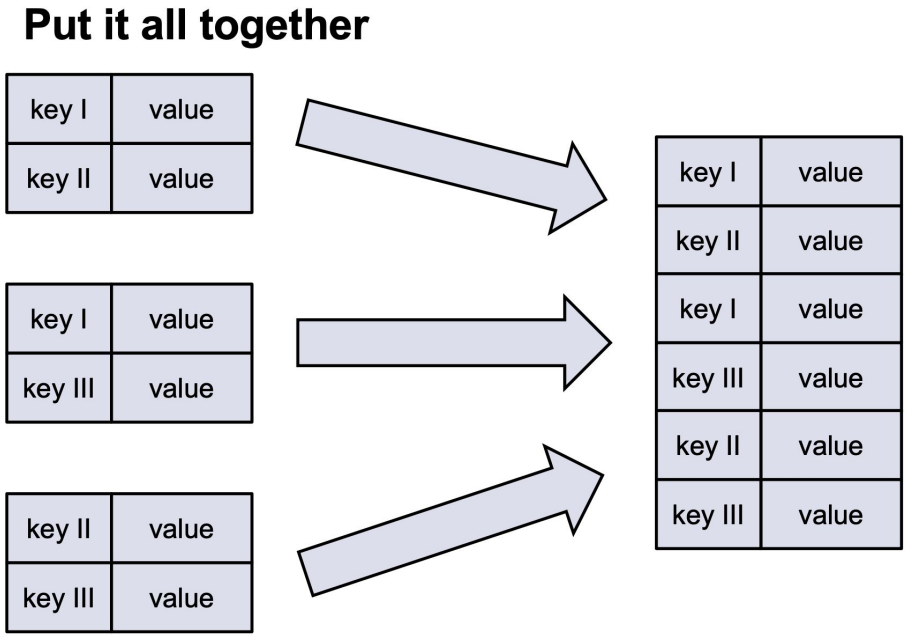
MapReduce

- Step 1: Map phase



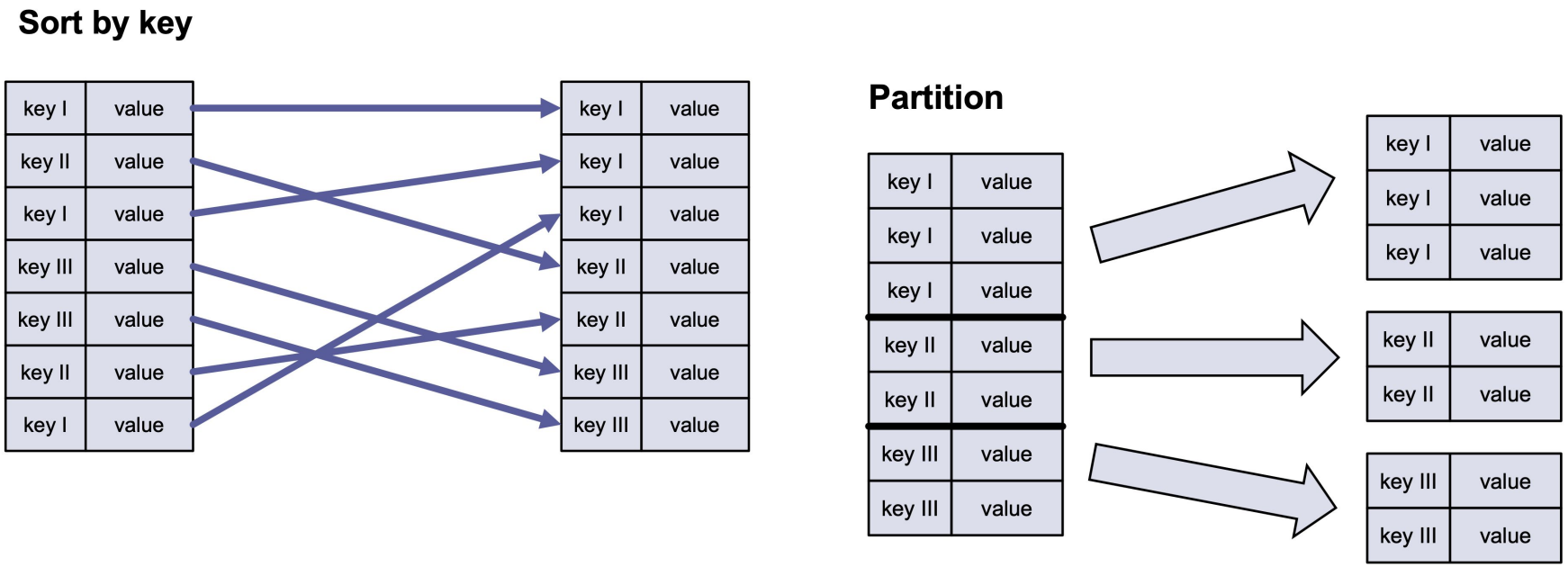
MapReduce

- Step 2: Shuffle phase



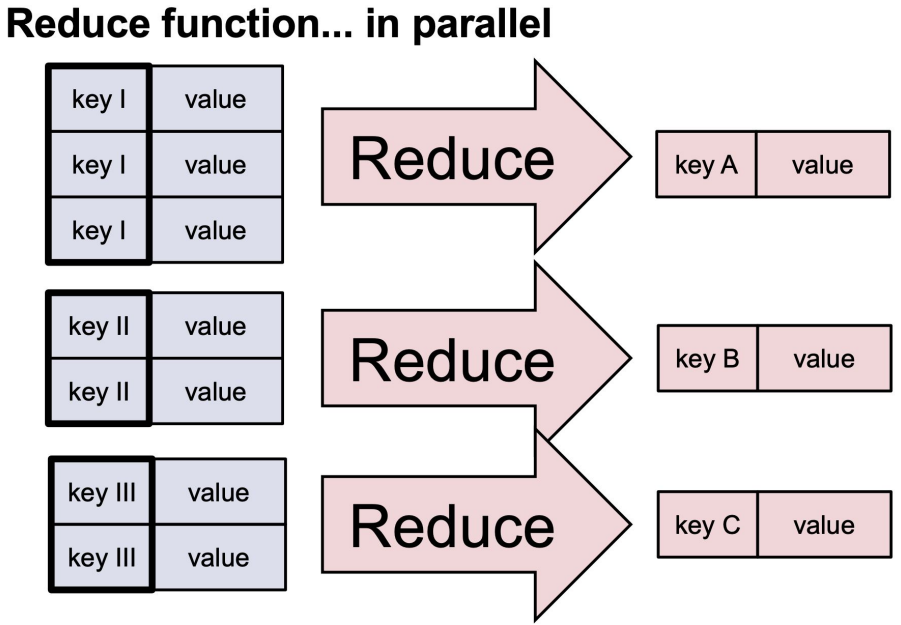
MapReduce

- Step 2: Shuffle phase



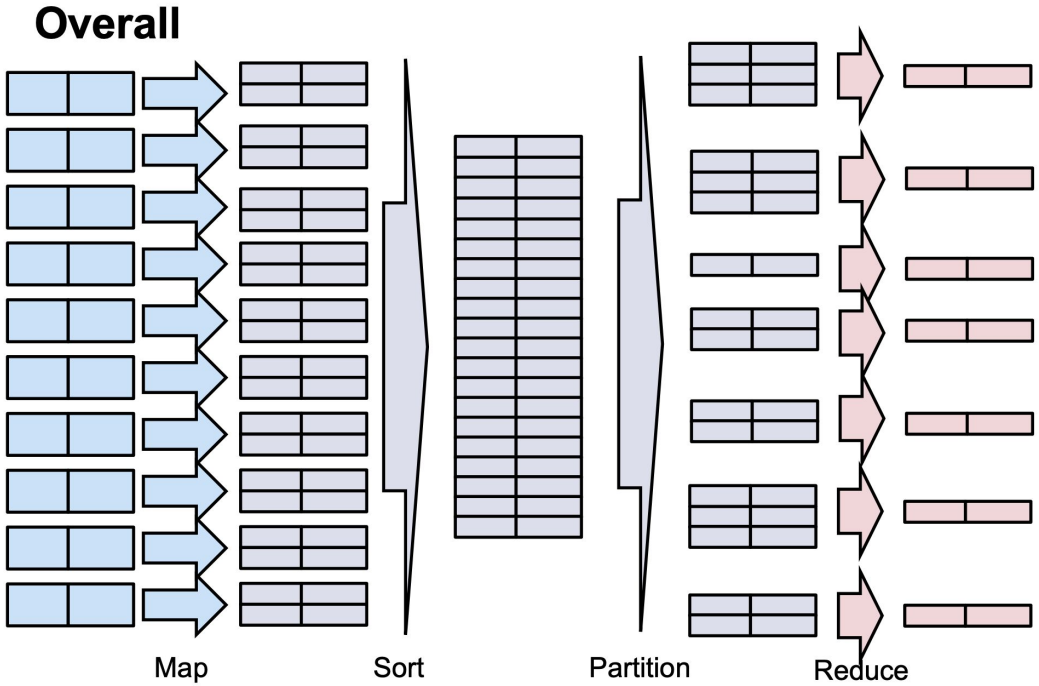
MapReduce

- Step 3: Reduce phase



MapReduce

- Summary

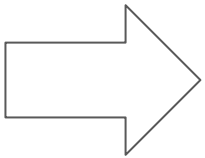


MapReduce Example

WordCount

- Count the occurence of each word in the document

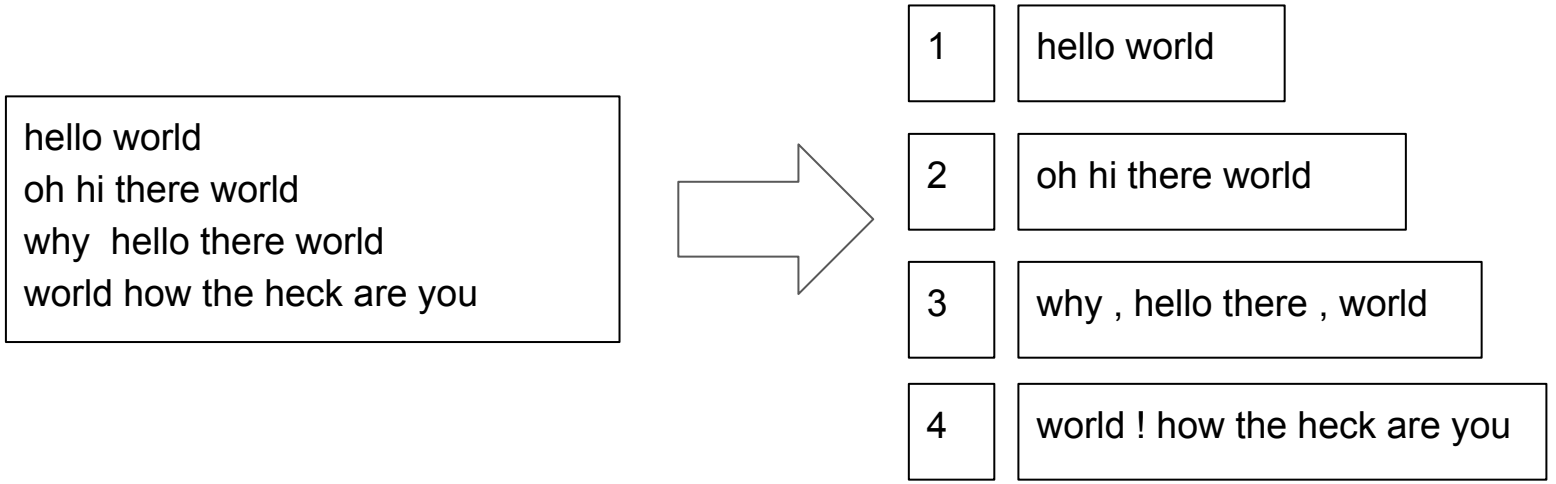
hello world
oh hi there world
why , hello there , world
world ! how the heck are you



hello: 2
world: 4
oh: 1
hi: 1
there: 2
the: 1
heck: 1
are: 1
you: 1
how: 1
why: 1

WordCount

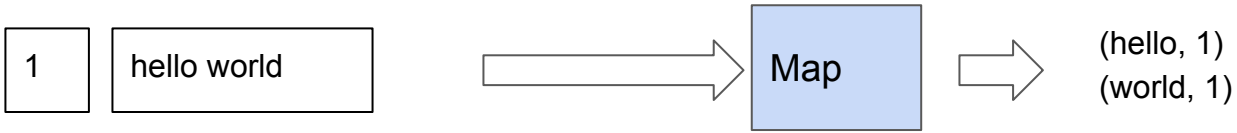
- Step 0: text input becomes key-value



WordCount

- Step 1: Map

```
map(x):
  xs = x.split()
  for s in xs:
    emit((s,1))
```

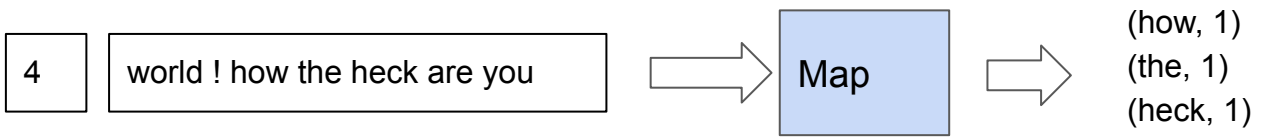


WordCount

- Step 1: Map

```
map(x):
  xs = x.split()
  for s in xs:
    emit((s,1))
```

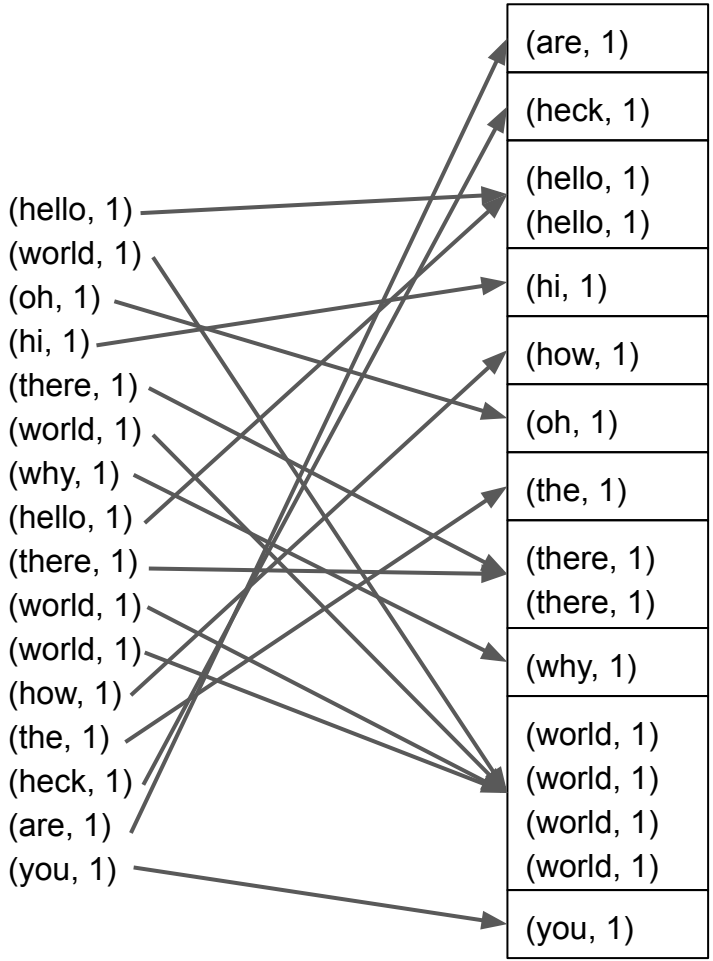
(why, 1)
(hello, 1)
(there, 1)
(world, 1)



(world, 1)
(how, 1)
(the, 1)
(heck, 1)
(are, 1)
(you, 1)

WordCount

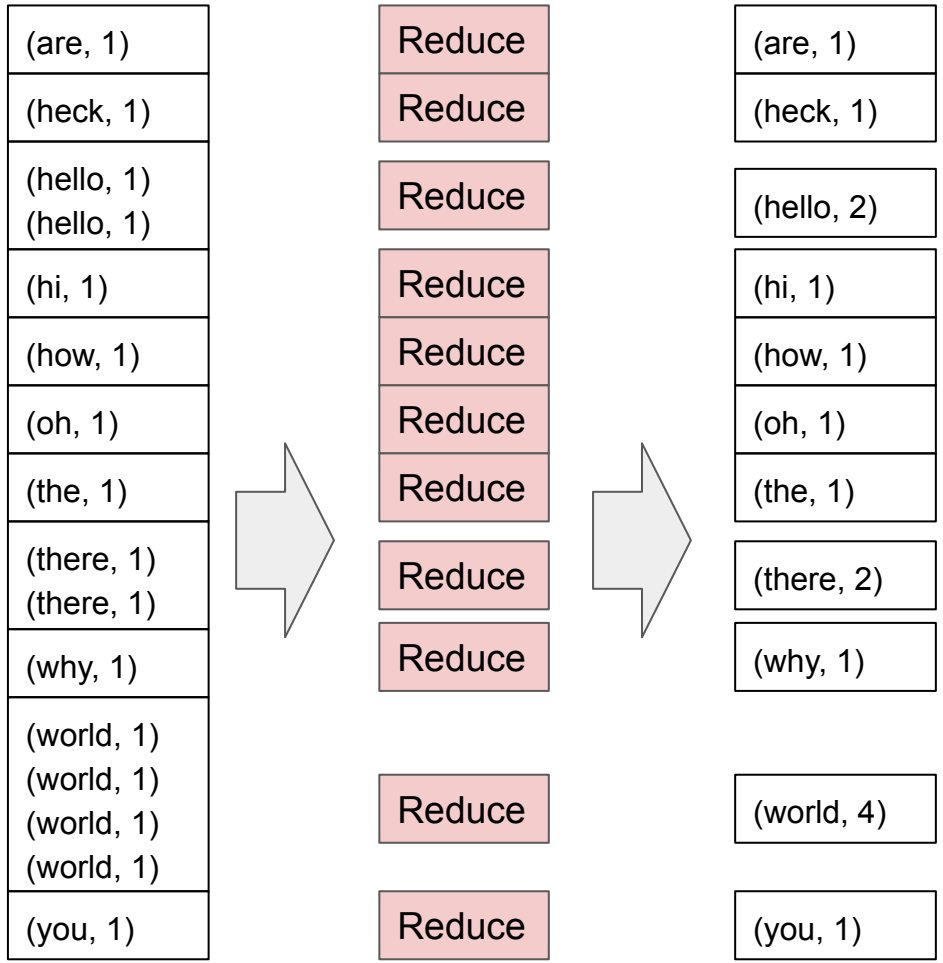
- Step 2: Shuffle



WordCount

- Step 3: Reduce

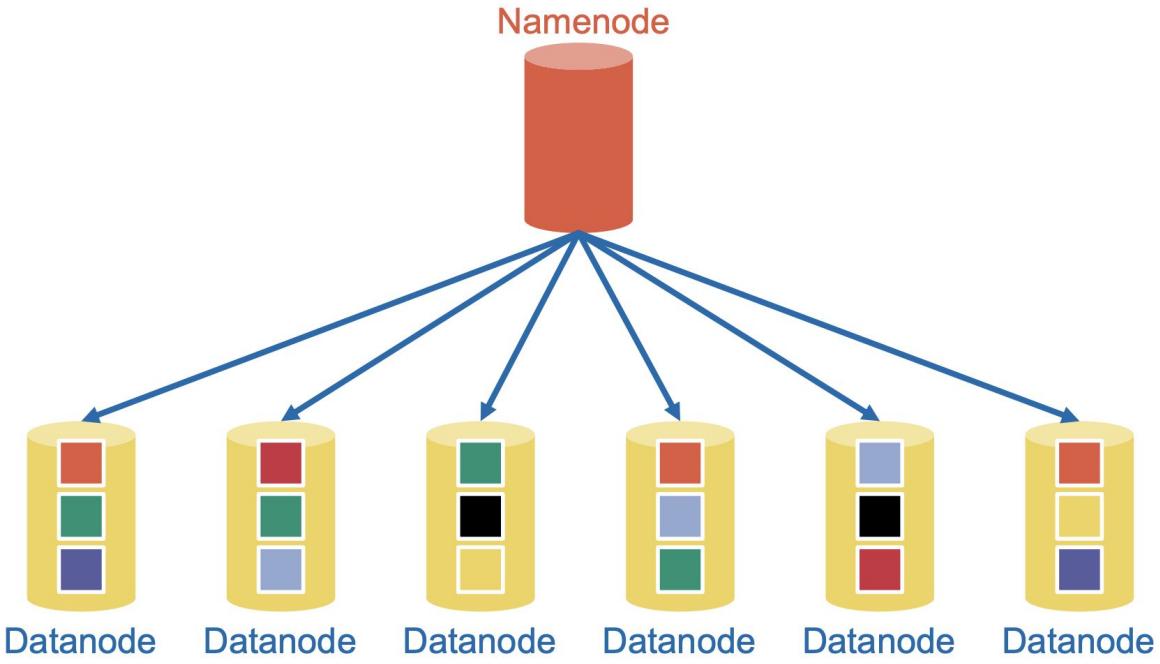
```
reduce(x, values):
  emit((x, sum(values))
```



MapReduce Architecture

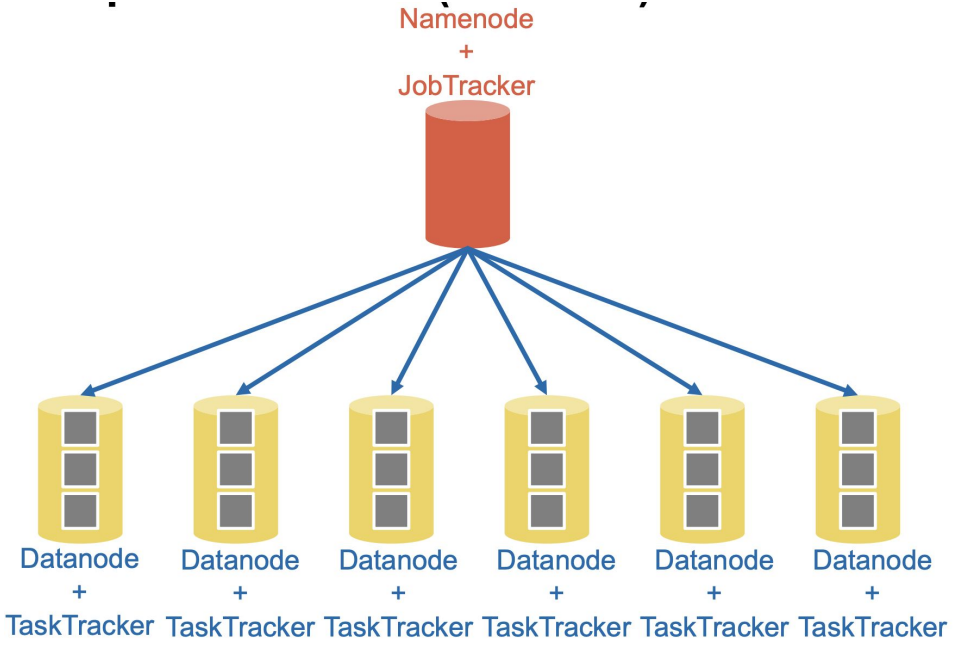
Hadoop Infrastructure

- Recall HDFS



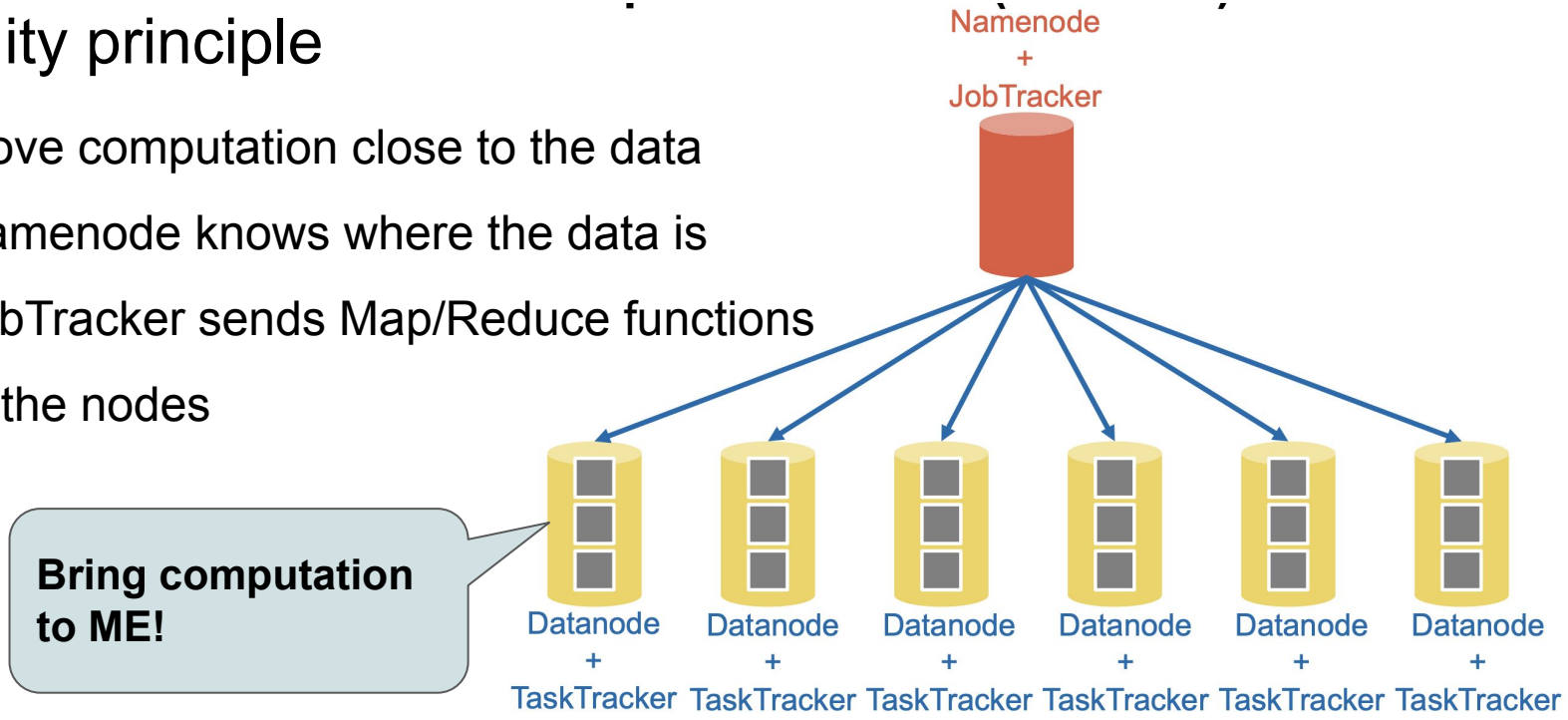
MapReduce Architecture

- Built on top of HDFS
  - JobTracker: master
    - Coordinate MapReduce execution
  - TaskTracker: slave
    - Actually execute it



MapReduce Architecture

- Locality principle
  - Move computation close to the data
  - Namenode knows where the data is
  - JobTracker sends Map/Reduce functions to the nodes

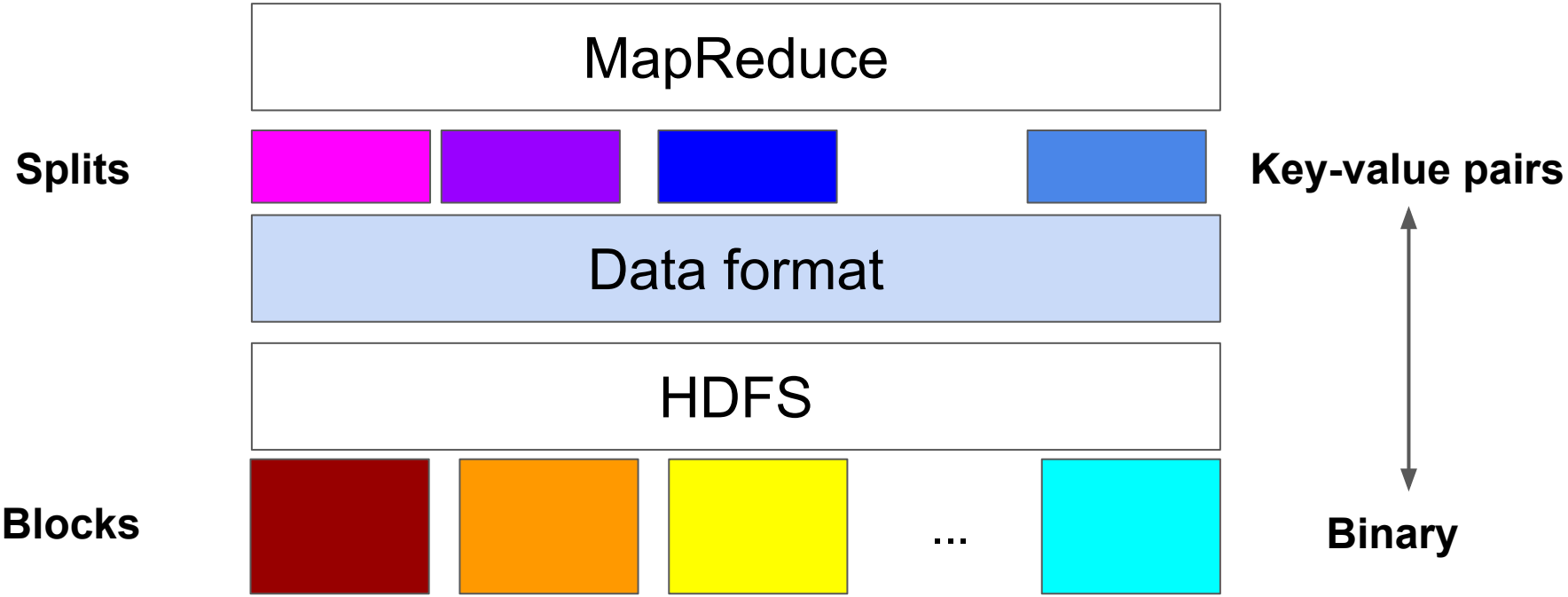


MapReduce Architecture

- Split vs. block

MapReduce Input Split VS Hadoop Block

If input split size is less than the block size, you will end up with using more # mappers vice versa.



BLOCK

Block is a continuous location on the hard drive where data is stored.
HDFS stores files into blocks.
Block -> min amount of data that can be read/write.

Split:

The data to be processed by an individual Mapper is represented by InputSplit.
#Map task = # inputsplit.
InputSplit doesnt contain actual data but a reference to the data.

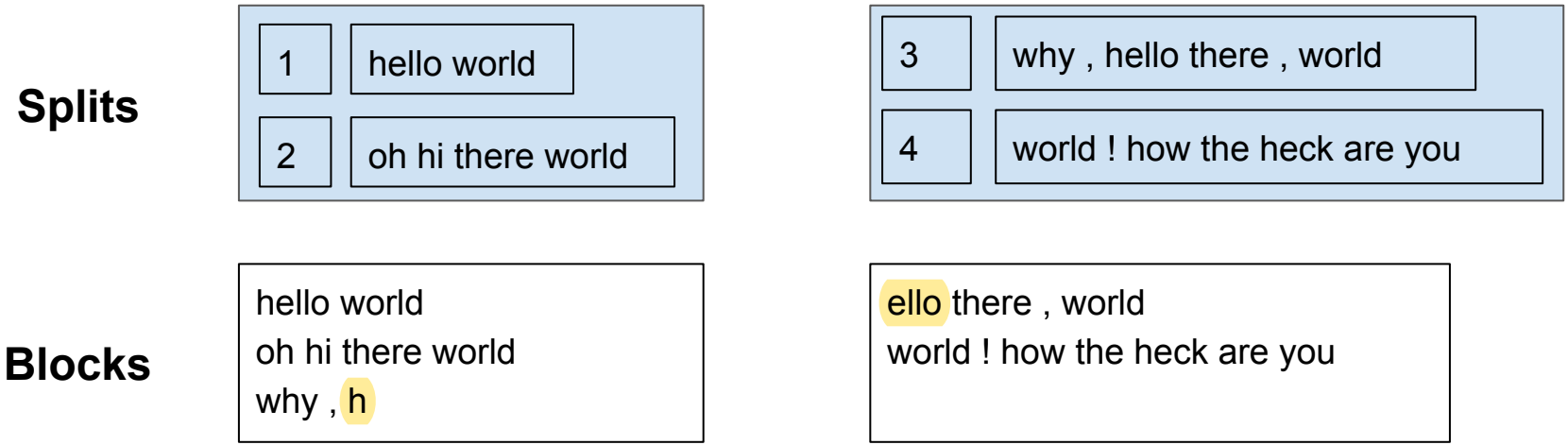
*During **MapReduce** execution, **Hadoop** scans through the **blocks** and create **InputSplits** and each **inputSplit** will be assigned to individual mappers for processing. Hence, Split act as a broker between block and mapper.*

Consider an example, where we need to store the file in HDFS. HDFS stores files as blocks. Block is the smallest unit of data that can be stored or retrieved from the disk and the default size of the block is 128MB. HDFS break files into blocks and stores these blocks on different nodes in the cluster. Suppose we have a file of 130 MB, so HDFS will break this file into 2 blocks. Now, if we want to perform MapReduce operation on the blocks, it will not process, because the 2nd block is incomplete. Thus, this problem is solved by InputSplit. InputSplit will form a logical grouping of blocks as a single block, because the InputSplit include a location for the next block and the byte offset of the data needed to complete the block.

MapReduce Architecture

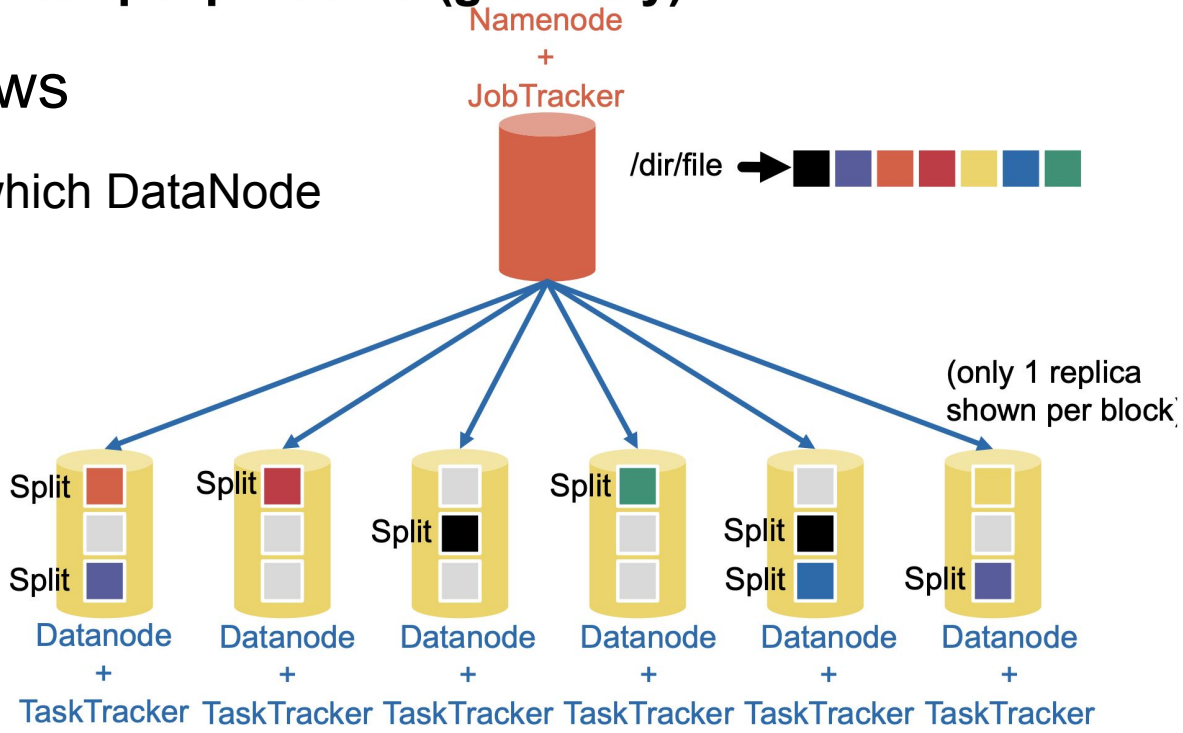
- Split vs. block:
  - Ideal case: split = block
  - But sometimes not. **Why?**

Block *is the physical representation of data.* ***Split*** *is the logical representation of data present in Block*



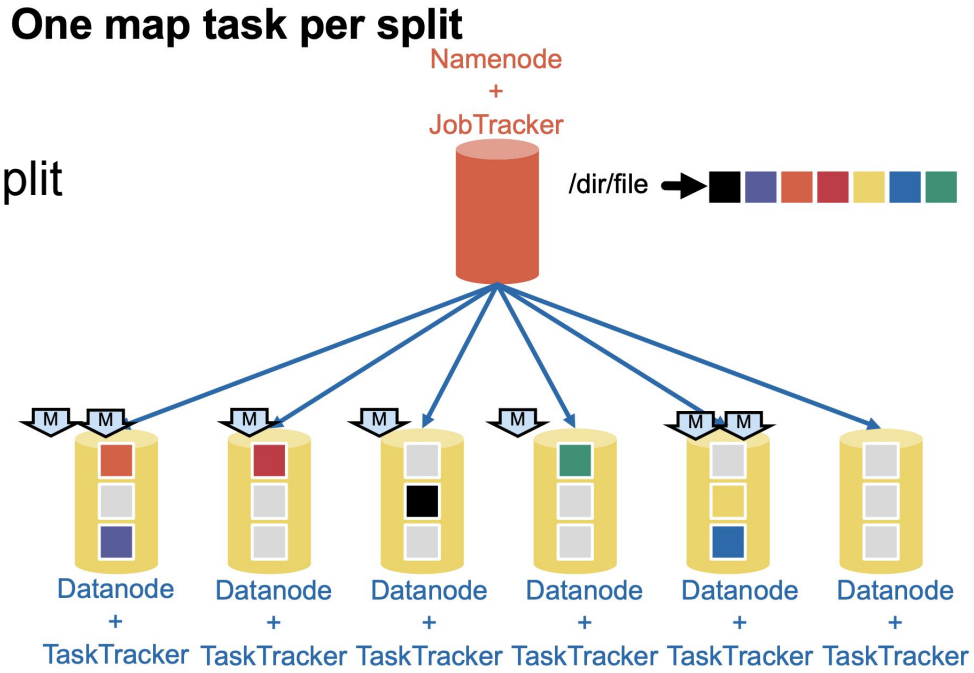
MapReduce

- Job tracker knows
  - Which splits in which DataNode



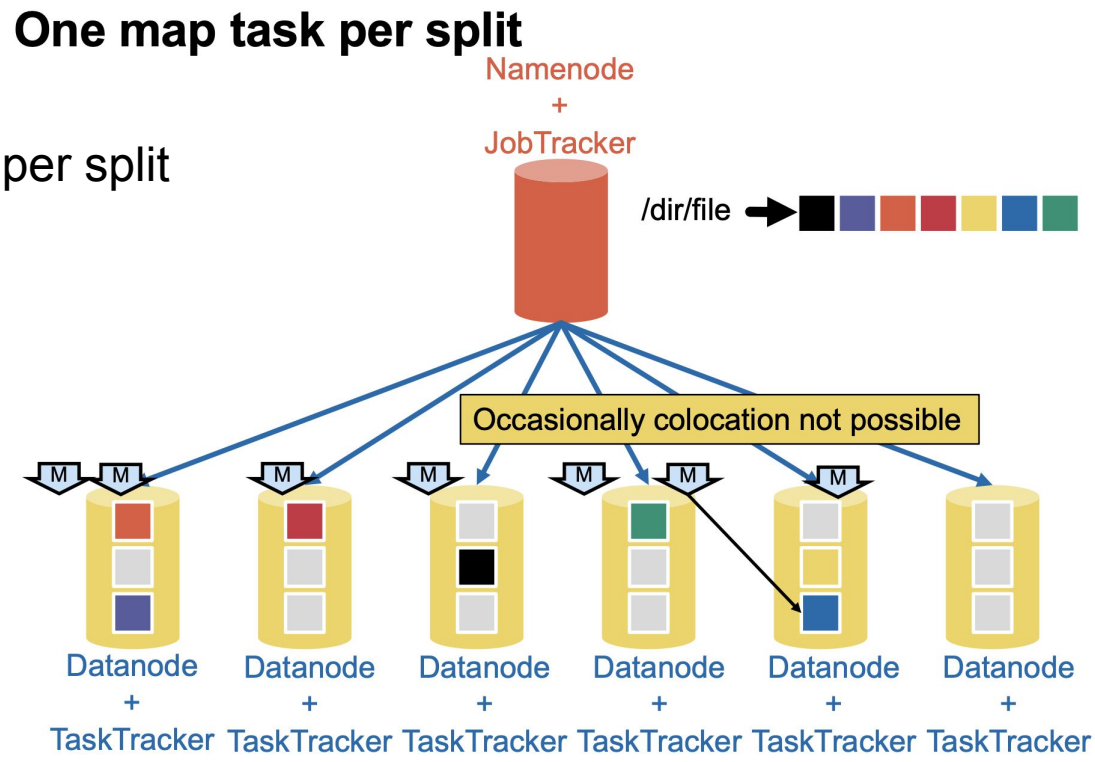
MapReduce

- Job tracker
  - Sends one map task per split



MapReduce

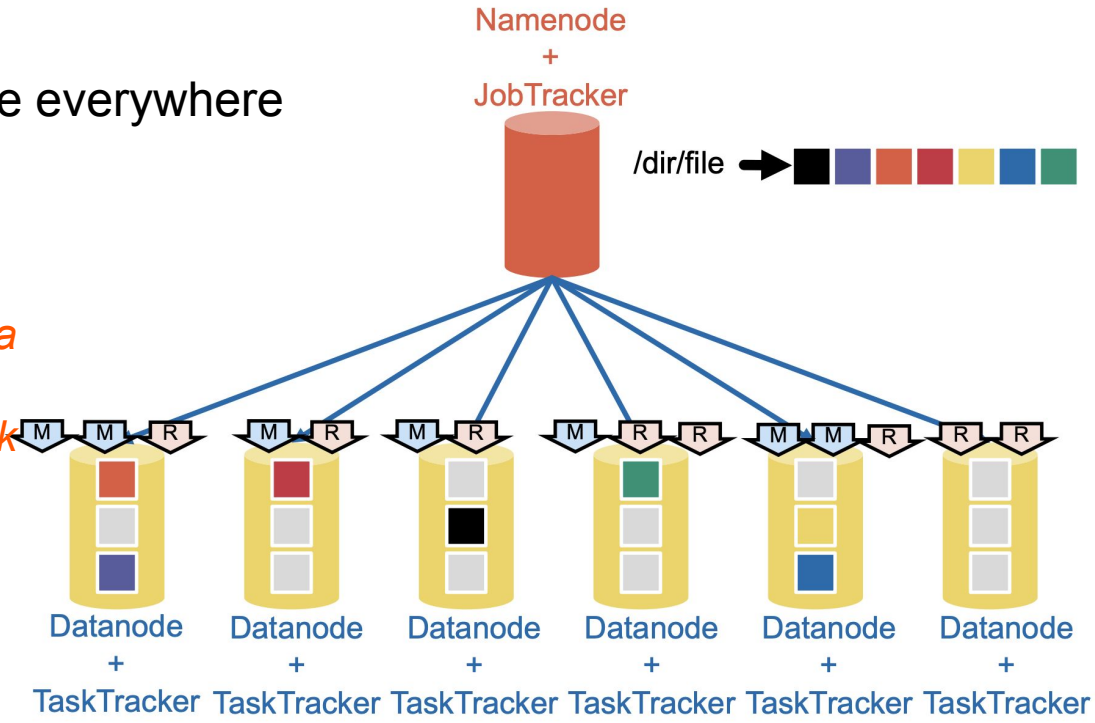
- Job tracker
  - Sends one map task per split



MapReduce

- Job tracker
  - Reduce tasks can be everywhere
  - **Why?**

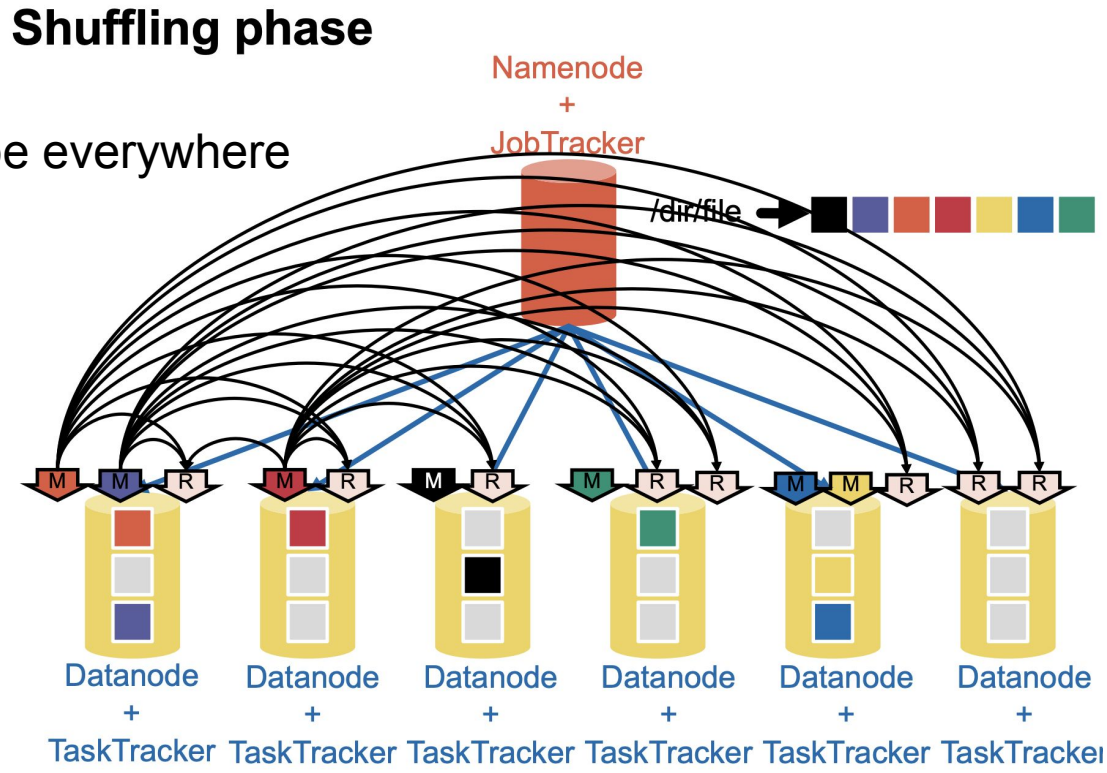
Reduce tasks



Mapping *-> need to be at the data nodes*
Reducer *-> does not need to work on the data node itself, gets the data from map stage*

MapReduce

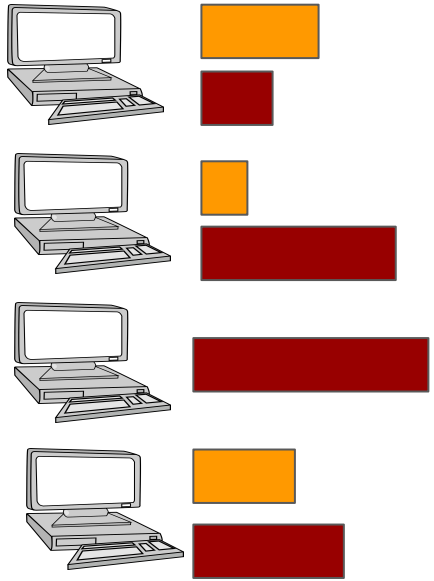
- Job tracker
  - Reduce tasks can be everywhere
  - **Why?**



MapReduce

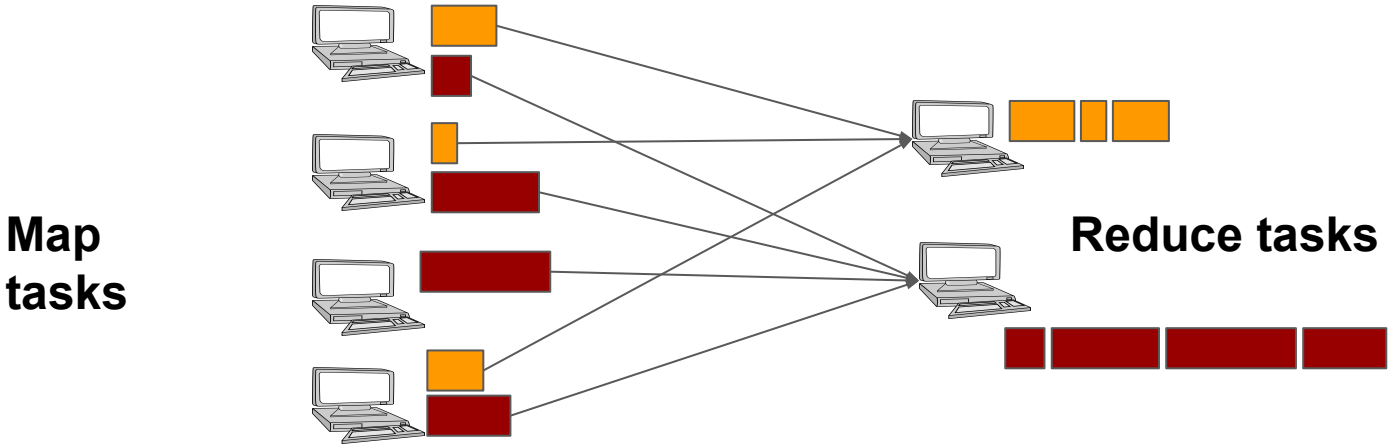
- TaskTracker:
  - Coordinate execution of map/reduce tasks
  - Sort map outputs, then spill to disk
- Shuffling phase
  - Output from map tasks stored on disks
    - Intermediate files
  - Reduce tasks fetch them over HTTP

Map tasks

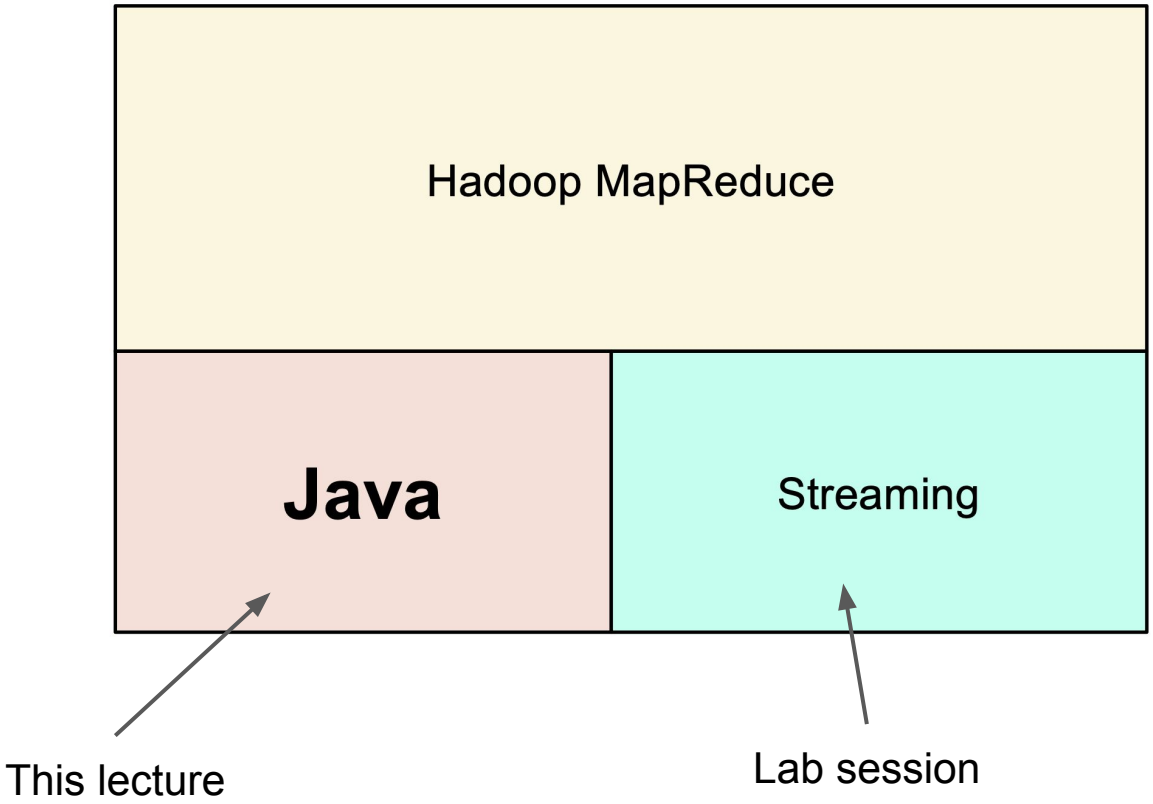


MapReduce

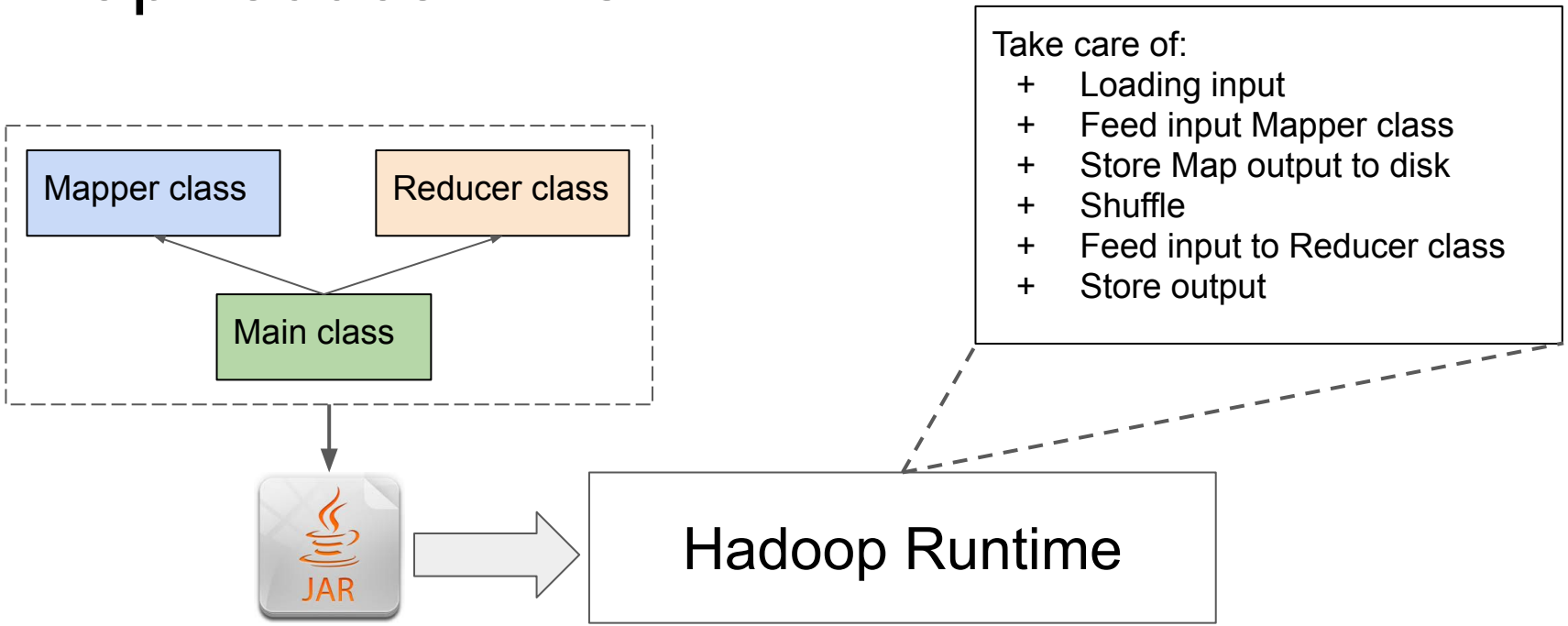
- Shuffling phase
  - Each map tasks stores output to **intermediate files** on local disk
    - Upto 1 file for 1 reduce task
  - Reduce tasks fetch these intermediate files over HTTP



MapReduce APIs



MapReduce APIs

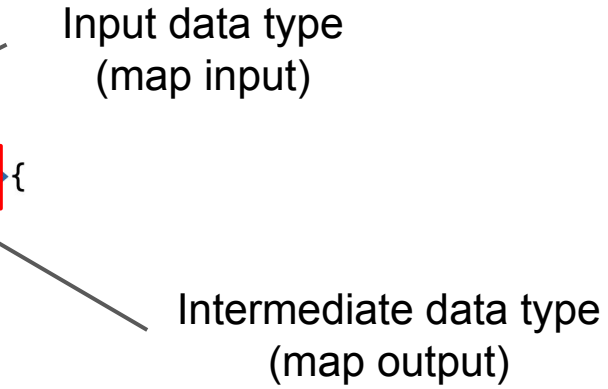



```
hadoop jar <job>.jar <class name> <input> <output> [options]
```

MapReduce APIs

Java API: Mapper

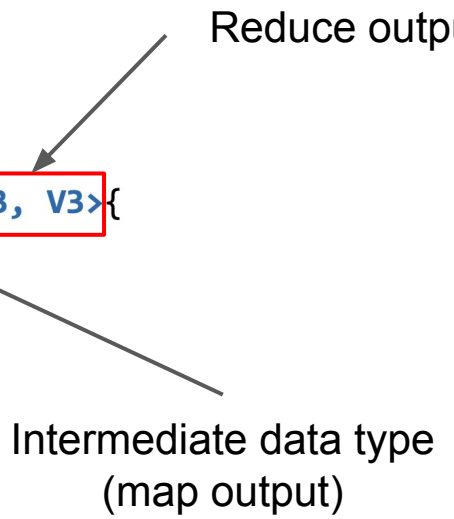
```
import org.apache.hadoop.mapreduce.Mapper;

public class MyOwnMapper extends Mapper<K1, V1, K2, V2>{

  public void map(K1 key, V1 value, Context context)
    throws IOException, InterruptedException
  {

        ….
        // K2 newK = ..
        // V2 newV = ...
        // context.write(newK, newV)

  }

}
```



MapReduce APIs

Java API: Reducer

```
import org.apache.hadoop.mapreduce.Reducer;

public class MyOwnReducer extends Reducer<K2, V2, K3, V3>{

  public void reduce
    (K2 key, Iterable<V2> values, Context context)
    throws IOException, InterruptedException
  {

        ….
        // K3 newK = ..
        // V3 newV = ...
        // context.write(newK, newV)

  }

}
```



MapReduce APIs

Java API: Job

Boilerplate code

```
import org.apache.hadoop.mapreduce.Job;

public class MyMapReduceJob {

   public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      Job job = Job.getInstance(conf, "word count");

      job.setMapperClass(MyOwnMapper.class);
      job.setReducerClass(MyOwnReducer.class);

      FileInputFormat.addInputPath(job, ...);
      FileOutputFormat.setOutputPath(job, ...);

      System.exit(job.waitForCompletion(true) ? 0 : 1);
   }
```

Summary

- MapReduce was a new computing paradigm in 2004
- Changed the database landscape
- Inspired many other systems
  - **Spark**
  - Flink
  - Dataflow
- But now deceased!

Question 1

Suppose you have a big data set (many terabytes) with many records of the following form:

```
(productID, supplierID, price)
```

where `productID` identifies a unique product, `supplierID` identifies a unique supplier, and `price` is the sale price.

You want a list of suppliers, listing for each supplier the average sale price of all items by the supplier.

Write a MapReduce job for this task.

Question 1

(productID, supplierID, price)
Want a list of suppliers, listing for each supplier the average sale price of all items by the supplier.

```
map(key, val):
   // key can be ignored
   // val contain the record
   // you can do like val.price to get the price

   …

   // you can use emit(key, val) to output
   // the result of the map function
```

```
map(key, val):
    emit(val.supplierID, val.price)

reduce(key, [val]):
    // average the prices across
one same key
    avgprice = avg(val)
    emit(key, avgprice)
```

** assume there is an average helper function*
or you can do like sum(val)/val.size

```
reduce(key, [val]):
   // [val] is a list of values for the reduce key

   // you can use emit(key, val) to output results
   // to HDFS
```

Question 2

Assume the following about your MapReduce job:

- Input size: 1TB
- Block size: 100MB
- Number of reduce tasks: 10,000
- Splits and blocks are perfectly aligned.

[Q1]: How many map tasks are there?

[Q2]: How many intermediate files are there?

Q1

1TB / 100MB = 1000 GB / 0.1GB
= 10,000 map tasks

(1 Map task read 1 block and run map function through all the key values of that block)

Q2

10^4 x 10^4 = 10^8

(For each map task, they need to output #reducer files)
(e.g. for each mapper, output 1 file for each word (for word count))