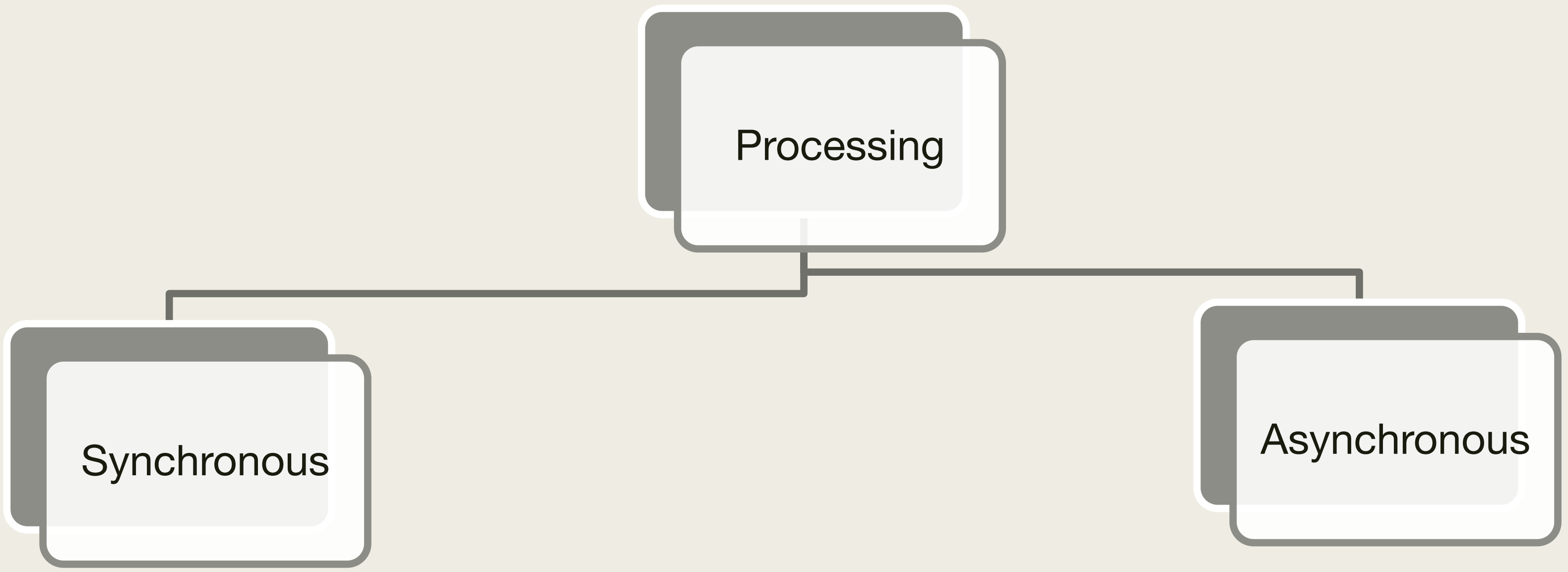
BATCH PROCESSING WITH MAP REDUCE

Prasad M Deshpande

Patterns in processing

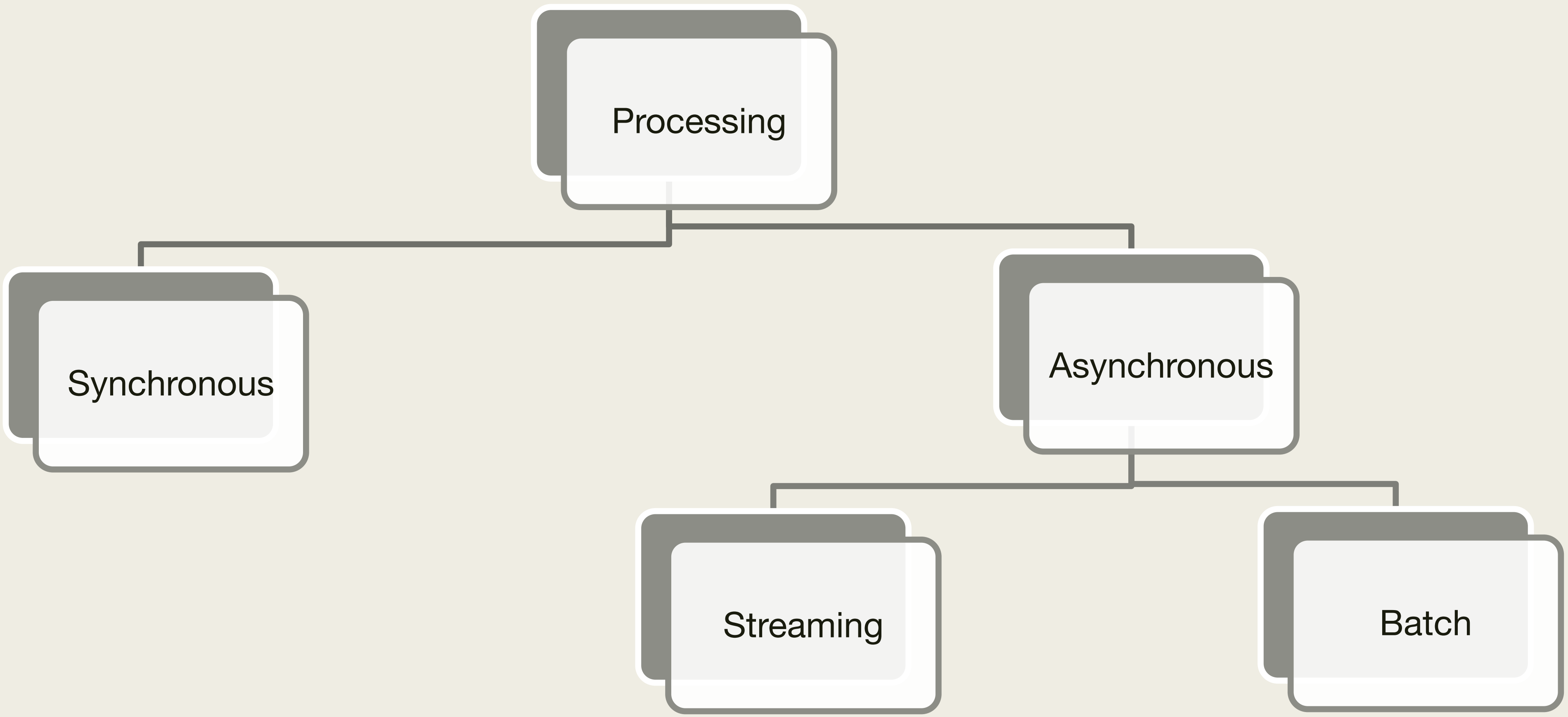


Synchronous vs Asynchronous

- Synchronous
  - *Request is processed and response sent back immediately*
  - *Client blocks for a response*
  - *Example - REST apis, GRPC*

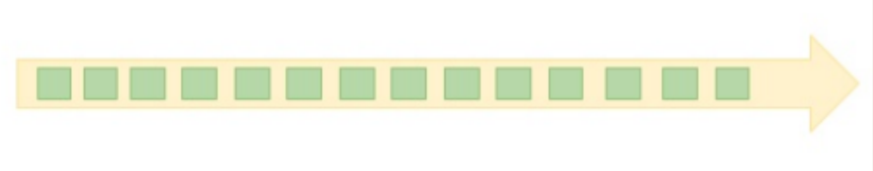
- Asynchronous
  - *Request is sent as an event/message*
  - *Client does not block*
  - *Event is put in a queue/file and processed later*
  - *Response is generated as another event*
  - *Consumer of response event can be a different service*
  - *Example - Kafka, Callbacks*

Patterns in processing



Data at rest Vs Data in motion

- At rest:
  - Dataset is fixed (file)
  - bounded
  - can go back and forth (iterate) on the data
- In motion:
  - continuously incoming data (queue)
  - unbounded
  - too large to store and then process
  - need to process in one pass



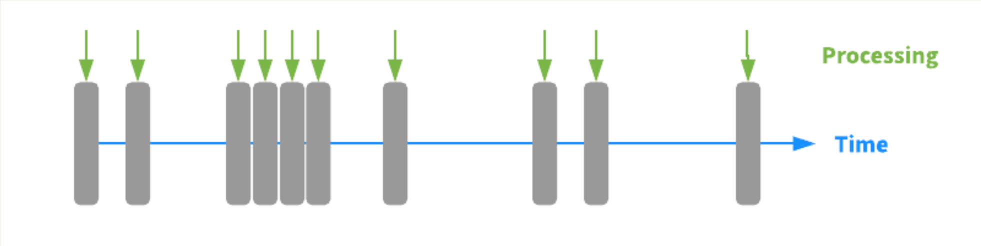
Batch processing

- Problem statement :
  - Process this entire data
  - give answer for X at the end

- Characteristics
  - Access to entire data
  - Split decided at the launch time.
  - Capable of doing complex analysis (e.g. Model training)
  - Optimize for Throughput (data processed per sec)
- Example frameworks : Map Reduce, Apache Spark

Stream processing

- Problem statement :
  - Process incoming stream of data
  - to give answer for X at this moment.



- Characteristics
  - Results for X are based on the current data
  - Incremental processing
  - Computes function on one record or smaller window.
  - Optimizations for latency (avg. time taken for a record)
- Example frameworks: Apache Storm, Apache Flink, Amazon Kinesis, Kafka, Pulsar

Batch vs Streaming

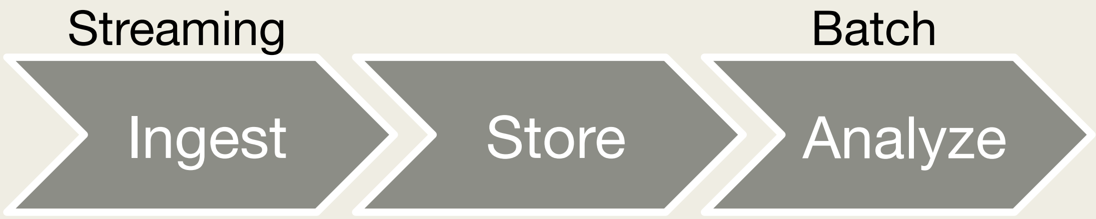
- Find stats about a group in a closed room
- Analyze sales data for last month to make strategic decisions

- Count number of runners crossing a point in a marathon
- Monitoring the health of a data center

When to use Batch vs Streaming

- Batch processing is designed for ‘data at rest’. ‘Data in motion’ becomes stale; if processed in batch mode.
- Real-time processing is designed for ‘data in motion’. Can be used for ‘data at rest’ as well (in many cases).

	Simple	Complex Iterative
Real time	Stream	Stream/ Batch
Non real time	Stream/ Batch	Batch



Big data flow

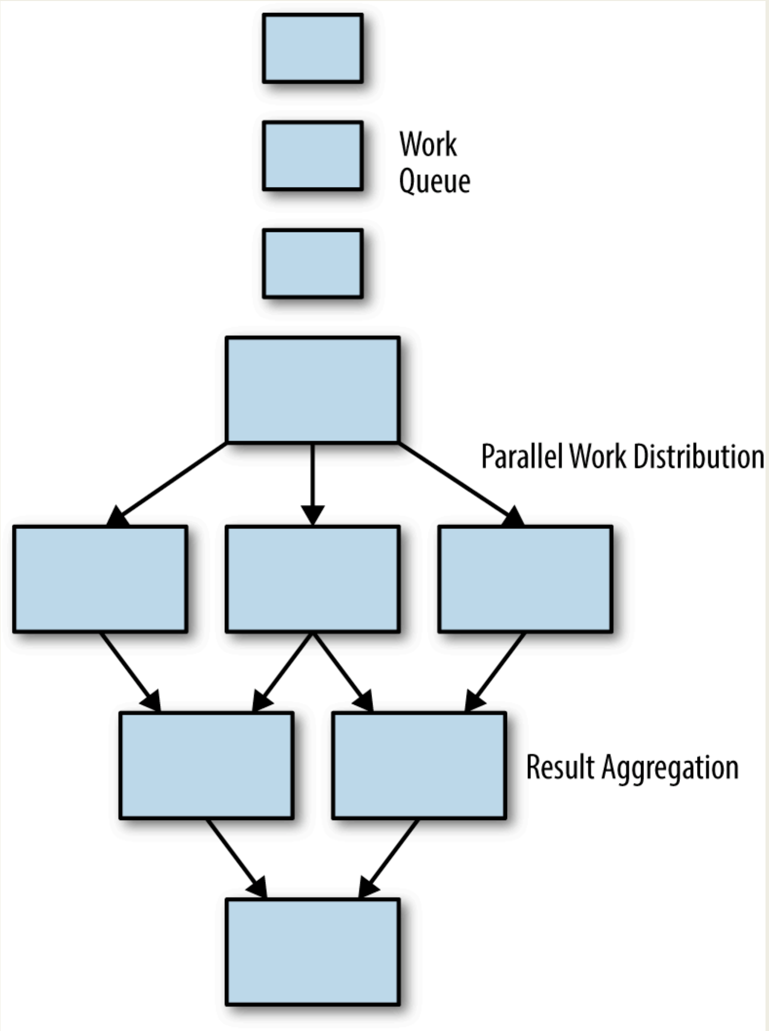
Design goals of batch processing systems

- Fast processing
  - *Data ought to be in primary storage, or even better, RAM*
- Scalable
  - *Should be able to handle growing data volumes*
- Reliable
  - *Should be able to handle failures gracefully*
- Ease of programming
  - *Right level of abstractions to help build applications*
- Low cost

➢ Need a whole ecosystem

Processing flows

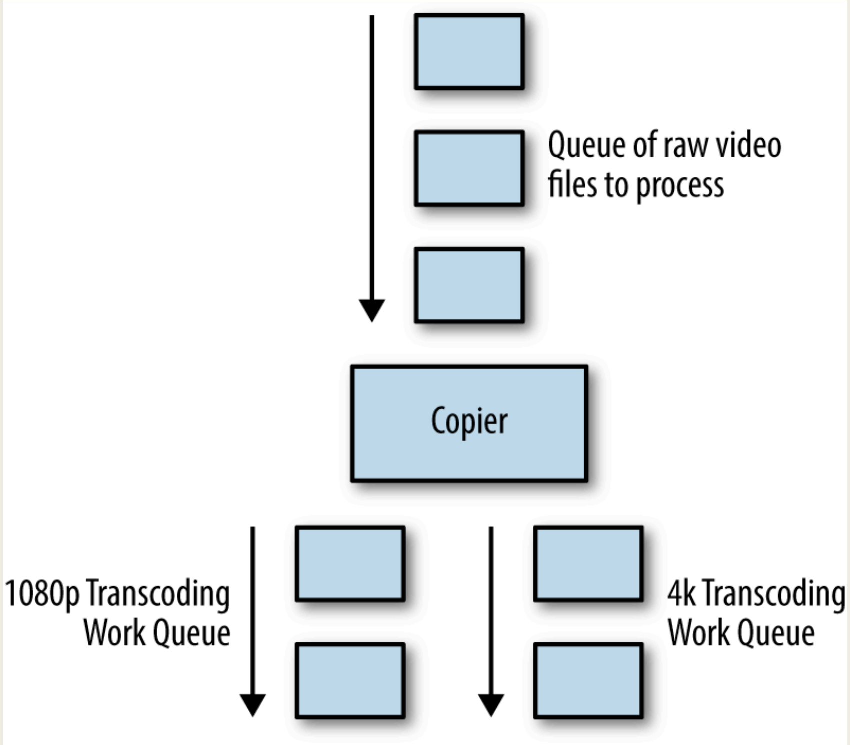
- Flow of work through a directed, acyclic graph
- Different operators for coordinating the flow
- Lets look at some common patterns



Copier

Example use case: process raw video files - transcode at 1080p and 4k resolution

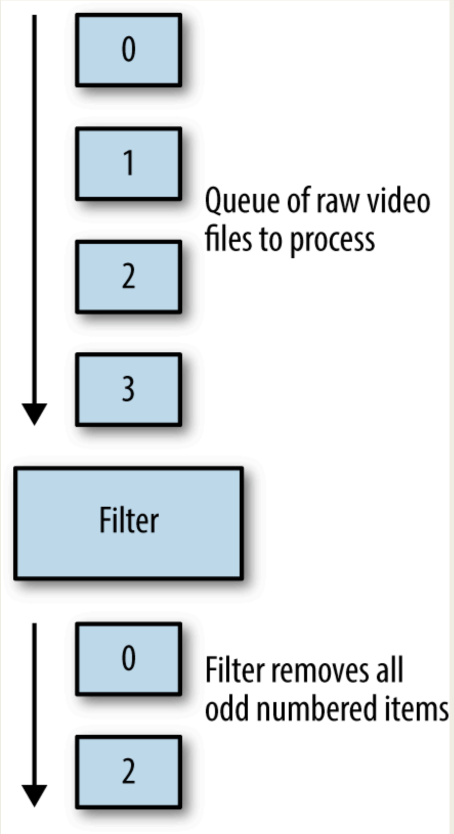
- Duplicate input to multiple outputs
- Useful when different independent processing steps need to be done on same input



Filter

Example use case: process raw video files - filter out all odd number items

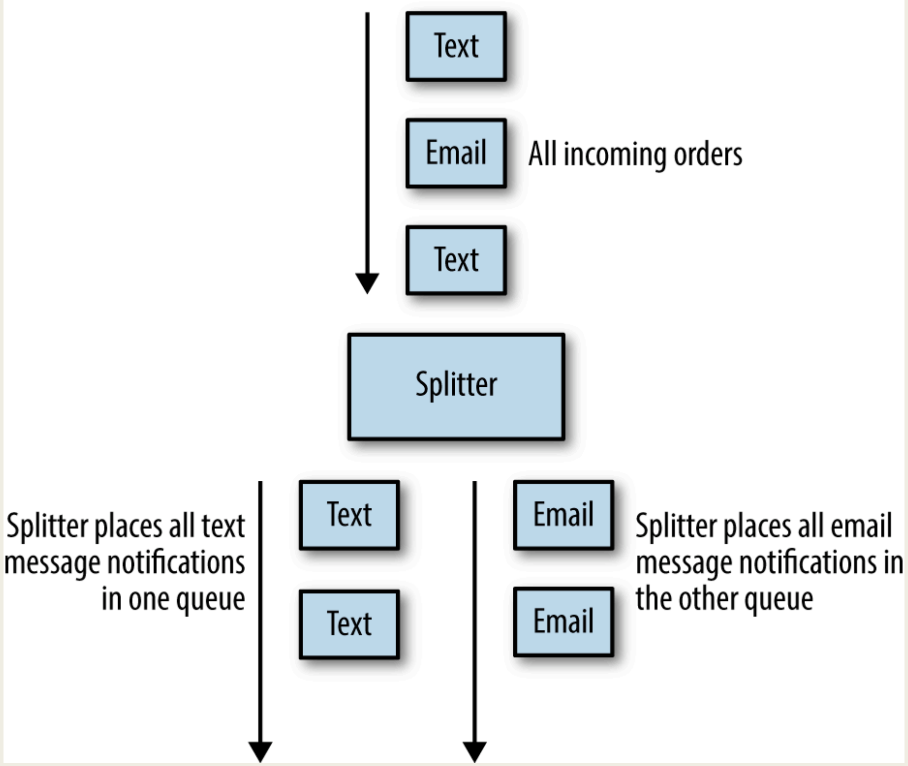
- Select a subset of the input items
- Usually based on a predicate on the input attribute values



Splitter

Example use case: separate out email from text notifications

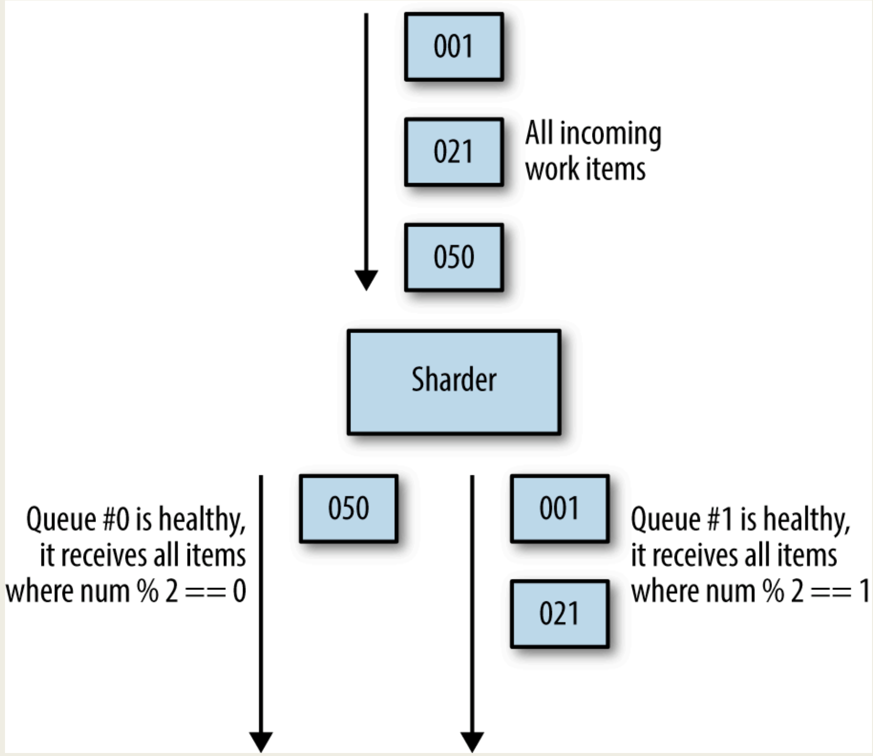
- Split input set into two or more different output sets
- Partitioning vs copy
- Usually based on some predicate – different processing to be done for each partition



Sharding

Example use case: parallelise the processing when identical processing is needed on all items

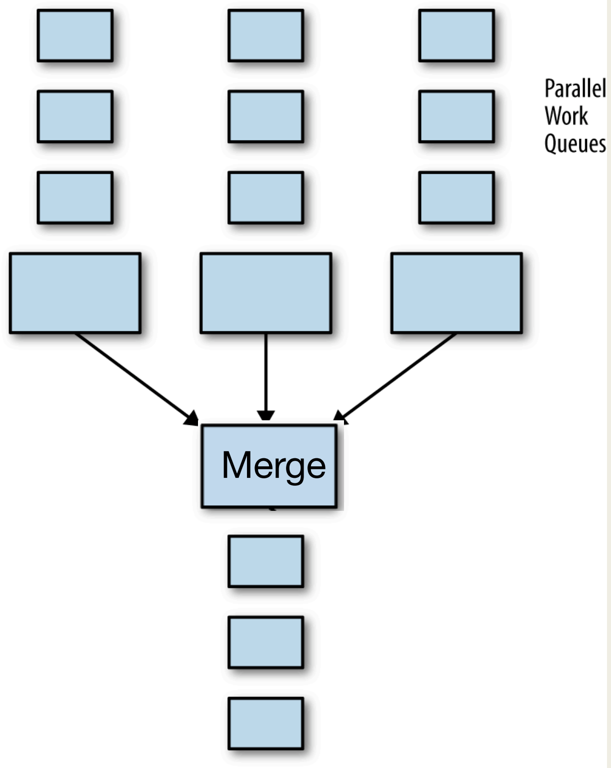
- Split based on some sharding function
- Same processing for all parititions
- Reasons for sharding
  - *To distribute load among multiple processors*
  - *Resilience to failures*



Merge

Example use case: combine the work done in parallel

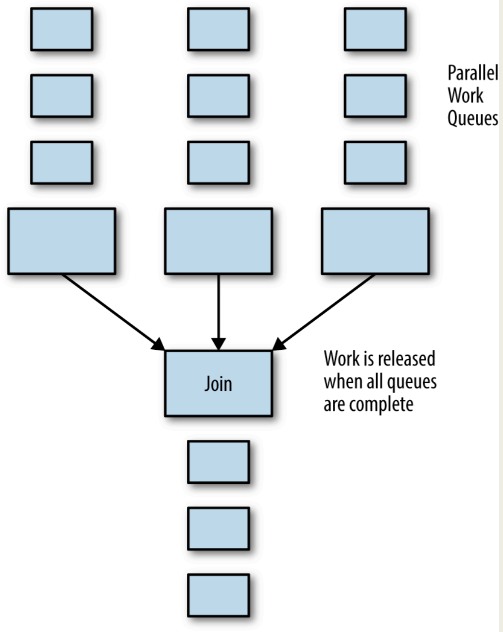
- Combine multiple input sets into a single output set
- A simple union



Join

Example use case: send out a summary email after all the work is done

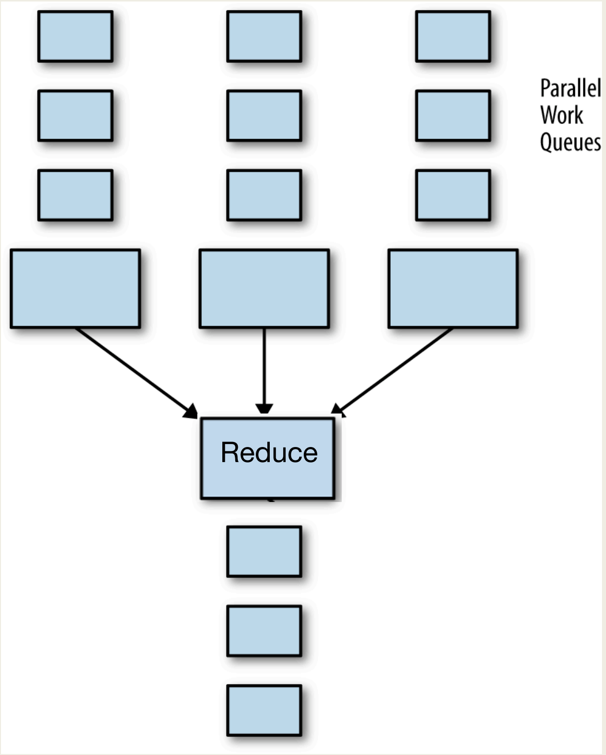
- Barrier synchronization
- Ensures that previous step is complete before starting the next step
- Reduces parallelism



Reduce

Example use case: count the number of emails and text messages sent

- Group and merge multiple input items into a single output item
- Usually, some form of aggregation
- Need not wait for all input to be ready



A simple problem

- Find transactions with sale >= 10
- Which patterns will you use?
- How will you parallelize?

Product	Sale
P1	10
P2	15
P1	5
P2	40
P5	15
P1	55
P2	10
P5	30
P3	25
P3	15

Copy, Filter, Split, Shard, Merge, Join, Reduce

Copy, **Filter**, Split, **Shard**, **Merge**, Join, Reduce

A simple problem - extended

- Find total sales by category for transactions with sale >= 10
- Which patterns will you use?
- How to parallelize?

e.g.: PC1, 105

Category	Product
PC1	P1, P3
PC2	P2, P4, P5

Product	Sale
P1	10
P2	15
P1	~~5~~
P2	40
P5	15
P1	55
P2	10
P5	30
P3	25
P3	15

Copy, Filter, Split, Shard, Merge, Join, Reduce

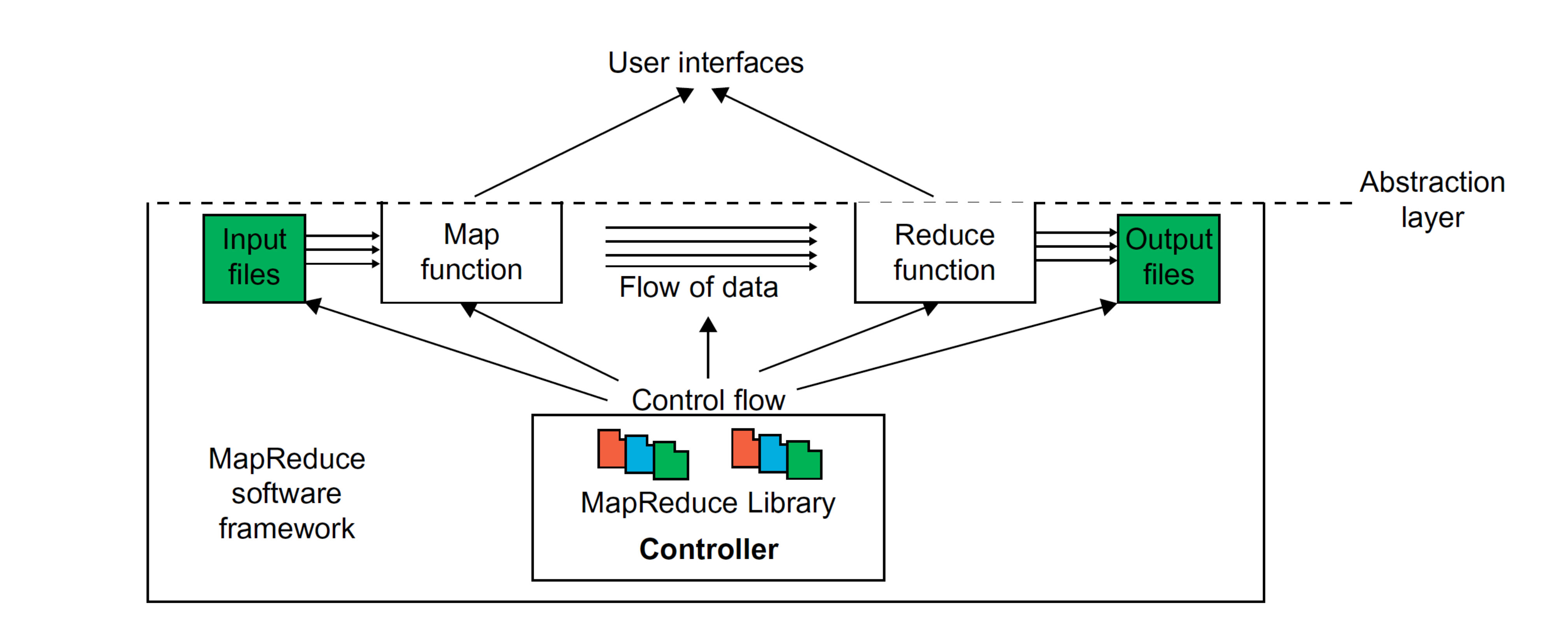
Copy, **Filter**, Split, **Shard**, Merge, Join, **Reduce**

Challenges in parallelization

- How to break a large problem into smaller tasks?
- How to assign tasks to workers distributed across machines?
- How to ensure that workers get the data they need?
- How to coordinate synchronization across workers?
- How to share partial results from one worker to another?
- How to handle software errors and hardware faults?

Programmer should not be burdened with all these details => need an abstraction

Map-reduce


User interfaces
Abstraction layer
Input files
Map function
Flow of data
Reduce function
Output files
Control flow
MapReduce software framework
MapReduce Library
Controller

Abstraction

Two processing layers/stages

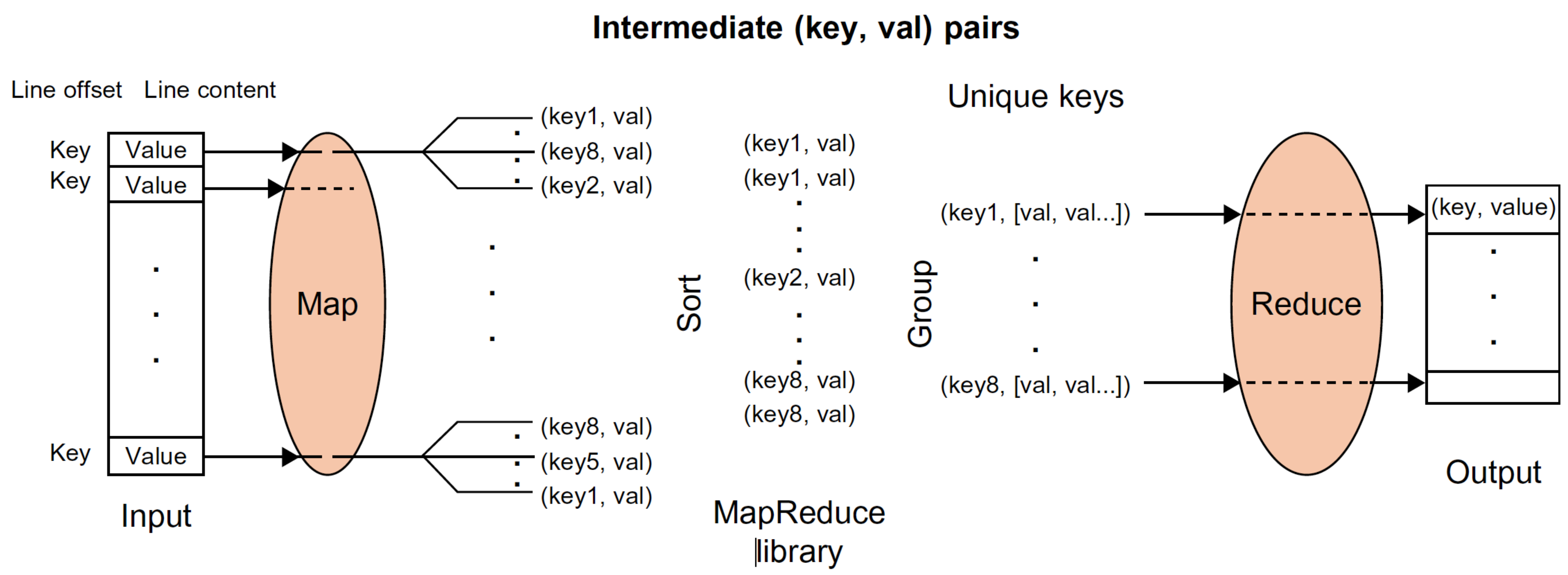
- map: (k1, v1) → [(k2, v2)]
- reduce: (k2, [v2]) → [(k3, v3)]

Revisiting the problem

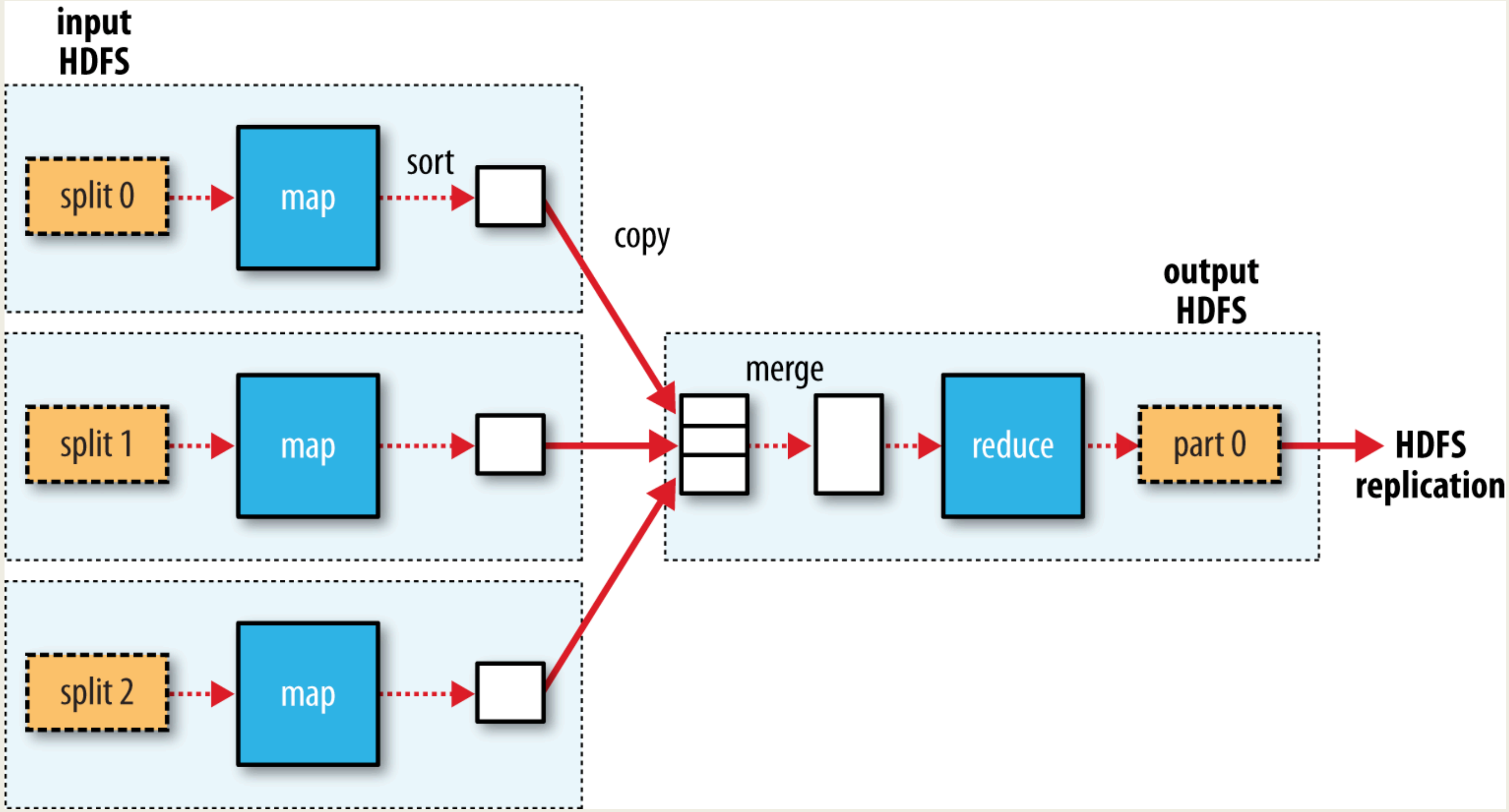
```
public class ProductMapper extends
Mapper<LongWritable, Text, Text, IntWritable> {

    @Override
    public void map(LongWritable key, Text value,
Context context)
            throws IOException,
InterruptedException {
        String line = value.toString();
        String parts[] = line.split(",");

        String product = parts[0];
        Integer sale = Integer.valueOf(parts[1]);

        if (sale >= 10) {
            String category = getCategory(product);
            context.write(new Text(category), new
IntWritable(sale));
        }
    }
}
```

```
public class ProductReducer extends
ReducerReducer<Text, IntWritable, Text,
IntWritable> {

    @Override
    public void reduce(Text key, Iterable<IntWritable>
values, Context context)
            throws IOException, InterruptedException {
        int total = 0;
        for (IntWritable val : values) {
            total += val;
        }
        context.write(key, new IntWritable(total));
    }

}
```

Processing stages

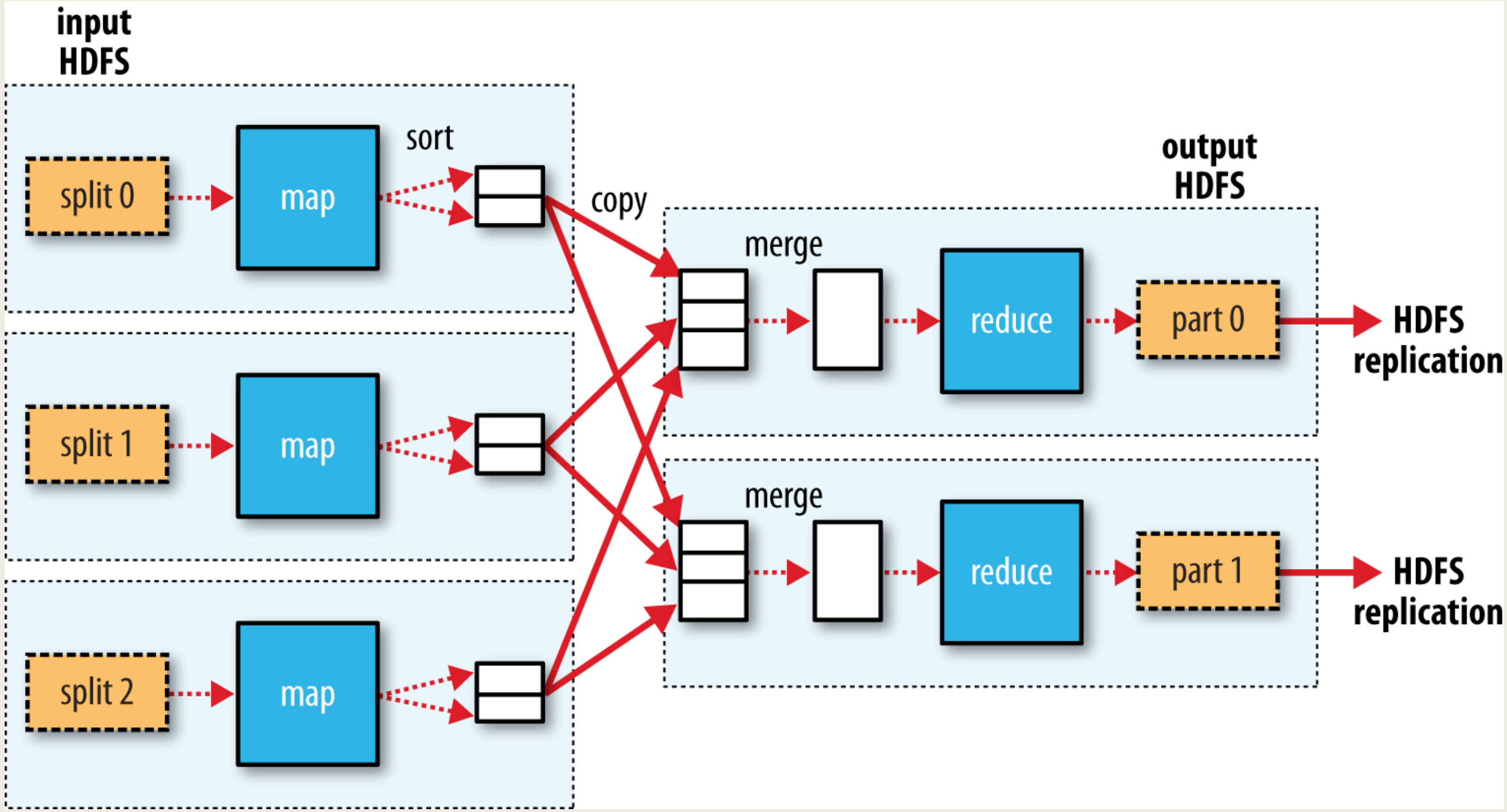


Scaling out



input
HDFS
split 0
map
sort
split 1
map
split 2
map
copy
output
HDFS
merge
reduce
part 0
HDFS
replication

Multiple reduce tasks



Our example - Mappers

Product	Sale
P1	10
P2	15
P1	~~5~~

Category	Sale
PC1	10
PC2	15
PC1	~~5~~

Product	Sale
P2	40
P5	15
P1	55
P2	10

Category	Sale
PC2	40
PC2	15
PC1	55
PC2	10

Product	Sale
P5	30
P3	25
P3	15

Category	Sale
PC2	30
PC1	25
PC1	15

Category	Product
PC1	P1, P3
PC2	P2, P4, P5

Shuffle, sort and partition

Category	Sale
PC1	10
PC2	15

Category	Sale
PC2	40
PC2	15
PC1	55
PC2	10

Category	Sale
PC2	30
PC1	25
PC1	15

- PC1, 10
- PC1, 55
- PC1, 25
- PC1, 15

Partition [reducer] 1 ➔ PC1, 105

- **PC2, 15**
- **PC2, 40**
- **PC2, 15**
- **PC2, 10**
- **PC2, 30**;

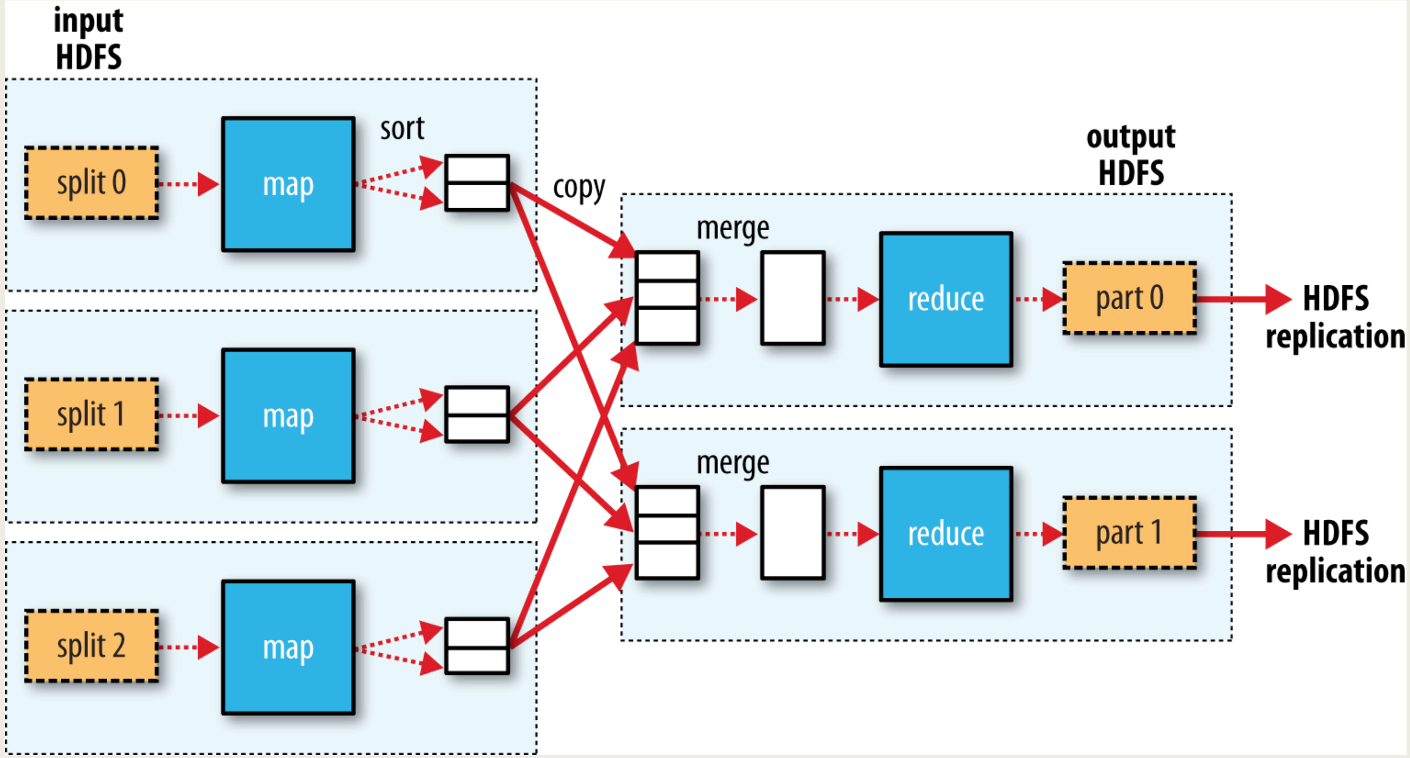
Partition [reducer] 2 ➔ ???

Partitions [reducers] ➔by product category

Can it be optimized further?

<u>Data from Mappers:</u>

- PC1, 10; *PC2, 15;*
- *PC2, 40*; PC2, 15; PC1, 55; PC2, 10
- PC2, 30; PC1, 25; PC1, 15



Combiner

- Runs on the output of mapper
- No guarantee on how many times it will be called by the framework
- Calling the combiner function zero, one, or many times should produce the same output from the reducer.
- Contract for combiner – same as reducer
  - *(k2, [v2]) → [(k3, v3)]*
- Reduces the amount of data shuffled between the mappers and reducers

Combiner example

Category	Sale
PC1	10
PC2	15

Category	Sale
PC2	40
PC2	15
PC1	55
PC2	10

Category	Sale
PC2	30
PC1	25
PC1	15

Category	Sale
PC1	10
PC2	15
PC1	~~5~~

Category	Sale
PC2	**65**
PC1	55

Category	Sale
PC2	30
PC1	**40**

After combining

Framework design

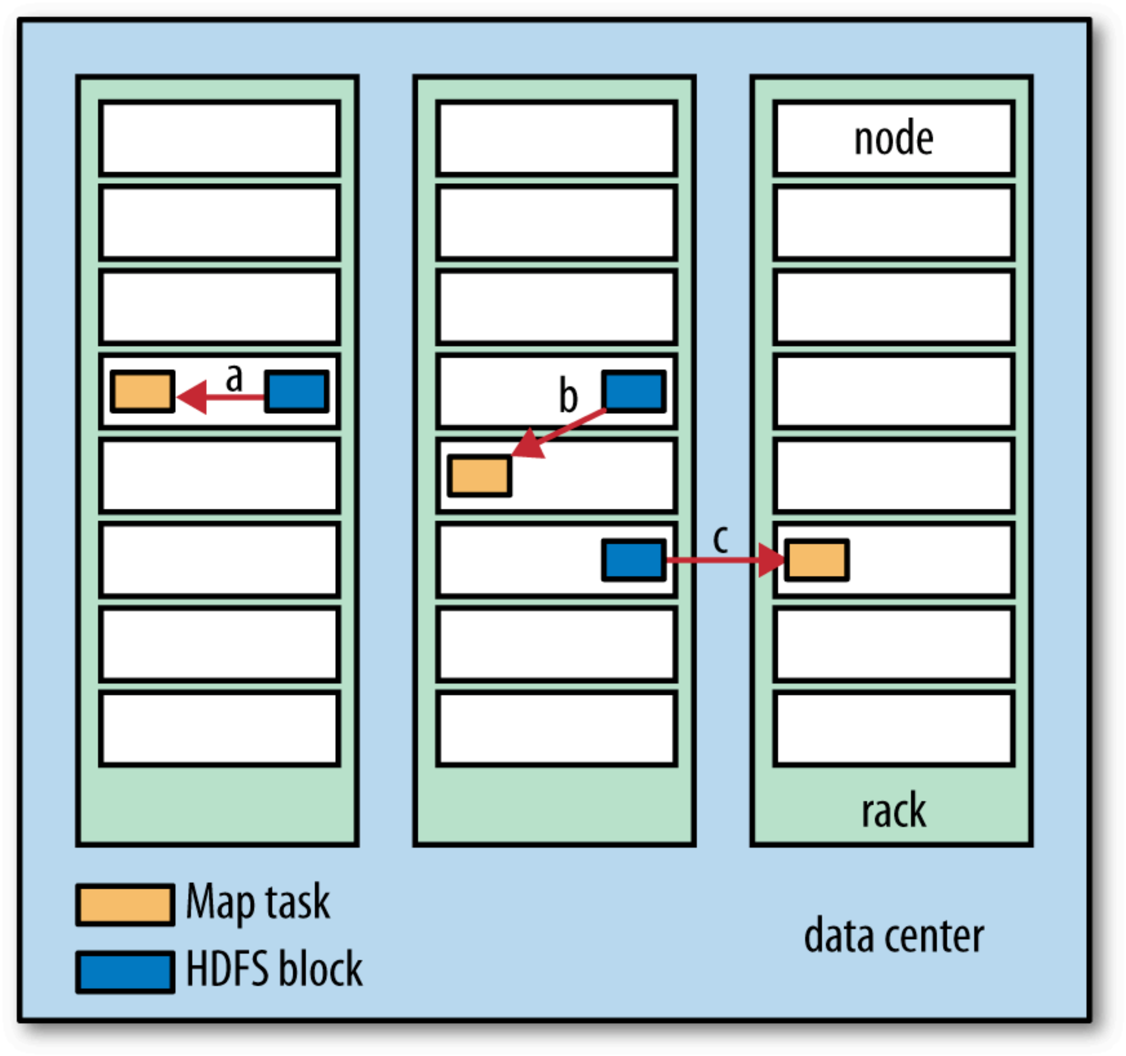
- So where should execution of mapper happen ?
- And how many map tasks ?

“Where to execute?” : Data Locality

- *Move computation close to the data rather than data to computation*”.
- A computation requested by an application is much more efficient if it is executed near the data it operates on when the size of the data is very huge.
- Minimizes network congestion and increases the throughput of the system
- Hadoop will try to execute the mapper on the nodes where the block resides.
  - *In case the nodes [think of replicas] are not available, Hadoop will try to pick a node that is closest to the node that hosts the data block.*
  - *It could pick another node in the same rack, for example.*

Data locality

Data-local (a), rack-local (b), and off-rack (c) map tasks



How many mapper tasks?

Number of mappers set to run are completely dependent on :

1) File Size and

2) Block [split] Size

Internals

- **Mapper** writes the output to the local disk of the machine it is working.
  - *This is the temporary data. Also called intermediate output.*

- As mapper finishes, data (output of the mapper) travels from mapper node to reducer node. Hence, this movement of output from mapper node to reducer node is called **shuffle**.

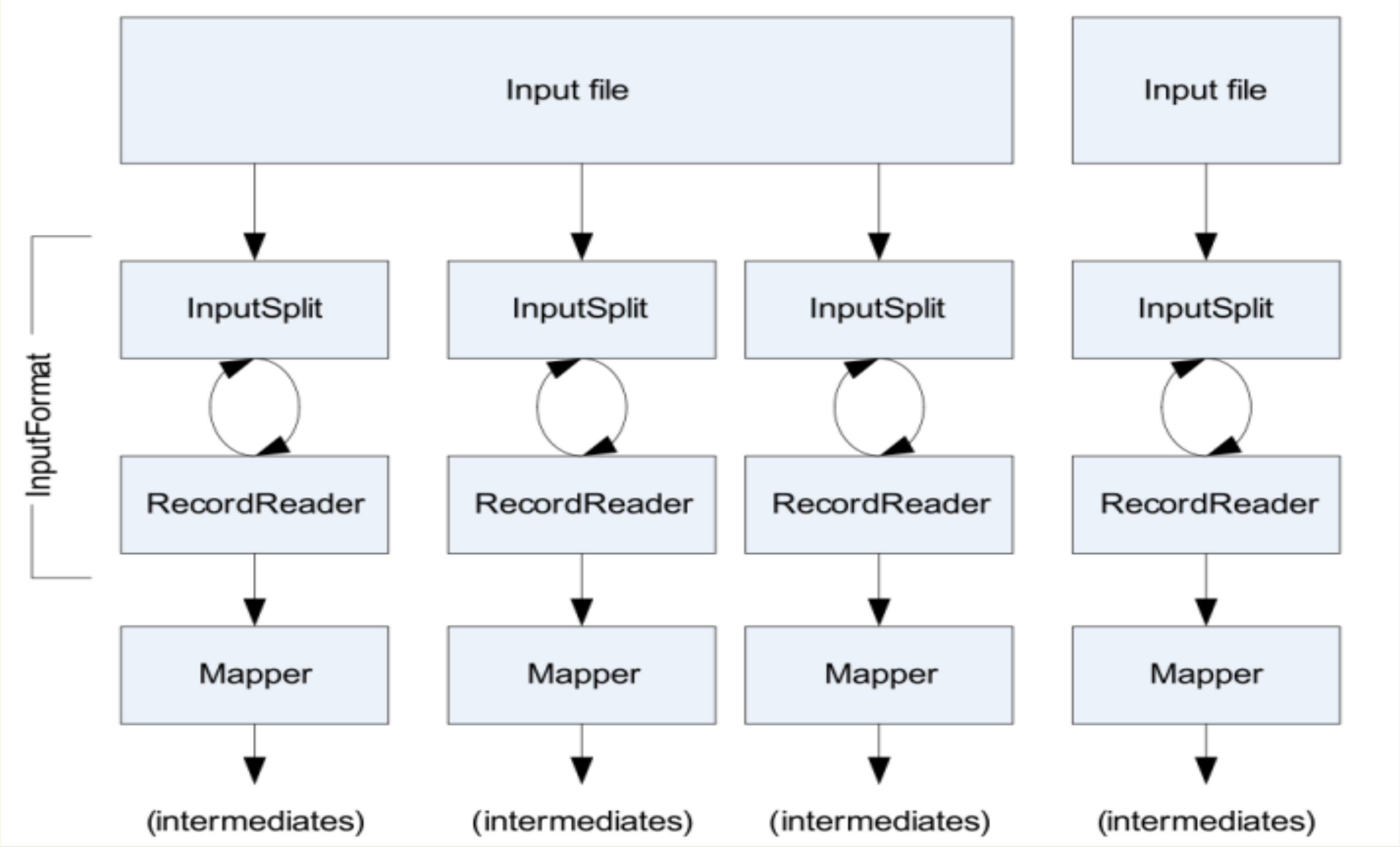
- An output from mapper is partitioned into many partitions;
  - *Each of this partition goes to a reducer based on some conditions*

Map Internals

InputSplits are created by InputFormat. Example formats – FileInputFormat, DBInputFormat

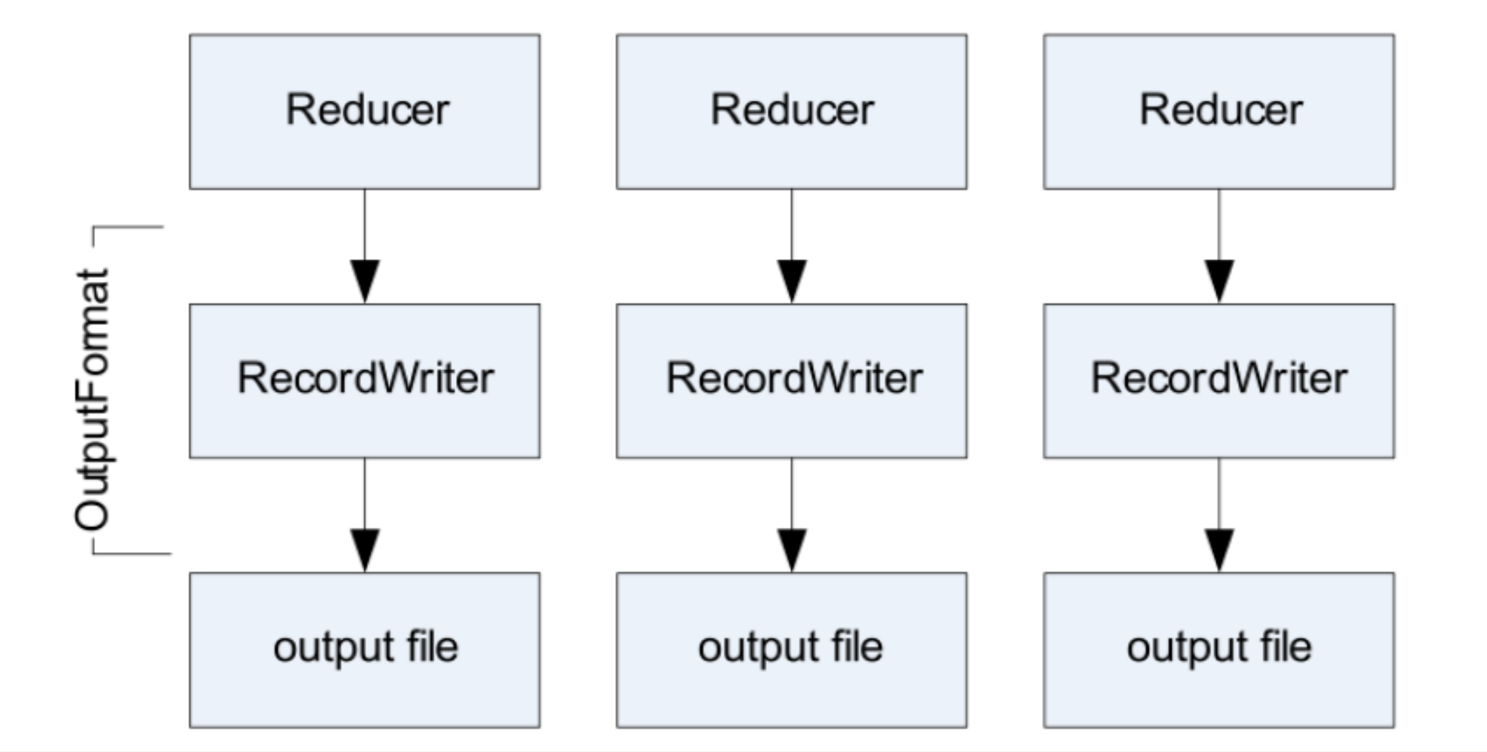
<u>RecordReader</u>'s responsibility is to keep reading/converting data into key-value pairs until the end; which is sent to the mapper.

Number of map tasks will be equal to the number of InputSplits
Mapper on any node should be able to access the split → need a distributed file system (HDFS)



Intermediate output is written to local disks

Same with Output Formats and Record Writers



MR Algorithm design

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)

class Reducer
    method Reduce(term t, counts [c1, c2, ...])
        sum ← 0
        for all count c ∈ counts [c1, c2, ...] do
            sum ← sum + c
        Emit(term t, count sum)
```

Pseudo-code for a basic word count algorithm

Improvement – local within document aggregation

```
class Mapper
    method Map(docid a, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1                ▷ Tally counts for entire documen
        for all term t ∈ H do
            Emit(term t, count H{t})
```

Local across document aggregation

```
class Mapper
    method Initialize
        H ← new AssociativeArray
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1                    ▷ Tally counts across documents
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

No longer pure functional programming – state maintained across function calls!

Do we still need combiners?

- Limitations of in-mapper combining
  - *State needs to be maintained*
  - *Scalability – size of the state can grow without bounds*
- Keep bounded state
  - *Write intermediate results*
  - *Use combiners*

Summary

- MR – powerful abstraction for parallel computation
- Framework handles the complexity of distribution, data transfer, coordination, failure recovery

Reading list

- Designing Distributed Systems, Brendan Burns
  - *Chapters 11 and 12, except Hands on sections*
- Distributed and cloud computing, Kai Hwang, Geoffrey C Fox, Jack J Dongarra
  - *Sections 6.2.2 except 6.2.2.7*

- Optional reading
  - *Data-Intensive Text Processing with MapReduce*
    - Sections 2.1 to 2.4