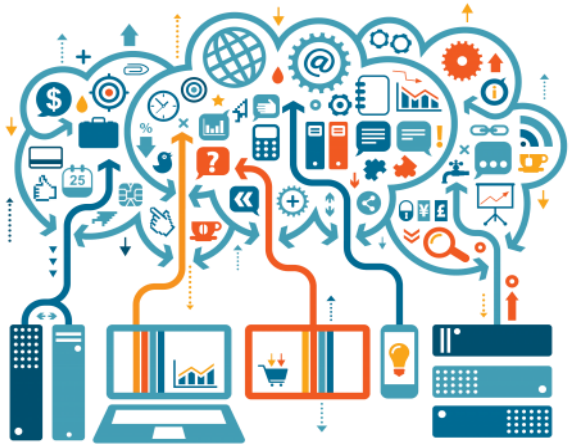
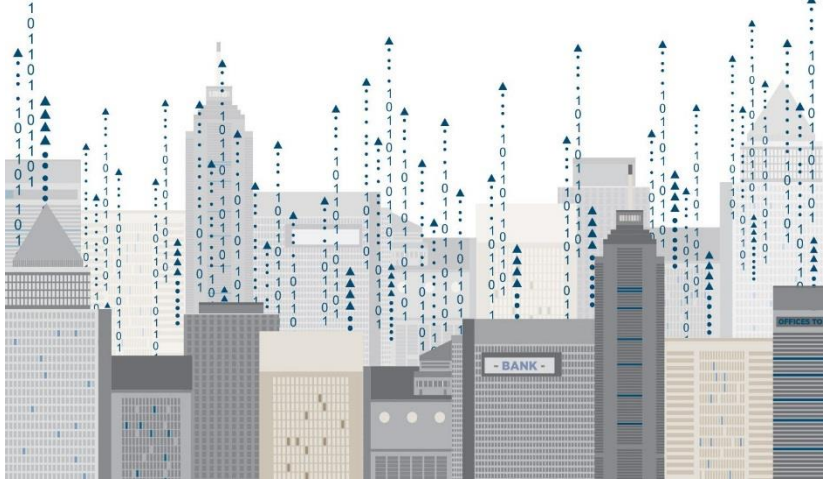




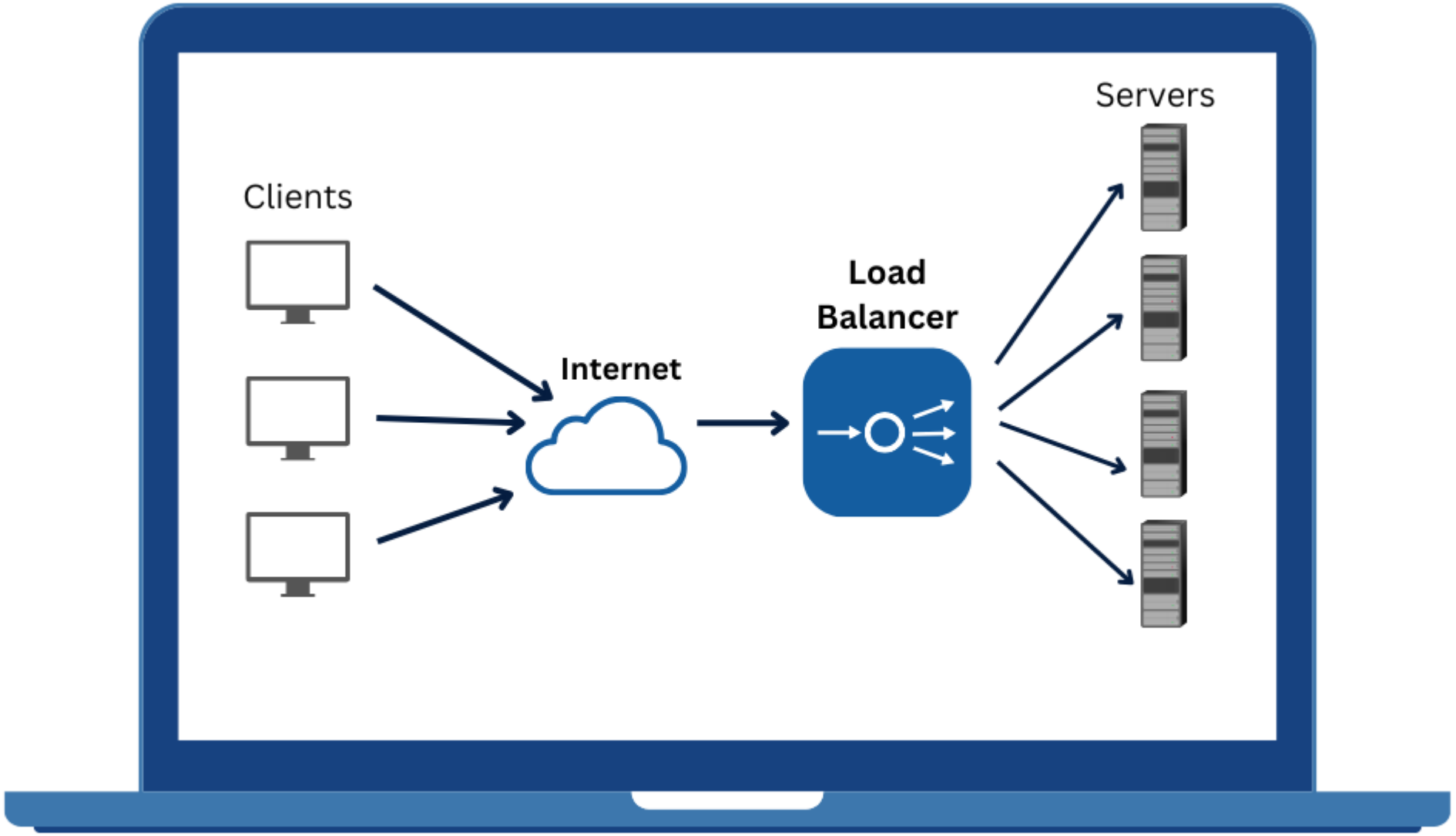
Big Data Analytics

Fall 2025

Lecture 7

MapReduce

- MR: input data, MapReduce program, and config information.
- Hadoop runs MR job the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks.
- The tasks are scheduled using YARN and run on nodes in the cluster.
- If a task fails, it will be automatically rescheduled to run on a different node.
- Hadoop divides the input to a MR job into fixed-size pieces called input splits, or just splits.
- Hadoop creates one map task for each split (map is run for each record in the split)


Servers
Clients
Load
Balancer
Internet

MapReduce

- When you put a file into HDFS, it’s automatically split into blocks (typically 128 MB or 256 MB each).
- Each block is replicated (by default 3 times) across different DataNodes for fault tolerance.

file.txt → block0, block1, block2
block0 → node1,node2,node3
block1 → node2,node3,node4
block2 → node3,node4,node5

MapReduce

- When you submit a MR job, the InputFormat (TextInputFormat) divides the input files into splits

- By default, one split ≈ one HDFS block (split is logical)

- But not *always exact* — some InputFormats may combine smaller blocks (CombineFileInputFormat)

- The ApplicationMaster (JobTracker) creates one Map task per split.

- If you have 20 input splits → then 20 map tasks

MapReduce

- The ApplicationMaster asks YARN's ResourceManager for containers to run map tasks.

- The scheduler tries to place each Map task on the node where its data block resides — called data locality.
  - Best case: *node-local* (same node)
  - Next best: *rack-local* (same rack)
  - Otherwise: *off-rack* (over network)

- Equal assignment is not guaranteed: (data locality not guaranteed ☹)
  - If one node happens to store more data blocks, it will likely get more Map tasks (to maintain data locality).
  - Hadoop's goal is minimize data movement, not force perfect task balance at start.
  - Over time, as tasks finish, idle nodes may get remaining unstarted tasks — this *eventually balances load dynamically*.

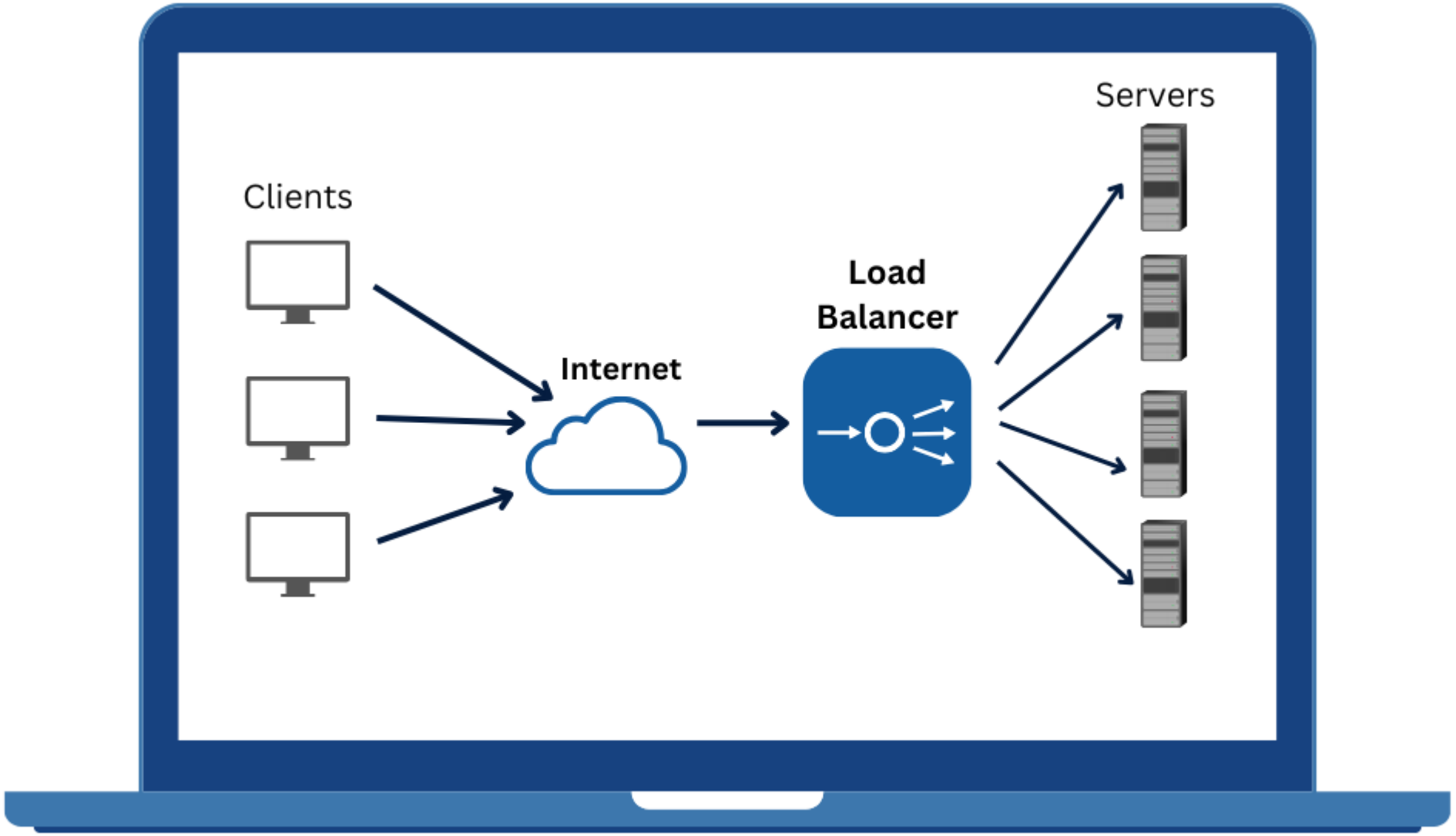
The Reality about Data Movement

Data *can* move between DataNodes during MR — but Hadoop tries very hard to *avoid* it.

The actual HDFS block doesn't move, but the data within that block may be *read remotely* if the mapper isn't running on the same DataNode.

If the cluster is busy or there are no free slots on the node that holds the data

- The scheduler may launch that map task on another node (rack-local or off-rack).
- The input split (the HDFS block) is transferred over the network from the DataNode where it's stored to the node running the map task (oops but can't be avoided)


Servers
Clients
Load
Balancer
Internet

Idea behind MR Split

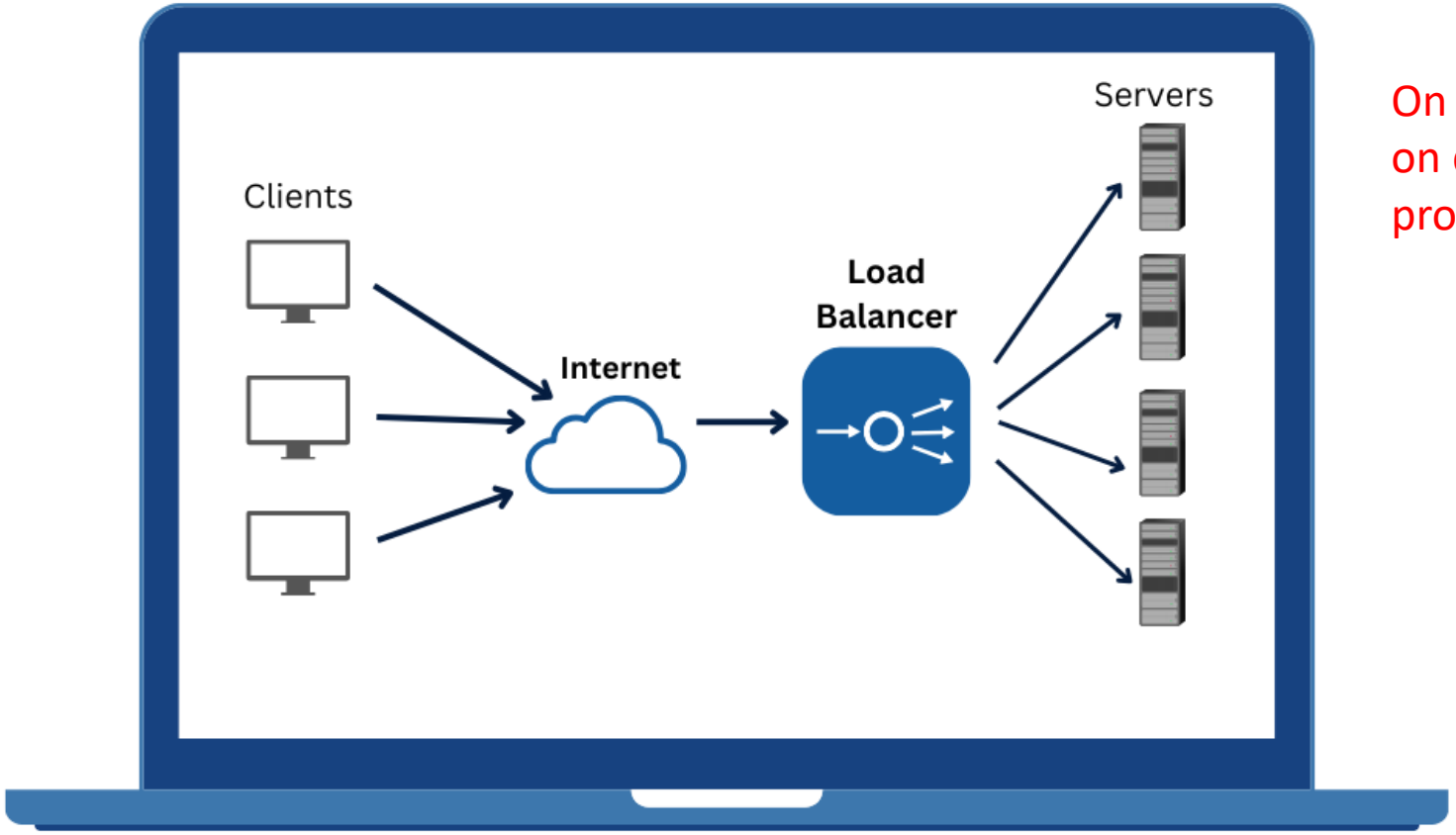
- When we process many input *splits* in parallel, the MR finishes when the slowest parallel task (“straggler”) finishes.
- If work is evenly spread across tasks, every worker has roughly the same amount of work and the job finishes close to the theoretical best time.
- If work is uneven, a few heavy tasks dominate total runtime - you lose most of the benefit of parallelism.

Maths behind Equal MR Split

- Let total work = $W = \sum_{i=1}^{n} w_i$ where w_i is work in split i.
- With p identical processors, **ideal** finish time (perfect balance) ≈ $T_{\text{ideal}} = W/p$.
- Actual finish time (makespan) = $\max_{j=1..p} W_j$ where W_j is work assigned to processor j.
- The closer $\max_j W_j$ is to W/p, the better the load balancing and the higher the parallel speedup.

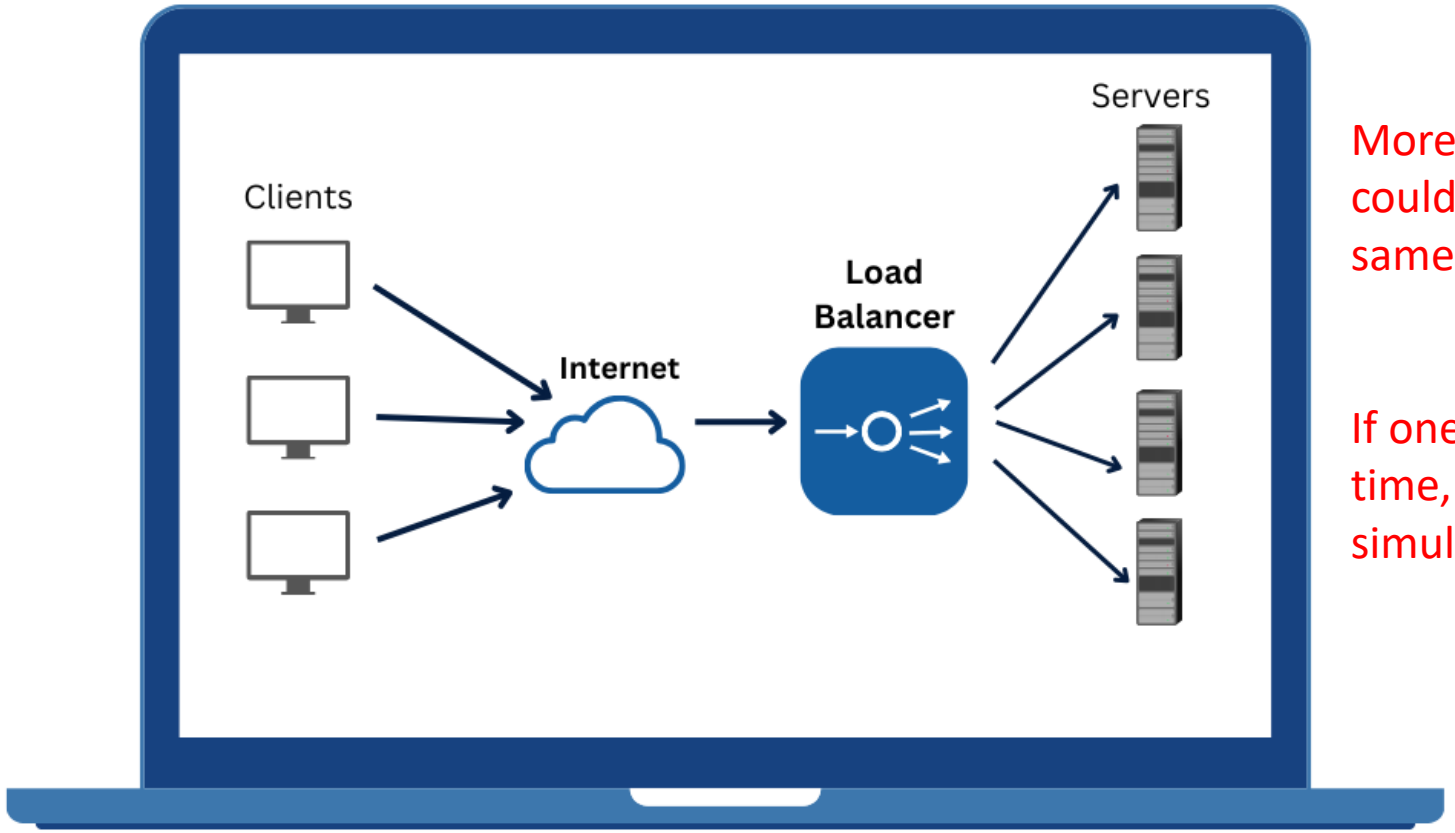
So parallelism improves performance **proportionally** to how evenly you can make each W_j — poor balance yields limited speedup.

Multiple map tasks read different HDFS blocks and process them simultaneously - increase aggregate disk throughput and CPU utilization.



On each node, all map tasks on different HDFS blocks proceed simultaneously

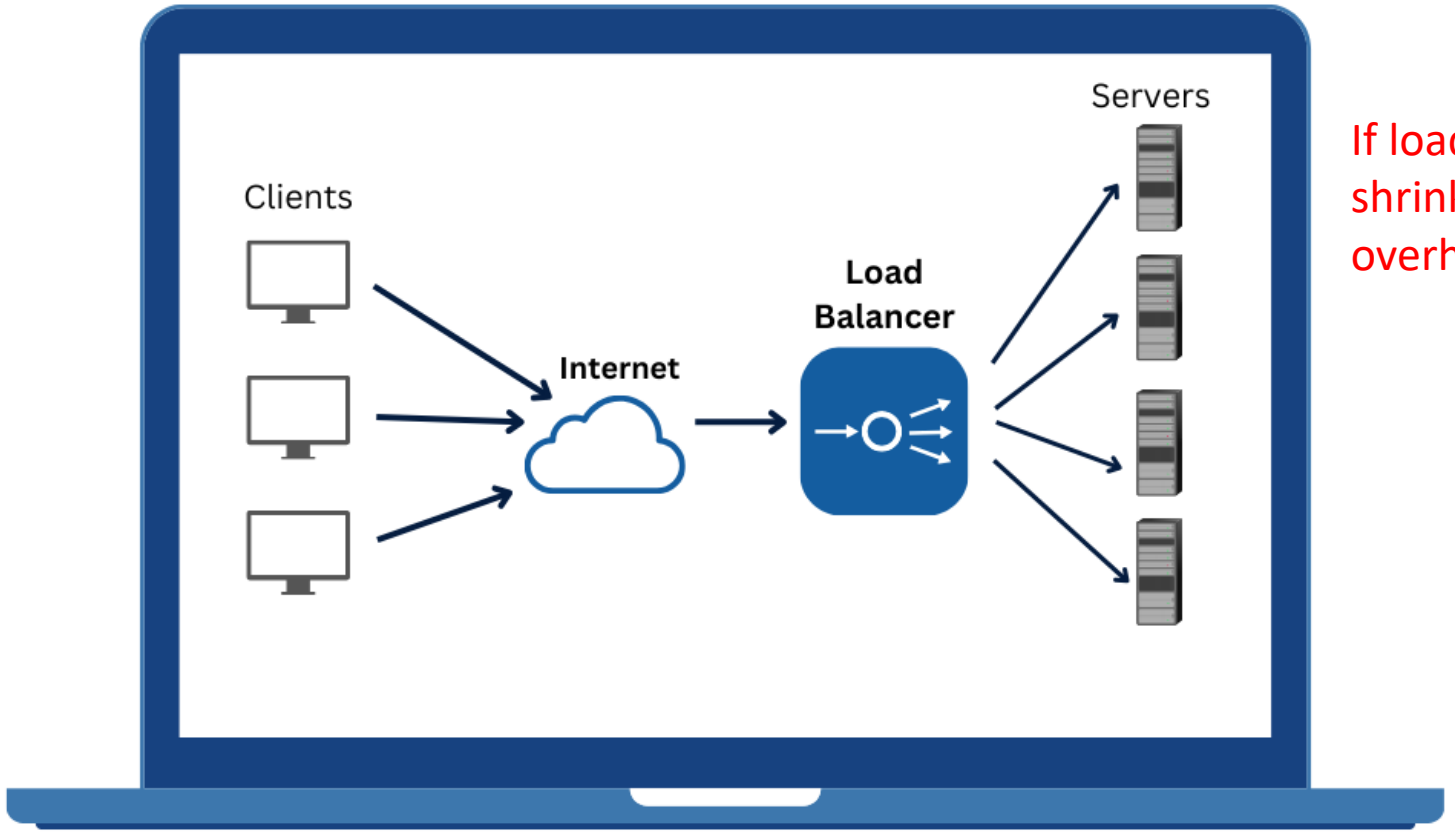
While one task waits for I/O, others can proceed, reducing idle time.



More than one MR task could be running on the same cluster

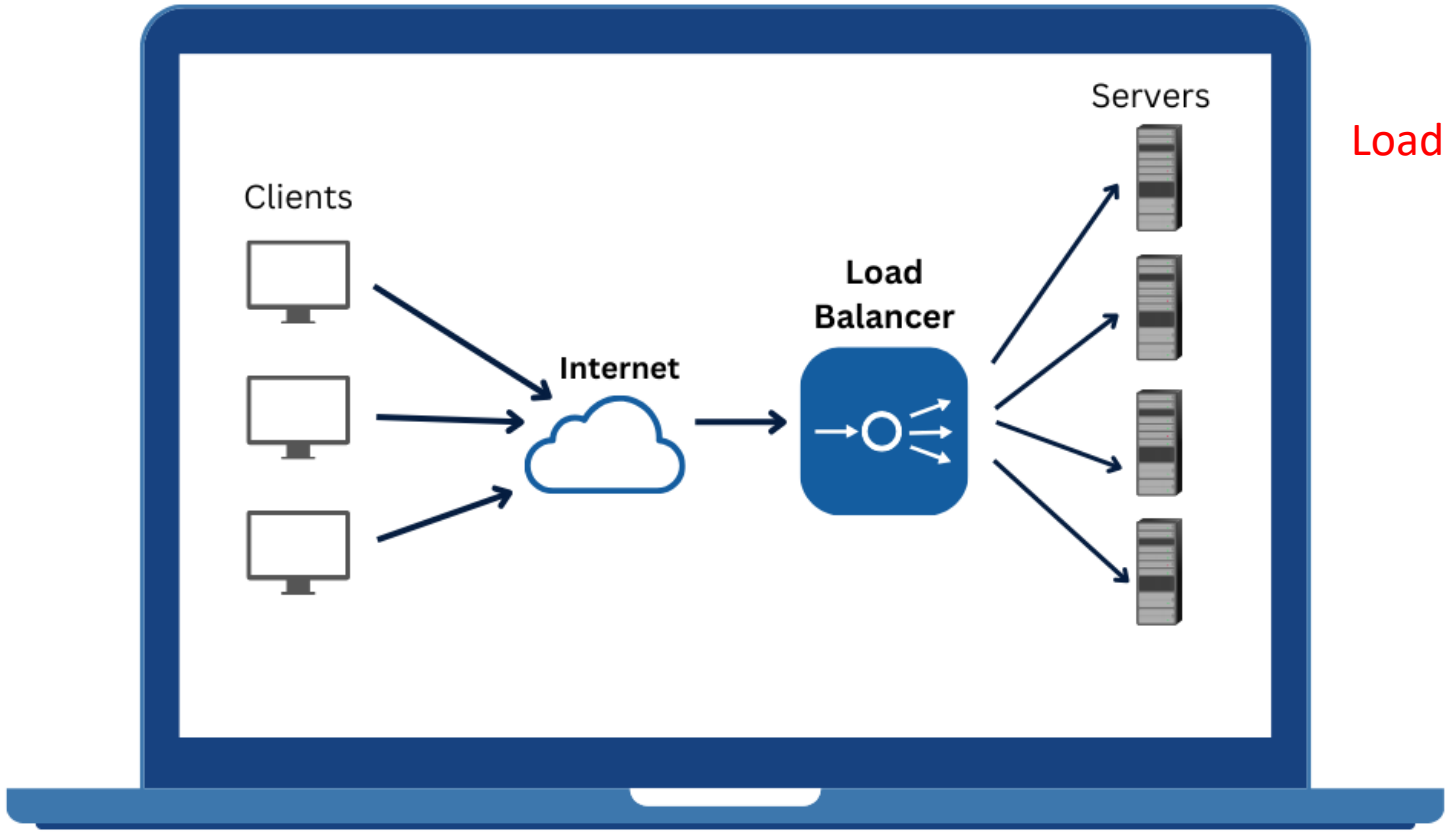
If one task is consuming time, other can proceed simultaneously.

Subdivide work: workload that a single data node must finish is smaller



If load balanced, overall time shrinks roughly by p (minus overhead).

Shorter tasks mean faster re-execution on failure; scheduler can reassign tasks to underutilized nodes.



Load balance as needed

MapReduce

- Having many splits means the time taken to process each split is small compared to the time to process the whole input.
- If we are processing the splits in parallel, the processing is better load balanced when the splits are small
  - a faster machine will be able to process proportionally more splits over the course of the job than a slower machine.
- Failed processes or concurrent jobs make load balancing desirable
- Quality of the load balancing increases as the splits become more fine grained

What IS load balancing?

- Load balancing in Hadoop MapReduce means distributing — and if necessary, shifting — *map and reduce tasks* across nodes to ensure all nodes are equally busy.

Load balancing Mappers

- During the map phase:
  - The ApplicationMaster looks at where each data block is stored (from the NameNode).
  - It assigns map tasks to the nodes holding that data (data-local scheduling) — no data movement.
  - If some nodes finish early or are idle, the scheduler can:
    - Assign new map tasks from unfinished splits (shift work)
    - Or launch speculative map tasks for slow ones

Load balancing Reducers

- During the reduce phase:
  - Reduce tasks don’t depend on local data; they pull intermediate data from all mappers.
  - YARN schedules reducers across nodes according to available resources.
  - If a node is overloaded or slow, reducers can be:
    - Started on less-busy nodes,
    - Re-attempted elsewhere if they fail or lag.

Summing it up

Scenario	What moves	Why
Node finishes early	New map/reduce tasks are assigned to it	Keep it busy
Node is slow or fails	Task re-run on another node	Maintain progress
Data not available locally	Map task scheduled on remote node	Continue job without waiting

Let’s Redo it again – from a different angle

MapReduce

- If splits are too small: overhead begins to dominate the total job execution time.
- A good split size tends to be the size of an HDFS block, which is 128 MB by default
- Run the map on a node where the input data resides in HDFS (locality)
- Sometimes: all nodes having replicas for a split are running other map tasks - so scheduler will look for a free map slot on a node in the same rack as one of the blocks.
- Very occasionally

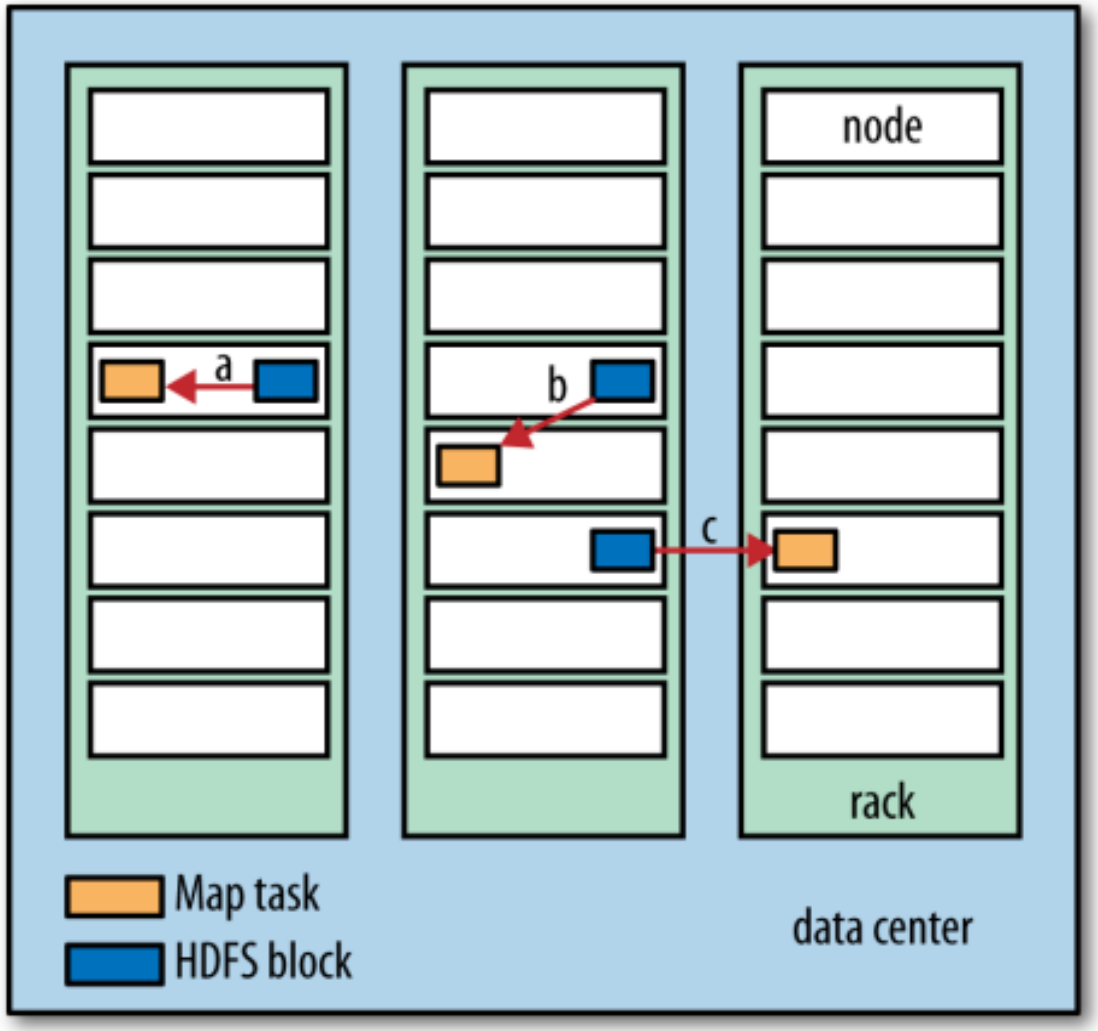


Figure 2-2. Data-local (a), rack-local (b), and off-rack (c) map tasks

MapReduce

- Logical: optimal split size should be the same as the block size
- If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some of the split would have to be transferred across the network
- Map tasks write output to local disk, not to HDFS: Map output is intermediate: processed by reduce to produce the final output, and once job is complete, the map output can be thrown away.
- So, storing it in HDFS with replication would be overkill.
- If the node running the map fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output.

Reduce Tasks

- Reduce tasks don’t have the advantage of data locality; the input to a single reduce task is normally the output from all mappers.
- In the present example, we have a single reduce fed by all map tasks.
- Thus, sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function.
- The output of reduce is normally stored in HDFS for reliability.
- For each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes for reliability.
- Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes.

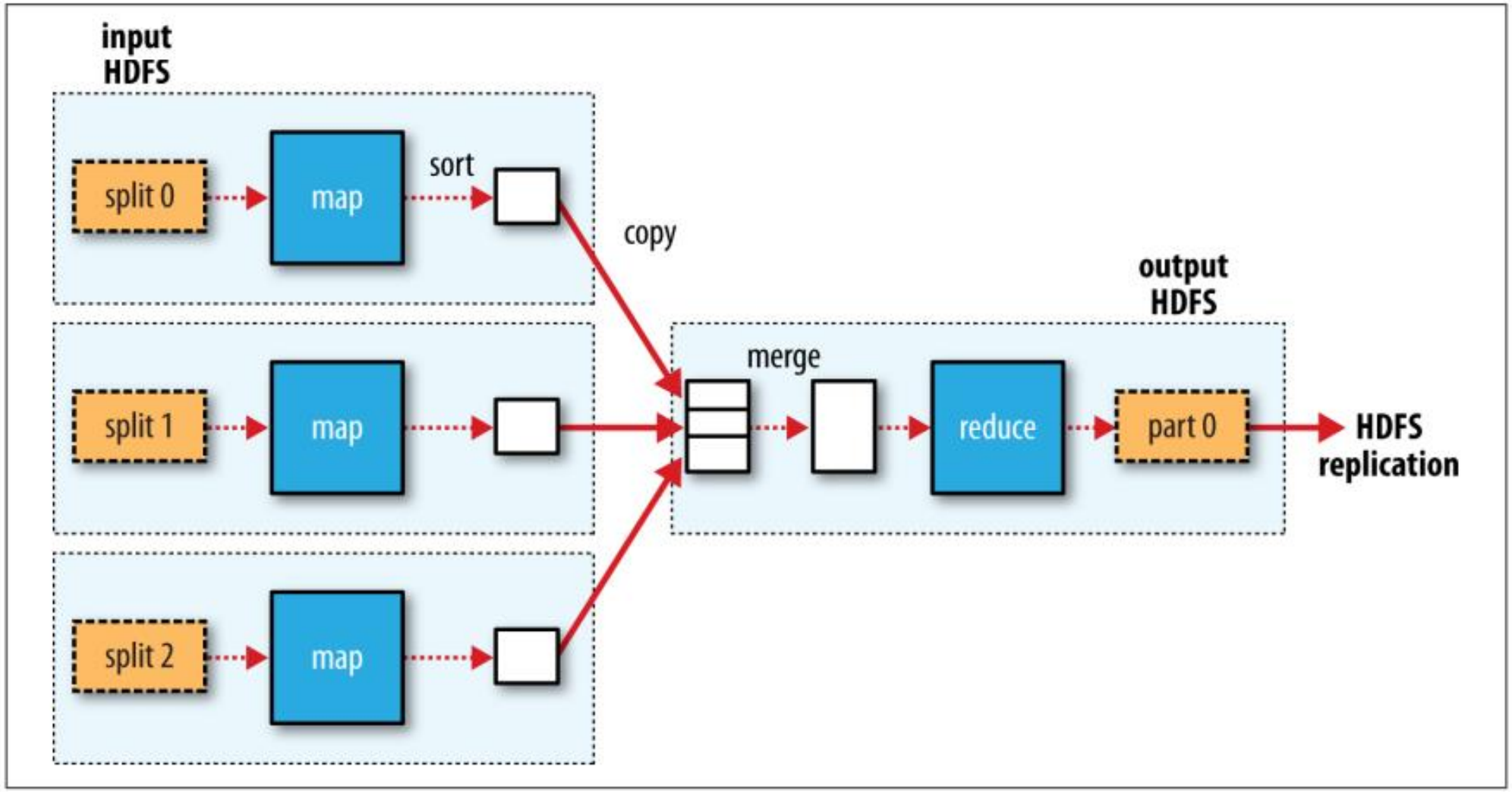


Figure 2-3. MapReduce data flow with a single reduce task

Reduce Tasks

- The number of reduce tasks is not governed by the size of the input, but instead is specified independently
- When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task.
- There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition.
- The partitioning can be controlled by a user-defined function, but default partitioner—buckets keys wit hash function—works very well.
- Data flow between map and reduce tasks is colloquially known as “the shuffle,” as each reduce task is fed by many map tasks – has a great impact on efficiency

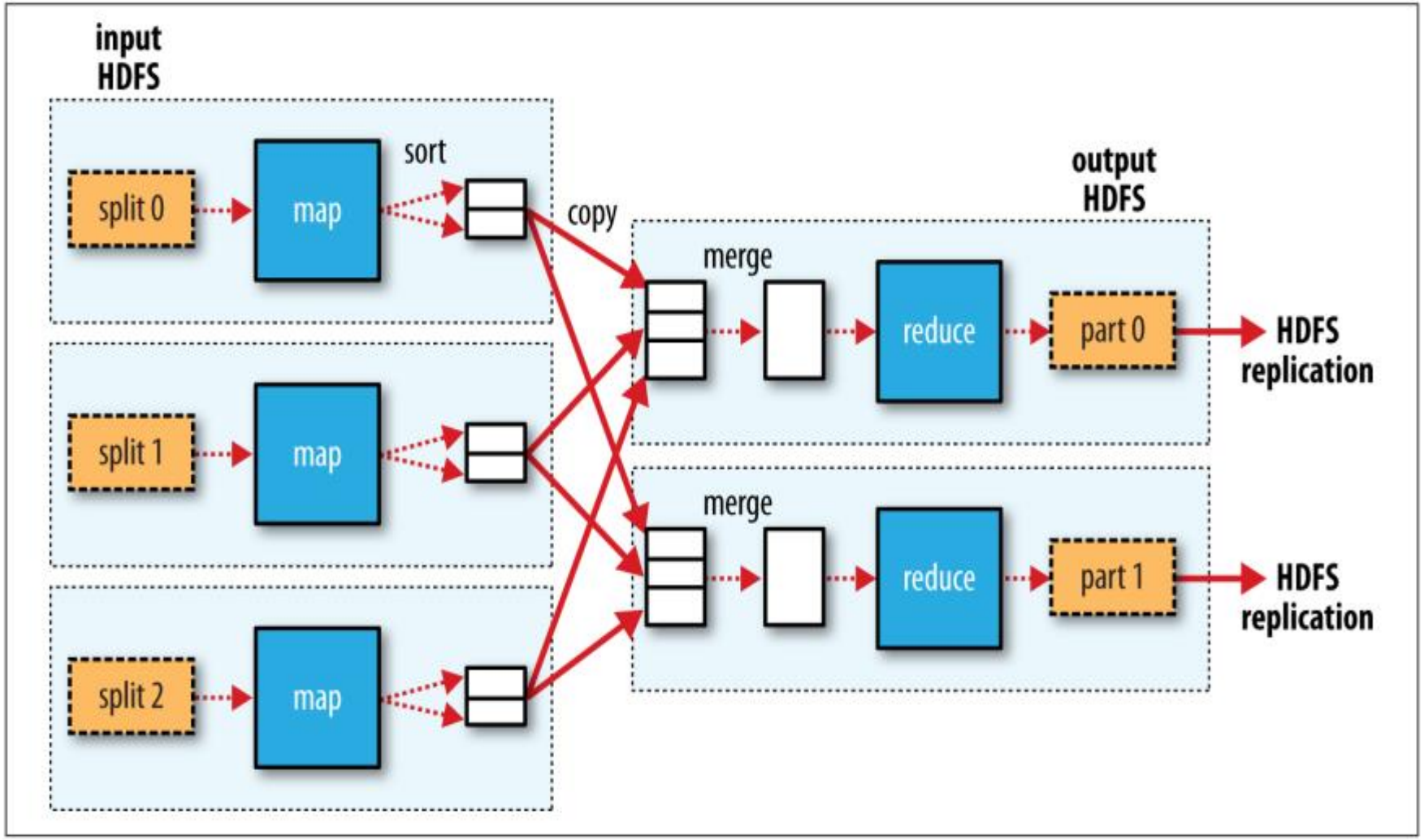


Figure 2-4. MapReduce data flow with multiple reduce tasks

Combiner Functions

- Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks.
- Hadoop allows the user to specify a combiner function to be run on the map output, and the combiner function's output forms the input to the reduce function.
- Because the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all.
- In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer.

Combiner Functions

- Limited by bandwidth: pays to minimize the data transferred between map and reduce tasks.
- Hadoop allows the user to specify a combiner function to be run on the map output, and the combiner function's output forms the input to the reduce function.
- Because the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all.
- In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer.

This is best illustrated with an example. Suppose that for the maximum temperature example, readings for the year 1950 were processed by two maps (because they were in different splits). Imagine the first map produced the output:

```
(1950, 0)
(1950, 20)
(1950, 10)
```

and the second produced:

```
(1950, 25)
(1950, 15)
```

The reduce function would be called with a list of all the values:

```
(1950, [0, 20, 10, 25, 15])
```

with output:

```
(1950, 25)
```

since 25 is the maximum value in the list. We could use a combiner function that, just like the reduce function, finds the maximum temperature for each map output. The reduce function would then be called with:

```
(1950, [20, 25])
```

and would produce the same output as before. More succinctly, we may express the function calls on the temperature values in this case as follows:

```
max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25
```

Example 2-6. Application to find the maximum temperature, using a combiner function for efficiency

```
public class MaxTemperatureWithCombiner {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCombiner <input path> " +
          "<output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperatureWithCombiner.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(IntWritable.class);

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

Example 2-9. Map function for maximum temperature in Python

```
#!/usr/bin/env python

import re
import sys

for line in sys.stdin:
  val = line.strip()
  (year, temp, q) = (val[15:19], val[87:92], val[92:93])
  if (temp != "+9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

Example 2-10. Reduce function for maximum temperature in Python

```
#!/usr/bin/env python

import sys

(last_key, max_val) = (None, -sys.maxint)
for line in sys.stdin:
  (key, val) = line.strip().split("\t")
  if last_key and last_key != key:
    print "%s\t%s" % (last_key, max_val)
    (last_key, max_val) = (key, int(val))
  else:
    (last_key, max_val) = (key, max(max_val, int(val)))

if last_key:
  print "%s\t%s" % (last_key, max_val)
```

We can test the programs and run the job in the same way we did in Ruby. For example, to run a test:

```
% cat input/ncdc/sample.txt | \
  ch02-mr-intro/src/main/python/max_temperature_map.py | \
  sort | ch02-mr-intro/src/main/python/max_temperature_reduce.py
1949    111
1950    22
```