MapReduce

ENGR689 (Sprint)

TEXAS A&M UNIVERSITY

Parallelism - Amdahl's Law

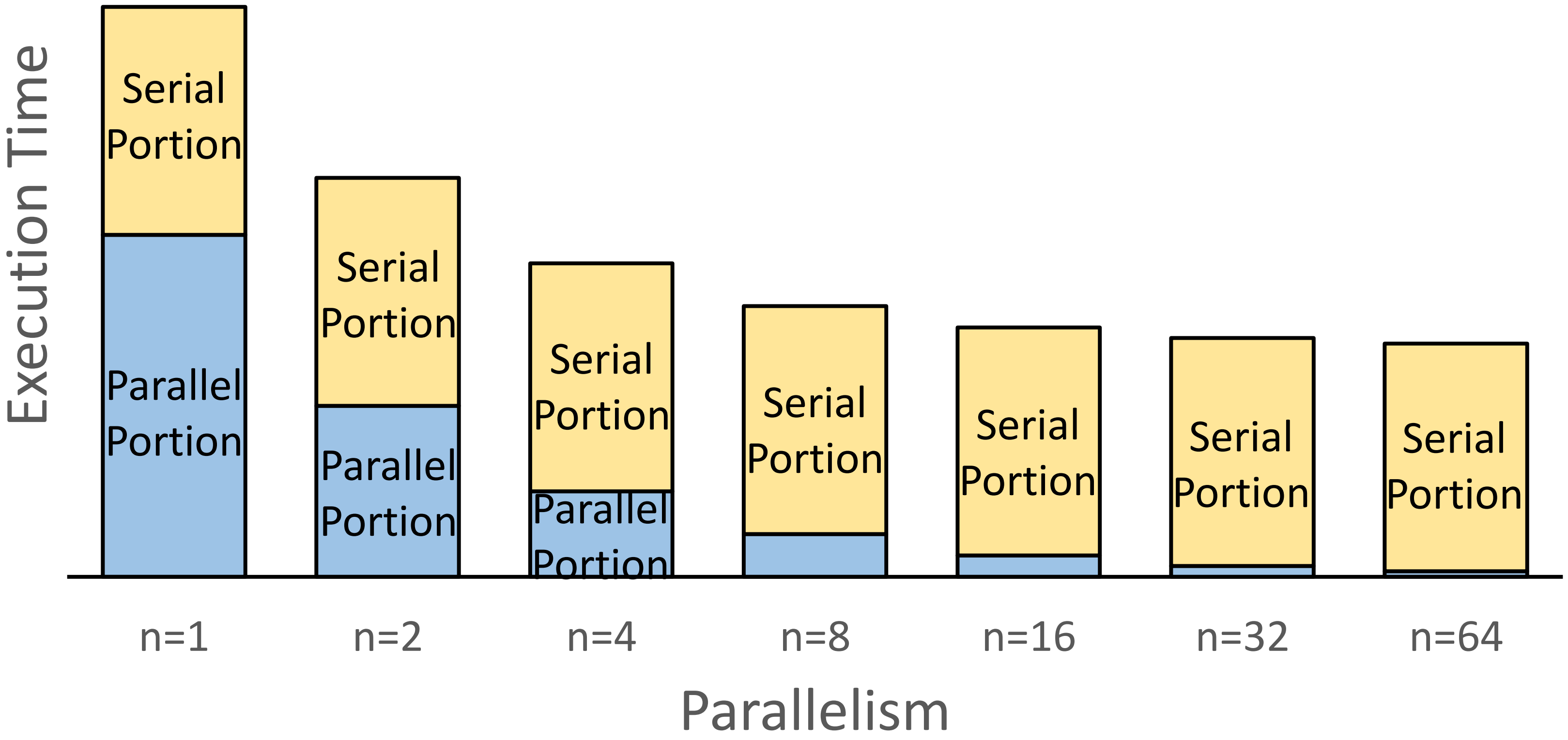
$$Speedup_n = \frac{1}{\frac{P_{parallel}}{n} + P_{serial}} = \frac{1}{\frac{P_{parallel}}{n} + 1 - P_{parallel}}$$

This part can be Parallelized.

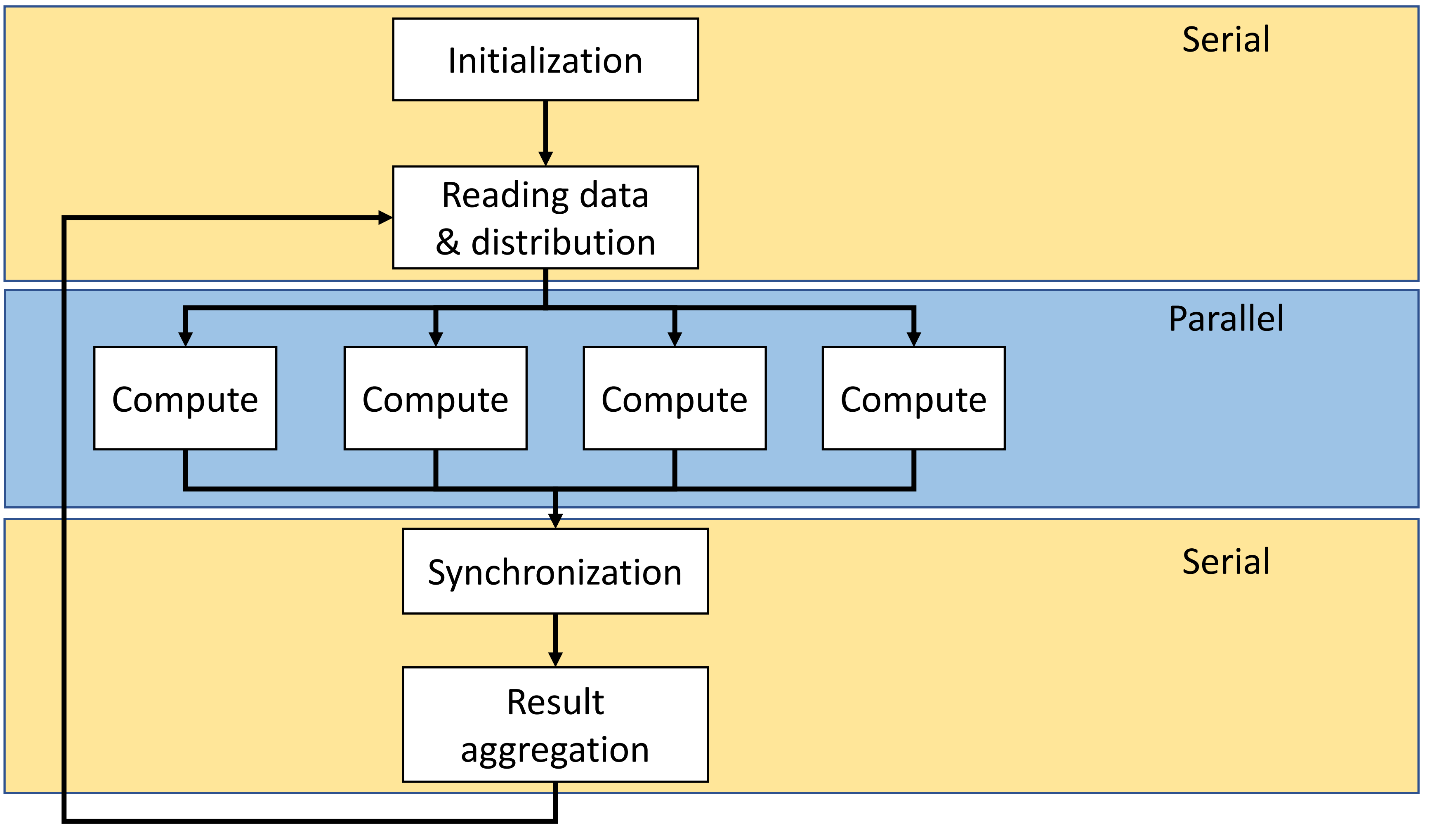
This part cannot be Parallelized.

- Not all portions of a program is parallelizable
- You lose the potential speedup by having lots of serial portions

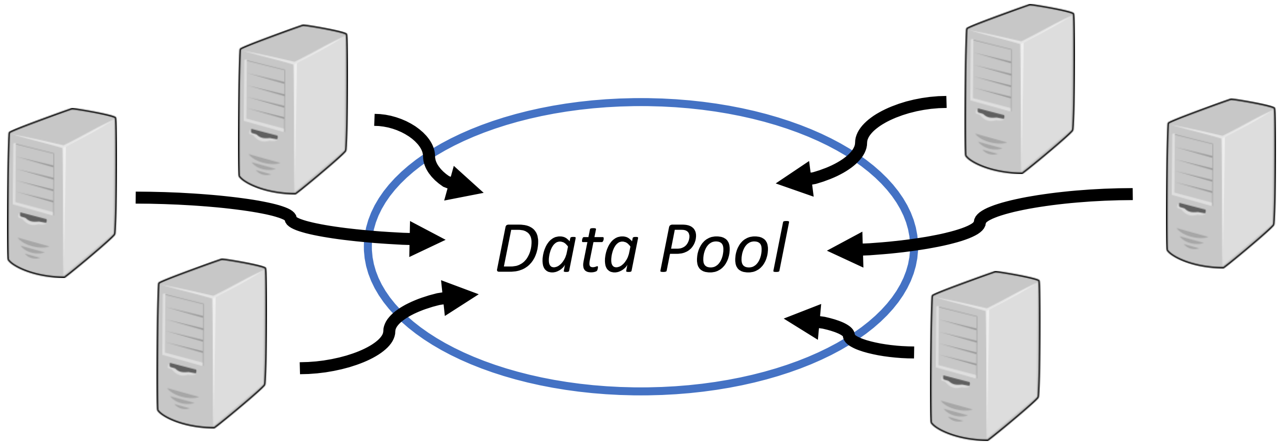
Result of Amdahl's Law



Parallelism in Programs

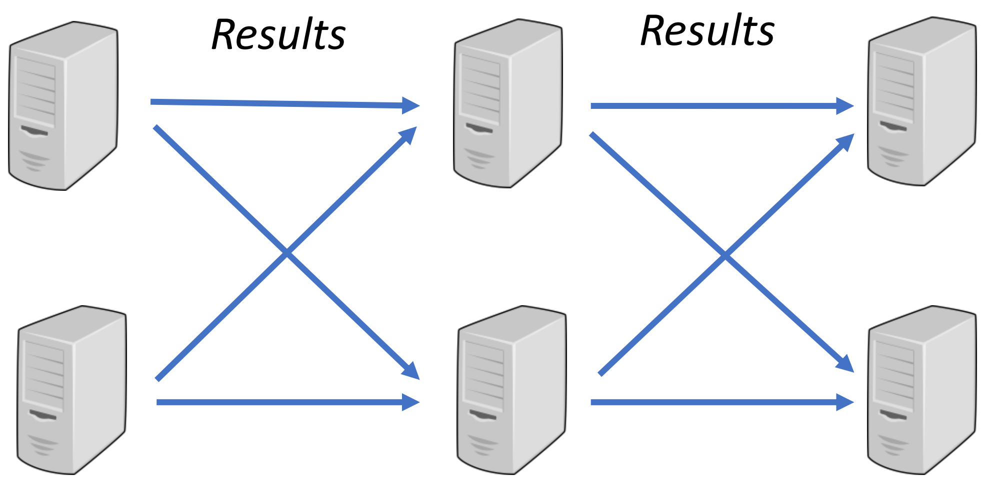


Challenges of Parallelism (1/3)



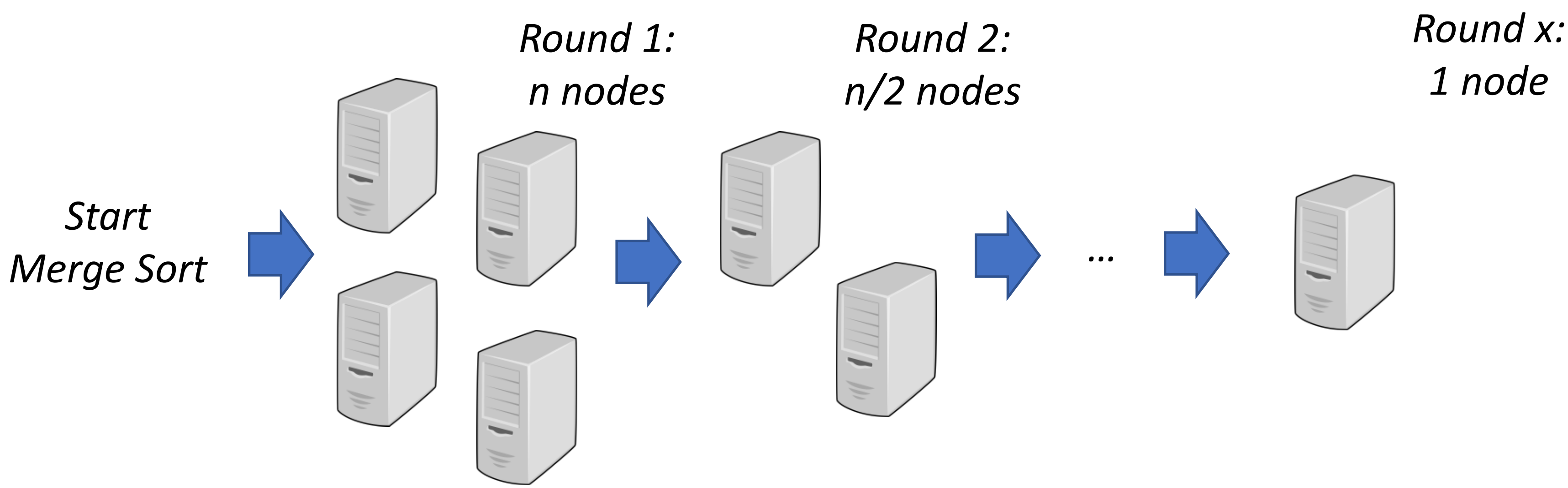
- **Challenge 1: Data transfer can be expensive**
  - Do you store all the data at one location?
  - If you duplicate the data on all nodes, how much cost?
  - If you split the data among the nodes, how?

Challenges of Parallelism (2/3)



- **Challenge 2: Synchronization & Dependency**
  - One node may wait for the result of another node
  - Wait for all nodes to complete?
  - Waiting increases the serial portions

Challenges of Parallelism (3/3)



- **Challenge 3: How to scale the computation**
  - Possible to achieve the same parallelism at every step?
  - How to scale down?
  - What if you certainly get a lot of new nodes? Increase parallelism midst the computation?

MapReduce

- Invented by Google in 2004
  - By Jeff Dean and Sanjay Ghemawat
- A programming model with APIs to help designing **naturally parallelizable** programs

MapReduce Operations

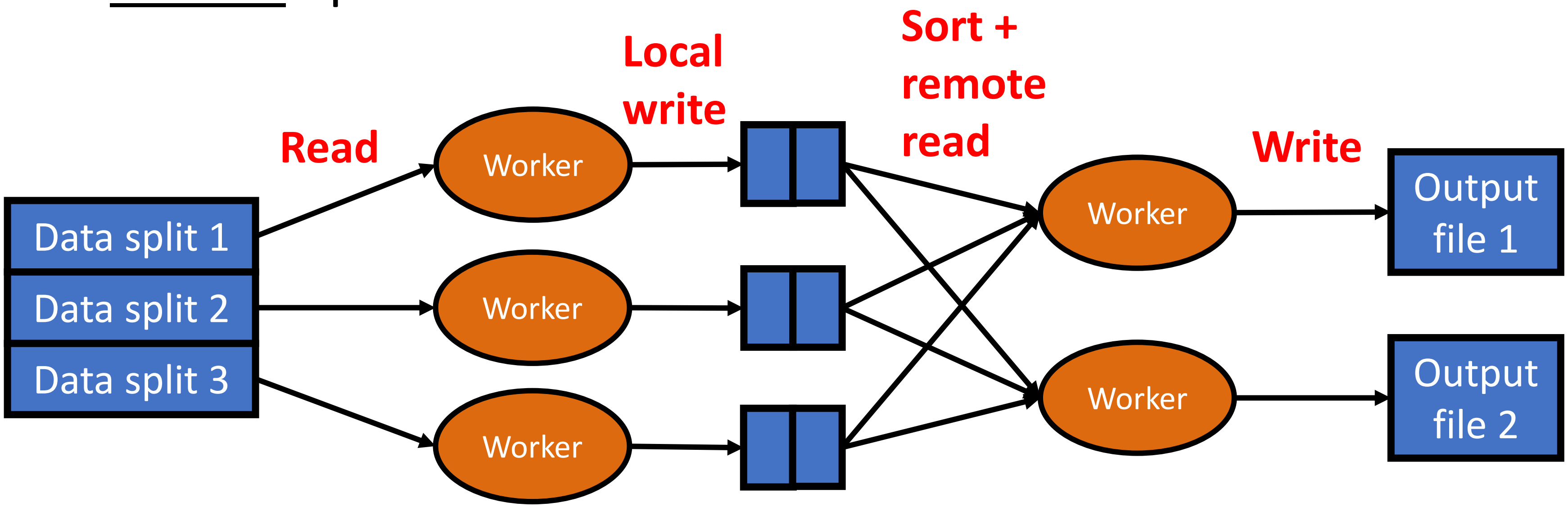
MapReduce separates two types of operations

- **“Map” operations:**
  Apply algorithm to one value or a small portion of the data

- **“Reduce” operations:**
  Aggregating the results from multiple parallel computations

Program with MapReduce

- Many data problems can be deconstructed as **<u>Map</u>** and **<u>Reduce</u>** operations



Map:
extract useful information from each data record

Reduce:
aggregating or filtering multiple records

MapReduce APIs

For each key-value pair,
generate a list of key-value outputs

- Map: `(k1, v1) → list(k2, v2)`

Collect all map outputs
with the same key

- Reduce: `(k2, list(v2)) → list(v2)`

Aggregated reduce outputs

Basic Example: Word Count

- **Problem: counting the occurrence of each word**
  - Map: **`(k1, v1) → list(k2, v2)`**

(A.txt, “Hello This Is Hello Michael”)

(B.txt, “Michael Hello This”)

  - Reduce: **`(k2, list(v2)) → list(v2)`**

Basic Example: Word Count

- **Problem: counting the occurrence of each word**
  - Map: **`(k1, v1) → list(k2, v2)`**

    For each word in each value, emit (word, 1)

    (A.txt, “Hello This Is Hello Michael”) → (Hello, 1), (This, 1), (Is, 1), (Hello, 1), (Michael, 1)

    (B.txt, “Michael Hello This”) → (Michael, 1), (Hello, 1), (This, 1)

  - Reduce: **`(k2, list(v2)) → list(v2)`**

    For each key, emit (key, sum of all values)

    (Hello, [1, 1, 1]) → Hello: 3

    (This, [1, 1]) → This: 2

    (Is, [1]) → Is: 1

    (Michael, [1, 1]) → Michael: 2

Basic Example: Word Count

It will be seen that this mere painstaking burrower and grub-worm of a poor devil of a Sub-Sub appears to have gone through the long Vaticans and streetstalls of the earth, picking up whatever random allusions to whales he could anyways find in any book whatsoever, sacred or profane. Therefore you must not, in every case at least, take the higgledy-piggledy whale statements however authentic, in these extracts gospel cetology. Far from it. As ancient authors generally, as well as the poet here appearing, these extracts are solely valuable or entertaining, as affording a glancing bird's eye view of what has been promiscuously said, thought, fancied, and sung of Leviathan, by many nations and generations, including our own.

map

Unsorted, Unaggregated

it 1
will 1
be 1
seen 1
that 1
this 1
mere 1
painstaking 1
burrower 1
and 1
grub-worm 1
of 1
a 1
poor 1
devil 1
of 1

reduce

Sorted, Aggregated

a 1
a 1
2

aback 1
aback 1
2

abaft 1
abaft 1
2

abandon 1
abandon 1
abandon 1
3

abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
7

Partitioner

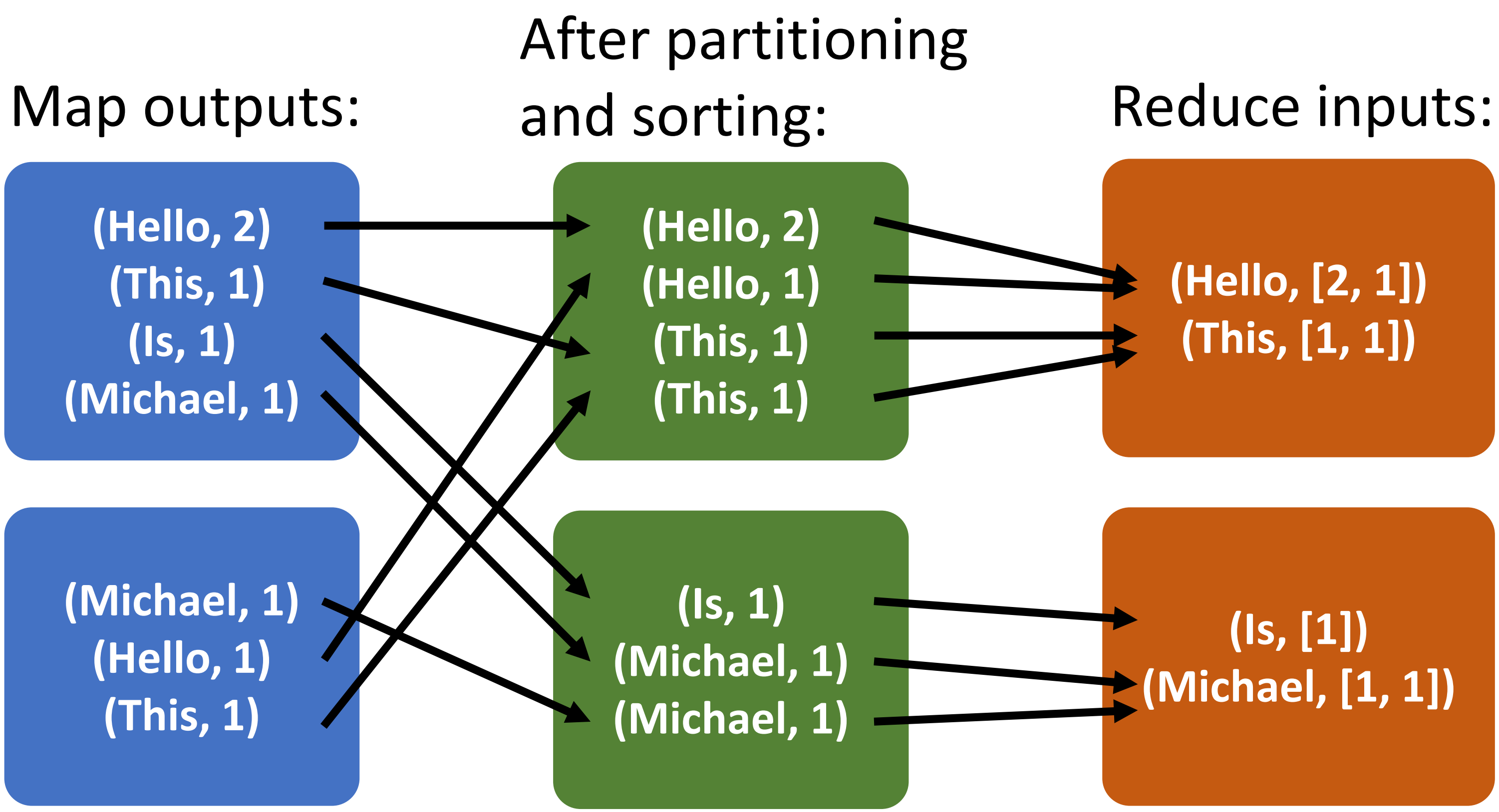
- Decides which reducer to process the map outputs
- Default **partitioner**:

  `(k, v) → Hash(k) mod #reducers`

  - Same key → always processed by the same reducer
- Users can customize partitioner
  - To change the way of grouping map outputs for reducers
    Ex: *Dates as keys* → *group by months*

Shuffling & Sorting

- After partitioning, map outputs are sorted by **keys**



Advanced Example: TeraSort

- Problem: How to sort terabytes of data

Map **Partition** **Reduce**

Advanced Example: TeraSort

- Problem: How to sort terabytes of data

Map

Default (no-op) mapper

`(k, v) → (k, v)`

k = 15
k = 4
k = 10

k = 7
k = 18
k = 3

Partition into ranges

Partition

$$k \,/\, \frac{\text{Max - Min}}{\text{\# Reducer}}$$

k = 4
k = 7
k = 3

0 <= k < 10

k = 15
k = 10
k = 18

10 <= k < 20

Sorted

Reduce

Default (no-op) reducer

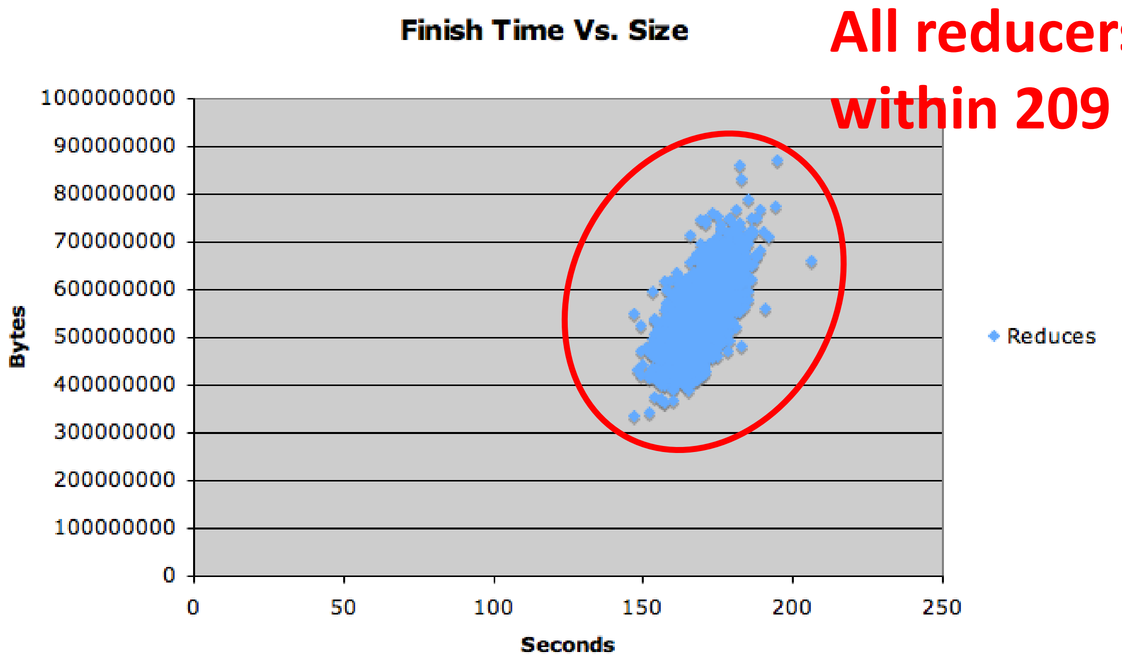
`(k, [v]) → (k, [v])`

k = 3
k = 4
k = 7

k = 10
k = 15
k = 18

TeraSort Performance

- TeraGen + TeraSort + TeraValidate (O'Malley 2008)
  - 10 billion key-value pairs
  - 910 machines with 4 dual-core Xeon CPUs, 8GB RAM
  - 1800 mappers and 1800 reducers



All reducers completed within 209 seconds

MapReduce Implementations

- The original proprietary implementation by Google
- Apache Hadoop
  - Most widely used, open-source version
- Cloud implementations (mainly based on Hadoop)
  - Amazon Elastic MapReduce
  - Azure HDInsight
  - Google Cloud Dataproc

Hadoop API Example

Word count:

- Map: `(k1, v1) → list(k2, v2)`

For each word in each value, emit (word, 1)

(A.txt, “Hello This Is Hello Michael”) → (Hello, 1), (This, 1), (Is, 1), (Hello, 1), (Michael, 1)

(B.txt, “Michael Hello This”) → (Michael, 1), (Hello, 1), (This, 1)

- Reduce: `(k2, list(v2)) → list(v2)`

For each key, emit (key, sum of all values)

(Hello, [1, 1, 1]) → Hello: 3

(This, [1, 1]) → This: 2

(Is, [1]) → Is: 1

(Michael, [1, 1]) → Michael: 2

Hadoop API Example

Define your own Mapper

```
class WordCountMapper
        extends Mapper<Object, Text, Text, IntWritable> {

    public void map(Object key, Text value, Context context )
            throws IOException, InterruptedException {

        StringTokenizer itr = new StringTokenizer(value.toString());

        while (itr.hasMoreTokens()) {
            context.write(new Text(itr.nextToken()),
                          new IntWritable(1));
        }
    }
}
```

Hadoop API Example

```
class WordCountMapper
        extends Mapper<Object, Text, Text, IntWritable> {

    public void map(Object key, Text value, Context context )
            throws IOException, InterruptedException {

        StringTokenizer itr = new StringTokenizer(value.toString());

        while (itr.hasMoreTokens()) {
            context.write(new Text(itr.nextToken()),
                          new IntWritable(1));
        }
    }
}
```

Process keys & values

Hadoop API Example

```
class WordCountMapper
        extends Mapper<Object, Text, Text, IntWritable> {

    public void map(Object key, Text value, Context context )
            throws IOException, InterruptedException {

        StringTokenizer itr = new StringTokenizer(value.toString());

        while (itr.hasMoreTokens()) {
            context.write(new Text(itr.nextToken()),
                          new IntWritable(1));
        }
    }
}
```

Store to context

Hadoop API Example

Define your own Reducer

```
class WordCountReducer
        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));
    }
}
```

Hadoop API Example

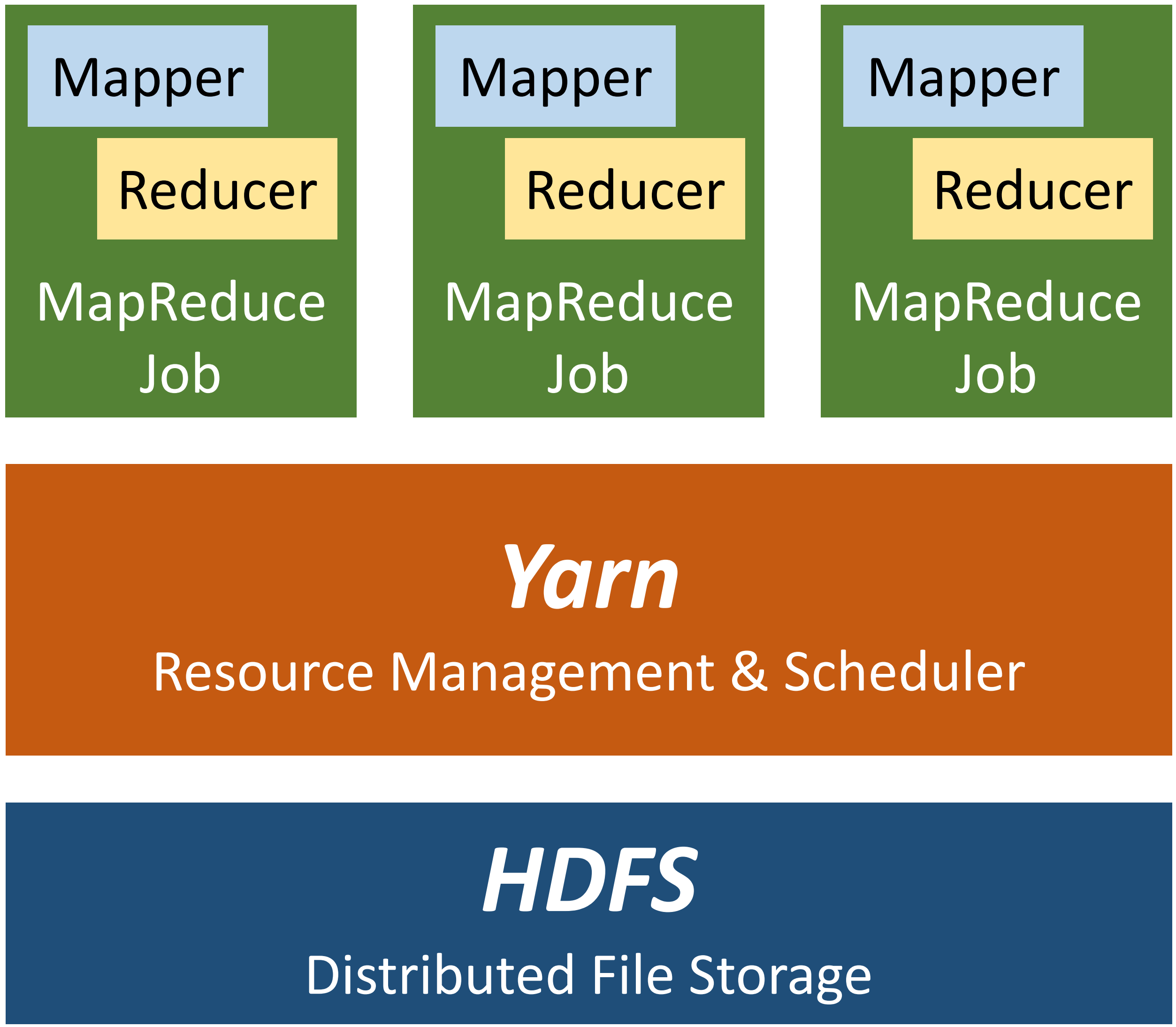
```
class WordCountReducer
        extends Reducer<Text,IntWritable,Text,IntWritable> {

    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));
    }
}
```

Store to context

Hadoop API Example

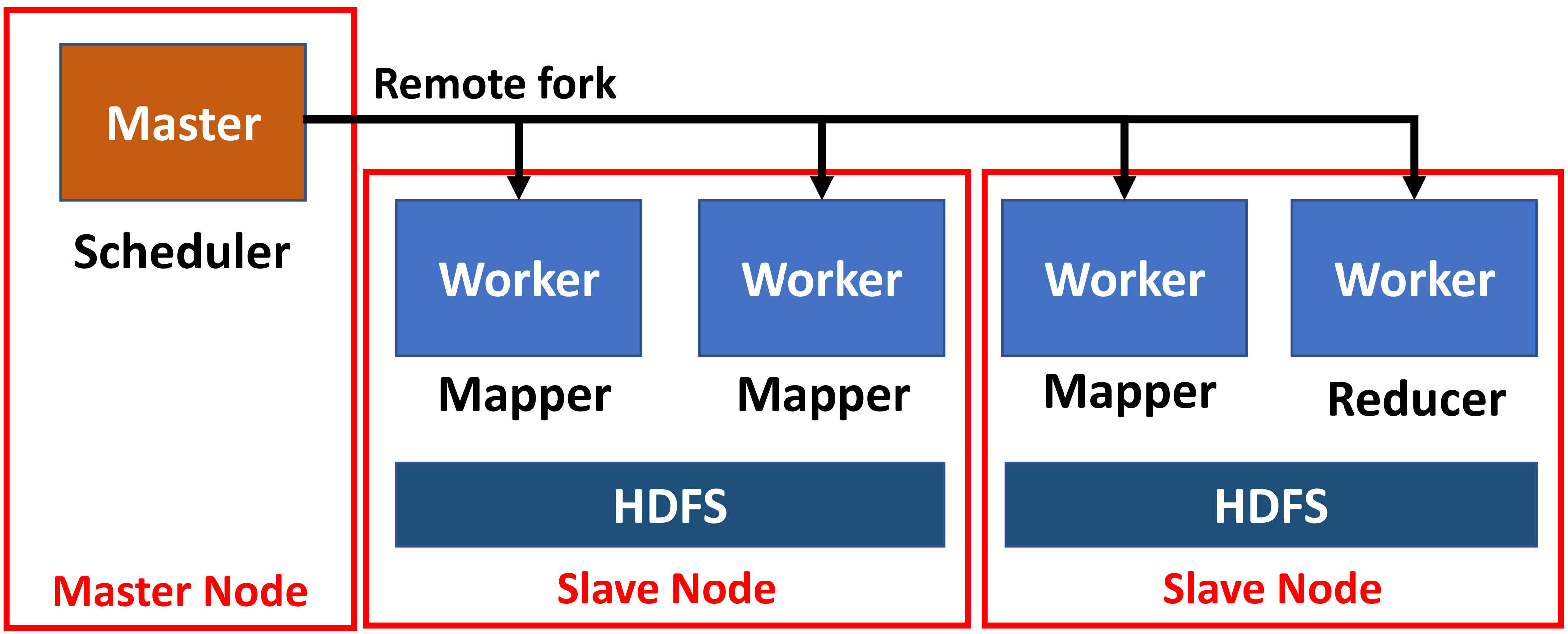
```
public class WordCount {

    public static void main(String[] args) throws Exception {
        /* Create configuration & job*/
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
         /* Set mapper & reducer */
         job.setMapperClass(WordCountMapper.class);
         job.setReducerClass(WordCountReducer.class);
         . . .
         /* Submit & wait for the job to complete */
         job.waitForCompletion(true);
    }
}
```

Hadoop Architecture



Resource Scheduling

- Hadoop master forks multiple workers across nodes
- Each idle worker can be assigned as:
  - **Mapper:** each work on a data split
  - **Reducer:** each work on a part of map outputs



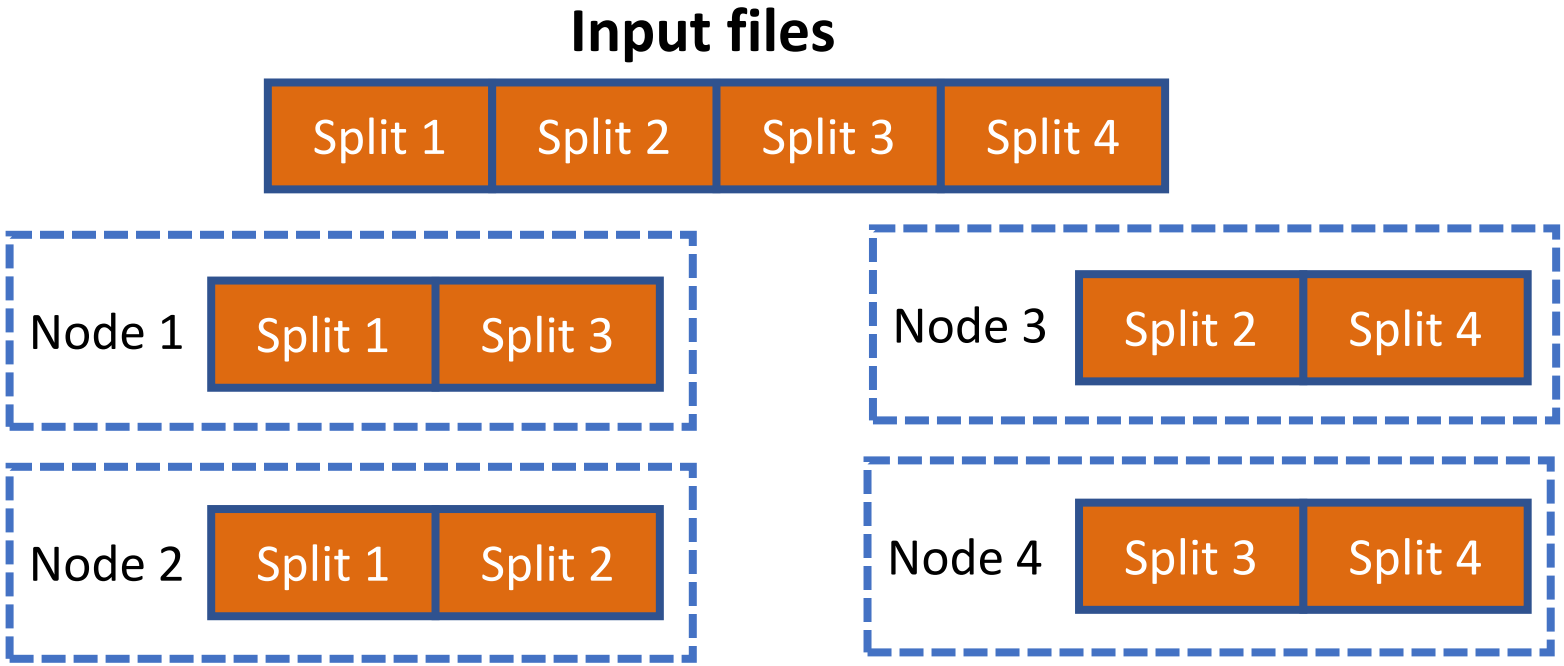
Data Partitioning & Movement

- HDFS (Hadoop Distributed file system)
  - Partitions input files into multiple splits
  - Replicating splits for efficiency and crash tolerance

Input files

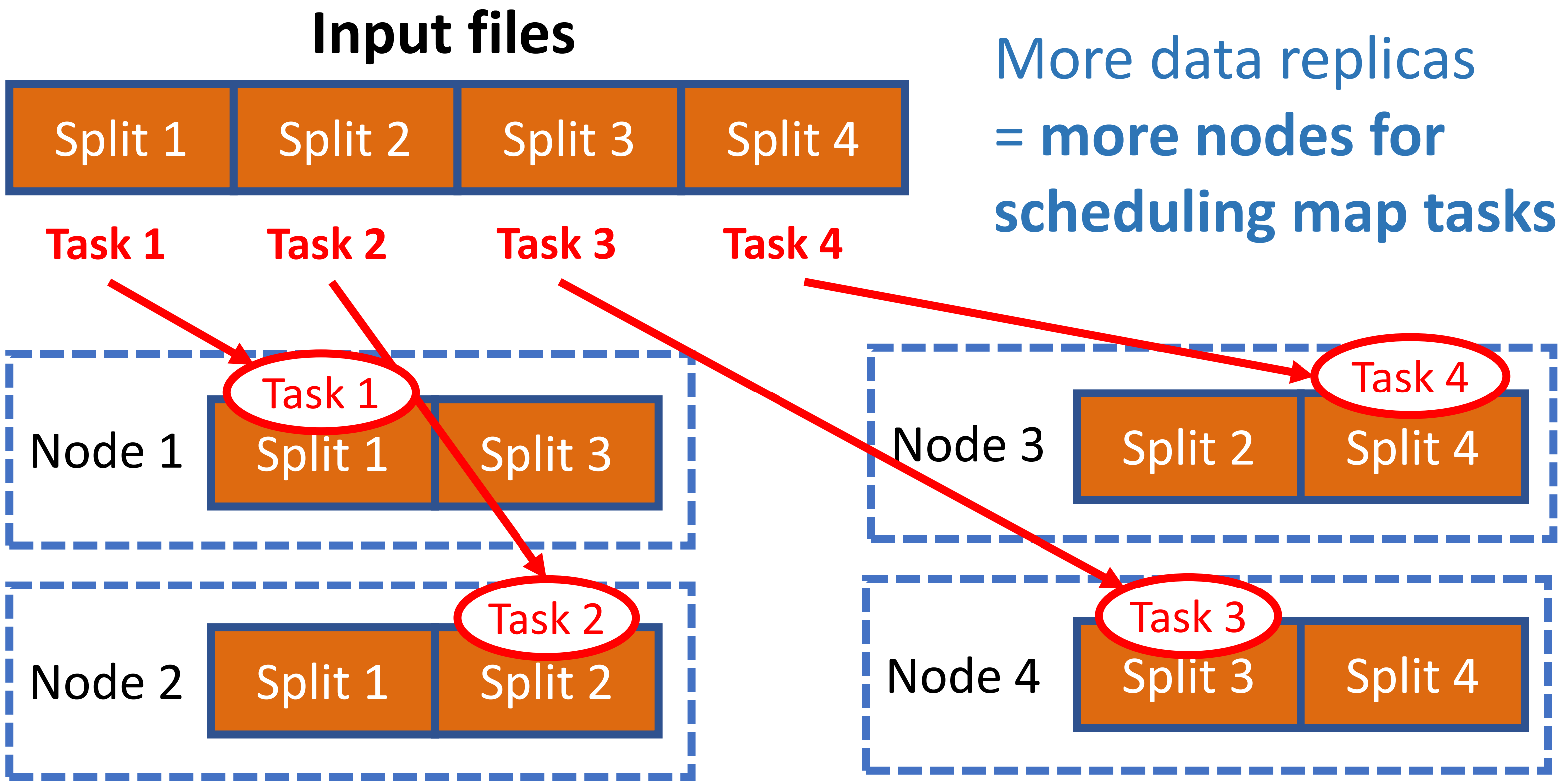
Data Partitioning & Movement

- HDFS (Hadoop file system)
  - Partitions input files into multiple splits (shards)
  - Replicating splits (shards) across nodes



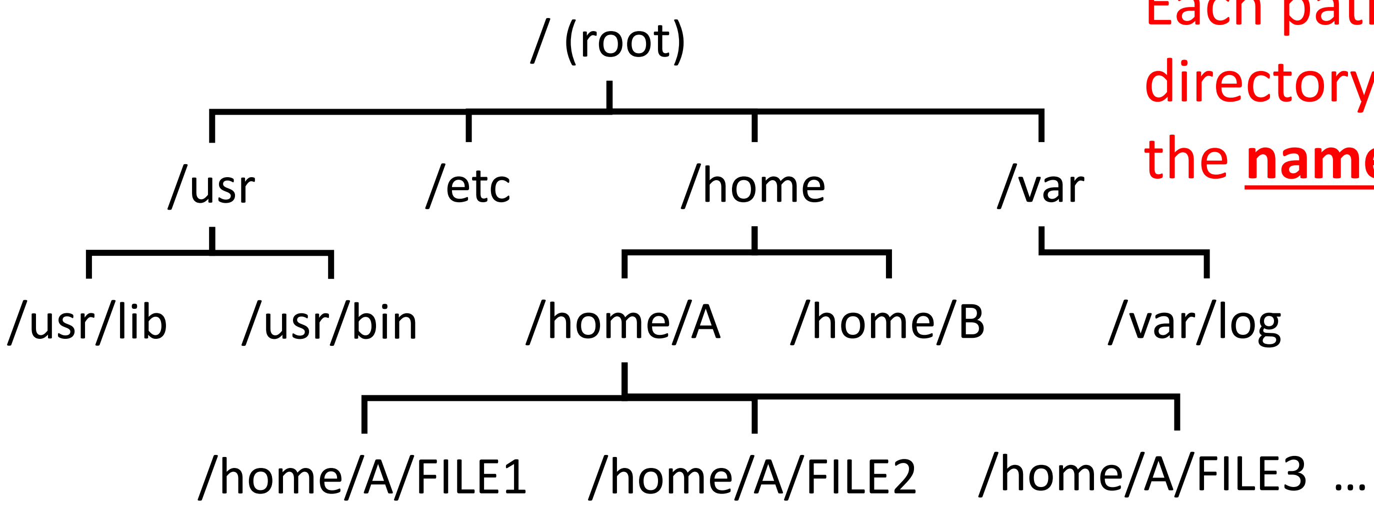
Data Partitioning & Movement

- Key optimization:
  Hadoop **moves computation to be near the data**, instead of moves data to be near computation.



HDFS (1/3)

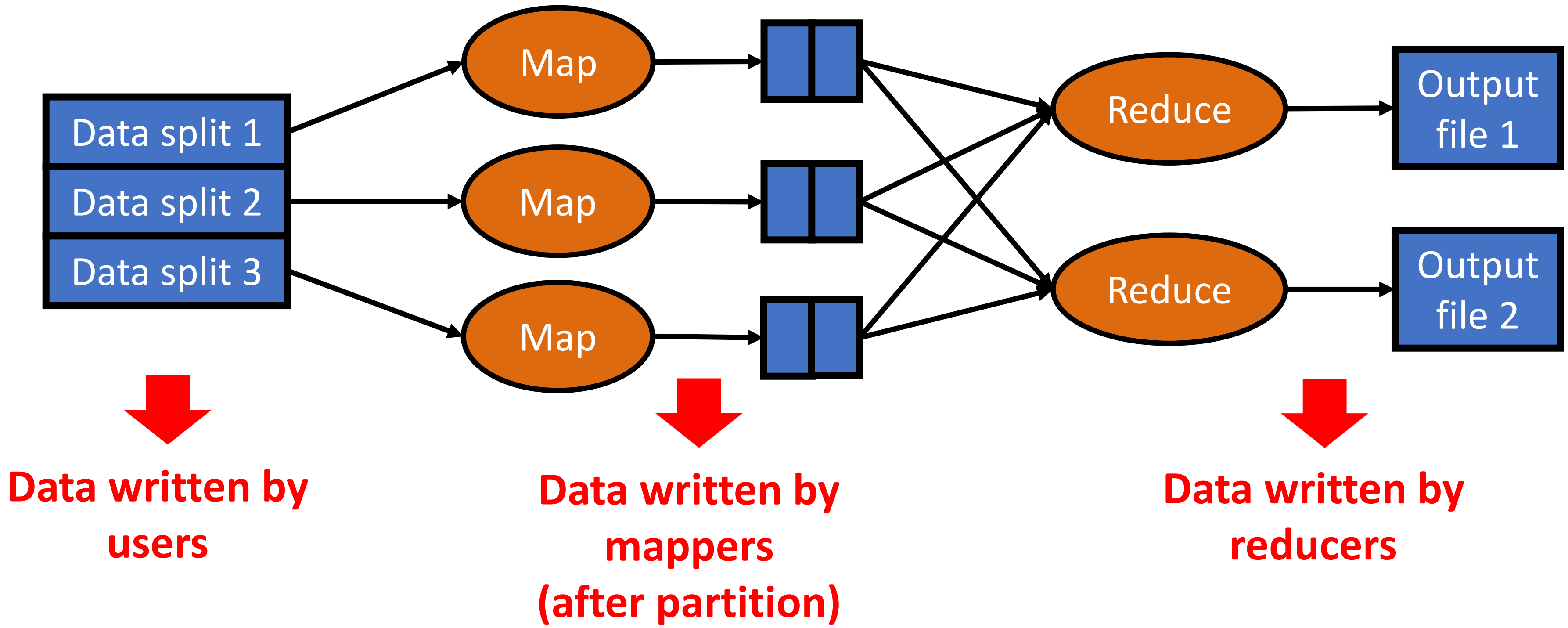
- HDFS works as a distributed <u>**file system**</u>
  - Not as database or key-value store



Each path is a unique directory or file in the <u>**namespace**</u>

HDFS (2/3)

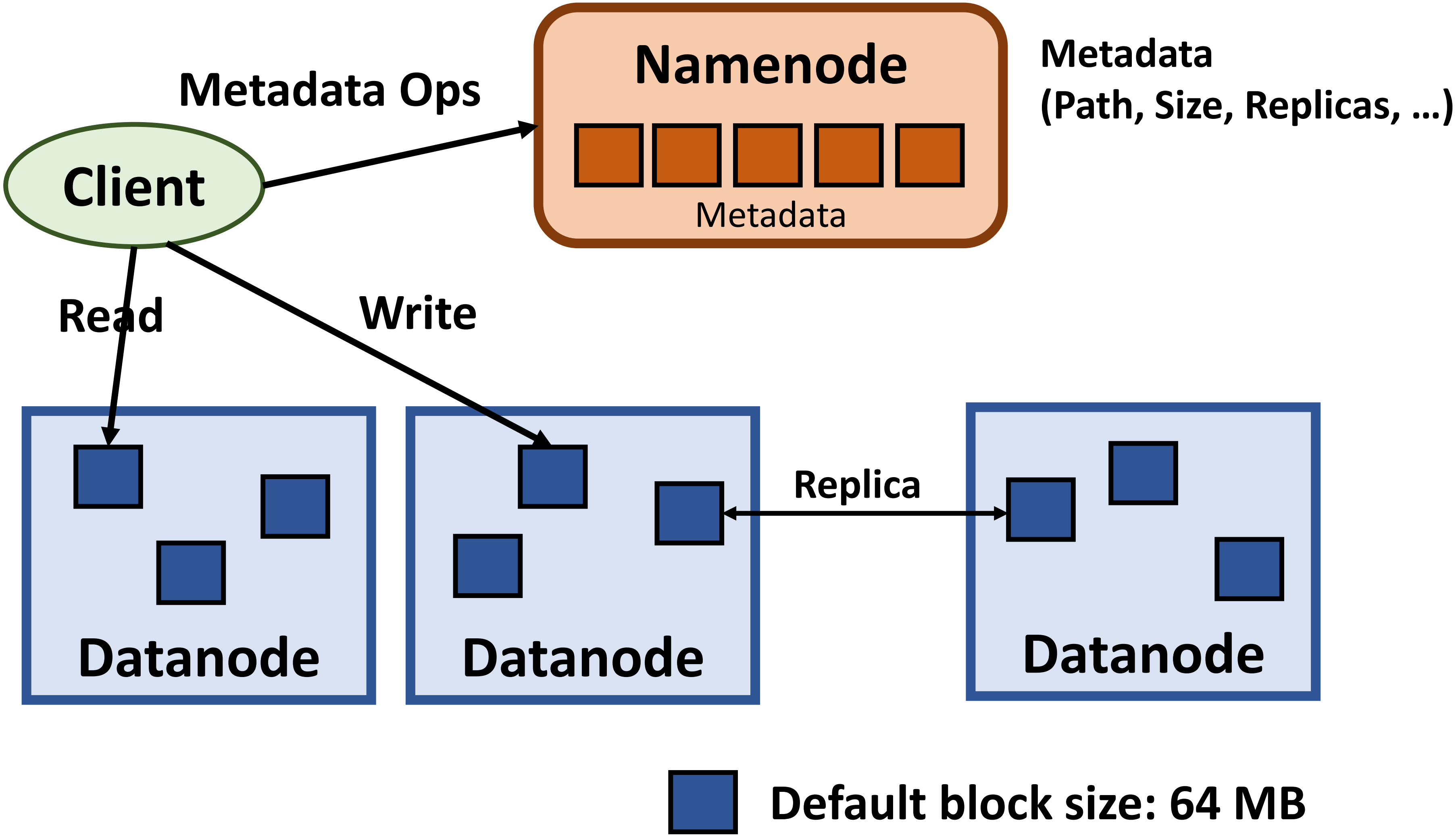
- **Hadoop chooses FS over DB and KV-store for its unique access patterns and requirements**



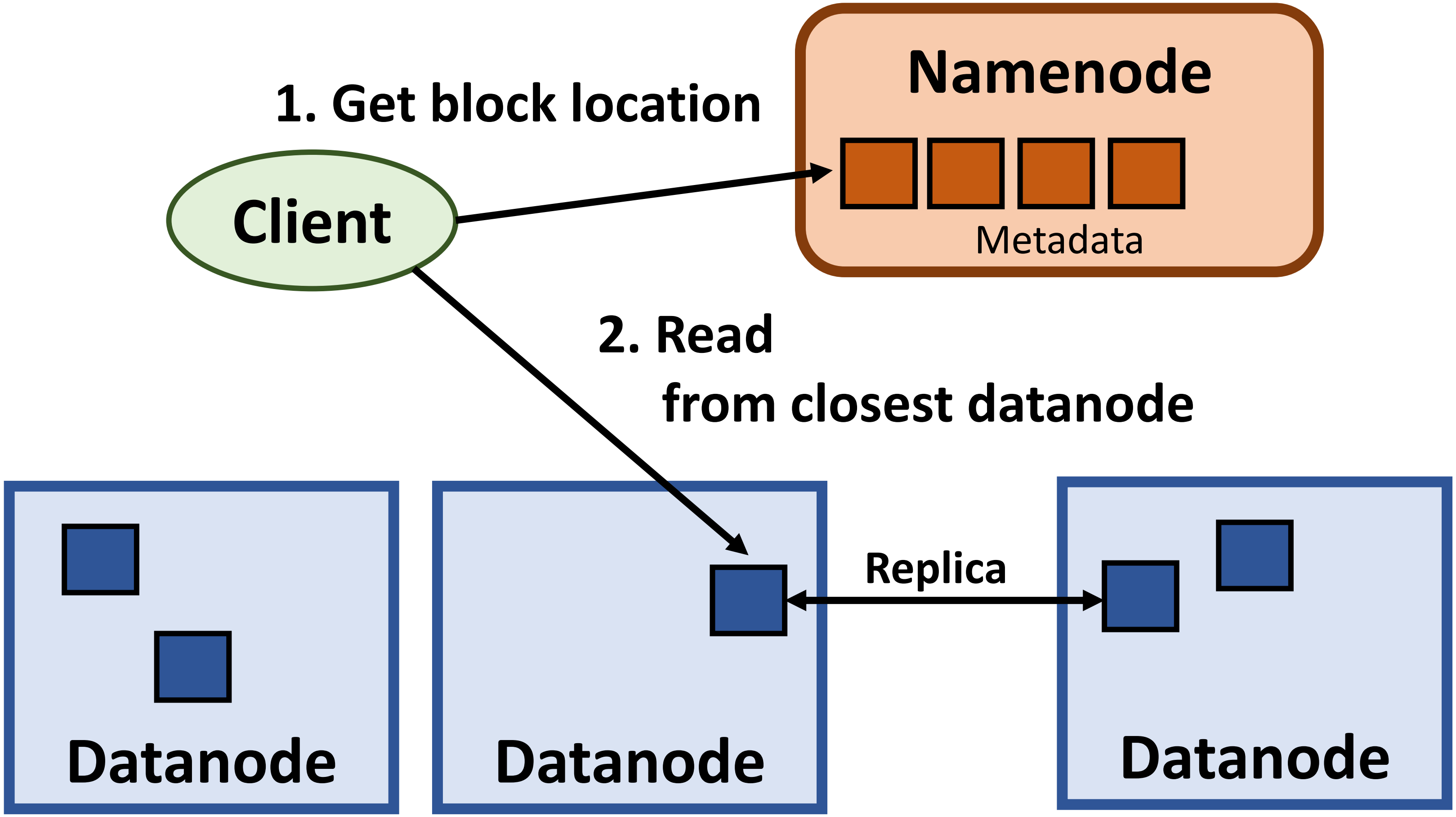
HDFS (3/3)

- **Access patterns for Hadoop**
  - Very large files: nearly GBs ~ TBs
  - Streaming (sequential) access instead of random access
  - Throughput is more important than latency
  - **Write-once-read-many:** once a file is created and written, it never changes again

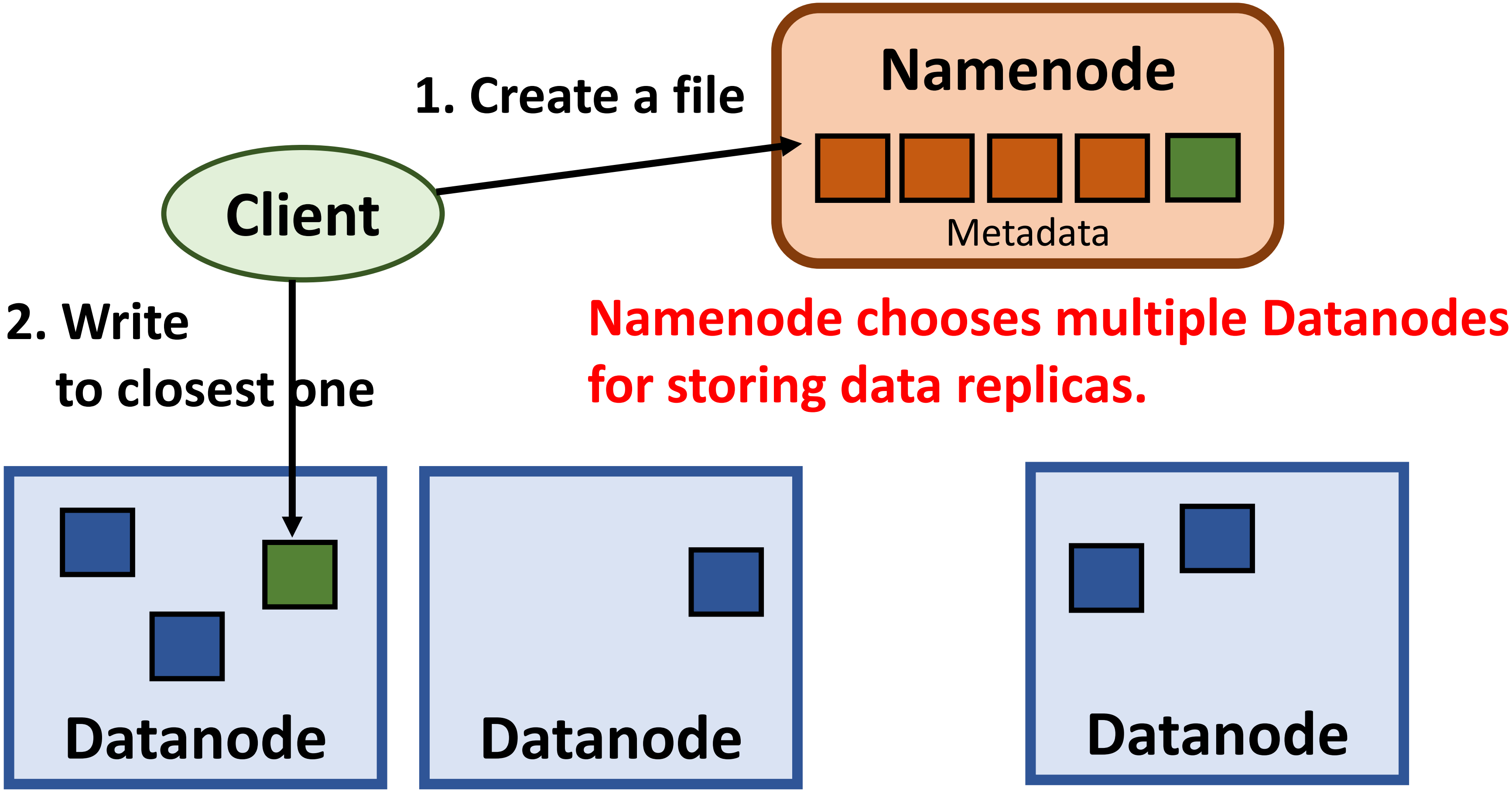
Namenodes and Datanodes



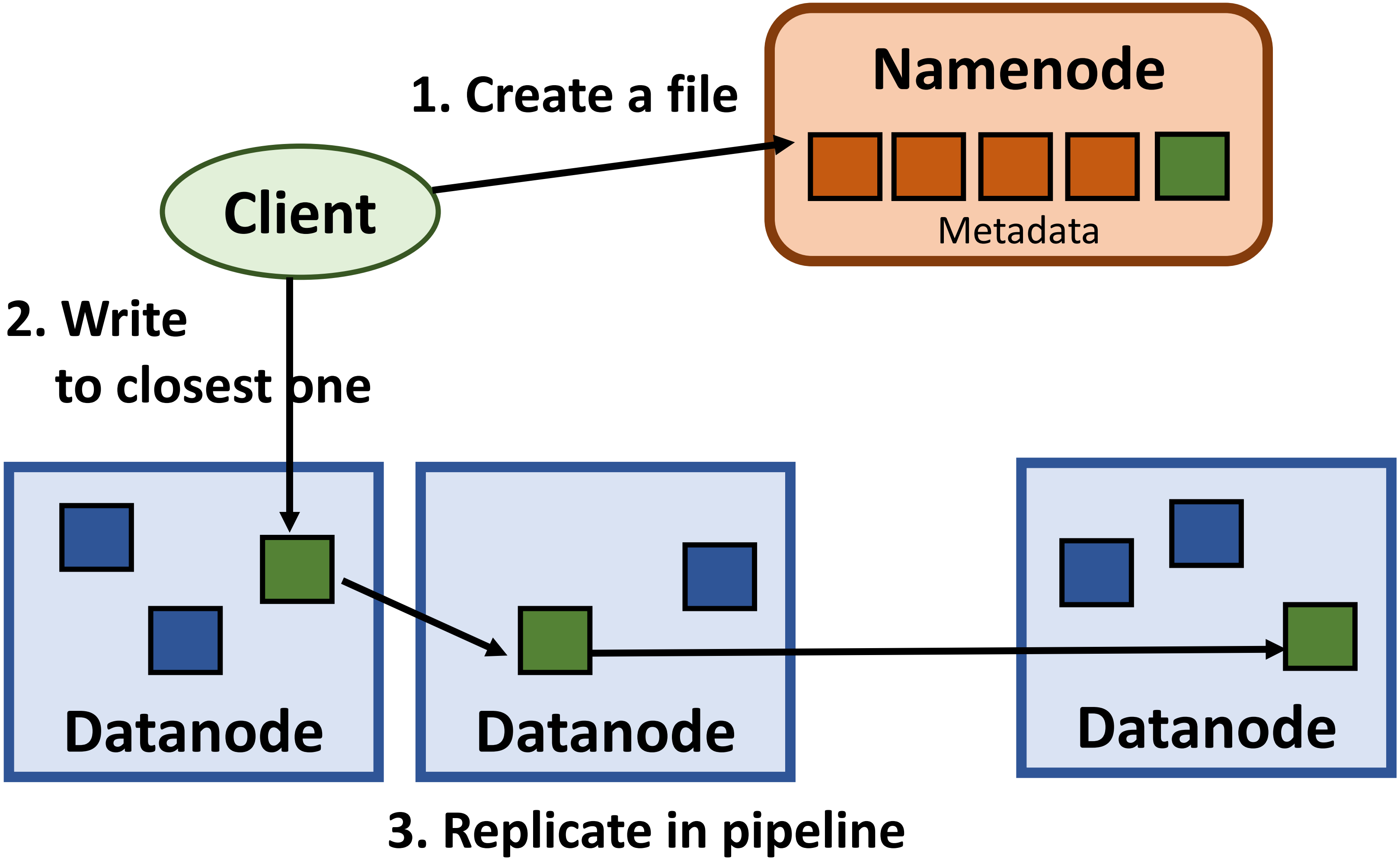
Read Operation



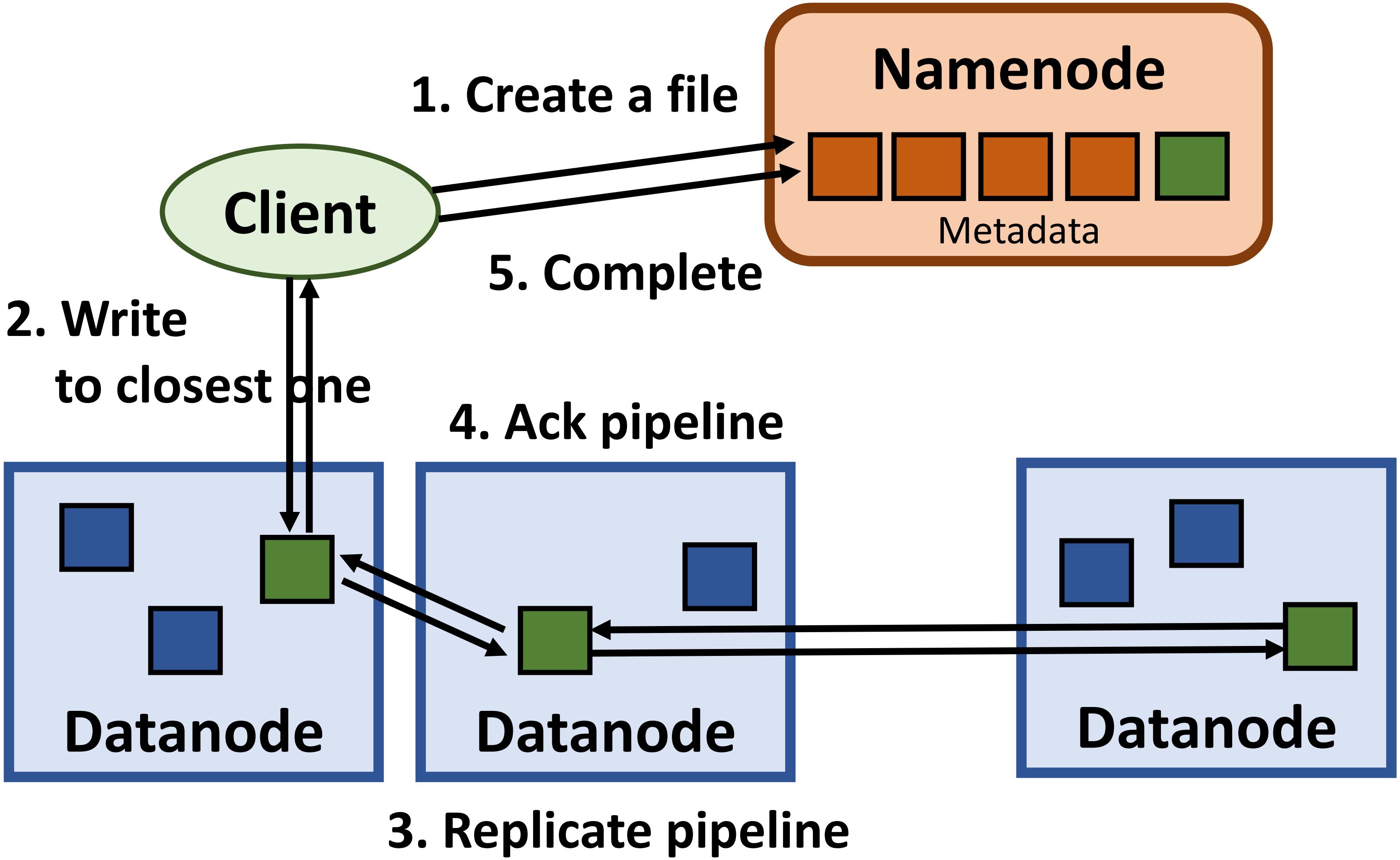
Write Operation (1/3)



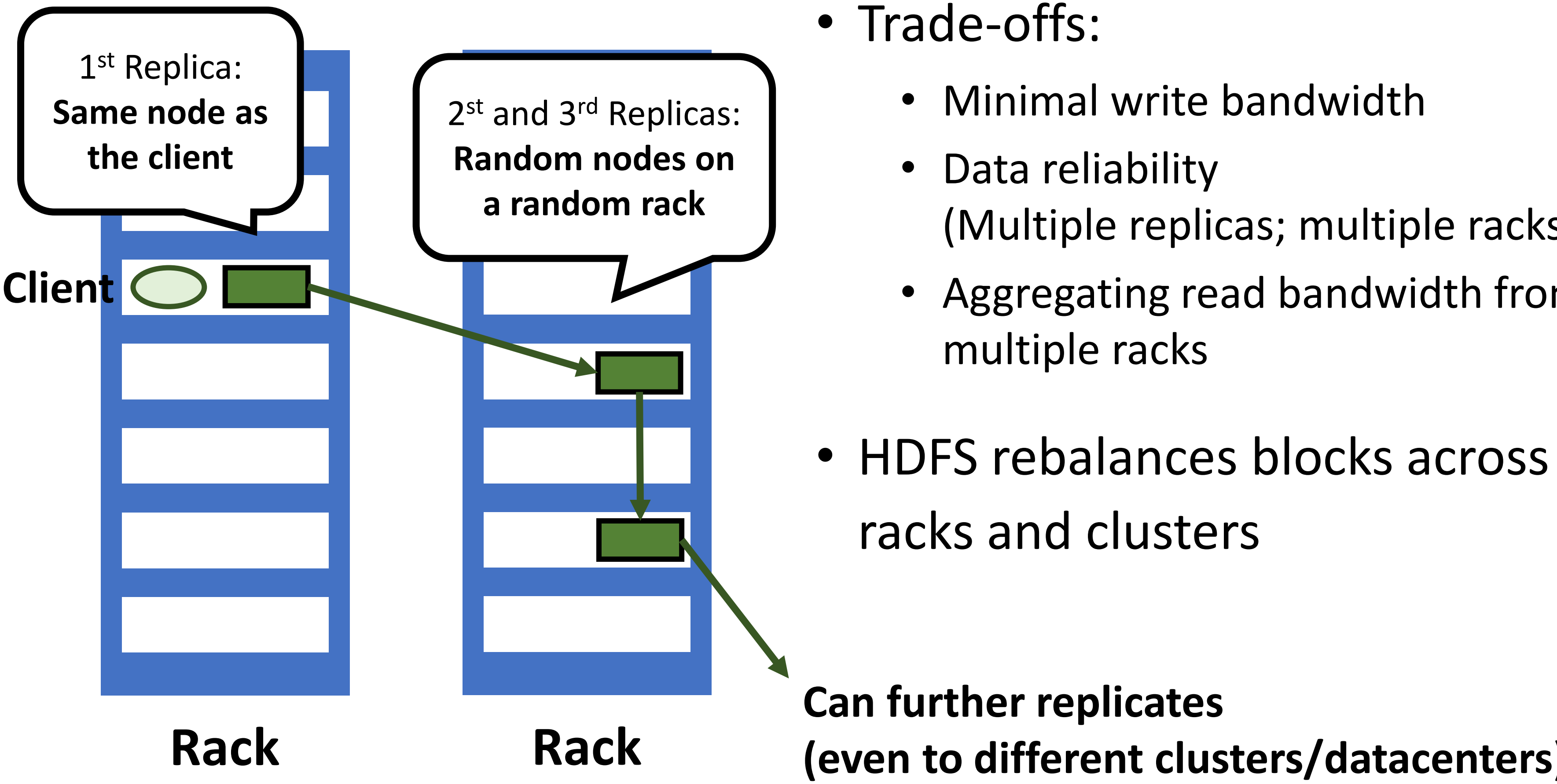
Write Operation (2/3)



Write Operation (2/3)



Block Placement



- Trade-offs:
  - Minimal write bandwidth
  - Data reliability (Multiple replicas; multiple racks)
  - Aggregating read bandwidth from multiple racks
- HDFS rebalances blocks across racks and clusters

No more than one replica per node;
No more than two replicas per rack;

Dealing with Stragglers

- **Stragglers:** workers that run unexpectedly long
  - Example: a machine with a bad disk can slow down its read from 30MB/s to 1MB/s
- Backup tasks:
  - Spawning backups of in-complete tasks when the whole computation is close to completion
  - If the backup task finishes first, kill the original task