CS172 INFORMATION RETRIEVAL

MapReduce &
Distributed File System

Announcements

- Please complete readings
  - MapReduce (from Mining of Massive Datasets Chapter 2)
    - infolab.stanford.edu/~ullman/mmds/book.pdf
  - Chapter 5 in our textbook

Timeline of Databases

- **1960s** – hierarchical databases which provided support for concurrency, recover, and fast access.
- **1970-1972** - Edgar Codd who was working at IBM proposed the 'relational database model'. Provdied support for more reliability, less redundancy, more flexibility, etc.
- **1970s** – two major RDBMS prototypes were proposed: Ingres and System R
- **Mid 1970s** – A DB model called Entity –Relationship(ER) was proposed
- **1980s** – Structured Query Language (SQL) became standard querying language.
- **Late 1980s - 1990s** – Parallel and distributed databases

Parallel

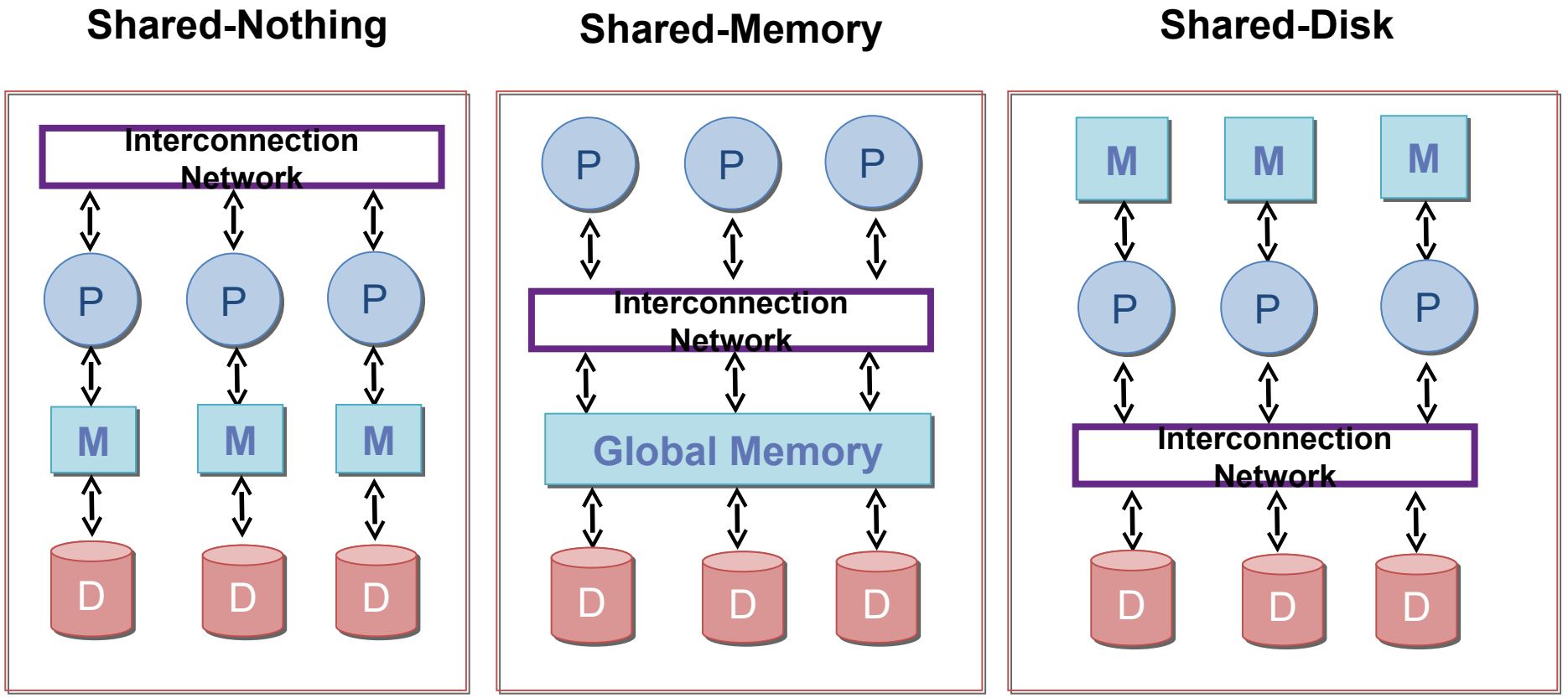
- Improve performance through parallel implementation
- Architecture:
  - Machines are physically close to each other (same server room)
  - Machines connected by dedicated high-speed LANs and switches
  - Communication cost is assumed to be small
  - Usually, shared-memory, shared-disk or shared-nothing architecture.

Distributed

- Data is physically stored across several sites, where each site is managed by a DBMS capable of running independently.
  - Architecture
  - Machines can be far from each other
  - Can be connected using public-purpose network (internet)
  - Communication cost and problems cannot be ignored here
  - Usually shared –nothing architecture

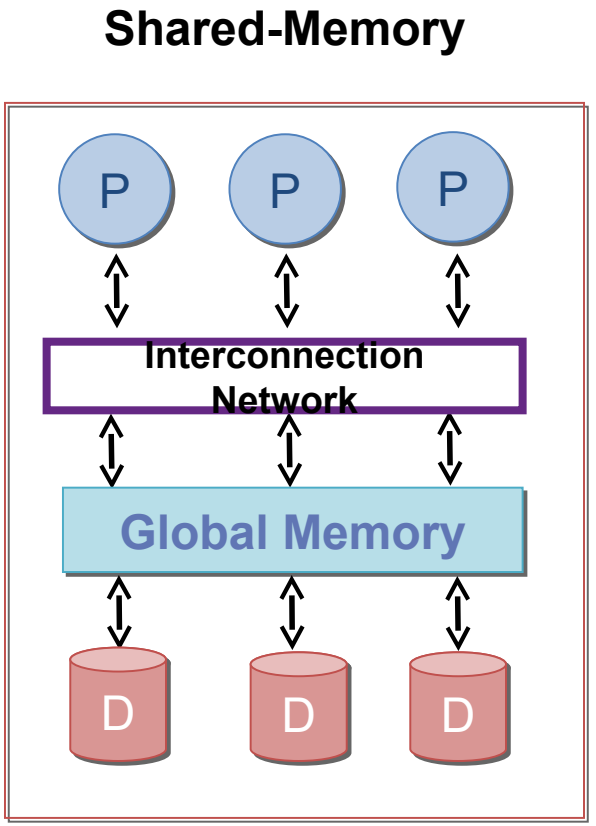
Parallel Databases - Architectures

- Three possible architectures for passing information



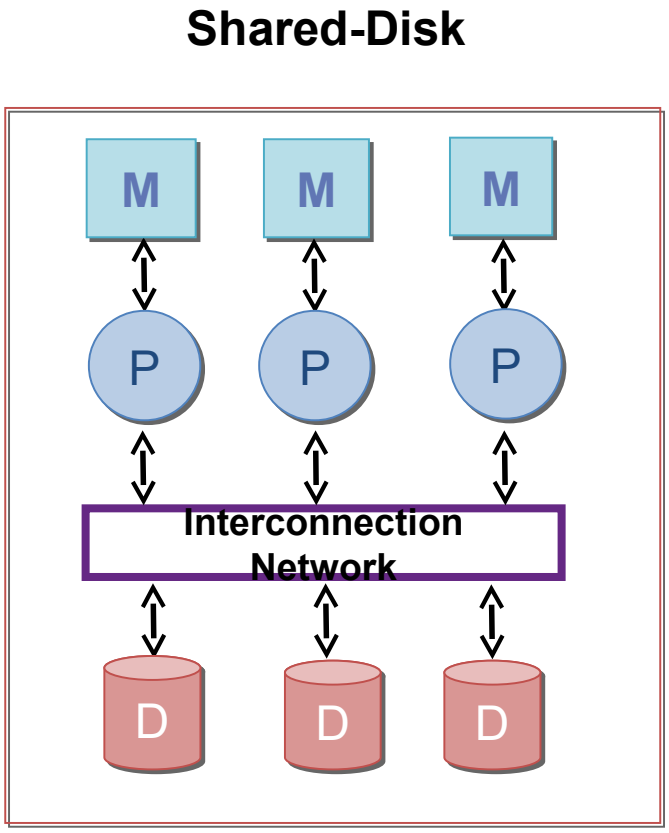
Shared-Memory Architecture

- Every processor has its own disk
- Single memory address-space for all processors
  - Reading or writing to far memory can be slightly more expensive.
- Every processor can have its own local memory and cache as well.



Shared-Disk Architecture

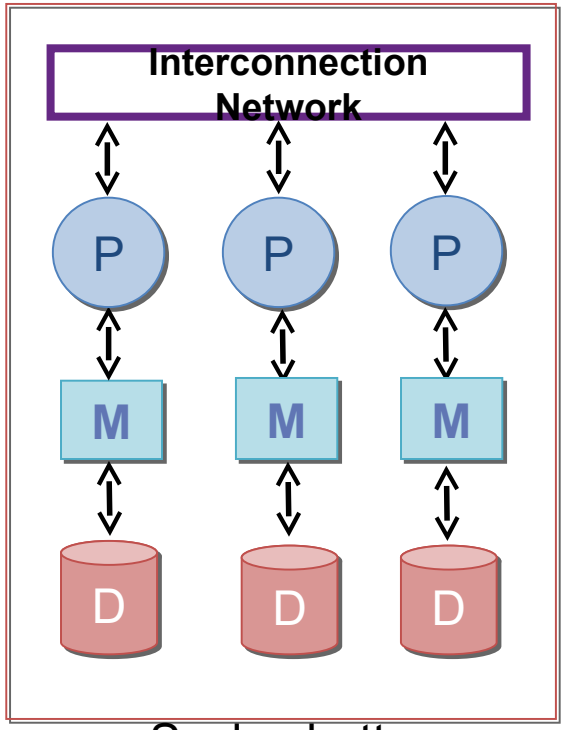
- Every processor has its own memory (not accessible by others).
- All machines can access all disks in the system.
- Number of disks does not necessary match the number of processors.



Shared-Nothing Architecture

- Most common architecture nowadays
- Every machine has its own memory and disk
  - Many cheap machines (commodity hardware).
- Communications is done through high speed network and switches.
- Usually machines can have a hierarchy
  - Machines on same rack
  - Then racks are connected through high speed switches.

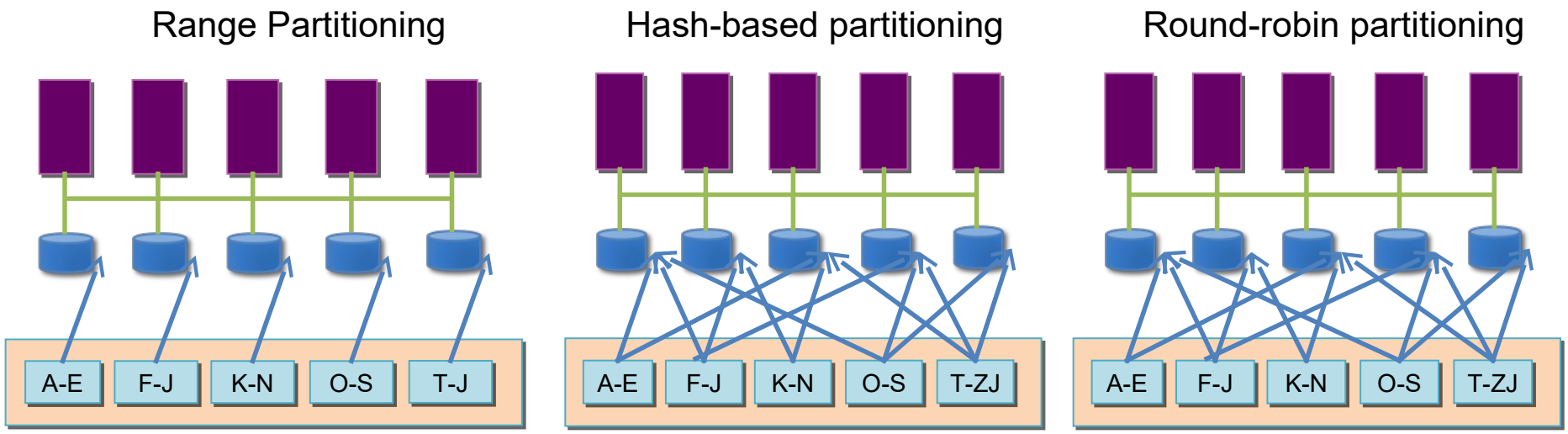
Shared-Nothing



Scales better
Easier to build
Cheaper cost

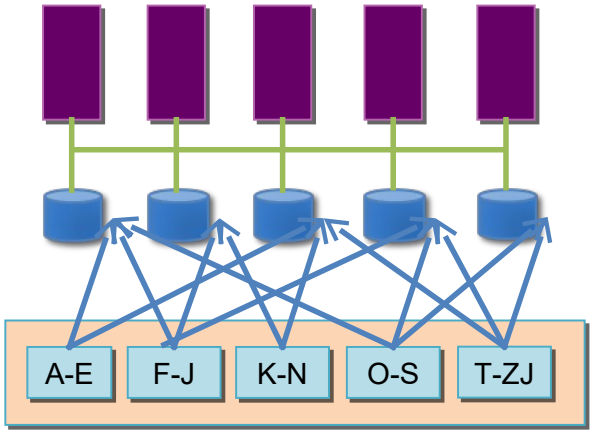
Partitioning of Data

- How to partition a relation R (table) over M machines?



- Shared-nothing architectures are sensitive to partitioning
- Good partitioning depends on what operations are commons
- load balancing is important – bad partitioning causes data skew

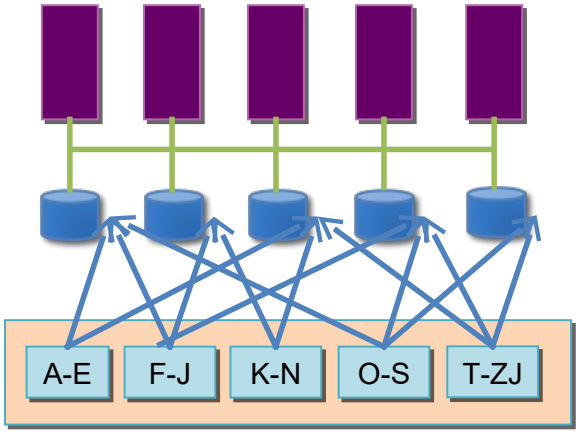
Round-robin partitioning



- Best suited for sequential scan of entire relation on each query.
- All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks!
- Range queries are difficult to process---tuples are scattered across all disks

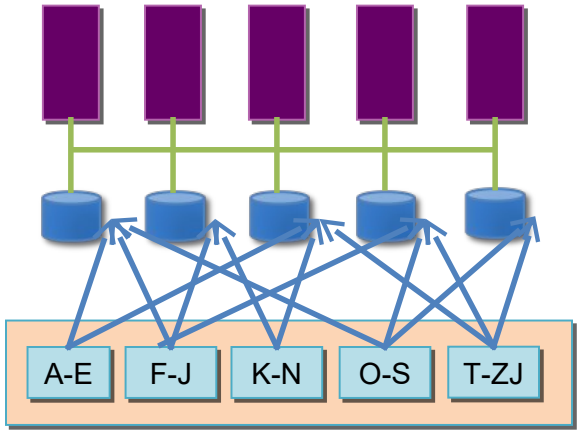
Hash-based partitioning

- Good for sequential access
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks! o Retrieval work is then well balanced between disks!
- Good for point queries on partitioning attribute!
- Can lookup single disk, leaving others available for answering other queries. !
- index on partitioning attribute can be local to disk, making lookup and update more efficient!
- No clustering, so difficult to answer range queries!



Parallel Selection: Range Partitioning

- Provides data clustering by partitioning attribute value!
  - Good for sequential access!
- Good for point queries on partitioning attribute: only one disk needs to be accessed.!
- For range queries on partitioning attribute, one to a few disks may need to be accessed!
  - Remaining disks are available for other queries!
- Caveat: badly chosen partition vector may assign too many tuples to some partitions and too few to others!



New Paradigm Shift

- In 2003/2004 Google released the research papers on the Google File System and MapReduce which started the craze!!

- MapReduce was really inspired because of the large and ever growing raw data generated by Web 2.0, social media (facebook, twitter, etc.), logs, user profiles (searches, click, transaction patterns, etc.).

- Parallel DB are expensive and unable to scale with the ever-growing amount of data.

New Paradigm Shift -

- Basic Idea:
  - Use large collections of commodity hardware, including conventional processors ("compute nodes") connected by Ethernet cables or inexpensive switches to perform processing of raw data.
  -
- The software stack :
  - **Distributed File System (DFS):** larger file units that provide replication of data or redundancy to protect against the frequent media failures that occur when data is distributed over thousands of low-cost compute nodes.
  - **MapReduce** : framework that allows users to run applications on computing clusters efficiently and in a way that is tolerant of hardware failures during the computation.

What is MapReduce?

- MapReduce is a programming model Google has used successfully is processing its “big-data” sets (~ 20000 peta bytes per day)
  - Users specify the computation in terms of a map and a reduce function,
  - Underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, and
  - Underlying system also handles machine failures, efficient communications, and performance issues.

Reference:

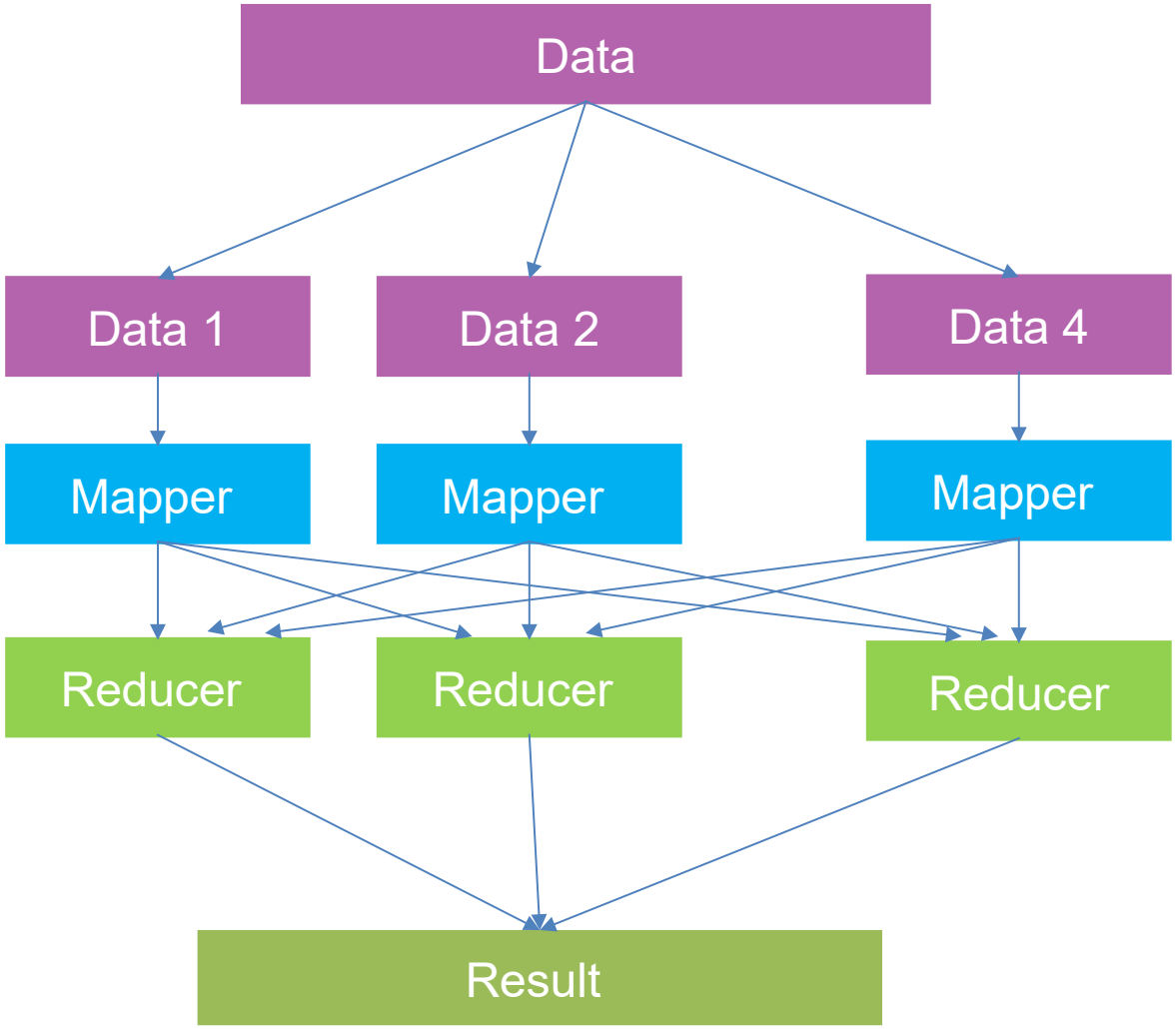
(1) Dean, J. and Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. Communication of ACM 51, 1 (Jan. 2008), 107-113.

(2) http://www.cs.rutgers.edu/~pxk/417/notes/content/mapreduce.html

MapReduce

- Parallel programming model meant for large clusters
  - User implements Map() and Reduce()
- Parallel computing framework
  - Libraries take care of EVERYTHING else
    - Parallelization
    - Fault Tolerance
    - Data Distribution
    - Load Balancing
- Useful model for many practical tasks (large data)

Basic Idea: Divide & Conquer



Map Function

Dividing the work into equal-size pieces

Reduce Function

Combining the results from independent processes

Map and Reduce

- MapReduce works by breaking the processing into two phases
  - The map phase – Process a key/value pair to generate intermediate key/value pairs
  - The reduce phase – Merge all intermediate values associated with the same key

- Programmers should specify
  - Types of input/output key-values
  - The map function
  - The reduce function

- MapReduce through WordCount example

Class MapReduce Example - WordCount

- Given a large collection of documents, count the number of occurrences of each word in this collection.

```
map (String key, String value) :
    // key : document name
    // value : 1 line of document content
    for each word w in value:
        Emit (w, "1") :

reduce (String key, Iterator values):
    // key : a word
    // values : a list of counts
    int sum = 0;
    for each v in values :
        sum += parseInt(v)
    Emit ( sum )
```

Parses 1 line of the document, and emits key value pair of <word , count>

Sums values for the same key and emails <word, total count>

Visualizing the way MapReduce works

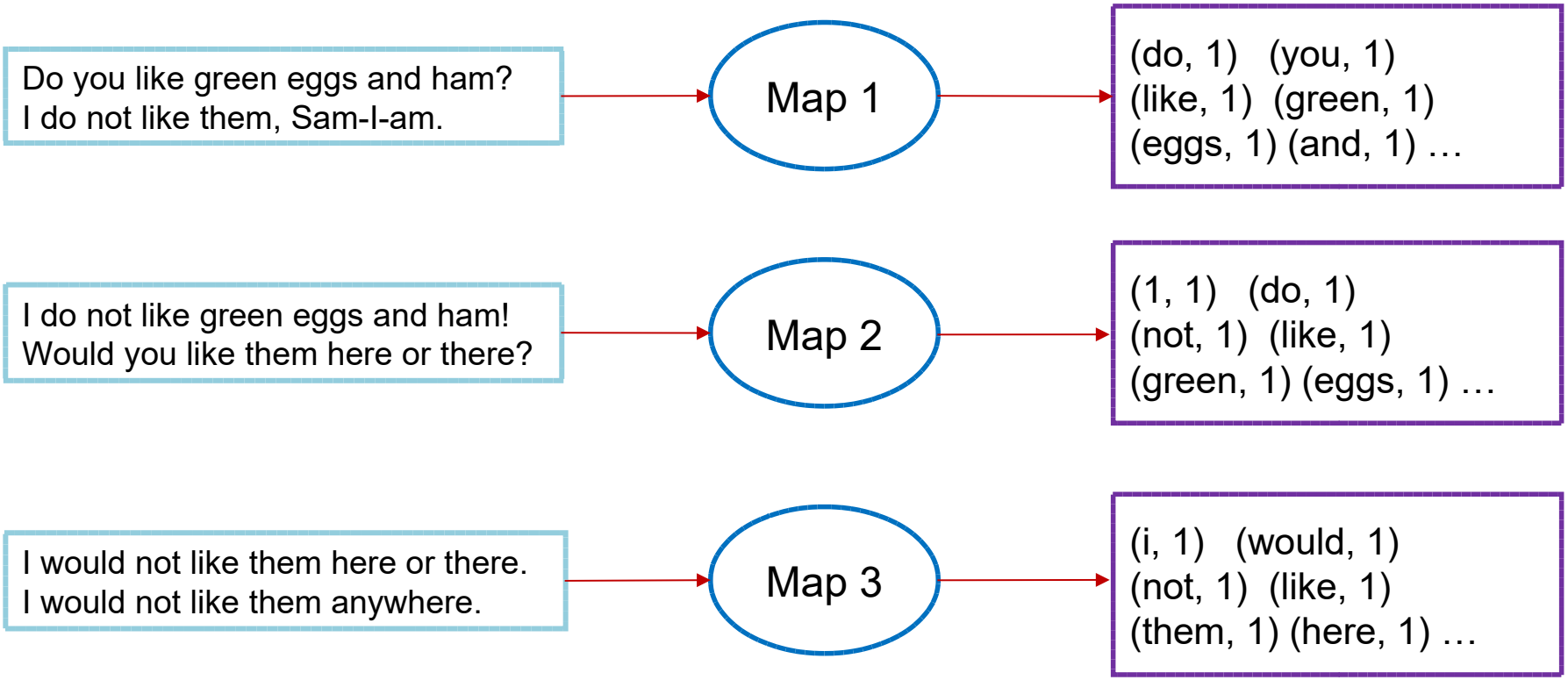
- Sample lines of input data

```
1  Do you like green eggs and ham?
2  I do not like them, Sam-I-am.
3  I do not like green eggs and ham!
4  Would you like them here or there?
5  I would not like them here or there. I would not like them
6  anywhere.
7  I do so like green eggs and ham!
8  Thank you! Thank you,
9  Sam-I-am!
```

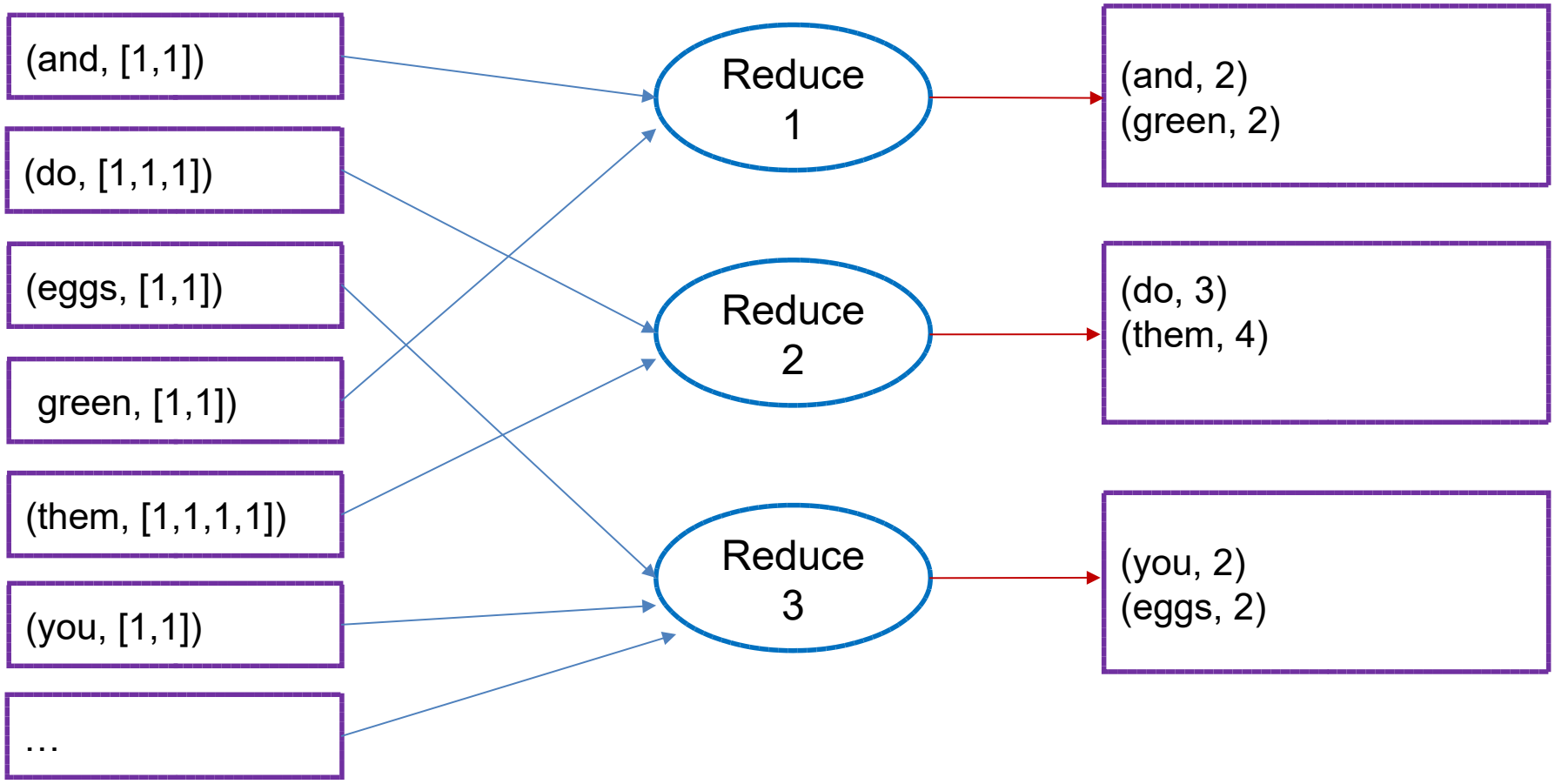
- These lines are presented to the map function as key-value pairs.

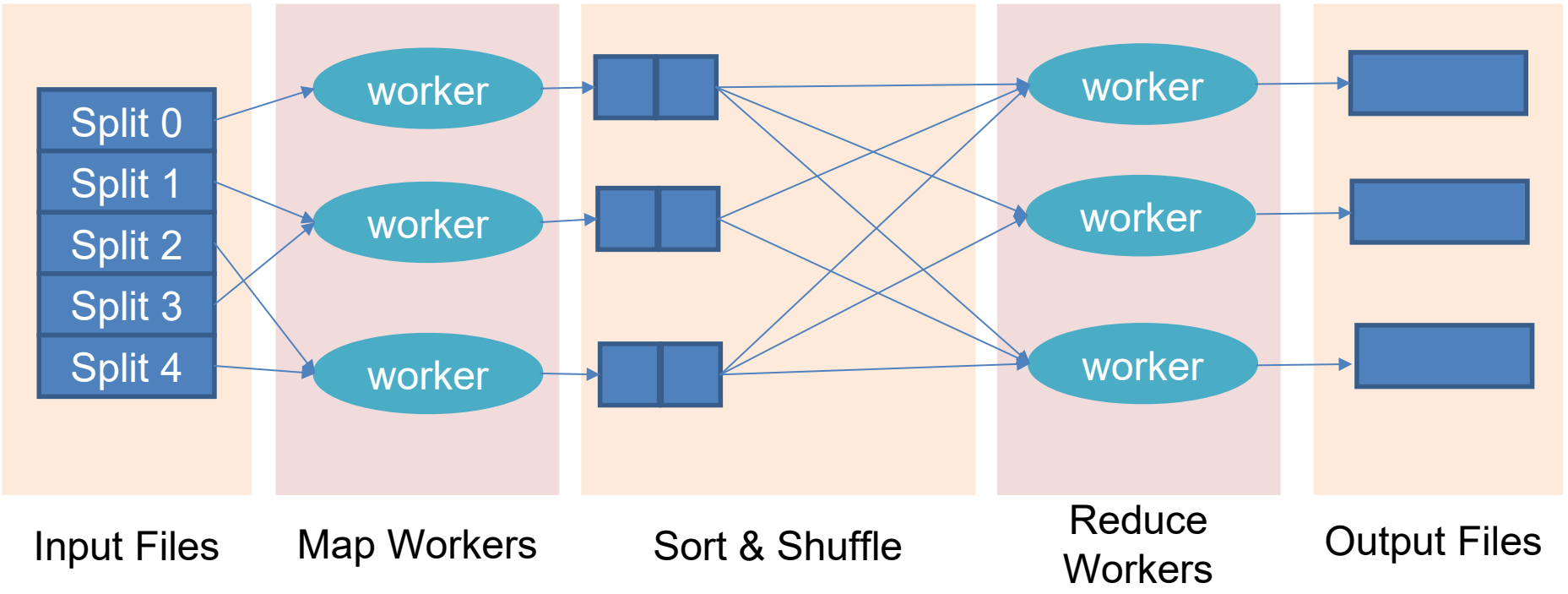
- The original document is divided into chunks, and each chunk is given to a mapper / worker.

- For each line/item in the chunk, the map function is called.
- The map function receives 1 line as input, parses the line to extract words and emits them as a key-value pair <word, 1>.

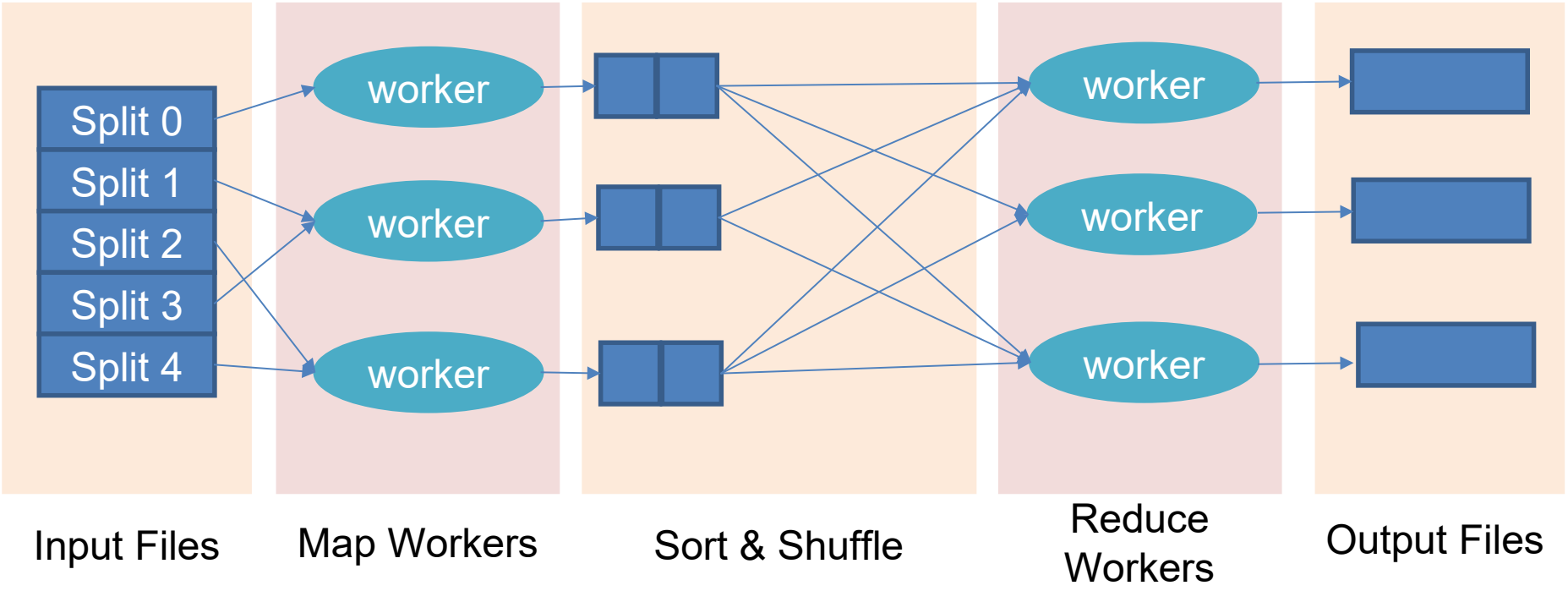


- The previous step groups the mapper output on the key, thus each reducer receives the counts for a given key / word.
- For each input, the reduce function iterates through the values for that key, and aggregates the result.





1. Shards the input files into M pieces
2. Distributes the shards of the input file to the many workers. (Each worker is performing the same computation on a different piece of data)
3. Each worker reads in the contents of the corresponding input shard, applies the map function and emits a key-value pair
4. In the Sort phase, the key-value pairs for each worker are sorted/grouped by key.
5. In the Shuffle phase, the key-value pairs are distributed to the many Reduce workers. (All pairs with the same key must be sent to the same reducer).
6. The workers apply the reduce function and emit the result.



Important Takeaways

The Sort & Shuffle phase is the intermediate step between the Mapper and the Reducer.

Each Mapper emits key-value pairs, the Sort phase sorts the key of a particular Mapper (essentially doing a Group By Operation)

The Shuffle phase distributes the key-value pairs to the Reducers such that a given Reducers receives all pair for a given key.

MapReduce Environment

- The ***Master*** keeps track of the status of each Map and Reduce task (idle, executing at a particular Worker, or completed).

- A ***Worker*** process reports to the Master when it finishes a task, and a new task is scheduled by the Master for that Worker process.

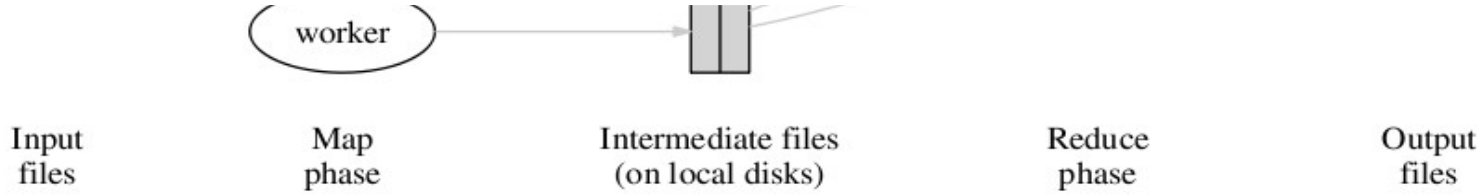


Figure 1: Execution overview

Fault tolerance: Handled via re-execution

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress *map* tasks
  - Re-execute in progress *reduce* tasks
  - Task completion committed through master

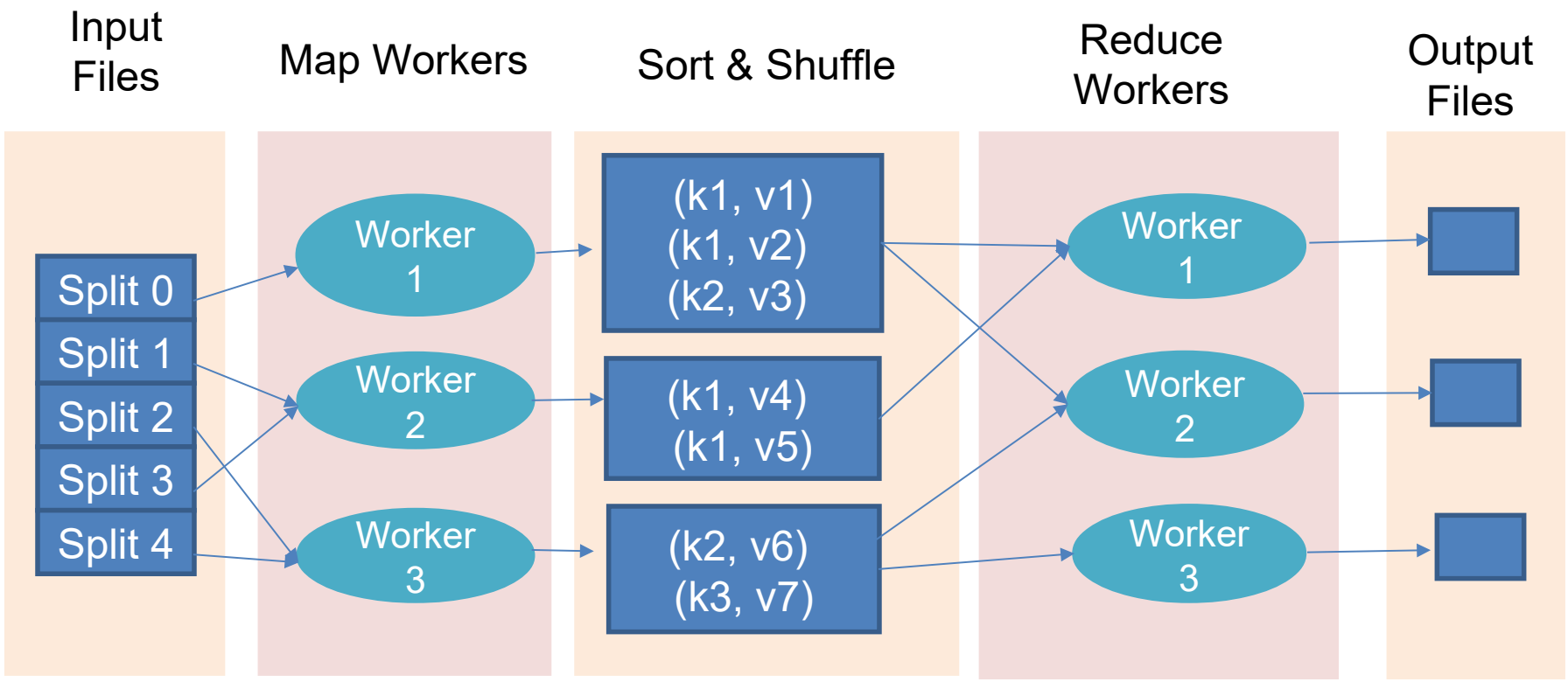
- Master failure:
  - Assume master failure unlikely.
  - Can have a shadow master node

Refinement: Combiner

- Can use Combiner if the Reducer function is associative and commutative (the values to be combined can be combined in any order, with the same result).

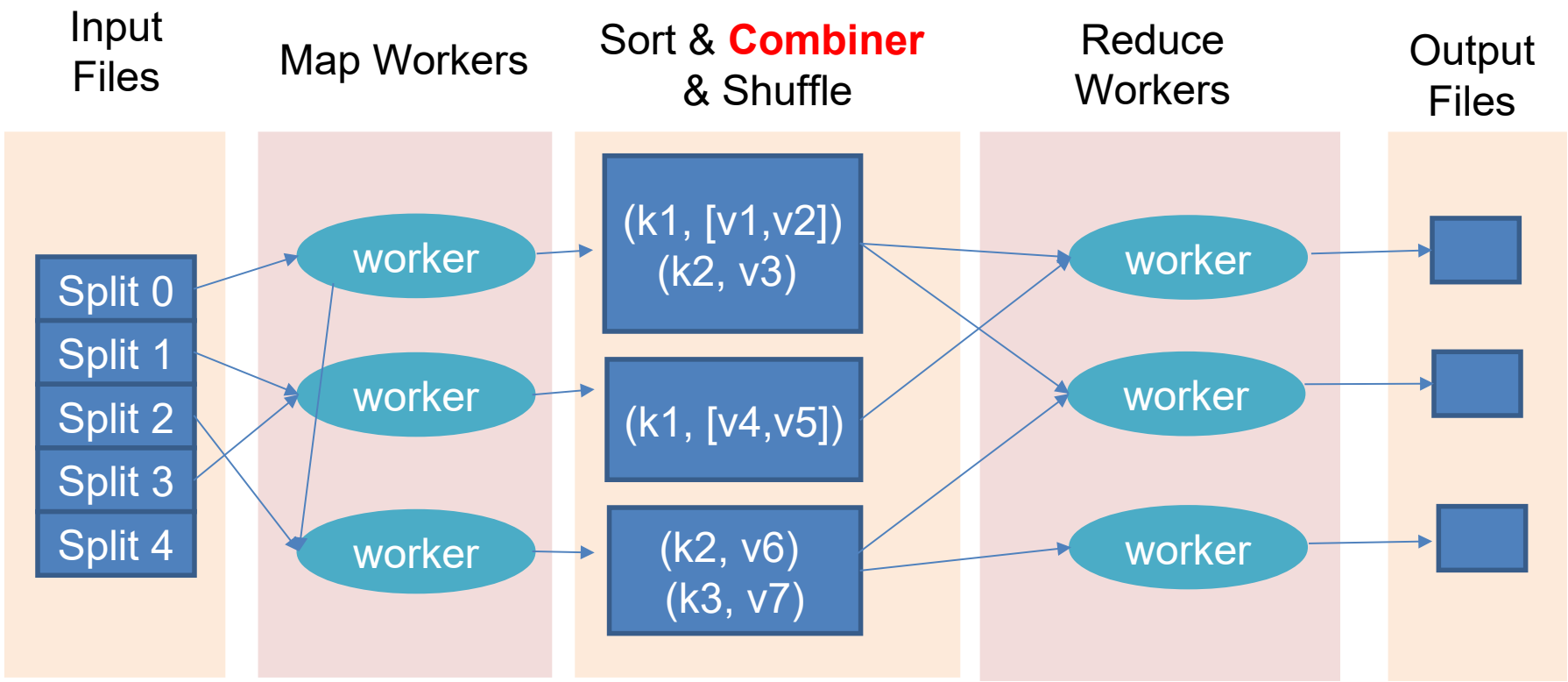
- The Combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Map and Reduce.
- Usually, the output of the map task is large and the data transferred to the reduce task is high.

- Can speed up processing by reducing amount of data sent to the Reducer.



Notice, that output of Worker 1 & 2 from the mapper phase produce redundant keys.

While this output will eventually be aggregated by the reducer, its best to summarize it during the Mapper Phase so that we reduce IO (i.e. the amount of data transferred to the Reducer).


Input Files
Map Workers
Sort & Combiner & Shuffle
Reduce Workers
Output Files
Split 0
Split 1
Split 2
Split 3
Split 4
worker
worker
worker
(k1, [v1,v2])
(k2, v3)
(k1, [v4,v5])
(k2, v6)
(k3, v7)
worker
worker
worker

Refinement: Partitioner

- The partition phase takes place after the Map phase and before the Reduce phase.

- To assign the Map phase output to a Reducer, a hash function is applied to each key to determine the Reducer where the key-value pair is routed

- If the default hash function is insufficient, we can add a partitioner to partition the data using a user-defined condition to evenly distribute the Map-out equally amon the Reducers.
  - The total number of partitions is same as the number of Reducer tasks for the job.

New Paradigm Shift -

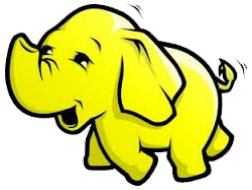
- Basic Idea:
  - Use large collections of commodity hardware, including conventional processors (“compute nodes”) connected by Ethernet cables or inexpensive switches to perform processing of raw data.

- The software stack :
  - **Distributed File System (DFS):** larger file units that provide replication of data or redundancy to protect against the frequent media failures that occur when data is distributed over thousands of low-cost compute nodes.

  - **MapReduce** : framework that allows users to run applications on computing clusters efficiently and in a way that is tolerant of hardware failures during the computation.

History of GFS

- 2003 - The Google File System (GFS) paper published
- Later on, an open source implementation of their papers was developed… called Hadoop
- Primarily developed within Yahoo then made open-source
- Some terminology differences between GFS and Hadoop Distributed File System (HDFS).



Terminology Differences

- We will see later that there is terminology differences between Google File System(GFS) and Hadoop Distributed File System (HDFS)
  - Namenode (HDFS) vs. Master (GFS)
  - Datanode (HDFS) vs. Chunckserver (GFS)
- Besides a few minor differences, HDFS is just the open source implementation of GFS.

Basic Features: HDFS

- Highly fault-tolerant
- High throughput
- Highly available
  - Files should be easily and quickly accessible by a number of users
- Suitable for applications with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware

Fault tolerance

- Failure is the norm, not the exception
- A HDFS instance may consist of thousands of server machines, each storing part of the file system's data.
- Since we have huge number of components and that each component has non-trivial probability of failure means that there is always some component that is non-functional.
- Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

Data Characteristics

- Individual file characteristics
  - Very large, multiple gigabytes per file
  - Files are updated by *appending new entries to the end* (faster than overwriting existing data)
  - Files are virtually never modified (other than by appends) and virtually never deleted.
  - Files are mostly read-only
- Write-once-read-many: a file once created, written and closed need not be changed – this assumption simplifies coherency
- A map-reduce application or web-crawler application fits perfectly with this model.

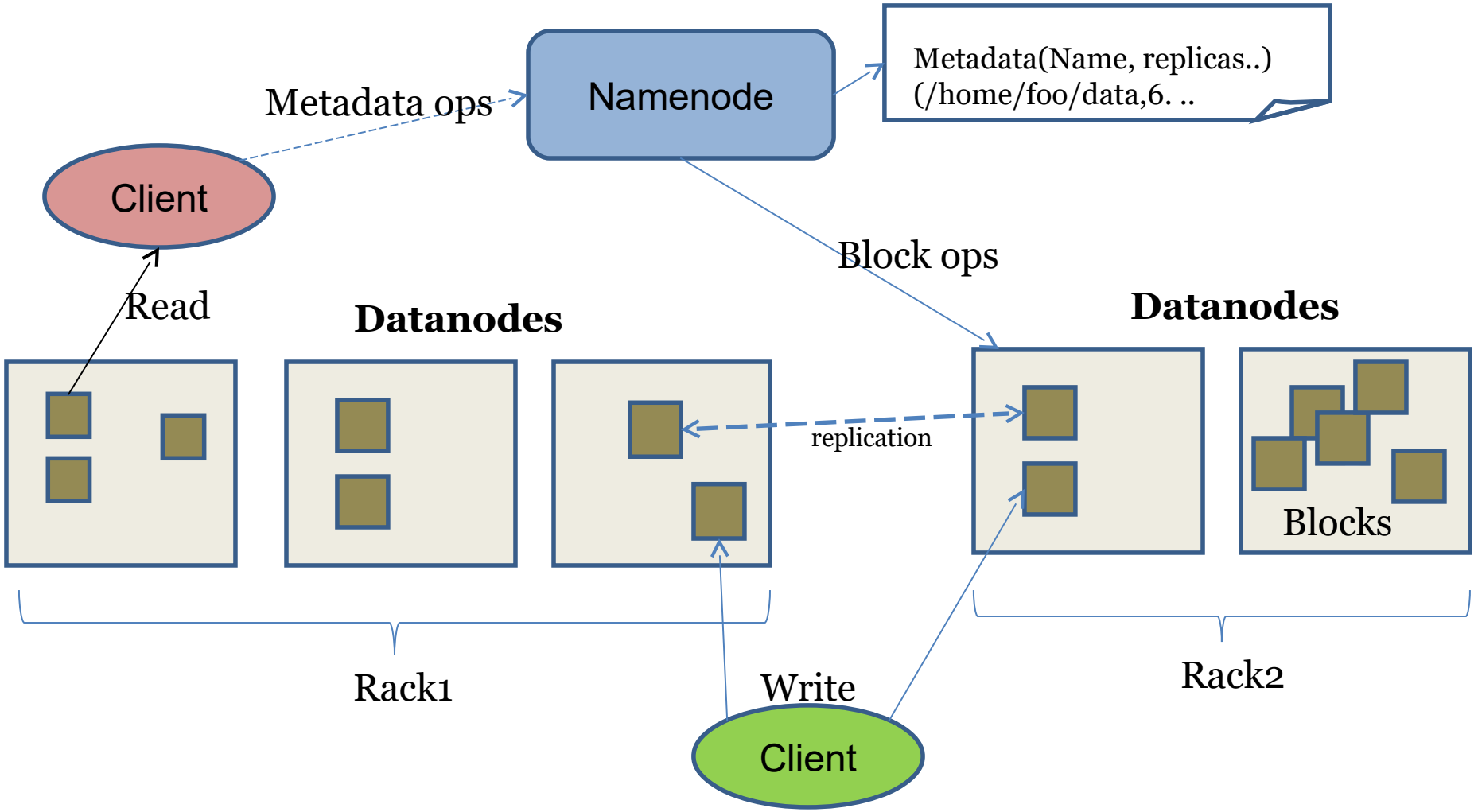
HDFS Storage

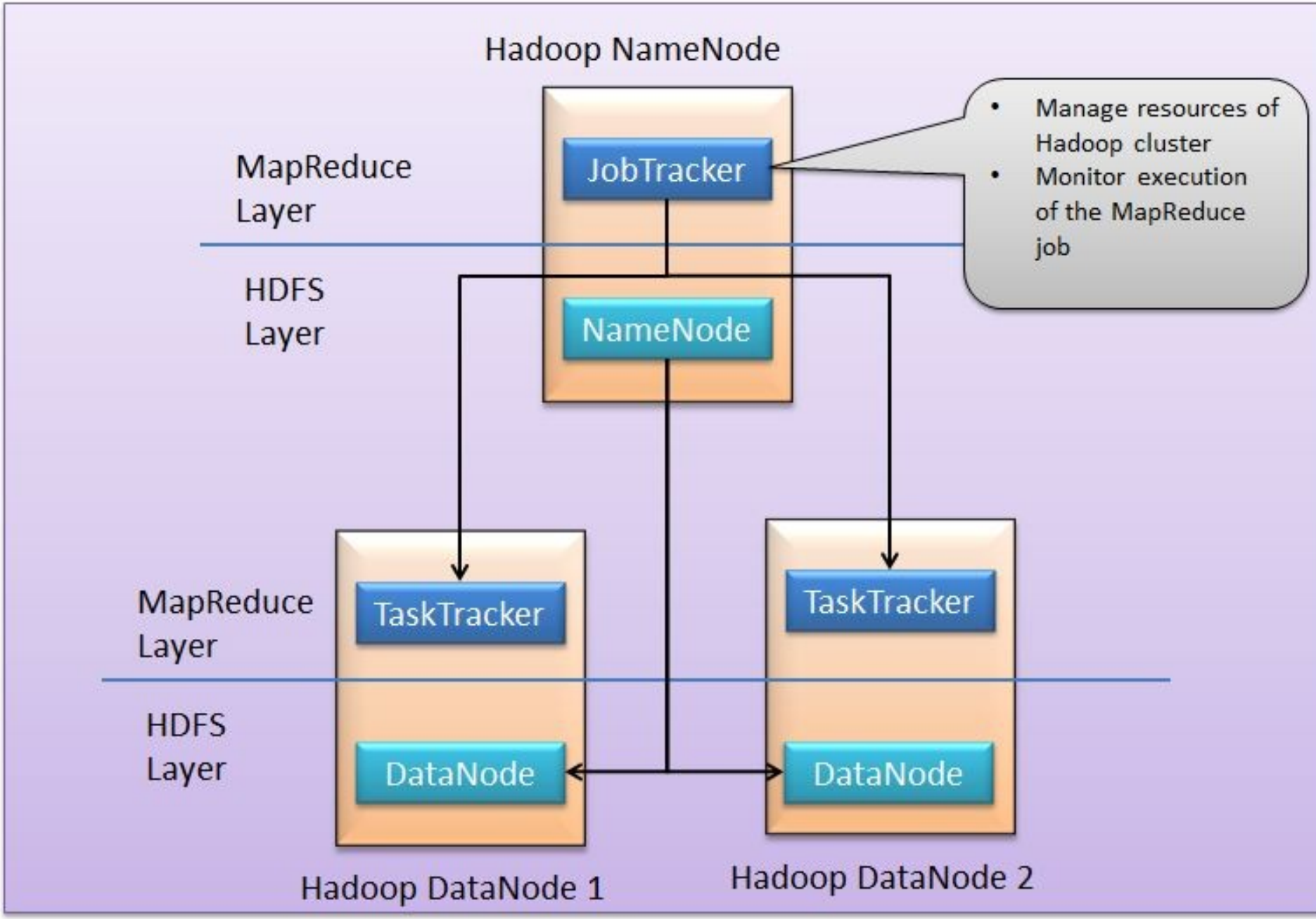
- HDFS Files are broken into Blocks
  - Basic until of reading/writing like disk blocks
  - Default to 64MB, may be larger in product env.
  - Make HDFS good for large file & high throughput
- Block may have multiple replicas
  - One block stored at multiple locations
  - Make HDFS storage fault tolerance

HDFS

- Master/slave cluster architecture
  - Consists of a single master called the NameNode
  - Consists of “chunk servers” called DataNodes
- Namenode
  - a master server that manages the file system namespace and regulates access to files by clients.
  - Maintains a mapping from file name to blocks & blocks to dataNodes
  - Clients contact the master to find where a particular block is located.
  - All further client communication goes to the dataNode where the blocks are stored.
- DataNodes
  - A file is split into one or more blocks and set of blocks are stored in DataNodes.
  - DataNodes: serves read, write requests, performs block creation, deletion, and replication upon instruction from Namenode

DFS Architecture




Hadoop NameNode
MapReduce Layer
HDFS Layer
JobTracker
NameNode
Manage resources of Hadoop cluster
Monitor execution of the MapReduce job
MapReduce Layer
HDFS Layer
TaskTracker
DataNode
TaskTracker
DataNode
Hadoop DataNode 1
Hadoop DataNode 2

Data Replication

- HDFS is designed to store very large files across machines in a large cluster.
- Each file is a sequence of blocks blocks are replicated for fault tolerance.
- The Namenode receives a Heartbeat and a BlockReport from each DataNode in the cluster.
- BlockReport contains all the blocks on a Datanode.

Replica Placement

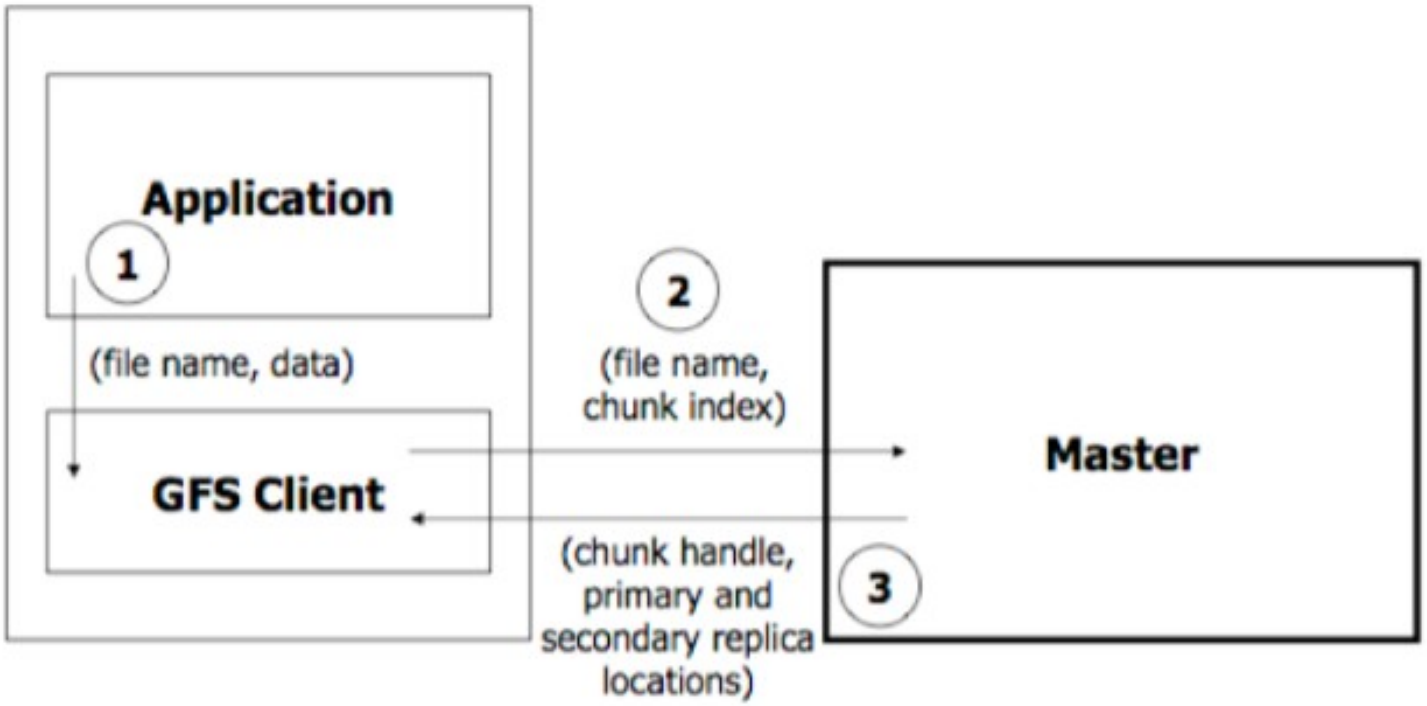
- Rack-aware replica placement:
  - Goal: improve reliability, availability and network bandwidth utilization
- Many racks, communication between racks are through switches.
- Network bandwidth between machines on the same rack is greater than those in different racks.
- Namenode determines the rack id for each DataNode.
- Replicas are placed: one on a node in a local rack, one on a different node in the local rack and one on a node in a different rack.
- 1/3 of the replica on a node, 2/3 on a rack and 1/3 distributed evenly across remaining racks.

Replica Selection

- Replica selection for READ operation: HDFS tries to minimize the bandwidth consumption and latency.
- If there is a replica on the Reader node then that is preferred.
- HDFS cluster may span multiple data centers: replica in the local data center is preferred over the remote one.

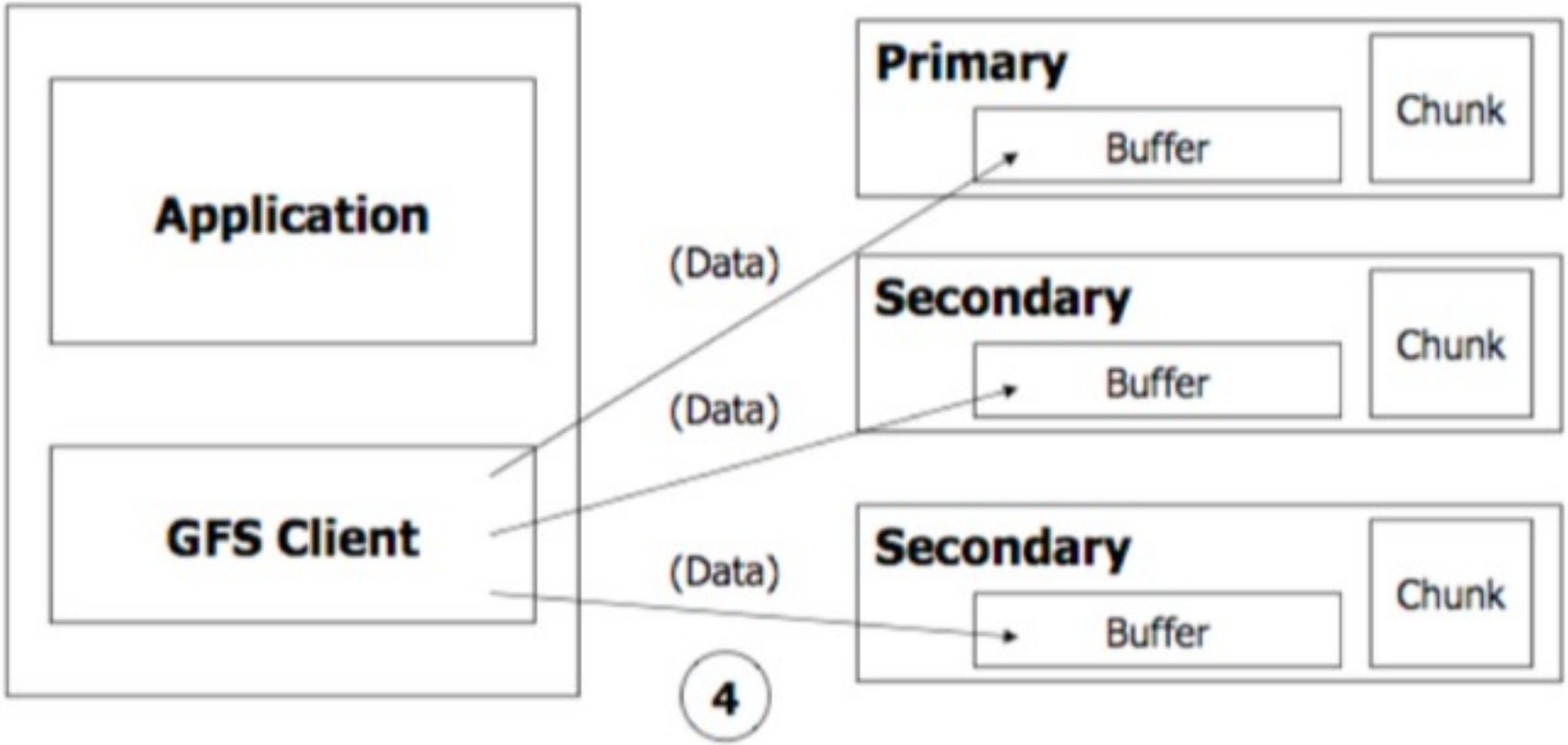
Client Write

1. Application originates the request
2. GFS client translates request and sends it to master
3. Master responds with chunk handle and replica locations



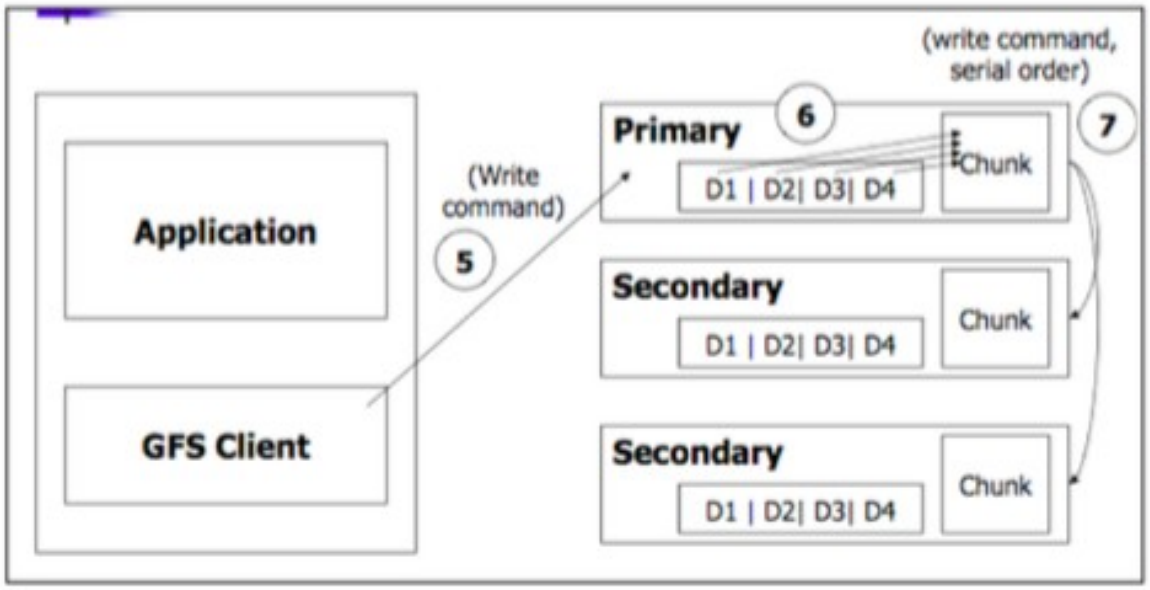
Client Write (Cont.)

4. Client pushes write data to all locations. Data is stored in chunkserver's internal buffers



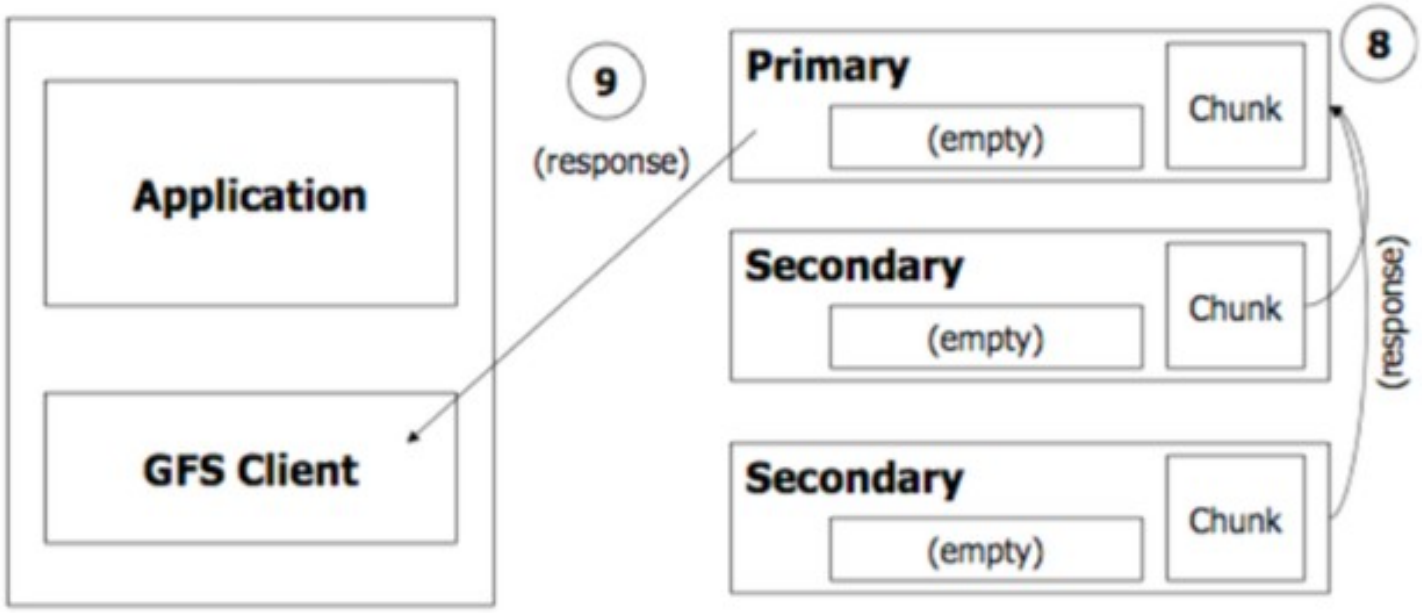
Client Write (Cont.)

5. Client sends write command to primary
6. Primary determines serial order for data instances in its buffer and writes the instances in that order to the chunk
7. Primary sends the serial order to the secondary and tells them to perform the write



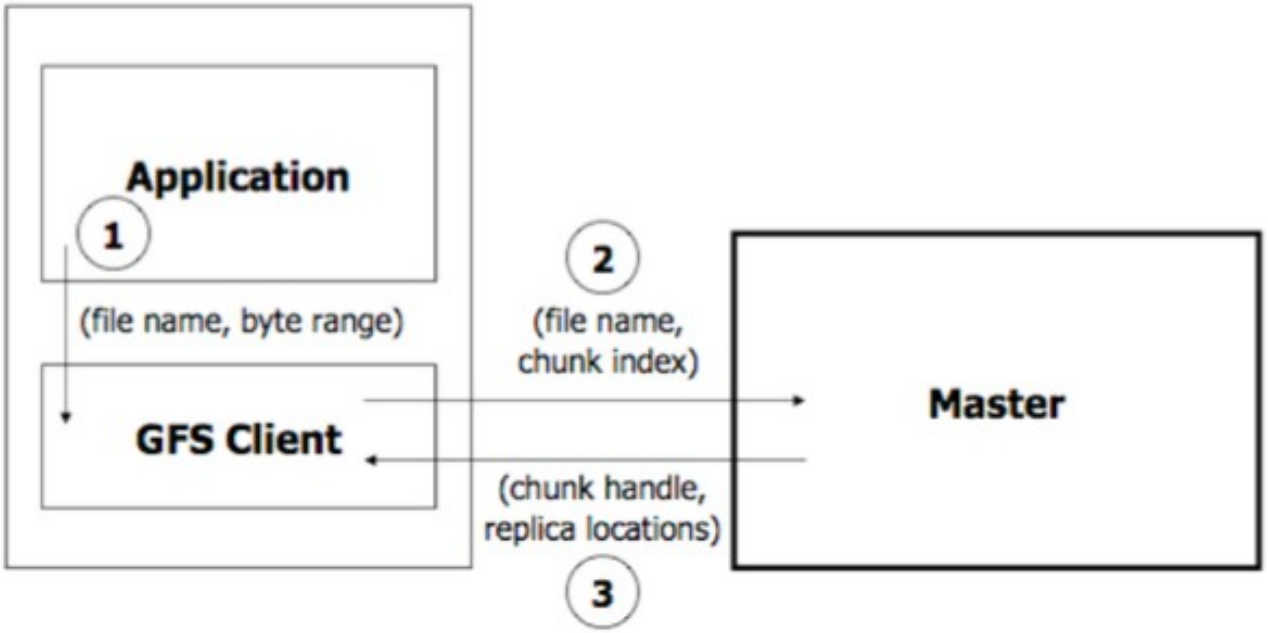
Client Write (Cont.)

8. Secondaries respond back to primary
9. Primary responds back to the client



Client Read

1. Application originates the read request
2. GFS client translates request and sends it to master
3. Master responds with chunk (block) handle and replica locations



Client Read (Cont.)

4. Client picks a location and sends the request
   - "Closest" determined by IP address on simple rack-based network topology
5. Chunkserver (Datanodes) sends requested data to the client
6. Client forwards the data to the application