Cloud Programming and Software Environments

Acknowledgement: Prof. Rajkumar Buyya for providing figures appear in this presentation

Introduction

- Commercial clouds need broad capabilities, these capabilities offer cost-effective utility computing with the elasticity to scale up and down in power. These include:
  - Physical or virtual computing platform
  - Massive data storage service, distributed file system
  - Massive database storage service
  - Massive data processing method and programming model

Introduction

- Workflow and data query language support
- Programming interface and service deployment
- Runtime support
- Support services

Infrastructure Cloud Features

- **Accounting:** Includes economies; clearly an active area for commercial clouds
- **Appliances:** Preconfigured virtual machine (VM) image supporting multifaceted tasks such as message-passing interface (MPI) clusters
- **Authentication and authorization:** Could need single sign-on to multiple systems

Infrastructure Cloud Features

- **Data transport:** Transports data between job components both between and within grids and clouds; exploits custom storage patterns as in BitTorrent
- **Operating systems:** Apple, Android, Linux, Windows
- **Program library:** Stores images and other program material
- **Registry:** Information resource for system (system version of metadata management)

Infrastructure Cloud Features

- **Security:** Security features other than basic authentication and authorization; includes higher level concepts such as trust
- **Scheduling:** Basic staple of Condor, Platform, Oracle Grid Engine, etc.; clouds have this implicitly as is especially clear with Azure Worker Role
- **Gang scheduling:** Assigns multiple (data-parallel) tasks in a scalable fashion; note that this is provided automatically by MapReduce

Infrastructure Cloud Features

- **Software as a Service (SaaS):** Shared between clouds and grids, and can be supported without special attention; Note use of services and corresponding service oriented architectures are very successful and are used in clouds very similarly to previous distributed systems.

Infrastructure Cloud Features

- **Virtualization:** Basic feature of clouds supporting elastic feature highlighted by Berkeley as characteristic of what defines a (public) cloud; includes virtual networking as in ViNe from University of Florida

Platform Features Supported by Clouds

- **Blob:** Basic storage concept typified by Azure Blob and Amazon S3
- **DPFS:** Support of file systems such as Google (MapReduce), HDFS (Hadoop), and Cosmos (Dryad) with compute-data affinity optimized for data processing
- **Fault tolerance:** A major feature of clouds

Platform Features Supported by Clouds

- **MapReduce:** Support MapReduce programming model including Hadoop on Linux, Dryad on Windows HPCS, and Twister on Windows and Linux. Include new associated languages such as Sawzall, Pregel, Pig Latin, and LINQ
- **Monitoring:** Can be based on publish-subscribe
- **Notification:** Basic function of publish-subscribe systems

Platform Features Supported by Clouds

- **Programming model:** Cloud programming models are built with other platform features and are related to familiar web and grid models
- **Queues:** Queuing system possibly based on publish-subscribe
- **Scalable synchronization:** Apache Zookeeper or Google Chubby. Supports distributed locks and used by BigTable.

Platform Features Supported by Clouds

- **SQL:** Relational database
- **Table:** Support of table data structures modeled on Apache Hbase or Amazon SimpleDB/Azure Table. Part of NOSQL movement
- **Web role:** Used in Azure to describe important link to user and can be supported otherwise with a portal framework. This is the main purpose of GAE

Platform Features Supported by Clouds

- **Worker role:** Implicitly used in both Amazon and grids but was first introduced as a high-level construct by Azure

Traditional Features Common to Grids and Clouds

- **Workflow**
- **Data Transport**
- **Security, Privacy, and Availability**

Technologies for Data-Intensive Computing

- Data-intensive computing concerns the development of applications that are mainly focused on processing large quantities of data
- Explosion of unstructured data in the form of blogs, web pages, software logs, and sensor readings.
- The relational model in its original formulation, does not seem to be the preferred solution for supporting data analytics at a large scale

Technologies for Data-Intensive Computing

- Growing of popularity of Big Data
- Growing importance of data analytics in the business chain
- Presence of data in several forms, not only structured
- New approaches and technologies for computing

High Performance Distributed & Parallel File Systems and Storages

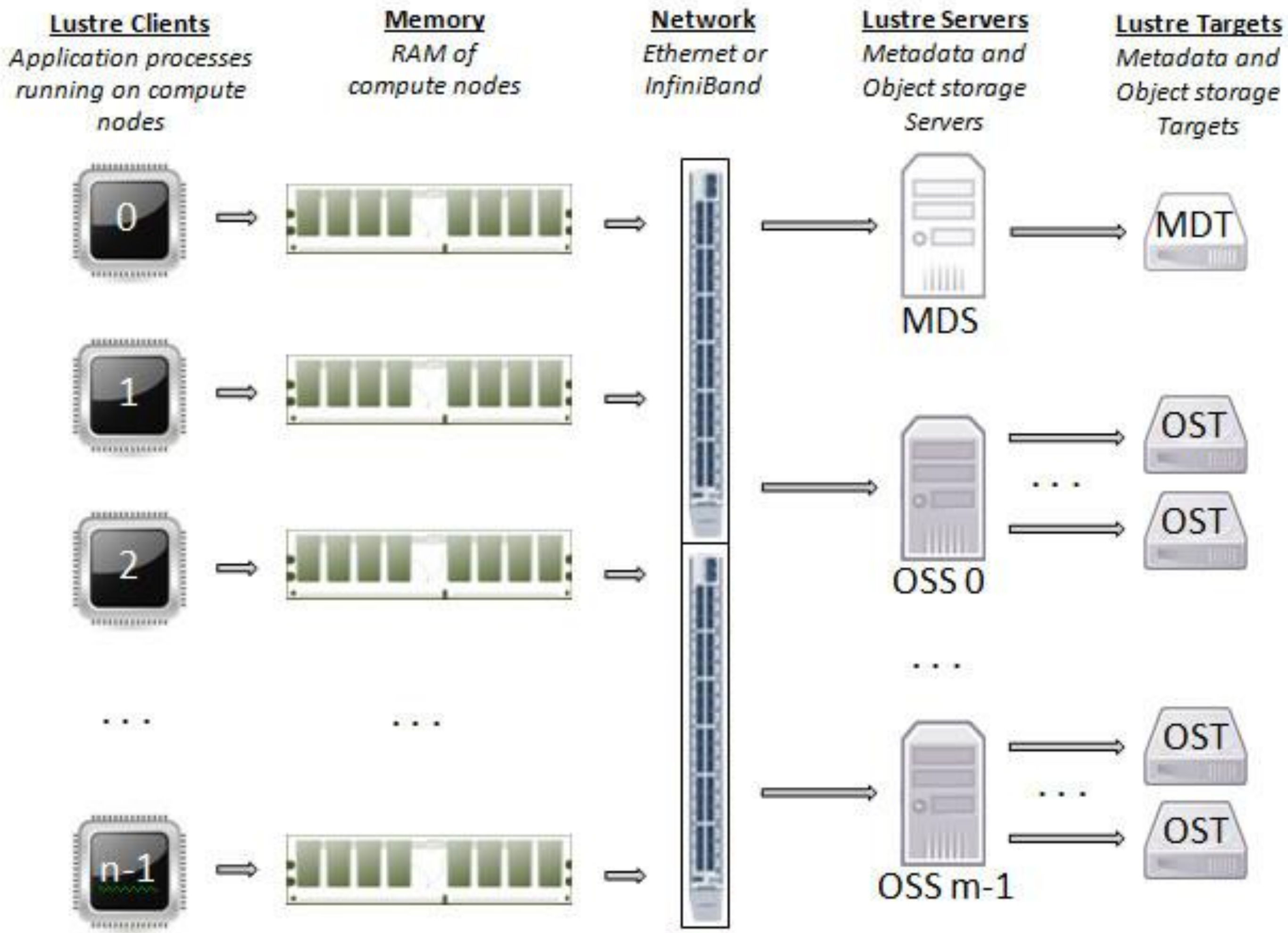
Lustre File System

- Lustre (Sun Microsystem now Oracle)
- A massively parallel distributed file system that covers the needs of a small workgroup of clusters to a large scale computing cluster.
- Lustre is designed to provide access to petabytes (PBs) of storage, to serve thousands of clients with an IO throughput of hundreds of gigabytes per second (GB/s).

Lustre File System

- The system is composed by a metadata server containing the metadata information about the file system and a collection of object storage servers that are in-charge of providing storage.
- Users access the file system via a POSIX compliant client, which can be either mounted as a module in the kernel or through a library.

Lustre File System



Lustre File System

- The Lustre file system is made up of an underlying set of I/O servers called **Object Storage Servers (OSSs)**
- disks called **Object Storage Targets (OSTs)**.
- The file metadata is controlled by a **Metadata Server (MDS)** and stored on a **Metadata Target (MDT)**.
- A single Lustre file system consists of one MDS and one MDT.

Lustre File System

- The file system implements a *robust failover strategy and recovery mechanism*, making server failures and recoveries transparent to clients.

General Parallel File System (GPFS)

- IBM General Parallel File System (GPFS)
- It is a high performance distributed file system developed providing support for RS/6000 supercomputer and Linux computing clusters.
- GPFS is a multi-platform distributed file system *built with advanced recovery mechanisms*.

General Parallel File System (GPFS)

- GPFS is built on the *concept of shared disks*, where a collection of disks is attached to the file systems nodes by means of some switching fabric.
- The file system makes this infrastructure transparent to users and *stripes large files over the disk array* also by *replicating* portion of the file in order to ensure *high availability*.

General Parallel File System (GPFS)

- By means of this infrastructure, the system is able to support petabytes of storage, which is accessed at a high throughput and without losing consistency of data.
- GPFS also distributes the metadata of the entire file system and provides transparent access to it, thus eliminating a single point of failure.

Google File System

- Google File System (GFS)
- GFS was built primarily as the fundamental storage service for Google’s search engine
- Google needed a distributed file system to *redundantly store massive amounts of data* on cheap and unreliable computers.
- In traditional file system design, there should be a clear interface between applications and the file system, such as a POSIX interface

Google File System (write)

- GFS typically will hold a large number of huge files, each 100 MB or larger. Thus, Google has chosen its file data block size to be 64 MB
- The I/O pattern in the Google application is also special. Files are typically written once, and the ***write*** *operations are often the appending data blocks to the end of files*.
- Multiple appending operations might be concurrent.

Google File System (read)

- There will be a lot of large streaming reads and only a little random access.
- *For large streaming reads, highly sustained throughput is given more important than low latency*.
- The workloads primarily consist of two kinds of reads: large streaming reads and small random reads.

Google File System

- Google made some special decisions regarding the design of GFS. *Reliability is achieved by using replications* (i.e., each chunk is replicated across *more than three* chunk servers).
- A single master coordinates access as well as keeps the metadata

Google File System

- There is no data cache in GFS as large streaming reads and writes represent neither time nor space locality.
- GFS provides a similar, but not identical, POSIX file system accessing interface.
- The architecture of the file system is organized into a *single master, containing the metadata of the entire file system*, and a collection of chunk servers, which provide storage space.

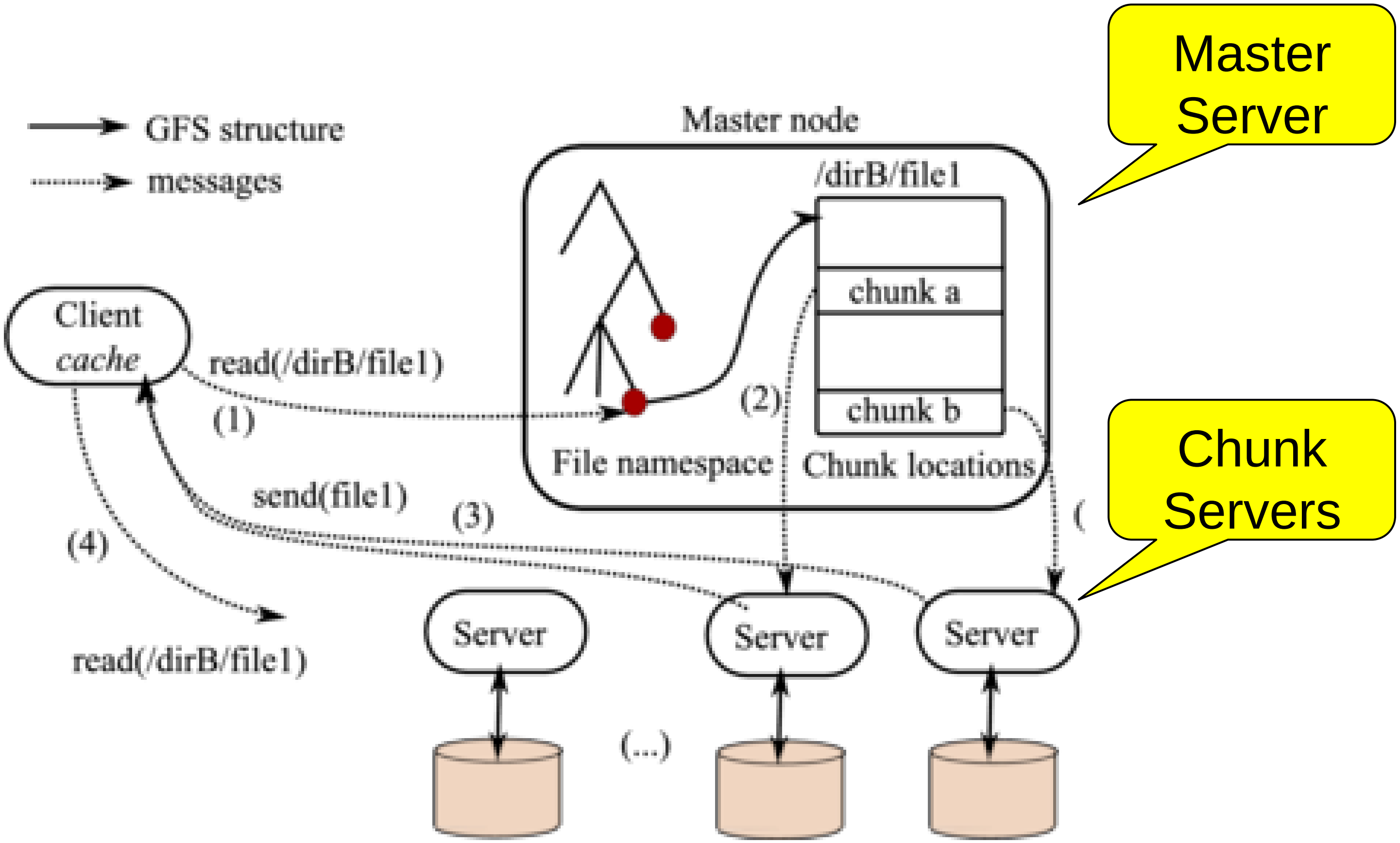
Google File System

- From a logical point of view the system is composed by a collection of software daemons, which *implement either the master server or the chunk server*.
- A file is a collection of chunks whose size can be configured at file system level.
- Chunks are replicated on multiple nodes in order to tolerate failures.

Google File System

- Clients look up the master server and identify the specific chunk of a file they want to access.
- Once the chunk is identified, the interaction happens between the client and the chunk server.

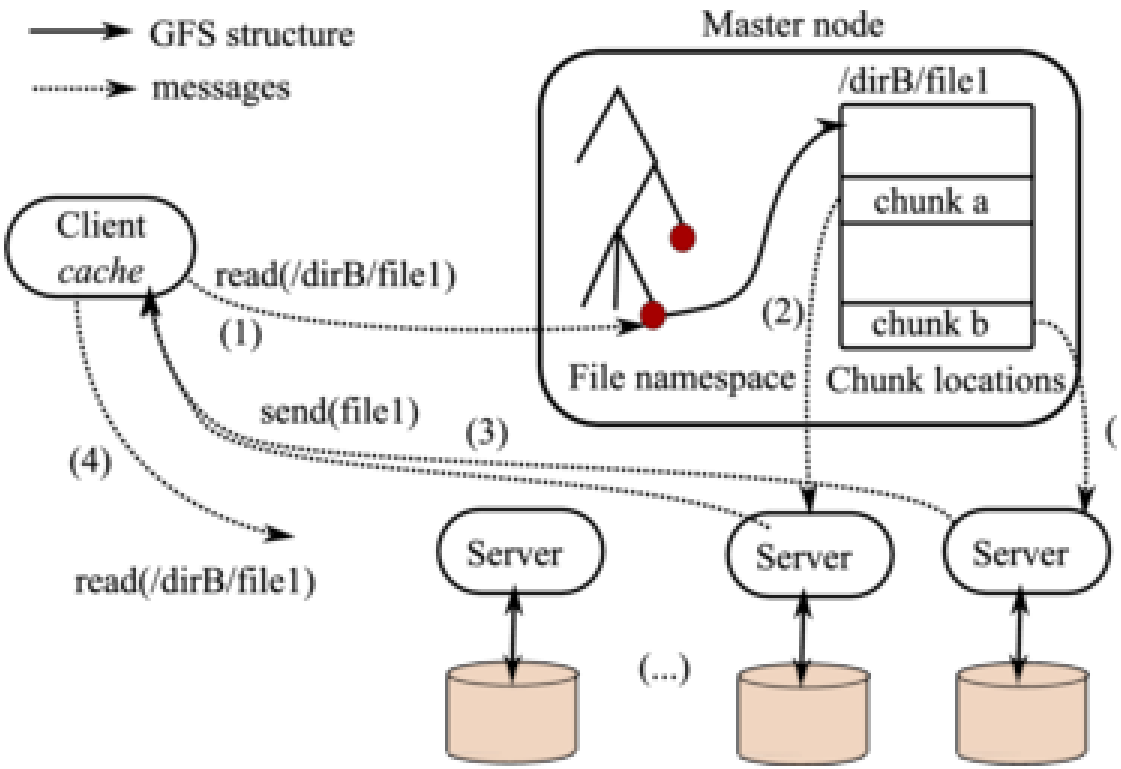
Architecture diagram of GFS



Google File System

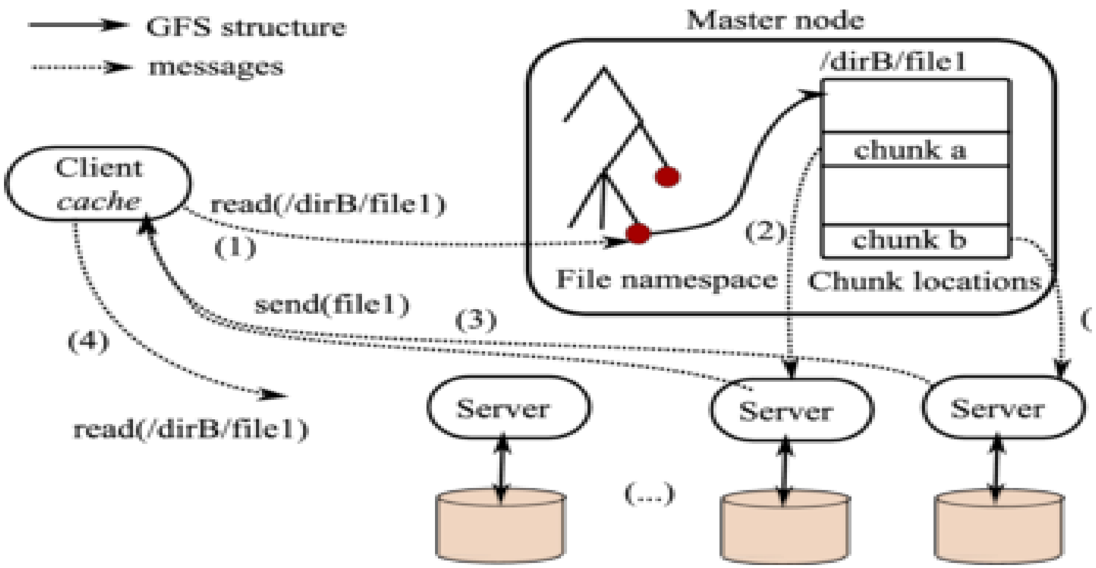
- A single-master architecture brings simplicity to the design of the system but gives rise to some *concern for its scalability and reliability*.
- The scalability concern is addressed by a *Client cache*, called *Client image* in the following way.
- Let us examine in detail how the system handles a *read()* request:

Architecture diagram of GFS



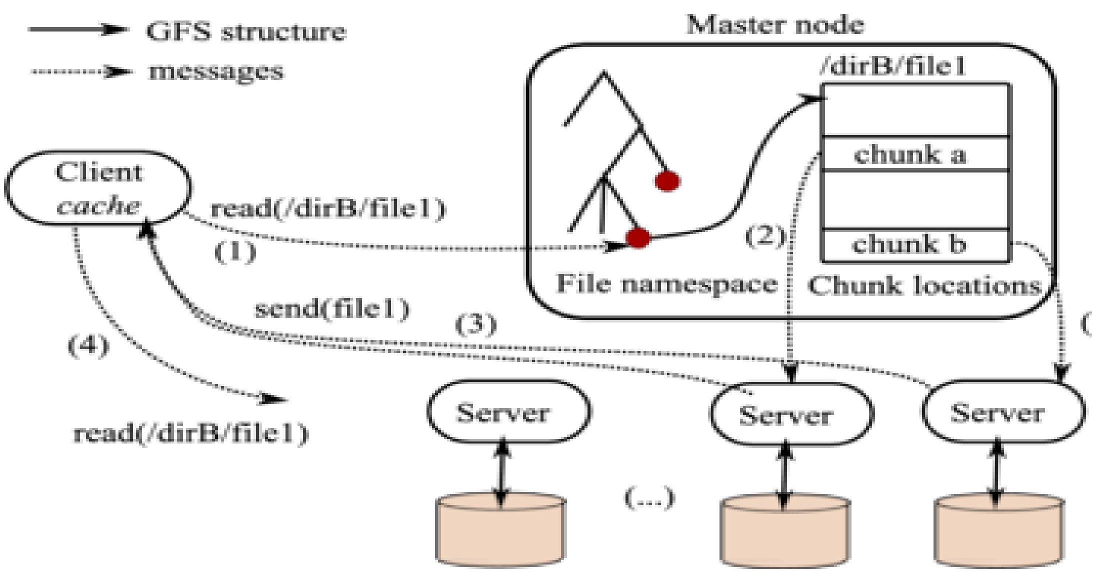
Google File System

1. The Client sends first *read(/dirB/file1)* request; since it knows nothing about the file distribution, the request is routed to the Master (1).



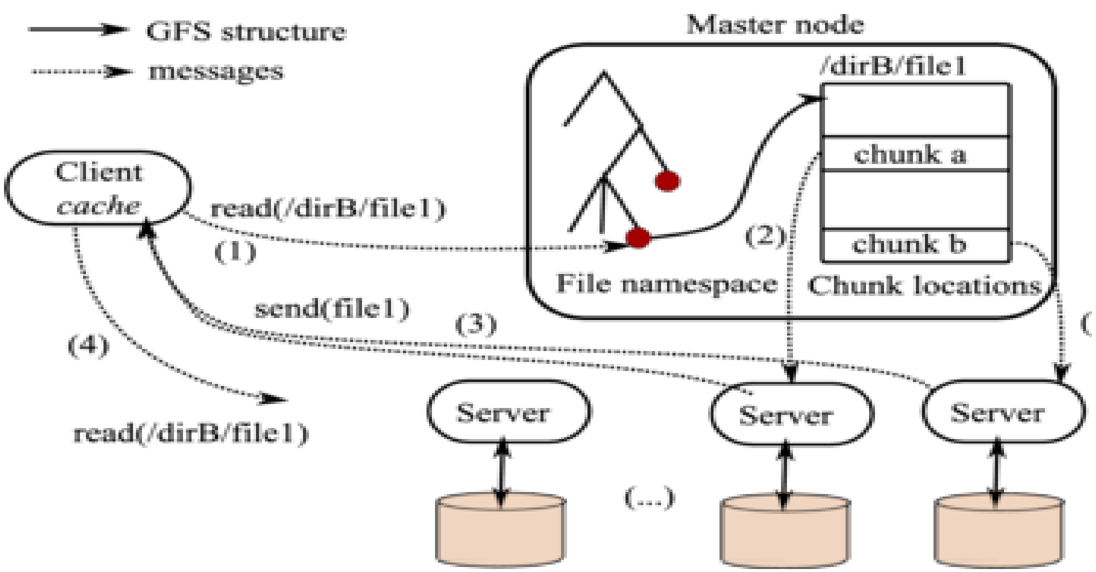
Google File System

2. The Master inspects the namespace and finds that *file1* is mapped to a list of chunks; their location is found in a local table (2).



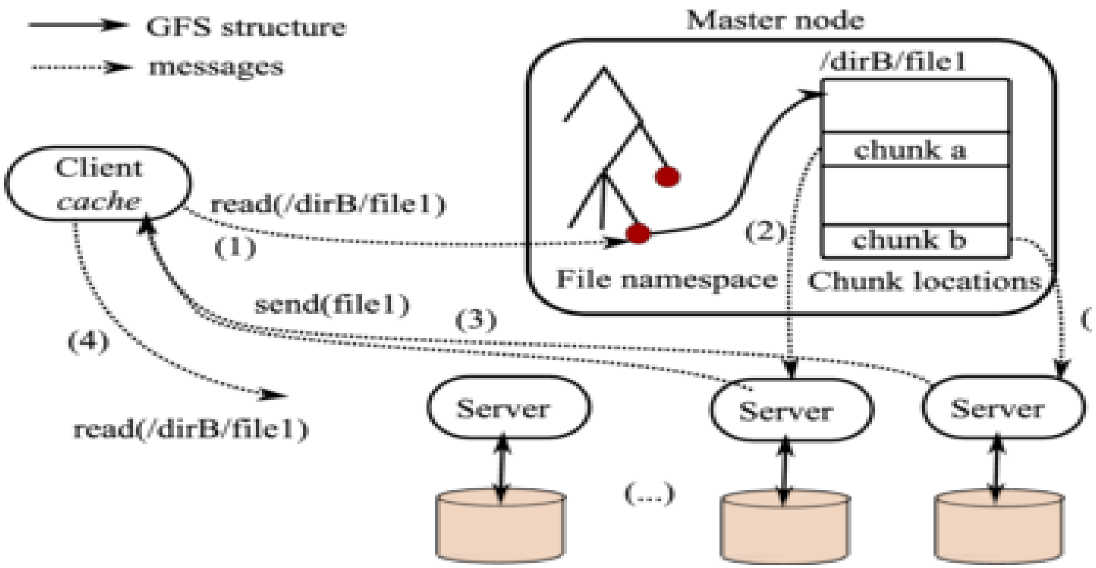
Google File System

3. Each server holding a chunk of *file1* is required to transmit this chunk to the Client (3).

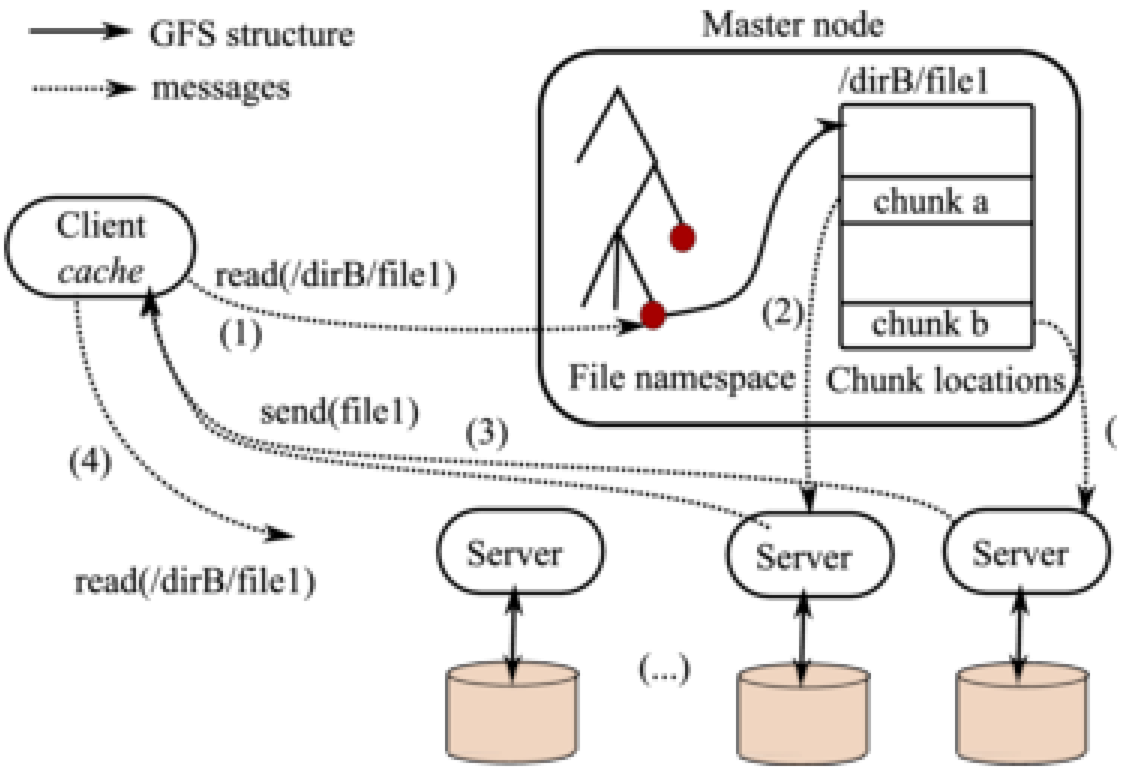


Google File System

4. The Client keeps in its cache the addresses of the nodes that serve *file1* (but *not* the file itself); this knowledge can be used for subsequent accesses to *file1*(4).



Architecture diagram of GFS



Google File System

- From the Client point of view, the distributed file system appears just like a directory hierarchy equipped with the usual Unix navigation (**chddir**, **ls**) and access (**read**, **write**) commands.

Hadoop Distributed File System (HDFS)

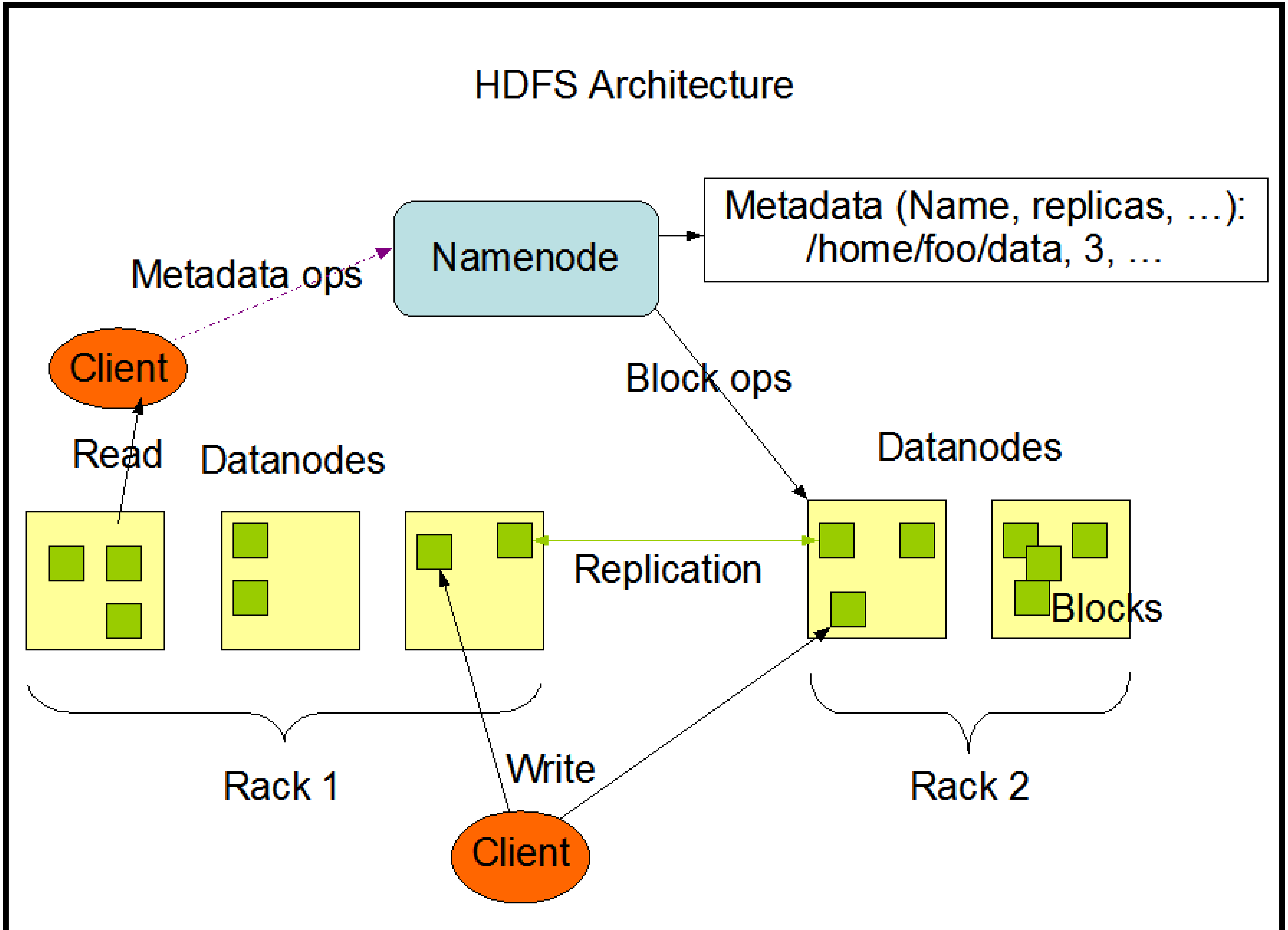
- **HDFS:** is a distributed file system inspired by GFS that organizes files and stores their data on a distributed computing system.
- **HDFS Architecture:** HDFS has a master/slave architecture containing a *single NameNode* as the master and a number of *DataNodes* as workers (slaves).

HDFS

- To store a file in this architecture, HDFS splits the file into fixed-size blocks (e.g., 64 MB) and stores them on workers (DataNodes).
- *The mapping of blocks to DataNodes is determined by the NameNode.*
- The NameNode (master) also manages the file system's metadata and namespace.

HDFS

- In such systems, the namespace is the area maintaining the metadata (location of input splits/blocks in all DataNodes)
- Each DataNode, usually one per node in a cluster, manages the storage attached to the node.
- Each DataNode is responsible for storing and retrieving its file blocks


HDFS Architecture
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Namenode
Metadata ops
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Rack 2
Client

HDFS

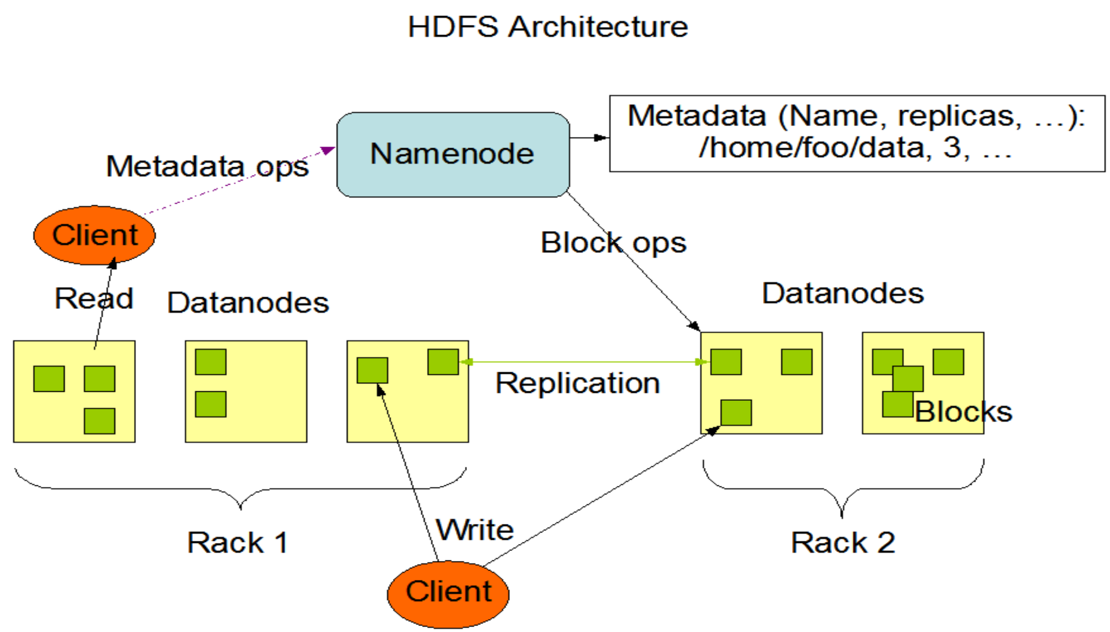
- **HDFS Fault Tolerance:** One of the main aspects of HDFS is its *fault tolerance characteristic.*
- Since Hadoop is designed to be deployed on low-cost hardware by default, a hardware failure in this system is considered to be common rather than an exception.

HDFS

- Therefore, Hadoop considers the following issues to fulfill reliability requirements of the file system
  - **Block replication:** The replication factor is set by the user and is *three by default*.
  - **Replica placement:** HDFS compromises its reliability to achieve lower communication costs. In the HDFS the default replication factor of **three**,
    - HDFS stores one replica in the same node the original data is stored,
    - one replica on a different node but in the same rack, and
    - one replica on a different node in a different rack to provide three copies of the data

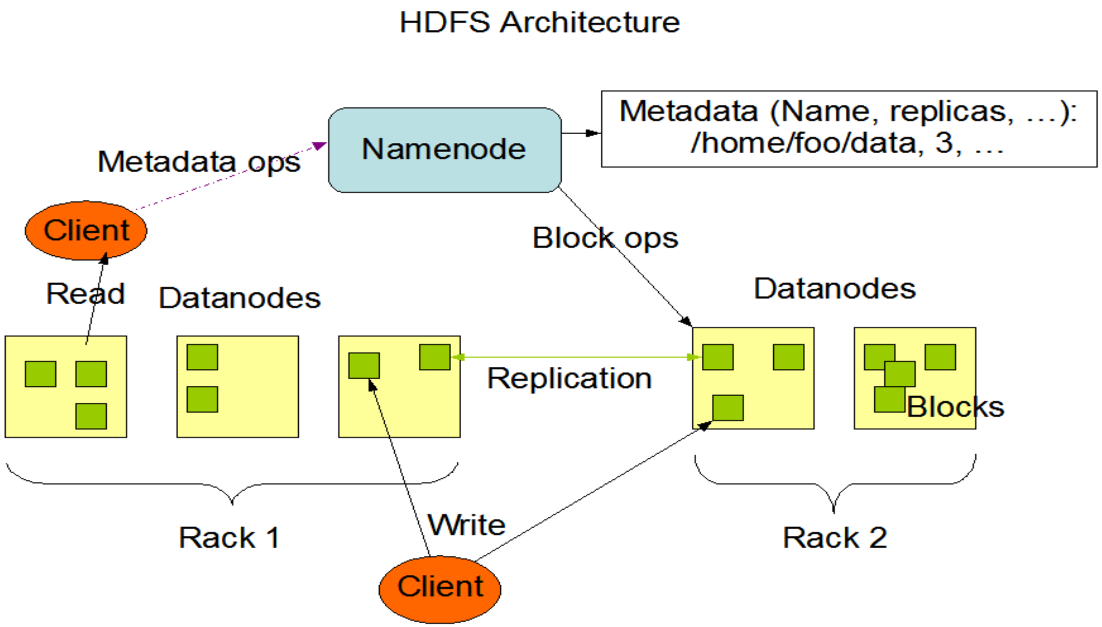
HDFS

- **Reading a file:** a user sends an open request to the NameNode to get the location of file blocks.



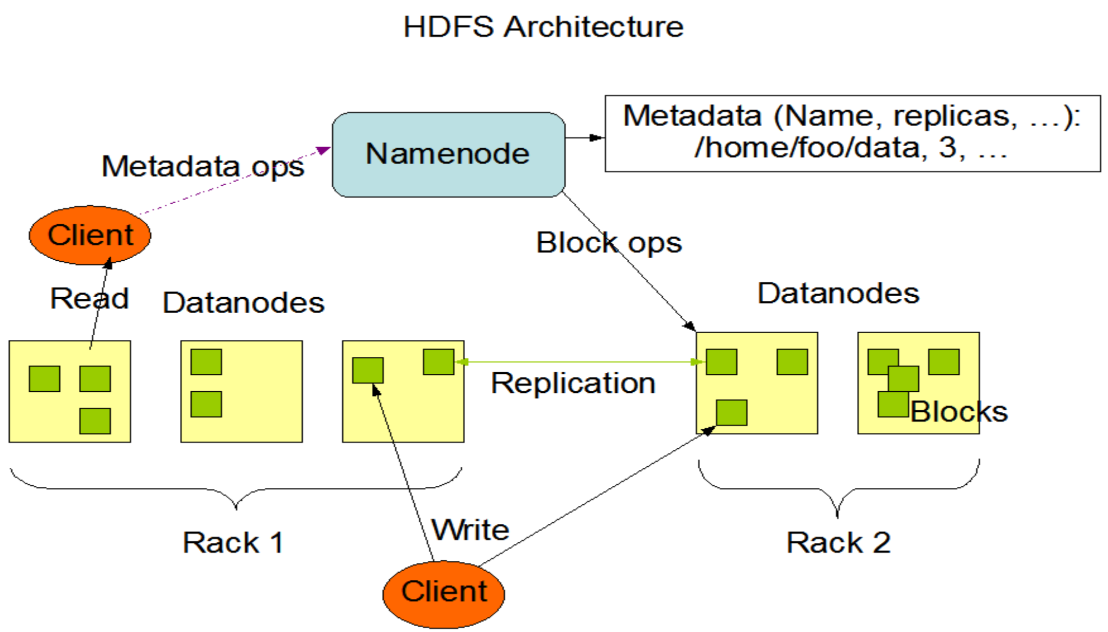
HDFS

- For each file block, the NameNode returns the address of a set of DataNodes containing replica information for the requested file.



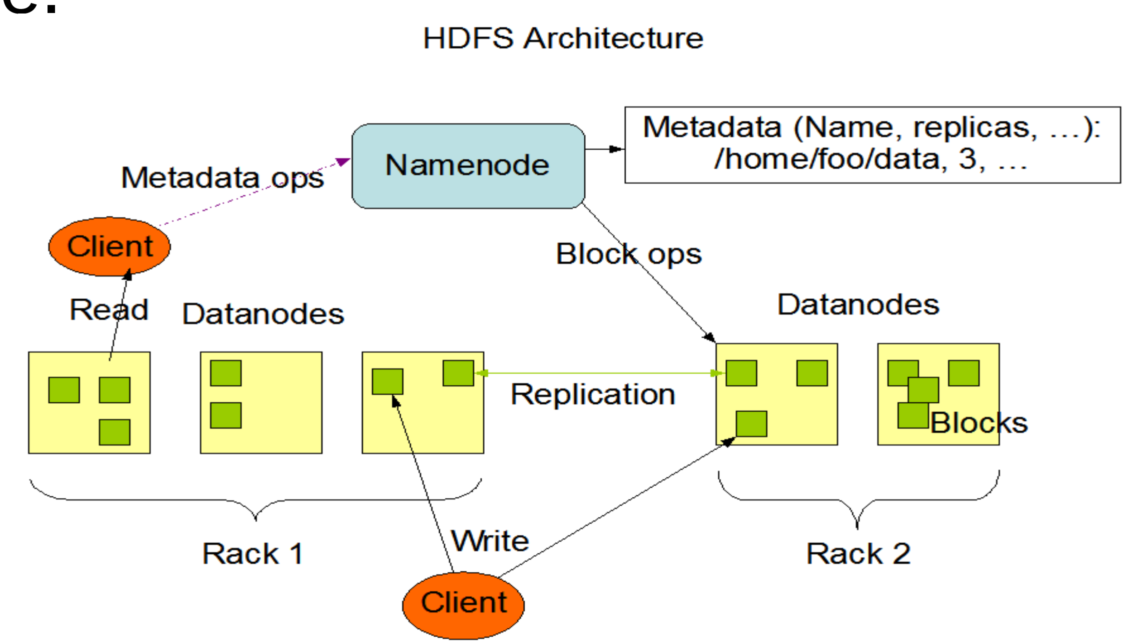
HDFS

- The number of addresses depends on the number of block replicas.



HDFS

- Upon receiving such information, the user calls the *read* function to connect to the closest DataNode containing the first block of the file.



HDFS

- After the first block is streamed from the respective DataNode to the user, the *established connection is terminated* and the same process is repeated for all blocks of the requested file until the whole file is streamed to the user.

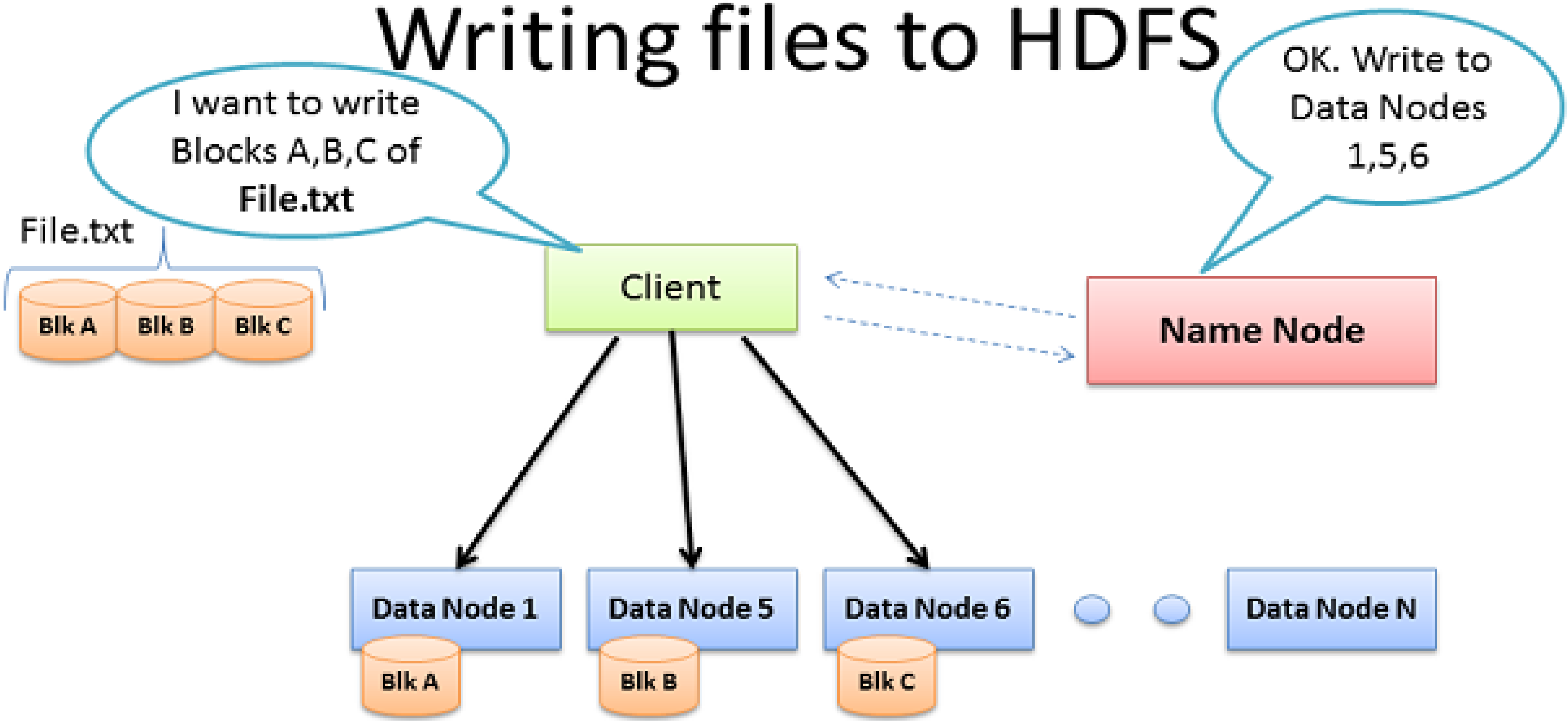
HDFS

- **Writing to a file:** a user sends a create request to the NameNode to create a new file in the file system namespace.
- If the file does not exist, the NameNode notifies the user and allows him to start writing data to the file by calling the *write* function.

HDFS

- The first block of the file is written to an internal queue termed the data queue while a data streamer monitors its writing into a DataNode.
- Since each file block needs to be replicated by a predefined factor, *the data streamer first sends a request to the NameNode to get a list of suitable DataNodes to store replicas of the first block.*

Writing files to HDFS



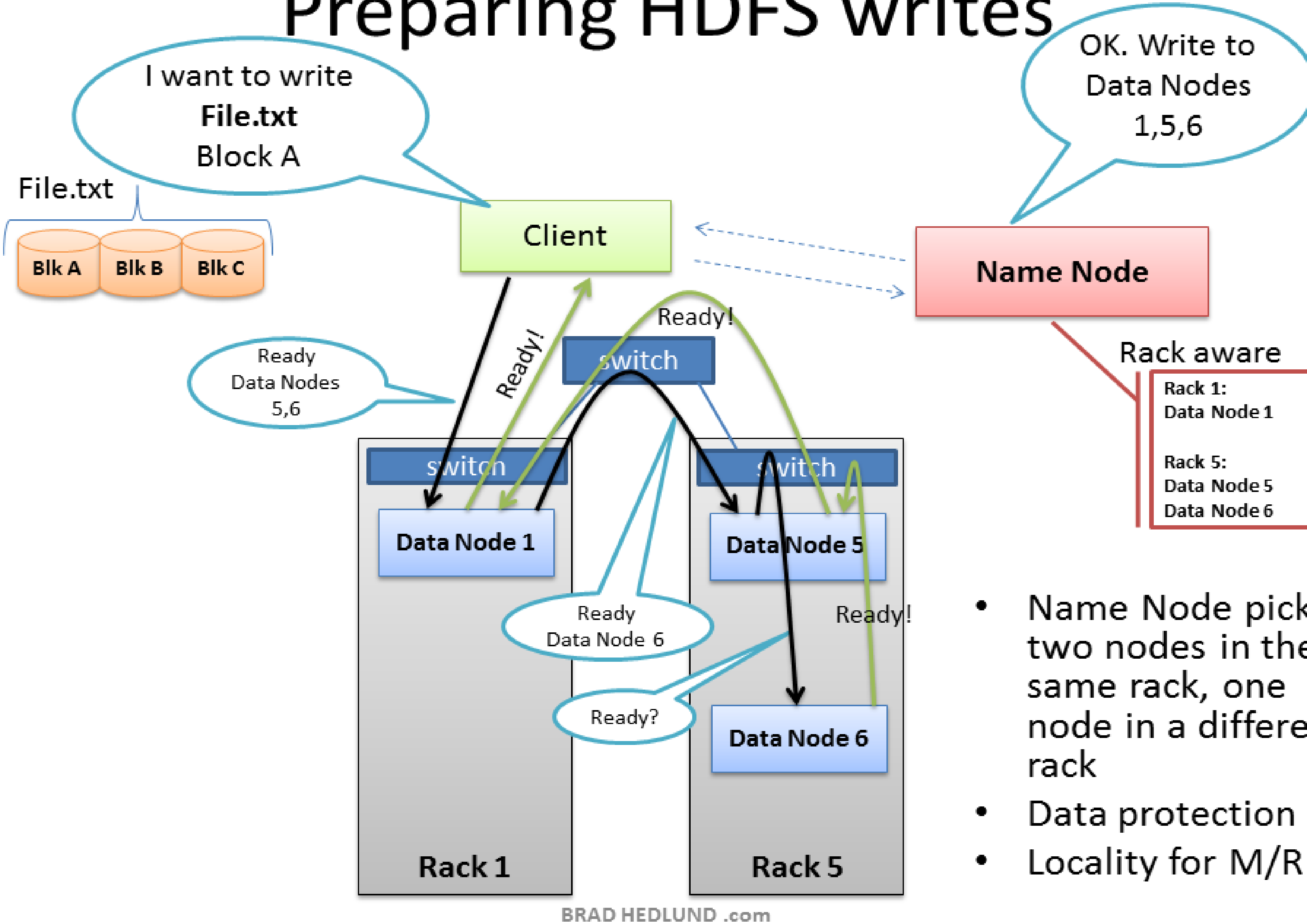
- Client consults Name Node
- Client writes block directly to one Data Node
- Data Nodes replicates block
- Cycle repeats for next block

BRAD HEDLUND .com

HDFS

- The steamer then stores the block in the first allocated DataNode. Afterward, *the block is forwarded to the second DataNode by the first DataNode*.
- The process continues until all allocated DataNodes receive a replica of the first block from the previous DataNode. Then process is repeated for other blocks.

Preparing HDFS writes

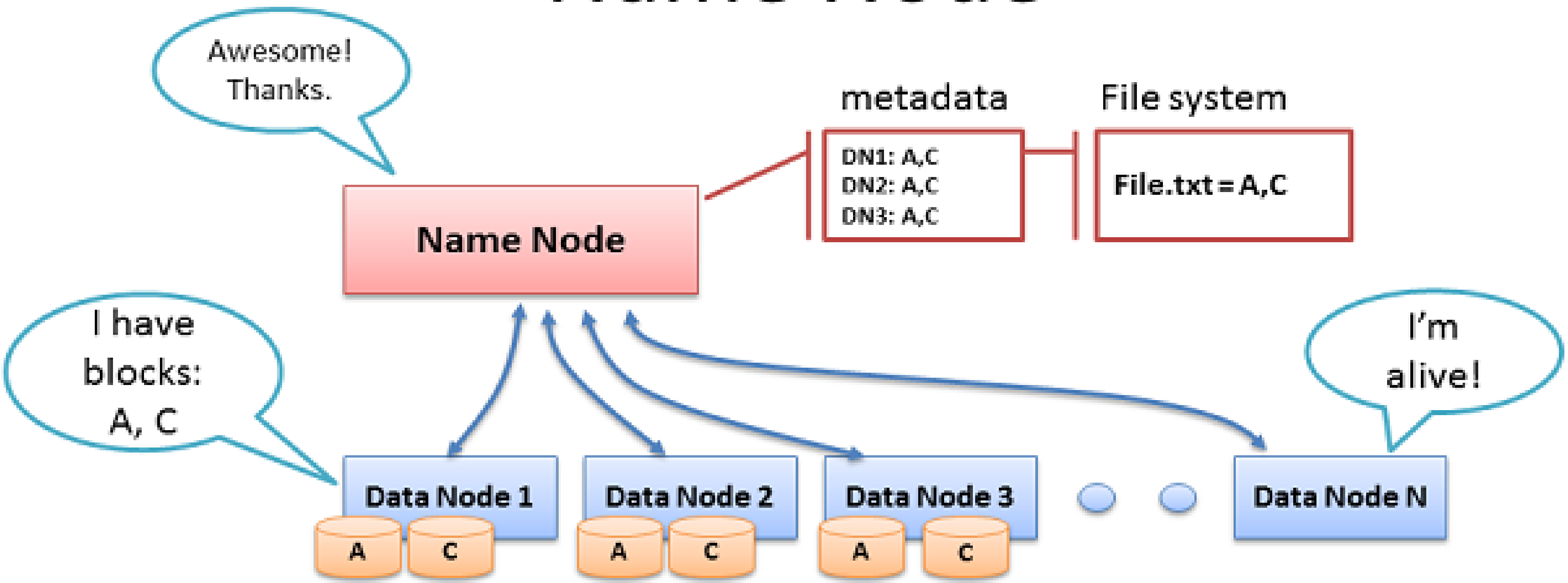


- Name Node picks two nodes in the same rack, one node in a different rack
- Data protection
- Locality for M/R

HDFS

- **Heartbeat and Blockreport messages:** are periodic messages sent to the NameNode by each DataNode in a cluster.
  - Receipt of a Heartbeat implies that the DataNode is functioning properly,
  - Blockreport contains a list of all blocks on a DataNode.
- The NameNode receives such messages because it is the sole decision maker of all replicas in the system.

Name Node



- Data Node sends Heartbeats
- Every 10^{th} heartbeat is a Block report
- Name Node builds metadata from Block reports
- TCP – every 3 seconds
- If Name Node is down, HDFS is down

BRAD HEDLUND .com

Differences from GFS

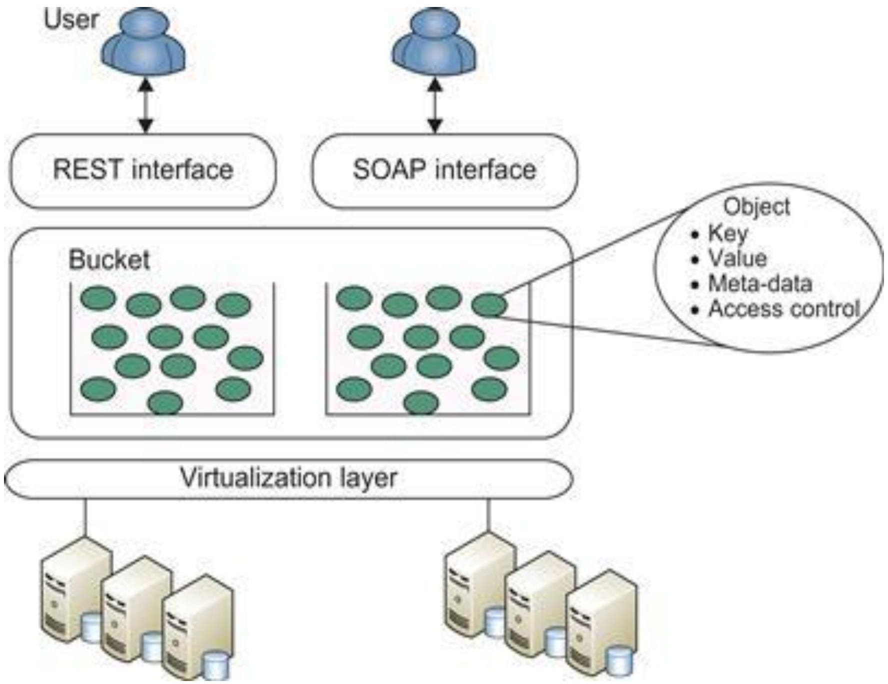
- Only single-writers per file.
  - No *record append* operation.
- Open source
  - Provides many interfaces and libraries for different file systems. Ex. S3, KFS, etc. Thrift IDE for C++, Python etc.

Amazon Simple Storage Service (S3)

- Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web.
- S3 provides the object-oriented storage service for users. Users can access their objects through SOAP/REST with either browsers or other client programs which support SOAP/REST.

Amazon Simple Storage Service (S3)

- SQS is responsible for ensuring a reliable message service between two processes, even if the receiver processes are not running.



Amazon Simple Storage Service (S3)

- The fundamental operation unit of S3 is called an *object*. Each object is stored in a *bucket* and retrieved via a unique, developer-assigned key.
- Besides unique key attributes, the object has other attributes such as values, metadata, and access control information.

Amazon Simple Storage Service (S3)

- From the programmer's perspective, the storage provided by S3 can be viewed as a very *coarse-grained key-value pair*.
- Through the key-value programming interface, users can write, read, and delete objects containing from 1 byte to 5 gigabytes of data each.

Amazon Simple Storage Service (S3)

- There are two types of web service interface for the user to access the data stored in Amazon clouds.
- One is a REST (web 2.0) interface, and the other is a SOAP interface.

Programming Platforms

Programming Platforms

- Platforms for programming data intensive applications provide abstractions
- Help us to express the computation over a large quantity of information, and runtime systems are able to manage huge volumes of data efficiently

MapReduce

- *MapReduce*, as introduced, is a software framework which supports parallel and distributed computing on large data sets.
- This software framework abstracts the data flow of running a parallel program on a distributed computing system by providing users with two interfaces in the form of two functions: *Map* and *Reduce* (originally from Functional Programming)

MapReduce

- Users can override these two functions to interact with and manipulate the data flow of running their programs.
- In this framework, the *value* part of the data, *(key, value),* is the *actual data*, and the *key part* is only used by the MapReduce controller to *control the data flow*.

logical data flow from the *Map* to the *Reduce*

```
Map Function
 {

 }
Reduce Function
 {

 }
Main Function
 {
  Initialize Spec object

  MapReduce (Spec, & Results)
}
```

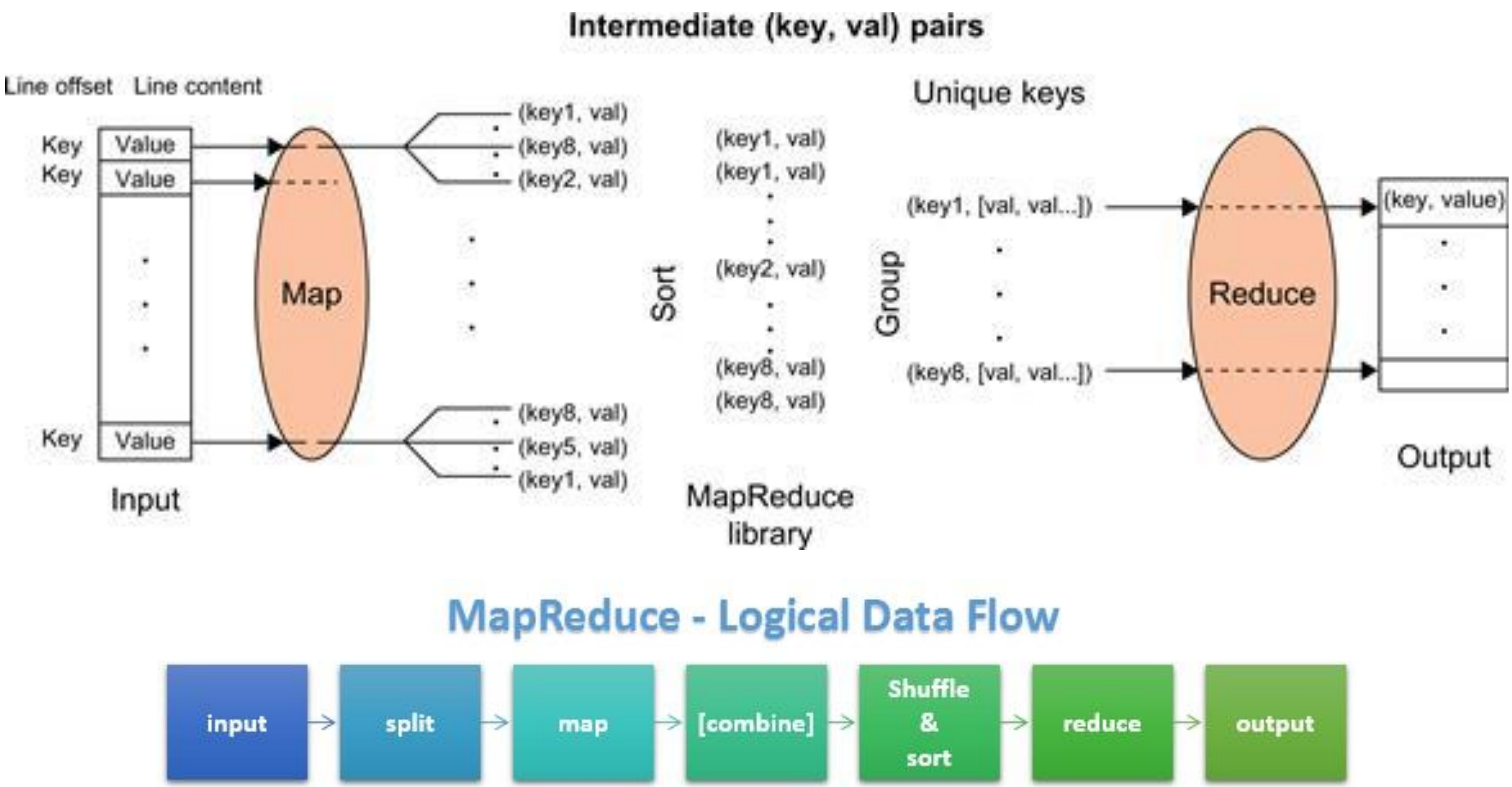
MapReduce Logical Data Flow

- The input data to both the *Map* and the *Reduce* functions has a particular structure. This also pertains for the output data.
- The input data to the *Map* function is in the form of a (key, value) pair. For example, the key is the line offset within the input file and the value is the content of the line.
- The output data from the *Map* function is structured as (key, value) pairs called *intermediate (key, value) pairs*.

MapReduce Logical Data Flow

- In other words, the user-defined *Map* function processes each input (key, value) pair and produces a number of (zero, one, or more) intermediate (key, value) pairs.
- Here, the goal is to process all input (key, value) pairs to the *Map* function in parallel

MapReduce logical data flow stages



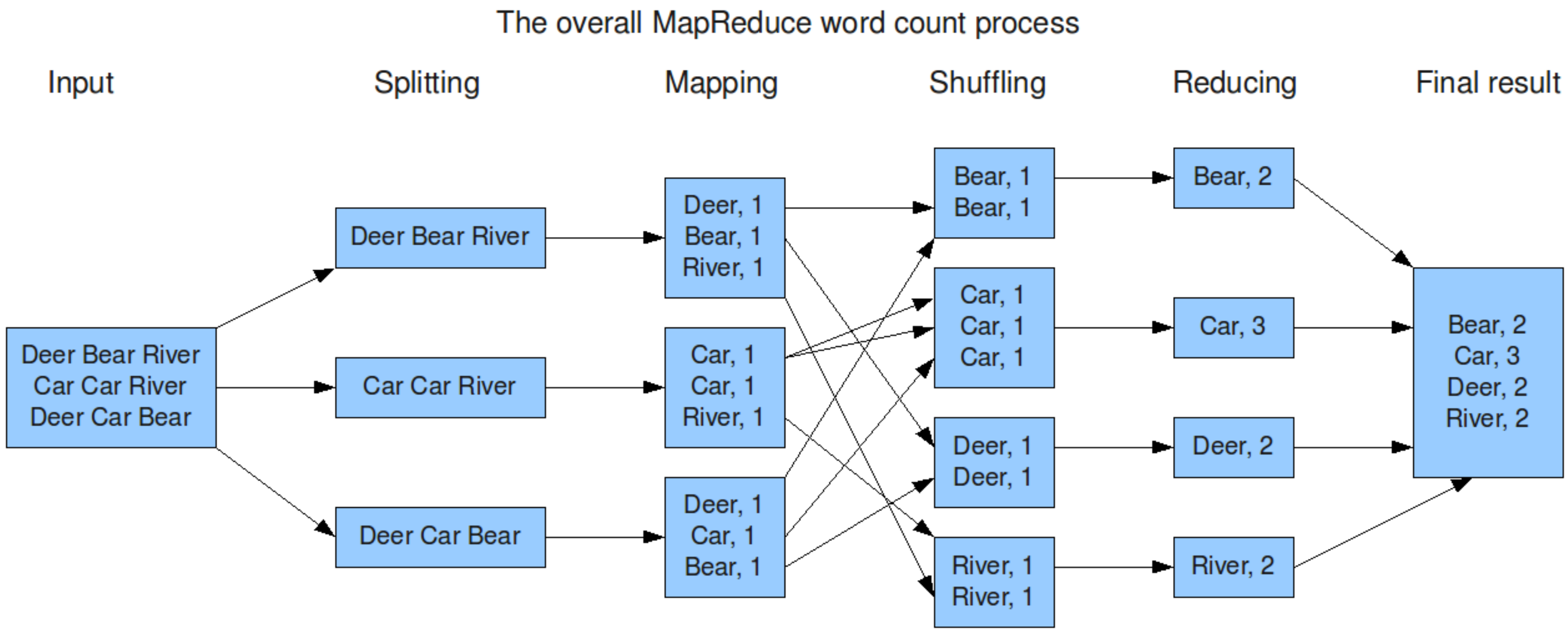
MapReduce

- The *Reduce* function receives the intermediate (key, value) pairs in the form of a group of intermediate values associated with one intermediate key, *(key, [set of values])*.
- In fact, the *MapReduce* framework forms these groups by first sorting the intermediate (key, value) pairs and then grouping values with the same key.

MapReduce

- It should be noted that the data is sorted to simplify the grouping process.
- The *Reduce* function processes each (key, [set of values]) group and produces a set of (key, value) pairs as output.

MapReduce



The overall MapReduce word count process
Input
Splitting
Mapping
Shuffling
Reducing
Final result
Deer Bear River
Car Car River
Deer Car Bear
Deer Bear River
Car Car River
Deer Car Bear
Deer, 1
Bear, 1
River, 1
Car, 1
Car, 1
River, 1
Deer, 1
Car, 1
Bear, 1
Bear, 1
Bear, 1
Car, 1
Car, 1
Car, 1
Deer, 1
Deer, 1
River, 1
River, 1
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

Formal Notation of MapReduce Data Flow

- The *Map* function is applied in parallel to every input (key, value) pair, and produces new set of intermediate (key, value) pairs as follows:

$$(key_1, val_1) \xrightarrow{Map\ Function} List\ (key_2, val_2)$$

Formal Notation of MapReduce Data Flow

- Then the MapReduce library collects all the produced intermediate (key, value) pairs from all input (key, value) pairs, and sorts them based on the key part.
- It then groups the values of all occurrences of the same key. Finally, the *Reduce* function is applied in parallel to each group producing the collection of values as output as illustrated here:

$$(key_2, List(val_2)) \xrightarrow{Reduce\ Function} List\ (val_2)$$

Formal Notation of MapReduce Data Flow

- The main responsibility of the *MapReduce* framework is to efficiently run a user's program on a distributed computing system.
- *MapReduce* framework meticulously handles all partitioning, mapping, synchronization, communication, and scheduling details of such data flows

Formal Notation of MapReduce Data Flow

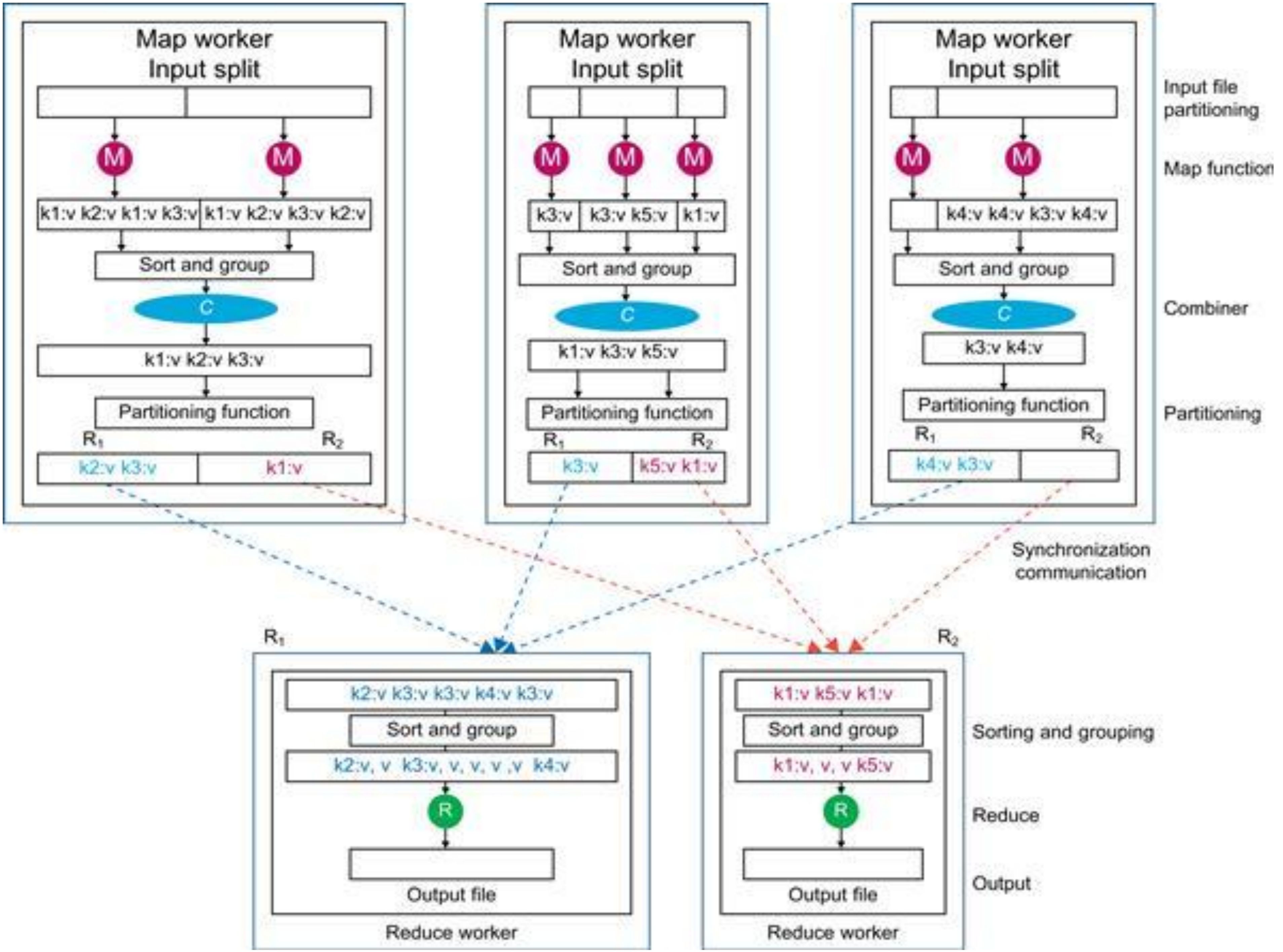
- **Problem 1:** Counting the number of occurrences of words having the same size, or the same number of letters, in a collection of documents. What is unique Key? What is intermediate value?

Solution: unique key: each word,
intermediate value: size of the word

Formal Notation of MapReduce Data Flow

- **Problem 2:** Counting the number of occurrences of anagrams in a collection of documents. Anagrams are words with the same set of letters but in a different order (e.g., the words listen and silent)
- What is unique Key? What is intermediate value?

Solution: unique key: alphabetically sorted sequence of letters for each word, intermediate value: number of occurrences



Map worker
Input split
k1:v k2:v k1:v k3:v
k1:v k2:v k3:v k2:v
Sort and group
C
k1:v k2:v k3:v
Partitioning function
R1
R2
k2:v k3:v
k1:v
Map worker
Input split
k3:v
k3:v k5:v
k1:v
Sort and group
C
k1:v k3:v k5:v
Partitioning function
R1
R2
k3:v
k5:v k1:v
Map worker
Input split
k4:v k4:v k3:v k4:v
Sort and group
C
k3:v k4:v
Partitioning function
R1
R2
k4:v k3:v
Input file partitioning
Map function
Combiner
Partitioning
Synchronization communication
R1
k2:v k3:v k3:v k4:v k3:v
Sort and group
k2:v, v k3:v, v, v, v ,v k4:v
R
Output file
Reduce worker
R2
k1:v k5:v k1:v
Sort and group
k1:v, v, v k5:v
R
Output file
Reduce worker
Sorting and grouping
Reduce
Output

MapReduce framework – Step 1

- **Data partitioning** The MapReduce library splits the input data (files), already stored in GFS, into M pieces that also *correspond to the number of map tasks*.

MapReduce framework– Step 2

- **Computation partitioning** is handled by allowing users to write their programs in the form of the *Map* and *Reduce* functions.
- Therefore, the MapReduce library only generates copies of a user program (e.g., by a fork system call) containing the *Map* and the *Reduce* functions, distributes them, and starts them up on a number of available computation engines.

MapReduce framework– Step 3

- **Determining the master and workers** one of the copies of the user program becomes the master and the rest become workers. The master picks idle workers, and assigns the map and reduce tasks to them.
- A map/reduce *worker* is typically a computation engine such as a cluster node to run map/reduce *tasks* by executing *Map/Reduce functions*.

MapReduce framework– Step 4

- **Reading the input data (data distribution)** Each *map worker* reads its corresponding portion of the input data, namely the input data split, and sends it to its *Map* function.
- Although a map worker may run more than one *Map* function, which means it has been assigned more than one input data split, each worker is usually assigned one input split only.

MapReduce framework– Step 5

- ***Map* function** Each *Map* function receives the input data split as a set of (key, value) pairs to process and produce the intermediated (key, value) pairs.

MapReduce framework– Step 6

- ***Combiner* function** This is an optional local function within the map worker which applies to intermediate (key, value) pairs. The user can invoke the *Combiner* function inside the user program.
- The *Combiner* function runs the same code written by users for the *Reduce* function as its functionality is identical to it. The *Combiner* function merges the local data of each map worker before sending it over the network to effectively reduce its communication costs.

MapReduce framework– Step 7

- ***Partitioning* function** the intermediate (key, value) pairs with identical keys are grouped together because all values inside each group should be processed by only one *Reduce* function to generate the final result.

MapReduce framework– Step 7 Conti…

- However, in real implementations, since there are *M* map and *R* reduce tasks, intermediate (key, value) pairs with the same key might be produced by different map tasks, although they should be grouped and processed together by one *Reduce* function only.

MapReduce framework– Step 7 Conti…

- Therefore, the intermediate (key, value) pairs produced by each map worker are partitioned into *R* regions, equal to the number of reduce tasks, by the *Partitioning* function to guarantee that all (key, value) pairs with identical keys are stored in the same region.

MapReduce framework– Step 7 Conti…

- As a result, since reduce worker *i* reads the data of region *i* of all map workers, all (key, value) pairs with the same key will be gathered by reduce worker *i* accordingly.

MapReduce framework– Step 7 Conti…

- To implement this technique, a *Partitioning* function could simply be a hash function (e.g., *Hash*(*key*) *mod R*) that forwards the data into particular regions.
- The locations of the buffered data in these *R* partitions are sent to the master for later forwarding of data to the reduce workers.

MapReduce framework– Step 8

- **Synchronization** MapReduce applies a simple synchronization policy to coordinate map workers with reduce workers, in which the communication between them starts when all map tasks finish.

MapReduce framework– Step 9

- **Communication** Reduce worker *i*, already notified of the location of region *i* of all map workers, uses a remote procedure call to read the data from the respective region of all map workers.
- Since all reduce workers read the data from all map workers, *all-to-all communication* among all map and reduce workers, which incurs network congestion, occurs in the network.

MapReduce framework– Step 9 Conti…

- This issue is one of the major bottlenecks in increasing the performance of such systems.
- A data transfer module was proposed to schedule data transfers independently.

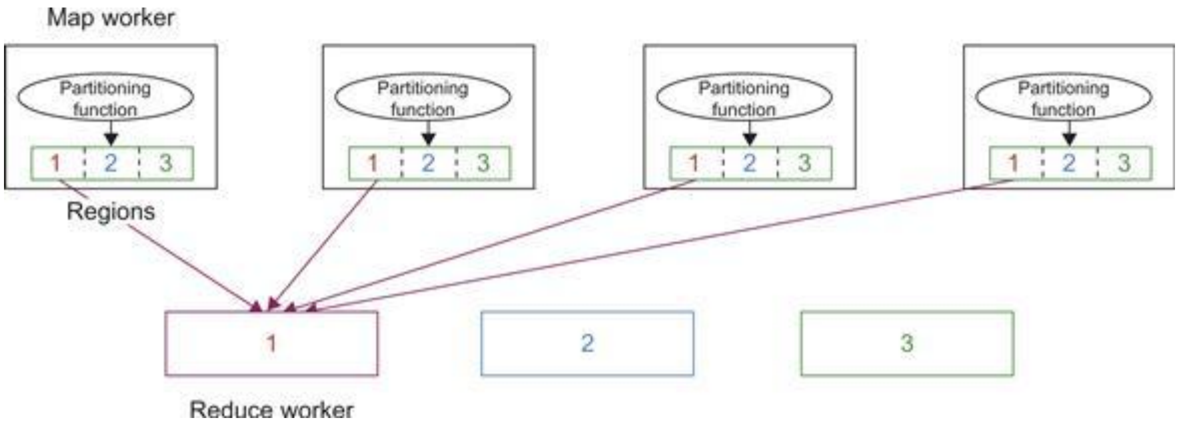
Steps 10 and 11 correspond to the *reduce worker domain:*

MapReduce framework– Step 10 Conti…

- **Sorting and Grouping** When the process of reading the input data is finalized by a reduce worker, the data is initially *buffered in the local disk of the reduce worker*.
- Then the reduce worker groups intermediate (key, value) pairs by sorting the data based on their keys, followed by grouping all occurrences of identical keys.

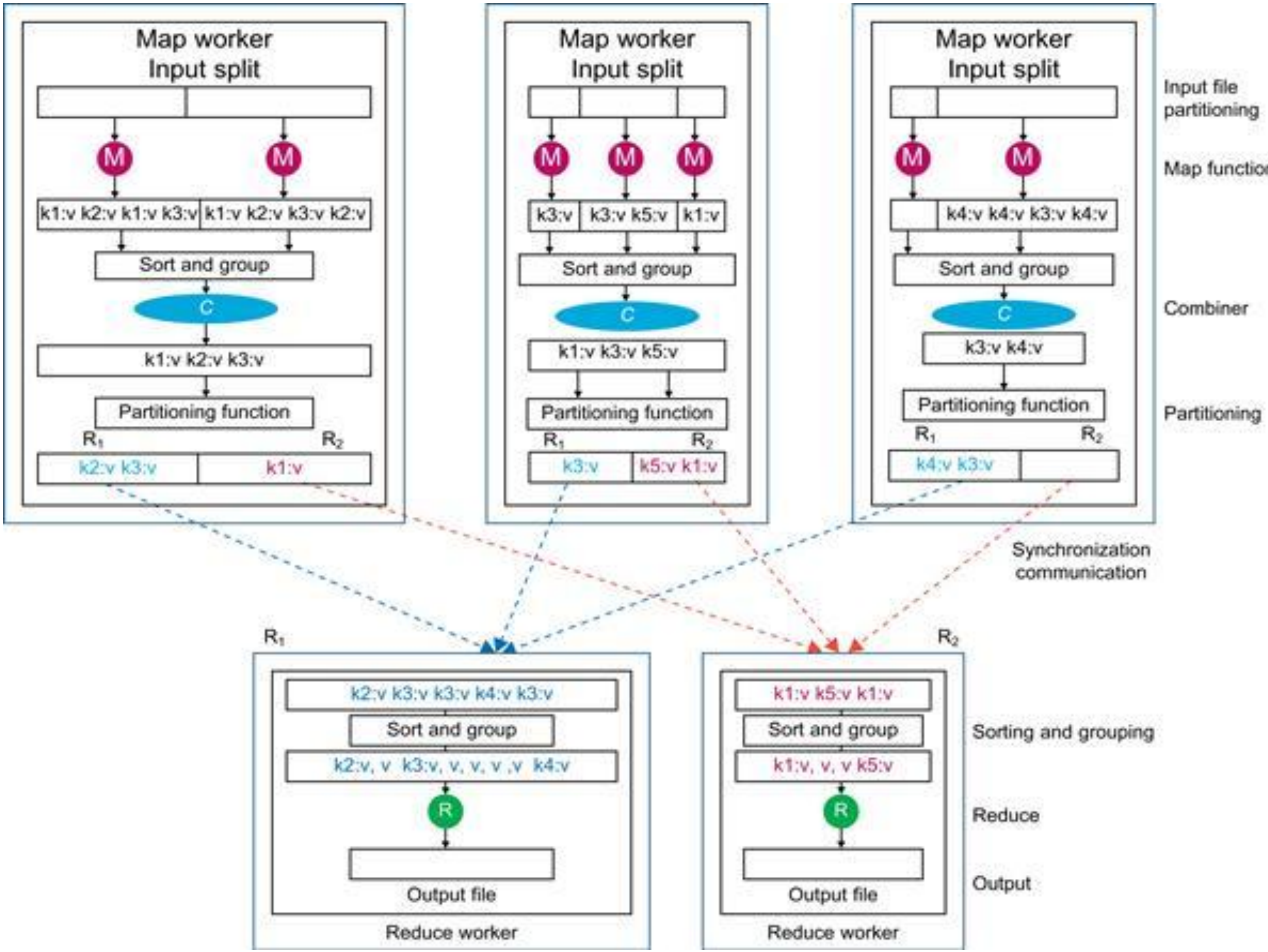
MapReduce framework– Step 10 Conti…

- Note that the buffered data is sorted and grouped because the number of unique keys produced by a map worker may be more than *R* regions in which more than one key exists in each region of a map worker



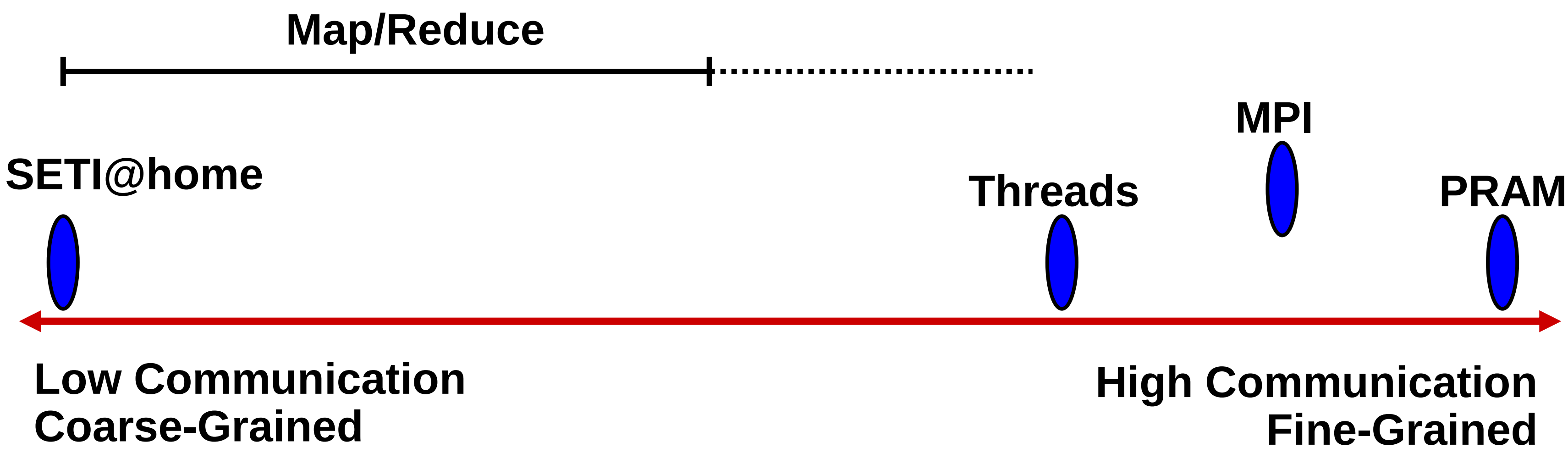
MapReduce framework– Step 11 Conti…

- ***Reduce* function** The reduce worker iterates over the grouped (key, value) pairs, and for each unique key, it sends the key and corresponding values to the *Reduce* function.
- This function then processes its input data and stores the output results in predetermined files in the user's program.



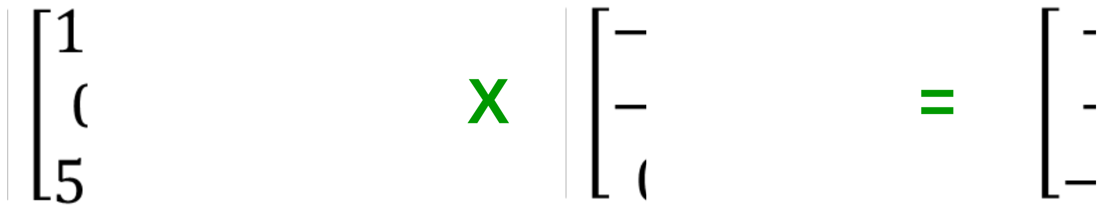
Map worker
Input split
Map worker
Input split
Map worker
Input split
Input file partitioning
M
Map function
k1:v k2:v k1:v k3:v
k1:v k2:v k3:v k2:v
k3:v
k3:v k5:v
k1:v
k4:v k4:v k3:v k4:v
Sort and group
Sort and group
Sort and group
C
Combiner
k1:v k2:v k3:v
k1:v k3:v k5:v
k3:v k4:v
Partitioning function
Partitioning function
Partitioning function
Partitioning
R1
R2
k2:v k3:v
k1:v
k3:v
k5:v k1:v
k4:v k3:v
Synchronization communication
R1
R2
k2:v k3:v k3:v k4:v k3:v
k1:v k5:v k1:v
Sort and group
Sort and group
Sorting and grouping
k2:v, v k3:v, v, v, v ,v k4:v
k1:v, v, v k5:v
R
Reduce
Output file
Output file
Output
Reduce worker
Reduce worker

Where does MapReduce Stands



- Map/Reduce Provides Coarse-Grained Parallelism
  - Computation done by independent processes
  - File-based communication
- Observations
  - Relatively “natural” programming model
  - Research issue to explore full potential and limits

Example: Sparse Matrices with Map/Reduce

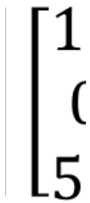


- Task: Compute product C = A·B
- Assume most matrix entries are 0

- Motivation
  - Core problem in scientific computing
  - Challenging for parallel execution
  - Demonstrate expressiveness of Map/Reduce

Computing Sparse Matrix Product

A



$1 \xrightarrow[A]{10} 1$

$1 \xrightarrow[A]{20} 3$

$2 \xrightarrow[A]{30} 2$

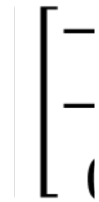
$2 \xrightarrow[A]{40} 3$

$3 \xrightarrow[A]{50} 1$

$3 \xrightarrow[A]{60} 2$

$3 \xrightarrow[A]{70} 3$

B



$1 \xrightarrow[B]{-1} 1$

$2 \xrightarrow[B]{-2} 1$

$2 \xrightarrow[B]{-3} 2$

$3 \xrightarrow[B]{-4} 2$

Represent matrix as list of nonzero entries

⟨**row, col, value, matrixID**⟩

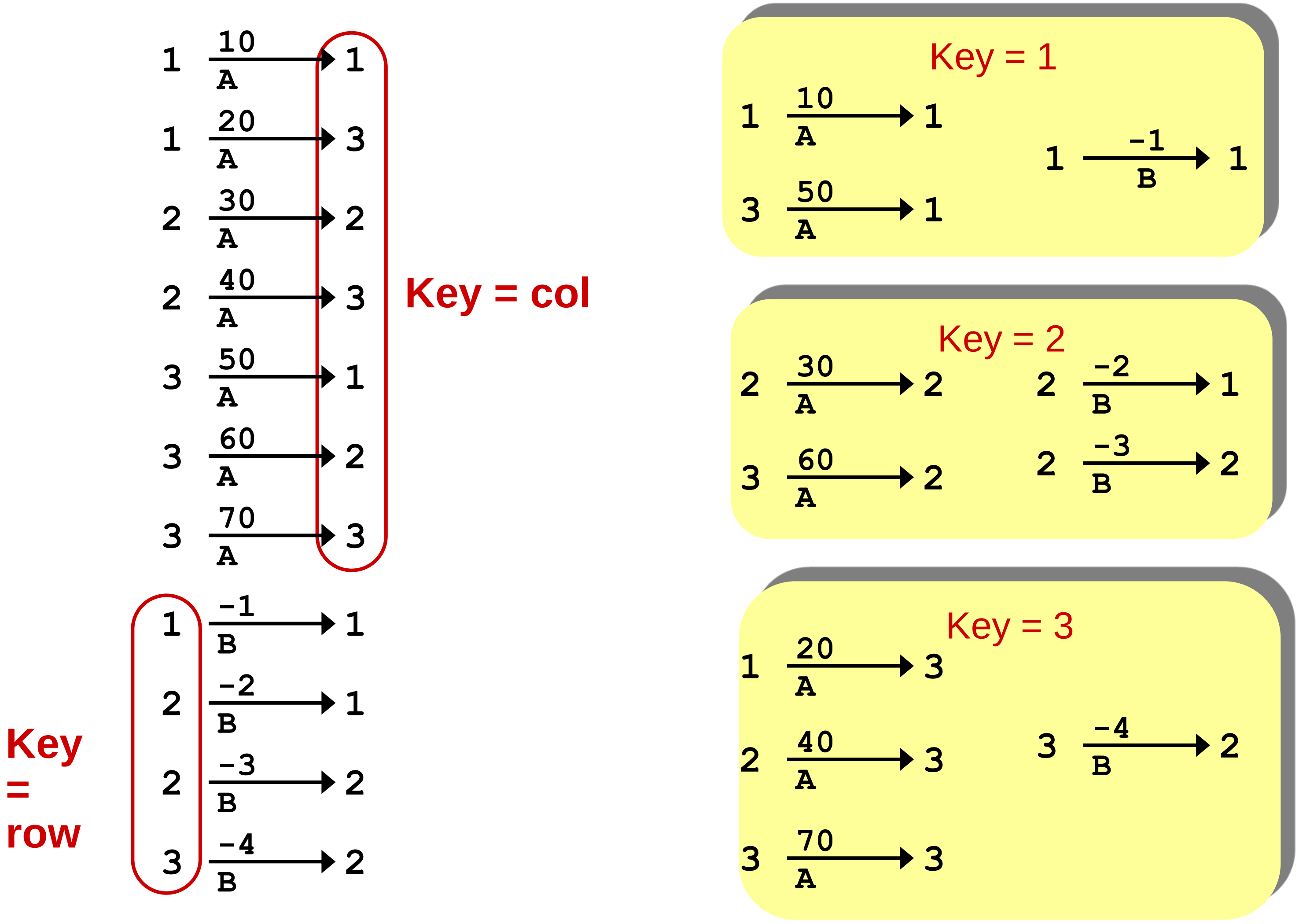
Strategy

Phase 1: Compute all products $a_{i,k} \cdot b_{k,j}$

Phase 2: Sum products for each entry i,j

Each phase involves a Map/Reduce

Phase 1 Map of Matrix Multiply



– Group values $a_{i,k}$ and $b_{k,j}$ according to key k

Phase 1 "Reduce" of Matrix Multiply

Key = 1

$1 \xrightarrow[A]{10} 1$

$3 \xrightarrow[A]{50} 1$

X $1 \xrightarrow[B]{-1} 1$

Key = 2

$2 \xrightarrow[A]{30} 2$

$3 \xrightarrow[A]{60} 2$

X $2 \xrightarrow[B]{-2} 1$

$2 \xrightarrow[B]{-3} 2$

Key = 3

$1 \xrightarrow[A]{20} 3$

$2 \xrightarrow[A]{40} 3$

$3 \xrightarrow[A]{70} 3$

X $3 \xrightarrow[B]{-4} 2$

$1 \xrightarrow[C]{-10} 1$

$3 \xrightarrow[C]{-50} 1$

$2 \xrightarrow[C]{-60} 1$

$2 \xrightarrow[C]{-90} 2$

$3 \xrightarrow[C]{-120} 1$

$3 \xrightarrow[C]{-180} 2$

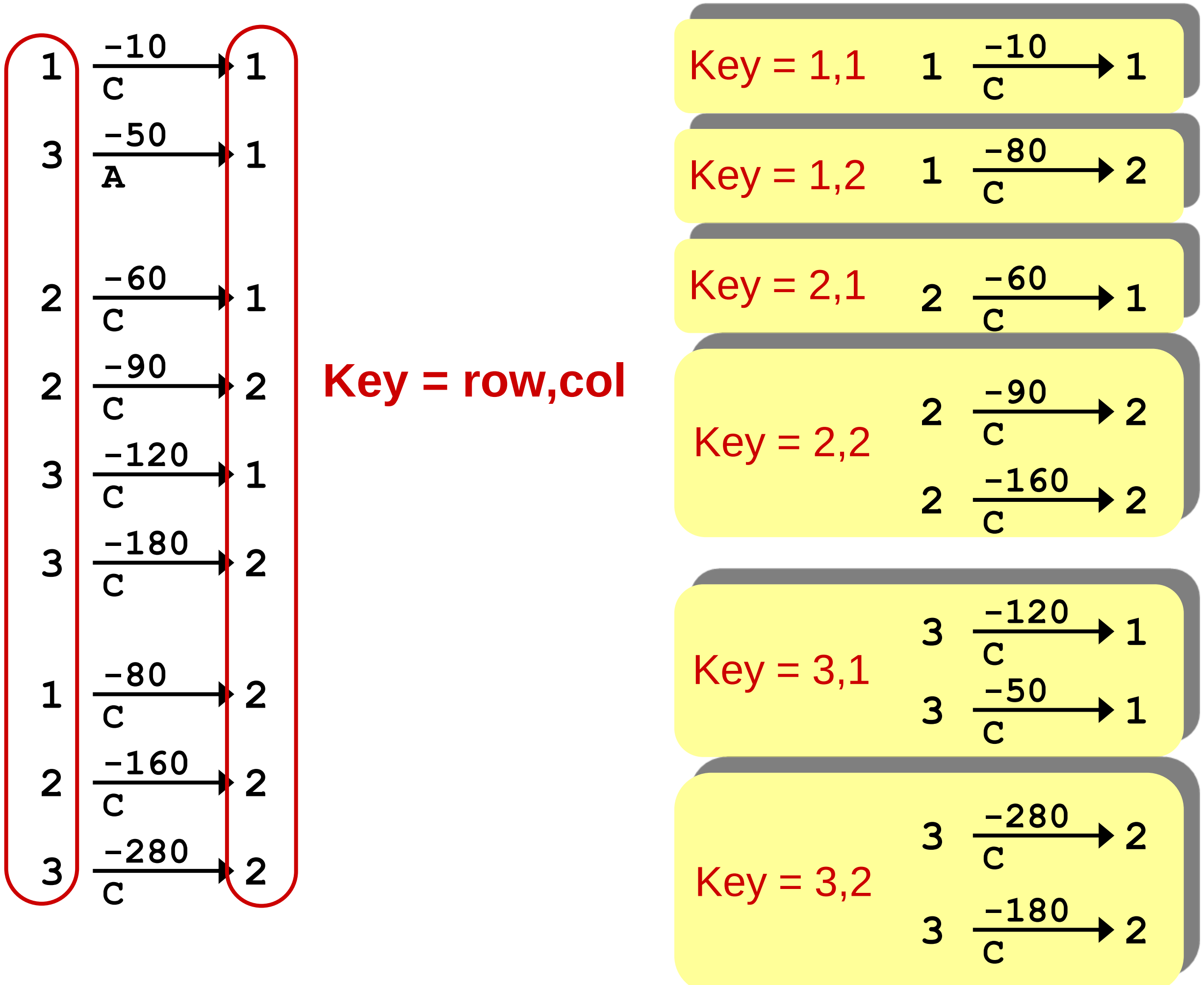
$1 \xrightarrow[C]{-80} 2$

$2 \xrightarrow[C]{-160} 2$

$3 \xrightarrow[C]{-280} 2$

– Generate all products $a_{i,k} \cdot b_{k,j}$

Phase 2 Map of Matrix Multiply



– Group products $a_{i,k} \cdot b_{k,j}$ with matching values of i and j

Phase 2 Reduce of Matrix Multiply

Key = 1,1	$1 \xrightarrow[C]{-10} 1$	$1 \xrightarrow[C]{-10} 1$
Key = 1,2	$1 \xrightarrow[C]{-80} 2$	$1 \xrightarrow[C]{-80} 2$
Key = 2,1	$2 \xrightarrow[C]{-60} 1$	$2 \xrightarrow[C]{-60} 1$
Key = 2,2	$2 \xrightarrow[C]{-90} 2$ $2 \xrightarrow[C]{-160} 2$	$2 \xrightarrow[C]{-250} 2$
Key = 3,1	$3 \xrightarrow[C]{-120} 1$ $3 \xrightarrow[C]{-50} 1$	$3 \xrightarrow[C]{-170} 1$
Key = 3,2	$3 \xrightarrow[C]{-280} 2$ $3 \xrightarrow[C]{-180} 2$	$3 \xrightarrow[C]{-460} 2$

C

– Sum products to get final entries