Unit II

21CSE221T – Big Data Tools and Techniques

Map Reduce

Map Reduce

- A **programming model** for data processing
- Hadoop can run MapReduce programs written in various languages
- MapReduce programs are inherently parallel
- No special hardware required
- Based on **divide-and-conquer** principle

Map Reduce

- Input data sets are split into **independent chunks**, which are processed by the **mappers in parallel**
- Execution of the maps is typically co-located with the data
- MR sorts the outputs of the maps, and uses them as an input to the reducers
- Responsibility of the user is to implement mappers and reducers
    - classes that extend Hadoop-provided base classes to solve a specific problem

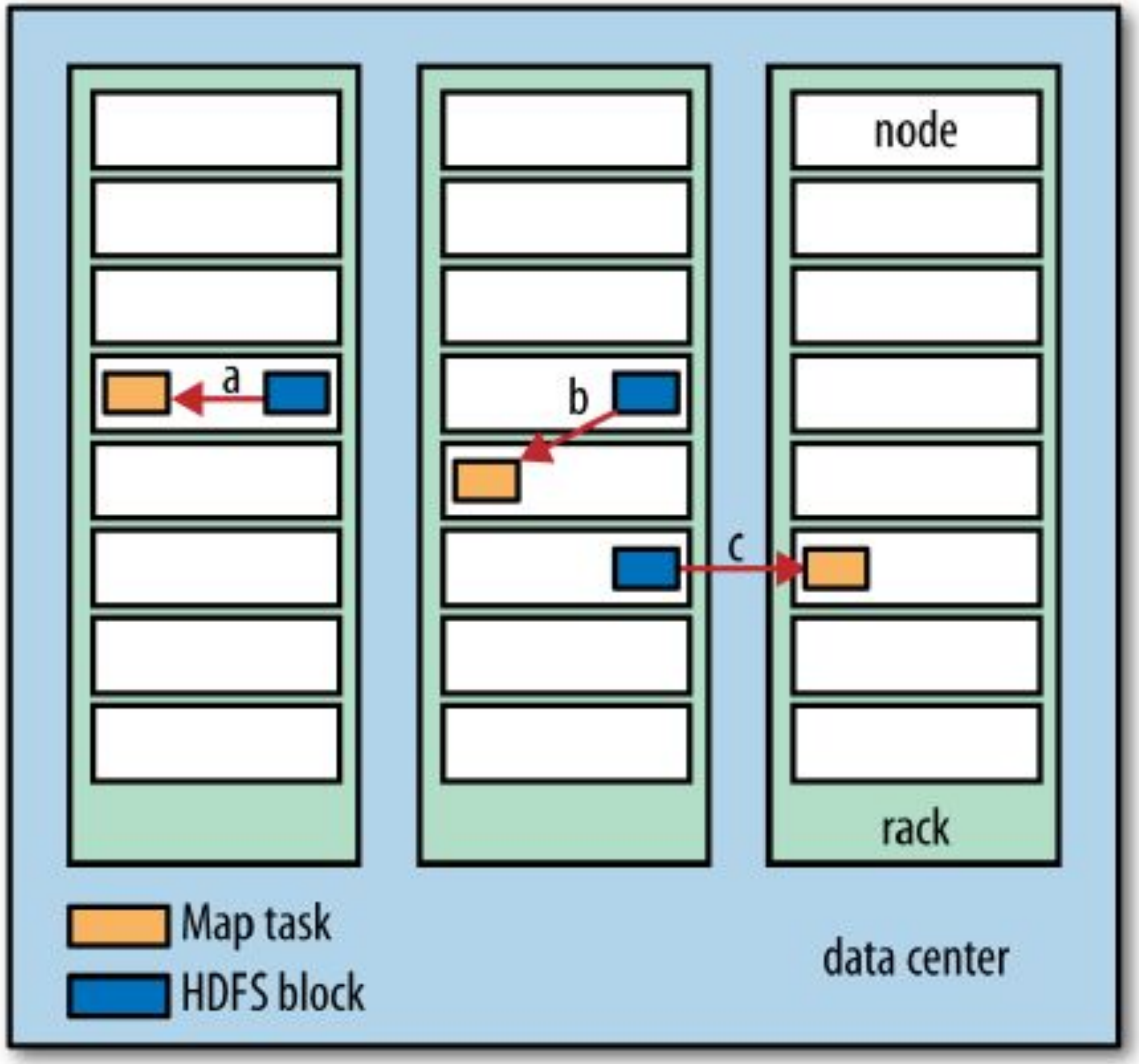
Map Reduce

- MapReduce works by breaking the processing into two phases
  - Map phase
  - Reduce phase
- Each phase has key-value pairs as input and output
  - the types of which may be chosen by the programmer
- The programmer also specifies two functions
  - the map function and the reduce function

Map Reduce

- Splits
  - Input is divided into fixed size chunks
  - Hadoop creates one map task for each split
  - Map function is executed for each record of split
  - Small splits
    - Less processing time for each split
    - Parallel – Better load balancing
  - Too small
    - Managing is difficult
  - Good split should be size of HDFS block

Map Reduce



Inputs and Outputs of Map Reduce

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

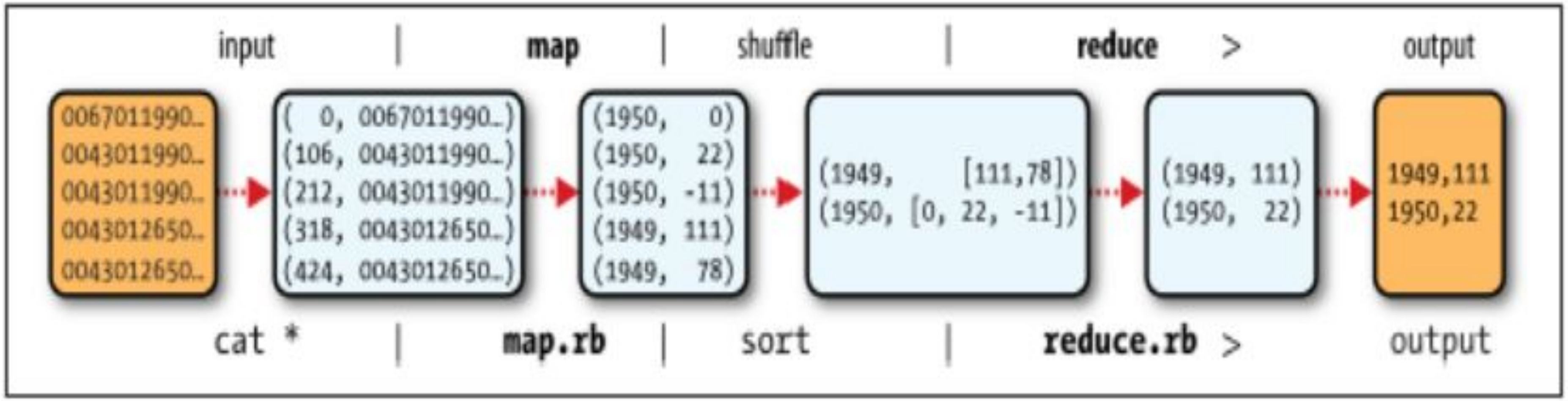
Inputs and Outputs of Map Reduce

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)

(1949, [111, 78])
(1950, [0, 22, -11])

(1949, 111)
(1950, 22)
```

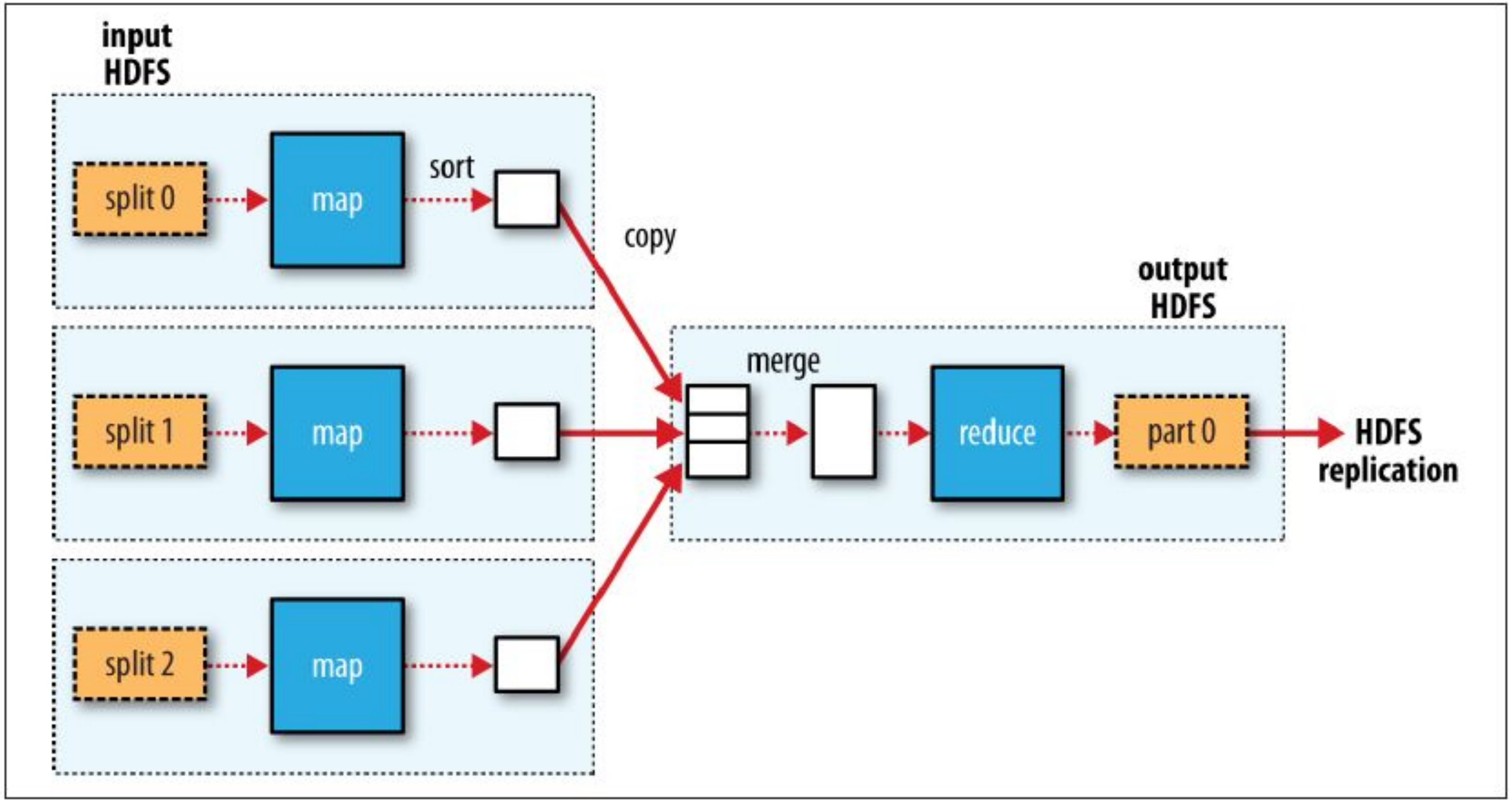
Map Reduce logical data flow



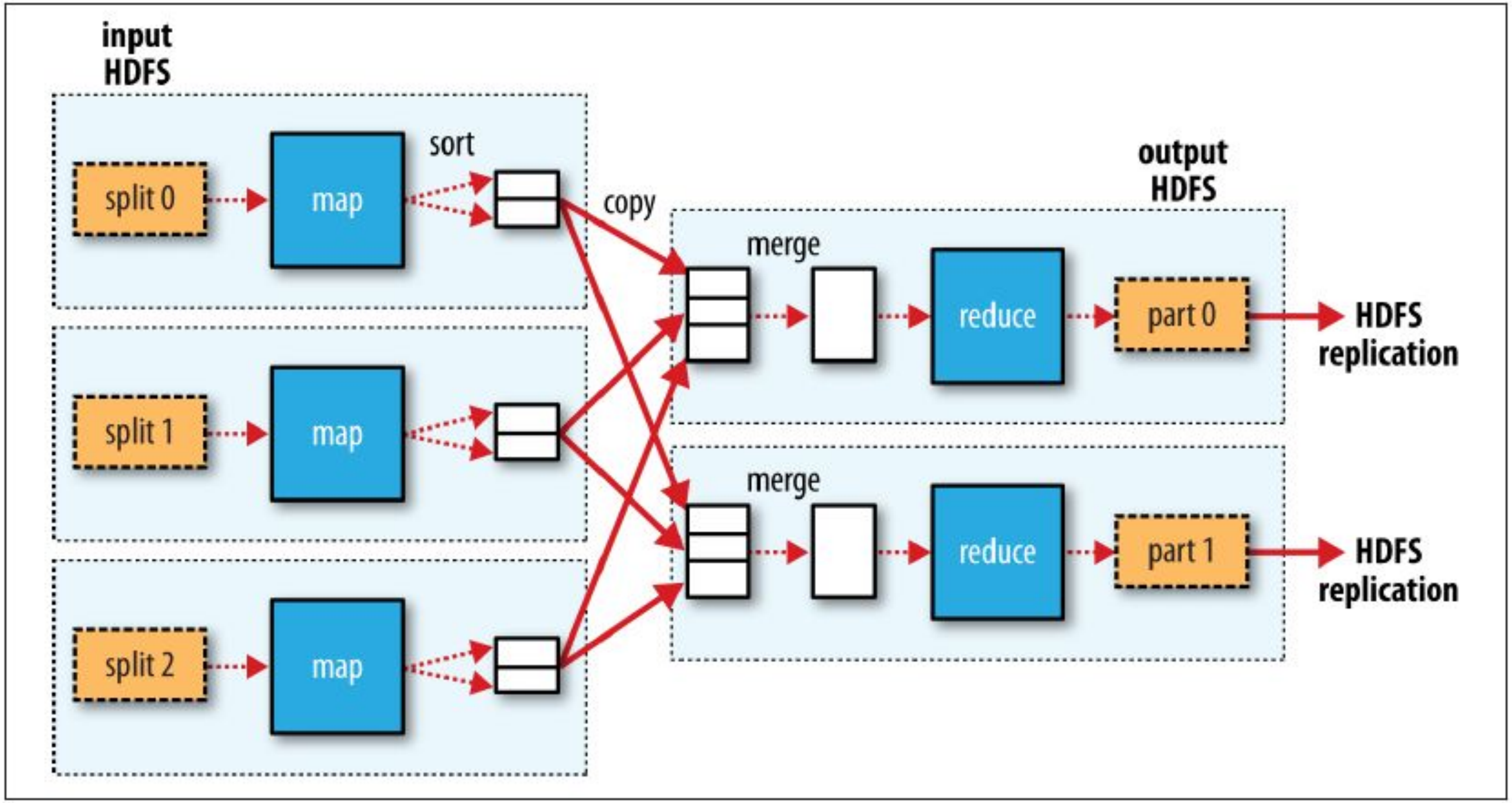
Map Reduce

- Mapper - Takes input in a form of key/value pairs (k1, v1) and transforms them into another key/value pair (k2, v2).
- The MapReduce framework sorts a mapper's output key/value pairs and combines each unique key with all its values (k2, {v2, v2,…}).
- These key/ value combinations are delivered to reducers, which translate them into yet another key/value pair (k3, v3).

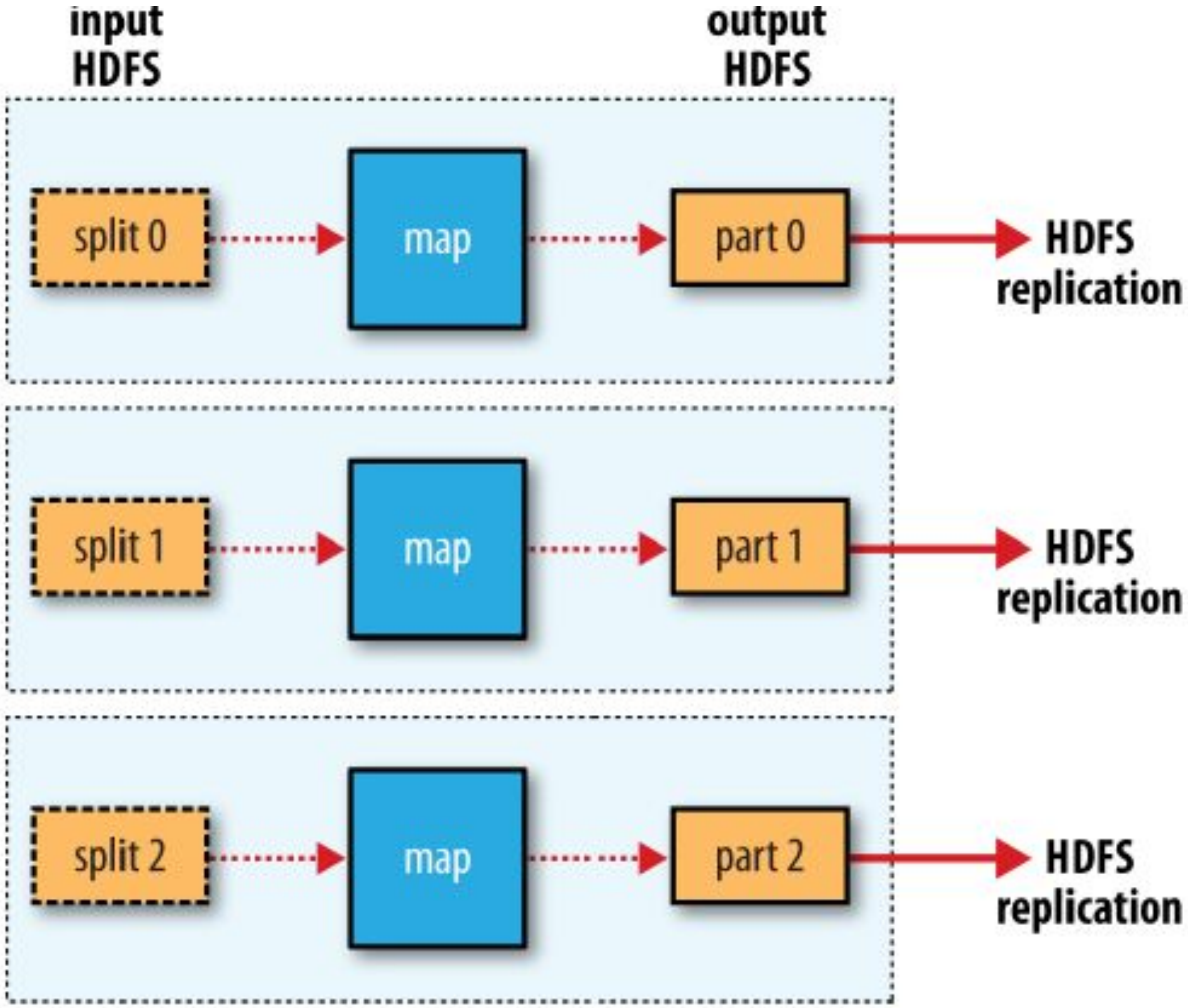
Map Reduce – Single Reduce



Map Reduce – Multiple reduce



Map Reduce – Zero Reduce



Combiners

```
(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:

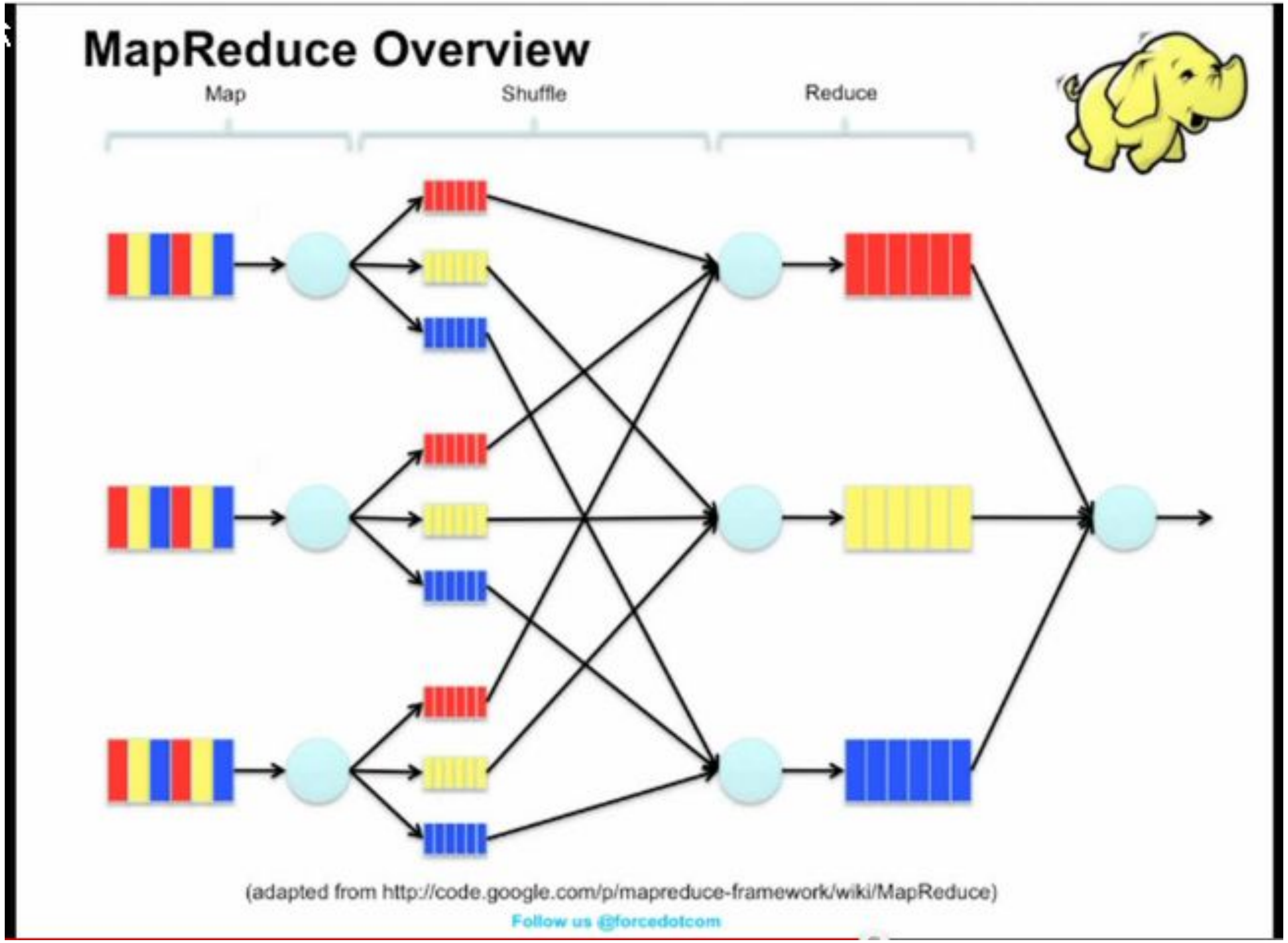
```
(1950, [0, 20, 10, 25, 15])
```

with output:

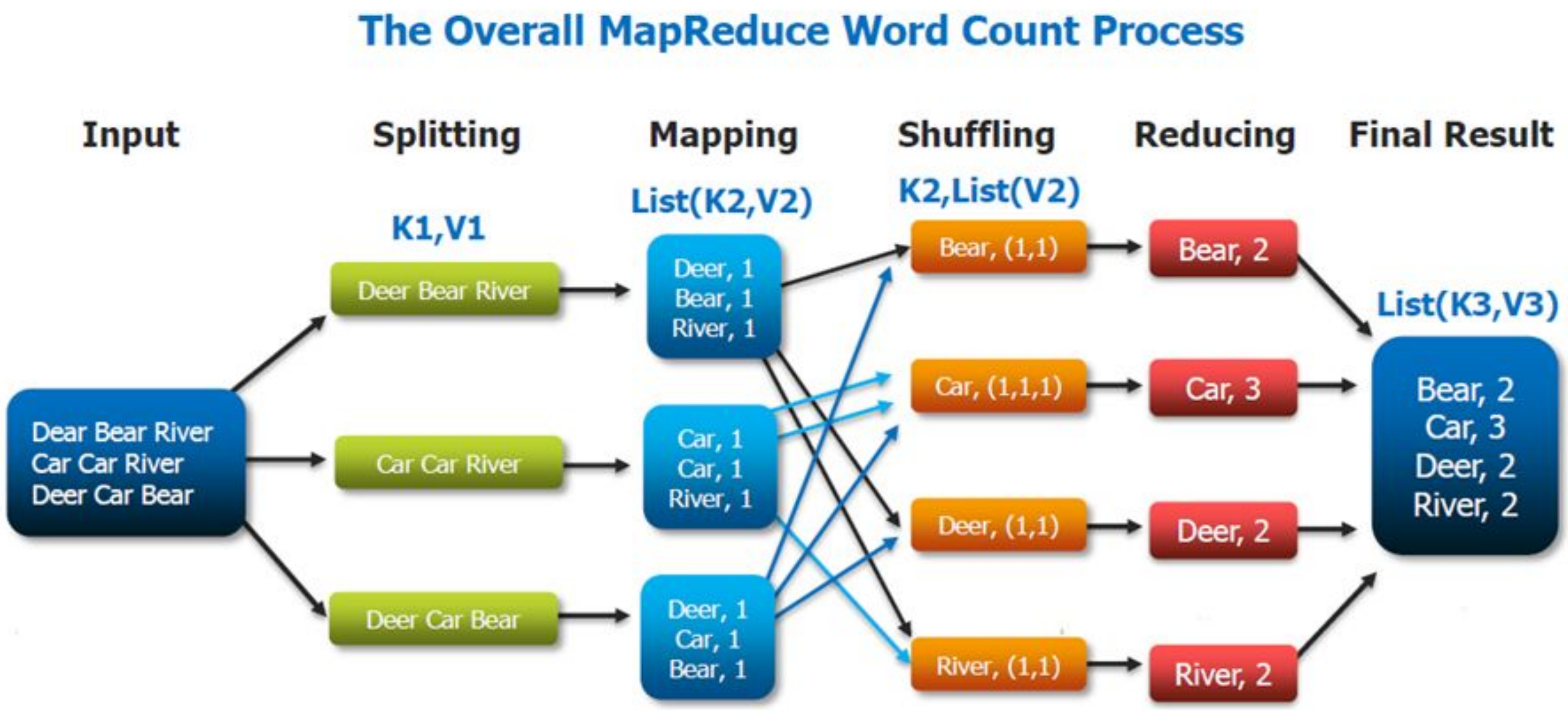
```
(1950, 25)
```

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

More examples



More examples



Hadoop Streaming

- A utility that comes with the Hadoop distribution
- Mapper and Reducer will be scripts or programs in any language
- For example, if we have mapper.py and reducer.py, then the execution is summarized as,

```
# $HADOOP_HOME/bin/hadoop jar
$HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-1.2.1.jar \
 -input input_dirs \
 -output output_dir \
 -mapper path/mapper.py \
 -reducer path/reducer.py
```

Hadoop I/O

Hadoop data I/O

- General Characteristics of data integrity in any environment
  - Data Integrity
  - Compression
- Hadoop specific
  - Serialization
  - On-disk data structures

1. Data Integrity

- No data loss – storage or processing
- Volume of data – chances of data corruption in Hadoop is high
- Detecting corrupted data – checksum
- Checksum - computed when data/file is entering the system for the **first time**
- When data transmitted acrosss unreliable network link – checksum computed at the end of transmission

Data Integrity

- Checksum only checks data corruption **does not fix it**
- Common method used is CRC-32 Cyclic redundancy check
- Computes a 32 bit checksum for input of any size
- HDFS – CRC32C – variant of original method

Data Integrity in HDFS

- Transparently checksums all data written to it
- When reading data, by default, verifies checksums
- Separate checksum for every *dfs.bytes-perchecksum* size of data
- 512 bytes – normal setting
- Checksum storage is less than 1 percent of file size

Data Integrity in HDFS - Writes

- Datanodes - responsible for verifying the data they receive before storing the data and its checksum
- Applicable for data from clients and from other datanodes during replication
- Last Datanode in the pipeline verifies the checksum
- If datanode detects error, client gets a IOException
- Handled app specific way like rewrites

Data Integrity in HDFS - Reads

- Client reads data – compares checksum stored in data node
- Datanode logs the latest verification time of each chunk
- Heal corrupted blocks
  - by copying one of the good replicas to produce a new, uncorrupt replica

Data Integrity in HDFS - Reads

- When error/corruption block is found
  - Reports bad block, Datanode information to Namenode
  - Throws ChecksumException
- Namenode
  - Marks block replica so that no other clients read from that block replica
  - schedules a copy of the block to be replicated on another datanode, so its replication factor is back at the expected level
  - Corrupt block deleted
- Find a file's checksum
  - hadoop fs -checksum

Data Integrity in HDFS

- Two classes in HDFS
  - LocalFileSystem
  - ChecksumFileSystem

LocalFileSystem

- Client side checksum computation
- On write, computes checksum and stores it in a file called ‘filename.crc’
- The file has checksums for all chunks in a file
- Chunk size is defined by the property *`dfs.bytes-perchecksum`*
- Default is 512 bytes

LocalFileSystem

- Chunk size is also written as meta data in the .crc file
  - Even if chunk size is reconfigured, no effect in checksum comparison as it retains the previous chunk size
- Read
  - Checksum verified for each chunk against ‘filename.crc’
  - Throws a CheckSumException during mismatch

2. Data compression

- File compression benefits
  - reduces the space needed to store files,
  - speeds up data transfer across the network or to or from disk

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

Data compression

- All compression algorithms exhibit a space/time trade-off
- faster compression and decompression speeds is at the expense of smaller space savings
- –1 means optimize for speed, and -9 means optimize for space

```
% gzip -1 file
```

Data compression

- gzip - middle of the space/time trade-off
- bzip2 – more effective than gzip but slower
- bzip2's decompression speed is faster than its compression speed, but it is still slower than the other formats

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

Data compression

- LZO, LZ4, and Snappy, all optimize for speed and are faster than gzip, but compress less effectively.
- Snappy and LZ4 are also significantly faster than LZO for decompression
- Splittable - whether you can seek to any point in the stream and start reading from some point further on

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

Codecs

- Codec is the implementation of a compression-decompression algorithm
- Hadoop - represented by an implementation of the interface *CompressionCodec*
- Example: GzipCodec class encapsulates the compression and decompression algorithm for gzip

Compression

- CompressionCodec has two methods that allow you to easily compress or decompress data.
- Compress (data written to an output stream)
  - ***createOutputStream***(*OutputStream* out) method to create a *CompressionOutputStream*
  - to which you write your uncompressed data to have it written in compressed form to the underlying stream

Example Compression

A program to compress data read from standard input and write it to standard output

```
public class StreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
      ReflectionUtils.newInstance(codecClass, conf);

    CompressionOutputStream out = codec.createOutputStream(System.out);
    IOUtils.copyBytes(System.in, out, 4096, false);
    out.finish();
  }
}
```

```
% echo "Text" | hadoop StreamCompressor org.apache.hadoop.io.compress.GzipCodec \
  | gunzip -
Text
```

Decompression

- To decompress data being read from an input stream
  - In the codec class instance, call ***createInputStream***(*InputStream* in) to obtain a *CompressionInputStream*

Compression in MR – Hypothetical Example

- Compression format should support splitting?
- Consider an **uncompressed** file stored in HDFS whose size is 1 GB.
- HDFS block size is 128 MB
- file will be stored as eight blocks
- 8 input splits processed separately by 8 mappers

Compression in MR – Hypothetical Example

- Compression format should support splitting?
- Consider an **gzip compressed** file stored in HDFS whose size is 1 GB.
- HDFS block size is 128 MB
- file will be stored as eight blocks
- Creating a split for each block wont work since gzip is not splittable
- impossible for a map task to read its split independently of the others

Compression in MR – Hypothetical Example

- MR will not try to split the gzipped file
- MR knows that the input is gzip-compressed and that gzip does not support splitting.
- A single map will process the eight HDFS blocks, most of which will not be local to the map. (Data locality is violated)
- Also, with fewer maps, the job is less granular and so may take longer to run

Compression in MR – Hypothetical Example

- Compression format should support splitting?
- Consider an **bzip2 compressed** file stored in HDFS whose size is 1 GB.
- Supports splitting

Which Compression Format to Use?

- Depends on file size, format, and the tools you are using for processing
- most to least effective
- Avro datafiles, ORCFiles,or Parquet files all of which support both compression and splitting
- A fast compressor such as LZO, LZ4, or Snappy
- Use a compression format that supports splitting, such as bzip2
- Input Split==HDFS block size

Serialization

- Process of **turning structured objects** into a byte stream for transmission over a network or for writing to persistent storage
- *Deserialization is the reverse* process of turning a byte stream back into a series of structured objects
- Hadoop – IPC implemented using Remote Procedure Calls(RPCs)

Serialization

- RPC – uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message
- Serialization format should be
    - Compact
    - Fast
    - Extensible
    - Interoperable

Hadoop Serialization Format

- Writable Interface
- Writables are central to Hadoop
- Most MapReduce programs use them for their key and value types

```
IntWritable writable = new IntWritable();
writable.set(163);
```

Writable wrapper classes for Java primitives

Java primitive	Writable implementation	Serialized size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1–5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1–9
double	DoubleWritable	8