



Inspire...Educate...Transform.

Big Data

Session 3: MapReduce, YARN, Spark

Suryaprakash Kompalli
Senior Mentor, INSOF

This presentation may contain references to findings of various reports available in the public domain. INSOF makes no representation as to their accuracy or that the organization subscribes to those findings.



Review of Last Lecture

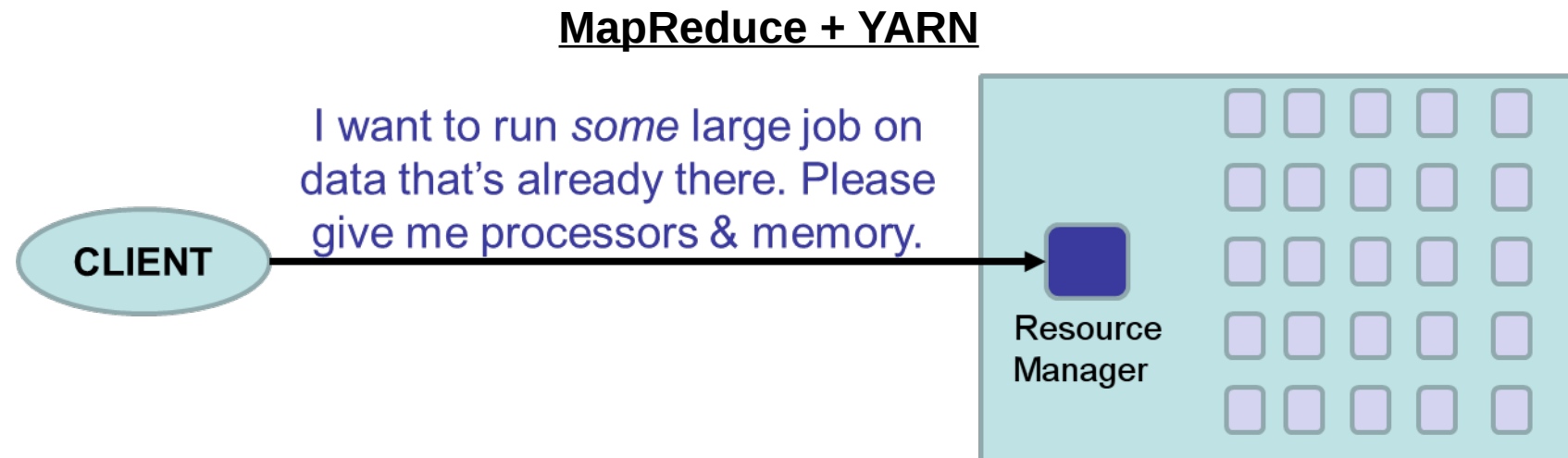
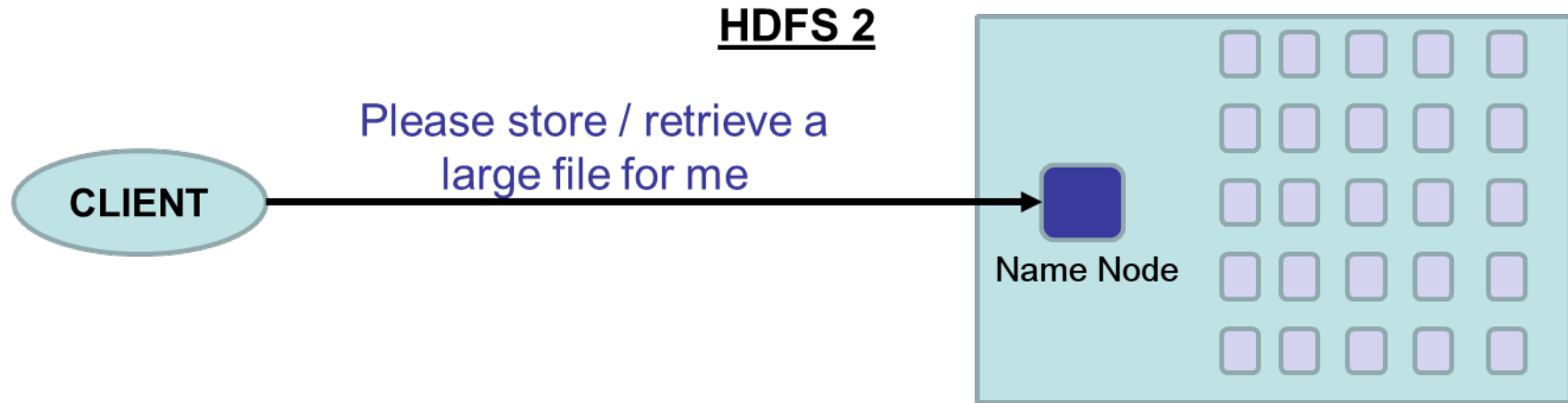
- **Class 1:**
 - Different architectures
 - Transition from Databases to data warehouses and data lakes
 - Thinking of Large Jobs as Task Decompositions
 - How BigData is changing IT and business operations
- **Class 2:**
 - Hadoop: Storage



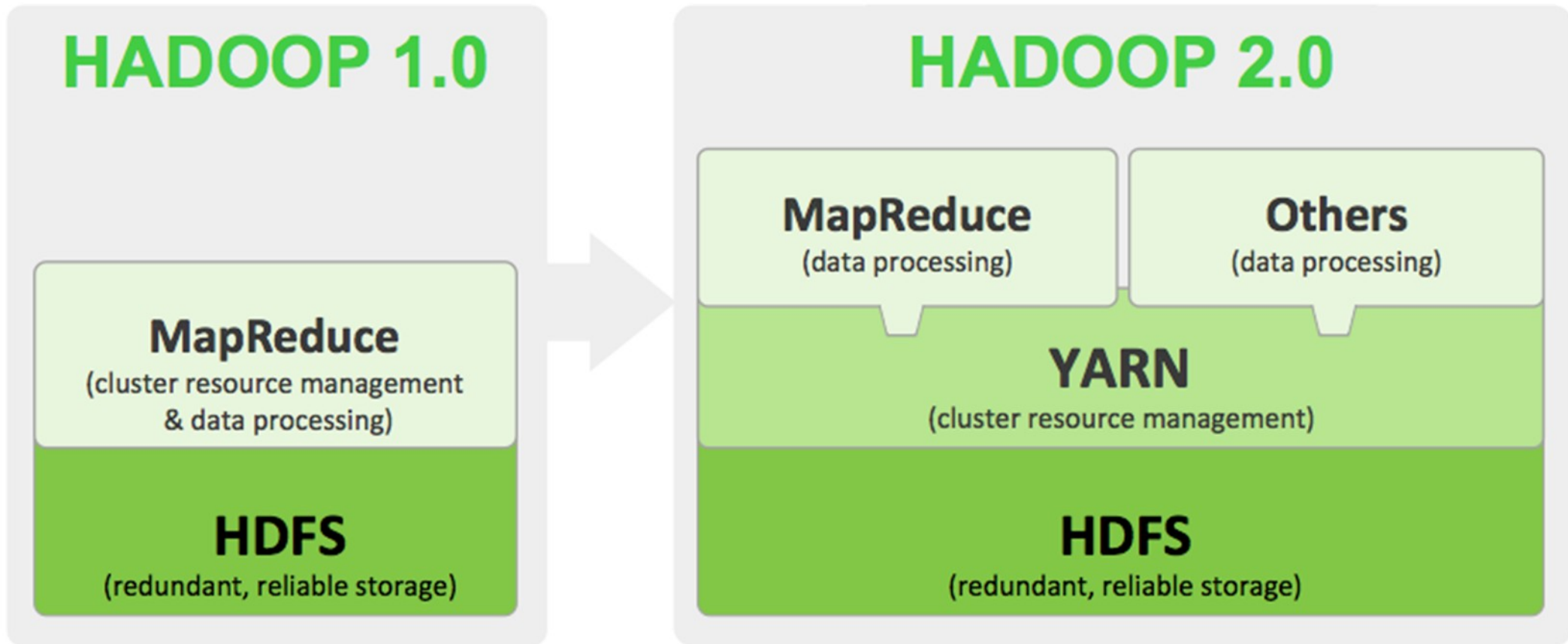
Agenda

- MapReduce and YARN
- Spark

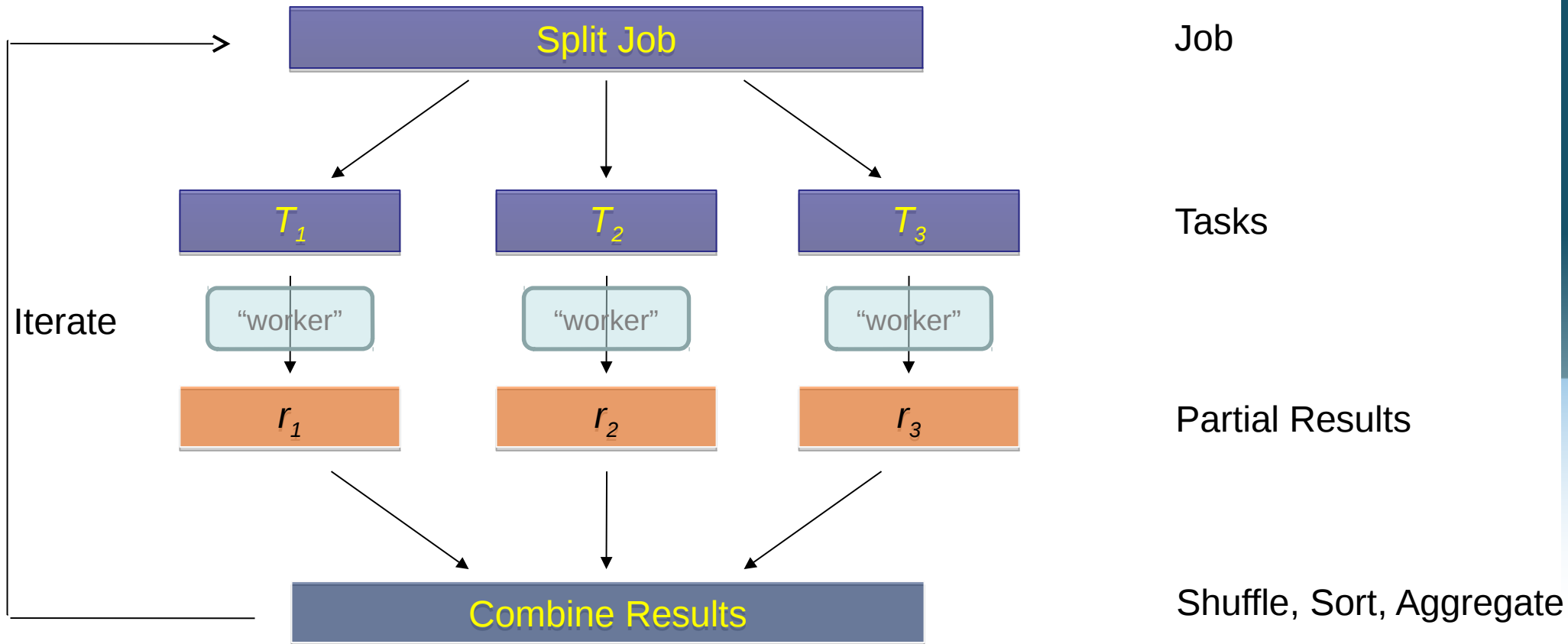
Context: MapReduce and YARN



Context: MapReduce and YARN



Map-Reduce



Map-Reduce: Solving large data problems

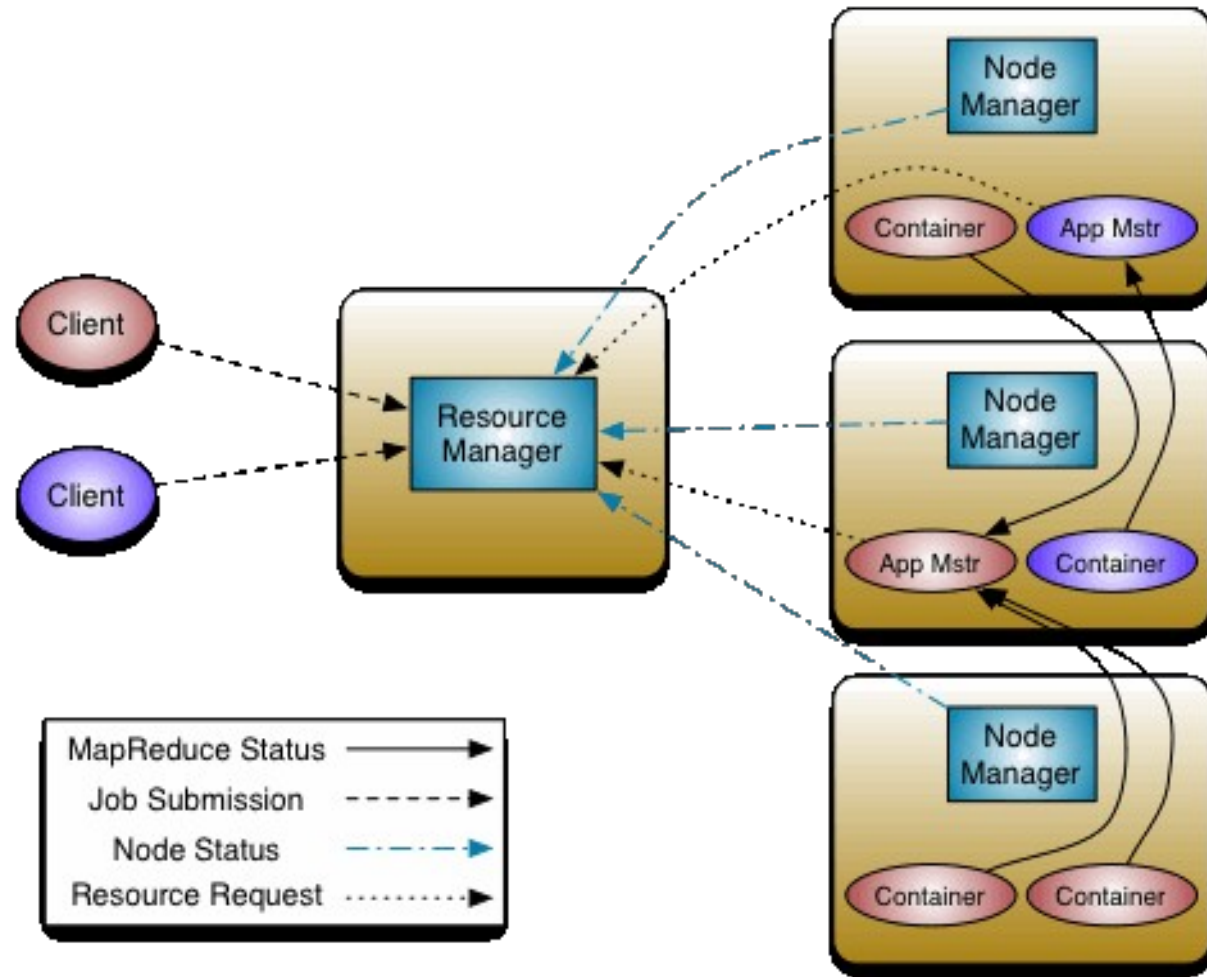


Map

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Reduce !!!

MRv2, YARN: JobTracker Redefined



Node Manager sits on each node. It can talk to Name Node and Resource Manager
App Master is executed on some of the nodes

Resource Manager/YARN (~2013)

<http://hortonworks.com/blog/apache-hadoop-yarn-concepts-and-applications/>
<https://wiki.apache.org/hadoop/Roadmap>

YARN Daemons

- **Resource Manager:**

- Runs on one node
- Global resource scheduler
- Negotiates system resources for apps



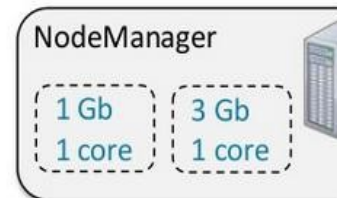
- **Node Manager:**

- Runs on all slave nodes
- Communicates with RM



- **Container:**

- Created by RM upon client/App Master request
- Allots a certain resource (RAM, CPU) on slave node
- Apps run in one or more container



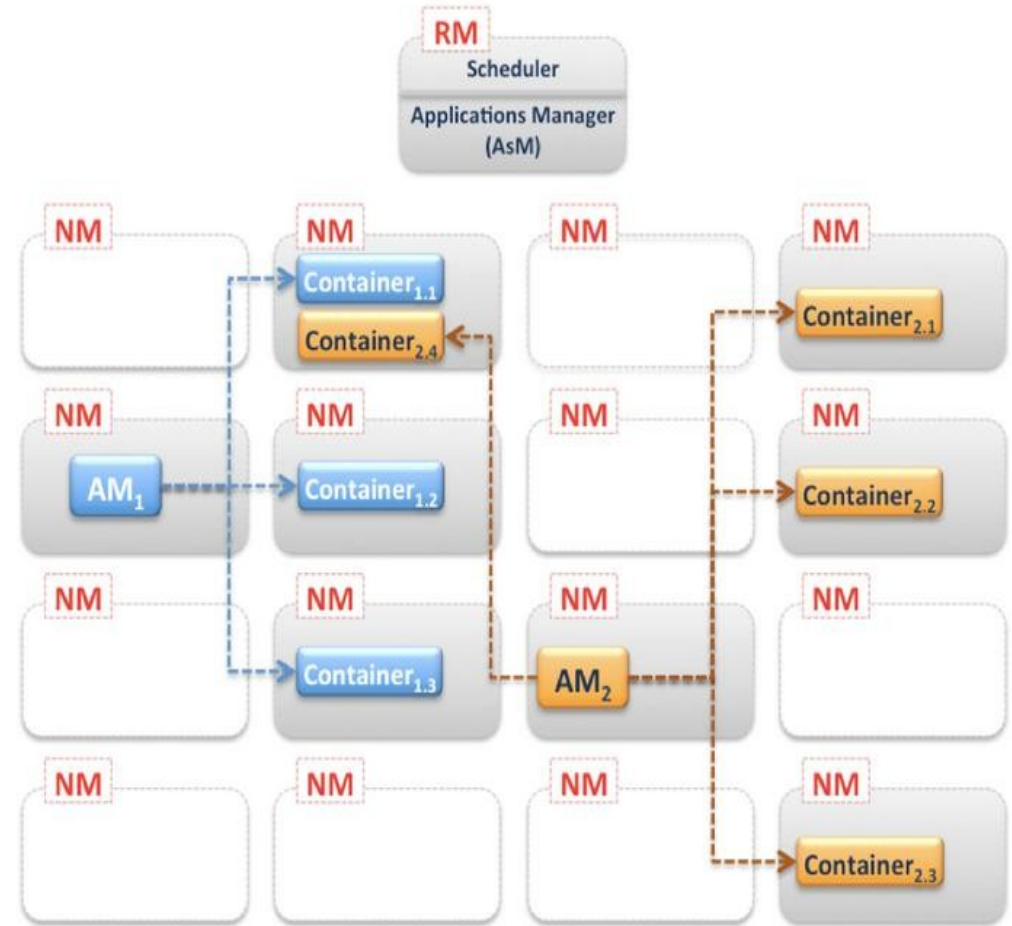
- **Application Master:**

- One per app
- Framework/app specific
- Runs in container
- Requests more containers as needed



YARN: Details of Initiating Jobs/Apps

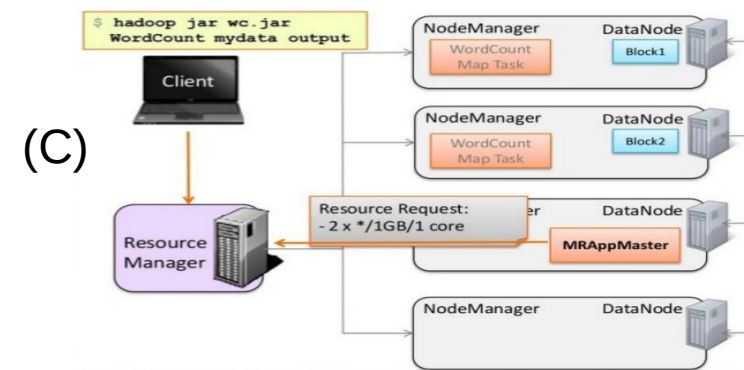
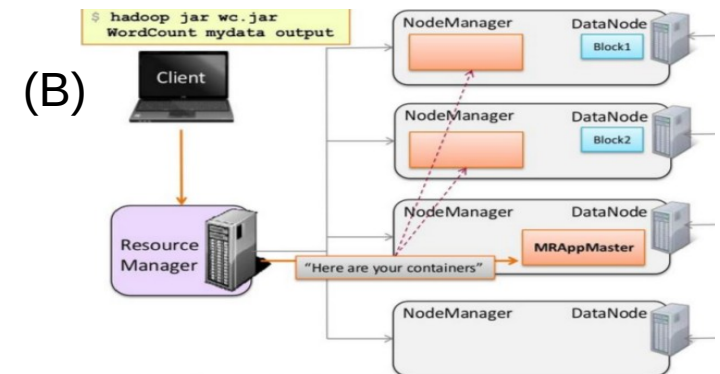
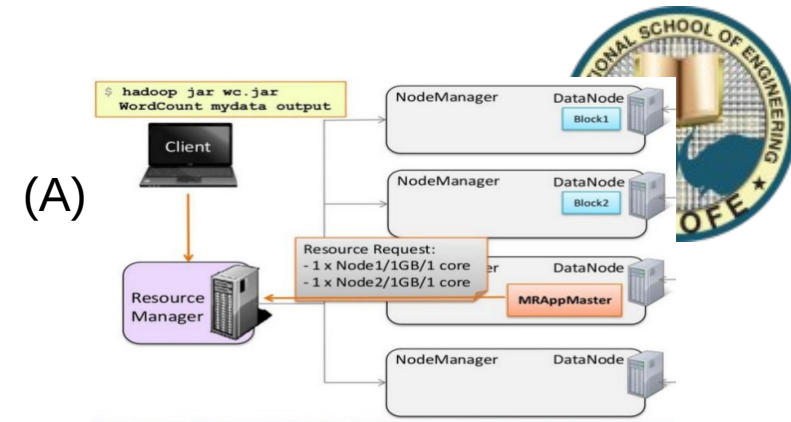
- Client program submits the app
- ResourceManager negotiates a container to start the ApplicationMaster
 - When container is available, it launches the ApplicationMaster
- ApplicationMaster on boot-up registers with the ResourceManager
- ApplicationMaster negotiates appropriate resource containers
 - On successful container allocations, ApplicationMaster launches the container by providing the container launch specification (with app code) to NodeManager
 - App code executing within the container provides necessary information (progress, status etc.) to its ApplicationMaster via an application-specific protocol.
- During the app execution, client that submitted the app communicates directly with the ApplicationMaster to get status, progress updates etc.
- Once app is complete, and all necessary work has been finished, the ApplicationMaster deregisters with the ResourceManager and shuts down, allowing its own container to be repurposed



Example: Application 1 (Blue) is using three containers, and Application 2 (Brown) is using four containers.

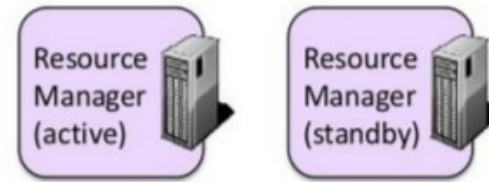
How does MR2 use YARN

- Note: Any app can use YARN
- MR2 has a MRAppMaster
- When you submit a MR job:
 - Job goes to MRAppMaster
 - MRAppMaster asks RM for resources (A)
 - RM allots the containers (B)
 - MRAppMaster runs the MR task on the containers (C)
- Typical container allocation:
 - Mapper containers located close to data nodes that contain required files (or 2 hops from them)
 - Reducer containers placed close to mappers

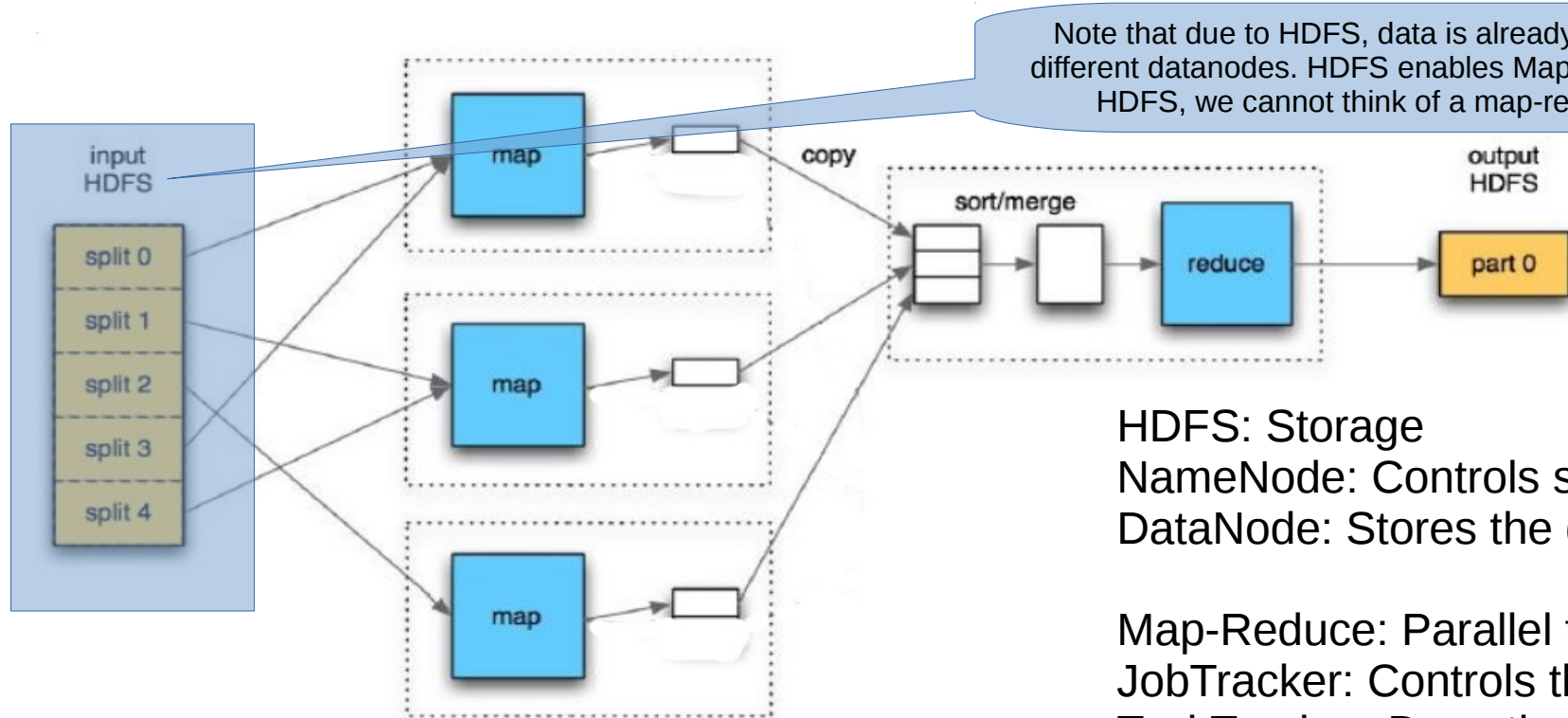


YARN Fault Tolerance

- **ResourceManager:**
 - No applications can run if RM is failing
 - Can be configured with High Availability (HA)
- **NodeManager:**
 - If RM stops getting heartbeat from NM, the NM is removed from list of active nodes
 - If AppMaster detects that an NM that has its container has failed, all tasks on that container are treated as failed
 - If a NodeManager is running an AppMaster and fails, the entire app is treated as failed
- **Task (Container) Failure:**
 - MRAppMaster will restart tasks that stop execution or stop responding (4 times default)
 - Applications with too many failed tasks are reported as failed
- **ApplicationMaster:**
 - If application fails or AM stops heartbeat, RM will re-attempt whole app (2 times default)
 - MRAppMaster settings: Job recovery flag
 - If set to false, all tasks are rerun when MRAppMaster restarts
 - If set to true, state of tasks on restart is reviewed, only incomplete tasks are re-run

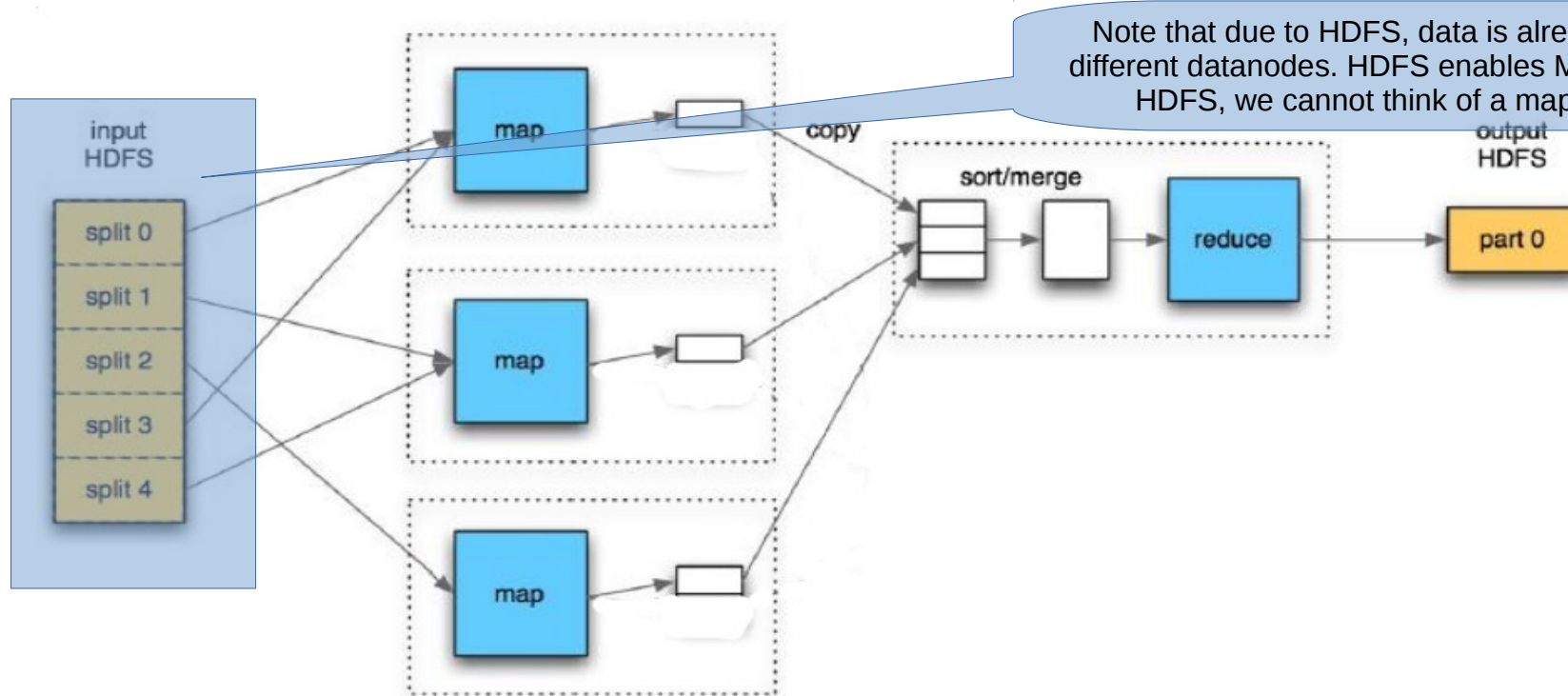
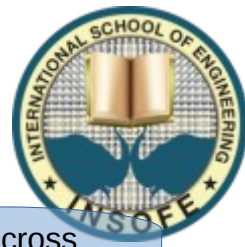


Map-Reduce: Splitting Jobs



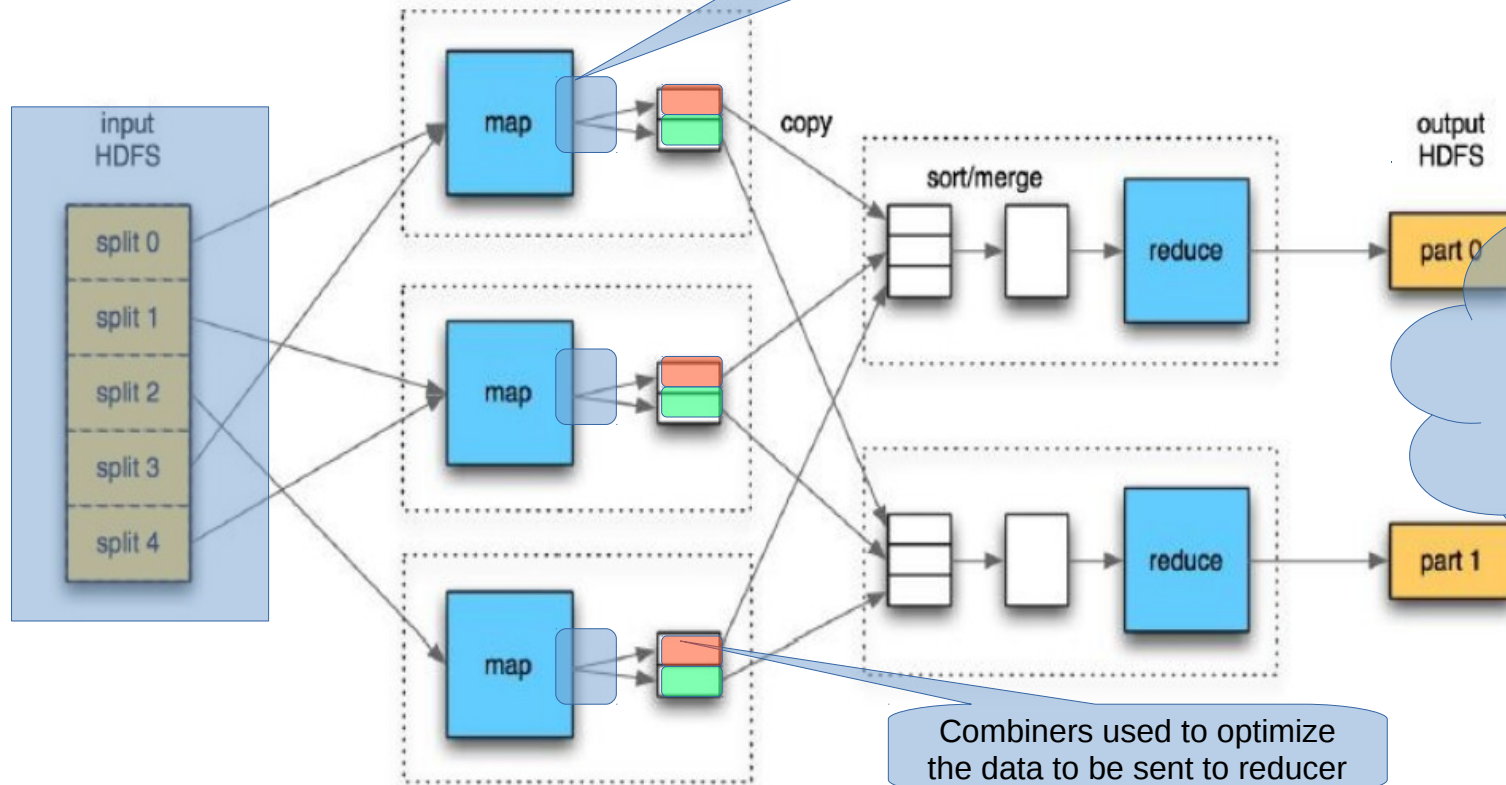
- App/client submits job to the **MRAppMaster**
 - The job could be in form of code for the map and reduce steps that are executed on **Containers**
 - There may optionally be intermediate steps (we will see a few intermediate steps in this and future classes)
- Typically there will be only one physical reducer for each map-reduce job
 - Applications may request multiple reducers
 - Reducer can complete only after all mappers are done

Map-Reduce



- Ex: word counting of 300K documents spread across 3 data nodes
 - Each data node:
 - Has ~100K documents, one mapper
 - Counts the word in its local set of 100K documents
 - Reducer combines the results
- Let us say mapper 1 generates following counts by analyzing its files:
 - [Bombay, 1] [Alaska, 1] [Buffalo, 1]
[Bombay, 1] [Bombay, 1] [Alaska, 1]
[London, 1] [Zaire, 1] etc
 - It will send these results to the reducer
- Reducer gets results from all mappers to compute the final results
- Output will be written to HDFS
 - [Alaska, 2] [Bombay, 3]
[Buffalo, 1] [London, 1]
[Zaire, 1]

Map-Reduce



Large jobs may require:
Multiple reducers,
partitioners, combiners

Combiners can be more complex: Build a partial ML model before transferring all training data to reducer

Sort all the GPS coordinates by distance (Remember MongoDB?)

Mappers are dependant on data, Programmer needs to specify number of combiners, partitioners, and reducers

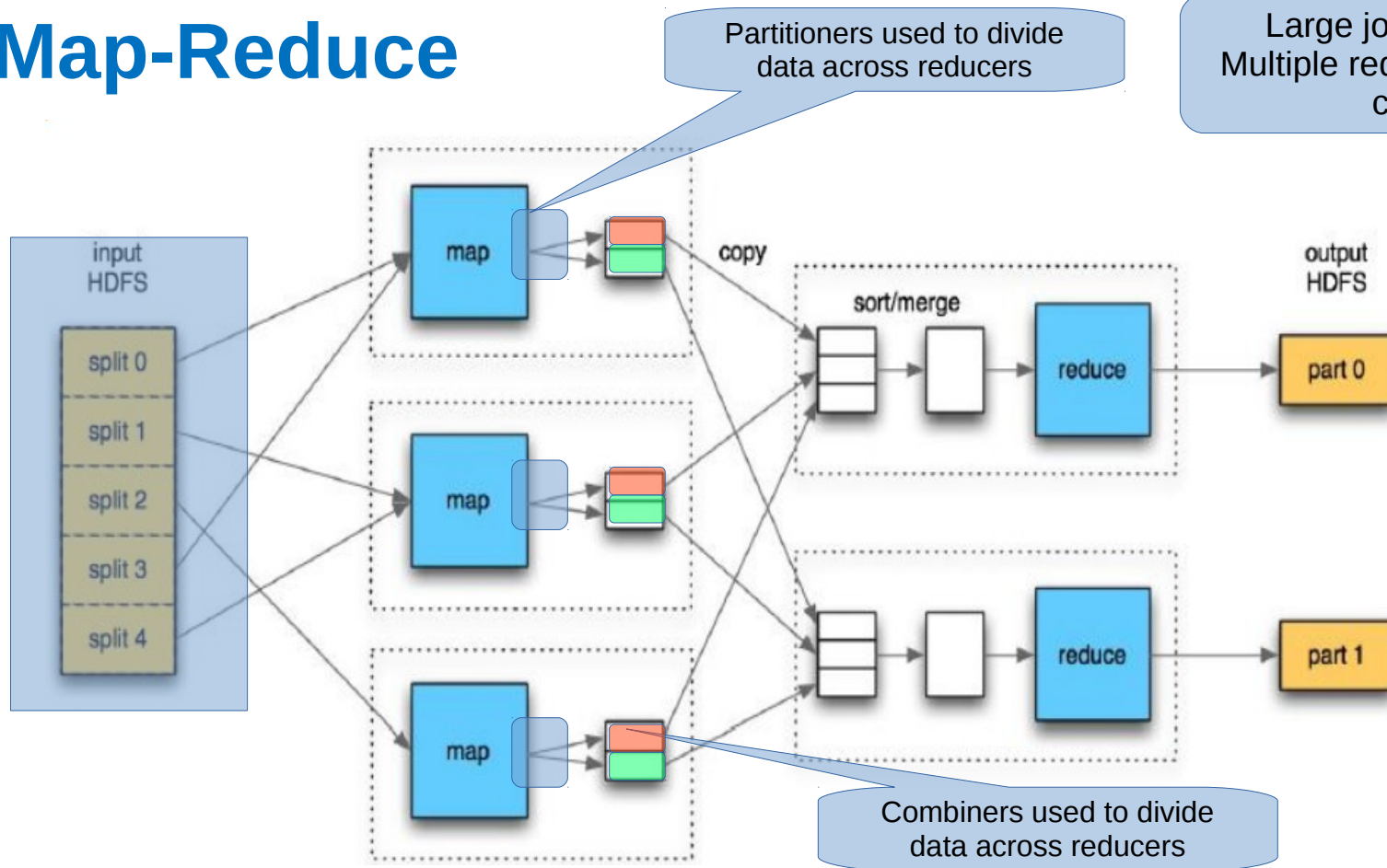
- Ex: word counting of 300K documents spread across 3 data nodes
 - Each data node:
 - Has ~100K documents, one mapper
 - Counts the word in its local set of 100K documents
 - Two reducers are used:
 - Reducer 1 to count all words starting with "A-M"
 - Reducer 2 to count all words starting with "N-Z"

- Further, let us say mapper 1 generates following counts by analyzing its files:
 - [Bombay, 1] [Alaska, 1] [Buffalo, 1] [Bombay, 1] [Bombay, 1] [Alaska, 1] [London, 1] etc
 - It will need to partition Alaska, Bombay, Buffalo data to reducer 1 and London Zaire data to reducer 2
 - To optimize the data, we will combine all counts together:
 - [Bombay, 3] [Buffalo, 1] [Alaska, 2] [London, 1]

- Reducer 1 and Reducer 2 get results from mappers
 - [Bombay, 3] [Buffalo, 1] [Alaska, 2]
 - [London, 1]
 - Similar results from all mappers will be used by reducers to compute the final results
- Part 0 and Part 1 will contain counts that the user needs



Map-Reduce



- Individual mappers do not talk to each other
- The mapper may decide to run or not run a combiner/partitioner

Use this to understand the rationale behind needing a combiner:

http://www.tutorialspoint.com/map_reduce/map_reduce_combiners.htm

<https://developer.yahoo.com/hadoop/tutorial/module4.html>

There are some debates on who runs first? <http://stackoverflow.com/questions/22061210/what-runs-first-the-partitioner-or-the-combiner>



The Hello World of MapReduce

Pseudo code:

```
map(String input_key, String input_value)
  foreach word w in input_value:
    emit(w, 1)
```

```
reduce(String output_key,
        Iterator<int> intermediate_vals)

  set count = 0
  foreach v in intermediate_vals:
    count += v
  emit(output_key, count)
```

Java code:

```
public void map(Object key, Text value, Context context
                ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
    }
```

Loop through all items in "value"

Map-reduce manager

```
public void reduce(Text key, Iterable<IntWritable> intermediateValues,
                  Context context
                  ) throws IOException, InterruptedException {

    int sum = 0;
    for (IntWritable val : intermediateValues) {
        sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
}
```

Loop through all intermediate results

The Hello World of MapReduce

Input to the mapper:

```
(3414, 'the cat sat on the mat')  
(3437, 'the aardvark sat on the sofa')
```

Output from mapper:

```
('the', 1), ('cat', 1), ('sat', 1), ('on', 1),  
( 'the', 1), ('mat', 1), ('the', 1), ('aardvark', 1),  
( 'sat', 1), ('on', 1), ('the', 1), ('sofa', 1)
```

Intermediate result sent to reducer:

```
('aardvark', [1])  
( 'cat', [1])  
( 'mat', [1])  
( 'on', [1, 1])  
( 'sat', [1, 1])  
( 'sofa', [1])  
( 'the', [1, 1, 1, 1])
```

Result from reducer to client (or HDFS):

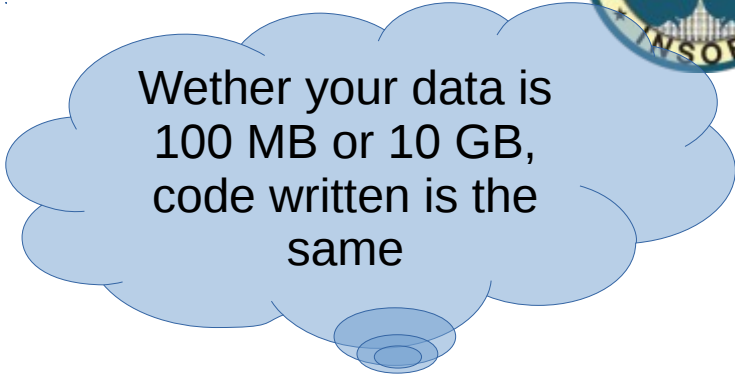
```
('aardvark', 1)  
( 'cat', 1)  
( 'mat', 1)  
( 'on', 2)  
( 'sat', 2)  
( 'sofa', 1)  
( 'the', 4)
```

The Hello World of MapReduce

```
map(String input_key, String input_value)
  foreach word w in input_value:
    emit(w, 1)
```

```
reduce(String output_key,
        Iterator<int> intermediate_vals)

  set count = 0
  foreach v in intermediate_vals:
    count += v
  emit(output_key, count)
```

A blue cloud shape with a white outline, containing text. It has several smaller circles at the bottom, suggesting a puff of smoke or a cloud rising.

Whether your data is
100 MB or 10 GB,
code written is the
same

Multiple tasks may be chained together as several HDFS/MR tasks

Example: Classify news articles into groups

HDFS: Crawler to get all comments from news websites

MR task: Remove ads, punctuations, HTML tags, store individual articles separately

MR task: Generate counts of certain words (TF-IDF computation), or other features from each article

MR task: Classify if an article is about: education, politics, science, sports etc



The Hello World of MapReduce

A challenge is that each MR step writes to a HDFS system

If an algorithm requires an MR step to be iterative, the process will slow down due to several read-writes to HDFS

HALOOP prevents this

Multiple tasks may be chained together as several HDFS/MR tasks

Example: Classify news articles into groups

HDFS: Crawler to get all comments from news websites

MR task: Remove ads, punctuations, HTML tags, store individual articles separately

MR task: Generate counts of certain words (TF-IDF computation), or other features from each article

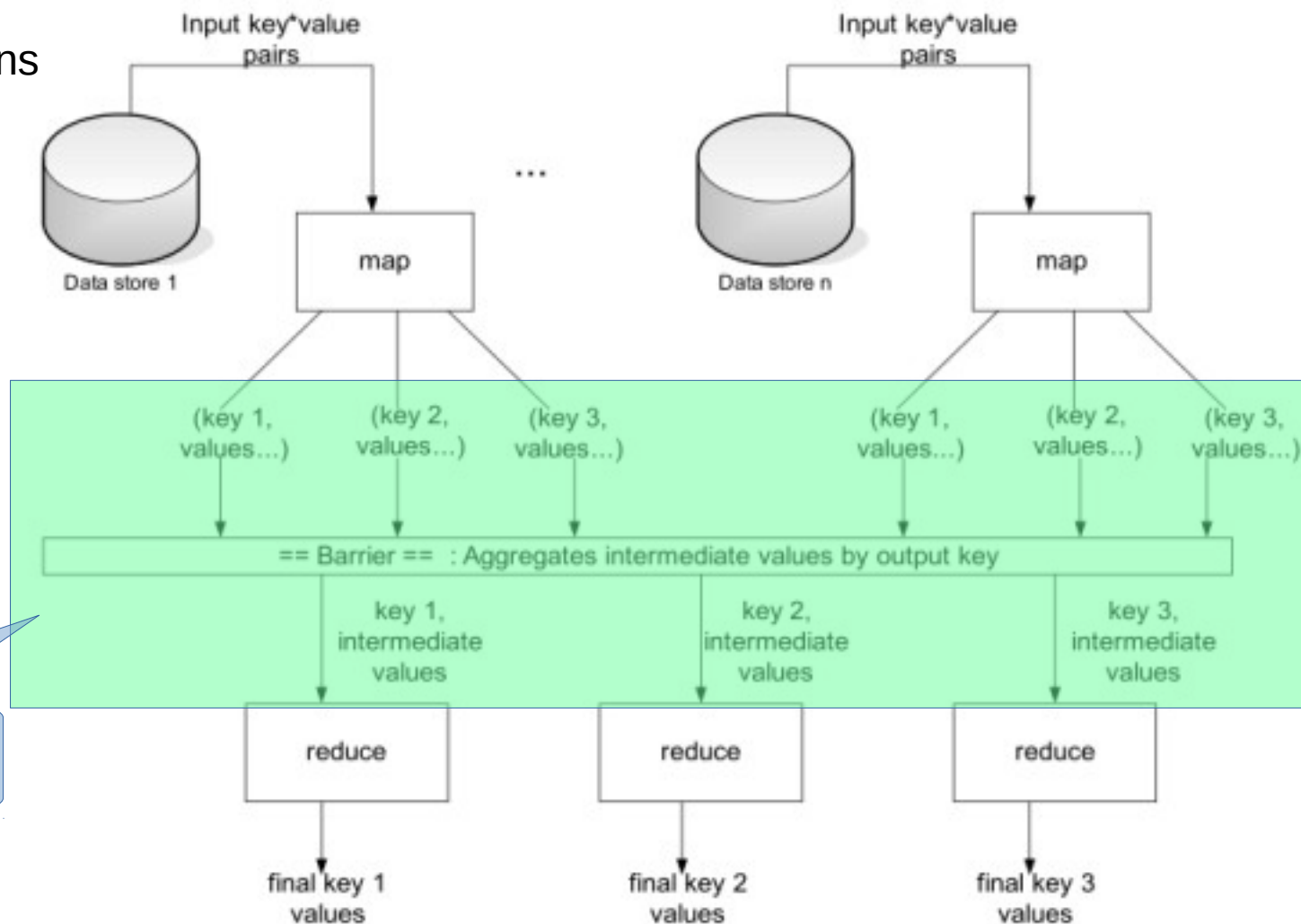
MR task: Classify if an article is about: education, politics, science, sports etc

http://sigmod.github.io/papers/HaLoop_camera_ready.pdf

MR Keys and Values

Understanding design patterns and OOP may be critical to good MapReduce

How can you reuse, extend, and simplify someone else's map-reduce code?



Partitioners and combiners may be present here

http://www.nataraz.in/data/ebook/hadoop/mapreduce_design_patterns.pdf

<https://www.safaribooksonline.com/library/view/mapreduce-design-patterns/9781449341954/ch01.html>

MR Keys and Values



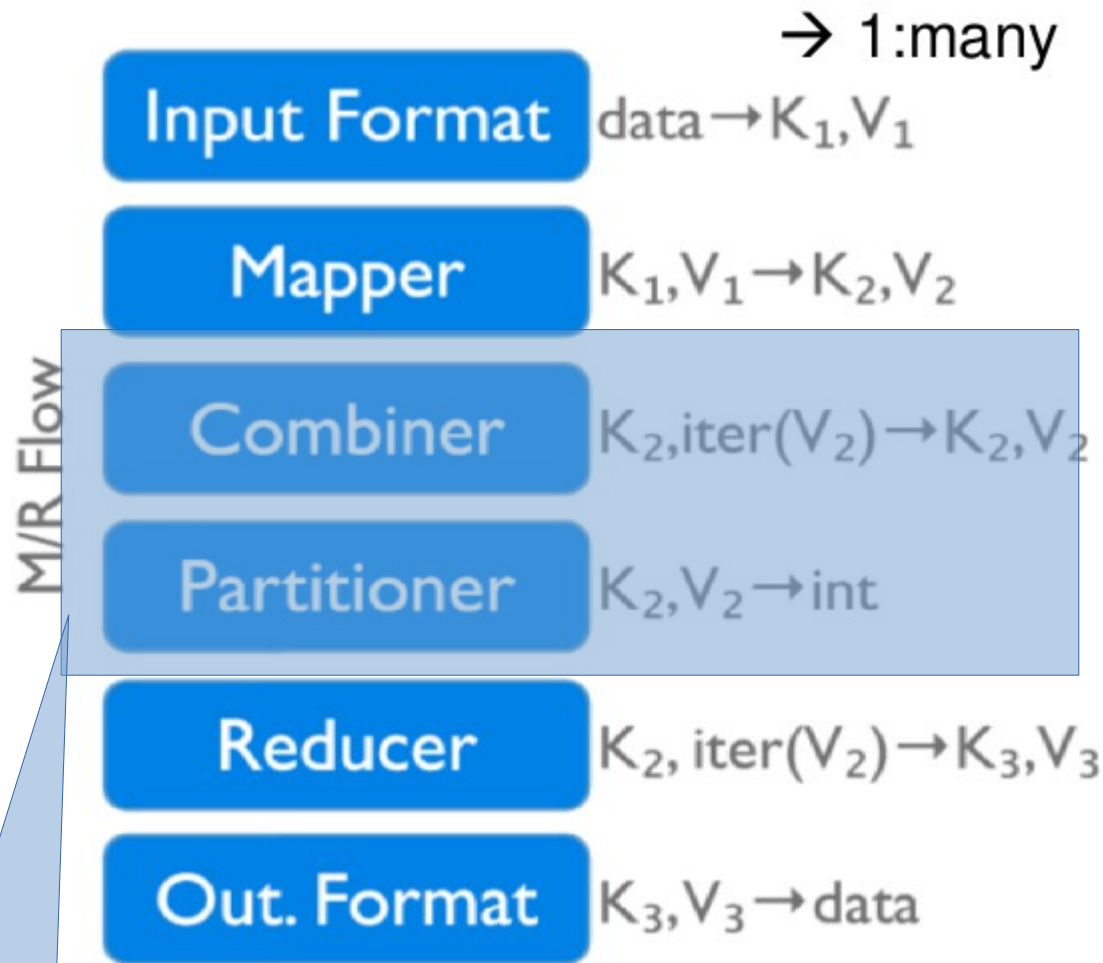
- Programmers specify two functions:
 - $\text{map}(k, v) \rightarrow \langle k', v' \rangle^*$
 - $\text{reduce}(k', v') \rightarrow \langle k', v' \rangle^*$
- All values with the same key are reduced together
 - E.x.: Key: (Bombay, Buffalo, Hyderabad), value: Number of times it occurs in a piece of text
- Keys and Values in Hadoop are *Objects*
- Values are objects which implement *Writable*
- Keys are objects which implement *WritableComparable*



Mappers

- Mappers run on nodes which hold their portion of the data locally, to avoid network traffic; a few exceptions:
 - In case a DN/Container is busy, the data and job may be moved to a DN/Container that is maximum 2-hops away
 - There may be dedicated processing and storage nodes (Esp in memory heavy apps)
- Multiple Mappers run in parallel, each processing a portion of the input data
- Mapper reads data in the form of key/value pair
 - Mapper may use, or completely ignore, the input key.
 - E.g., The task could be to read a paragraph from file at a time, while performing some operation like stemming. Key then is the byte offset into the file at which the paragraph starts. Value is processed text itself. In this case, the key is irrelevant.
- It outputs zero or more key/value pairs
 - `let map(k, v) = emit(k.toUpper(), v.toUpper())`
 - `('foo', 'bar') -> ('FOO', 'BAR')`

Data Flow in a MapReduce Program



- Input Format may need to be complex
 - A key-value pair may have been divided across 64/128MB blocks in HDFS
 - Imagine a json file that was split
- There may be a few exceptions to this sequence
- Now, let us see what is Input Format and Output Format

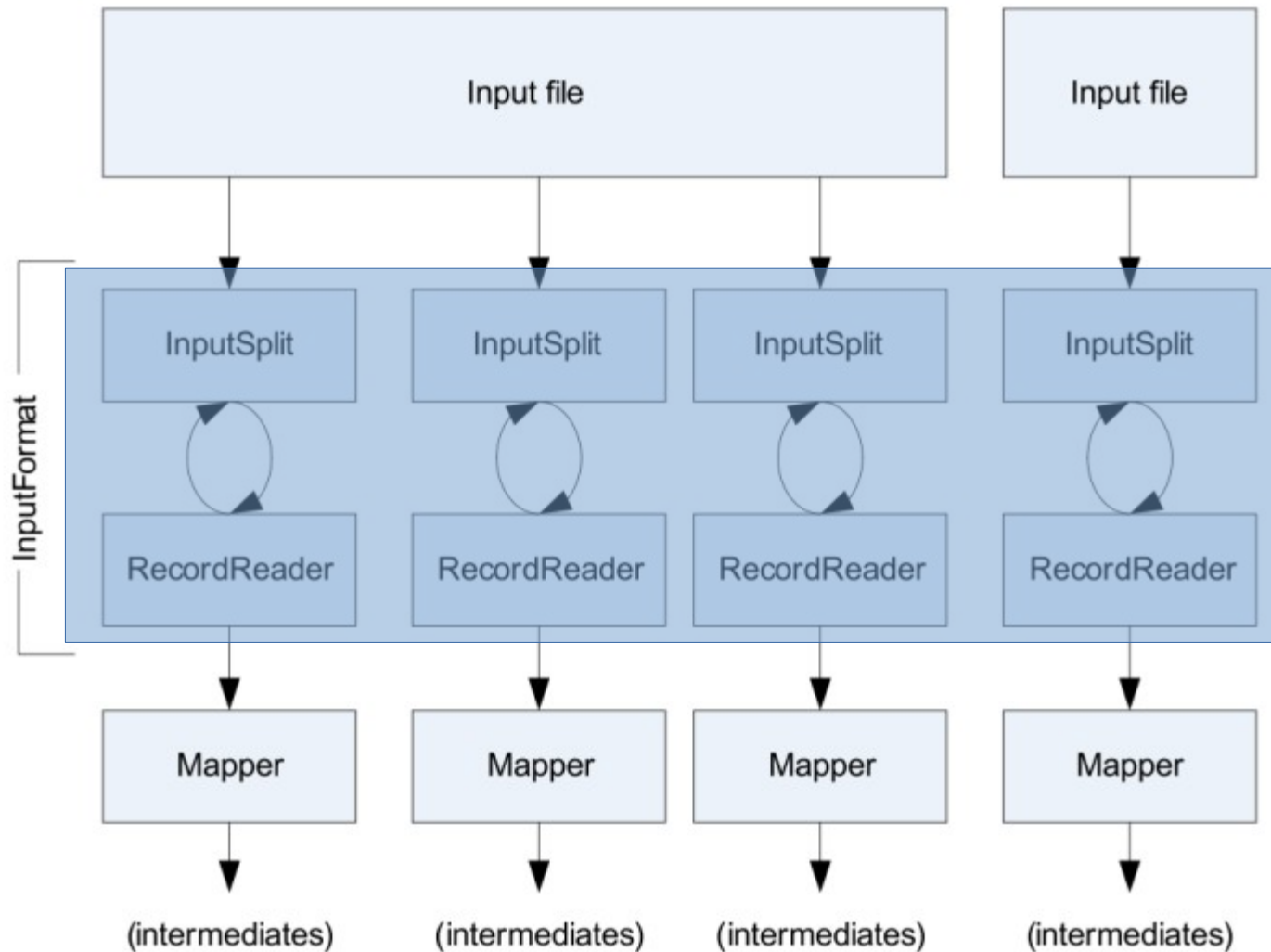
Sequence can
change



InputFormat and OutputFormat: Basics

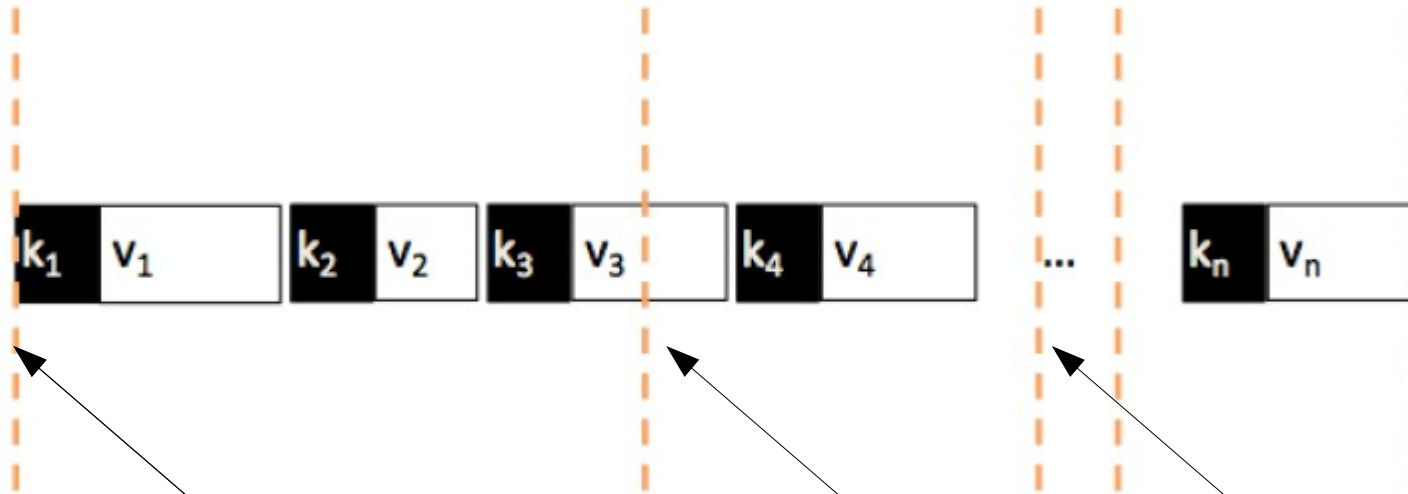
- **Serialization:** Converting in memory data structures into a format suitable for writing to i/o
 - i/o can be HDD, network, or any other port
- **De-serialization:** Converting bytes read from i/o into in-memory data structures
- **Trivially serializable data:**
 - Text, XML, or JSON
 - Arrays
- **Some data will need special processing to serialize:**
 - Any data that has memory pointers
 - Linked lists

Input Format



- InputSplit assumes an approximate **size** of “key-value” pair
- Uses **size** to read a block, spit it and process it in parallel
- RecordReader knows what types of records you are looking for, delimiters etc
- Neighboring TTs can have InputFormats that can talk to each other

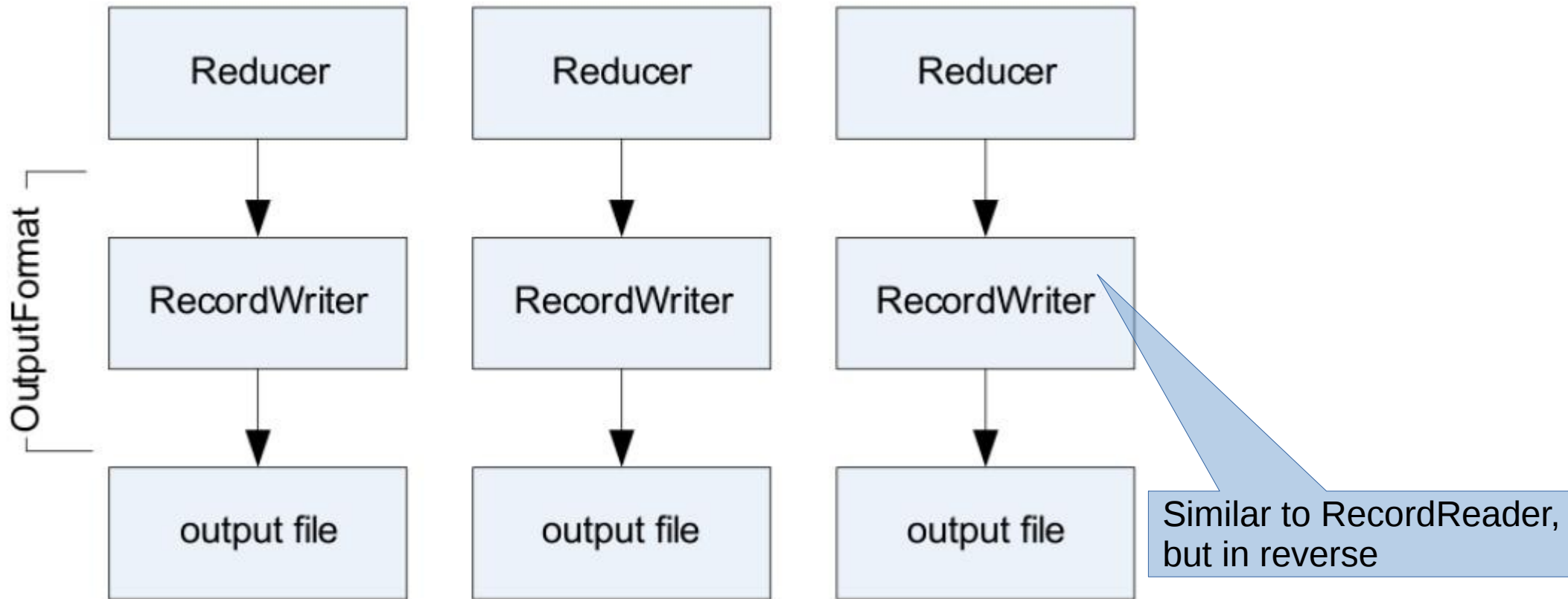
Sample Input Splits



InputSplits

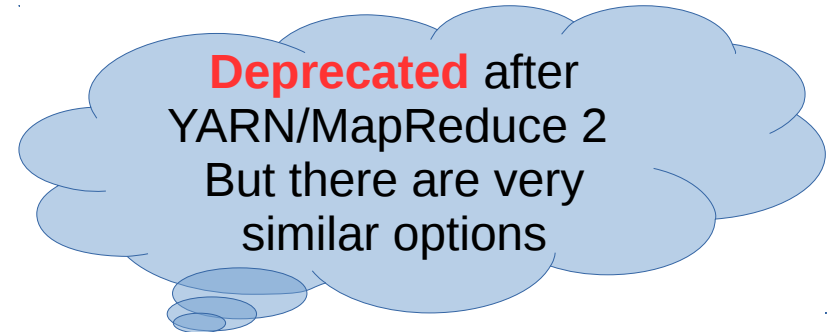
RecordReader intervenes here to decide if the key-value is complete or not

OutputFormats and RecordWriters



The Distributed Cache

- Often times, a Mapper or Reducer needs access to “side data”
 - Dictionary
 - GPS to city name mapping
 - Configuration values
- One option is to read directly from HDFS
 - But this is not scalable
- Distributed cache provides API to push data to all nodes
 - Transfer happens behind scene before task execution
 - Cache is read-only
 - DistributedCache files are deleted after job execution



The Distributed Cache

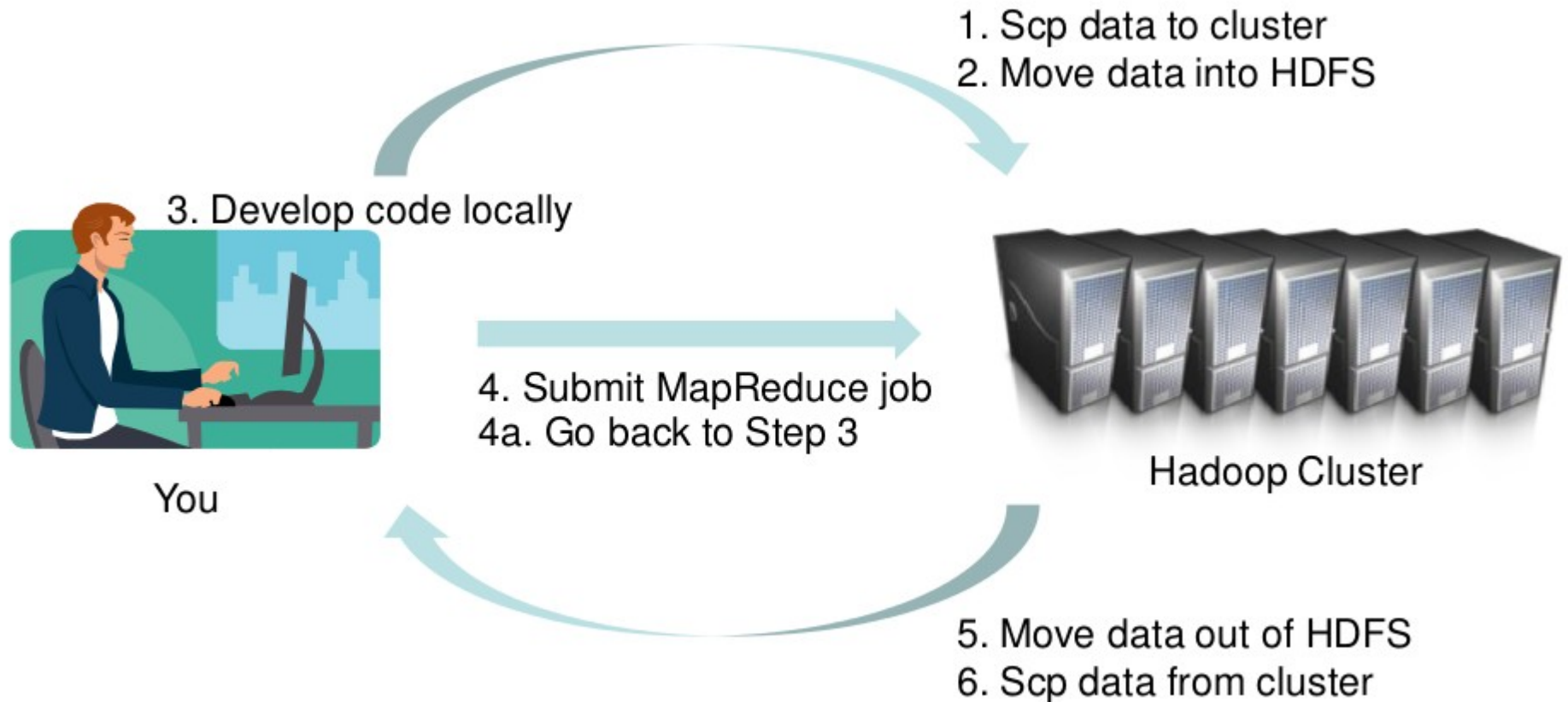
- Use the `-files` option to add files

```
hadoop jar myjar.jar MyDriver -files file1, file2, file3, ...
```

- **Files added to the DistributedCache are made available in your task's local working directory**
 - Access them from your Mapper or Reducer the way you would read any ordinary local file

```
File f = new File("file_name_here");
```

Map Reduce Development Cycle





Some features of map-reduce jobs

- MapReduce jobs tend to be relatively short in terms of lines of code
- It is typical to combine multiple small MapReduce jobs together in a single workflow
 - Oozie
- You are likely to find that many of your MapReduce jobs use very similar code



Map-Reduce Refresher

- All algorithms must be expressed in m, r, c, p
- The execution framework handles everything else...
 - Scheduling: assigns workers to map and reduce tasks
 - Data distribution: moves processes to data
 - Synchronization: gathers, sorts, and shuffles intermediate data
 - Errors and faults: detects worker failures and restarts
- You don't know:
 - Where mappers and reducers run
 - When a mapper or reducer begins or finishes
 - Which input a particular mapper is processing
 - Which intermediate key a particular reducer is processing



How well does it Work?

- A 2009 study at Yahoo!, lost 650 out of 329 Million blocks on 10 clusters with 20K nodes
 - 7-9's of reliability
 - The 650 losses reduced to 19 after some bug fixes
- NN is very robust and can take a lot of abuse
 - NN is resilient against overload caused by misbehaving apps
- However, there are limitations
 - These were resolved using High Availability (HA) NameNode

<http://hortonworks.com/wp-content/uploads/2012/01/HA-NameNode.pdf>

<http://blog.cloudera.com/blog/2012/03/high-availability-for-the-hadoop-distributed-file-system-hdfs/>



CDH3 HDFS and Map Reduce: Limitations

- Scalability:
 - Max cluster size: 4000 nodes
 - Max concurrent tasks: 40,000
 - Coarse synchronization in AppMaster
- Single point of failure
 - Failure kills all queued and running jobs
 - Jobs restart on reboot of RM or MRAppMaster



Map Reduce Limitations – Contd.

- Hard partition of containers into map and reduce
 - Low resource utilization
- Lacks support for:
 - Iterative applications (10x slower)
 - Hacks for graph/MPI
- Lack of wire-compatible protocols
 - Client and cluster must be same version
 - Applications and workflows cannot migrate to different clusters

Quick Recap



- HDFS:

- NameNode, Secondary NameNode, DataNode, Blocks, HeartBeat
 - Few defaults: 128MB block size, heartbeat every 3 secs
- High Availability HDFS (CDH3) splits NameNode into: NameNode (active), NameNode (standby), control maintained using zookeeper

- MR:

- Mapper, Reducer, Partitioners, Combiners, InputFormat, OutputFormat
- YARN: ResourceManager, NodeManager, Container, AppMaster
 - A few defaults to remember: Tasks on failed containers are restarted by AppMaster 4 times, Apps on failed AppMasters are restarted by RM 2 times

YARN EcoSystem

- Apache Giraph: Graph Processing
- Apache HAMA – BSP
- Apache MapReduce – Batch
- Apache Tez – Batch/Interactive
- Apache S4 – Stream Processing
- Apache Samza – Stream Processing
- Apache Storm – Stream Processing
- Apache Spark – Iterative Applications
- Elastic Search – Scalable Search
- Cloudera Llama – Impala on YARN
- DataTorrent – Data Analysis
- HOYA – HBase on YARN



Frameworks powered by YARN

Apache Twill

Apache REEF (Previously Microsoft REEF, stdlib of YARN)

Spring for Hadoop 2 (manage MR, Hive and Pig jobs)

<http://twill.apache.org/>
<http://reef.apache.org/>

Yet Another Resource Negotiator (YARN)

- YARN provides the daemons and APIs necessary to develop generic distributed applications of any kind
- YARN handles and schedules resource requests (such as memory and CPU) from applications, and supervises their execution
- YARN can run applications that do not follow the Map Reduce model
 - Having built a cluster, some admins may want to run programs that do not fall into MR
 - MR2 is modeled as “just another” client application on YARN
- YARN is like the component of an OS that performs context switching and resource management



Agenda

- MapReduce and YARN
- Spark

Intro to Spark



- **Key Concept:**

- HDFS shares Hard disk of multiple nodes, Spark shares RAM
- HDFS had distributed files, Spark enables distributed data structures
- Write programs in form of transformation on distributed datasets

Datastructures

Resilient Distributed Datasets (RDD)

Objects spread across a cluster, stored in RAM, sometimes on disk

Built through parallel transformations

Automatically rebuilt on failure

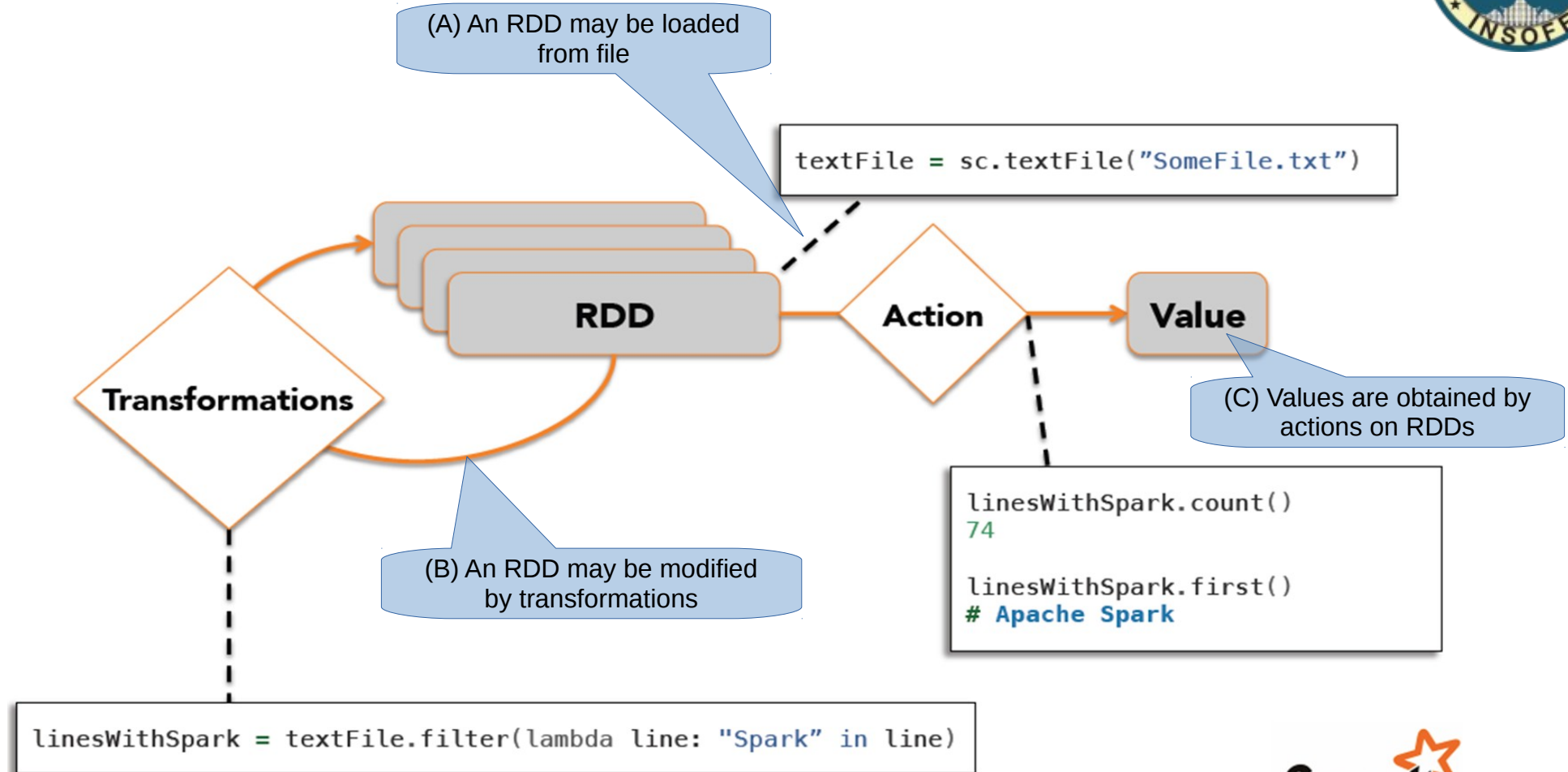
Originally published in 2012 at UC Berkeley: https://people.eecs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

Operations

Transformations (e.g.: Map, filter, groupBy)

Actions (e.g.: Count, collect, save)

Working with RDDs



In case of failure, Spark recreates RDD by repeating file reads or transformations
RDDs are deleted when: application finishes running, or application deletes an RDD explicitly

<http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence>



Spark: Fault Tolerance

- Checkpointing and multiple persistence: Just like in HDFS, multiple nodes store the data
 - Multiple storage options: MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY, MEMORY_ONLY_SER (memory only serialized), MEMORY_AND_DISK_SER
 - A) Replication: MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc. Same as in (A), but replicated on two nodes
- WriteAhead Logs (or journalling): Any operation performed on an RDD is first written to a log, in case of failure the steps may be repeated
 - `spark.streaming.receiver.writeAheadLog.enable` to true (default is false)

<http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence>

<https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html>

<http://spark.apache.org/docs/latest/streaming-programming-guide.html#fault-tolerance-semantics>

Spark: Example

- Load error messages from a log into memory and search for patterns

Base RDD

Transformed
RDD

Transformed
RDD

```
val lines = spark.textFile("hdfs://...")  
val errors = lines.filter(_.startsWith("ERROR"))  
val messages = errors.map(_.split('\t')(2))  
messages.cache()
```

Cache the
RDD

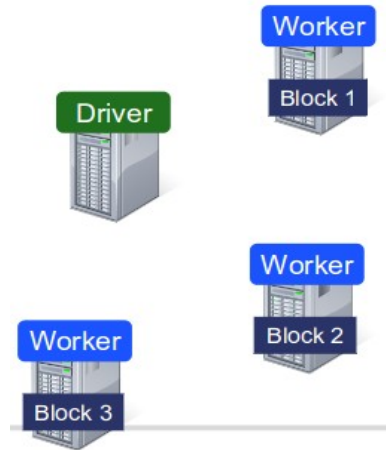
```
messages.filter(_.contains("mysql")).count()  
messages.filter(_.contains("php")).count()
```

Action(s) on
the RDD

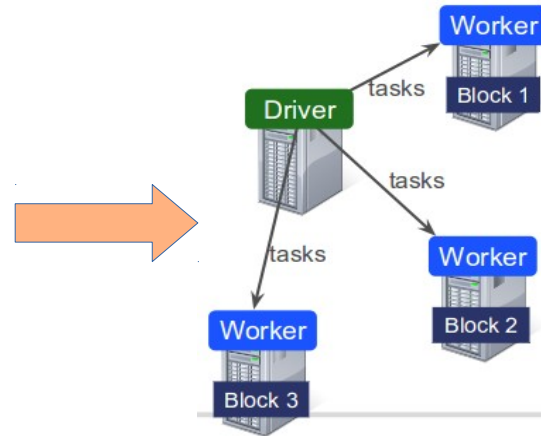
There may be many transformations: Base readers, filters, maps that may be implemented on HDFS, MR, or BSP

Similarly, many actions

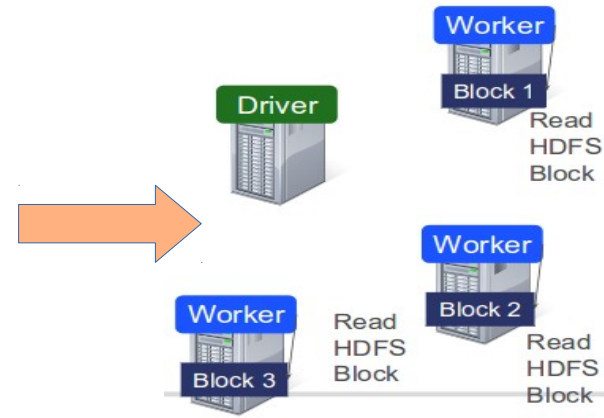
Spark: Flow



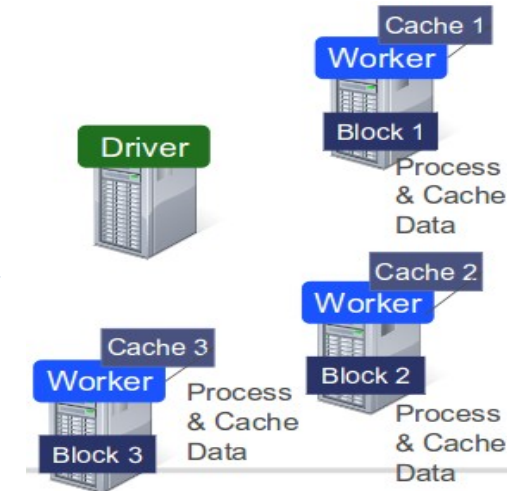
(A) Driver gets the task



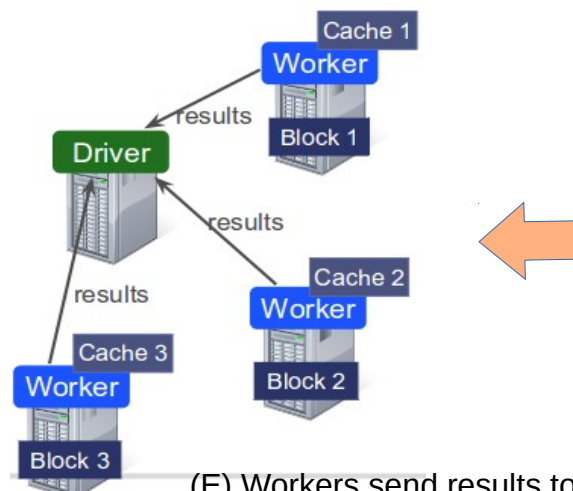
(B) Tasks are sent to workers where the blocks are located



(C) Workers read the blocks into memory



(D) Workers process and cache the data



(E) Workers send results to driver

If multiple operations are to be performed, better to cache the data

1 TB of data to search
~8 seconds from cache,
vs ~150s from disk

https://stanford.edu/~rezab/sparkclass/slides/itas_workshop.pdf
<http://stanford.edu/~rezab/sparkworkshop/slides/holden.pdf>

<http://www.insofe.edu.in>

International School of Engineering

Plot 63/A, 1st Floor, Road # 13, Film Nagar, Jubilee Hills, Hyderabad - 500 033

For Individuals: +91-9502334561/63 or 040-65743991

For Corporates: +91-9618483483

Web: <http://www.insofe.edu.in>

Facebook: <https://www.facebook.com/insofe>

Twitter: <https://twitter.com/Insofeedu>

YouTube: <http://www.youtube.com/InsofeVideos>

SlideShare: <http://www.slideshare.net/INSOFE>

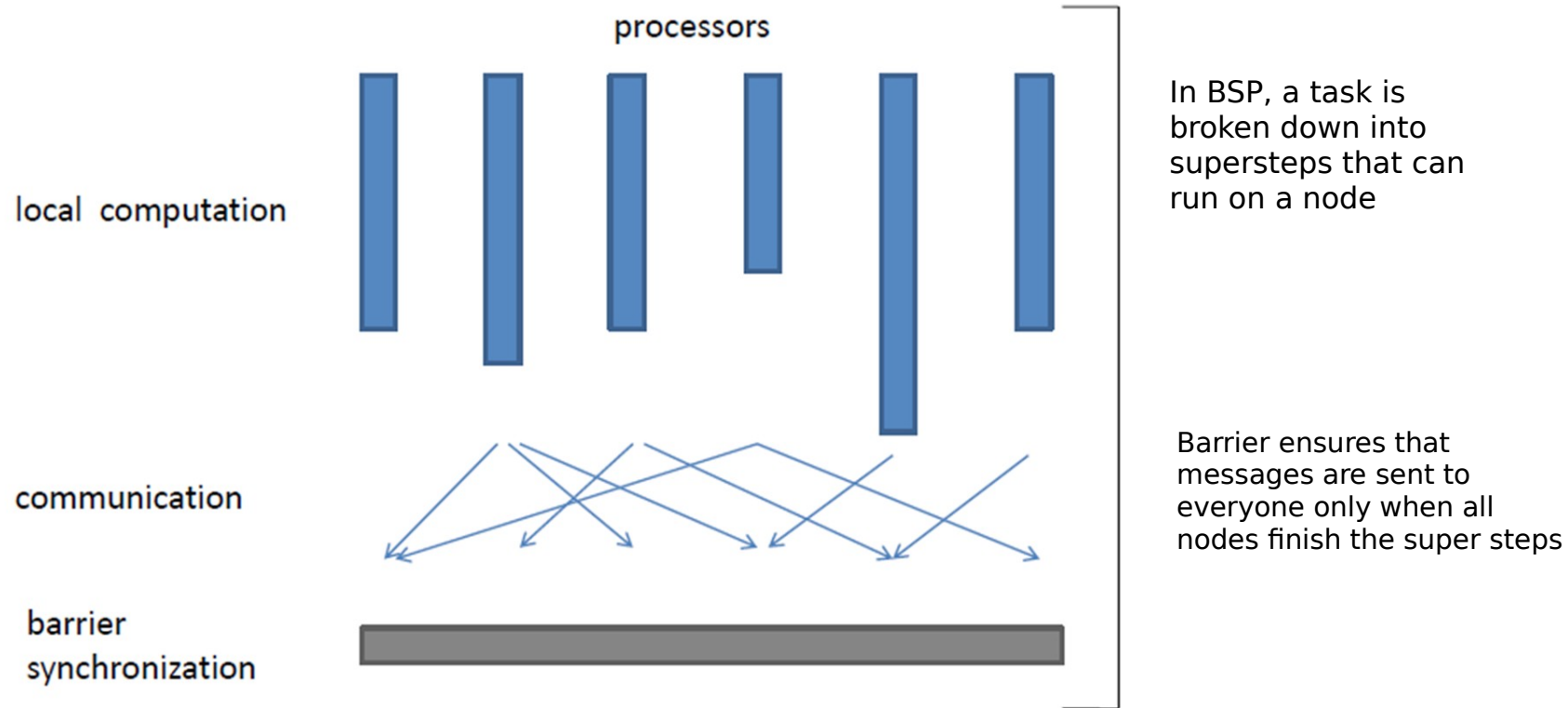
LinkedIn: <http://www.linkedin.com/company/international-school-of-engineering>



Additional Material

Bulk Synchronous Processing

Bulk Synchronous Processing (BSP)



- Developed in the 1990s
 - Parallel local computation
 - Synchronized peer to peer communication

Example (max of numbers, matrix multiplication): <http://sbrinz.di.unipi.it/~peppe/FilesPaginaWeb/BSP.pdf>

https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

Bulk Synchronous Processing (BSP)

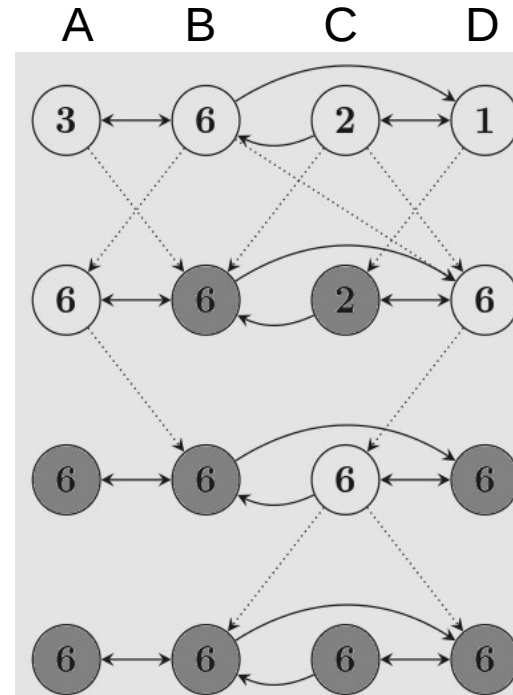
- Input:
 - Directed edge
 - Each vertex associated with id and value
 - Edges may also have values
- Edges have no computation
 - Vertices may modify its value, active/halt state or edges
- Computation ends when all vertices reach halt state



Active vertex



Halted vertex



Superstep 0

D gets message from B, C
C from D
B from C, A
A from B
B and C decide they are large A and D change their values

Superstep 1

B, D, A decide they are max, C changes value

Superstep 2

B, D, A, C decide they are max

Superstep 3

Everyone is in halt stage, read out max value

Maximum value example, dotted lines indicate messages, dark lines indicate edges

https://kowshik.github.io/JPregel/pregel_paper.pdf

<https://www.cs.duke.edu/courses/spring13/compsci590.2/slides/lec14.pdf>



Examples of Mappers and Reducers

Explode mapper

- Ex: Output each character separately, Useful in text mining, where we output a word separately

```
let map(k, v) =  
  foreach char c in v:  
    emit (k, c)
```

```
('foo', 'bar') -> ('foo', 'b'), ('foo', 'a'),  
                  ('foo', 'r')  
  
('baz', 'other') -> ('baz', 'o'), ('baz', 't'),  
                    ('baz', 'h'), ('baz', 'e'),  
                    ('baz', 'r')
```

Filter Mapper

- Apply filter on value
 - Example: A certain word is present, the value is prime number, bloom filter (How likely is the value?)

```
let map(k, v) =  
  if (isPrime(v)) then emit(k, v)
```

```
('foo', 7) -> ('foo', 7)  
('baz', 10) -> nothing
```

Changing Key Spaces Mapper

- Output word length as key
- Ex: Sort dictionary entries, Extract features from “value” part

```
let map(k, v) =  
    emit(v.length(), v)
```

```
('foo', 'bar') -> (3, 'bar')  
( 'baz', 'other') -> (5, 'other')  
( 'foo', 'abracadabra') -> (11, 'abracadabra')
```



Reducer

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
 - There may be one or more reducers
 - If a reducer is assigned handle a particular intermediate key, all key-value pairs of that key will go to that reducer only
 - Intermediate keys and value lists go to reducer in sorted order, and they can come from multiple mappers
 - Hence this step is called “shuffle and sort”
- Remember, we spoke about reducer result being written to HDFS
 - HALOOP lets you combine multiple MR without writing to HDFS

Sum Reducer

- Add all values associated with a key

```
let reduce(k, vals) =  
  sum = 0  
  foreach int i in vals:  
    sum += i  
  emit(k, sum)
```

```
('bar', [9, 3, -17, 44]) -> ('bar', 39)  
('foo', [123, 100, 77]) -> ('foo', 300)
```

Other examples: Identify (pass the data with no changes)
Explode reducer?



Simple Examples of MR

- Grep: Find all lines matching some pattern
 - Mapper: Looks at each line and outputs it if it matches the pattern
 - Reducer: Identity, or do nothing
- Count of URL Access Frequency:
 - Mapper (url, 1), reducer: (url, sum)
- Reverse Web-Link Graph: For a URL (target), get all URLs that point to it (source)
 - Mapper: emit(target, source), reducer(target, cons) => {target, [sources]}
- Term-Vector per Host - {hostname, [term vector]}
- Inverted Index
- Distributed Sort

Bulk Synchronous Processing (BSP)

- Example: Page rank
- Note: In map-reduce Task Trackers cannot talk to each other
- BSP (HAMA) allows you to do that

```
class PageRankVertex
    : public Vertex<double, void, double> {
public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
                sum += msgs->Value();
            *MutableValue() =
                0.15 / NumVertices() + 0.85 * sum;
        }

        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue() / n);
        } else {
            VoteToHalt();
        }
    }
};
```

<https://prezi.com/tabqzlvzohii/apache-hama-introduction/>
<http://arasan-blog.blogspot.in/>

Bulk Synchronous Processing (BSP)

HDFS

BSP

Map-Reduce

HAMA



Suitable for iterative tasks

BSP

Giraph



Primarily for graph processing

<https://prezi.com/tabqzlvzohii/apache-hama-introduction/>
<http://www.hadoopsphere.com/2015/06/large-scale-graph-processing-with.html>
<http://arasan-blog.blogspot.in/>