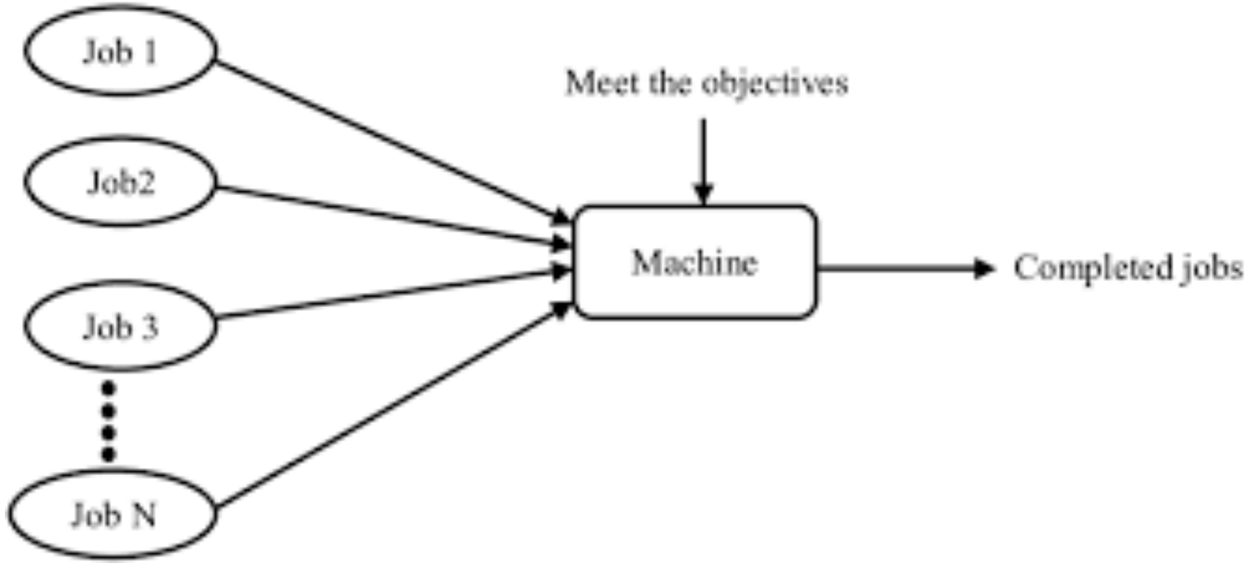
MapReduce

Ravi Kumar Gupta

https://kravigupta.in

Why we need MapReduce?

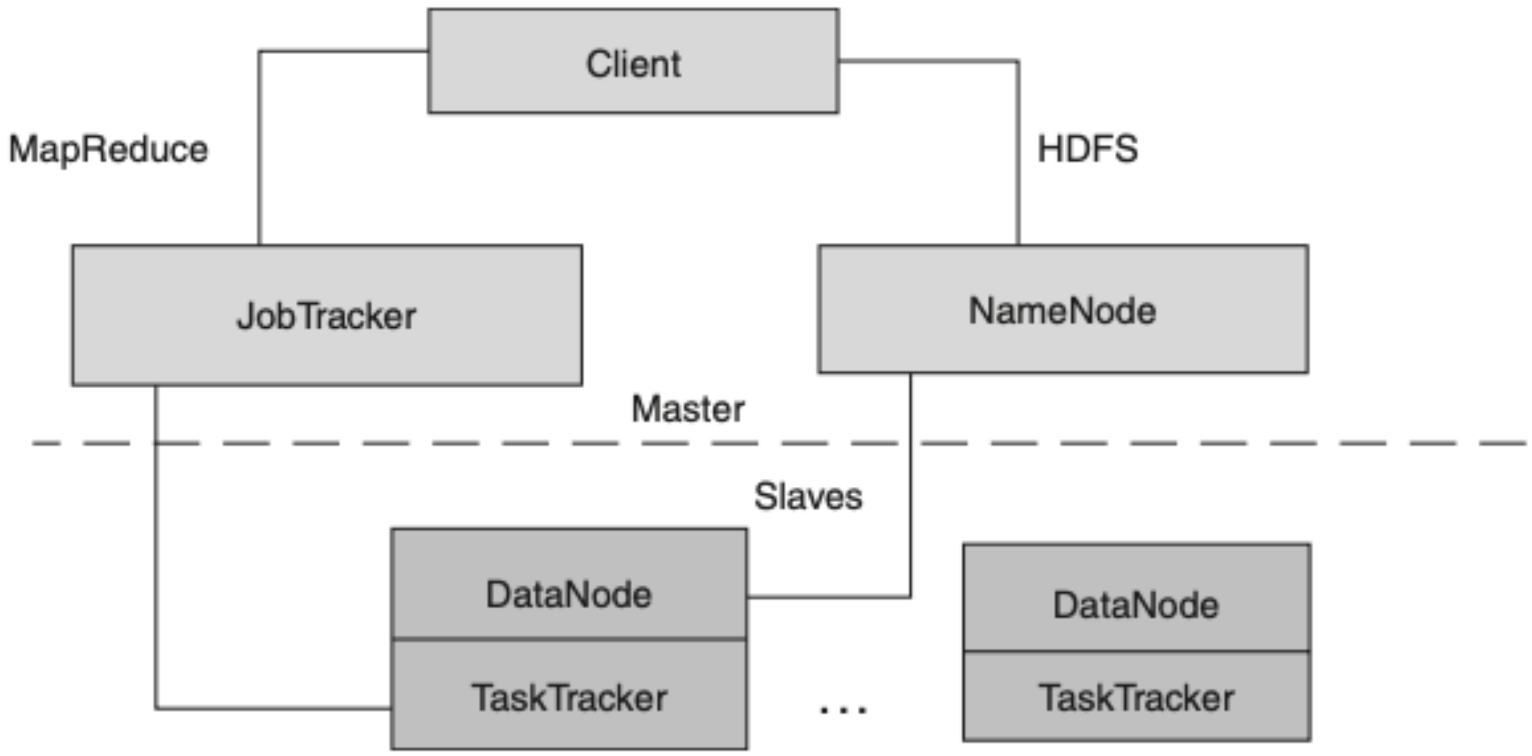
- Need to analyse and process
  - Massive amount of data
  - In a short period of time
- If done on a single machine will take huge amount of time



Why we need MapReduce?

- Idea is to divide the work
  - Divide the data into smaller chunks
  - Send it to a cluster of machine
  - Process simultaneously
  - Combine the results after processing
- Achieving parallelism by using several commodity hardware
  - Connected by ethernet or switches.
- Hadoop provides
  - DFS – distributed file system
    - Splits the data and sends to all nodes in clusters
  - MapReduce does the computation in parallel

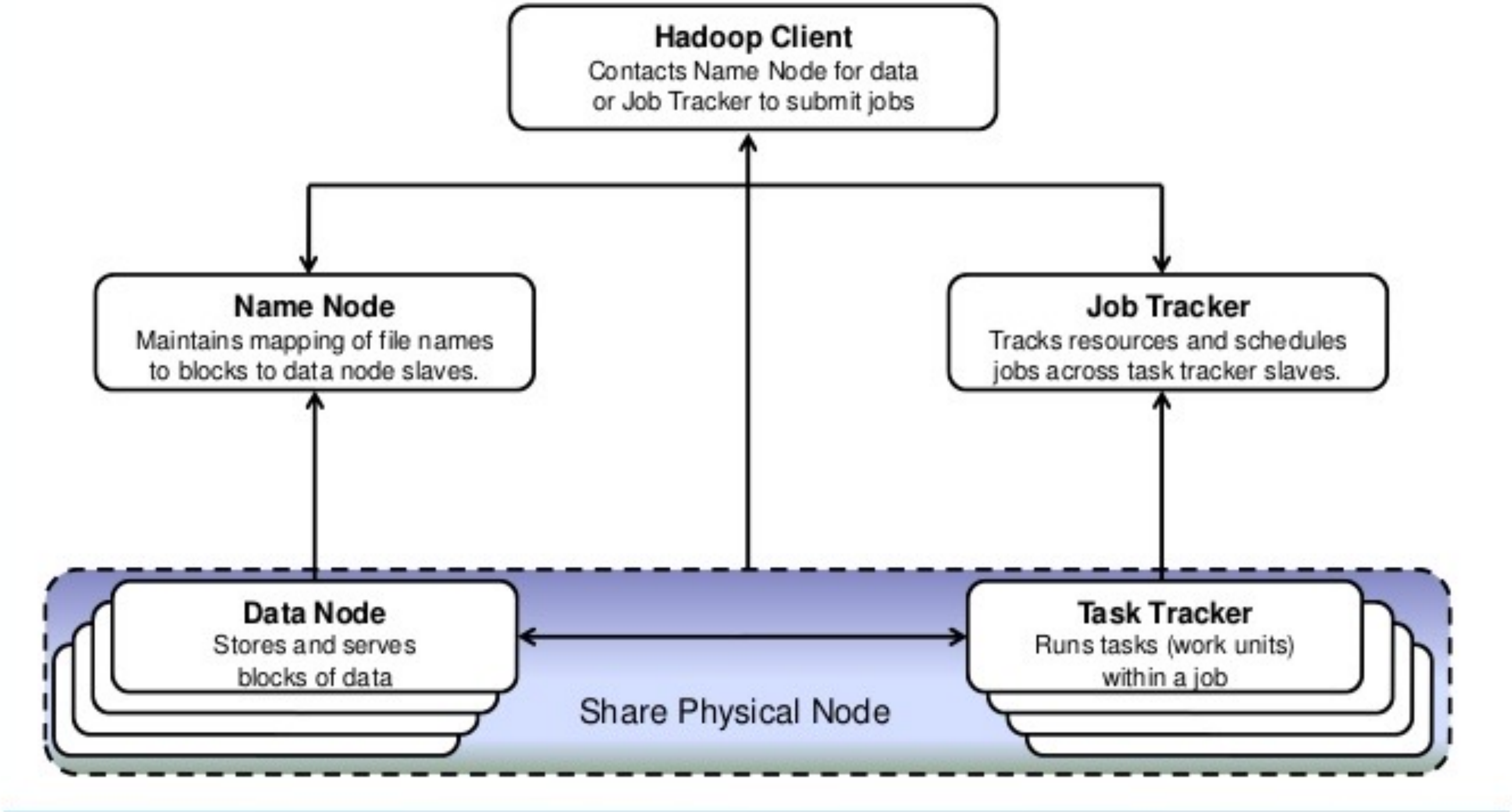
Hadoop - High Level Architecture



Hadoop - High Level Architecture

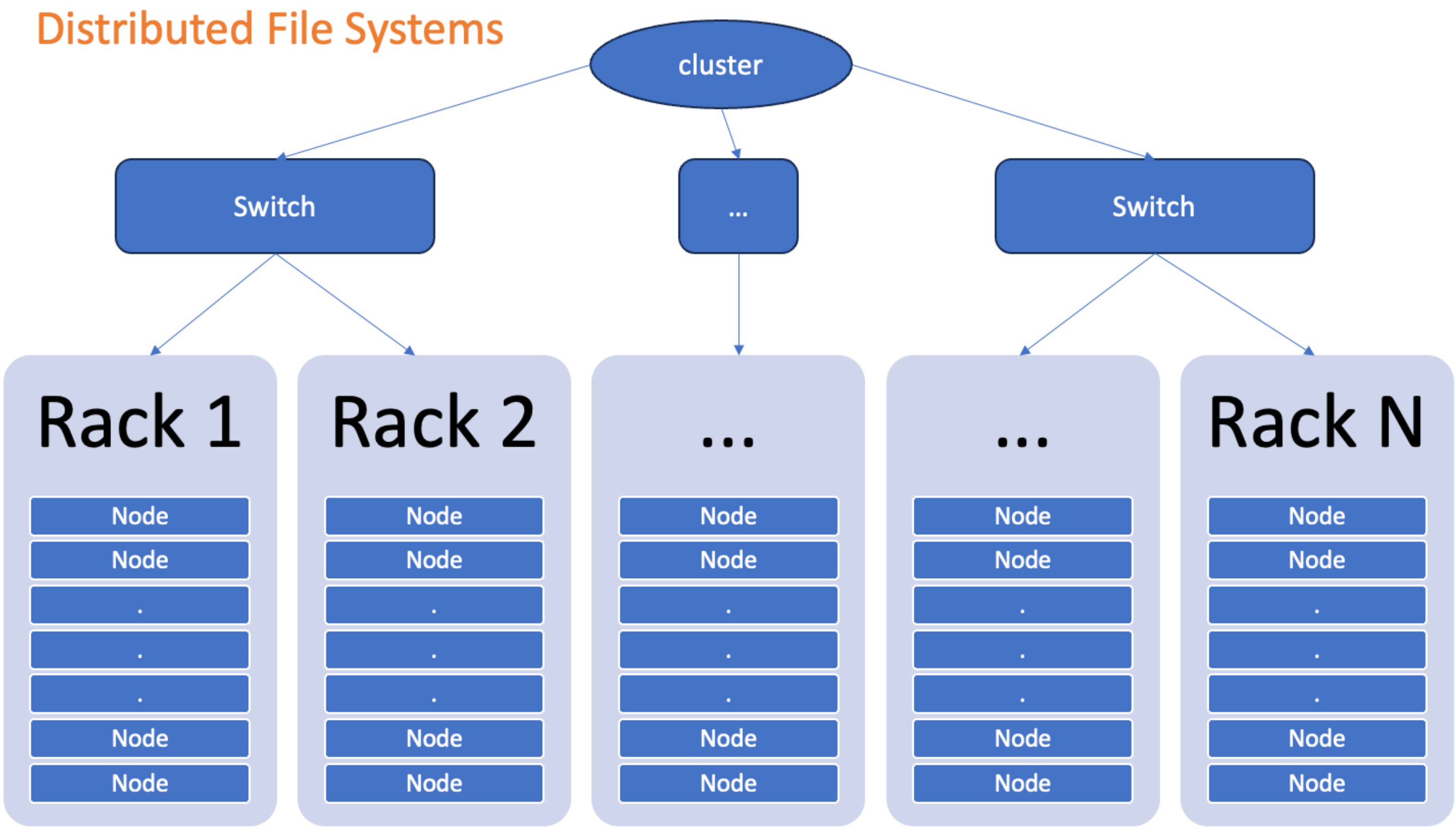
- Master node
  - NameNode Coordinates and monitors the data storage function
  - JobTracker coordinates the parallel processing of data using MapReduce
- Slave node
  - Runs a DataNode and TaskTracker
  - DataNode Takes instruction from NameNode
  - Does the actual work
    - Stores the data
    - Runs computation on data
- DataNode is a slave to NameNode
- TaskTracker is a slave to JobTracker

More detailed view ..



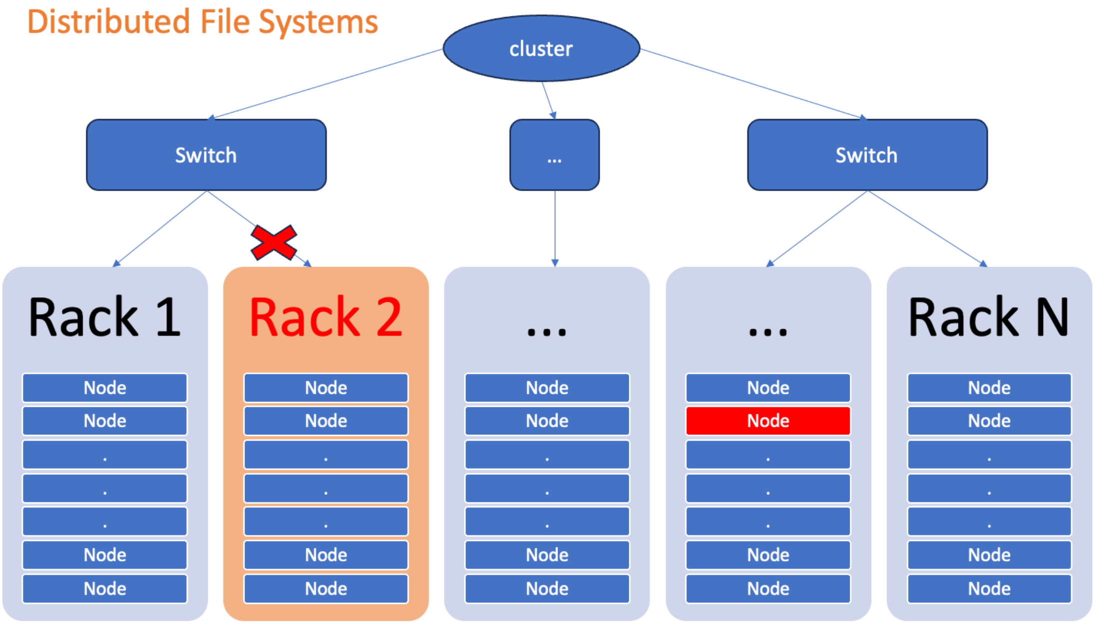
Distributed File System

- Uses **Cluster computing**
  - The new parallel computation architecture
  - Compute nodes in range of **8-64** are stored in racks and connected
  - Connected by switch or ethernet



Distributed File System

- **Failures** at a node are taken care by the replication
  - Failure can be node level e.g. disk, cpu
  - Or it can be rack level e.g. network failure, power failure
- All tasks are completed **Independently**
- Any node can be restarted without affecting computation on other nodes



Distributed File System

- DFS supports access to files stored on remote servers
- Offers replication and local caching
- Concurrent access can be taken care by locking conditions
- Different types of implementations – based on complexity of the application.

DFS

- What do we mean by **Locking Conditions** ?

- Mechanisms or rules used by DFS to manage concurrent access.
- Help avoid data inconsistency or corruption
- e.g. Allow multiple clients to read but block writes. Only one client will be able to write at once.

DFS - Google File System

- Google had to store a massive amount of data
- Stored on commodity hardware – not so reliable, hence redundant storage
  - Reduced cost because commodity hardware
- Mostly read operations, sometimes append
- Requires large streaming reads
  - High sustained throughput is needed with low latency
- File sizes are in GBs but stored as 64MB chunks with rep factor 3.
- Chunks managed through a single master node
  - Master stores the metadata of these chunks

DFS - Google File System

- Metadata contains
  - File and chunk namespaces
  - Mapping of the file to chunks
  - Locations of the replicas of each chunk
- Master is replicated in a shadow master – in case master fails
- Master chooses one of the replicas as primary and delegates the authority for taking care of the data mutation.

DFS – Hadoop Distributed File System

- Very Similar to GFS
- Master is called NameNode
- Shadow Master is called Secondary NameNode
- Chunks are called blocks
- Chunk Server is called DataNode
- DN stores and retrieve blocks and also reports the list of blocks to NN
- No appends like GFS
- Open Source

Organization of Nodes

- Hadoop runs best of Linux
- For smaller clusters where nodes are < 40
  - Single physical server can host both NameNode and JobTracker
- For bigger clusters,
  - Both of them can be on different physical servers
- Server virtualization or hypervisor layer adds to overhead
  - Impedes the Hadoop performance

Case Study
Feedbacks Processing

Case Study – Feedbacks Processing

- **Purpose –**

- A huge file contains feedbacks mails from customers
- Need to find out the number of times when
  - Goods were returned and
  - Refund was requested

- Will help business to measure the performance of a vendor
- This is a word count problem.
- E.g. find the frequency of words – refund, return etc.

Feedbacks System – Words of Interest

- Return Requested
  - Returned
  - Return Request
  - Goods Returned
  - Item Returned
  - Damaged
  - Defective
  - Exchange
- Refund Requested
  - Refund
  - Refund requested
  - Refund required
  - Request for refund
  - Cancellation
  - Cancel order
  - Money back

Feedback System - Process

- The big file – Feedback.txt contains all feedback mails

- **High Level steps**

- Client loads this file in cluster
- Submits a job describing how to analyse this data
- Cluster will process and return a new file – Returned.txt
- This file will have the desired result
- The client will read this file.

Processing – a deep dive

- Client breaks the file into three blocks
- For each block, client consults NN and receives a list of 3 DN
- NN provides – Rack number, Hostname, port number, ip address etc
- Client writes block directly to the DN
- The receiving DN replicates to another DN and so on..
- Two DN are in the same Rack and one DN in another Rack
    - To avoid data loss in case one rack fails

Processing – Storing and Replication of A Block

- Client initiates TCP connections to DN1
- To DN1 it sends the location of DN2 and DN3
- DN1 initiates TCP to DN2 and sends info about DN3
- DN2 initiates TCP to DN3 and sends info about the client
- On successful replication, “Block Received” report is sent to NN
- “Success” message is also sent to the Client to close TCP connections
- Client informs NN that the block was successfully written
- NN updates its metadata info with the node locations of block A of Feedback.txt
- Client processes another block.

Processing – Metadata and Health Check

- NN not only holds Metadata but also supervise the health of DN
- NN acts as central controller of HDFS
- DN sends heartbeats every 3 seconds
  - TCP connection to port used by NN daemon.
- Every 10^{th} heartbeat, DN sends block report
- Block report contains info about all of the blocks a DN has
- NN ACKs the heartbeat/Block report
- DN acknowledges the ACK

Processing – Role of Secondary NN

- Every hour, by default, secondary NN connects to the NN
- Copies from NN
  - The in-memory metadata information - X
  - Files that used to store metadata – Y
- X and Y may or may not be in sync
- Secondary NN combines X and Y in a fresh set of files
- Delivers the new set of files to NN
- Keeps a copy for itself.

Processing – Receiving the Output

- When Client wants to retrieve the output of a job
- Client communicates to NN and asks for the block locations of result file
- NN provides a unique list of 3 DN for each block
- Client chooses first DN in the list
- Blocks are read sequentially
- Subsequent blocks are read only after the previous block is read completely.
- NN checks the DN in the same rack to avoid traversing to other switches
- If the data can be retrieved from the same rack, processing can begin soon and job completes faster

Processing – MapReduce

- MapReduce - Parallel processing framework
- MapReduce => Map and Reduce
- Map process runs computation on the local block

- In our case, we're counting the occurrence of word – Refund
- Let's do one by one – First **Map** and then **Reduce**

Processing – The Map Process

- Client submits the MapReduce job to the JobTracker
- JobTracker finds from NN which DN has blocks of Feedback.txt
- JobTracker provides required Java code to execute Map computation to TaskTracker on DN
- TaskTracker starts a Map Task and monitors the progress
- TaskTracker provides Heartbeats and task status back to JobTracker
- As Map task completes on DN, it stores the results of local computation as **intermediate data**
- This intermediate data is sent over network to a node running Reduce task for final computation

Processing – The Reduce Process

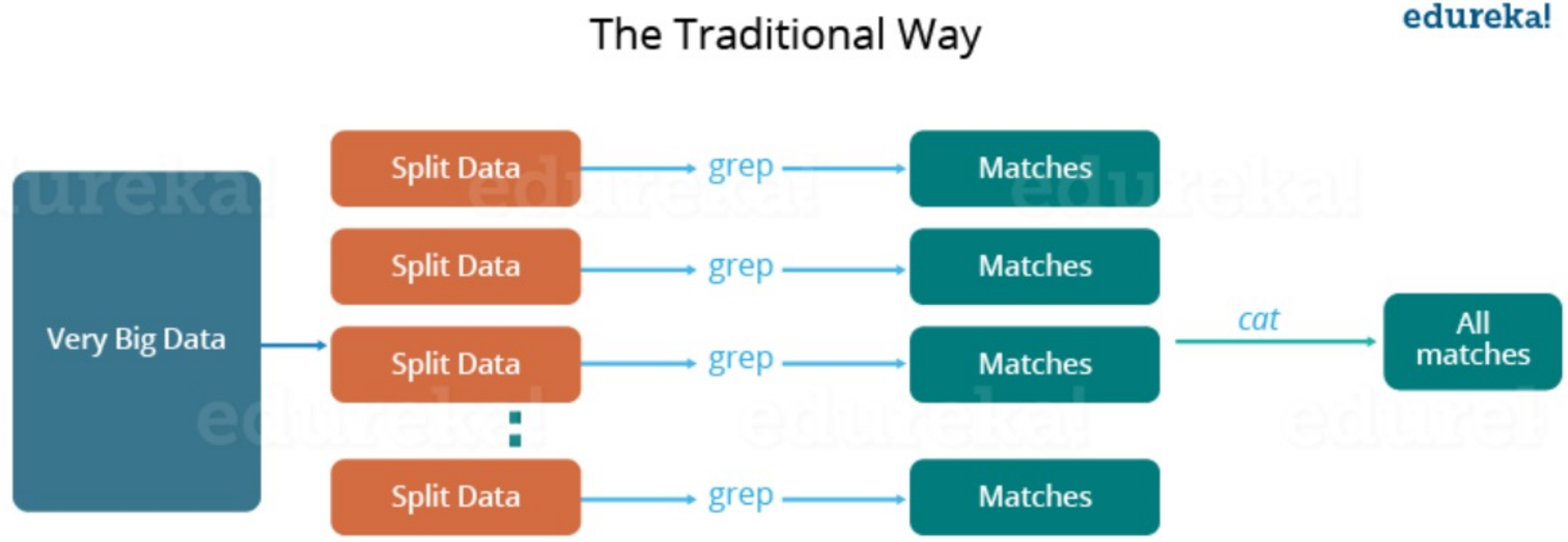
- JobTracker starts a Reduce Task on any one of the nodes
- Instructs Reduce task to go and grab the data from Map Tasks
- Map task may send data simultaneously
- In-Cast of Fan-in can occur
  - A traffic situation where number of nodes sending TCP data to a single node, all at once.
- The network switches have to manage the internal traffic to handle all in-cast conditions
- After collecting the data, final computation phase starts
- Results are written to file – Results.txt to HDFS
- Client can read the Results.txt from HDFS
- Job is marked as completed.

Points to Ponder

- Complexity of distributed and concurrent apps makes it harder or more expensive to scale up(vertically)
- Additional resources need to be added to existing node to keep up with data growth
- Hadoop follows scale-out approach
  - More nodes can be added or removed
  - Scale up and down both are easy
- RDBMS also scale up but very expensive
- Code is moved to data, local computation
- Hadoop Defines smaller number of components and well-defined interfaces of interaction between the components
- Allows developers to focus on app and business logic
- No worry about the system level challenges

MapReduce Programming Model

Before MapReduce



Problems with Traditional way

Critical path problem

- Amount of time taken to finish the job without delaying next milestone
- Or the actual completion date
- If machine delays the job, the project gets delayed

Reliability issue

- What if any of the machine involved fails.
- Failover management is a challenge

Equal Split issue

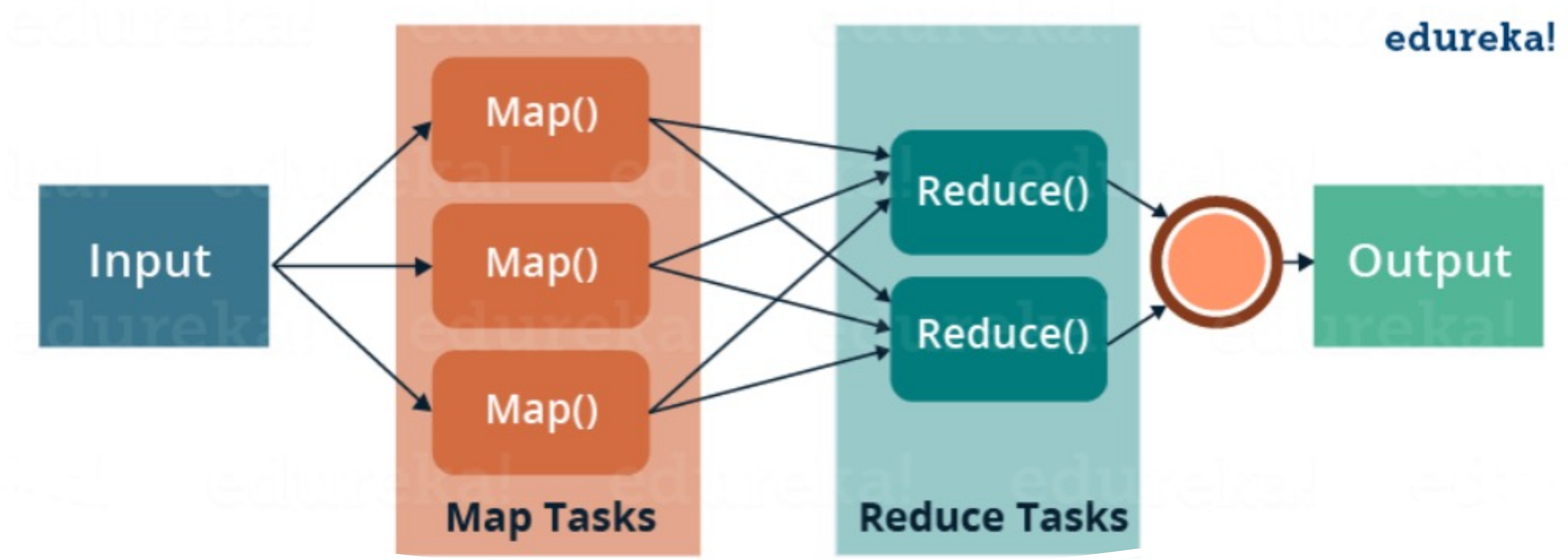
- Need to split the data so that no individual machine is overloaded or under utilized.

Single split may fail

- Even a single machine could not process and return output, the final outcome may not be valid
- Fault Tolerance needed

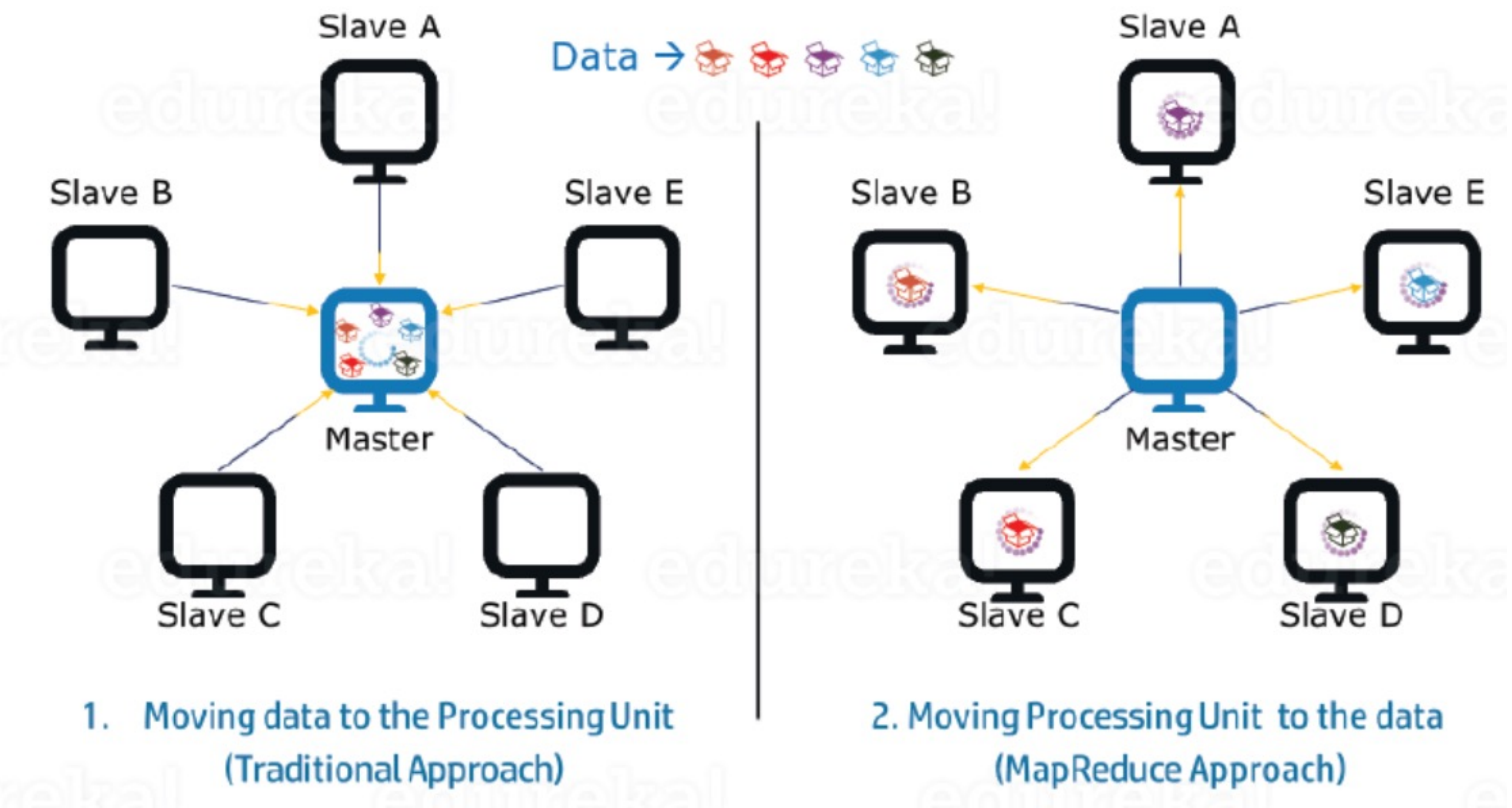
Aggregation of result

- Need of a mechanism to aggregate the result from each machine to make a final output



MapReduce Approach

- MapReduce is a programming framework that allows us to perform distributed and parallel processing on large datasets in a distributed environment
- Write code logic without bothering about reliability, fault-tolerance, design issues etc.



MapReduce Advantage

MapReduce

- Works on Divide-and-conquer principle
- Huge amount of input data is split into 64 MB chunks/blocks
- Mappers process this data in parallel
- Mappers exist and run at the same location as data chunks
- Intermediate result is shuffled and sorted
- Output of shuffling and sorting is sent to Reducers
- Mappers and Reducers are written by Programmers (Us).
  - Need to extend the Base Classes provided by Hadoop
  - May need specific implementation – depends on the problem in context

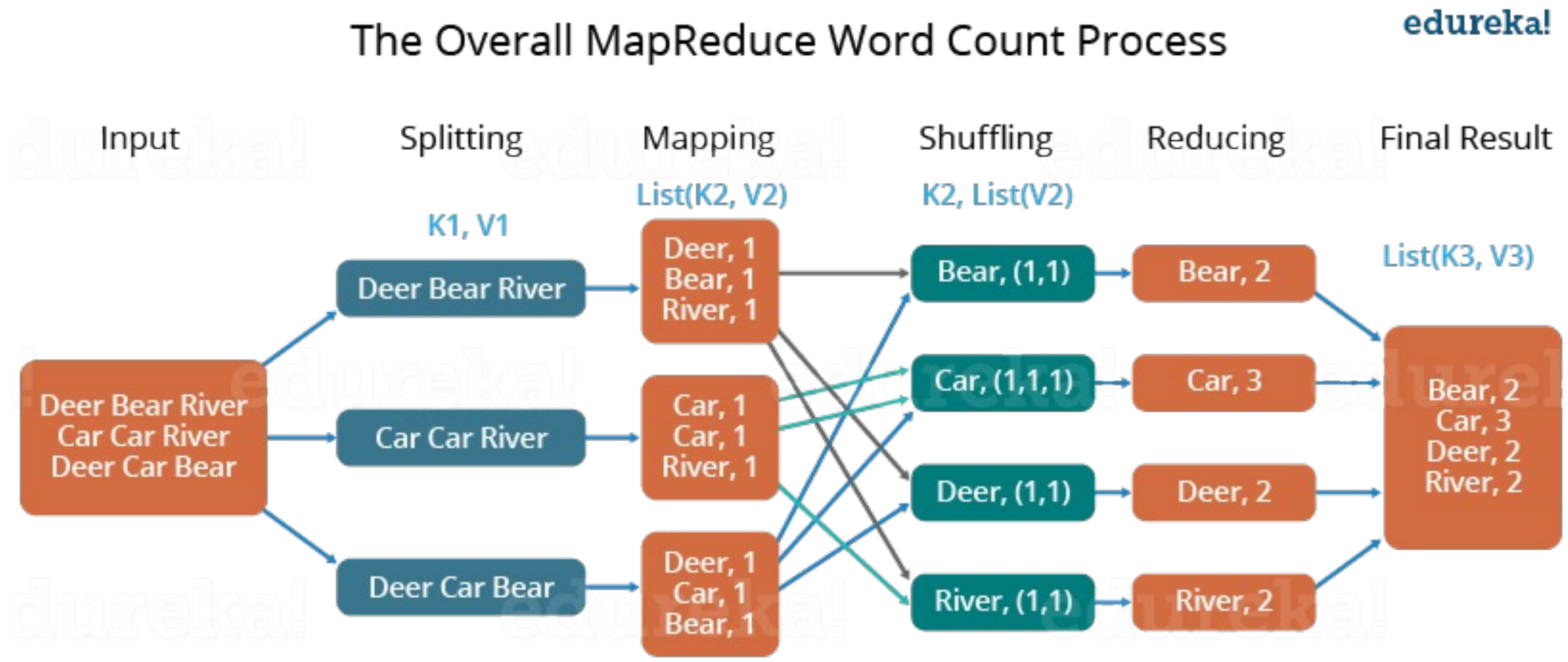
MapReduce

$$(k_1, v_1) \xrightarrow{\text{Map}} \text{list}[(k_2, v_2)] \xrightarrow{\text{Sort}} \text{list}[(k_2, \text{list}[v_2])] \xrightarrow{\text{Reduce}} (k_3, v_3)$$

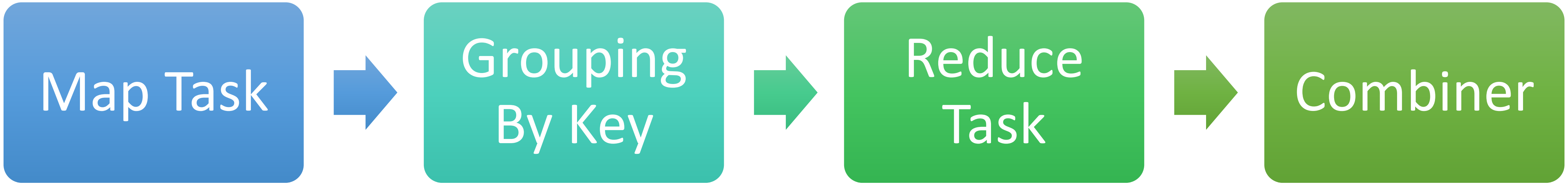
Three steps

- Map: (k1,v1) ↦ list[(k2,v2)]
- Sort: list[(k2,v2)] ↦ list[(k2,list[v2])]
- Reduce: (k2,list[v2]) ↦ (k3,v3)

MapReduce


The Overall MapReduce Word Count Process
edureka!
Input
Splitting
Mapping
Shuffling
Reducing
Final Result
K1, V1
List(K2, V2)
K2, List(V2)
List(K3, V3)
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, (1,1)
Car, (1,1,1)
Deer, (1,1)
River, (1,1)
Bear, 2
Car, 3
Deer, 2
River, 2
Bear, 2
Car, 3
Deer, 2
River, 2

MapReduce Components



Map Task

- A chunk is a collection of element
- An element can be a tuple, line, document
- All input to Map tasks are key-value pairs
- Map converts to zero or more key-value pairs
- Keys may not be unique
  - It is possible to have same key-value pairs from same element
- Example – word count – count the words in a line, document etc

Map Task

- If we need to count the words in documents.
- The element here in chunks would be document
- Map function reads the document and breaks into sequence of words
- Each word is counted as one → k1, 1
- A word can be repeated hence the keys may not be unique.
- Final output of the Map task would be –
  - (Word1, 1), (word2, 1),…. (wordn, 1)
- This way all of the documents in chunk are processed by Map task.

Grouping By Key

- Grouping is performed by system automatically regardless of what Map or Reduce do.
- Each key from the result of the Map task is hashed
- The key-value pairs are saved in one of the “i” local files.
- “i” => number of reducers to be executed
- To perform the grouping by key, Master controller merges the files from Map tasks.
- Example -
  - Map Tasks output --> (Word1, v1), (word1, v2), … (word1, vn)
  - Input to Reducer --> (word1, [v1, v2, v3 … vn])

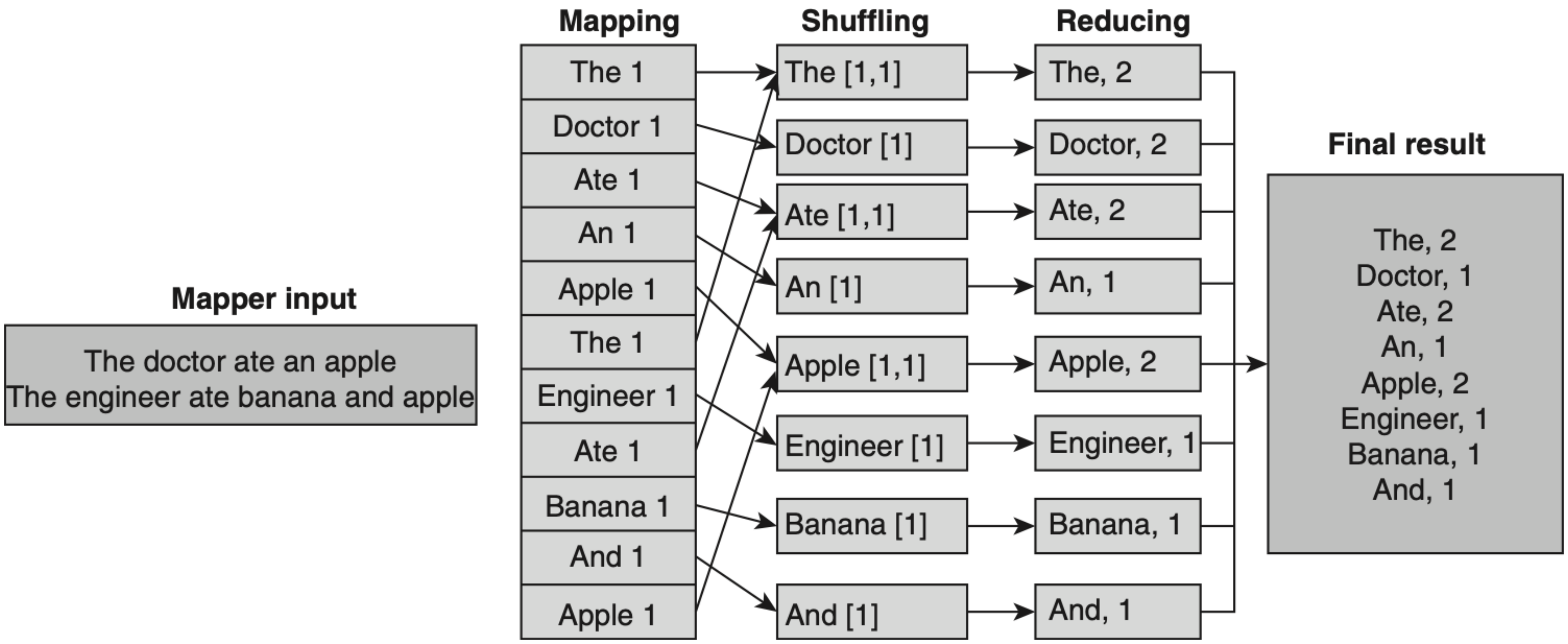
Reduce Task

- Input to Reducer
  - Key and the associated files [0 – (i-1)]
- Each reduce function uses one or more Reducers
- Output of all reducers is merged into a single file
- Output is a sequence of zero or more key-value pairs – one for each key
- For word count, reducer will add all of the values in the list
  - Input => word1, [v1, v2, v3.. vn]
  - Output => word1, v
  - Where v => sum(v1, v2, v3… vn)

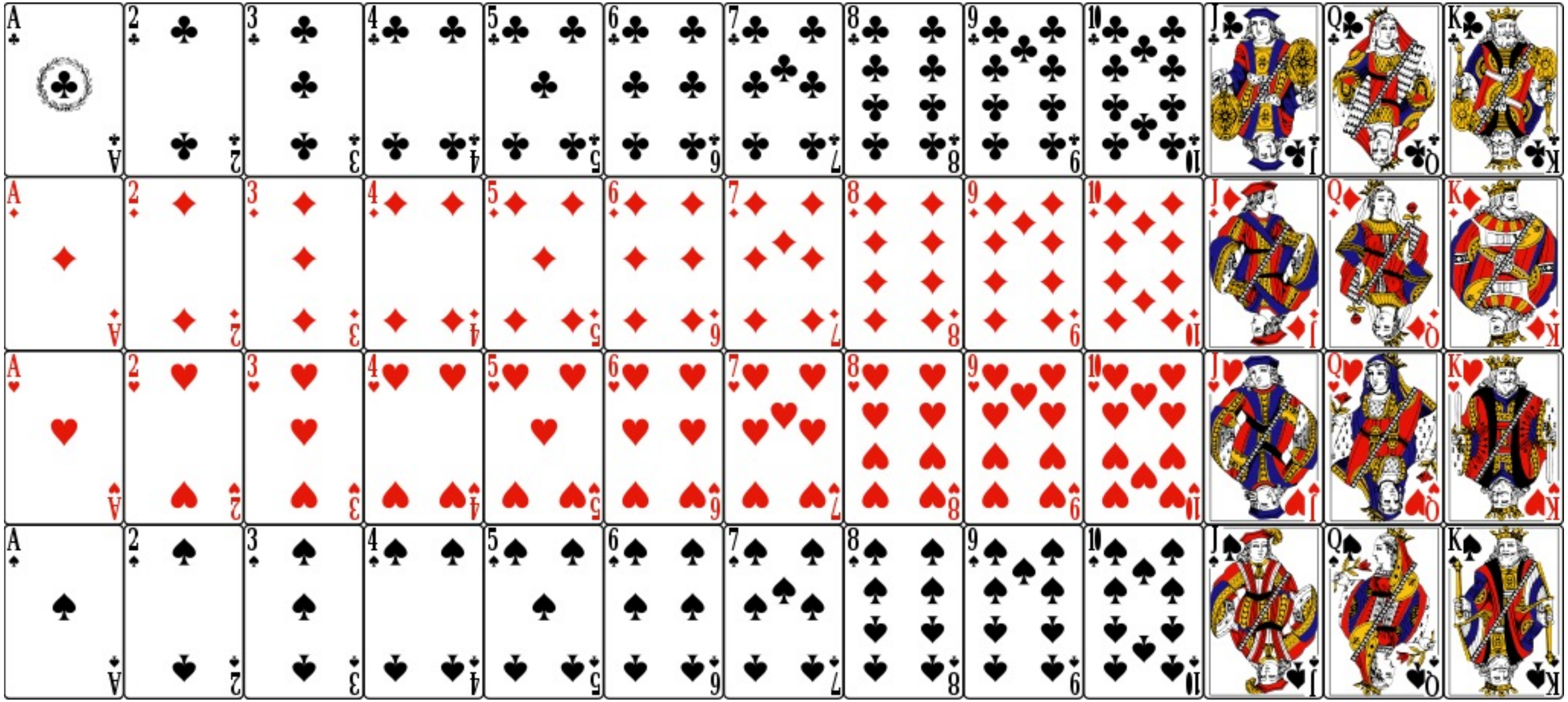
Combiners

- Reduce function can be both associative and commutative
  - E.g. sum, max, average etc
- Instead of sending all of the mapper output to reducer,
  - Some values are computed at the Map side using Combiners
  - Then sent to Reducer
  - To optimize the input/output operations between Mappers and Reducers
- Example –
  - After map task, the word xyz may appear in k times. Then (xyz, 1) will appear k times.
  - This can be grouped as (xyz, k) and then can be send to Reducer

Word Count Exercise



Playing Card – MapReduce Demo



Map Process

- Map Process runs these computation on blocks
- Client submits MapReduce to JobTracker
- JT provides tasks to TaskTracker with required Java code to execute Map
- TT executes Map Task, monitors, provides heartbeat and status to JT
- After Map task, result of the computation is stored as intermediate data
- The intermediate data is sent to a node running Reducer

Reduce Process

- JobTracker starts a Reduce Task on any one of the nodes
- Instructs Reduce task to go and grab the data from Map Tasks
- Map task may send data simultaneously
- In-Cast of Fan-in can occur
  - A traffic situation where number of nodes sending TCP data to a single node, all at once.
- The network switches have to manage the internal traffic to handle all in-cast conditions
- After collecting the data, final computation phase starts
- Results are written to file – Results.txt to HDFS
- Client can read the Results.txt from HDFS
- Job is marked as completed.

MapReduce Execution

Runtime Coordination

- MapReduce handles distributed code execution transparently
- MR takes care of scheduling and synchronization
- MR ensures that job submitted by all of the users get fair share of cluster's execution
- Scheduling optimizations
  - If a machine executes a task very slowly, JT assigns different instance(node) using a different TT
  - It is true for both Map and Reduce part
- Synchronization optimizations
  - Reduce Phase can not start until all Map tasks are completed
  - Shuffling and sorting is done using all of the nodes where Map tasks are executed and where Reduce will be executed

Responsibilities

- Takes care of scheduling, monitoring, and rescheduling of failed tasks
  - Provides overall coordination of execution.
  - Selects nodes for running mappers.
  - Starts and monitors mappers execution.
  - Sorts and shuffles output of mappers.
  - Chooses locations for reducers execution.
  - Delivers the output of mapper to reducer node.
  - Starts and monitors reducers execution.

Execution Pipeline

- **Key Components**

- Driver
- Input Data
- Mapper
- Shuffle and Sort
- Reducer
- Optimized MapReduce process by using Combiner(optional)
- Distributed Cache

Execution Pipeline

Driver

- Main program that initialize job & get back status of job execution.
- Defines configuration & specification of all components – For each job
- Include I/O format

Input Data

- I/p reside in HDFS or Hbase
- *InputFormat* : defines number of map task in mapping phase
- *InputSplit:* unit of work task, jobdriver invokes InputFormat to decide numberof (split) & location of map task execution.
- *RecordReader:* reads data in maper task ,converts data to key,value.

Execution Pipeline

Mapper

- For each map task, mapper is initiated.

Shuffle and Sort

- Process of moving mappers output to reducer
- It is triggered when mapper completes its task
- Grouping is performed regardless what mapper, reducer does, Pairing with same key is grouped & passed to single reducer

Execution Pipeline

Reducer

- Executes user-define code
- Reduce method receive the key
- Record writer is used for storing data in specified location

Optimization using combiners [Optional]

- Combiner takes the input and combines the value with same key to reduce the number of keys

Distributed Cache

- Resource used globally by all nodes in cluster.
- Can be shared library that each task can access.
- User code (driver, mapper, reducer) jar file can be placed in cache .

Process Pipeline

- JobDriver uses InputFormat to partition a map's execution & initiates a JobClient.
- Job Client communicates with JobTracker and submits the job for execution.
- JobTracker creates one Map task for each split as well as a set of Reducer tasks
- TaskTracker that is present on every node of cluster, controls the actual execution.
- Once TT starts the Map job, periodically send heartbeat to JT, indicates its ready to accept job.
- JT uses scheduler to allocate task to TT – uses the info from heartbeat
- TT copies the job files, files needed for execution, and creates a child process.
- Child process informs the TT every few seconds with the status
- TT informs the JT as soon the last job is completed and JT changes job status to complete
- By periodically polling JT, JobClient recognizes the job status.

Coping with Node Failure

- MapReduce job are submitted and tracked by JobTracker
- Only one JobTracker for a Hadoop Cluster and runs on its own JVM
- All slave node are configured with JT node location
- If JT fails, all job are halted. & restarted again.
- JT monitors all TT so if a TT fails JT detects failure.
- All tasks of a failing TT node are restarted
- Even if some task is completed it must be redone, since the output of reducer(at the failing node) would be unavailable
- JT informs all Reducer tasks that the input to Reducer will be available from the new location
- If failure at Reducer, then JT sets it to idle & reschedules the reducer task in another node.

Algorithms using MapReduce

Algorithm using MapReduce

- **Use Cases for MapReduce**
  - Ideal for data-intensive computations
  - Well-suited for analytics, like identifying similar buying patterns
- **When Not to Use**
  - Not suitable for real-time, transactional tasks like online retail
  - Processes involving little calculations
- Examples where MapReduce can be used
  - To solve analytic queries on large amount of data
  - Relation Algebra operations
- Google implemented MapReduce
  - Large matrix-vector multiplications for its PageRank

Algorithm using MapReduce

- **MapReduce Program**
- Three components
  - Driver
    - Initializes the job configuration
    - Defines the mapper and reducer for the job
    - Specifies the path for input and output files
  - Mapper and Reducer
    - Extends classes from **org.apache.hadoop.mapreduce**
    - https://hadoop.apache.org/docs/r2.7.3/api/org/apache/hadoop/mapreduce/package-summary.html

Mapper and Reducer

- Mapper and Reducer classes are generic types
- Users can specify the types for input/output keys and values.
- Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
- Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
- Types
  - KEYIN – The type of the input key
  - VALUEIN – The type of the input value
  - KEYOUT – The type of the output key
  - KEYOUT – The type of the output value

Mapper and Reducer

```
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
        // Mapper implementation here
}

public class WordCountReducer extends Reducer< Text, IntWritable, Text, IntWritable > {
        // Reducer implementation here
}
```

- Types
  - KEYIN – The type of the input key - LongWritable → long value
  - VALUEIN – The type of the input value - Text
  - KEYOUT – The type of the output key - Text
  - KEYOUT – The type of the output value - IntWritable → integer value

Mapper and Reducer

- Mapper and Reducer implementations are classes.
- Actual functions doing the job are usually map() and reduce().

```
public class TokenCounterMapper extends Mapper<Object, Text, Text, IntWritable>{
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        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);
                }
        }
}
```

Mapper and Reducer

- The **Key** is a generic type parameter
  - Provides flexibility by allowing the reducer to operate on any key type

```
public class IntSumReducer<Key> extends Reducer<Key, IntWritable, Key, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(Key key, Iterable<IntWritable> values, Context context)
                throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
}
```

Mapper and Reducer

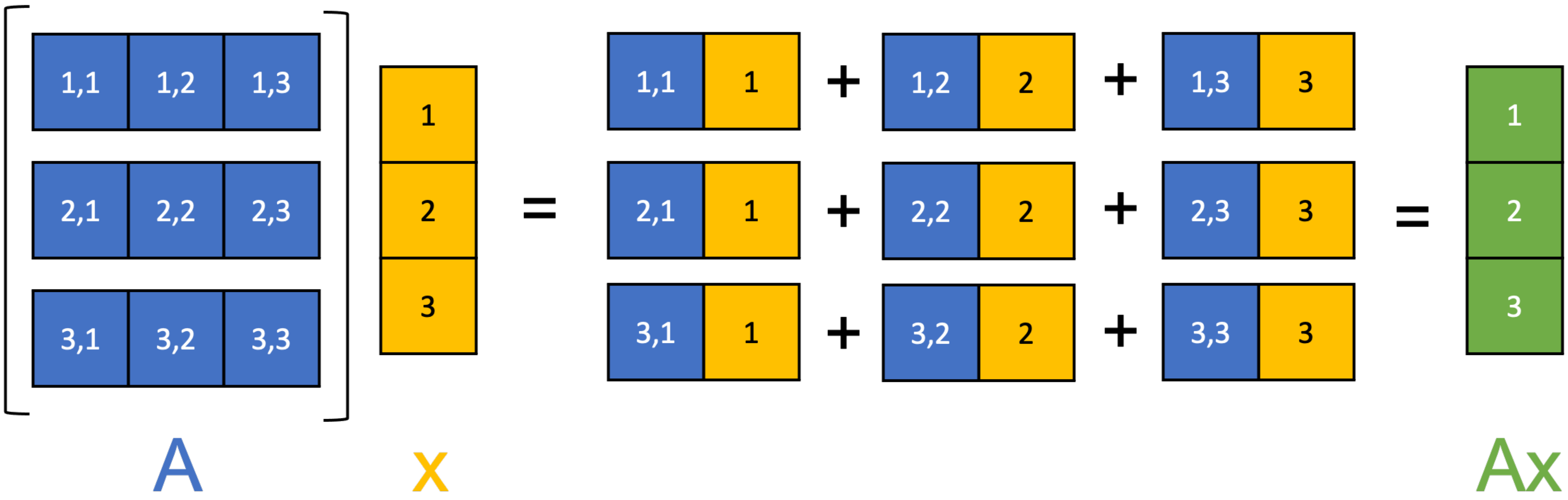
- Map function is –
  - **map (*InputKeyType* inputKey, *InputValueType* inputValue):**
  - Process the *inputKey* and *inputValue*
  - Result will be *intermediateKey* , *intermediateValue* pair
  - **Emit(intermediateKey, intermediateValue);**
  - map can emit more than one intermediate key–value pairs

Mapper and Reducer

- Reduce function is –
  - **reduce (*IntermediateKeyType* intermediateKey, *Iterator* values):**
  - All the values for a particular intermediateKey is first iterated
  - A user-defined operation is performed over the values.
  - The number of reducers is specified by the user
  - Reducers run in parallel
  - *outputValue* will contain the value that is output for that *outputKey*
  - **Emit(outputKey, outputValue);**
  - reduce() method can emit more than one output key–value pairs

Matrix Vector Multiplication

Matrix-Vector Multiplication



Matrix-Vector Multiplication

- Let's say we have a matrix A and a vector x

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}, x = \begin{pmatrix} 7 \\ 8 \end{pmatrix}$$

- A multiply by x

1. First row: (1×7)+(2×8) =7+16 =23
2. Second row: (3×7)+(4×8) =21+32 =53
3. Third row: (5×7)+(6×8) =35+48 =83

- So, b = Ax =>

$$b = \begin{pmatrix} 23 \\ 53 \\ 83 \end{pmatrix}$$

Matrix-Vector Multiplication

- Using MapReduce
- Let's consider matrices A (L x M) and B(M x N)
- L = 2, M = 3, N = 2

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}$$

Matrix-Vector Multiplication

- Map Phase

For each element (i,j) of I, emit ((i,k), A[i,j]) for k in 1,...,N.

For each element (j,k) of B, emit ((i,k), B[j,k]) for i in 1,...,L.

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}$$

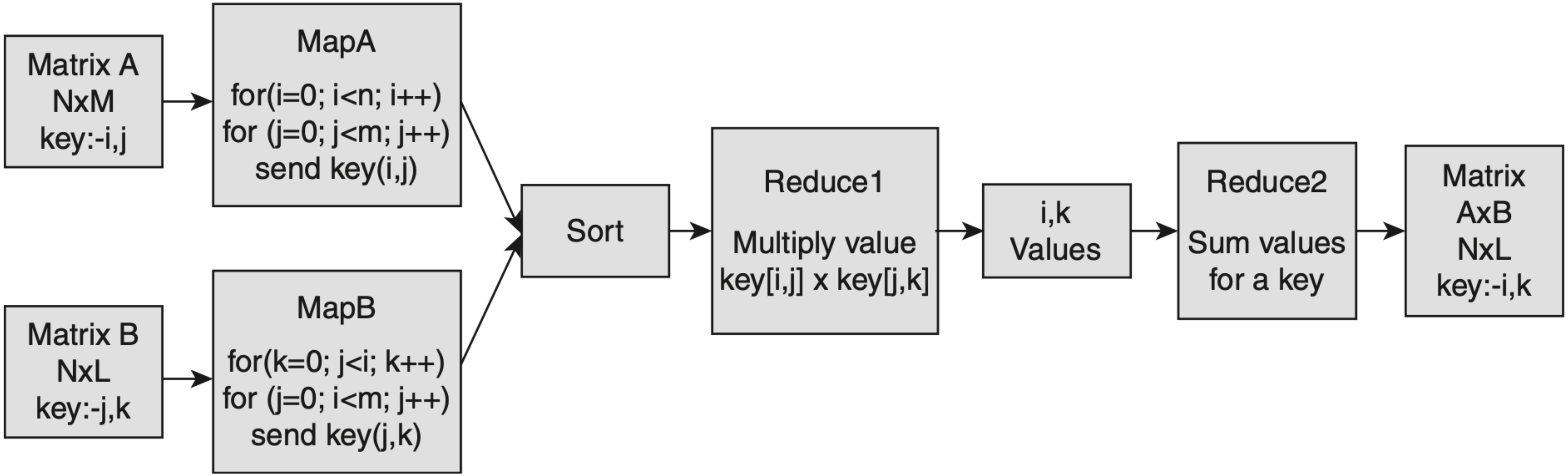
1. **Map for Matrix A:**
   - For element $A[1,1] = 1$, emit $((1,1),1), ((1,2),1)$
   - For element $A[1,2] = 2$, emit $((1,1),2), ((1,2),2)$
   - For element $A[1,3] = 3$, emit $((1,1),3), ((1,2),3)$
   - For element $A[2,1] = 4$, emit $((2,1),4), ((2,2),4)$
   - For element $A[2,2] = 5$, emit $((2,1),5), ((2,2),5)$
   - For element $A[2,3] = 6$, emit $((2,1),6), ((2,2),6)$
2. **Map for Matrix B:**
   - For element $B[1,1] = 7$, emit $((1,1),7), ((2,1),7)$
   - For element $B[1,2] = 8$, emit $((1,2),8), ((2,2),8)$
   - For element $B[2,1] = 9$, emit $((1,1),9), ((2,1),9)$
   - For element $B[2,2] = 10$, emit $((1,2),10), ((2,2),10)$
   - For element $B[3,1] = 11$, emit $((1,1),11), ((2,1),11)$
   - For element $B[3,2] = 12$, emit $((1,2),12), ((2,2),12)$

Matrix-Vector Multiplication

- **Reduce Phase**
- key = (i,k)
- value = Sum$j$ (A[i,j] * B[j,k])
- One reducer is used per output cell
- Each reducer calculates Sum$j$ (A[i,j] * B[j,k])

- First let's see how to calculate → C[1,1]
- Emitted key value pairs for C[1,1] are =
  - ((1,1),1), ((1,1),2), ((1,1),3), ((1,1),7), ((1,1),9), ((1,1),11)
- C[1,1]= A[1,1]×B[1,1] + A[1,2]×B[2,1] + A[1,3]×B[3,1]
- C[1,1] = 1 × 7 + 2 × 9 + 3 × 11
- **Similarly, we calculate other values for C**

Matrix-Vector Multiplication



MapReduce and Relational Operators

- Shuffle/Sort Handles Grouping
  - Shuffle and Sort phases automatically take care of sorting and grouping the intermediate key-value pairs emitted by the mappers
  - This can be compared with GROUP BY operation in an SQL Query

UserID	AmountSpent
1	10
2	20
1	30
3	40
1	50
3	60

SELECT UserID, SUM(AmountSpent) FROM Transactions GROUP BY UserID;

Map Output: (1, 10), (2, 20), (1, 30), (3, 40), (1, 50), (3, 60)

Sorted/Grouped Output: (1, [10, 30, 50]), (2, [20]), (3, [40, 60])

Reduce Output: (1, 90), (2, 20), (3, 100)

MapReduce and Relational Operators

Following operation are performed either at Mapper or Reducer

Selection

- SELECT * FROM Employees **WHERE** Salary > 50000;

Projection

- SELECT FirstName, LastName FROM Employees;

Union, Intersection, and Difference

- SELECT City FROM Table1 **UNION** SELECT City FROM Table2;
- SELECT City FROM Table1 **INTERSECT** SELECT City FROM Table2;
- SELECT City FROM Table1 **EXCEPT** SELECT City FROM Table2;

Natural Join

- SELECT * FROM Orders NATURAL JOIN Customers;

Grouping and Aggregation

- SELECT Department, **COUNT(*)** FROM Employees **GROUP BY** Department;
- SELECT Department, **AVG(Salary)** FROM Employees **GROUP BY** Department;

MapReduce and Relational Operators

Following operation are performed either at Mapper or Reducer

Selection

- SELECT * FROM Employees **WHERE** Salary > 50000;
- Filter rows based on some criteria
- Can be done either at Mapper or Reducer

Projection

Union, Intersection, and Difference

Natural Join

Grouping and Aggregation

MapReduce and Relational Operators

Following operation are performed either at Mapper or Reducer

Selection

Projection

- SELECT FirstName, LastName FROM Employees;
- Selects Columns from Database, Like the SQL SELECT statement
- Usually done at the Mapper side
- Why at Mapper?

Union, Intersection, and Difference

Natural Join

Grouping and Aggregation

MapReduce and Relational Operators

Following operation are performed either at Mapper or Reducer

Selection

Projection

Union, Intersection, and Difference

- SELECT City FROM Table1 **UNION** SELECT City FROM Table2;
  - Combines Multiple Datasets/tables
  - In simple words, and with respect to SQL, adds the data from two tables
- SELECT City FROM Table1 **INTERSECT** SELECT City FROM Table2;
  - Finds the intersection of two tables
- SELECT City FROM Table1 **EXCEPT** SELECT City FROM Table2;
  - Finds element from one table which are not part of another table
- Performed at either Mapper or Reducer

Natural Join

Grouping and Aggregation

MapReduce and Relational Operators

Following operation are performed either at Mapper or Reducer

Selection

Projection

Union, Intersection, and Difference

Natural Join

- SELECT * FROM Orders NATURAL JOIN Customers;
- Combines two or more tables on a related column
- Can be done in multiple ways – map side join, reduce side join

Grouping and Aggregation

- SELECT Department, **COUNT(*)** FROM Employees **GROUP BY** Department;
- SELECT Department, **AVG(Salary)** FROM Employees **GROUP BY** Department;
- Involves grouping the data and then performing some calculations e.g. sum, average etc.
- Aggregation is done on each group
- Often done in reducer, benefitting from the automatic sorting and grouping done in shuffle and sort phase

MapReduce and Relational Operators

Join Strategies

Reduce Side Join

- Reducer performs the Join Operation
- Mapper outputs the pairs with Join Key
- Shuffle phase ensures that all related records are sent to same Reducer.

Map Side Join

- Mapper performs the join operation
- By loading smaller dataset into memory

In-Memory Join

- These are specialized map-side joins where data is pre-loaded into memory for faster access
- Two Variants
- **Striped Variant** – Stores the smaller table in an efficient in-memory data structure e.g. hash table.
  - So that the mapper can quickly perform the join
- **Memcached Variant –** Stores data in external in-memory key-value store such as Memcached
  - To hold a smaller table for very fast lookups during the map phase

Computing Selections

- Selections may not need both mapper and reducer
- Mostly can be done in Map part only
- Ex. SELECT * FROM Employees WHERE Salary > 60000;
- Map function –
  - Takes each tuple t in relation R and checks if it meets a given condition C (e.g., Salary>60000).
  - If yes, emits a key-value pair as (t, t) where both key and value are tuple t
- Reduce function –
  - An identity function – does not modify the data
  - Simply passes each key-value pair as received to the output

Pseudo Code for Selection

```
map(key, value):
    for tuple in value:
        if tuple satisfies C:
            emit(tuple, tuple)
reduce(key, values):
    emit(key, key)
```

Computing Selections

Map

- Acts as a filter.
- If R = {(1,2), (3,4), (5,6)}
- And condition is – first element is >=3
- Then output would be –
  - ((3,4), (3,4)), ((5,6),(5,6))

Reduce

- Does not modify anything – Identity function
- Input - ((3,4), (3,4)), ((5,6),(5,6))
- Output – same as input - ((3,4), (3,4)), ((5,6),(5,6))
- Because the filtering has already happened in Map.

Pseudo Code for Selection

```
map(key, value):
    for tuple in value:
        if tuple satisfies C:
            emit(tuple, tuple)
reduce(key, values):
    emit(key, key)
```

Computing Projections

- Ex. SELECT name, department FROM Employees;
- Projection may result in duplicate tuples
- Reduce fn is used to remove duplicates
- Map function –
  - Goal is to eliminate unwanted columns
  - For each tuple t in relation R
  - Construct a tuple ts with only those components which are needed
  - E.g. only keep name and department from all columns
  - Emit (ts, ts)
- Let t = (a,b,c), S = {a,c}, then ts = {a,c}
- R = {(1,2,3),(4,5,6)}, S = {1,3}
- Map output => ts = ((1,3),(1,3)), ((4,6),(4,6))

Pseudo Code for Projection

```
map(key, value):
    for tuple in value:
        ts = tuple with only the components for the attributes in S
        emit(ts, ts)

reduce(key, values):
    emit(key, key)
```

Computing Projections

Duplicates after Map Output

Given Relation R

a	b	c
1	2	3
4	5	6
1	7	3

Projection

- S={ a, c }

Map output

- For the first row (1,2,3), the output would be ((1,3),(1,3))
- For the second row (4,5,6), the output would be ((4,6),(4,6))
- For the third row (1,7,3), the output would again be ((1,3),(1,3))

Final output ->

- ((1,3),(1,3)), ((4,6),(4,6)), ((1,3),(1,3)) ---
- ((1,3),[(1,3),(1,3)]), ((4,6),[(4,6)])

Computing Projections

- Reduce function –
  - For each key ts, there will be one or more key-value pairs (ts, ts).
  - Reduce turns [ts,ts,ts,…,ts] into (ts,ts)
  - Produces exactly one pair (ts,ts) for key ts
- Let t = (a,b,c), S = {a,c}, then ts = {a,c}
- R = {(1,2,3),(1,4,3),(4,5,6)}, S = {1,3}
- Map output => ts = ((1,3),(1,3)), ((4,6),(4,6)), ((1,3),(1,3))
- Reduce output => ((1,3),(1,3)), ((4,6),(4,6))

Pseudo Code for Projection

```
map(key, value):
    for tuple in value:
        ts = tuple with only the components for the attributes in S
        emit(ts, ts)

reduce(key, values):
    emit(key, key)
```

Union Operation

- Union is done between two relations – R and S
- For Union, both R and S should have same schema
- Map function –
  - Map fn is given chunk(tuples here) of Relation R or S
  - Simply turns each tuple t into key-value pair (t,t)
  - Emit (t, t)
  - No processing on data is done by Map function
- Reduce function –
  - Ensures that duplicates are removed
  - Emit (t,t)

Pseudo Code for Union

```
map(key, value):
    for tuple in value:
        emit(tuple, tuple)

reduce(key, values):
    emit(key, key)
```

Union Operation

- R = {(1,2),(3,4)}
- S = {(3,4),(5,6)}
- Map output
  - `(1,2) => (1,2),(1,2)`
  - `(3,4) => (3,4),(3,4)`
  - `(3,4) => (3,4),(3,4)`
  - `(5,6) => (5,6),(5,6)`
- Input to Reduce
  - `(1,2),[(1,2)]`
  - `(3,4),[(3,4),(3,4)]`
  - `(5,6),[(5,6)]`
- Reduce output
  - `(1,2),[(1,2)]          => Output (1,2),(1,2)`
  - `(3,4),[(3,4),(3,4)]    => Output (3,4),(3,4)`
  - `(5,6),[(5,6)]          => Output (5,6),(5,6)`
  - `Final output => {(1,2),(3,4),(5,6)}`

Pseudo Code for Union

```
map(key, value):
    for tuple in value:
        emit(tuple, tuple)

reduce(key, values):
    emit(key, key)
```

Intersection Operation

- Intersection is done between two relations – R and S
- Both R and S should have same schema
- Map function –
  - Map fn is given tuples of Relations R or S
  - Simply turns each tuple t into key-value pair (t,t)
  - Emit (t, t)
- Reduce function –
  - If a key t has value list as [t,t], which means that it exists in both R and S
  - If yes, then outputs single key value pair (t,t)
  - If a key t has single value as [t], then no output.
  - Emit (t,t) if tuple t exists in both R and S
  - otherwise no output

Pseudo Code for Intersection

```
map(key, value):
    for tuple in value:
        emit(tuple, tuple)

reduce(key, values):
    if values == [key, key]
        emit(key, key)
```

Intersection Operation

- Let R={(1,2),(3,4)} and S={(3,4),(5,6)}
- Map output
  - (1,2) -> (1,2),(1,2)
  - (3,4) -> (3,4),(3,4)
  - (3,4) -> (3,4),(3,4)
  - (5,6) -> (5,6),(5,6)
- Input to Reduce
  - (1,2),[(1,2)]
  - (3,4),[(3,4),(3,4)]
  - (5,6),[(5,6)]
- Reduce
  - (1,2),[(1,2)] -> Output nothing
  - (3,4),[(3,4),(3,4)] -> Output (3,4),(3,4)
  - (5,6),[(5,6)] -> Output nothing
- Final output
  - {(3,4)}

Pseudo Code for Intersection

```
map(key, value):
    for tuple in value:
        emit(tuple, tuple)

reduce(key, values):
    if values == [key, key]
        emit(key, key)
```

Difference Operation

- Difference is done between two relations – R and S
- Both R and S should have same schema
- We're trying to do (R – S)
- Map function –
  - Map fn is given chunk(tuples) of Relations
  - Simply turns each tuple t into key-value pair (t,Rname) or (t, Sname)
  - Rname and Sname here are names of the relations R and S respectively
  - Emit (t, Rname)
  - Can also be written as Emit(t,R)
    - Meaning that a tuple t was found in R
- Reduce function –
  - Receives inputs as (t,[R]), (t,[R,S]), or (t,[S])
  - We are interested in only those tuples which were part of R and not S

Pseudo Code for Difference

```
map(key, value):
  if key == R:
    for tuple in value:
      emit(tuple, R)
  else:
    for tuple in value:
      emit(tuple, S)
reduce(key, values):
  if values == [R]
    emit(key, key)
```

Difference Operation

- R={1,2,3,4}
- S={3,4,5}

Map

- (1,R),(2,R),(3,R),(4,R),(3,S),(4,S),(5,S)

Input to Reduce

- 1:[R], 2:[R], 3:[R,S], 4:[R,S], 5:[S]

Reduce

- For 1:[R] and 2:[R], it emits (1,1) and (2,2)
- For 3:[R,S] and 4:[R,S], it emits nothing
- For 5:[S], it also emits nothing

Final output –

- R – S => {(1,1), (2,2)}

Pseudo Code for Difference

```
map(key, value):
  if key == R:
    for tuple in value:
      emit(tuple, R)
  else:
    for tuple in value:
      emit(tuple, S)
reduce(key, values):
  if values == [R]
    emit(key, key)
```

Natural Join

- Relations R(A,B) and S(B,C)
- For natural join it is required to find tuples that agree on their B component.
  - Second component from R and first component from S
- Map
  - For each tuple (a,b) in R, produce the key-value pair (b,(R,a))
  - For each tuple (b,c) in S, produce the key-value pair (b,(S,c))
- Reduce
  - For each key b, if there are pairs or R and S as (R,a) and (S, c) then Emit (a,b,c)

Pseudo Code for Natural Join

```
map(key, value):
  if key == R:
    for (a, b) in value:
      emit(b, (R, a))
  else:
    for (b, c) in value:
      emit(b, (S, c))
```

Natural Join

- Let R(A,B)={(1,2),(2,3),(3,4)}
- Let S(B,C)={(2,7),(4,9),(3,8)}
- Map
  - For each tuple (a,b) in R, produce the key-value pair (b,(R,a))
  - For each tuple (b,c) in S, produce the key-value pair (b,(S,c))
  - Output => 2:(R,1), 3:(R,2), 4:(R,3), 2:(S,7), 4:(S,9), 3:(S,8)
- Grouping For Reduce function – Grouping by the key b
  - 2:[(R,1),(S,7)],
  - 3:[(R,2),(S,8)],
  - 4:[(R,3),(S,9)]
- Reduce
  - For each key b, if there are pairs or R and S as (R,a) and (S, c) then Emit (a,b,c)
  - 2:[(R,1),(S,7)] produces (1,2,7)
  - 3:[(R,2),(S,8)] produces (2,3,8)
  - 4:[(R,3),(S,9)] produces (3,4,9)
  - Final output => (1,2,7),(2,3,8),(3,4,9)

Grouping and Aggregation

- Can be performed in one MapReduce Job
- Mapper extracts each tuple values to “group by and aggregate” and emits them
- Reducer receives values to be aggregated that are already grouped
- Reducer calculates an aggregation function

Pseudo Code for Grouping and Aggregation

```
map(key, value):
    for (a, b, c) in value:
        emit(a, b)

reduce(key, values):
    emit(key, theta(values))
```

Example

- R(A,B,C) => {(1,2,3),(1,4,7),(2,3,1),(2,5,8),(3,1,2),(3,6,3)}
- Map: For each tuple(a,b,c), emit (a,b) => 1:2, 1:4, 2:3, 2:5, 3:1, 3:6
- Grouping for reduce: Key value pairs are grouped by their key a => 1: [2,4], 2: [3,5], 3: [1,6]
- Reduce: if aggregate fn is Sum then => 1: 6, 2: 8, 3: 7 ||| 1:2+4, 2: 3+5, 3: 1+6
    - Final output would be => {(1,6),(2,8),(3,7)}
- Reduce: if aggregate fn is Max then => 1: 4, 2: 5, 3: 6
    - Final output would be => {(1,4),(2,5),(3,6)}

Grouping and Aggregation

- In case there are multiple aggregations
- Reduce function applies each aggregation to the list of values
- Produces a tuple consisting of the key, including components for all grouping attributes
- Followed by the results of each of the aggregation
- Example : R(A,B,C,D), Group by A and B, then apply theta1(C) and theta2(D)

Mapper output might look like:

$$
\begin{aligned}
&(a_1, b_1) : (c_1, d_1), \\
&(a_1, b_1) : (c_2, d_2), \\
&(a_2, b_2) : (c_3, d_3), \\
&(a_2, b_2) : (c_4, d_4), \\
&(a_3, b_3) : (c_5, d_5)
\end{aligned}
$$

Shuffle and Sort Phase

$$
\begin{aligned}
&(a_1, b_1) : [(c_1, d_1), (c_2, d_2)], \\
&(a_2, b_2) : [(c_3, d_3), (c_4, d_4)], \\
&(a_3, b_3) : [(c_5, d_5)]
\end{aligned}
$$

Reducer Phase

$$
\begin{aligned}
&(a_1, b_1) : (\mathrm{SUM}(c_1 + c_2), \mathrm{MAX}(d_1, d_2)), \\
&(a_2, b_2) : (\mathrm{SUM}(c_3 + c_4), \mathrm{MAX}(d_3, d_4)), \\
&(a_3, b_3) : (\mathrm{SUM}(c_5), \mathrm{MAX}(d_5))
\end{aligned}
$$

Grouping and Aggregation

- If necessary, multiple rounds of MapReduce could be applied
- Especially for more complex aggregation and transformation requirements

- If multiple aggregations are to be performed on same values, then output pair might look like -

$$(a_1, b_1) : (\text{SUM}(c_1 + c_2), \text{MAX}(c_1, c_2))$$

MapReduce Job Structure

- **Consider two MapReduce Jobs for Block Multiplication and Summing up the results**

Job 1: Block Multiplications

- **Mappers**:
  - Distribute blocks of data to the reducers.
  - Use a carefully chosen intermediate key structure.
  - Key comparator and partitioning functions help in distributing data efficiently.
- **Reducers**:
  - Perform the block multiplications.

Job 2: Summing Up Results

- **Mappers**:
  - Distribute the multiplication results to the reducers.
  - Similar key structure and partitioning used as in Job 1.
- **Reducers**:
  - Sum up the multiplication results.

Performance Characteristics and Trade-offs

- Block multiplication tasks can be assigned to reducers in different ways.
- Each method has its own performance characteristics and trade-offs.
- Choose the method that best fits the performance requirements.

Block Multiplication

Block Multiplication

$$A = \begin{bmatrix} 1 & 3 & 4 & 2 \\ 0 & 5 & -1 & 6 \\ 1 & 0 & 3 & -1 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 0 & 3 \\ 1 & 2 & 0 \\ 2 & -1 & 2 \\ 0 & 3 & 1 \end{bmatrix}$$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$= \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

Multiply as the small matrices are scalar

Don't switch the order

Block Multiplication

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix} \quad A = \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \end{array}\right]$$

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} \quad B = \left[\begin{array}{cc|c} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ \hline b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{array}\right]$$

Blocks of A:

- $A1 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$
- $A2 = \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix}$
- $A3 = \begin{bmatrix} a_{31} & a_{32} \\ 0 & 0 \end{bmatrix}$

Blocks of B:

- $B1 = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$
- $B2 = \begin{bmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix}$
- $B3 = \begin{bmatrix} b_{13} & 0 \\ b_{23} & 0 \end{bmatrix}$

$$C1 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \qquad C1 = \begin{bmatrix} c1_{11} & c1_{12} \\ c1_{21} & c1_{22} \end{bmatrix}$$

$$C2 = \begin{bmatrix} a_{31} & a_{32} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \qquad C2 = \begin{bmatrix} c2_{11} & c2_{12} \end{bmatrix}$$

Block Multiplication

$$A = \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \end{array}\right]$$

$$B = \left[\begin{array}{cc|c} 2 & 0 & 1 \\ 1 & 2 & 1 \\ \hline 1 & 1 & 0 \\ 1 & 0 & 2 \end{array}\right]$$

$$C1 = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 4 & 4 \\ 12 & 12 \end{bmatrix}$$

$$C2 = \begin{bmatrix} 9 & 10 \end{bmatrix} \times \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 18 & 20 \end{bmatrix}$$

$$C3 = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 11 & 5 \end{bmatrix}$$

$$C4 = \begin{bmatrix} 9 & 10 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 19 & 9 \end{bmatrix}$$