The MapReduce Paradigm

- Platform for reliable, scalable parallel computing
- Abstracts issues of distributed and parallel environment from programmer.
- Runs over distributed file systems
  - Google File System (GFS)
  - Hadoop File System (HDFS)

Problem Scope

- *MapReduce* is a parallel programming model for data processing
- The power of MapReduce lies in its ability to scale to 100s or 1000s of computers, each with several processor cores
- How large an amount of work?
  - Web-Scale data on the order of 100s of GBs to TBs or PBs
  - It is likely that the input data set will not fit on a single computer's hard drive
  - Hence, a distributed file system (e.g., Google File System- GFS) is typically required

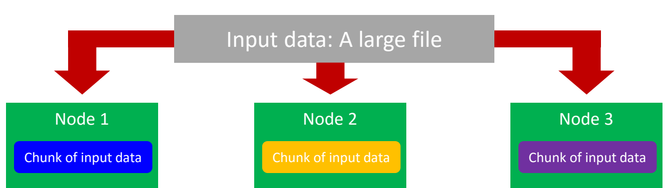
2

Commodity Clusters

- MapReduce is designed to efficiently process large volumes of data by connecting many commodity computers together to work in parallel
- MapReduce divides the workload into multiple *independent tasks* and schedule them across cluster nodes
- A work performed by each task is done *in isolation* from one another

Data Distribution

- In a MapReduce cluster, data is distributed to all the nodes of the cluster as it is being loaded in
- An underlying distributed file systems (e.g., GFS) splits large data files into chunks which are managed by different nodes in the cluster



- Even though the file chunks are distributed across several machines, they form *a single namesapce*

Functional Abstractions Hide Parallelism

- Map and Reduce
- Functions borrowed from functional programming languages (eg. Lisp)
- Map()
  - Process a key/value pair to generate intermediate key/value pairs•
- Reduce()
  - Merge all intermediate values associated with the same key

11/23/10

Relational Databases vs. MapReduce

- Relational databases:
  - Multipurpose: analysis and transactions; batch and interactive
  - Data integrity via ACID transactions
  - Lots of tools in software ecosystem (for ingesting, reporting, etc.)
  - Supports SQL (and SQL integration, e.g., JDBC)
  - Automatic SQL query optimization
- MapReduce (Hadoop):
  - Designed for large clusters, fault tolerant
  - Data is accessed in "native format"
  - Supports many query languages
  - Programmers retain control over performance
  - Open source

Source: O'Reilly Blog post by Joseph Hellerstein (11/19/2008)

MapReduce

- Map: Inputs a key/value pair
  - Key is a reference to the input value
  - Value is the data set on which to operate
- Reduce:
  - Starts with intermediate Key / Value pairs
  - Ends with finalized Key / Value pairs

MapReduce

- MapReduce programs are executed in two main phases, called mapping and reducing:
  - Map: the map function is written to convert input elements to key-value pairs.
  - Reduce: the reduce function is written to take pairs consisting of a key and its list of associated values and combine those values in some way.

WordCount in MapReduce

- Map:
  - For a pair <k1,document> produce a sequence of pairs <token,1>, where token is a token/word found in the document.

- Reduce
  - For a pair <word, list(1, 1, ..., 1)> sum up all ones appearing in the list and return <word, sum>, where sum is the sum of ones.

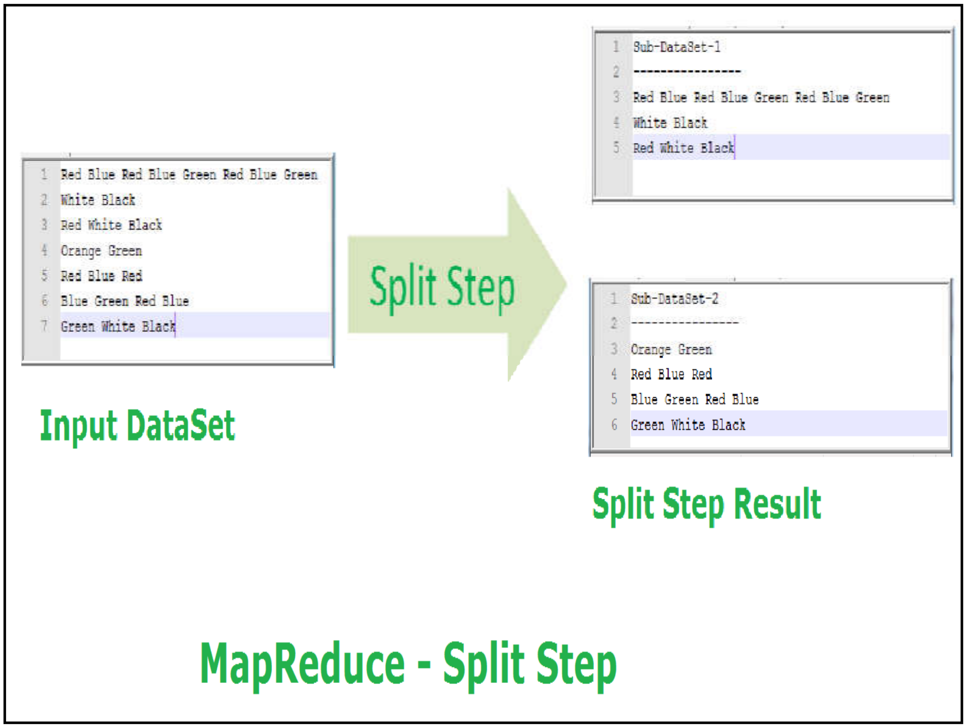
Example

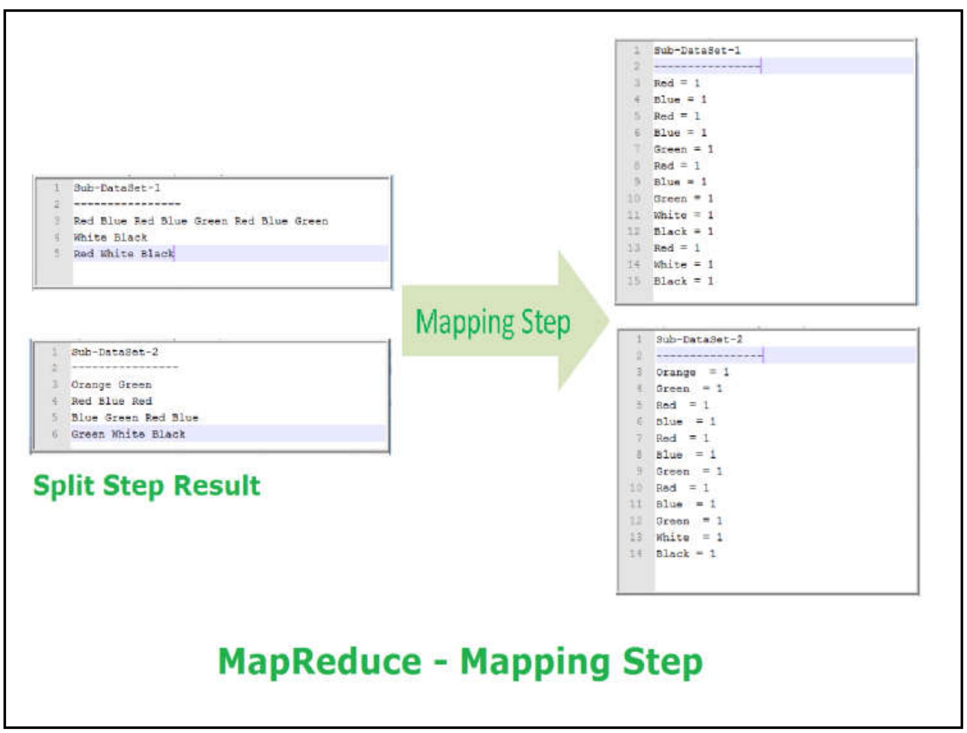
- Count the number of occurrences of each word available in a DataSet given below.

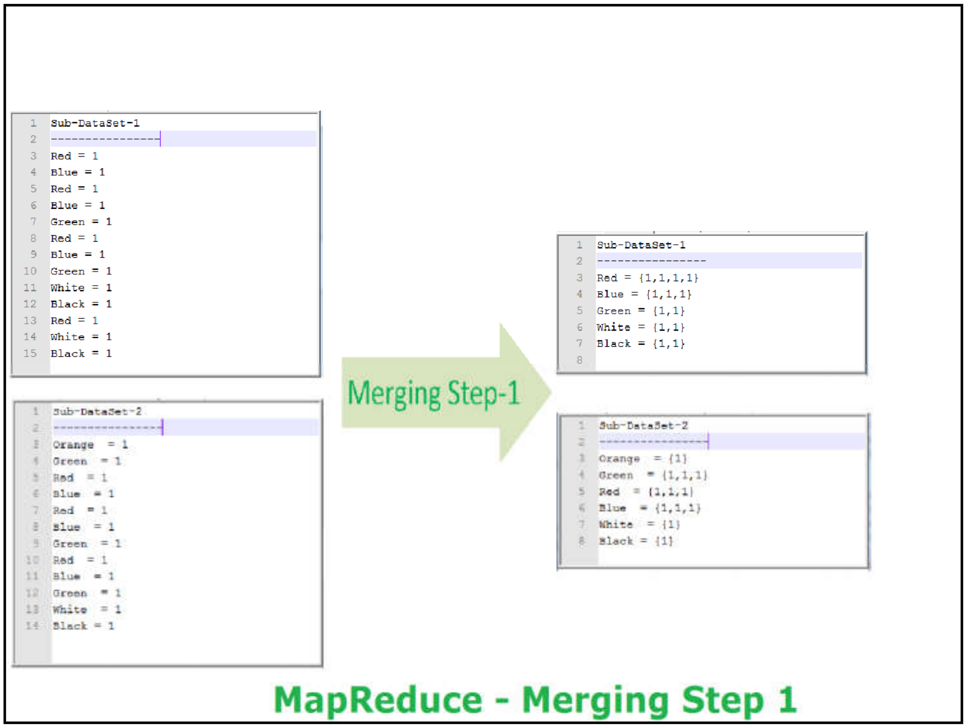
```
Red Blue Red Blue Green Red Blue Green
White Black
Red White Black
Orange Green
Red Blue Red
Blue Green Red Blue
Green White Black
```

```
Black = 3
Blue = 6
Green = 5
Orange = 1
Red = 7
White = 3
```

Final Ouput


1 Red Blue Red Blue Green Red Blue Green
2 White Black
3 Red White Black
4 Orange Green
5 Red Blue Red
6 Blue Green Red Blue
7 Green White Black
Input DataSet
Split Step
1 Sub-DataSet-1
2 ----------------
3 Red Blue Red Blue Green Red Blue Green
4 White Black
5 Red White Black
1 Sub-DataSet-2
2 ----------------
3 Orange Green
4 Red Blue Red
5 Blue Green Red Blue
6 Green White Black
Split Step Result
MapReduce - Split Step


1 Sub-DataSet-1
2 ----------------
3 Red Blue Red Blue Green Red Blue Green
4 White Black
5 Red White Black
1 Sub-DataSet-2
2 ----------------
3 Orange Green
4 Red Blue Red
5 Blue Green Red Blue
6 Green White Black
Split Step Result
Mapping Step
1 Sub-DataSet-1
2 ----------------
3 Red = 1
4 Blue = 1
5 Red = 1
6 Blue = 1
7 Green = 1
8 Red = 1
9 Blue = 1
10 Green = 1
11 White = 1
12 Black = 1
13 Red = 1
14 White = 1
15 Black = 1
1 Sub-DataSet-2
2 ----------------
3 Orange = 1
4 Green = 1
5 Red = 1
6 Blue = 1
7 Red = 1
8 Blue = 1
9 Green = 1
10 Red = 1
11 Blue = 1
12 Green = 1
13 White = 1
14 Black = 1
MapReduce - Mapping Step

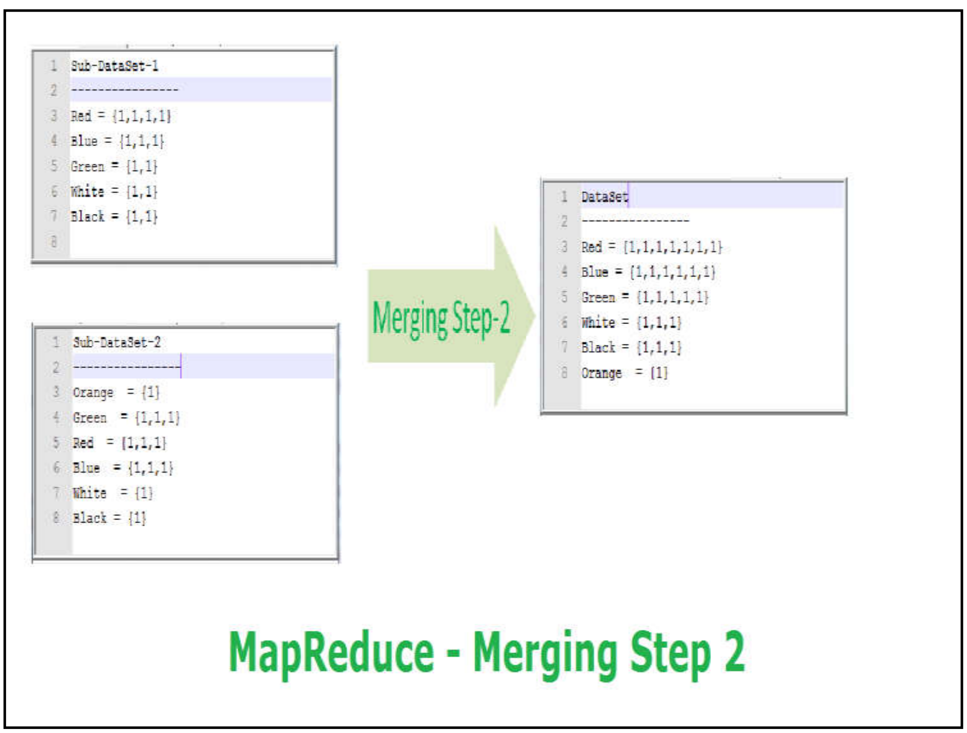

Sub-DataSet-1

Red = 1
Blue = 1
Red = 1
Blue = 1
Green = 1
Red = 1
Blue = 1
Green = 1
White = 1
Black = 1
Red = 1
White = 1
Black = 1
Sub-DataSet-2

Orange = 1
Green = 1
Red = 1
Blue = 1
Red = 1
Blue = 1
Green = 1
Red = 1
Blue = 1
Green = 1
White = 1
Black = 1
Merging Step-1
Sub-DataSet-1

Red = {1,1,1,1}
Blue = {1,1,1}
Green = {1,1}
White = {1,1}
Black = {1,1}
Sub-DataSet-2

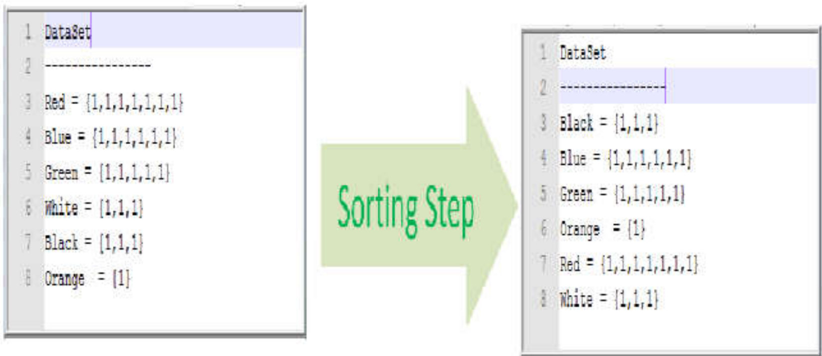
Orange = {1}
Green = {1,1,1}
Red = {1,1,1}
Blue = {1,1,1}
White = {1}
Black = {1}
MapReduce - Merging Step 1


Sub-DataSet-1

Red = {1,1,1,1}
Blue = {1,1,1}
Green = {1,1}
White = {1,1}
Black = {1,1}
Sub-DataSet-2

Orange = {1}
Green = {1,1,1}
Red = {1,1,1}
Blue = {1,1,1}
White = {1}
Black = {1}
Merging Step-2
DataSet

Red = {1,1,1,1,1,1,1}
Blue = {1,1,1,1,1,1}
Green = {1,1,1,1,1}
White = {1,1,1}
Black = {1,1,1}
Orange = {1}
MapReduce - Merging Step 2



MapReduce - Sorting Step

Final Output

MapReduce - Reduce Step

MapReduce Algorithms for Relational Operations

- Selection
- Projection
- Union, and Intersection
- Natural Join

MapReduce Algorithms for Relational Operations

- R, S - relation
- t, t' - a tuple
- $\mathcal{C}$ - a condition of selection
- A, B, C - subset of attributes
- a, b, c - attribute values for a given subset of attributes

Selection

Selections really do not need the full power of map-reduce. They can be done most conveniently in the map portion alone, although they could also be done in the reduce portion alone

$\sigma_C(R)$

- **Map**: For each tuple t in R, test if it satisfies C. If so, produce the key-value pair (t, t). That is, both the key and value are t.
- **Reduce**: The Reduce function is the identity. It simply passes each key-value pair to the output.

Selection

Selections really do not need the full power of map-reduce. They can be done most conveniently in the map portion alone, although they could also be done in the reduce portion alone. Here is a map-reduce implementation of selection $\sigma_C(R)$.

The Map Function: For each tuple t in R, test if it satisfies C. If so, produce the key-value pair (t, t). That is, both the key and value are t.

The Reduce Function: The Reduce function is the identity. It simply passes each key-value pair to the output.

Note that the output is not exactly a relation, because it has key-value pairs. However, a relation can be obtained by using only the value components (or only the key components) of the output.

Projection

Projection is performed similarly to selection, because projection may cause the same tuple to appear several times, the Reduce function must eliminate duplicates. We may compute $\pi_S(R)$ as follows.

The Map Function: For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in S. Output the key-value pair (t', t').

The Reduce Function: For each key t' produced by any of the Map tasks, there will be one or more key-value pairs (t', t'). The Reduce function turns $(t', [t', t', \ldots, t'])$ into (t', t'), so it produces exactly one pair (t', t') for this key t'.

Union

- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: Associated with each key t there will be either one or two values. Produce output (t, t) in either case.

Intersection

- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: If key t has value list $[t, t]$, then produce (t, t). Otherwise, produce nothing.

Example: Join By Map-Reduce

- **Compute the natural join *R(A,B)* ⋈ *S(B,C)***
- *R* and *S* are each stored in files
- Tuples are pairs *(a,b)* or *(b,c)*

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

⋈

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Join in MapReduce (Reduce-side Join)

- Assume to have two relations: R(A, B) and S(B, C)
  - We must find tuples that agree on their B components
- A MapReduce implementation of Natural Join

Map: For a tuple *(a,b)* in R emit a key/value pair (*b*, ('R',a))
For a tuple *(b,c)* in S, emit a key/value pair (*b*, ('S',c))

Reduce: If key *b* has value list [('R',a),('S',c)], emit a key/value pair (*b*, *(a,b,c)*)