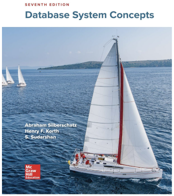
UNIVERSITY OF MARYLAND

UMD DATA605 - Big Data Systems

8.2: Map Reduce



- **Instructor**: Dr. GP Saggese - gsaggese@umd.edu
- **Resources**
  - Silbershatz: Chap 10
  - Ghemawat et al.: The Google File System, 2003
  - Dean et al.: MapReduce: Simplified Data Processing on Large Clusters, 2004

SCIENCE ACADEMY

MapReduce: Overview

- **MapReduce programming model**
  - Inspired by functional programming (e.g., Lisp)
  - Common pattern of parallel programming
  - Basic algorithm
    - Process large number of records
    - Apply `map()` to each record
    - Group results by key
    - Apply `reduce()` to results of `map()`
- **Example**
  - **Goal**: Sum length of all tuples in a document
    - E.g.,
      `[() (a,) (a, b) (a, b, c)]`
  - **map(function, set of values)**
    - Apply function to each value (e.g., len)
      `map(len, [(), (a), (a, b), (a, b, c))]) -> [0, 1, 2, 3]`
  - **reduce(function, set of values)**
    - Combine values using a binary function (e.g., add)
      `reduce(add, [0, 1, 2, 3]) -> 6`

MapReduce: Overview

- Structure of computation
  - **Read input**
    - Sequentially or in parallel
  - **Map**
    - Extract / compute from records
  - **Group by key**
    - Sort and shuffle
  - **Reduce**
    - Aggregate, summarize, filter, transform
  - **Write the result**
- MapReduce framework (e.g., Hadoop, Spark) implements algorithm
- User specifies `map()` and `reduce()` functions to solve problem

MapReduce: Word Count

- **Word Count**
  - "Hello world" of MapReduce
  - Huge text file (can't fit in memory)
  - Count occurrences of each distinct word
- **Sample application**
  - Analyze web server logs for popular URLs
- **Linux solution**

```
> more doc.txt
One a penny, two a penny, hot cross buns.
> words doc.txt | sort | uniq -c
a 2
buns 1
cross 1
...
```

Hot cross buns!
Hot cross buns!
One a penny, two a penny,
Hot cross buns!

If you have no daughters,
Give them to your sons.
One a penny, two a penny,
Hot cross buns![1]

- `words` outputs words one per line
- Unix pipeline is parallelizable in MapReduce sense

MapReduce: Word Count

Action

Read input

Map:

- Invoke **map**() on each input record
- Emit 0 or more output data items

Group by key:

- Gather all outputs from **map**() stage
- Collect outputs by keys

Reduce:

- Combine the list of outputs with same keys

Python code

```
values = read(file_name)

def map(values):
  # values: words in document
  for word in values:
        emit(word, 1)

def reduce(key, values):
   # key: a word
   # value: a list of counts
   result = 0
   # result = sum(values)
   for count in values:
    result += count
   emit(key, result)
```

Example

"One a penny, two a penny, hot cross buns."

Map:

[("one", 1), ("a", 1), ("penny", 1),("two", 1), ("a", 1), ("penny", 1), ("hot", 1), ("cross", 1), ("buns", 1)]

Group by key:

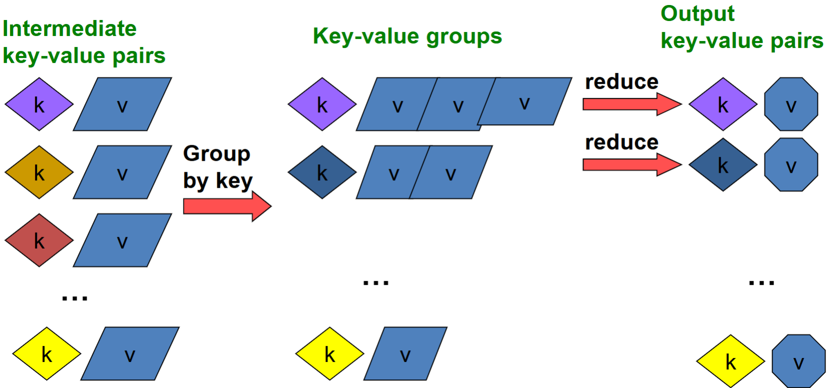
[("a", [1, 1]), ("buns", [1]), ("cross", [1]), ("hot", [1]), ("one", [1]), ("penny", [1, 1]), ("two", [1])]

Reduce:

[("one", 1), ("a", 2), ("penny", 2), ("two", 1), ("hot", 1), ("cross", 1), ("buns", 1)]

MapReduce: Reduce Step

```
reduce(key: word, values: List[int]):
   # key: a word
   # value: an iterator over counts
   result = 0
   for count in values:
    result += count
   emit(key, result)
```



MapReduce: Interfaces

- Input: read key-value pairs **List[Tuple[k, v]]**
- Programmer specifies two methods map and reduce
- **Map(Tuple[k, v]) → List[Tuple[k, v]]**
  - Take a key-value pair and output a set of key-value pairs
    - E.g., key is a file, value is the number of occurrences
    - "One a penny" → [("One", 1), ("a", 1), ("penny", 1)]
  - There is one **Map** call for every *(*k, *v) pair*
- **GroupBy(List[Tuple[k, v]]) → List[Tuple[k, List[v]]]**
  - Group and optionally sort all the records with the reduce key
- **Reduce(Tuple[k, List[v]]) → Tuple[k, v]**
  - All values *v'* with same key *k'* are reduced together
  - There is one **Reduce** call per unique key *k'*
- Output: write key-value pairs **List[Tuple[k, v]]**

SCIENCE
ACADEMY

MapReduce: Log Processing

- Log file recording access to a website with format

```
date, hour, filename
```

- **Goal**: find how many times each files is accessed during Feb 2013
- **Input**
  - Read the file and split into lines
- **Map**
  - Parse each line into the 3 fields
  - If the date is in the required interval emit(dir_name, 1)
- **GroupBy**
  - The reduce key is the filename
  - Accumulate all the (key, value) with the same filename
- **Reduce**
  - Add the values for each list of (key, value) since they have the same filename
  - Output the number of access to each file
- **Output**
  - Write results on disk separated by newline

After Input

```
...
2013/02/21 10:31:22.00EST /slide-dir/11.ppt
2013/02/21 10:43:12.00EST /slide-dir/12.ppt
2013/02/22 18:26:45.00EST /slide-dir/13.ppt
2013/02/22 18:26:48.00EST /exer-dir/2.pdf
2013/02/22 18:26:54.00EST /exer-dir/3.pdf
2013/02/22 20:53:29.00EST /slide-dir/12.ppt
...
```

After Map

```
[`/slide-dir/11.ppt`, 1), ...)]
```

After GroupBy

```
[`/slide_dir/11.ppt`, 1), ...,
(` /slide-dir/12.ppt`, [1, 1]), ..
```

After Reduce

```
[`/slide_dir/11.ppt`, 1), ...,
(`/slide-dir/12.ppt`, 2), ...)]
```

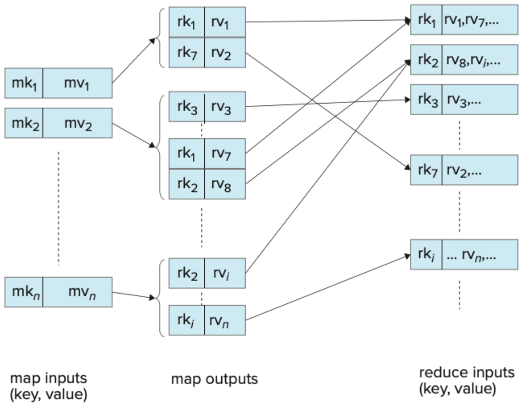
Output

```
/slide_dir/11.ppt 1
...
/slide-dir/12.ppt 2
...
```

SCIENCE ACADEMY

MapReduce: Data Flow

`Focusing on MapReduce functionality / flow of the data to expose the parallelism`

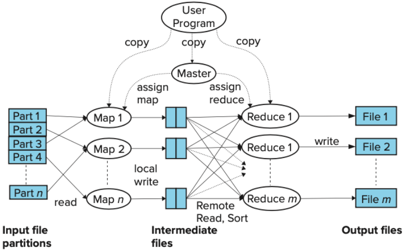


- **Input**
- **Map**
  - mki = map keys
  - mvi = map input values
- **GroupBy**
  - Shuffle / collect the data
- **Reduce**
  - rki = reduce keys
  - rvi** = reduce input values
  - Reduce outputs are not shown

Input **Map** **GroupBy** **Reduce** .

MapReduce: Parallel Data Flow

- **User program** specifies map/reduce code
- **Input data** is partitioned across multiple machines (HDFS)
- **Master** node sends copies of the code to all computing nodes
- **Map**
  - *n* data chunks to process
  - Functions executed in parallel on multiple *k* machines
  - Output data from *Map* is saved on disk
- **GroupBy / Sort**
  - Output data from *Map* is sorted and partitioned based on reduce key
  - Different files are created for each *Reduce* task
- **Reduce**
  - Functions executed in parallel on multiple machines
  - Each work on some part of the data
  - Output data from Reduce is saved on disk
- **Write to disk**

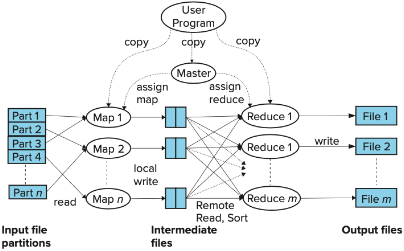


- All operations use HDFS as storage
- Machines are reused for multiple computations (Map, GroupBy, Reduce) at different times

SCIENCE ACADEMY

Master Node Responsibilities

- Master node coordinates
  - Task status: idle, in-progress, completed
  - Schedule idle tasks as workers become available
  - Map task completion sends location and sizes of intermediate files to Master
  - Master informs Reduce tasks
  - Schedule idle Reduce tasks
- Master node pings workers to detect failures



SCIENCE ACADEMY

Dealing with Failures

- **Map worker failure**
  - Reset failed map tasks to idle
  - Notify reduce workers when task is rescheduled
- **Reduce worker failure**
  - Reset in-progress tasks to idle
  - Restart reduce task
- **Master failure**
  - Abort MapReduce task
  - Notify client

How many Map and Reduce jobs?

- **M map tasks**
- **R reduce tasks**
- **N worker nodes**
- Rules of thumb
  - $M \gg N$
    - Pros: Improve dynamic load balancing, Speed up recovery from worker failures
    - Cons: More communication between *Master* and *Worker Nodes*, Lots of smaller files
  - $R > N$
    - Usually $R < M$, Output is spread across fewer files

Refinements: Backup Tasks

- **Problem**
  - Slow workers significantly lengthen the job completion time
  - Slow workers due to:
    - Older processor
    - Not enough RAM
    - Other jobs on the machine
    - Bad disks
    - OS thrashing / virtual memory hell
- **Solution**
  - Near the end of Map / Reduce phase
    - Spawn backup copies of tasks
    - Whichever one finishes first "wins"
- **Result**
  - Shorten job completion time

Refinement: Combiners

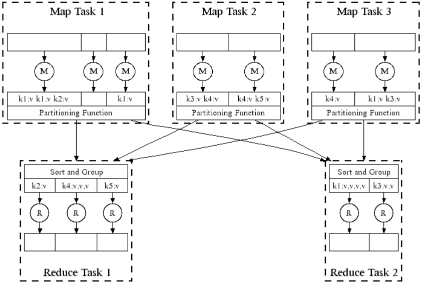
- **Problem**
  - Often a *Map* task produces many pairs for the same key *k*

`[(k1, v1), (k1, v2), ...]`

  - E.g., common words in the word count example
  - Increase complexity of the GroupBy stage
- **Solution**
  - Pre-aggregate values in the *Map* with a *Combine*

`[k1, (v1, v2, ...), k2, ([...])]`

  - *Combine* is usually the same as the *Reduce* function
  - Works only if *Reduce* function is commutative and associative
- **Result**
  - Better data locality
  - Less shuffling and reordering
  - Less network / disk traffic



Refinement: Partition Function

- **Problem**
  - Sometimes users want to control how keys get partitioned
  - Inputs to *Map* tasks are created by contiguous splits of input file
  - MapReduce uses a default partition function **hash(key) mod R**
  - Reduce needs to ensure that records with the same intermediate key end up at the same worker
- **Solution**
  - Sometimes useful to override the hash function:
  - E.g., **hash(hostname(URL)) mod R** ensures URLs from a host end up in the same output file