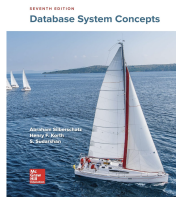


8.2: Map Reduce

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



MapReduce: Overview

- **MapReduce programming model**

- Inspired by functional programming (e.g., Lisp)
- Common pattern of parallel programming to process large number of records

- **Basic algorithm**

- 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 result*

- **Division of responsibilities**

- User specifies `map()` and `reduce()` functions to solve problem
- MapReduce framework (e.g., Hadoop, Spark) implements algorithm

MapReduce: Word Count

- **Word Count**

- “Hello world” of MapReduce
- Huge text file (can't fit in memory)
- Count occurrences of each distinct word

- **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
```

```
...
```

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

- **Sample application**

- Analyze web server logs for popular URLs

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]

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: Log Processing

- **Goal:**
 - Log file records website access (date, hour, filename)
 - Count file accesses in Feb 2013
- **Input**
 - Read file, split into lines
- **Map**
 - Parse line into 3 fields
 - If date is in interval, emit(dir_name, 1)
- **GroupBy**
 - Reduce key: filename
 - Accumulate (key, value) with same filename
- **Reduce**
 - Sum values for each (key, value) list with same filename
 - Output file access count
- **Output**
 - Write results to disk, newline-separated

After Input

```
2013/02/21 10:31:22.00EST /slide-dir/1
2013/02/21 10:43:12.00EST /slide-dir/1
2013/02/22 18:26:45.00EST /slide-dir/1
2013/02/22 18:26:48.00EST /exer-dir/2
2013/02/22 18:26:54.00EST /exer-dir/3
2013/02/22 20:53:29.00EST /slide-dir/1
```

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
...
```

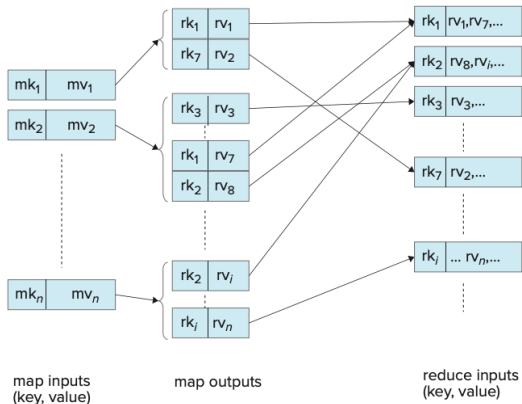


MapReduce: Interfaces

- **Input:** Read key-value pairs `List[Tuple[k, v]]`
- **Programmer** specifies two methods `map` and `reduce`
- **Map**
 - `Map(Tuple[k, v]) \rightarrow 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 occurrences
 - "One a penny" \rightarrow [("One", 1), ("a", 1), ("penny", 1)]
 - One Map call for every (k, v) pair
- **GroupBy**
 - `GroupBy(List[Tuple[k, v]]) \rightarrow List[Tuple[k, List[v]]]`
 - Group and optionally sort records with the reduce key
- **Reduce**
 - `Reduce(Tuple[k, List[v]]) \rightarrow Tuple[k, v]`
 - Reduce all values `v'` with same key `k'`
 - One Reduce call per unique key `*k'`
- **Output:** Write key-value pairs `List[Tuple[k, v]]`

MapReduce: Data Flow

- Focusing on MapReduce flow of the data to expose the parallelism



- Input**

- Map**

- mk_i = map keys
- mv_i = map values

- GroupBy**

- Shuffle / collect the data

- Reduce**

- rk_i = reduce keys
- rv_i = reduce values
- Reduce outputs are not shown

Input

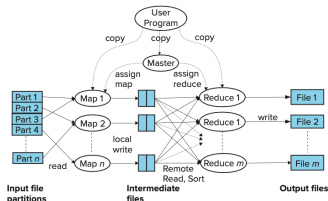
Map

GroupBy

Reduce

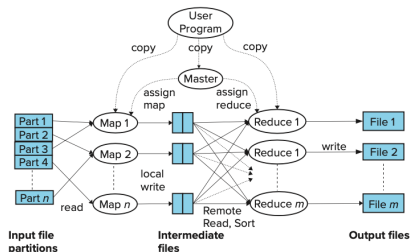
MapReduce: Parallel Data Flow

- **User program** specifies map/reduce code
 - *MasterNode* sends code to all computing nodes
 - *Machines* reused for multiple computations (Map, Reduce) at different times
 - All operations use HDFS as storage
- **Map**
 - n data chunks to process
 - Functions executed in parallel on k machines
 - Output data saved on disk
- **GroupBy / Sort**
 - Output data sorted and partitioned by reduce key
 - Files created for each Reduce task
- **Reduce**
 - Functions executed in parallel on multiple machines
 - Each works on part of the data
 - Output data saved on disk



MasterNode Responsibilities

- *MasterNode* **coordinates / schedule tasks**
 - 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
- *MasterNode* **pings workers to detect failures**
 - Heartbeat



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?

- Number of map tasks = M
- Number of reduce tasks = R
- Number of worker nodes = N
- Typically $M \gg N$
 - Pros:
 - Improve dynamic load balancing
 - Speed up recovery from worker failures
 - Cons:
 - More communication between *MasterNode* and *WorkerNodes*
 - Lots of smaller files
- Typically $R > N$
- 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
[(k_1 , v_1), (k_1 , v_2), ...]
- 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
[(k_1 , (v_1 , v_2 , ...)), (k_2 , [...])]
- 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**

- Users want to control key partitioning
- Inputs to Map tasks created by contiguous input file splits
- Default partition function: `hash(key) mod R`
- Ensure records with the same intermediate key go to the same worker

- **Solution**

- Override hash function:
- E.g., `hash(hostname(URL)) mod R` ensures URLs from a host end up in the same output file

Implementations of MapReduce

- There are many implementations of MapReduce:
 - **Google**
 - Proprietary implementation
 - **Hadoop**
 - Open-source in Java
 - Uses HDFS for storage
 - Used by Yahoo, Facebook for large-scale data processing
 - **Amazon Elastic MapReduce (EMR)**
 - Hadoop MapReduce on Amazon EC2
 - Supports Spark, HBase, Hive, other big data frameworks
 - Used for processing large datasets on AWS
 - **Spark**
 - Open-source cluster-computing framework
 - Known for speed, ease of use
 - Used by Netflix for real-time analytics
 - **Dask**
 - Parallel computing with task scheduling
 - Integrates with Python's ecosystem
 - Suitable for large datasets on a single machine or cluster