



UMD DATA605 - Big Data Systems

8.4: Map Reduce Algorithms

Instructor: Dr. GP Saggese, gsaggese@umd.edu

MapReduce: Applications

- **Intuition**

- Break massive jobs into independent map tasks
- Aggregate via shuffle
- Combine in reduce

- **Major classes of applications**

- Text processing and search
 - E.g., tokenization, inverted index, log analysis
- Large data transforms
 - E.g., ETL pipelines, joins, global sort, deduplication across petabytes
- Data mining and machine learning
 - E.g., Counting co occurrences, feature extraction, *k*-means iterations
- Graph and link analysis
 - E.g., PageRank, connected components

- **Typical outputs**, e.g.,

- Counts
- Aggregates
- Reorganized datasets for downstream systems

Cost Measures for Distributed Algorithms

- What matter is the **real dollar cost** not just $O(\cdot)$
 - *Total cost* \approx CPU + storage + network
 - *Communication cost*
 - Total I/O across all processes, e.g., shuffling 1TB
 - *Elapsed communication cost*
 - Max I/O along the critical path
 - *Elapsed computation cost*
 - Wall clock with p workers, sensitive to skew and stragglers
- **Dominant term heuristic**
 - *"If one cost dominates, ignore the others for first order reasoning"*
- **Practical note**
 - *"Adding more machines trades \$ for time and may not fix skew"*

Total Cost Model for MapReduce

- **Total cost** of a computation

$$C_{total} = C_{compute} + C_{io} + C_{network} + C_{storage}$$

- **Notation**

- $|I|$ input GB, $|S|$ shuffle GB, $|O|$ output GB
- p_m mappers, p_r reducers
- T_m map hours, T_r reduce hours
- $c_{compute} = \$/\text{VM hour}$, $c_{io} = \$/\text{GB I/O}$, $c_{shuffle} = \$/\text{GB shuffle}$

- **Compute cost**

$$C_{compute} = c_{compute} (p_m T_m + p_r T_r)$$

where T_m, T_r include skew and stragglers

- Skew effects = a heavy key or hotspot inflates the heaviest task and the critical path

- **I/O cost**

$$C_{io} = c_{io} (|I| + 2|S| + |O|)$$

- Shuffle often dominates since $\sum_i |S_i| \gg |I|, |O|$

Total Cost Model for MapReduce

- **Notation**

- $|I|$ input GB, $|S|$ shuffle GB, $|O|$ output GB
- $c_{shuffle} = \$/\text{GB shuffle}$ $c_{egress} = \$/\text{GB egress}$, $c_{storage} = \$/\text{GB hour}$
- R HDFS replication

- **Network cost**

$$C_{network} = c_{shuffle}|S| + c_{egress}|O|_{egress}$$

where $|O|_{egress}$ is data leaving the provider

- **Storage cost**

$$C_{storage} = c_{storage}R(|I| + |O|)$$

- **Putting all together**

- Plug in variables and unit prices to get C_{total}
- Apply the dominant term heuristic to prioritize optimization
- Tuning levers
 - Use Combiners
 - Compression
 - Better partitioning
 - Early filtering to reduce $\sum_i |S_i|$ and stragglers

Inverted Index using MapReduce

- **Goal:** build a mapping from words to the list of documents they appear in
- **Example**
 - Input in doc1 = MapReduce is powerful
 - Output = [(MapReduce, doc1), (is, doc1), (powerful, doc1)]
 - *Map phase:*
 - Input: (docID, content)
 - Emit: (word, docID) for each word in content
 - *Reduce phase:*
 - Input: (word, [docID_1, docID_2, ...])
 - Emit: (word, list of unique docIDs)
- **Implementation considerations**
 - Useful in search engines and information retrieval
 - Requires tokenization and normalization of content
 - Deduplication of document IDs in reducer

Join Operations using MapReduce

- **Goal:** join two datasets based on a common key
 - Types of joins:
 - Inner join, left/right outer join, full outer join
- **Example**
 - Join `EmployeeRecords(empID, name, deptID)` with `Department(deptID, deptName)`
 - Join key: `deptID`
 - *Map phase:*
 - Emit key as `deptID`, tag each record with source label (e.g., "E" or "D")
 - Example output: `(deptID, ("E", empRecord))`, `(deptID, ("D", deptRecord))`
 - *Reduce phase:*
 - Input: `(deptID, [list of tagged records])`
 - Cross-product logic based on tags to perform the actual join
- **Implementation considerations**
 - Use composite values with source tag to distinguish record origin
 - Optimize data layout to minimize network shuffle
 - Ensure proper partitioning so all records with the same key go to the same reducer

Sorting and Grouping in MapReduce

- **Goal:** organize data by keys or values for further analysis
- **Example**
 - Sort sales data by date or group by product ID
 - *Map phase:*
 - Emit data with key as sort/group criterion
 - *Shuffle phase:*
 - Shuffle and sort phase automatically sorts by key
 - *Reduce phase:*
 - Receives sorted keys and can perform grouped aggregation
- Often used as a preprocessing step for reporting

Graph Processing with MapReduce

- Many graph algorithms are iterative
 - Requires multiple MapReduce rounds for convergence
 - Each iteration refines the scores
- PageRank:
 - Compute importance score of web pages
 - Graph is represented as adjacency lists
 - *Map phase*:
 - Emit contributions to neighboring nodes
 - *Reduce phase*:
 - Sum contributions to update PageRank score

Statistical Aggregation and Log Analysis

- **Statistical Aggregation**
 - Example: (sensorID, temperature) → compute average temperature per sensor
 - *Map phase*: emit relevant quantities (e.g., value, 1 for count)
 - *Reduce phase*: compute sums, averages, variances
- **Log Analysis**
 - Example: (status code, 1) → count occurrences
 - *Map phase*: parse logs, extract fields (timestamp, IP, status) and emit (key, value) for desired metrics
 - *Reduce phase*: aggregate (e.g., count errors, hits per IP)
- **Implementation considerations**
 - Useful for monitoring, alerting, and trend analysis
 - Can handle large-scale logs from distributed systems