SE256:Jan16 (2:1)

L11:MapReduce Advanced Topics

Yogesh Simmhan

16 Mar, 2016

Midterm Exam Review (YS)

- MapReduce features (a)
  - Map
    - **Required** to be Commutative across different <K,V>. Order in which Map applied to input data items must not matter.
    - Map called concurrently across different <K,V>, in no given order.
  - Reduce
    - **Required** to be Commutative over intermediate <K,V[]>. Order across reducer tasks not defined (sorted only within a reducer).
    - **Required** to be Commutative over V[] within a Reduce function.
    - **Optionally** Associative over V[] within a Reduce function.
      - Allows for Reduce to be *reused* as Combiner.
      - If Reduce is not associative, separate Combiner **still possible**.

- MapReduce features (b) [Lin, Ch 3.1]
  - Combiner
    - Called 0, 1 or more times in Map task. No minimum guarantee.
    - Typically, Hadoop MapReduce calls combiner in
      - Map task, once per spill file
      - Map task, once over merged spill files, if spill files > 3
      - Reduce task, during merge
    - Combiner called after (mini) shuffle/sort
  - Using state-object
    - Guarantees map.configure() and map.close() called once
    - Faster due to light-weight management
    - But requires additional user code (shuffle)
    - Requires active memory management
    - Potential for bugs

- MapReduce features (c) [White, Ch 7]
  - Map/Reduce being idempotent allows only failing tasks to be rerun, rather than application to be rerun
  - Allows speculative execution of Map/Reduce over splits if one of them is slower (fastest one wins)
  - Effects of Map/Reduce functions should not be visible to other Map/Reduce function

- Scalability (a) [Leskovec, Ch 2.5, 2.6]
  - Typically will benefit from Weak scaling
    - As many Map tasks as the number of splits. Controlling size/number of splits controls weak scaling. Number of splits proportional to data size.
    - As many Reduce tasks as defined in code, or number of unique keys/partitions. Controlling partitioner will control weak scaling of Reduce. Weak scaling limited by value distribution for keys too.
    - Shuffle & sort may also benefit from weak scaling on Map and Reduce sides.
  - May benefit from strong scaling too
    - Map tasks operate on large splits
    - Reduce tasks operate on large partitions

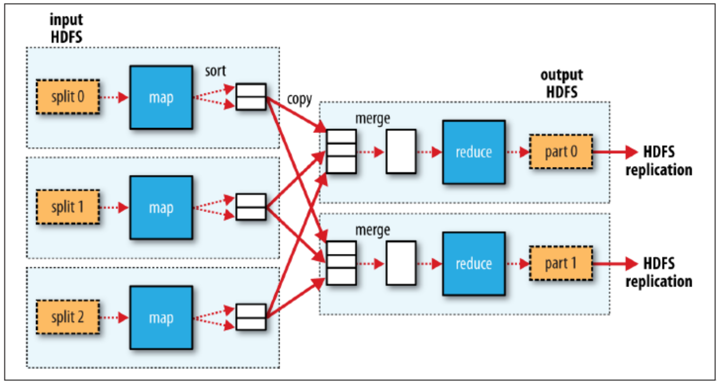


Figure 2-4. MapReduce data flow with multiple reduce tasks

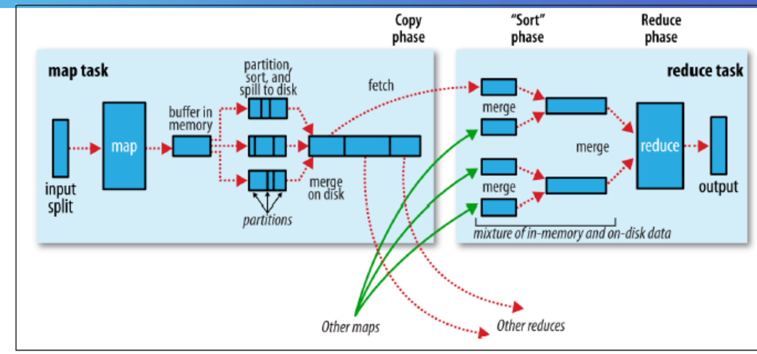


Figure 7-4. Shuffle and sort in MapReduce

- Scalability (b)

  m=d/b. As many Map tasks as the number of splits. Typically HDFS block size, but can be overridden.

- Scalability (c) [Leskovec, Ch 2.5, 2.6]
  - *Makespan* includes Map , shuffle/sort and Reduce times
  - r=1
    - Less partition, shuffle, merge on Map to be done.
    - Merge/Sort done by single Reduce task.
    - Single reducer may be overwhelmed if large number of <K,V[]>
    - BW on reducer is constrained
  - r=k
    - Extreme number of partitions, merges in Map if k is large.
    - Merge/Sort done in parallel by reducer tasks
    - Overheads of creating reducer tasks if few values per key
  - Pick r to trade-off number of <k,v[]> per reducer to offset overhead of creating reducers; balance computation per reducer.

- Hadoop & HDFS
  - Replication
    - Reliability of data in HDFS since it runs on commodity hardware. Control over degree of replication/reliability.
    - Availability of data even with node failures.
    - Flexibility in scheduling Map tasks due to multiple copies. Can improve compute and network performance by picking “closest” replica.
    - Low sync cost of replication due to write-once, ready-many model.
  - Location of replicas [White, Ch 3]
    - Rack aware placement of replicas to improve reliability, availability, and network bandwidth.
    - With 3 replicas: one replica on node in local/random rack, another on a node in a remote rack, and third on a different node in remote rack
    - Reliability, performance, load distribution

- Hadoop & HDFS [White, Ch 7]
  - Spill files
    - Allows Map to generate more output than local memory
    - Prevents out of memory exceptions
    - Allows incremental shuffle/sort
  - Sorting
    - In-memory sort over each spill file for a Map task
    - One or more rounds of Merge-sort of all spill files for a Map task
    - One or more rounds of Merge-sort of input files for a Reduce task

Assignment B
5pm IST Mar 30, 2016

- Replicate Google's search engine from 10 years back using the *Common Crawl* dataset.
  - Find the web pages that match a given set of search terms
  - Rank the webpages and return the top ranked 100 matches

a) MapReduce job to build an inverted index based on Common Crawl web pages, *ignoring stop words* [1]

```
<URL,Webpage> → <Word,URL>*
```

This MapReduce job will be *run once* to build the index and save it to HDFS.

b) Using *web graph* of Common Crawl data (Assg A), write MapReduce job to run PageRank algorithm.

```
<v1, <v2*>> → <v1, pr1>*
```

This MapReduce job will be *run once* to build the webpage's ranks and saved to HDFS.

c) Given a search phrase, write MapReduce job(s) to:

- Find ALL web pages (URLs) that contain ALL the search terms using the inverted index.
- For matching URLs, lookup their PageRank and identify the top 100 pages with the highest page rank.
- Return the ranked list of 100 matching URL for the search
- These will be run for each search phrase

- Report scalability of (a) and (b).
- Report correctness and latency for (c).

d) Train a classifier in a distributed manner such that given a webpage's content (e.g., title, content, etc.) from the Common Crawl dataset, it identifies the *country* in which it is hosted.

- Training and evaluation data for this classifier may be obtained by doing a reverse IP geo-lookup on the IP address associated with each page.
- Of course, this information will be hidden to the classifier during test time.
- Please report on data preparation, classifier training strategies, brief implementation details, and final evaluation accuracy.

E.g. http://lite.ip2location.com/

More MR Topics

Inverted Indexes

- Each Map task parses one or more webpages
  - Input: A stream of webpages (WARC)
  - Output: A stream of (term, URL) tuples
    - (long, http://gb.com) (long, http://gb.com) (ago, http://gb.com) … (long, http://jn.in) (years, http://jn.in) (ago, http://jn.in) …
- Shuffle sorts by key and routes tuples to Reducers
- Reducers convert streams of keys into streams of inverted lists
  - Sorts the values for a key (why?) and builds an inverted list
  - Output: (long, [http://gb.com, http://jn.in]), (ago, [http://gb.com, http://jn.in]), (years, [http://jn.in])

Optimizations & Extensions

- URL sizes may be large
  - Replace URLs with unique longs, URL ID
    - Mapping from URL ID to URL saved as a file
    - Inverted Index has <term, [URL ID]+>
  - Skip stop words with lot of matching URLs
  - Use combiners
- Partition term by prefix alphabet(s)
  - One reducer for each term starting with “a”, “b”, etc.
  - Part file from each reducer has terms with unique a starting letter
- Additional metadata
  - Idea: Include a mapping from URL ID to <URL, PageRank>?
  - Include “term frequency” of term occurrence per URL ID in Inverted Index?

Challenge

- Even using URL IDs, all IDs per term *may not fit in reduce memory for sorting*
    - E.g. 17M URLs in 1% of CC data.
    - Say 1000 unique words per URL.
    - So 17B keys and values generated by Mappers.
    - Say 50,000 unique words (keys) in English
    - One key would on average have 17B/50K=340K URL IDs
        - Peak values would be much higher
- Use a value-to-key conversion design pattern
    - Let MR perform sorting, Reducer just emits result

```
class Mapper
    method Map(docid n, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
        for all term t ∈ H do
            Emit(tuple ⟨t, n⟩, tf H{t})

class Reducer
    method Initialize
        t_prev ← ∅
        P ← new PostingsList
    method Reduce(tuple ⟨t, n⟩, tf [f])
        if t ≠ t_prev ∧ t_prev ≠ ∅ then
            Emit(term t, postings P)
            P.Reset()
        P.Add(⟨n, f⟩)
        t_prev ← t
    method Close
        Emit(term t, postings P)
```

- Mapper emits <**<term, URL ID>**, tf>
    - i.e. compound key
- Partitioner sends all terms to the same reducer
- Per reducer, MR sorts based on compound key <term, URL ID>
- Only one value for each compound key
- Reduce task gets list of term and URL ID in sorted order
    - When new term seen, flush index for “prev” term and start new term
- E.g.
    - <<Ago, 1>, tf1>
    - <<Ago, 7>, tf7>
        - Flush <Ago, [<1,tf1>,<7,tf7>]
    - <<Long, 3>, tf3>
    - <<Long, 4>, tf4>
    - <<Long, 6>, tf6>
        - Flush <Long, [<3,tf3>,<4,tf4>,<6,tf6>]

Lin, Figure 4.4

Lookup of Terms

- Each Map task loads one of the index files, say, by alphabet
- Input terms e.g. “t1 & t2 & t3” passed to each Map task as AND search
- Map does lookup and sends <URL ID, t_i> to reducer
  - Optionally send <<PR, URL ID>, t_i> for sorting by PR
- Reducer does set intersection of all t_i for a URL ID
  - If all terms match, looks up URL for the URL ID
  - If PR stored for each URL, that is returned too

PageRank

- Centrality measure of web page quality based on the web structure
  - How important is this vertex in the graph?
- Random walk
  - Web surfer visits a page, randomly clicks a link on that page, and does this repeatedly.
  - How frequently would each page appear in this surfing?
- Intuition
  - Expect high-quality pages to contain "endorsements" from many other pages thru hyperlinks
  - Expect if a high-quality page links to another page, then the second page is likely to be high quality too

PageRank, recursively

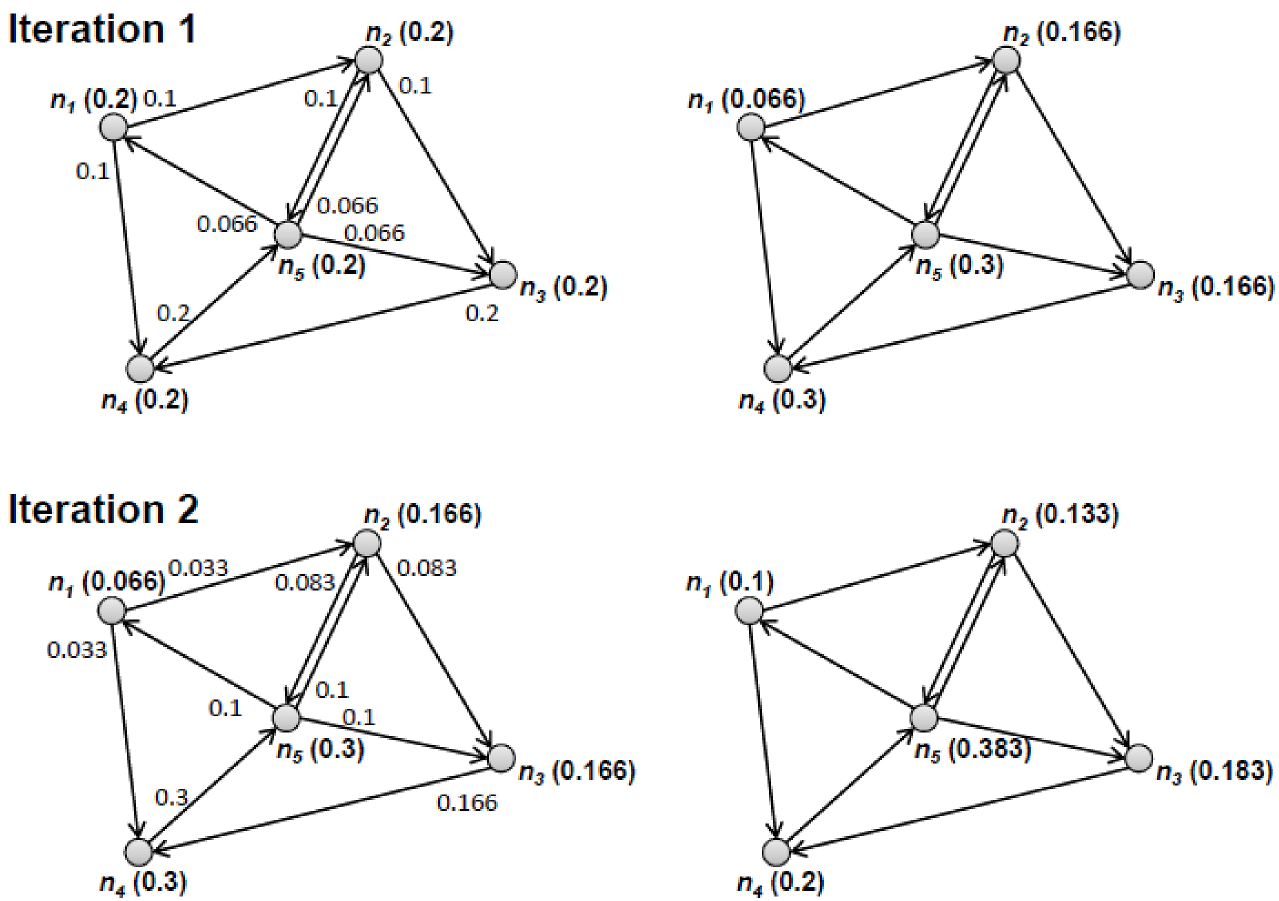
$$P(n) = \alpha \left(\frac{1}{|G|} \right) + (1-\alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

- P(n) is PageRank for webpage/URL 'n'
  - Probability that you're in vertex 'n'
- |G| is number of URLs (vertices) in graph
- α is probability of random jump
- L(n) is set of vertices that link to 'n'
- C(m) is out-degree of 'm'

PageRank Iterations

$\alpha=0$

Initialize $P(n)=1/|G|$



Lin, Fig 5.7

PageRank using MapReduce

```
class Mapper
    method Map(nid n, node N)
        p ← N.PageRank/|N.AdjacencyList|
        Emit(nid n, N)                              ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, p)                          ▷ Pass PageRank mass to neighbors
```

```
class Reducer
    method Reduce(nid m, [p1, p2, ...])
        M ← ∅
        for all p ∈ counts [p1, p2, ...] do
            if IsNode(p) then
                M ← p                               ▷ Recover graph structure
            else
                s ← s + p                           ▷ Sum incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```


Lin, Fig 5.8

PageRank using MapReduce

- MR run over multiple iterations (typically 30)
  - *The graph structure itself must be passed from iteration to iteration!*
- Mapper will
  - Initially, load adjacency list and initialize default PR
    - `<v1, <v2>+>`
  - Subsequent iterations will load adjacency list and new PR
    - `<v1, <v2>+, pr1>`
  - Emit two types of messages from Map
    - PR messages and Graph Structure Messages
- Reduce will
  - Reconstruct the adjacency list for each vertex
  - Update the PageRank values for the vertex based on neighbour's PR messages
  - Write adjacency list and new PR values to HDFS, to be used by next Map iteration
    - `<v1, <v2>+, pr1'>`

Reading

- Lin, Chapters 3, 4, 5