MapReduce: Simplified Data Processing on Large Clusters

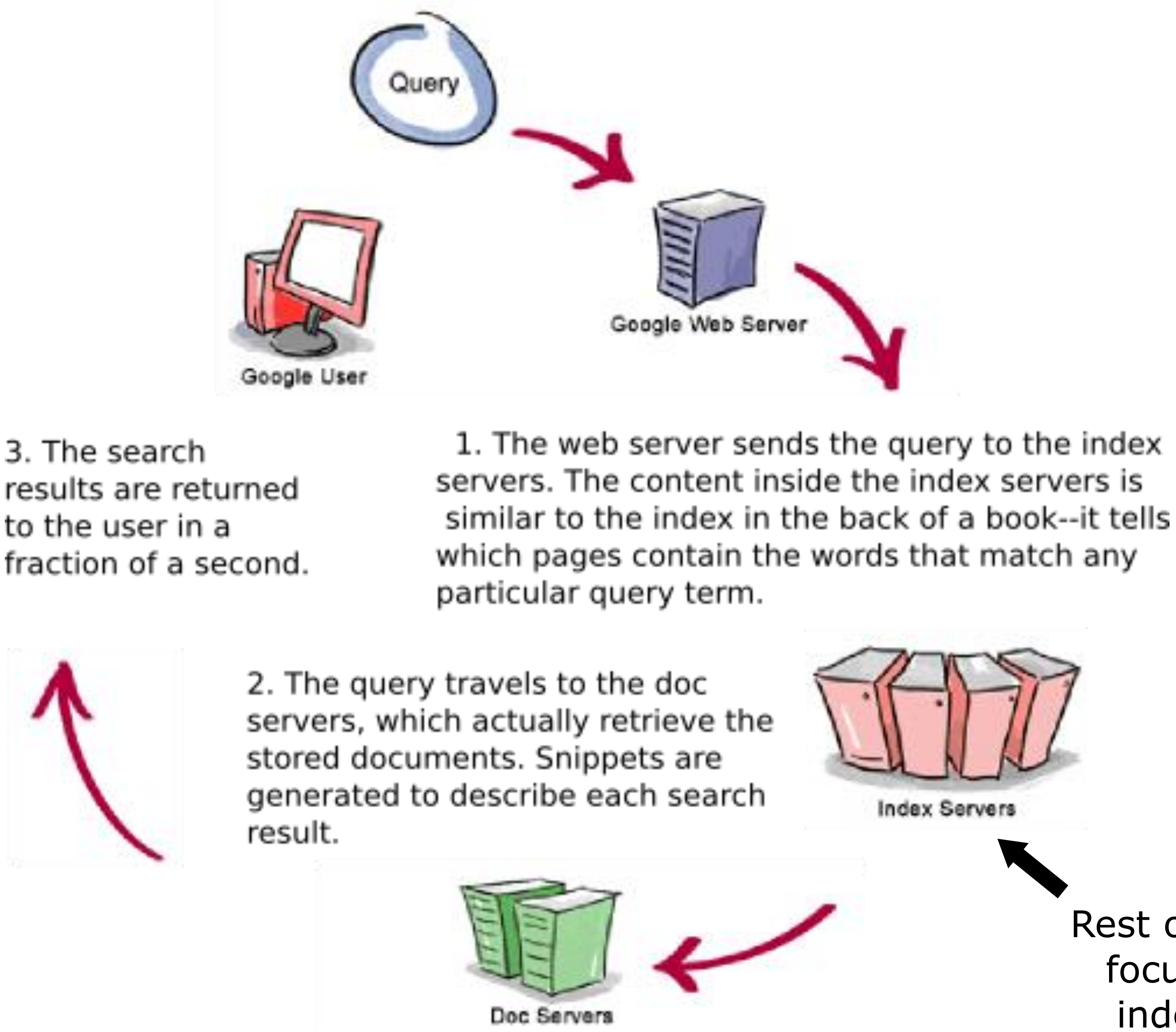
Ashvin Goel

Electrical and Computer Engineering
University of Toronto

ECE1724

Authors: Jeffrey Dean and Sanjay Ghemawat

How Google Works



1. The web server sends the query to the index servers. The content inside the index servers is similar to the index in the back of a book--it tells which pages contain the words that match any particular query term.

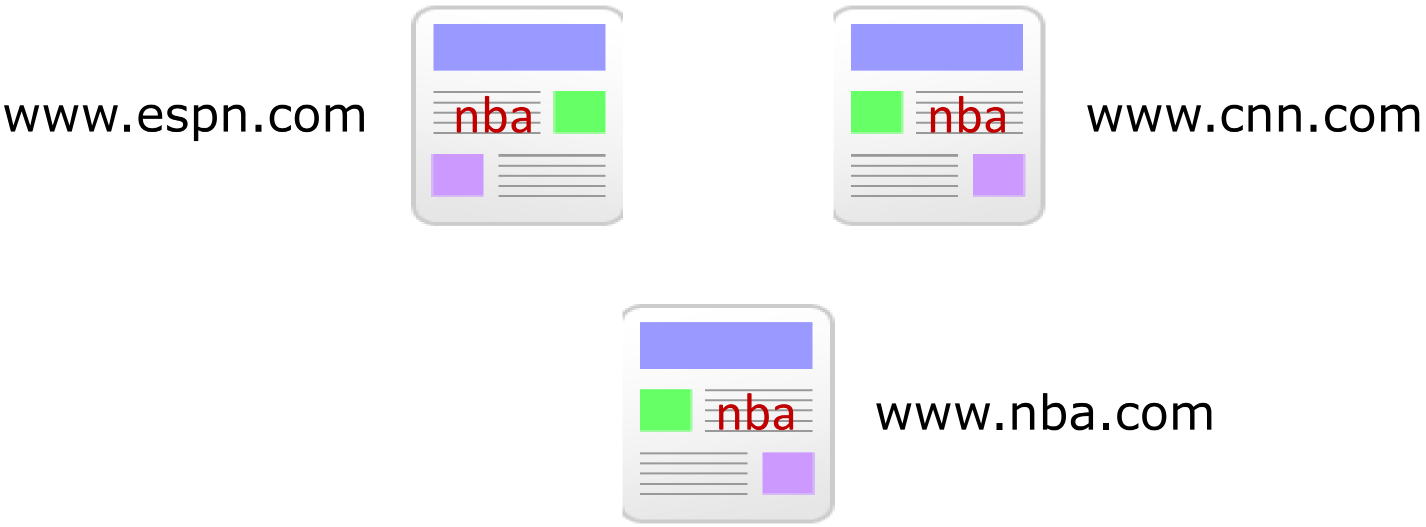
2. The query travels to the doc servers, which actually retrieve the stored documents. Snippets are generated to describe each search result.

3. The search results are returned to the user in a fraction of a second.

Rest of the lecture focuses on the index servers

Two Indexing Challenges

- Web page indexing: which webpages contain given keyword (e.g., “NBA”)?
  - Need to crawl and analyze all web pages
  - Output: <word, list(URLs)>
    - Example: <“NBA”, (www.nba.com, www.espn.com, …)>



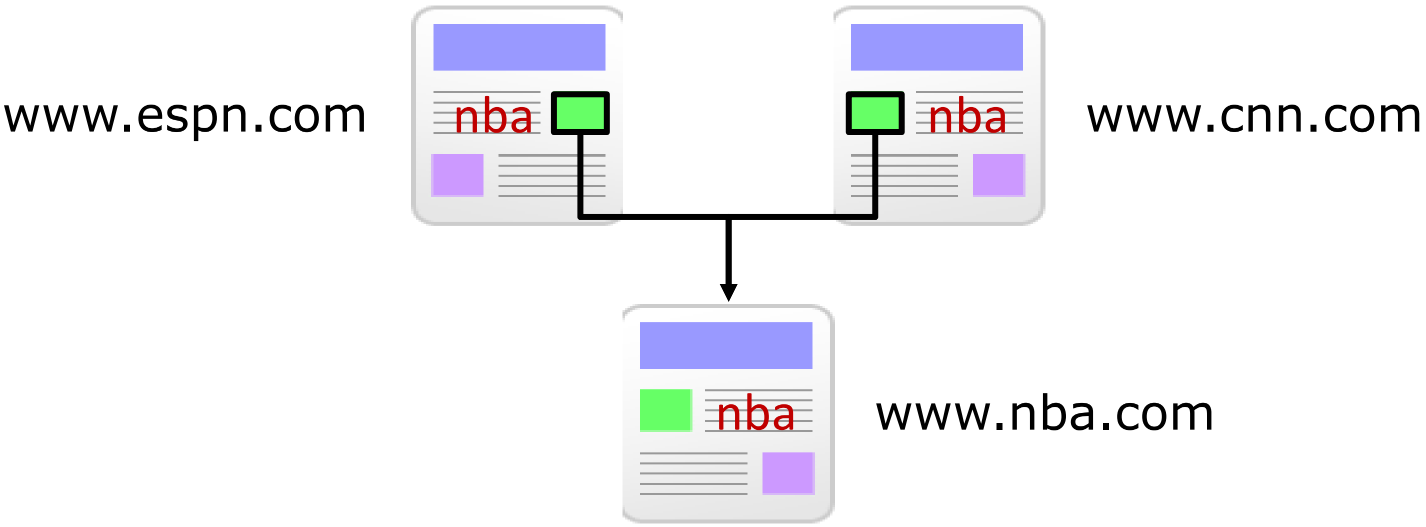
www.espn.com

www.cnn.com

www.nba.com

Two Indexing Challenges

- Web page ranking: which webpages are important for a given keyword?
  - Need to first find source pages that link to a target page
  - Output: <target url, list(source url)>
    - Example: <www.nba.com, (www.espn.com, www.cnn.com, …)>



  - Need to rank pages based on output (PageRank)

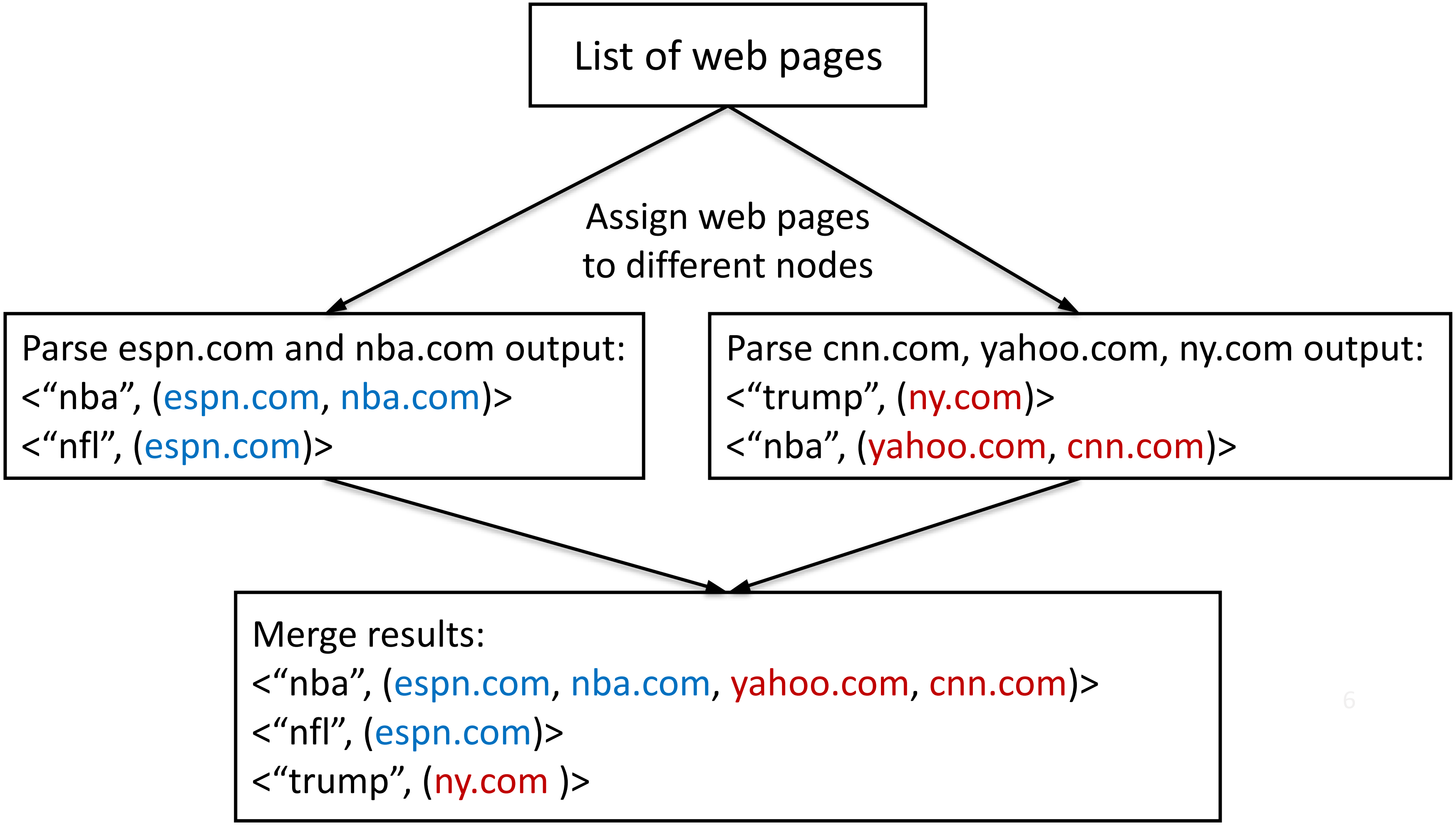
Web Page Indexing

```
// input: list of all web pages
// output: for each word, list of web pages that contain the word

index(List webpages) {
  Hash output = new Hash<string word, List<string url>>;

  for each page p in webpages {
    for each word w in p {
      if (!output.exists(w))
        output{w} = new List<string>;
      // append web page for this word
      output{w}.push(URL(p));
    }
  }
}
```

How can we scale with billions of web pages?

Parallel Web Page Indexing

- Need to parallelize indexing on multiple machines



List of web pages

Assign web pages
to different nodes

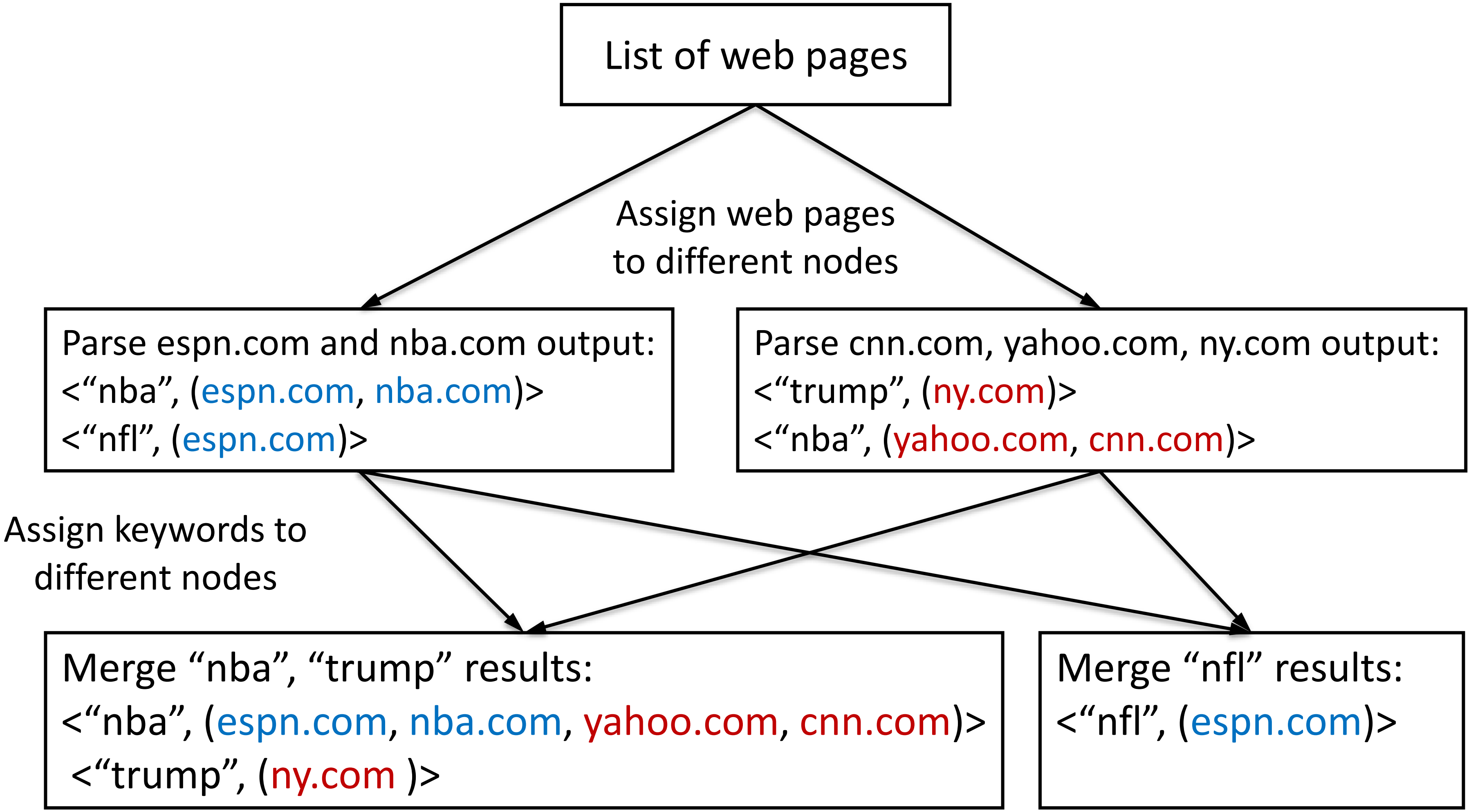
Parse espn.com and nba.com output:
<“nba”, (espn.com, nba.com)>
<“nfl”, (espn.com)>

Parse cnn.com, yahoo.com, ny.com output:
<“trump”, (ny.com)>
<“nba”, (yahoo.com, cnn.com)>

Merge results:
<“nba”, (espn.com, nba.com, yahoo.com, cnn.com)>
<“nfl”, (espn.com)>
<“trump”, (ny.com)>

Parallel Web Page Indexing

- What if we also want to parallelize the merge process?



List of web pages

Assign web pages to different nodes

Parse espn.com and nba.com output:
<"nba", (espn.com, nba.com)>
<"nfl", (espn.com)>

Parse cnn.com, yahoo.com, ny.com output:
<"trump", (ny.com)>
<"nba", (yahoo.com, cnn.com)>

Assign keywords to different nodes

Merge "nba", "trump" results:
<"nba", (espn.com, nba.com, yahoo.com, cnn.com)>
<"trump", (ny.com)>

Merge "nfl" results:
<"nfl", (espn.com)>

```
// index a subset of web pages
index(List webpages) {
  Hash output = new Hash<string word,
                List<string url>>;

  foreach page p in webpages {
    for each word w in p {
      if (!output.exists(w))
        output{w} = new List<string>;
      // append web page for word w
      output{w}.push(URL(p));
    }
  }

  // partition data
  // send output to merge servers
  foreach word w in keys(output) {
   if (w in range [‘a’ – ‘d’])
    send(merge_serverA, output{w});
   else if (w in range [‘e’ – ‘h’]
    send(merge_serverB, output{w});
   .. ..
  }
}
```

```
merge() {
  // while any index server has data
  while (index_serverN sends data) {
    // receive data
    recv(index_serverN, output{w});
    // merge results in final_output
    final_output{w}.push(output{w});
  }
}
```

Problem

final_output stores results for all words, what if it is so large that merge() runs out of memory?

```
// index a subset of web pages
index(List webpages) {
  Hash output = new Hash<string word,
                List<string url>>;

  foreach page p in webpages {
    for each word w in p {
      if (!output.exists(w))
        output{w} = new List<string>;
      // append web page for word w
      output{w}.push(URL(p));
    }
  }

  // partition data
  // send output to merge servers
  foreach word w in keys(output) {
   if (w in range [‘a’ – ‘d’])
    send(merge_serverA, output{w});
   else if (w in range [‘e’ – ‘h’]
    send(merge_serverB, output{w});
   .. ..
  }
}
```

```
merge() {
  // while any index server has data
  while (index_serverN sends data) {
    // receive and buffer data
    // in output, possibly on disk
    output += recv(index_serverN,
                   output{w});
  }
  // group output by word,
  // may require disk-based sort
  group_by_word(output);

  foreach w in keys(output) {

    // merge results in final_output
    final_output{w}.push(output{w});

    if (w != prev_w) {
      // done with prev_w
      // write prev_w output to disk
      write(final_output{prev_w});
    }
  }
}
```

Are we done?

Not So Fast!

- Need to handle failures
  - What if indexer is slow or fails?
    - Need to restart the indexer, mergers need to wait
  - What if merger fails?
    - Need to restart merger, need to wait for all mergers to finish
  - Need to ensure idempotent operation under all failures
    - Operation can be run multiple times, without additional side-effects
- What if partitioning is skewed?
  - E.g., frequency of words by initial letters is not the same
    - S (12%), C (9%), P, …. Y, Z (0.38%), X (0.09%)
  - Leads to load imbalance at merger
    - Need to repartition output of indexer for better performance

```
// index a subset of web pages
index(List webpages) {
  Hash output = new Hash<string word,
                    List<string url>>;

  foreach page p in webpages {
    for each word w in p {
      if (!output.exists(w))
        output{w} = new List<string>;
      // append web page for word w
      output{w}.push(URL(p));
    }
  }

  // partition data
  // send output to merge servers
  foreach word w in keys(output) {
   if (w in range [‘a’ – ‘d’])
    send(merge_serverA, output{w});
}
```

```
merge() {
  // while any index server has data
  while (index_serverN sends data) {
    // receive and buffer data
    // in output, possibly on disk
    output += recv(index_serverN,
                   output{w});
  }
  // group output by word,
  // may require disk-based sort
  group_by_word(output);

  foreach w in keys(output) {

    // merge results in final_output
    final_output{w}.push(output{w});

    if (w != prev_w) {
      // done with prev_w
}
```

What if programmers only had to write code inside the boxes?

Solution: MapReduce

- Programming model for big data analytics
- Programmer writes two fns, called map and reduce

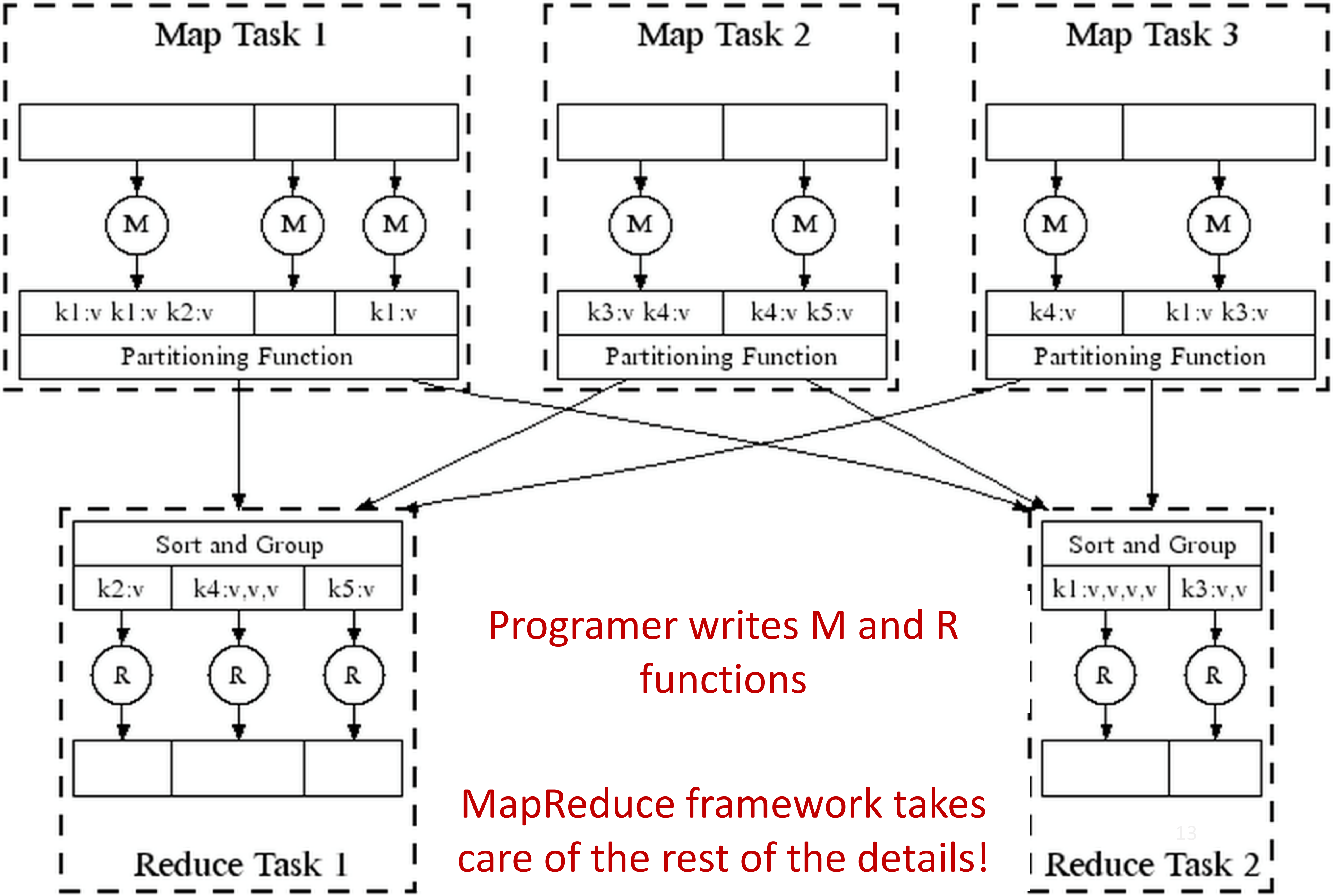
```
map(in_key, in_value)-> list(out_key, intermediate_val)
```

Processes input key/value pair, produces set of intermediate pairs

```
reduce(out_key, list(intermediate_val))-> list(out_key, outvalue)
```

Processes a set of intermediate key-values, produces value for each key

- Widely used model
  - At Google, used for indexing and many analytic jobs
  - Hadoop (open-source version)
    - Used by > 50% of the Fortune 50 companies



Programer writes M and R functions

MapReduce framework takes care of the rest of the details!

Web Page Indexing With MapReduce

```
// input: <url, web page content>
map(url, content) {
  for each word w in content {
    // output: <word, url>
    Emit(<w, url>);
  }
}

// input: <word, list of url>
reduce(char *word, List<url> l) {
  if (!final_output.exists(word))
    final_output{word} = new List<url>;

  // output: <word, list(url)>
  foreach url in l {
    final_output{word}.push(url);
  }
}
```

MapReduce Framework:

Mapper:

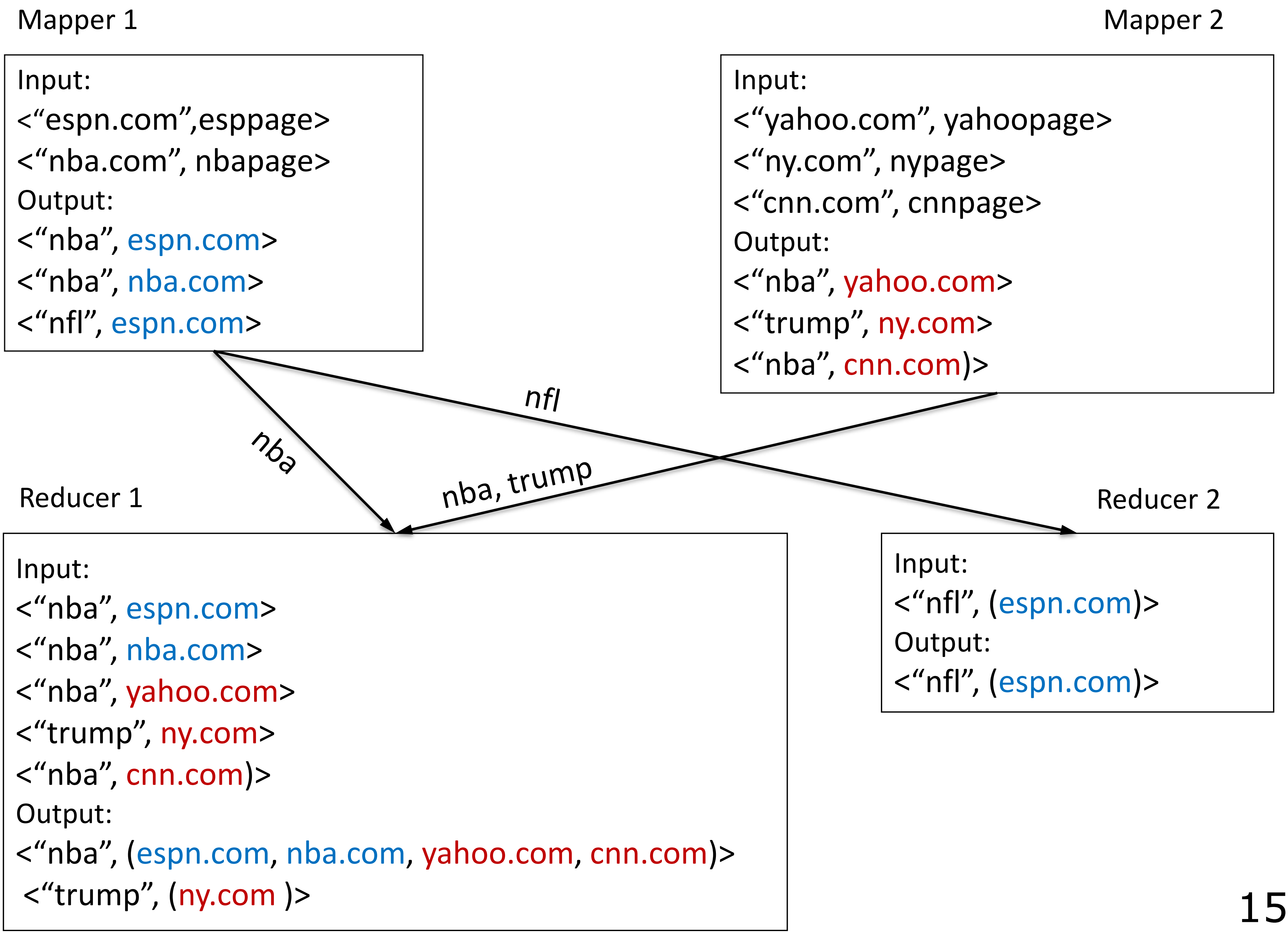
- Partitions intermediate output
- Sends same keys to same reducer

Reducer:

- Receives data
- Sorts and groups data by key

Master:

- Performs error handling



Mapper 1

Input:
<“espn.com”,esppage>
<“nba.com”, nbapage>
Output:
<“nba”, espn.com>
<“nba”, nba.com>
<“nfl”, espn.com>

Mapper 2

Input:
<“yahoo.com”, yahoopage>
<“ny.com”, nypage>
<“cnn.com”, cnnpage>
Output:
<“nba”, yahoo.com>
<“trump”, ny.com>
<“nba”, cnn.com)>

Reducer 1

Input:
<“nba”, espn.com>
<“nba”, nba.com>
<“nba”, yahoo.com>
<“trump”, ny.com>
<“nba”, cnn.com)>
Output:
<“nba”, (espn.com, nba.com, yahoo.com, cnn.com)>
<“trump”, (ny.com)>

Reducer 2

Input:
<“nfl”, (espn.com)>
Output:
<“nfl”, (espn.com)>

Reverse Web Links With MapReduce

```
// input: <url, web page content>
map(url, content) {
  for each target_url in content {
    // output: <target_url, url>
    Emit(<target_url, url>);
  }
}

// input: <target_url, list of url>
reduce(target_url, List<url>l) {
  if (!final_output.exists(target_url))
    final_output{target_url} = new List<url>;

  // output: <target_url, list(url)>
  foreach url in l {
    final_output{target_url}.push(url);
  }
}
```

Just need to replace word with target_url!

Map-Reduce Architecture

Locality optimization: map/reduce tries to run map task on machine storing the file split

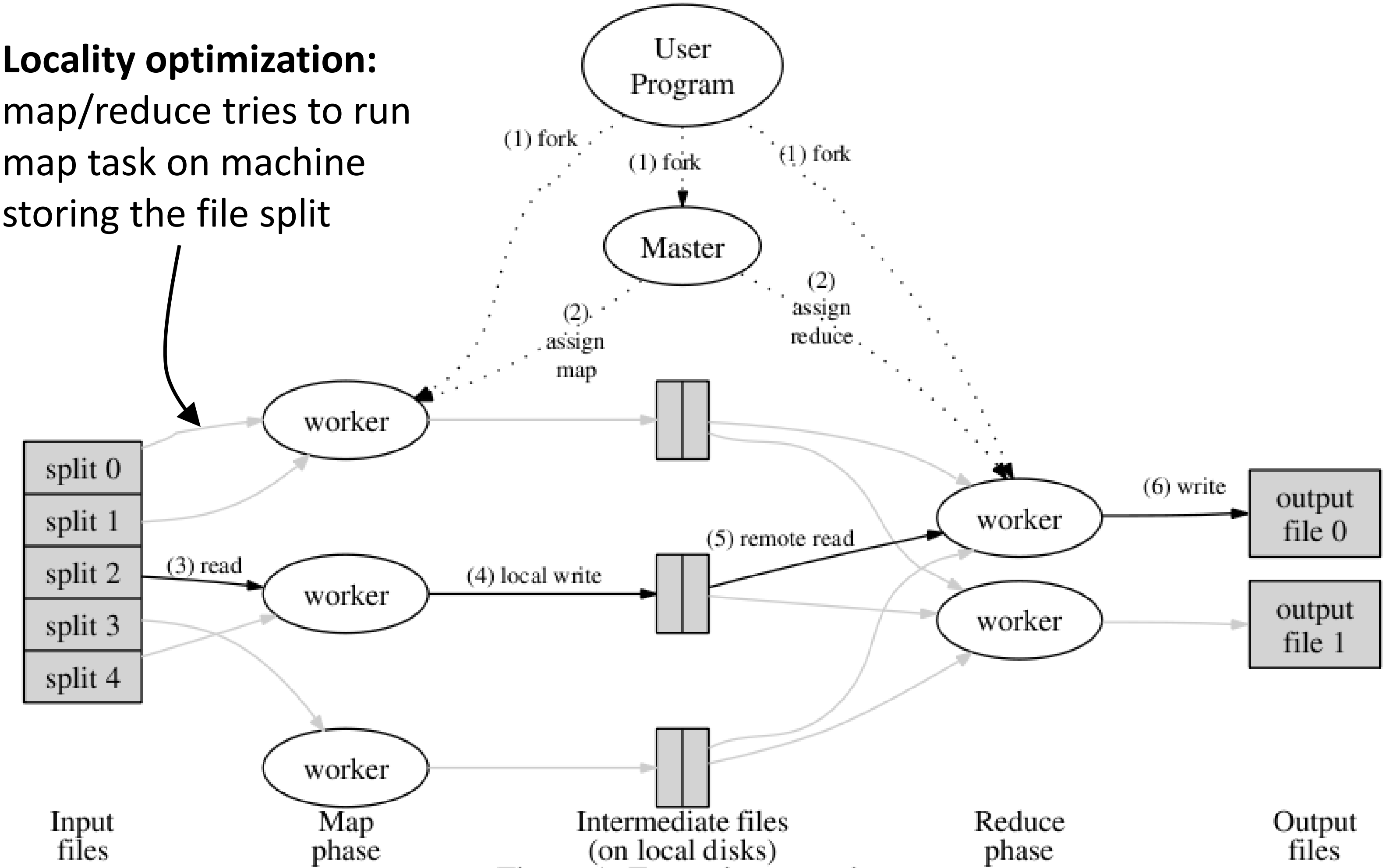


Figure 1: Execution overview

Map-Reduce Implementation

- Map task:
  - Reads a data partition (e.g., GFS chunk)
  - Runs mapper fn on each data item in the partition
  - Writes intermediate file per reduce task on local disk
  - On completion, informs master about its map output files
  - Master informs all reduce tasks about their map output files
- Reduce task:
  - Reads (pulls) data from its map output files
  - After reading all map output files, sorts the data in all the files
  - Runs the reduce fn on each data item

Handling Failures

- Machine failures are common in large systems
  - “One node crashes per day in a 10K node cluster” - Jeff Dean
- Worker failure
  - Master detects worker failure via periodic heartbeats
  - Re-executes map/reduce tasks whose results are not available
    - Assumption: map/reduce tasks are deterministic
- Master failure
  - Single point of failure
  - Master writes periodic checkpoints
  - Another master started from the last checkpointed state
- Google: Lost 200 of 1800 workers but finished fine!

Refinement: Redundant Execution

- Slow workers significantly lengthen completion time
  - Called stragglers
- Caused by many reasons
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Software bugs
- Solution
  - Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"
  - Doesn't cause overhead if stragglers don't exist

Various Advancements

- Master can become bottleneck
  - Split functionality of master
    - Scheduling, monitoring, recovery, etc.
  - Only scheduler is centralized
- I/O on intermediate results is slow
  - Buffer intermediate result in memory
- Other programming models
  - E.g., SQL on distributed systems (HIVE)

Conclusions

- Powerful, simple-to-use distributed programming model
- Scales well since many analytic tasks are embarrassingly parallel
- Ensures that computation produces the same output as running the computation sequentially, even in the presence of failures
- Highly influential
  - Apache Hadoop builds on map-reduce design

Discussion

Q1

- How are data partitions created for map tasks, and for reduce tasks, and why?

Q2

- Why is sorting required on reduce side? What impact does this sorting have on concurrent operation?

Q3

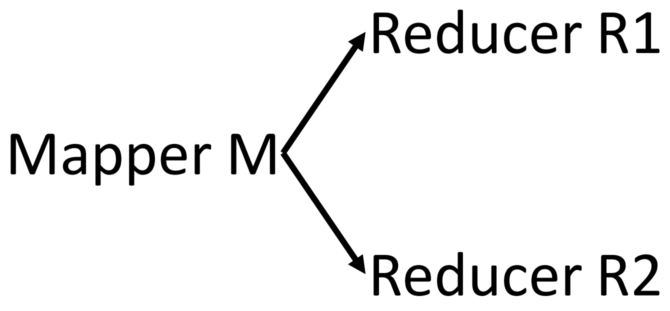
- Why is data stored on disk on map side (and not on the reduce side)?

Q4

- Why is data stored on map side made visible to the reducer only after the mapper ends?

Q5

- Why do the map tasks need to be deterministic? Hint: what might happen if M dies and is restarted?



Mapper M

Reducer R1

Reducer R2