Distributed Systems 1

CUCS Course 4113
https://systems.cs.columbia.edu/ds1-class/

Instructor: Roxana Geambasu

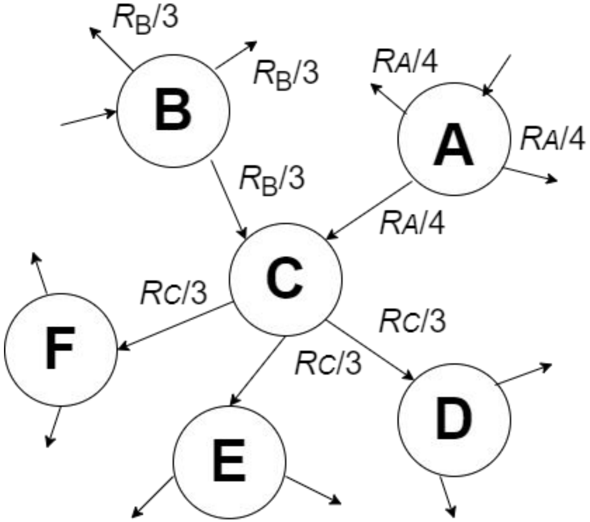
MapReduce

Outline

- Today: MapReduce
  - Analytics programming interface
  - Word count example
  - Chaining
  - Reading/write from/to HDFS
  - Dealing with failures
- In future: Spark
  - Motivation
  - Resilient Distributed Datasets (RDDs)
  - Programming interface
  - Transformations and actions

Large-scale Analytics

- Compute the frequency of words in a corpus of documents.
- Count how many times users have clicked on each of a (large) set of ads.
- PageRank: Compute the "importance" of a web page based on the "importances" of the pages that link to it.
- ….



Option 1: SQL

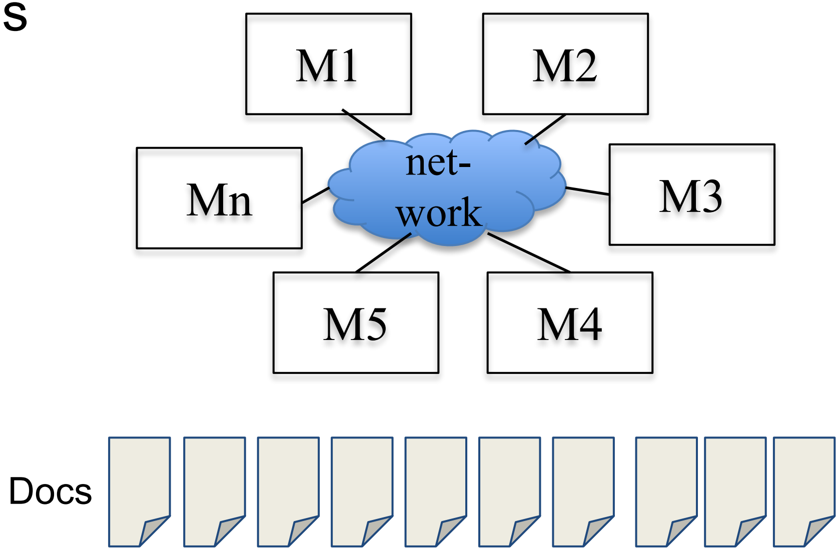
- Before MapReduce, analytics mostly done in SQL, or manually.
- Example: Count word appearances in a corpus of documents.
- With SQL, the rough query might be:

```
SELECT COUNT(*)
 FROM (
        SELECT UNNEST(string_to_array(doc_content, ‘ ’)) as word
        FROM Corpus
 )
 GROUP BY word
```

- Very **expressive**, **convenient to program**, but no one knew **how to scale** SQL execution!

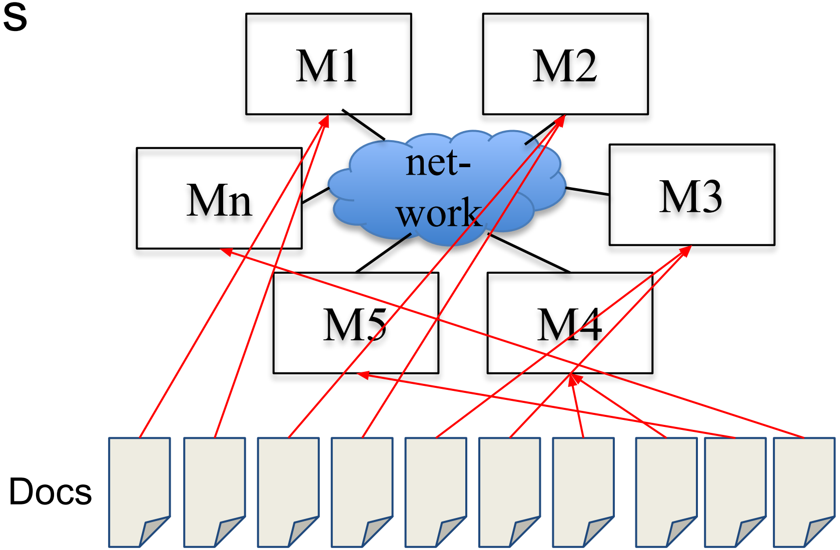
Option 2: Manual

- Example: Count word appearances in a corpus of documents.



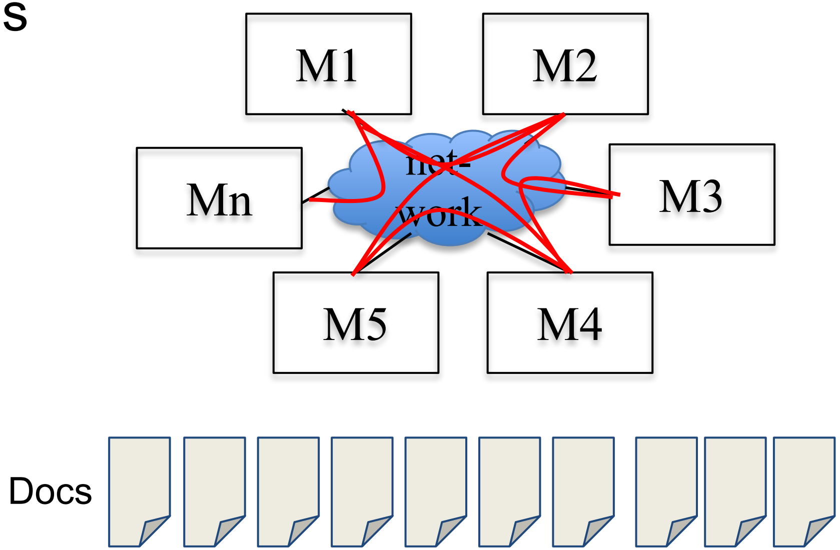
Option 2: Manual

- Example: Count word appearances in a corpus of documents.
- **Phase 1:** Assign documents to different machines/nodes.
  - Each computes a dictionary: {word: local_freq}.



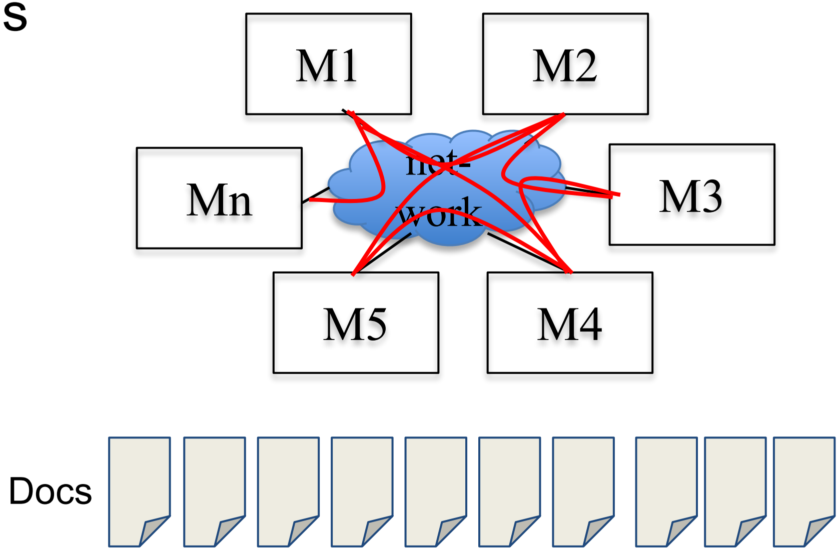
Option 2: Manual

- Example: Count word appearances in a corpus of documents.
- **Phase 1:** Assign documents to different machines/nodes.
  - Each computes a dictionary: {word: local_freq}.
- **Phase 2:** Nodes exchange dictionaries (how?) to aggregate local_freq's.



Option 2: Manual

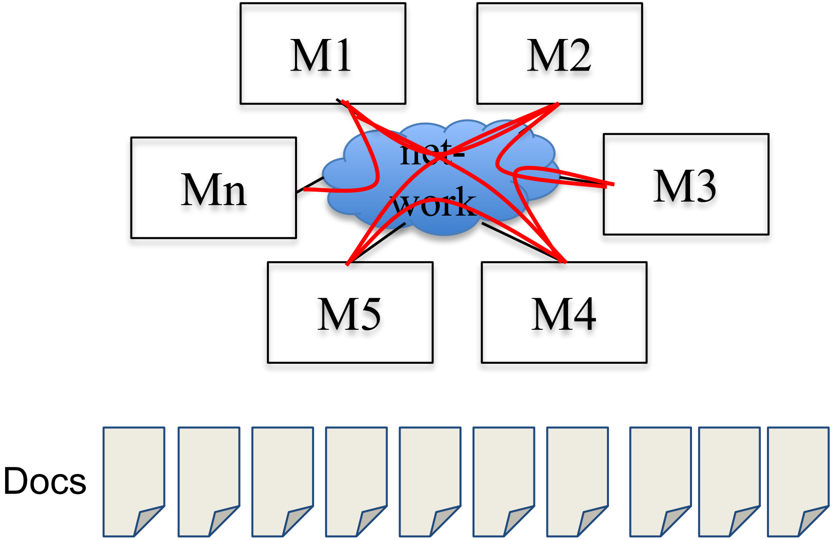
- Example: Count word appearances in a corpus of documents.
- **Phase 1:** Assign documents to different machines/nodes.
  - Each computes a dictionary: {word: local_freq}.
- **Phase 2:** Nodes exchange dictionaries (how?) to aggregate local_freq's.
  - **But how to make this scale??**



Option 2: Manual (cont'ed)

- **Phase 2, Option a:** Send all {word: local_freq} dictionaries to one node, who aggregates.
  - But what if it's too much data for one node?

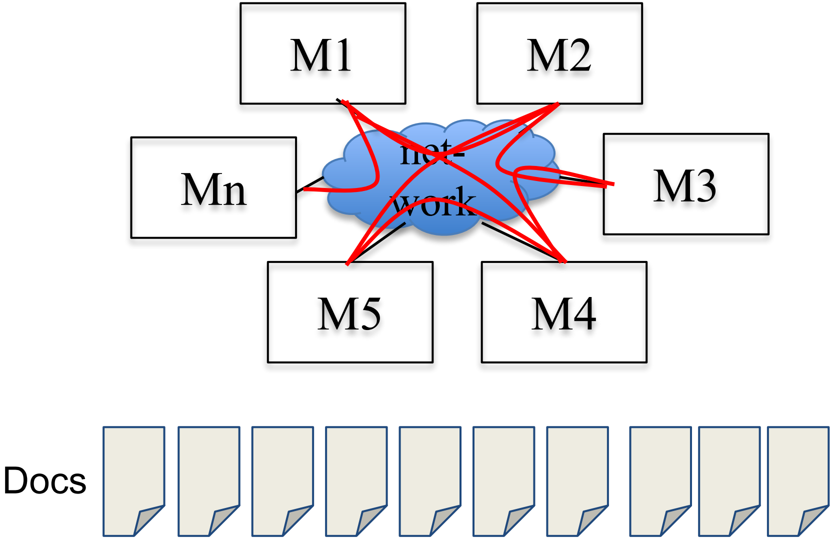
- **Phase 2, Option b:** Each node sends (word, local_freq) to a designated node, e.g., node with ID hash(word) % N.



Option 2: Manual (cont'ed)

- **Phase 2, Option a:** Send all {word: local_freq} dictionaries to one node, who aggregates.
  - But what if it's too much data for one node?

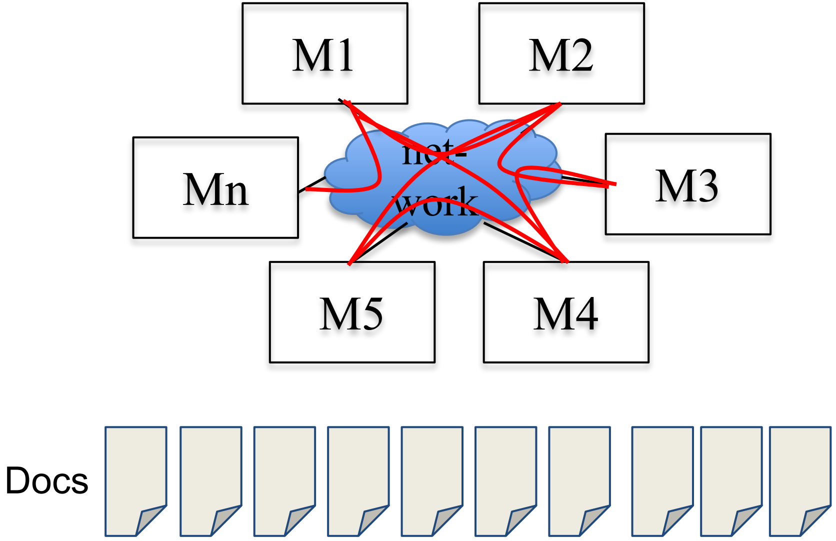
- **Phase 2, Option b:** Each node sends (word, local_freq) to a designated node, e.g., node with ID hash(word) % N.



We've roughly discovered an app-specific version of MapReduce!

Option 2: Challenges

- How to generalize to other applications?
- How to deal with failures?
- How to deal with slow nodes?
- How to deal with load balancing (some docs are very large, others small)?
- How to deal with skew (some words are very frequent, so nodes designated to aggregate them will be pounded)?
- …



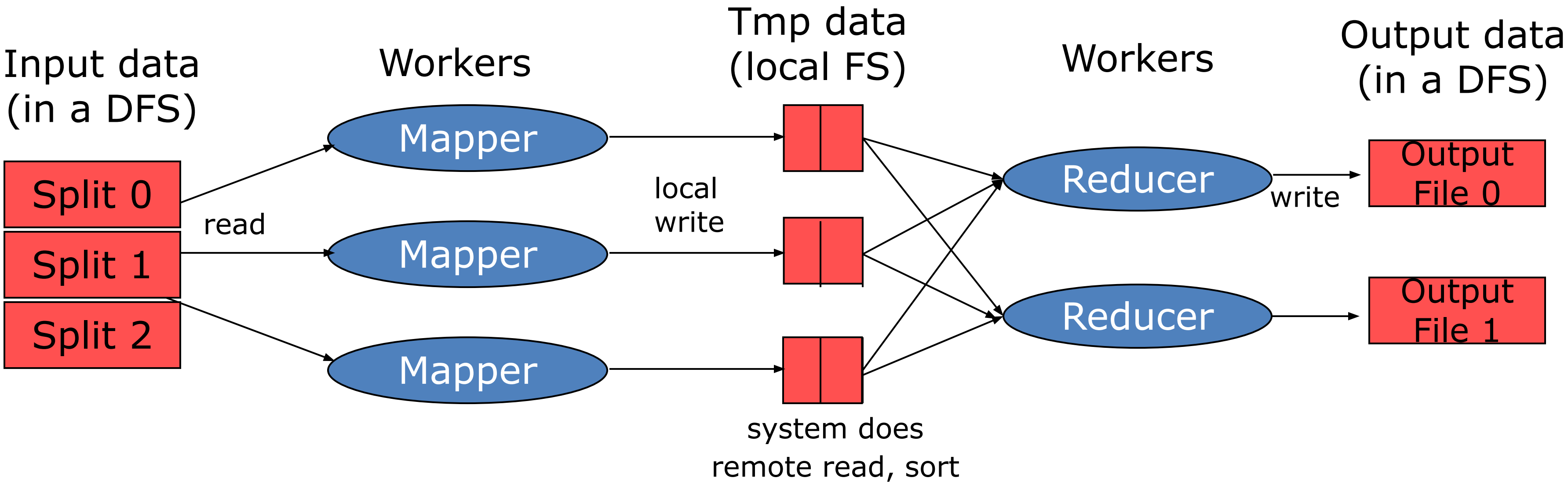
MapReduce

- **Parallelizable programming model**:
  - Applies to a broad class of analytics applications.
  - Isn’t as expressive as SQL but it is easier to scale.
  - Consists of three phases, each intrinsically parallelizable:
    - **Map**: processes input elements independently to emit relevant (key, value) pairs from each.
    - Transparently, the runtime system groups all the values for each key together: (key, [list of values]).
    - **Reduce**: aggregates all the values for each key to emit a global value for each key.
- **Scalable, efficient, fault tolerant runtime system**.
  - Addresses preceding challenges (and more).
- Technical paper: please read it!

How it works

- Input: a collection of elements of **(key, value) pair type**.
- Programmer defines two functions:
  - **Map**(key, value) → 0, 1, or more (key’, value’) pairs
  - **Reduce**(key, value-list) → output
- Execution:
  - Apply **Map** to each input key-value pair, in parallel for different keys.
  - Sort emitted (key’, value’) pairs to produce **(key’ value’-list) pairs.**
  - Apply **Reduce** to each (**key’, value’-list)** pair, in parallel for different keys.
- Output is the union of all Reduce invocations’ outputs.

Workflow



Map phase:
extract something you care
about from each record

Reduce phase:
aggregate

Example: Word count

- We have a directory, which contains many documents.
- The documents contain words separated by whitespace and punctuation.
- Goal: Count the number of times each distinct word appears across the files in the directory.

Word count with MapReduce

```
Map(key, value):    // key: document ID; value: document content
  FOR (each word w IN value)
      emit(w, 1);


Reduce(key, value-list):   // key: a word; value-list: a list of integers
        result = 0;
        FOR (each integer v on value-list)
                result += v;
        emit(key, result);
```

Word count with MapReduce

```
Map(key, value):    // key: document ID; value: document content
  FOR (each word w IN value)
    emit(w, 1);


Reduce(key, value-list):   // key: a word; value-list: a list of integers
    result = 0;
    FOR (each integer v on value-list)
        result += v;
    emit(key, result);
```

Expect to be all 1’s, but “combiners” allow local summing of integers with the same key before passing to reducers.

Map Phase

- Mapper is given key: document ID; value: document content, say:
  `(D1,"The teacher went to the store. The store was closed;   the store opens in the morning. The store opens at 9am.")`

- It will emit the following pairs:
  ```
  <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store,1>
  <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store,1>
  <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1>
  <opens, 1> <at, 1> <9am, 1>
  ```

- Normally, there will be many documents, hence many Mappers that emit such pairs in parallel, but for simplicity, let's say that these are all the pairs emitted from the Map phase.

Intermediary Phase

- Transparently, the runtime **sorts** emitted (key, value) pairs by key:

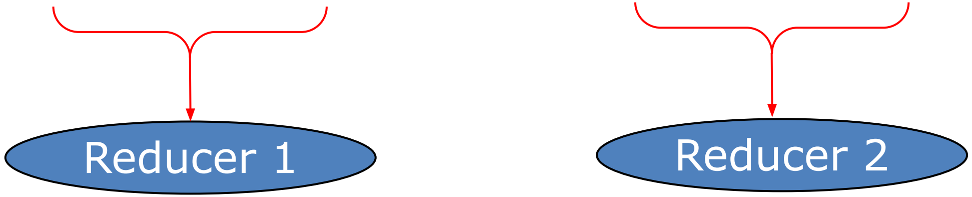
```
<9am, 1>
<at, 1>
<closed, 1>
<in, 1>
<morning, 1>
<opens, 1>
<opens, 1>
<store, 1>
<store,1>
<store, 1>
<store,1>
<teacher, 1>
<the, 1>
<the, 1>
<the, 1>
<the, 1>
<the 1>
<to, 1>
<went, 1>
<was, 1>
<The, 1>
```

Intermediary Phase

- Transparently, the runtime **sorts** emitted (key, value) pairs by key:

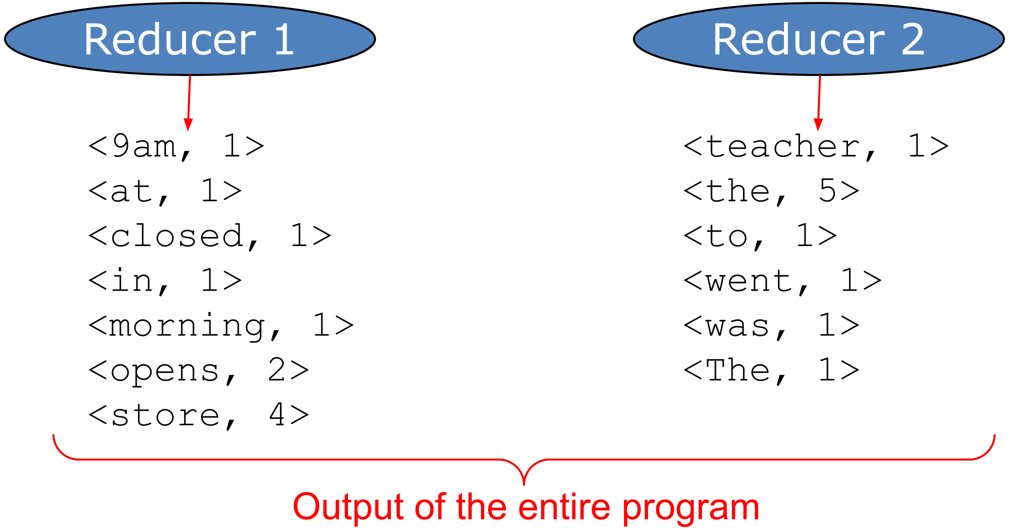
```
<9am, 1>
<at, 1>
<closed, 1>
<in, 1>
<morning, 1>
<opens, 1>
<opens, 1>
<store, 1>
<store,1>
<store, 1>
<store,1>
```

```
<teacher, 1>
<the, 1>
<the, 1>
<the, 1>
<the, 1>
<the 1>
<to, 1>
<went, 1>
<was, 1>
<The, 1>
```



Reduce Phase

- For each unique key emitted from the Map Phase, function Reduce(key, value-list) is invoked on Reducer 1 or Reducer 2.
- Across their invocations, these Reducers will emit:



Reducer 1

```
<9am, 1>
<at, 1>
<closed, 1>
<in, 1>
<morning, 1>
<opens, 2>
<store, 4>
```

Reducer 2

```
<teacher, 1>
<the, 5>
<to, 1>
<went, 1>
<was, 1>
<The, 1>
```

Output of the entire program

Chaining MapReduce

- The programming model seems pretty restrictive.
- But quite a few analytics applications can be written with it, especially with a technique called **chaining**.
- If the output of reducers is (key, value) pairs, then their output can be passed onto other Map/Reduce processes.
- This chaining can support a variety of analytics (though certainly not all types of analytics, e.g., no ML b/c no loops).

Example: Word Frequency

- Suppose instead of word count, we wanted to compute word frequency: the probability that a word would appear in a document.
- This means computing the fraction of times a word appears, out of the total number of words in the corpus.

Solution: Chain two MapReduce’s

- **First Map/Reduce:** Word Count (like before)
  - Map: process documents and output <word, 1> pairs.
  - Multiple Reducers: emit <word, word_count> for each word.

- **Second MapReduce:**
  - Map: process <word, word_count> and output (1, (word, word_count)).
  - **1 Reducer**: perform two passes:
    - In first pass, sum up all word_count’s to calculate overall_count.
    - In second pass calculate fractions and emit multiple <word, word_count/overall_count>.

- Scalability is not too bad, as first stage’s output is a rather small dictionary (maximum # of English words with an integer for each).

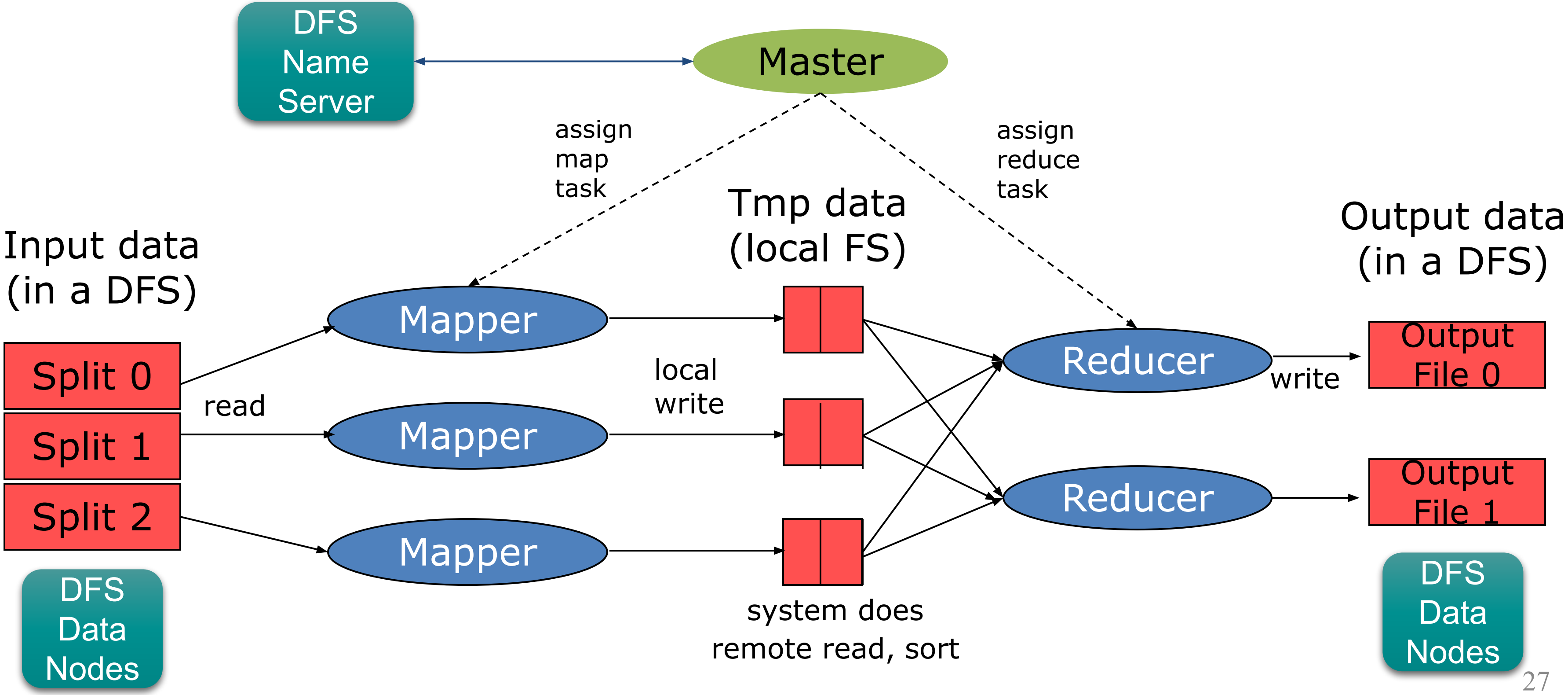
Recall: Option 2’s Challenges

- How to generalize to other applications?
  - See original MapReduce paper for more examples.

- How to deal with **failures**?
- How to deal with **slow nodes**?
- How to deal with **load balancing**?
- How to deal with **skew**?

- The MapReduce runtime tries to hide these challenges as much as possible. We’ll talk about a few of the **performance** and **fault tolerance** challenges here.

Architecture



Data locality for performance

- Master scheduling policy:
  - Asks DFS for locations of replicas of input file blocks.
  - Map tasks are scheduled so DFS input block replica are on the same machine or on the same rack.
- Effect: Thousands of machines read input at local speed.
  - Don’t need to transfer input data all over the cluster over the network: eliminate network bottleneck!

Heartbeats & replication for fault tolerance

- Failures are the norm in data centers.
- **Worker failure:**
  - Master detects if workers failed by periodically pinging them (this is called a “heartbeat”).
  - Re-execute in-progress map/reduce tasks.
- **Master failure:**
  - Initially, was single point of failure; Resume from Execution Log. Subsequent versions used replication and consensus.
- Effect: From Google’s paper: once, a Map/Reduce job lost 1600 of 1800 machines, but it still finished fine.

Redundant execution for performance

- Slow workers or **stragglers** significantly lengthen completion time.
- Slowest worker can determine the total latency!
  - Other jobs consuming resources on machine.
  - Bad disks with errors transfer data very slowly.
  - This is why many systems measure **99th percentile latency**.
- **Solution**: spawn backup copies of tasks.
  - Whichever one finishes first "wins.”
  - I.e., treat slow executions as failed executions!

Take-Aways

- MapReduce is a **programming model** and **runtime** for scalable, fault tolerant, and efficient analytics.
  - Well, some types of analytics, it's not very expressive.
- It hides common at-scale challenges under convenient abstractions.
  - Programmers need only implement Map and Reduce and the system takes care of running those at scale on lots of data.
  - Failures raise semantic/performance challenges for MapReduce, which it typically handles through redundancy.
- There exist open-source implementations, including Hadoop, Flink.
- Spark, a popular data analytics framework, also supports MapReduce but also a wider range of programming models.

Acknowledgement

- Slides prepared based on Asaf Cidon’s 2020 DS-1 invited lecture.