Map Reduce

CSCI 333

Williams College

This Video

Map Reduce

- The problem
  - Examples
- The model
- Fault tolerance
- The straggler problem
- Moving data vs. moving computation

When Reading a Paper

Look at authors

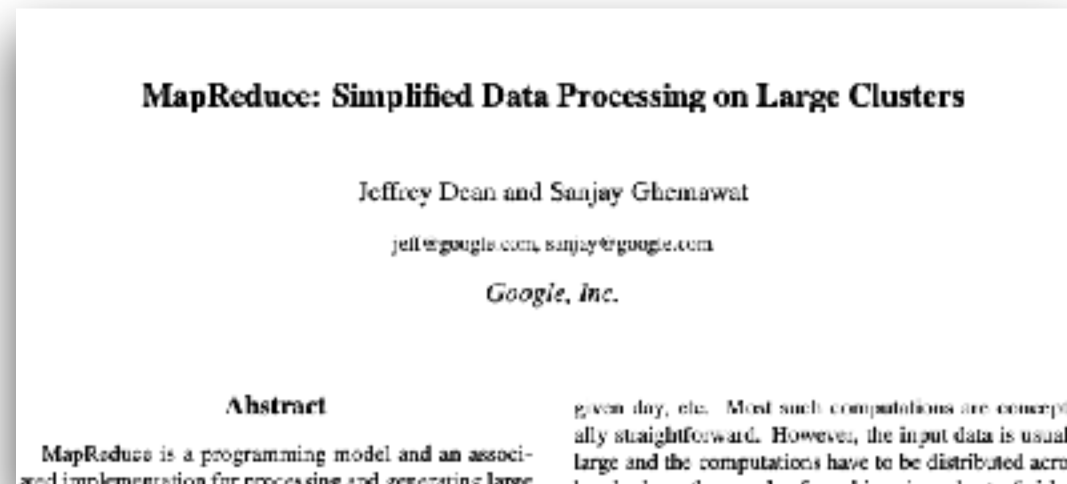
Look at institution

Look at past/future research

Look at publication venue

These things will give you insight into the

- motivations
- perspectives
- agendas
- resources



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associ-
ated implementation for processing and generating large

given day, etc. Most such computations are conceptu
ally straightforward. However, the input data is usuall
large and the computations have to be distributed acros

Think: Are there things that they are promoting? Hiding? Building towards?

Why?

Thought Experiment

What is it that Google actually does?

- Sells ads

How do they sell ads?

- NLP on your emails, harvesting Android GPS data, etc. (in general by creeping on our personal lives)

But what does the average person mean when they use “Google” as a verb?

- Search!

Reverse Indexes

World-wide-web is a graph of webpages

- URI -> content (set of words)

Reverse index does the opposite

- word -> set of URIs

We can compute over an inverted index to rank pages.

How would you implement a reverse index?

The Problem

Hundreds of special-purpose computations per day that

- Consume data distributed over thousands of machines
- Can be parallelized, and must be in order to finish in a reasonable timeframe

Challenges that each computation must solve:

- Parallelization
- Fault tolerance
- Data distribution
- Load balancing

Want one computation model that can use to abstract away these concerns

The Model

Map Reduce uses a functional model

- User-supplied **map** function
  - {*key-value pair*} -> {**set** of *key-value pairs*}
- User-supplied **reduce** function
  - {**set** of all *key-value pairs* with a given key} -> {*key-value pair*}

- The system applies the **map** function to each key-value pair, yielding a set of intermediate key-value pairs
- The system then gathers all intermediate key-value pairs, and for each unique key, calls **reduce** on the set of key-value pairs with that key

Example: Word Frequency

Pseudo code (section 2.1):

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
     EmitIntermediate(w, “1”);

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
     result += ParseInt(v);

  Emit(AsString(result));
```

Emits each word plus an associated “**count**” (1 here; duplicates possible)

Aggregates all counts for individual words and sums the entries.

Design

Input data is distributed across multiple systems

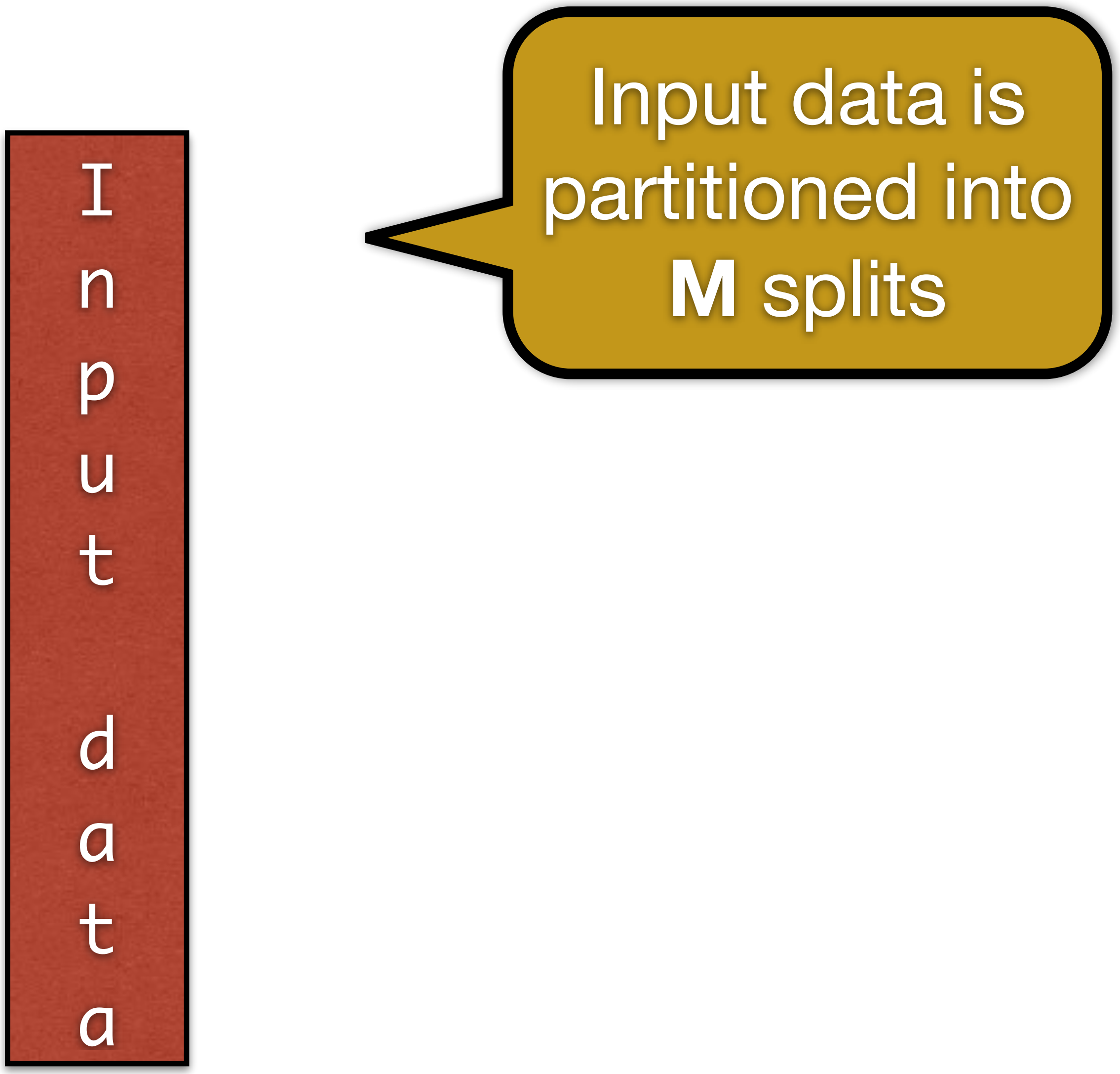
- Input data is divided into **M** (evenly sized) splits
- System schedules a mapper to run on each of the **M** splits
  - No guarantees how evenly target contents are distributed among splits

Intermediate (i.e., pre-reduced) data is distributed across multiple systems

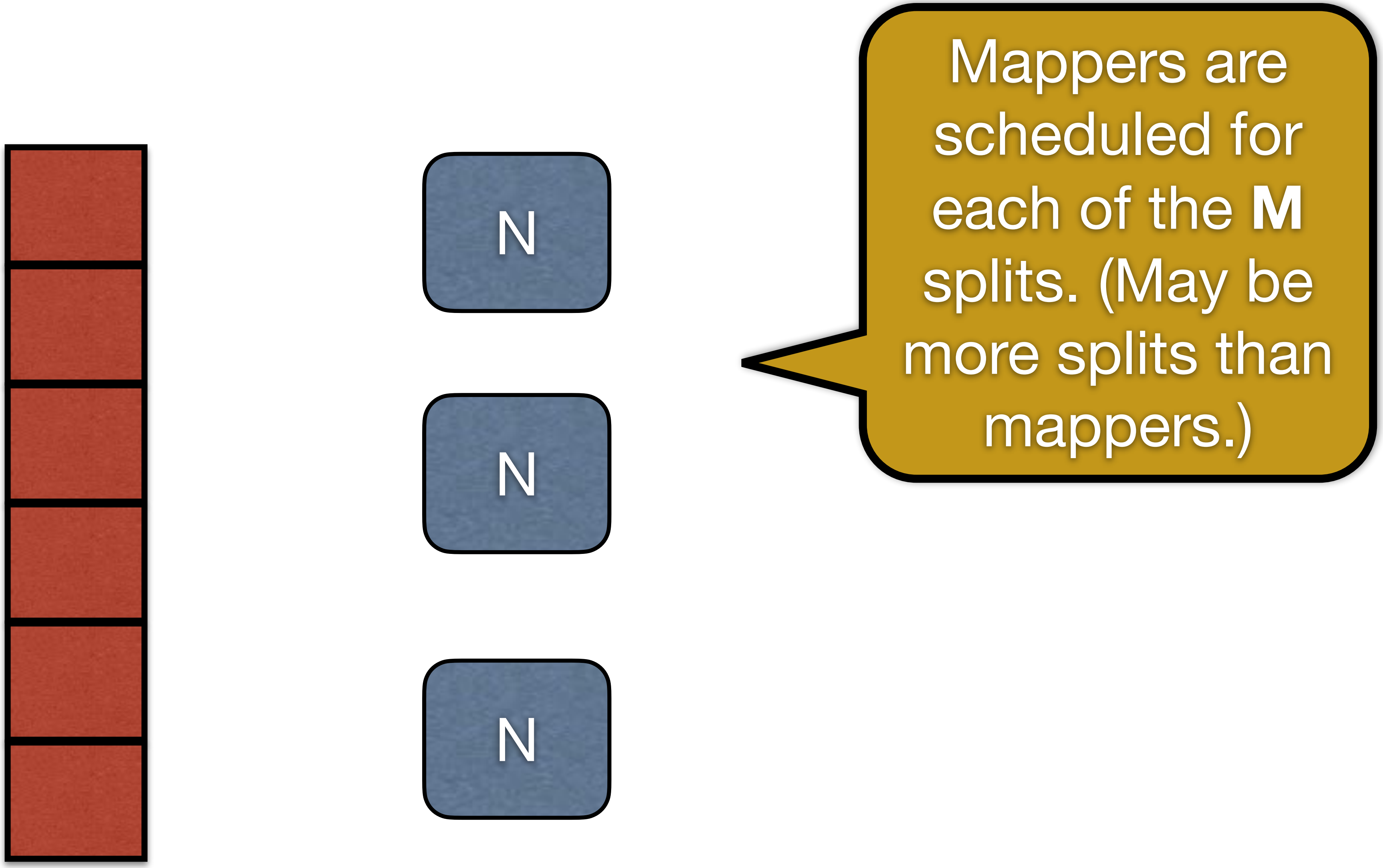
- Users provide a "partitioning" function (e.g., `hash(key) mod R`) that is used to distribute the mapper outputs
- System schedules a reducer for each of the **R** pieces of the intermediate outputs

Result of computation is located in R output files

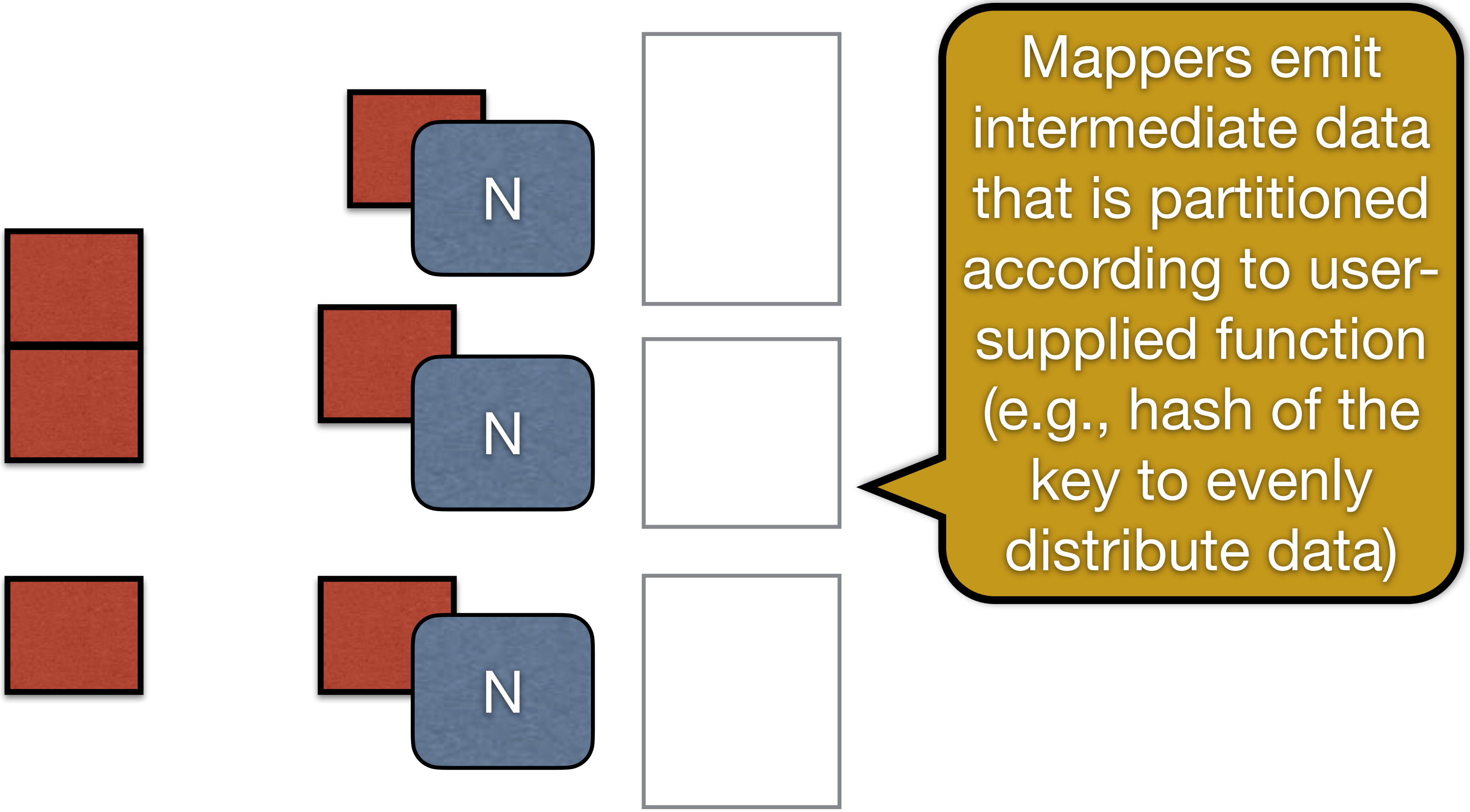
Map Reduce



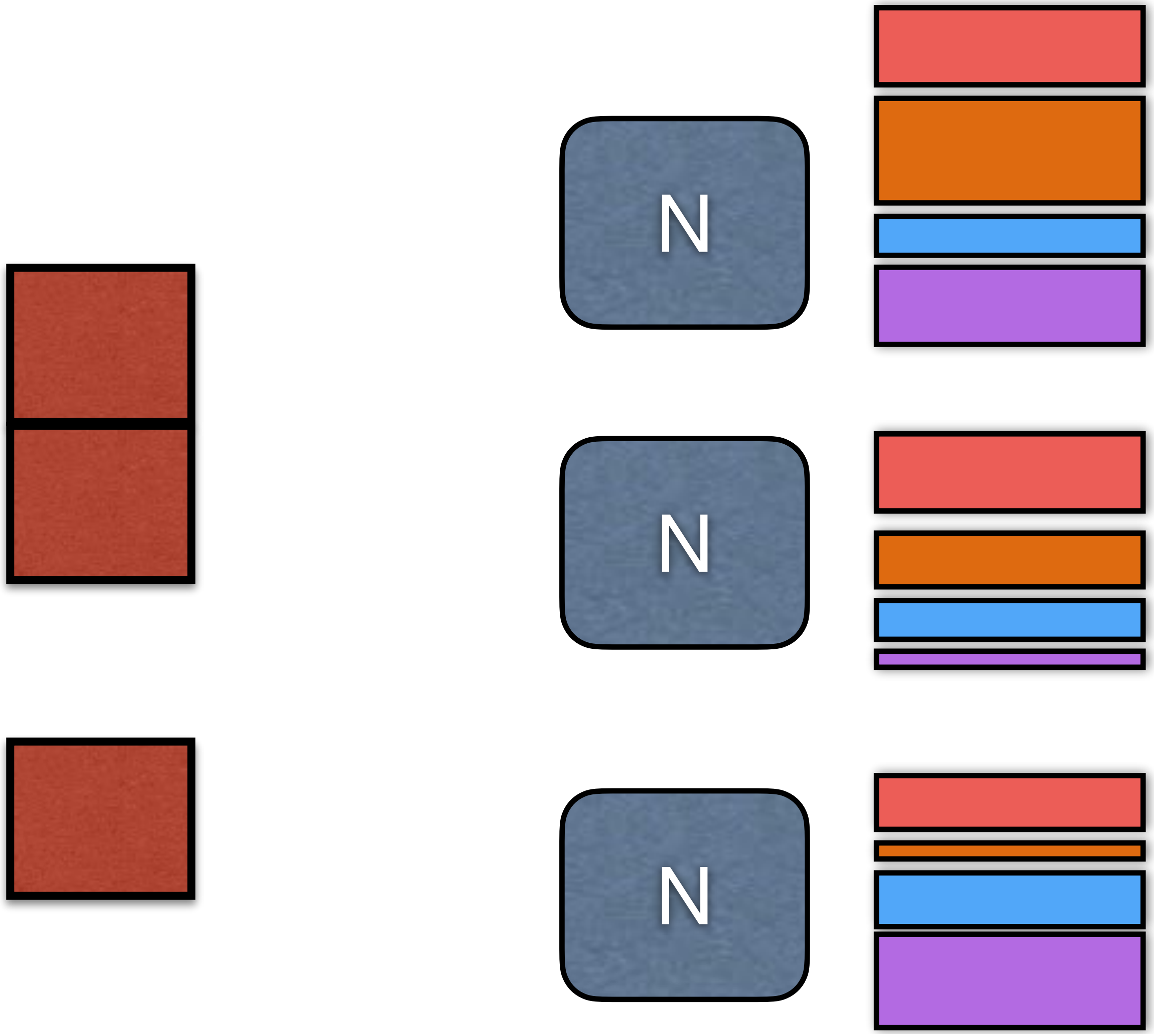
Map Reduce


N
N
N
Mappers are scheduled for each of the **M** splits. (May be more splits than mappers.)

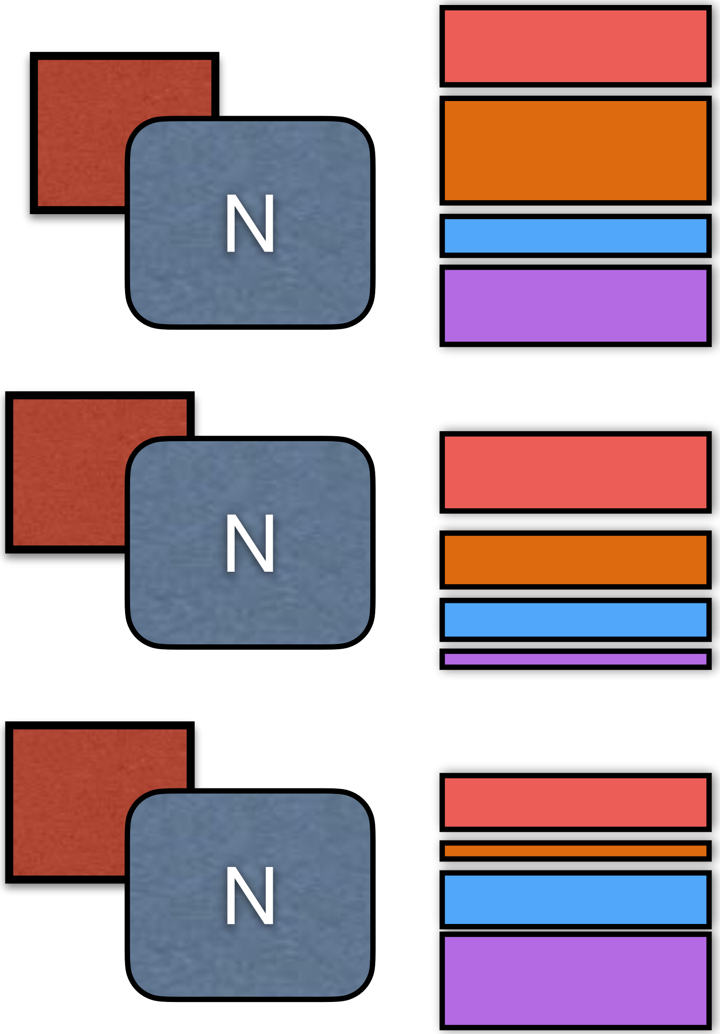
Map Reduce


N
N
N
Mappers emit intermediate data that is partitioned according to user-supplied function (e.g., hash of the key to evenly distribute data)

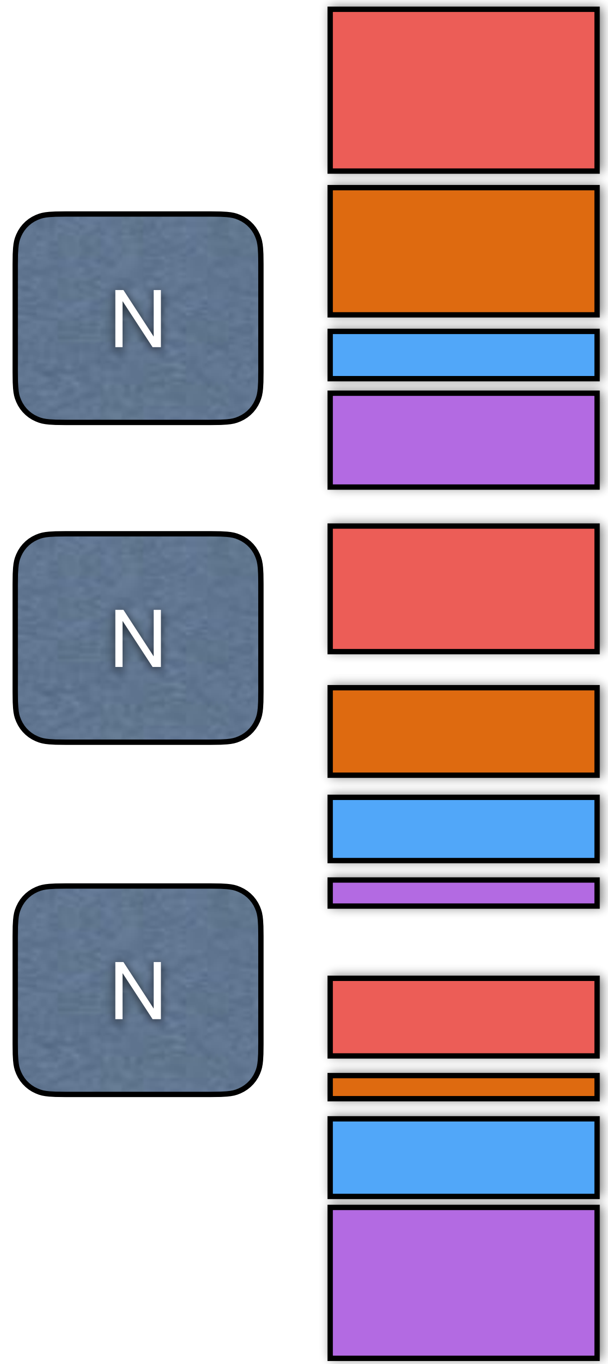
Map Reduce



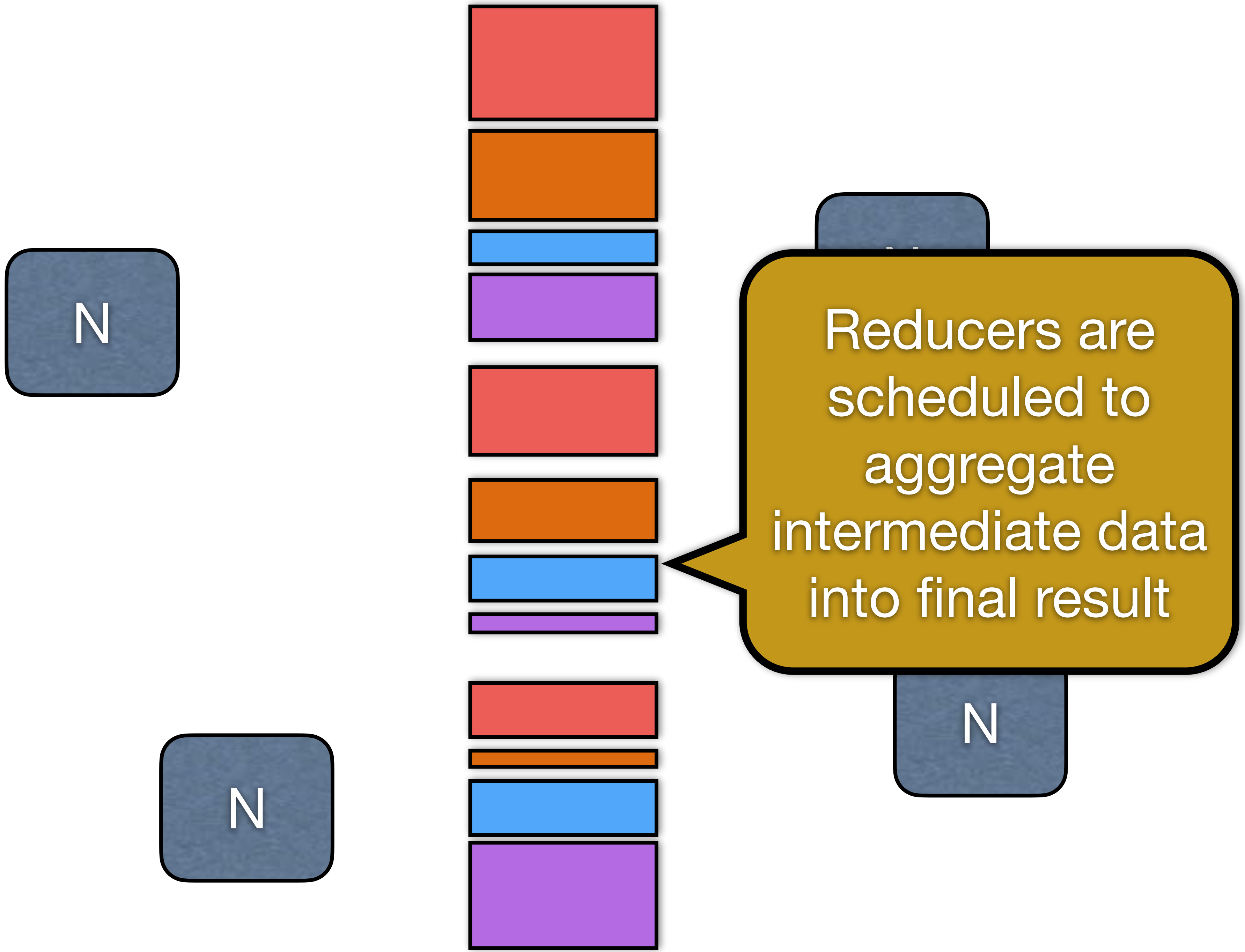
Map Reduce



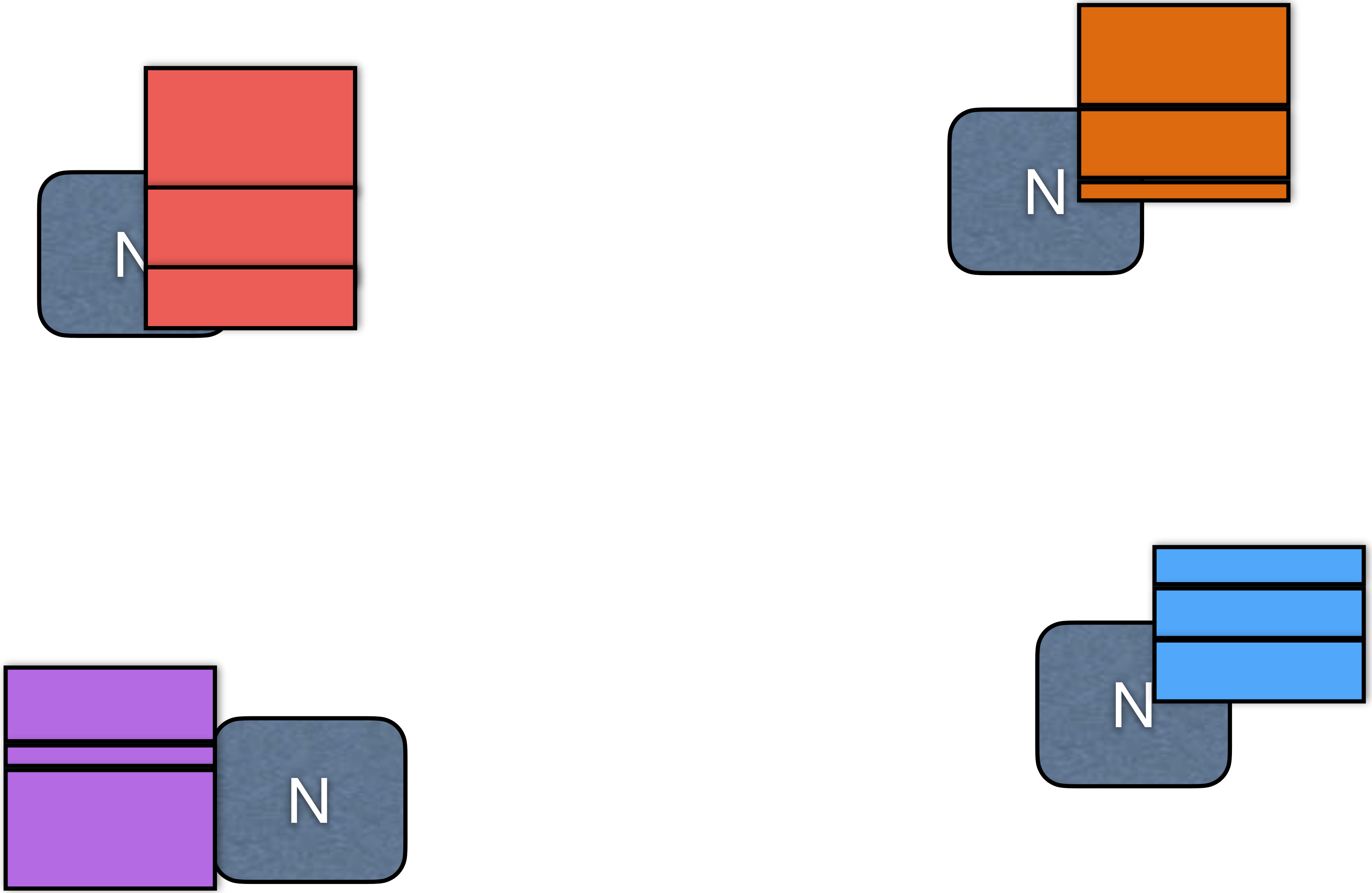
Map Reduce



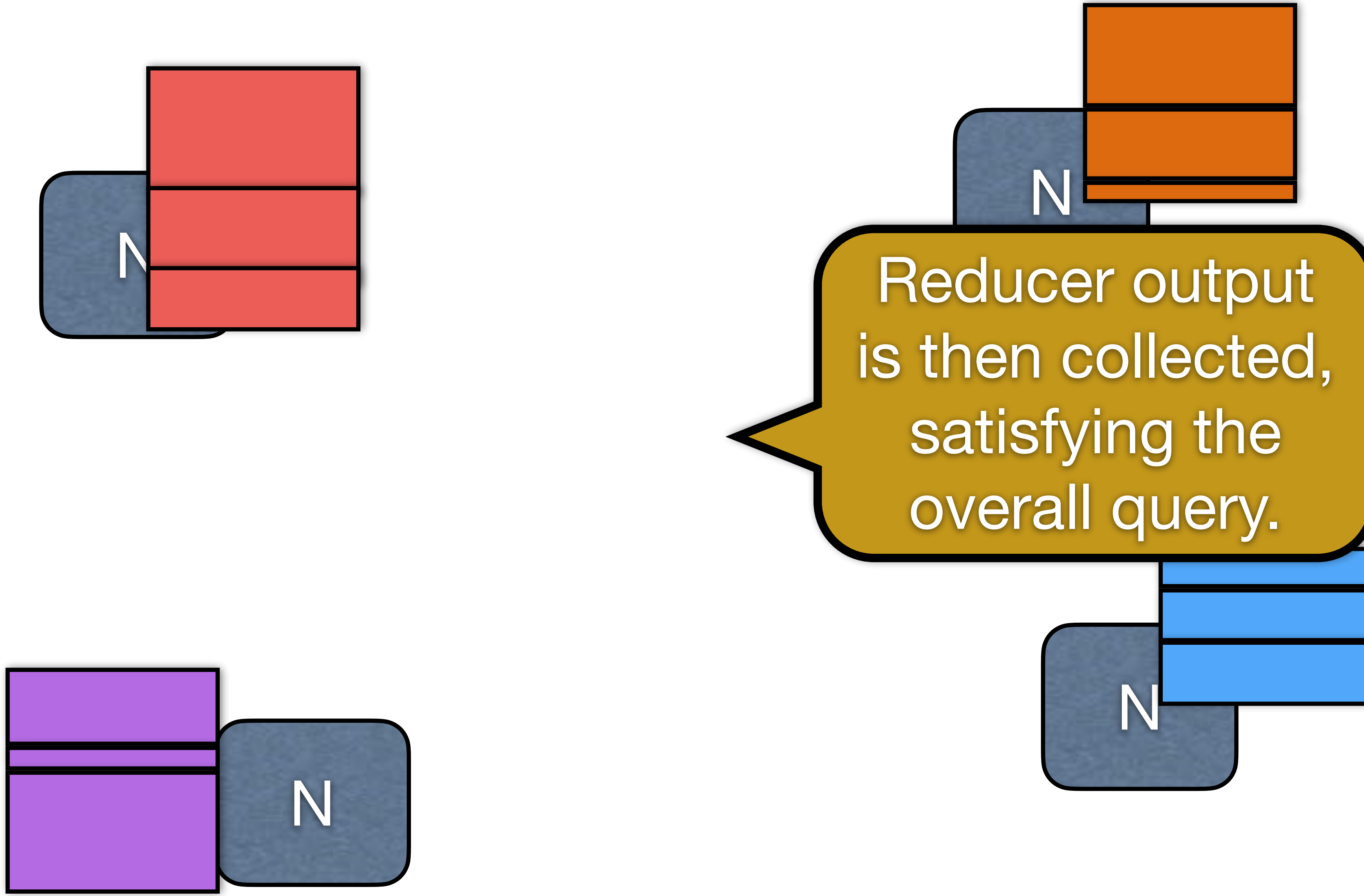
Map Reduce


N
N
N
N
Reducers are scheduled to aggregate intermediate data into final result

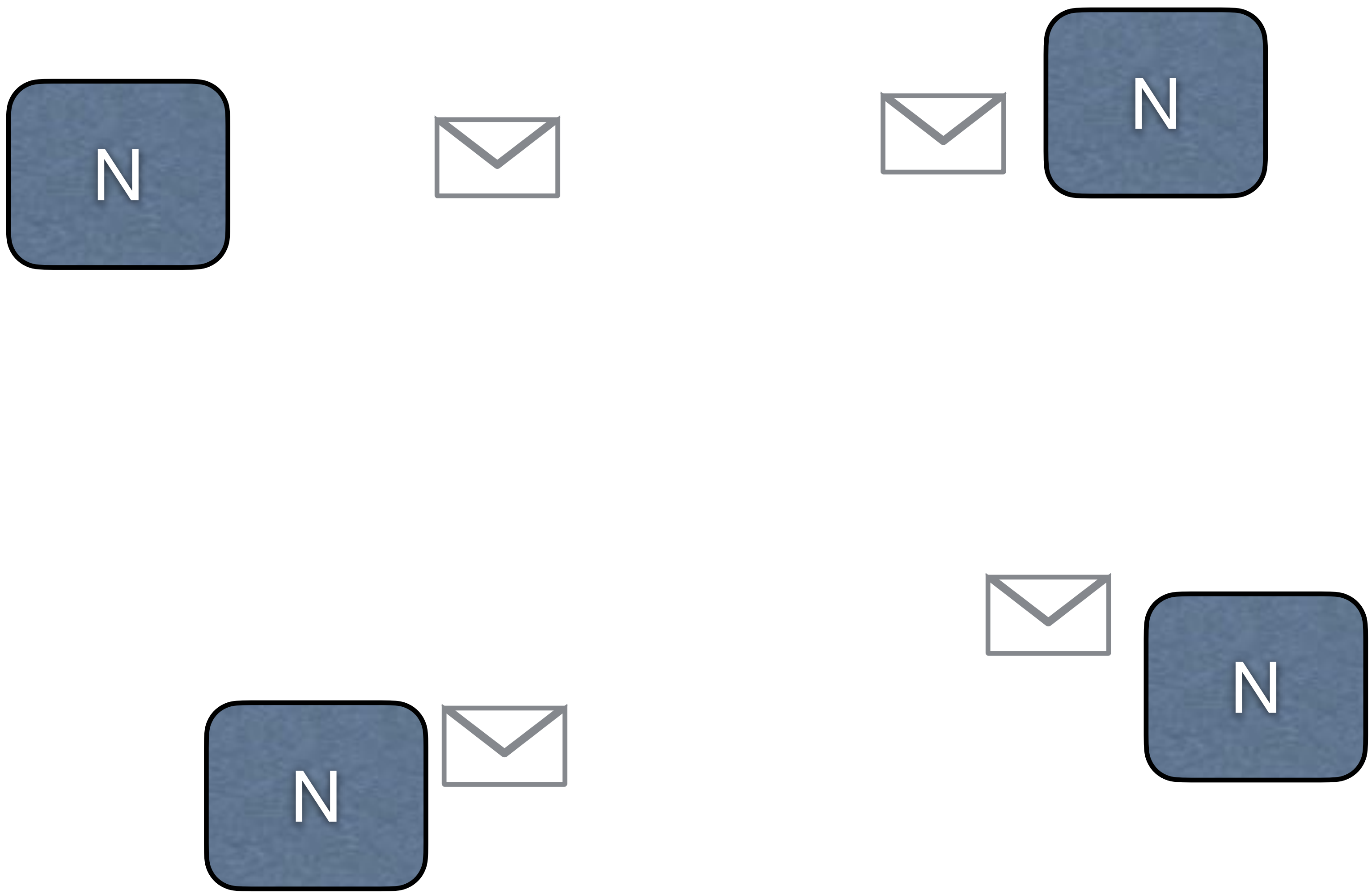
Map Reduce



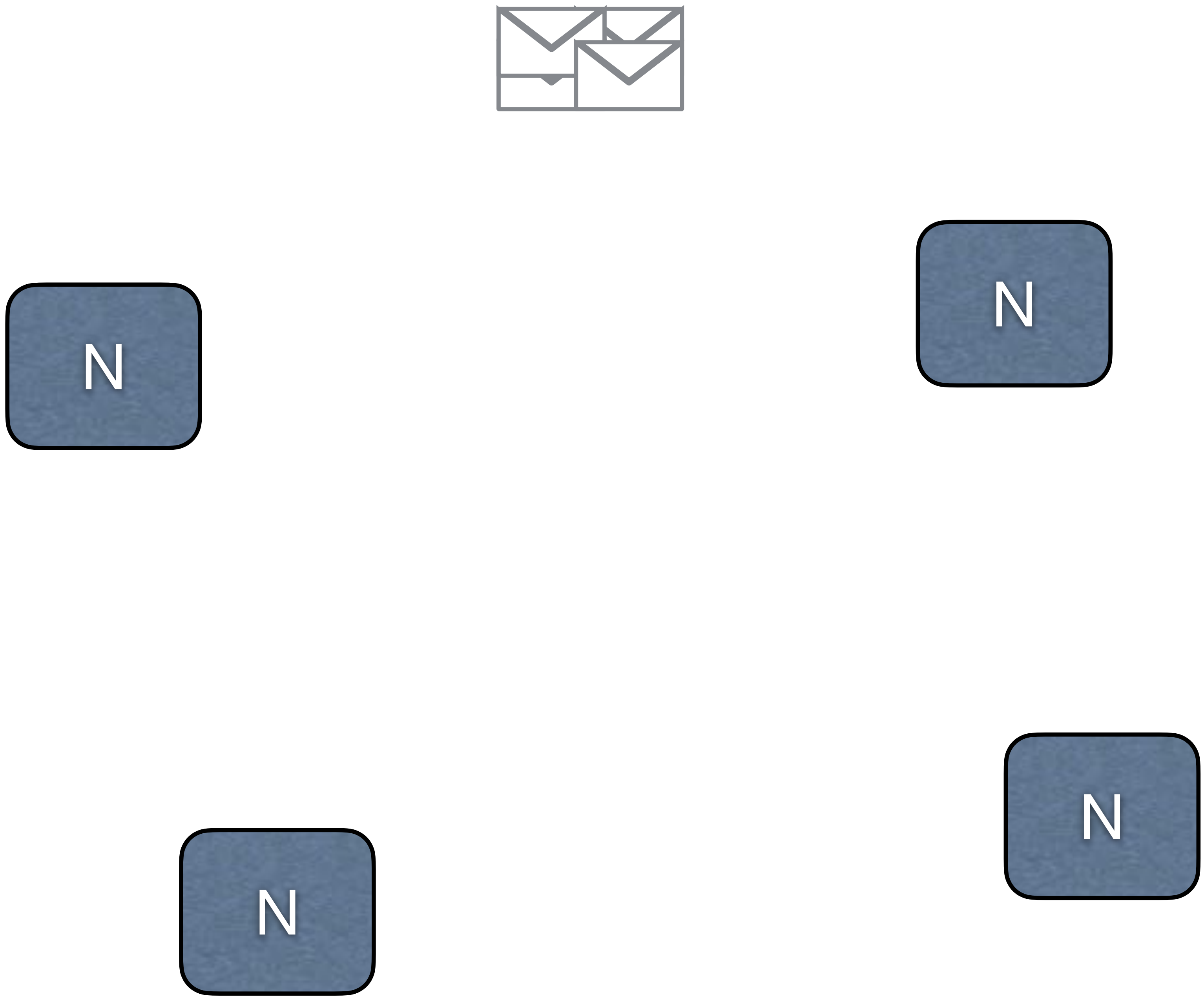
Map Reduce


N
N
Reducer output is then collected, satisfying the overall query.
N
N

Map Reduce



Map Reduce



Other Considerations

Fault Tolerance

- Functional model makes this easy:
  - If a failure occurs, schedule (sub)task again on another node!
  - Caveat: requires deterministic functions, otherwise may get different results

The "Straggler" problem

- What if you have a few slow machines?
  - When near end of the run, reschedule all remaining tasks
  - Use first version of task that returns

Tradeoff: Moving data vs. moving computation

- It is expensive to copy large amounts of data around
  - The MapReduce Scheduler tries as hard as possible to locate mappers/ reducers where the data lives, avoiding data copies (if the underlying system uses replication, then there is more flexibility in scheduling)

Thoughts

MapReduce is a programming model, not necessarily a storage system

- But it relies on the storage system and builds on many of the themes we've discussed in this course
  - Locality matters
    - Moving the computation to the data, partitioning intermediate outputs, etc.
  - Abstraction and layering let us build cohesive and easy-to-reason-about systems
    - drop/replace components without altering surrounding stack

Whether or not you work in "storage", understanding storage system designs and tradeoffs will help you build better systems.