CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 23: Overview of MapReduce

Credit Phil Levis, Chris Gregg, and Nick Troccoli for these slides.

- We're in the home stretch and have studied four broad areas of systems, three of which have expanded our understanding of parallelism and out ability to harness the computation power of an unbounded number of CPUs.

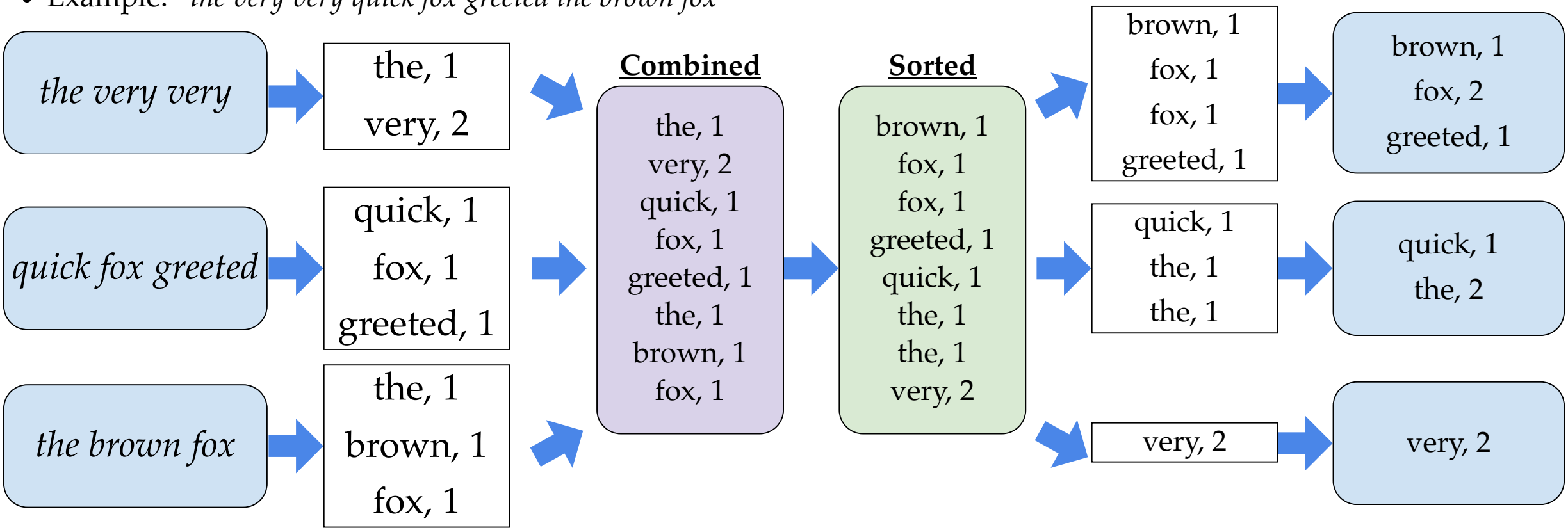
- So, how can we parallelize data processing across many, many machines?
  - There are many ways to do so, but one of the most common frameworks for parallel processing is **MapReduce**, which given our prior work in CS110 is easily described and even implemented.

- Today, we will:
  - Leverage our understanding of networking and concurrency to study MapReduce.
  - Learn about MapReduce's programming model and how it parallelizes operations.
  - Understand how to write a program that can be run using MapReduce.

- An ***optional*** Assignment 7 is your chance to implement your very own MapReduce framework.
  - Assignment 7 will go out this Thursday and can be submitted as late as December 10th at 11:59pm.
  - No lateness on this one, and it will only be graded on functionality.
  - If you complete Assignment 7 and do well on it, I'll replace your lowest assignment or assessment score with your Assignment 7 functionality score.

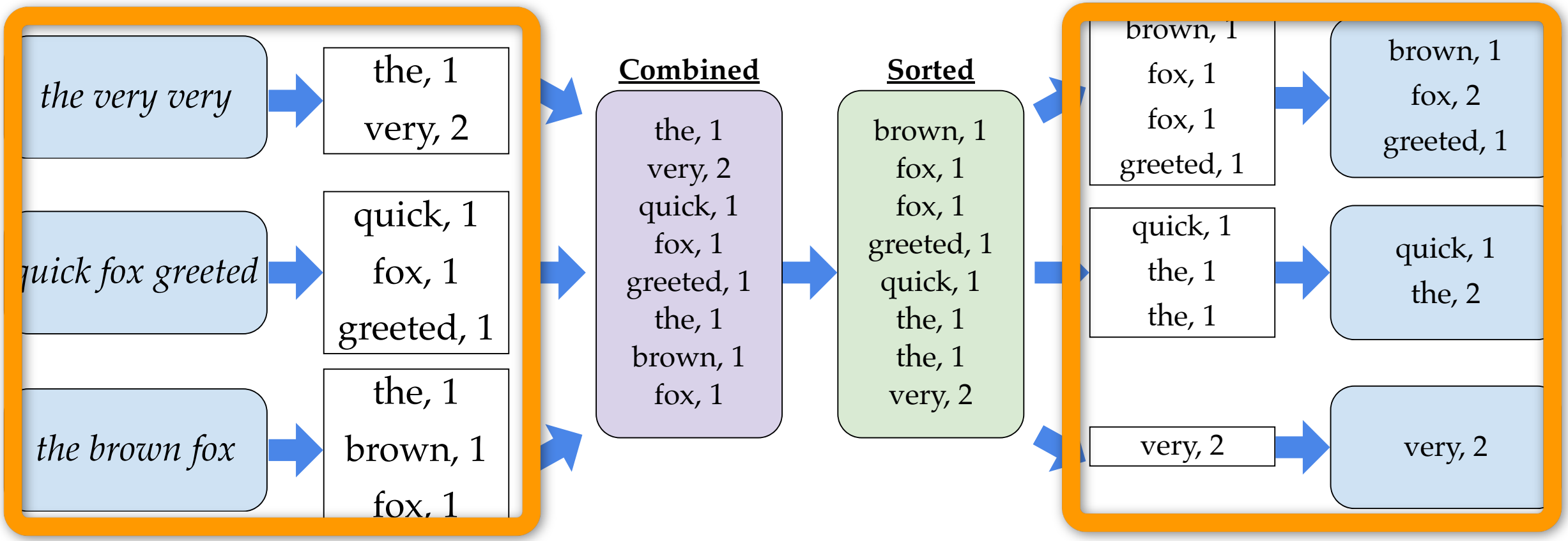
Lecture 23: Overview of MapReduce

- Representative Problem: we want to parse a document and count how many times each word appears.
- Possible Approach: program that reads document and builds a word -> frequency map.
- How can we parallelize this?
  - Idea: Split document into chunks, count words in each chunk concurrently
  - Problem: What if a word appears in multiple chunks?
  - Better Idea: Combine all the output, sort, split into chunks, combine counts in each one (in parallel).
- Example: *"the very very quick fox greeted the brown fox"*



Lecture 23: Overview of MapReduce

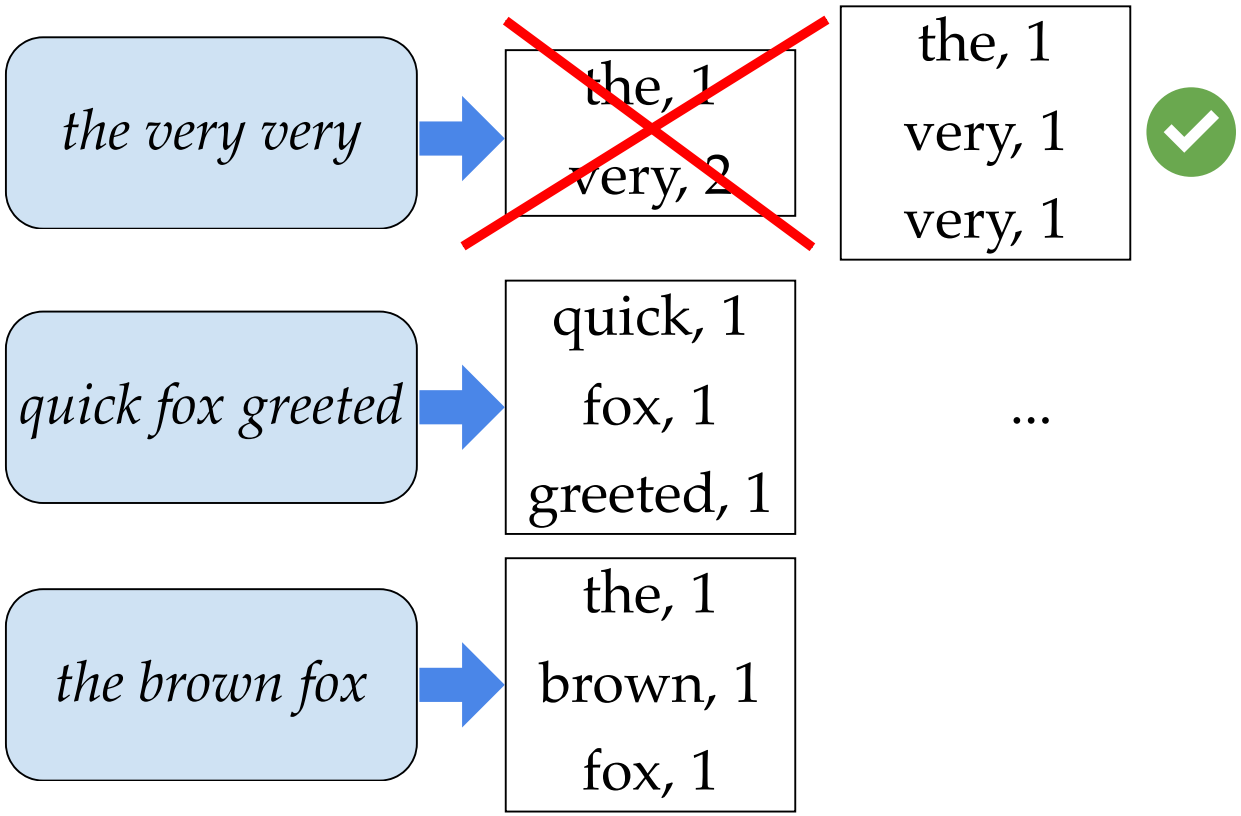
- There are 2 "phases" where we can install parallelism.
  - **map** the input to some intermediate data representation
  - **reduce** the intermediate data representation into final result



Lecture 23: Overview of MapReduce

- The first phase focuses on finding, and the second phase focuses on summing. So the first phase should only output 1's, and leave the summing for later.

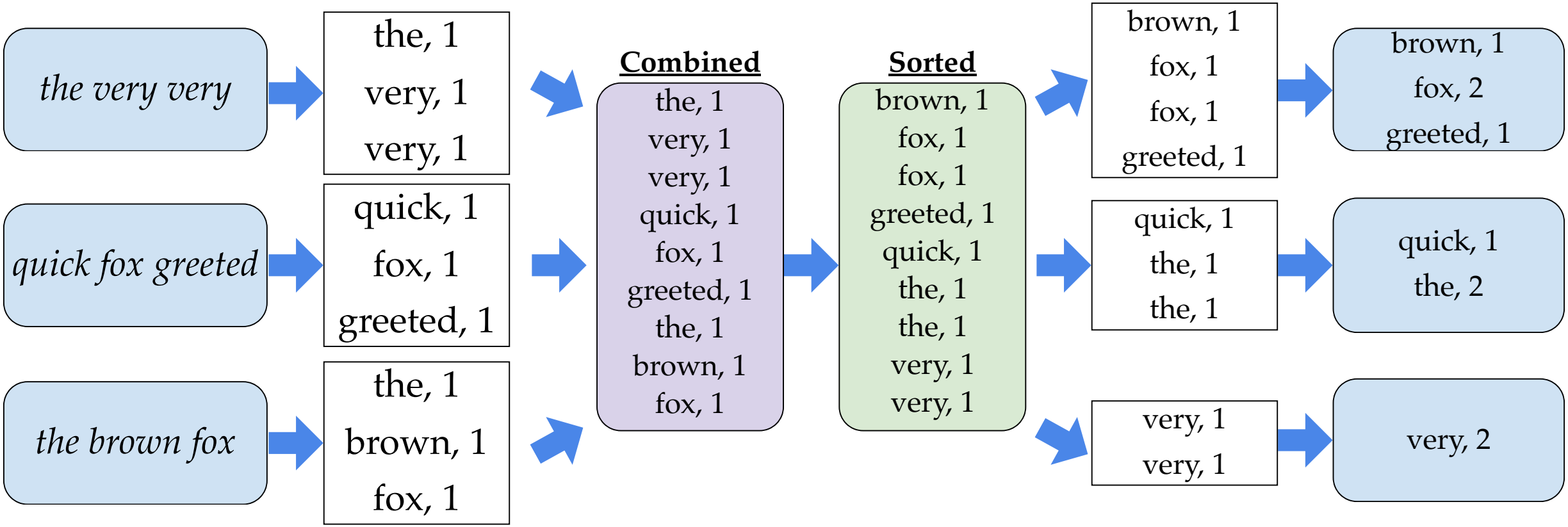
- Our example is still: *"the very very quick fox greeted the brown fox"*



Lecture 23: Overview of MapReduce

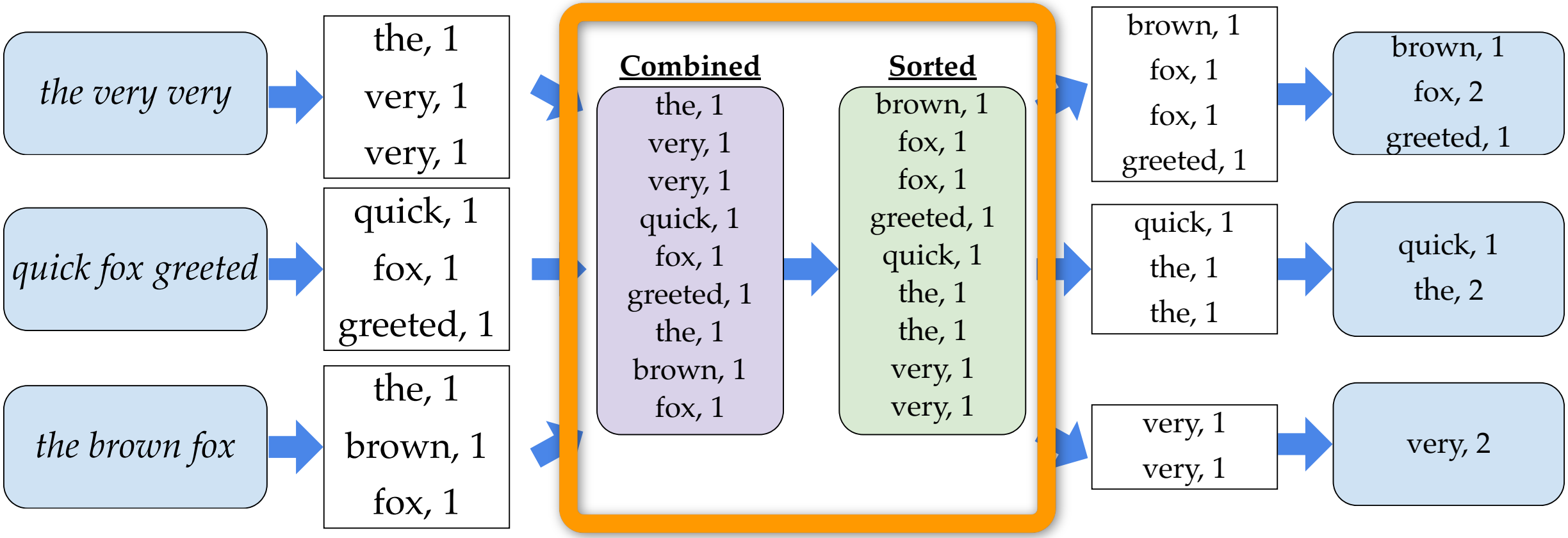
- There are 2 "phases" where we can install parallelism.
  - **map** the input to some intermediate data representation
  - **reduce** the intermediate data representation into final result

- Note the diagram below includes a chunk with two "very, 1"s, as opposed to the one "very, 2" that generated prior.



Lecture 23: Overview of MapReduce

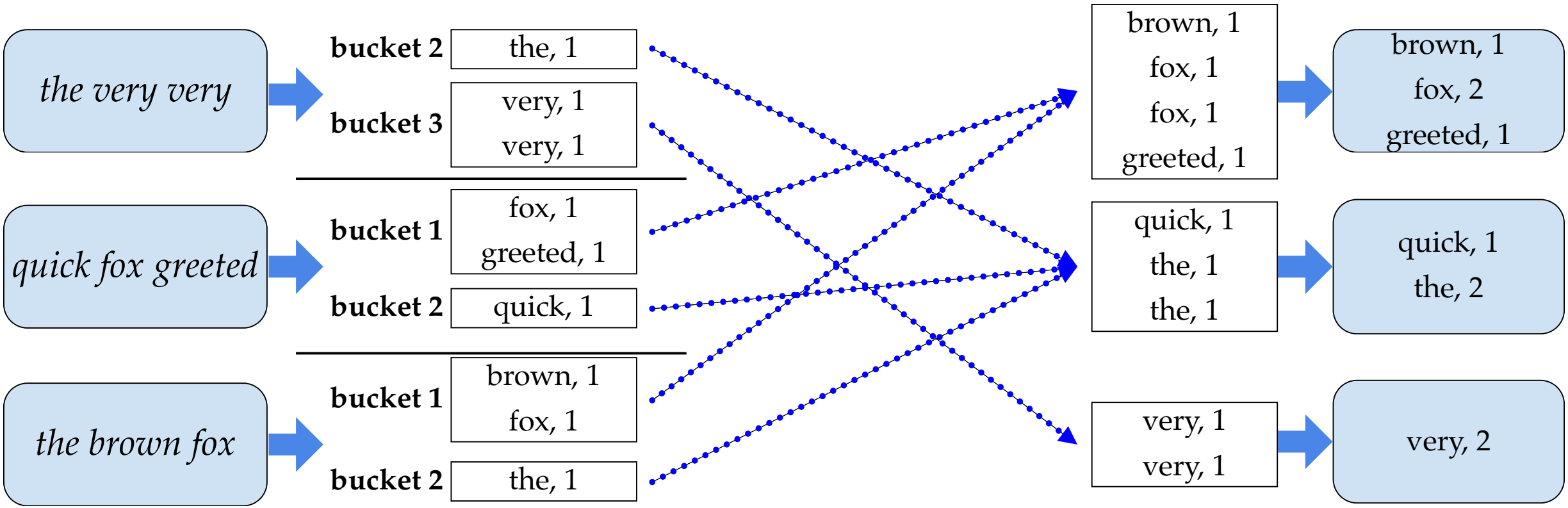
- Question: Is there a way to parallelize this operation as well?
- Another idea: Have each map task separate its data in advance of each reduce task. Then each reduce task can combine and sort its own data.



Lecture 23: Overview of MapReduce

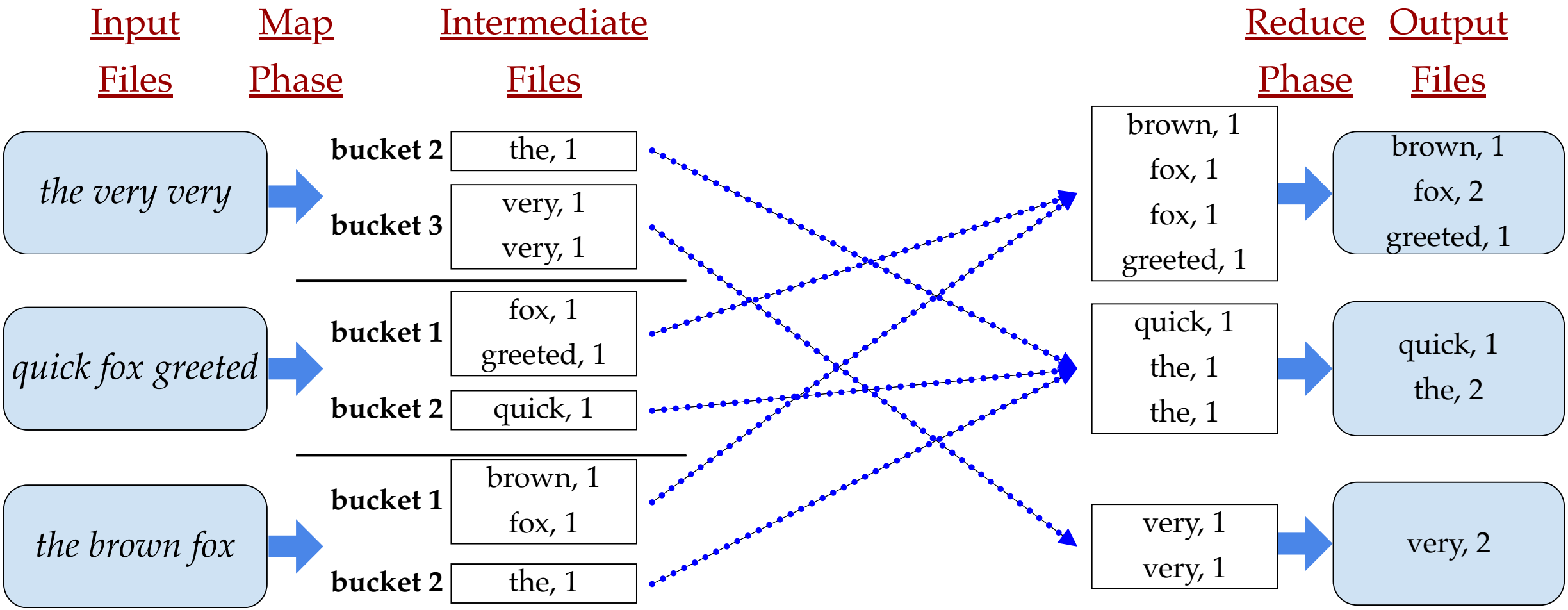
- Question: Is there a way to parallelize this operation as well?
- Another idea: Have each map task separate its data in advance of each reduce task. Then each reduce task can combine and sort its own data.

bucket # = *hash(key)* % *R* where R = # reduce tasks (3)



Lecture 23: Overview of MapReduce

- Question: Is there a way to parallelize this operation as well?
- Another idea: Have each map task separate its data in advance for each reduce task. Then each reduce task can combine and sort its own data.



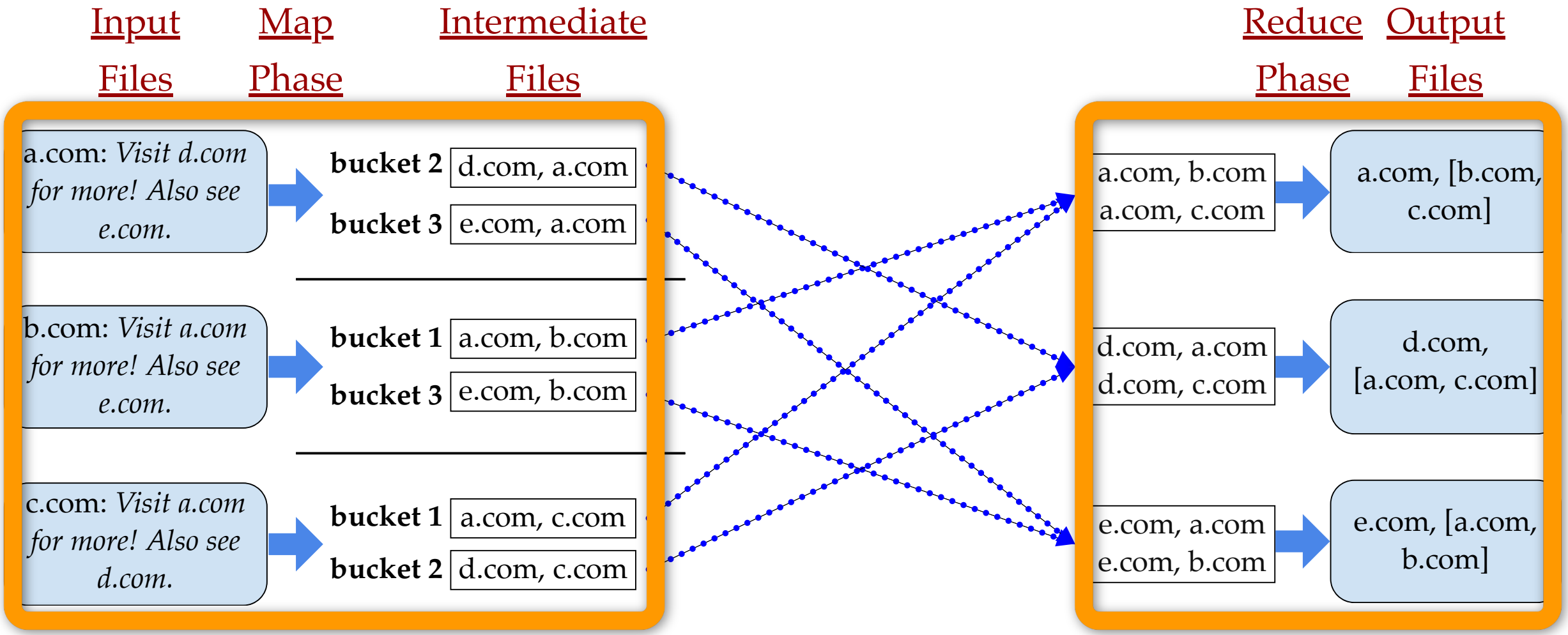
Lecture 23: Overview of MapReduce

- Task: We have web pages, and want to make a list of what web pages link to a given URL.
- Possible Approach: Leverage our existing idea to read web pages and build a URL to list(web page) map.
- **Fundamentally, this is a simplified version of PageRank 😊. In fact, Google once used MapReduce to build an in-memory index of the entire web.**

- How can we parallelize this?
  - Idea: Split web pages into chunks, find URLs in each chunk concurrently
  - Problem: What if a URL appears in multiple chunks? We need to merge the lists of web pages.
  - Better Idea: Use hashing to split the intermediate output by reduce task, and have each reduce task merge, sort and reduce concurrently.
  - Example: 3 web pages (1 per group): a.com, b.com, c.com

Lecture 23: Overview of MapReduce

- How can we parallelize this?
  - Current Thinking: Use hashing to split the intermediate output by reduce task, and have each reduce task merge, sort and reduce concurrently.
  - Example: 3 web pages (1 per group): a.com, b.com, c.com



Lecture 23: Overview of MapReduce

- Case Study: **Counting Word Frequencies**
  - Sequential Approach: program that reads a document and builds a word to frequency map
  - Parallel Approach: split document into chunks, count words in each chunk concurrently, partitioning output. Then, sort and reduce each chunk concurrently.

- Case Study: **Inverted Web Index**
  - Sequential Approach: program that reads web pages and builds a URL to list(web page) map
  - Parallel Approach: split web pages into chunks, find URLs in each chunk concurrently, partitioning output. Then, sort and reduce each chunk concurrently.

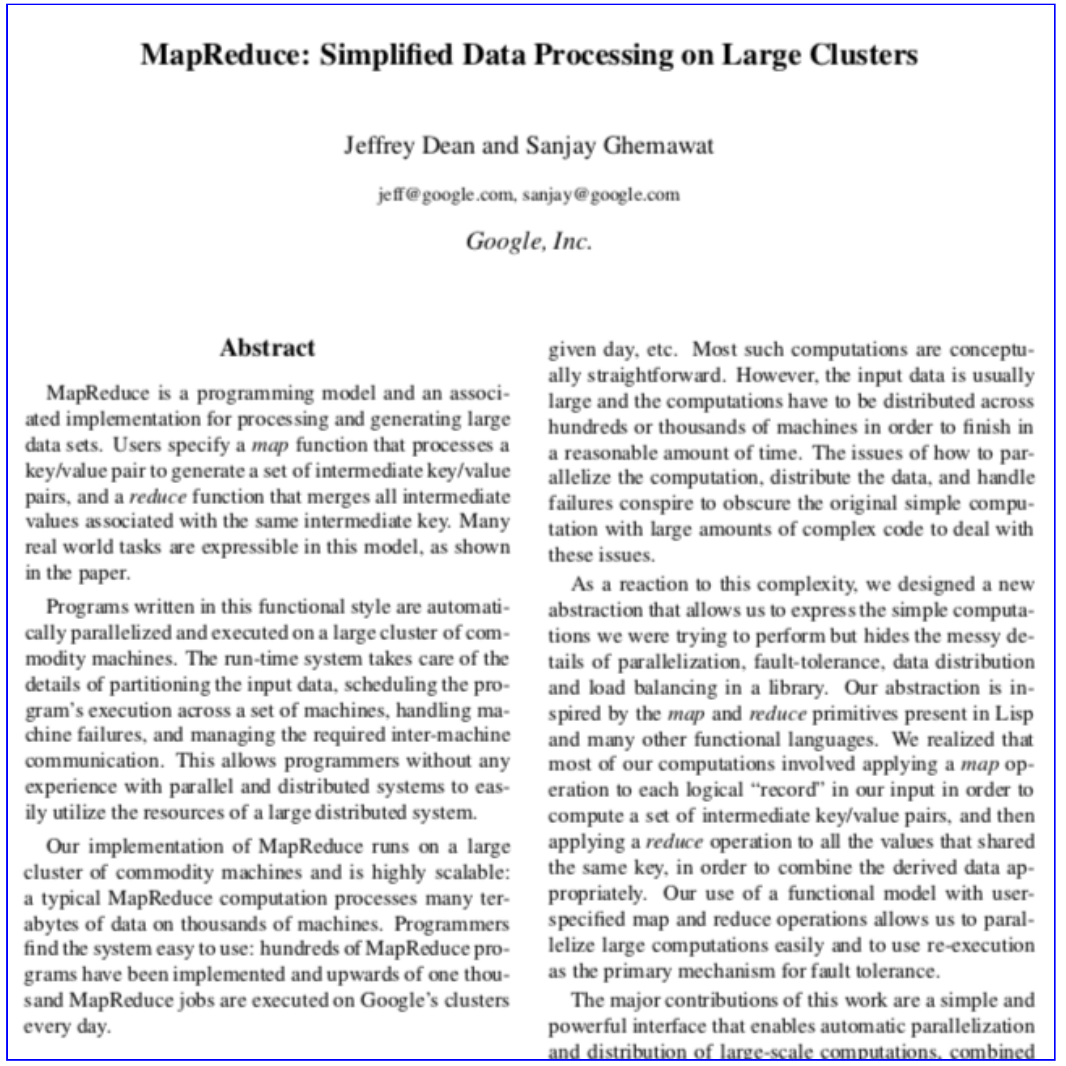
- We framed these problems as a two-step process:
  - map the input to some intermediate data representation
  - reduce the intermediate data representation into final result

- Not all problems conform to this approach. But if they do, we can parallelize them more or less the same way we have for these two examples.
  - The map and reduce steps are specific to the problem, but the overarching approach is the same regardless.

Lecture 23: Overview of MapReduce

- **MapReduce** executes programs in parallel, provided you specify the input, the *map* step and the *reduce* step.
- **MapReduce**'s primary goal is to make running programs across multiple machines as easy as possible.
- Many challenges in writing programs spanning many machines, including:
  - machines failing
  - communicating over the network
  - coordinating tasks
- MapReduce handles all of these! Users needs to frame their problem as a MapReduce-compatible one—that is, they need to provide clear map + reduce steps—and they can easily run it within a MapReduce framework.

- This is an example of how the right *abstraction* can revolutionize computing.
  - You'll see if you tackle Assignment 7!
  - MapReduce is so well-designed and specified that implementing it is much easier than you might think.
- An open source implementation immediately appeared: Hadoop.



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 associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined

Published by Google in 2004. Read it [here].

Lecture 23: Overview of MapReduce

- Programmer must implement map and reduce steps:
  - `map(k1, v1) -> list(k2, v2)`
  - `reduce(k2, list(v2)) -> list(v2)`
- Here's pseudocode for the map and reduce components needed for our word counting example:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for word w in value:
    EmitIntermediate(w,"1")
```

```
reduce(String key, List values):
  // key: a word
  // values: a list of counts
  int result = 0
  for v in values:
    result += ParseInt(v)
  Emit(AsString(result))
```

- Some terminology
  - **map step** - the step that goes from input data to intermediate data
  - **reduce step** - the step that goes from intermediate data to output data
  - **worker** - a machine that performs map or reduce tasks
  - **leader** or **orchestrator** - a machine that coordinates all the workers
  - **task** - a single piece of work—either a map a reduce step—managed by a worker

Lecture 23: Overview of MapReduce

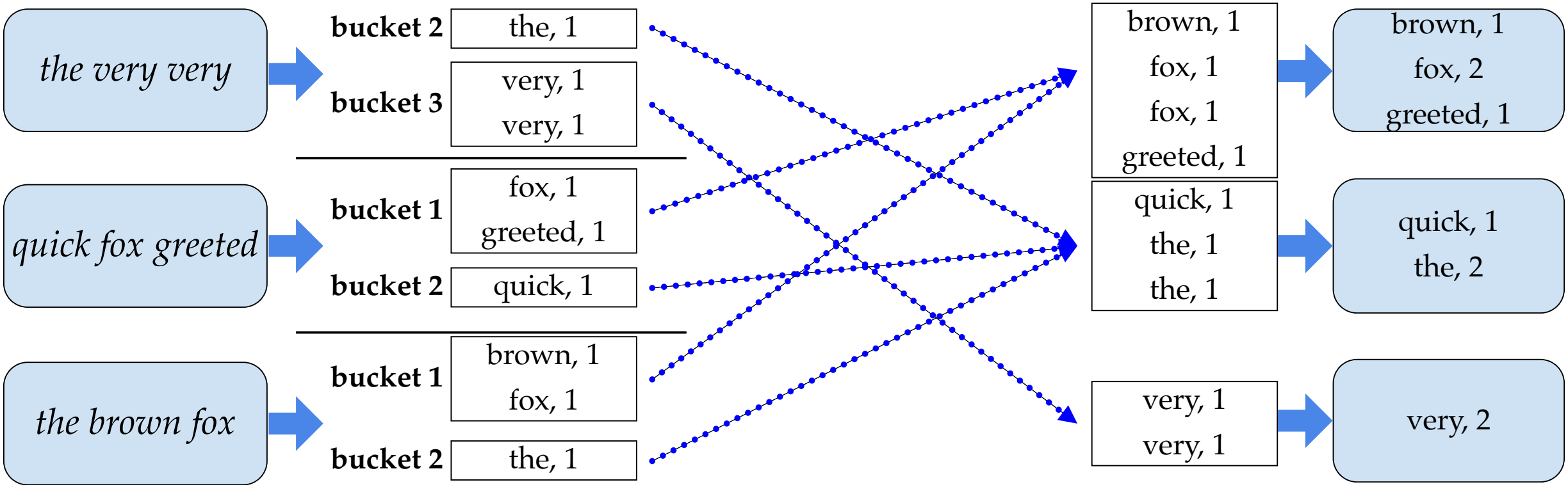
- User specifies information about the job they wish to run:
  - input data
  - map component
  - reduce component
  - # map tasks M (perhaps set to reach some target size for task data)
  - # reduce tasks R (perhaps set to reach some target size for task data)
- MapReduce partitions input data into M pieces, starts program running on cluster on machines - one will be the orchestrator, the others will be workers
- Leader assigns tasks (map or reduce) to idle workers until job is done
  - Map task - worker reads slice of input data, calls **map()**, output is broadcast across R partitions on disk using **hashing % R**.
  - Reduce task - reducer is told by orchestrator where its relevant partitions are, it reads them / sorts them by intermediate key. For each intermediate key and set of intermediate values, calls reduce(), output is appended to output file.
  - If a worker fails during execution, the orchestrator re-assigns the task with hopes it'll succeed the second time.

Lecture 23: Overview of MapReduce

- User specifies information about the job they wish to run:
  - input data -> *the very very quick fox greeted the brown fox*
  - map code: to the right
  - reduce code: to the right
  - # map tasks M = 3
  - # reduce tasks R = 3

```
map(String key, String value):
  // key: document name
  // value: document contents
  for word w in value:
    EmitIntermediate(w,"1")
```

```
reduce(String key, List values):
  // key: a word
  // values: a list of counts
  int result = 0
  for v in values:
    result += ParseInt(v)
  Emit(AsString(result))
```

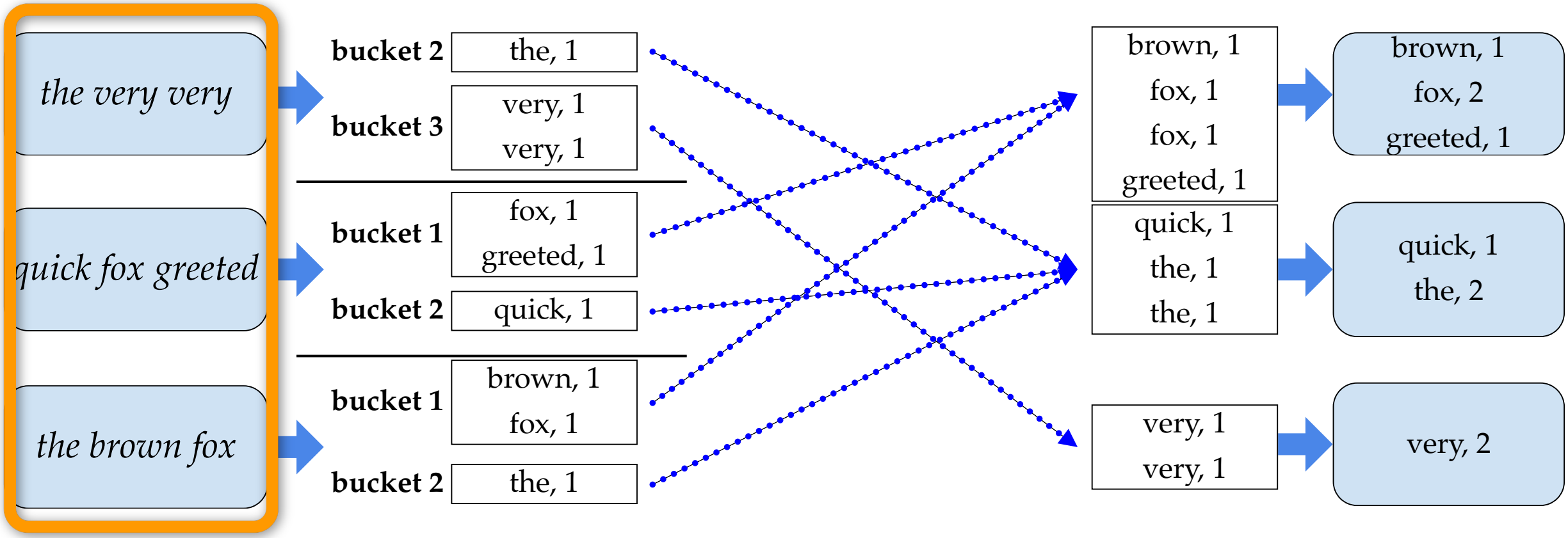


Lecture 23: Overview of MapReduce

- MapReduce partitions input data into M pieces, starts program running on cluster on machines - one will be the orchestrator, the others will be workers.
- Orchestrator assigns tasks (map or reduce) to idle workers until job is done.

```
map(String key, String value):
  // key: document name
  // value: document contents
  for word w in value:
    EmitIntermediate(w,"1")
```

```
reduce(String key, List values):
  // key: a word
  // values: a list of counts
  int result = 0
  for v in values:
    result += ParseInt(v)
  Emit(AsString(result))
```

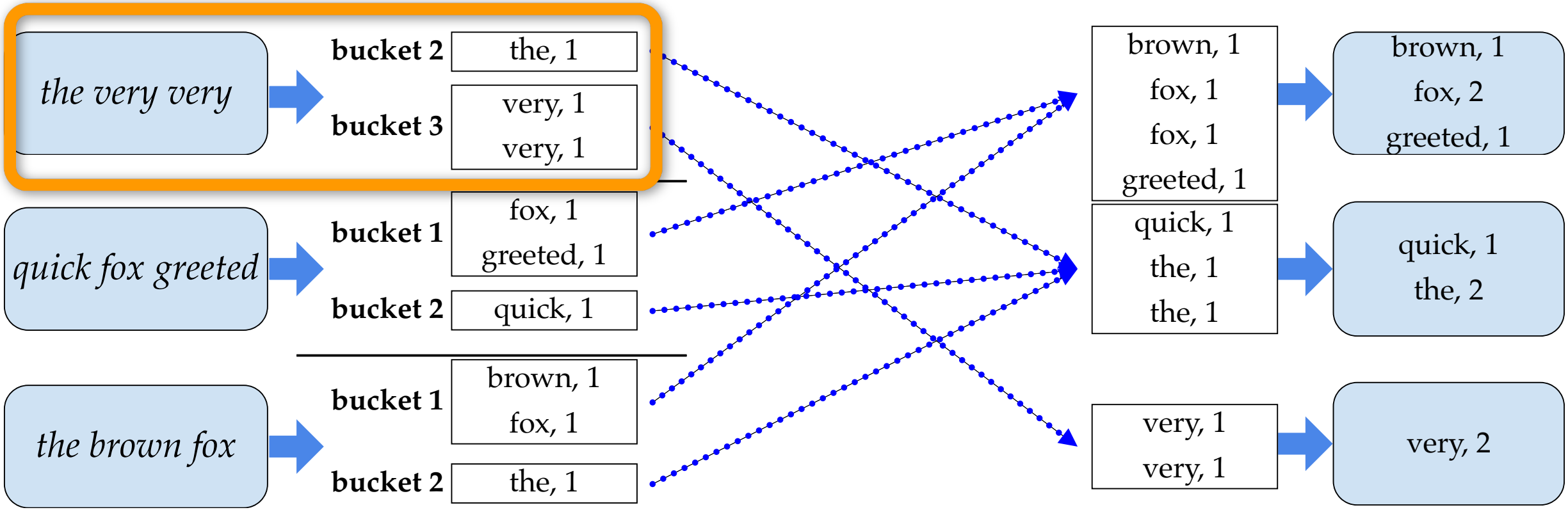


Lecture 23: Overview of MapReduce

- Map task - worker reads slice of input data, calls **map()**, output is broadcast across R (= 3) partitions on disk using hashing % R.

```
map(String key, String value):
  // key: document name
  // value: document contents
  for word w in value:
    EmitIntermediate(w,"1")
```

```
reduce(String key, List values):
  // key: a word
  // values: a list of counts
  int result = 0
  for v in values:
    result += ParseInt(v)
  Emit(AsString(result))
```

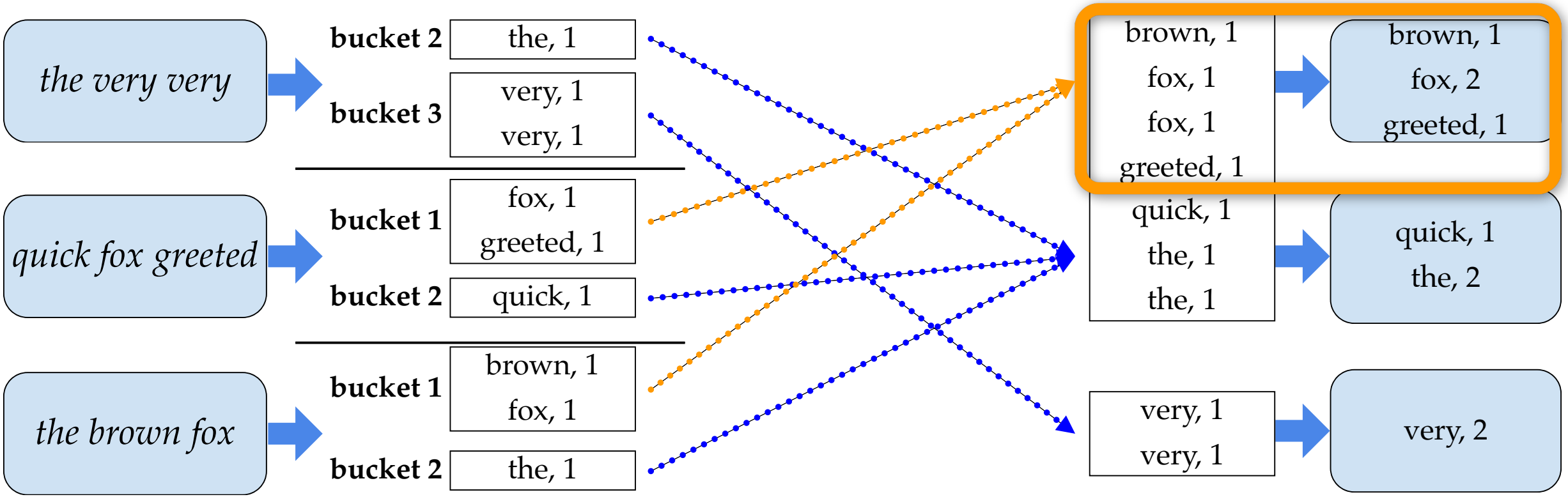


Lecture 23: Overview of MapReduce

- Reduce task - reducer is told by orchestrator where its relevant partitions are, it reads them / sorts them by intermediate key. For each intermediate key and set of intermediate values, calls reduce(), output is appended to output file.

```
map(String key, String value):
  // key: document name
  // value: document contents
  for word w in value:
    EmitIntermediate(w,"1")
```

```
reduce(String key, List values):
  // key: a word
  // values: a list of counts
  int result = 0
  for v in values:
    result += ParseInt(v)
  Emit(AsString(result))
```

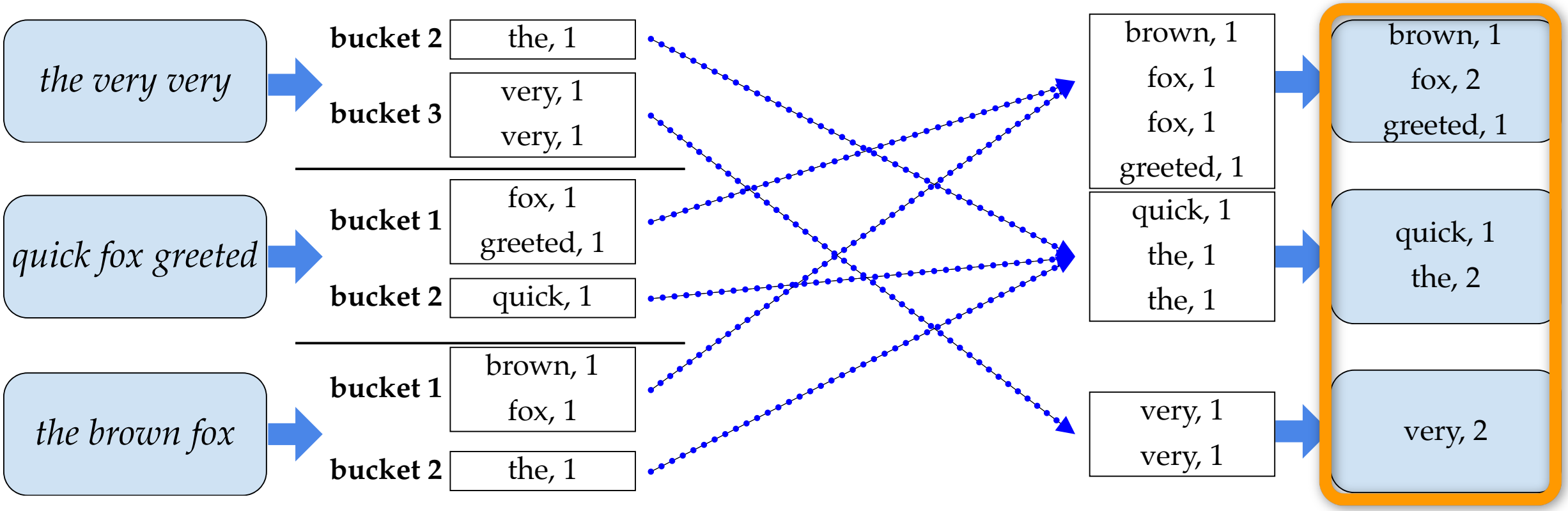


Lecture 23: Overview of MapReduce

- Orchestrator repeatedly assigns tasks (map or reduce) to idle workers until job is done.

```
map(String key, String value):
  // key: document name
  // value: document contents
  for word w in value:
    EmitIntermediate(w,"1")
```

```
reduce(String key, List values):
  // key: a word
  // values: a list of counts
  int result = 0
  for v in values:
    result += ParseInt(v)
  Emit(AsString(result))
```



Lecture 23: Overview of MapReduce

- Takeaways:
  - We have to execute all maps before we execute any reduces, because maps may all feed into a single reduce.
  - The number of workers is separate from the number of tasks; e.g. in the word count example, we could have 1, 2, 3, 4, ... etc. workers. A worker can execute multiple tasks, and tasks can run anywhere.
  - MapReduce framework handles parallel processing, networking, error handling, etc...
  - MapReduce relies heavily on keys to distribute load across many machines while avoiding the need to move data around unnecessarily.
    - Hashing lets us collect keys into larger units of work, e.g. for N tasks, a key K falls under the responsibility of the task whose ID is hash(K) % N.
    - All data with the same key is processed by the same working with the same reduce task.

Lecture 23: Overview of MapReduce

- Optional Assignment 7: You code to the standard MapReduce design, with tons of scaffolding and a few differences:
  - Intermediate files aren't saved locally and fetched - we rely on AFS
    - e.g. if **myth53** maps and **myth65** reduces, if **myth53** creates a file, it's also visible to **myth65** because AFS is a networked filesystem! No need to transmit over the network ourselves.
  - You specify # mappers and # reducers, not # workers
  - You provide input file chunks, and that determines the number of map tasks
  - # reduce tasks = # mappers x # reducers

- We'll demo Assignment 7 if we have time. :)