Overview of MapReduce and Spark

Diego Rivera Correa

Key Learning Goals

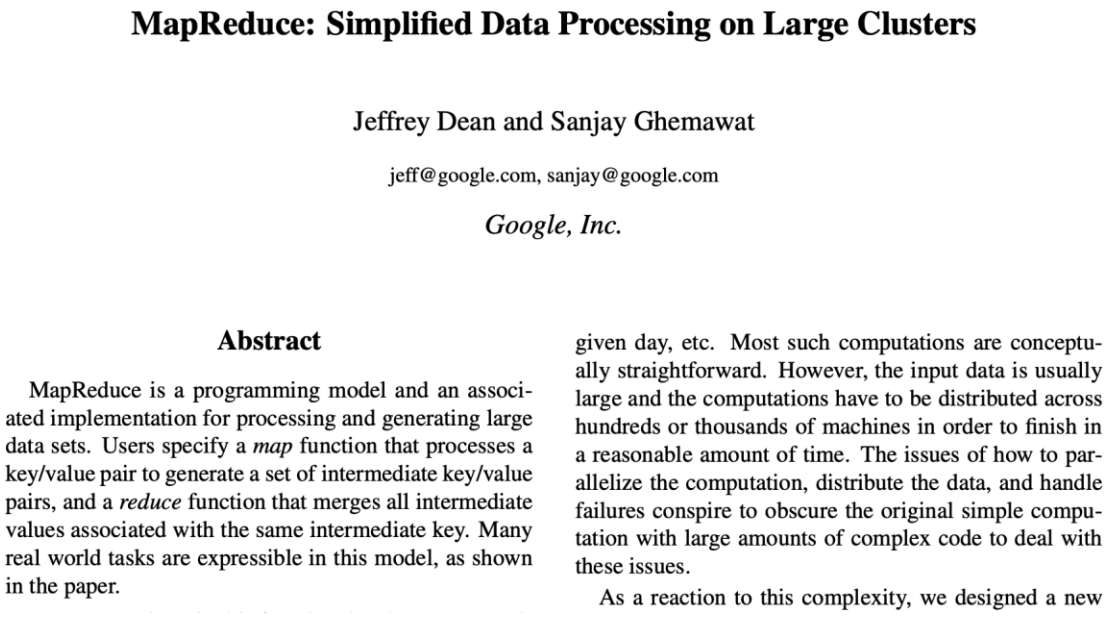
- How many tasks should be created for a job running on a cluster with w worker machines?
- What is the main difference between Hadoop MapReduce and Spark?
- For a given problem, how many Map tasks, Map function calls, Reduce tasks, and Reduce function calls does MapReduce create?
- How can we tell if a MapReduce program aggregates data from different Map calls before transmitting it to the Reducers?
- How can we tell if an aggregation function in Spark aggregates locally on an RDD partition before transmitting it to the next downstream operation?

Key Learning Goals

- Why do input and output type have to be the same for a Combiner?
- What data does a single Mapper receive when a file is the input to a MapReduce job? And what data does the Mapper receive when the file is added to the distributed file cache?
- Does Spark use the equivalent of a shared-memory programming model?

Introduction

- MapReduce was proposed by Google in a research paper. Hadoop MapReduce implements it as an open-source system.



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.

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

Compilers don't warn Jeff Dean. Jeff Dean warns compilers.

Introduction

- Spark originated in academia—at UC Berkeley—and was proposed as an improvement of MapReduce.

Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.

- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it

Word Count

- To get a feel for MapReduce and Spark, let's dive right in and look at Word Count—the *hello-world* program for distributed big-data processing.

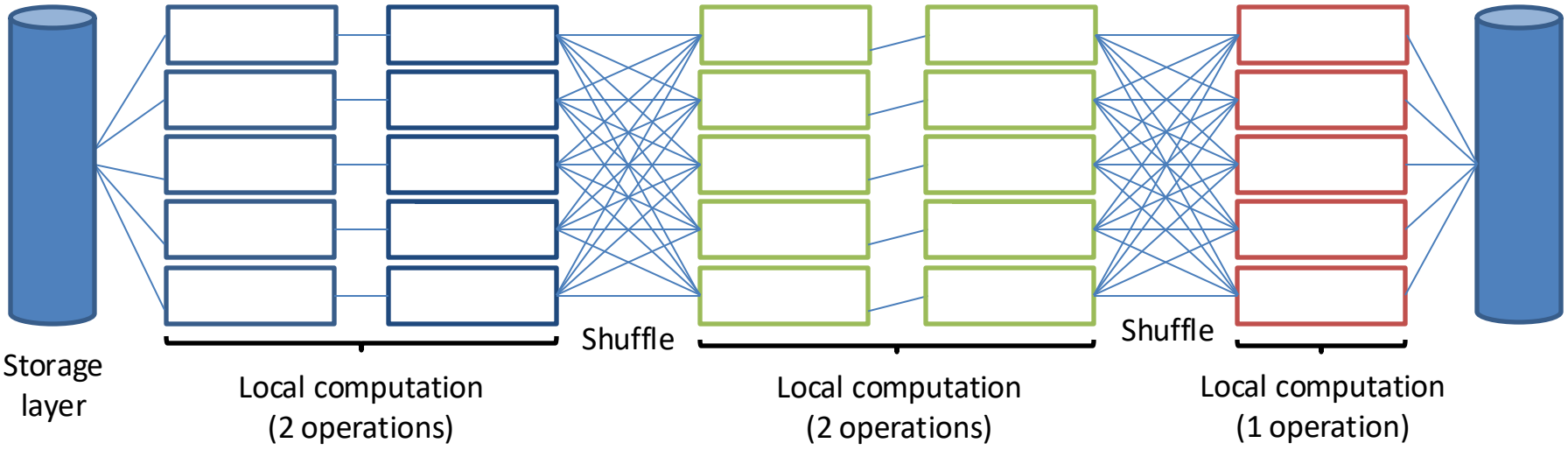
- When designing a parallel data-processing program, it is helpful to ask "what data should be processed together to get the desired result?"

Quick MapReduce Terminology

- **Worker machine -** a physical node in the cluster.
  - It can run multiple containers simultaneously.
- **Container -** a process allocated by YARN on a worker machine.
  - It's the resource envelope (some CPU, some RAM) in which a task executes.
- **Task** - the logical unit of work (a Map task or a Reduce task). It runs inside a container.
  - One container runs one task at a time.
- **Driver**: Your main() program on the client machine.
  - Configures the job (Mapper class, Reducer class, input/output paths, etc.), submits it to the cluster, and reads results after completion.

Anatomy of a Distributed Data-Processing Job

- A distributed data-processing job consists of two alternating phases:
  - **Shuffle phase**: ensures that data ends up on the right machine.
  - **Local computation phase**: each partition of the data is processed independently.
- The central question of this course: how do we partition data and computation so that local computation can proceed independently with minimal need for shuffling?



Word Count: The Algorithm

- How can we count words using local computation and shuffling? Let us think about the problem and *abstract away all implementation details*.

- The input are documents in the storage layer. Hence, abstractly, what does each worker receive?

Word Count: The Algorithm

- What can the worker do locally before shuffling becomes necessary?

Word Count: The Algorithm

- What can the worker do locally before shuffling becomes necessary?
  - It counts the word frequencies in its local list.
- To get the totals, workers must communicate their local counts. We want to distribute the summation work by making each worker *responsible* for totaling a subset of the words.
  - This means the worker responsible for word x collects *all* local counts for x and emits the total.
  - This requires shuffling! Any worker in the system may have encountered word x.

Algorithm Details

- How do we decide which worker is responsible for which word?
- We want to distribute load evenly, i.e., give each worker approximately the same number of words.
- We need to make sure that there is exactly one responsible worker per word. This is a distributed consensus problem!

Algorithm Details

- Fortunately, there exists a simple answer—hashing! Given w workers, we assign word x to the worker with number *h(x) mod w*.
  - Function h() maps a string to an integer. The mod operation determines the remainder when dividing h(x) by w. A good hash function achieves near-uniform load distribution.
  - If everybody uses the same hash function, they all assign words in the same way to workers. This elegantly solves the consensus problem.

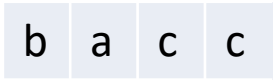
Algorithm Details

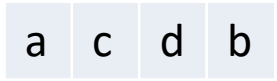
- What happens when a worker fails? Then things become complicated. Hence instead of hashing to workers, in practice we hash to tasks.

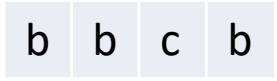
- With t tasks, we assign word x to task h(x) mod t, leaving it to the application master to assign tasks to worker machines. If a worker fails, its tasks are assigned to another worker.

Algorithm Illustration

Input (a letter represents a word)

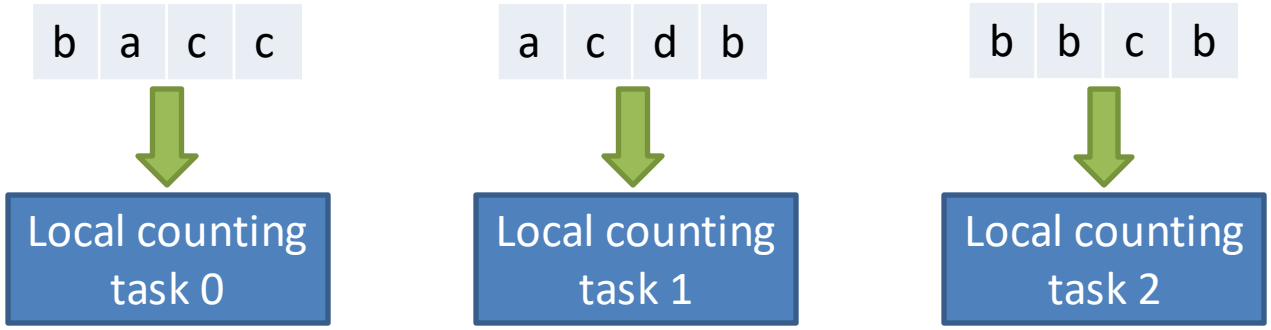






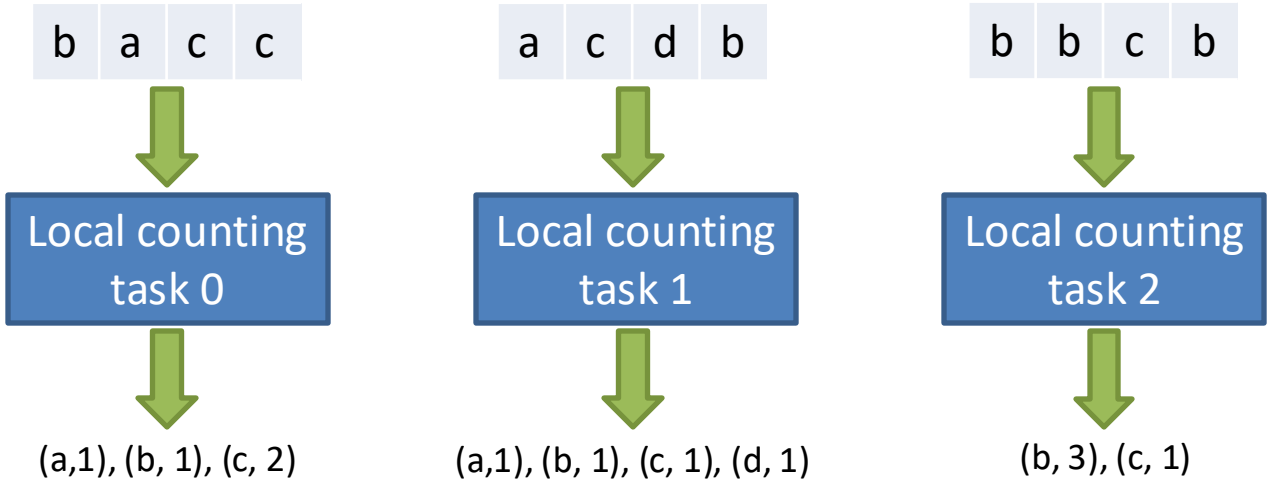
Algorithm Illustration

Input (a letter represents a word)

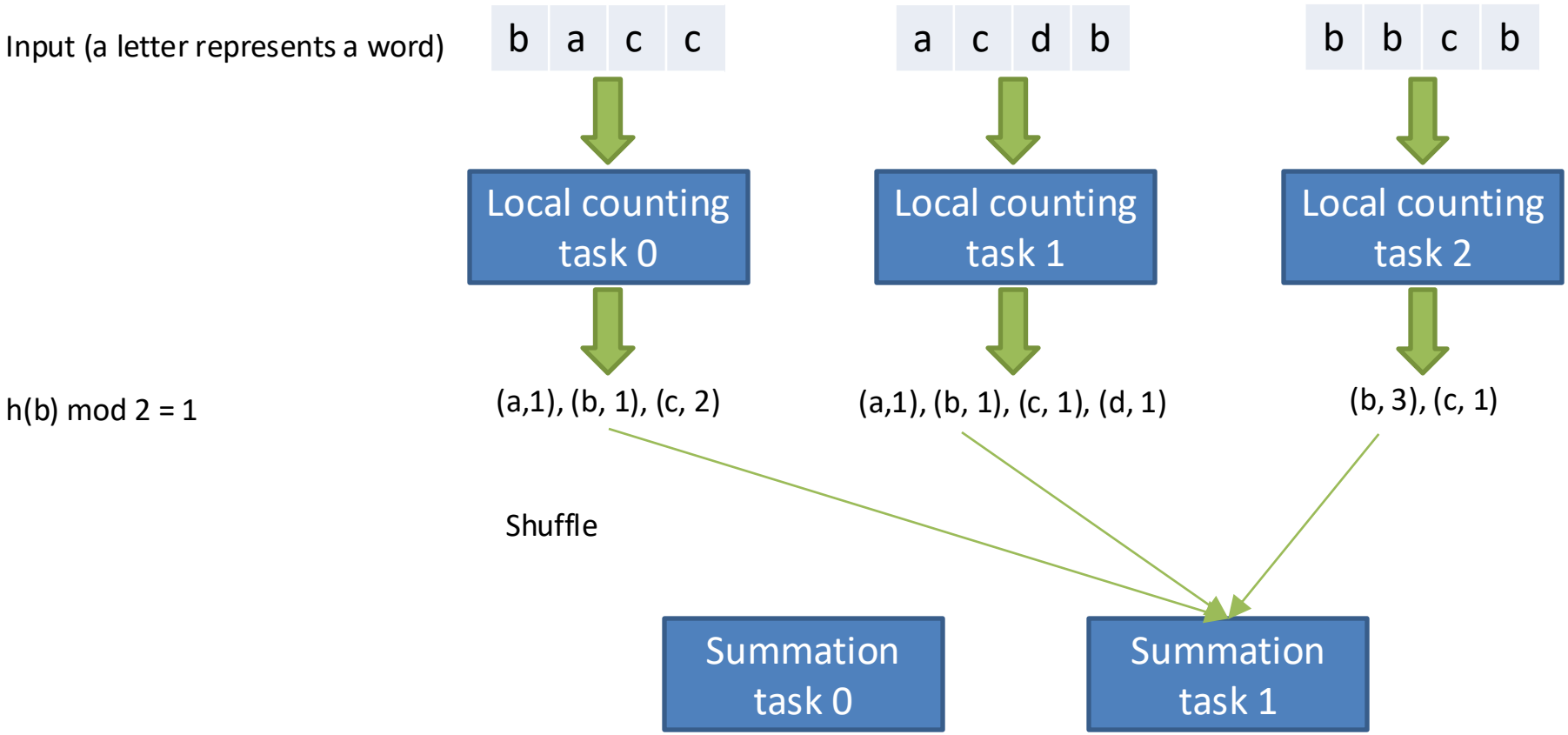


Algorithm Illustration

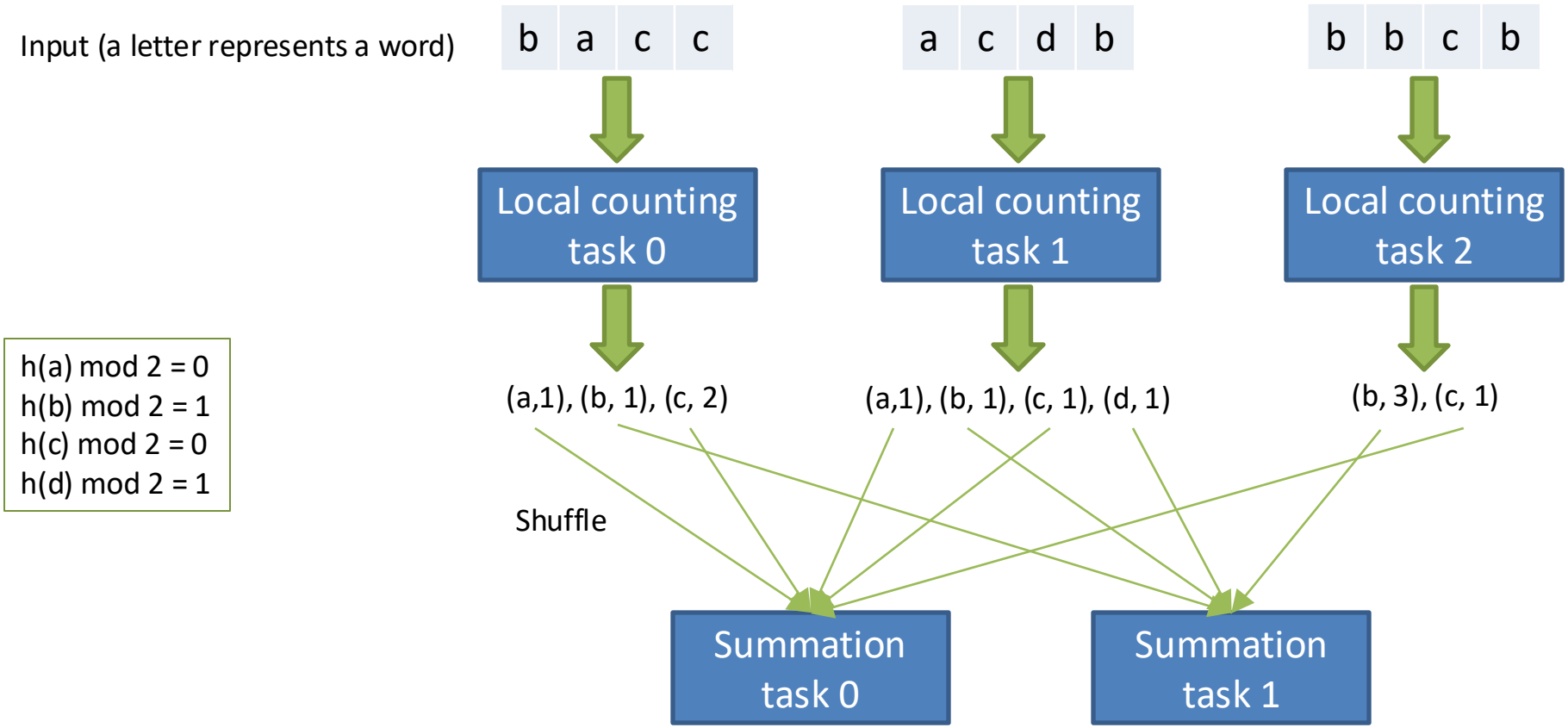
Input (a letter represents a word)



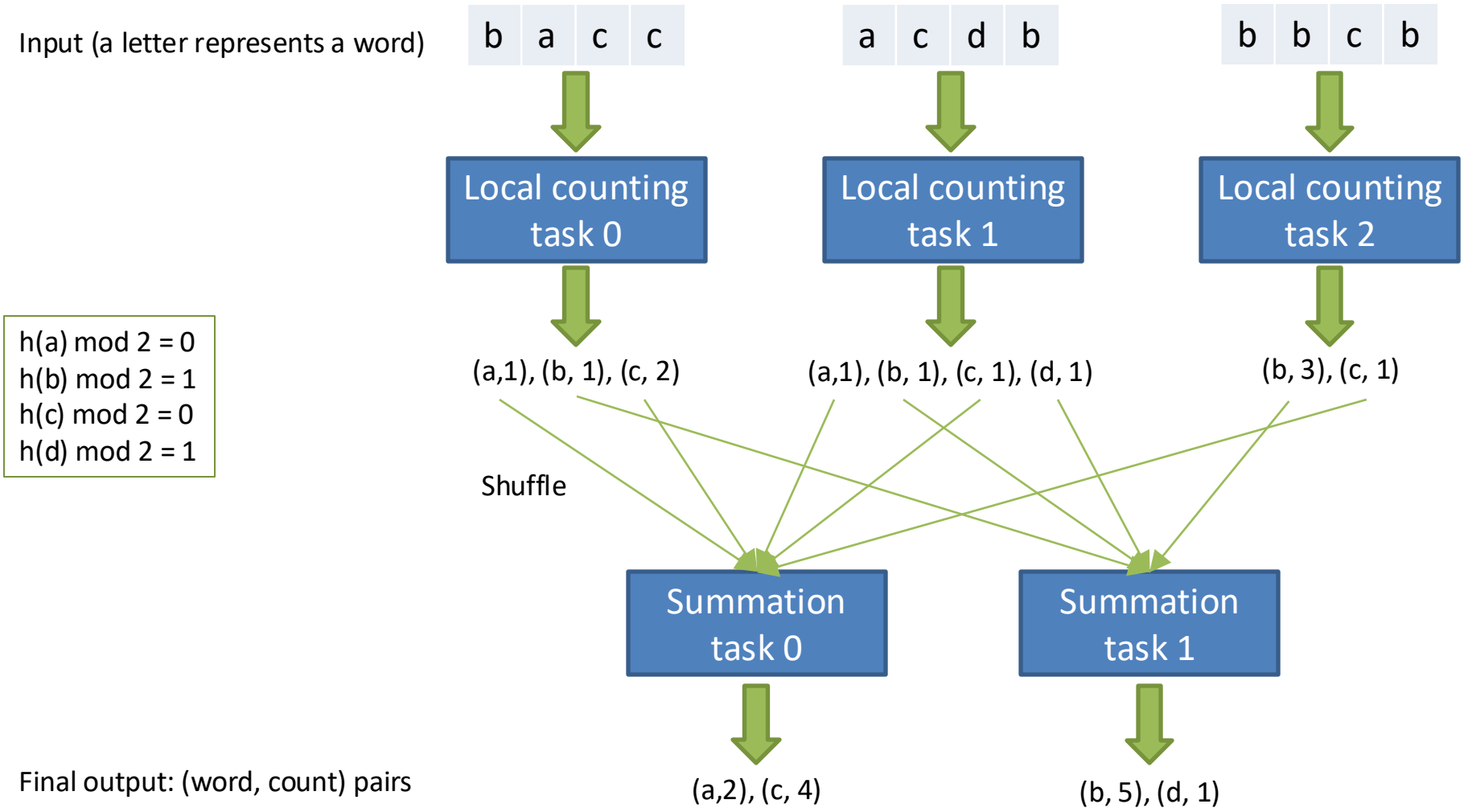
Algorithm Illustration



Algorithm Illustration



Algorithm Illustration



Algorithm Discussion

- In the example, there are five different tasks: three for local counting and two for final summation. We could have created more, or fewer, tasks for both rounds.
  - How many tasks should we create for best performance?

- The application master assigns tasks to containers on worker machines. Those containers were allocated earlier when the application master requested resources from the resource manager.
  - In what order should the tasks be scheduled?

Number of Tasks

- **Approach 1: Exactly w Tasks (One Per Worker Machine)**

Number of Tasks

- **Approach 1: Exactly w Tasks (One Per Worker Machine)**
- Simple idea: give each worker machine exactly one task with L/w of the total load.
  - Problem 1: If a worker machine dies, another machine takes over its task. That machine now has double the work. Everyone else is done and waiting.
  - Problem 2: If one machine is 2× slower than the rest, the entire job waits for it. No way to redistribute its work.
- Both problems cause the same thing: one overloaded machine, everyone else idle, and the job finishes at the speed of the slowest machine.

Number of Tasks

- **Approach 2: 0.9w Tasks (Leave Some Machines Idle)**

Number of Tasks

- **Approach 2: 0.9w Tasks (Leave Some Machines Idle)**
- Idea: Leave 10% of machines idle as standby. If a machine fails or is slow, an idle machine picks up the task.
- Better than Approach 1, but still fragile:
  - Each task is still large ($L/0.9w \approx L/w$). If failure happens near the end of a task, nearly all of that work is lost and must be restarted from scratch.
  - The idle machines are wasted capacity during normal (failure-free) execution.

Number of Tasks

- **Approach 3: 10w Tasks (Many Small Tasks)**

Number of Tasks

- **Approach 3: 10w Tasks (Many Small Tasks)**
- Idea: Create 10 tasks per machine. Application master assigns one task per container. When a container finishes, it gets the next task. This solves both:
  - Failure: a machine dies and only 1/10 of the per-machine work (L/10w) is lost and needs re-execution.
  - Slow machine: fast machines automatically receive more tasks. Load balances itself — no manual tuning needed.

How Many Tasks (cont.)?

- Does this imply tasks should be as small as possible? No!
  - For some problems, more fine-grained partitioning increases data duplication and hence total cost. We see this in future modules (Joins).
  - Managing, starting up, and terminating a task adds a cost, no matter how small the task. Assuming this cost is in the order of milliseconds or seconds, then a task should perform at least a few minutes' worth of useful work.

How Many Tasks (cont.)?

- Map tasks: determined by the input, not by your code.
  - # Map tasks ≈ total input size / file split size.
  - Default split size = HDFS block size.
  - Want more Map tasks? Decrease the split size (configuration setting) or split your input into more files before the job.
- Reduce tasks: determined by the programmer.
  - Set directly in the driver.
  - More Reduce tasks = more partitions = smaller workload per Reducer.

Rules of Thumb for Number of Tasks

- For problems where more fine-grained partitioning does *not* increase data duplication, it is recommended to set the number of tasks to a small multiple of the number of workers, e.g., 10w.
  - If that setting creates tasks that run for more than about 30-60 min, increase the number of tasks further. Long-running tasks are more likely to fail and they waste more resources for restarting.

Rules of Thumb for Number of Tasks

- When more fine-grained partitioning significantly increases data duplication, consider fewer tasks, e.g., w tasks.

- In some cases, e.g., for Map tasks in MapReduce, there are good default settings like “create one Map task per input file split.”

Order of Task Execution

- Each of the local counting tasks is independent of the others, therefore they can be executed in any order. The same holds true for the summation tasks.

- MapReduce and Spark are designed to create and manage this type of independent tasks.

Order of Task Execution

- Local counting must happen before the summation.
  - For this specific problem, the summation tasks could incrementally update their sums as data is arriving from the local counting tasks.
  - For other problems, e.g., finding the median, tasks in round i can only start processing data after all tasks in round i-1 have completed emitting and transferring their output.

From Algorithm Design to Implementation

- Now that we understand how to count words in parallel, we need to determine how to implement our ideas.
- As we will soon see, there are tradeoffs in terms of ease of use/programming, available options for controlling program behavior, and the resulting performance.
- We will look at 3 competing approaches: distributed relational database systems (DBMS), MapReduce, and Spark Scala.

Word Count in a DBMS

- DBMS queries are written in SQL, as shown for Word Count below. Here we assume that documents are stored in a table with schema (documentID, offsetInDocument, word).

- The query looks the same, no matter if the DBMS is distributed or not. This is the beauty of SQL: we specify what we want, not how to get it. The latter is determined automatically by an optimizer.

SELECT word, count(*) FROM Documents GROUP BY word

Word Count in MapReduce

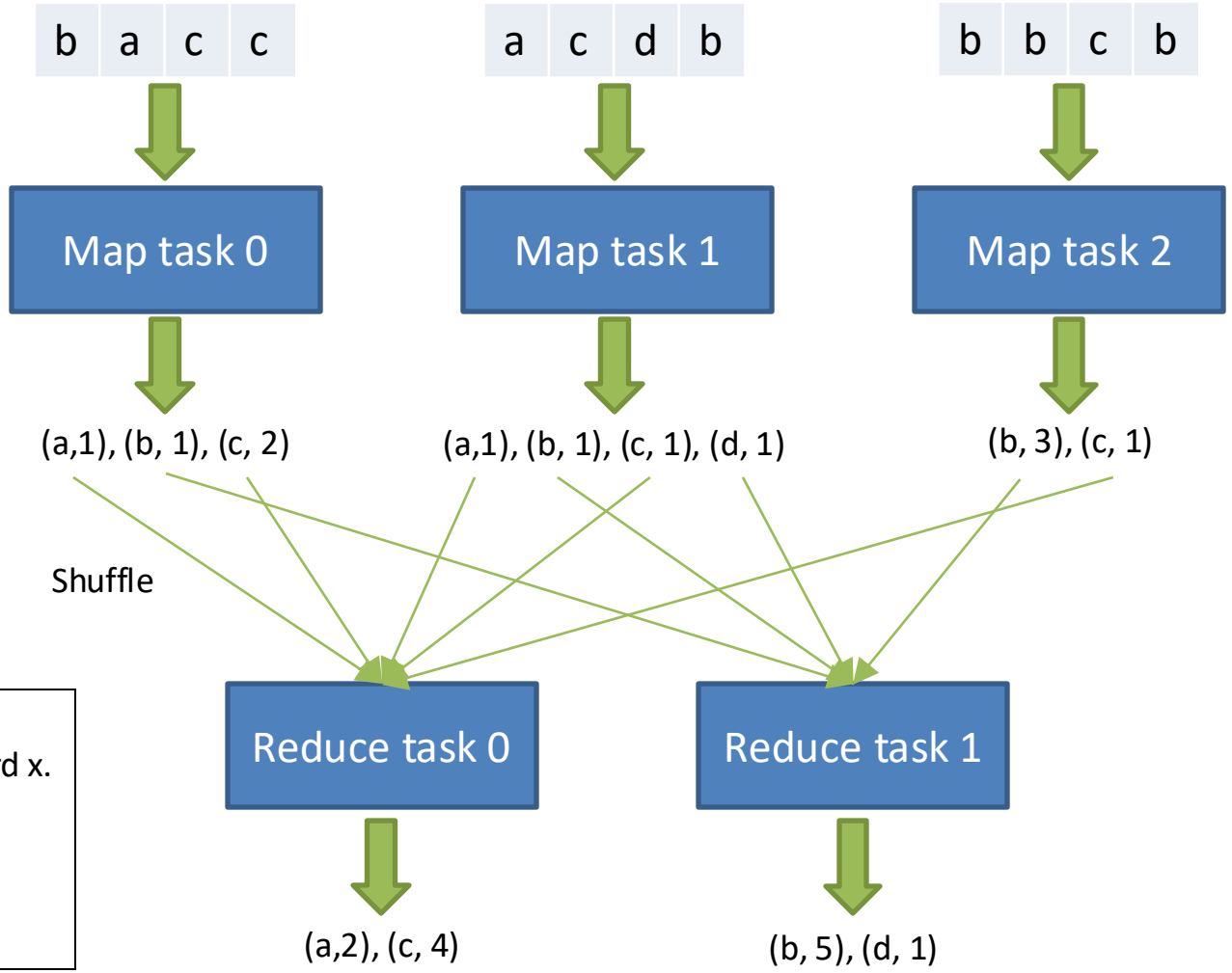
- The programmer specifies two functions: Map and Reduce. They look like sequential functions, i.e., there is no parallelism in them. We simply define how to process a given input.
  - The program has a few more components, but we will cover those later.
- Map sets up the shuffle phase, so that each Reduce call receives the right data for local computation.

Word Count in MapReduce

```
map( documentID d, word x )
   emit( x, 1 )

combine( word x, [n1, n2,…] )
   // Same as reduce below
```

```
reduce( word x, [n1, n2,…] )
   // Each n is a partial count for word x.
   total = 0
   for each n in input list
      total += n
   emit( x, total )
```



Program Discussion

- By default, for each file split, there is exactly one Map task processing it. The Map task calls the Map function one-by-one on each record in the input file split. For word x, the Map call emits (x, 1) to signal that 1 more occurrence of x was found.
  - MapReduce works with key-value pairs. By convention, the first component is the key; the second is the value.

Program Discussion

- There is exactly one Reduce function call for every key, i.e., distinct word. It receives all local counts for that word and totals them.
  - Data is automatically moved by the MapReduce environment: it groups all Mapper output by key and sends it to the corresponding Reduce task.
  - The Reduce function call for a key receives *all* the values associated with that key.

Program Discussion

- Note that in Map task 0, Map would emit (c, 1) twice instead of the desired (c, 2). Since Map here works on a single word at a time, it cannot produce the partial count.

- Fortunately, MapReduce supports a Combiner. It works like a Reducer but is executed in the Map task. Here it is identical to the Reducer.

Program Discussion (cont.)

- The assignment of words to Reduce tasks is determined by the Partitioner. The default hash Partitioner in MapReduce implements the desired assignment.

- MapReduce does all the heavy lifting of grouping the intermediate results (i.e., those emitted by Map) by their key and delivering them to the right Reduce calls.

Let's take a closer look at MapReduce's features. Understanding MapReduce will help understanding the more complex Spark approach.

MapReduce Overview

- MapReduce is a programming model and its implementation for processing large data sets.

- The programmer essentially only specifies two (sequential) functions: Map and Reduce. At runtime, program execution is automatically parallelized.

MapReduce Overview

- MapReduce provides a clever abstraction that works well for many real-world problems. It lets the programmer focus on the algorithm, while the system takes care of messy details such as:
  - Input partitioning and delivering data chunks to worker machines.
  - Creating and scheduling all tasks for execution on many machines.
  - Handling machine failures.

MapReduce Programming Model

- MapReduce program converts a list of key-value pairs to another list of key-value pairs.
- Map: (k1, v1) → list(k2, v2)
  - Takes one input record, outputs a list of intermediate key-value pairs.
  - Map calls are independent of each other — Map can only perform per-record computation.
- MapReduce automatically groups all intermediate pairs by key and passes them to Reduce.
- Reduce: (k2, list(v2)) → list(k3, v3)
  - Combines all values that share the same intermediate key.

MapReduce Programming Model

- k1 = documentID, v1 = word
- k2 = word, v2 = 1 (integer)
- k3 = word, v3 = total count
- Map: ("doc1", "hello") → [("hello", 1)]
- Reduce: ("hello", [1, 1]) → [("hello", 2)]
- Terminology: there are three key types (k1, k2, k3), but only the intermediate key k2 matters for program design.
  - It determines which Reduce call receives which data. We will usually just say "key" to mean k2.

MapReduce Execution Highlights

- Each MapReduce job has its own application master, which schedules Map tasks before Reduce tasks. It may schedule some Reduce tasks during the Map phase, so they can pull their data early from the Mappers.
  - The terms "Mapper" and "Reducer" are not used consistently. We use Mapper as a synonym for *Map task*, which is an instance of the Mapper class; analogously for Reducer.

MapReduce Execution Highlights

- When assigning Map tasks, the application master considers data location: It tries to assign a Map task to a machine that stores the corresponding input data chunk locally on disk.
  - This avoids sending data across the network—an important optimization for big data.
  - If no available machine has the chunk, then the master tries to assign the task to a machine in the same rack as the data chunk.

MapReduce Execution Highlights

- A Mapper reads its file split from the distributed file system and emits its output to the local file system—partitioned by Reduce task and sorted by key within each of these partitions.
  - We will see what this means in a few slides...

- The Mapper then informs the master about the file locations, who in turn informs the Reducers.

MapReduce Execution Highlights

- Each Reducer pulls the corresponding data directly from the appropriate Mapper disk locations and merges the pre-sorted files locally by key before making them available to the Reduce function calls.

- The output emitted by Reduce is written to a local file. As soon as the Reduce task completes, this local file is *atomically* renamed (and moved) to a file in the DFS.

Summary of Important Hadoop Execution Properties

- Map function calls inside a single Map task happen sequentially, consuming one input record at a time.

- Reduce function calls inside a single Reduce task happen sequentially in increasing key order.

Features Beyond Map and Reduce

- Even though the Map and Reduce functions usually represent the main functionality of a MapReduce program, there are several additional features that are crucial for achieving high performance and good scalability.

- We first take a closer look at the Combiner.

Combiner

- Like a Reducer, the Combiner works on Map output by processing values that share the same key. Often the code for a Combiner will be the same as the Reducer. Here are examples for when it is *not*:
  - Consider Word Count, but assume we are only interested in words that occur at least 100 times in the document collection. The Reduce function computes the total count for a word, but outputs it only if it is at least 100. Should the Combiner do the same?

Combiner

- Even though in Hadoop a Combiner is defined as a Reducer class, it is still executed in the Map phase.

- This has an interesting implication for its input and output types. Note that the Combiner's input records are of type (k2, v2), i.e., Map's output type. Since the Combiner's output will be processed by a Reducer, and the Reducer input type is (k2, v2) as well, it follows that a Combiner's input and output types must be identical.

Combiner (cont.)

- Such multi-step combining does not work for all problems.
  - For example, it cannot be applied to holistic aggregates.

- A Combiner can reduce data transfer cost from Mappers to Reducers but tends to increase Mapper-side costs for the additional processing.
  - Due to this tradeoff, Hadoop does not guarantee if or when a Combiner will be executed. The programmer needs to ensure correctness.

Partitioner

- The Partitioner determines which Reduce task receives each intermediate key. Each Reduce task calls the Reduce function once per key assigned to it.

- Default: hash-based assignment. Given key k and t Reduce tasks, assign k to task h(k) mod t. This distributes keys roughly evenly across Reduce tasks.

Maintaining Global State

- Tasks cannot communicate with each other. But sometimes we need job-wide statistics, e.g., how many input records failed to parse.
- Hadoop's solution: global counters, maintained by the application master.
  - Tasks increment/decrement counters locally. Updates are propagated to the application master.
  - The driver program (your main() method, running on the client side) reads the final counter values after the job completes.

Broadcasting Data to All Tasks

- Hadoop supports two ways of sharing the same data with all tasks.

- A small number of constants should be shared through the Context object, which is defined in the driver and passed to all tasks.

- You can also use the file cache. By adding a file to the file cache, it will be copied to all worker machines executing MapReduce tasks.

Terminology Cheat Sheet

- A MapReduce job consists of a Map phase and a Reduce phase, connected through a shuffle phase.
- **Map phase:**
  - Default number of Map tasks = number of input file splits.
  - Each Map task calls the Map function once per input record in its split.
    - What counts as a "record" is defined by the InputFormat. Default (TextInputFormat): one record = one line of text, a single word, etc.
  - Number of Map function calls = number of input records.
- **Reduce phase:**
  - The number of Reduce tasks is set by the programmer.
  - Each Reduce task calls the Reduce function once per key assigned to it by the Partitioner.
  - Reduce function calls within a task happen in key order (a consequence of the merge-sort in the shuffle phase).
  - Number of Reduce function calls = number of distinct intermediate keys.

Important Hadoop System Details

- The features discussed so far help the programmer implement their algorithm ideas.

- Below the surface, MapReduce does a lot of work that is hidden from the programmer, but still affects program performance. We will now take a closer look at some of these hidden features and processes.

Moving Data From Mappers to Reducers

- Data is transferred from Mappers to Reducers in the shuffle phase. **No Reduce function call executes** until all Map output has been transferred — this is a synchronization barrier.

- Why? A Reduce call needs the complete list of values for its key. If some Mapper hasn't finished yet, that list may be incomplete.

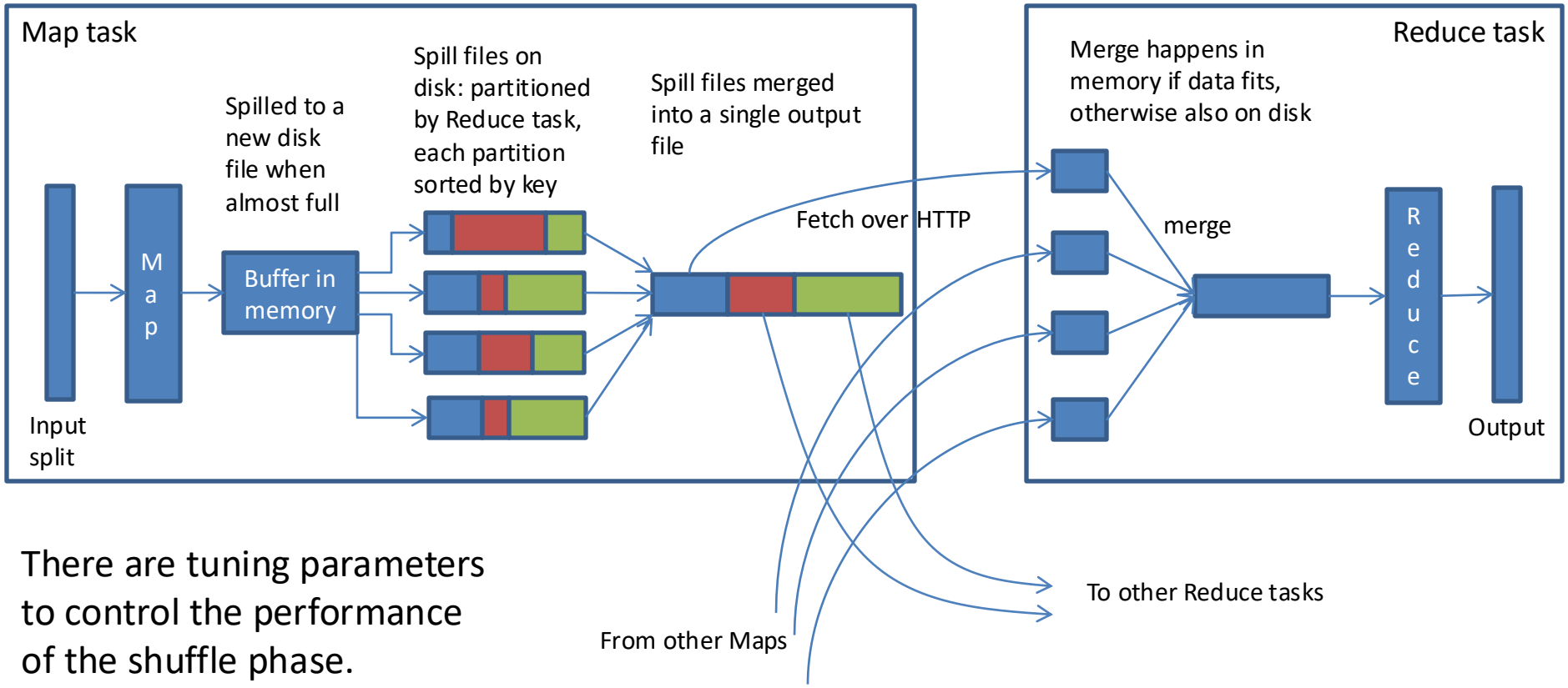
- Note: Reducers (tasks) can begin *pulling data* from completed Mappers before all Mappers finish. The barrier applies only to Reduce *function calls*, not to data transfer.

Moving Data From Mappers to Reducers

- The shuffle phase is often the most expensive part of a MapReduce job. Here is what happens on the Mapper side:
  - Map function calls emit key-value pairs to an in-memory buffer.
  - When the buffer fills, its contents are partitioned by Reduce task (using the Partitioner) and sorted by key within each partition.
  - Each sorted partition is written to a spill file on the local file system.
  - Spill files are merged and the resulting partitions are copied to the corresponding Reducers.

Shuffle Phase Overview

Reduce tasks can start copying data from a Map task as soon as it completes. Reduce cannot start processing the data until all Mappers have finished and all their output has arrived.



There are tuning parameters to control the performance of the shuffle phase.

Shuffle Phase Overview

- Suppose 2 Reduce tasks and the Partitioner assigns: h(key) mod 2
  - Partition 0 (blue): keys where h(key) mod 2 = 0, e.g., "apple," "cat," "the"
  - Partition 1 (red): keys where h(key) mod 2 = 1, e.g., "dog," "hello"

- Inside one Map task, the buffer fills and spills 3 times:
  - Spill 1: blue: [("apple",1), ("the",1)] — red: [("dog",1)]
  - Spill 2: blue: [("cat",1), ("the",1)] — red: [("hello",1)]
  - Spill 3: blue: [("the",1)] — red: [("dog",1), ("hello",1)]

- Spill files are merged into one output file:
  - Blue partition: [("apple",1), ("cat",1), ("the",1), ("the",1), ("the",1)]
  - Red partition: [("dog",1), ("dog",1), ("hello",1), ("hello",1)]

- Reducer 0 fetches all blue partitions from every Mapper. Reducer 1 fetches all red partitions.

Moving Data From Mappers to Reducers (Cont.)

- Each Reducer merges the pre-sorted partitions it receives from different Mappers. The sorted file on the Reducer is then made available for its Reduce calls.

- The iterator for accessing the Reduce function's list of input values simply iterates over this sorted file.

Moving Data From Mappers to Reducers (Cont.)

- Each Reducer pulls its partition from every Mapper via HTTP.
- These partitions are already sorted by key (the Mapper did that). The Reducer merge-sorts them into a single sorted stream.
- The Reduce function's iterator simply scans forward through this sorted file. Every time the key changes, a new Reduce call begins.
  - This is why Reduce calls within a task happen in key order.

Backup-Task Optimization

- Toward the end of the Map phase, the application master might create backup Map tasks to deal with machines that take unusually long for the last in-progress tasks.
  - Recall that the Reducers cannot start their work until all Mappers have finished. Hence even a single slow Map task would delay the start of the entire Reduce phase.
  - Backup tasks, i.e., “copies” of a task that are started while the original instance of the task is still in progress, can address delays caused by slow machines.
- The same optimization can be applied toward the end of the Reduce phase.

Backup-Task Optimization

- Will duplicate tasks jeopardize the correctness of the job result? No, they are handled by the MapReduce system. We will soon see how, when discussing failures.

- Side note: Sometimes the execution log of your MapReduce job will show error messages indicating task termination.
  - Some of those may have been caused by backup task optimization when duplicate tasks were actively terminated by the application master.

Handling Failures

- Since MapReduce was proposed for Warehouse-Scale Computers, it must deal with failures as a common problem.
  - The programmer does not need to write code for dealing with slow responses or failures of machines in the cluster.

- When failures happen, typically the only observable effect is a moderate delay in program execution due to re-assignment of tasks from the failed machine.
  - In Hadoop, hanging tasks are detected through timeout. The application master will automatically re-schedule failed tasks.

Handling Worker Failures: Map

- The master pings every worker machine periodically. Workers who do not respond in time are marked as failed.
- A failed worker's in-progress and completed Map tasks are reset to idle state and become available for re-assignment to another worker.
  - Why do we need to re-execute a completed Map task? Mappers write their output to the local file system. When the machine that executed this Map task fails, the local file system is considered inaccessible and hence the result must be re-created on another machine.
- Reducers are notified by the master about the Mapper failure, so that they do not attempt to read from the failed machine.

Handling Worker Failures: Reduce

- A failed worker's in-progress Reduce tasks are reset to idle state and become available for re-assignment to another worker machine. There is no need to restart *completed* Reduce tasks.
  - Why not? Reducers write their output to the distributed file system. Hence even if the machine that produced the data fails, the file is still accessible.

Handling Master Failure

- The application master rarely fails, because its a single process. In contrast, the probability of experiencing at least one failure among 100 or 1000 worker processes is higher.

- The simplest way of handling a master failure is to abort the job. Users re-submit aborted jobs once the new master process is up. This works well if the master's mean time to failure is significantly greater than the execution time of most MapReduce jobs

Handling Master Failure

- Example: Assume the master has a mean time to failure of 1 week.
- Jobs running for a few hours are unlikely to experience a master failure.
- However, a job running for 2 weeks has a high probability of suffering from a master failure.
  - Abort-and-resubmit would not work well for such jobs, because the newly re-started instance of the job is likely to experience a master failure as well.

Handling Master Failure

- As an alternative to abort-and-resubmit, the master could write periodic checkpoints of its data structures so that it can be re-started from a checkpointed state.

- Long-running jobs would pick up from this state, instead of starting from scratch. The downside of checkpointing is the higher cost during normal operation, compared to abort-and-resubmit.

Program Semantics with Failures

- If Map, Combine, and Partitioner functions are deterministic, MapReduce guarantees that the output is the same as a failure-free sequential execution — no matter how many failures occur.
- This relies on atomic commits:
  - Tasks write output to a private temp file.
  - A Map task reports its temp file to the application master upon completion. If the task was already completed (e.g., due to backup-task optimization), the master ignores the duplicate.
  - A Reduce task atomically renames its temp file to a final output file in the distributed file system.
- Takeaway: if your functions are deterministic, you never have to think about failures affecting correctness. MapReduce handles it.

We now discuss general principles for approaching a MapReduce programming problem, starting with the programming model's expressiveness. In future modules we will see a variety of concrete examples.

Programming Model Expressiveness

- The MapReduce programming model might appear limited, because it relies on only a handful of functions and seems to have a comparably rigid structure for transferring data between worker nodes.

- However, it turns out that MapReduce is as powerful as the host language (e.g., Java) in which the programmer implements Map and Reduce functions.

Programming Model Expressiveness

- To see why, consider the following construction:
  - Consider a function F written in the host language that, given data set D, outputs F(D). We show that we can write a MapReduce program that also returns F(D), no matter the input D.
  - Map function: For input record r from D, emit (X, r).
  - Reduce function: Execute F on the list of input values.
- Since each record r is emitted with the same key X, the Reduce call for key X has access to the entire data set D. Hence it can execute F locally in the Reducer.

Programming Model Expressiveness

- Even though the above construction shows that every function F from the MapReduce host language can be computed in MapReduce, it would usually not result in an *efficient* or *scalable* parallel implementation.

- Our challenge therefore remains to find the best MapReduce implementation for a given problem.

Multiple MapReduce Steps

- Many real problems require a pipeline of MapReduce jobs.
  - We model this as a directed acyclic graph (DAG): nodes are jobs, edges are data dependencies.

- Jobs execute in topological order. Job J2 cannot start until all jobs it depends on have completed.

- This is one motivation for Spark: in MapReduce, each job writes its output to the distributed file system and the next job reads it back. Spark can keep intermediate data in memory across stages, avoiding this repeated disk I/O.

MapReduce Program Design Principles

- Focus on the question how to partition your input data and intermediate results.
- For tasks that can be performed independently on a data partition, use Map.
- For computation requiring data objects from different partitions, use Reduce.
- Sometimes it is easier to start program design with Map, sometimes Reduce.
  - A good rule of thumb is to select intermediate keys and values such that the right objects end up together in the same Reduce function call.

High-Level MapReduce Development Steps

- Write Map and Reduce functions and create unit tests for them.
- Write a driver program and run the job locally.
  - Use a small data subset for testing and debugging on your development machine. Local testing can be performed using Hadoop's local or pseudo-distributed mode.
- Once the local tests succeeded, run the program on the cluster using small input and few machines. Increase data size gradually to determine the relationship between input size and running time.

High-Level MapReduce Development Steps

- Once you have an estimate of the running time on the full input, try it. Keep monitoring the cluster and cancel the job if it takes much longer than expected.
- Profile the code and fine-tune the program.
  - Analyze and think: where is the bottleneck and how do I address it?
  - Choose the appropriate number of Mappers and Reducers.
  - Consider Combiners. (We will discuss an alternative called in-Mapper combining in a future module.)
  - Consider Map-output compression.
  - Explore Hadoop tuning parameters to optimize the shuffle phase between Mappers and Reducers.

MapReduce Summary

- The MapReduce programming model hides details of parallelization, fault tolerance, locality optimization, and load balancing.
- The model is comparably simple, but it fits many common problems. The programmer provides a Map and a Reduce function, if needed also a Combiner and a Partitioner.
- The MapReduce implementation on a cluster scales to 1000s of machines.