Acknowledgements

- **Credits**
  - *Part of the course material is based on slides provided by the following authors*
    - Pietro Michiardi, Jimmy Lin
Basic example: Word count

- Assume to have a large collection of texts
  - e.g., Web pages from the whole Internet

- We would like to count how many times each word is mentioned all over the collection
  - it represents the basis for more complex computations, such as frequencies, pairings, etc

- Assuming that the collection is distributed among N machines, how would you proceed?

In a single machine, the solution is trivial
- final output: [(fog, 3), (winter, 2), (and, 4), ...]

With multiple machines
1. Use the solution for the single machine in each machine
   - intermediate output: [(fog, 3), (winter, 2), (and, 4), ...]
2. Join the results collected from the different machines and produce the final output
   - final output: [(tree, 8), (fog, 13), (cold, 3), (winter, 6), (and, 22), ...]
Divide and Conquer

Word count: pseudo-code

```java
1: class Mapper
2:    method Map(docid a, doc d)
3:       for all term t ∈ doc d do
4:          Emit(term t, count 1)
1: class Reducer
2:    method Reduce(term t, counts [c1, c2, ...])
3:       sum ← 0
4:       for all count c ∈ counts [c1, c2, ...] do
5:          sum ← sum + c
6:       Emit(term t, count sum)
```

- The two computational steps materializes into two methods, Map and Reduce
- MapReduce is then a programming model
- These two methods are included in a framework that takes care of different aspects
Parallel computing: Concerns

- A parallel system needs to provide:
  - Data distribution
  - Computation distribution
  - Fault tolerance
  - Job scheduling

  The execution framework should hide these system-level details
  - Separate the what from the how
  - MapReduce abstracts away the “distributed” part of the system
    - MapReduce is then an execution framework

What is MapReduce

- A programming model:
  - Inspired by functional programming
  - Allows expressing distributed computations on massive amounts of data

- An execution framework:
  - Designed for large-scale data processing
  - Designed to run on clusters of commodity hardware
The Programming Model

MapReduce: Programming model

- MapReduce is a new use of an old idea in Computer Science

- Map: Apply a function to every object in a list
  - Each object (e.g. document) is independent
    - Order is unimportant
    - Maps can be done in parallel
  - The function produces an intermediate result

- Reduce: Combine the intermediate results to produce a final result
What can we do with MapReduce?

- There are several important problems that can be adapted to MapReduce
  - Inverted indexing (web search), graph algorithms (PageRank),...

- The key point is how to design algorithms with the MapReduce programming model
  - We will show some “design patterns”
    - How to transform a problem and its input
    - How to save memory and bandwidth in the system

Data structures

- Key-value pairs are the basic data structure
  - Keys and values can be: integers, float, strings, raw bytes
    - E.g.: for a collection of Web pages, input keys may be URLs and values may be the HTML content
  - They can also be arbitrary data structures

- The design of MapReduce algorithms involves:
  - Imposing the key-value structure on arbitrary datasets
    - E.g.: for a collection of Web pages, input keys may be URLs and values may be the HTML content
  - In some algorithms, input keys are not used, in others they uniquely identify a record
  - Keys can be combined in complex ways to design various algorithms
MapReduce jobs

- The programmer defines a mapper and a reducer as follows:
  - map: \((k_1, v_1) \rightarrow [(k_2, v_2)]\)
  - reduce: \((k_2, [v_2]) \rightarrow [(k_3, v_3)]\)

- A MapReduce job consists in:
  - A dataset, stored on the underlying distributed filesystem, which is split in a number of blocks across machines
  - The mapper, applied to every input key-value pair to generate intermediate key-value pairs
  - The reducer, applied to all values associated with the same intermediate key to generate output key-value pairs

Where the magic happens

- Implicit between the map and reduce phases is a distributed “group by” operation on intermediate keys
  - Intermediate data arrive at each reducer in order, sorted by the key
  - No ordering is guaranteed across reducers

- Output keys from reducers are written back to the distributed filesystem
  - The output may consist of \(r\) distinct files, where \(r\) is the number of reducers
  - Such output may be the input to a subsequent MapReduce phase

- Intermediate keys are transient:
  - They are not stored on the distributed filesystem
  - They are “spilled” to the local disk of each machine in the cluster
A Simplified view of MapReduce

The Execution Framework
MapReduce: Execution framework

- MapReduce program, a.k.a. a job:
  - Code of mappers and reducers
  - Code for combiners and partitioners (optional)
  - Configuration parameters
  - All packaged together

- A MapReduce job is submitted to the cluster
  - The framework takes care of everything else
  - Next, we will delve into (some) details

Scheduling

- Each Job is broken into tasks
  - Map tasks work on fractions of the input dataset, as defined by the underlying distributed filesystem
  - Reduce tasks work on intermediate inputs and write back to the distributed filesystem

- The number of tasks may exceed the number of available machines in a cluster
  - The scheduler takes care of maintaining something similar to a queue of pending tasks to be assigned to machines with available resources

- Jobs to be executed in a cluster requires scheduling as well
  - Different users may submit jobs
  - Jobs may be of various complexity
  - Fairness is generally a requirement
Data/code co-location

- How to feed data to the code
  - In MapReduce, this issue is intertwined with scheduling and the underlying distributed filesystem

- How data locality is achieved
  - The scheduler starts the task on the node that holds a particular block of data required by the task
  - If this is not possible, tasks are started elsewhere, and data will cross the network
    - Note that usually input data is replicated
  - Distance rules help dealing with bandwidth consumption
    - Same rack scheduling

Synchronization

- In MapReduce, synchronization is achieved by the “shuffle and sort” barrier
  - Intermediate key-value pairs are grouped by key
  - This requires a distributed sort involving all mappers, and taking into account all reducers
  - If you have m mappers and r reducers this phase involves up to m \times r copying operations

- IMPORTANT: the reduce operation cannot start until all mappers have finished
  - This is different from functional programming that allows “lazy” aggregation
  - In practice, a common optimization is for reducers to pull data from mappers as soon as they finish
Errors and faults

The MapReduce framework deals with:

- **Hardware failures**
  - Individual machines: disks, RAM
  - Networking equipment
  - Power / cooling
- **Software failures**
  - Exceptions, bugs
- **Corrupt and/or invalid input data**

Programming model: Optimizations
Local aggregation

- In the context of data-intensive distributed processing, the most important aspect of synchronization is the **exchange of intermediate results**
  - This involves copying intermediate results from the processes that produced them to those that consume them
  - In general, this involves **data transfers over the network**
  - In Hadoop, also disk I/O is involved, as intermediate results are written to disk

- Network and disk latencies are expensive
  - Reducing the amount of intermediate data translates into algorithmic efficiency

- Combiners and preserving state across inputs
  - Reduce the number and size of key-value pairs to be shuffled

Combiners

- Combiners are a general mechanism to reduce the amount of intermediate data
  - They could be thought of as “mini-reducers”

- Back to our running example: word count
  - Combiners aggregate term counts across documents processed by each map task
  - If combiners take advantage of all opportunities for local aggregation we have at most \( m \times V \) intermediate key-value pairs
    - \( m \): number of mappers
    - \( V \): number of unique terms in the collection
  - Note: due to Zipfian nature of term distributions, not all mappers will see all terms
Combiners: an illustration

In general, the code is very similar to the reducer’s code
- sometimes it is possible to use the reducers themselves
  - but this is not always true

The execution of the combiners is not under control of the programmer
- e.g., when the combiners are called
In-Mapper Combiners

- In-Mapper Combiners, a possible improvement

- Use an associative array to cumulate intermediate results
  - The array is used to sum up term counts within a single document
  - The Emit method is called only after all InputRecords have been processed

- Example (see next slide)
  - The code emits a key-value pair for each unique term in the document

In-Mapper Combiners: example

```java
1: class MAPPER
2:   method MAP(docid a, doc d)
3:     H ← new AssociativeArray
4:     for all term t ∈ doc d do
5:       H{t} ← H{t} + 1 // Tally counts for entire document
6:     for all term t ∈ H do
7:       Emit(term t, count H{t})
```
In-Memory Combiners

- Taking the idea one step further
  - Exploit implementation details in Hadoop
  - A Java mapper object is created for each map task
  - JVM reuse must be enabled

- Preserve state within and across calls to the Map method
  - Initialize method, used to create a across-map persistent data structure
  - Close method, used to emit intermediate key-value pairs only when all map task scheduled on one machine are done

---

In-Memory Combiners: example

1: class MAPPER
2:   method INITIALIZE
3:     H ← new ASSOCIATIVEARRAY
4:   method MAP(docid a, doc d)
5:     for all term t ∈ doc d do
6:       H{t} ← H{t} + 1
7:   method CLOSE
8:     for all term t ∈ H do
9:       Emitted(term t, count H{t}) → Tally counts across documents
In-Memory Combiners: Considerations

- **Precautions**
  - In-memory combining breaks the functional programming paradigm due to state preservation
  - Preserving state across multiple instances implies that algorithm behavior might depend on execution order
    - Ordering-dependent bugs are difficult to find

- **Scalability bottleneck**
  - The in-memory combining technique strictly depends on having sufficient memory to store intermediate results
    - And you don’t want the OS to deal with swapping
  - Multiple threads compete for the same resources