Chapter 12: Data-intensive processing using MapReduce

Chapter 12: outline

- Introduction
  - Motivation
  - Big ideas
- MapReduce
  - Programming model
  - Framework
  - Design patterns for MapReduce
    - In-mapper combining
    - Pairs versus stripes
    - Order reversal
    - Value-to-key conversion

Chapter 12: outline

- Introduction
  - Motivation
  - Big ideas
- MapReduce
  - Programming model
  - Framework
  - Design patterns for MapReduce
    - In-mapper combining
    - Pairs versus stripes
    - Order reversal
    - Value-to-key conversion

Introduction

- **MapReduce** is a **programming model** and **execution framework**
  - Used for processing **massive** amounts of **data** ("web-scale")…
  - … in a **distributed** way
  - Originally developed by **Google**
  - **Hadoop**: open-source implementation (Java)
- What is "**big data**" ?
  - *Google* processes 100 TByte/day (2004) – 20 PByte/day (2008)
  - *Ebay* keeps track of 170 trillion records (~ 6.5 PByte)
  - *Facebook* analyses 2.5 PByte of user data (grows with 15 Tbyte/day)
  - *Large Hadron Collider* (Cern): 15 PByte / year.
  - *EMBL*: DNA sequencing database: 5 PByte (2009)

Material reproduced from Lin and Dyer

MapReduce is a framework that is specially designed to handle very large datasets ("Big Data problems"). It provides solutions to problems such as:

- Moving TBytes of data from central storage to a cluster of worker nodes is costly
- How do we deal with the fact that a (small) fraction of TBytes of input data is probably corrupt / incorrectly formatted?
- How do we deal with node failure? We do not want to recompute everything in case a single node fails
- Worker nodes cannot keep TBytes of data in their RAM, we should hence rely on disk I/O.
- etc.

These problem are not trivial to solve using e.g. the Message Passing Interface. In any case, an implementation in MPI that deals with the above points will likely be very complex, even tough the computational task itself may be very simple. MapReduce provides a framework (a piece of software) and a programming model (a methodology to solve a problem) to solve these issues without burden for the programmer.

Motivation (why ?)

- Improvements in **capacity** are outpacing **bandwidth**
  - Capacity: has improved **order of magnitude** of the past decades
  - Bandwidth of commodity networks: **50x**
  - Latency of commodity networks: **2x**
- Why such **large amounts of data**
  - Successfully retrieving information from data depends on
    - **Algorithms**
    - **Features**
    - **Data itself**
  - Growing evidence that having **more data and simple models** outperforms complex features/algorithms with less data
  - This is called "**Data-driven philosophy**"

Material reproduced from Lin and Dyer

Seemingly simple questions like:

- what fraction of consumers in in supermarket X that buy product Y also buy Z ?
- given the fact that you liked movies X, Y and Z, what other movies can we recommend for you?
- etc.

can be solved by computing very simple correlation metrics. The real difficulty lies in the fact the raw datasets can be huge, e.g., most supermarkets now collect, for all of their customers, all products that were bought. Over time, these datasets grow extremely big and they cannot trivially be queried, ever for simple questions.

Motivation (how ?)

- Related to "**cloud**" computing
  - Cloud = the *current* buzz-word
  - Includes web-based apps that store user-generated data
  - **Major idea (1):** "Utility computing" (like water, gas, electricity, …)
    - Frees users from upfront investments
    - Elasticity (computer power on demand, when needed)
  - **Major idea (2):** "Everything as a service"
    - Infrastructure (IaaS)
    - Platform (PaaS), e.g. Google App Engine
    - Software (SaaS)
- Link to **MapReduce**:
  - Just as "clouds", MapReduce tries to represent the right level of abstraction: separate the "**what**" from the "**how**"

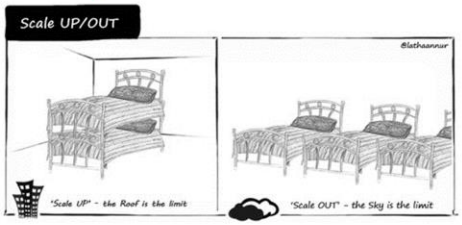
Material reproduced from Lin and Dyer

Chapter 12: outline

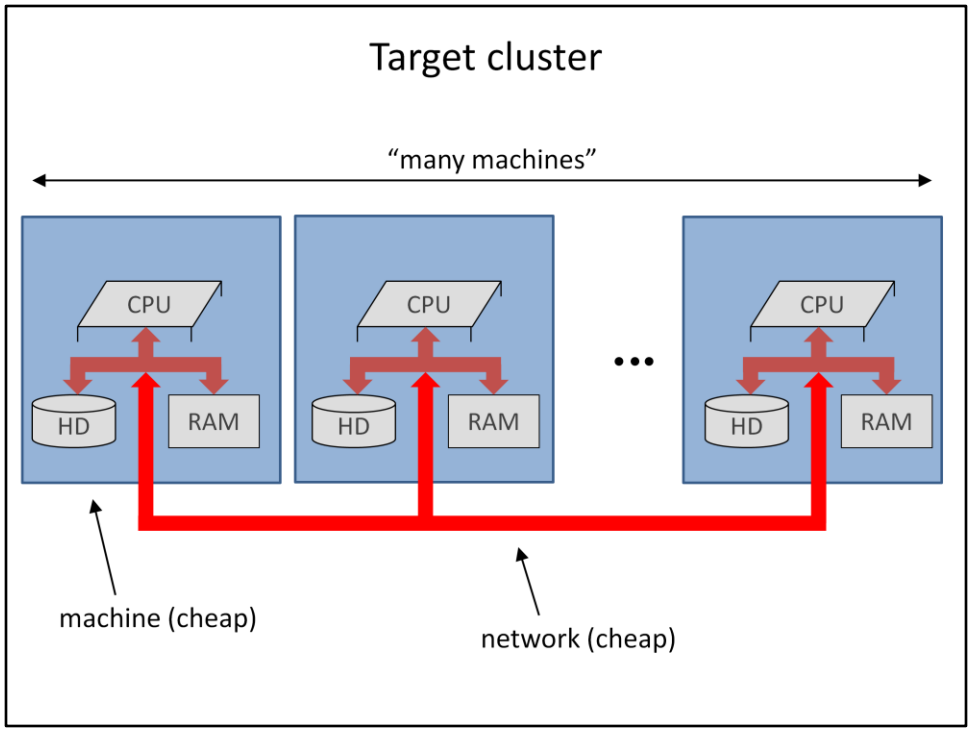
- Introduction
  - Motivation
  - Big ideas
- MapReduce
  - Programming model
  - Framework
  - Design patterns for MapReduce
    - In-mapper combining
    - Pairs versus stripes
    - Order reversal
    - Value-to-key conversion

Big ideas of MapReduce (1)

- **Scale "out", not "up"**



  - Prefer a **larger number of low-end workstations** over a smaller number of high-end workstations (for big-data problems)
    - Buying low-end workstations ("desktop-like machines") is 4-12x as cost efficient as buying high-end workstations ("high-memory SMP machines with a high number of CPUs / cores").
  - Idea is also valid for **network interconnects**
    - E.g. Infiniband versus "commodity" Gigabit Ethernet
    - True HPC applications do require fast networks though…



In MapReduce, it crucially important that each worker node has fast access to hard disk space as both input and output needs to be read from / written to disk (as opposed to the HPC approach of keeping data in memory as much as possible). The cheapest solution is by simply providing one or more local disks per node. See further information on the Hadoop Distributed File System (HDFS).

Big ideas of MapReduce (2)

- **Assume failures are common**
  - Example: 10,000-server cluster, MTBF = 1000 days
    - Expected value of **10 failures a day**
    - Restarting a large-scale job every time a node fails? **NO**
  - **Ideas**:
    - In case of **node failure**,
      - only the work attributed to that node should be recomputed
      - remaining work should be redistributed among other nodes
    - In case a node is **repaired/replaced**
      - node should be able to rejoin an existing job

Material reproduced from Lin and Dyer

MTBF = Mean time before failure.

Big ideas of MapReduce (3)

- **"Move processing to the data"**
  - In a **supercomputer**
    - "compute nodes" are typically linked to "storage nodes" through a high-capacity interconnect
  - In **big data** problems
    - moving this data itself becomes a bottleneck
    - MapReduce assumes a system where **storage and processing power are co-located**
    - In MapReduce: data is located on the local disk of the compute node (requires a distributed file system – see further)
    - Try to assign computations to the nodes that contain all necessary data

Material reproduced from Lin and Dyer

The idea of "moving processing to the data" is important for MapReduce. A certain (sub)task will preferentially be executed by a worker node that contains the input data on local disk. This again differs from the HPC approach of copying the input data to a worker node prior to doing the actual computations. In MapReduce, the (hard disks in the) worker nodes effectively serve as permanent data storage system, rather than a temporary scratch file system.

Big ideas of MapReduce (4)

- **Process data sequentially and avoid random access**
  - **Example**: 1 Terabyte database: 10^{10} 100-byte records
    - Disc latency: ~10 ms
    - Disc throughput: ~100 MByte /s
    - Updating 1% using random access
      – $10^8 \times 10^{-2}$ (latency only) = **10^6 seconds**
    - Reading and rewriting the entire database
      – $2 \times 10^{12} \times 10^{-8}$ = **2.10^4 seconds**

Big ideas of MapReduce (5)

- **Hide system-level details from the application developer**
  - Parallel programming is *difficult*
  - Code runs in *unpredictable* order
  - Race conditions, deadlocks, livelocks, …
  - Main idea is to be able to program the "**what**" (i.e. the functionality of the code) and hide the "**how**" (i.e. the low-level details)

Responsibility of the programmer

Responsibility of the framework

Material reproduced from Lin and Dyer

Big ideas of MapReduce (6)

- **Scalability of the framework**
  - **Ideal algorithm**:
    - Given twice the amount of data, the runtime should be (at most) twice as high (algorithm has linear complexity)
    - Doubling the number of resources, the runtime should be roughly halved
  - For **complex algorithms**, this is usually unobtainable
    - Require true HPC computing environments to get reasonable speedups
  - For **data-driven problems**, MapReduce will effectively scale to very large volumes / very large number of CPUs

Material reproduced from Lin and Dyer

Chapter 12: outline

- Introduction
  - Motivation
  - Big ideas
- MapReduce
  - Programming model
  - Framework
  - Design patterns for MapReduce
    - In-mapper combining
    - Pairs versus stripes
    - Order reversal
    - Value-to-key conversion

MapReduce basics

- **MapReduce** programming model
  - **Divide & conquer strategy**
    - **divide**: partition dataset into smaller, independent chunks to be processed in parallel (= "map")
    - **conquer**: combine, merge or otherwise aggregate the results from the previous step (= "reduce")
  - Roots in **functional programming** (e.g. Lisp)
  - Operates on **<key, value>** pairs
    - Web-based example: key = URL ; value = webpage
    - Graph-based example: key = nodes ; value = adjacency list
  - Users specifies two functions:

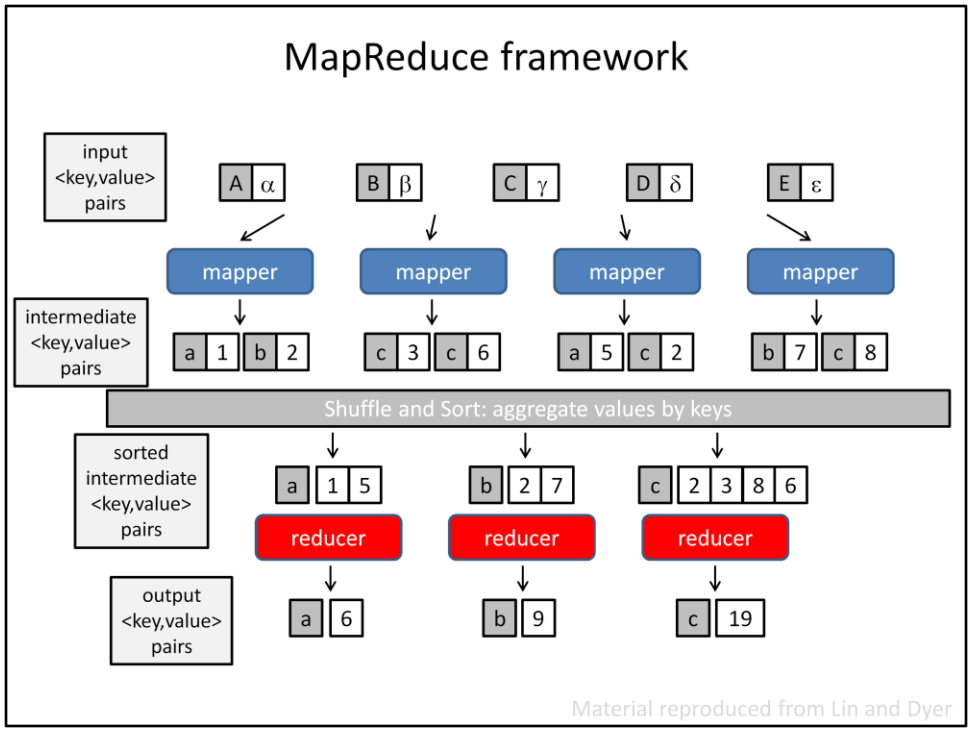
$$\textbf{map:}\ (k_1, v_1) \rightarrow \text{list}[k_2, v_2]$$

$$\textbf{reduce:}\ (k_2, \text{list}[v_2]) \rightarrow \text{list}[k_3, v_3]$$

  - Sorting of intermediate keys between map and reduce phase

Material reproduced from Lin and Dyer

The input "key" for the map function is typically not used. It can be metadata like the filename, a record identifier, and so on. The value is a part of the actual input data, e.g. a line in a file, a record and so on. The map function (user-defined) transforms a certain input <key, value> pair to zero or more **intermediate** <key, value> pairs, where both "key" and "value" can be arbitrary, user-defined objects (a string, an integer, a composite structure, etc.). **All** intermediate <key, value> pairs are sorted (on disk and in parallel) by the MapReduce framework according to key, grouping values that belong to the same key. One key and the list of all corresponding values then serve as the input for the reduce function (user-defined) which further processes these data and produces zero or more output <key, value> pairs (type is again user-defined).



Mappers run on the worker nodes and execute map tasks by applying the map function to a part of the input data. Different map tasks can be executed in parallel. This means that there can be no dependencies between different map tasks. In other words, it should be possible to process part of the input data without having access the remaining parts.

Reducers run on the worker nodes and execute reduce tasks by applying the reduce function on a single intermediate key and its corresponding values. Different reduce tasks can be executed in parallel. This means that there can be no dependencies between different reduce tasks. In other words, it should be possible to process a certain intermediate key and all of its corresponding values without having access to other intermediate keys and values.

MapReduce basics

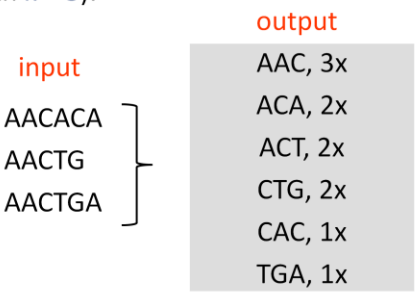
- **Mappers and reducers**
  - objects that implement "`map`" and "`reduce`" methods
  - one mapper object per **map "task"**
    - map task processes a subset of the **input** <key, value> pairs
  - one reduce object per **reduce "task"**
    - reduce task processes a subset of the **intermediate** pairs
- **`map` function**
  - Operates on a **single input pair**
  - Outputs a list of **intermediate pairs**
- **`reduce` function**
  - Operates on a **single key and all of its values**
  - Outputs of list of **resulting** <key, value> pairs

"Mapper" and "Reducer" is the name of two classes within the MapReduce framework. Within these classes, users should implement a map() and reduce() function, respectively.

Multiple instances ("objects") of these classes are created. These mapper/reducer objects run, in parallel, on the worker nodes and process map tasks and reduce tasks.

Example: k-mers in DNA

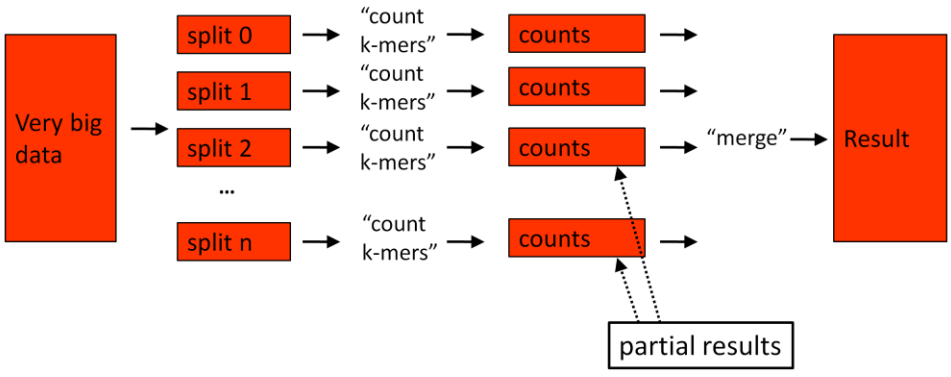
- **Problem**: given a number of DNA sequences, count the occurences of all "words" of length k (**= k-mers**).
- Example (with k = 3):

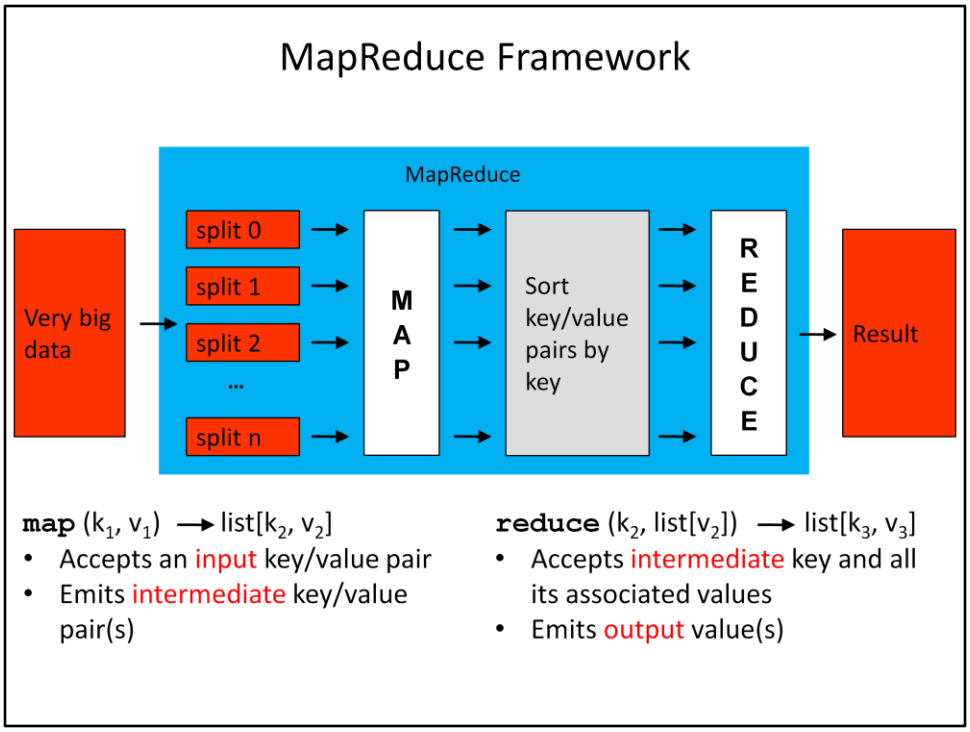


- Now, assume that we have **lots of input** sequences

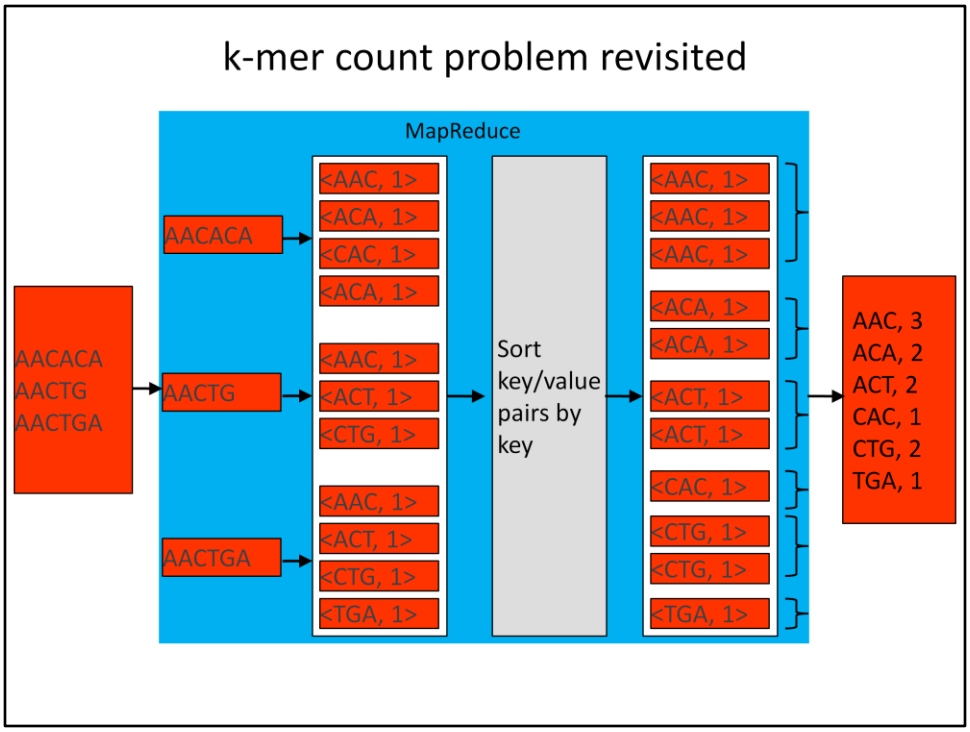
Distributed k-mer count

How would we implement this using e.g. MPI ?





• Map, written by the user, takes an input key/value pair and produces a set of intermediate output key/value pairs.
• MapReduce library groups together all intermediate with the same intermediate key and passes them to the reduce function
• Reduce, also written by the user, merges values that correspond to a specific key.



In this case, the k-mer (e.g. a string) serves as “key” in both intermediate and output <key, value> pairs. An number (e.g. unsigned integer) serves as “value” in both intermediate and output <key, value> pairs. Note that in general, different types for keys/values can be used for intermediate and output <key,value> pairs.

k-mer count in DNA

Pseudocode (**basic implementation**)

```
class Mapper
    method Map(sequenceID id, sequence s)
        for all kmer k ∈ sequence s do
            Emit(kmer k, count 1)
```

```
class Reducer
    method Reduce(kmer k, counts[c1,c2,…])
        sum = 0
        for all count c ∈ counts[c1,c2,…] do
            sum = sum + c
        Emit(kmer k, count sum)
```

Chapter 12: outline

- Introduction
  - Motivation
  - Big ideas
- MapReduce
  - Programming model
  - Framework
  - Design patterns for MapReduce
    - In-mapper combining
    - Pairs versus stripes
    - Order reversal
    - Value-to-key conversion

MapReduce Basics

- **OpenMP / Pthread / MPI: lower-level constructs**
  - Manually break up problems into **smaller subtasks**
  - Explicitly **assign tasks** to workers
  - Ensure that workers get the input **data** they require
  - Explicitly **synchronize** these tasks
  - Explicitly **communicate** (partial) results
  - Not really any support for **fault tolerance**
- **MapReduce framework**
  - Handles the above points "**automatically**"

Material reproduced from Lin and Dyer

MapReduce framework

- **Scheduling**: "**map tasks**" = blocks of input <key, value> pairs
  - #map tasks >> #worker nodes (typically)
  - MapReduce automatically assigns **map tasks to worker** nodes
  - MapReduce automatically assigns **reduce tasks to worker** nodes
  - **Load balance** by keeping track of which tasks are finished
  - Stragglers (tasks that take excessively long) can be difficult to achieve good load balance
- **Data / code co-localization**
  - **Move code to data**: schedule tasks such that few data movements are required
  - Otherwise: **copy data** between worker nodes (slow)
- **Synchronization / communication**
  - **Barrier** between map and reduce phase
  - Efficient **sorting of intermediate** <key, value> pairs
- **Error handling** through job resubmission

Material reproduced from Lin and Dyer

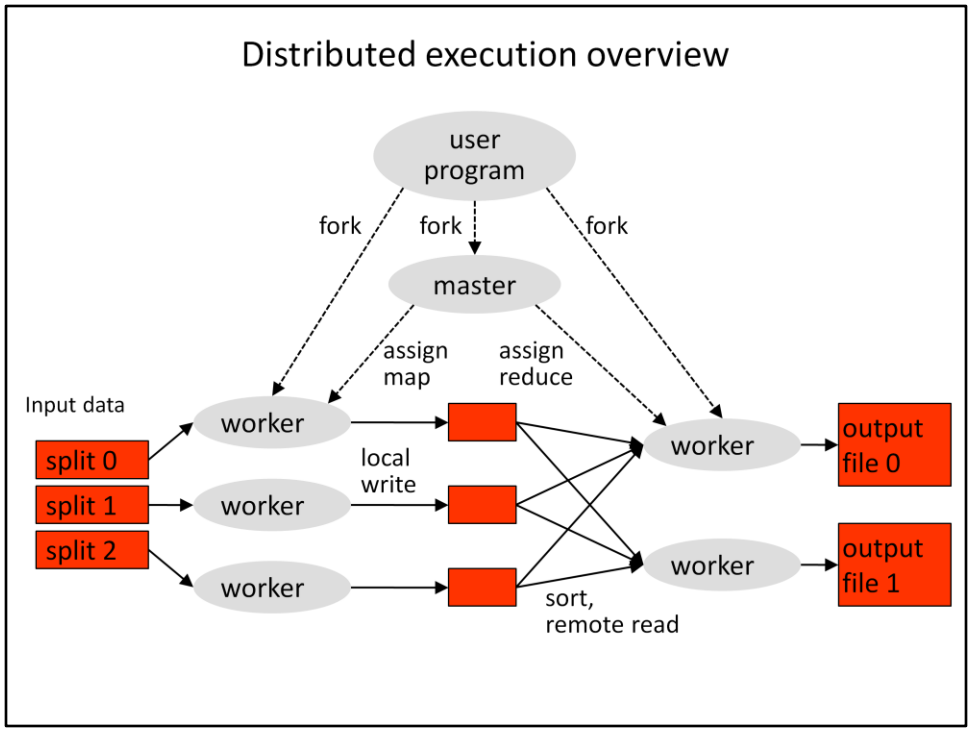
Distributed file system

- **HDFS**: Hadoop Distributed File System
- Master-slave concept
  - Master = **namenode** (keeps track of file names and where data is stored)
  - Slaves = **data nodes** = contain piece of the actual data
- Data nodes are also the **worker nodes** that process map and reduce routines!
- Redundancy concept
  - Each block of data is present in multiple (default=3) data nodes
  - Allows to recover from failures
  - Increases possibilities for improved data locality

Material reproduced from Lin and Dyer

The Hadoop Distributed File System (HDFS) consists of (part of) the local hard disks of the worker nodes. Data stored onto the HDFS is partitioned in blocks (default size = 64 MByte) and each block is redundantly stored on the disk of a number (default = 3) of worker nodes. A "namenode" keeps track of which blocks are stored on which nodes. All worker nodes can read all blocks. In case the block is stored on the same node that is reading the block, this results in (fast) local disk access. In case the block is stored on a different node, this results in (slower) network communication, referred to as "remote reading / writing".

The redundancy in storage of blocks has two advantages: (1) in case a node fails, two copies are still available on the HDFS. The namenode will automatically make a new, third copy of all data that was initially stored on the failed node. (2) When scheduling map tasks, the job scheduler can chose from multiple worker nodes that have the input block on local disk. This facilitates the load balancing.



In MapReduce, one node in the cluster is typically reserved for job scheduling and management (“master” node). The other nodes are “worker” nodes that perform the actual computations in the map / reduce functions. The nodes also serve as storage device, that is, their local hard disks are used to store both the input, intermediate and output data. As the disks of the nodes contain part of the input data (stored on the HDFS), a map task will preferentially be scheduled on the worker node that contains the corresponding input data on local disk, in order to avoid network communication. Intermediate <key, value> pairs, produced by the worker nodes, are also streamed to disk in order to avoid keeping them in RAM (which has limited capacity). The sorting of intermediate <key, value> pairs is performed by the MapReduce framework and obviously relies on network communication. Finally, reduce tasks will also read their input data from disk and write the output to disk.

Chapter 12: outline

- Introduction
  - Motivation
  - Big ideas
- Programming model: map - reduce
  - Basic functionality
  - Framework
  - Design patterns for MapReduce
    - In-mapper combining
    - Pairs versus stripes
    - Order reversal
    - Value-to-key conversion

MapReduce design patterns

- Responsibilities of the **user**
  - Prepare the **data**
  - Implement the **mapper/reducer** (+ combiner/partitioner – see further)
- Rest is handled through the **framework** (no user control!)
- What "**extras**" can be done by the user?
  - **Complex, user defined data types** in <key, value> pairs
  - User-specific **initialization** code in mapper/reducer
  - Ability to **preserve state** across multiple inputs
  - Determine the **sort order of intermediate** terms
  - Control the **partitioning** of the intermediate key space

Material reproduced from Lin and Dyer

MapReduce design patterns

- Many **complex algorithms**
  - Cannot be cast in a **single** MapReduce jobs
  - Rather, a **sequence of jobs** might be required
  - … or even, a **hierarchy** (a mapper calling a new MapReduce job)
  - external (sequential) program ("**driver**") can control and launch MapReduce jobs
- Goals of **design patterns**
  - Provide **optimizations** applicable to many algorithms
  - Ultimate goal: **scalability**
    - If data x 2 -> time x 2
    - If P x 2 -> time / 2

Local aggregation

Word count problem (**basic implementation**)

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)
```

```
class Reducer
    method Reduce(term t, counts[c1,c2,…])
        sum = 0
        for all count c ∈ counts[c1,c2,…] do
            sum = sum + c
        Emit(term t, count sum)
```

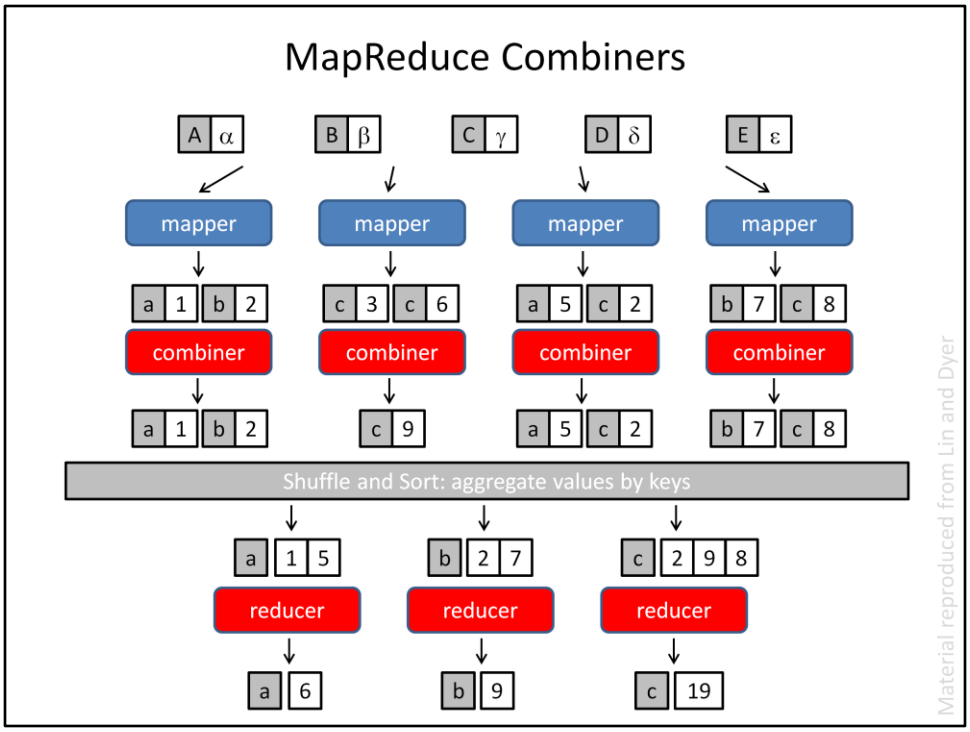
Correct code, however, number of emitted <k, v> pairs is equal to the **total number of terms** in all documents (= huge)

Even though this implementation is correct, it is not really efficient. The number of emitted <key, value> pairs equals the total number of words in a document (key = a word, value = always the number 1). All these data have to written to disk and sorted, which can be time-consuming for extremely large datasets.

Local aggregation

- **Local aggregation**: **reduce** amount of **intermediate <k,v> pairs**
  - While not altering the final results
  - Reduces the amount of network traffic
  - Reduces the amount of disc I/O (pairs are streamed to disc)
  - Alleviates effect of "stragglers"
- **How** ?
  - 1) MapReduce **combiners**
    - method than can optionally be implemented by user
  - 2) "**In-mapper combining**"
    - relies on the ability to preserve state in mapper

Material reproduced from Lin and Dyer



Combiners do **not** reduce the number of intermediate <key, value> pairs emitted by the mappers. They do, however, reduce the number of intermediate <key, value> pairs that have to be **sorted**.

MapReduce Combiners

- **Combiners**
  - Can be provided to the framework as **optimization**
  - Can be thought of a "**mini-reducers**" prior to sorting phase
  - Work on the output of a **single** mapper (locally)
  - Used at the discretion of the framework
    - User cannot force the use of combiners
  - Can sometimes be identical to reducers
    - In the general case, they can be different
  - Alternative: "**in-mapper combining**"

In-mapper combining

"In-mapper combining" ***within*** **a document**

```
class Mapper
    method Map(docid a, doc d)
        H = new AssociativeArray
        for all term t ∈ doc d do
            H{t} = H{t} + 1
        for all term t ∈ H do
            Emit(term t, count H{t})
```

```
class Reducer
    method Reduce(term t, counts[c1,c2,…])
        sum = 0
        for all count c ∈ counts[c1,c2,…] do
            sum = sum + c
        Emit(term t, count sum)
```

The total number of emitted <k, v> pairs equals the number of **unique terms in each document**

Material reproduced from Lin and Dyer

In-mapper combining on the other hand, does reduce the number of intermediate <key, value> pairs by combining certain intermediate <key, value> pairs prior to actually emitting them. One should take care however, that the associative array H can be kept in RAM. If H is close to filling the entire RAM, it should be emptied by emitting all content.

In-mapper combining

In-mapper combining ***within*** **a mapper,** ***across*** **documents**

```
class Mapper
    method Initialize
        H = new AssociativeArray
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} = H{t} + 1
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

```
class Reducer
    method Reduce(term t, counts[c1,c2,…])
        sum = 0
        for all count c ∈ counts[c1,c2,…] do
            sum = sum + c
        Emit(term t, count sum)
```

The total number of emitted <k, v> pairs equals

the number of **unique terms encountered in a mapper**

An more advanced variant of in-mapper combining that relies on the ability of mappers to preserve state. In this case, in-mapper combining is achieved over different map tasks.

In-mapper combining

- **Advantage** over using combiners:
  - Programmer **forces** combining <k,v> pairs
  - Reduces **number of emitted <k,v> pairs by mappers**
    - Reduced overhead from creation, deletion, disc I/O, serialization, etc.
- **Disadvantage** over using combiners:
  - **Memory constraints** of a single node
  - Associative Array needs to be emptied from time to time
    - Every so many elements
    - Or when memory requirements exceed node capacity
- Combiners do not necessarily perform same operations as reducers
  - See next example

Local aggregation correctness

Compute the **mean of values** associated with the same key

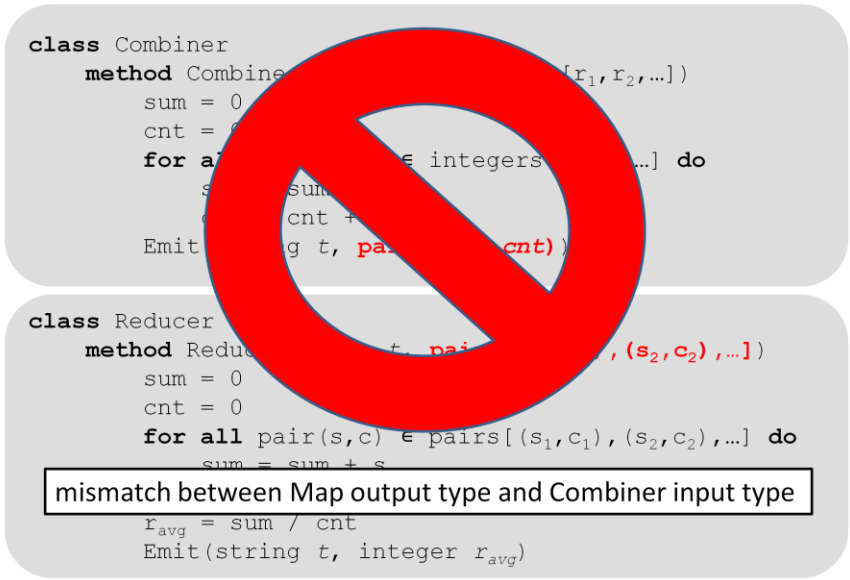
```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)
```

```
class Reducer
    method Reduce(string t, integers[r1,r2,…])
        sum = 0
        cnt = 0
        for all integer r ∈ integers[r1,r2,…] do
            sum = sum + r
            cnt = cnt + 1
        ravg = sum / cnt
        Emit(string t, integer ravg)
```

Cannot use **reducer** as **combiner**:
mean(1,2,3,4,5) ≠ mean(mean(1,2), mean(3,4,5))

Local aggregation correctness

First try at the use of a **combiner** …



```
class Combiner
    method Combine                                  [r1, r2, …])
        sum = 0
        cnt =
        for a                    integers          …] do
            s        sum
                     cnt +
        Emit          g t, pa              cnt)

class Reducer
    method Redu          t, pai                  , (s2, c2), …])
        sum = 0
        cnt = 0
        for all pair(s,c)  pairs[(s1, c1), (s2, c2), …] do
            sum = sum + s
        ravg = sum / cnt
        Emit(string t, integer ravg)
```

mismatch between Map output type and Combiner input type

```
class Mapper
    method Map(string t, integer r)
        Emit(term t, pair(r,1))
```

```
class Combiner
    method Combine(string t, pairs[(s1,c1),(s2,c2),…])
        sum = 0
        cnt = 0
        for all pair(s,c) ∈ pairs[(s1,c1),(s2,c2),…] do
            sum = sum + s
            cnt = cnt + c
        Emit(string t, pair(sum, cnt))
```

```
class Reducer
    method Reduce(string t, pairs[(s1,c1),(s2,c2),…])
        sum = 0
        cnt = 0
        for all pair(s,c) ∈ pairs[(s1,c1),(s2,c2),…] do
            sum = sum + s
            cnt = cnt + c
        ravg = sum / cnt
        Emit(string t, integer ravg)
```

Correct!

In-mapper combining

Alternative: use **in-mapper combining** …

```
class Mapper
    method Initialize
        S = new AssociativeArray
        C = new AssociativeArray
    method Map(string t, integer r)
        S{t} = S{t} + r
        C{t} = C{t} + 1
    method Close
        for all term t ∈ S do
            Emit(term t, pair(S{t}, C{t}))
```

Design pattern 2: “pairs” vs. “stripes”

- Example: Build a **co-occurrence matrix**
  - Given n words in a text
  - **Construct n x n matrix N**, where N_{ij} denotes the number of co-occurences of words w_i and w_j (e.g. w_j followed by w_i)
- **“Pairs” method** (= naïve approach)
  - Map routine emits $<w_i, w_j>$ pairs and 1 as value
  - Reduce count the co-occurrences for each $<w_i, w_j>$ pair
- **“Stripes” method** (= improved approach)
  - Map routine emits “**stripes**” of word counts of w_j associated to w_i

A “stripe” is a collection of objects, e.g. an array.

“Pairs” approach

“Pairs” implementation

```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            for all term u ∈ Neighbors(w) do
                Emit(pair(w,u), count 1)
```

```
class Reducer
    method Reduce(pair p, counts[c1,c2,…])
        sum = 0
        for all count c ∈ counts[c1,c2,…] do
            sum = sum + c
        Emit(pair p, count sum)
```

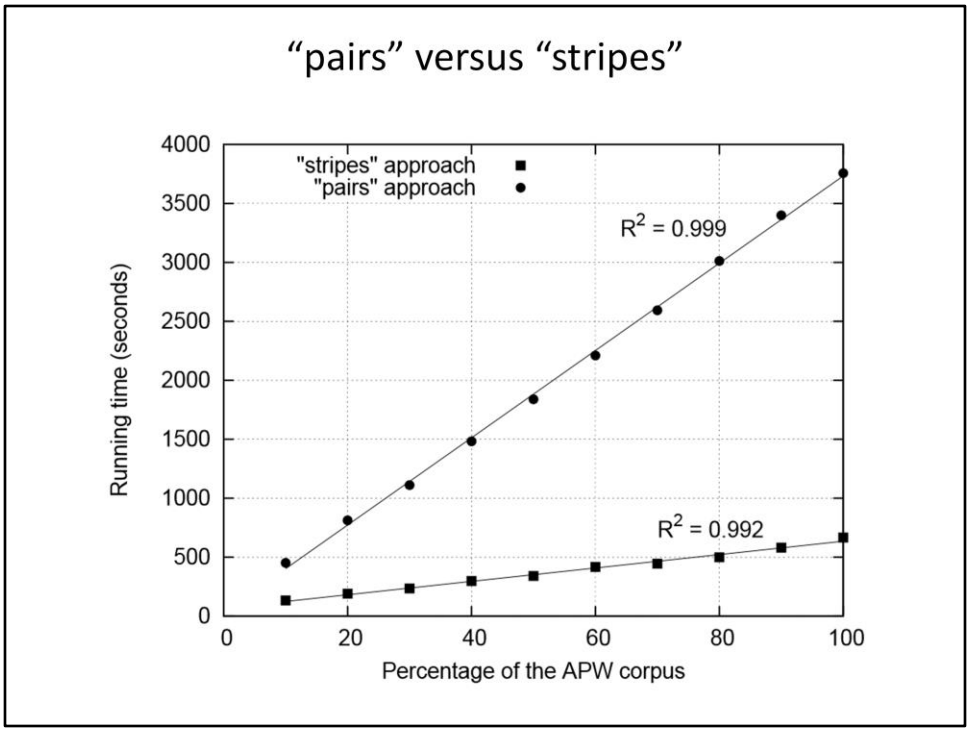
code can be optimized using in-mapper combining

"Stripes" approach

```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            H = new AssociativeArray
            for all term u ∈ Neighbors(w) do
                H{u} = H{u} + 1
            Emit(term w, stripe H)
```

```
class Reducer
    method Reduce(term w, stripes[H1,H2,H3…])
        Hf = new AssociativeArray
        for all stripe H ∈ stripes[H1,H2,H3,…] do
            sum(Hf, H)
        Emit(term w, stripe Hf)
```

Potential memory bottleneck as stripes can be very big



APW = Associated Press Wordstream

Order inversion

- Example: compute **relative frequencies** of co-occurrence matrix

$$f(w_j \mid w_i) = \frac{N(w_i, w_j)}{\sum_{w'} N(w_i, w')}$$

  - Using the **stripes approach**: all needed data is present in the reducer = trivial
    - However, memory bottleneck remains
  - Using the **pairs approach**: not trivial
    - Corresponding pairs <w_i, w_j> are sent to reducer
    - The marginal (= denominator) is not available in the reducer
  - **Idea:**
    - Compute the marginal before processing the joint counts
    - Let mapper emit an **extra <<w_i, *>, 1> pair** for each <w_i, w_j> pair
    - Summing over <w_i, *> yields the marginal
    - If **<w_i, *> pairs are "smaller"** than the regular <w_i, w_j> pairs, we can indeed compute the marginal before handling joint counts

Order inversion

Mapper code:

```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            for all term u ∈ Neighbors(w) do
                Emit(pair(w,u), count 1)
                Emit(pair(w,*), count 1)
```

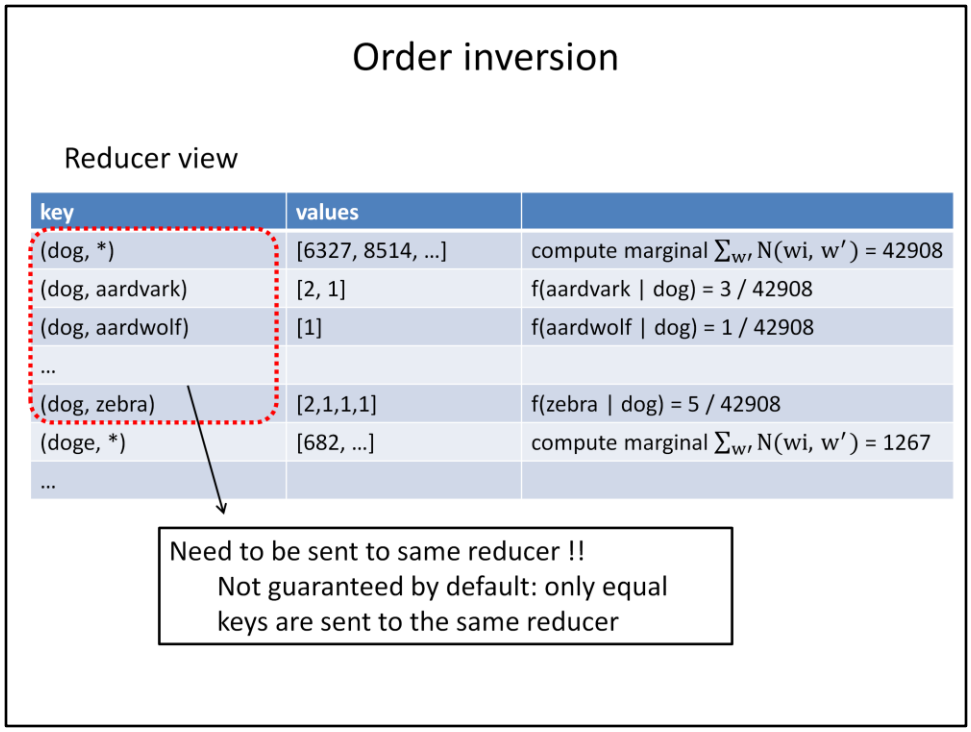
<w, * > is always smaller than any other <w, u>

Optimizations (not shown):

- In-mapper combining

  OR

- MapReduce combiners



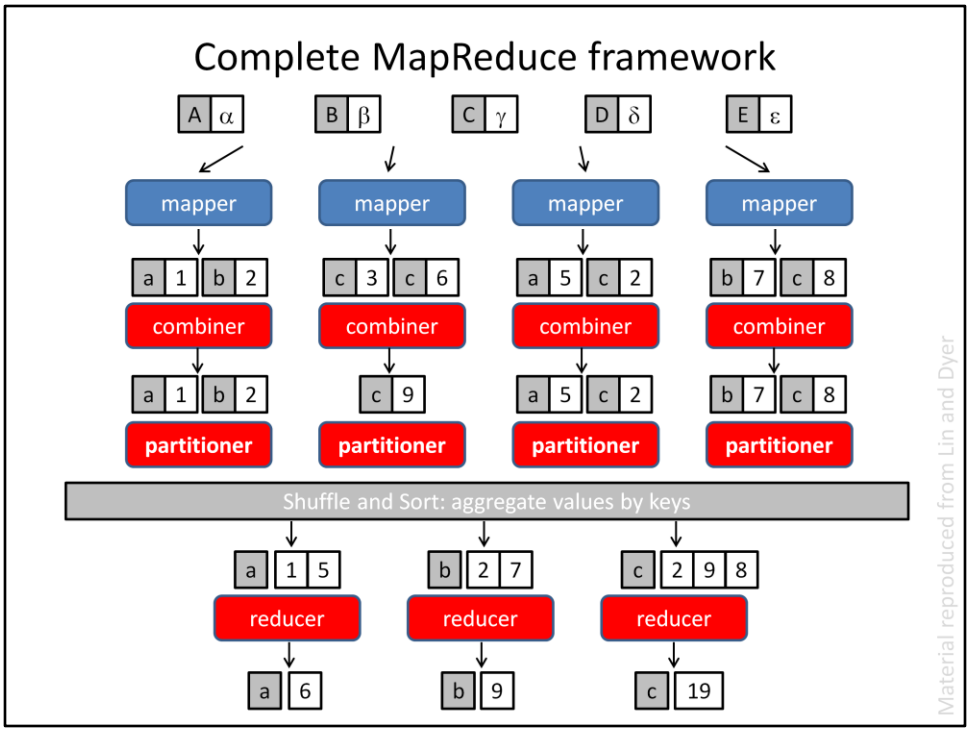
It is important to realize that a certain takes as input only a **single** key and all of its corresponding values. Even if a reducer would first process (dog, *) in order to compute the marginal (denominator) and store this number (reducers can also preserve state across multiple reduce calls), there is no guarantee whatsoever that (dog, aardvark) will be processed by the same reducer later on.

Design patterns: Order inversion

- **Partitioner** determines to which reducer intermediate pairs are sent
  - **Default implementation**:
    - Compute a hash value for each intermediate key and send to reducer *i*

      i = h modulo *#reducers*
  - **Override** this implementation:
    - Such that hash value only depends upon w_i for each $<w_i, w_j>$ pair
    - Thus, all $<w_i, w_j>$ pairs will be sent to the same reducer

A "Partitioner" is a class within the MapReduce framework that controls to which reducer a certain key is being sent. One can override the default implementation to direct certain keys to certain reducers.


Complete MapReduce framework
A α
B β
C γ
D δ
E ε
mapper
mapper
mapper
mapper
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combiner
combiner
combiner
combiner
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partitioner
partitioner
partitioner
partitioner
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reducer
reducer
reducer
a 6
b 9
c 19
Material reproduced from Lin and Dyer

Design patterns: value-to-key conversion

- **Example: sensor data**
  - Large number of sensors s_1, s_2, s_3…
  - Each collect values (raw data) r_x
  - During a large number of time intervals t_1, t_2, …
- **Example log:**

  (t_1, s_1, r_{80521})
  (t_1, s_2, r_{14209})
  (t_1, s_3, r_{76042})
  …
  (t_2, s_1, r_{21823})
  (t_2, s_2, r_{66508})
  (t_2, s_3, r_{98347})
  …
- **Goal**: restructure data per sensor over time (ordered)
  - s_1 : {$[t_1, r_{80521}]$, $[t_2, r_{21823}]$, …}

Design patterns: value-to-key-conversion

- **Simply:**
  emit <s, pair<t, r> >
  will only sort on key (sensor), but not on time

- **Solution 1:**
  Perform "secondary" sorting of values by reducer
  Suffers from scalability bottlenecks
- **Solution 2:**
  emit <pair<s, t> , r>
  Will sort both on sensor and time
  Requires a "partitioner" to assign same sensors to the same reducer

Conclusions

- **MapReduce**
  - Simple programming model for processing large datasets on a large computer cluster.
  - Focus on the problem, let the library deal with the (messy) details.
- **References**
  - Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", proceedings of the OSDI 2004. http://labs.google.com/papers/mapreduce-osdi04.pdf
  - Jimmy Lin and Chris Dyer, "Data-intensive text processing using MapReduce"

The end