MapReduce

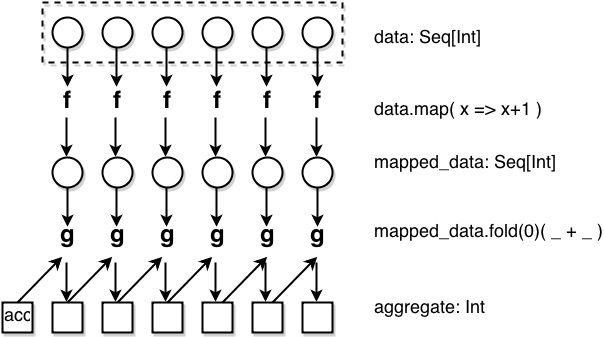
* ***Move Processing to the Data***
* HIgh performance computing model: distinction between processing nodes and storage nodes
* the network becomes the bottleneck not the processors → MapReduce assumes processing and storage nodes to be collocated
* ***Process Data Sequentially and Avoid Random Access***
* Relevant datasets are too large to fit in memory → such data resides on disks
* Disk performance is a bottleneck → organize computation for sequential reads
* ***Data access patterns***
* MapReduce is designed for: batch processing + involving full scans of the data
* data is collected “elsewhere” and copied to the *distributed filesystem*
* ***Hide System-level details***
* MapReduce abstracts away the “distributed” part of the system → handled by the framework
* In-depth knowledge of the framework: custom data reader/writer, custom data partitioning, memory utilization
* ***Scalability***
* In terms of data: given twice the amount of data, the same algorithm should take no more than twice as long to run
* In terms of resources: given a cluster twice the size, the same algorithm should take no more than half as long to run
* Parallelism: independent (shared nothing) computations on fragments of the dataset

The Programming model

***Map phase*** : Given a list, map takes as an argument a function f (that takes a single argument) and applies it to all element in a list

***Fold phase*** : Given a list, fold takes as arguments a function g (that takes two arguments) and an initial value (an accumulator)

* g is first applied to the initial value and the first item in the list
* result is stored in an intermediate variable, which is used as an input together with the next item to a second application of g and so on...



Map → transformation (specified by f) over a dataset:

* each functional application happens in isolation
* application of f to each element can be parallelized

Fold → aggregation operation

* Data locality: elements in the list must be “brought together”
* If we can group elements of the list, also the fold phase can proceed in parallel

The framework coordinates the map and reduce phases: grouping intermediate results happens in parallel.

In practice:

* User-specified computation is applied (in parallel) to all input records of a dataset
* Intermediate results are aggregated by another user-specified computation

***Data Structures***

Key-value pairs = basic data structure in MapReduce

The design of MapReduce algorithms involves:

* imposing the key-value structure on arbitrary datasets (E.g.: for a collection of Web pages : (key = URL, value = HTML content) )
* Keys can be not used or used uniquely to identify a record

***MapReduce Algorithm***

**Map** takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

Map(k1,v1) → list(k2,v2)

The *Map* function is applied in parallel to every pair in the input dataset. This produces a list of pairs for each call. After that, the MapReduce framework collects all pairs with the same key from all lists and groups them together, creating one group for each key.

The *Reduce* function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

Reduce(k2, list (v2)) → list(v3)

A dataset stored on an underlying ***distributed filesystem***, which is split in a number of ***blocks*** across machines.

The mapper is applied to every input key-value pair to generate intermediate key-value pairs.

The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs.

* **"Map" step:** Each worker node applies the "map()" function to the local data, and writes the output to a temporary storage. A master node orchestrates that for redundant copies of input data, only one is processed.
* **"Shuffle" step:** Worker nodes redistribute data based on the output keys (produced by the "map()" function), such that all data belonging to one key is located on the same worker node.
* **"Reduce" step:** Worker nodes now process each group of output data, per key, in parallel.

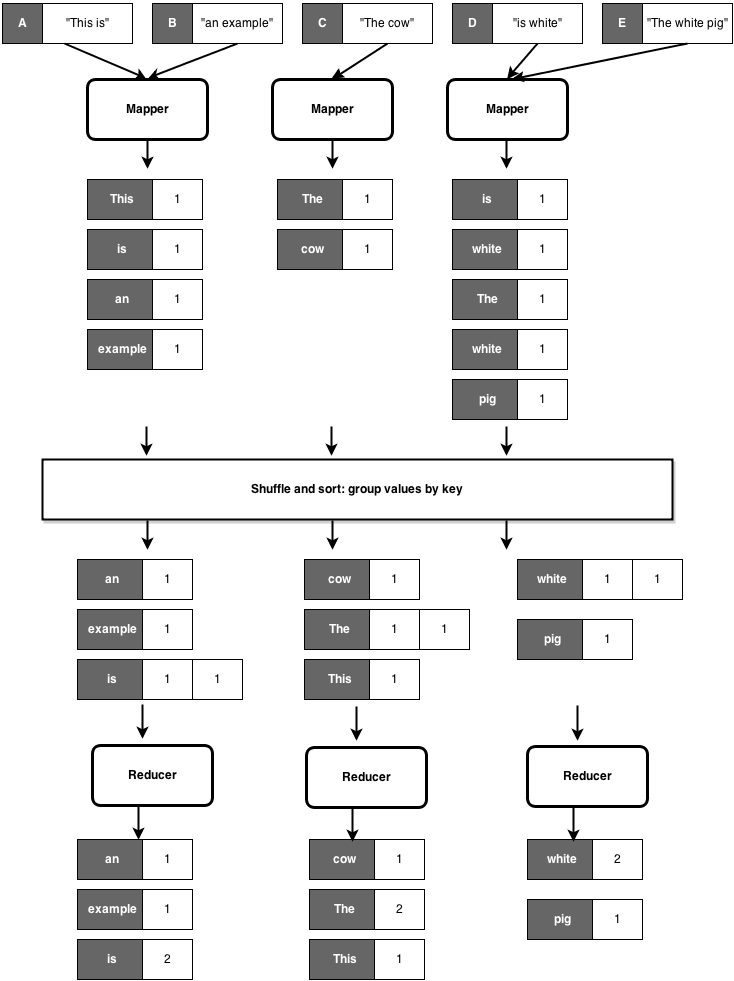
Output keys from reducers are written back to the distributed filesystem. The output may consist of r distinct files, where r is the number of reducers.

Paralellism:   
MapReduce allows for ***distributed processing*** of the map and reduction operations. Provided that each mapping operation is independent of the others, all maps can be performed in parallel – though in practice this is limited by the number of independent data sources and/or the number of CPUs near each source. Similarly, a set of 'reducers' can perform the reduction phase, provided that all outputs of the map operation that share the same key are presented to the same reducer at the same time, or that the reduction function is associative.

The parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled – assuming the input data is still available.

→ 5-step parallel and distributed computation:

1. **Prepare the Map() input** – the "MapReduce system" designates Map processors, assigns the input key value *K1* that each processor would work on, and provides that processor with all the input data associated with that key value.
2. **Run the user-provided Map() code** – Map() is run exactly once for each *K1* key value, generating output organized by key values *K2*.
3. **"Shuffle" the Map output to the Reduce processors** – the MapReduce system designates Reduce processors, assigns the *K2* key value each processor should work on, and provides that processor with all the Map-generated data associated with that key value.
4. **Run the user-provided Reduce() code** – Reduce() is run exactly once for each *K2* key value produced by the Map step.
5. **Produce the final output** – the MapReduce system collects all the Reduce output, and sorts it by *K2* to produce the final outcome.

Example of MapReduce:

**class** MAPPER

**method** MAP (offset a, line l)

for all term t ∈ line l 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)

***Input***:

Key-value pairs: (offset, line) of a file stored on the distributed filesystem

a: unique identifier of a line offset

l: is the text of the line itself

***Mapper***:

Takes an input key-value pair, tokenize the line

Emits intermediate key-value pairs: the word is the key and the integer is the value

***The framework***:

Guarantees all values associated with the same key (the word) are

brought to the same reducer

***The reducer***:

Receives all values associated to some keys

Sums the values and writes output key-value pairs: the key is the

word and the value is the number of occurrences

The frozen part of the MapReduce framework is a large distributed sort. The hot spots, which the application defines, are:

### Input reader

The *input reader* divides the input into appropriate size 'splits' (in practice typically 64 MB to 128 MB) and the framework assigns one split to each *Map* function. The *input reader* reads data from stable storage (typically a distributed file system) and generates key/value pairs.

A common example will read a directory full of text files and return each line as a record.

### Map function

### Partition function

Each *Map* function output is allocated to a particular *reducer* by the application's *partition* function for sharding purposes. The *partition* function is given the key and the number of reducers and returns the index of the desired *reducer*.

A typical default is to hash the key and use the hash value modulo the number of *reducers*. It is important to pick a partition function that gives an approximately uniform distribution of data per shard for load-balancing purposes, otherwise the MapReduce operation can be held up waiting for slow reducers (reducers assigned more than their share of data) to finish.

Between the map and reduce stages, the data is *shuffled* (parallel-sorted / exchanged between nodes) in order to move the data from the map node that produced it to the shard in which it will be reduced. The shuffle can sometimes take longer than the computation time depending on network bandwidth, CPU speeds, data produced and time taken by map and reduce computations.

### Comparison function

The input for each *Reduce* is pulled from the machine where the *Map* ran and sorted using the application's *comparison* function.

### Reduce function

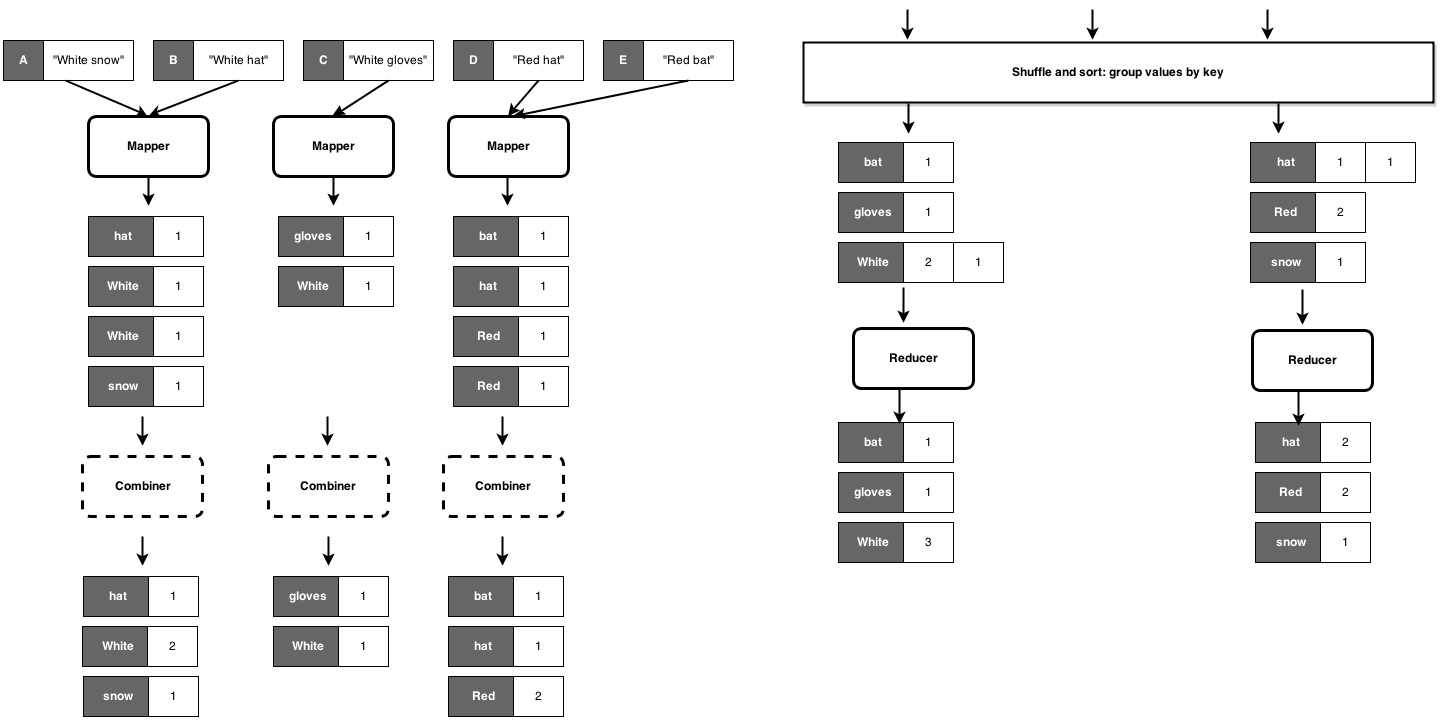
### Output writer

The *Output Writer* writes the output of the *Reduce* to the stable storage.

***Combiners***

When the map operation outputs its pairs they are already available in memory. For efficiency reasons, sometimes it makes sense to take advantage of this fact by supplying a combiner class to perform a reduce-type function. If a combiner is used then the map key-value pairs are not immediately written to the output. Instead they will be collected in lists, one list per each key value. When a certain number of key-value pairs have been written, this buffer is flushed by passing all the values of each key to the combiner's reduce method and outputting the key-value pairs of the combine operation as if they were created by the original map operation.

For example, a word count MapReduce application whose map operation outputs (*word*, 1) pairs as words are encountered in the input can use a combiner to speed up processing. A combine operation will start gathering the output in in-memory lists (instead of on disk), one list per word. Once a certain number of pairs is output, the combine operation will be called once per unique word with the list available as an iterator. The combiner then emits (*word*, count-in-this-part-of-the-input) pairs. From the viewpoint of the Reduce operation this contains the same information as the original Map output, but there should be far fewer pairs output to disk and read from disk.



Basic Design Patterns

***Algorithm Design***

Aspects that are not under the control of the designer

* Where a mapper or reducer will run
* When a mapper or reducer begins or finishes
* Which input key-value pairs are processed by a specific mapper
* Which intermediate key-value pairs are processed by a specific reducer

Aspects that can be controlled

* Construct data structures as keys and values
* Execute user-specified initialization and termination code for mappers and reducers
* Preserve state across multiple input and intermediate keys in mappers and reducers
* Control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys
* Control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer

***Local Aggregation***

In the context of data-intensive distributed processing, the most important aspect of synchronization is the exchange of intermediate results

→ copying intermediate results from the processes that produced them to those that consume them

→ data transfers over the network

→ Hadoop: also disk I/O is involved, as intermediate results are written to disk

**In-Mapper Combiners**

A combiner is a process that runs locally on each Mapper machine to pre-aggregate data before it is shuffled across the network to the various cluster Reducers.

The in-mapper combiner takes this optimization a bit further: the aggregations do not even write to *local* disk: they occur *in-memory* in the Mapper itself.

An in-mapper combiner is much more efficient than a traditional combiner because it continually aggregates the data. As soon as it receives two values with the same key it combines them and stores the resulting key-value pair in a HashMap. However, if there are too many distinct keys, it may run out of memory. To avoid this problem, we use Least Recently Used (LRU) caching. For each incoming key-value pair, we either add it as a new entry in the HashMap, or combine it with the existing entry for that key. If the HashMap grows bigger than the cache capacity, then the least recently used key-value pair will write to the context, and then there will be space in HashMap to store the new incoming key. Finally, in the cleanup method, any remaining entries are written to the context. In contrast, when a mapper with a traditional combiner (the mini-reducer) emits the key-value pair, they are collected in the memory buffer and then the combiner aggregates a batch of these key-value pairs before sending them to the reducer. The drawbacks of this approach are

1. The execution of combiner is not guaranteed; so MapReduce jobs cannot depend on the combiner execution.
2. Hadoop may store the key-value pairs in local filesystem, and run the combiner later which will cause expensive disk IO.
3. A combiner only combines data in the same buffer. Thus, we may still generate a lot of network traffic during the shuffle phase even if most of the keys from a single mapper are the same. To see this, consider the word count example, assuming that buffer size is 3, and we have <key, value> = <Stanford, 3>, <Berkeley, 1>, <Stanford, 7>, <Berkeley, 7>, and <Stanford, 2> emitted from one mapper. The first three items will be in one buffer, and last two will be in the the other buffer; as a result, the combiner will emit <Stanford, 10>, <Berkeley, 1>, <Berkeley, 7>, <Stanford, 2>. If we use in-mapper combiner, we will get <Stanford, 12>, <Berkeley, 8>.

The pseudo code for a basic M/R algorithm which computes average marks is as given:

**class** Mapper

**method** Map(integer s\_id, integer m)

Emit(integer s\_id, integer m)

**class** Reducer

**method** Reduce(integer s\_id, integer [m1,m2,...])

sum ← 0

cnt ← 0

for all integer m ∈ integer [m1,m2,...] do

sum ← sum + m

cnt ← cnt + 1

avg\_m ← sum/cnt

Emit(integer s\_id, float avg\_m )

If we have a large number of input records then the same number of records emitted from map task will be shuffled and sorted before being passed on to reducer. This large amount of data transfer could be deterrent in the speed of execution of overall M/R job.

We can make this algorithm faster by decreasing the number of records emitted by the mapper. To achieve this we can use an associative array to store partial sums of marks, and another associative array to store the count of marks and finally emit these values in close method. The pseudo code for in-mapper combiner is shown below:

**class** Mapper

**method** Initialize

S ← new AssociativeArray

C ← new AssociativeArray

**method** Map(integer s\_id, integer m)

S{s\_id} ← S{s\_id} + m

C{s\_id} ← C{s\_id} + 1

**method** Close

for all integer s\_id ∈ S do

Emit(integer s\_id, pair (S{s\_id}, C{s\_id}))

**class** Reducer

method Reduce(integer s\_id, 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

avg\_m ← sum/cnt

Emit(integer s\_id, float avg\_m )

Using this algorithm we can improve the performance of M/R job by reducing the number of intermediary key-value pairs emitted from mappers to reducers.

**Design Pattern: in-memory combining**

An **in-memory database** is a database management system that primarily relies on main memory for computer data storage. It is contrasted with database management systems that employ a disk storage mechanism. Main memory databases are faster than disk-optimized databases since the internal optimization algorithms are simpler and execute fewer CPU instructions. Accessing data in memory eliminates seek time when querying the data, which provides faster and more predictable performance than disk.

Advantages of in-memory combining:

* Provides control over when local aggregation occurs
* Designer can determine how exactly aggregation is done

Compared to vanilla combiners:

* No additional overhead due to the materialization of key-value pairs  
  With vanilla combiners:  
  → un-necessary object creation and destruction (garbage collection)  
  → serialization, deserialization when memory bounded
* With combiners, mappers still need to emit all key-value pairs; combiners “only” reduce network traffic

Drawbacks of in-memory combining:

* in-memory combining breaks the functional programming paradigm due to state preservation

*Functionnal programming:*In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result f(x) each time. Eliminating side effects, i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

→ preserving state across multiple instances implies that algorithm behavior might depend on execution order

* in-memory combining strictly depends on having sufficient memory to store intermediate results

***Pairs and Stripes***

*Word Co-Occurrence Problem*

Co-occurrence matrix of a corpus is a square n × n matrix where n is the number of unique words in the corpus (i.e., the vocabulary size). A cell mij contains the number of times word wi co-occurs with word wj within a specific context—a natural unit such as a sentence, paragraph, or a document, or a certain window of m words (where m is an application-dependent parameter). Note that the upper and lower triangles of the matrix are identical since co-occurrence is a symmetric relation.

**Pairs approach**The mapper processes each input document and emits intermediate

key-value pairs with each co-occurring word pair as the key and the integer one (i.e.,

the count) as the value. This is straightforwardly accomplished by **two nested loops**:

* the outer loop iterates over all words (the left element in the pair)
* the innerloop iterates over all neighbors of the first word (the right element in the pair).

The neighbors of a word can either be defined in terms of a sliding window or some other

contextual unit such as a sentence.

The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Thus, in this case the reducer simply sums up all the values associated with the same co-occurring word pair to arrive at the absolute count of the joint event in the corpus, which is then emitted as the final key-value pair. Each pair corresponds to a cell in the word co-occurrence matrix. This algorithm illustrates the use of complex keys in order to coordinate distributed computations.

**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 , . . .])

s←0

for all count c ∈ counts [c1 , c2 , . . .] do

s←s+c

Emit(pair p, count s)

**Stripes approach**Like the pairs approach, co-occurring word pairs are generated by **two nested**

**loops**. However, the major difference is that instead of emitting intermediate key-value

pairs for each co-occurring word pair, ***co-occurrence information is first stored in an***

***associative array***, denoted H.

The mapper emits key-value pairs with words as keys and corresponding associative arrays as values, where each associative array encodes the co-occurrence counts of the neighbors of a particular word (i.e., its context).

The MapReduce execution framework guarantees that all associative arrays with the same

key will be brought together in the reduce phase of processing.

The reducer performs an element-wise sum of all associative arrays with the same key, accumulating counts that correspond to the same cell in the co-occurrence matrix. The final associative array is emitted with the same word as the key.

In contrast to the pairs approach, each final key-value pair encodes a row in the co-occurrence matrix.

**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 )

**Comparison of the two approaches**

Pairs:

* Generates a large number of key-value pairs
* The benefit from combiners is limited, as it is less likely for a mapper to process multiple occurrences of a word
* Does not suffer from memory paging problems

Stripes:

* More compact
* Generates fewer and shorted intermediate keys → the framework has less sorting to do
* The values are more complex and have serialization/deserialization overhead
* Greatly benefits from combiners, as the key space is the vocabulary
* Suffers from memory paging problems

**Relative frequencies**

Instead of absolute counts, we take into consideration the fact that some words appear more frequently than others. We need to convert absolute counts to relative frequencies f(wj | wi ).

N() is the number of times a co-occurring word pair is observed

The denominator is called the marginal

Stripes approach:

In the reducer, the counts of all words that co-occur with the conditioning variable (wi ) are available in the associative array. → the sum of all those counts gives the marginal.

We divide the joint counts by the marginal and we’re done.

Pairs approach:

The reducer receives the pair (wi , wj ) and the count → **not possible** to compute f (wj, wi ).

The mapper and the reducer can preserve state across multiple keys:

→ buffer in memory all the words that co-occur with wi and their counts

→ basically building the associative array in the stripes method

***Order Inversion***

The order inversion pattern exploits the sorting phase of MapReduce to push data needed for calculations to the reducer *ahead of* the data that will be manipulated. The **order inversion** pattern is a nice trick that lets a reducer see intermediate results before it processes the data that generated them.

The mapper counts word pairs in the corpus, so its output looks like

((dog, cat), 125)

((dog, foot), 246)...

But it also keeps a running total of all the word pairs containing "dog", outputting this as

((dog,\*), 5348)

Using a suitable partitioner, so that all (dog,...) pairs get sent to the same reducer, and choosing the "\*" token so that it occurs before any word in the sort order, the reducer sees the total ((dog,\*), 5348) first, followed by all the other counts, and can trivially store the total and then output relative frequencies. The benefit of the pattern is that it avoids an extra MapReduce iteration without creating any additional scalability bottleneck.

We must define an appropriate partitioner:

* The default partitioner is based on the hash value of the intermediate key, modulo the number of reducers
* For a complex key, the raw byte representation is used to compute the hash value → no guarantee that the pair (dog, aardvark) and (dog,zebra) are sent to the same reducer
* we want all pairs with the same left word to be sent to the same reducer

The key is to properly sequence data presented to reducers.

The mapper:

* additionally emits a “special” key of the form (wi , \*)
* The value associated to the special key is one, that represents the contribution of the word pair to the marginal
* Using combiners, these partial marginal counts will be aggregated before being sent to the reducers

The reducer:

* We must make sure that the special key-value pairs are processed before any other key-value pairs where the left word is wi
* We also need to modify the partitioner as before, i.e., it would take into account only the first word

Summary of order inversion:

* Emit a special key-value pair to capture the marginal
* Control the sort order of the intermediate key, so that the special key-value pair is processed first
* Define a custom partitioner for routing intermediate key-value pairs
* Preserve state across multiple keys in the reducer