MapReduce

WHAT IS MAPREDUCE

MapReduce is:

- a **PROGRAMMING MODEL**
  - inspired by **Functional Programming**
  - for expressing *distributed* computation on *massive amounts of data* and

- an **EXECUTION FRAMEWORK**
  - designed for *large-scale data* processing
  - designed to run on clusters of commodity servers.

***Commodity computing**, or commodity cluster computing, is the use of large numbers of already-available computing components for parallel computing, to get the greatest amount of useful computation at low cost.*

Why? There are many answers to this question, but we focus on two.

- First, big data is a fact of the world, and therefore an issue that real-world systems must deal with.
- Second, across a wide range of text processing applications, more data translates into more effective algorithms, and thus it makes sense to take advantage of the plentiful amounts of data that surround us.

SOME IDEAS BEHIND MAPREDUCE

1. Scale out, not up

For data-intensive workloads, a large number of commodity low-end servers (i.e., the **scaling out** approach) is preferred over a small number of high-end servers (i.e., the **scaling up** approach). The latter approach is not cost effective, since *the costs of such machines do not scale linearly* (i.e., a machine with twice as many processors is often significantly more than twice as expensive). Therefore, most existing implementations of the MapReduce programming model are designed around clusters of low-end commodity servers.

However, cost in acquiring servers is, of course, only one component of the total cost. Operational costs are dominated by the cost of electricity to power the server as well as other aspects such as power distribution, cooling etc… As a result, **ENERGY EFFICIENCY** has become a key issue in building warehouse-scale computers for large-data processing.
Which are the implications of scaling out?

- **Processing data is quick, I/O is very slow**
- **Sharing vs Shared nothing**
  - Sharing: manage a common/global state
  - Shared nothing: independent entities, no common state (OUR CASE)

- **Sharing is difficult**
    - Synchronization, deadlocks
    - Finite bandwidth to access data from SAN
    - Temporal dependencies are complicated (restart)

2. Failure are the norm, not the exception

At warehouse scale, failures are not only inevitable, but commonplace. Mature implementations of the MapReduce programming model are able to robustly cope with failures through a number of mechanisms such as automatic task restarts on different cluster nodes.

3. Data Locality: Move Processing to the Data

In traditional high-performance computing (HPC) applications (e.g., for climate or nuclear simulations), it is commonplace for a supercomputer to have **PROCESSING NODES** and **STORAGE NODES** linked together by a *high-capacity interconnect*.

Many data-intensive workloads are not very processor-demanding, which means that the separation of compute and storage creates a bottleneck in the network. As an alternative to moving data around, it is more efficient to move the processing around. That is, MapReduce assumes an architecture where processors and storage (disk) are co-located. In such a setup, we can take advantage of **DATA LOCALITY** by running code on the processor directly attached to the block of data we need.

The **DISTRIBUTED FILE SYSTEM** is responsible for managing the data over which MapReduce operates.

4. Process data sequentially and avoid random access

Data-intensive processing by definition means that the relevant datasets *are too large to fit in memory* and must be held on disk. Seek times for random disk access are fundamentally limited by the mechanical nature of the devices: read heads can only move so fast and platters can only spin so rapidly. As a result, it is desirable to avoid random data access, and instead organize computations so that data is processed sequentially.

MapReduce is designed for batch processing (elaborazione a blocchi) involving full scans of the dataset $sequential\ access >> random\ access$.

5. Hide system-level details from the application developer

MapReduce addresses the challenges of distributed programming by providing an abstraction that isolates the developer from system-level details (e.g., locking of data structures, data starvation issues in the processing pipeline, etc.). MapReduce maintains a separation of **what** computations are to be performed and **how** those computations are actually carried out on a cluster of machines.
The advantage is that the execution framework only needs to be designed once.

6. Seamless scalability

For data-intensive processing, it goes without saying that scalable algorithms are highly desirable. As an aspiration, let us sketch the behavior of an ideal algorithm.

We can define scalability along at least two dimensions:

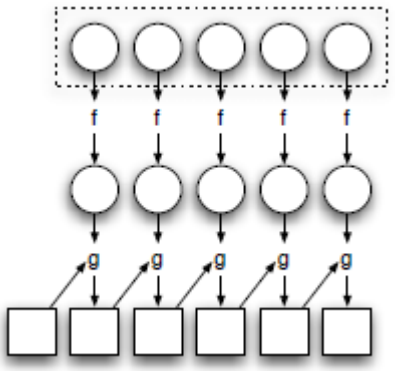
- First, in terms of data: given twice the amount of data, the same algorithm should take at most twice as long to run, all else being equal.
- Second, in terms of resources: given a cluster twice the size, the same algorithm should take no more than half as long to run.

Furthermore, an ideal algorithm would maintain these desirable scaling characteristics across a wide range of settings: on data ranging from gigabytes to petabytes, on clusters consisting of a few to a few thousand machines. Finally, the ideal algorithm would exhibit these desired behaviors without requiring any modifications whatsoever, not even tuning of parameters.
However, other than for embarrassingly parallel problems, algorithms with the characteristics sketched above are, of course, unobtainable. Perhaps the most exciting aspect of MapReduce is that it represents a small step toward algorithms that behave in the ideal manner discussed above.

EQUIVALENCE OF MAPREDUCE AND FUNCTIONAL PROGRAMMING

MapReduce has its roots in functional programming. The key feature of functional languages is the concept of **higher-order functions**, or *functions that can accept other functions as arguments*.
Two common built-in higher order functions are:

- **MAP**
  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**
  Given a list, fold takes as arguments a function g (that takes two arguments) and an initial value g is first applied to the initial value and the first item in the list. The result is stored in an intermediate variable, which is used as an input together with the next item to a second application of g. The process is repeated until all items in the list have been consumed.



We can view **MAP** as a **TRANSFORMATION** over a dataset. This transformation is specified by the function f and happens in *isolation*. In fact, the application of f to each element of a dataset can be parallelized in a straightforward manner.

We can view **FOLD** as an **AGGREGATION** operation. The aggregation is defined by the function g. Unlike the previous case, the fold operation has more restrictions on data locality: elements in the list must be "brought together" before the function g can be applied (in parallel).

In summary, we have described MapReduce.

- The **MAP** phase in MapReduce roughly corresponds to the **MAP** operation in functional programming.
- The **REDUCE** phase in MapReduce roughly corresponds to the **FOLD** operation in functional programming.

As we will discuss in detail shortly, the MapReduce execution framework coordinates the map and reduce phases of processing over large amounts of data on large clusters of commodity machines.

In practice:

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

MAPPERS AND REDUCERS

Data Structures

KEY-VALUE pairs form the basic data structure in MapReduce.

- Keys and values may be primitives such as integers, float values, strings, and raw bytes, or
- They may be arbitrarily complex structures (lists, tuples, associative arrays, etc.).

Part of the design of MapReduce algorithms involves imposing the key-value structure on arbitrary datasets.

- For a collection of web pages, keys may be URLs and values may be the actual HTML content.
- For a graph, keys may represent node ids and values may contain the adjacency lists of those nodes.
- In some algorithms, input keys are not particularly meaningful and are simply ignored during processing.
- In other cases input keys are used to uniquely identify a datum (such as a record id). Keys can be also combined in complex ways to design various algorithms.

A MapReduce job

In MapReduce, the programmer defines a mapper and a reducer with the following signatures:

$$map: \left(k_1, v_1\right) \rightarrow [(k_2, v_2)]$$

$$reduce: \left(k_2, [v_2]\right) \rightarrow [(k_3, v_3)]$$

The convention […] is used to denote a list.

- The input to a MapReduce job starts as data stored on the underlying distributed file system.
- The mapper is applied to every input key-value pair (split across an arbitrary number of files) to generate an *arbitrary* number of intermediate key-value pairs.
- The reducer is 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 arrives at each reducer in order, sorted by the key. However, no ordering relationship is guaranteed for keys across different reducers.

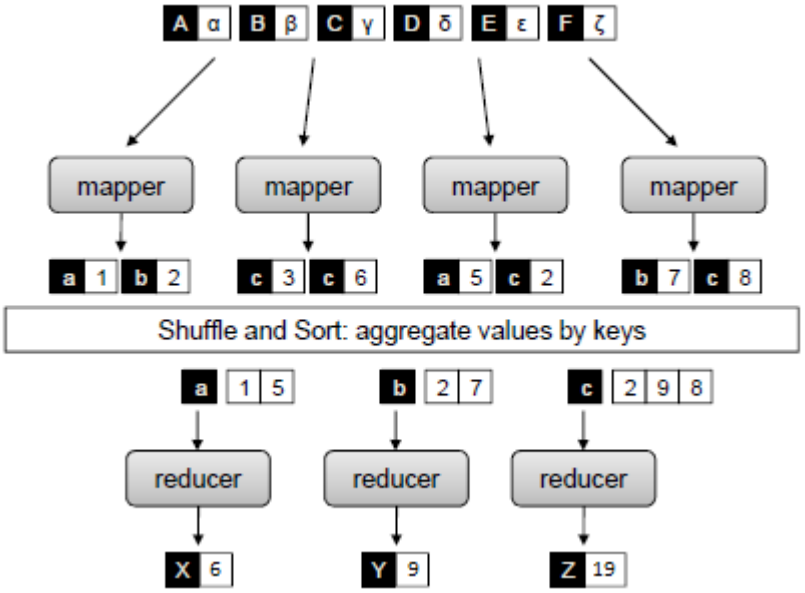
Output key-value pairs from each reducer are written **persistently** back onto the distributed file system (whereas intermediate key-value pairs are transient and not preserved).
The output ends up in r files on the distributed file system, where r is the number of reducers. For the most part, there is no need to consolidate reducer output, since the r files often serve as input to yet another MapReduce job.

An example

A simple word count algorithm in MapReduce is shown in Figure.

This algorithm counts the number of occurrences of every word in a text collection.

Input key-values pairs take the form of (*docid*, *doc*) pairs stored on the distributed file system, where the former is a unique identifier for the document, and the latter is the text of the document itself.



1. The **MAPPER** takes an input key-value pair, tokenizes the document, and emits an intermediate key-value pair for every word: the word itself serves as the key, and the integer one serves as the value (denoting that we've seen the word once).

   $map: \left(k_1, v_1\right) \rightarrow [(k_2, v_2)]$ $(docid, doc) \rightarrow [(word, count)]$

2. The **MAPREDUCE EXECUTION FRAMEWORK** guarantees that all values associated with the same key are brought together in the reducer (every 'a' will go to reducer1 and so on). Therefore, in our word count algorithm, we simply need to sum up all counts (ones) associated with each word. In reality, the figure above is a little bit different. In the first case, for example, we should have a:1, b:1, b:1 instead of a:1, b:2 □ *see* ***COMBINERS***.

3. The **REDUCER** does exactly this, and emits final key-value pairs (only one for each reducer in our case) with the word as the key, and the count as the value. Final output is written to the distributed file system, one file per reducer.

   $reduce: \left(k_2, [v_2]\right) \rightarrow [(k_3, v_3)]$ $(word, [count]) \rightarrow [(WORD, SUM)]$

N.B. Words within each file will be sorted by alphabetical order, ***and each file will contain roughly the same number of words***. The ***PARTITIONER***, which we discuss later, controls the assignment of words to reducers.

The output can be examined by the programmer or used as input to another MapReduce program.

A pseudo code for MAP and REDUCE functions could be:

```
class MAPPER
      method MAP(offset a, docid d_id)
            for all word w belonging to docid d_id do
                  EMIT(word w, count 1);

class REDUCER
      method REDUCE(word w, counts [c1, c2, …])
            sum = 0;
            for all count ci do
                  sum += ci;
            EMIT(word w, sum);
```

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: combiners aggregate term counts across documents processed by each map task.

- **The use of combiners must be thought carefully**
  - The correctness of the algorithm cannot depend on computation (or even execution) of the combiners
- **Combiners I/O types**
  - Input: (k2; [v2]) [Same input as for Reducers]
  - Output: [(k2; v2)] [Same output as for Mappers]
- **Commutative and Associative computations**
  - Reducer and Combiner code may be interchangeable
  - This is not true in the general case (Example: compute the mean of word in some documents)

MapReduce Algorithm Design

A large part of the power of MapReduce comes from its **simplicity**: the programmer needs only to:

- preparing the **INPUT DATA**
- implement the **MAPPER** and the **REDUCER**
- implement, *optionally*, the **COMBINER** and the **PARTITIONER**

All other aspects of execution are handled transparently by the execution framework.
However, this also means that any possible algorithm that a programmer wishes to develop must be expressed in terms of a small number of rigidly-defined components that must fit together in very specific ways. Thus, it may not appear obvious how a multitude of algorithms can be recast into this programming model.

- **Synchronization**
  It is perhaps the most tricky aspect of designing MapReduce algorithms.
  Within a single Map-Reduce job, there is only one opportunity for cluster-wide synchronization - during the shuffle and sort stage where intermediate key-value pairs are copied from the mappers to the reducers and grouped by key. Beyond that, mappers and reducers run in isolation without any mechanisms for direct communication.

- Furthermore, **the programmer has not control over many aspects of execution**:
  - Where a mapper or reducer runs (i.e., on which node in the cluster)
  - 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

- **Nevertheless, the programmer does have a number of techniques for controlling execution and managing the flow of data in MapReduce**.
  In summary, they are:
  - The ability to **construct complex data structures** as keys and values to store and communicate partial results.
  - The ability to execute **user-specified initialization code** at the beginning of a map or reduce task, and the ability to execute **user-specified termination code** at the end of a map or reduce task.
  - The ability to **preserve state in both mappers and reducers** across multiple input or intermediate keys.
  - The ability to **control the sort order of intermediate keys**, and therefore the order in which a reducer will encounter particular keys.
  - The ability to **control the partitioning of the key space**, and therefore the set of keys that will be encountered by a particular reducer.

- **Many algorithms cannot be easily expressed as a single MapReduce job**.
  One must often decompose complex algorithms into a sequence of jobs, which requires orchestrating data so that the output of one job becomes the input to the next.
  Many algorithms are iterative in nature, requiring repeated execution until some convergence criteria. Often, the convergence check itself cannot be easily expressed in MapReduce. The standard solution is an external (non-MapReduce) program that serves as an "**EXTERNAL DRIVER**" to coordinate MapReduce iterations.

Now we are going to study some design patterns.

1. LOCAL AGGREGATION

In the context of data-intensive distributed processing, the single most important aspect of synchronization is the ***exchange of intermediate results***, from the processes that produced them to the processes that will ultimately consume them.

This necessarily involves:

- ***transferring data over the network*** (**NETWORK DELAYS**)
- ***intermediate results are sometimes written to local disk*** (**DISK DELAYS**) before being sent over the network (*Hadoop*)

 Since network and disk latencies are relatively expensive compared to other operations, *reductions in the amount of intermediate data translate into increases in algorithmic efficiency*.

In MapReduce, local aggregation of intermediate results is one of the keys to efficient algorithms. Through use of the **combiner** and by taking advantage of the ability to **preserve state** across multiple inputs, it is often possible to substantially reduce:

- the number of key-value pairs and
- the size of key-value pairs

that need to be shuffled from the mappers to the reducers.

In-mappers Reducer

The first technique for local aggregation is the **combiner**, already discussed previously. Combiners provide a general mechanism within the MapReduce framework to reduce the amount of intermediate data generated by the mappers – recall that they can be understood as "mini reducers" that process the output of mappers. However, Hadoop does not guarantee combiners to be executed. We have already seen an example – the word count example – but now, we are going to speak about **IN-MAPPERS REDUCER**.

In this case, the mapper is modified but the reducer remains the same before.

```
class MAPPER
      method MAP(offset a, docid d_id)
            H = new AssociativeArray
            for all word w belonging to docid d_id do
                  H(w) += 1
            for all word w belonging to H do
                  EMIT(word w, count H(w));

class REDUCER
      method REDUCE(word w, counts [c1, c2, …])
            sum = 0;
            for all count ci do
                  sum += ci;
            EMIT(word w, sum);
```

An **ASSOCIATIVE ARRAY** is introduced inside the mapper to tally up term counts within a single document: instead of emitting a key-value pair for each term in the document, this version emits a key-value pair for each unique term in the document.

Exploiting the implementation details in Hadoop we can split the process in 3 sub-tasks:

1. **INITIALIZE**
2. **MAP**
3. **CLOSE**

```
class MAPPER
      method INITIALIZE
            H = new AssociativeArray
      method MAP(offset a, docid d_id)
            for all word w belonging to docid d_id do
                  H(w) += 1
      method CLOSE
            for all word w belonging to H do
                  EMIT(word w, count H(w));
```

Prior to processing any input key-value pairs, the mapper's ***Initialize*** method is called. In this case, we initialize an associative array for holding term counts. Since it is possible to preserve state across multiple calls of the Map method (for each input key-value pair), we can continue to accumulate partial term counts in the associative array across multiple documents, and emit key-value pairs only when the mapper has processed all documents. That is, emission of intermediate data is deferred until the ***Close*** method in the pseudo-code.

N.B. *Recall that this API hook provides an opportunity to execute user-specified code after the Map method has been applied to all input key-value pairs of the input data split to which the map task was assigned.*

There are two main advantages to using this design pattern:

- First, it provides control over when local aggregation occurs and how it exactly takes place – in fact the person who writes the map function is the programmer himself.
- Second, in-mapper combining will typically be more efficient than using actual combiners. In fact, combiners reduce the amount of intermediate data that is shuffled across the network, but don't actually reduce the number of key-value pairs that are emitted by the mappers: intermediate key-value pairs are still generated on a per-document basis and this process involves unnecessary object creation and destruction (garbage collection takes time), and furthermore, object serialization and deserialization (write operation on disks could be performed). In contrast, with in-mapper combining, the mappers will generate only those key-value pairs that need to be shuffled across the network to the reducers.

There are, however, drawbacks to the in-mapper combining pattern.

- First, it breaks the functional programming paradigm of MapReduce due to state preservation. Preserving state across multiple input instances means that ***algorithmic behavior may depend on the order in which input key-value pairs are encountered***. This creates the potential for ordering-dependent bugs, which are difficult to debug on large datasets in the general case.
- Second, there is a fundamental scalability bottleneck associated with the in-mapper combining pattern. It critically depends on having sufficient ***memory to store intermediate results*** – the associative array in our example – until the mapper has completely processed all key-value pairs in an input split.
  One common solution to limiting memory usage when using the in-mapper combining technique is to "block" input key-value pairs and "flush" in-memory data structures periodically.
  The idea is simple: instead of emitting intermediate data only after every key-value pair has been processed, emit partial results after processing every n key-value pairs.

Local aggregation is also effective to deal with reduce stragglers (See later).

2. PAIRS AND STRIPES

One common approach for synchronization in MapReduce is to construct complex keys and values.
Two common design patters exemplify this strategy.

As a running example, we focus on the problem of building word **CO-OCCURRENCE MATRIX** from large corpora.

- The co-occurrence matrix of a corpus is a square matrix $n \times n$, m
- n is the number of unique words (i.e., the vocabulary size)
- A cell m_{ij} contains the number of times the word w_i co-occurs with word w_j *within a specific context*
- Context: a sentence, a paragraph a document or a window of m words
- NOTE: the matrix may be symmetric in some cases

The computation of the word co-occurrence matrix is quite simple if the entire matrix fits in memory. However, in the case where the matrix is too big to fit in memory, a naive implementation on a single machine can be very slow as memory is **paged** to disk.

Although compression techniques can increase the size of corpora for which word co-occurrence matrices can be constructed on a single machine, it is clear that there are inherent scalability limitations. We describe two MapReduce algorithms for this task that can scale to large corpora.

Word co-occurrence – Pairs Approach

INPUT			
w1	w2	w3	w1
w2	w3	w1	
w2	w3	w1	

OUTPUT			
	w1	w2	w3
w1	0	4	4
w2	4	0	3
w3	4	3	0

$(w_1, w_1) = [0]$
$(w_1, w_2) = [1,1,1,1]$
$(w_1, w_3) = [1,1,1,1]$

Pseudo-code for the first algorithm, dubbed the "Pairs" approach:

```
class MAPPER
        method MAP(offset a, line l)
                for all word w belonging to line l do
                        for all word u belonging to NEIGHBORS(w) do
                                EMIT(pair(w, u), count 1);

class REDUCER
        method REDUCE(pair p, counts [c1, c2, …])
                sum = 0;
                for all count ci do
                        sum += ci;
                EMIT(pair p, count sum);
```

Input to the problem:

- Key-value pairs in the form of an offset and a line.

The mapper:

- Processes each input document
- Emits key-value pairs with:
    - Each co-occurring word pair as the key (w_i, w_j)
    - The integer one (count 1) as the value
- This is done with two nested loops:
    - The outer loop iterates over all words
    - The inner loop iterates over all neighbors

The reducer:

- Receives pairs related to co-occurring words
    - This requires modifying the partitioner
- Computes an absolute count of the joint event
- Emits the pair and the count as the final key-value output
    - Basically reducers emit the cells of the output matrix

Word co-occurrence – Stripes Approach

INPUT			
w1	w2	w3	w1
w2	w3	w1	
w2	w3	w1	

w1 [[<w1,0>, <w2,1>, <w3,1>],
[<w1,0>, <w2,1>, <w3,1>],
[<w1,0>, <w2,1>, <w3,1>],
[<w1,0>, <w2,1>, <w3,1>]]

OUTPUT			
	w1	w2	w3
w1	0	4	4
w2	4	0	3
w3	4	3	0

Pseudo-code for the first algorithm, dubbed the "Stripes" approach:

```
class MAPPER
        method MAP(offset a, line l)
                for all word w belonging to line l do
                        Stripe H = new AssociativeArray
                        for all word u belonging to NEIGHBORS(w) do
                                H(w) += 1
                        EMIT(word w, Stripe H);

class REDUCER
        method REDUCE(word w, Stripes [H1, H2, …])
                Hf = new AssociativeArray
                for all Stripe H do
                        SUM(Hf, H)
                EMIT(word w, Stripe Hf);
```

Input to the problem:

- Key-value pairs in the form of an offset and a line (the same as before).

The mapper:

- Same two nested loops structure as before
- Co-occurrence information is first stored in an associative array
- Emit key-value pairs with:
    - Words as keys
    - The corresponding arrays as values

The reducer:

- Receives all associative arrays related to the same word
- Performs an element-wise sum of all associative arrays with the same key
- Emits key-value output in the form of word, associative array

Pairs and Stripes, a comparison

The pairs approach:

- Generates a large number of key-value pairs
    - In particular, intermediate ones, that fly over the network
- 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 because it does not need to hold intermediate data in memory (no intermediate associative array)

The stripes approach:

- Generates fewer and shorted intermediate keys ($word\ w$ vs $pair\ p$)
    - The framework has less sorting to do
- However, stripes values are more complex and have serialization / deserialization overhead ($Stripe\ H$ vs $Count\ 1$)
- Greatly benefits from combiners, as the key space is the vocabulary
- Suffers from memory paging problems, if not properly engineered (intermediate associative array)

3. COMPUTING RELATIVE FREQUENCIES

The drawback of absolute counts is that it doesn't take into account the fact that some words appear more frequently than others. Word w_i may co-occur frequently with w_j simply because one of the words is very common. A simple remedy is to convert absolute counts into relative frequencies:

$$f(w_i) = \frac{\#(w_i, w_j)}{\sum_{w_{-i}} \#(w_i, w_{-i})}$$

- $\#(w_i, w_j)$ indicates the number of times a particular pair is observed in the corpus
- $\sum_{w_{-i}} \#(w_i, w_{-i})$ indicates the sum of the counts the conditioning variable w_i co-occurs with anything else. The denominator is also called **MARGINAL**

3.1 Stripes approach

Computing relative frequencies with the stripes approach is straightforward since counts of all words that co-occur with the conditioning variable w_i are available in the associative array associated to w_i.

3.2 Pair approach

In the pairs approach, the reducer receives (w_i, w_j) as the key and the $count$ as the value. From this alone it is not possible to compute $f(w_i)$ since we do not have the marginal.

- **Solution with associative array – The bad solution**

Inside the reducer, we might buffer in memory all the words that co-occur with w_i and their $counts$, in essence building the associative array in the stripes approach. To make this work, we must define the sort order of the pair so that keys are first sorted by the left word, and then by the right word. However, we must ensure that all pairs with the same left word are sent to the same reducer and this does not happen automatically. To produce the desired behavior, we must define a custom partitioner that only looks at the left word.
This algorithm could work, and effectively it works, but it suffers from the same memory drawback as the stripes approach. So, is there a way to modify the basic pairs approach so that this advantage is retained?

- **ORDER INVERSION Design – The good solution**

If it were possible to somehow compute (or otherwise obtain access to) the marginal in the reducer before processing the joint counts, the reducer could simply divide the joint counts by the marginal to compute the

relative frequencies. But we already know that the ***ordering of key-value pairs*** can be explicitly controlled by the programmer: the programmer can define the sort order of keys so that data needed earlier is presented to the reducer before data that is needed later.
However, we still need to compute the marginal counts. To compute relative frequencies, we modify the **mapper** so that it additionally emits a "special" key of the form $(w_i, *)$, with a value of one, that represents the contribution of the word pair to the marginal.
Through use of combiners, these partial marginal counts will be aggregated before being sent to the reducers. Alternatively, the in-mapper combining pattern can be used to even more efficiently aggregate marginal counts.

In the reducer, we must make sure that the special key-value pairs $(w_i, *)$ representing the partial marginal contributions are processed before the normal key-value pairs representing the joint counts. This is accomplished by defining the **sort order of the keys** so that pairs with the special symbol of the form $(w_i, *)$ are ordered before any other key-value pairs where the left word is w_i. In addition, as with before we must also properly define the partitioner to pay attention to only the left word in each pair.

An example of the sequence key-value pairs a reducer could receive is the following:

$(dog, *)$ $[6327, 8514, \ldots]$
$(dog, aardvark)$ $[2, 1]$
$(dog, aardwolf)$ $[1]$
…

N.B. *Observe that the memory requirement for this algorithm is minimal, since only the marginal (an integer) needs to be stored. No buffering of individual co-occurring word counts is necessary, and therefore we have eliminated the scalability bottleneck of the previous algorithm.*

This design pattern, which we call **ORDER INVERSION**, occurs surprisingly often and across applications in many domains. It is so named because through proper coordination, we can access the result of a computation in the reducer before processing the data needed for that computation.