

# COMP9313: Big Data Management



**Lecturer: Xin Cao**

**Course web site: <http://www.cse.unsw.edu.au/~cs9313/>**

# **Chapter 3.1: MapReduce III**

# **Design Pattern 3: Order Inversion**

# Computing Relative Frequencies

- ❖ “Relative” Co-occurrence matrix construction
  - Similar problem as before, same matrix
  - Instead of absolute counts, we take into consideration the fact that some words appear more frequently than others
    - ▶ Word  $w_i$  may co-occur frequently with word  $w_j$  simply because one of the two is very common
  - We need to convert absolute counts to relative frequencies  $f(w_j|w_i)$ 
    - ▶ What proportion of the time does  $w_j$  appear in the context of  $w_i$ ?
- ❖ Formally, we compute:

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

- $N(\cdot, \cdot)$  is the number of times a co-occurring word pair is observed
- The denominator is called the marginal

# $f(w_j|w_i)$ : “Stripes”

- ❖ In the reducer, the counts of all words that co-occur with the conditioning variable ( $w_i$ ) are available in the associative array
- ❖ Hence, the sum of all those counts gives the marginal
- ❖ Then we divide the joint counts by the marginal and we're done

$$a \rightarrow \{b_1:3, b_2:12, b_3:7, b_4:1, \dots\}$$

$$f(b_1|a) = 3 / (3 + 12 + 7 + 1 + \dots)$$

- ❖ Problems?
  - Memory

# $f(w_j|w_i)$ : “Pairs”

- ❖ The reducer receives the pair  $(w_i, w_j)$  and the count
- ❖ From this information alone it is not possible to compute  $f(w_j|w_i)$ 
  - Computing relative frequencies requires marginal counts
  - But the marginal cannot be computed until you see all counts

$((a, b_1), \{1, 1, 1, \dots\})$

No way to compute  $f(b_1|a)$  because the marginal is unknown

# $f(w_j|w_i) : \text{“Pairs”}$

- ❖ Solution 1: Fortunately, as for the mapper, also the reducer can preserve state across multiple keys
  - We can buffer in memory all the words that co-occur with  $w_i$  and their counts
  - This is basically building the associative array in the stripes method

$$a \rightarrow \{b_1:3, b_2:12, b_3:7, b_4:1, \dots\}$$

is now buffered in the reducer side

- Problems?

# $f(w_j|w_i)$ : “Pairs”

If reducers receive pairs not sorted

$((a, b_1), \{1, 1, 1, \dots\})$   
 $((c, d_1), \{1, 1, 1, \dots\})$   
 $((a, b_2), \{1, 1, 1, \dots\})$   
... ..

When can we compute the marginal?

- ❖ We must define the sort order of the pair !!
  - In this way, the keys are first sorted by the left word, and then by the right word (in the pair)
  - Hence, we could detect if all pairs associated with the word we are conditioning on ( $w_i$ ) have been seen
  - At this point, we can use the in-memory buffer, compute the relative frequencies and emit



# $f(w_j|w_i)$ : “Pairs”

$((a, b_1), \{1, 1, 1, \dots\})$  and  $((a, b_2), \{1, 1, 1, \dots\})$  may be assigned to different reducers!

Default partitioner computed based on the whole key.

- ❖ 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
    - ▶ Hence, there is no guarantee that the pair (dog, aardvark) and (dog, zebra) are sent to the same reducer
  - What we want is that all pairs with the same left word are sent to the same reducer
- ❖ Still suffer from the memory problem!

# $f(w_j|w_i)$ : “Pairs”

## ❖ Better solutions?

$(a, *) \rightarrow 32$

Reducer holds this value in memory, rather than the stripe

$(a, b_1) \rightarrow 3$

$(a, b_2) \rightarrow 12$

$(a, b_3) \rightarrow 7$

$(a, b_4) \rightarrow 1$

...



$(a, b_1) \rightarrow 3 / 32$

$(a, b_2) \rightarrow 12 / 32$

$(a, b_3) \rightarrow 7 / 32$

$(a, b_4) \rightarrow 1 / 32$

...

- ❖ The key is to properly sequence data presented to reducers
  - If it were possible to compute the marginal in the reducer before processing the join counts, the reducer could simply divide the joint counts received from mappers by the marginal
  - The notion of “before” and “after” can be captured in the **ordering of key-value pairs**
  - 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

# $f(w_j|w_i)$ : “Pairs” – Order Inversion

- ❖ A better solution based on order inversion
- ❖ The mapper:
  - additionally emits a “special” key of the form  $(w_i, *)$
  - 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  $w_i$  (define sort order)
  - We also need to guarantee that all pairs associated with the same word are sent to the same reducer (use partitioner)

# $f(w_j|w_i)$ : “Pairs” – Order Inversion

## ❖ Example:

- The reducer finally receives:

key	values	
(dog,*)	[6327, 8514, ...]	compute marginal: $\sum_{w'} N(\text{dog}, w') = 42908$
(dog, aardvark)	[2,1]	$f(\text{aardvark} \text{dog}) = 3/42908$
(dog, aardwolf)	[1]	$f(\text{aardwolf} \text{dog}) = 1/42908$
...		
(dog, zebra)	[2,1,1,1]	$f(\text{zebra} \text{dog}) = 5/42908$
(doge,*)	[682, ...]	compute marginal: $\sum_{w'} N(\text{doge}, w') = 1267$
...		

- The pairs come in order, and thus we can compute the relative frequency immediately.

# $f(w_j|w_i)$ : “Pairs” – Order Inversion

- ❖ Memory requirements:
  - Minimal, because only the marginal (an integer) needs to be stored
  - No buffering of individual co-occurring word
  - No scalability bottleneck
- ❖ Key ingredients for 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

# Order Inversion

- ❖ Common design pattern
  - Computing relative frequencies requires marginal counts
  - But marginal cannot be computed until you see all counts
  - Buffering is a bad idea!
  - Trick: getting the marginal counts to arrive at the reducer before the joint counts
- ❖ Optimizations
  - Apply in-memory combining pattern to accumulate marginal counts

# Synchronization: Pairs vs. Stripes

- ❖ Approach 1: turn synchronization into an ordering problem
  - Sort keys into correct order of computation
  - Partition key space so that each reducer gets the appropriate set of partial results
  - Hold state in reducer across multiple key-value pairs to perform computation
  - Illustrated by the “pairs” approach
  
- ❖ Approach 2: construct data structures that bring partial results together
  - Each reducer receives all the data it needs to complete the computation
  - Illustrated by the “stripes” approach

# **How to Implement Order Inversion in MapReduce?**



# Implement a Custom Partitioner

- ❖ You need to implement a “pair” class first as the key data type
- ❖ A customized partitioner extends the *Partitioner* class

```
public static class YourPatitioner extends Partitioner<Key, Value>{
```

- The *key* and *value* are the intermediate key and value produced by the map function
- In the relevant frequencies computing problem

```
public static class FirstPatitioner extends Partitioner<StringPair, IntWritable>{
```

- ❖ It overrides the *getPartition* function, which has three parameters

```
public int getPartition(WritableComparable key, Writable value, int numPartitions)
```

- The *numPartitions* is the number of reducers used in the MapReduce program and it is specified in the driver program (by default 1)
- In the relevant frequencies computing problem

```
public int getPartition(StringPair key, IntWritable value, int numPartitions){  
    return (key.getFirst().hashCode() & Integer.MAX_VALUE) % numPartitions;  
}
```

# Partitioner in Hadoop Streaming

- ❖ Hadoop has a library class, `KeyFieldBasedPartitioner`, that is useful for many applications. This class allows the Map/Reduce framework to partition the map outputs based on certain key fields, not the whole keys.

```
mapred streaming \  
-D stream.map.output.field.separator=. \  
-D stream.num.map.output.key.fields=4 \  
-D map.output.key.field.separator=. \  
-D mapreduce.partition.keypartitioner.options=-k1,2 \  
-D mapreduce.job.reduces=12 \  
-input myInputDirs \  
-output myOutputDir \  
-mapper /bin/cat \  
-reducer /bin/cat \  
-partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

- “-D stream.map.output.field.separator=.” specifies “.” as the field separator for the map outputs. By default, the separator is ‘\t’
- “-D stream.num.map.output.key.fields=4” means the prefix up to the fourth “.” in a line will be the key and the rest of the line (excluding the fourth “.”) will be the value.

# Partitioner in Hadoop Streaming

```
mapred streaming \  
-D stream.map.output.field.separator=. \  
-D stream.num.map.output.key.fields=4 \  
-D map.output.key.field.separator=. \  
-D mapreduce.partition.keypartitioner.options=-k1,2 \  
-D mapreduce.job.reduces=12 \  
-input myInputDirs \  
-output myOutputDir \  
-mapper /bin/cat \  
-reducer /bin/cat \  
-partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

- “-D map.output.key.field.separator=.” means the separator for the key is also “.”
- “-D mapreduce.partition.keypartitioner.options=-k1,2” means MapReduce will partition the map outputs by the first two fields of the keys
- This guarantees that all the key/value pairs with the same first two fields in the keys will be partitioned into the same reducer.

# Partitioner in Hadoop Streaming

- ❖ For the relative frequency computation task, you can do like:

```
hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \  
-D stream.map.output.field.separator=\t \  
-D stream.num.map.output.key.fields=1 \  
-D map.output.key.field.separator=, \  
-D mapreduce.partition.keypartitioner.options=-k1,1 \  
-D mapreduce.job.reduces=2 \  
-input input \  
-output output \  
-mapper mapper.py \  
-reducer reducer.py \  
-partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner \  
-file mapper.py \  
-file reducer.py
```

# Partitioner in MRJob

- ❖ In your class, configure JOBCONF, like:

```
JOBCONF = {  
    'stream.map.output.field.separator':'\t',  
    'stream.num.map.output.key.fields':1,  
    'map.output.key.field.separator': ',',  
    'mapred.reduce.tasks':2,  
    'mapred.output.key.comparator.class':'org.apache.hadoop.mapred.lib.KeyFieldBasedComparator',  
    'mapreduce.partition.keypartitioner.options':'-k1,1'  
}
```

# **Design Pattern 4: Value-to-key Conversion**

# Secondary Sort

- ❖ MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered
- ❖ What if want to sort value as well?
  - E.g.,  $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r) \dots$
  - Google's MapReduce implementation provides built-in functionality
  - Unfortunately, Hadoop does not support
- ❖ Secondary Sort: sorting values associated with a key in the reduce phase, also called “value-to-key conversion”

# Secondary Sort

- ❖ Sensor data from a scientific experiment: there are  $m$  sensors each taking readings on continuous basis

$(t_1, m_1, r_{80521})$

$(t_1, m_2, r_{14209})$

$(t_1, m_3, r_{76742})$

...

$(t_2, m_1, r_{21823})$

$(t_2, m_2, r_{66508})$

$(t_2, m_3, r_{98347})$

- ❖ We wish to reconstruct the activity at each individual sensor over time
- ❖ In a MapReduce program, a mapper may emit the following pair as the intermediate result

$m_1 \rightarrow (t_1, r_{80521})$

- We need to sort the value according to the timestamp



# Secondary Sort

## ❖ Solution 1:

- Buffer values in memory, then sort
- Why is this a bad idea?

## ❖ Solution 2:

- “Value-to-key conversion” design pattern: form composite intermediate key,  $(m_1, t_1)$ 
  - ▶ The mapper emits  $(m_1, t_1) \rightarrow r_{80521}$
- Let execution framework do the sorting
- Preserve state across multiple key-value pairs to handle processing
- Anything else we need to do?
  - ▶ Sensor readings are split across multiple keys. Reducers need to know when all readings of a sensor have been processed
  - ▶ All pairs associated with the same sensor are shuffled to the same reducer (use partitioner)

# **How to Implement Secondary Sort in MapReduce?**

# Secondary Sort: Another Example

- ❖ Consider the temperature data from a scientific experiment. Columns are year, month, day, and daily temperature, respectively:

```
2012, 01, 01, 5
2012, 01, 02, 45
2012, 01, 03, 35
2012, 01, 04, 10
...
2001, 11, 01, 46
2001, 11, 02, 47
2001, 11, 03, 48
2001, 11, 04, 40
...
2005, 08, 20, 50
2005, 08, 21, 52
2005, 08, 22, 38
2005, 08, 23, 70
```



```
2012-01: 5, 10, 35, 45, ...
2001-11: 40, 46, 47, 48, ...
2005-08: 38, 50, 52, 70, ...
```

- ❖ We want to output the temperature for every year-month with the values sorted in ascending order.

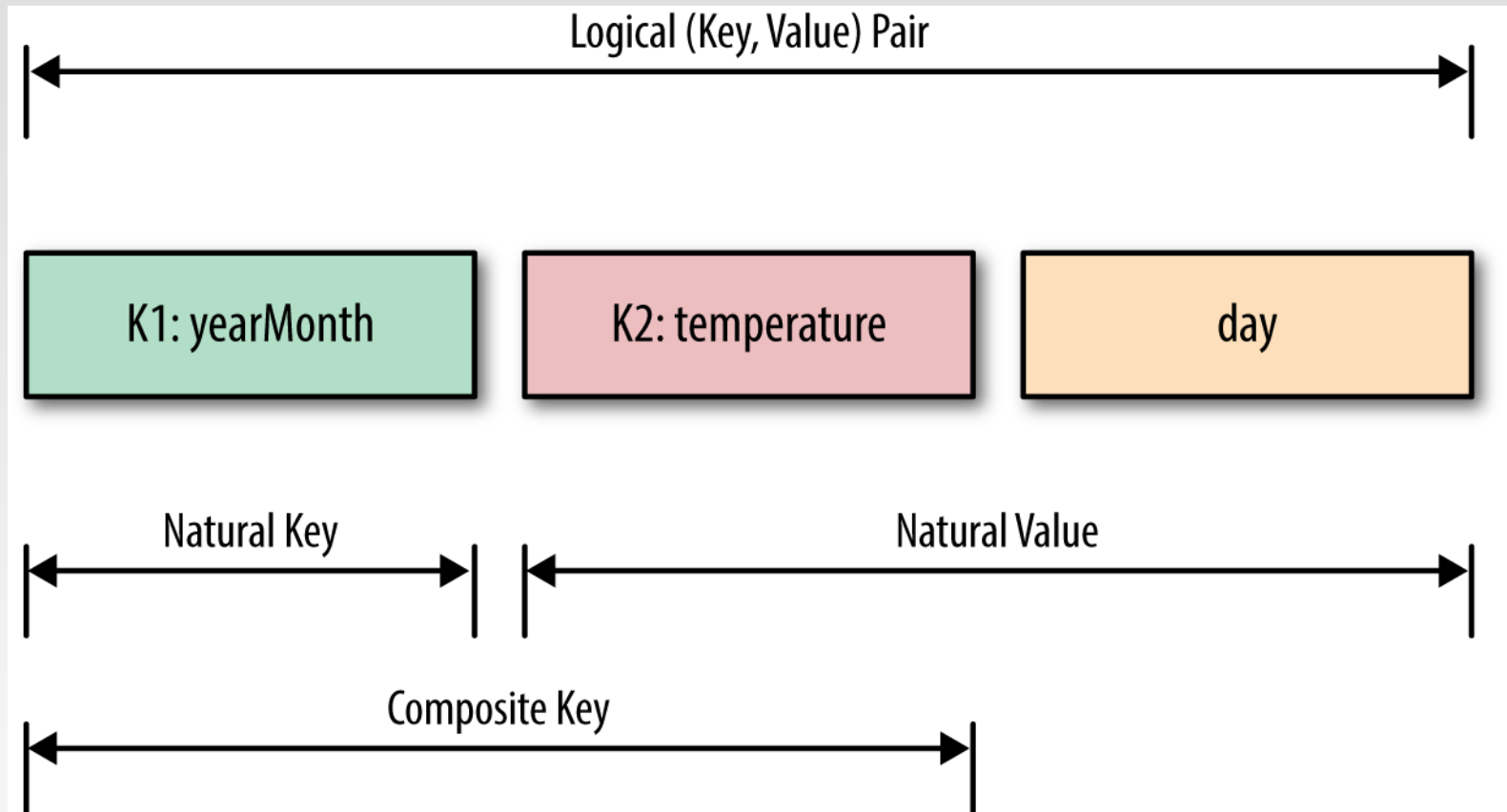
# Solutions to the Secondary Sort Problem

- ❖ Use the *Value-to-Key Conversion* design pattern:
  - form a composite intermediate key,  $(K, V)$ , where  $V$  is the secondary key. Here,  $K$  is called a *natural key*. To inject a value (i.e.,  $V$ ) into a reducer key, simply create a composite key
    - ▶  $K$ : year-month
    - ▶  $V$ : temperature data

Let the MapReduce execution framework do the sorting (rather than sorting in memory, let the framework sort by using the cluster nodes).

- ❖ Preserve state across multiple key-value pairs to handle processing. Write your own partitioner: partition the mapper's output by the natural key (year-month).

# Secondary Sorting Keys

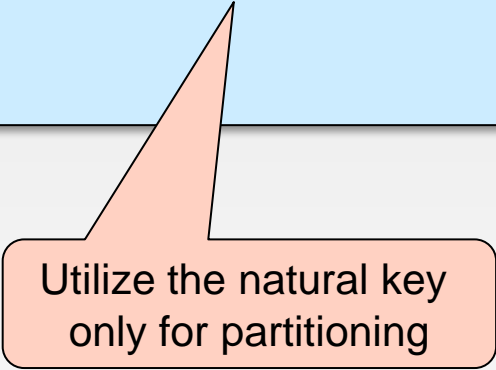


# Customize The Composite Key

```
public class DateTemperaturePair
    implements Writable, WritableComparable<DateTemperaturePair> {
    private Text yearMonth = new Text(); // natural key
    private IntWritable temperature = new IntWritable(); // secondary key
    ... ..
    @Override
    /**
    * This comparator controls the sort order of the keys.
    */
    public int compareTo(DateTemperaturePair pair) {
        int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
        if (compareValue == 0) {
            compareValue = temperature.compareTo(pair.getTemperature());
        }
        return compareValue; // sort ascending
    }
    ... ..
}
```

# Customize The Partitioner

```
public class DateTemperaturePartitioner
    extends Partitioner<DateTemperaturePair, Text> {
    @Override
    public int getPartition(DateTemperaturePair pair, Text text, int numberOfPartitions) {
        // make sure that partitions are non-negative
        return Math.abs(pair.getYearMonth().hashCode() % numberOfPartitions);
    }
}
```



Utilize the natural key  
only for partitioning

# Grouping Comparator

- ❖ Controls which keys are grouped together for a single call to `Reducer.reduce()` function.

```
public class DateTemperatureGroupingComparator extends WritableComparator {  
    ....  
    protected DateTemperatureGroupingComparator(){  
        super(DateTemperaturePair.class, true);  
    }  
    @Override  
    /* This comparator controls which keys are grouped together into  
    reduce() method */  
    public int compare(WritableComparable wc1, WritableComparable wc2) {  
        DateTemperaturePair pair = (DateTemperaturePair) wc1;  
        DateTemperaturePair pair2 = (DateTemperaturePair) wc2;  
        return pair.getYearMonth().compareTo(pair2.getYearMonth());  
    }  
}
```

Consider the natural key  
only for grouping

- ❖ Configure the grouping comparator using Job object:

```
job.setGroupingComparatorClass(DateTemperatureGroupingComparator.class);
```



# Secondary Sort by Hadoop Streaming

- ❖ Hadoop has a library class, `KeyFieldBasedComparator`, that is useful for secondary sort.

```
mapred streaming \  
-D mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapreduce.lib.partition.KeyFieldBasedComparator \  
-D stream.map.output.field.separator=. \  
-D stream.num.map.output.key.fields=4 \  
-D mapreduce.map.output.key.field.separator=. \  
-D mapreduce.partition.keycomparator.options=-k2,2nr \  
-D mapreduce.job.reduces=1 \  
-input myInputDirs \  
-output myOutputDir \  
-mapper /bin/cat \  
-reducer /bin/cat
```

- The map output keys of the above Map/Reduce job have four fields separated by “.”
- MapReduce will sort the outputs by the second field of the keys using the `-D mapreduce.partition.keycomparator.options=-k2,2nr` option
  - ▶ `-n` specifies that the sorting is numerical sorting
  - ▶ `-r` specifies that the result should be reversed

# MapReduce 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 (pre-process and post-process)
  - **Preserve state** across multiple input and intermediate keys in mappers and reducers (in-mapper combining)
  - **Control the sort order** of intermediate keys, and therefore the order in which a reducer will encounter particular keys (order inversion)
  - **Control the partitioning of the key space**, and therefore the set of keys that will be encountered by a particular reducer (partitioner)

# **Application: Building Inverted Index**

# MapReduce in Real World: Search Engine

## ❖ Information retrieval (IR)

- Focus on textual information (= text/document retrieval)
- Other possibilities include image, video, music, ...

## ❖ Boolean Text retrieval

- Each document or query is treated as a “bag” of words or terms. Word sequence is not considered
- Query terms are combined logically using the Boolean operators AND, OR, and NOT.
  - ▶ E.g., ((data AND mining) AND (NOT text))
- Retrieval
  - ▶ Given a Boolean query, the system retrieves every document that makes the query logically true.
  - ▶ Called exact match
- The retrieval results are usually quite poor because term frequency is not considered and results are not ranked

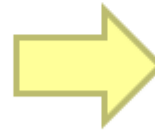
# Boolean Text Retrieval: Inverted Index

- ❖ The inverted index of a document collection is basically a data structure that
  - attaches each distinctive term with a list of all documents that contains the term.
  - The documents containing a term are sorted in the list
- ❖ Thus, in retrieval, it takes constant time to
  - find the documents that contains a query term.
  - multiple query terms are also easy handle as we will see soon.

# Boolean Text Retrieval: Inverted Index

**Doc 1**      **Doc 2**      **Doc 3**      **Doc 4**  
**one fish, two fish**      **red fish, blue fish**      **cat in the hat**      **green eggs and ham**

	1	2	3	4
blue		1		
cat			1	
egg				1
fish	1	1		
green				1
ham				1
hat			1	
one	1			
red		1		
two	1			



blue	→	2
cat	→	3
egg	→	4
fish	→	1 → 2
green	→	4
ham	→	4
hat	→	3
one	→	1
red	→	2
two	→	1

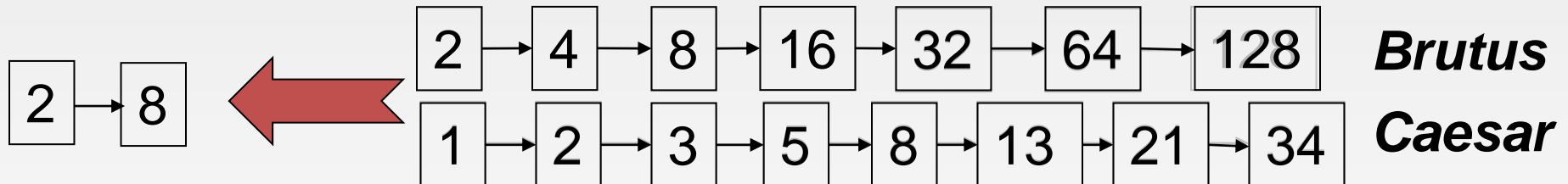
# Search Using Inverted Index

- ❖ Given a query  $q$ , search has the following steps:
  - Step 1 (vocabulary search): find each term/word in  $q$  in the inverted index.
  - Step 2 (results merging): Merge results to find documents that contain all or some of the words/terms in  $q$ .
  - Step 3 (Rank score computation): To rank the resulting documents/pages, using:
    - ▶ content-based ranking
    - ▶ link-based ranking
    - ▶ Not used in Boolean retrieval

# Boolean Query Processing: AND

Consider processing the query: **Brutus AND Caesar**

- Locate **Brutus** in the Dictionary;
  - ▶ Retrieve its postings.
- Locate **Caesar** in the Dictionary;
  - ▶ Retrieve its postings.
- “Merge” the two postings:
  - ▶ Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are  $x$  and  $y$ , the merge takes  $O(x+y)$  operations.

Crucial: postings sorted by docID.



# MapReduce it?

## ❖ The indexing problem

- Scalability is critical
- Must be relatively fast, but need not be real time
- Fundamentally a batch operation
- Incremental updates may or may not be important
- For the web, crawling is a challenge in itself

**Perfect for MapReduce!**

## ❖ The retrieval problem

- Must have sub-second response time
- For the web, only need relatively few results

**Uh... not so good...**

# MapReduce: Index Construction

- ❖ Input: documents: (docid, doc), ..
- ❖ Output: (term, [docid, docid, ...])
  - E.g., (long, [1, 23, 49, 127, ...])
    - ▶ The docid are sorted !! (used in query phase)
  - docid is an internal document id, e.g., a unique integer. Not an external document id such as a URL
- ❖ How to do it in MapReduce?

# MapReduce: Index Construction

- ❖ A simple approach:
  - Each Map task is a document parser
    - ▶ Input: A stream of documents
      - (1, long ago ...), (2, once upon ...)
    - ▶ Output: A stream of (term, docid) tuples
      - (long, 1) (ago, 1) ... (once, 2) (upon, 2) ...
  - Reducers convert streams of keys into streams of inverted lists
    - ▶ Input: (long, [1, 127, 49, 23, ...])
    - ▶ The reducer sorts the values for a key and builds an inverted list
      - Longest inverted list must fit in memory
    - ▶ Output: (long, [1, 23, 49, 127, ...])
- ❖ Problems?
  - Inefficient
  - docids are sorted in reducers

# Ranked Text Retrieval

- ❖ Order documents by how likely they are to be relevant
  - Estimate  $\text{relevance}(q, d_i)$
  - Sort documents by relevance
  - Display sorted results
- ❖ User model
  - Present hits one screen at a time, best results first
  - At any point, users can decide to stop looking
- ❖ How do we estimate relevance?
  - Assume document is relevant if it has a lot of query terms
  - Replace  $\text{relevance}(q, d_i)$  with  $\text{sim}(q, d_i)$
  - Compute similarity of vector representations
- ❖ Vector space model/cosine similarity, language models, ...

# Term Weighting

- ❖ Term weights consist of two components
  - Local: how important is the term in this document?
  - Global: how important is the term in the collection?
- ❖ Here's the intuition:
  - Terms that appear often in a document should get high weights
  - Terms that appear in many documents should get low weights
- ❖ How do we capture this mathematically?
  - TF: Term frequency (local)
  - IDF: Inverse document frequency (global)

# TF.IDF Term Weighting

$$w_{i,j} = \text{tf}_{i,j} \cdot \log \frac{N}{n_i}$$

$w_{i,j}$  weight assigned to term  $i$  in document  $j$

$\text{tf}_{i,j}$  number of occurrence of term  $i$  in document  $j$

$N$  number of documents in entire collection

$n_i$  number of documents with term  $i$

# Retrieval in a Nutshell

- ❖ Look up postings lists corresponding to query terms
- ❖ Traverse postings for each query term
- ❖ Store partial query-document scores in accumulators
- ❖ Select top  $k$  results to return

# MapReduce: Index Construction

- ❖ Input: documents: (docid, doc), ..
- ❖ Output: (t, [(docid,  $w_t$ ), (docid, w), ...])
  - $w_t$  represents the term weight of t in docid
  - E.g., (long, [(1, 0.5), (23, 0.2), (49, 0.3), (127, 0.4), ...])
    - ▶ The docid are sorted !! (used in query phase)
- ❖ How this problem differs from the previous one?
  - TF computing
    - ▶ Easy. Can be done within the mapper
  - IDF computing
    - ▶ Known only after all documents containing a term t processed
  - Input and output of map and reduce?



# Inverted Index: TF-IDF

**Doc 1**

one fish, two fish

**Doc 2**

red fish, blue fish

**Doc 3**

cat in the hat

**Doc 4**

green eggs and ham

	<i>tf</i>				
	1	2	3	4	<i>df</i>
blue		1			1
cat			1		1
egg				1	1
fish	2	2			2
green				1	1
ham				1	1
hat			1		1
one	1				1
red		1			1
two	1				1



blue	→	1	→	2	1			
cat	→	1	→	3	1			
egg	→	1	→	4	1			
fish	→	2	→	1	2	→	2	2
green	→	1	→	4	1			
ham	→	1	→	4	1			
hat	→	1	→	3	1			
one	→	1	→	1	1			
red	→	1	→	2	1			
two	→	1	→	1	1			

# MapReduce: Index Construction

- ❖ A simple approach:
  - Each Map task is a document parser
    - ▶ Input: A stream of documents
      - (1, long ago ...), (2, once upon ...)
    - ▶ Output: A stream of (term, [docid, tf]) tuples
      - (long, [1,1]) (ago, [1,1]) ... (once, [2,1]) (upon, [2,1]) ...
  - Reducers convert streams of keys into streams of inverted lists
    - ▶ Input: (long, {[1,1], [127,2], [49,1], [23,3] ...})
    - ▶ The reducer sorts the values for a key and builds an inverted list
      - Compute TF and IDF in reducer!
    - ▶ Output: (long, [(1, 0.5), (23, 0.2), (49, 0.3), (127,0.4), ...])

# MapReduce: Index Construction

Map

Doc 1  
one fish, two fish

one	1	1
two	1	1
fish	1	2

Doc 2  
red fish, blue fish

red	2	1
blue	2	1
fish	2	2

Doc 3  
cat in the hat

cat	3	1
hat	3	1

Shuffle and Sort: aggregate values by keys

Reduce

cat	3	1		
fish	1	2	2	2
one	1	1		
red	2	1		

blue	2	1
hat	3	1
two	1	1

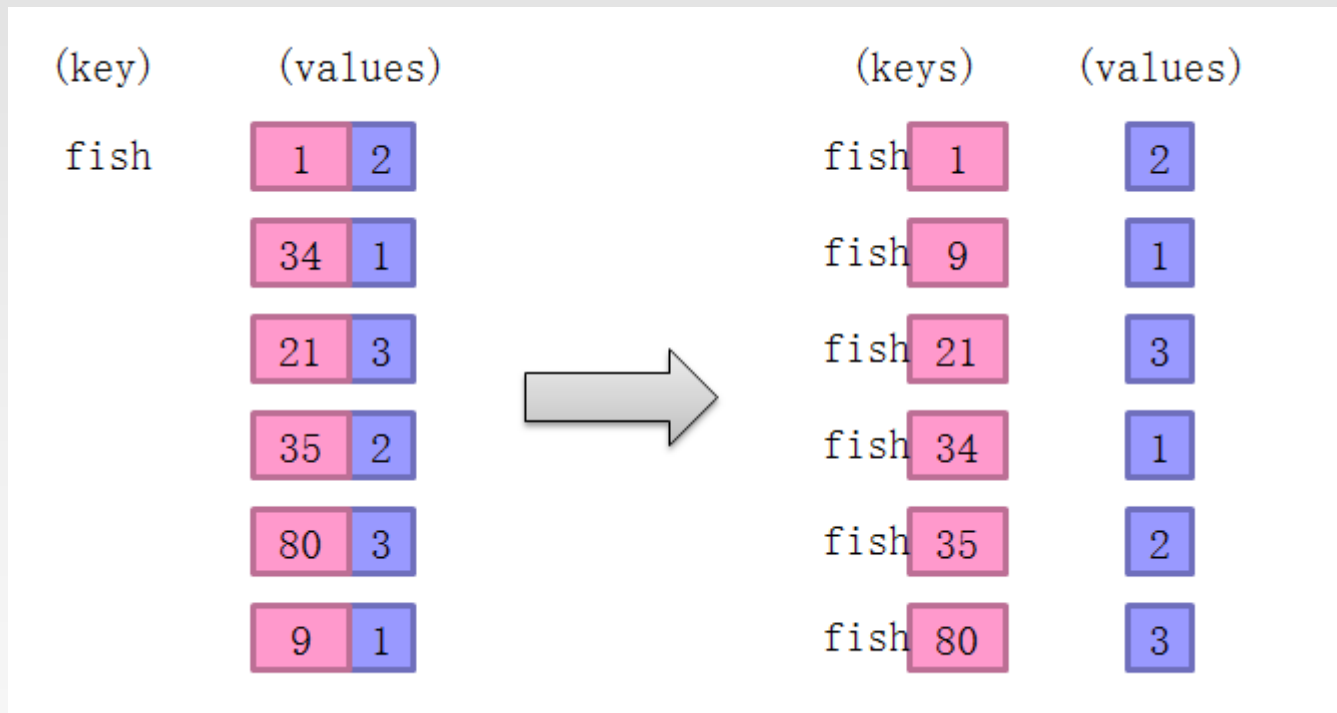
# MapReduce: Index Construction

- ❖ Inefficient: terms as keys, postings as values
  - docids are sorted in reducers
  - IDF can be computed only after all relevant documents received
  - Reducers must buffer all postings associated with key (to sort)
    - ▶ What if we run out of memory to buffer postings?
  - Improvement?

# The First Improvement

- ❖ How to make Hadoop sort the docid, instead of doing it in reducers?
- ❖ Design pattern: value-to-key conversion, secondary sort

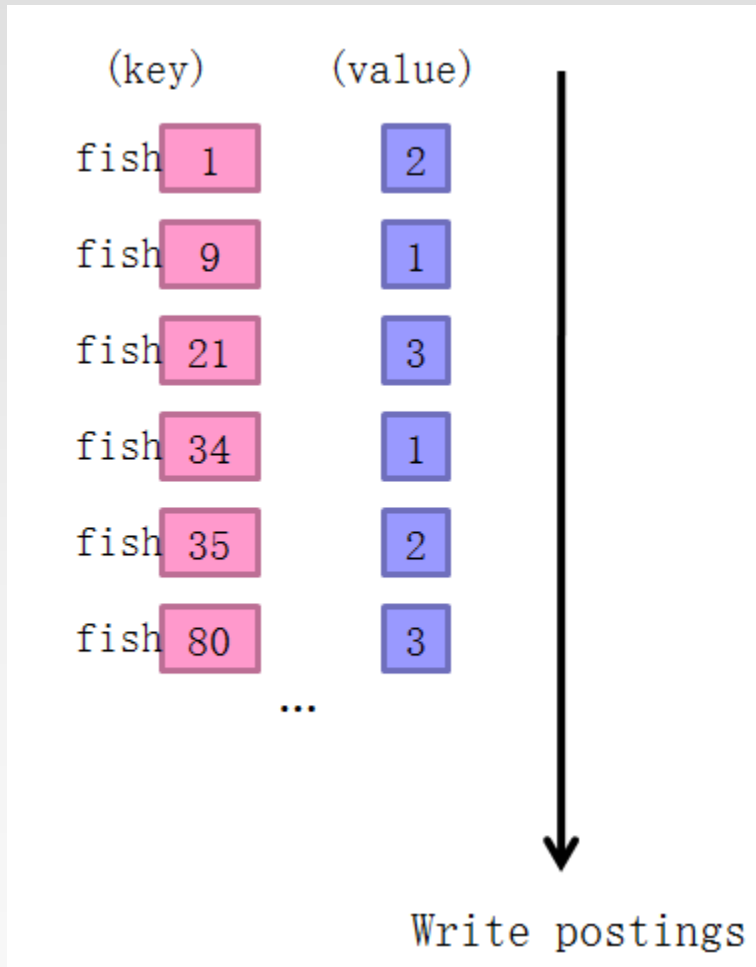
Mapper output a stream of ([term, docid], tf) tuples



Remember: you must implement a partitioner on term!

# The Second Improvement

- ❖ How to avoid buffering all postings associated with key?



We'd like to store the DF at the front of the postings list

But we don't know the DF until we've seen all postings!

Sound familiar?

Design patten: Order inversion

# The Second Improvement

- ❖ Getting the DF
  - In the mapper:
    - ▶ Emit “special” key-value pairs to keep track of DF
  - In the reducer:
    - ▶ Make sure “special” key-value pairs come first: process them to determine DF
  - Remember: proper partitioning!

	(key)	(value)
fish	1	2
one	1	1
two	1	1
fish	★	1
one	★	1
two	★	1

Emit normal key-value pairs...

Emit “special” key-value pairs to keep track of df...

**Doc1:** one fish, two fish

# The Second Improvement

	(key)	(value)
fish	★	21 32 ...

Write the DF...

fish	1	2
fish	9	1
fish	21	3
fish	34	1
fish	35	2
fish	80	3

...

Write postings

First, compute the DF by summing contributions from all “special” key-value pair...

Important: properly define sort order to make sure “special” key-value pairs come first!



# More Detailed Hadoop MapReduce I/O

## ❖ InputSplit

- A **chunk** of the input processed by a single map
- Each split is divided into records
- Split is just a reference to the data (doesn't contain the input data)

```
public interface InputSplit extends Writable {  
    long getLength() throws IOException;  
    String[] getLocations() throws IOException;  
}
```

## ❖ RecordReader

- Iterate over records
- Used by the map task to generate record key-value pairs

- ❖ As a MapReduce application programmer, we do not need to deal with InputSplit directly, as they are created in InputFormat

# MapReduce InputFormat/OutputFormat

- ❖ InputFormat describes the input-specification for a MapReduce job
  - The Map-Reduce framework relies on the InputFormat to:
    - ▶ Validate the input-specification of the job
    - ▶ Split-up the input file(s) into logical InputSplits, each of which is then assigned to an individual Mapper
    - ▶ Provide the RecordReader implementation to be used to glean input records from the logical InputSplit for processing by the Mapper
  - The application usually has to implement a RecordReader for the individual task
  - In MapReduce by default TextInputFormat and LineRecordReader

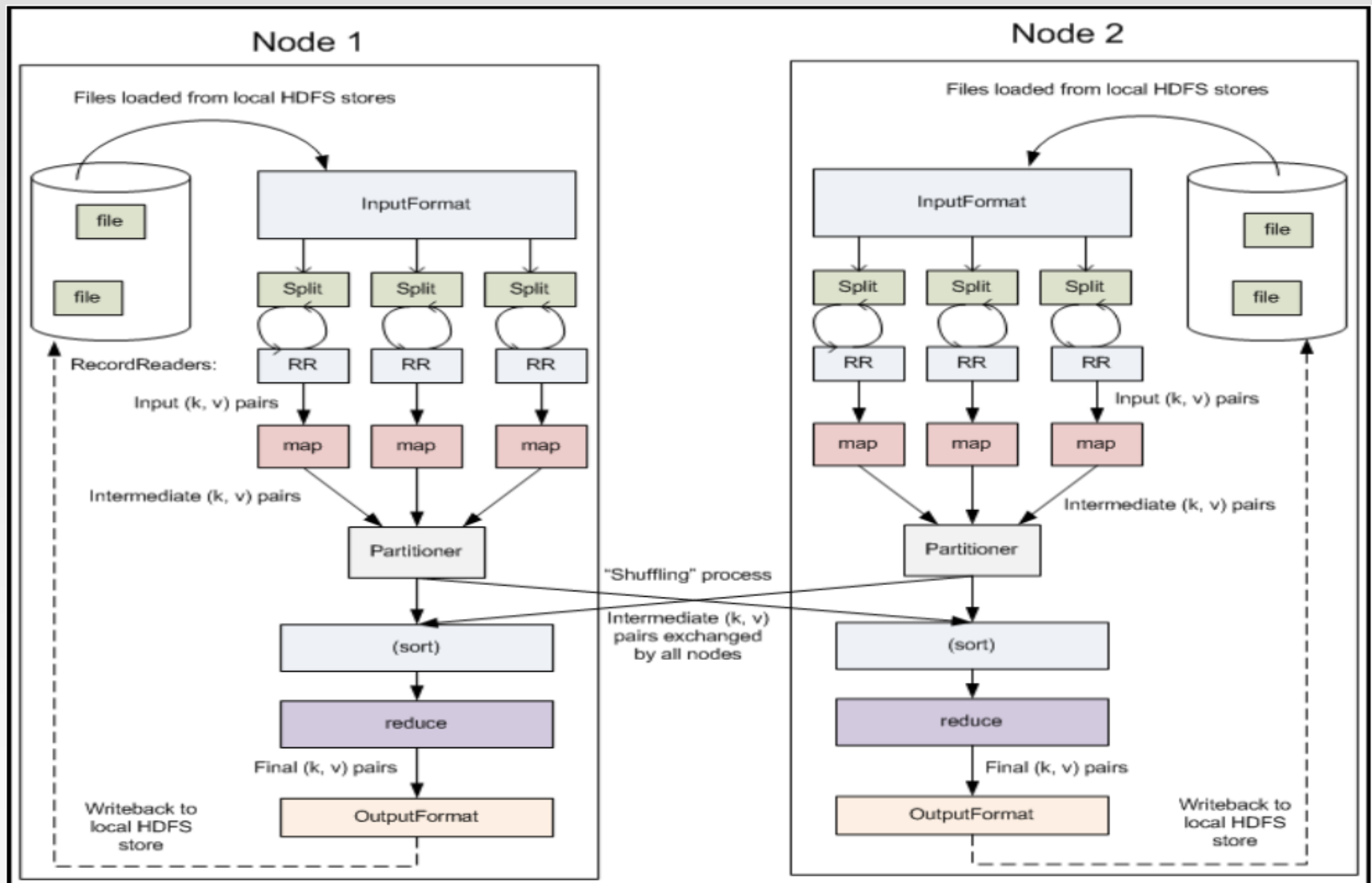
# MapReduce SequenceFile

- ❖ File operations based on binary format rather than text format
- ❖ SequenceFile class provides a persistent data structure for binary key-value pairs, e.g.,
  - Key: timestamp represented by a LongWritable
  - Value: quantity being logged represented by a Writable
- ❖ Use SequenceFile in MapReduce:
  - `job.setInputFormatClass(SequenceFileInputFormat.class);`
  - `job.setOutputFormatClass(SequenceFileOutputFormat.class);`

# MapReduce OutputFormat

- ❖ OutputFormat describes the output-specification for a MapReduce job
  - The Map-Reduce framework relies on the OutputFormat to:
    - ▶ Validate the output-specification of the job. For e.g. check that the output directory doesn't already exist.
    - ▶ Provide the RecordWriter implementation to be used to write out the output files of the job. Output files are stored in a FileSystem.
  - In Mapreduce by default TextOutputFormat and LineRecordWriter

# Detailed Hadoop MapReduce Data Flow



# Example: Creating Inverted Index

- ❖ Given you a large text file containing the contents of huge amount of webpages, in which each webpage starts with “<DOC>” and ends with “</DOC>”, your task is to create an inverted index for these documents.
  - [A sample file](#)
- ❖ Procedure:
  - Implement a custom RecordReader
  - Implement a custom InputFormat, and overwrite the CreateRecordReader() function to return your self-defined RecordReader object
  - Configure the InputFormat class in the main function using `job.setInputFormatClass()`

# References

- ❖ Data-Intensive Text Processing with MapReduce. Jimmy Lin and Chris Dyer. University of Maryland, College Park.
- ❖ Hadoop The Definitive Guide. Hadoop I/O, and MapReduce Features chapters.

**End of Chapter 3.1**