COMP9313: Big Data Management

MapReduce

Data Structure in MapReduce

- Key-value pairs are the basic data structure in MapReduce
  - Keys and values can be: integers, float, strings, raw bytes
  - They can also be arbitrary data structures

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

Recall of Map and Reduce

- Map
  - Reads data (split in Hadoop, RDD in Spark)
  - Produces key-value pairs as intermediate outputs

- Reduce
  - Receive key-value pairs from multiple map jobs
  - aggregates the intermediate data tuples to the final output

MapReduce in Hadoop

- Data stored in HDFS (organized as blocks)
- Hadoop MapReduce Divides input into fixed-size pieces, input splits
  - Hadoop creates one map task for each split
  - Map task runs the user-defined map function for each record in the split
  - Size of a split is normally the size of a HDFS block
- Data locality optimization
  - Run the map task on a node where the input data resides in HDFS
  - This is the reason why the split size is the same as the block size
    - The largest size of the input that can be guaranteed to be stored on a single node
    - If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks

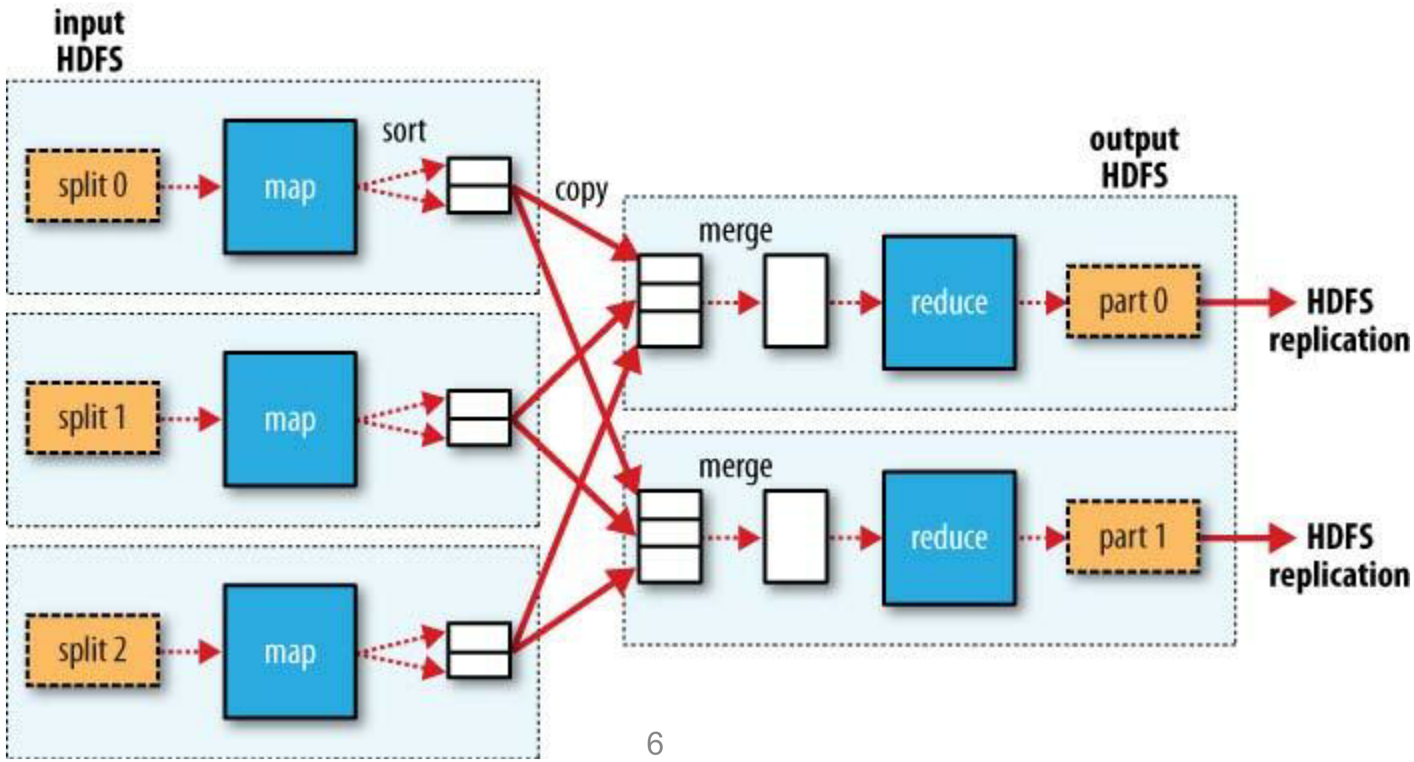
MapReduce in Hadoop

- Map tasks write their output to local disk (not to HDFS)
  - Map output is intermediate output
  - Once the job is complete the map output can be thrown away
  - Storing it in HDFS with replication, would be overkill
  - If the node of map task fails, Hadoop will automatically rerun the map task on another node

- Reduce tasks don't have the advantage of data locality
  - Input to a single reduce task is normally the output from all mappers
  - Output of the reduce is stored in HDFS for reliability
  - The number of reduce tasks is not governed by the size of the input, but is specified independently

More Detailed MapReduce Dataflow

- When there are multiple reducers, the map tasks partition their output:
  - One partition for each reduce task
  - The records for every key are all in a single partition
  - Partitioning can be controlled by a user-defined partitioning function



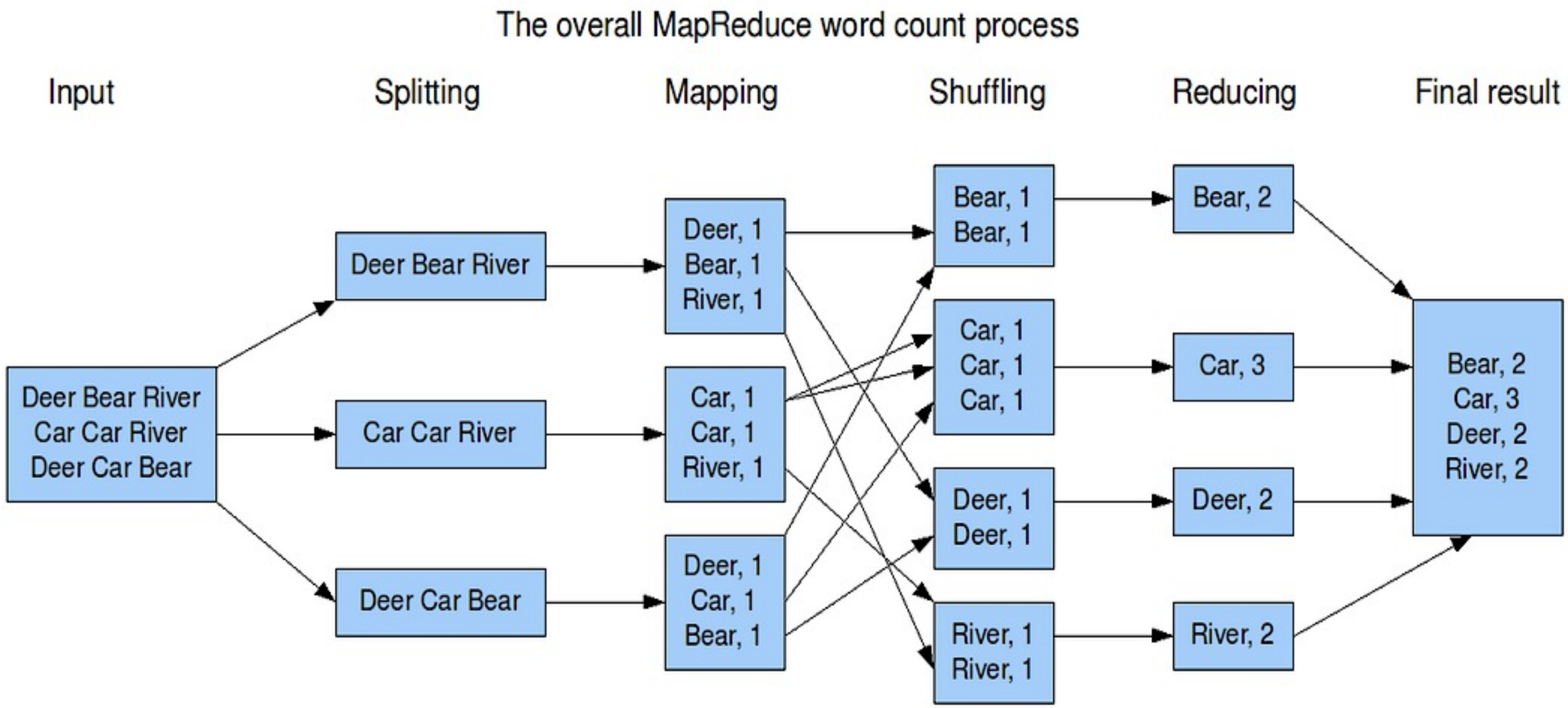
Shuffle

- Shuffling is the process of data redistribution
  - To make sure each reducer obtains all values associated with the same key.
  - It is needed for all of the operations which require grouping
    - E.g., word count, compute avg. score for each department, …
- Spark and Hadoop have different approaches implemented for handling the shuffles.

Shuffle in Hadoop (handled by framework)

- Happens between each Map and Reduce phase
- Use Shuffle and Sort mechanism
  - Results of each Mapper are sorted by the key
  - Starts as soon as each mapper finishes
- Use combiner to reduce the amount of data shuffled
  - Combiner combines key-value pairs with the same key in each par
  - This is not handled by framework!

Example of MapReduce in Hadoop



Shuffle in Spark (handled by Spark)

- Triggered by some operations
  - Distinct, join, repartition, all *By, *ByKey
  - I.e., Happens between stages
- Hash shuffle
- Sort shuffle
- Tungsten shuffle-sort
  - More on https://issues.apache.org/jira/browse/SPARK-7081

Hash Shuffle

- Data are hash partitioned on the map side
  - Hashing is much faster than sorting
- Files created to store the partitioned data portion
  - # of mappers X # of reducers
- Use consolidateFiles to reduce the # of files
  - From M * R => E*C/T * R
- Pros:
  - Fast
  - No memory overhead of sorting
- Cons:
  - Large amount of output files (when # partition is big)

Sort Shuffle

- For each mapper 2 files are created
  - Ordered (by key) data
  - Index of beginning and ending of each 'chunk'
- Merged on the fly while being read by reducers
- Default way
  - Fallback to hash shuffle if # partitions is small
- Pros
  - Smaller amount of files created
- Cons
  - Sorting is slower than hashing

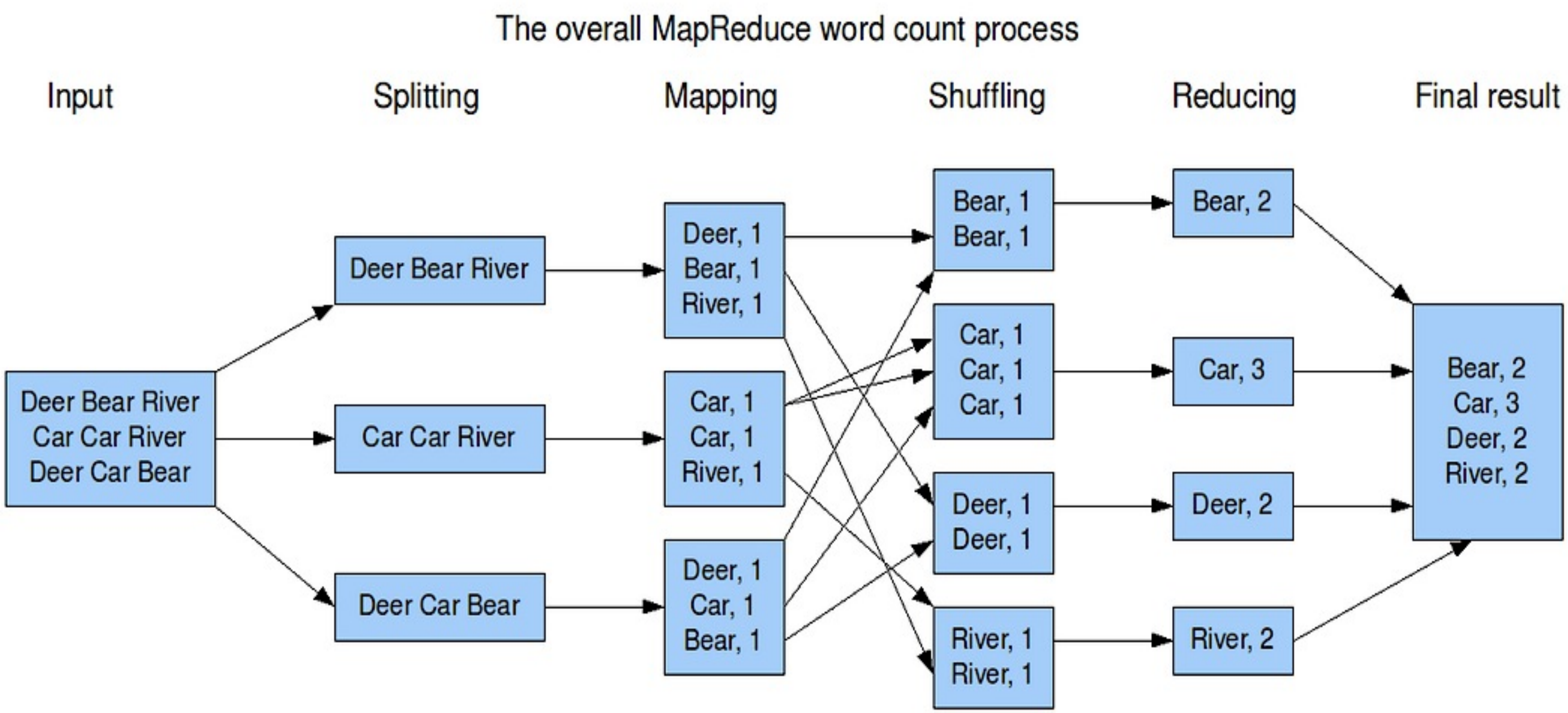
MapReduce in Spark

MapReduce Functions in Spark (Recall)

- Transformation
  - Narrow transformation
  - Wide transformation
- Action

- The job is a list of Transformations followed by one Action
  - Only action will trigger the 'real' execution
    - I.e., lazy evaluation

Transformation = Map? Action = Reduce?



combineByKey

- RDD([K, V]) to RDD([K, C])
  - K: key, V: value, C: combined type
- Three parameters (functions)
  - createCombiner
    - What is done to a single row when it is FIRST met?
    - V => C
  - mergeValue
    - What is done to a single row when it meets a previously reduced row?
    - C, V => C
    - In a partition
  - mergeCombiners
    - What is done to two previously reduced rows?
    - C, C => C
    - Across partitions

Example: word count

- createCombiner
  - What is done to a single row when it is FIRST met?
  - V => C
  - lambda v: v
- mergeValue
  - What is done to a single row when it meets a previously reduced row?
  - C, V => C
  - lambda c, v: c+v
- mergeCombiners
  - What is done to two previously reduced rows?
  - C, C => C
  - lambda c1, c2: c1+c2

Example 2: Compute Max by Keys

- createCombiner
  - What is done to a single row when it is FIRST met?
  - V => C
  - lambda v: v
- mergeValue
  - What is done to a single row when it meets a previously reduced row?
  - C, V => C
  - lambda c, v: max(c, v)
- mergeCombiners
  - What is done to two previously reduced rows?
  - C, C => C
  - lambda c1, c2: max(c1, c2)

Example 3: Compute Sum and Count

- createCombiner
  - V => C
  - lambda v: (v, 1)
- mergeValue
  - C, V => C
  - lambda c, v: (c[0] + v, c[1] + 1)
- mergeCombiners
  - C, C => C
  - lambda c1, c2: (c1[0] + c2[0], c1[1] + c2[1])

Example 3: Compute Sum and Count

- data = [('A', 2.), ('A', 4.), ('A', 9.), ('B', 10.), ('B', 20.), ('Z', 3.), ('Z', 5.), ('Z', 8.)]
  - Partition 1: ('A', 2.), ('A', 4.), ('A', 9.), ('B', 10.)
  - Partition 2: ('B', 20.), ('Z', 3.), ('Z', 5.), ('Z', 8.)
- Partition 1 ('A', 2.), ('A', 4.), ('A', 9.), ('B', 10.)
  - A=2. --> createCombiner(2.) ==> accumulator[A] = (2., 1)
  - A=4. --> mergeValue(accumulator[A], 4.) ==> accumulator[A] = (2. + 4., 1 + 1) = (6., 2)
  - A=9. --> mergeValue(accumulator[A], 9.) ==> accumulator[A] = (6. + 9., 2 + 1) = (15., 3)
  - B=10. --> createCombiner(10.) ==> accumulator[B] = (10., 1)
- Partition 2 ('B', 20.), ('Z', 3.), ('Z', 5.), ('Z', 8.), ('Z', 12.)
  - B=20. --> createCombiner(20.) ==> accumulator[B] = (20., 1)
  - Z=3. --> createCombiner(3.) ==> accumulator[Z] = (3., 1)
  - Z=5. --> mergeValue(accumulator[Z], 5.) ==> accumulator[Z] = (3. + 5., 1 + 1) = (8., 2)
  - Z=8. --> mergeValue(accumulator[Z], 8.) ==> accumulator[Z] = (8. + 8., 2 + 1) = (16., 3)
- Merge partitions together
  - A ==> (15., 3)
  - B ==> mergeCombiner((10., 1), (20., 1)) ==> (10. + 20., 1 + 1) = (30., 2)
  - Z ==> (16., 3)
- Collect
  - ([A, (15., 3)], [B, (30., 2)], [Z, (16., 3)])

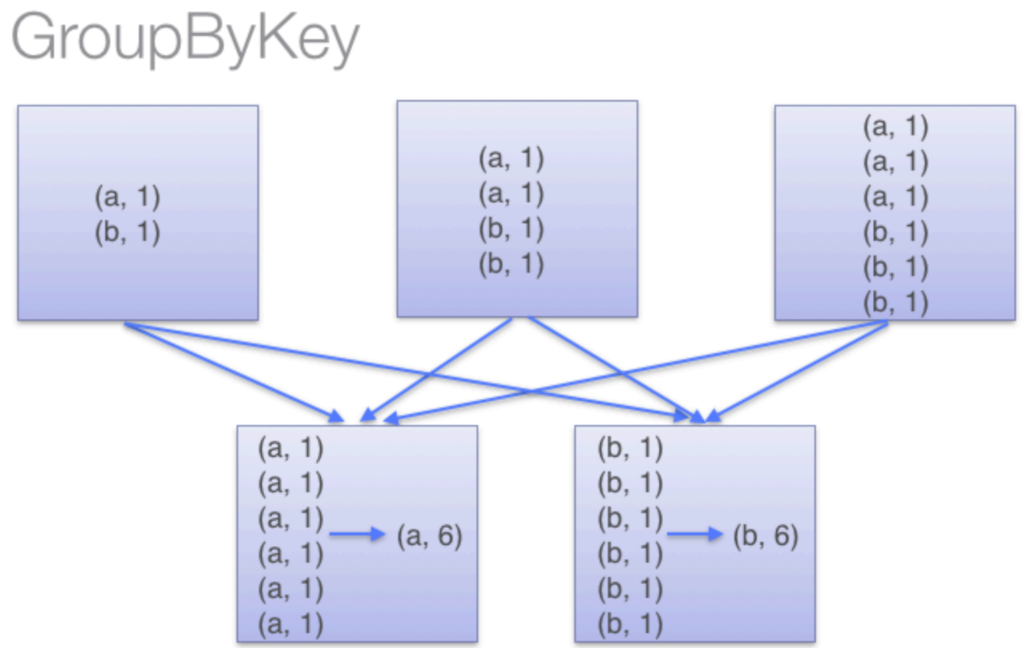
reduceByKey

- reduceByKey(func)
  - Merge the values for each key using func
  - E.g., reduceByKey(lambda x, y: x + y)

- createCombiner
  - lambda v: v
- mergeValue
  - func
- mergeCombiners
  - func

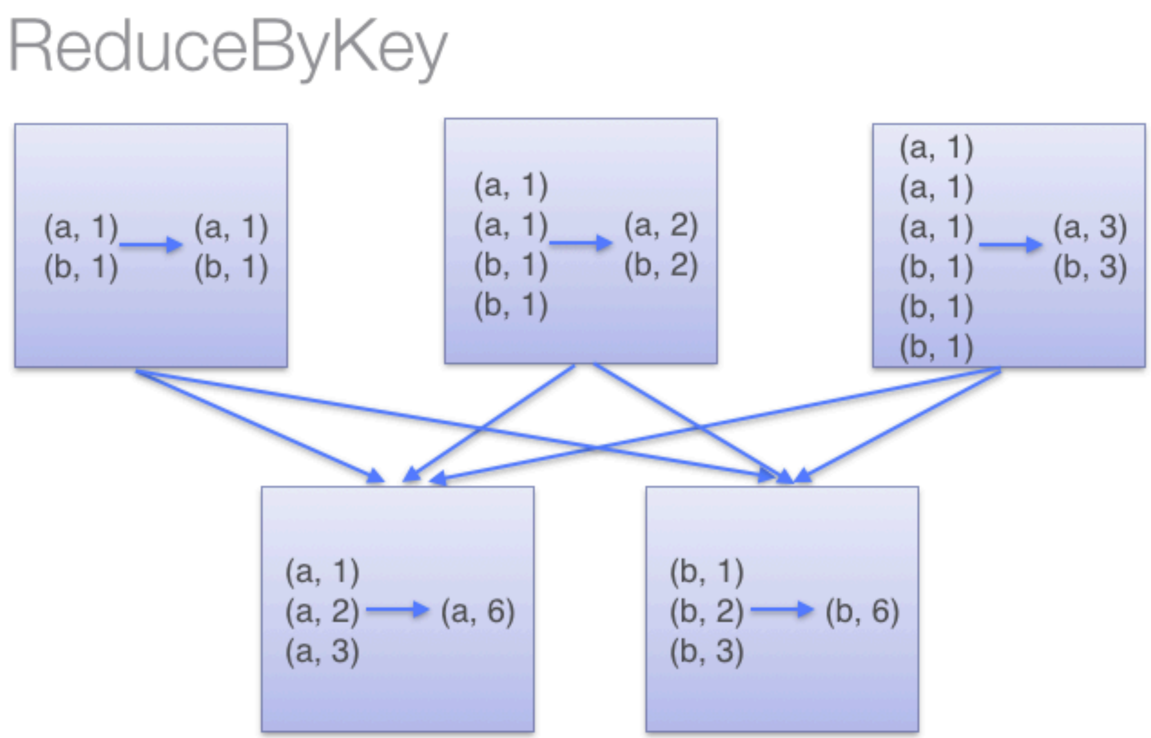
groupByKey

- groupByKey()
  - Group the values for each key in the RDD into a single sequence.
  - Data shuffle according to the key value in another RDD



reduceByKey

- Combines before shuffling
- Avoid using groupByKey



The Efficiency of MapReduce in Spark

- Number of transformations
  - Each transformation involves a linearly scan of the dataset (RDD)
- Size of transformations
  - Smaller input size => less cost on linearly scan
- Shuffles
  - data transferring between partitions is costly
    - especially in a cluster!
      - Disk I/O
      - Data serialization and deserialization
      - Network I/O

Number of Transformations (and Shuffles)

rdd = sc.parallelize(data)

- data: (id, score) pairs

- Bad design

  maxByKey = rdd.combineByKey(…)
  sumByKey = rdd.combineByKey(…)
  sumMaxRdd = maxByKey.join(sumByKey)

- Good design

  sumMaxRdd = rdd.combineByKey(…)

Size of Transformations

rdd = sc.parallelize(data)

- data: (word, 1) pairs

- Bad design

  countRdd = rdd.reduceByKey(…)

  fileteredRdd = countRdd.filter(…)

- Good design

  fileteredRdd = countRdd.filter(…)

  countRdd = fileteredRdd.reduceByKey(…)

Partition

rdd = sc.parallelize(data)

- data: (word, 1) pairs

- Bad design

  countRdd = rdd.reduceByKey(…)

  countBy2ndCharRdd = countRdd.map(…).reduceByKey(…)

- Good design

  paritionedRdd = data.partitionBy(…)

  countBy2ndCharRdd = paritionedRdd.map(…).reduceByKey(…)

How to Merge Two RDDs?

- Union
  - Concatenate two RDDs
- Zip
  - Pair two RDDs
- Join
  - Merge based on the keys from 2 RDDs
  - Just like join in DB

Union

- How do A and B union together?
  - What is the number of partitions for the union of A and B?
  - Case 1: Different partitioner:
    - Note: default partitioner is None
  - Case 2: Same partitioner:

Zip

- Key-Value pairs after A.zip(B)
  - Key: tuples in A
  - Value: tuples in B

- Assumes that the two RDDs have
  - The same number of partitions
  - The same number of elements in each partition
  - E.g., 1-to-1 map

Join

- E.g., A.*Join(B)
- join
  - All pairs with matching Keys from A and B
- leftOuterJoin
  - Case 1: in both A and B
  - Case 2: in A but not B
  - Case 3: in B but not A
- rightOuterJoin
  - Opposite to leftOuterJoin
- fullOuterJoin
  - Union of leftOuterJoin and rightOuterJoin

Application: Term Co-occurrence Computation

Term Co-occurrence Computation

- Term co-occurrence matrix for a text collection
  - Input: A collection of documents/sentences
  - Output: M = N x N matrix (N = vocabulary size)
    - M_{ij}: number of times *i-th* and *j-th term* co-occur in some context
  - Why we need MapReduce (also the difficulties)
    - A large event space (number of terms)
    - A large number of observations (the collection itself)
  - Applications: data mining, language model, information retrieval, bioinformatics, etc.

Naïve Solution: “Pairs”

- Map a sentence into pairs of terms with its count
  - Generate all co-occurring term pairs

```
ForAll term u in sent s do:
    ForAll term v in Neighbors(u) do:
        emit((u,v), 1)
```

- Reduce by key (i.e., the term pair) and sum up the counts
- Example:
  - *A boy can do everything for girl.*

“Pairs” Analysis

- Advantages
  - Easy to implement, easy to understand

- Disadvantages
  - Lots of pairs to sort and shuffle around
    - upper bound?
  - Not many opportunities for combiners to work

Alternative Solution: "Stripes"

- Motivation
  - The NxN matrix is sparse
    - You cannot expect relationship between every pair of words!
- Idea

(a, b) → 1
(a, d) → 5
(a, e) → 3

a → { b: 1, d: 5, e: 3}

(a, b) → 1
(a, c) → 2
(a, d) → 2
(a, f) → 2

a → { b: 1, c: 2, d: 2, f: 2 }

   a → { b: 1, d: 5, e: 3 }
\+ a → { b: 1, c: 2, d: 2, f: 2 }
____
   a → { b: 2, c: 2, d: 7, e: 3, f: 2 }

MapReduce of "Stripes"

- Map a sentence into stripes

```
ForAll term u in sent s do:
    H_u = new dictionary
    ForAll term v in Neighbors(u) do:
        H_u(v) = H_u(v)+1
```

- Reduce by key and merge the dictionaries
  - element-wise sum of dictionaries

“Stripes” Analysis

- Advantages
  - Far less sorting and shuffling of key-value pairs

- Disadvantages
  - More difficult to implement
  - Underlying object more heavyweight
  - Fundamental limitation in terms of size of event space

Pairs vs. Stripes

- The pairs approach
  - Keeps track of each pair of co-occur terms separately
  - Generates a large number of key-value pairs (also intermediate)
  - The benefit from combiners is limited, as it is less likely for a mapper to process multiple occurrences of a pair of words
- The stripe approach
  - Keeps track of all terms that co-occur with the same term
  - Generates fewer and shorted intermediate keys
    - The framework has less sorting to do
  - Greatly benefits from combiners, as the key space is the vocabulary
  - More efficient, but may suffer from memory problem

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
    - Exact match

Boolean Text Retrieval: Inverted Index

- The inverted index of a document collection is 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
    - Why sorted?

- 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

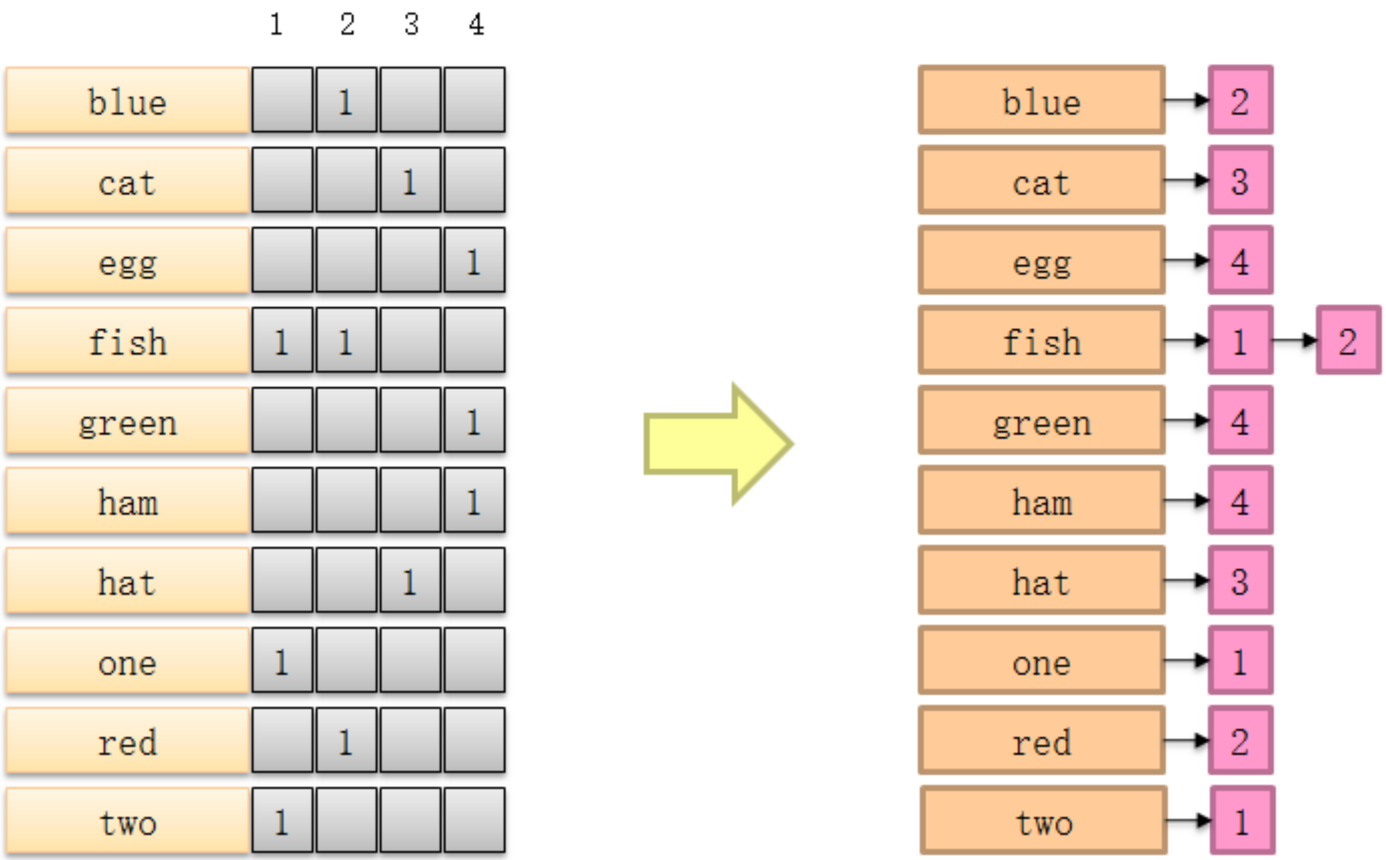
Boolean Text Retrieval: Inverted Index

Doc 1 one fish, two fish

Doc 2 red fish, blue fish

Doc 3 cat in the hat

Doc 4 green eggs and ham

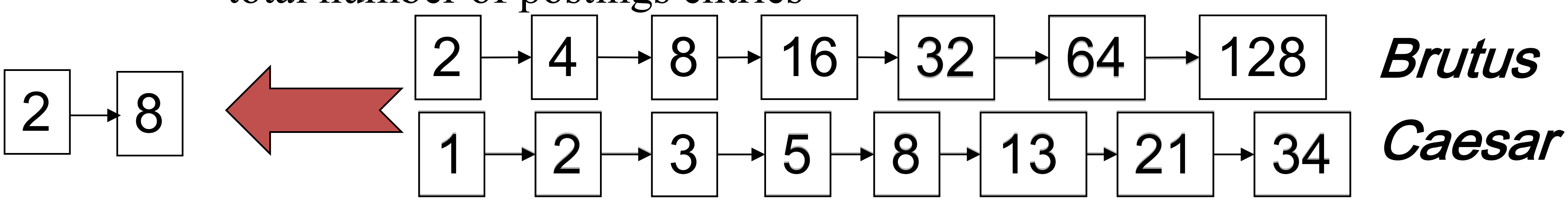


Search Using Inverted Index

- 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.

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

MapReduce: Index Construction

- Input: documents
  - (docid, doc), ..

- Output: (term, [docid, docid, …])
  - E.g., (long, [1, 23, 49, 127, …])
    - The docid are sorted
  - 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, …])

Ranked Text Retrieval

- Order documents by how likely they are to be relevant
  - Estimate 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 relevance(q, d_i) with 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

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

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

$\mathrm{tf}_{i,j}$ **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

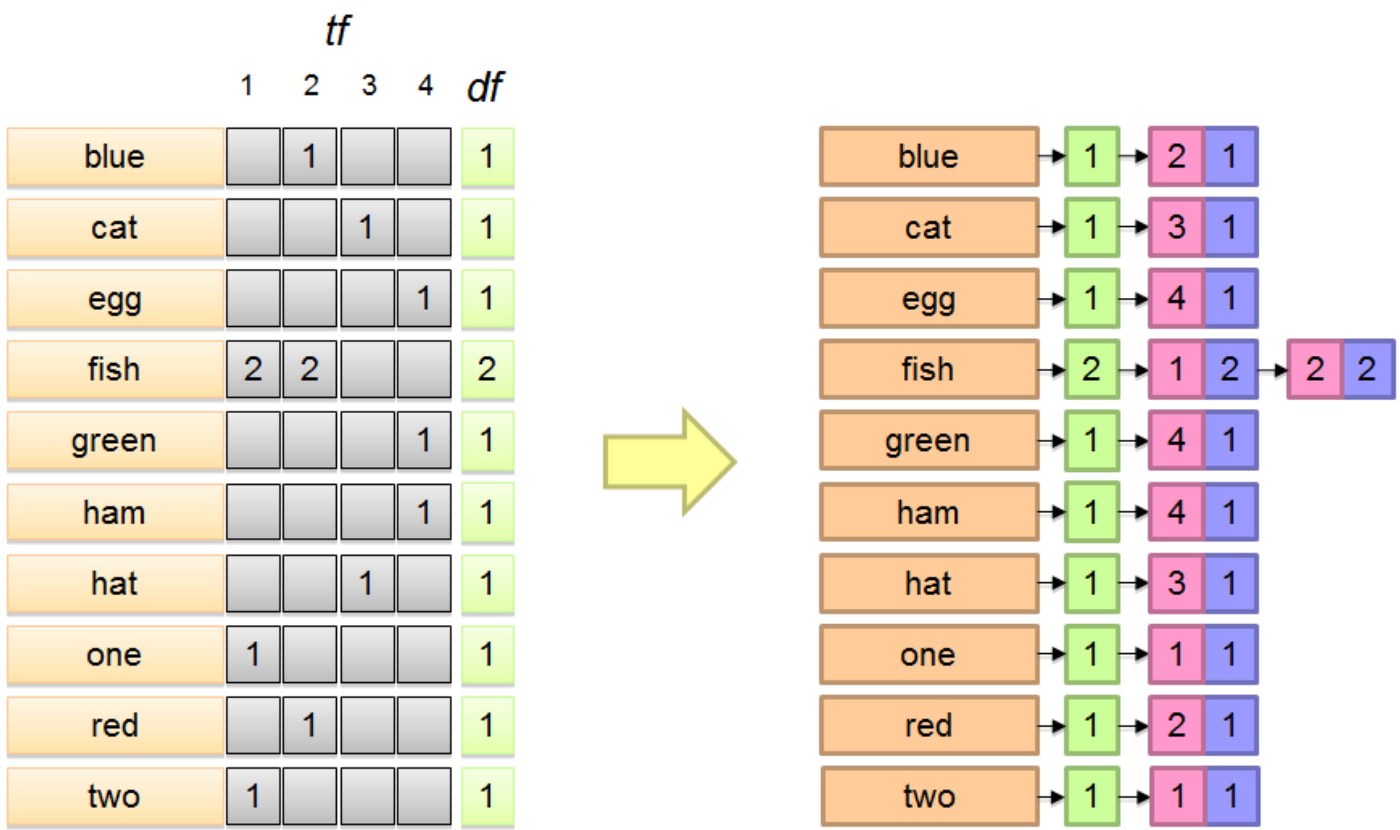
Inverted Index: TF-IDF

Doc 1 one fish, two fish

Doc 2 red fish, blue fish

Doc 3 cat in the hat

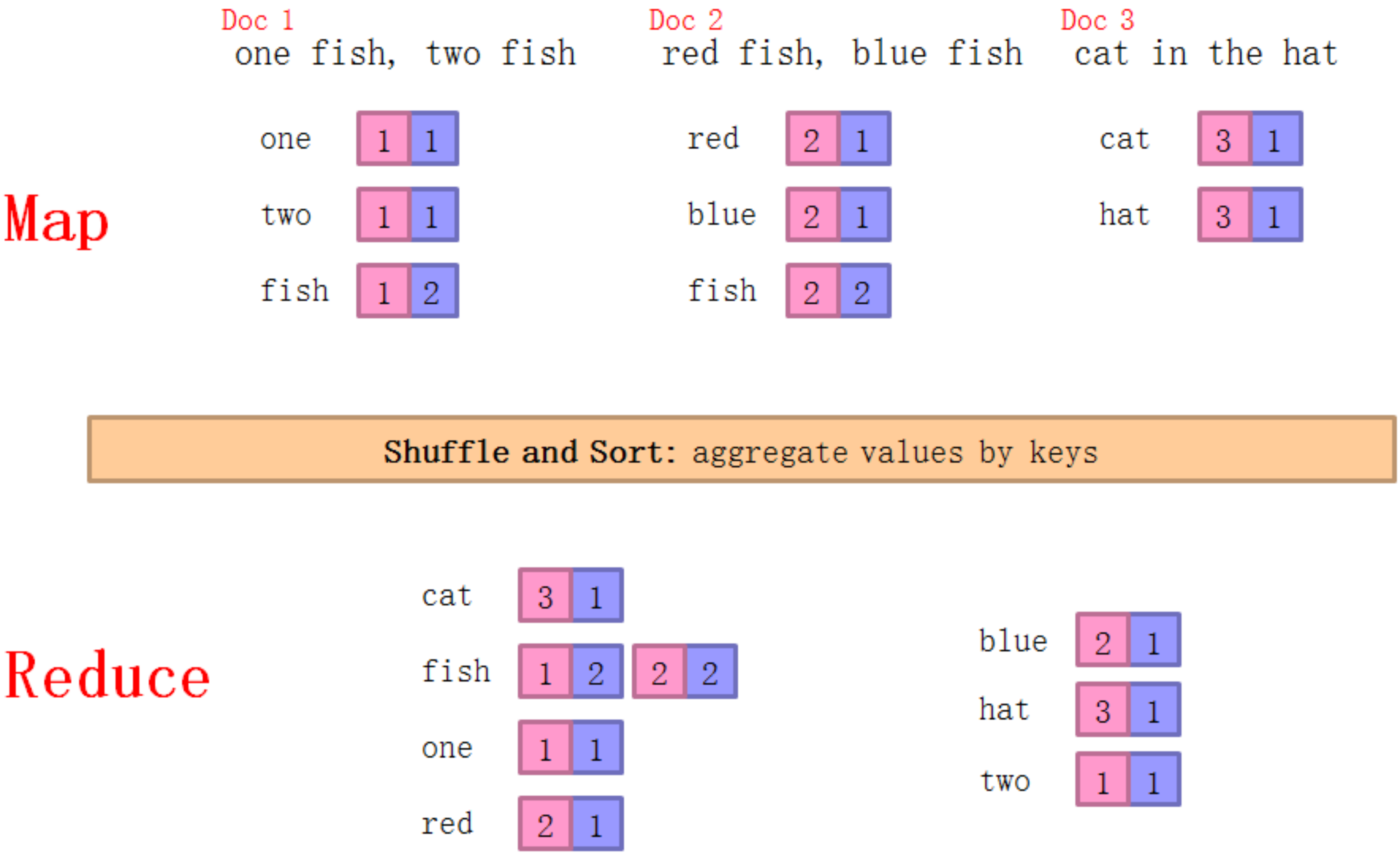
Doc 4 green eggs and ham



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



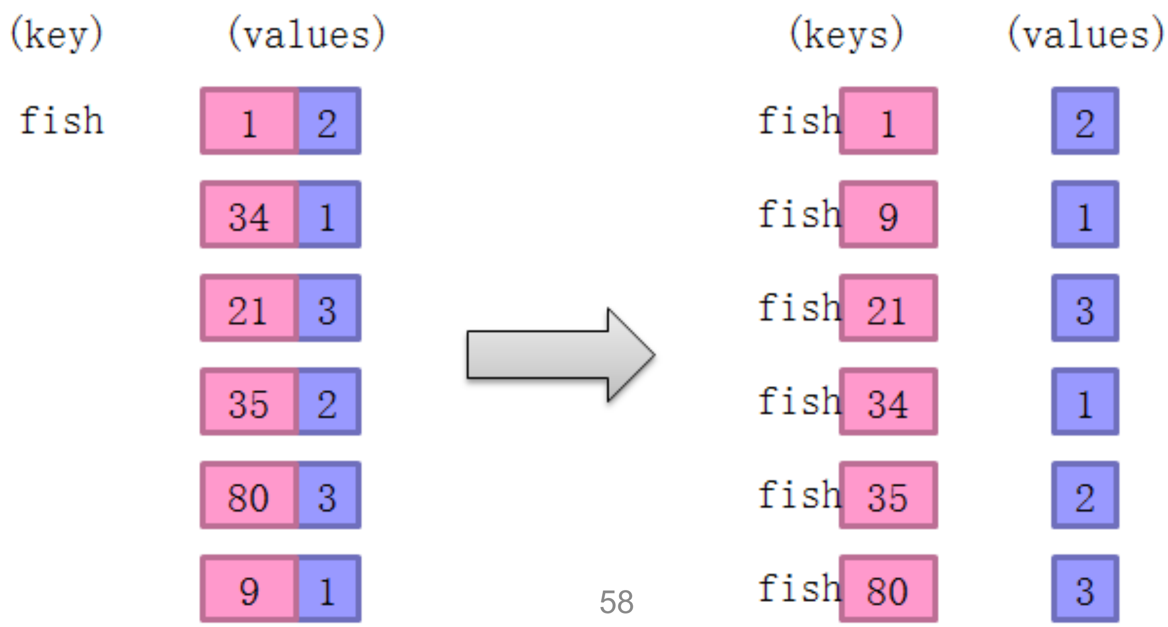
MapReduce: Index Construction

- 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

- Sorting docId in reducer is costly!
- However, key is always sorted by the framework…
  - Value-to-key conversion (Secondary sort)
  - Mapper output a stream of ([term, docid], tf) tuples

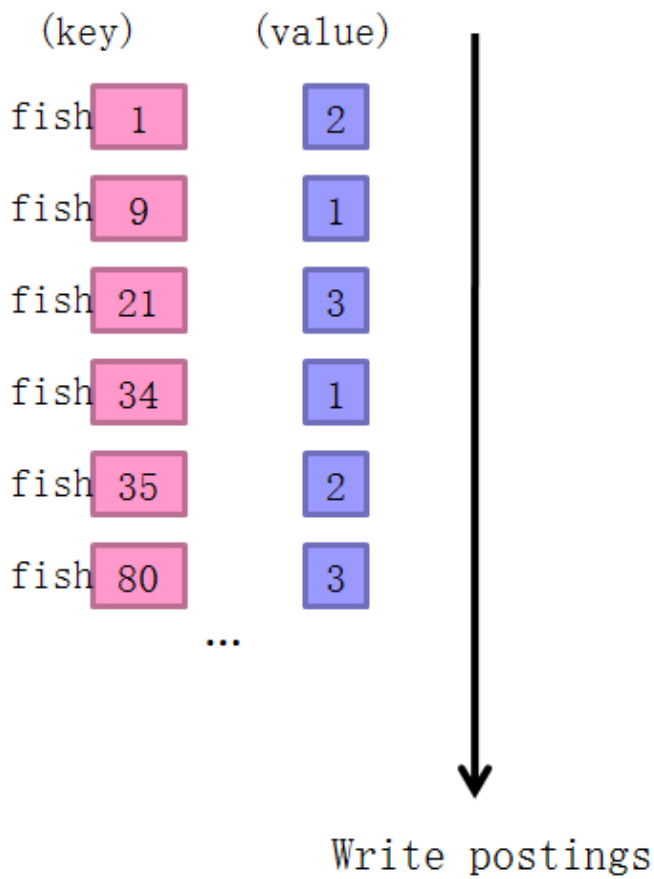


Secondary Sort

- Buffer values in memory, then sort
  - bad idea
- Value-to-key conversion
  - form composite intermediate key, (w_t, docId)
    - The mapper emits (w_t, docId) -> tf_i
  - Let execution framework do the sorting
  - Anything else we need to do?
    - All pairs associated with the same term are shuffled to the same reducer (use partitioner)

The Second Improvement

- How to avoid buffering all postings associated with key?

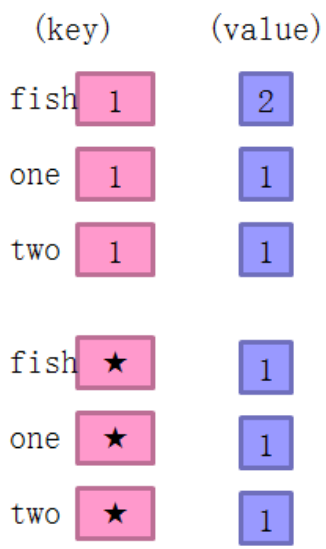


We'd like to store the DF at the front of the postings list

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

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

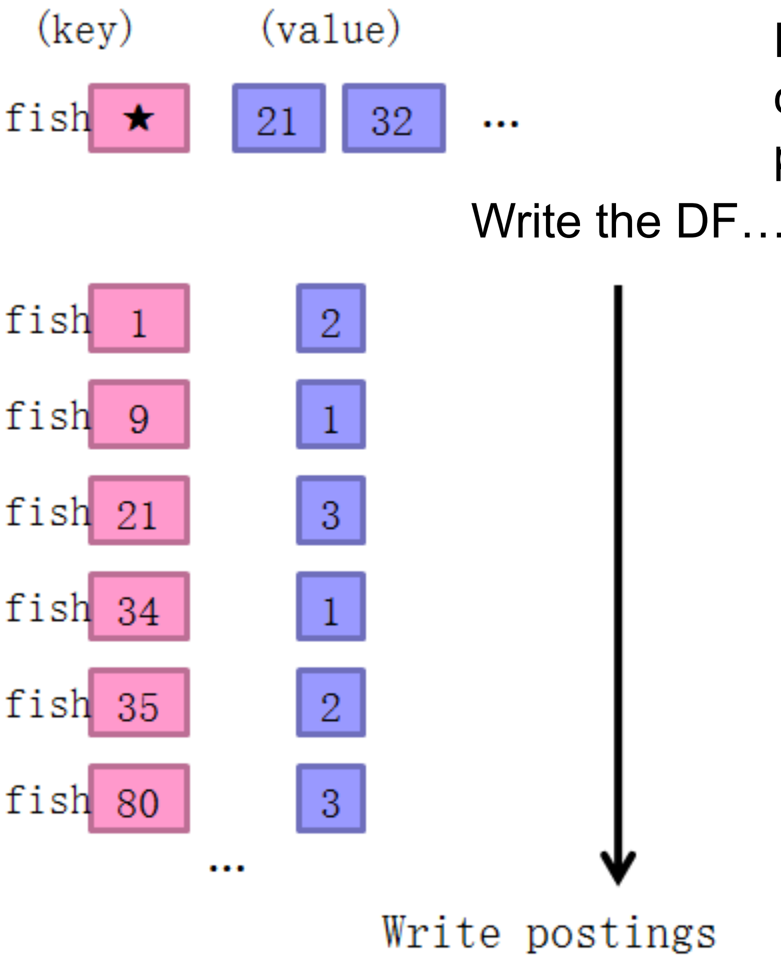


Emit normal key-value pairs…

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

Doc1: one fish, two fish

The Second Improvement



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!

Order Inversion

- The mapper:
  - additionally emits a “special” key of the form $(t_i, *)$
  - The value associated to the special key is 1
    - represents the contribution of the word pair to the marginal
  - 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 t_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

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