COMP9313: Big Data Management

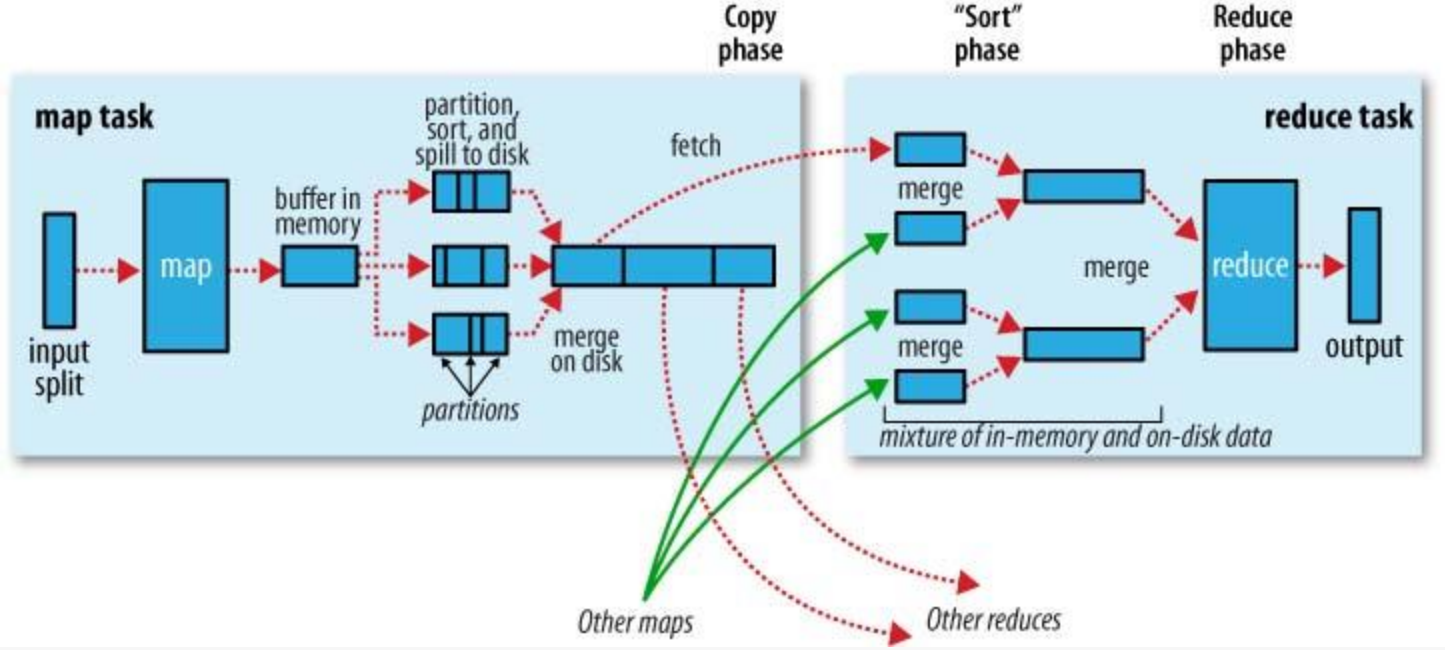
Lecturer: Siqing Li

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

Chapter 3.2: MapReduce IV

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



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.

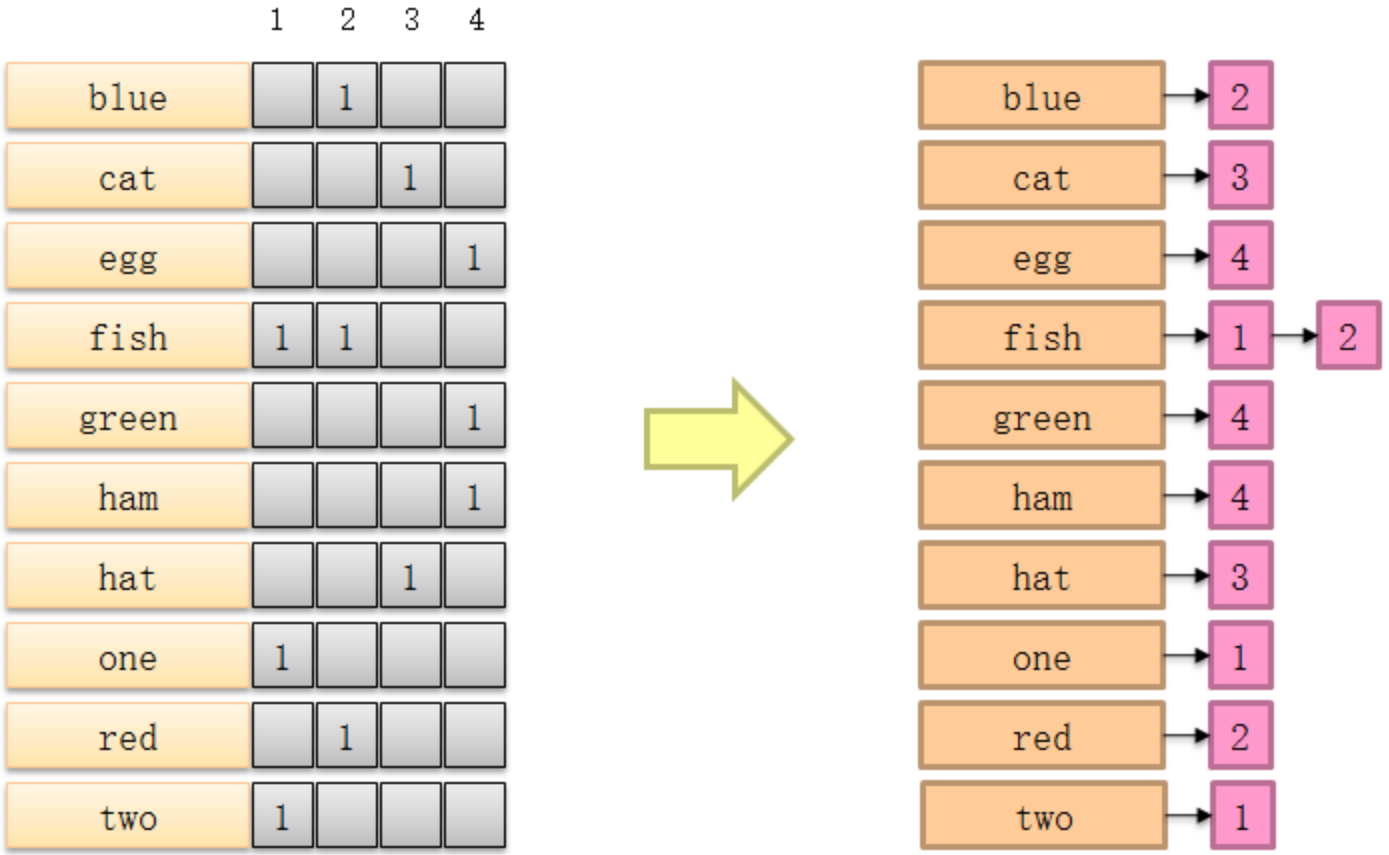
Boolean Text Retrieval: Inverted Index

Doc 1 one fish, two fish

Doc 2 red fish, blue fish

Doc 3 cat in the hat

Doc 4 green eggs and ham

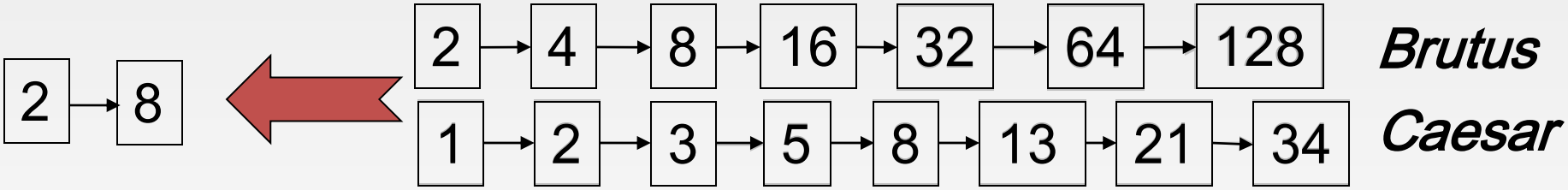


Search Using Inverted Index

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

Perfect for MapReduce!

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 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
  - Input and output of map and reduce?

Inverted Index: TF-IDF

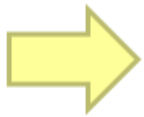
Doc 1 one fish, two fish

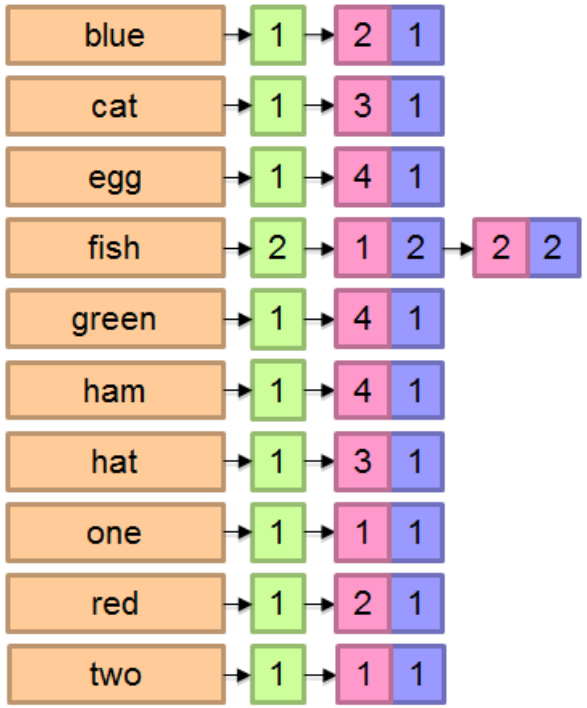
Doc 2 red fish, blue fish

Doc 3 cat in the hat

Doc 4 green eggs and ham

	tf 1	2	3	4	*df*
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

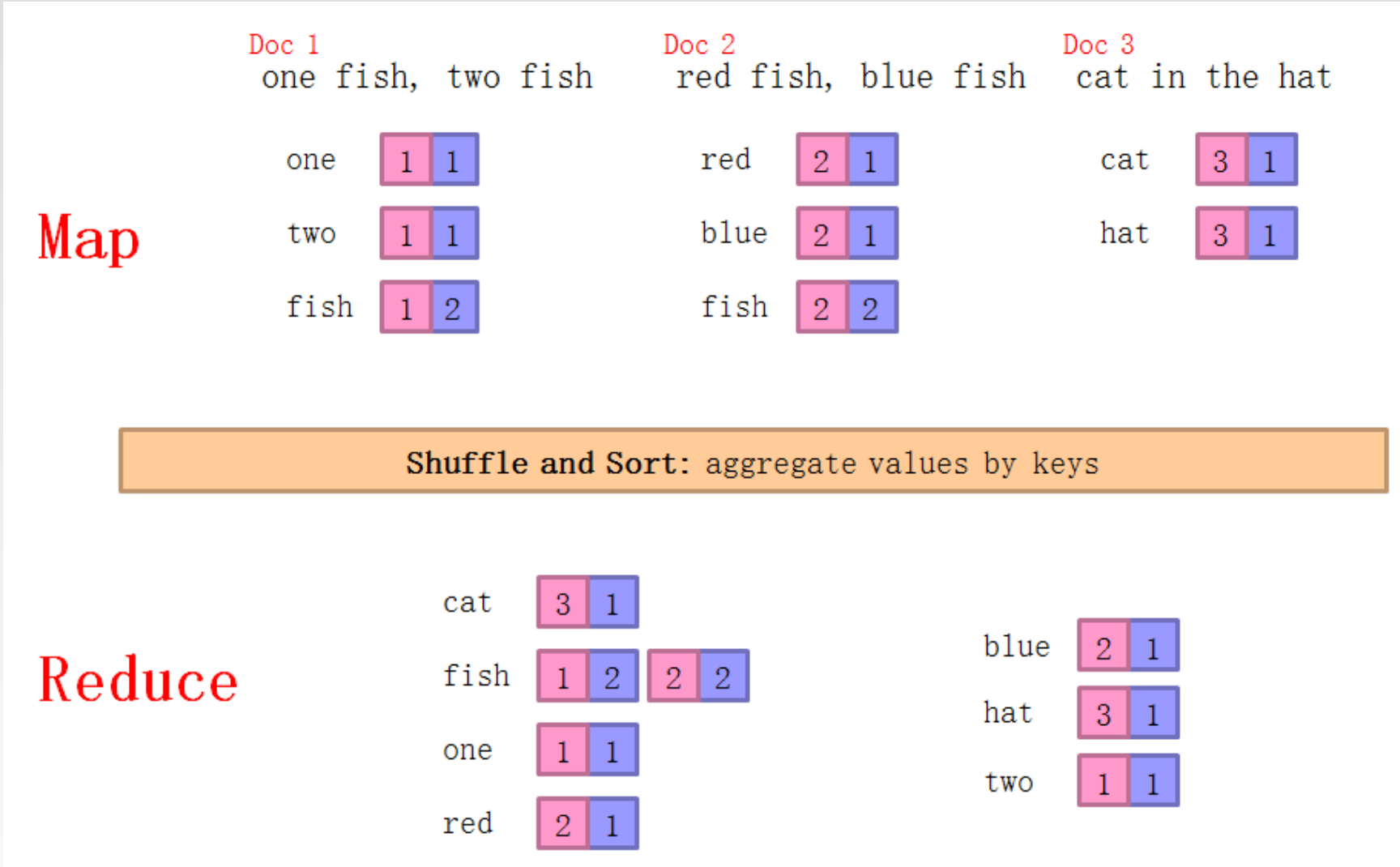




MapReduce: Index Construction

- 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

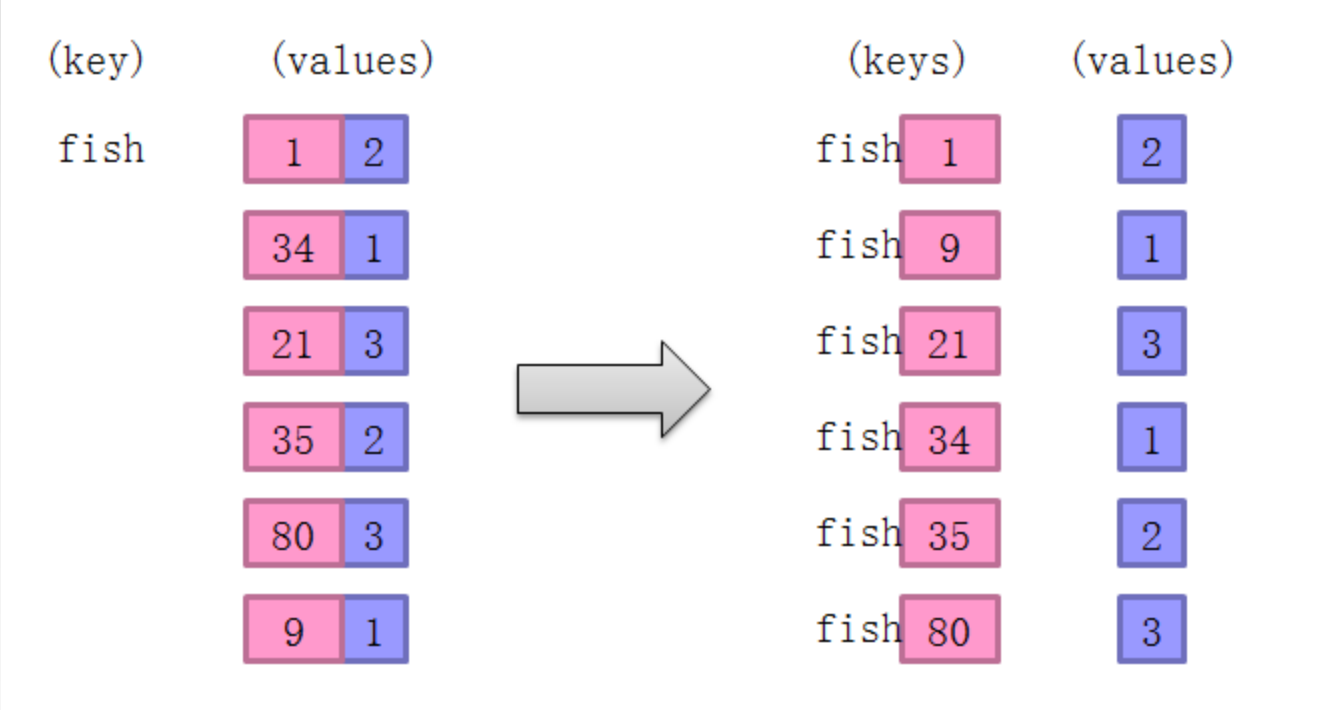


MapReduce: Index Construction

- 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

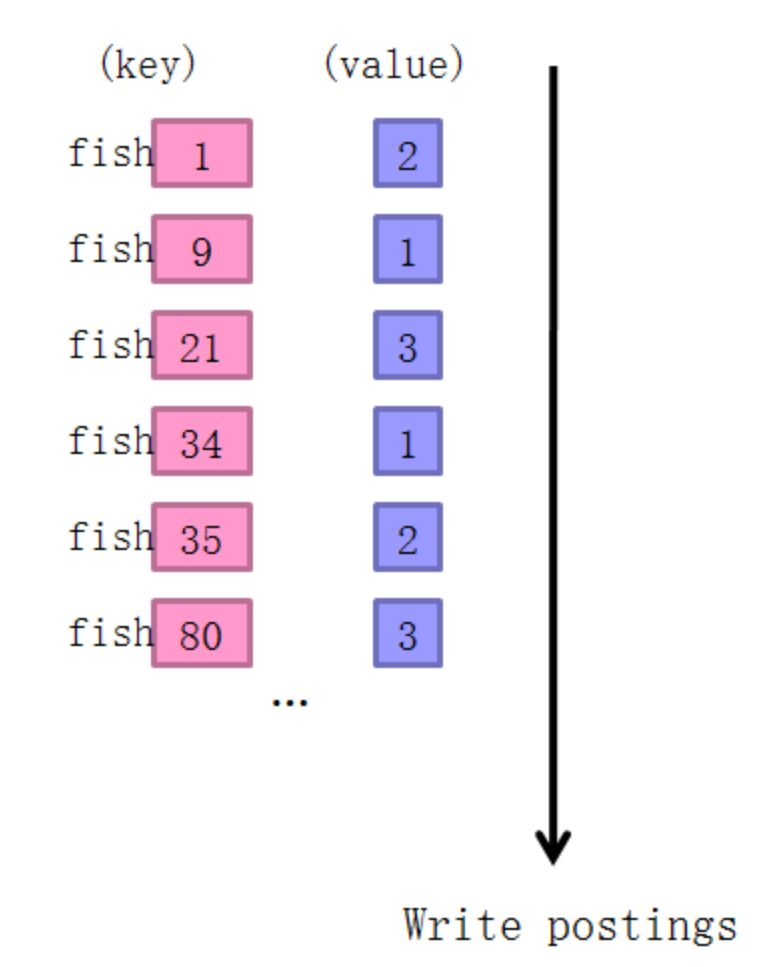
- How to make Hadoop sort the docid, instead of doing it in reducers?
- Design pattern: value-to-key conversion, secondary sort
- Mapper output a stream of ([term, docid], tf) tuples



- Remember: you must implement a partitioner on term!

The Second Improvement

- How to avoid buffering all postings associated with key?



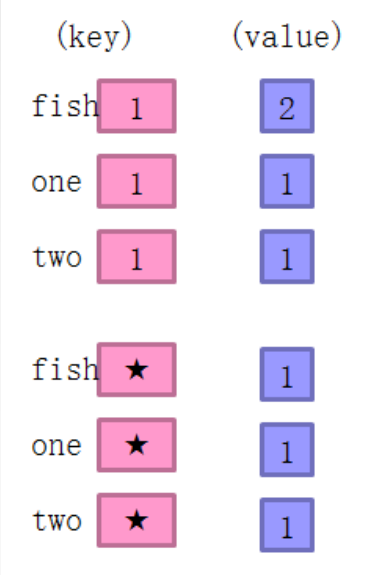
We'd like to store the DF at the front of the postings list

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

Sound familiar?
Design patter: 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!

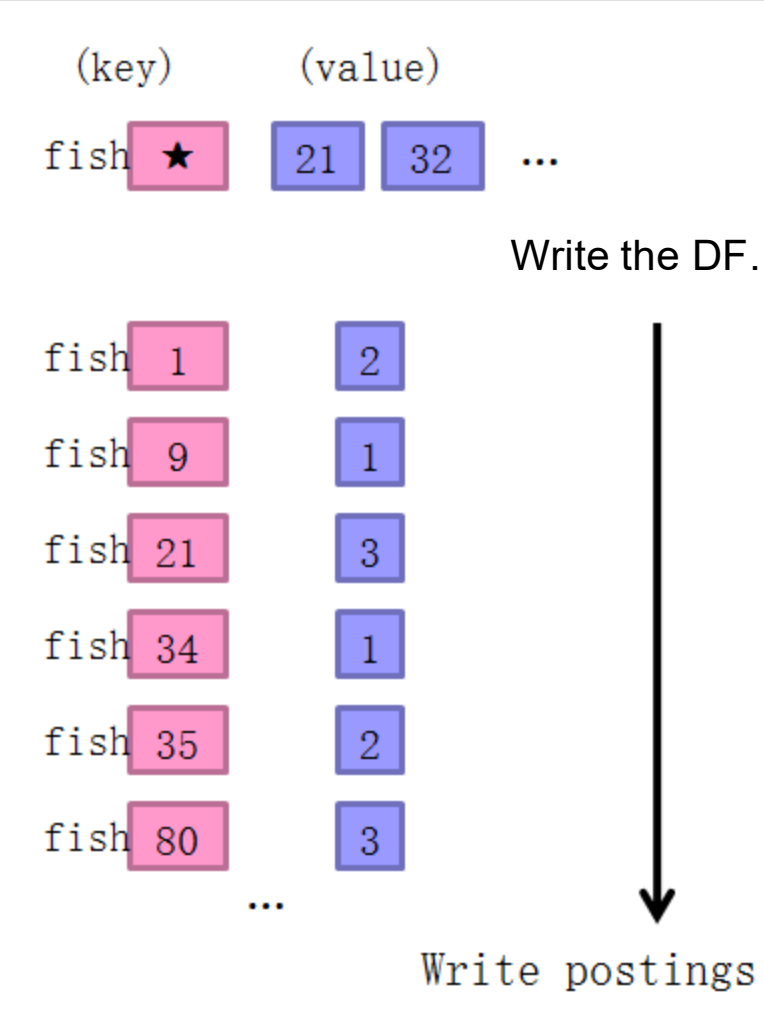


Emit normal key-value pairs…

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!

Chained MapReduce Job (MRJob)

- To define multiple steps, override steps() to return a list of MRSteps:

```
class MRMostUsedWord(MRJob):

    def mapper_get_words(self, _, line):
        # yield each word in the line
        for word in WORD_RE.findall(line):
            yield (word.lower(), 1)

    def combiner_count_words(self, word, counts):
        # sum the words we've seen so far
        yield (word, sum(counts))

    def reducer_count_words(self, word, counts):
        # send all (num_occurrences, word) pairs to the same reducer.
        # num_occurrences is so we can easily use Python's max() function.
        yield None, (sum(counts), word)

    # discard the key; it is just None
    def reducer_find_max_word(self, _, word_count_pairs):
        # each item of word_count_pairs is (count, word),
        # so yielding one results in key=counts, value=word
        yield max(word_count_pairs)

    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_words,
                   combiner=self.combiner_count_words,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.reducer_find_max_word)
        ]
```

Practices

Practice: Design MapReduce Algorithms

- Counting total enrollments of two specified courses

- Input Files: A list of students with their enrolled courses

  Jamie: COMP9313, COMP9318

  Tom: COMP9331, COMP9313

  … …

- Mapper selects records and outputs initial counts
  - Input: Key – student, value – a list of courses
  - Output: (COMP9313, 1), (COMP9318, 1), …

- Reducer accumulates counts
  - Input: (COMP9313, [1, 1, …]), (COMP9318, [1, 1, …])
  - Output: (COMP9313, 16), (COMP9318, 35)

Practice: Design MapReduce Algorithms

- Remove duplicate records
- Input: a list of records

  2013-11-01 aa
  2013-11-02 bb
  2013-11-03 cc
  2013-11-01 aa
  2013-11-03 dd

- Mapper
  - Input (record_id, record)
  - Output (record, “”)
    - E.g., (2013-11-01 aa, “”), (2013-11-02 bb, “”), …
- Reducer
  - Input (record, [“”, “”, “”, …])
    - E.g., (2013-11-01 aa, [“”, “”]), (2013-11-02 bb, [“”]), …
  - Output (record, “”)

Practice: Design MapReduce Algorithms

- Calculate the common friends for each pair of users in Facebook. Assume the friends are stored in format of Person->[List of Friends], e.g.: A -> [B C D], B -> [A C D E], C -> [A B D E], D -> [A B C E], E -> [B C D]. Note that the "friendship" is bi-directional, which means that if A is in B's list, B would be in A's list as well. Your result should be like:
  - (A B) -> (C D)
  - (A C) -> (B D)
  - (A D) -> (B C)
  - (B C) -> (A D E)
  - (B D) -> (A C E)
  - (B E) -> (C D)
  - (C D) -> (A B E)
  - (C E) -> (B D)
  - (D E) -> (B C)

Practice: Design MapReduce Algorithms

- Mapper:
  - Input(user u, List of Friends [f_1, f_2, …,])
  - map(): for each friend f_i, emit (<u, f_i>, List of Friends [f_1, f_2, …,])
    - Need to generate the pair <u, f_i> according to an order! Thus <u, fi> and <fi, u> will be the same key

- Reducer:
  - Input(user pair, list of friends lists[])
  - Get the intersection from all friends lists

- Example: http://scaryscientist.blogspot.com/2015/04/common-friends-using-mapreduce.html

Practice: Design MapReduce Algorithms

- Assume that in an online shopping system, a huge log file stores the information of each transaction. Each line of the log is in format of "userID\t product\t price\t time". Your task is to use MapReduce to find out the top-5 expensive products purchased by each user in 2016

- Mapper:
  - Input(transaction_id, transaction)
  - mapper_init(): initialize an associate array H(UserID, priority queue Q of log record based on price)
  - map(): get local top-5 for each user
  - mapper_final(): emit the entries in H

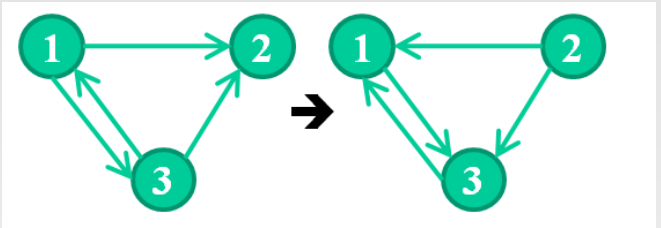
- Reducer:
  - Input(userID, list of queues[])
  - get top-5 products from the list of queues

Practice: Design MapReduce Algorithms

- Reverse graph edge directions & output in node order

- Input: adjacency list of graph (3 nodes and 4 edges)

(3, [1, 2])
(1, [2, 3]) ➔ (2, [1, 3])
(1, [3])
(3, [1])



- Note, the node_ids in the output values are also sorted. But Hadoop only sorts on keys!

- Solutions: Secondary sort

Practice: Design MapReduce Algorithms

- Map
  - Input: (3, [1, 2]), (1, [2, 3]).
  - Intermediate: (1, [3]), (2, [3]), (2, [1]), (3, [1]). (reverse direction)
  - Output: (<1, 3>, [3]), (<2, 3>, [3]), (<2, 1>, [1]), (<3, 1>, [1]).
    - Copy node_ids from value to key.
- Partition on Key.field1, and Sort on whole Key (both fields)
  - Input: (<1, 3>, [3]), (<2, 3>, [3]), (<2, 1>, [1]), (<3, 1>, [1])
  - Output: (<1, 3>, [3]), (<2, 1>, [1]), (<2, 3>, [3]), (<3, 1>, [1])
- Reducer
  - Merge according to part of the key
  - You need to preserve the state across input key-value pairs
  - Output: (1, [3]), (2, [1, 3]), (3, [1])

Practice: Design MapReduce Algorithms

- Given a large text dataset, find the top-k frequent terms (considering that you can utilize multiple reducers, and the efficiency of your method is evaluated).

- Two rounds: first round compute term frequency in multiple reducers, and each reducer only stores local top-k. Second round get the local top-k and compute the final top-k using a single reducer.

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 3.2