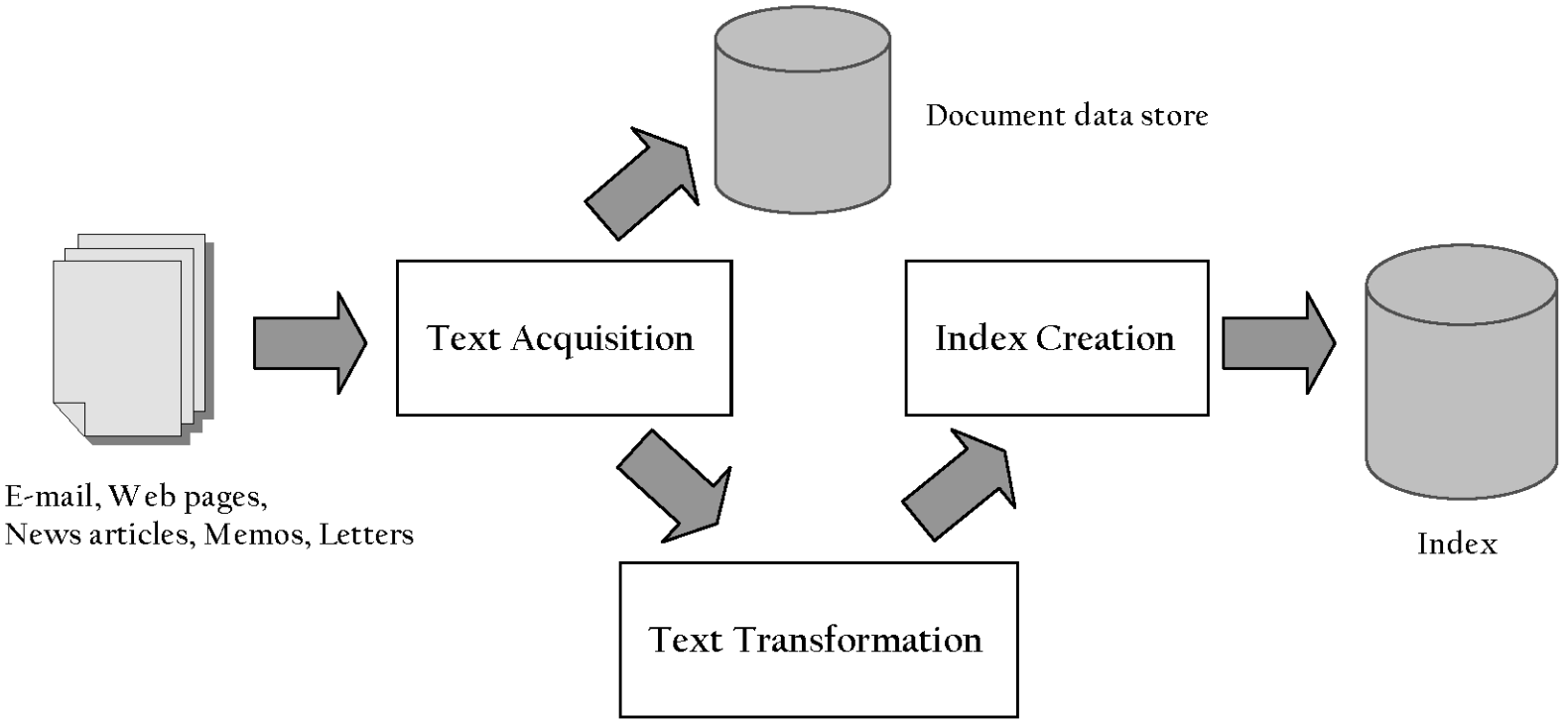
Indexing

- UCSB 293S, 2017
- Mainly based on slides from the text books of Croft/Metzler/Strohman and Manning/Raghavan/Schutze

Table of Content

- **Inverted index with positional information**
- **Compression**
- **Distributed indexing**

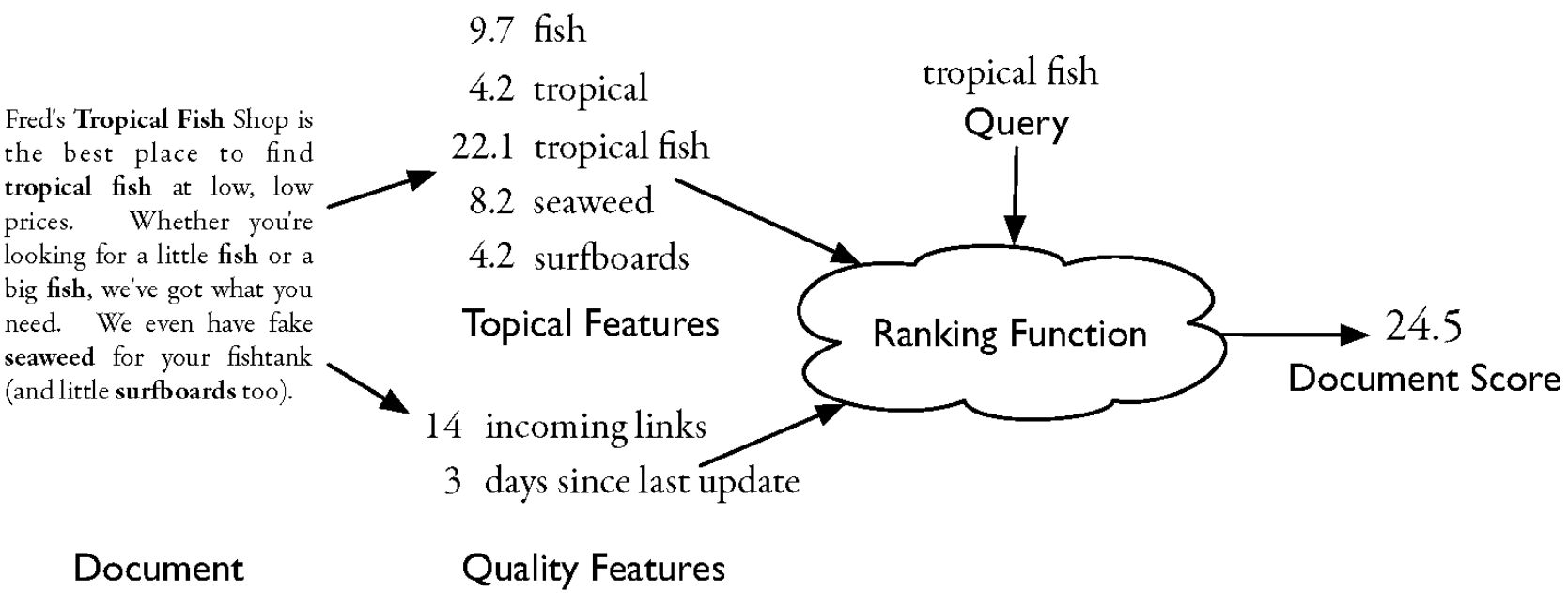
Indexing Process



Indexes

- ***Indexes* are data structures designed to make search faster**
- **Most common data structure is *inverted index***
  - general name for a class of structures
  - “inverted” because documents are associated with words, rather than words with documents
    - similar to a *concordance*
- **What is a reasonable abstract model for ranking?**
  - enables discussion of indexes without details of retrieval model

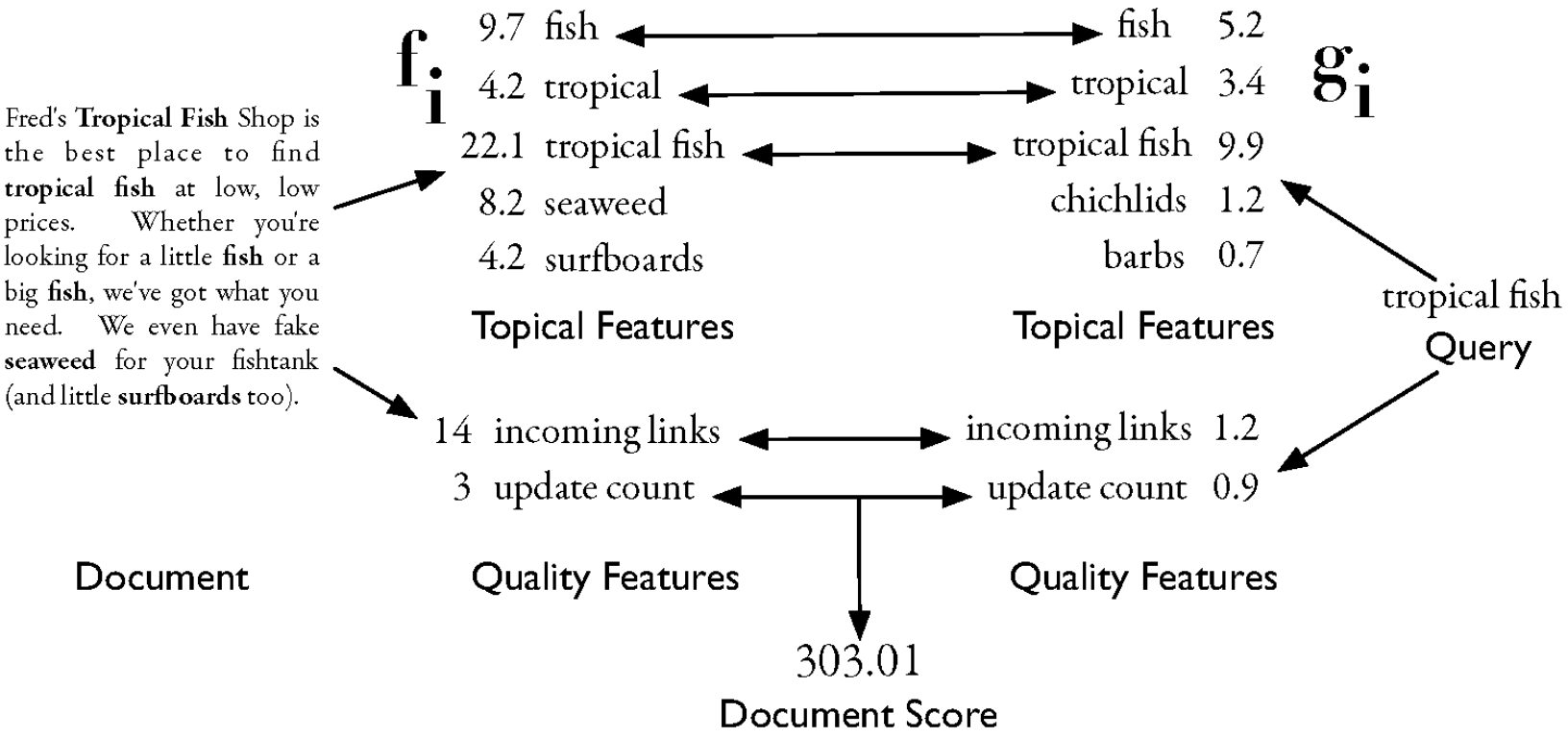
Simple Model of Ranking



More Concrete Model

$$R(Q,D) = \sum_i g_i(Q) f_i(D)$$

f_i is a document feature function
g_i is a query feature function



Inverted Index

- **Each index term is associated with an *inverted list***
  - Contains lists of documents, or lists of word occurrences in documents, and other information
  - Each entry is called a *posting*
  - The part of the posting that refers to a specific document or location is called a *pointer*
  - Each document in the collection is given a unique number
  - Lists are usually *document-ordered* (sorted by document number)

Example "Collection"

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

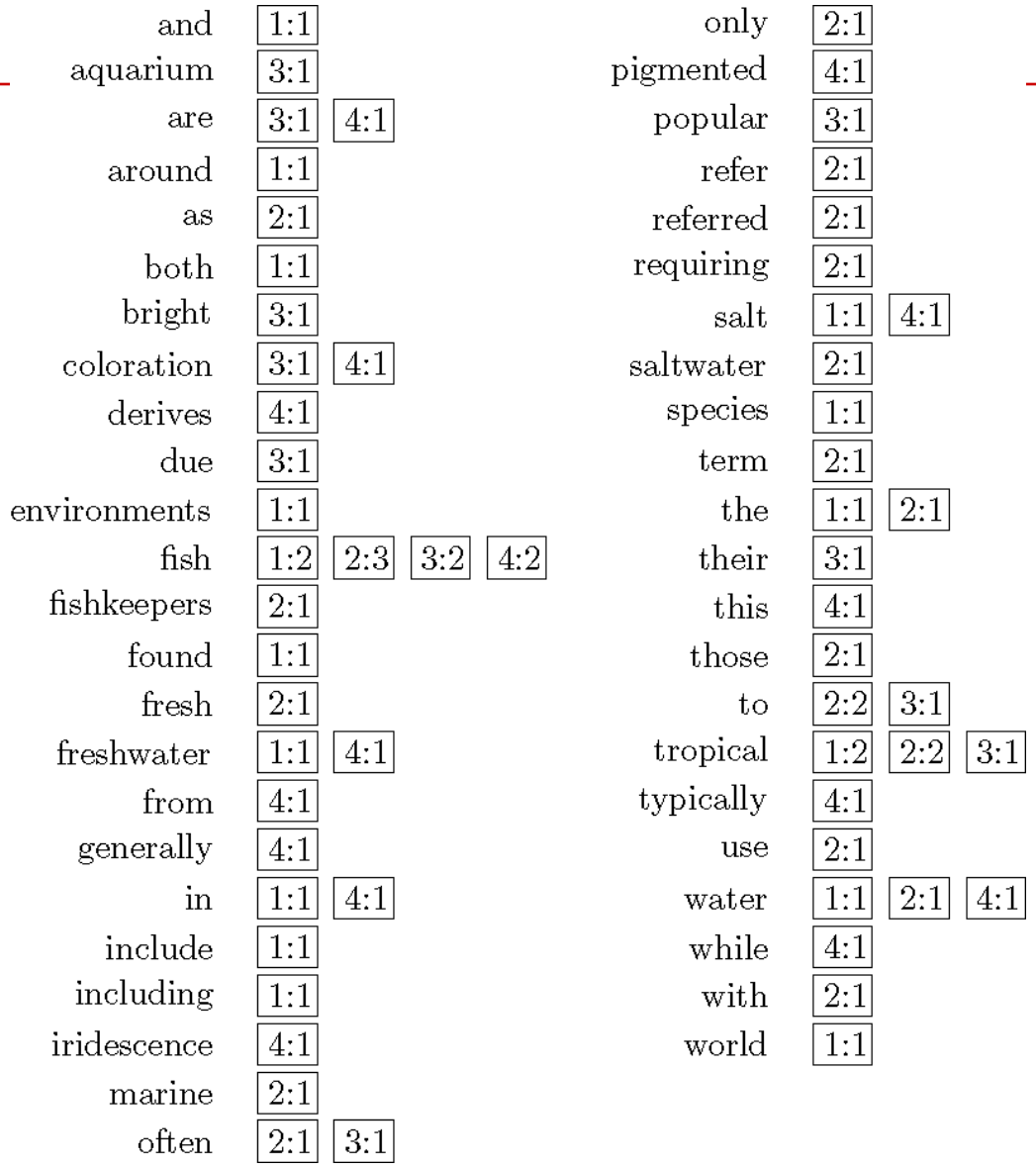
Four sentences from the Wikipedia entry for *tropical fish*

Simple Inverted Index

and	1
aquarium	3
are	3 4
around	1
as	2
both	1
bright	3
coloration	3 4
derives	4
due	3
environments	1
fish	1 2 3 4
fishkeepers	2
found	1
fresh	2
freshwater	1 4
from	4
generally	4
in	1 4
include	1
including	1
iridescence	4
marine	2
often	2 3
only	2
pigmented	4
popular	3
refer	2
referred	2
requiring	2
salt	1 4
saltwater	2
species	1
term	2
the	1 2
their	3
this	4
those	2
to	2 3
tropical	1 2 3
typically	4
use	2
water	1 2 4
while	4
with	2
world	1

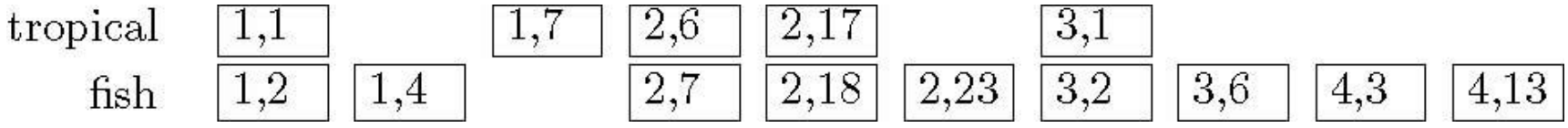
Inverted Index with counts

- supports better ranking algorithms



Proximity Matches

- **Matching phrases or words within a window explicitly or implicitly.**
  - e.g., "`tropical fish`", or “find tropical within 5 words of fish”
- **Word positions in inverted lists make these types of query features efficient**
  - e.g.,



Positional indexes

- **Store, for each *term*, entries of the form:**

  <number of docs containing ***term***;

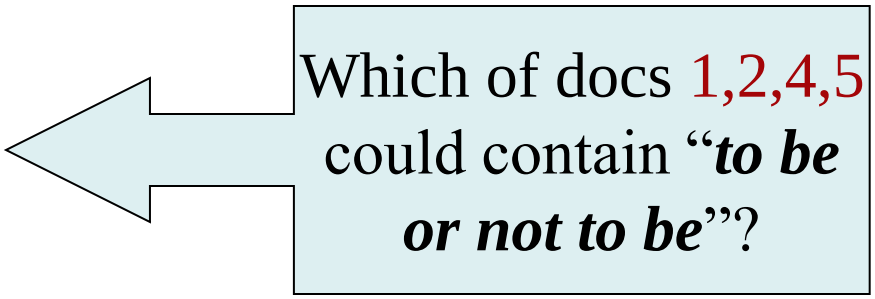
  doc1: position1, position2 … ;

  doc2: position1, position2 … ;

  etc.>

Positional index example

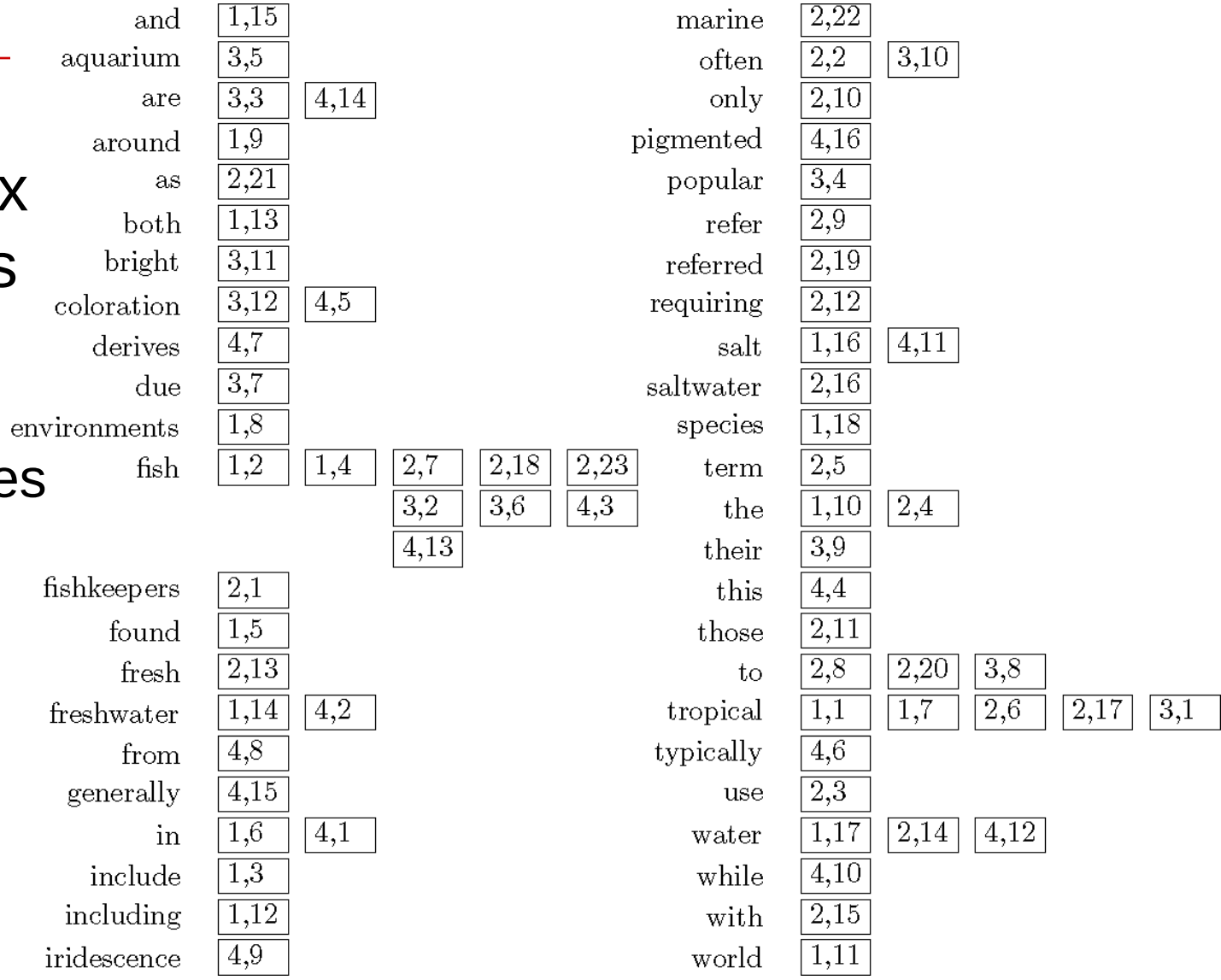
<***be***: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367, …>



- **this expands postings storage *substantially***

Inverted Index with positions

- supports proximity matches

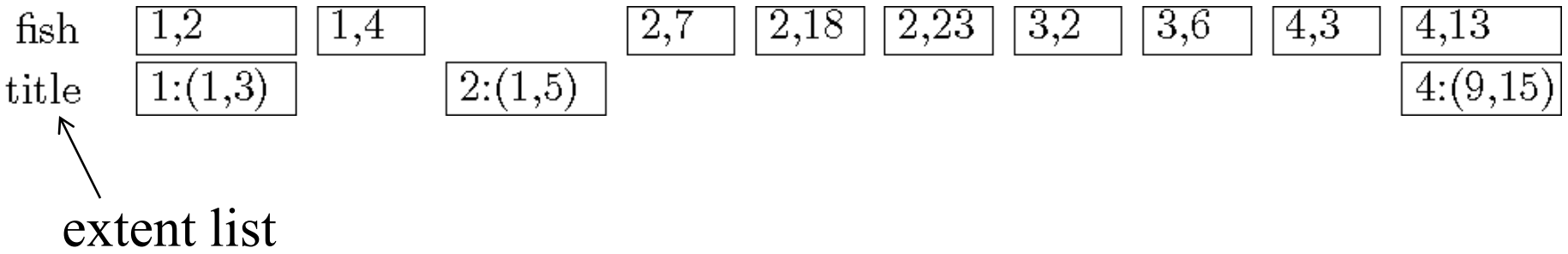


Fields and Extents

- **Document structure is useful in search**
  - *field* restrictions
    - e.g., date, from:, etc.
  - some fields more important
    - e.g., title
- **Options:**
  - separate inverted lists for each field type
  - add information about fields to postings
  - use *extent lists to mark special areas in a document*

Extent Lists

- **An *extent* is a contiguous region of a document**
  - represent extents using word positions
  - inverted list records all extents for a given field type
  - e.g.
    - 1:(1,3) → title in document 1 is from 1 to 3

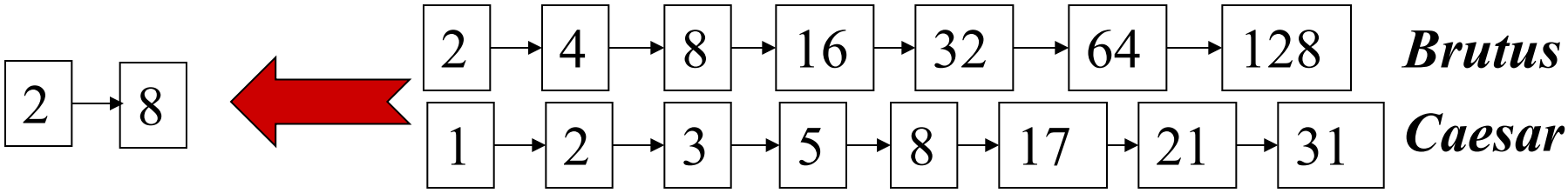


Other Issues

- **Precomputed scores in inverted list**
  - e.g., list for “fish” [(1:3.6), (3:2.2)], where 3.6 is total feature value for document 1
  - improves speed but reduces flexibility
- **Score-ordered lists**
  - query processing engine can focus only on the top part of each inverted list, where the highest-scoring documents are recorded
  - very efficient for single-word queries

Basic merge

- **Walk through the two postings simultaneously, in time linear in the total number of postings entries**



If the list lengths are *m* and *n*, the merge takes O(*m*+*n*) operations.

Can we do better?
Yes, if index isn't changing too fast.

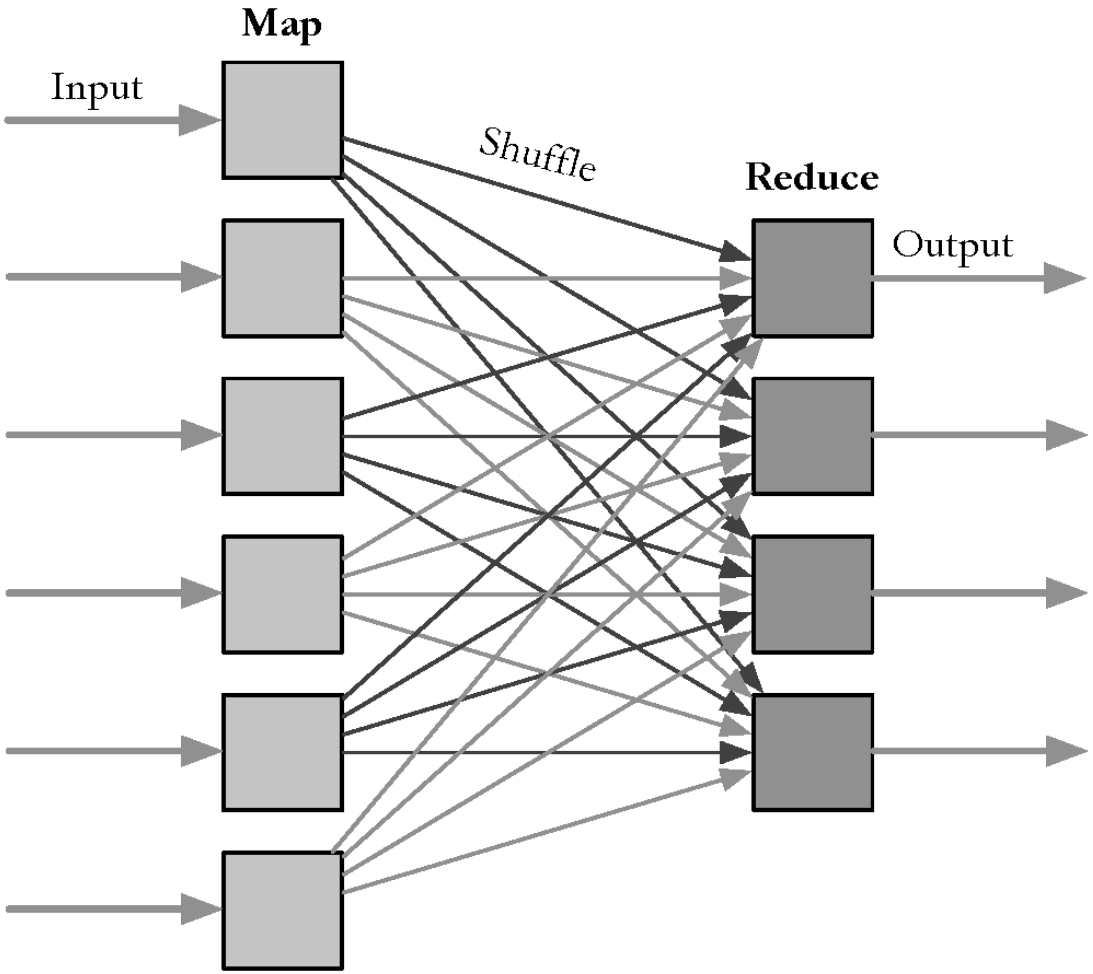
Distributed Indexing

- **Distributed processing driven by need to index and analyze huge amounts of data (i.e., the Web)**
- **Large numbers of inexpensive servers used rather than larger, more expensive machines**
- ***MapReduce* is a distributed programming tool designed for indexing and analysis tasks**

MapReduce

- **Distributed programming framework that focuses on data placement and distribution**
- ***Mapper***
  - Generally, transforms a list of items into another list of items of the same length
- ***Reducer***
  - Transforms a list of items into a single item
  - Definitions not so strict in terms of number of outputs
- **Many mapper and reducer tasks on a cluster of machines**

MapReduce



MapReduce

- **Basic process**
  - *Map* stage which transforms data records into pairs, each with a key and a value
  - *Shuffle* uses a hash function so that all pairs with the same key end up next to each other and on the same machine
  - *Reduce* stage processes records in batches, where all pairs with the same key are processed at the same time
- ***Idempotence* of Mapper and Reducer provides fault tolerance**
  - multiple operations on same input gives same output

Indexing Example

```
procedure MapDocumentsToPostings(input)
    while not input.done() do
        document ← input.next()
        number ← document.number
        position ← 0
        tokens ← Parse(document)
        for each word w in tokens do
            Emit(w, number:position)
            position = position + 1
        end for
    end while
end procedure

procedure ReducePostingsToLists(key, values)
    word ← key
    WriteWord(word)
    while not input.done() do
        EncodePosting(values.next())
    end while
end procedure
```