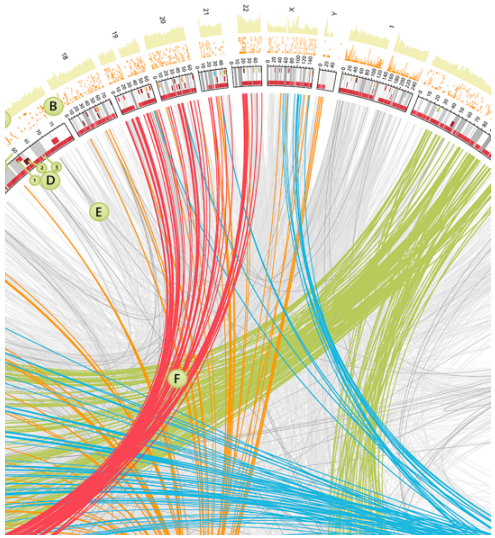
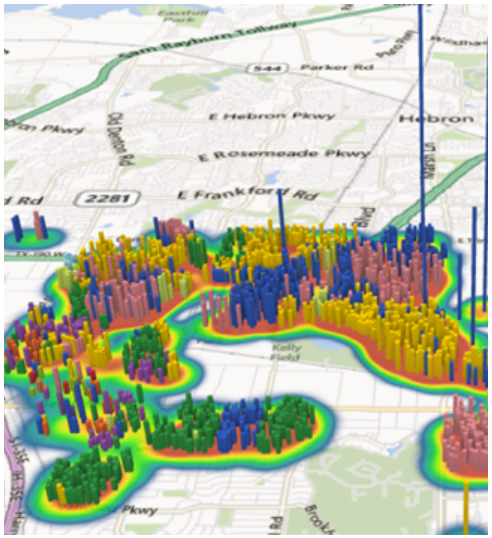
Big Data Analytics - Information retrieval



Prof. Nastaran Fatemi | HES-SO Yverdon-les-Bains (HEIG-VD)

Plan

- Introduction to information retrieval
- Basics of indexing and retrieval
- Inverted indexing in MapReduce

Introduction

- Web search is the quintessential big-data problem
  - Given an information need expressed as a short query consisting of a few terms, the system's task is to retrieve relevant Web objects and present them to the user.
- How big is the web?
  - Difficult to compute exactly
  - A conservative estimation : several tens of billions of pages, or hundreds of terabytes
- Users demand results quickly from a search engine
  - query latencies longer than a few hundred milliseconds will try a user's patience
- Fulfilling these requirements is quite an engineering challenge!

Search applications

- Search and communication are most popular uses of the computer
- Applications involving search are everywhere
  - Web search is a prominent example, but there are also other applications,
  - Enterprise business applications (enterprise resource planning, supply chain management, compliance & discovery, etc.)
  - Personal information retrieval (email search, spam filtering, etc.)
- The field of computer science that is most involved with R&D for search is information retrieval

Basic concepts

- Information retrieval (IR)
  - Focus on textual information (= text/document retrieval)
  - Other possibilities include image, video, music, …
- What do we search?
  - Generically, “collections”
  - Less-frequently used, “corpora”
- What do we find?
  - Generically, “documents”
  - Even though we may be referring to web pages, PDFs, PowerPoint slides, paragraphs, etc.

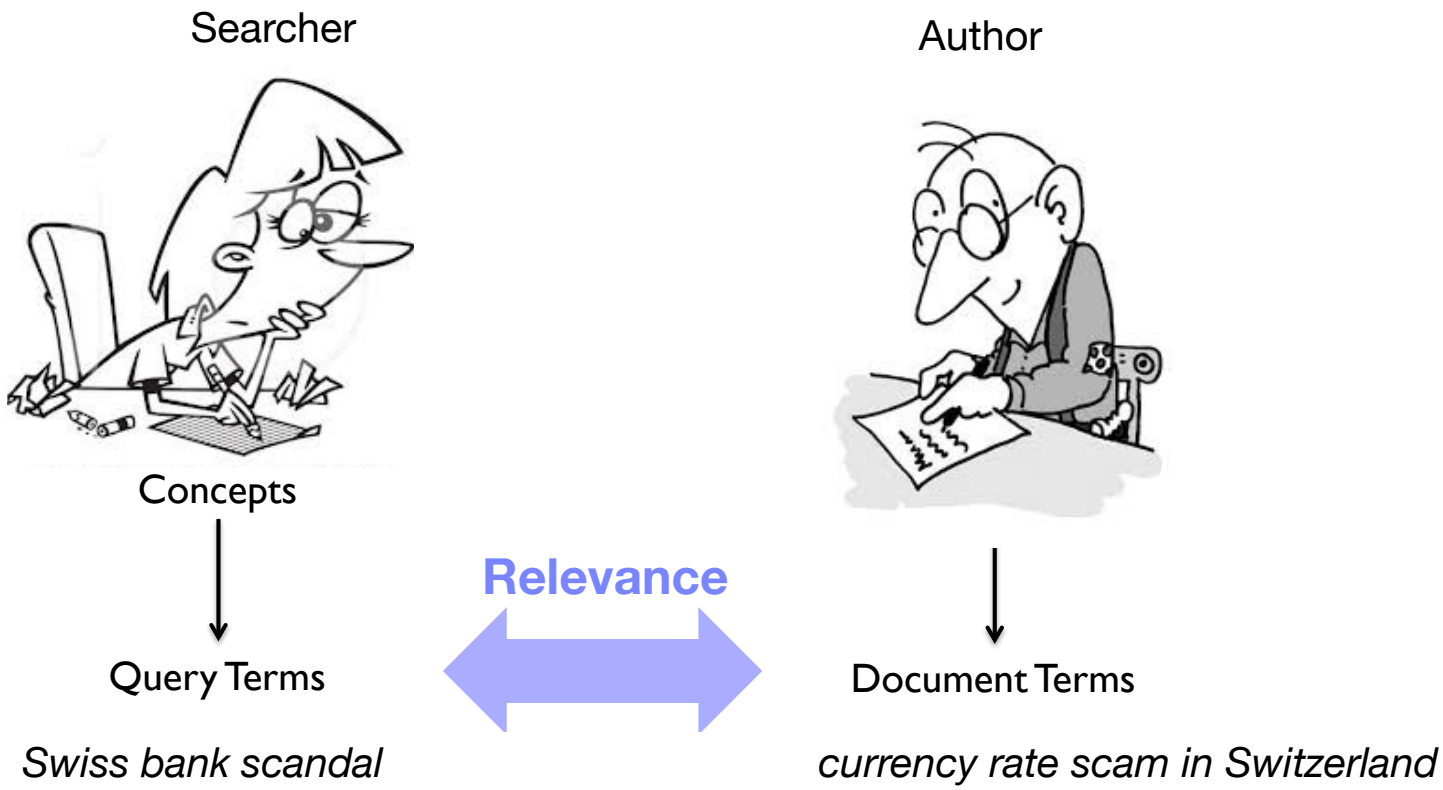
Documents vs. database records

- Database records (tuples in relational databases) are typically made up of well-defined attributes
  - E.g., bank records with account numbers, balances, names, addresses, social security numbers, etc.
  - Easy to compare fields with well-defined semantics to queries in order to find matches
- Example of bank database query
  - Find records with balance > 100,000 CHF in branches located in Zurich.
  - Matches easily found by comparison with field values of records
- Example of search engine query
  - "Swiss bank scandals"
  - This text must be compared to the text of entire news stories

Comparing text

- Defining the meaning of a word, a sentence, a paragraph, or a whole news story is much more difficult that defining an account number.
- Comparing the query text to the document text and determining what is a good match is the core issue of information retrieval
- Exact matching of words is not enough
  - Many different ways to write the same thing in a “natural language” like English
  - Some stories will be better matches than others

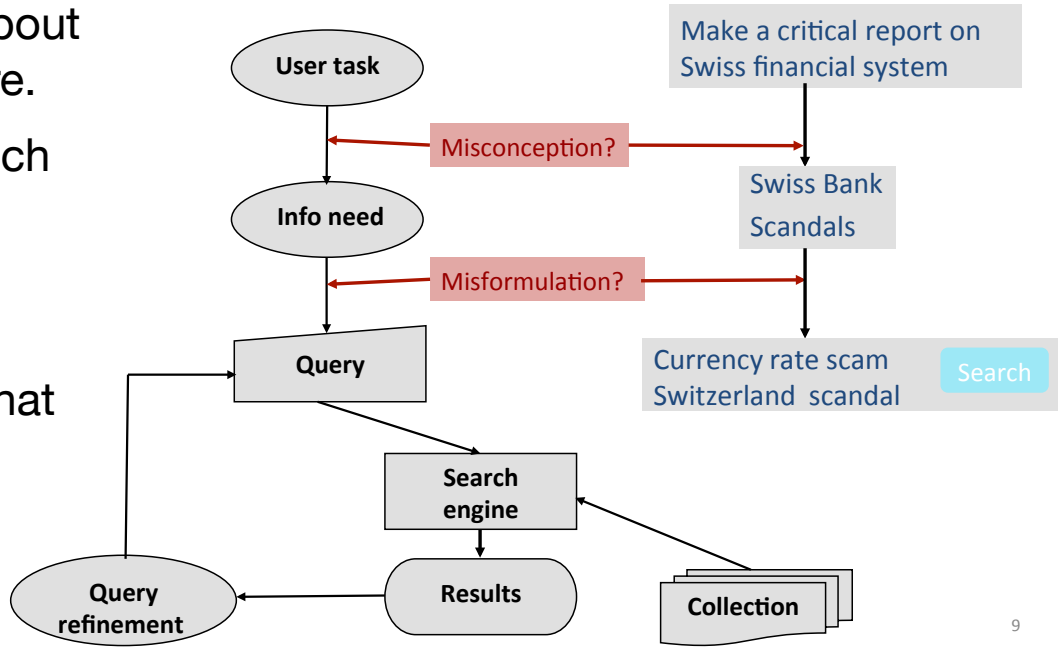
The Central Problem in Search



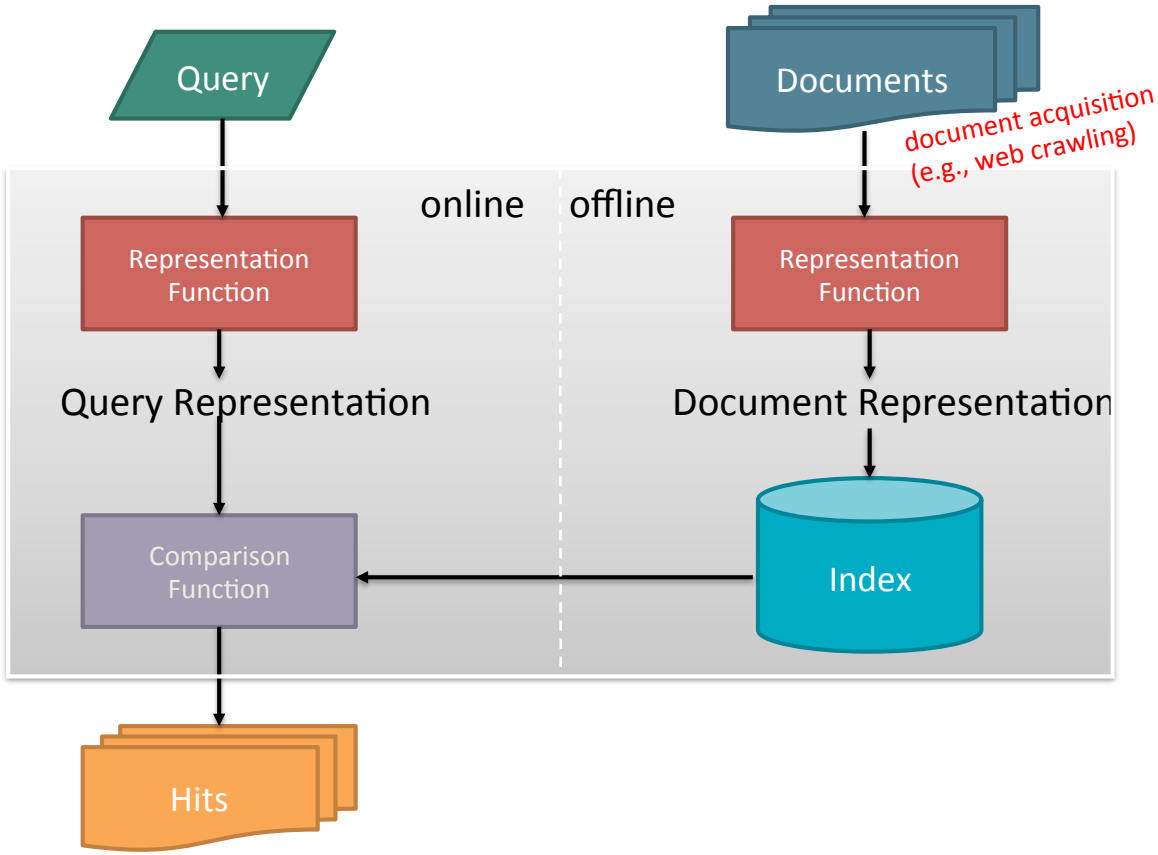
Do these represent the same concepts?

Query relevance

- An **information need** is the topic about which the user desires to know more.
- It is differentiated from a **query**, which is what the user conveys to the computer in an attempt to communicate the information need.
- A document is **relevant** if it is one that the user perceives as containing information of value with respect to their personal information need.



Abstract IR Architecture



11

How do we represent text?

- Computers don't "understand" anything!
- "Bag of words"
  - Treat all the words in a document as index terms
  - Assign a "weight" to each term based on "importance" (or, in simplest case, presence/absence of word)
  - Disregard order, structure, meaning, etc. of the words
  - Simple, yet effective!
- Assumptions
  - Term occurrence is independent
  - "Words" are well-defined

What's a word?

天主教教宗若望保祿二世因感冒再度住進醫院。
這是他今年第二度因同樣的病因住院。

وق ال مكارم فيجيري - النطاق بسام
الخارجية الإسرائيلية - إن شارون قبل
الدعوة وسيقوم للمرة الأولى بزيارة
تونس، التي كانت لفترة طويلة المقر
الرسمي لمنظمة التحرير الفلسطينية بعد خروجها من لبنان عام 1982.

Выступая в Мещанском суде Москвы экс-глава ЮКОСа заявил не совершал ничего противозаконного, в чем обвиняет его генпрокуратура России.

भारत सरकार ने आर्थिक सर्वेक्षण में वित्तीय वर्ष 2005-06 में सात फ़ीसदी विकास दर हासिल करने का आकलन किया है और कर सुधार पर ज़ोर दिया है

日米連合で台頭中国に対処…アーミテージ前副長官提言

조재영 기자= 서울시는 **25**일 이명박 시장이 `행정중심복합도시" 건설안에 대해 `군대라도 동원해 막고싶은 심정"이라고 말했다는 일부 언론의 보도를 부인했다.

Sample Document

McDonald's slims down spuds
Fast-food chain to reduce certain types of fat in its french fries with new cooking oil.

NEW YORK (CNN/Money) - McDonald's Corp. is cutting the amount of "bad" fat in its french fries nearly in half, the fast-food chain said Tuesday as it moves to make all its fried menu items healthier.

But does that mean the popular shoestring fries won't taste the same? The company says no. "It's a win-win for our customers because they are getting the same great french-fry taste along with an even healthier nutrition profile," said Mike Roberts, president of McDonald's USA.

But others are not so sure. McDonald's will not specifically discuss the kind of oil it plans to use, but at least one nutrition expert says playing with the formula could mean a different taste.

Shares of Oak Brook, Ill.-based McDonald's (MCD: down $0.54 to $23.22, Research, Estimates) were lower Tuesday afternoon. It was unclear Tuesday whether competitors Burger King and Wendy's International (WEN: down $0.80 to $34.91, Research, Estimates) would follow suit. Neither company could immediately be reached for comment.

...

“Bag of Words”

14 × McDonalds

12 × fat

11 × fries

8 × new

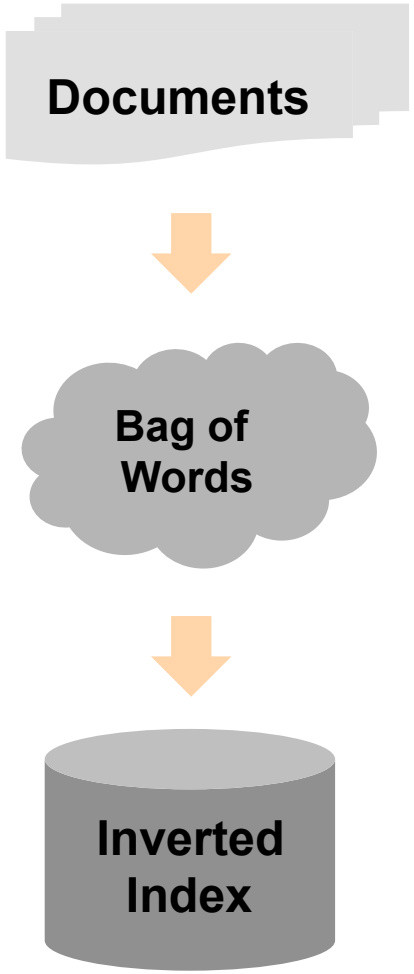
7 × french

6 × company, said, nutrition

5 × food, oil, percent, reduce, taste, Tuesday

…

Counting Words…



case folding, tokenization, stopword removal, stemming

syntax, semantics, world knowledge, etc.

Boolean Retrieval

- Users express queries as a Boolean expression
  - AND, OR, NOT
- Can be arbitrarily nested
- Retrieval is based on the notion of sets
  - Any given query divides the collection into two sets: retrieved, not-retrieved
  - Pure Boolean systems do not define an ordering of the results

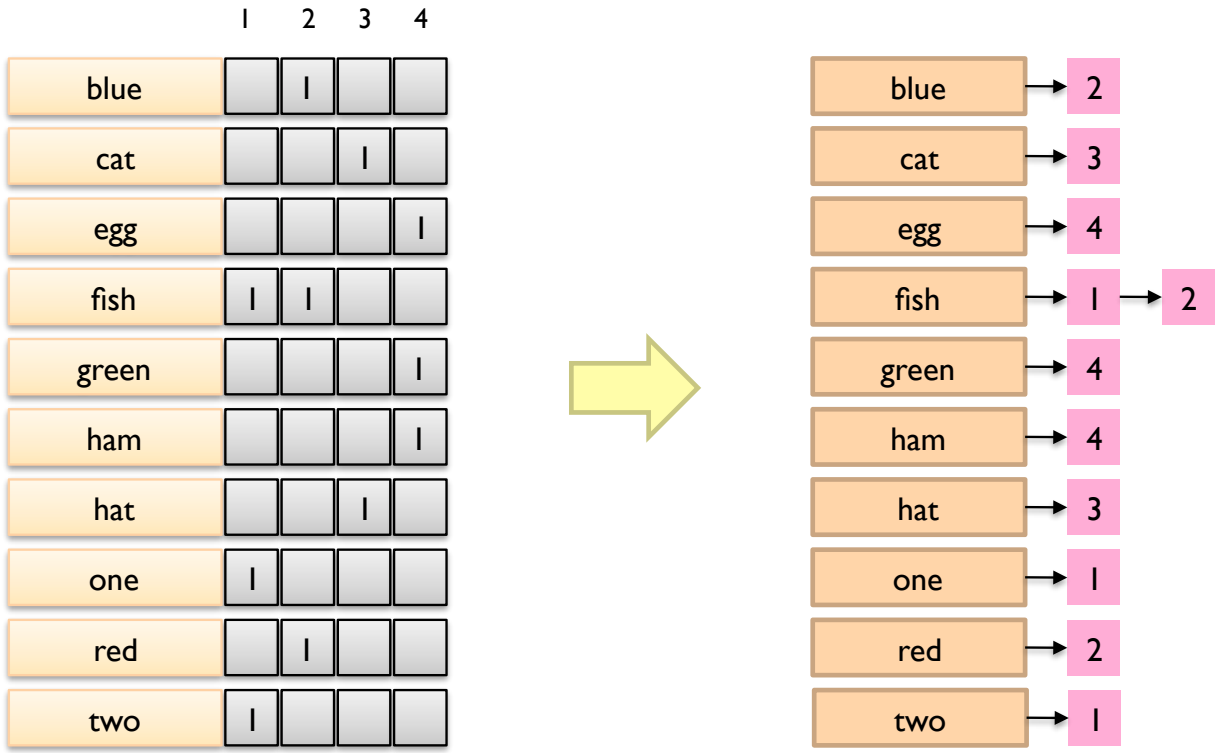
Inverted Index: Boolean Retrieval

Doc 1 one fish, two fish

Doc 2 red fish, blue fish

Doc 3 cat in the hat

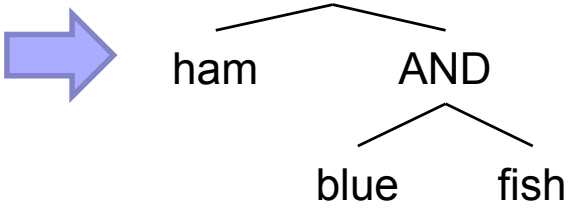
Doc 4 green eggs and ham



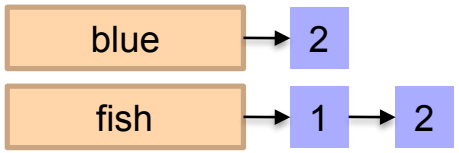
Boolean Retrieval

- To execute a Boolean query:
  - Build query syntax tree

(blue AND fish) OR ham



  - For each clause, look up postings

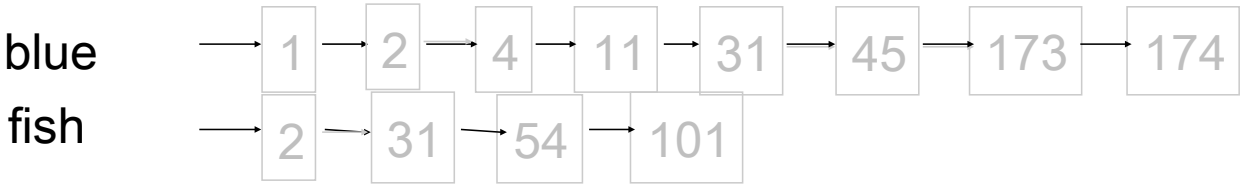


  - Traverse postings and apply Boolean operator

Simple conjunctive query (two terms)

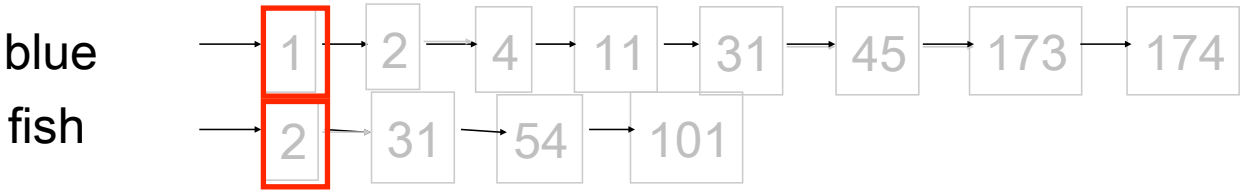
- Consider processing the query:
  - *blue* **AND** *fish*
- To find all matching documents using inverted index:
  - Locate *blue* in the Dictionary
  - Retrieve its postings list from the postings file
  - Locate *fish* in the Dictionary
  - Retrieve its postings list from the postings file
  - Intersect the the two postings lists
  - Return intersection to user

Intersecting two posting lists



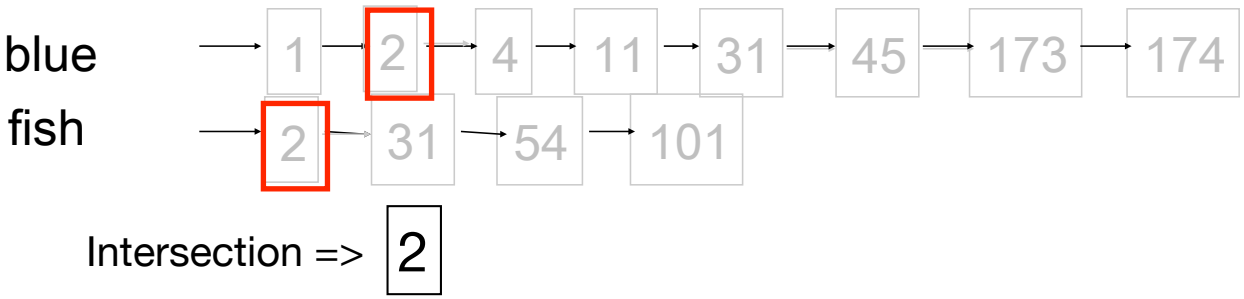
Intersection =>

Intersecting two posting lists

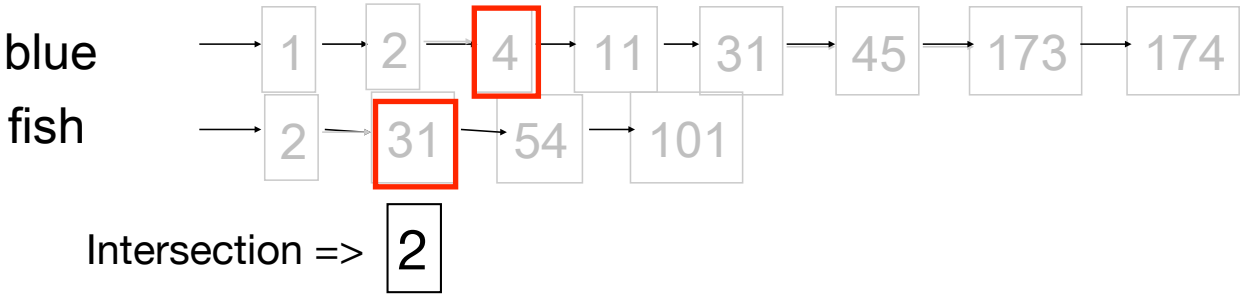


Intersection =>

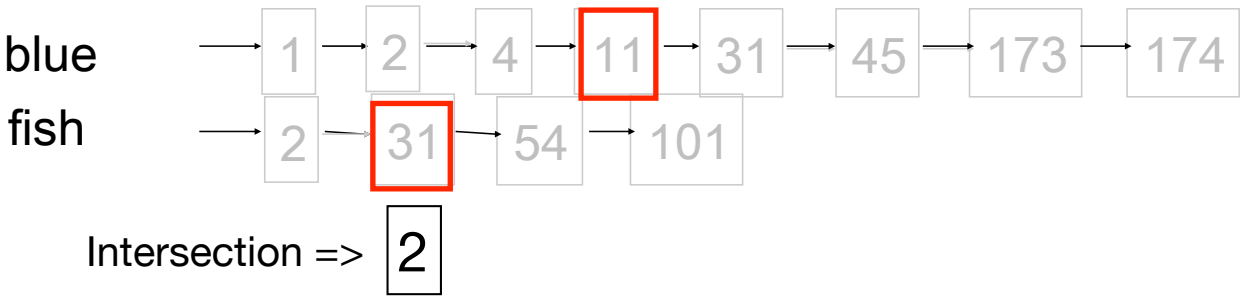
Intersecting two posting lists



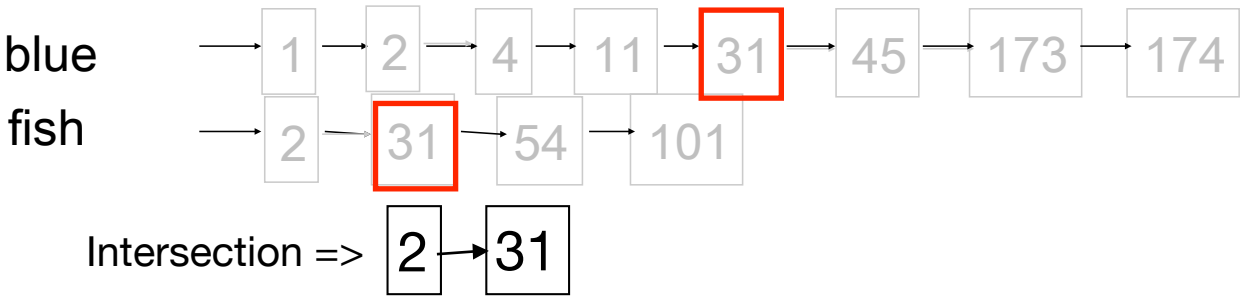
Intersecting two posting lists



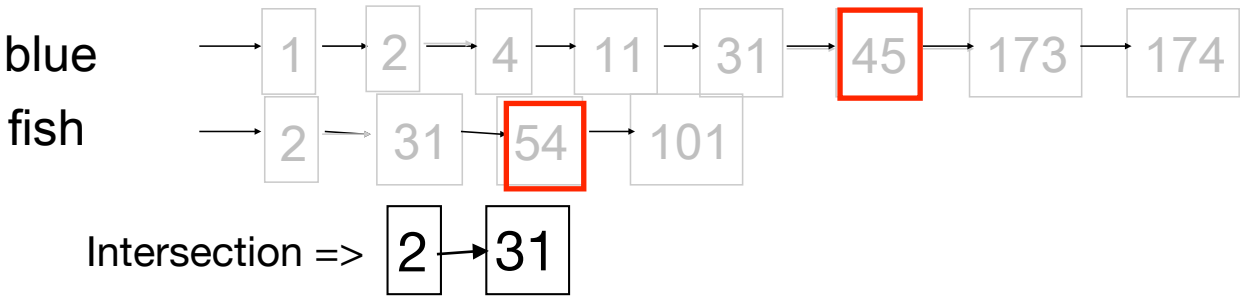
Intersecting two posting lists



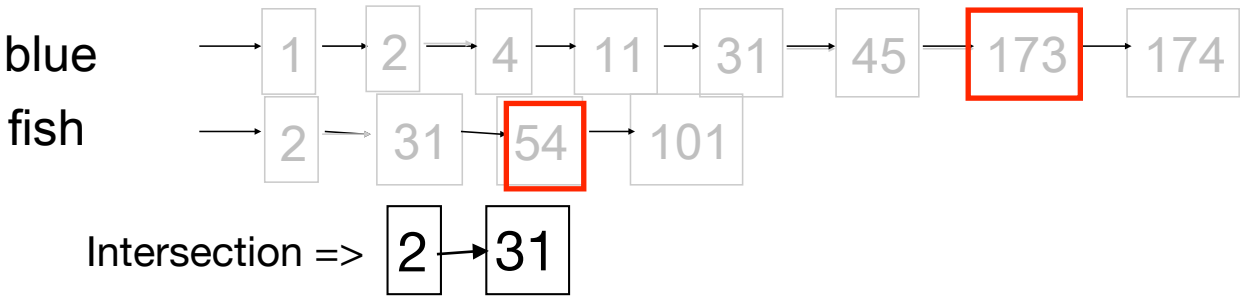
Intersecting two posting lists



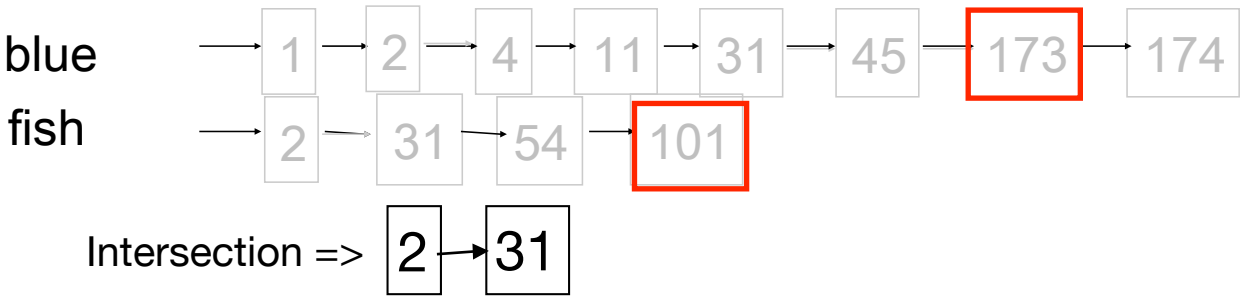
Intersecting two posting lists



Intersecting two posting lists



Intersecting two posting lists



- Crucial: This only works if postings lists are sorted.
- Efficiency analysis:
  - Postings traversal is linear (assuming sorted postings)
  - If the list lengths are x and y, the merge takes **O(x+y)** Operations.
  - Start with shortest posting first

Intersecting two postings lists: AND operator

```
INTERSECT(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then p1 ← next(p1)
 9                else p2 ← next(p2)
10  return answer
```

Question: How to implement the **OR** operator?

Boolean model: Strengths and Weaknesses

- Strengths
  - Precise, if you know the right strategies
  - Precise, if you have an idea of what you're looking for
  - Implementations are fast and efficient
- Weaknesses
  - Users must learn Boolean logic
  - Boolean logic insufficient to capture the richness of language
  - Feast or famine: no control over size of result set: either too many hits or none
  - When do you stop reading? All documents in the result set are considered "equally good"
  - What about partial matches? Documents that "don't quite match" the query may be useful also

Ranked 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?

Documents as vectors

- Represent documents as vector representation of terms
- Given |V| the size of our vocabulary, we have a |V|-dimensional vector space.
- Terms are **axes** of the space.
- Documents are **points** or vectors in this space
- Very high-dimensional: tens of millions of dimensions when you apply this to a web search engine
- These are very sparse vectors — most entries are zero.

Queries as vectors

- Do the same for queries: represent them as vectors in the space
- Key idea: Rank documents according to their proximity to the query in this space
- proximity = similarity of vectors
- proximity ≈ contrary to distance
- Recall: We do this because we want to get away from the you're-either-in-or-out Boolean model.
- Instead: rank more relevant documents higher than less relevant documents

Example: vector representation of documents

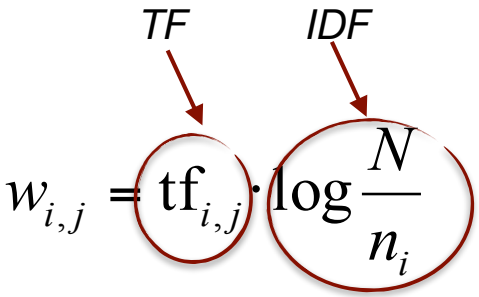
- Take the following documents:
  - D1: Information, Retrieval, Text, Information
  - D2: Conference, Information
  - D3: Information, Retrieval, TREC
  - D4: Video, Retrieval
- And the query :
  - Q: Video, Information, Retrieval

- Construct the vector representation of the documents and the query:

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?
  - Term frequency (local) : **tf**
  - Inverse document frequency (global): **idf**

TF.IDF Term Weighting



$w_{i,j}$ 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*

Example: vector representation of documents (revisited)

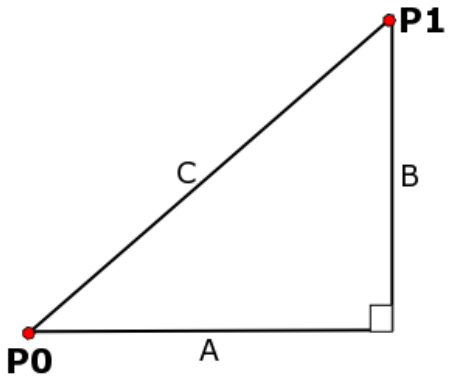
- Take the following documents:
  - D1: Information, Retrieval, Text, Information
  - D2: Conference, Information
  - D3: Information, Retrieval, TREC
  - D4: Video, Retrieval
- And the query :
  - Q: Video, Information, Retrieval

- Construct the vector representation of the documents and the query:
  - Weighting schema: *tf* for document, tf.idf for queries (assume N = 10)

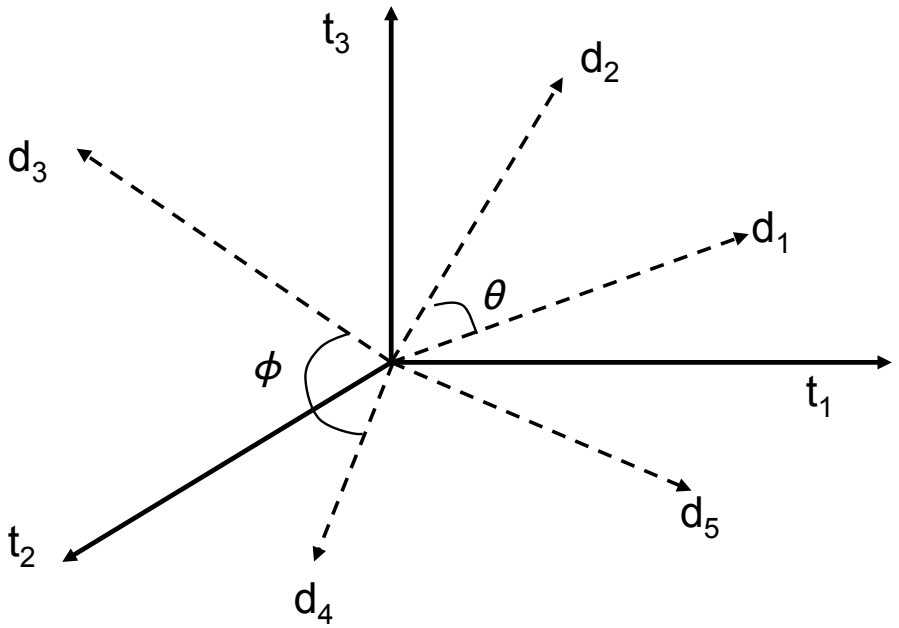
- *Reflection: Why did we weight queries using tf.idf but documents using only tf ?*

How do we formalize vector space similarity?

- First cut: distance between two points
  (= distance between the end points of the two vectors)
- Euclidean distance?
- Euclidean distance is a bad idea . . .
- . . . because Euclidean distance is large for vectors of different lengths.



Vector Space Model



Assumption: Documents that are “close together” in vector space “talk about” the same things

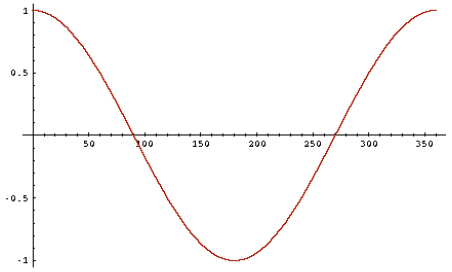
Therefore, retrieve documents based on how close the document is to the query (i.e., similarity ~ “closeness”)

Similarity Metric

- Use “angle” between the vectors.
- Rank documents according to angle with query.
- Thought experiment:
  - take a document d and append it to itself. Call this document d′.
  - “Semantically” d and d′have the same content
  - The angle between the two documents is 0, corresponding to maximal similarity.
  - The Euclidean distance between the two documents can be quite large

From angles to cosinus

- The following two notions are equivalent:
  - **Rank** documents in **decreasing** order of the **angle** between query and document
  - **Rank** documents in **increasing** order of **cosine** (query,document)
- Cosine is a monotonically decreasing function for the interval [0°, 180°]



$$d_j = [w_{j,1}, w_{j,2}, w_{j,3}, \ldots w_{j,n}]$$
$$d_k = [w_{k,1}, w_{k,2}, w_{k,3}, \ldots w_{k,n}]$$

$$\cos\theta = \frac{d_j \cdot d_k}{|d_j||d_k|}$$

$$\text{sim}(d_j, d_k) = \frac{d_j \cdot d_k}{|d_j||d_k|} = \frac{\sum_{i=0}^{n} w_{j,i} w_{k,i}}{\sqrt{\sum_{i=0}^{n} w_{j,i}^2} \sqrt{\sum_{i=0}^{n} w_{k,i}^2}}$$

Example: Ranking the documents in response to a given query

- Take the following documents:
  - D1: Information, Retrieval, Text, Information
  - D2: Conference, Information
  - D3: Information, Retrieval, TREC
  - D4: Video, Retrieval
- And the query :
  - Q: Video, Information, Retrieval

- Construct the vector representation of the documents and the query:
  - Weighting schema: tf for document, tf.idf for queries
- Calculate the similarity of the query with the given documents and rank the results.

- *Reflection: Is this approach efficient for ranking a big number of documents against a query?*

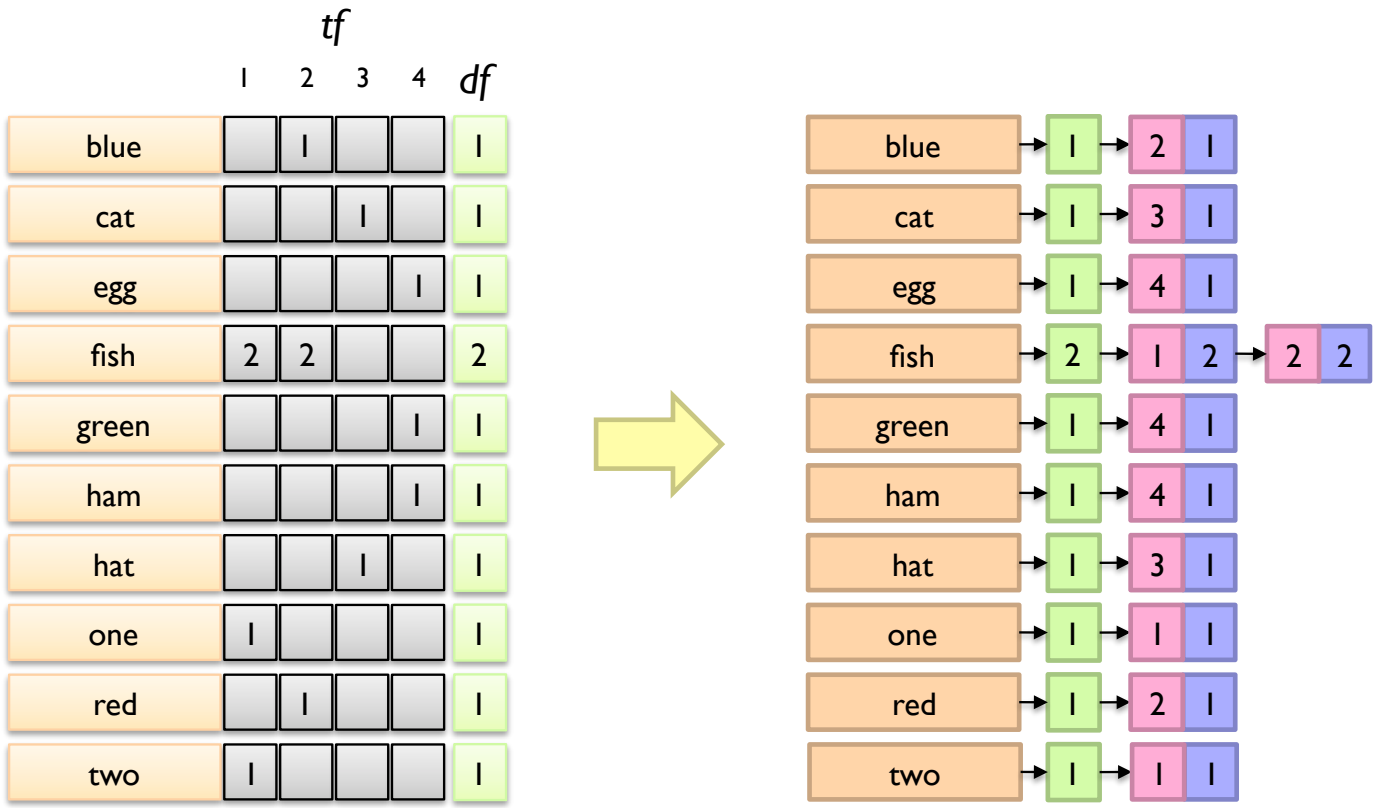
Inverted Index: TF and IDF

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham



Positional Indexes

- Store term position in postings
- Supports richer queries (e.g. proximity)
- Naturally, leads to larger indexes…

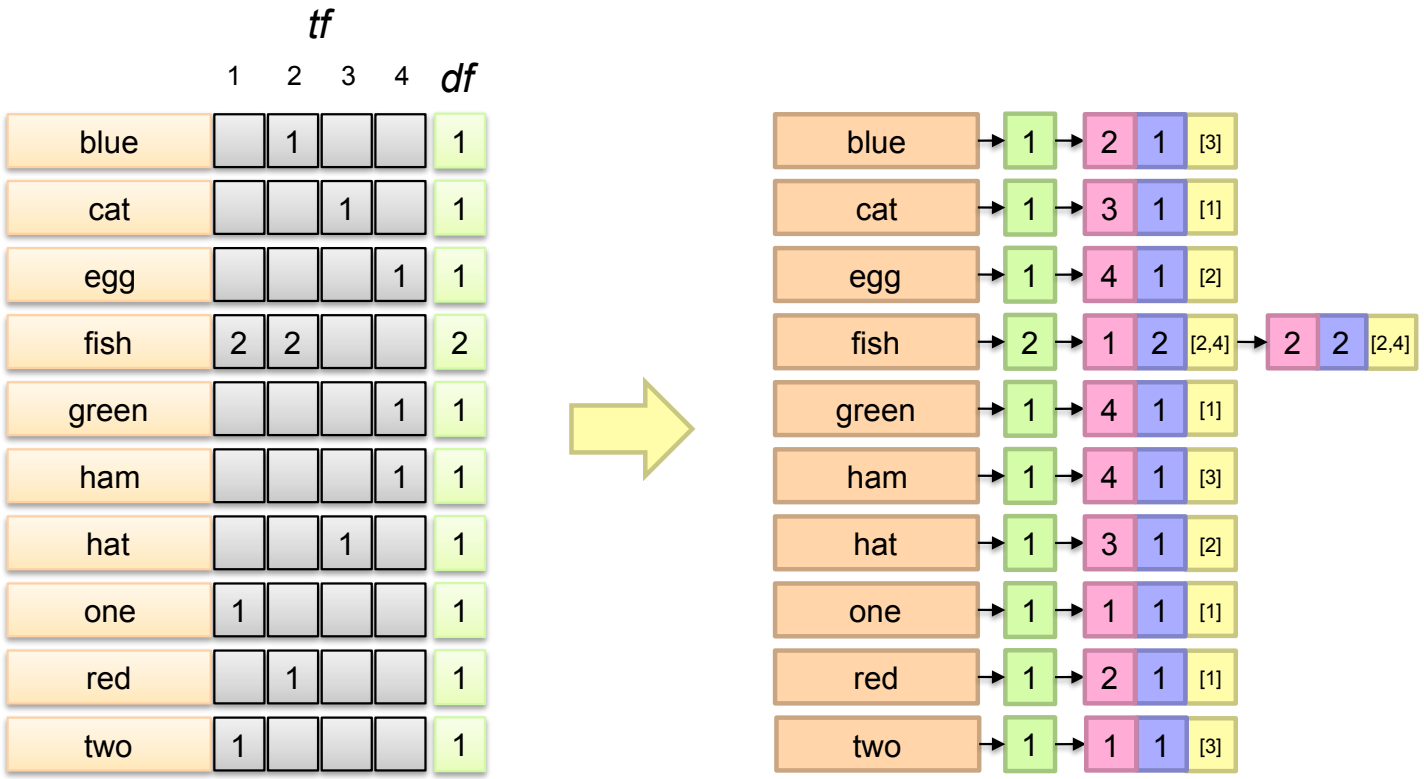
Inverted Index: Positional Information

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham



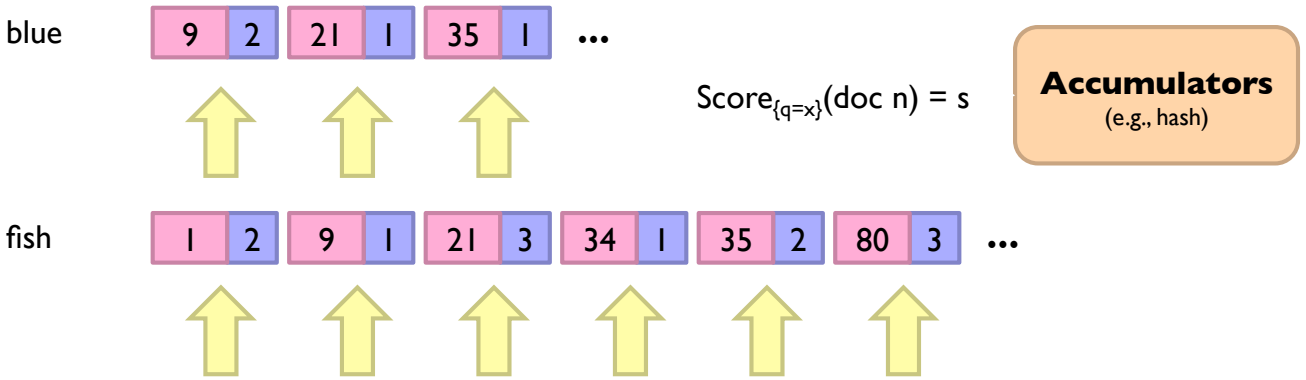
Retrieval using the inverted index

- 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

```
CosineScore(q)
 1  float Scores[N] = 0
 2  float Length[N]
 3  for each query term t
 4  do calculate w_{t,q} and fetch postings list for t
 5      for each pair(d, tf_{t,d}) in postings list
 6      do Scores[d] += w_{t,d} × w_{t,q}
 7  Read the array Length
 8  for each d
 9  do Scores[d] = Scores[d]/Length[d]
10  return Top K components of Scores[]
```

Retrieval: Query-At-A-Time

- Evaluate documents one query term at a time



- Tradeoffs
  - Early termination heuristics (good)
  - Large memory footprint (bad), but filtering heuristics possible

Example: Constructing the index and using the index for retrieval

- Take the following documents:
  - D1: Information, Retrieval, Text, Information
  - D2: Conference, Information
  - D3: Information, Retrieval, TREC
  - D4: Video, Retrieval
- And the query :
  - Q: Video, Information, Retrieval

- Construct the vector representation of the documents and the query:
  - Weighting schema: tf for document, tf.idf for queries (assume N = 10)
- Apply the retrieval algorithm and rank documents against the query.

- *Reflection: What are the tasks involved in the index construction?*

MapReduce it?

- The indexing problem
  - Scalability is critical
  - Must be relatively fast, but need not be real time
  - Fundamentally a batch operation
- The retrieval problem
  - Must have sub-second response time
  - For the web, only need relatively few results

Perfect for MapReduce!

Uh... not so good...

Indexing: Performance Analysis

- Fundamentally, a large sorting problem
  - Terms usually fit in memory
  - Postings usually don't
- How is it done on a single machine?
- How can it be done with MapReduce?
- First, let's characterize the problem size:
  - Size of vocabulary
  - Size of postings

Vocabulary Size: Heaps' Law

- The motivation for Heaps' Law: the simplest possible relationship between collection size and vocabulary size is linear in log-log space.
- The assumption of linearity is usually born out in practice as shown on the next figure.
- Vocabulary size grows unbounded!

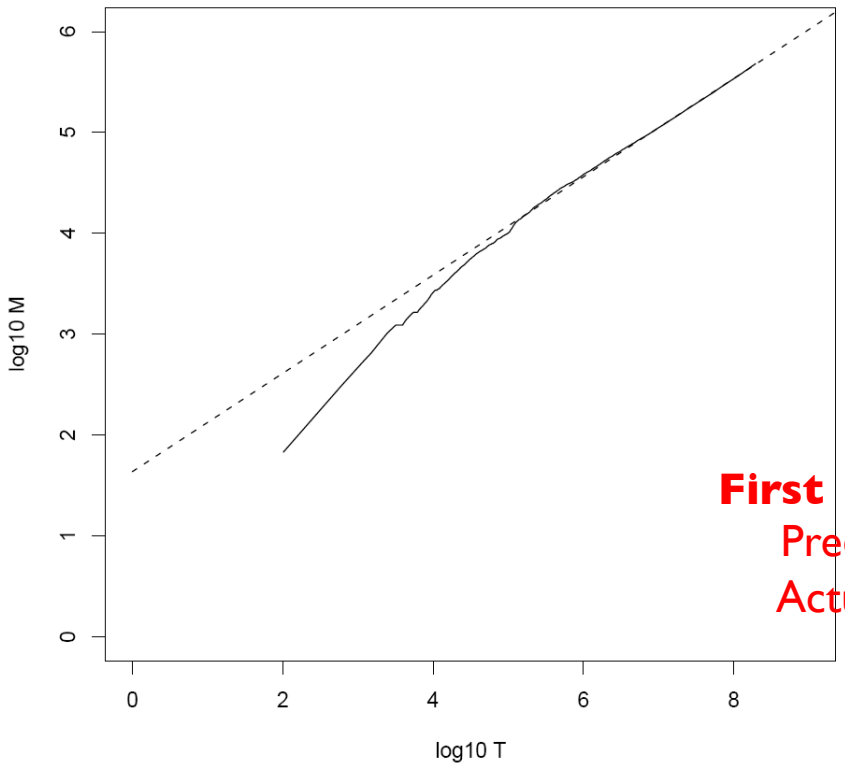
$$M = kT^b$$

M is vocabulary size
T is collection size (number of documents)
k and *b* are constants

Typically, *k* is between 30 and 100, *b* is between 0.4 and 0.6

Heaps' Law for RCV1

In this figure, the fit is excellent for T>100,000 for the parameter values b= 0.49 and k=44. For the first 1,000,020 terms (document size) Heap's law predicts 38,323 terms.



k = 44
b = 0.49

First 1,000,020 terms:
Predicted = 38,323
Actual = 38,365

Reuters-RCV1 collection: 806,791 newswire documents (Aug 20, 1996-August 19, 1997)

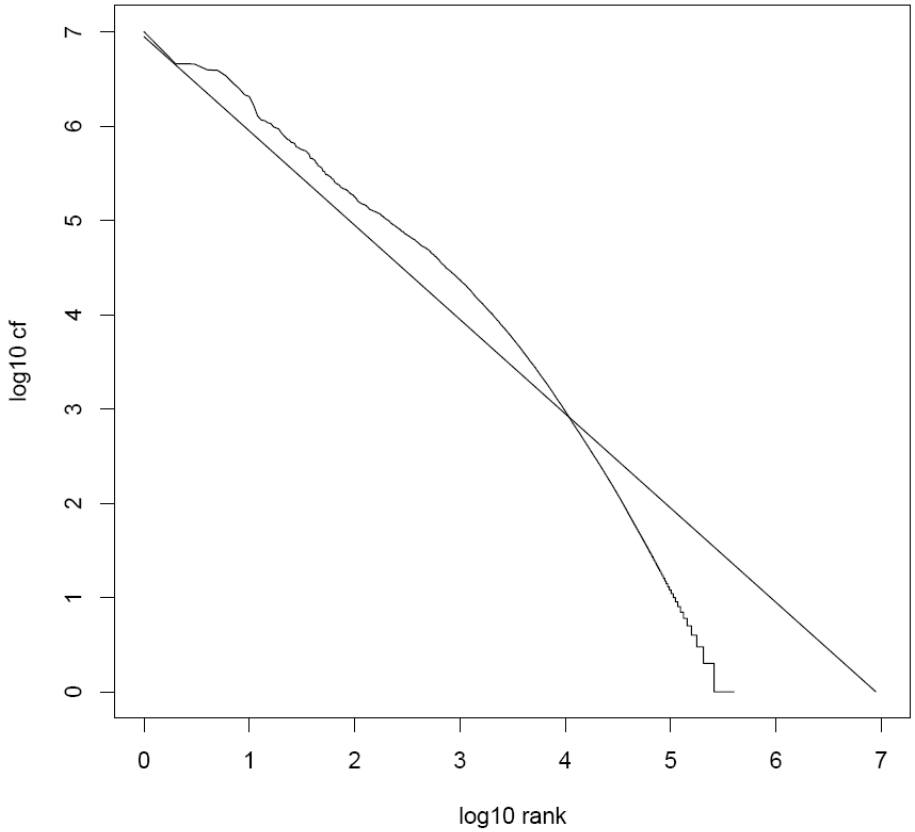
Zipf's Law: modeling the distribution of terms (postings lists)

- Zipf's Law: (also) linear in log-log space
  - If t_1 is the most frequent term in the collection, t_2 is the next most common, and so on, then the collection frequency cf_i of the ith most common term is proportional to 1/i.
  - So if the most frequent term occurs cf_1 times, then the second most frequent term has half as many occurrences, the third most frequent term a third as many occurrences, and so on.
- In other words:
  - A few elements occur very frequently

$$\mathrm{cf}_i = \frac{c}{i}$$

cf is the collection frequency of *i*-th common term
c is a constant

Zipf's Law for RCV1



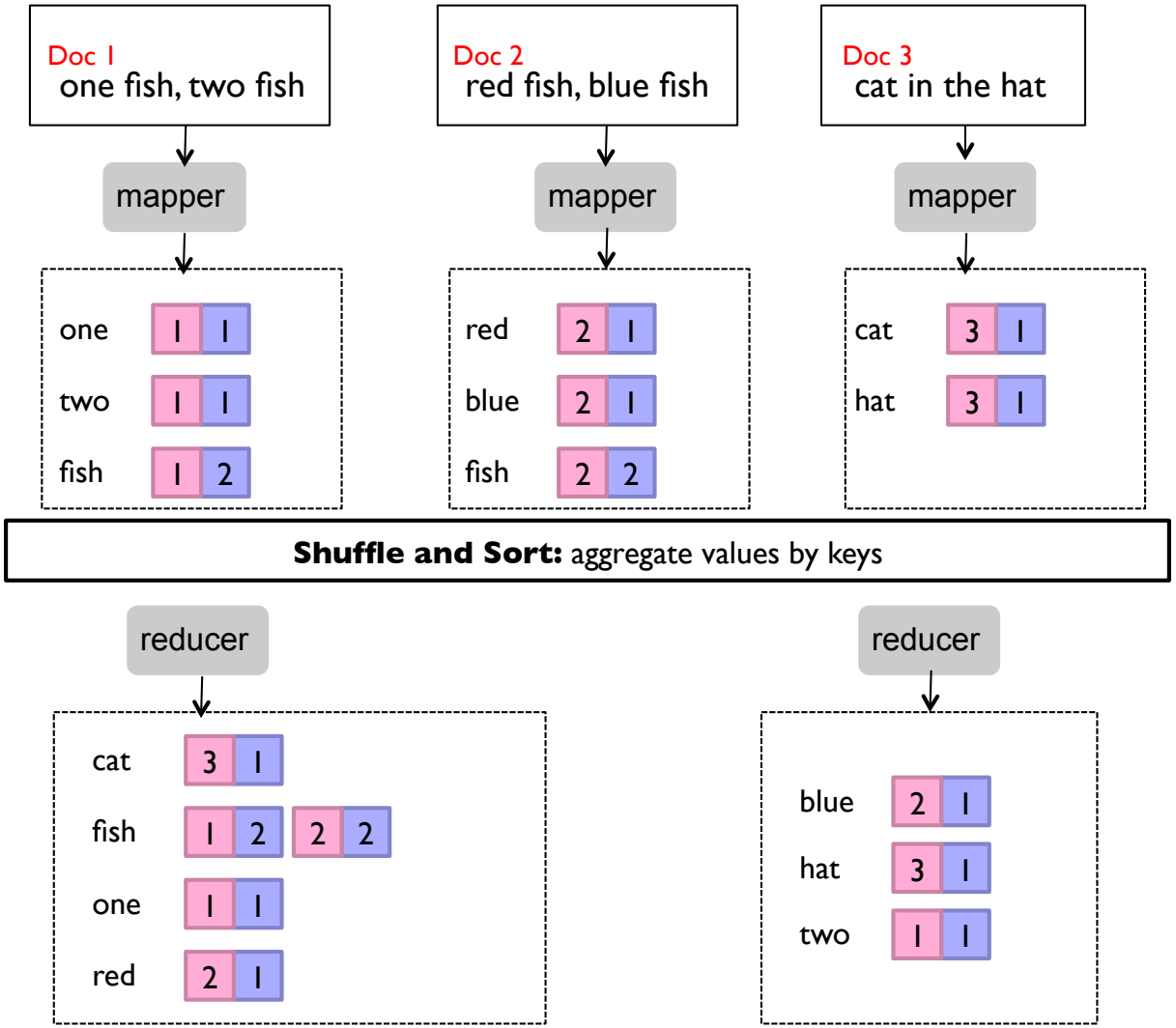
Fit isn't that good…
but good enough!

Reuters-RCV1 collection: 806,791 newswire documents (Aug 20, 1996-August 19, 1997)

MapReduce: Index Construction

- Map over all documents
  - Emit *term* as key, *(docno, tf)* as value
  - Emit other information as necessary (e.g., term position)
- Sort/shuffle: group postings by term
- Reduce
  - Gather and sort the postings (e.g., by *docno* or *tf*)
  - Write postings to disk
- MapReduce does all the heavy lifting!

Inverted Indexing with MapReduce

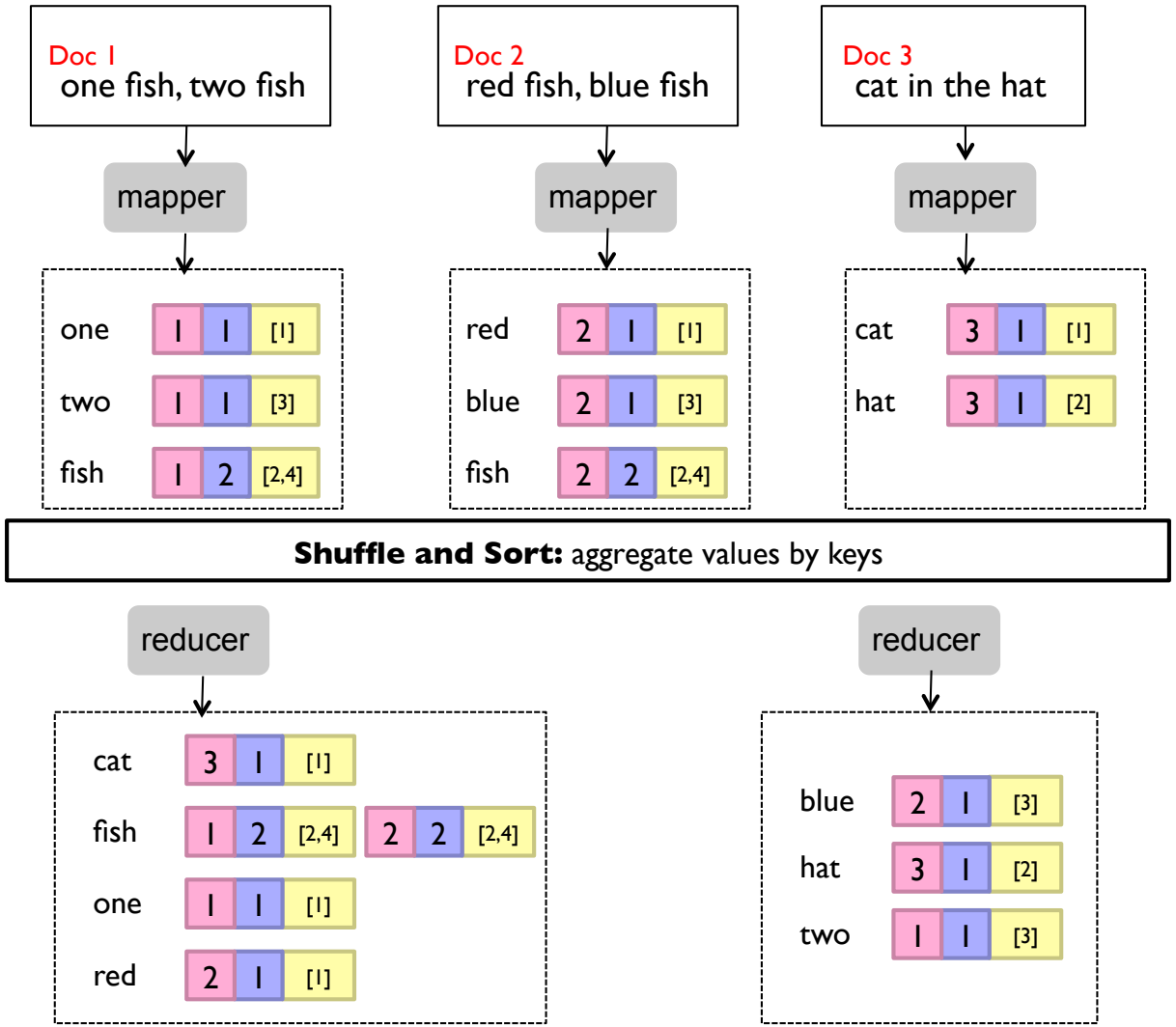


Inverted Indexing: Pseudo-Code

```
1: class Mapper
2:     procedure Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)

1: class Reducer
2:     procedure Reduce(term t, postings [⟨a1, f1⟩, ⟨a2, f2⟩ ...])
3:         P ← new List
4:         for all posting ⟨a, f⟩ ∈ postings [⟨a1, f1⟩, ⟨a2, f2⟩ ...] do
5:             Append(P, ⟨a, f⟩)
6:         Sort(P)
7:         Emit(term t, postings P)
```

Positional Indexes



Inverted Indexing: Pseudo-Code

```
1: class Mapper
2:     procedure Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(term t, posting ⟨n, H{t}⟩)

1: class Reducer
2:     procedure Reduce(term t, postings [⟨a1, f1⟩, ⟨a2, f2⟩ ...])
3:         P ← new List
4:         for all posting ⟨a, f⟩ ∈ postings [⟨a1, f1⟩, ⟨a2, f2⟩ ...] do
5:             Append(P, ⟨a, f⟩)
6:         Sort(P)
7:         Emit(term t, postings P)
```

What's the problem?

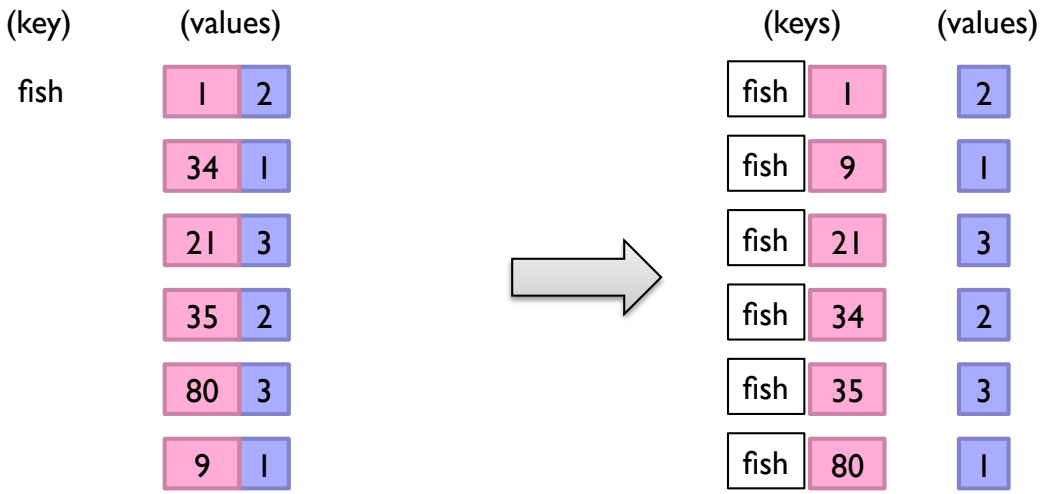
Scalability Bottleneck

- Initial implementation: terms as keys, postings as values
- Reducers must buffer all postings associated with key (to sort)
- As collections become larger, posting lists grow longer and at some point time, reducers will run out of memory
- Uh oh!

Another Try…

Instead of emitting key-value pairs of the type
(term *t*, posting ⟨*docid,f*⟩)

We emit intermediate key-value pairs of the type
(tuple ⟨*t, docid*⟩ tf *f*)



- How is this different?
  - Let the framework do the sorting
  - Term frequency implicitly sorted
  - Directly write postings to disk!

General case: Secondary Sorting Problem

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered
- What if want to sort value also?
  - E.g., k → (v1, r), (v3, r), (v4, r), (v8, r)…

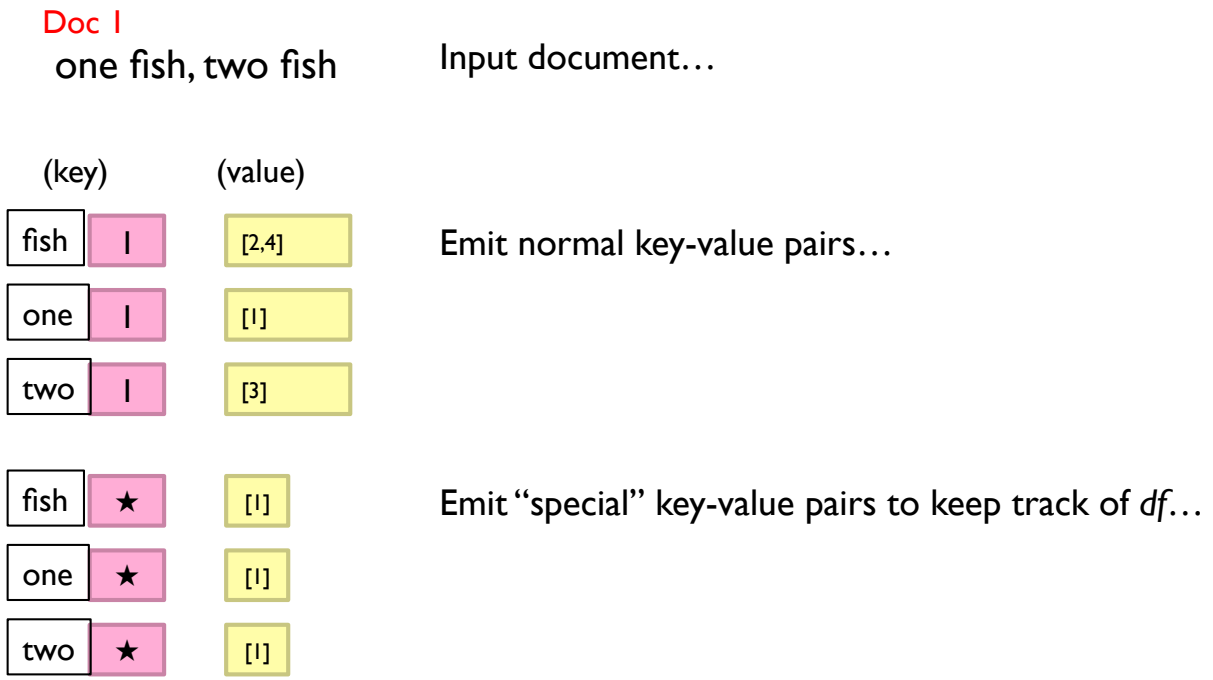
Secondary Sorting: Solutions

- 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, (k, v1)
  - Let execution framework do the sorting
  - Preserve state across multiple key-value pairs to handle processing
  - Anything else we need to do?

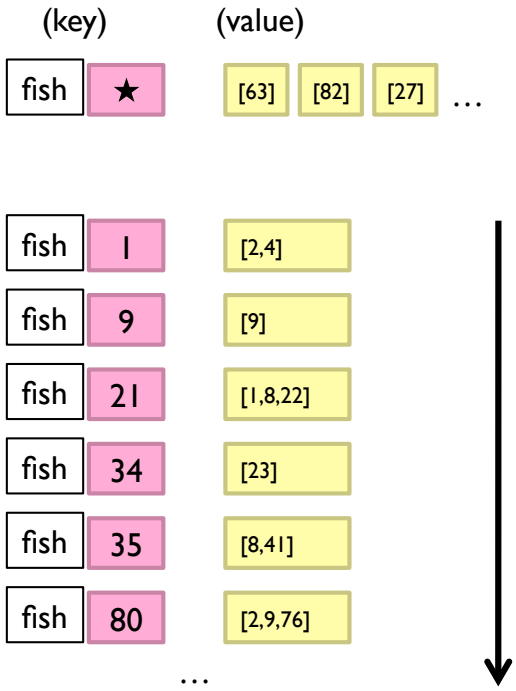
Getting the df

- 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*

Getting the df: Modified Mapper



Getting the df: Modified Reducer



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!

Write postings directly to disk

Where have we seen this before?

Inverted Indexing: Scalable version

```
1: class Mapper
2:     method Map(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             Emit(tuple ⟨t, n⟩, tf H{t})
```

```
1: class Reducer
2:     method Initialize
3:         t_prev ← ∅
4:         P ← new PostingsList
5:     method Reduce(tuple ⟨t, n⟩, tf [f])
6:         if t ≠ t_prev ∧ t_prev ≠ ∅ then
7:             Emit(term t_prev, postings P)
8:             P.Reset()
9:         P.Add(⟨n, f⟩)
10:        t_prev ← t
11:    method Close
12:        Emit(term t, postings P)
```

References

- The slides of this section are based on the following sources:
  - Jimmy Lin's course "Data Intensive Computing with MapReduce", University of Maryland.
  - Christopher D. Manning's course "Information Retrieval and Web Search", Stanford University.