After Sessional I

Mansi A. Radke

Index construction

Mansi A. Radke

Note

- Assumption: Collection is static i.e. it never changes …. There are no additions or deletions to the collection
- Desirable to keep entire index in RAM for performance reasons, however in many applications it is not feasible
- Hard drive is much cheaper than the RAM. In 2010 1GB RAM costed 200 times of the cost of 1GB hard drive space. This difference might be even more today
- An in memory index is 10 to 20 times faster than an on-disk index.
- Hence throughout the chapter we focus on hybrid organizations of the index in which some parts of the index are kept in main memory and majority of the data are on the hard disk. We do so assuming that main memory is a scarce resource

Index components

- Dictionary
- Posting lists

- Note that for every query term in an incoming search query, the search engine first needs to locate the term’s posting list before it can start actually processing the query

- It is the dictionary which provides a mapping between the terms and the location of their posting lists in the index

Life cycle of an inverted index

- Index construction
  - The text collection is processed sequentially, one token at a time, and a postings list is built for each term in the collection in an incremental fashion
  - Also called phase 1
  - The time required for this is called indexing time
- Query processing
  - The information stored in the index built in the phase 1 is used to process search queries
  - Also called as phase 2
  - The time required for this is called as query time

Dictionary

- Dictionary is the central data structure that is used to manage the set of terms in a text collection
- Provides mapping between index term and the location of its posting list
- At query time locating the posting list if one of the first operations performed when processing an incoming keyword query
- At indexing time, the dictionary’s lookup capability allows the search engine to quickly obtain the memory address of the inverted list for each incoming term and append a new posting at the end of that list

Set of operations supported by the dictionary

- Insert a new entry for term T
- Find and return the entry for term T (if present)
- Find and return the entries of all terms that start with a given prefix P
  - For e.g. prefix queries like “inform*” – all words that begin with prefix inform

Size of the dictionary

- For a typical natural language text collection, the size of the dictionary is very small compared to the total size of the index
    - Examples: GOV 2 copus 0.6 percent, Shakespeare corpus – 7%
    - (This is because of the Zipf’s law – assume for now as we will study it later)

- Hence we assume that the dictionary is small enough to fit into the main memory

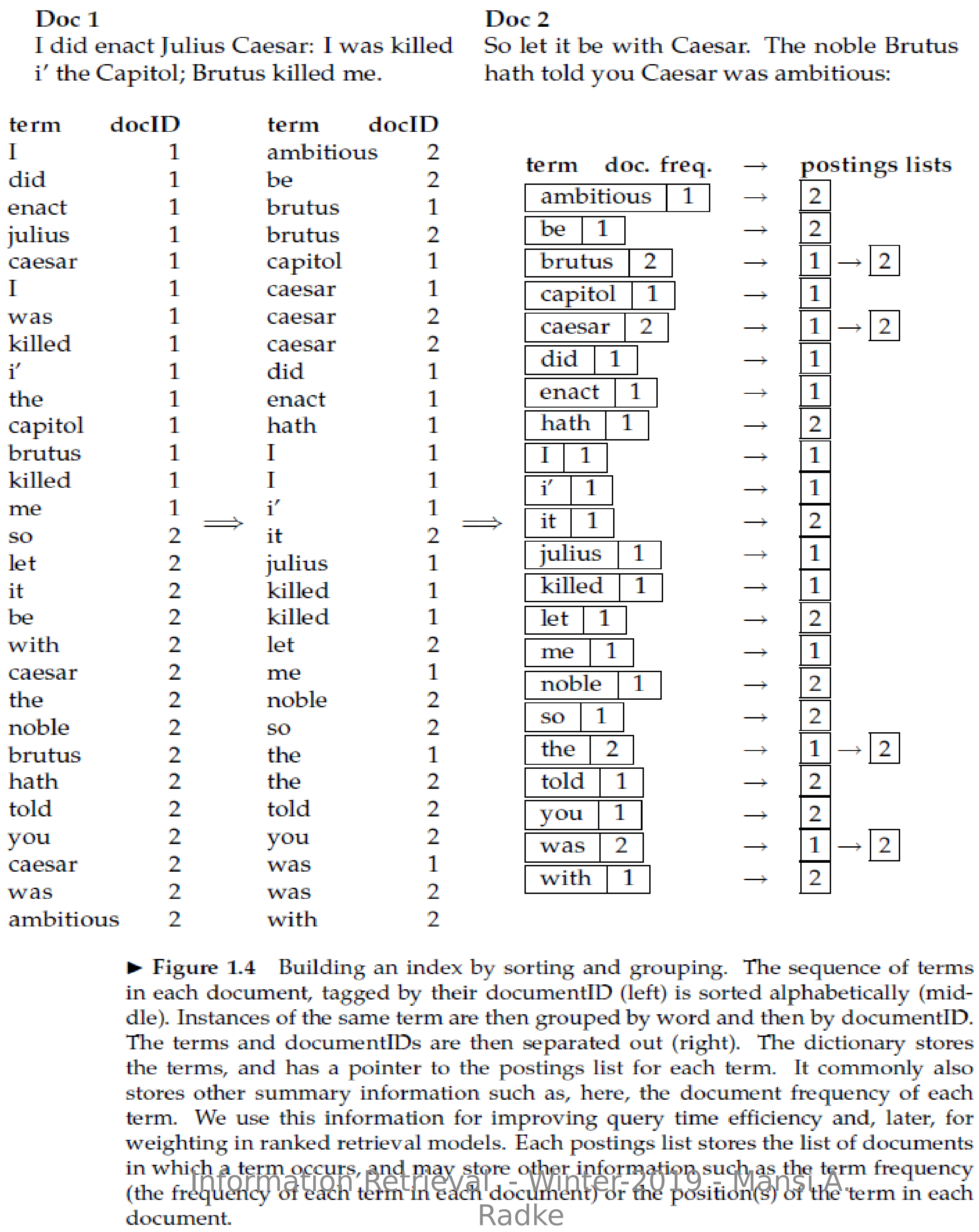
Two most common ways to realise an in-memory dictionary are:

- A sort-based dictionary, in which all the terms that appear in the text collection are arranged in a sorted array or in a search tree, in lexicographical (i.e. alphabetical) order. Lookup operations are realised through tree traversals typically using a BST when the list is sorted

- A hash based dictionary, in which each index term as a corresponding entry in a hash table. Collisions in the table (i.e. two terms are assigned the same hash value) are resolved using chaining

Blocked Sort Based Indexing

- Basic steps in constructing a non positional index are:
  - Make a pass through the collection collecting all term id - document id pairs
  - Sort the term id – document id pairs with term id as the dominant key and the document id as the secondary key
  - Organise the document ids for each term into a posting list
  - Compute the statistics like term and document frequency
- For small collections, all this can happen in the memory. Our focus is on methods for large collections that require the use of secondary storage
- Note: termIDs are represented as numbers instead of strings where each termID is a unique serial number



Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

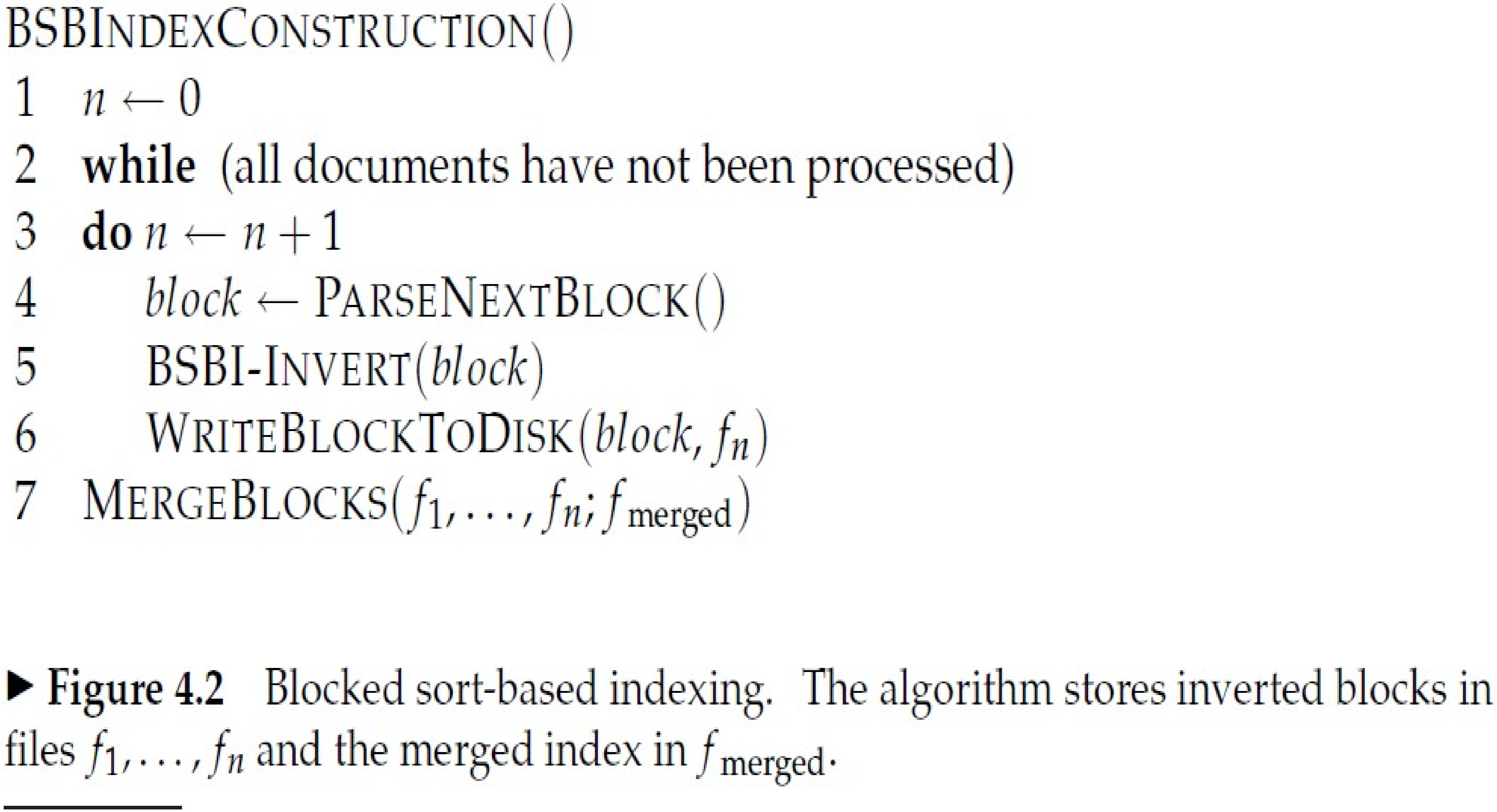
⟹

term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

⟹

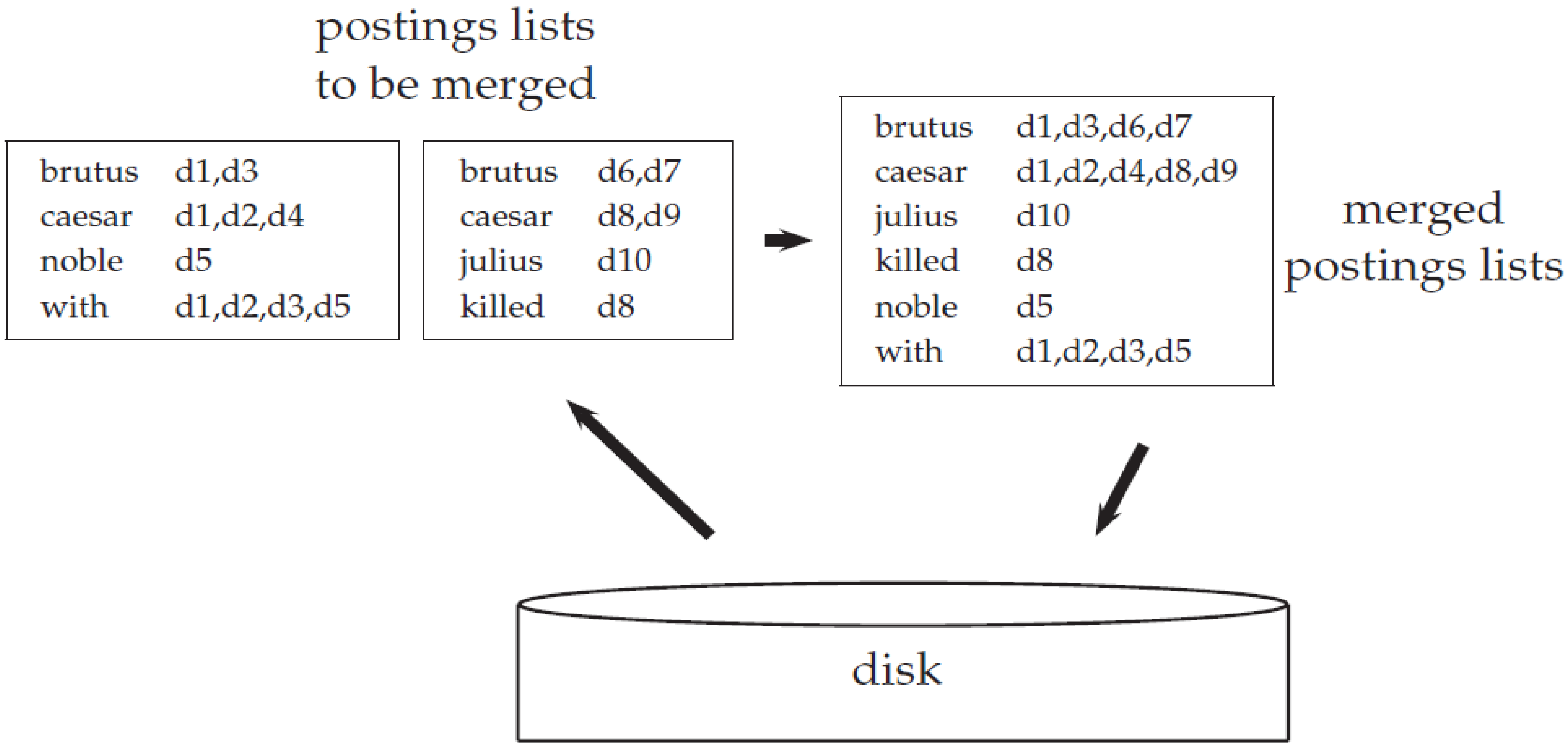
term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
I	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

► **Figure 1.4** Building an index by sorting and grouping. The sequence of terms in each document, tagged by their documentID (left) is sorted alphabetically (middle). Instances of the same term are then grouped by word and then by documentID. The terms and documentIDs are then separated out (right). The dictionary stores the terms, and has a pointer to the postings list for each term. It commonly also stores other summary information such as, here, the document frequency of each term. We use this information for improving query time efficiency and, later, for weighting in ranked retrieval models. Each postings list stores the list of documents in which a term occurs, and may store other information such as the term frequency (the frequency of each term in each document) or the position(s) of the term in each document.



```
BSBIndexConstruction()
$n \leftarrow 0$
while (all documents have not been processed)
do $n \leftarrow n + 1$
    $block \leftarrow$ ParseNextBlock()
    BSBI-Invert($block$)
    WriteBlockToDisk($block, f_n$)
MergeBlocks($f_1, \ldots, f_n; f_{\text{merged}}$)
```

▶ **Figure 4.2** Blocked sort-based indexing. The algorithm stores inverted blocks in files $f_1, \ldots, f_n$ and the merged index in f_{merged}.



brutus	d1,d3
caesar	d1,d2,d4
noble	d5
with	d1,d2,d3,d5

brutus	d6,d7
caesar	d8,d9
julius	d10
killed	d8

brutus	d1,d3,d6,d7
caesar	d1,d2,d4,d8,d9
julius	d10
killed	d8
noble	d5
with	d1,d2,d3,d5

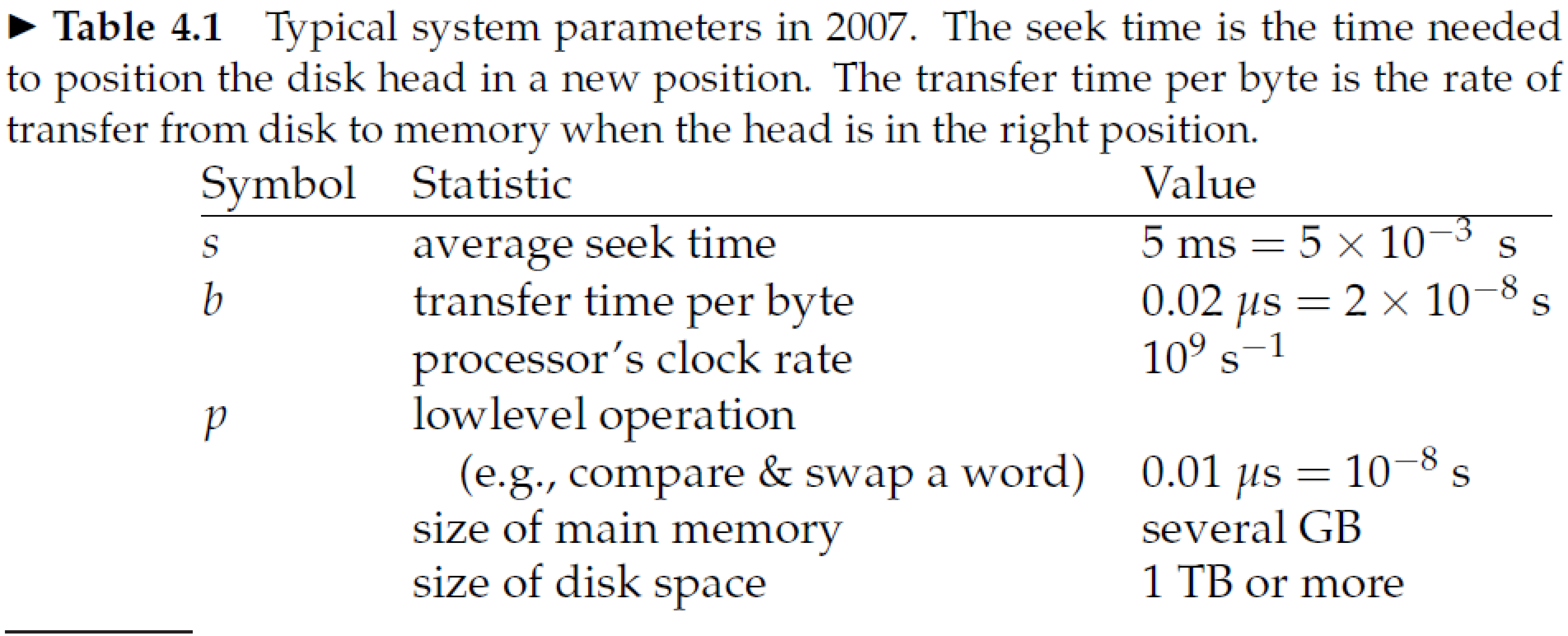
► **Figure 4.3** Merging in blocked sort-based indexing. Two blocks ("postings lists to be merged") are loaded from disk into memory, merged in memory ("merged postings lists") and written back to disk. We show terms instead of termIDs for better readability.

Merging – suppose 10 blocks

- Open all the block files simultaneously
- Maintain small read buffers for the 10 blocks which we are reading
- A small write buffer for the final merged index which we are writing
- In each iteration, the smallest term id is selected which has not been processed yet using a priority queue or similar data structure
- All posting lists for this term id are read and merged
- Merged list is written back to the disk
- Each buffer is filled from its file when necessary

Time complexity of BSBI

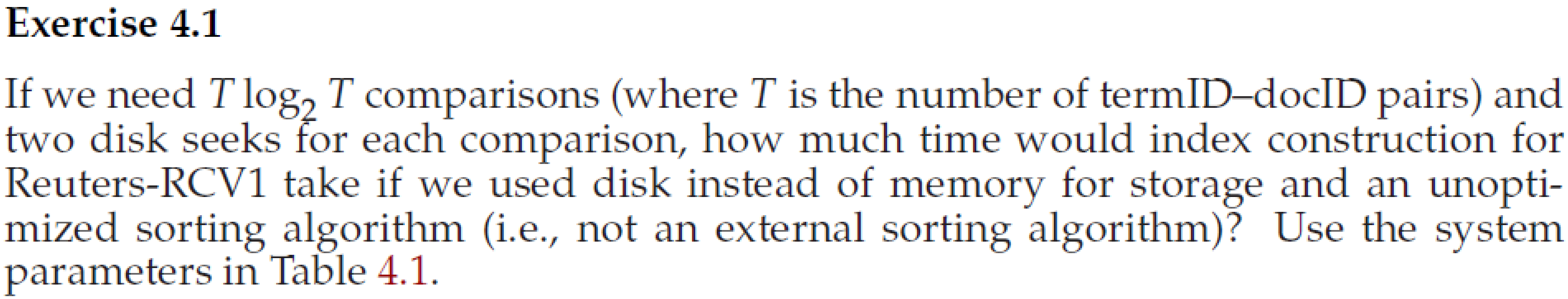
- O(T Log T) where T is the number of term id – doc id pairs
- The step with highest time complexity is sorting which takes O (T Log T) time
- But, ParseNextBlock and MergeBlocks usually dominates time taken due to disk access time



► **Table 4.1** Typical system parameters in 2007. The seek time is the time needed to position the disk head in a new position. The transfer time per byte is the rate of transfer from disk to memory when the head is in the right position.

Symbol	Statistic	Value
s	average seek time	$5 \text{ ms} = 5 \times 10^{-3}$ s
b	transfer time per byte	$0.02\ \mu\text{s} = 2 \times 10^{-8}$ s
	processor's clock rate	$10^9\ \text{s}^{-1}$
p	lowlevel operation	
	(e.g., compare & swap a word)	$0.01\ \mu\text{s} = 10^{-8}$ s
	size of main memory	several GB
	size of disk space	1 TB or more

Solve the following question…



Exercise 4.1

If we need $T \log_2 T$ comparisons (where T is the number of termID–docID pairs) and two disk seeks for each comparison, how much time would index construction for Reuters-RCV1 take if we used disk instead of memory for storage and an unoptimized sorting algorithm (i.e., not an external sorting algorithm)? Use the system parameters in Table 4.1.

Solution

- An unoptimized sorting algorithm would be to have all the postings stored on the disk and transfer them back and forth from disk to memory to make comparisons
- A trivial index construction would consist of these 2 steps:
  - Parsing the documents and creating the postings O(n)
  - Sorting the postings 2*n log n * diskseektime
- For RCV1
  - N = 100,000,000 = 10^8
- Step 2 dominates the total time by a large factor
- Answer = 2 * 10^8 * log 10^8 * 5 * 10^-3 seconds

Answer the following question…

- How would you create the dictionary in blocked sort based indexing on the fly to avoid an extra pass through the data?

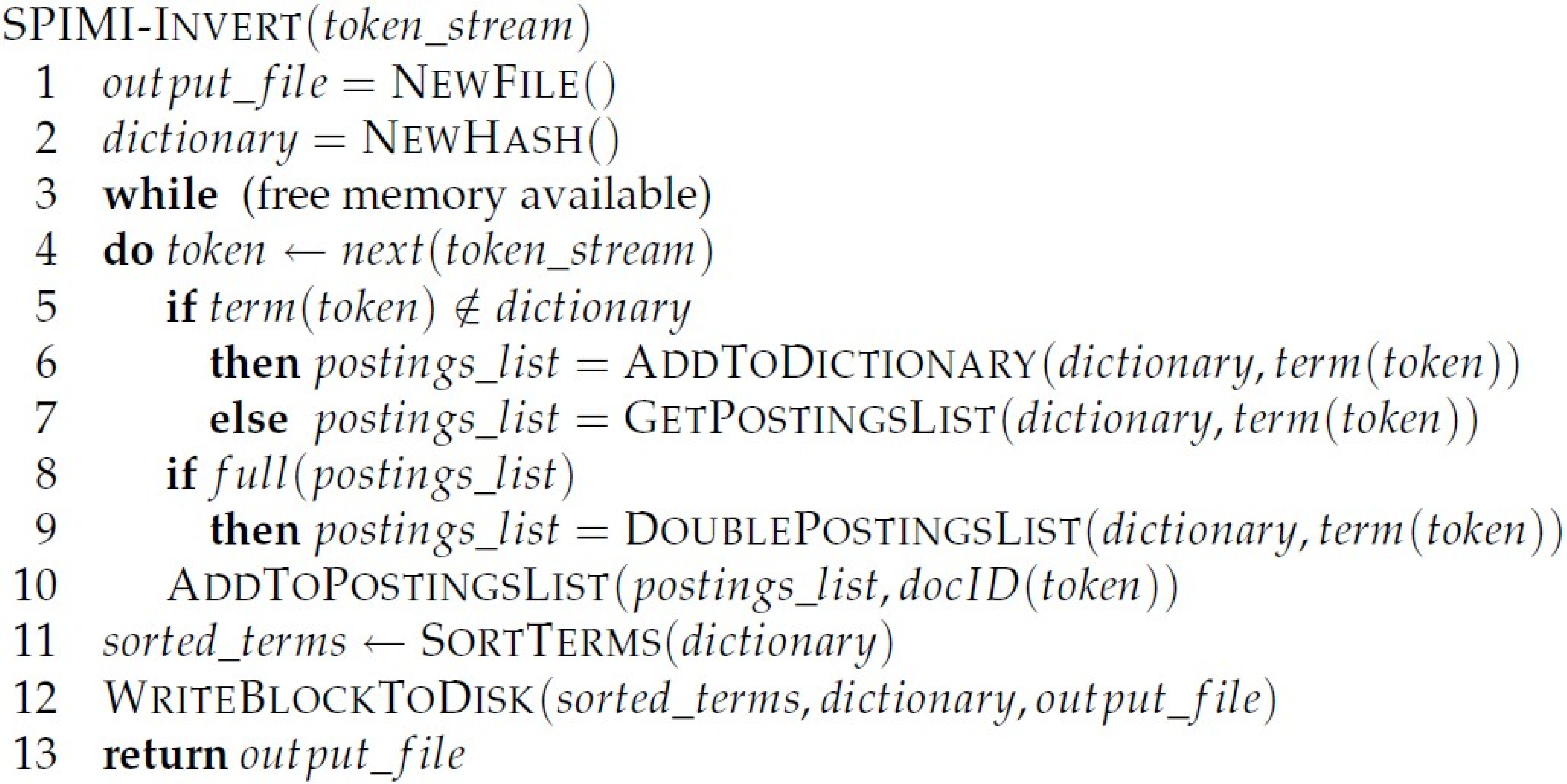
- Solution: Simply accumulate the vocabulary in memory using a hash

BSBI advantages and disadvantages

- Advantage:
  - Excellent scaling properties
- Disadvantage:
  - Needs a data structure for mapping terms to term ids. For very large collections this data structure does not fit into memory

SPIMI (Single pass in memory indexing)

- SPIMI uses terms instead of term ids
- It writes a block of dictionary to the disk and starts a new dictionary for the next block
- It can index collections of any size as long as disk space is available



```
SPIMI-Invert(token_stream)
output_file = NewFile()
dictionary = NewHash()
while (free memory available)
do token ← next(token_stream)
    if term(token) ∉ dictionary
      then postings_list = AddToDictionary(dictionary, term(token))
      else  postings_list = GetPostingsList(dictionary, term(token))
    if full(postings_list)
      then postings_list = DoublePostingsList(dictionary, term(token))
    AddToPostingsList(postings_list, docID(token))
sorted_terms ← SortTerms(dictionary)
WriteBlockToDisk(sorted_terms, dictionary, output_file)
return output_file
```

▶ **Figure 4.4** Inversion of a block in single-pass in-memory indexing

Time complexity of SPIMI

- $O(T)$

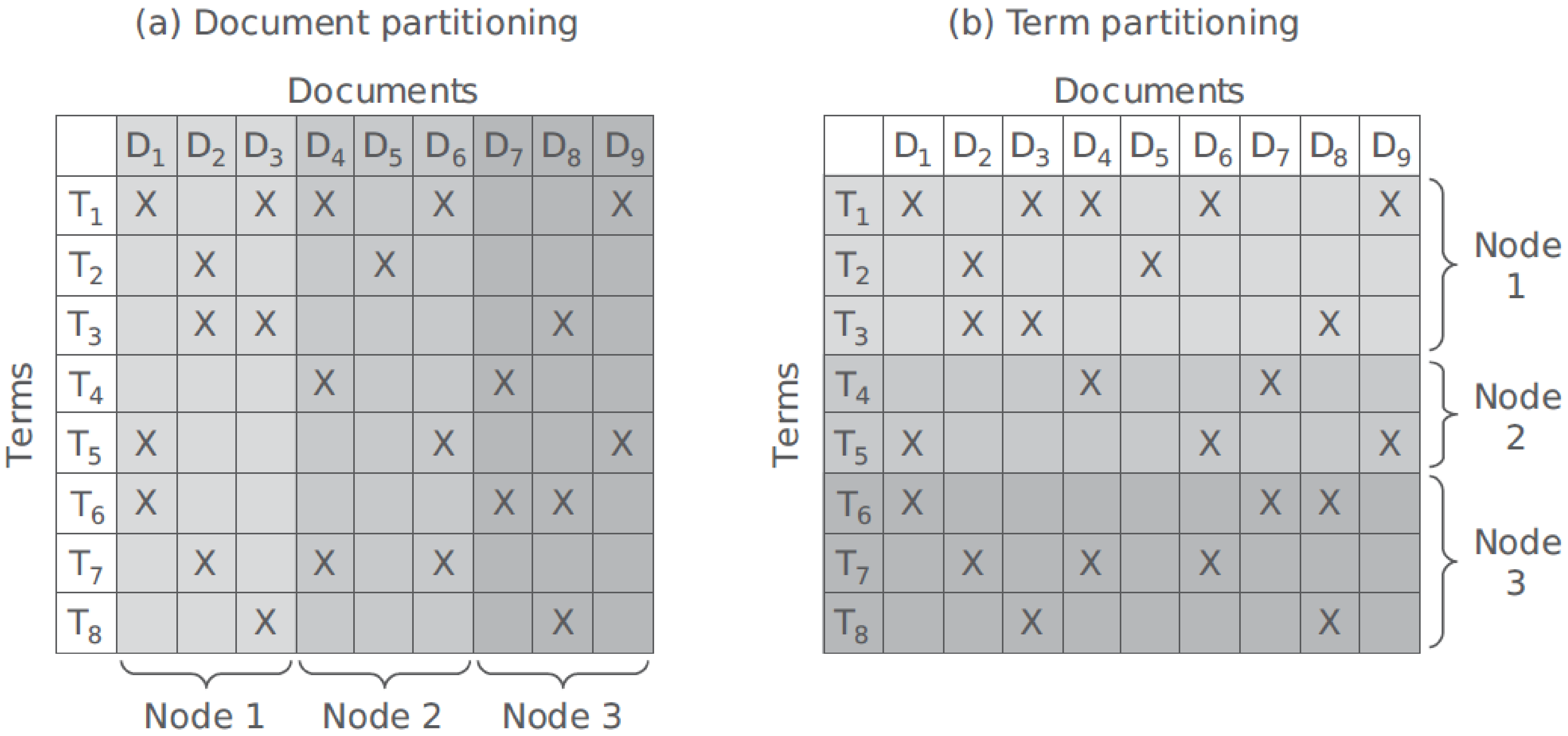
BSBI versus SPIMI

- SPIMI does not collect term id –doc id pairs
- SPIMI does not need to sort the term id document id pairs
- SPIMI adds postings directly to its posting list
- SPIMI uses a dynamic posting list (i.e. its size is adjusted as it grows)
- SPIMI keeps track of the term a postings list belongs to and no term ids are stored due to which the blocks that individual calls of SPIMI-INVERT can process are larger and so the index construction process as a whole is more eficient

Distributed Indexing

- Collections are huge
- Indexing cannot be done on single machine (e.g world wide web)
- Hence distributed indexing algorithms used
- Distributed index is partitioned across several machines
  - Acccording to term

    or
  - According to document
- In the next slides we discuss both these partitioning techniques



(a) Document partitioning

	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9
T_1	X		X	X		X			X
T_2		X			X				
T_3		X	X					X	
T_4				X			X		
T_5	X					X			X
T_6	X						X	X	
T_7		X		X		X			
T_8			X					X	

(b) Term partitioning

	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9
T_1	X		X	X		X			X
T_2		X			X				
T_3		X	X					X	
T_4				X			X		
T_5	X					X			X
T_6	X						X	X	
T_7		X		X		X			
T_8			X					X	

Figure 14.1 The two prevalent index partitioning schemes: document partitioning and term partitioning (shown for a hypothetical index containing 8 terms and 9 documents).

Term partitioning and document partitioning

In a document-partitioned index, each node holds an index
for a subset of the documents in the collection. For instance, the index
maintained by node 2
in Figure 14.1(a) contains the following docid lists:
L_1 = 4, 6, L_2 = 5, L_4 = 4, L_5 = 6, L_7 = 4, 6.

In a term-partitioned index, each node is responsible for a subset of
the terms in the collection.
The index stored in node 1 in Figure 14.1(b) contains the following
lists:
L1 = 1, 3, 4, 6, 9, L2 = 2, 5, L3 = 2, 3, 8.

- In this chapter we will discuss about distributed indexing for term partitioned index

MapReduce

- MapReduce is an architecture for distributed computing
- Master node
- Worker nodes
- Master node directs the process of assigning and reassigning tasks to individual worker nodes
- The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time\
- The chunks should be not too small and not too large

Role of master node

- Assigns and reassigns tasks to worker nodes
- As machine finishes one split, it is assigned next one
- If machine dies or has hardware problems, the split it is working on is assigned to another machine

Key value pairs indexing are <term id, doc id >

MAP Phase:

mapping splits of input data to key value pairs

The map phase writes its output to local intermediate files called as segment files

Reduce Phase:

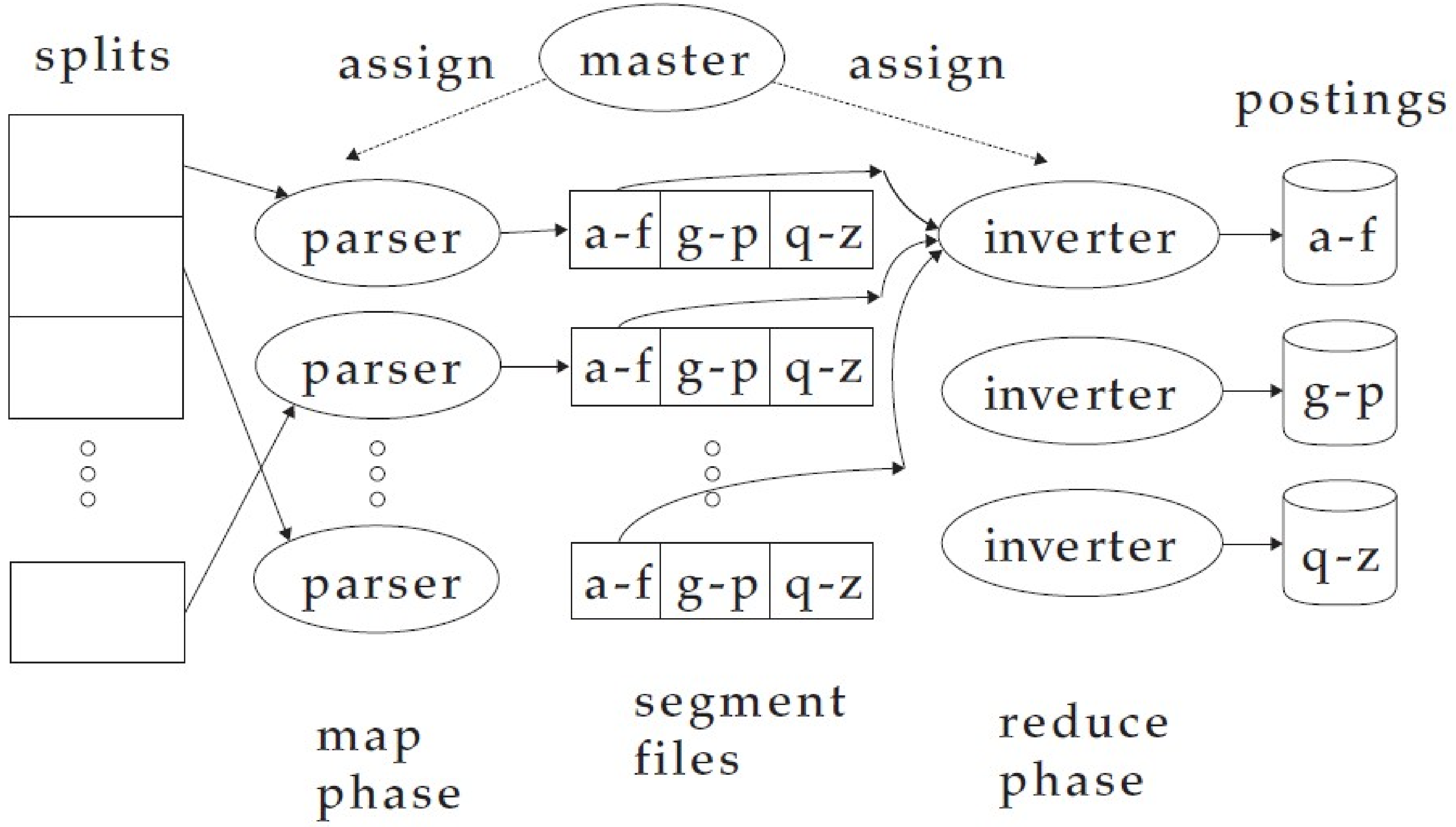
All values of a given key get stored together

the keys are partitioned into j term partitions

Note: The term partitions are defined by the person who operates the indexing system

Parsers and inverters

- Parsers and inverters are not separate sets of machines.
- The master identifies idle machines and assigns tasks to them.
- The same machine can be a parser in the map phase and an inverter in the reduce phase.
- And there are often other jobs that run in parallel with index construction, so in between being a parser and an inverter a machine might do some crawling or another unrelated task.



► **Figure 4.5** An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).

Problem on map reduce

Computing scores in a complete search system

Mansi A. Radke

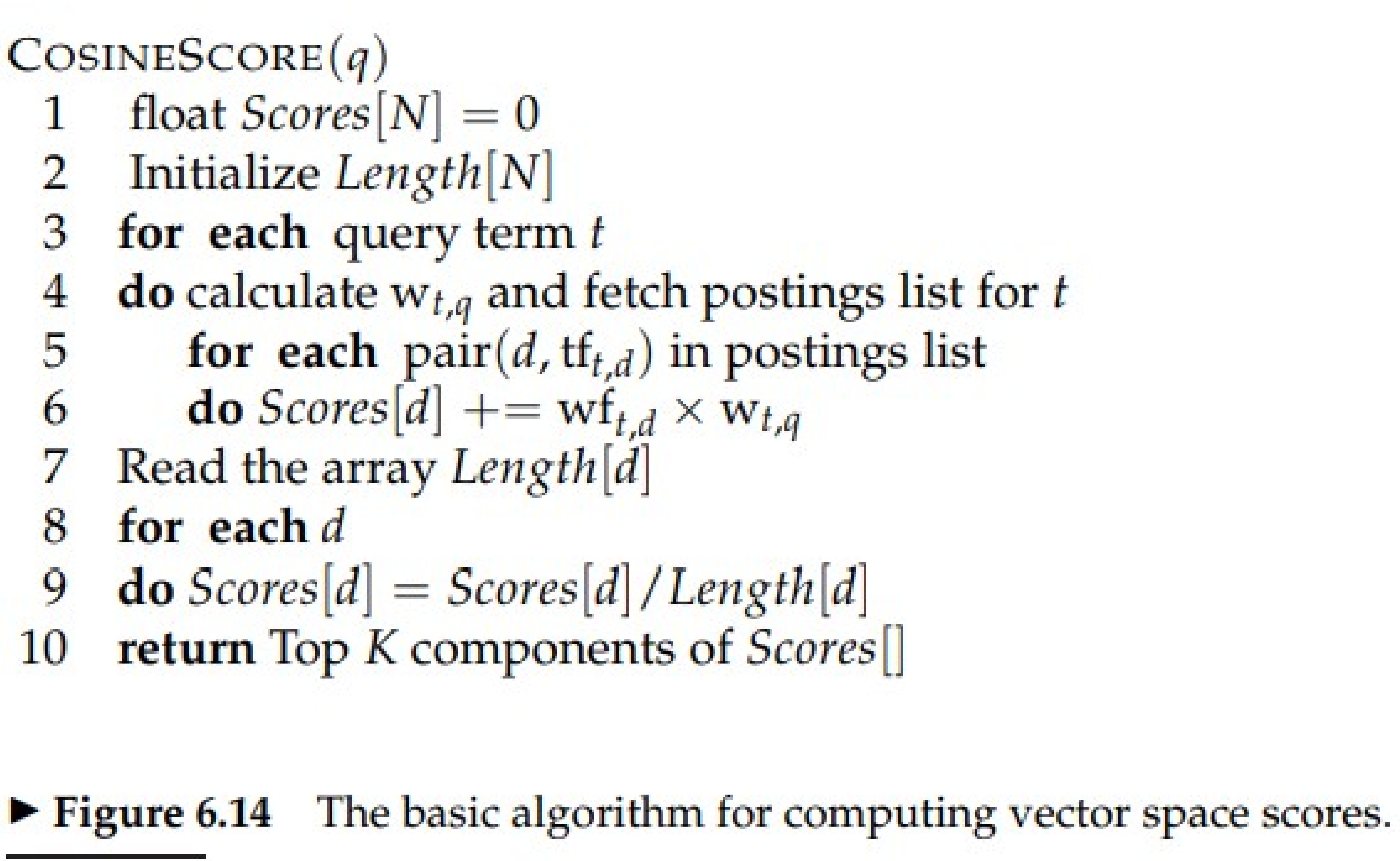
Efficient scoring and ranking in a complete search system

- For the purpose of ranking the documents matching a given query, we are really interested in the relative (rather than absolute) scores of the documents in the collection. To this end, it suffices to compute the cosine similarity from each document unit vector $v(d)$ to $V(q)$ (in which all non-zero components of the query vector are set to 1), rather than to the unit vector $v(q)$ for any two documents d1 and d2.

- For any document d, the cosine similarity $V(q) \cdot v(d)$ is the weighted sum, over all terms in the query q, of the weights of those terms in d.

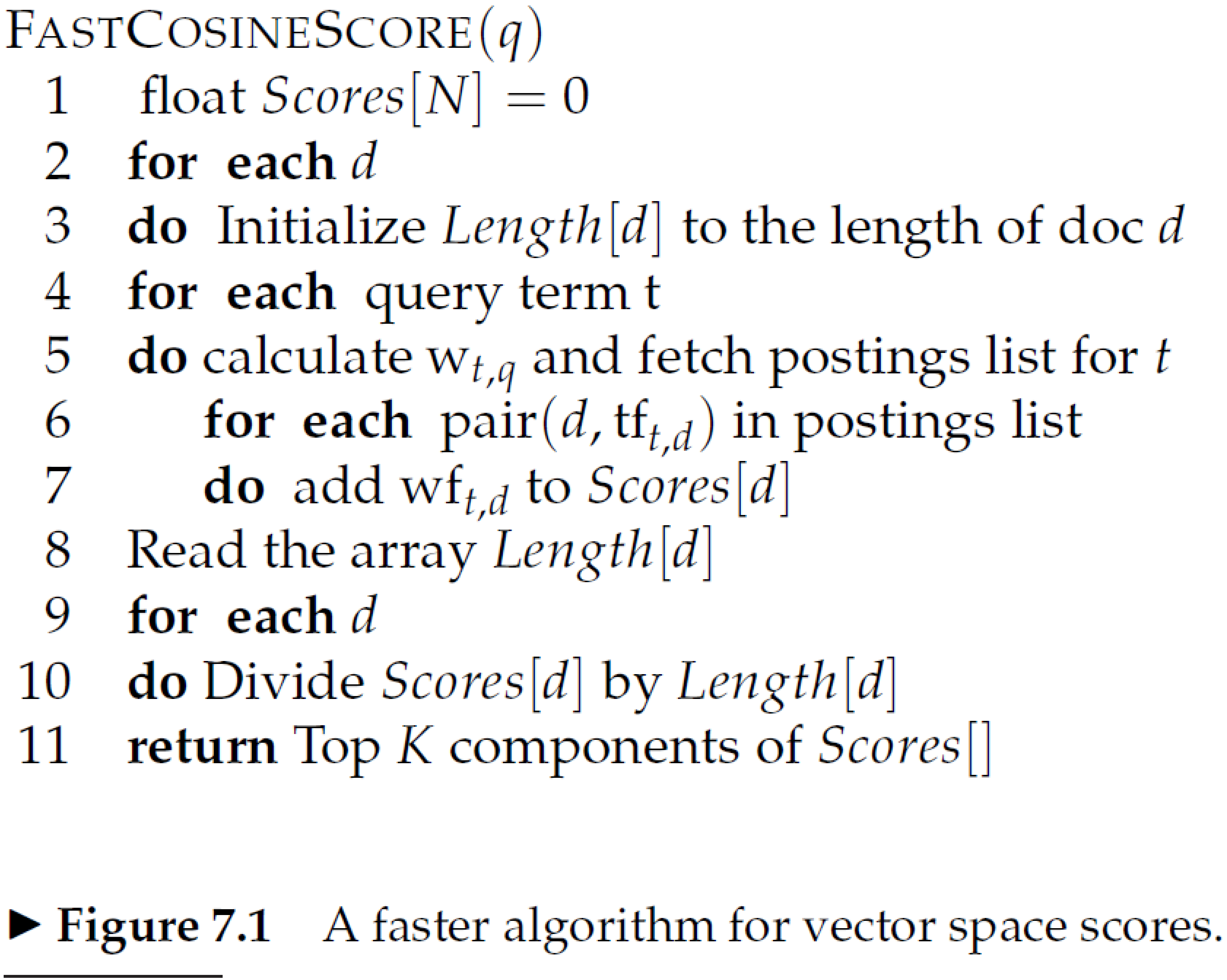
Efficient scoring and ranking in a complete search system

- Given these scores, the final step before presenting results to a user is to pick out the *K* highest-scoring documents. While one could sort the complete set of scores, a better approach is to use a heap to retrieve only the top *K* documents in order
- Where *J* is the number of documents with non-zero cosine scores, constructing such a heap can be performed in 2*J* comparison steps, following which each of the *K* highest scoring documents can be “read off” the heap with log *J* comparison steps



```
CosineScore($q$)
  float $Scores[N] = 0$
  Initialize $Length[N]$
  for each query term $t$
  do calculate $w_{t,q}$ and fetch postings list for $t$
      for each pair($d$, $\mathrm{tf}_{t,d}$) in postings list
      do $Scores[d] \mathrel{+}= \mathrm{wf}_{t,d} \times w_{t,q}$
  Read the array $Length[d]$
  for each $d$
  do $Scores[d] = Scores[d] / Length[d]$
  return Top $K$ components of $Scores[]$
```

► **Figure 6.14** The basic algorithm for computing vector space scores.



```
FASTCOSINESCORE($q$)
float $Scores[N] = 0$
for each $d$
do Initialize $Length[d]$ to the length of doc $d$
for each query term t
do calculate $w_{t,q}$ and fetch postings list for $t$
    for each pair$(d, \mathrm{tf}_{t,d})$ in postings list
    do add $\mathrm{wf}_{t,d}$ to $Scores[d]$
Read the array $Length[d]$
for each $d$
do Divide $Scores[d]$ by $Length[d]$
return Top $K$ components of $Scores[]$
```

▶ **Figure 7.1** A faster algorithm for vector space scores.

Inexact top-k document retrieval

- In most applications, it suffices to retrieve k documents whose scores are very close to those of the k – best
- By this we can potentially avoid computing the scores for most of N documents in the collection
- We use some heuristics and find a set A of documents that are potential candidates for top k   (k < |A| << N)
- Note that A does not necessarily contain the k top scoring documents for the query, but is likely to have many documents with scores near to those of top k
- Return the top k scoring documents in A

Index Elimination

- For a multiterm query, we only consider documents containing at least one of the query terms
- Additional Heuristics
  - We consider only those documents containing terms whose idf exceeds a preset threshold
    - Thus in the postings traversal, we only traverse the postings for terms with high idf. This has a fairly significant benefit as the postings list for low idf terms are generally long. With these removed, the set of documents for which we compute the cosine scores is greatly reduced
    - In short low idf terms are treated like stop words
    - The cut-off threshold can be adopted in query dependent manner
  - We contain the documents that contain many (as a special case, all) the query terms
    - Danger of this is : We may end up in fewer than k candidate documents in the output

Champion Lists

- Champion lists = fancy lists = top docs
- The idea is to precompute, for each term t in the dictionary, the set of r documents with highest weights for t The value of r is chosen in advance
- For tf-idf weighting, these would be the *r* documents with the highest tf values for term *t*. We call this set of *r* documents the *champion list* for term *t*.
- Now, given a query *q* we create a set *A* as follows: we take the union of the champion lists for each of the terms comprising *q*.
- We now restrict cosine computation to only the documents in *A*.
- A critical parameter in this scheme is the value *r*, which is highly application dependent. Intuitively, *r* should be large compared with *K*, especially if we use any form of the index elimination

Issue In champion lists

- One issue here is that the value *r* is set at the time of index construction, whereas *K* is application dependent and may not be available until the query is received; as a result we may (as in the case of index elimination) find ourselves with a set *A* that has fewer than *K* documents. There is no reason to have the same value of *r* for all terms in the dictionary; it could for instance be set to be higher for rarer terms.
- Note : r need not be same for all terms in dictionary. R could be greater for rarer terms

Static Quality scores and ordering

- In many search engines, we have available a measure of quality g(d) for each document d that is query independent and thus static
- Static quality measure has a value between 0 and 1
- Example: In context of news stories on the web, g(d) may be derived from the number of favourable reviews if the story by web surfers
- Net score = combination of (g(d) and query dependent score)
- For e.g. net score(q,d) = g(d) + cosine score of d with respect to q
  - Both components can have equal contributions assuming each is between 0 and 1 .
  - Other relative weightings are also possible, the effectiveness of our heuristics depends on the specific relative weighting

Ideas for static quality scores and ranking

- Idea 1:
    - Consider ordering the documents in the posting list for each term by decreasing order of g(d)
- Idea 2:
    - Extension of champion lists
    - Global champion list: for a well chosen r, we maintain for each term ‘t’, a global champion list of r documents with highest values of g(d) + tf-idf(t,d)
    - At query time, we only compute the net scores for documents in the union of global champion lists

Ideas for static quality scores and ranking

- Idea 3:
    - We maintain for each term t, two posting lists consisting of disjoint sets of documents, each sorted by g(d) values
        - First list -> high-> m docs with highest tf values for t
        - Second list -> low-> remaining docs containing t
    - While processing a query, scan only the high lists. If we get k docs in the process terminate. If not, go to the low lists

Impact ordering

- In all the postings lists described thus far, we order the documents consistently by some common ordering: typically by document ID or by static quality scores.
- Such a common ordering supports the concurrent traversal of all of the query terms' postings lists, computing the score for each document as we encounter it. Computing scores in this manner is sometimes referred to as document-at-atime scoring.
- We will now introduce a technique for inexact top-*K* retrieval in which the postings are not all ordered by a common ordering, thereby precluding such a concurrent traversal. We will therefore require scores to be "accumulated" one term at a time so that we have term-at-a-time scoring.
- The idea is to order the documents *d* in the postings list of term *t* by decreasing order of tf*t*,*d*. Thus, the ordering of documents will vary from one postings list to another, and we cannot compute scores by a concurrent traversal of the postings lists of all query terms.

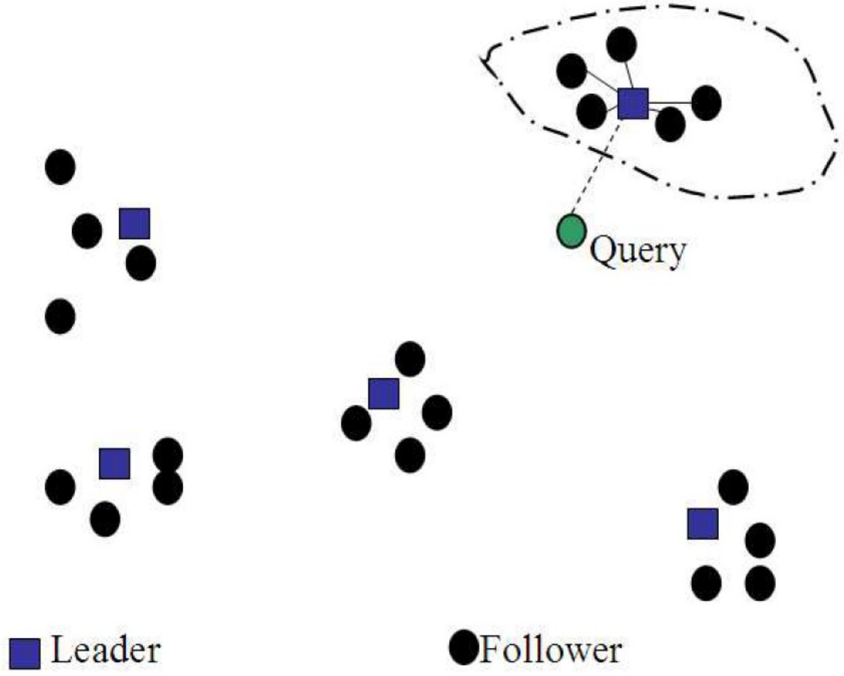
Impact ordering

- Given postings lists ordered by decreasing order of tft,d, two ideas have been found to significantly lower the number of documents for which we accumulate scores:
  - (1) when traversing the postings list for a query term t, we stop after considering a prefix of the postings list – either after a fixed number of documents r have been seen, or after the value of tft,d has dropped below a threshold;
  - (2) when accumulating scores, we consider the query terms in decreasing order of idf, so that the query terms likely to contribute the most to the final scores are considered first.
    - This latter idea too can be adaptive at the time of processing a query: as we get to query terms with lower idf, we can determine whether to proceed based on the changes in document scores from processing the previous query term. If these changes are minimal, we may omit accumulation from the remaining query terms, or alternatively process shorter prefixes of their postings lists.

Impact ordering

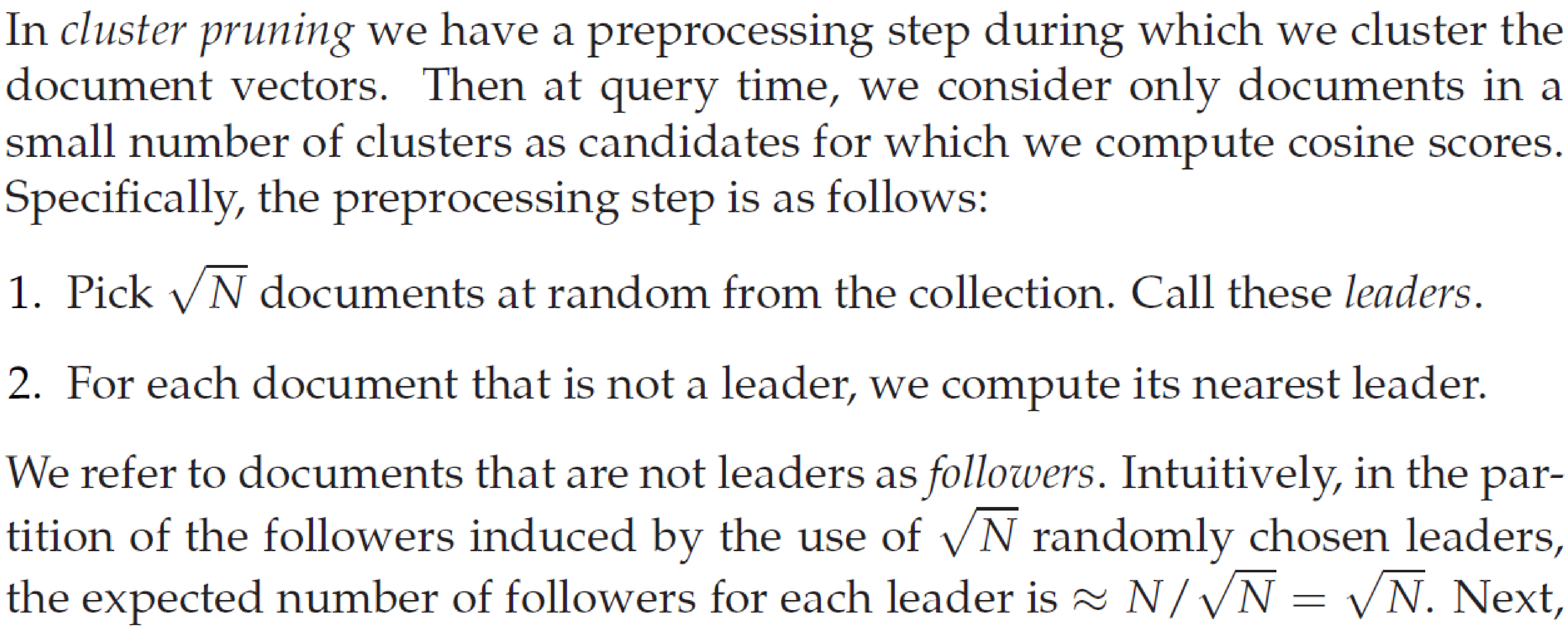
- We may also implement a version of static ordering in which each postings list is ordered by an additive combination of static and query-dependent scores. We would again lose the consistency of ordering across postings, thereby having to process query terms one at time accumulating scores for all documents as we go along. Depending on the particular scoring function, the postings list for a document may be ordered by other quantities than term frequency; under this more general setting, this idea known as impact ordering.

Cluster Pruning



► **Figure 7.3** Cluster pruning.

Cluster pruning



In *cluster pruning* we have a preprocessing step during which we cluster the document vectors. Then at query time, we consider only documents in a small number of clusters as candidates for which we compute cosine scores. Specifically, the preprocessing step is as follows:

1. Pick $\sqrt{N}$ documents at random from the collection. Call these *leaders*.
2. For each document that is not a leader, we compute its nearest leader.

We refer to documents that are not leaders as *followers*. Intuitively, in the partition of the followers induced by the use of $\sqrt{N}$ randomly chosen leaders, the expected number of followers for each leader is $\approx N/\sqrt{N} = \sqrt{N}$. Next,

Query processing in cluster pruning

- 1. Given a query *q*, find the leader *L* that is closest to *q*. This entails computing cosine similarities from *q* to each of the $\sqrt{N}$ leaders.
- 2. The candidate set *A* consists of *L* together with its followers. We compute the cosine scores for all documents in this candidate set.
  - Note: The use of randomly chosen leaders for clustering is fast and likely to reflect the distribution of the document vectors in the vector space: a region of the vector space that is dense in documents is likely to produce multiple leaders and thus a finer partition into sub-regions.

Variations of cluster pruning

- Variations of cluster pruning introduce additional parameters $b1$ and $b2$, both of which are positive integers. In the pre-processing step we attach
- each follower to its $b1$ closest leaders, rather than a single closest leader.
- At query time we consider the $b2$ leaders closest to the query q.
- Clearly, the basic scheme above corresponds to the case $b1 = b2 = 1$. Further, increasing $b1$ or $b2$ increases the likelihood of finding K documents that are more likely to be in the set of true top-scoring K documents, at the expense of more computation.

Points to ponder!

- When discussing champion lists, we simply used the *r* documents with the largest tf values to create the champion list for *t*. But when considering global champion lists, we used idf as well, identifying documents with the largest values of *g*(*d*) +tf-idf*t,d*. Why do we differentiate between these two cases?

- When query = a single term t, union of champion lists for each term is the same as that of t. Ranking by netscore=g(d)+ tf(d) idf(t,d) is equivalent to Ranking by netscore=g(d)+ tf(d) So use of global champion list within m=K suffices for finding K highest scoring documents.

Points to Ponder!

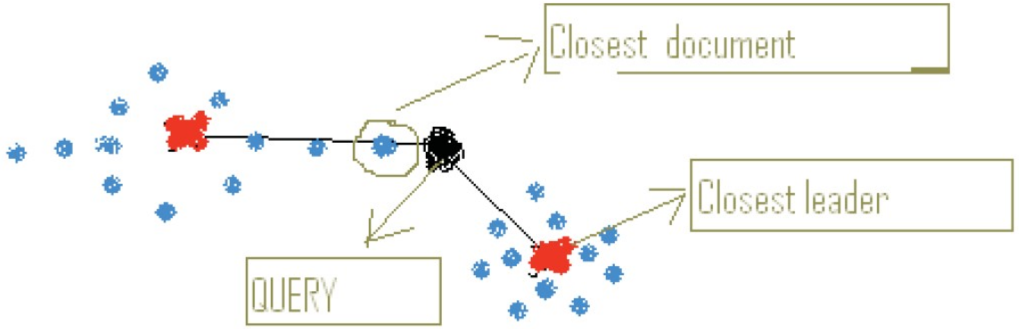
- If we were to only have one-term queries, explain why the use of global champion lists with $r = K$ suffices for identifying the K highest scoring documents. What is a simple modification to this idea if we were to only have s-term queries for any fixed integer $s > 1$?
- For a one-term query, the ordering of documents is independent of   of the term in the query. It only depends on the weight of the term in the documents. The truncated weight-ordered postings list contains the K documents with the highest weights. Thus, it is identical to the ranked list for the one-term query.

- Explain how the common global ordering by *g*(*d*) values in all high and low lists helps make the score computation efficient.
- In the general case, the postings list of query terms are traversed to find the documents which contains all or most of query terms and then scores are calculated for these documents to find the top K. Thus more importance is given to v(q) .v(d) in the short listing stage and g(d)is accounted for later. If common global ordering by g(d) is implemented, not just it increases the efficiency of results but also less documents would have to undergo score calculation process thereby reducing some work through at the cost of initial ordering.

Points to ponder!

- The nearest-neighbor problem in the plane is the following: given a set of *N* data points on the plane, we preprocess them into some data structure such that, given a query point *Q*, we seek the point in *N* that is closest to *Q* in Euclidean distance. Clearly cluster pruning can be used as an approach to the nearest-neighbor problem in the plane, if we wished to avoid computing the distance from *Q* to every one of the query points. Devise a simple example on the plane so that with two leaders, the answer returned by cluster pruning is incorrect (it is not the data point closest to *Q*).

Solution



Points to Ponder!

- Explain how the postings intersection algorithm first introduced in Section 1.3 can be adapted to find the smallest integer ω that contains all query terms.
- Move concurrently through the positional postings of all query terms, until a document D is found that contains all the query terms.
  - 1. Move the cursor on each postings to the first positional occurrence of the term.
  - 2. Measure the window size to be the distance between the leftmost and rightmost cursors.
  - 3. Repeat the following until some cursor exits document D:
  - 4. Pick the left-most cursor among all postings and move right to the next occurrence; measure the window size as above.
  - 5. Output the minimum of all of these measurements.

Adapt this procedure in previous question to work when not all query terms are present in a document

a. Check whether all k query terms are present in a document. If yes, apply algorithm of k nearest neighbor cluster pruning.

b. If k>=2,Make all possible combinations of (k-1) query terms and check for a document D which contains all terms of any of the combinations.

If yes, change the query[] to exclude all query terms not in that combination and recur.

If not, repeat step b until a document D is found. If k=1, return all documents containing the remaining query term.

Evaluation in IR

Mansi A. Radke

3 key things !

- To measure ad hoc information retrieval effectiveness in the standard way, we need a test collection consisting of three things:
- 1. A document collection
- 2. A test suite of information needs, expressible as queries
- 3. A set of relevance judgments, standardly a binary assessment of either *relevant* or *nonrelevant* for each query-document pair.

Gold standard / Ground truth

- With respect to a user information need, a document in the test collection is given a binary classification as either relevant or nonrelevant. This decision is referred to as the *gold standard* or *ground truth* judgment of relevance.

- This process is called as *Annotation*

- Gold labour indicates the people who do this annotation

How many queries?

- As a rule of thumb, 50 information needs has usually been found to be a sufficient minimum.

- (Whatever be the size of the corpus)

- Relevance is assessed relative to an information need, *not* a query

- A document is relevant if it addresses the stated information need, not because it just happens to contain all the words in the query. This distinction is often misunderstood in practice, because the information need is not overt

Development test collection

- Many systems contain various weights (often known as parameters) that can be adjusted to tune system performance. It is wrong to report results on a test collection which were obtained by tuning these parameters to maximize performance on that collection. That is because such tuning overstates the expected performance of the system, because the weights will be set to maximize performance on one particular set of queries rather than for a random sample of queries. In such cases, the correct procedure is to have one or more *development test collections*, and to tune the parameters on the development test collection. The tester then runs the system with those weights on the test collection and reports the results on that collection as an unbiased estimate of performance.

Precision and Recall

	Relevant	Non-relevant	
Retrieved	$A \cap B$	$\bar{A} \cap B$	B
Not retrieved	$A \cap \bar{B}$	$\bar{A} \cap \bar{B}$	$\bar{B}$
	A	$\bar{A}$	

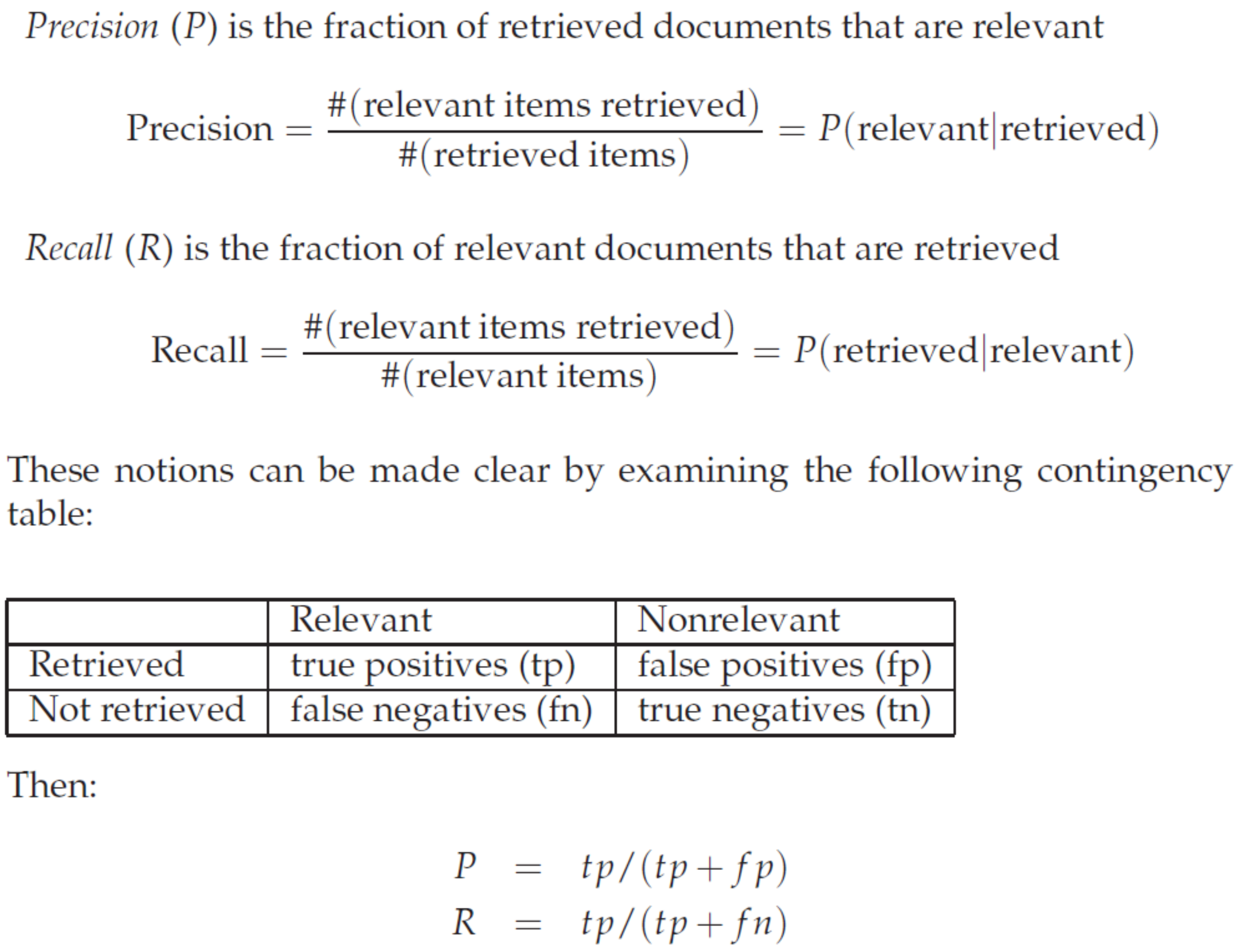
$N = \text{number of documents in the system}$

- **Precision** $\frac{|A \cap B|}{|B|}$
- **Recall** $\frac{|A \cap B|}{|A|}$
- **Accuracy** $\frac{|A \cap B| + |\bar{A} \cap \bar{B}|}{N}$

Careful! May not make sense to use this!

Which one, P or R, is more important?

Evaluation in unranked retrieval



Precision (P) is the fraction of retrieved documents that are relevant

$$\text{Precision} = \frac{\#(\text{relevant items retrieved})}{\#(\text{retrieved items})} = P(\text{relevant}|\text{retrieved})$$

Recall (R) is the fraction of relevant documents that are retrieved

$$\text{Recall} = \frac{\#(\text{relevant items retrieved})}{\#(\text{relevant items})} = P(\text{retrieved}|\text{relevant})$$

These notions can be made clear by examining the following contingency table:

	Relevant	Nonrelevant
Retrieved	true positives (tp)	false positives (fp)
Not retrieved	false negatives (fn)	true negatives (tn)

Then:

$$\begin{aligned} P &= tp/(tp+fp) \\ R &= tp/(tp+fn) \end{aligned}$$

Accuracy

- An obvious alternative that may occur to the reader is to judge an information retrieval system by its *accuracy*, that is, the fraction of its classifications that are correct.
- Accuracy = $(tp + tn)/(tp + f\,p + f\,n + tn)$.

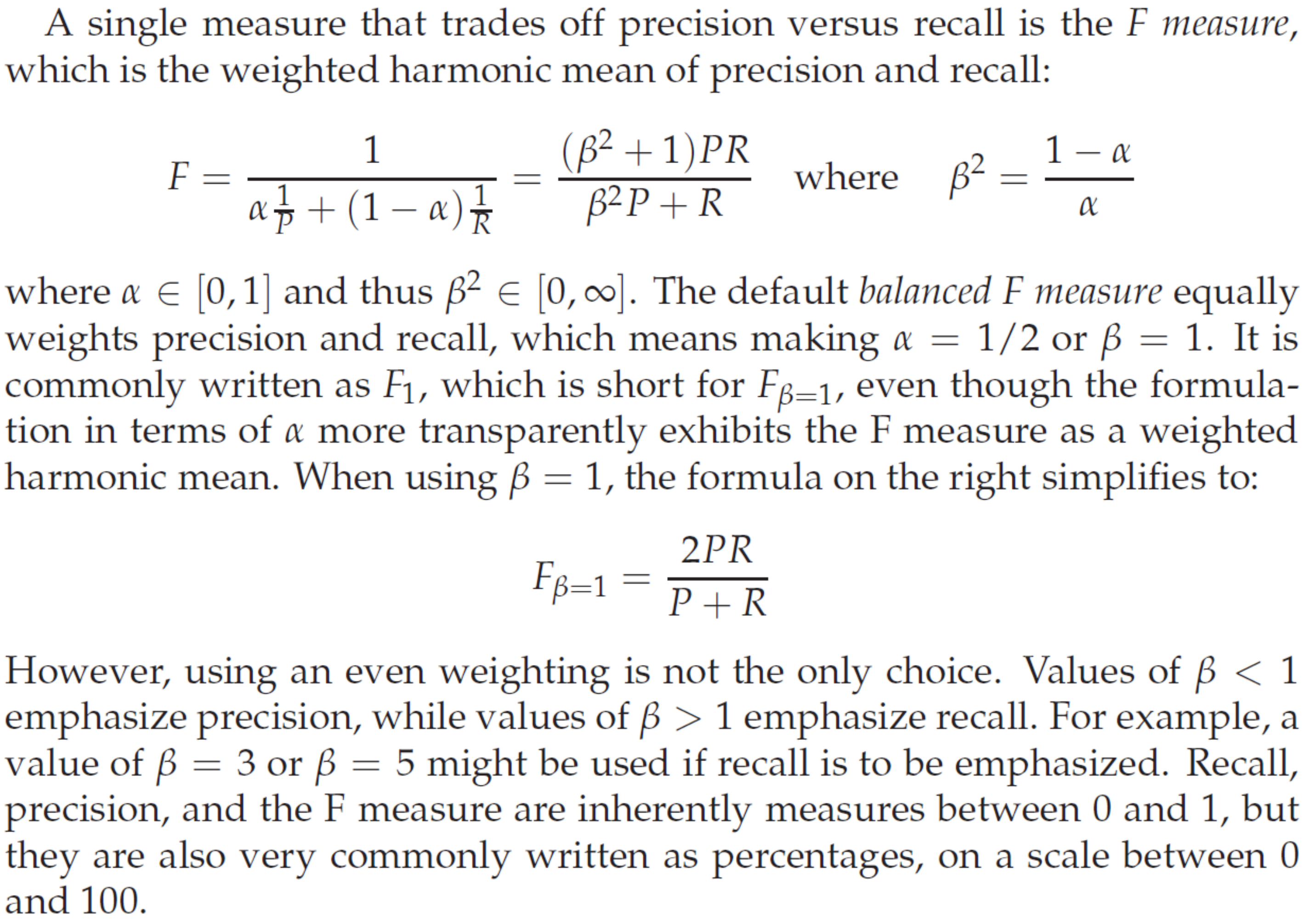
Why accuracy is not a good measure?

- There is a good reason why accuracy is not an appropriate measure for information retrieval problems. In almost all circumstances, the data is extremely skewed: normally over 99.9% of the documents are in the nonrelevant category. A system tuned to maximize accuracy can appear to perform well by simply deeming all documents nonrelevant to all queries. Even if the system is quite good, trying to label some documents as relevant will almost always lead to a high rate of false positives. However, labeling all documents as nonrelevant is completely unsatisfying to an information retrieval system user.

Why precision and recall both?

- The advantage of having the two numbers for precision and recall is that one is more important than the other in many circumstances. Typical web surfers would like every result on the first page to be relevant (high precision) but have not the slightest interest in knowing let alone looking at every document that is relevant. In contrast, various professional searchers such as paralegals and intelligence analysts are very concerned with trying to get as high recall as possible, and will tolerate fairly low precision results in order to get it. Individuals searching their hard disks are also often interested in high recall searches.

F1 measure



A single measure that trades off precision versus recall is the *F measure*, which is the weighted harmonic mean of precision and recall:

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R} \quad \text{where} \quad \beta^2 = \frac{1-\alpha}{\alpha}$$

where $\alpha \in [0,1]$ and thus $\beta^2 \in [0,\infty]$. The default *balanced F measure* equally weights precision and recall, which means making $\alpha = 1/2$ or $\beta = 1$. It is commonly written as F_1, which is short for $F_{\beta=1}$, even though the formulation in terms of α more transparently exhibits the F measure as a weighted harmonic mean. When using $\beta = 1$, the formula on the right simplifies to:

$$F_{\beta=1} = \frac{2PR}{P+R}$$

However, using an even weighting is not the only choice. Values of $\beta < 1$ emphasize precision, while values of $\beta > 1$ emphasize recall. For example, a value of $\beta = 3$ or $\beta = 5$ might be used if recall is to be emphasized. Recall, precision, and the F measure are inherently measures between 0 and 1, but they are also very commonly written as percentages, on a scale between 0 and 100.

Values of Beta

- <1
- >1
- =1

Arithmetic, geometric and harmonic mean

- Arithmetic mean of 2 and 18 is 10
- What is the Geometric Mean of 2 and 18? First we multiply them: 2 × 18 = 36. Then (as there are two numbers) take the square root: √36 = 6.
- Harmonic mean of 2 and 18 is
  - ½ + 1/18
  - = 10/18
  - As there are 2 numbers we divide by 2
  - 10/(18*2)
  - Then take reciprocal = reciprocal of 10/36 = 36/10 = 3.6

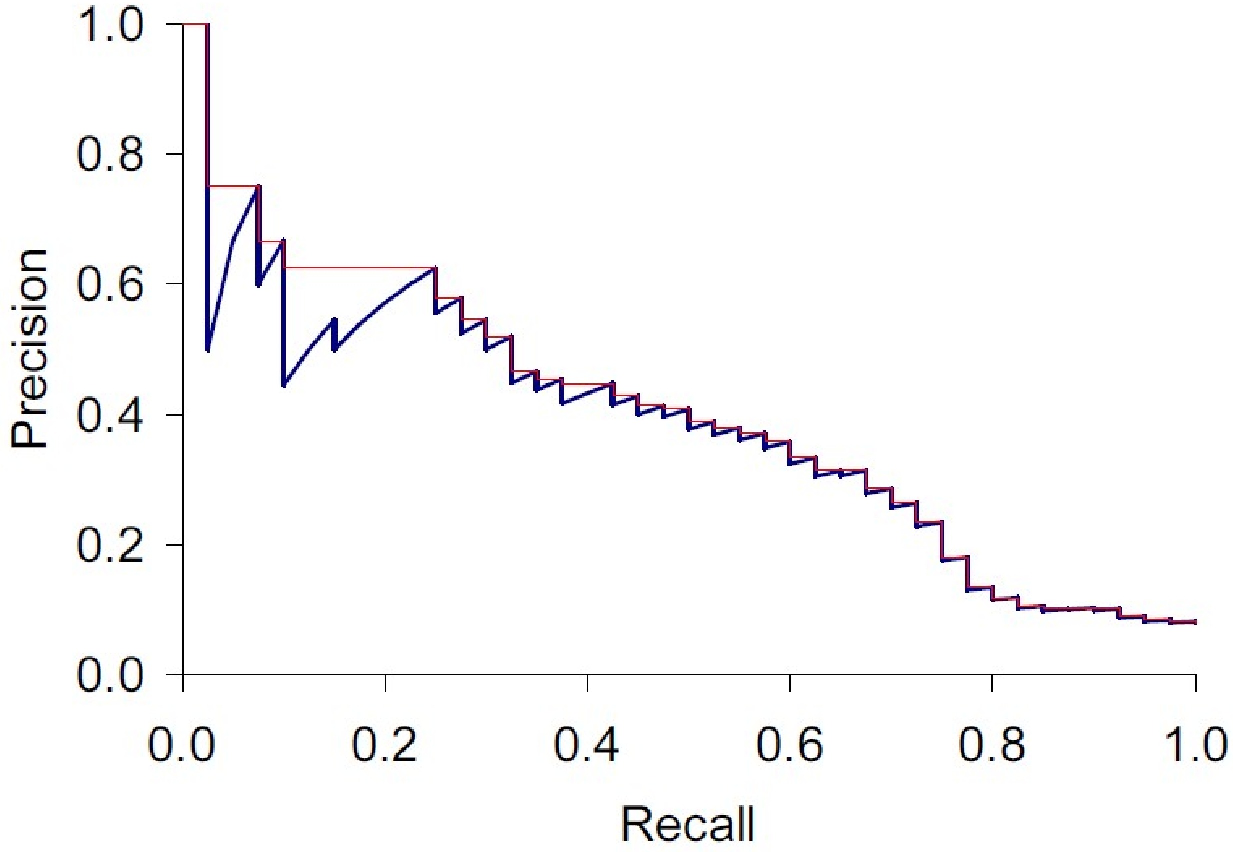
Why harmonic mean?

- we can always get 100% recall by just returning all documents, and therefore we can always get a 50% arithmetic mean by the same process. This strongly suggests that the arithmetic mean is an unsuitable measure to use.
- In contrast, if we assume that 1 document in 10,000 is relevant to the query, the harmonic mean score of this strategy is 0.02%.
- The harmonic mean is always less than or equal to the arithmetic mean and the geometric mean.
- When the values of two numbers differ greatly, the harmonic mean is closer to their minimum than to their arithmetic mean

Evaluation of ranked retrieval

- In a ranked retrieval context, appropriate sets of retrieved documents are naturally given by the top *k* retrieved documents. For each such set, precision and recall values can be plotted to give a *precision-recall curve*, such as the one shown in next slide
- Precision-recall curves have a distinctive saw-tooth shape: if the $(k + 1)$th document retrieved is nonrelevant then recall is the same as for the top *k* documents, but precision has dropped. If it is relevant, then both precision and recall increase, and the curve jags up and to the right.

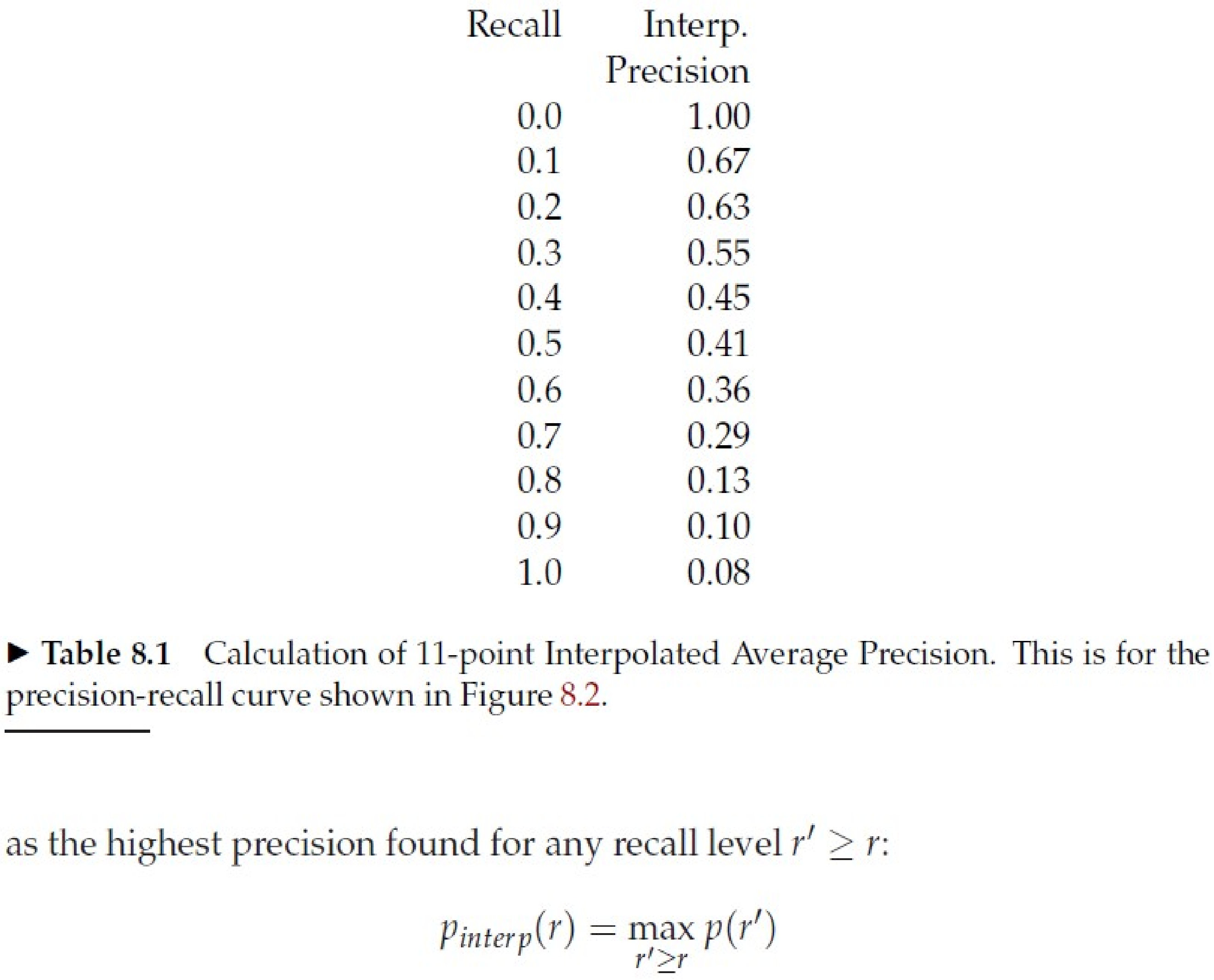
Precision recall curve



► **Figure 8.2** Precision/recall graph.

- It is often useful to remove these jiggles and the standard way to do this is with an interpolated precision: the *interpolated precision pinterp* at a certain recall level r is defined as the highest precision found for any recall level $r' \geq r$:

Interpolated precision



Recall	Interp. Precision
0.0	1.00
0.1	0.67
0.2	0.63
0.3	0.55
0.4	0.45
0.5	0.41
0.6	0.36
0.7	0.29
0.8	0.13
0.9	0.10
1.0	0.08

► **Table 8.1** Calculation of 11-point Interpolated Average Precision. This is for the precision-recall curve shown in Figure 8.2.

as the highest precision found for any recall level $r' \geq r$:

$$p_{interp}(r) = \max_{r' \geq r} p(r')$$

11 point interpolated average precision

- For each information need, the interpolated precision is measured
- at the 11 recall levels of 0.0, 0.1, 0.2, . . . , 1.0. For the precision-recall curve. For each recall level, we then calculate the arithmetic mean of the interpolated precision at that recall level for each information need in the test collection

Average precision

- average of the precision value obtained for the set of top *k* documents existing after each relevant document is retrieved

Mean average precision

$$\mathrm{MAP}(Q) = \frac{1}{|Q|}\sum_{j=1}^{|Q|}\frac{1}{m_j}\sum_{k=1}^{m_j}\mathrm{Precision}(R_{jk})$$

Note:

- Using MAP, fixed recall levels are not chosen, and there is no interpolation

PRECISION AT *k*

- Precision at the kth document

R-PRECISION

- Count the number of relevant documents say k

- Calculate the precision at that point i.e. kth document.

- Report it as the R-precision

BREAK-EVEN POINT

- When P = R!

Normalised discounted cumulative gain

$$\mathrm{NDCG}(Q,k) = \frac{1}{|Q|}\sum_{j=1}^{|Q|} \mathrm{Z}_{kj} \sum_{m=1}^{k} \frac{2^{R(j,m)}-1}{\log_2(1+m)},$$

- In large collection settings, use of binary assessments failed to
- distinguish “very good” and “fair” retrieval systems, so NDCG is used

Non-binary assessments and NDCG

- For each relevance level, assign a gain value
- **nDCG** normalised Discounted Cumulative Gain
    - **Gain vector** from ranked results

      $G = \langle 5, 10, 0, 5, 1, 10, 0, 0, \ldots \rangle$
    - **Cumulative gain** vector

      $CG[k] = \sum_{i=1}^{k} G[i]$ $\quad$ $CG = \langle 5, 15, 15, 20, 21, 31, 31, 31, \ldots \rangle$
        - Apply **discount** function - typically logarithm

          $DCG[k] = \sum_{i=1}^{k} \frac{G[i]}{\log_2(1+i)}$
    - Compute **ideal** cumulative gain vector $G = \langle 10,10,5,5,1,0,0,0\ldots \rangle$
    - **Normalize** using ideal cumulative gain vector

$$nDCG[k] = \frac{DCG[k]}{DCG'[k]}$$

highly relevant	10
relevant	5
marginally relevant	1
non relevant	0
malicious	-1

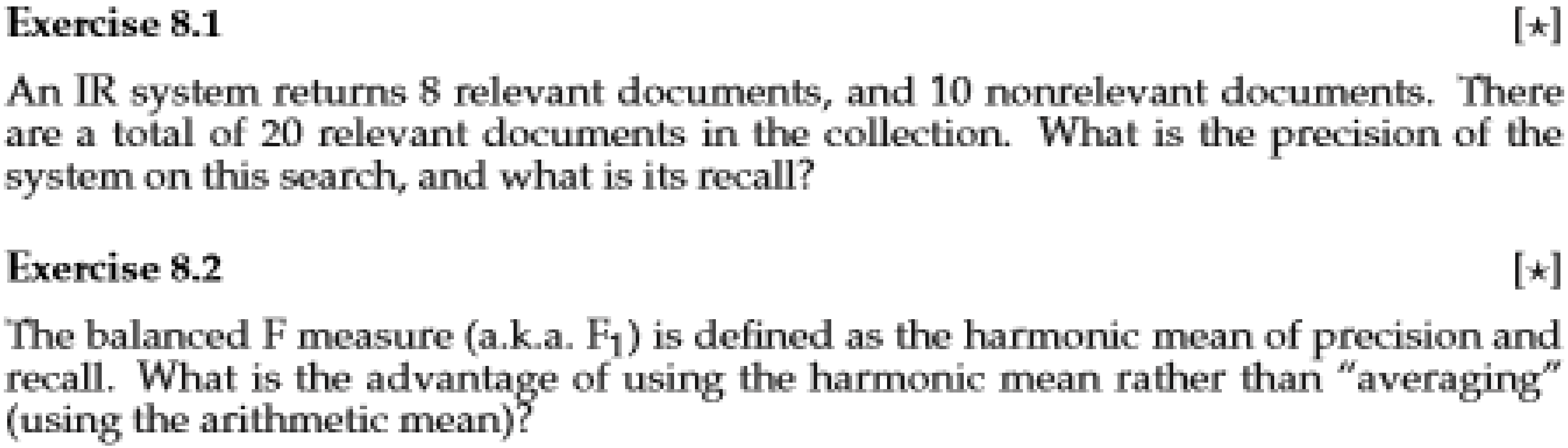
Difficulty in IR

- Information Retrieval is an Empirical Discipline
- •Lack of “Correct answer”
- •Requiring thorough and careful evaluation

Relevance

- “Something (A) is relevant to a task (T) if it increases the likelihood of accomplishing the goal (G), which is implied by T.”[Hjørland & Sejer Christensen, 2002]
- Is subjective - and not entirely so!
- • A document is judged relevant within the context of the query (or more precisely, *the information need*)
- • Who judges?
- • What is useful?

Problems

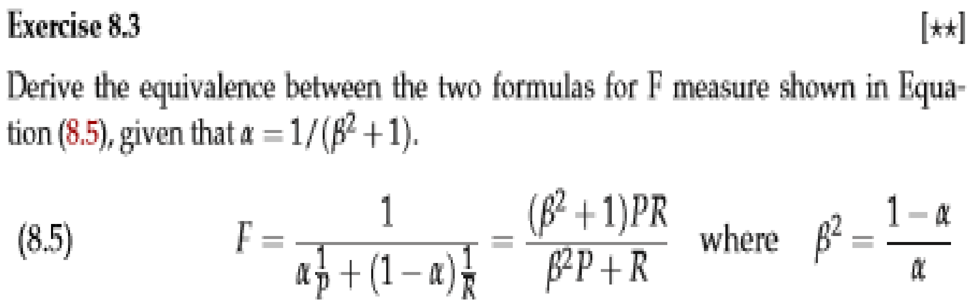


Exercise 8.1 [⋆]

An IR system returns 8 relevant documents, and 10 nonrelevant documents. There are a total of 20 relevant documents in the collection. What is the precision of the system on this search, and what is its recall?

Exercise 8.2 [⋆]

The balanced F measure (a.k.a. F_1) is defined as the harmonic mean of precision and recall. What is the advantage of using the harmonic mean rather than "averaging" (using the arithmetic mean)?



Exercise 8.3 [⋆⋆]

Derive the equivalence between the two formulas for F measure shown in Equation (8.5), given that $\alpha = 1/(\beta^2+1)$.

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R} \quad \text{where} \quad \beta^2 = \frac{1-\alpha}{\alpha} \tag{8.5}$$

Exercise 8.4 [⋆]

What are the possible values for interpolated precision at a recall level of 0?

Exercise 8.5 [⋆⋆]

Must there always be a break-even point between precision and recall? Either show there must be or give a counter-example.

Exercise 8.6 [⋆⋆]

What is the relationship between the value of F_1 and the break-even point?

Exercise 8.7 [⋆⋆]

The *Dice coefficient* of two sets is a measure of their intersection scaled by their size (giving a value in the range 0 to 1):

$$\text{Dice}(X, Y) = \frac{2|X \cap Y|}{|X| + |Y|}$$

Show that the balanced F-measure (F_1) is equal to the Dice coefficient of the retrieved and relevant document sets.

Exercise 8.8 [⋆]

Consider an information need for which there are 4 relevant documents in the collection. Contrast two systems run on this collection. Their top 10 results are judged for relevance as follows (the leftmost item is the top ranked search result):

System 1	R	N	R	N	N	N	N	N	R	R
System 2	N	R	N	N	R	R	R	N	N	N

a. What is the MAP of each system? Which has a higher MAP?

b. Does this result intuitively make sense? What does it say about what is important in getting a good MAP score?

c. What is the R-precision of each system? (Does it rank the systems the same as MAP?)

Exercise 8.9 [⋆⋆]

The following list of Rs and Ns represents relevant (R) and nonrelevant (N) returned documents in a ranked list of 20 documents retrieved in response to a query from a collection of 10,000 documents. The top of the ranked list (the document the system thinks is most likely to be relevant) is on the left of the list. This list shows 6 relevant documents. Assume that there are 8 relevant documents in total in the collection.

R R N N N   N N N R N   R N N N R   N N N N R

a. What is the precision of the system on the top 20?

b. What is the F_1 on the top 20?

c. What is the uninterpolated precision of the system at 25% recall?

d. What is the interpolated precision at 33% recall?

e. Assume that these 20 documents are the complete result set of the system. What is the MAP for the query?

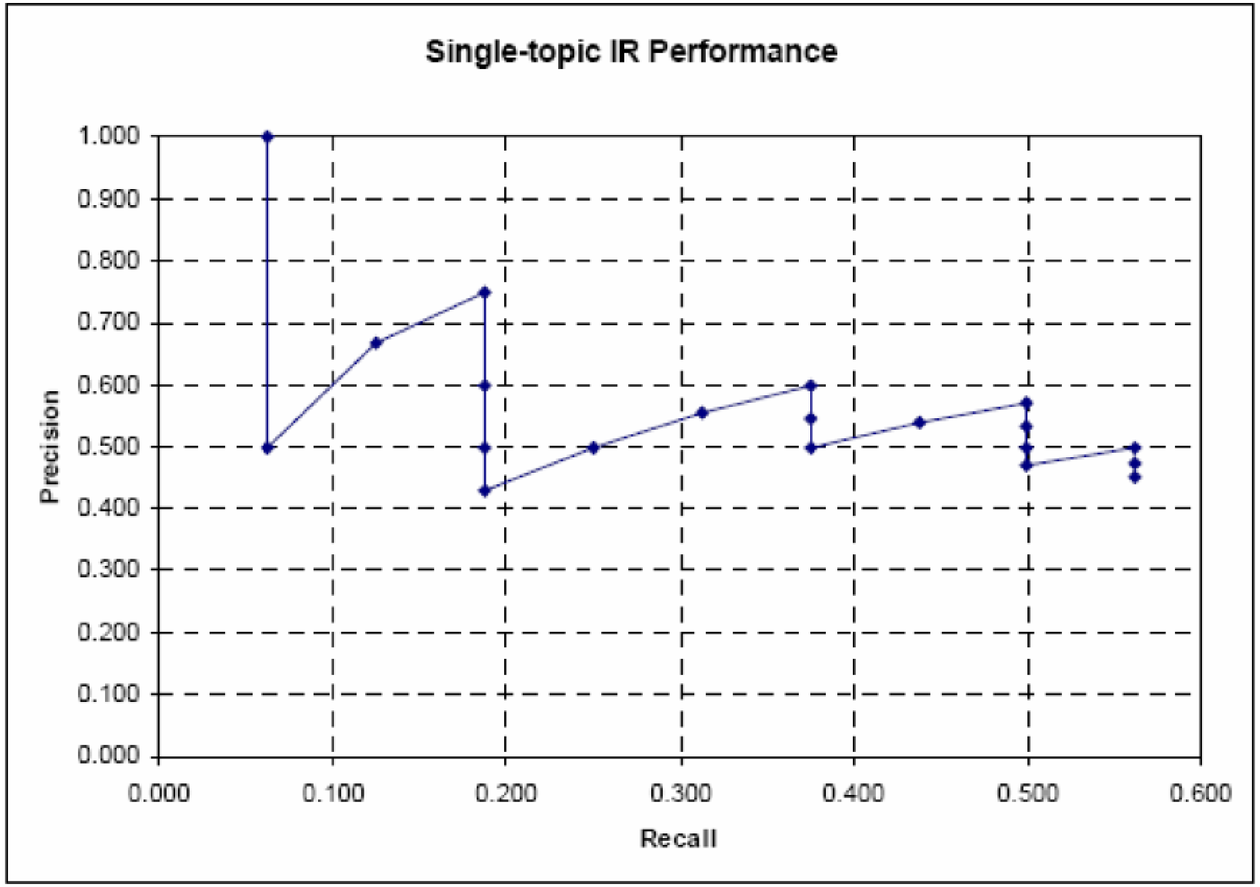
Assume, now, instead, that the system returned the entire 10,000 documents in a ranked list, and these are the first 20 results returned.

f. What is the largest possible MAP that this system could have?

g. What is the smallest possible MAP that this system could have?

h. In a set of experiments, only the top 20 results are evaluated by hand. The result in (e) is used to approximate the range (f)–(g). For this example, how large (in absolute terms) can the error for the MAP be by calculating (e) instead of (f) and (g) for this query?

The following is the uninterpolated Precision-Recall Graph of an IR system, for one topic. You know that 20 hits were retrieved, and that there are 16 relevant documents for this topic (not all of which are retrieved).



a. What does the interpolated graph look like? Draw neatly on the graph above.

b. In the diagram below, each box represents a hit. Based on the above Precision-Recall graph, which hits are relevant? Write an “R” on the relevant hits. Leave the non-relevant hits alone.

□	□	□	□	□	□	□	□	□	□
1	2	3	4	5	6	7	8	9	10
□	□	□	□	□	□	□	□	□	□
11	12	13	14	15	16	17	18	19	20

c. What is the MAP?

d. What is the R-precision?

e. What is Precision at 10?

Pooling

- For large modern collections, it is usual for relevance to be assessed only for a subset of the documents for each query. The most standard approach is *pooling*, where relevance is assessed over a subset of the collection that is formed from the top *k* documents returned by a number of different IR systems (usually the ones to be evaluated), and perhaps other sources such as the results of Boolean keyword searches or documents found by expert searchers in an interactive process.

Kappa measure

		Judge 2 Relevance		
		Yes	No	Total
Judge 1 Relevance	Yes	300	20	320
	No	10	70	80
	Total	310	90	400

Observed proportion of the times the judges agreed

$P(A) = (300 + 70)/400 = 370/400 = 0.925$

Pooled marginals

$P(\mathit{nonrelevant}) = (80 + 90)/(400 + 400) = 170/800 = 0.2125$

$P(\mathit{relevant}) = (320 + 310)/(400 + 400) = 630/800 = 0.7878$

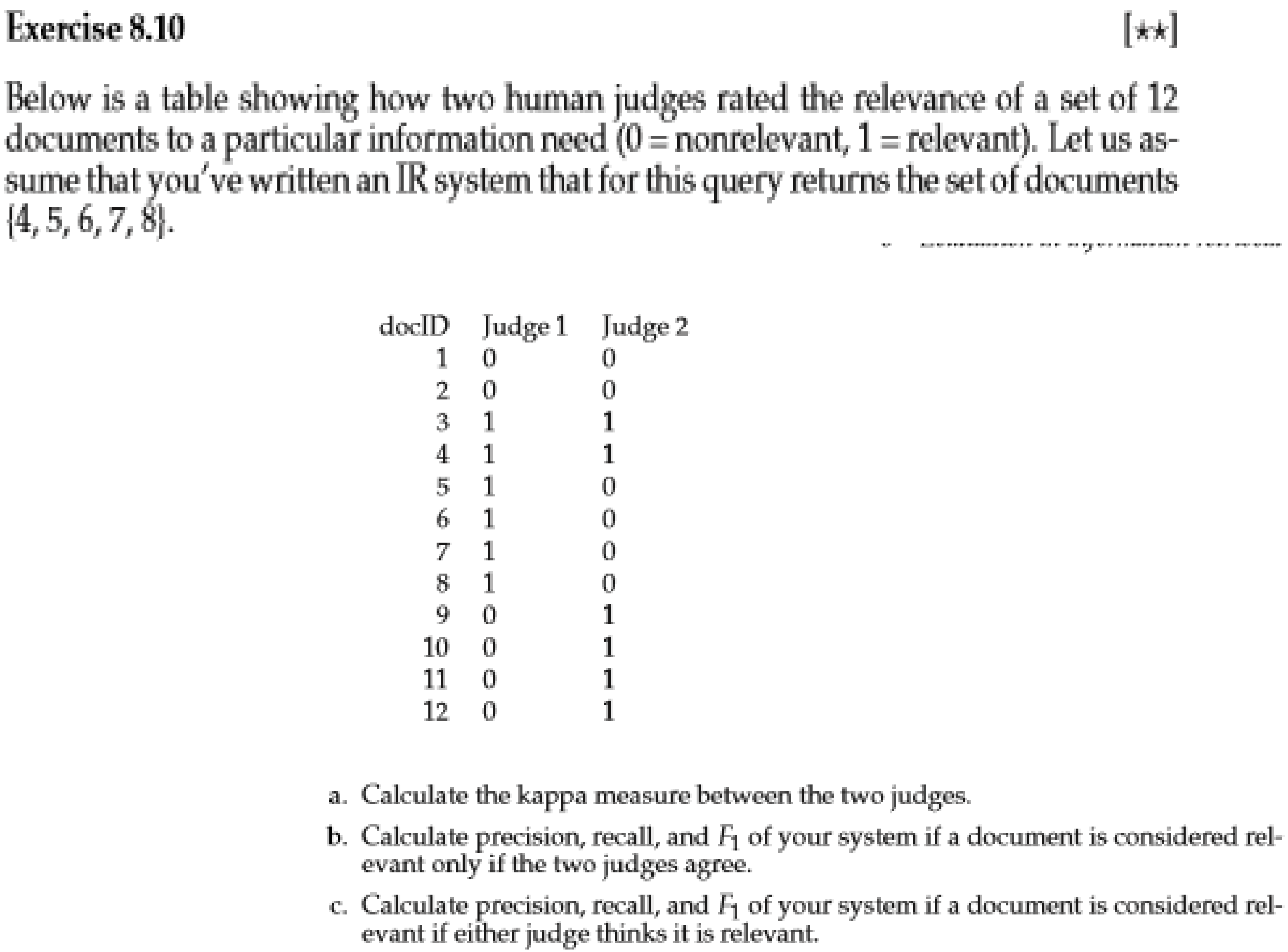
Probability that the two judges agreed by chance

$P(E) = P(\mathit{nonrelevant})^2 + P(\mathit{relevant})^2 = 0.2125^2 + 0.7878^2 = 0.665$

Kappa statistic

$\kappa = (P(A) - P(E))/(1 - P(E)) = (0.925 - 0.665)/(1 - 0.665) = 0.776$

▶ **Table 8.2** Calculating the kappa statistic.



Exercise 8.10 [⋆⋆]

Below is a table showing how two human judges rated the relevance of a set of 12 documents to a particular information need (0 = nonrelevant, 1 = relevant). Let us assume that you've written an IR system that for this query returns the set of documents {4, 5, 6, 7, 8}.

docID	Judge 1	Judge 2
1	0	0
2	0	0
3	1	1
4	1	1
5	1	0
6	1	0
7	1	0
8	1	0
9	0	1
10	0	1
11	0	1
12	0	1

a. Calculate the kappa measure between the two judges.

b. Calculate precision, recall, and F_1 of your system if a document is considered relevant only if the two judges agree.

c. Calculate precision, recall, and F_1 of your system if a document is considered relevant if either judge thinks it is relevant.

Relevance Feedback in Information retrieval

- Synonymy
  - E.g. aircraft and airplane
- Global vs Local methods
  - Global methods are techniques for expanding or reformulating query terms independent of the query and results returned from it, so that changes in the query wording will cause the new query to match other semantically similar terms
    - Query expansion with Thesaurus or wordnet
    - Query expansion through spelling correction
    - Automatic thesaurus generation
  - Local methods
    - Relevance feedback
    - Pseudo relevance feedback
    - Indirect relevance feedback

Relevance feedback process

- The user issues a (short, simple) query.
- The system returns an initial set of retrieval results.
- The user marks some returned documents as relevant or nonrelevant.
- The system computes a better representation of the information need based on the user feedback.
- The system displays a revised set of retrieval results.

Relevance feedback can go through one or more iterations of this sort.

The idea behind relevance feedback

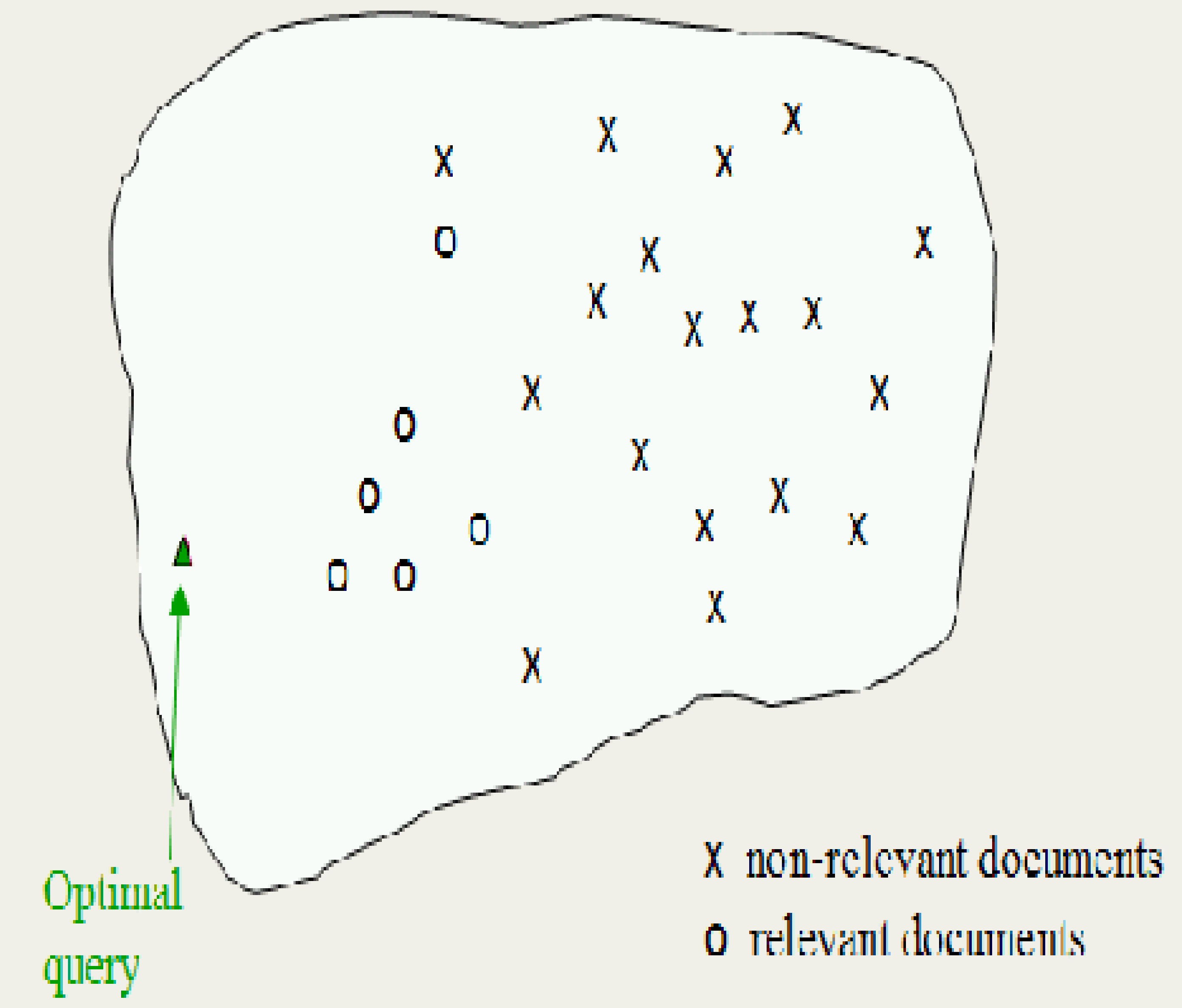
- The process exploits the idea that it may be difficult to formulate a good query when you don’t know the collection well, but it is easy to judge particular documents, and so it makes sense to engage in iterative query refinement of this sort.
- In such a scenario, relevance feedback can also be effective in tracking a user’s evolving information need:
  - seeing some documents may lead users to refine their understanding of the information they are seeking.
  - E.g. Image retrieval

- Rocchio Algorithm

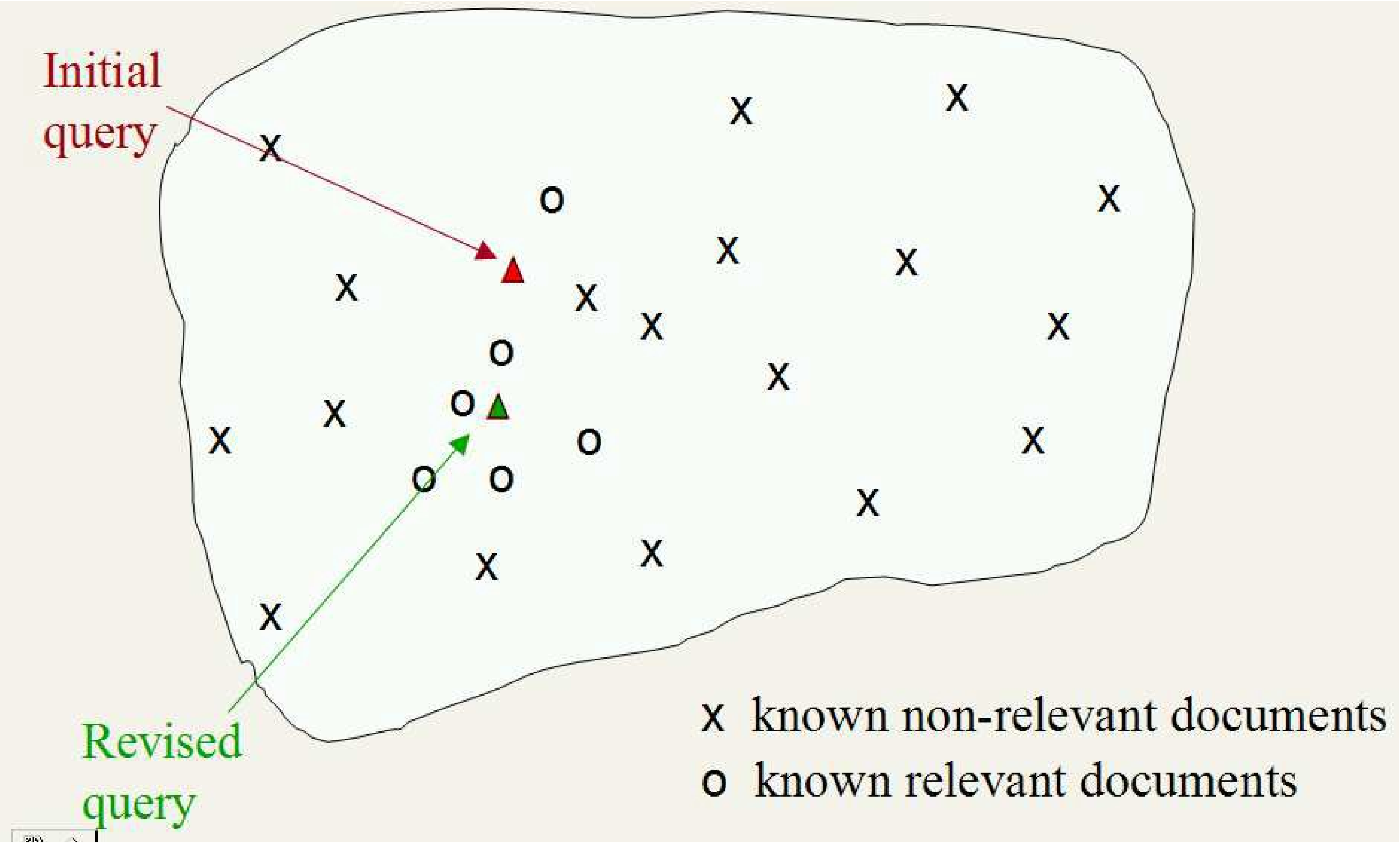
  Move the centroid towards of relevant documents and away from non relevant documents

$$\vec{q}_{opt} = \arg\max_{\vec{q}}[\text{sim}(\vec{q}, C_r) - \text{sim}(\vec{q}, C_{nr})],$$

$$\vec{q}_{opt} = \frac{1}{|C_r|}\sum_{\vec{d}_j \in C_r}\vec{d}_j - \frac{1}{|C_{nr}|}\sum_{\vec{d}_j \in C_{nr}}\vec{d}_j$$



Revised query is more close to the optimal query!



Formula for modified query vector

- Relevance feedback can improve both recall and precision. But, in practice, it has been shown to be most useful for increasing recall in situations where recall is important. This is partly because the technique expands the query, but it is also partly an effect of the use case: when they want high recall, users can be expected to tak [illegible] search.

$$\vec{q}_m = \alpha\vec{q}_0 + \beta\frac{1}{|D_r|}\sum_{\vec{d}_j\in D_r}\vec{d}_j - \gamma\frac{1}{|D_{nr}|}\sum_{\vec{d}_j\in D_{nr}}\vec{d}_j$$

Positive vs negative feedback

- Positive feedback also turns out to be much more valuable than negative feedback, and so most IR systems set $\gamma < \beta$. Reasonable values might be $\alpha = 1$, $\beta = 0.75$, and $\gamma = 0.15$. In fact, many systems, such as the image search system in Figure 9.1, allow only positive feedback, which is equivalent to setting $\gamma = 0$. Another alternative is to use only the marked nonrelevant document which received the highest ranking from the IR system as negative feedback (here, $|Dnr| = 1$). While many of the experimental results comparing various relevance feedback variants are rather inconclusive
- some studies have suggested that this variant, called *Ide dec-hi* is the most effective or at least the most consistent performer.

Probabilistic relevance feedback

- Rather than reweighting the query in a vector space, if a user has told us some relevant and nonrelevant documents, then we can proceed to build a classifier. One way of doing this is with a Naive Bayes probabilistic model. If *R* is a Boolean indicator variable expressing the relevance of a document, then we can estimate *P*(*xt* = 1|*R*), the probability of a term *t* appearing in a document, depending on whether it is relevant or not, as:

$$\begin{aligned}\hat{P}(x_t = 1|R = 1) &= |VR_t|/|VR| \\ \hat{P}(x_t = 1|R = 0) &= (df_t - |VR_t|)/(N - |VR|)\end{aligned}$$

When does relevance feedback work? Assumptions behind relevance feedback

- Firstly, the user has to have sufficient knowledge to be able to make an initial query which is at least somewhere close to the documents they desire.

- Secondly, the relevance feedback approach requires relevant documents to be similar to each other. That is, they should cluster. Ideally, the term distribution in all relevant documents will be similar to that in the documents marked by the users, while the term distribution in all nonrelevant documents will be different from those in relevant documents. Things will work well if all relevant documents are tightly clustered around a single prototype, or, at least, if there are different prototypes, if the relevant documents have significant vocabulary overlap, while similarities between relevant and nonrelevant documents are small. Implicitly, the Rocchio relevance feedback model treats relevant documents as a single *cluster*, which it models via the centroid of the cluster. This approach does not work as well if the relevant documents are a multimodal class, that is, they consist of several clusters of documents within the vector space.

Why relevance feedback fails?

- Relevance feedback is not necessarily popular with users. Users are often reluctant to provide explicit feedback, or in general do not wish to prolong the search interaction.
- Furthermore, it is often harder to understand why a particular document was retrieved after relevance feedback is applied.
- Relevance feedback can also have practical problems.
    - The long queries that are generated by straightforward application of relevance feedback techniques are inefficient for a typical IR system.
    - This results in a high computing cost for the retrieval and potentially long response times for the user.
    - A partial solution to this is to only reweight certain prominent terms in the relevant documents, such as perhaps the top 20 terms by term frequency. Some experimental results have also suggested that using a limited number of terms like this may give better results (Harman 1992) though other work has suggested that using more terms is better in terms of retrieved document quality

Cases where relevance feedback alone is not sufficient include:

- Misspellings. If the user spells a term in a different way to the way it is spelled in any document in the collection, then relevance feedback is unlikely to be effective. This can be addressed by the spelling correction techniques.

- Cross-language information retrieval. Documents in another language are not nearby in a vector space based on term distribution. Rather, documents in the same language cluster more closely together.

- Mismatch of searcher’s vocabulary versus collection vocabulary. If the user searches for laptop but all the documents use the term notebook computer, then the query will fail, and relevance feedback is again most likely ineffective

Evaluation of relevance feedback

- Interactive relevance feedback can give very substantial gains in retrieval performance. Empirically, one round of relevance feedback is often very useful. Two rounds is sometimes marginally more useful.
- Successful use of relevance feedback requires enough judged documents, otherwise the process is unstable in that it may drift away from the user’s information need.
    - Accordingly, having at least five judged documents is recommended.

Evaluation of relevance feedback

- The obvious first strategy is to start with an initial query *q*0 and to compute a precision-recall graph. Following one round of feedback from the user, we compute the modified query *qm* and again compute a precision-recall graph. Here, in both rounds we assess performance over all documents in the collection, which makes comparisons straightforward. If we do this, we find spectacular gains from relevance feedback: gains on the order of 50%inmean average precision.
- But unfortunately it is cheating. The gains are partly due to the fact that known relevant documents (judged by the user) are now ranked higher.
- Fairness demands that we should only evaluate with respect to documents not seen by the user

Evaluation of relevance feedback

- A second idea is to use documents in the *residual collection* (the set of documents minus those assessed relevant) for the second round of evaluation.
- This seems like a more realistic evaluation. Unfortunately, the measured performance can then often be lower than for the original query. This is particularly the case if there are few relevant documents, and so a fair proportion of them have been judged by the user in the first round.
- The relative performance of variant relevance feedback methods can be validly compared, but it is difficult to validly compare performance with and without relevance feedback because the collection size and the number of relevant documents changes from before the feedback to after it.

Evaluation of relevance feedback

- A third method is to have two collections, one which is used for the initial query and relevance judgments, and the second that is then used for comparative evaluation. The performance of both $q0$ and qm can be validly compared on the second collection.
- Perhaps the best evaluation of the utility of relevance feedback is to do user studies of its effectiveness, in particular by doing a time-based comparison:
- how fast does a user find relevant documents with relevance feedback vs. another strategy (such as query reformulation), or alternatively, how many relevant documents does a user find in a certain amount of time. Such notions of user utility are fairest and closest to real system usage.

Pseudo relevance feedback

- *Pseudo relevance feedback*, also known as *blind relevance feedback*, provides a method for automatic local analysis. It automates the manual part of relevance feedback, so that the user gets improved retrieval performance without an extended interaction. The method is to do normal retrieval to find an initial set of most relevant documents, to then *assume* that the top *k* ranked documents are relevant, and finally to do relevance feedback as before under this assumption.

Implicit Relevance Feedback

- We can also use indirect sources of evidence rather than explicit feedback on relevance as the basis for relevance feedback. This is often called i*mplicit (relevance) feedback*.
- Implicit feedback is less reliable than explicit feedback, but is more useful than pseudo relevance feedback, which contains no evidence of user judgments.
- Moreover, while users are often reluctant to provide explicit feedback, it is easy to collect implicit feedback in large quantities for a high volume system, such as a web search engine.

Global methods for query reformulation

- **Vocabulary tools for query reformulation**
  - The IR system might also suggest search terms by means of a thesaurus or a controlled vocabulary.
  - A user can also be allowed to browse lists of the terms that are in the inverted index, and thus find good terms that appear in the collection
- **Query expansion**
  - The most common form of query expansion is global analysis, using some form of thesaurus. For each term *t* in a query, the query can be automatically expanded with synonyms and related words of *t* from the thesaurus. Use of a thesaurus can be combined with ideas of term weighting: for instance, one might weight added terms less than original query terms.

Query expansion continued

- Use of a controlled vocabulary is quite common for well resourced domains. A well-known example is the Unified Medical Language System (UMLS) used with MedLine for querying the biomedical research literature.
- User query: cancer
- • PubMed query: (“neoplasms”[TIAB] NOT Medline[SB]) OR “neoplasms”[MeSH
- Terms] OR cancer[Text Word]
- • User query: skin itch
- • PubMed query: (“skin”[MeSH Terms] OR (“integumentary system”[TIAB] NOT
- Medline[SB]) OR “integumentary system”[MeSH Terms] OR skin[Text Word]) AND
- ((“pruritus”[TIAB] NOT Medline[SB]) OR “pruritus”[MeSH Terms] OR itch[Text
- Word])

Query expansion continued

- A manual thesaurus. Here, human editors have built up sets of synonymous names for concepts, without designating a canonical term.
- An automatically derived thesaurus. Here, word co-occurrence statistics over a collection of documents in a domain are used to automatically induce a thesaurus

- An automatically derived thesaurus. Here, word co-occurrence statistics over a collection of documents in a domain are used to automatically induce a thesaurus

Automatic thesaurus generation

- One is simply to exploit word co-occurrence We say that words co-occurring in a document or paragraph are likely to be in some sense similar or related in meaning, and simply count text statistics to find the most similar words.

- The other approach is to use a shallow grammatical analysis of the text and to exploit grammatical relations or grammatical dependencies. For example, we say that entities that are grown, cooked, eaten, and digested, are more likely to be food items.

- Simply using word cooccurrence is more robust (it cannot be misled by parser errors), but using grammatical relations is more accurate.

Points to ponder!

- In Rocchio’s algorithm, what weight setting for α/β/γ does a “Find pages like this one” search correspond to?

- Give three reasons why relevance feedback has been little used in web search.

- Under what conditions would the modified query qm in Equation 9.3 be the same as the original query q0? In all other cases, is qm closer than q0 to the centroid of the relevant documents?

- Why is positive feedback likely to be more useful than negative feedback to an IR system? Why might only using one non relevant document be more effective than using several?

Exercise

- Suppose that a user’s initial query is cheap CDs cheap DVDs extremely cheap CDs. The user examines two documents, d1 and d2. She judges d1, with the content CDs cheap software cheap CDs relevant and d2 with content cheap thrills DVDs non relevant. Assume that we are using direct term frequency (with no scaling and no document frequency). There is no need to length-normalize vectors. Using Rocchio relevance feedback as in Equation (9.3) what would the revised query vector be after relevance feedback? Assume $\alpha = 1, \beta = 0.75, \gamma = 0.25$.

Exercise

- Omar has implemented a relevance feedback web search system, where he is going to do relevance feedback based only on words in the title text returned for a page(for efficiency). The user is going to rank 3 results. The first user, Jinxing, queries for:
- banana slug
- and the top three titles returned are:
- banana slug Ariolimax columbianus
- Santa Cruz mountains banana slug
- Santa Cruz Campus Mascot
- Jinxing judges the first two documents relevant, and the third non relevant. Assume that Omar’s search engine uses term frequency but no length normalization nor IDF. Assume that he is using the Rocchio relevance feedback mechanism, with α = β = γ = 1. Show the final revised query that would be run. (Please list the vector elements in alphabetical order.)

Probabilistic Retrieval

Mansi A. Radke

Probability Review

- P(A,B) = P(A ∩ B) = P(A| B)*P(B) = P(B|A)*P(A)
-
  - P(A,B) = P(A ∩ B) = P(A| B)*P(B) = P(B|A)*P(A)

- P(B|)* P()
  - P$(\bar{A}, B) = P($B| $\bar{A}$)* P($\bar{A}$)

- P(B) = P (A,B) + P(,B)
  - P(B) = P (A,B) + P($\bar{A}$,B)

Probability Ranking Principle (PRP)

If an IR system’s response to each query is a ranking of the documents in the collection in order of decreasing probability of relevance, then the overall effectiveness of the system to the user will be maximised

How to model?

- D -> documents
  - Sample space: collection of documents indexed by search engine
- Q –> query
  - Sample space: any query consisting of one or more terms
- R -> user’s judgement of relevance (R is a binary random variable)
  - Sample space: 0 or 1

What is the probability that a user will judge a particular document relevant to a given query?

- Suppose we want to model this question, we need
- P(R = 1 | D = d, Q = q) --------------Equation 1

- We can write this as:
- P(r | D, Q) = 1 – P (|D, Q)

Log Odds / Logit

- Given a probability p, the logit (p) is defined as:

- $\mathrm{Logit}(p) = \log(p/(1-p))$

- Where the base of the logarithm may be chosen arbitrarily

Properties of log Odds or Logit

- As p varies from 0 to 1, logit(p) varies from -∞ to ∞

- If the odds are even i.e. p = 0.5, then logit(p) = 0

- Given two probabilities p and q, logit(p) > logit (q) only if p>q

- Thus log odds or logit and probability are rank equivalent

Modelling relevance - Assumptions

The binary independence assumption is that documents are binary vectors. That is only the presence or absence of terms in the documents are recorded

The terms are independently distributed in the set of relevant documents and they are also independently distributed in the set of irrelevant documents. The representation is an ordered set of Boolean vectors

A document is represented by a vector d = (x1, x2, …….., xm) where xt = 1 if term t is present in the document and xt = 0 if it is not.

Queries represented in similar way.

Notion of independence

- Independence signifies that the terms in the document are considered independently from each other and no association between the terms is modelled.

Go back to Equation 1

- P(R = 1 | D = d, Q = q) --------------Equation 1

- We can write this as:
- P(r | D, Q)

- Also we have
- P(r | D, Q) = 1 – P (|D, Q)

Applying Baye’s theorem

Bayes theorem is: $P(A|B) = P(B|A) * P(A) / P(B)$

$P(r| D, Q) = P(D, Q|r)* P(r)/ P(D,Q)$

And

$P(| D, Q) = P(D, Q|) * P()/ P(D,Q)$

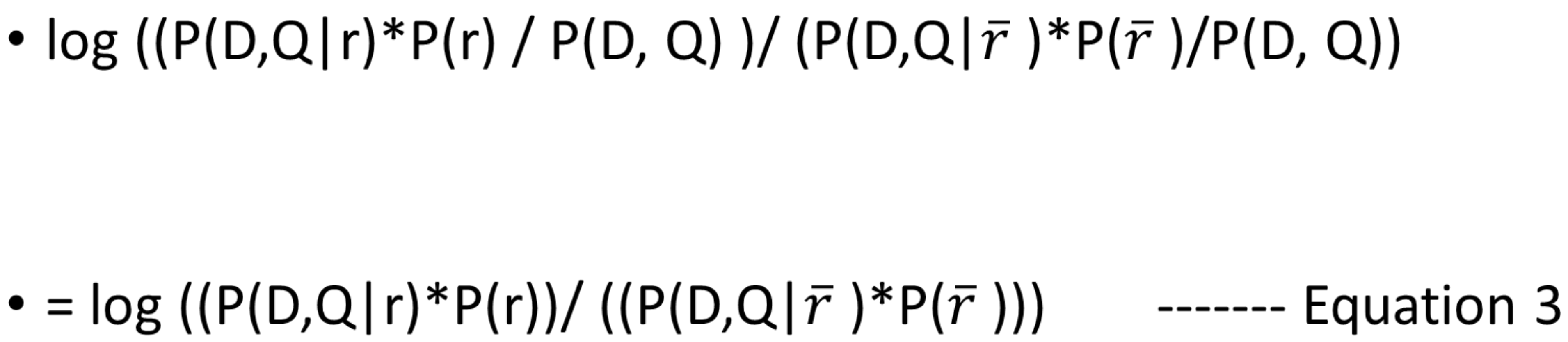
Taking log odds of equation 1

- $\log(P(r|D,Q)/(1-P(r|D,Q))$

- = $\log(P(r|D,Q)/(\mathrm{P}(\overline{r}$ | *D, Q))* ----------*Equation 2*

Applying Baye’s theorem equation 2 becomes

-



- log ((P(D,Q|r)*P(r) / P(D, Q))/ (P(D,Q|$\bar{r}$)*P($\bar{r}$)/P(D, Q))

- = log ((P(D,Q|r)*P(r))/ ((P(D,Q|$\bar{r}$)*P($\bar{r}$))) ------- Equation 3

We may expand the joint probabilities in equation 3 into conditional probabilities by using the equality

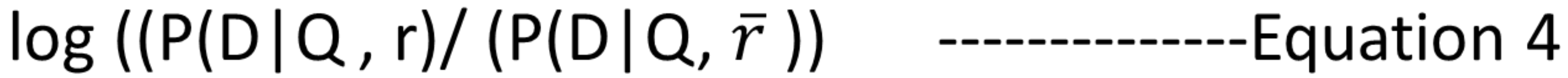
- P(D,Q|R) = P(D|Q,R). P(Q|R)

- log ((P(D,Q|r)*P(r))/ ((P(D,Q|)*P( )))
- = log ((P(D|Q , r)*P(Q|r)*P(r))/ ((P(D|Q,) P(Q|)*P( )))
- = log ((P(D|Q , r)*P(r|Q)*P(Q))/ ((P(D|Q,) P(|Q)*P(Q)))
- = log ((P(D|Q , r)*P(r|Q))/ ((P(D|Q,) P(|Q)))
- =log ((P(D|Q , r)/ (P(D|Q,)) + log (P(r|Q)/ P(|Q))
- = log ((P(D|Q , r)/ (P(D|Q,))

This term is independent of D. Hence can be safely ignored for ranking purposes

So we get to the ranking formula

-



log ((P(D|Q , r)/ (P(D|Q, $\bar{r}$))   --------------Equation 4

log ((P(D|Q , r)/ (P(D|Q,))   --------------Equation 4

The Binary Independence Model – 1st assumption

- Given relevance terms are statistically independent
- Given a positive relevance judgement, the presence or absence of one term does not depend on the presence or absence of any other terms
- Given a negative relevance judgement, the presence or absence of one term does not depend on the presence or absence of any other terms

- With this assumption, we rewrite the probabilities of equation 4 as

$$\begin{aligned}
p(D \,|\, Q, r) &= \prod_{i=1}^{|\mathcal{V}|} p(D_i \,|\, Q, r) \quad \text{and} \\
p(D \,|\, Q, \bar{r}) &= \prod_{i=1}^{|\mathcal{V}|} p(D_i \,|\, Q, \bar{r}).
\end{aligned}$$

So equation 4 becomes :

$$\log \frac{p(D \,|\, Q, r)}{p(D \,|\, Q, \bar{r})} = \sum_{i=1}^{|\mathcal{V}|} \log \frac{p(D_i \,|\, Q, r)}{p(D_i \,|\, Q, \bar{r})} \,.$$

Equation 5

BIM-We make the second strong assumption

- The presence of a term in a document depends on relevance only when the term is present in the query

- As a result of this 2nd assumption, if qi = 0, then

$$p(D_i \,|\, Q, r) \;=\; p(D_i \,|\, Q, \bar{r})$$

$$\log \frac{p(D_i \,|\, Q, r)}{p(D_i \,|\, Q, \bar{r})} \;=\; 0.$$

- The effect of this assumption is to change the summation in Equation 5 from a summation over all terms in the vocabulary to a summation over all terms in the query. Because conditioning on the query is now redundant, we can drop it and our ranking formula then becomes

$$\sum_{t \in q} \log \frac{p(D_t \,|\, r)}{p(D_t \,|\, \bar{r})} \,,$$

------ Equation 6

- We now subtract from Equation 6 its own value when no query terms appear in the document

$$\sum_{t \in q} \log \frac{p(D_t = d_t \,|\, r)}{p(D_t = d_t \,|\, \bar{r})} \; - \; \sum_{t \in q} \log \frac{p(D_t = 0 \,|\, r)}{p(D_t = 0 \,|\, \bar{r})}$$

When all query terms are absent, this term is a constant

We can subtract this constant as it has no impact on ranking

- Rearranging the terms

$$\sum_{t\in(q\cap d)} \log \frac{p(D_t=1\,|\,r)\ p(D_t=0\,|\,\bar{r})}{p(D_t=1\,|\,\bar{r})\ p(D_t=0\,|\,r)} - \sum_{t\in(q\setminus d)} \log \frac{p(D_t=0\,|\,r)\ p(D_t=0\,|\,\bar{r})}{p(D_t=0\,|\,\bar{r})\ p(D_t=0\,|\,r)}\,,$$

This term is 0

So our formula becomes !

$$\sum_{t \in (q \cap d)} \log \frac{p(D_t = 1 \,|\, r)\; p(D_t = 0 \,|\, \bar{r})}{p(D_t = 1 \,|\, \bar{r})\; p(D_t = 0 \,|\, r)} \,.$$

So, this refinement of the equation 4 under the assumptions 1 and 2 and considering only the presence an absence of terms is called as the Binary Independence model

Thank You !