Information Retrieval

Unit I

Definition

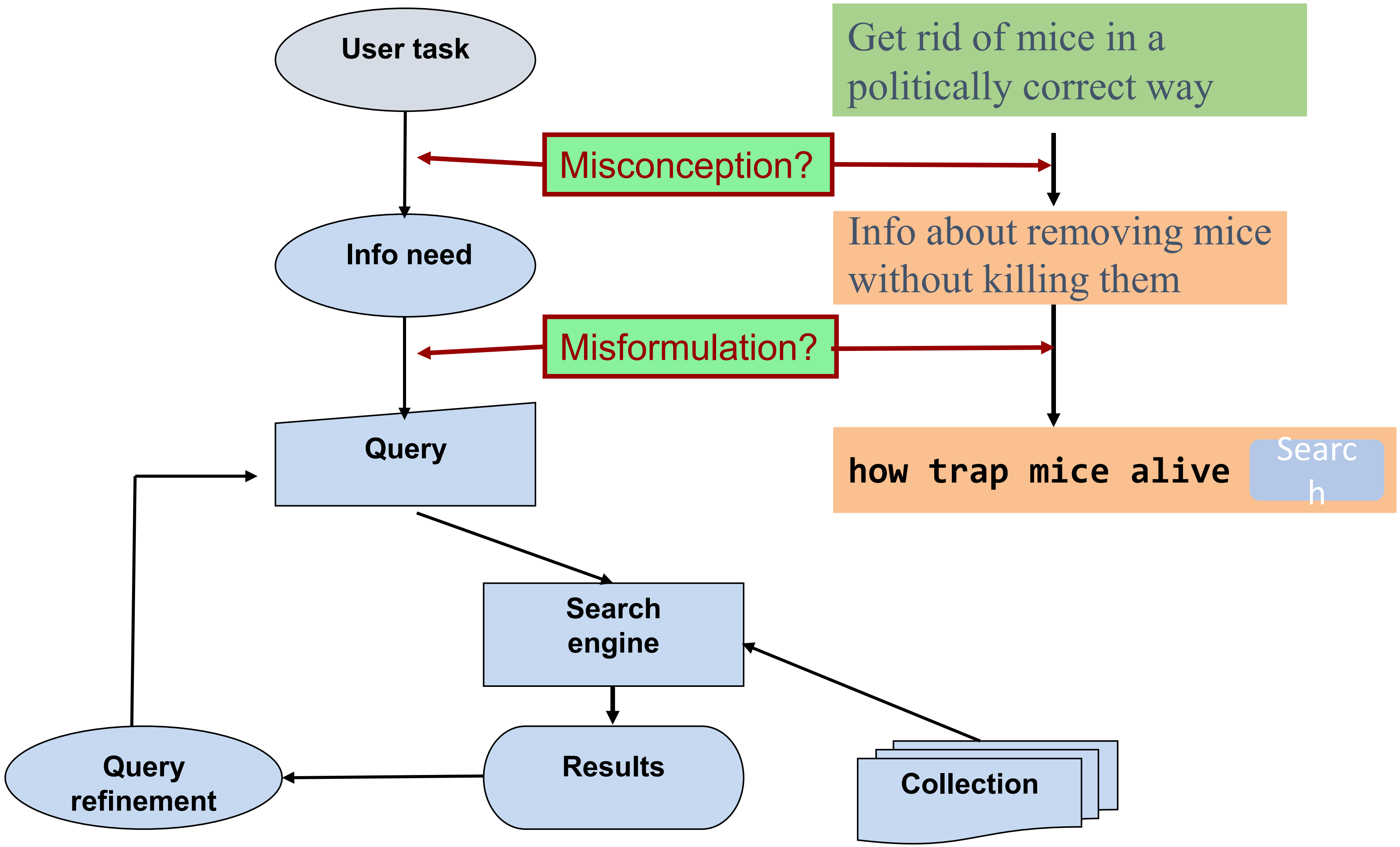
- Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).

Basic assumptions of Information Retrieval

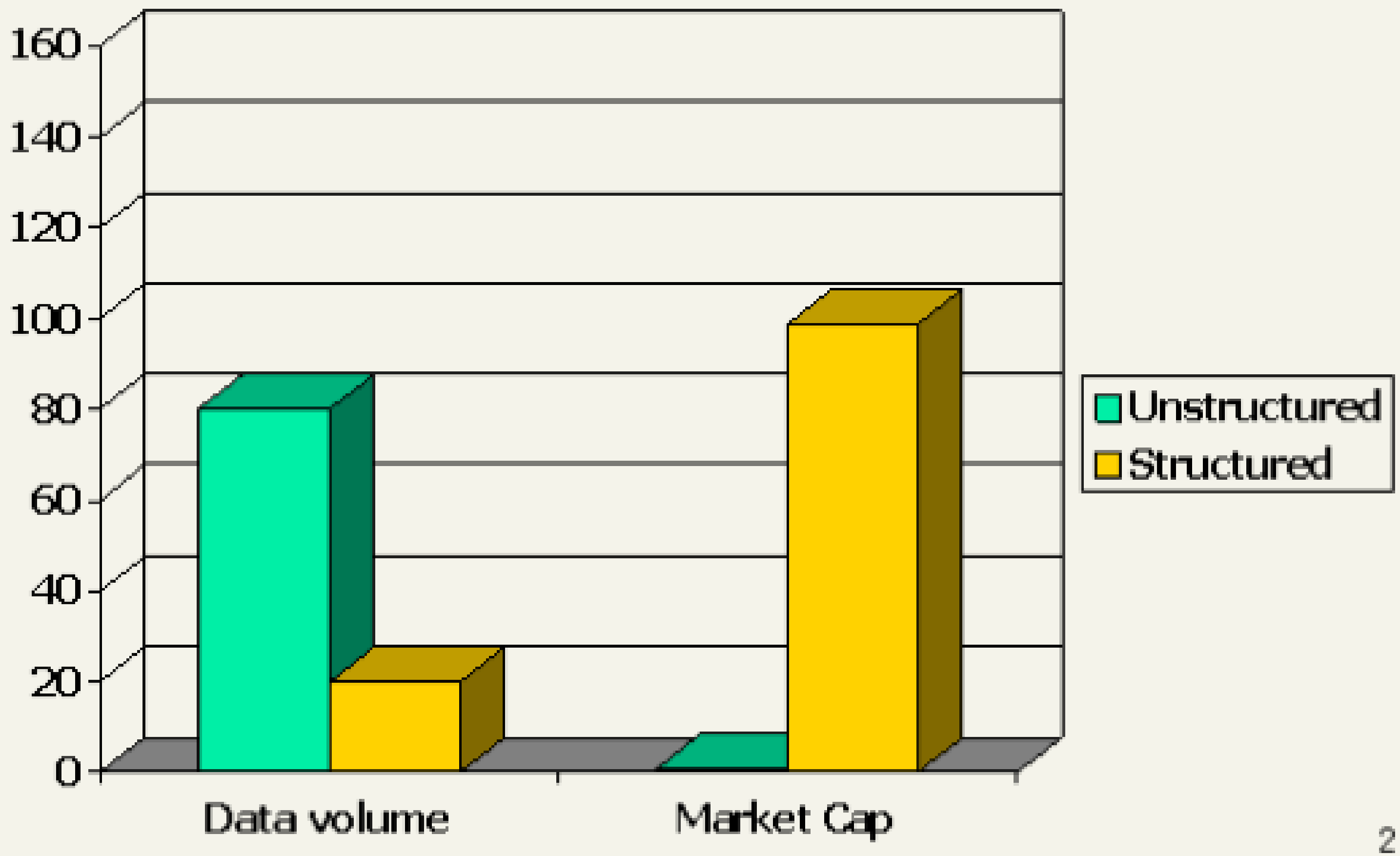
- Collection: A set of documents
  - Assume it is a static collection for the moment

- Goal: Retrieve documents with information that is relevant to the user's information need and helps the user complete a task

The classic search model



Unstructured (text) vs. structured (database) data in 1996



2

Unstructured (text) vs. structured (database) data in 2006

6

IR today

- Web search (Google bing)
  - Search ground are billions of documents on millions of computers
  - issues: spidering; efficient indexing and search; malicious manipulation to boost search engine rankings
  - Link analysis covered in Lecture 8
- Enterprise and institutional search (PubMed LexisNexis)
  - e.g company's documentation, patents, research articles
  - often domain-specific
  - Centralised storage; dedicated machines for search.
  - Most prevalent IR evaluation scenario: US intelligence analyst's searches
- Personal information retrieval (email, pers. documents;)
  - e.g., Mac OS X Spotlight; Windows' Instant Search
  - Issues: different file types; maintenance-free, lightweight to run in background

Basic IR System Architecture

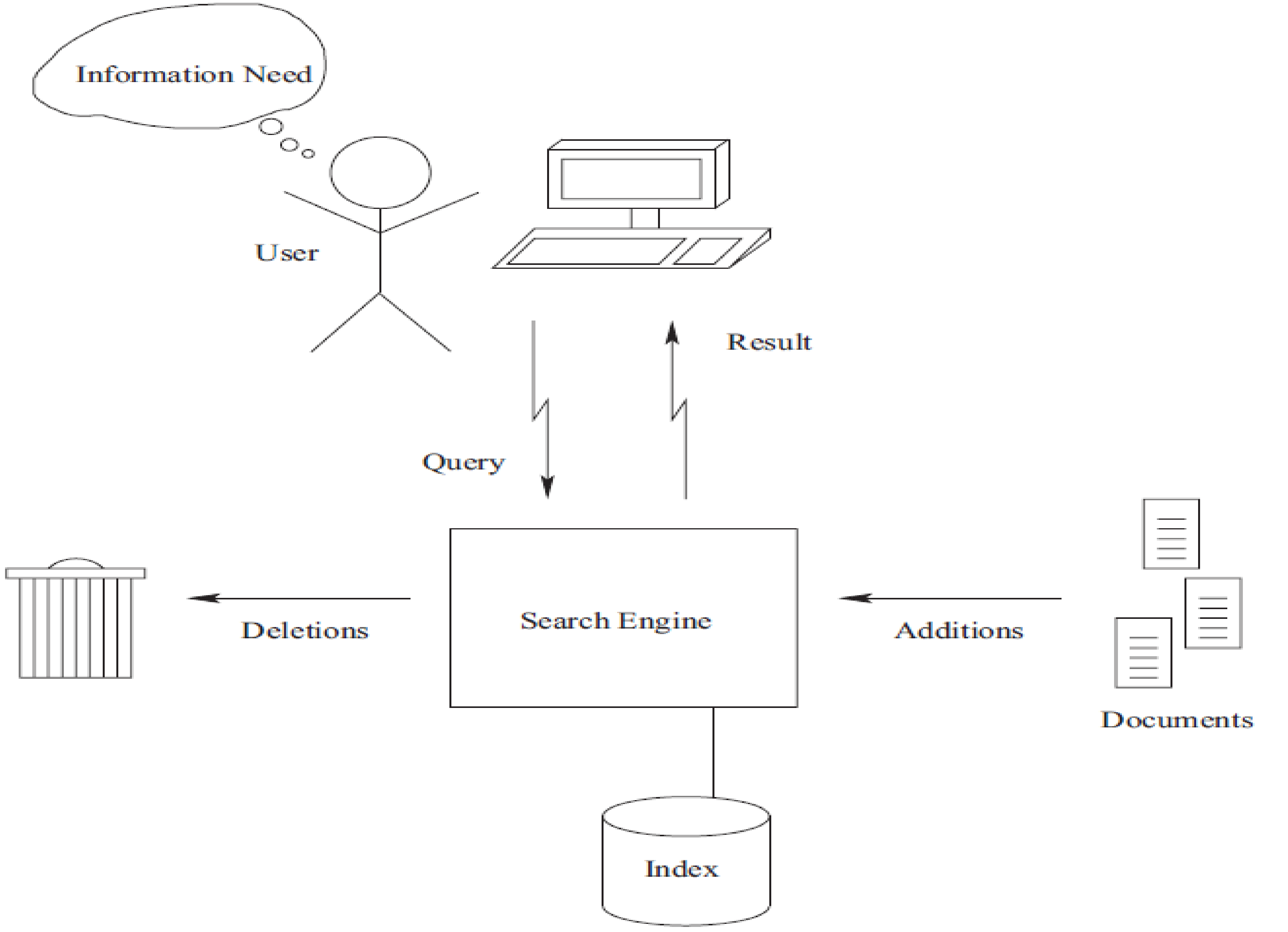


Figure 1.1 Components of an IR system.

How good are the retrieved docs?

- *Precision* : Fraction of retrieved docs that are relevant to the user's information need
- *Recall* : Fraction of relevant docs in collection that are retrieved

  - More precise definitions and measurements to follow later

Relevance

- Relevance is the core concept in IR, but nobody has a good definition
  - Relevance = useful
  - Relevance = topically related
  - Relevance = new
  - Relevance = interesting
  - Relevance = ???
- Relevance is very dynamic – it depends on the needs of a person at a specific point in time
- *The same result for the same query may be relevant for a user and not relevant for another*

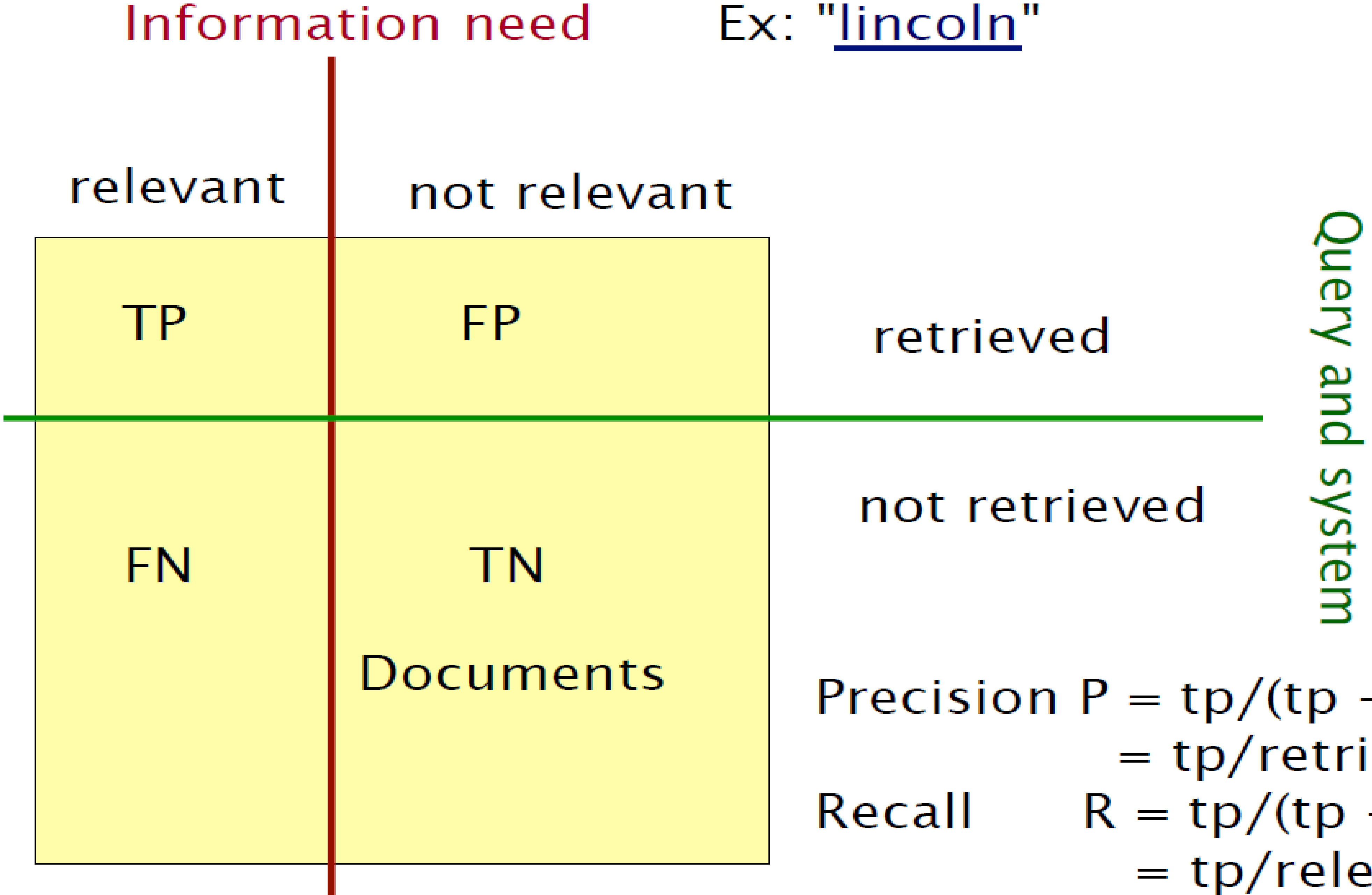
Boolean Retrieval and Relevance

- **Assumption:** A document is relevant to the information need expressed by a query if it satisfies the Boolean expression of the query.

- Question: *Is it always true?*

- No: consider for instance a collection of documents dated before 2014, and the query is "oscar AND 2014". Would the documents retrieved by this query relevant?

Relevance and Retrieved documents



Precision $P = tp/(tp + fp)$
$= tp/retrieved$

Recall $R = tp/(tp + fn)$
$= tp/relevant$

Term-document incidence matrices

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

1 if play contains word, 0 otherwise

Brutus *AND* ***Caesar*** *BUT NOT* ***Calpurnia***

AD HOC RETRIEVAL

- System aims to provide documents from within the collection that are relevant to an arbitrary user information need, communicated to the system by means of a one-off, user-initiated query.

Incidence vectors

- So we have a 0/1 vector for each term.
- To answer query: take the vectors for ***Brutus, Caesar*** and ***Calpurnia*** (complemented) ➔ bitwise *AND*.
  - 110100 *AND*
  - 110111 *AND*
  - 101111 =
  - **100100**

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

Answers to query

- Antony and Cleopatra, Act III, Scene ii

Agrippa [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus,
When Antony found Julius ***Caesar*** dead,
He cried almost to roaring; and he wept
When at Philippi he found ***Brutus*** slain.

- Hamlet, Act III, Scene ii

Lord Polonius: I did enact Julius ***Caesar*** I was killed i' the
Capitol; ***Brutus*** killed me.

Bigger collections

- Consider $N = 10^6$ documents, each with about 1000 tokens
- ⇒ total of 10^9 tokens
- On average 6 bytes per token, including spaces and
- punctuation ⇒ size of document collection is about 6
  - $10^9 = 6$ GB
- Assume there are $M = 500{,}000$ distinct terms in the collection
- (Notice that we are making a term/token distinction.)

Can’t build the incidence matrix

- M = 500,000 × 10^6 = half a trillion 0s and 1s.
- But the matrix has no more than one billion 1s.
  - Matrix is extremely sparse.
- What is a better representations?
  - We only record the 1s.

18

Inverted Index

For each term *t*, we store a list of all documents that contain *t*.

Identify each doc by a **docID**, a document serial number

Can we use fixed size array for this?

Brutus	1	2	4	11	31	45	173	174
Caesar	1	2	4	5	6	16	57	132
Calpurnia	2	31	54	101				

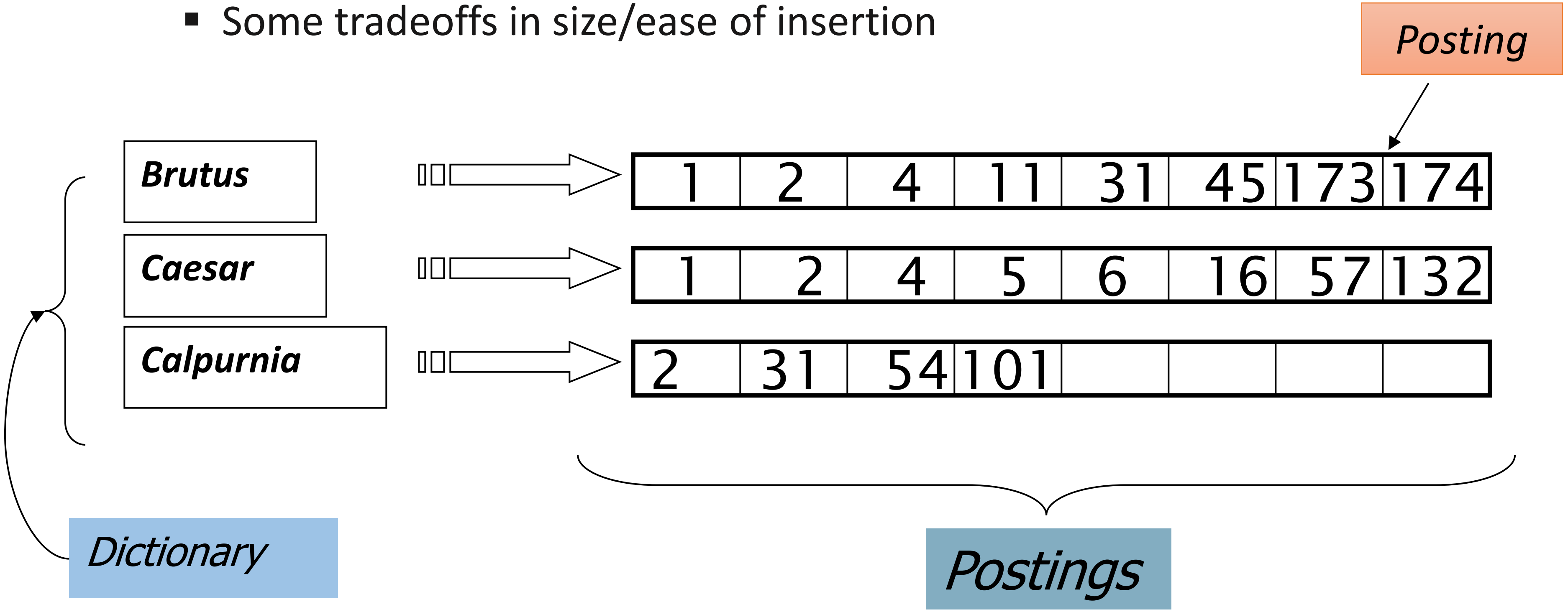
What happens if the word ***Caesar*** is added to document 14?

19

Inverted Index

- We need variable-size postings lists
  - On disk, a continuous run of postings is normal and best
  - In memory, can use linked lists or variable length arrays
    - Some tradeoffs in size/ease of insertion

Each item in the list – which records that a term appeared in a document– is conventionally called a *posting*.



Sorted by docID (more later on why).

How to build an inverted index

1. Collect the documents to index

2. Tokenize the content: from string to tokens

"I am not going there to be imprisoned," said Dantes. ----->

“I | am | not | going | there

to | be | Imprisoned,” | said | Dantes.

3. Normalize the tokens (preprocessing)

“I | am | not | going | there

to | be | Imprisoned,” | said | Dantes.

----->

i | am | not | **go** | there

to | be | **imprison** | said | **dantes**

-----> *(imprison,L3)*

(term,docid) pair

4. Index the documents

Tokenization & normalization I

- Tokenization is not always straight-forward
  - E-mail: *email* or *{e,mail}?*
  - It's: *its* or *{it,s}?*
  - What about *O-β-D-galactopyranosyl-(1→4)-D-glucopyranose*?
  - What about documents containing many floats 2.4325453423432323432 4325.... ?
  - "The sun is shining." in simplified Chinese: 阳光普照。
- Case folding [2]
  - *{the,The,THE,tHE}* are all matched to *the*
  - *General AND Motors* should not retrieve *"general repairs to all kinds of motors" (exception can be handled by a postretrieval scan)*

Tokenization & normalization II

- Stopword removal
  - Term frequencies: *The Count of Monte Cristo*
  - *Stopwords occur with very high frequencies often not adding any value*
  - *What about the query "to be or not to be"?*
  - Standard stopword list vs. corpus-dependent (domain-dependent) lists

	Term	#tf
1.	the	28388
2.	to	12841
3.	of	12834
4.	and	12447
5.	a	9328
6.	i	8174
7.	you	8128

- Stemming
  - Reduce terms to their root form (strip suffixes), e.g.

    {compressed,compression} ➔ *compress*

    {walking,walked,walks} ➔ *walk*

Tokenization & normalization III

- Stemming cont.
  - It is not appropriate for all types of documents or parts of documents
    - Author names in scientific papers or book catalogues, etc.
  - Two standard stemmers (for English): Krovetz (1993) and Porter stemmer (1979) [3,4]

Clear sky, swift-flitting boats, and brilliant sunshine disappeared; the heavens were hung with black, and the gigantic structure of the Chateau d'If seemed like the phantom of a mortal enemy.

Clear sky swift flit boat and brilliant sunshin disappear the heaven were hung with black and the gigant structur of the Chateau d If seem like the phantom of a mortal enemi

Porter stemmed

Indexer steps: Token sequence

- Sequence of (Modified token, Document ID) pairs.

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

Indexer steps: Sort

- Sort by terms
  - And then docID

Core indexing step

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

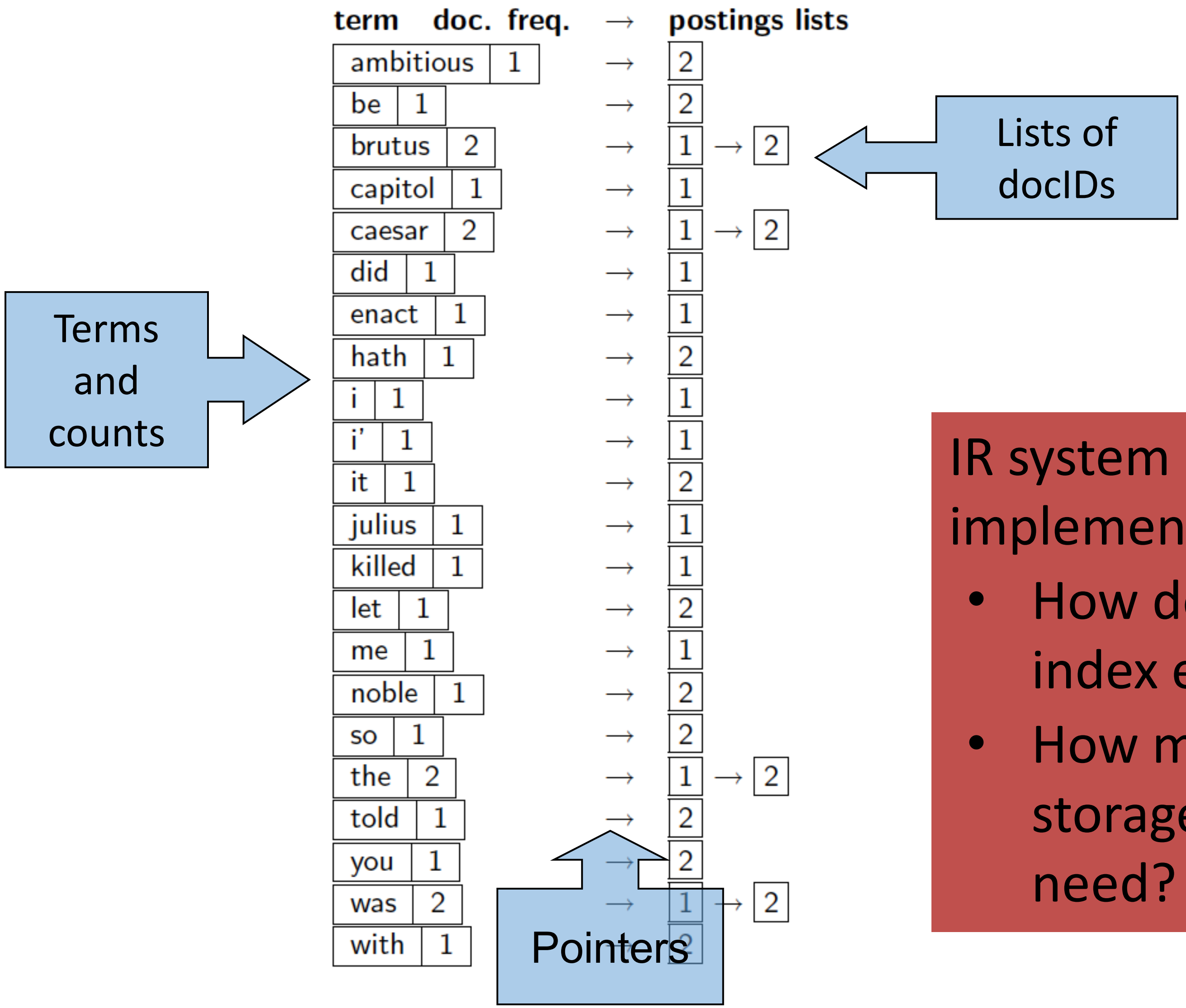
Indexer steps: Dictionary & Postings

- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Doc. frequency information is added.

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

Where do we pay in storage?



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

IR system implementation

- How do we index efficiently?
- How much storage do we need?

Exercise

Doc 1: “I am not going there to be imprisoned,” said Dantes

Doc 2 : “ You are Edmonds Dantes,” cried villefort seizing the count by the wrist; “then come here!”

Exercise

Exercise 1.1 [⋆]

Draw the inverted index that would be built for the following document collection. (See Figure 1.3 for an example.)

Doc 1	new home sales top forecasts
Doc 2	home sales rise in july
Doc 3	increase in home sales in july
Doc 4	july new home sales rise

Exercise 1.2 [⋆]

Consider these documents:

Doc 1	breakthrough drug for schizophrenia
Doc 2	new schizophrenia drug
Doc 3	new approach for treatment of schizophrenia
Doc 4	new hopes for schizophrenia patients

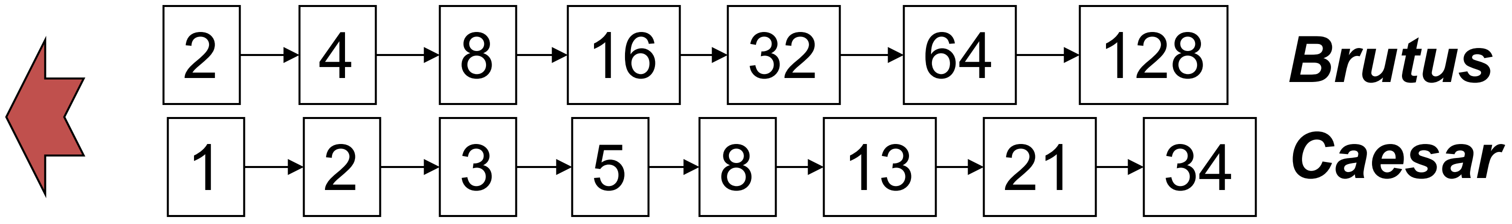
a. Draw the term-document incidence matrix for this document collection.

Query processing: AND

- Consider processing the query:

  Brutus *AND* ***Caesar***

  - Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.
  - Locate ***Caesar*** in the Dictionary;
    - Retrieve its postings.
  - "Merge" the two postings (intersect the document sets):



The merge

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

2 → 4 → 8 → 16 → 32 → 64 → 128 **_Brutus_**

1 → 2 → 3 → 5 → 8 → 13 → 21 → 34 **_Caesar_**

If the list lengths are x and y, the merge takes $O(x+y)$ operations.

Crucial: postings sorted by docID.

Intersecting two posting lists

```
INTERSECT(p1, p2)
  answer ← ⟨ ⟩
  while p1 ≠ NIL and p2 ≠ NIL
  do if docID(p1) = docID(p2)
        then ADD(answer, docID(p1))
             p1 ← next(p1)
             p2 ← next(p2)
        else if docID(p1) < docID(p2)
                then p1 ← next(p1)
                else p2 ← next(p2)
  return answer
```

Posting lists

Common docid found in both lists

Increase the posting list counters

33

term1 OR term2

Query: *Brutus OR Caesar*

- Retrieve the postings list for *Brutus*.
- Retrieve the postings list for *Caesar*.
- Compute the union of the postings lists.

Note: The intersection should be ordered again. consider:
(Brutus OR Caesar) AND Calpuria

NOT term

Query: *NOT Caesar*

- Retrieve the postings list for *Caesar*.
- Construct a list with all documents.
- Remove the postings for *Caesar* from the list of all documents.

- Inefficient!
- Not frequently used in isolation.

NOT term

Query: *Brutus AND NOT Caesar*

Evaluation:

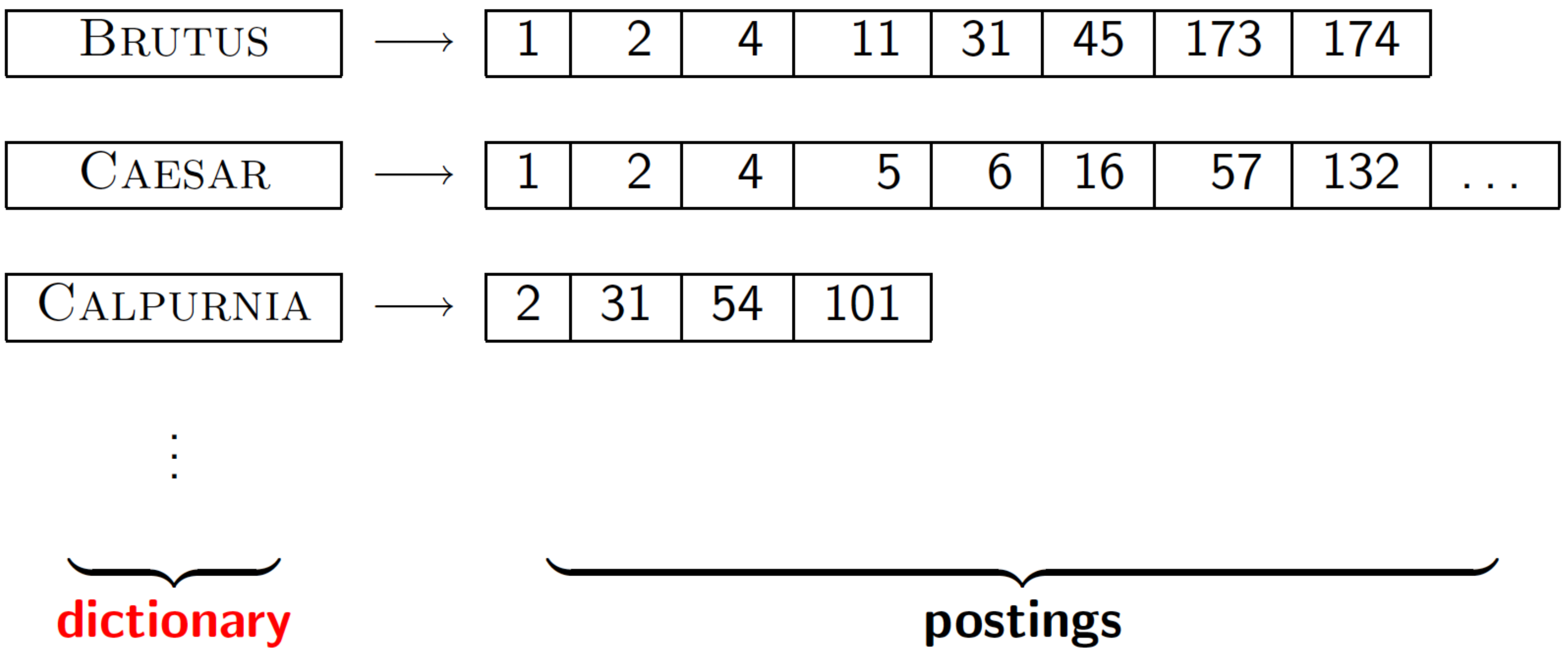
- Retrieve the postings list for *Brutus*.
- Retrieve the postings list for *Caesar*.
- Compute the difference between the postings lists for *Brutus* and *Caesar*

This lecture

- Dictionary data structures
- “Tolerant” retrieval
  - Wild-card queries
  - Spelling correction
  - Soundex

Dictionary data structures for inverted indexes

- The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list … in what data structure?



A naïve dictionary

- An array of struct:

term	document frequency	pointer to postings list
a	656,265	$\longrightarrow$
aachen	65	$\longrightarrow$
...	...	...
zulu	221	$\longrightarrow$

char[20]	int	Postings *
20 bytes	4/8 bytes	4/8 bytes

- How do we store a dictionary in memory efficiently?
- How do we quickly look up elements at query time?

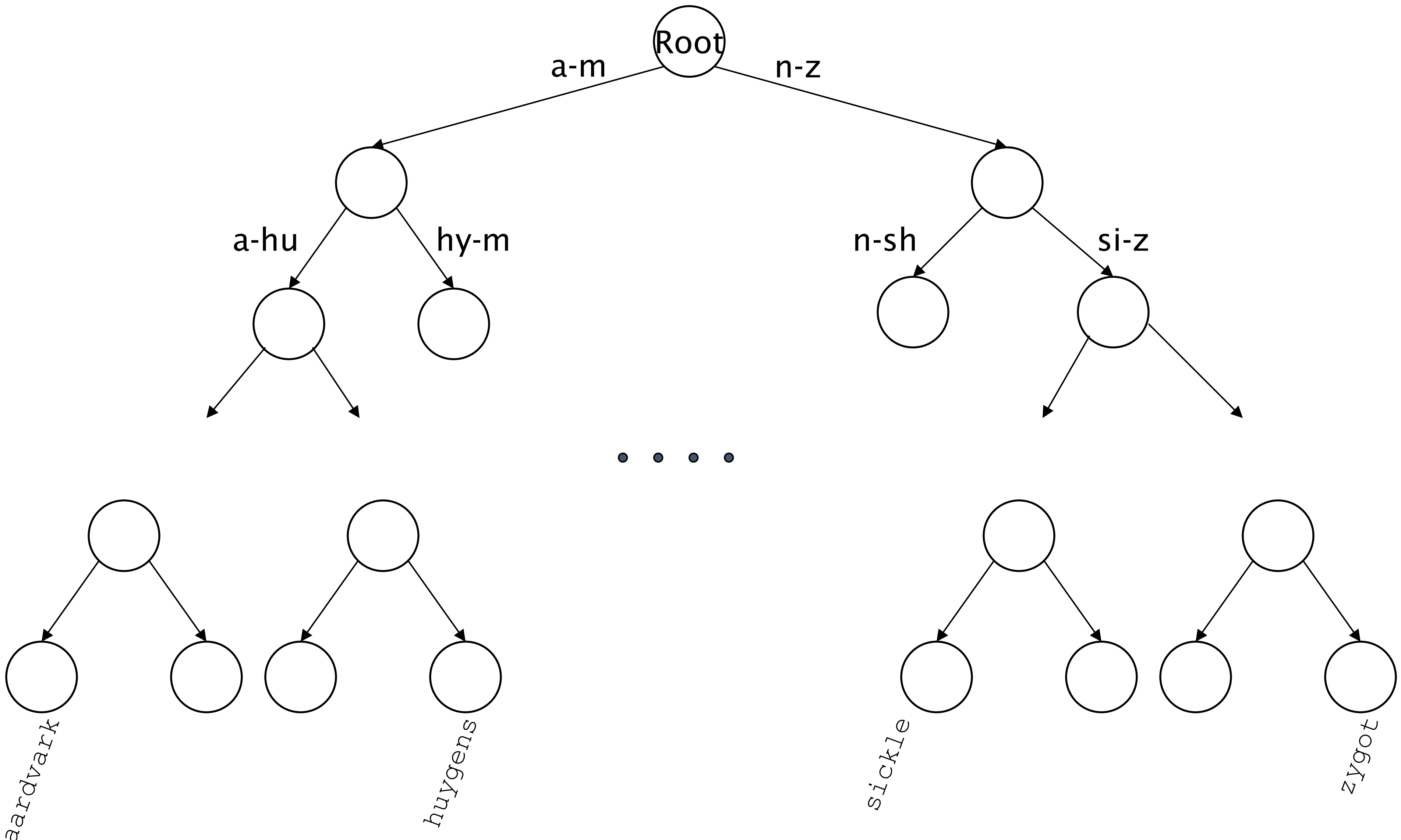
Dictionary data structures

- Two main choices:
  - Hashtables
  - Trees
- Some IR systems use hashtables, some trees

Hashtables

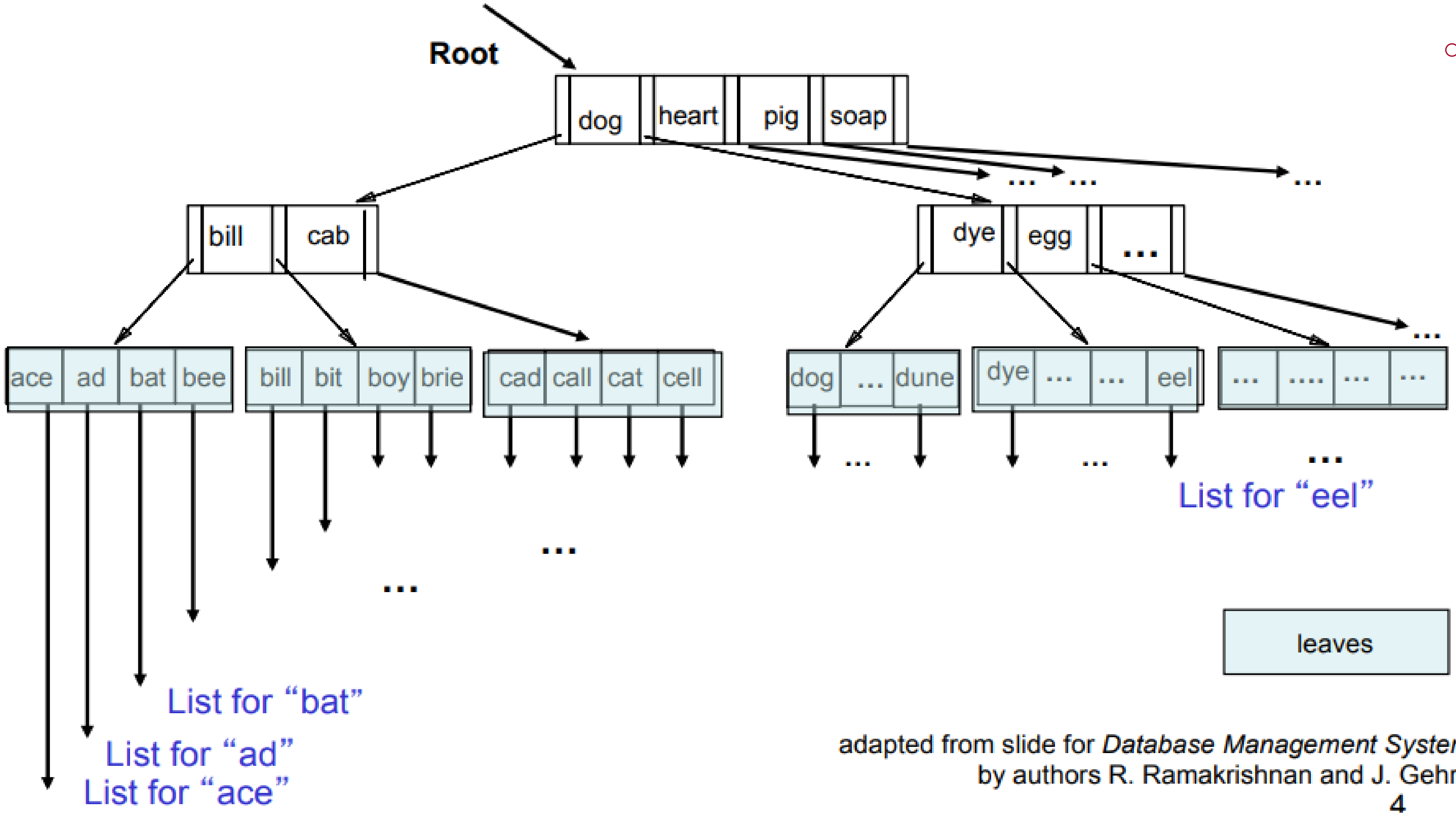
- Each vocabulary term is hashed to an integer
  - (We assume you've seen hashtables before)
- Pros:
  - Lookup is faster than for a tree: O(1)
- Cons:
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search [tolerant retrieval]
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing *everything*

Tree: binary tree



Example B+ Tree

order = 2: 2 to 4 search keys per interior node



- Definition: Every internal nodel has a number of children in the interval [*a*,*b*] where *a*, *b* are appropriate natural numbers, e.g., [2,4].

adapted from slide for *Database Management Systems* by authors R. Ramakrishnan and J. Gehrke

Trees

- Simplest: binary tree
- More usual: B-trees
- Trees require a standard ordering of characters and hence strings … but we typically have one
- Pros:
  - Solves the prefix problem (terms starting with *hyp*)
- Cons:
  - Slower: O(log *M*) [and this requires *balanced* tree]
  - Rebalancing binary trees is expensive
    - But B-trees mitigate the rebalancing problem

Wild-card queries

Wildcard queries I

- Commonly employed when
  - There is uncertainty about the spelling of a term (Dantes vs. Dantès)
  - Multiple spelling variants of a term exist (labour vs. labor)
  - All terms with the same stem are sought (restoration and restore)

- Trailing wildcard query: *restor**
  - Search trees are perfect in such situations: walk along the edges and enumerate the W terms with prefix *restor*, followed by $|W|$ lookups of the respective posting lists to retrieve all docIDs

Wildcard queries II

- Leading wildcard query: **building* (building vs. rebuilding)
  - **Reverse** dictionary B-tree: constructed by reading each term in the vocabulary **backwards**
  - Solved analogously to the trailing wildcard query on a b-tree
    - reverse b-tree is traversed with *building backwards: g-n-i-d-l-i-u-b

- Single wildcard query: analy*ed (analysed vs. analyzed)
  - Traverse the regular b-tree to find the W terms with prefix *analy*
  - Traverse the reverse b-tree to find the R terms with suffix *ed*
  - Final result:intersect W and R

Wild-card queries: *

- ***mon*:*** find all docs containing any word beginning with “mon”.
- Easy with binary tree (or B-tree) lexicon: retrieve all words in range: ***mon ≤ w < moo***
- ****mon:*** find words ending in “mon”: harder
  - Maintain an additional B-tree for terms *backwards.*

  Can retrieve all words in range: ***nom ≤ w < non***.

Exercise: from this, how can we enumerate all terms meeting the wild-card query ***pro*cent*** ?

Query processing

- At this point, we have an enumeration of all terms in the dictionary that match the wild-card query.
- We still have to look up the postings for each enumerated term.
- E.g., consider the query:

  se*ate *AND* ***fil*er***

  This may result in the execution of many Boolean *AND* queries.

B-trees handle *'s at the end of a query term

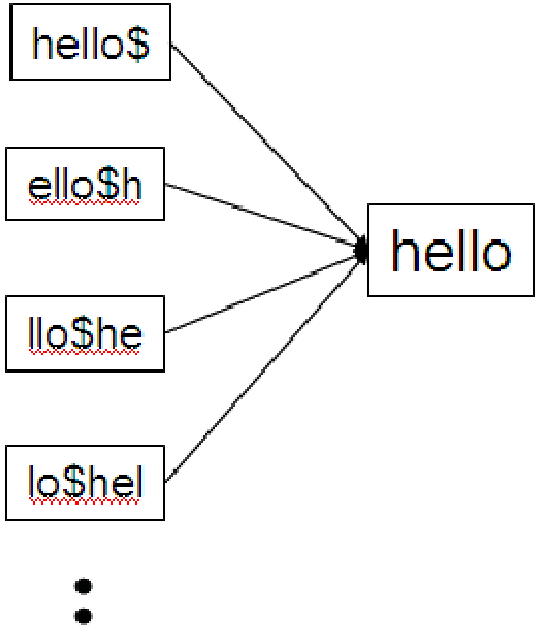
- How can we handle *'s in the middle of query term?
  - ***co*tion***
- We could look up ***co**** AND ***tion*** in a B-tree and intersect the two term sets
  - Expensive
- The solution: transform wild-card queries so that the *'s occur at the end
- This gives rise to the **Permuterm** Index.

Permuterm index

- For term ***hello***, index under:
  - ***hello$, ello$h, llohe, lohel, o$hell, $hello***

  where $ is a special symbol.
- Queries:
  - **X** lookup on **X$** **X*** lookup on **$X***
  - ***X** lookup on **X$*** ***X*** lookup on **X***
  - **X*Y** lookup on **Y$X*** **X*Y*Z** ??? Exercise!

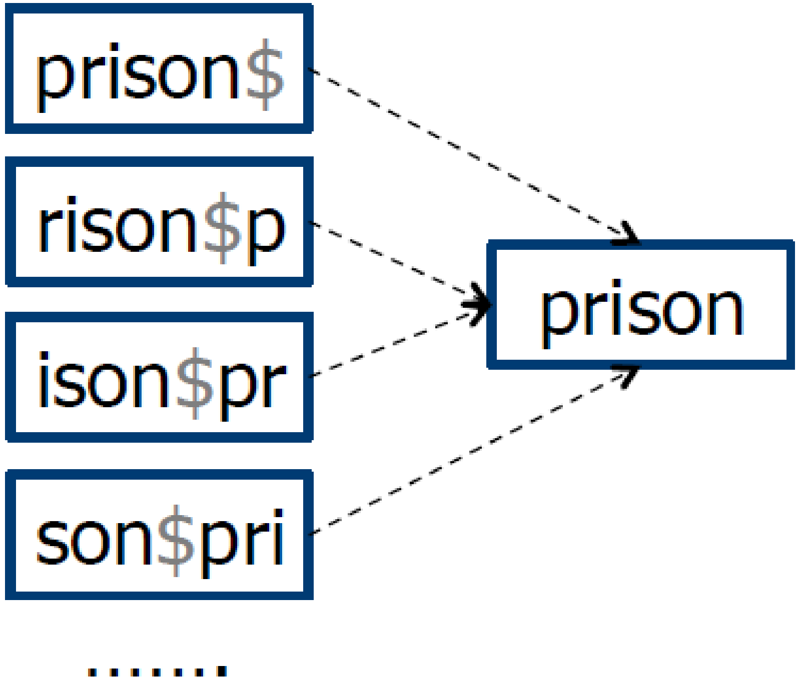
Query = ***hel*o***
X=*hel*, Y=*o*
Lookup ***o$hel****



General wildcard queries I

Permuterm index

- Query *pr*son*→*pr*son$*
  - Move * to the end: *son$pr**
  - Look up the term in the permuterm index (search tree)
  - Look up the found terms in the standard inverted index



- Query *pr*s*n*
  - Start with *n$pr**
  - Filter out all results not containing 's' in the middle (exhaustive)
  - Look up the found terms in the standard inverted index

Dictionary increases substantially in size!!

Permuterm query processing

- Rotate query wild-card to the right
- Now use B-tree lookup as before.
- *Permuterm problem: ≈ quadruples lexicon size*

Empirical observation for English.

Bigram (*k*-gram) indexes

- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term
- *e.g.,* from text "***April is the cruelest month***" we get the 2-grams (*bigrams*)

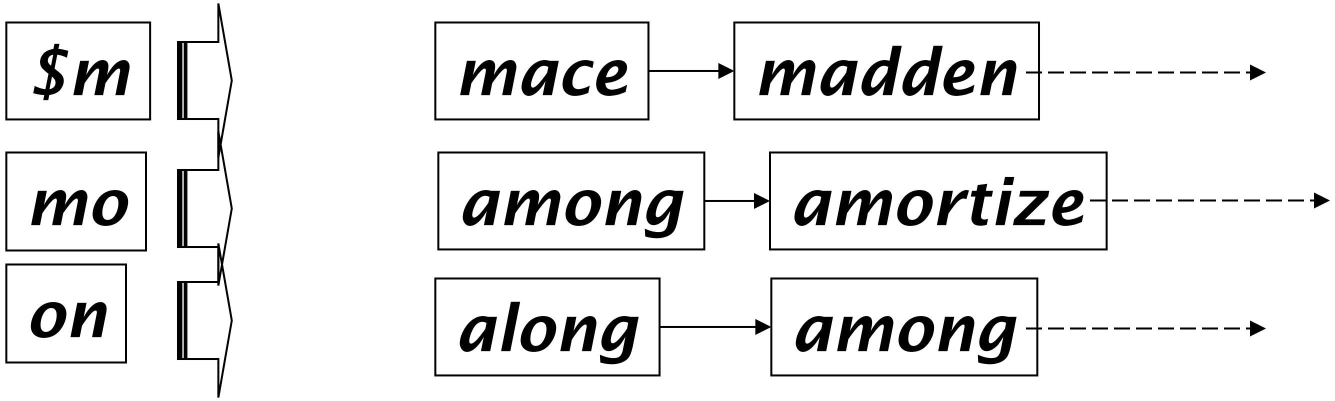
a,ap,pr,ri,il,l,i,is,s,t,th,he,e,$c,cr,ru,
ue,el,le,es,st,t$, m,mo,on,nt,h

  - $ is a special word boundary symbol

- Maintain a <u>*second*</u> inverted index <u>*from bigrams to*</u> <u>*dictionary terms*</u> that match each bigram.

Bigram index example

- The *k*-gram index finds *terms* based on a query consisting of *k*-grams (here *k*=2).



Processing wild-cards

- Query ***mon**** can now be run as
  - ***$m** AND **mo** AND **on***
- Gets terms that match AND version of our wildcard query.
- But we'd enumerate ***moon***.
- Must post-filter these terms against query.
- Surviving enumerated terms are then looked up in the term-document inverted index.
- Fast, space efficient (compared to permuterm).

General wildcard queries II

N-gram index

- each N-gram in the dictionary points to all terms containing the N-gram

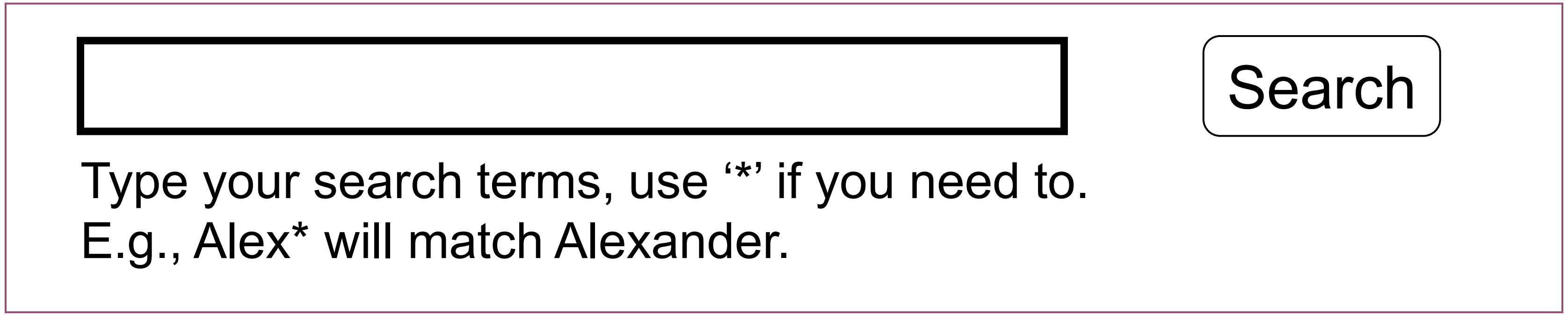
ris → grisly → prison → pristine → rise

lexicographical ordering

- Wildcard query: *pr*on*
  - Boolean query *$pr AND on$*
  - Look up in a 3-gram index yields a list of matching terms
  - Look up the matching terms in a standard inverted index
- Wildcard query: red*
  - Boolean query *$re AND red* (also retrieves *retired*)
  - Post-filtering step to ensure enumerated terms match red*

Processing wild-card queries

- As before, we must execute a Boolean query for each enumerated, filtered term.
- Wild-cards can result in expensive query execution (very large disjunctions…)
  - pyth* AND prog*
- If you encourage “laziness” people will respond!



Which web search engines allow wildcard queries?

Spelling correction

Spell correction

- Two principal uses
  - Correcting document(s) being indexed
  - Correcting user queries to retrieve “right” answers
- Two main flavors:
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words
    - e.g., ***from*** $\rightarrow$ ***form***
  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew <u>form</u> Heathrow to Narita.***

Document correction

- Especially needed for OCR'ed documents
  - Correction algorithms are tuned for this: rn/m
  - Can use domain-specific knowledge
    - E.g., OCR can confuse O and D more often than it would confuse O and I (adjacent on the QWERTY keyboard, so more likely interchanged in typing).
- But also: web pages and even printed material have typos
- Goal: the dictionary contains fewer misspellings
- But often we don't change the documents and instead fix the query-document mapping

Query mis-spellings

- Our principal focus here
  - E.g., the query ***Alanis Morisett***
- We can either
  - Retrieve documents indexed by the correct spelling, OR
  - Return several suggested alternative queries with the correct spelling
    - *Did you mean … ?*

Isolated word correction

- Fundamental premise – there is a lexicon from which the correct spellings come
- Two basic choices for this
  - A standard lexicon such as
    - Webster's English Dictionary
    - An "industry-specific" lexicon – hand-maintained
  - The lexicon of the indexed corpus
    - E.g., all words on the web
    - All names, acronyms etc.
    - (Including the mis-spellings)

Isolated word correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
- What’s “closest”?
- We’ll study several alternatives
  - Edit distance (Levenshtein distance)
  - Weighted edit distance
  - *n*-gram overlap

Edit distance

- Given two strings S_1 and S_2, the minimum number of operations to convert one to the other
- Operations are typically character-level
  - Insert, Delete, Replace, (Transposition)
- E.g., the edit distance from ***dof*** to ***dog*** is 1
  - From ***cat*** to ***act*** is 2 (Just 1 with transpose.)
  - from ***cat*** to ***dog*** is 3.
- Generally found by dynamic programming.
- See http://www.merriampark.com/ld.htm for a nice example plus an applet.

Defining Min Edit Distance (Levenshtein)

- Initialization

  D(i,0) = i

  D(0,j) = j

- Recurrence Relation:

  For each i = 1…M

  For each j = 1…N

$$
D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 1; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases} \end{cases}
$$

- Termination:

  D(N,M) is distance

-	0	E	X	E	C	U	T	I	O	N
0	0	1	2	3	4	5	6	7	8	9
I	1	1	2	3	4	5	6	6	7	8
N	2	2	2	3	4	5	6	7	7	7
T	3	3	3	3	4	5	5	6	7	8
E	4	3	4	3	4	5	6	6	7	8
N	5	4	4	4	4	5	6	7	7	7
T	6	5	5	5	5	5	5	6	7	8
I	7	6	6	6	6	6	6	5	6	7
O	8	7	7	7	7	7	7	6	5	6
N	9	8	8	8	8	8	8	7	6	5

- Recurrence Relation:

For each $i = 1..M$

For each $j = 1..N$

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases} \end{cases}$$

Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture OCR or keyboard errors
    Example: ***m*** more likely to be mis-typed as ***n*** than as ***q***
  - Therefore, replacing ***m*** by ***n*** is a smaller edit distance than by ***q***
  - This may be formulated as a probability model
- Requires weight matrix as input
- Modify dynamic programming to handle weights

Using edit distances

- Given query, first enumerate all character sequences within a preset (weighted) edit distance (e.g., 2)
- Intersect this set with list of “correct” words
- Show terms you found to user as suggestions
- Alternatively,
  - We can look up all possible corrections in our inverted index and return all docs … slow
  - We can run with a single most likely correction
- The alternatives disempower the user, but save a round of interaction with the user

Edit distance to all dictionary terms?

- Given a (mis-spelled) query – do we compute its edit distance to every dictionary term?
  - Expensive and slow
  - Alternative?
- How do we cut the set of candidate dictionary terms?
- One possibility is to use *n*-gram overlap for this
- This can also be used by itself for spelling correction.

n-gram overlap

- Enumerate all the *n*-grams in the query string as well as in the lexicon
- Use the *n*-gram index (recall wild-card search) to retrieve all lexicon terms matching any of the query *n*-grams
- Threshold by number of matching *n*-grams
  - Variants – weight by keyboard layout, etc.

Example with trigrams

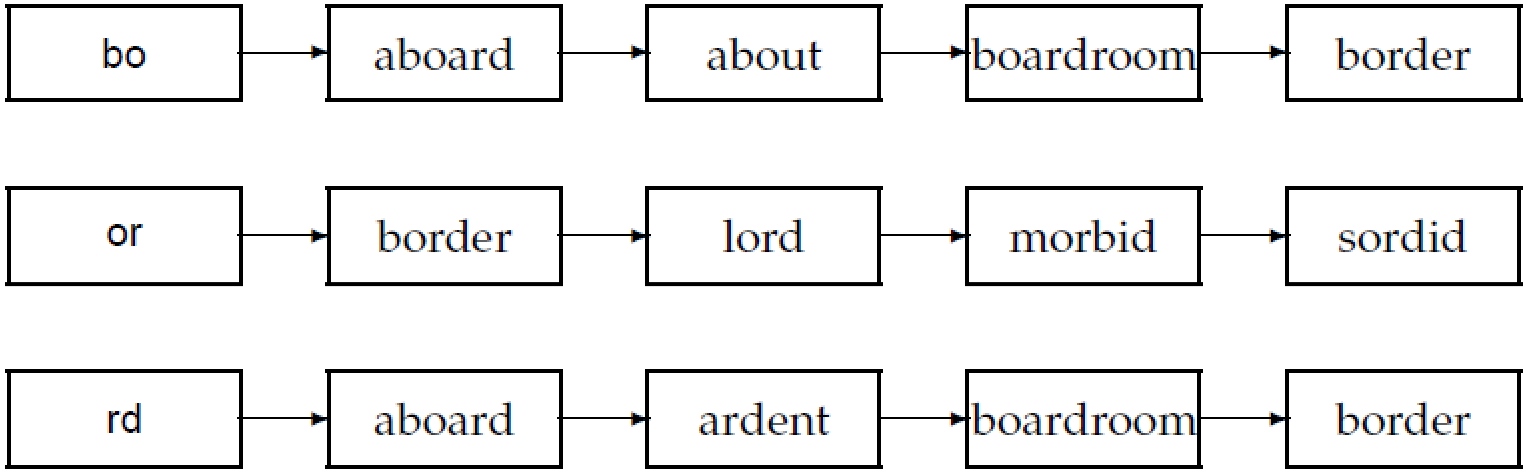
- Suppose the text is ***november***
  - Trigrams are *nov, ove, vem, emb, mbe, ber*.
- The query is ***december***
  - Trigrams are *dec, ece, cem, emb, mbe, ber*.
- So 3 trigrams overlap (of 6 in each term)
- How can we turn this into a normalized measure of overlap?

One option – Jaccard coefficient

- A commonly-used measure of overlap
- Let *X* and *Y* be two sets; then the J.C. is

$$|X \cap Y| / |X \cup Y|$$

- Equals 1 when *X* and *Y* have the same elements and zero when they are disjoint
- *X* and *Y* don't have to be of the same size
- Always assigns a number between 0 and 1
  - Now threshold to decide if you have a match
  - E.g., if J.C. > 0.8, declare a match



► **Figure 3.7** Matching at least two of the three 2-grams in the query bord.

If the postings stored the (pre-computed) number of bigrams in boardroom (namely, 8), we have all the information we require to compute the Jaccard coefficient to be 2/(8+3−2); the numerator is obtained from the number of postings hits (2, from bo and rd) while the denominator is the sum of the number of bigrams in bord and boardroom, less the number of postings hits.

Context-sensitive spell correction

- Text: ***I flew from Heathrow to Narita.***
- Consider the phrase query ***"flew form Heathrow"***
- We'd like to respond

  Did you mean "***flew from Heathrow***"?

because no docs matched the query phrase.

Context-sensitive correction

- Need surrounding context to catch this.
- First idea: retrieve dictionary terms close (in weighted edit distance) to each query term
- Now try all possible resulting phrases with one word “fixed” at a time
    - ***flew from heathrow***
    - ***fled form heathrow***
    - ***flea form heathrow***
- **Hit-based spelling correction:** Suggest the alternative that has lots of hits.

Context sensitive spelling correction

- If a query phrase yields a small set of retrieved documents, search engines often offer potential corrections
  - *animals form Australia* is corrected to *animals **from** Australia*

- Approach
  - Enumerate all possible corrections of each query term
  - Substitute each correction into the phrase
  - Run a query against the index, find number of matching documents
  - Offer most common phrasings

```
    8 animals form australia
    6 animal form australia
    0 animal form austria
  155 animal from austria
 3850 animals from austria
55500 animals from australia
```

Exercise

- Suppose that for ***“flew form Heathrow”*** we have 7 alternatives for flew, 19 for form and 3 for heathrow.

How many “corrected” phrases will we enumerate in this scheme?

Phonetic Correction

Soundex

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific – mainly for names
  - E.g., ***chebyshev*** $\rightarrow$ ***tchebycheff***
- Invented for the U.S. census … in 1918

Soundex – typical algorithm

- Turn every token to be indexed into a 4-character reduced form
- Do the same with query terms
- Build and search an index on the reduced forms
  - (when the query calls for a soundex match)

- http://www.creativyst.com/Doc/Articles/SoundEx1/SoundEx1.htm#Top

Soundex – typical algorithm

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to '0' (zero):
   'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:

- B, F, P, V $\rightarrow$ 1
- C, G, J, K, Q, S, X, Z $\rightarrow$ 2
- D,T $\rightarrow$ 3
- L $\rightarrow$ 4
- M, N $\rightarrow$ 5
- R $\rightarrow$ 6

Soundex continued

4. Remove all pairs of consecutive digits.
5. Remove all zeros from the resulting string.
6. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., ***Herman*** becomes H655.

Ashcraft
Ashcroft
Pfister

Will ***hermann*** generate the same code?