

---

# COMP9319 Web Data Compression and Search

Distributed path queries,  
Compressed inverted index

# Intro to distributed query evaluation

---

Web data is inherently distributed

Reuse some techniques from distributed RDBMS if some schema info is known

**New techniques required if no schema info is known**

In XML, these links are denoted in XLinks and XPointers.

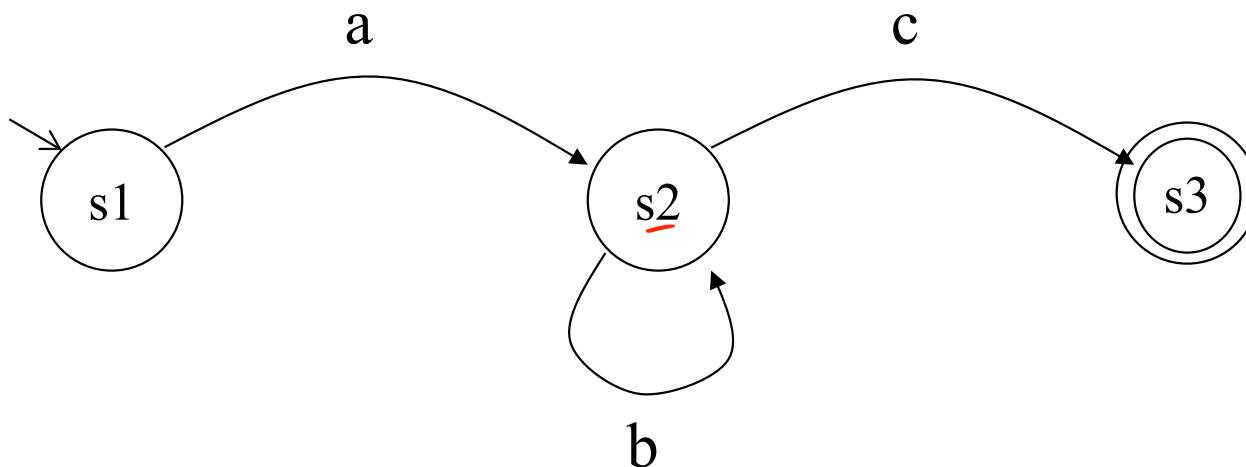
# Example query

---

Assume data are distributed in 3 sites

Assume the RPE: a.b\*.c       $a/b^*/c$

Assume the query starts from Site 1



# Regular path expressions

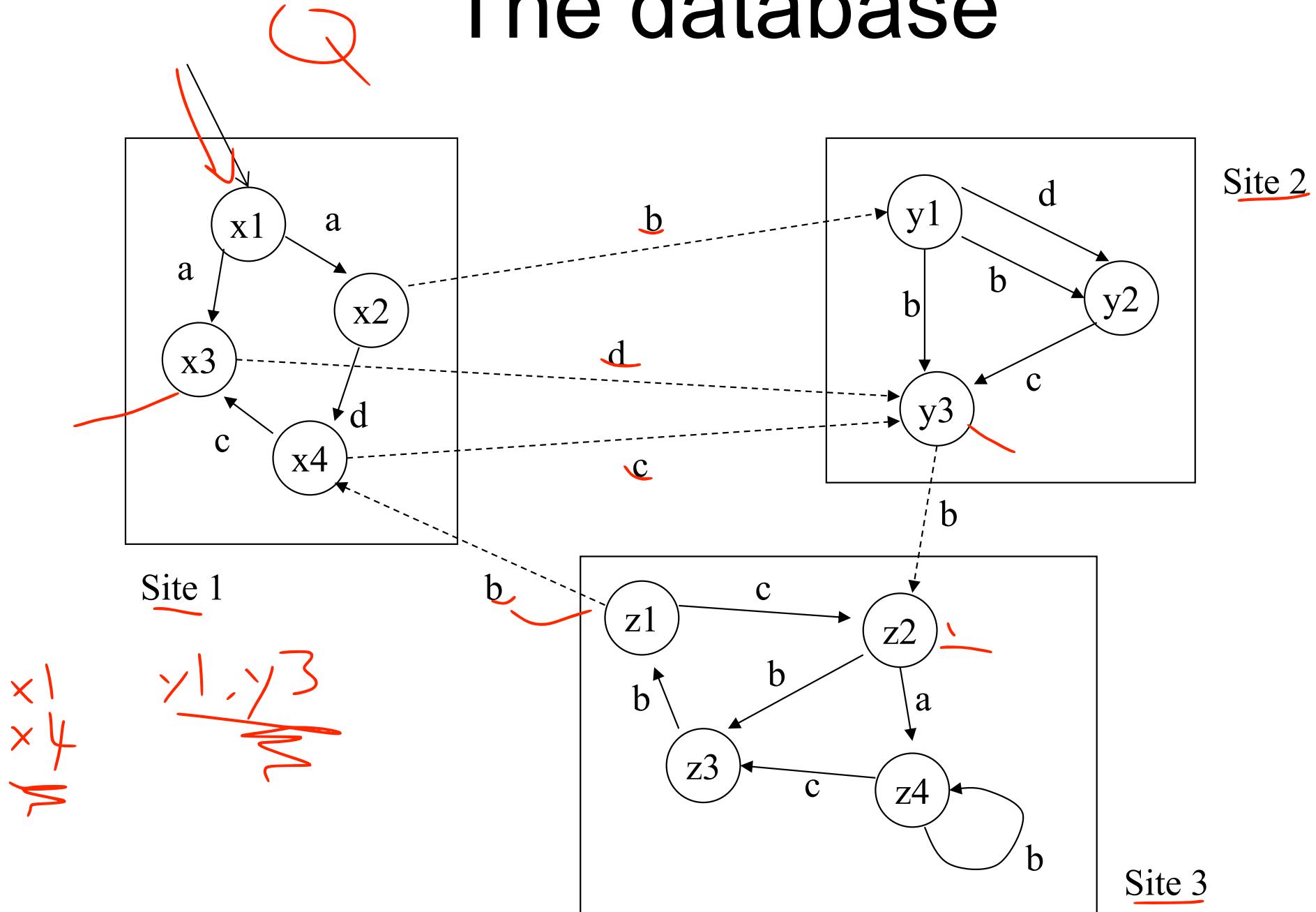
---

Regular expressions for path, e.g.:

a.b\*.c

a.b+.c

# The database



# Naïve approach

---

A naïve approach takes too many communication steps

=> we have to do more work locally

A better approach needs to

1. identify all external references
2. identify targets of external references

# Input and output nodes

---

## Site 1

Inputs:  $x_1$  (root),  $x_4$

Outputs:  $y_1$ ,  $y_3$

## Site 2

Inputs:  $y_1$ ,  $y_3$

Outputs:  $z_2$

## Site 3

Inputs:  $z_2$

Outputs:  $x_4$

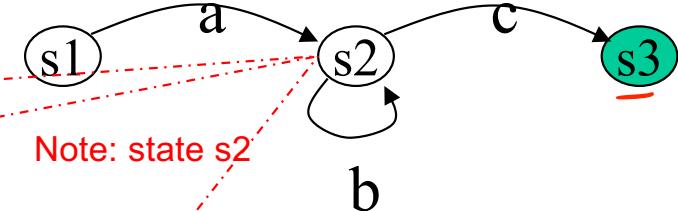
# Query Processing

---

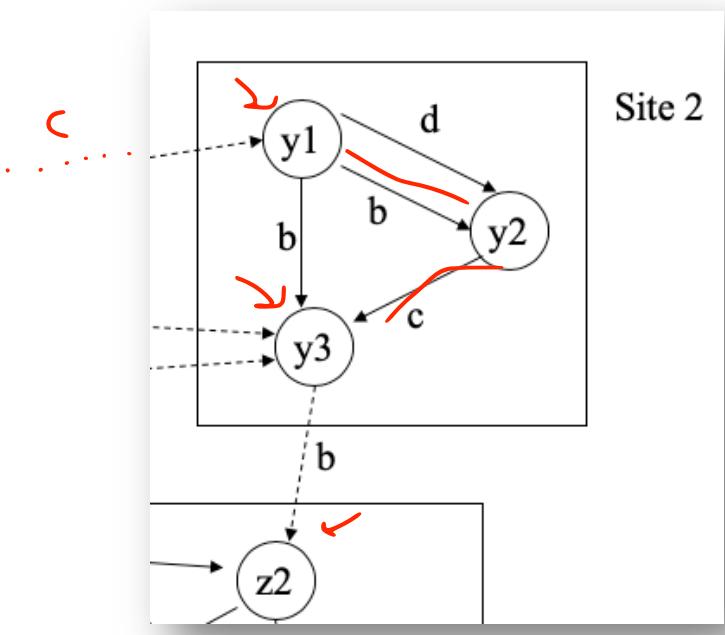
- 1 Given a query, we compute its automaton
- 2 Send it to each site
- 3 Start an identical process at each site
- 4 Compute two sets Stop(n, s) and Result(n, s)
- 5 Transmits the relations to a central location  
and get their union

# Stop and Result at site 2

| Start        | Stop         |
|--------------|--------------|
| $(y_1, s_2)$ | $(z_2, s_2)$ |
| $(y_3, s_2)$ | $(z_2, s_2)$ |



| Start        | Result |
|--------------|--------|
| $(y_1, s_2)$ | $y_3$  |
| $(y_1, s_3)$ | $y_1$  |
| $(y_3, s_3)$ | $y_3$  |



# Union the relations from all sites

| Start    | Stop     |  |
|----------|----------|--|
| (x1, s1) | (y1, s2) |  |
| (x4, s2) | (y3, s3) |  |
| (y1, s2) | (z2, s2) |  |
| (y3, s2) | (z2, s2) |  |
| (z2, s2) | (x4, s2) |  |

| Start    | Result |  |
|----------|--------|--|
| (x1, s3) | x1     |  |
| (x4, s2) | x3     |  |
| (x4, s3) | x4     |  |
| (y1, s2) | y3     |  |
| (y1, s3) | y1     |  |
| (y3, s3) | y3     |  |
| (z2, s1) | z3     |  |
| (z2, s2) | z2     |  |
| (z2, s3) | z2     |  |

The result of the query  
is {y3, z2, x3}

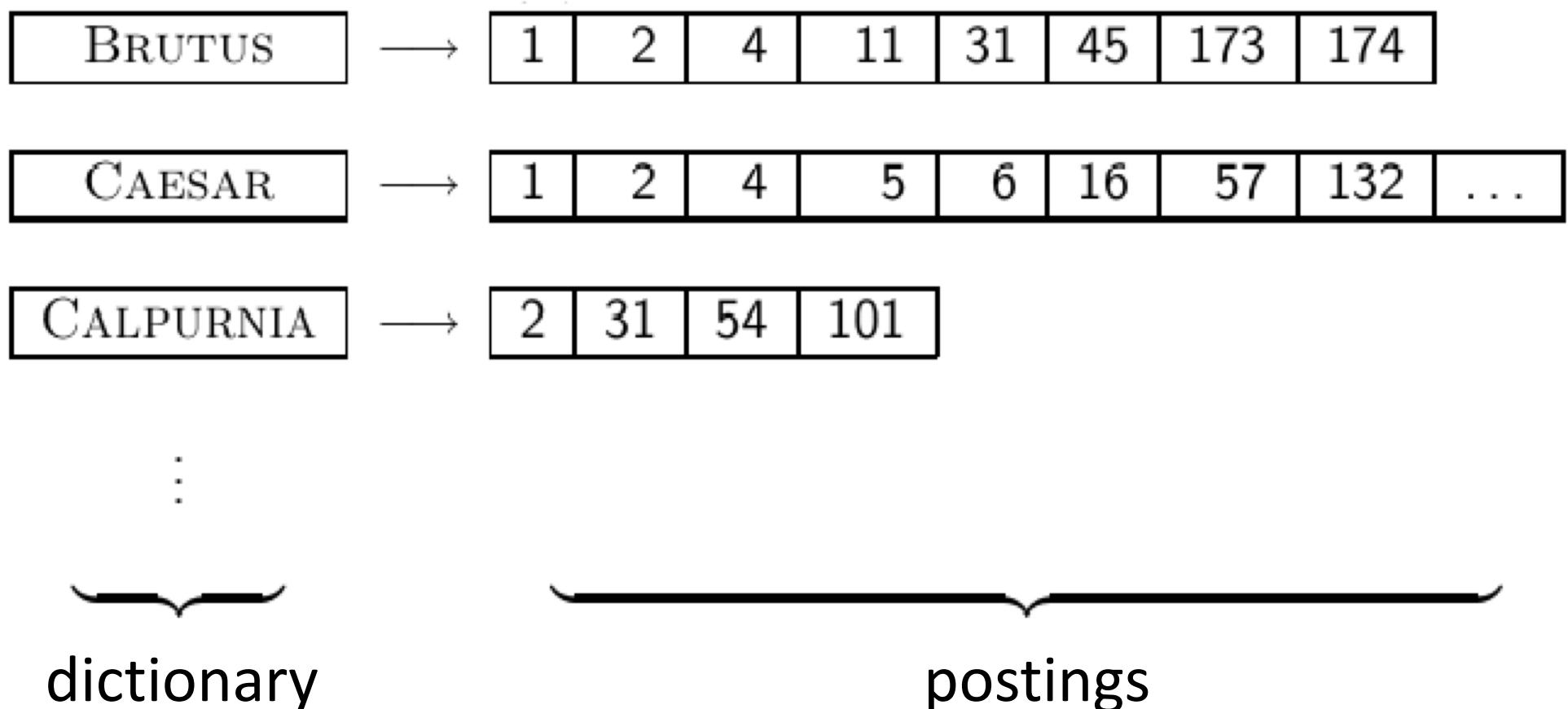
# COMP9319: Web Data Compression and Search

## Inverted index revisit & its compression

Slides modified from Hinrich Schütze and Christina Lioma slides on IIR

# Inverted Index

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



# Inverted index construction

---

- 1 Collect the documents to be indexed:

Friends, Romans, countrymen.

So let it be with Caesar ...

- 2 Tokenize the text, turning each document into a list of tokens:

Friends

Romans

countrymen

So

...

- 3 Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms:

friend

roman

countryman

so

...

- 4 Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.

# Tokenizing and preprocessing

---

**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:



**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

# Generate posting

---

| 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     |

**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



# Sort postings

---

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



# Create postings lists, determine document frequency

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 1

noble 2

so 2

the 1

the 2

told 2

you 2

was 2

was 1

with 2

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

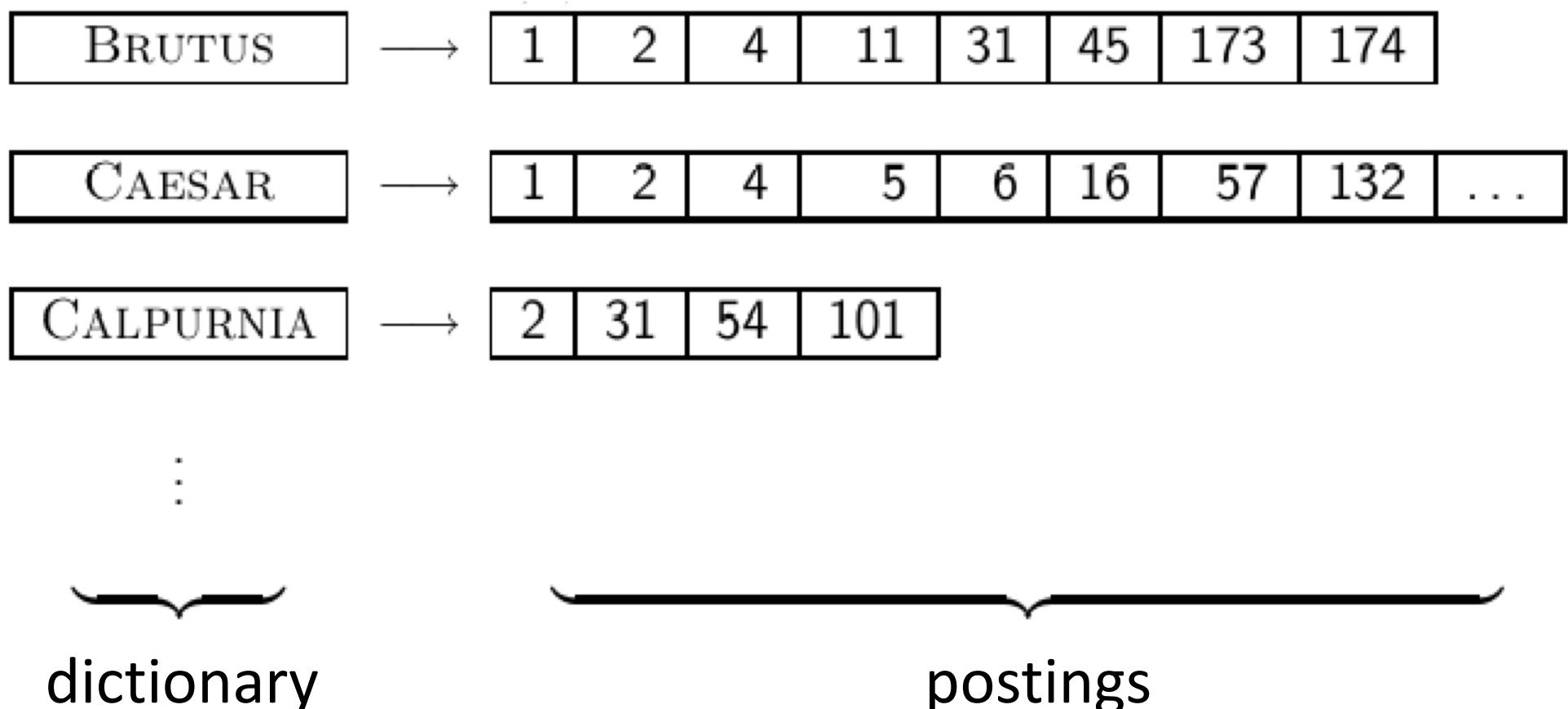


| term      | doc. | freq. |
|-----------|------|-------|
| ambitious | 1    | 1     |
| be        | 1    | 1     |
| brutus    | 1    | 2     |
| brutus    | 2    | 1     |
| capitol   | 1    | 1     |
| caesar    | 1    | 1     |
| caesar    | 2    | 2     |
| caesar    | 2    | 2     |
| did     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

→ postings lists

|         |
|---------|
| → 2     |
| → 2     |
| → 1 → 2 |
| → 1     |
| → 1 → 2 |
| → 1     |
| → 1     |
| → 2     |
| → 1     |
| → 1     |
| → 2     |
| → 1     |
| → 2     |
| → 1     |
| → 2     |
| → 1     |
| → 2     |
| → 1 → 2 |
| → 2     |
| → 2     |
| → 2     |
| → 1 → 2 |
| → 2     |

Split the result into dictionary and postings file



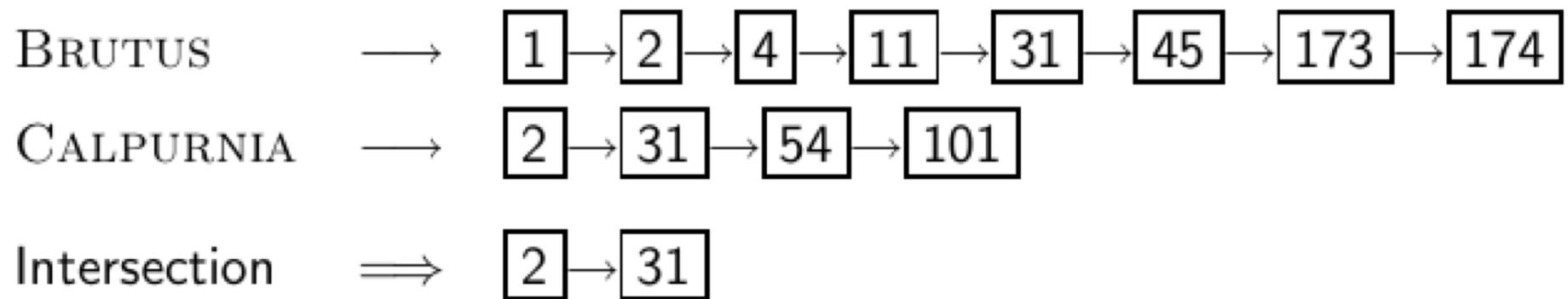
# Simple conjunctive query (two terms)

---

- Consider the query: BRUTUS AND CALPURNIA
- To find all matching documents using inverted index:
  - ① Locate BRUTUS in the dictionary
  - ② Retrieve its postings list from the postings file
  - ③ Locate CALPURNIA in the dictionary
  - ④ Retrieve its postings list from the postings file
  - ⑤ Intersect the two postings lists
  - ⑥ Return intersection to user

# Intersecting two posting lists

---



- This is linear in the length of the postings lists.
- Note: This only works if postings lists are sorted.

# Intersecting two posting lists

---

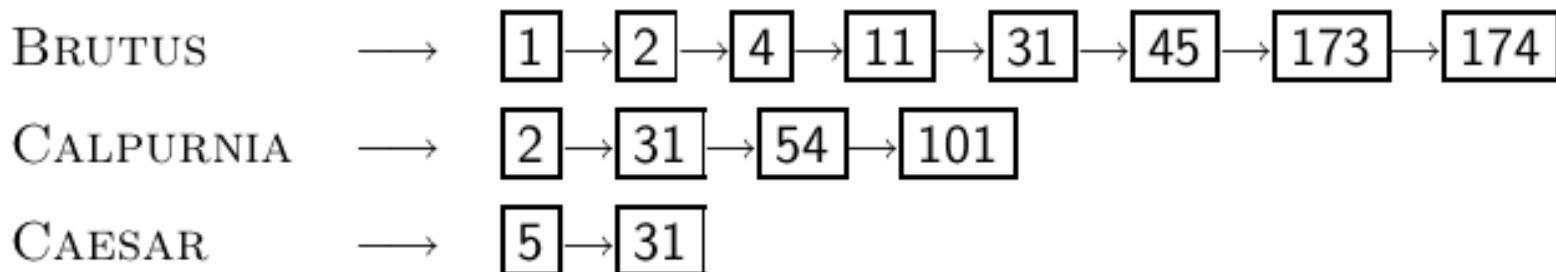
INTERSECT( $p_1, p_2$ )

- 1     $answer \leftarrow \langle \rangle$
- 2    **while**  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$
- 3    **do if**  $docID(p_1) = docID(p_2)$ 
  - 4        **then** ADD( $answer, docID(p_1)$ )
  - 5             $p_1 \leftarrow next(p_1)$
  - 6             $p_2 \leftarrow next(p_2)$
- 7    **else if**  $docID(p_1) < docID(p_2)$ 
  - 8        **then**  $p_1 \leftarrow next(p_1)$
  - 9        **else**  $p_2 \leftarrow next(p_2)$
- 10    **return**  $answer$

# Typical query optimization

---

- Example query: BRUTUS AND CALPURNIA AND CAESAR
- Simple and effective optimization: **Process in order of increasing frequency**
- Start with the shortest postings list, then keep cutting further
- In this example, first CAESAR, then CALPURNIA, then BRUTUS



# Optimized intersection algorithm for conjunctive queries

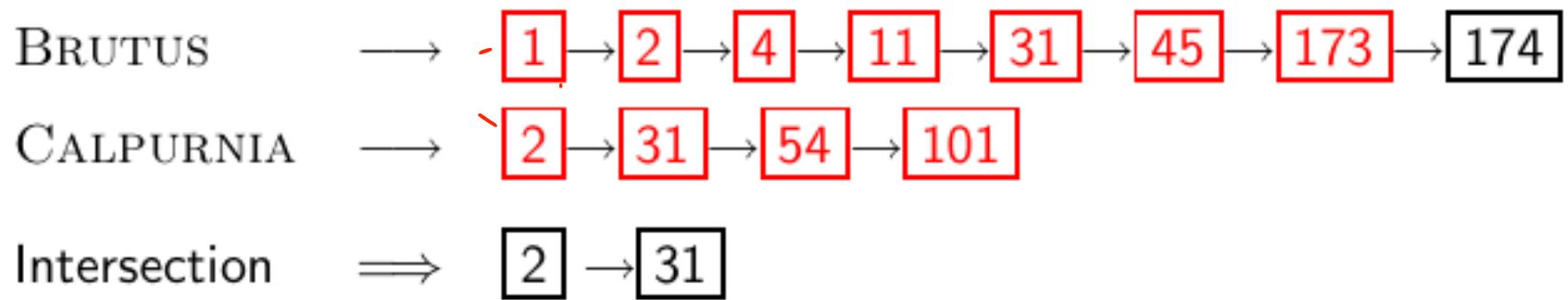
---

INTERSECT( $\langle t_1, \dots, t_n \rangle$ )

- 1  $terms \leftarrow \text{SORTBYINCREASINGFREQUENCY}(\langle t_1, \dots, t_n \rangle)$
- 2  $result \leftarrow \text{postings}(\text{first}(terms))$
- 3  $terms \leftarrow \text{rest}(terms)$
- 4 **while**  $terms \neq \text{NIL}$  and  $result \neq \text{NIL}$
- 5   **do**  $result \leftarrow \text{INTERSECT}(result, \text{postings}(\text{first}(terms)))$
- 6     $terms \leftarrow \text{rest}(terms)$
- 7 **return**  $result$

# Recall basic intersection algorithm

---



- Linear in the length of the postings lists.
- Can we do better?

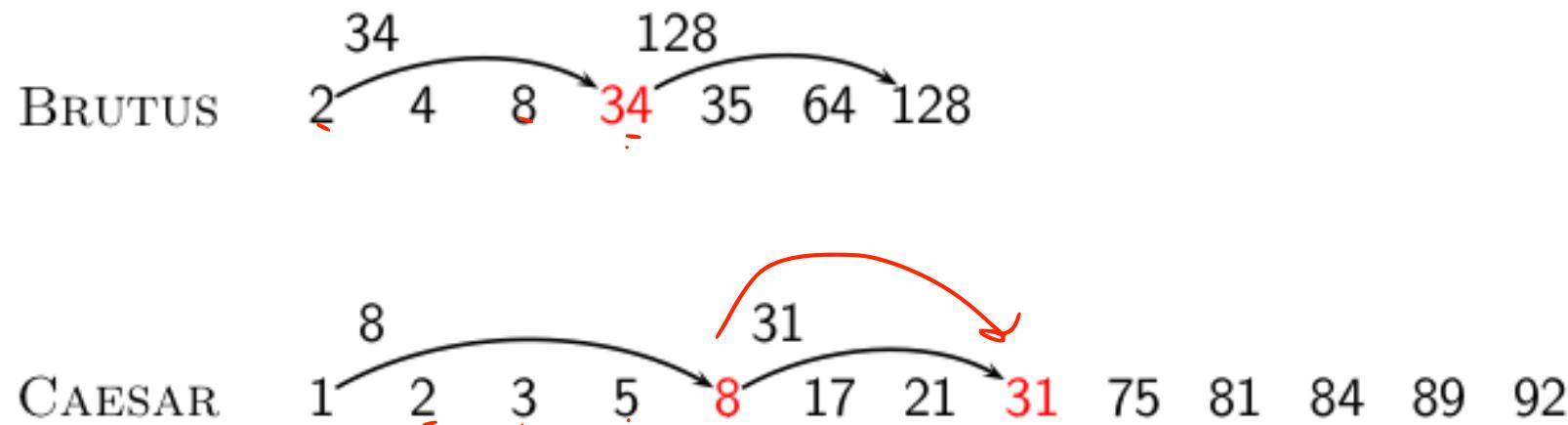
# Skip pointers

---

- Skip pointers allow us to **skip** postings that will not figure in the search results.
- This makes intersecting postings lists more efficient.
- Some postings lists contain several million entries – so efficiency can be an issue even if basic intersection is linear.
- Where do we put skip pointers?
- How do we make sure intersection results are correct?

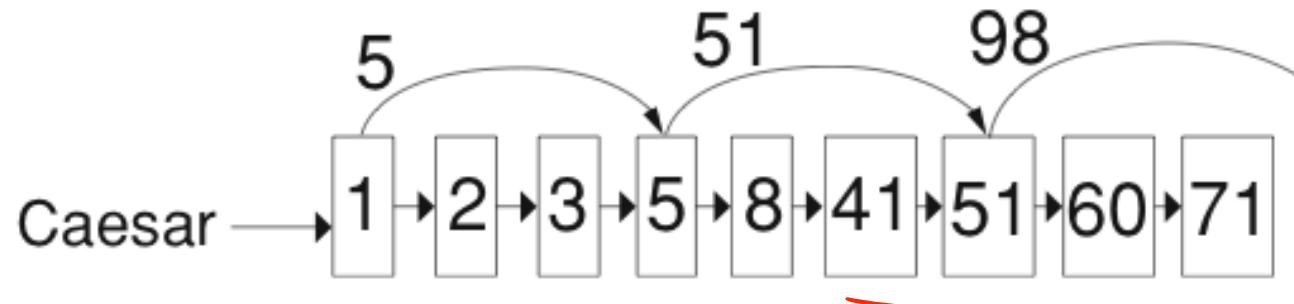
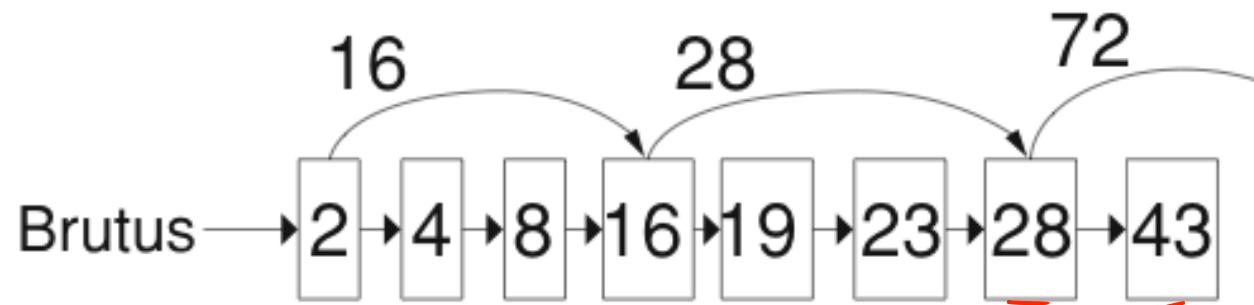
# Basic idea

---



# Skip lists: Larger example

---



# Intersection with skip pointers

---

INTERSECTWITHSKIPS( $p_1, p_2$ )

```
1  answer ← ⟨ ⟩
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $\text{docID}(p_1) = \text{docID}(p_2)$ 
4    then ADD( $answer, \text{docID}(p_1)$ )
5     $p_1 \leftarrow \text{next}(p_1)$ 
6     $p_2 \leftarrow \text{next}(p_2)$ 
7  else if  $\text{docID}(p_1) < \text{docID}(p_2)$ 
8    then if  $\text{hasSkip}(p_1)$  and ( $\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2)$ )
9      then while  $\text{hasSkip}(p_1)$  and ( $\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2)$ )
10     do  $p_1 \leftarrow \text{skip}(p_1)$ 
11     else  $p_1 \leftarrow \text{next}(p_1)$ 
12   else if  $\text{hasSkip}(p_2)$  and ( $\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1)$ )
13     then while  $\text{hasSkip}(p_2)$  and ( $\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1)$ )
14     do  $p_2 \leftarrow \text{skip}(p_2)$ 
15     else  $p_2 \leftarrow \text{next}(p_2)$ 
16 return  $answer$ 
```

# Where do we place skips?

---

- Tradeoff: number of items skipped vs. frequency skip can be taken
- More skips: Each skip pointer skips only a few items, but we can frequently use it.
- Fewer skips: Each skip pointer skips many items, but we can not use it very often.

# Phrase queries

---

- We want to answer a query such as [stanford university] – as a phrase.
- Thus *The inventor Stanford Ovshinsky never went to university* should **not** be a match.
- The concept of phrase query has proven easily understood by users.
- About 10% of web queries are phrase queries.
- Consequence for inverted index: it no longer suffices to store docIDs in postings lists.
- Two ways of extending the inverted index:
  - biword index (cf. COMP6714)
  - positional index

# Positional indexes

---

- Postings lists in a **nonpositional** index: each posting is just a docID
- Postings lists in a **positional** index: each posting is a docID and a **list of positions**

# Positional indexes: Example

---

Query: “ $to_1 be_2 or_3 not_4 to_5 be_6$ ”

TO, 993427:

```
< 1: <7, 18, 33, 72, 86, 231>;  
 2: <1, 17, 74, 222, 255>;  
 4: <8, 16, 190, 429, 433>;  
 5: <363, 367>;  
 7: <13, 23, 191>; . . . >
```

BE, 178239:

```
< 1: <17, 25>;  
 4: <17, 191, 291, 430, 434>;  
 5: <14, 19, 101>; . . . > Document 4 is a match!
```

# Inverted index

---

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

|        |   |   |   |   |    |    |    |     |     |
|--------|---|---|---|---|----|----|----|-----|-----|
| BRUTUS | → | 1 | 2 | 4 | 11 | 31 | 45 | 173 | 174 |
|--------|---|---|---|---|----|----|----|-----|-----|

|        |   |   |   |   |   |   |    |    |     |     |
|--------|---|---|---|---|---|---|----|----|-----|-----|
| CAESAR | → | 1 | 2 | 4 | 5 | 6 | 16 | 57 | 132 | ... |
|--------|---|---|---|---|---|---|----|----|-----|-----|

|           |   |   |    |    |     |
|-----------|---|---|----|----|-----|
| CALPURNIA | → | 2 | 31 | 54 | 101 |
|-----------|---|---|----|----|-----|

⋮

 **dictionary**

 **postings**

# Dictionaries

---

- The dictionary is the data structure for storing the term vocabulary.
- Term vocabulary: the **data**
- Dictionary: the **data structure** for storing the term vocabulary

# Dictionary as array of fixed-width entries

---

- For each term, we need to store a couple of items:
  - document frequency
  - pointer to postings list
  - ...
- Assume for the time being that we can store this information in a fixed-length entry.
- Assume that we store these entries in an array.

# Dictionary as array of fixed-width entries

---

| term   | document frequency | pointer to postings list |
|--------|--------------------|--------------------------|
| a      | 656,265            | →                        |
| aachen | 65                 | →                        |
| ...    | ...                | ...                      |
| zulu   | 221                | →                        |

space needed: 20 bytes 4 bytes 4 bytes

How do we look up a query term  $q_i$  in this array at query time?  
That is: which data structure do we use to locate the entry (row)  
in the array where  $q_i$  is stored?

# Data structures for looking up term

---

- Two main classes of data structures: hashes and trees
- Some IR systems use hashes, some use trees.
- Criteria for when to use hashes vs. trees:
  - Is there a fixed number of terms or will it keep growing?
  - What are the relative frequencies with which various keys will be accessed?
  - How many terms are we likely to have?

# Hashes

---

- Each vocabulary term is hashed into an integer.
- Try to avoid collisions
- At query time, do the following: hash query term, resolve collisions, locate entry in fixed-width array
- Pros: Lookup in a hash is faster than lookup in a tree.
  - Lookup time is constant.
- Cons
  - no way to find minor variants (*resume* vs. *résumé*)
  - no prefix search (all terms starting with *automat*)
  - need to rehash everything periodically if vocabulary keeps growing

# Trees

---

- Trees solve the prefix problem (find all terms starting with *automat*).
- Simplest tree: binary tree
- Search is slightly slower than in hashes:  $O(\log M)$ , where  $M$  is the size of the vocabulary.
- $O(\log M)$  only holds for **balanced** trees.
- Rebalancing binary trees is expensive.
- **B-trees** mitigate the rebalancing problem.
- B-tree definition: every internal node has a number of children in the interval  $[a, b]$  where  $a, b$  are appropriate positive integers, e.g.,  $[2, 4]$ .

# Sort-based index construction

---

- As we build index, we parse docs one at a time.
- The final postings for any term are incomplete until the end.
- Can we keep all postings in memory and then do the sort in-memory at the end?
- No, not for large collections
- At 10–12 bytes per postings entry, we need a lot of space for large collections.
- But in-memory index construction does not scale for large collections.
- Thus: We need to store intermediate results on disk.

# Same algorithm for disk?

---

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?
- No: Sorting for example 100,000,000 records on disk is too slow – too many disk seeks.
- We need an **external** sorting algorithm.

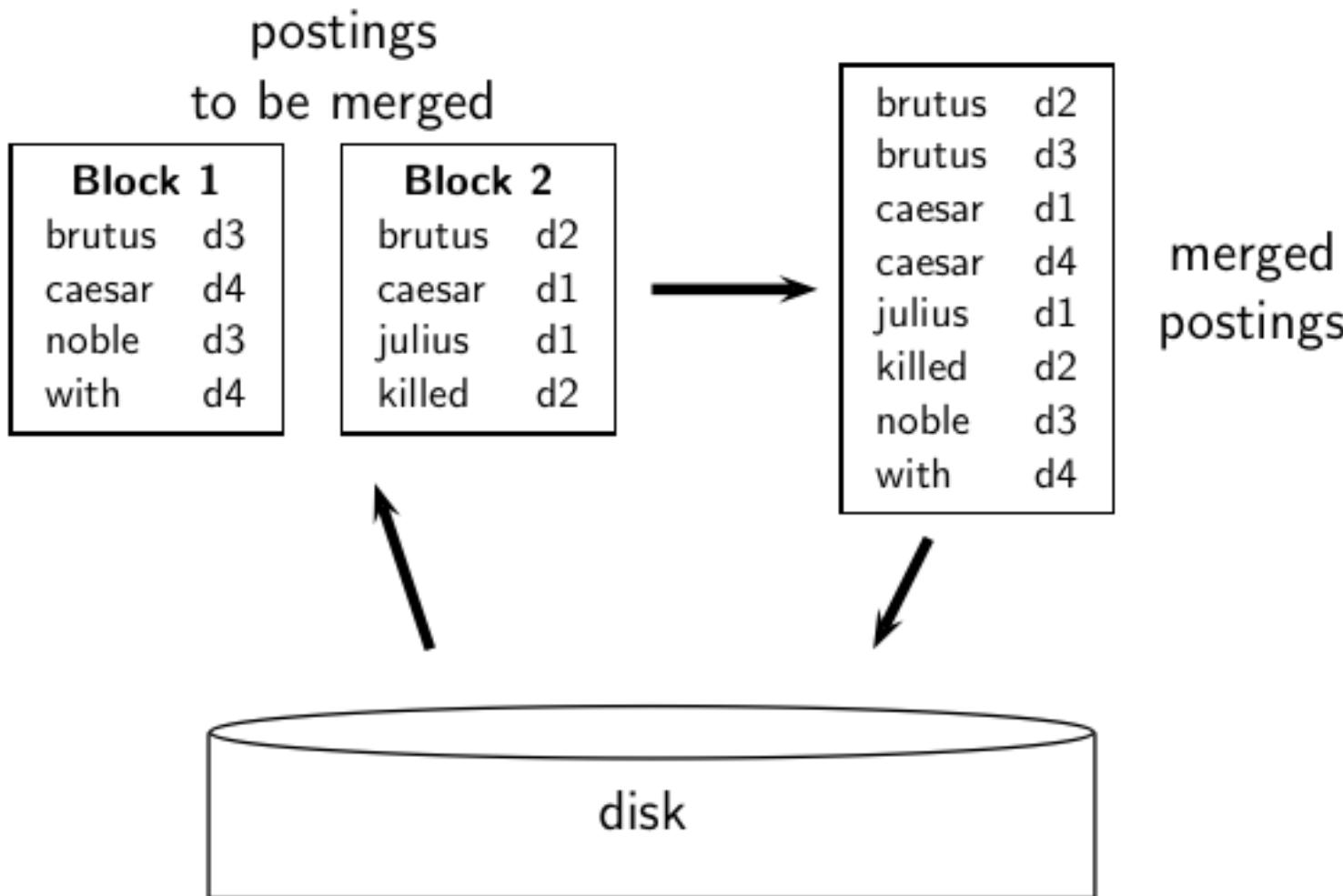
# “External” sorting algorithm (using few disk seeks)

---

- We must sort 100,000,000 non-positional postings.
  - Each posting has size 12 bytes (4+4+4: termID, docID, document frequency).
- Define a **block** to consist of 10,000,000 such postings
  - We can easily fit that many postings into memory.
  - We will have 10 such blocks.
- Basic idea of algorithm:
  - For each block: (i) accumulate postings, (ii) sort in memory, (iii) write to disk
  - Then merge the blocks into one long sorted order.

# Merging two blocks

---



# Why compression in information retrieval?

---

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory
- Then for the postings file
  - Motivation: reduce disk space needed, decrease time needed to read from disk
  - Note: Large search engines keep significant part of postings in memory
- We will devise various compression schemes for dictionary and postings.

# Dictionary compression

---

- The dictionary is small compared to the postings file.
- But we want to keep it in memory.
- Also: competition with other applications, cell phones, onboard computers, fast startup time
- So compressing the dictionary is important.

# Recall: Dictionary as array of fixed-width entries

---

| term   | document frequency | pointer to postings list |
|--------|--------------------|--------------------------|
| a      | 656,265            | →                        |
| aachen | 65                 | →                        |
| ...    | ...                | ...                      |
| zulu   | 221                | →                        |

Space needed: 20 bytes    4 bytes    4 bytes

for Reuters:  $(20+4+4)*400,000 = 11.2 \text{ MB}$

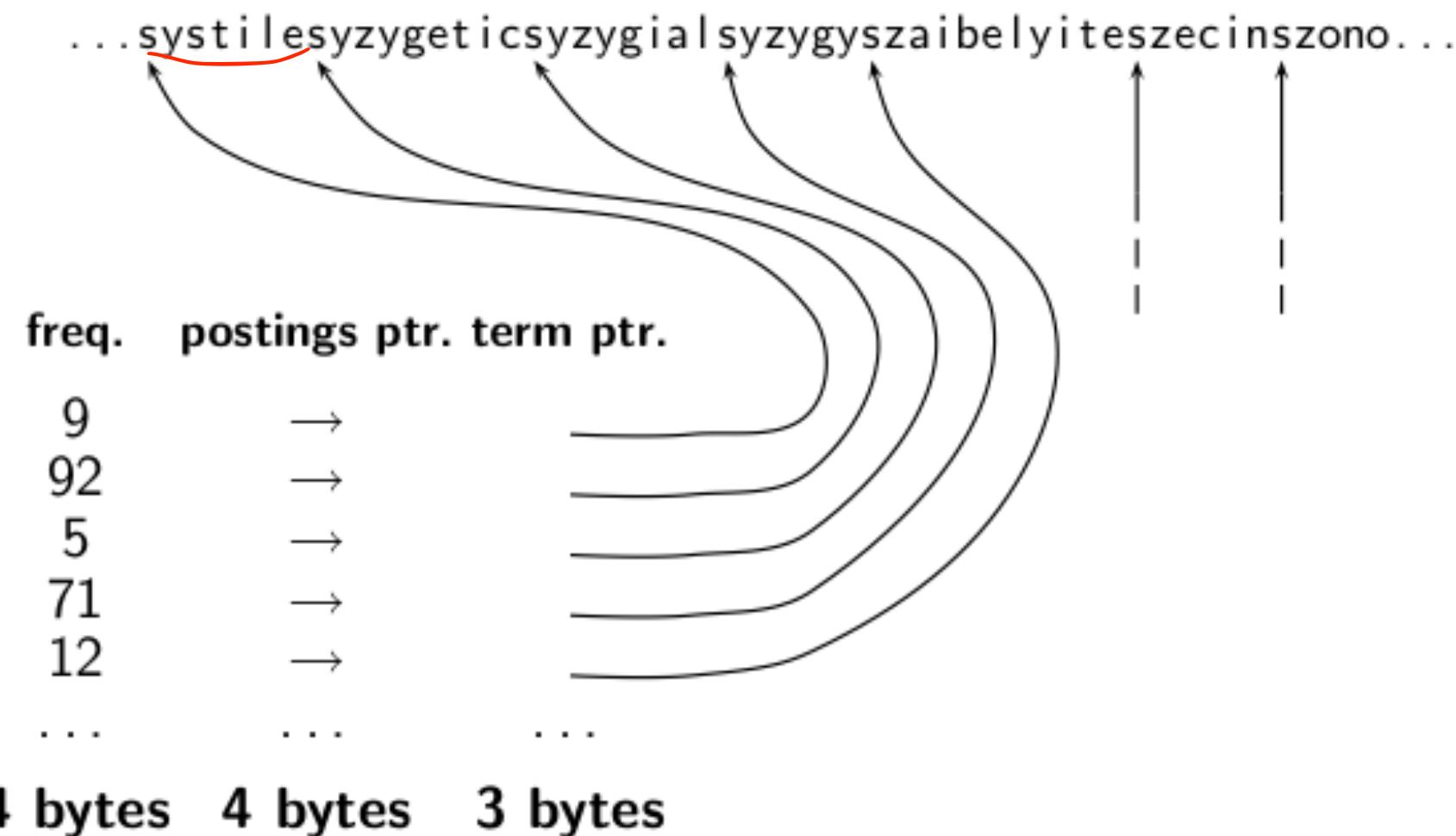
# Fixed-width entries are bad.

---

- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.
- We can't handle HYDROCHLOROFLUOROCARBONS and SUPERCALIFRAGILISTICEXPIALIDOCIOUS
- Average length of a term in English: 8 characters
- How can we use on average 8 characters per term?

# Dictionary as a string

---



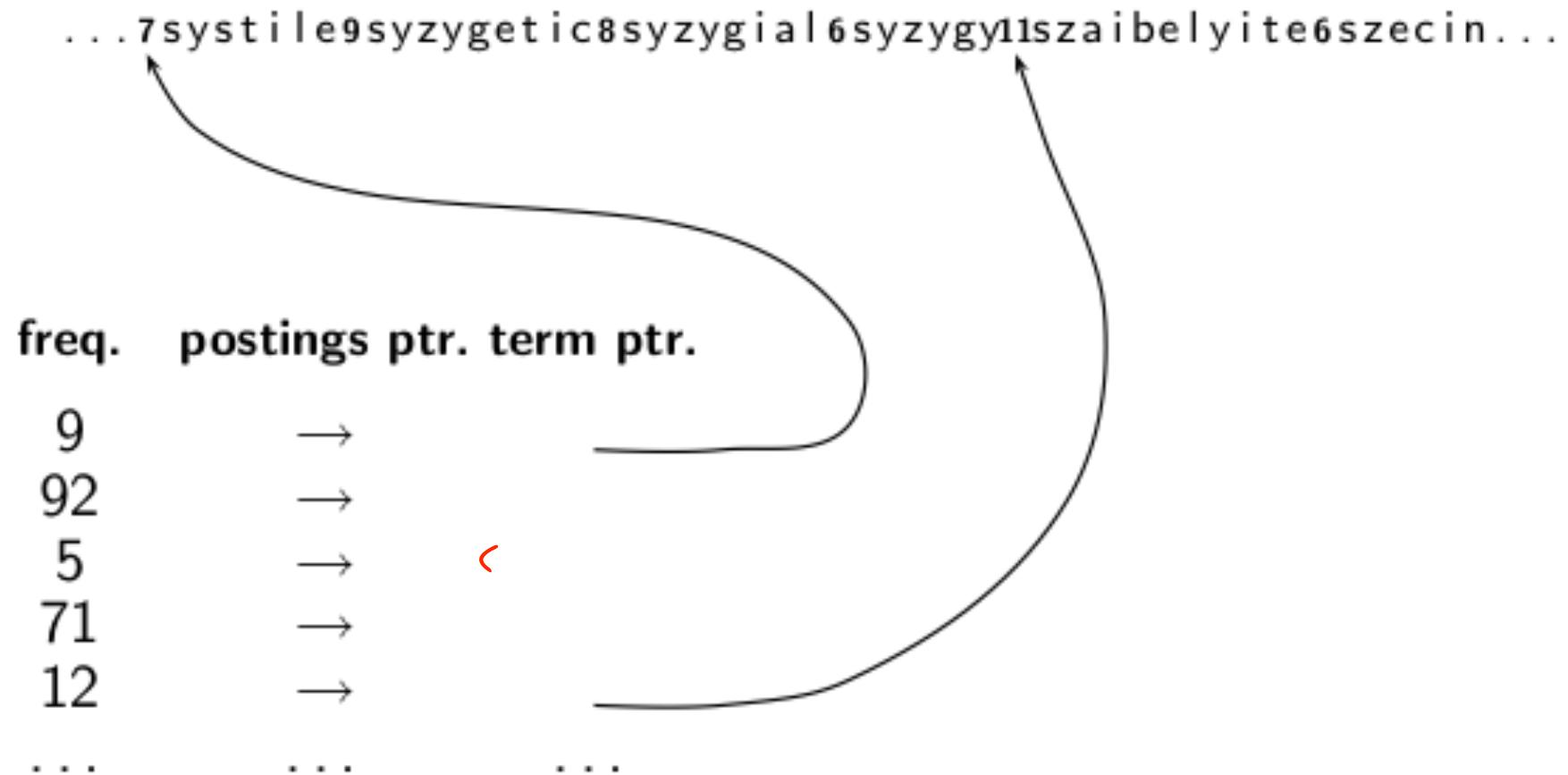
# Space for dictionary as a string

---

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
- 3 bytes per pointer into string (need  $\log_2 8 \cdot 400000 < 24$  bits to resolve  $8 \cdot 400,000$  positions)
- Space:  $400,000 \times (4 + 4 + 3 + 8) = 7.6\text{MB}$  (compared to 11.2 MB for fixed-width array)

# Dictionary as a string with blocking

---



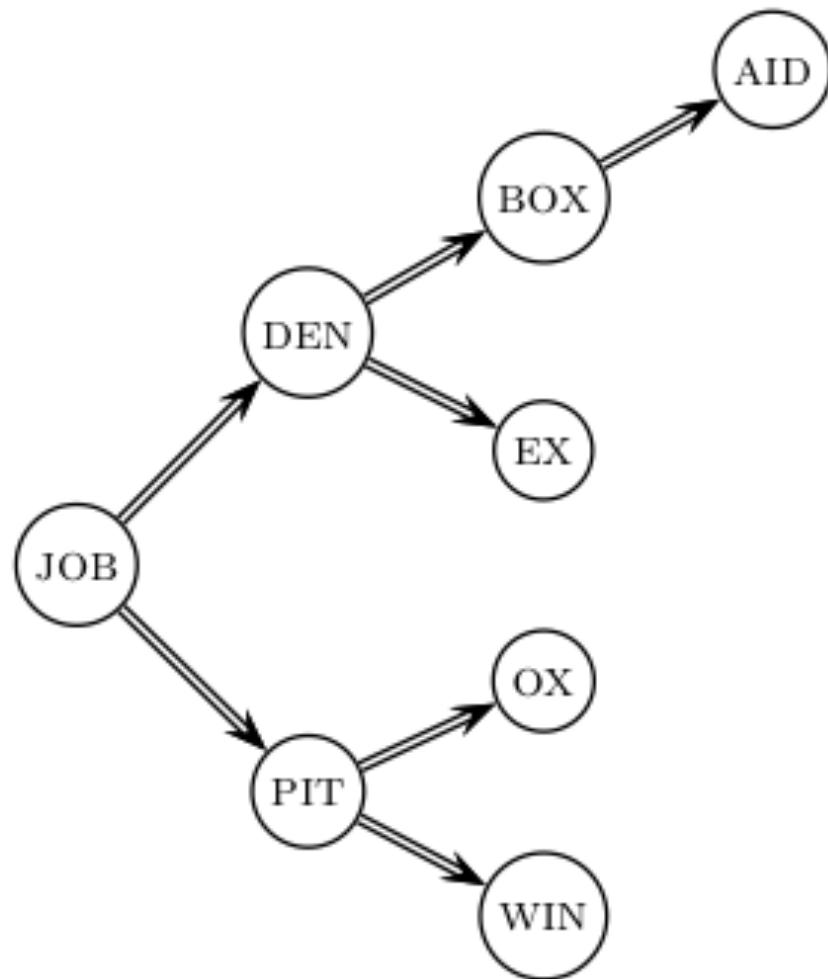
# Space for dictionary as a string with blocking

---

- Example block size  $k = 4$
- Where we used  $4 \times 3$  bytes for term pointers without blocking . . .
- . . .we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
- We save  $12 - (3 + 4) = 5$  bytes per block.
- Total savings:  $400,000/4 * 5 = 0.5$  MB
- This reduces the size of the dictionary from 7.6 MB to 7.1 MB.

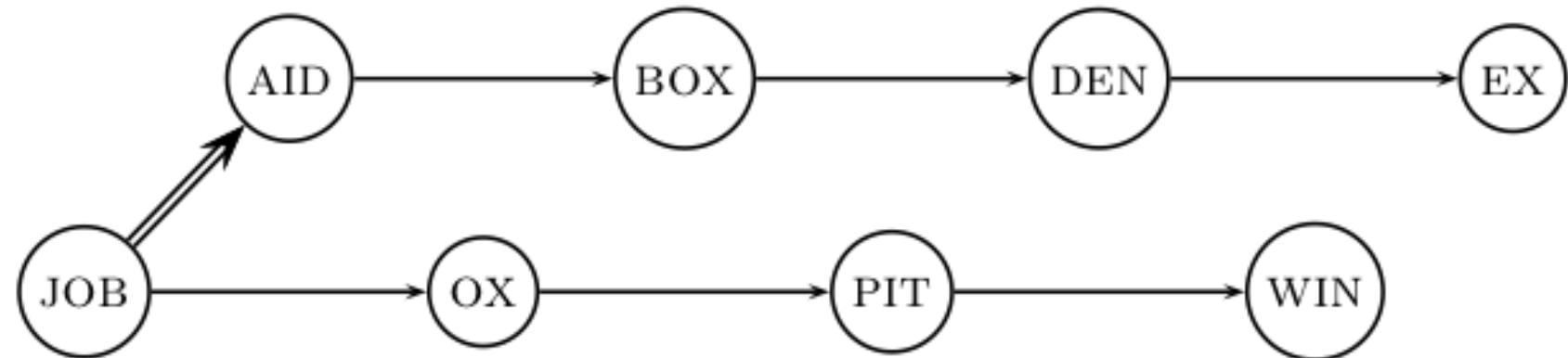
# Lookup of a term without blocking

---



# Lookup of a term with blocking: (slightly) slower

---



# Front coding

---

One block in blocked compression ( $k = 4$ ) . . .

**8** automata **8** automate **9** automatic **10** automation



. . . further compressed with front coding.

**8** automat\*a 1°e 2°ic 3°ion

# Dictionary compression for Reuters: Summary

---

| data structure                        | size in MB |
|---------------------------------------|------------|
| dictionary, fixed-width               | 11.2       |
| dictionary, term pointers into string | 7.6        |
| ~, with blocking, k = 4               | 7.1        |
| ~, with blocking & front coding       | 5.9        |

# Postings compression

---

- The postings file is much larger than the dictionary, factor of at least 10.
- Key desideratum: store each posting compactly
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use  $\log_2 800,000 \approx 19.6 < 20$  bits per docID.
- Our goal: use a lot less than 20 bits per docID.

# Key idea: Store gaps instead of docIDs

---

- Each postings list is ordered in increasing order of docID.
- Example postings list: COMPUTER: 283154, 283159, 283202, . . .
- It suffices to store **gaps**:  $283159 - 283154 = 5$ ,  $283202 - 283154 = 43$
- Example postings list using gaps : COMPUTER: 283154, 5, 43, . . .
- Gaps for frequent terms are small.
- Thus: We can encode small gaps with fewer than 20 bits.

# Gap encoding

---

|                | encoding | postings list |        |        |        |        |     |  |
|----------------|----------|---------------|--------|--------|--------|--------|-----|--|
| THE            | docIDs   | ...           | 283042 | 283043 | 283044 | 283045 | ... |  |
|                | gaps     |               | 1      | 1      | 1      | 1      | ... |  |
| COMPUTER       | docIDs   | ...           | 283047 | 283154 | 283159 | 283202 | ... |  |
|                | gaps     |               | 107    | 5      | 43     |        | ... |  |
| ARACHNOCENTRIC | docIDs   | 252000        | 500100 |        |        |        |     |  |
|                | gaps     | 252000        | 248100 |        |        |        |     |  |

# Variable length encoding

---

- Aim:
  - For ARACHNOCENTRIC and other rare terms, we will use about 20 bits per gap (= posting).
  - For THE and other very frequent terms, we will use only a few bits per gap (= posting).
- In order to implement this, we need to devise some form of **variable length encoding**.
- Variable length encoding uses few bits for small gaps and many bits for large gaps.

# Variable byte (VB) code

---

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a **continuation bit  $c$** .
- If the gap  $G$  fits within 7 bits, binary-encode it in the 7 available bits and set  $c = 1$ .
- Else: encode lower-order 7 bits and then use one or more additional bytes to encode the higher order bits using the same algorithm.
- At the end set the continuation bit of the last byte to 1 ( $c = 1$ ) and of the other bytes to 0 ( $c = 0$ ).

# VB code examples

---

|         |          |          |          |
|---------|----------|----------|----------|
| docIDs  | 824      | 829      | 215406   |
| gaps    |          | 5        | 214577   |
| VB code | 00000110 | 10111000 | 10000101 |

00001101 00001100 10110001

# VB code encoding algorithm

---

VBENCODENUMBER( $n$ )

```
1  bytes ← ⟨⟩  
2  while true  
3  do PREPEND(bytes,  $n \bmod 128$ )  
4    if  $n < 128$   
5      then BREAK  
6     $n \leftarrow n \text{ div } 128$   
7  bytes[LENGTH(bytes)] += 128  
8  return bytes
```

VBEENCODE( $numbers$ )

```
1  bytestream ← ⟨⟩  
2  for each  $n \in numbers$   
3  do bytes ← VBENCODENUMBER( $n$ )  
4    bytestream ← EXTEND(bytestream, bytes)  
5  return bytestream
```

# VB code decoding algorithm

---

```
VBDECODE(bytestream)
1  numbers ← ⟨⟩
2  n ← 0
3  for i ← 1 to LENGTH(bytestream)
4  do if bytestream[i] < 128
5    then n ← 128 × n + bytestream[i]
6    else n ← 128 × n + (bytestream[i] – 128)
7    APPEND(numbers, n)
8    n ← 0
9  return numbers
```

# Gamma codes for gap encoding

# Gamma code

---

- Represent a gap G as a pair of **length** and **offset**.
- Offset is the gap in binary, with the leading bit chopped off.
- For example  $13 \rightarrow 1101 \rightarrow 101 = \text{offset}$
- Length is the length of offset.
- For 13 (offset 101), the length is 3.
- Encode length in **unary** code: 1110.
- Gamma code of 13 is the concatenation of length and offset: 1110101.

1101

# Gamma code examples

---

| number | unary code | length     | offset     | $\gamma$ code         |
|--------|------------|------------|------------|-----------------------|
| 0      | 0          |            |            |                       |
| 1      | 1          | 10         | 0          | 0                     |
| 2      | 10         | 110        | 10         | 10,0                  |
| 3      | 11         | 1110       | 10         | 10,1                  |
| 4      |            | 11110      | 110        | 110,00                |
| 9      |            | 111111110  | 1110       | 1110,001              |
| 13     |            | 1110       | 101        | 1110,101              |
| 24     |            | 11110      | 1000       | 11110,1000            |
| 511    |            | 111111110  | 11111111   | 111111110,11111111    |
| 1025   |            | 1111111110 | 0000000001 | 1111111110,0000000001 |

# Properties of gamma code

---

- Gamma code is prefix-free
- The length of offset is  $\lfloor \log_2 G \rfloor$  bits.
- The length of length is  $\lfloor \log_2 G \rfloor + 1$  bits,
- So the length of the entire code is  $2 \times \lfloor \log_2 G \rfloor + 1$  bits.
- $\gamma$  codes are always of odd length.
- Gamma codes are within a factor of 2 of the optimal encoding length  $\log_2 G$ .

# Gamma codes: Alignment

---

- Machines have word boundaries – 8, 16, 32 bits
- Compressing and manipulating at granularity of bits can be slow.
- Variable byte encoding is aligned and thus potentially more efficient.
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost.

# Compression of Reuters

| data structure                        | size in MB |
|---------------------------------------|------------|
| dictionary, fixed-width               | 11.2       |
| dictionary, term pointers into string | 7.6        |
| ~, with blocking, k = 4               | 7.1        |
| ~, with blocking & front coding       | 5.9        |
| collection (text, xml markup etc)     | 3600.0     |
| collection (text)                     | 960.0      |
| T/D incidence matrix                  | 40,000.0   |
| postings, uncompressed (32-bit words) | 400.0      |
| postings, uncompressed (20 bits)      | 250.0      |
| postings, variable byte encoded       | 116.0      |
| postings, gamma encoded               | 101.0      |