

# Informatics 141

## Computer Science 121

### Information Retrieval

#### Lecture 17

*Duplication of course material for any commercial purpose without the explicit written permission of the professor is prohibited.*

*These course materials borrow, with permission, from those of Prof. Cristina Videira Lopes, Addison Wesley 2008, Chris Manning, Pandu Nayak, Hinrich Schütze, Heike Adel, Sascha Rothe, Jerome H. Friedman, Robert Tibshirani, and Trevor Hastie. Powerpoint theme by Prof. André van der Hoek.*

# Index Construction

# Index Construction

- Simple in-memory indexer

```
procedure BUILDINDEX( $D$ )  
   $I \leftarrow$  HashTable()  
   $n \leftarrow 0$   
  for all documents  $d \in D$  do  
     $n \leftarrow n + 1$   
     $T \leftarrow$  Parse( $d$ )  
    Remove duplicates from  $T$   
    for all tokens  $t \in T$  do  
      if  $I_t \notin I$  then  
         $I_t \leftarrow$  List<Posting>()  
      end if  
       $I_t.append(Posting(n))$   
    end for  
  end for  
  return  $I$   
end procedure
```

- ▷  $D$  is a set of text documents
  - ▷ Inverted list storage
  - ▷ Document numbering

- ▷ Parse document into tokens

# Index Construction

- Simple in-memory indexer

```
procedure BUILDINDEX( $D$ )  
     $I \leftarrow$  HashTable()  
     $n \leftarrow 0$   
    for all documents  $d \in D$  do  
         $n \leftarrow n + 1$   
         $T \leftarrow$  Parse( $d$ )  
        Remove duplicates from  $T$   
        for all tokens  $t \in T$  do  
            if  $I_t \notin I$  then  
                 $I_t \leftarrow$  List<Posting>()  
            end if  
             $I_t.append(Posting(n))$   
        end for  
    end for  
    return  $I$   
end procedure
```

▷  $D$  is a set of text documents  
▷ Inverted list storage  
▷ Document numbering  
▷ Parse document into tokens

Could this be a file,  
directly?

# Jeff Dean's (\*)

## “Latency Numbers Every Programmer Should Know”

- Latency Comparison Numbers (updated 2020)

• L1 cache reference	0.5-1.5 ns	
• L2 cache reference	5-7 ns	
• L3 cache reference	16-25 ns	
• Mutex lock/unlock	25 ns	
• 64MB Main memory reference	50-75 ns	
• Send 4KB over 100 Gbps HPC fabric	1,040 ns	
• Compress 1K bytes with Zippy	2,000 ns	2 us
• Read 1 MB sequentially from memory	3,000 ns	3 us
• Send 4KB over 10 Gbps ethernet	10,000 ns	10 us
• Read 1 MB sequentially from SSD	49,000 ns	49 us
• Read 1 MB sequentially from disk	825,000 ns	825 us
• Disk seek	2,000,000 ns	2,000 us
• Send packet CA->Netherlands->CA	150,000,000 ns	150,000 us

(\*) <https://ai.google/research/people/jeff/>

Original: <http://norvig.com/21-days.html#answers>

2019 : <https://gist.github.com/eshelman/343a1c46cb3fba142c1afdcdeec17646>

2020: [https://colin-scott.github.io/personal\\_website/research/interactive\\_latency.html](https://colin-scott.github.io/personal_website/research/interactive_latency.html)

# Use disk as “memory”?

- Can we use the same index construction algorithm for larger collections, but by using disk instead of memory?
- No: accessing/modifying  $T \sim 10^{8-9}$  records on disk
  - too slow: too many disk seeks.
- We need an *external* algorithm.

# Partial Indexes + Merging

---

- Build the inverted list structure until a certain size
  - Which size?

# Partial Indexes + Merging

---

- Build the inverted list structure until a certain size
  - Which size?
- Then write the partial index to disk, start making a new one



# Partial Indexes + Merging

---

- Build the inverted list structure until a certain size
  - Which size?
- Then write the partial index to disk, start making a new one
- At the end of this process, the disk is filled with many partial indexes, which are merged

# Partial Indexes + Merging

- Build the inverted list structure until a certain size
  - Which size?
- Then write the partial index to disk, start making a new one
- At the end of this process, the disk is filled with many partial indexes, which are merged
- Partial lists (=partial indexes) must be designed so they can be merged in small pieces
  - e.g., storing in alphabetical order

# Partial indexes

```
procedure BuildIndex(D)
```

```
  I ← HashTable
```

```
  n ← 0
```

```
  B ← [] # batch of documents
```

```
  while D is not empty do
```

```
    B ← GetBatch(D)
```

```
    for all documents d in B do
```

```
      n ← n+1
```

```
      T ← Parse(d)
```

```
      RemoveDuplicates(T)
```

```
      for all tokens e in T do
```

```
        if t not in I then
```

```
          I[t] = []
```

```
          I[t].append(Posting(n))
```

```
        end for
```

```
    end for
```

```
    SortAndWriteToDisk(I, name)
```

```
    I.empty()
```

```
  end while
```

# Merging

Index A	aardvark	2	3	4	5	apple	2	4
Index B	aardvark	6	9	actor	15	42	68	

# Merging

Index A	aardvark	2	3	4	5	apple	2	4
---------	----------	---	---	---	---	-------	---	---

Index B	aardvark	6	9	actor	15	42	68
---------	----------	---	---	-------	----	----	----

Index A	aardvark	2	3	4	5
---------	----------	---	---	---	---

Index B	aardvark					6	9
---------	----------	--	--	--	--	---	---

Combined index	aardvark	2	3	4	5	6	9
----------------	----------	---	---	---	---	---	---

# Merging

Index A     

aardvark	2	3	4	5	apple	2	4
----------	---	---	---	---	-------	---	---

Index B     

aardvark	6	9	actor	15	42	68
----------	---	---	-------	----	----	----

Index A     

aardvark	2	3	4	5
----------	---	---	---	---

Index B     

aardvark	6	9	actor	15	42	68
----------	---	---	-------	----	----	----

Combined index     

aardvark	2	3	4	5	6	9	actor	15	42	68
----------	---	---	---	---	---	---	-------	----	----	----

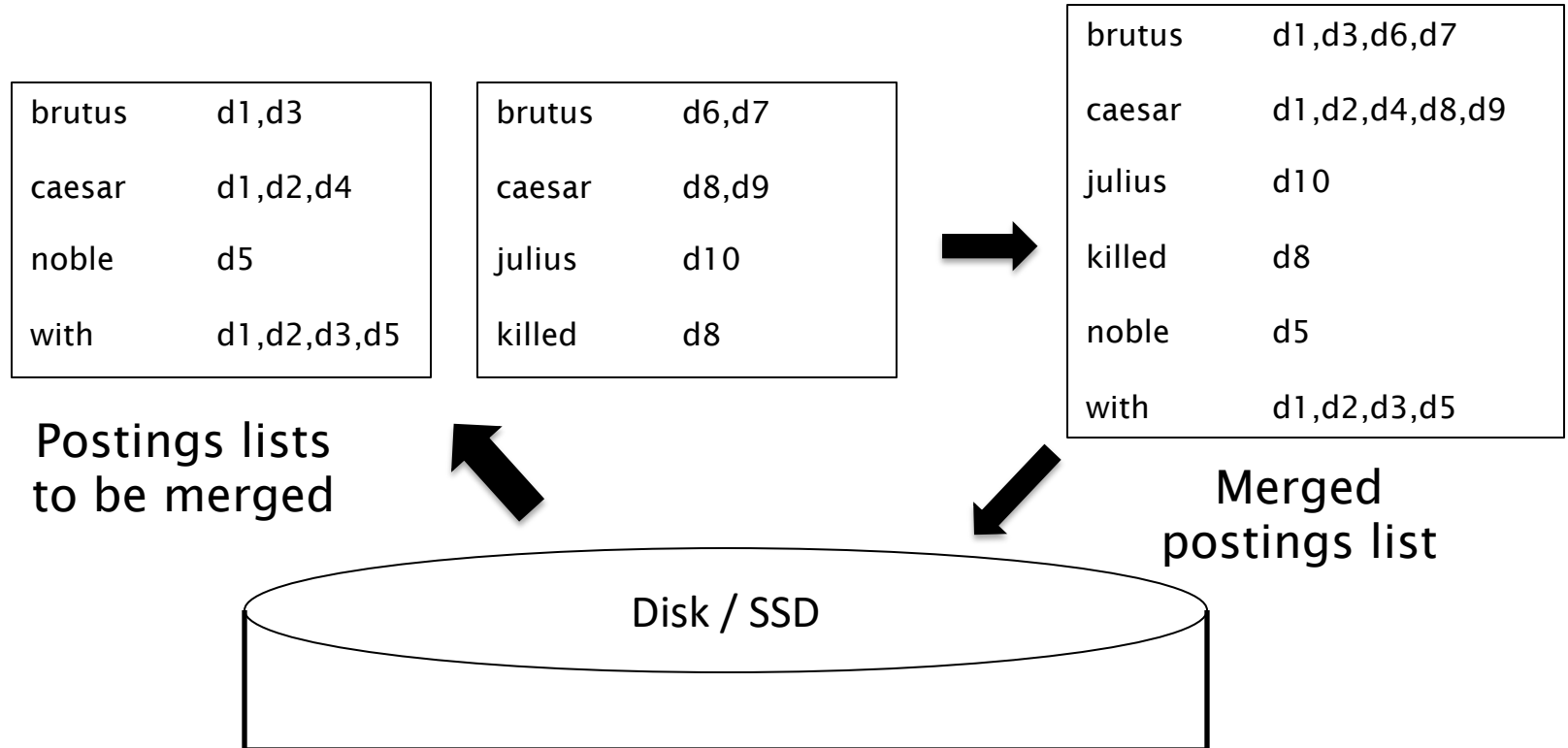
Index B	aardvark	6	9	actor	15	42	68
---------	----------	---	---	-------	----	----	----

apple	2	4
-------	---	---

Combined index	aardvark	2	3	4	5	6	9	actor	15	42	68	apple	2	4
----------------	----------	---	---	---	---	---	---	-------	----	----	----	-------	---	---

# How to merge the sorted runs?

- Can do binary merges, 2 files at a time
- During each layer, **read** into memory in blocks of a few MB (~10MB is not uncommon for HDDs: but you should profile!), **merge**, **write back**.





# How to merge the sorted runs?

---

- But it is more efficient to do a multi-way merge, where you are reading from all files simultaneously

# How to merge the sorted runs?

---

- But it is more efficient to do a multi-way merge, where you are reading from all files simultaneously
  - Open all partial index files simultaneously and maintain a read buffer for each one and a write buffer for the output file
  - In each iteration, pick the lowest termID that hasn't been processed
  - Merge all postings lists for that termID and write it out

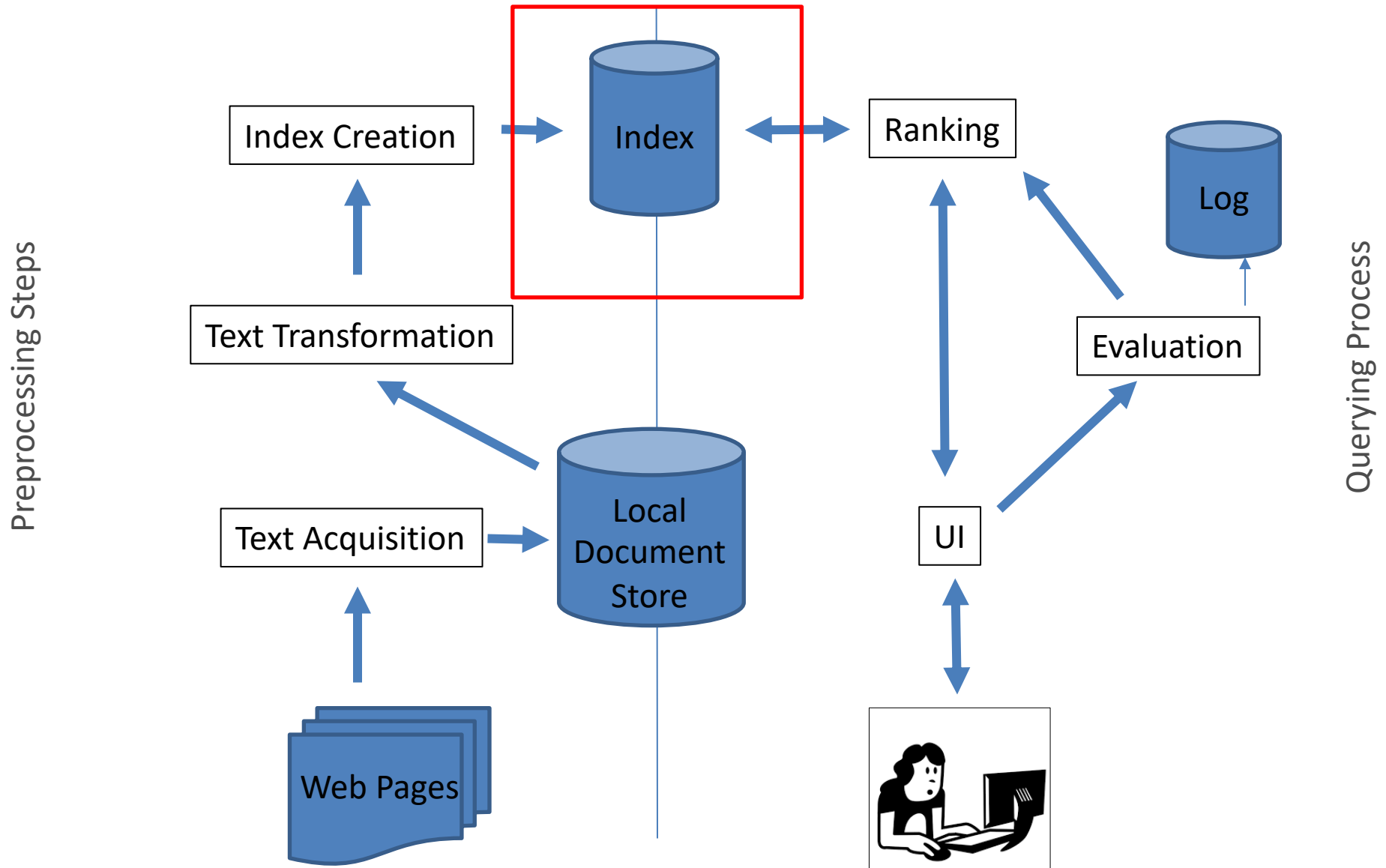
# How to merge the sorted runs?

- But it is more efficient to do a multi-way merge, where you are reading from all files simultaneously
  - Open all partial index files simultaneously and maintain a read buffer for each one and a write buffer for the output file
  - In each iteration, pick the lowest termID that hasn't been processed
  - Merge all postings lists for that termID and write it out
- Providing that you read decent-sized chunks of each block into memory and then write out a decent-sized output chunk, then you're not killed by disk seeks

# How to merge the sorted runs?


- But it is more efficient to do a multi-way merge, where you are reading from all files simultaneously
  - Open all partial index files simultaneously and maintain a read buffer for each one and a write buffer for the output file
  - In each iteration, pick the lowest termID that hasn't been processed
  - Merge all postings lists for that termID and write it out
- Providing that you read decent-sized chunks of each block into memory and then write out a decent-sized output chunk, then you're not killed by disk seeks
  - The actual size of the “decent-sized chunks” need to be actively probed as it is hardware and filesystem dependent.

# Architecture



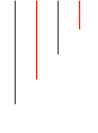
# The index we just built

---

- How do we process a query? 
  - Later – what kinds of complex queries can we process?

# Boolean queries: Exact match

- The **Boolean retrieval model** is being able to ask a query that is a **Boolean expression**:
  - Boolean Queries are queries using *AND*, *OR* and *NOT* to join query terms
    - Views each document as a set of words
    - Is precise: document matches condition or not.
  - Perhaps the simplest model to build an IR system on
- Primary commercial retrieval tool for 3 decades.
- Many search systems you use are Boolean:
  - Email, library catalog, macOS Spotlight

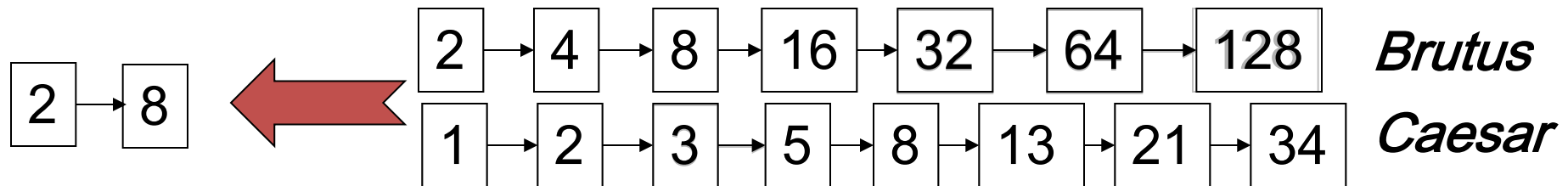


- Two possible outcomes for query processing
  - TRUE and FALSE
  - “exact-match” retrieval
  - simplest form of “ranking”
- Query usually specified using Boolean operators
  - AND, OR, NOT
  - proximity operators also used



# The merge

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

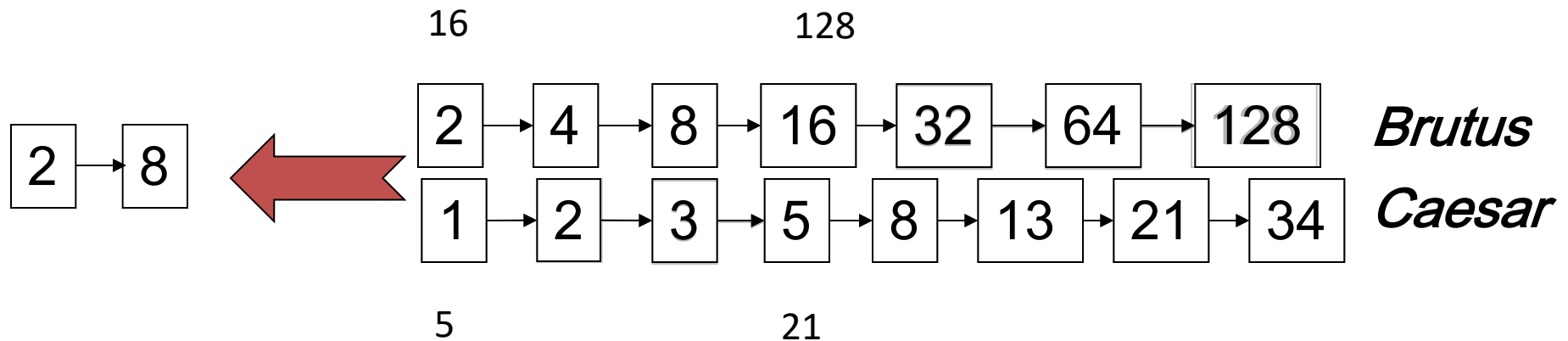


If the list lengths are  $x$  and  $y$ , the merge takes  $O(x+y)$  operations.

**Crucial: postings sorted by docID.**

# The merge: optimization

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

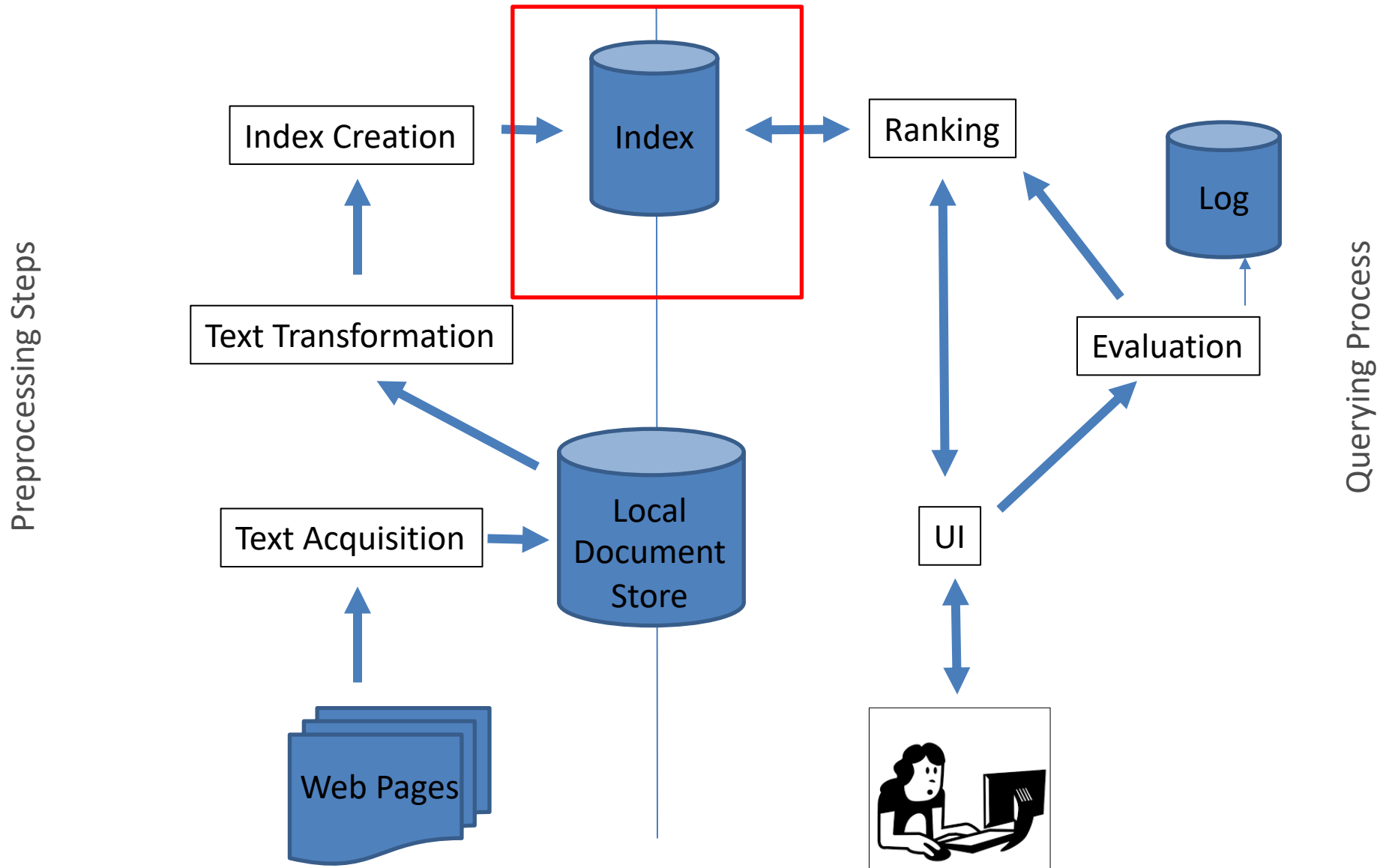


If the list lengths are  $x$  and  $y$ , the merge takes  $O(x+y)$  operations.

Crucial: postings sorted by docID.

You can also add skip pointers.

# Can you optimize also at QUERY TIME?



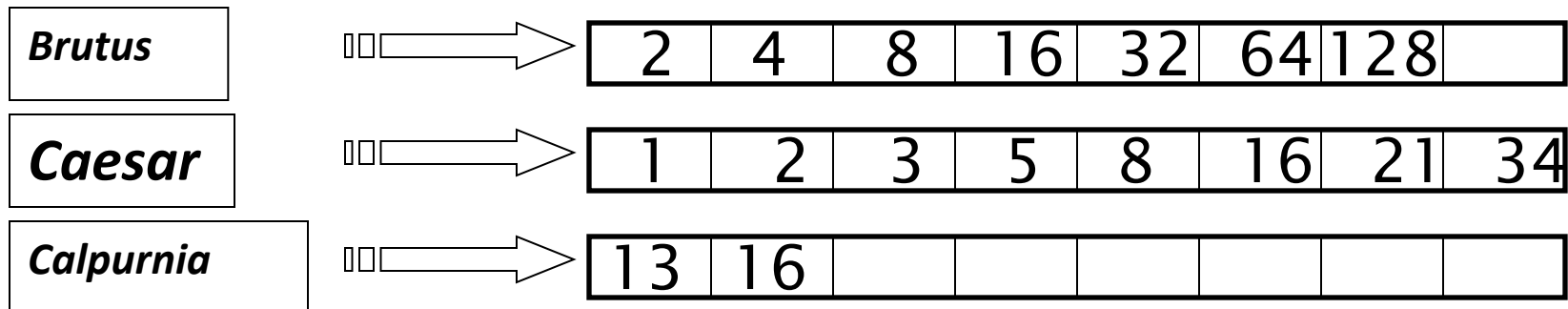
# Query optimization

---

- What is the best order for query processing?

# Query optimization

- What is the best order for query processing?
- Consider a query that is an *AND* of  $n$  terms.
- For each of the  $n$  terms, get its postings, then *AND* them together.



Query: **Brutus AND Calpurnia AND Caesar**

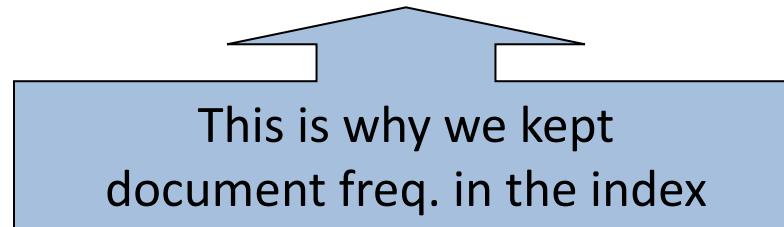
# Query optimization example

---

- Process in order of increasing freq:
  - *start with smallest set, then keep cutting further.*

# Query optimization example

- Process in order of increasing freq:
  - *start with smallest set, then keep cutting further.*



# Query optimization example

- Process in order of increasing freq:
  - *start with smallest set, then keep cutting further.*

This is why we kept  
document freq. in the index

<b>Brutus</b>	→	2	4	8	16	32	64	128	
<b>Caesar</b>	→	1	2	3	5	8	16	21	34
<b>Calpurnia</b>	→	13	16						

Execute the query as (***Calpurnia AND Brutus***) ***AND Caesar***.



# Query: Phrase queries

- We want to be able to answer queries such as “*university of california*” – as a phrase
- Thus the sentence “*I went to university in california*” is **not** a match.
  - The concept of phrase queries has proven easily understood by users; one of the few “advanced search” ideas that works
  - Many more queries are *implicit phrase queries*
- For this, it no longer suffices to store only *<term : docs>* entries

# A first attempt: Biword/bigrams indexes

- Index every consecutive pair of terms in the text as a phrase
- For example the text “Friends, Romans, Countrymen” would generate the biwords (= bi-grams!)
  - *friends romans*
  - *romans countrymen*
- Each of these biwords/bigrams is now a dictionary term :
  - All bigrams are indexed
- Two-word phrase query-processing is now immediate.
- Adopted for small n. but **unfeasible solution for arbitrary sizes.**

# Longer phrase queries

---

- Longer phrases can be processed by breaking them down
- ***University of California Irvine*** can be broken into the Boolean query on bigrams:

***university of AND of california AND california irvine***

# Longer phrase queries

- Longer phrases can be processed by breaking them down
- *University of California Irvine* can be broken into the Boolean query on bigrams:

*university of* AND *of california* AND *california irvine*

**Without the docs**, we cannot verify that the docs matching the above Boolean query do contain the continuous phrase.



Can have false positives!

# Issues for bigram indexes usage in phrases

- False positives, as noted before
- Possible index blowup due to bigger dictionary
  - Infeasible for more than biwords, big even for all of them
  - But very large scale search engines can support some n-grams for small n
- Bigram indexes are not the standard solution (for all possible bigrams) but they can be part of a compound strategy

# Solution 2: Positional indexes

- In the postings, store, for each ***term*** the position(s) in which tokens of it appear:

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

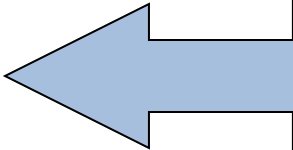
*doc1*: position1, position2 ... ;

*doc2*: position1, position2 ... ;

etc.>

# Positional index example

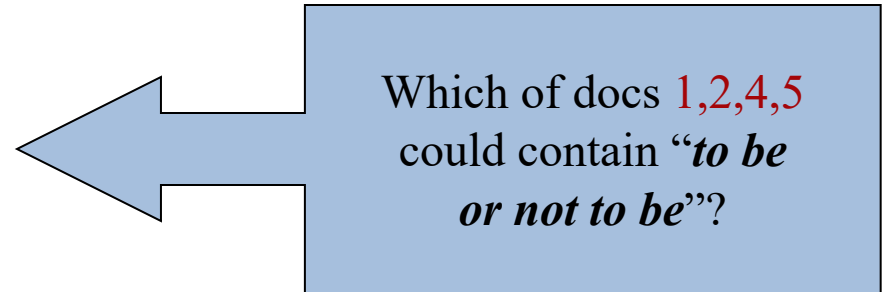
<*be*: 993427;  
*1*: 7, 18, 34, 72, 86, 231;  
*2*: 3, 149;  
*4*: 17, 191, 291, 430, 434;  
*5*: 363, 367, ...>



Which of docs *1,2,4,5*  
could contain “*to be*  
*or not to be*”?

# Positional index example

<*be*: 993427;  
*1*: 7, 18, 34, 72, 86, 231;  
*2*: 3, 149;  
*4*: 17, 191, 291, 430, 434;  
*5*: 363, 367, ...>

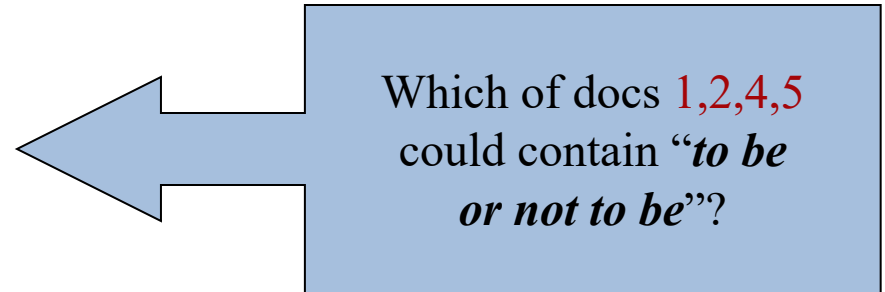


- For phrase queries, we use a **merge algorithm recursively** at the document level



# Positional index example

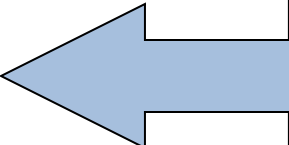
<*be*: 993427;  
*1*: 7, 18, 34, 72, 86, 231;  
*2*: 3, 149;  
*4*: 17, 191, 291, 430, 434;  
*5*: 363, 367, ...>



- For phrase queries, we use a **merge algorithm recursively** at the document level
- **Pre-clean the list to keep only documents that respect the distance** between words that appear more than once in the query (e.g. distance between two "be"s).

# Positional index example

<*be*: 993427;  
*1*: 7, 18, 34, 72, 86, 231;  
*2*: 3, 149;  
*4*: 17, 191, 291, 430, 434;  
*5*: 363, 367, ...>



Which of docs *1,2,4,5*  
could contain “*to be*  
*or not to be*”?

- For phrase queries, we use a **merge algorithm recursively** at the document level
- **Pre-clean the list to keep only documents that respect the distance** between words that appear more than once in the query (e.g. distance between two “be”s).
- But we now need to deal with **more than just equality**

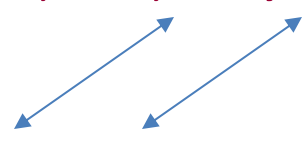
# Processing a phrase query

---

- Extract inverted index entries for each distinct term: *to*, *be*, *or*, *not*.

# Processing a phrase query

- Extract inverted index entries for each distinct term: *to*, *be*, *or*, *not*.
- Merge their *doc:position* lists to enumerate all positions with “*to be or not to be*”.
  - *to*:
    - 2:1,17,74,222,551; 4:8,16,190,429,433; 7:13,23,191; ...
  - *be*:
    - 1:17,19; 4:17,191,291,430,434; 5:14,19,101; ...
- Same general method for proximity searches



# Positional index size

---

- A positional index expands postings storage *substantially*
  - Even though indices can be compressed

# Positional index size

---

- A positional index expands postings storage *substantially*
  - Even though indices can be compressed
- Nevertheless, a **positional index is now standardly used** because of the power and usefulness of phrase and proximity queries whether used **explicitly or implicitly** in a ranking retrieval system.

# Positional index size

---

- Need an entry for each occurrence, not just once per document
- Index size depends on average document size
  - Average web page has <1000 terms
  - SEC filings, books, even some epic poems ... easily 100,000 terms

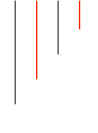
# Rules of thumb

---

- A compressed positional index is 2–4 as large as a non-positional index
- Positional index size 35–50% of volume of original text
  - *Caveat: these numbers hold for “English-like” languages*



- N-grams and positional approaches can be profitably combined
  - For particular phrases (“*Michael Jackson*”, “*Britney Spears*”) it is inefficient to keep on merging positional postings lists
    - Even more so for phrases like “*The Who*”
- More sophisticated mixed indexing scheme (e.g. Williams et al., 2004)
  - A typical web query mixture was executed in  $\frac{1}{4}$  of the time of using just a positional index
  - And it requires 26% more space than having a positional index alone



- **Advantages**

- Results are predictable, relatively easy to explain
- Many different features can be incorporated
- Efficient processing since many documents can be eliminated from search

- **Disadvantages**

- Effectiveness depends entirely on user
- Simple queries usually don't work well
- Complex queries are difficult