# Indexing and Query Processing (Inverted Index)

## CS4422/7263 Information Retrieval
## Lecture 03

Jiho Noh

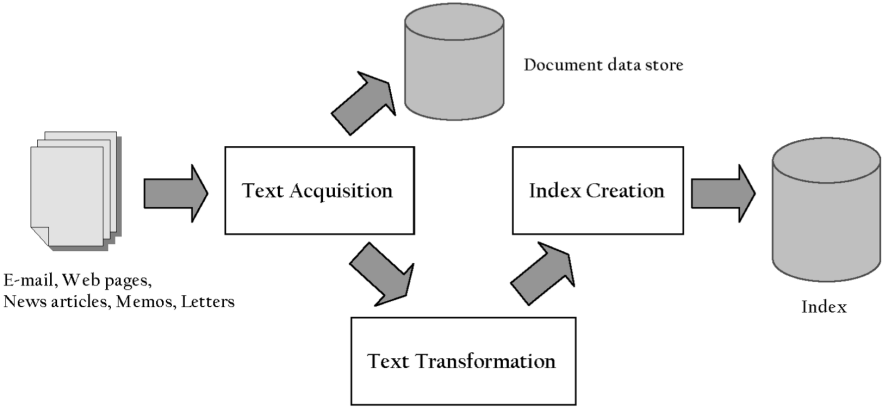
Department of Computer Science
Kennesaw State University

CS4422/7263 Spring 2025

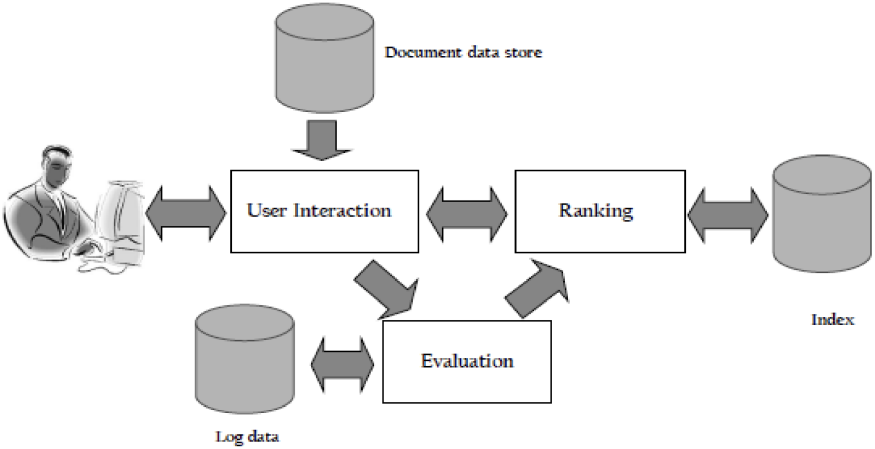
1 **Indexing Process**
- Motivation
- Inverted Index
- Index Compression
- Index Construction

2 **Query Processing**

# The Indexing Process



# The Querying Process



# Where are we now?

- We **crawled** webpages from the Web
- We **extracted** documents (body text) from the webpages
- We **tokenized and normalized** the documents
- Now, we can **represent** a document as a Bag-of-Words

Query: "What are the side effects of COVID-19 AstraZeneca vaccines?"

A bag of words with their frequencies

{side: 1, effects: 1, COVID-19: 1, AstraZeneca: 1, vaccines: 1}

*Vocabulary*

```
0: <UNK>
1: the
2: a
…
267: effects
…
347: side
…
```

word mapped to its index in the pre-defined vocabulary

{347: 1, 267: 1, 1657: 1, 2110: 1, 943: 1}

# Document-Term Matrix

V

D

| | a | able | about | academic | access | according | account | ... |
|---|---|---|---|---|---|---|---|---|
| D1 | 11 | 0 | 2 | 0 | 0 | 0 | 0 | |
| D2 | 5 | 0 | 2 | 0 | 0 | 0 | 1 | |
| D3 | 12 | 2 | 0 | 0 | 0 | 0 | 0 | |
| D4 | 4 | 0 | 0 | 0 | 3 | 0 | 0 | |
| D5 | 24 | 2 | 2 | 2 | 2 | 1 | 0 | |
| ... | | | | | | | | |

Query: "academic success"

# Vectore Representations for Documents

- A document is represented as a vector of term frequencies

$$d = (4, 0, 0, 0, 3, 0, \ldots, tf_{|V|})$$

- A query can also be represented as a vector of term frequencies

$$q = (0, 0, 0, 1, 0, 0, \ldots, tf_{|V|})$$

- The matching terms between the query and the documents are used to compute a similarity score
- *dot product* is a common similarity measure

$$sim(q, d) = q \cdot d = \sum_{i=1}^{|V|} q_i \cdot d_i$$

# Complexity Analysis (Time)

```
1  rel_docs = list()
2  for qt in q:
3      for d in C:
4          for w in d:
5              if qt == w:
6                  rel_docs.append(d)
7  return rel_docs
```

- $O(q \times |D| \times l)$
  - where $q$ is the average length of queries, and
  - $l$ is the average length of documents
- line 3 is the bottleneck, where $|D|$ can be huge

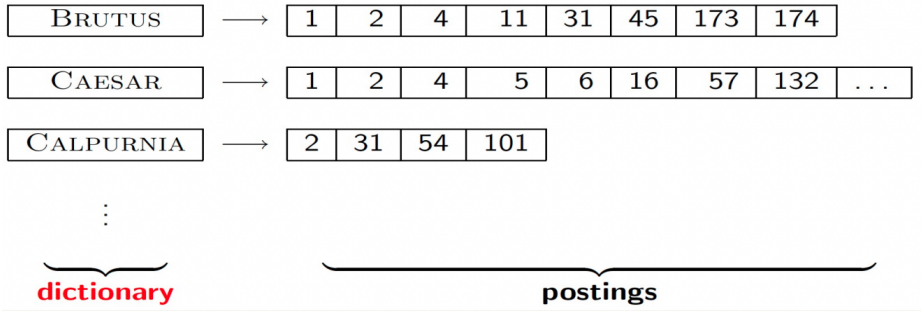
# Complexity Analysis (Space)

- $O(|D| \times |V|)$
  - where V is the vocabulary size
- Zipf's law: Each document contains only a small fraction of the vocabulary
- Space efficiency can be greatly improved by storing only the non-zero entries of the document-term matrix
- Any Solution??

1 **Indexing Process**
- Motivation
- Inverted Index
- Index Compression
- Index Construction

2 **Query Processing**

# Solution: Use Inverted Index




- **Inverted Index** is the term-to-documents mapping
- **Dictionary** is the data structure for storing the term vocabulary
- **Posting** is a list that records which documents the term occur in

# Library Catalog

**Figure:** The card catalog in a library

# Library Catalog

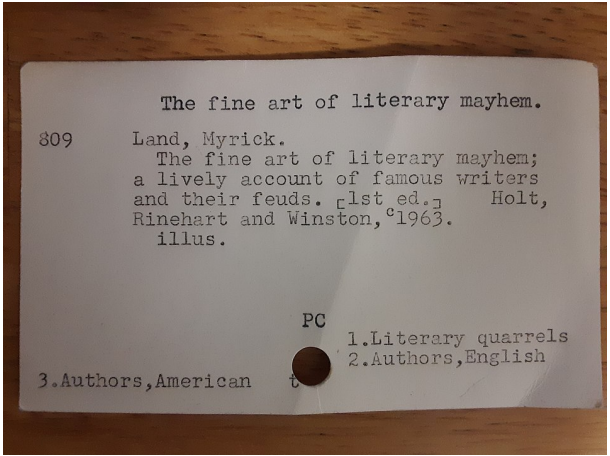



**Figure:** An title index card

# Example Document Collection

$S_1$ Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

$S_2$ Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

$S_3$ Tropical fish are popular aquarium fish, due to their often bright coloration.

$S_4$ In freshwater fish, the coloration typically derives from iridescence, while salt water fish are generally pigmented.

Four sentences from the Wikipedia entry for *tropical fish*

# Forward Index

$S_1$ tropical, fish, include, found, in, environments, around, the, world, ...
$S_2$ fishkeepers, often, use, the, term, tropical, to, fish, refer, only, ...
$S_3$ tropical, fish, are, popular, aquarium, fish, due, to, their, often, ...
$S_4$ in, freshwater, fish, the, coloration, typically, derives, from, iridescence, ...

documents (sentences) and their terms

- **Inverted-Index**, an inversion of the forward index
- Simple inverted index for the documents (sentences)

| | |
|---|---|
| and | 1 |
| aquarium | 3 |
| are | 3 4 |
| around | 1 |
| as | 2 |
| both | 1 |
| bright | 3 |
| coloration | 3 4 |
| derives | 4 |
| due | 3 |
| environments | 1 |
| fish | 1 2 3 4 |
| fishkeepers | 2 |
| found | 1 |
| fresh | 2 |
| freshwater | 1 4 |
| from | 4 |
| generally | 4 |
| in | 1 4 |
| include | 1 |
| including | 1 |
| iridescence | 4 |
| marine | 2 |
| often | 2 3 |
| only | 2 |
| pigmented | 4 |
| popular | 3 |
| refer | 2 |
| referred | 2 |
| requiring | 2 |
| salt | 1 4 |
| saltwater | 2 |
| species | 1 |
| term | 2 |
| the | 1 2 |
| their | 3 |
| this | 4 |
| those | 2 |
| to | 2 3 |
| tropical | 1 2 3 |
| typically | 4 |
| use | 2 |
| water | 1 2 4 |
| while | 4 |
| with | 2 |
| world | 1 |

- Inverted index with counts (**term frequencies**) supports better ranking algorithms

| | | | | |
|---|---|---|---|---|
| and | 1:1 | | | |
| aquarium | 3:1 | | | |
| are | 3:1 | 4:1 | | |
| around | 1:1 | | | |
| as | 2:1 | | | |
| both | 1:1 | | | |
| bright | 3:1 | | | |
| coloration | 3:1 | 4:1 | | |
| derives | 4:1 | | | |
| due | 3:1 | | | |
| environments | 1:1 | | | |
| fish | 1:2 | 2:3 | 3:2 | 4:2 |
| fishkeepers | 2:1 | | | |
| found | 1:1 | | | |
| fresh | 2:1 | | | |
| freshwater | 1:1 | 4:1 | | |
| from | 4:1 | | | |
| generally | 4:1 | | | |
| in | 1:1 | 4:1 | | |
| include | 1:1 | | | |
| including | 1:1 | | | |
| iridescence | 4:1 | | | |
| marine | 2:1 | | | |
| often | 2:1 | 3:1 | | |

| | | | |
|---|---|---|---|
| only | 2:1 | | |
| pigmented | 4:1 | | |
| popular | 3:1 | | |
| refer | 2:1 | | |
| referred | 2:1 | | |
| requiring | 2:1 | | |
| salt | 1:1 | 4:1 | |
| saltwater | 2:1 | | |
| species | 1:1 | | |
| term | 2:1 | | |
| the | 1:1 | 2:1 | |
| their | 3:1 | | |
| this | 4:1 | | |
| those | 2:1 | | |
| to | 2:2 | 3:1 | |
| tropical | 1:2 | 2:2 | 3:1 |
| typically | 4:1 | | |
| use | 2:1 | | |
| water | 1:1 | 2:1 | 4:1 |
| while | 4:1 | | |
| with | 2:1 | | |
| world | 1:1 | | |

- term/document frequencies can be inferred
- more features for better ranking

| | | | | | |
|---|---|---|---|---|---|
| and | 1,15 | | | | |
| aquarium | 3,5 | | | | |
| are | 3,3 | 4,14 | | | |
| around | 1,9 | | | | |
| as | 2,21 | | | | |
| both | 1,13 | | | | |
| bright | 3,11 | | | | |
| coloration | 3,12 | 4,5 | | | |
| derives | 4,7 | | | | |
| due | 3,7 | | | | |
| environments | 1,8 | | | | |
| fish | 1,2 | 1,4 | 2,7 | 2,18 | 2,23 |
| | | | 3,2 | 3,6 | 4,3 |
| | | | 4,13 | | |
| fishkeepers | 2,1 | | | | |
| found | 1,5 | | | | |
| fresh | 2,13 | | | | |
| freshwater | 1,14 | 4,2 | | | |
| from | 4,8 | | | | |
| generally | 4,15 | | | | |
| in | 1,6 | 4,1 | | | |
| include | 1,3 | | | | |
| including | 1,12 | | | | |
| iridescence | 4,9 | | | | |

| | | | | | |
|---|---|---|---|---|---|
| marine | 2,22 | | | | |
| often | 2,2 | 3,10 | | | |
| only | 2,10 | | | | |
| pigmented | 4,16 | | | | |
| popular | 3,4 | | | | |
| refer | 2,9 | | | | |
| referred | 2,19 | | | | |
| requiring | 2,12 | | | | |
| salt | 1,16 | 4,11 | | | |
| saltwater | 2,16 | | | | |
| species | 1,18 | | | | |
| term | 2,5 | | | | |
| the | 1,10 | 2,4 | | | |
| their | 3,9 | | | | |
| this | 4,4 | | | | |
| those | 2,11 | | | | |
| to | 2,8 | 2,20 | 3,8 | | |
| tropical | 1,1 | 1,7 | 2,6 | 2,17 | |
| typically | 4,6 | | | | |
| use | 2,3 | | | | |
| water | 1,17 | 2,14 | 4,12 | | |
| while | 4,10 | | | | |
| with | 2,15 | | | | |
| world | 1,11 | | | | |

# Data Structure — Dictionary

- Modest size
- Stay in memory
- Choices of data structure: **Hash, B-Tree, B+Tree...**
- Some IR systems use hashes, some trees
- Criteria in choosing *hash* or *search tree*
  - Fixed number of terms or keep growing
  - Relative frequencies with which various keys are accessed
  - How many terms

# Data Structure — Postings

- Sequential access is expected
- Stay on disk
- Information to store: docID, term frequency, positions, ...
- Likely to be huge: Compression is needed

# Hashes

- Each vocabulary term is hashed to a unique integer
- **Pros**:
  - Lookup is fast: $\mathcal{O}(1)$
- **Cons**:
  - No easy way to find all terms starting with prefix (e.g., 'automat')
  - Need to rehash when the dictionary grows

# Search Trees — Binary Tree

- Binary tree (simple), B-tree, B+tree (more usual)
- **Pros**:
  - Solves the prefix problem (finding all terms starting with 'automat')
- **Cons**:
  - Slower: $\mathcal{O}(\log M)$ [and this requires balanced tree]
    - $M$ is called the order of the tree, that is the number of allowed children per node
  - Rebalancing binary trees is expensive
  - But B+trees mitigate the rebalancing problem

# B+tree[1]

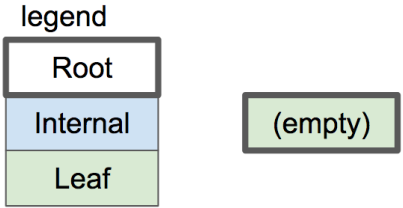
- *B+tree* is a good data structure for indexes:
  - Searching for a particular value is fast (logarithmic time)
  - Inserting / deleting a value you've already found is fast
  - Traversing a range of values is fast (unlike a hash map)
- Unlike a binary tree, each node in a B+Tree can have more than 2 children.
- Each node can have up to $M$ children

# B+Tree Properties

- Every node has at most $M$ children.
- Every (internal) node, except for the root and the leaves, has at least $\lceil M/2 \rceil$ children.
- The root node has at least two children unless it is a leaf.
- All leaves appear on the same level.
- A non-leaf node with $k$ children contains $k - 1$ keys.

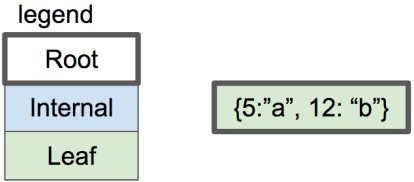
# B+tree example

- Let's take an example to see how a B-tree grows. To keep things simple, the tree will be order 3 ($M = 3$). That means:
- An empty B+tree has a single node: the root node.
- The root node starts as a leaf node with zero key/value pairs:



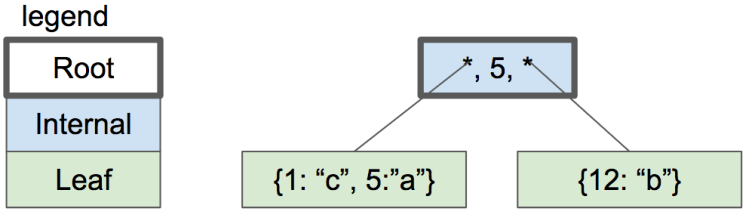
# B+tree example

- If we insert a couple key/value pairs, they are stored in the leaf node in sorted order.
- Two items are added. Let's insert a new item (1: "c"), where 1 is the key and "c" is the value.



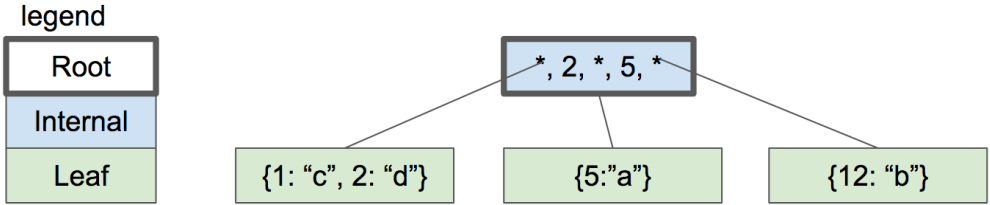
# B+tree example

- When we insert another, we have to split the leaf node and put half the pairs in each node.
- Let's insert a new item (2: "d")



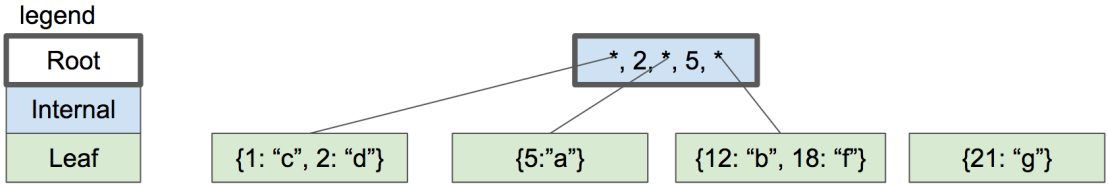
# B+tree example

- Now let's insert the key 2. First we look up which leaf node it would be in if it was present, and we arrive at the left leaf node. The node is full, so we split the leaf node and create a new entry in the parent node.
- Now, we are adding items with keys 18 and 21.



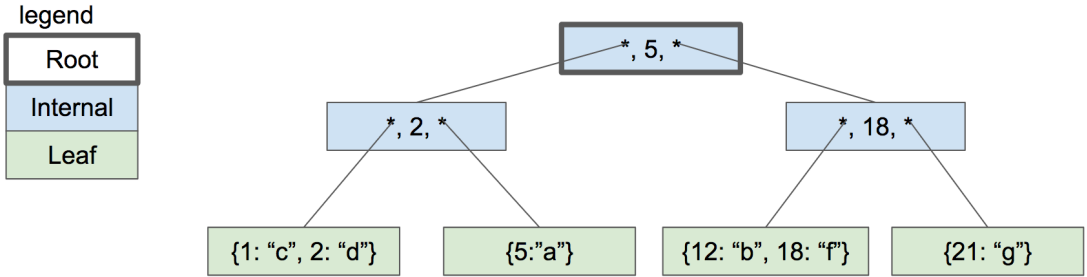
# B+tree example

- We get to the point where we have to split again, but there's no room in the parent node for another key/pointer pair.



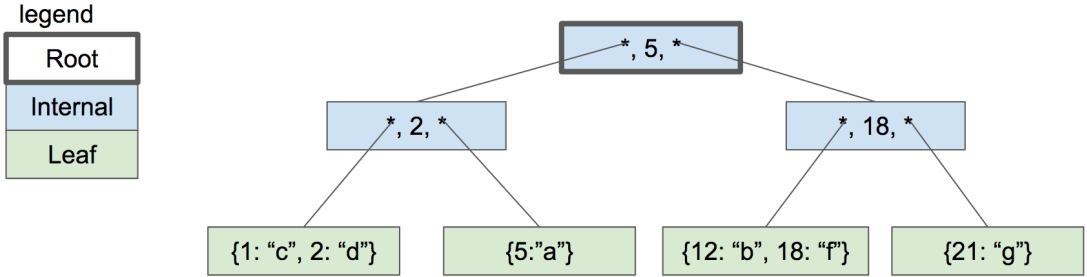
# B+tree example

- The solution is to split the root node into two internal nodes, then create new root node to be their parent.



# B+tree example

- The depth of the tree only increases when we split the root node.
- Every leaf node has the same depth and close to the same number of key/value pairs, so the tree remains balanced and quick to search.



1 **Indexing Process**
- Motivation
- Inverted Index
- Index Compression
- Index Construction

2 **Query Processing**

# Reuters RCV1 Statistics

| Statistics | Value |
|---|---:|
| documents (N) | 800,000 |
| avg. # tokens per doc (L) | 200 |
| vocabulary size (M) (# terms) | 400,000 |
| avg. # bytes per token (incl. spaces/punct.) | 6 |
| avg. # bytes per token (without spaces/punct.) | 4.5 |
| avg. # bytes per term | 7.5 |
| non-positional postings | 100,000,000 |

# Reuters RCV1 Statistics (Cont.)

| | Dictionary | Non-positional Index | Positional Index |
|---|---|---|---|
| Unfiltered | 484,494 | 109,971,179 | 197,879,290 |
| Numbers removed | 473,723 | 100,680,242 | 179,158,204 |
| Case folding | 39,523 | 96,969,056 | 179,158,204 |
| Stopwords removed (30) | 391,493 | 83,390,443 | 121,857,825 |
| Stopwords removed (50) | 391,373 | 67,001,847 | 94,516,599 |
| Stemming | 322,383 | 63,812,300 | 94,516,599 |

Size in tokens (= number of position entries in postings)

# Index Compression

- The Postings file is LARGE
  - For the most common term 'the', the posting likely to have $|D|$ entries
- We need to store each posting compactly
- For Reuters ($|D| = 800,000$), we would use 32 bits per docID when using 4-byte integers
- Alternatively, we can use $\log_2 800,000 \approx 20$ bits per docID.
- *Our goal is to use far fewer than 20 bits per docID*

# Posting Compression — Intuition

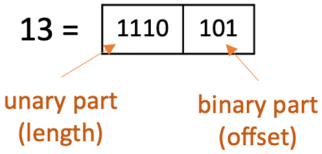
- Posting is a docID list in ascending order
- Instead of using docID, we can store the gaps between docIDs
- Zipf's law:
  - More frequent word: a long list with smaller gaps
  - Less frequent word: a short list with longer gaps
- Use fewer bits for encoding small (high frequency) integers
  - **Variable-length coding**
    - Unary code
    - Gamma $\gamma$-code
    - Delta $\delta$-code

# Unary Code

- Represent $n$ as $n$ 1s with a final 0.
- Unary code for 3 is 1110.
- Unary code for 40 is 11111111111111111111111111111111111111110.
- This doesn't look promising, but...
  - Optimal if $P(n) = 2^{-n}$

# Gamma Code

- Bit-level codes
- Represent a gap G as a pair of length and offset
- Offset is G in binary, with the leading bit cut off
  - For example, 13 →1101 →101
- Length is the length of offset
  - For 13 (offset is 101), the length is 3
- *Why do we need a length code?*
- *Why is it okay to remove the leading bit?*



# Gamma Code (Cont.)

- How many bits are needed to encode a gap $G$?
  - To encode the offset: $\lfloor \log_2 G \rfloor$ bits
  - To encode the length: $\lfloor \log_2 G \rfloor + 1$ bits
  - Total: $2 \times \lfloor \log_2 G \rfloor + 1$ bits
- That implies, the coding scheme is optimal for $P(n) \approx \frac{1}{2n^2}$
- Cons:
  - Bit-level; machines have word boundaries
  - Schemes using byte or word aligned code are favorable

# Variable Byte (VB) Code

- Begin with one byte to store G and dedicate 1 bit in it to be a continuation bit $c$ (*Similar to the Unicode coding scheme*)
- If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
- Else encode G's lower-order 7 bits and then use 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

| **docIDs** | **824** | **829** | **215406** |
|---|---|---|---|
| gaps | - | 5 | 214577 |
| VB code (binary) | 00000101<br>00000000 | 10000101 | 00001101<br>00001100<br>10110001 |
| VB code (hexadecimal) | 06 B8 | 85 | 0D 0C D1 |

# Reuters RCV1 Statstics

| Statistics | Value |
|---|---|
| documents (N) | 800,000 |
| avg. # tokens per doc (L) | 200 |
| vocabulary size (M) (# terms) | 400,000 |
| avg. # bytes per token (incl. spaces/punct.) | 6 |
| avg. # bytes per token (without spaces/punct.) | 4.5 |
| avg. # bytes per term | 7.5 |
| non-positional postings | 100,000,000 |
| postings, uncompressed (32-bit words) | 400 MB |
| postings, uncompressed (20-bits $= \log_2 1M$) | 250 MB |
| postings, Gamma encoded | 101 MB |
| postings, variable byte encoded | 116 MB |

# Delta Encoding

- Delta encoding is a way of storing or transmitting data in the form of **differences (deltas)** between sequential data.
- **Assuming that the docids, positions, etc. in our lists are *small and repetitive*,**
- We can utilize the encoding schemes with differences between the numbers to reduce the index size.

# Byte-aligned Code

- In production systems, inverted lists are stored using byte-aligned codes for delta-encoded integer sequences.
- For better computing performance, our index lists can be encoded using byte-aligned code schemes (e.g., VB).

# Putting it All Together

- Let's compress an inverted index postings with delta encoding.
- The raw postings contain a sequence of tuples as `(docid, tf, [pos1, pos2, ...])`
  `(1,2,[1,7]), (2,3,[6,17,197]), (3,1,[1])`
- We **delta-encode** the docid and position sequences independently.
  `(1,2,[1,6]), (1,3,[6,11,180]), (1,1,[1])`
- Finally, we encode the integers using VByte.
  `81 82 81 86 81 83 86 8B 01 B4 81 81 81`

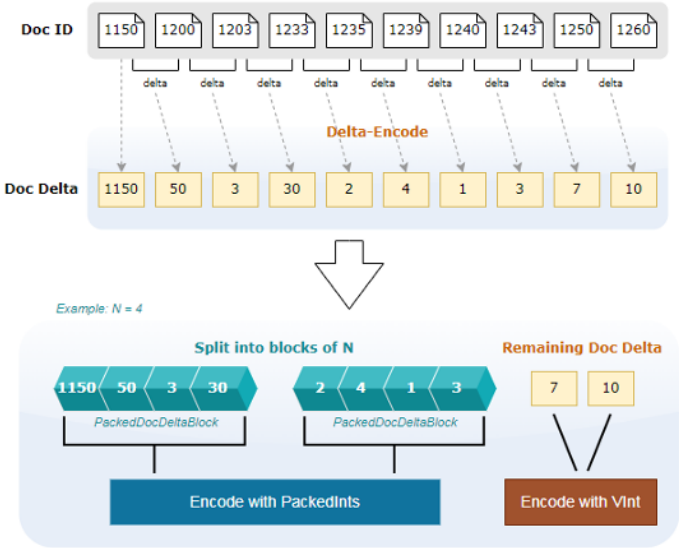



**Figure:** Lucene's compression method utilizing PackedDocDeltaBlock of 128 deltas. In this figure N=4 for simplicity

1 **Indexing Process**
- Motivation
- Inverted Index
- Index Compression
- Index Construction

2 **Query Processing**

# Challenges of Indexing

**Index construction (or indexing)** is the process of building an inverted index from a corpus.

1. Document (postings) size exceeds the memory limit
   - Say, one posting entry (termID, docID) requires 8 bytes; $100,000,000 \times 8/1024^2$ is near 800 Mb
   - Typical document collections are much larger
2. While indexing, we parse docs one at a time; final postings for a term are incomplete until the end.

We need to store intermediate results on disk.

# Simple Indexer

```
def build_index(D):      # D is a document collection
  I = HashTable()        # inveted list storage
  docID = 0
  for d in Dd:
    docID += 1           # consecutive integers for docID
    T = parse_document(d)  # prase document into tokens
    remove_duplicates(T)
    for t in T:
      if t not in I:
        I_t = []         # Create a new postings list
      I_t.append(docID) # Otherwise, add document to the postings list
  return I               # Return inverted index
```

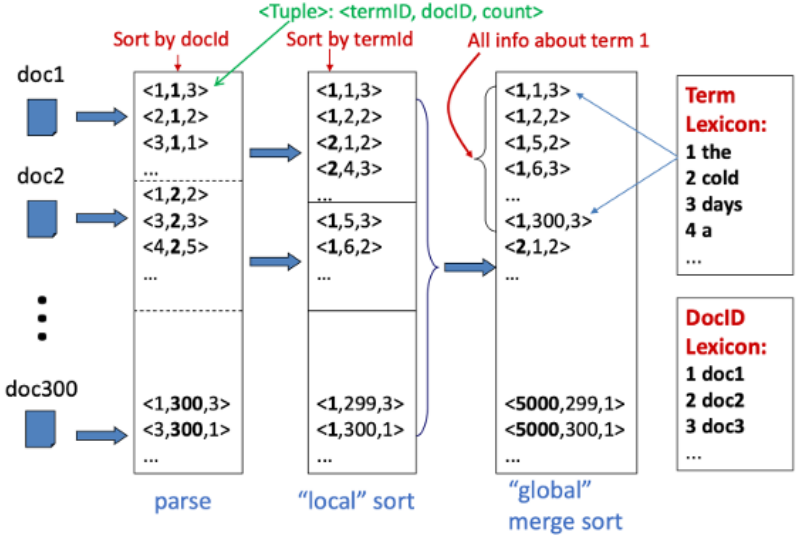
# Sort using disk as "memory"?

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

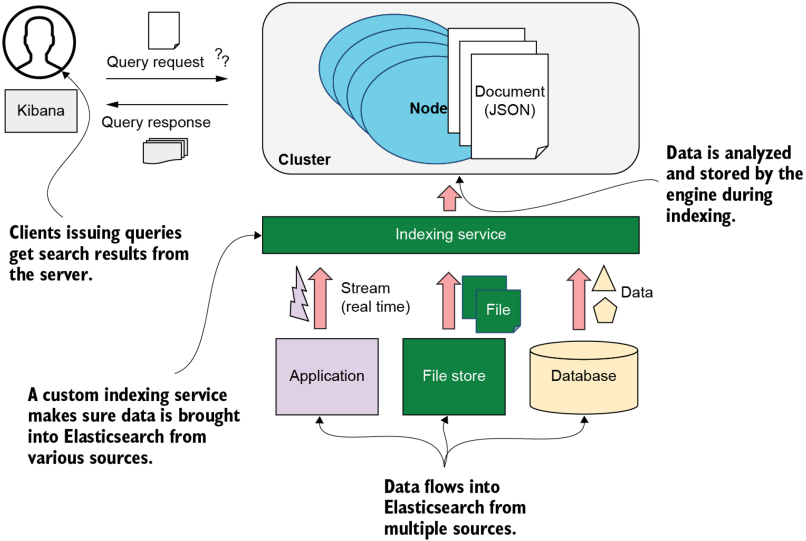
**External Sorting?**

**External Sorting** is a class of sorting algorithms that can handle massive amounts of data. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted subfiles are combined into a single larger file.

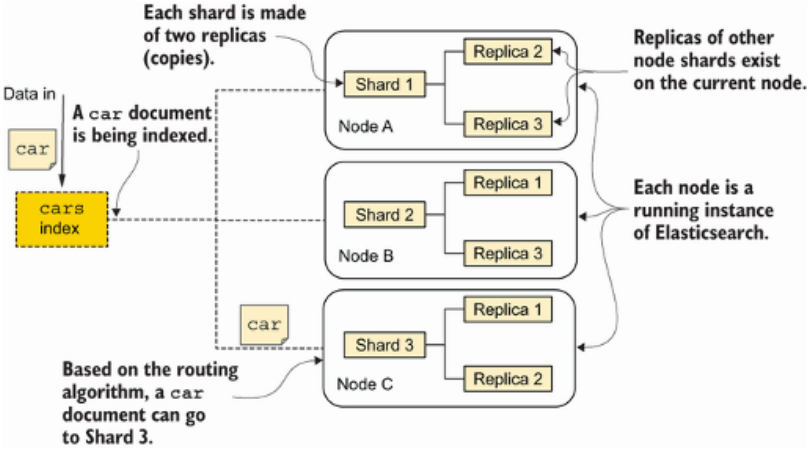
# Merge Sort


<Tuple>: <termID, docID, count>
Sort by docId
Sort by termId
All info about term 1
doc1
doc2
doc300
<1,1,3>
<2,1,2>
<3,1,1>
...
<1,2,2>
<3,2,3>
<4,2,5>
...
<1,300,3>
<3,300,1>
...
<1,1,3>
<1,2,2>
<2,1,2>
<2,4,3>
...
<1,5,3>
<1,6,2>
...
<1,299,3>
<1,300,1>
...
<1,1,3>
<1,2,2>
<1,5,2>
<1,6,3>
...
<1,300,3>
<2,1,2>
...
<5000,299,1>
<5000,300,1>
...
Term Lexicon:
1 the
2 cold
3 days
4 a
...
DocID Lexicon:
1 doc1
2 doc2
3 doc3
...
parse
"local" sort
"global" merge sort

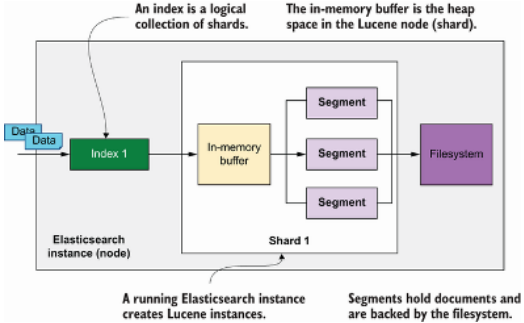
# ElasticSearch (ES) Indexing/Querying Pipeline

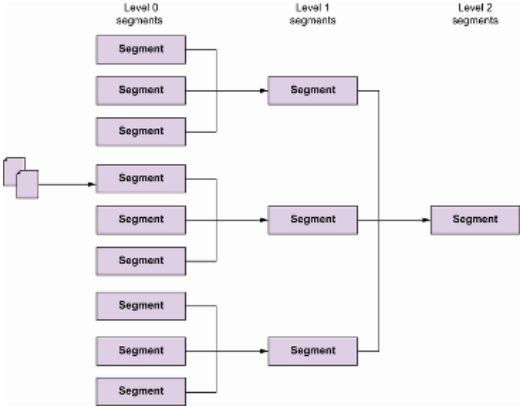


# ElasticSearch (ES) Indexes



# Mechanics of Indexing



# Lucene's Index

- Lucene index file formats

1 Indexing Process
- Motivation
- Inverted Index
- Index Compression
- Index Construction

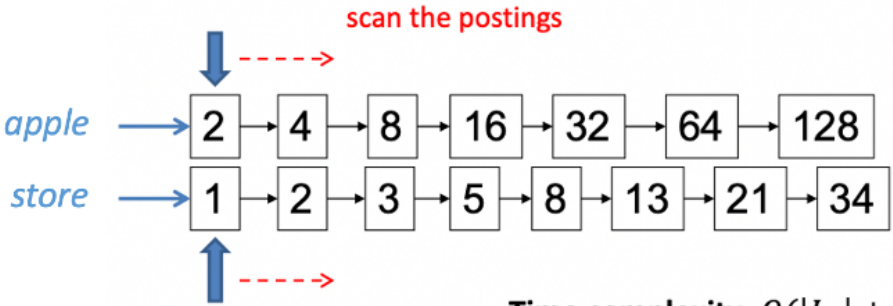
2 Query Processing

# Search with an inverted index

- Parse query syntax
  - e.g., `Apple AND store, orange OR apple`
- Perform the same text processing procedures as on documents
  - **tokenization** (e.g., John's →John + 's)
  - **normalization** (e.g., U.S.A. →usa)
  - **stemming** (e.g., beautiful →→ beauti)
  - **stopwords** (e.g., remove the, a, to, ...)

# Query, Conjunction

AND example: *apple AND store*



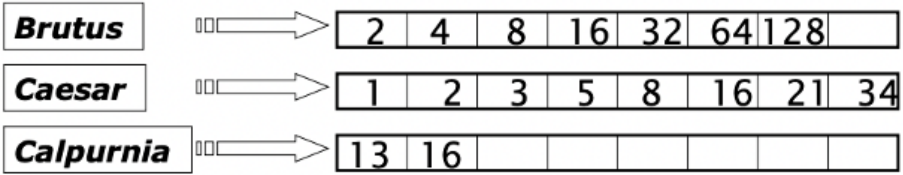

**Time complexity**: $O(|L_1| + |L_2|)$

# Query, Conjunction (pseudo code)

```
def intersect(pl, p2):
  answer = []
  while pl is not Null and p2 is not Null:
  if docID(p1) == docID(p2):
    answer.add(docID(p1))
    p1.next()
    p2.next( )
  else:
    if docID(p1) < docID(p2):
      p1.next( )
    else:
      p2.next( )
```

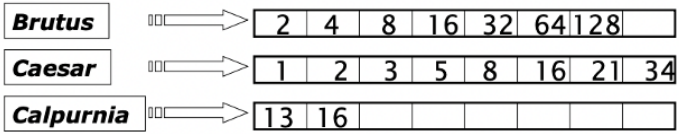
# Query Optimization

- Consider a query that is an "AND" of n terms, for example,
- **q:** `(Brutus AND Calpurnia AND Caesar)`




- For each of the n terms, get its postings, then "AND" them together
- **What is the best order for query processing?**

# Query Optimization (Cont.)




- Process in order of increasing term frequency (i.e., posting list lengths) can help increase the speed of merging posting lists.
- Start with the smallest set, then keep cutting further; This is why we keep document frequencies in our dictionary.

(**Brutus** AND **Calpurnia** AND **Caesar**)

*optimize* → (**Calpurnia** AND **Brutus**) AND **Caesar**

# Query Optimization — General Rules

- Query: `(london OR beijing) AND (train OR plane)`
- Get document frequencies for all terms
- Estimate the size of each OR by the sum of its doc. freq.'s (conservative)
- Process in increasing order of OR sizes.

# Phrase Queries

- For example, we want to be able to answer queries such as "`Apple Store`" as a phrase.
- Thus, the sentence "Which **store** should I go to buy an **apple**?" is not supposed to be a match.
- n-grams generally do not work for this.
- Large dictionary size, how to break long phrase into n-grams?
- For this, we need to store term positions in the postings lists.

# (proof of concept) Positional Indexes

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

```
<term, document frequency;
doc1: position1, position2, ... ;
doc2: position1, position2, ...>
```

# Positional indexes — example

```
<be, 993427;
1: 7, 18, 33, 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* (not the term level).
- But we now need to deal with more than just equality.

# Processing a Phrase Query

- Extract inverted index entries (i.e., postings) 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; ...
  - {to, be, or, not}: ... 4:8,16,17,190,191,291,429,430,431,432,433,434; ...
- If doc4 contains the phrase, you'll see a sequence like 429, 430, 431, 432, 433, 434 from the merged *doc:position* list.
- Generally, the same strategy can be used for proximity search.

# Processing a Phrase Query (Cont.)

- `(employment place)~3`
  - Here, ~k means "within k words of (on either side)"
  - Clearly, positional indexes can be used for such queries
- Equality condition: $|t_2.\text{pos} - t_1.\text{pos}| = 1$
  - $t_1$ must be immediately before or after $t_2$ in any matched document
- Inequality condition: $|t_2.\text{pos} - t_1.\text{pos}| \leq k$
  - $t_1$ and $t_2$ must be positioned within $k$ words

# Processing a Phrase Query (Cont.)

- 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.
- A positional index is 2–4 times as large as a non-positional index.
- Positional index size is 35–50% of the volume of the original text.

# Summary

- Questions?
- Discussion?