Information Retrieval

Index-Structures for Information Retrieval

Ulf Leser

Content of this Lecture

- Inverted files
- Storage structures
- Phrase and proximity search
- Building and updating the index
- Using a RDBMS

Full-Text Indexing

- Fundamental operation for all BoW approaches: find(q, D)
  - Given a term q, find all docs from D containing q
- Can be implemented using online search
  - Boyer-Moore, Keyword-Trees, etc.
- But
  - We search for terms (after tokenization), not arbitrary substrings
  - We assume that D is stable compared to q
  - The number of unique terms does not grow much with growing D
- Technique: Pre-compute a term index over D
  - "Full-text index" or "inverted file" or "inverted index"

Inverted Files (or Inverted Index)

- Simple and effective index structure for terms
- Builds on the Bag of words approach
  - We give up on order of terms in docs (reappears later)
  - We cannot reconstruct docs based on index only
- Start from "docs containing terms" (~ "docs") and invert to "terms appearing in docs" (~ "inverted docs")

```
d1: t1,t3
d2: t1
d3: t2,t3
d4: t1
d5: t1,t2,t3
d6: t1,t2
d7: t2
d8: t2
```

→

```
t1: d1,d2,d4,d5,d6
t2: d3,d5,d6,d7,d8
t3: d1,d3,d5
```

Building an Inverted File [Andreas Nürnberger, IR-2007]

Doc1:
Now is the time for all good men to come to the aid of their country.

Doc2:
It was a dark and stormy night in the country manor. The time was past midnight.

Term	Doc ID
now	1
is	1
the	1
time	1
for	1
all	1
good	1
men	1
to	1
come	1
to	1
the	1
aid	1
of	1
their	1
country	1
it	2
was	2
a	2
dark	2
and	2
stormy	2
night	2
in	2
the	2
country	2
manor	2
the	2
time	2
was	2
past	2
midnight	2

Sort

Term	Doc ID
a	2
aid	1
all	1
and	2
come	1
country	1
country	2
dark	2
for	1
good	1
in	2
is	1
it	2
manor	2
men	1
midnight	2
night	2
now	1
of	1
past	2
stormy	2
the	1
the	1
the	2
the	2
their	1
time	1
time	2
to	1
to	1
was	2
was	2

Merge

a	2
aid	1
all	1
and	2
come	1
country	1,2
dark	2
for	1
good	1
in	2
is	1
it	2
manor	2
men	1
midnight	2
night	2
now	1
of	1
past	2
stormy	2
the	1,2
their	1
time	1,2
to	2
was	2

Boolean Retrieval

- For each query term k_i, look-up doc-list D_i containing k_i
- Evaluate query in the usual order
  - $k_i \wedge k_j$: $D_i \cap D_j$
  - $k_i \vee k_j$: $D_i \cup D_j$
  - NOT k_i : $D \setminus D_i$
- Example

a	2
aid	1
all	1
and	2
come	1
country	1,2
dark	2
for	1
good	1
in	2
is	1
it	2
manor	2
men	1
midnight	2
night	2
now	1
of	1
past	2
stormy	2
the	1,2
their	1
time	1,2
to	2
was	2

(time AND past AND the) OR (men)

$$= (D_{time} \cap D_{past} \cap D_{the}) \cup D_{men}$$
$$= (\{1,2\} \cap \{2\} \cap \{1,2\}) \cup \{1\}$$
$$= \{1,2\}$$

Necessary and Obvious Tricks

- How do we efficiently look-up doc-list D_i?
  - Bin-search on sorted array with all terms from K: O(log(|K|))
  - How to do this if D and index is very large? (later)
- How do we support union and intersection efficiently?
  - Naïve algorithm requires $O(|D_i|*|D_j|)$
  - Better: Keep doc-lists sorted
  - Intersection $D_i \cap D_j$: Sort-Merge in $O(|D_i| + |D_j|)$
  - Union $D_i \cup D_j$: Sort-Merge in $O(|D_i| + |D_j|)$
  - If $|D_i| << |D_j|$, use binsearch in D_j for all terms in D_i
    - Whenever $|D_i| + |D_j| > |D_i|*\log(|D_j|)$

Less Obvious Tricks

- Define selectivity $sel(k_i) = df_i / |D|$
- Expected size of result is $|q| = |D| * sel(q) = |D| * \prod_i sel(k_i)$
  - Assuming AND and independence of query terms
- Intermediate result sizes vary greatly with different orders
  - These sizes have a large influence on runtime
  - How to keep sizes of intermediate results small?
  - Consider terms in order of increasing selectivity
  - Typically creates a few intersections at the beginning, then only look-ups
- General queries: Disjunctions, negations, …
  - $\mathbf{sel(k_i \cup k_j) = sel(k_i)+sel(k_j)-(sel(k_i)*sel(k_j))}$
  - Optimization problem: Find optimal order of evaluation

Adding Frequency

- VSM with TF*IDF requires term and document frequencies
- Split up inverted file into dictionary (term, df) and posting list (<docID, tf>)

Term	docIDs	DF
a	2	1
aid	1	1
all	1	1
and	2	1
come	1	1
country	1,2	2
dark	2	1
…	…	…
of	1	1
past	2	1
stormy	2	1
the	1,2	2
their	1	1
time	1,2	2
to	1	1
was	2	1

Dictionary / Postings

Term	DF	Posting
a	1	(2,1)
aid	1	(1,1)
all	1	(1,1)
and	1	(2,1)
come	1	(1,1)
country	2	(1,1), (2,1)
dark	1	(2,1)
…	…	…
of	1	(1,1)
past	1	(2,1)
stormy	1	(2,1)
the	2	(1,2), (2,1)
their	1	(1,1)
time	2	(1,1), (2,1)
to	1	(1,2)
was	1	(2,2)

Searching in VSM

- Assume we want to retrieve the top-r docs
- Algorithm
  - Initialize an empty doc-list S (as hash table or priority queue)
    - Will manage pairs (docID, score)
    - Score is computed incrementally, query term by query term
  - Iterate through query terms k_i
    - Retrieve posting list P
    - For all (docID, TF) $\in$ P
      - If docID$\in$S: S[docID] =+ IDF[k_i]*TF
      - else: S = S.append((docID, IDF[k_i]*TF))
  - For all pairs in S: Normalize scores
  - Return top-r docs in S
- S contains all and only those docs containing at least one k_i

$$sim(d,q) = \frac{\sum \left(v_q[i] * v_d[i] \right)}{\sqrt{\sum v_d[i]^2}}$$

Improvement

- Sort query terms by decreasing IDF Values – later terms have smaller IDF values – less impact on score
- Sort posting lists by decreasing TF values – later docs have smaller TF values – less impact on score
- Several heuristics for faster approximate VSM
  - H1: Stop adding docs to S in each posting if TF value too small
  - H2: Drop query terms whose IDF value is too small
    - Typically stop words with long posting lists – much work, little effect
  - H3: Assume we look at term k_i and are at position TF_j in the posting list. If $s^r - s^{r+1} > IDF_i * TF_j$, stop searching this posting list
  - …

Illustration H1

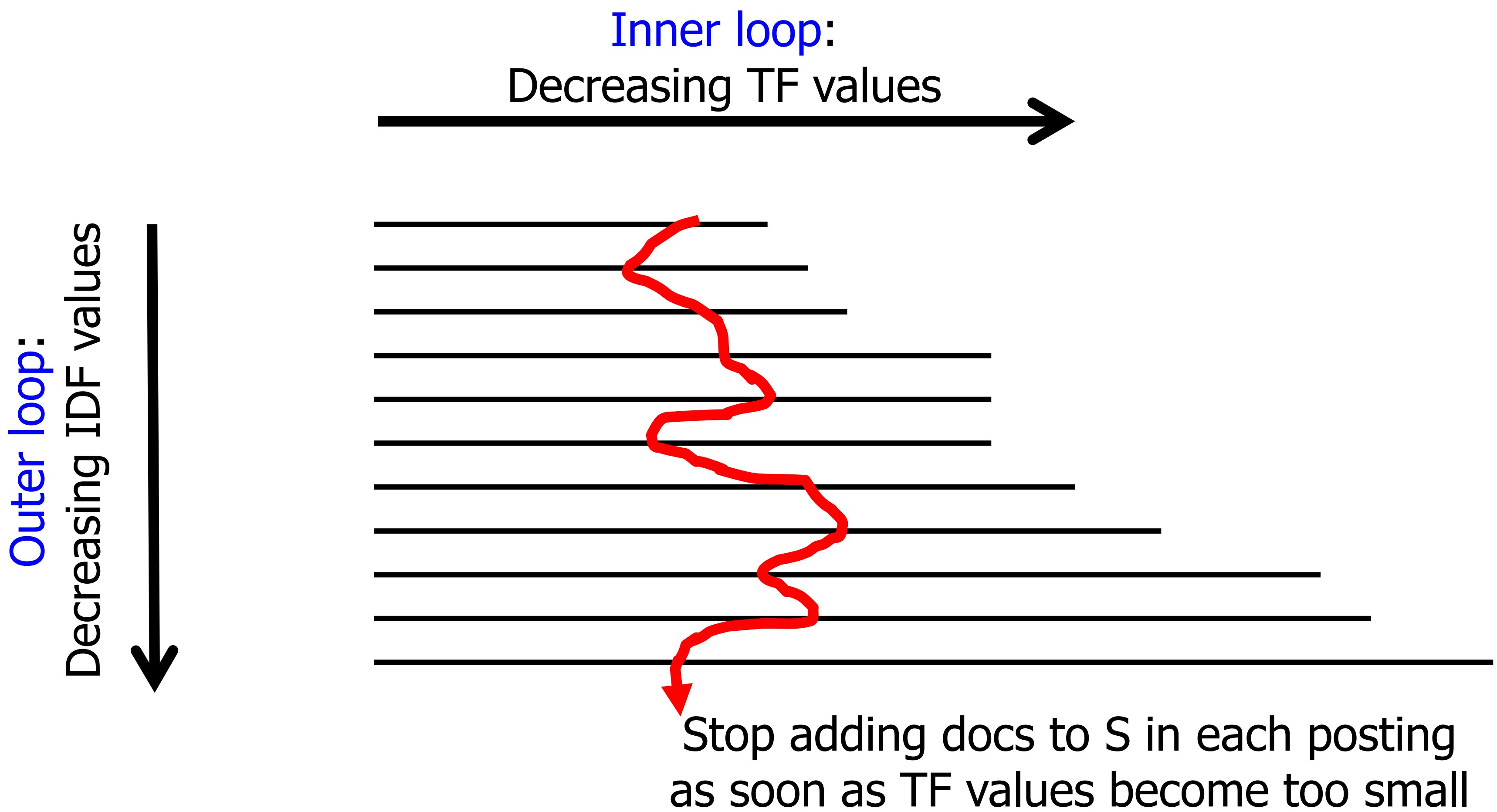
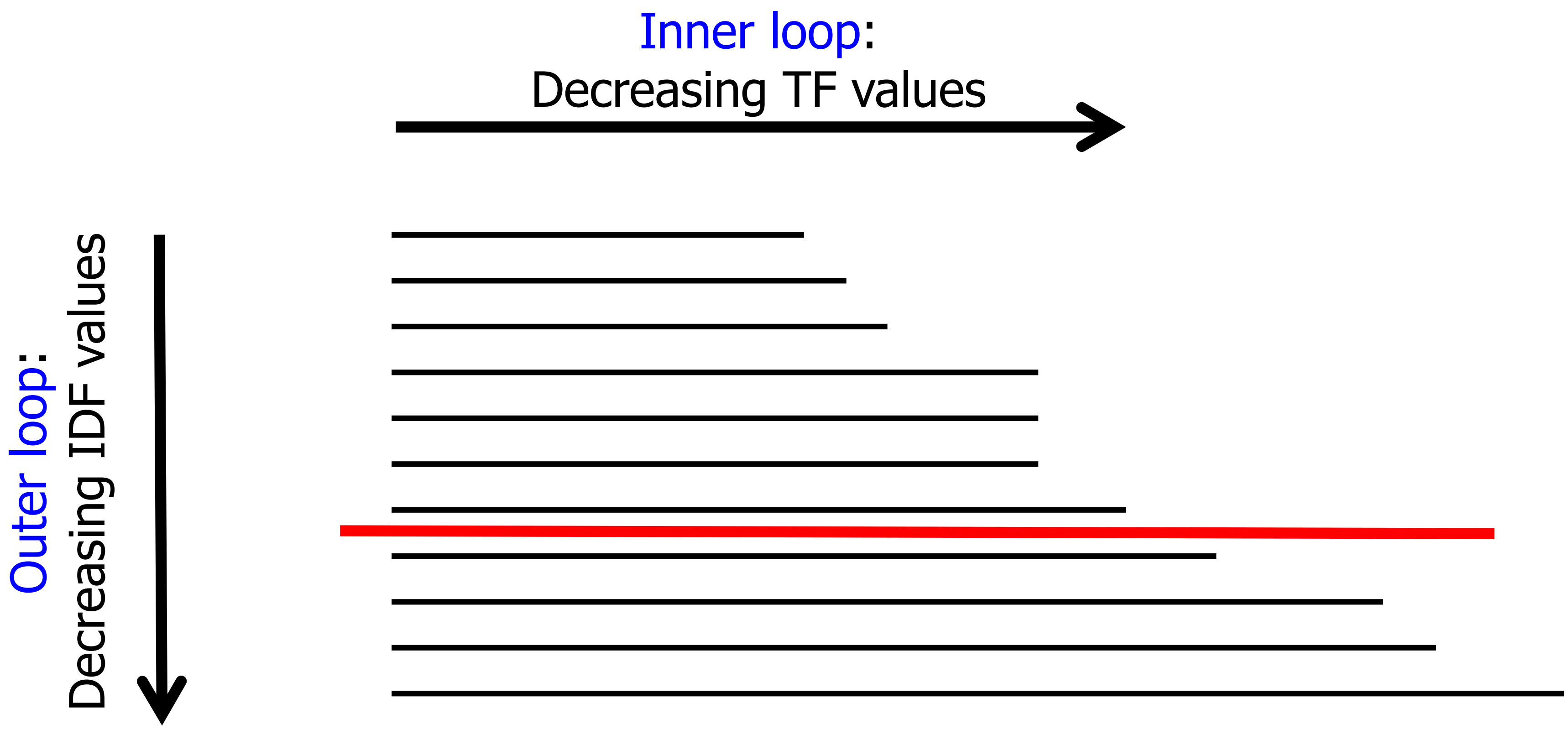
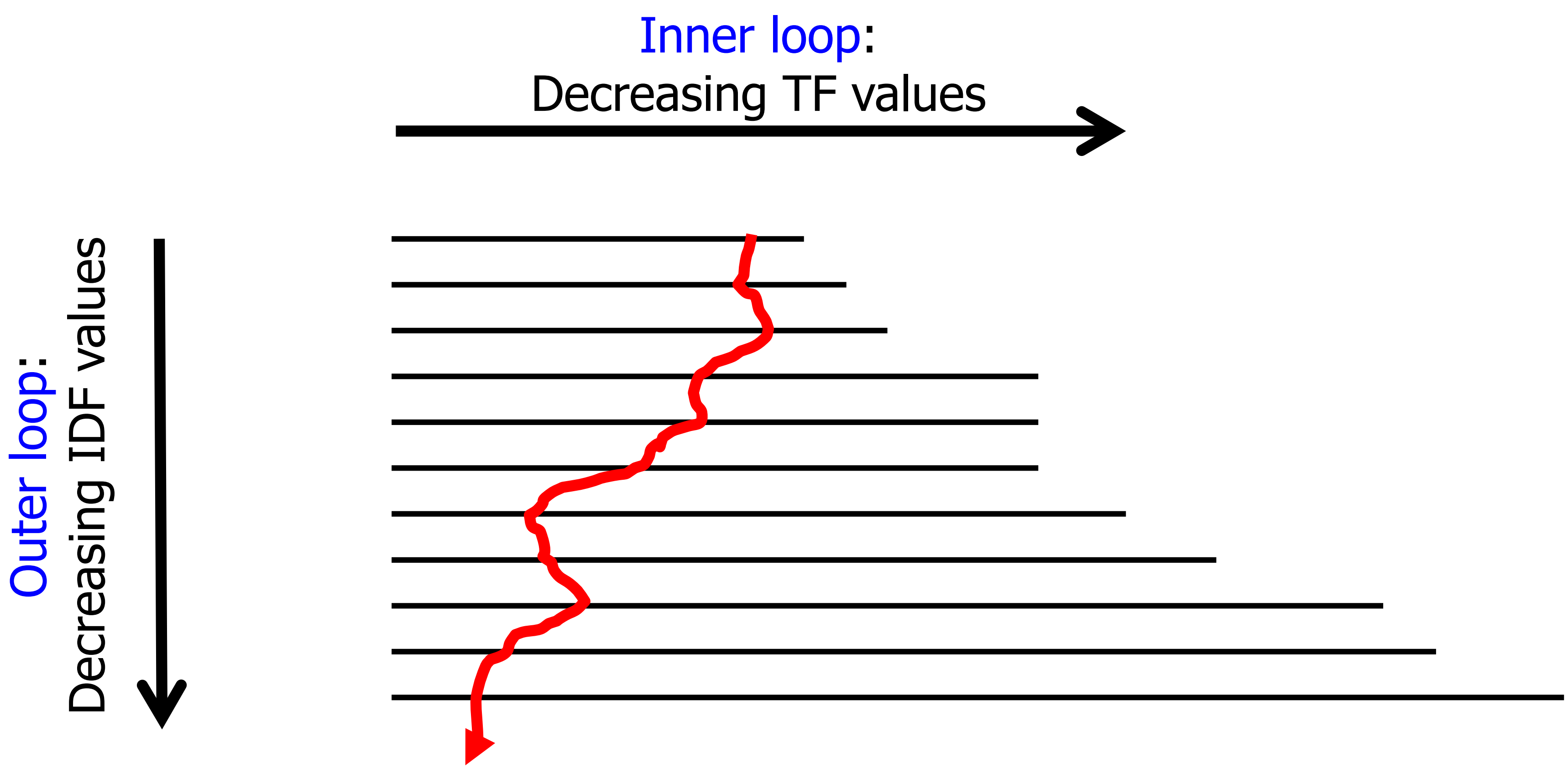


Illustration H2



Drop query terms whose IDF value are too small

Illustration H3



If $s^r - s^{r+1} > IDF_i * TF_j$, stop searching this posting list

Space Usage

- Size of dictionary: |K|
  - Zipf's law: If D is large, adding docs to D adds only few terms to K
    - But there are always new terms, no matter how large D
    - Example: 1GB text (TREC-2) generates only 5MB dictionary
    - Herdan–Heaps law: $|K| = a*terms(D)^b$
      - With a~50 and b~0,5 (for English)
  - Typically: |K|<1 Million
    - Not true in multi-lingual corpora, web corpora, etc.
- Size of posting list
  - Theoretic worst case: $O(|K|*|D|)$
  - Average case analysis is difficult, but certainly still large (in |D|)
- Implementation
  - Dictionary is small; should always fit into main memory
  - Posting lists is large; keep on disk

Content of this Lecture

- General approach
- Storage structures
  - The dictionary
  - The posting lists
- Phrase and proximity search
- Building and updating the index
- Using a RDBMS

Storing the Dictionary

- Dictionary are always kept in main memory
- Suitable data structures?

Storing the Dictionary

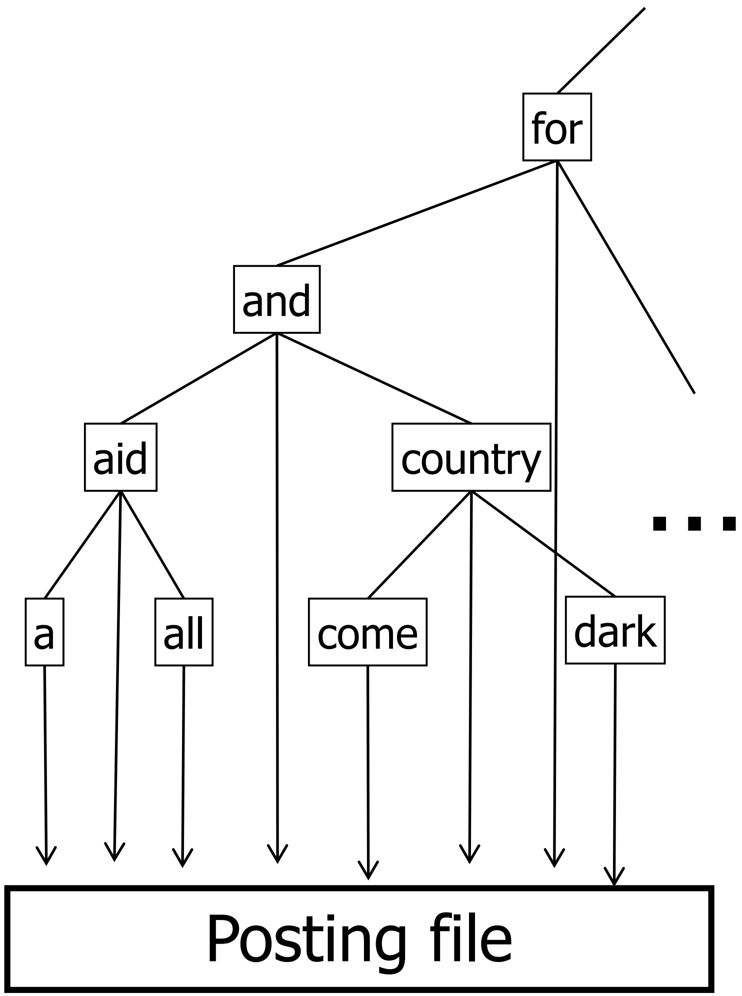
- Dictionary are always kept in main memory
- Suitable data structures?
  - Sorted term array: Small and fast (binsearch), static
  - Balanced binary (AVL) tree: Larger and fast, dynamic
  - Hashing: Either small and slow or large and very fast
  - (Compressed) Prefix-tree: Much larger and very fast
- In the following
  - Assume |ptr|=|DF|=4; |K|=1M
  - Let |q| be total length of query in characters
    - Usually small; used as upper bound on the number of char comparisons
  - Let n=8*|K|=8M be the sum of lengths of all terms in dictionary
    - Assuming average word length = 8

Dictionary as Sorted Array

- Elements: <term, DF, ptr>
- Since terms have different lengths: Implementation will be (ptr1, DF, ptr2)
  - ptr1: To string (the keyword)
  - ptr2: To posting list
- Search: Compute log(|K|) memory addresses, follow ptr1, compare strings: O(log(|K|)*|q|)
- Construction: O(|K|*log(|K|))
  - Sorting K
- Space: (4+4+4)*1M +n ~ 20M bytes
- But: Adding new terms is painful

Term	DF	
a	1	ptr
aid	1	ptr
all	1	ptr
and	1	ptr
come	1	ptr
country	2	ptr
dark	1	ptr
for	1	ptr
good	1	ptr
in	1	ptr
is	1	ptr
it	1	ptr
manor	1	ptr
men	1	ptr
midnight	1	ptr
night	1	ptr
now	1	ptr

Dictionary as AVL Tree



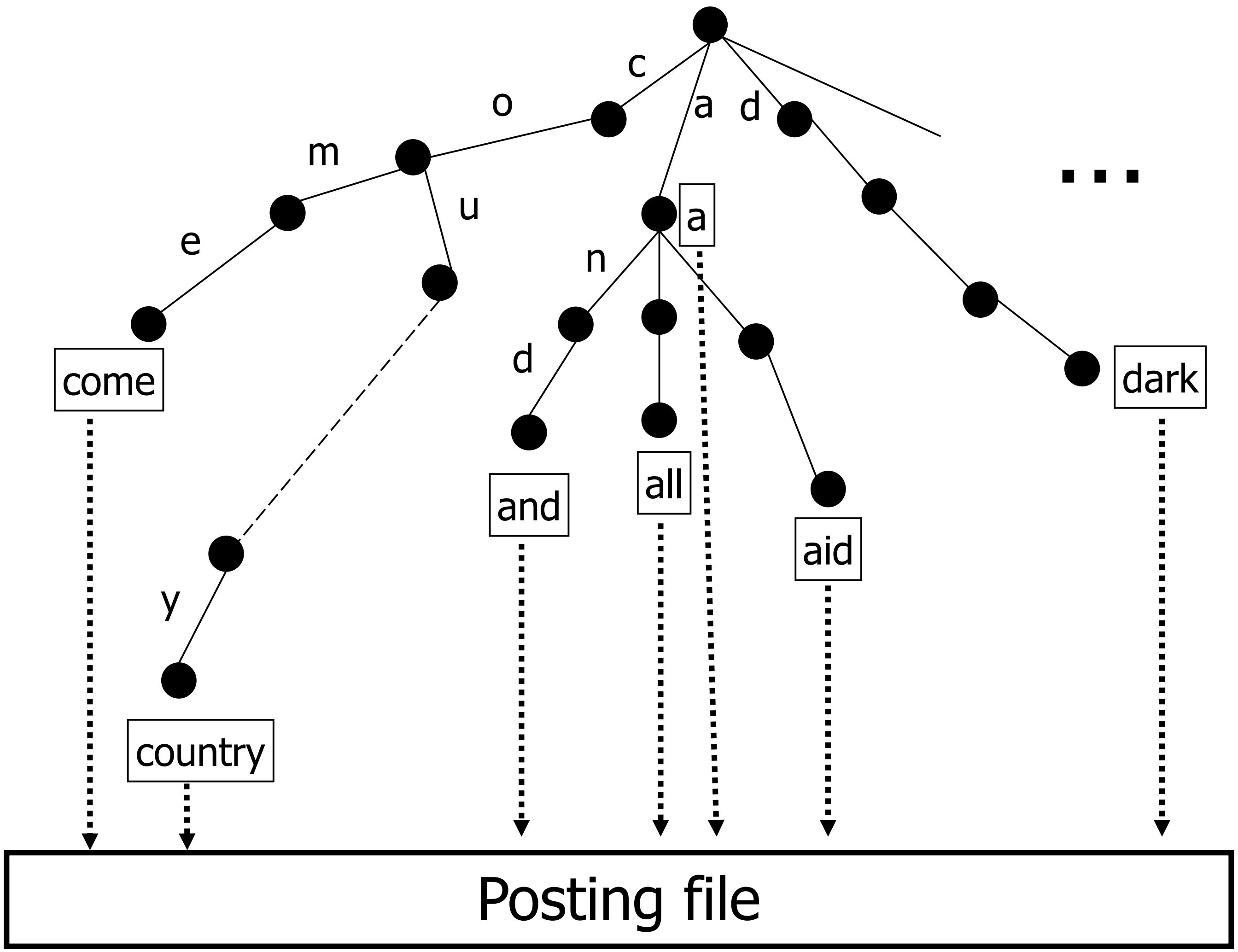
- Internal node: (ptr1, ptr2, ptr3, ptr4, DF)
  - String, posting, child1, child2
- Leaf: (ptr1, ptr2, DF)
- Search: Follow pointer, compare strings: O(log(|K|)*|q|)
- Construction: O(|K|*log(|K|))
- Space
  - Internal: 0.5M*(4+4+4+4+4)
  - Leaves: 0.5M*(4+4+4)
  - Together: 16M+n ~ 24MB
- Adding terms is simple

Dictionary as Hash Table

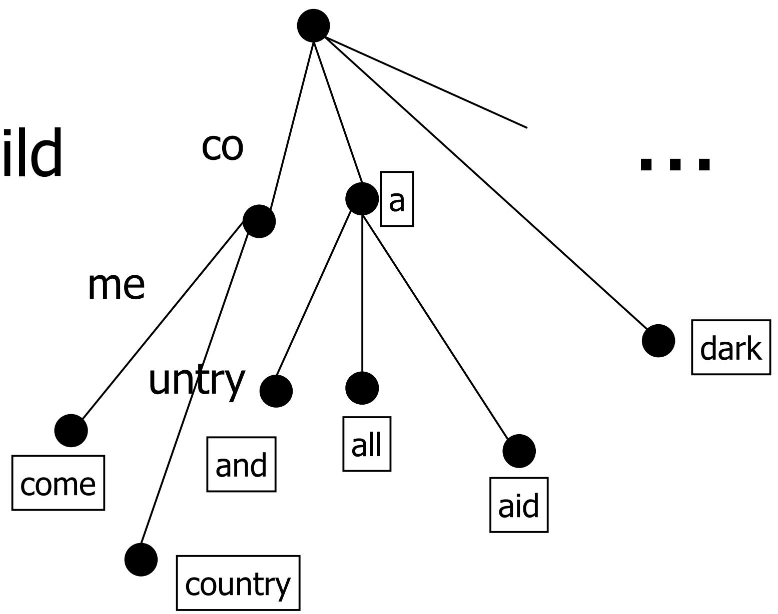
- Idea: Hash keywords into a hash table T (with overflow)
  - Value is <ptr-to-posting-list,DF>
- In principle, O(1) access is possible …
  - Construction: O(|K|)
  - Search time: O(|q|)
    - O(1) key comparisons, typical STRING hash functions look at all chars
  - Space: Difficult
    - Depends on size of hash table and implementation of overflow chains
- Only if collision are rare is used
  - Which requires that T is much larger than |K|
- Adding terms is simple (if collisions are rare)

Dictionary as Prefix Tree (TRIE: Information ReTRIEval)

Term	IDF
a	1
aid	1
all	1
and	1
come	1
country	2
dark	1
for	1
good	1
in	1
is	1
it	1
manor	1
men	1
midnight	1
night	1
now	1



Compressed Tries (Patricia Trees)



- Remove nodes with only one child
- Label edges with substrings, not single characters
- Saves space and pointers
- Search: O(|q|)
  - Maximally |q| char-comps + max |q| ptr to follow
  - Assumes O(1) for decision on child-pointer within each node
- Construction: O(n)
- Adding terms: Simple (search and insert new nodes)
- Space …

Space of a Trie

- Space: Difficult to estimate
- Assume 4 full levels, then each last inner node having two different suffixes (1M leaves, alphabet size 26)
  - 26 nodes in 1st, 26^2~700 in 2nd, 26^3~17.000 in 3rd, 26^4~450K in 4th
  - Assume each incoming edge stores only 1 character
  - Nodes in first 3 levels store 26 pointer, nodes in 4th only two
    - Beware: No O(|q|) search any more
- Inner: (26+700+17K)*(26*ptr+1)+450K*(2*ptr+1) ~ 6M
- Leaves: |K|*(string-ptr, posting-ptr, DF)+(n-|K|*4) ~ 16M
  - We only need to store a suffix of each term, prefix is in tree
- Together: ~22M
  - But assumptions are optimistic (Four full levels, equal-length terms)
  - Prefix trees are typically very space-consuming

Content of this Lecture

- General approach
- Storage structures
  - The dictionary
  - The posting lists
- Phrase and proximity search
- Building and updating the index
- Using a RDBMS

Storing the Posting File

- Posting file is usually kept on disk
- Suggestions?

Storing the Posting File

- Posting file kept on disk
- Static
  - Store posting lists one after the other in large file
  - Posting-ptr is offset in this file
- Less static: Prepare for inserts
  - Reserve additional space per posting
    - Good idea: Large initial posting lists get some extra space
    - Many inserts can be handled internally
  - Upon overflow, append entire posting list at the end of the file
    - Place pointer at old position – at most two access per posting list
    - Or update pointer in dictionary – better if only one copy around
    - Generates unused space (holes) – regular reorganization
    - Reorganization requires updating all pointers in the dictionary

Content of this Lecture

- General approach
- Storage structures
- Phrase and proximity search
- Building and updating the index
- Using a RDBMS

Positional Information

- What if we search for phrases: "Bill Clinton", "Ulf Leser"
  - ~10% of web searches are phrase queries
- What if we search by proximity "car AND rent/5"
  - "We rent cars", "cars for rent", "special care rent", "if you want to rent a car, click here", "Cars and motorcycles for rent", …
- We need positional information

Doc1:
Now is the time for all good men to come to the aid of their country.

: … a dark and stormy night in the country manor. The time was past midnight.

Term	Doc #
a	1
aid	1
all	1
and	1
come	1
country	2
…	…
the	2
their	1
time	2
…	…

Doc #	TF	Pos
2	1	6
1	1	1
1	1	14
2	1	15
1	1	6
1	1	16
2	1	10
1	1	15
	…	
1	2	3,12
2	2	9,12
	…	

Answering Phrase Queries

- Search posting lists of all query terms
- Boolean: During intersection, also positions must fit
  - If q="k_1 k_2" and $(d,tf_1,<P_1>) \in post(k_1)$ and $(d,tf_2,<P_2>) \in post(k_2)$
    - I.e.: d contains k1 in positions <P1> and k2 in positions <P2>
  - Find all pairs p_1, p_2 with $p_1 \in P_1$ and $p_2 \in P_2$ and $p_1+1=p_2$
  - For longer phrases, e.g. q="k_1 k_2 k_3 …", maintain a list P with all positions that are the end of a prefix of q
- VSM: More complicated (details skipped)

Effects

- Dictionary is not affected
- Posting lists get much larger
  - Store (docID, TF, <pos>) instead of (docID,TF)
  - Positional index typically 30-50% larger than the corpus itself
  - Especially frequent words create excessively long positing lists
- One trick: Compression of docIDs (delta encoding)
  - In large corpora, docID is a large integer
  - Trick: Store length of gaps instead of docID
    - **`t1: 17654,3,17655,12,17862,8,17880,4,17884,9, …`**
    - **`t1: 17654,3,1    ,12,207  ,8,18   ,4,4    ,9, …`**
  - In contrast, positions are small ints – no compression

Encoding

- Only pays off if we need few bits for small numbers but still have many bits for large numbers
- Variable-byte encoding
  - Always use at least 1 byte
  - Reserve first bit as "continuation bit" (cb) and 7 bit as payload
  - If cb=1, also use payload of next byte
    - **`t1: 17654,3,1       ,12,207                  ,8, …`**
    - **`t1: 17654,3,00000001,12,11001111|00000001,8, …`**
  - Simple, small numbers not encoded optimally
- γ (gamma) codes (details skipped)
  - Always use minimal number of bits for value
  - Encode length in unary encoding

Bi-Gram Index

- Alternative for phrase queries: Index over bi-grams
  - „The fat cat ate a rat“ – „the fat“, „fat cat“, „cat ate“, …
- Phrase query with |q| keywords gets translated into |q|-1 lookups
- Done?

Bi-Gram Index

- Alternative for phrase queries: Index over bi-grams
  - „The fat cat ate a rat" – „the fat", „fat cat", „cat ate", …
- Phrase query with |q| keywords gets translated into |q|-1 lookups
- Done?
  - Bi-gram need not appear sequentially in the doc
  - Need to confirm match after loading the doc
- Advantage
  - Very high selectivity of each bi-grams query leads to few confirmations
- Disadvantage: Very large dictionary

Proximity Search

- Phrase search = proximity search with distance one
- Proximity search
  - Search doc-lists with positional information for each term
  - Boolean: Upon intersection, check all pairs of positions
  - Can get quite involved for multi-term queries
    - "car AND rent/5 AND cheap/2 AND toyota/20" – "cheap" between 1 and 7 words from "car", "toyota" between 1 and 22 words from rent …
  - All conditions must be satisfied

Content of this Lecture

- General approach
- Storage structures
- Phrase and proximity search
- Building and updating the index
- Using a RDBMS

Building an Inverted File

- Assume a very large corpus: Billions of documents
  - We still assume that dictionary fits in memory
- How can we efficiently build the index?

Blocked, Sort-Based Indexing

- Partition corpus in blocks fitting into memory
- Algorithm
  - Keep dictionary always in memory
  - For each block: Load, create postings, flash to disk
  - Merge all posting lists
    - Open all at once
    - Skip through all posting lists term-by-term in sort-order
    - Merge doc-lists of equal keywords and flash to disk
      - If doc-lists are very large, we need special means here
- Requires 2 reads and 2 writes of all data
  - If there are enough file handles to open all posting lists at once

Updating an index: INSERT d_{new}

- What has to be done?
  - Foreach $k_i \in d_{new}$
    - Search k_i in dictionary
    - If present
      - Follow pointer to posting file
      - Add d_{new} to posting list of k_i
      - If list overflows, move posting list to end of file and place pointer
    - If not present
      - Insert k_i into dictionary
      - Add new posting list $\{d_{new}\}$ at end of posting file
- Disadvantage
  - Degradation: Many pointers in file, many terms require 2 IO
    - Especially the frequent ones
  - Index partly locked during updates

Using Auxiliary Indexes

- All inserts are performed on a second, auxiliary index
  - Keep it small: Always in memory
- Searches need to search real and auxiliary index
- When aux index grows too large, merge into real index
  - Try to append in-file: Same problem with degradation
  - Or read both indexes and write a new "fresh" index
  - In both cases, the index is locked
  - Solution: Work on a copy, then switch file pointers
- Alternative: Ignore new docs during search and periodically rebuild index on all docs

Content of this Lecture

- General approach
- Storage structures
- Phrase and proximity search
- Building and updating the index
- Using a RDBMS

Implementing an Inverted File using a RDBMS

Term-ID	Term	IDF
T1	Night	1
T2	To	2
…	…	…

Term-ID	Doc-ID	TF
T1	2	1
T2	1	2
…	…	…

Term-ID	Doc-ID	Pos
T1	2	6
T2	1	9
T2	1	11
…	…	…

night	2
now	1
of	1
past	2
stormy	2
the	1,2
their	1
time	1,2
to	1,2
was	1,2

Doc #	TF	Pos
2	1	6
1	1	1
1	1	14
2	1	15
1	1	6
1	2	3
1	2	12
2	2	9
2	2	12
1	1	15
1	1	4
2	1	13
1	2	9
1	2	11

Example Query 1

- Boolean: All docs containing terms “night” and “to”
  - ```
    SELECT D1.docid
    FROM terms T1, terms T2, termdoc D1, termdoc D2
    WHERE   T1.term=‘night’ AND T2.term=‘to’ AND
            D1.termid=T1.termid AND
            D2.termid=T2.termid AND
            D1.docid = D2.docid;
    ```

terms

Term-ID	Term	IDF
T1	Night	1
T2	To	2

pos

Term-ID	Doc-ID	Pos
T1	2	6
T2	1	9
T2	1	11

termdoc

Term-ID	Doc-ID	TF
T1	2	1
T2	1	2
...	...	...
    ```

Example Query 2

- VSM queries
  - We need to compute the inner product of two vectors
    - We ignore normalization
    - We assume TF-values of query terms are 1, others are 0
  - It suffices to aggregate TF values of matching terms per doc
- Example: Compute score for "night rider" (two terms)

```
SELECT did, sum(tf)
FROM (   SELECT D.docid did, T.term term, tf
         FROM terms T, termdoc D
         WHERE T.term='night' AND D.termid=T.termid)
         UNION
         SELECT D.docid did, T.term term, tf
         FROM terms T, termdoc D
         WHERE T.term='rider' AND D.termid=T.termid) docs
GROUP BY did;
```

Access Methods in a RDBMS

- Use B*-Indices on ID columns
- Searching a term
  - Requires O(log(|K|) random-access IO
    - Mind the base of the logarithm: Block size
    - For <100M terms, this usually means <3 IO (cache!)
  - Accessing the posting list: O(log(n)) quasi-random-access IO
    - Where n is the number of term occurrences in D
    - Access is a lookup with term-ID, then seq. scan along the B*-leaves
  - Compared to IR: Dictionary in memory, posting is accessed by direct link, then only sequential IO
- Advantages: Simple, easy to build
- Disadvantages: Much slower
  - More IO, general RDBMS overhead, space overhead for keys, …

Self Assessment

- Explain idea and structure of inverted files?
- What are possible data structures for the dictionary? Advantages / disadvantages?
- How can posting lists be managed?
- How much bigger is an inverted file with positions than without?
- How can one efficiently build a large inverted file from scratch?