Week 2 Overview

- Recap
  - Qs? Boolean Representation, Inverted Files, Tokenization
  - Q? Homework #1
  - Q? Chapters 1, 2
    - NOTs, Skip lists, Positional index, stopping
- Tonight:
  - Tolerant Retrieval (Ch. 3)
  - Building an Index (Ch. 4)

Chapter 3: Tolerant Retrieval

- Wildcard queries
  - Permuterm dictionary
  - k-grams (or n-grams or q-grams)
- Spelling Correction
  - General purpose
  - Proper names: Soundex algorithm
- Character N-grams (not in IIR)

Wild-card queries: *

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

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

Courtesy of Manning and Raghavan

Query processing

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

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

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

Courtesy of Manning and Raghavan

*’s inside a query term

- **How can we handle *’s in the middle of query term?**
  - **With one lookup**
  - **(Especially multiple *’s)**
- **The solution: transform every wild-card query so that the *’s occur at the end**
- **This gives rise to the Permuterm Index.**

Courtesy of Manning and Raghavan

Permuterm index

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

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

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

Courtesy of Manning and Raghavan

Permuterm query processing

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

Courtesy of Manning and Raghavan

k-gram conversion

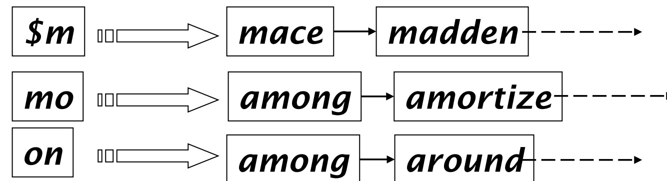
- **Enumerate all *k*-grams (sequence of *k* chars) occurring in any term**
- ***e.g.,* from text "*April is the cruelest month*" we get the 2-grams (*bigrams*)**

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

  – **$ is a special word boundary symbol**
- **Maintain a mapping from bigrams to *dictionary terms* that match each bigram.**

Courtesy of Manning and Raghavan

k-gram example



Courtesy of Manning and Raghavan

Wild-cards with character k-grams

- **Query** *mon** **can now be expanded by combining bigrams**
  - *$m* ***WITH*** *mo* ***WITH*** *on*
- **Fast, space efficient**
- **Gets terms that match the n-gram version of our wildcard query.**
- **But we'd generate** *moon***.**
  - **Must post-filter these terms against query.**
- **Surviving enumerated terms are then looked up in the term-document inverted index.**

Courtesy of Manning and Raghavan

Spell correction

- Two principal uses
  - Correcting document(s) being indexed
  - Retrieve matching documents when query contains a spelling error
- Two main flavors
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words e.g., *from* → *form*
  - Context-sensitive
    - Look at surrounding words, e.g., *I flew form Heathrow to Narita.*

Courtesy of Manning and Raghavan

Document correction

- Primarily for OCR'ed documents
  - Correction algorithms tuned for this
- Goal: the index (dictionary) contains fewer OCR-induced misspellings
- Can use domain-specific knowledge
  - E.g., OCR can confuse O and D more often than it would confuse O and I (adjacent on the QWERTY keyboard, so more likely interchanged in typing).

Courtesy of Manning and Raghavan

Query mis-spellings

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

Courtesy of Manning and Raghavan

Isolated word correction

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

Courtesy of Manning and Raghavan

Isolated word correction

- Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q
- What's "closest"?
- There are several alternatives
  - Edit distance
  - Weighted edit distance
  - *n*-gram overlap

Courtesy of Manning and Raghavan

Edit distance

- Given two strings S_1 and S_2, the minimum number of basic operations to covert one to the other
- Basic operations are typically character-level
  - Insert
  - Delete
  - Replace
- E.g., the edit distance from *swims* to *swam* is 2
- Generally found by dynamic programming
- Also called "Levenshtein distance"

Courtesy of Manning and Raghavan

Weighted edit distance

- As above, but the weight of an operation depends on the character(s) involved
  - Meant to capture keyboard errors, e.g. *m* more likely to be mis-typed as *n* than as *q*
  - Therefore, replacing *m* by *n* is a smaller edit distance than by *q*
  - (Same ideas usable for OCR, but with different weights)
- Require weight matrix as input
- Modify dynamic programming to handle weights

Courtesy of Manning and Raghavan

Edit distance to all dictionary terms?

- Given a (mis-spelled) query – do we compute its edit distance to every dictionary term?
  - Expensive and slow
- How do we cut the set of candidate dictionary terms?
- Here we use *n*-gram overlap for this

Courtesy of Manning and Raghavan

n-gram overlap

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

Courtesy of Manning and Raghavan

Example with trigrams

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

Courtesy of Manning and Raghavan

One option – Jaccard coefficient

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

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

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

Courtesy of Manning and Raghavan

Context-sensitive spell correction

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

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

because no docs matched the query phrase.

Courtesy of Manning and Raghavan

Issues in spell correction

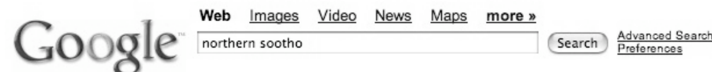
- Will enumerate multiple alternatives for "Did you mean"
- Need to figure out which one (or small number) to present to the user
- Use heuristics
  - The alternative hitting most docs
  - Query log analysis + tweaking
    - For especially popular, topical queries
- Spell-correction is computationally expensive
  - Run only on queries that matched few docs

Courtesy of Manning and Raghavan

northern sotho

Web

FDA Enforcement Report
Selig Chemical Industries brand **Sootho** Food Processing Sanitizing Lotion Soap ... FILED February 15, 1994; U.S. District Court for the **Northern** District of ...
www.fda.gov/bbs/topics/ENFORCE/ENF00307.html - 38k - Cached - Similar pages



Web

Did you mean: northern ***sotho***

Language Modeling Demo (KWIC)

- **Examination of co-occurrence patterns can help inform spelling correction**
  - **http://www.someya-net.com/concordancer/bigram.html**
- **Google wildcard search**
  - **"turn * at the * traffic"**
  - **"graduated with * degree in * science"**

Soundex

- **Class of heuristics to expand a query into phonetic equivalents**
  - **Language specific – mainly for names**
  - **E.g.,** *chebyshev → tchebycheff*
- **Turn every token to be indexed into a 4-character reduced form**
  - **Do the same with query terms**
- **Build and search an index on the reduced forms**
  - **(when the query calls for a soundex match)**

Courtesy of Manning and Raghavan

Soundex – typical algorithm

- Retain the first letter of the word.
- Change all occurrences of the following letters to '0' (zero):
  'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
- Change letters to digits as follows:
- B, F, P, V → 1
- C, G, J, K, Q, S, X, Z → 2
- D,T → 3
- L → 4
- M, N → 5
- R → 6

Courtesy of Manning and Raghavan

Soundex continued

- Replace consecutive identical digits with only the first, if they came from letters in the same category in the original string.
- Remove all zeros from the resulting string.
- Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.
- E.g., Herman becomes H655.

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

Courtesy of Manning and Raghavan

Word-based Information Retrieval

- **Most traditional information retrieval systems index documents according to the words in those documents.**
- **Word-based retrieval is language-specific (e.g., a retrieval system for English will not work as well for Arabic, Japanese, Korean, Turkish, and other languages).**
- **Word-based retrieval performs poorly when the documents to be retrieved are garbled or contain spelling mistakes (e.g., from OCR or speech transcription).**

N-grams as Indexing Terms

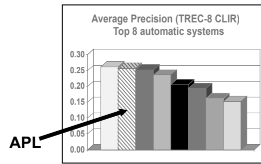
- **An *n-gram* is a sequence of *n* consecutive characters in a text**
- **N-grams can span words**
- **Advantages of n-grams:**
  - **language-independent**
  - **robust against errors in text**
  - **capture information about phrases**
- **This is not a way to find matching terms for wildcards, but rather a different way of indexing text**
- **n=4, n=5 good choices for indexing**

Four score and seven...

Words	N-Grams
four	four_
score	our_s
and	ur_sc
seven	r_sco
...	...

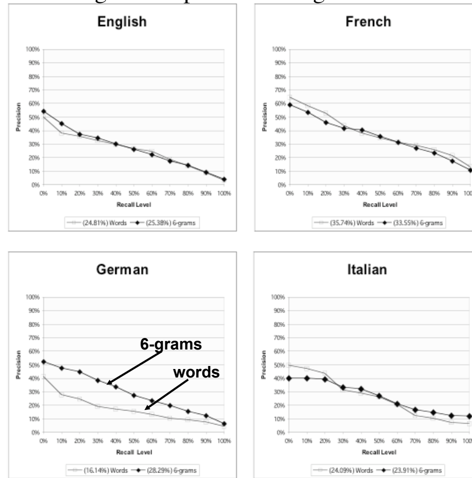
N-grams in non-Asian Languages

- N-grams are overlapping sequences of n characters
  - 6-grams of “Johns Hopkins” include: Johns_, ohns_H, hns_Ho, ns_Hop, and so forth
  - N-grams have been used chiefly in Asian languages where word identification is difficult
- TREC-8 Cross-Language Task
  - English, French, German, Italian



- N-grams and words comparable
  - But German words worse than 6-grams due to compounding
  - “einzelnefamilienstadtwohnung” vs. “single family townhouse”

Monolingual comparision of n-grams and words



Chapter 4: Index Construction

- Basic Dictionary Representations
- Indexing methods
  - Inverted File Algorithms
    - Study IIR 4.1, 4.2, 4.3, Skip 4.4, Read 4.5-4.7
  - Dynamic Indexes (4.5)

Sample Text

- Doc 1: Socrates is a man
- Doc 2: All men are mortal
- Doc 3: Socrates is mortal, mortal

Dictionary

Term	ID	DF	#Occur	Pointer
socrates	0	2	2	
is	1	2	2	
a	2	1	1	
man	3	1	1	
all	4	1	1	
men	5	1	1	
are	6	1	1	
mortal	7	2	3	

How should the dictionary be represented?

- Sorted Array
  - slow to update
- Tree
- Tries
- Hashtables

Arrays

- **Arrays provide access to a set of items**
  - Items in the array are stored contiguously in memory
  - Changing the length of an array is generally expensive (portions must be copied)
  - Length of arrays is fixed (known)
- **Fast access to data**
  - Reading or setting the ith item is O(1)
  - No pointers are required
  - Contiguous data may reduce paging if virtual memory is being used (compared to pointers)

fruit

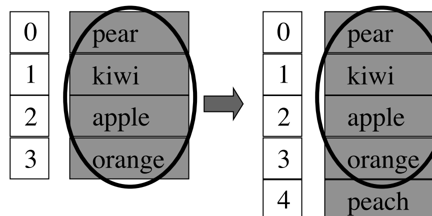
0	pear
1	kiwi
2	apple
3	orange
4	peach
5	apple

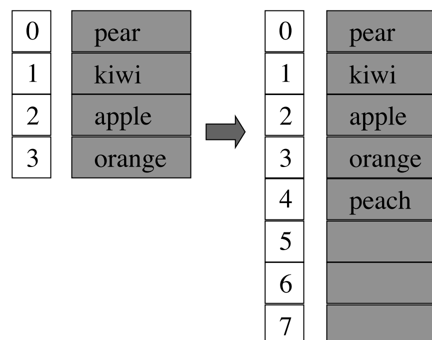
Array Operations

Operation	Time	Purpose
size()	O(1)	Determine the number of items
firstItem() lastItem()	O(1)	Return the first/last item (array is unchanged)
itemAt(i)	O(1)	Return the ith item (array is unchanged)
insert(x)	O(N)	Add item *x* to the array
search(x)	O(N)	Find item *x* and return it, or indicate failure
remove(x)	O(N)	Find item *x*, and delete it if present
set(i,x)	O(1)	Replace i^{th} item with x
createArray(n)	O(N)	Create an array with capacity n

Amortizing Costs

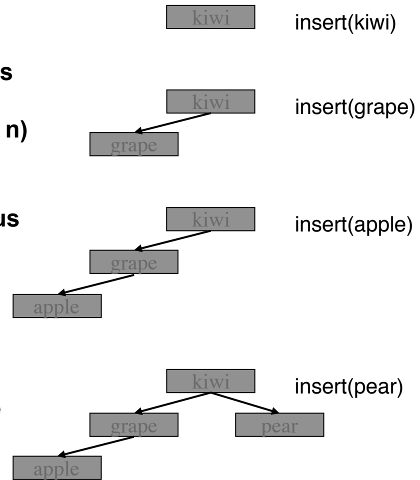
- When an array grows, many values need to be copied into a new array
  - By allocating extra storage space up front, the total cost over many growth operations can be reduced
  - For example, one policy would be to double the size of the array whenever it needs to grow
- Example
  - Suppose the 'fruit' array has 4 items and we want to add a fifth item peach
  - Now the array has extra capacity; adding a 6th, 7th, and 8th does not require any copying





Binary Trees

- **Generally Lists and Arrays require linear time to search for a given item**
  - **If items can be ordered much better is possible (just like a phone book)**
  - **Insertion and Deletion become O(log n) operations**
  - **Search becomes O(log n)**
  - **Data is preserved in sorted order, thus there is no need to sort the data**
- **Tree implementation**
  - **2 pointers & 1 item per node**
  - **Trees can become unbalanced**
    - **Special approaches maintain balance**
    - **Red-Black trees, AVL trees**



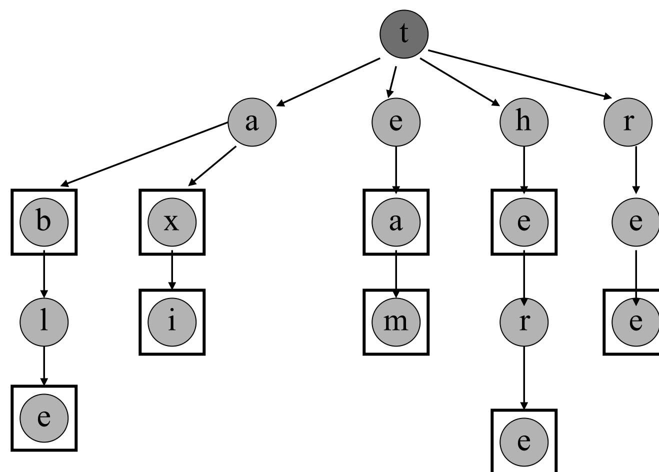
Tree Operations

Operation	Time	Purpose
size()	O(1)	Determine the number of items in the list
firstItem() lastItem()	O(1)	Return the first or last item (tree is unchanged)
itemAt(i)	O(N)	Return the ith item (tree is unchanged)
insert(x)	O(log n)	Add item *x* to the tree
search(x)	O(log n)	Find item *x* and return it, or indicate failure
remove(x)	O(log n)	Find item *x*, and delete it if present

Tries (de la Briandais - 1959)

- Given an ordered alphabet, represent words on a lexicographic tree
  - Root of tree is first character in word
  - Children of root represent second character, and so on
  - Height of tree is the number of nodes on the longest path from any leaf to the root
  - During search $O(\log_{\text{alphabet}} n)$ internal nodes examined, worst case
- Variants
  - Patricia Trees (collapse single child nodes)
  - Suffix tree

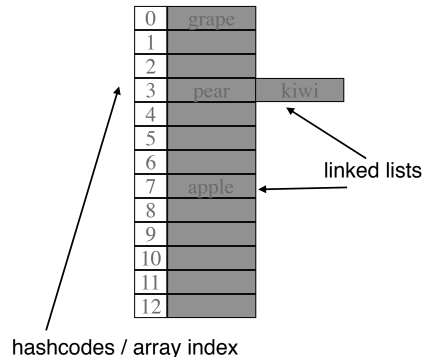
A Trie



How should each level (i.e., the green nodes) be stored?

Hashtables

- **Logarithmic access is fairly efficient**
  - **Search in a tree with 1,000,000 entries requires only 20 comparisions**
- **O(1) time is possible, if**
  - **Order information is unimportant**
  - **Data values are unique**
    - **that is, duplicates are not maintained**
  - **Data items can be tested for equality**
  - **Items can be mapped to the natural numbers using a ‘hash’ function**
- **Table implementation**
  - **An array of size much smaller than the universe of possible items (keys)**
  - **A value may be stored with each key**
  - **Many variants**



Hashtable Operations

Operation	Time	Purpose
size()	O(1)	Determine the number of items in the list
firstItem() lastItem()	N/A	No order information is preserved
itemAt(i)	N/A	No order information is preserved
insert(x)	O(1)	Add item *x* to the table
search(x)	O(1)	Find item *x* and return it’s value, or indicate failure
remove(x)	O(1)	Find item *x*, and delete it (and its value) if present

Technically, the O(1) performance of insert/search/remove depends on having a hash function that distributes keys uniformly

Guidance

- Easy to implement…
  - Hashtables, Trees, & Tries
  - Each of these is reasonable for the smallish collections we will use in class
- Minimal memory (next chapter)
  - Front-coding (and arrays, or B-trees)
  - Perfect hash functions
    - But, first need to get set of all terms

Lexicon payload

- Must include
  - term itself
  - pointer to binary file
- Could store on disk, but useful in memory
  - document frequency
  - don’t normally store postings in memory
    - They go into the inverted file
- Don’t strictly need
  - termid, number of occurrences
- How to store on disk
  - Binary data, length-encoded strings

Building Inverted Files

- Doc 1: Socrates is a man
- Doc 2: All men are mortal
- Doc 3: Socrates is mortal, mortal

Dictionary

Term	ID	DF	#Occur	Pointer
socrates	0	2	2	→
is	1	2	2	→
a	2	1	1	
man	3	1	1	
all	4	1	1	
men	5	1	1	
are	6	1	1	
mortal	7	2	3	

Inverted File

Doc	times	Doc	times
1	1	3	1
1	1	3	1
1	1		
1	1		
2	1		
2	1		
2	1		
2	1	3	2

Data structures usually rely on termids vs. strings

Notation (differs from text, see IIR 4.1)

- ***t***: the size of the text, in words
- ***N***: the number of documents
- ***v***: vocabulary size
- ***k***: the average number of *unique* terms per document
  - t > k * N *but O(k*N) = O(t), usually*
  - t << v * N
- ***M***: the amount of main memory available

Space Requirements

- The space required for the vocabulary is rather small. According to *Heaps'* law the vocabulary, v, grows as $O(t^{\beta})$, where β is a constant between 0.4 and 0.6 in practice

- On the other hand, the occurrences (inverted file) demand much more space. Since each word appearing in the text is referenced once in that structure, the space is $O(t)$

Plan A: Memory-based inversion

- All the vocabulary is kept in a suitable data structure
  - for each word also store a list of its occurrences in documents

- Each word of the text is read and searched in the vocabulary

- If it is not found, it is added to the vocabulary with a empty list of occurrences and the new position is added to the end of its list of occurrences

- Count duplicates in a document

Method A cont'd

- **Once the text is exhausted the vocabulary is written to disk with the list of occurrences. Two binary files are created:**
  - **in the first file, the list of occurrences are stored contiguously**
  - **in the second file, the vocabulary is stored in lexicographical order and, for each word, a pointer to its list in the first file is also included. This allows the vocabulary to be kept in memory at search time**

- **The overall process is *O(t)* worst-case time**
  - **O(t) scanning text, O(t) lookups*, O(t) writing to disk**

- **Sunk if we can't fit the inverted file in memory?**

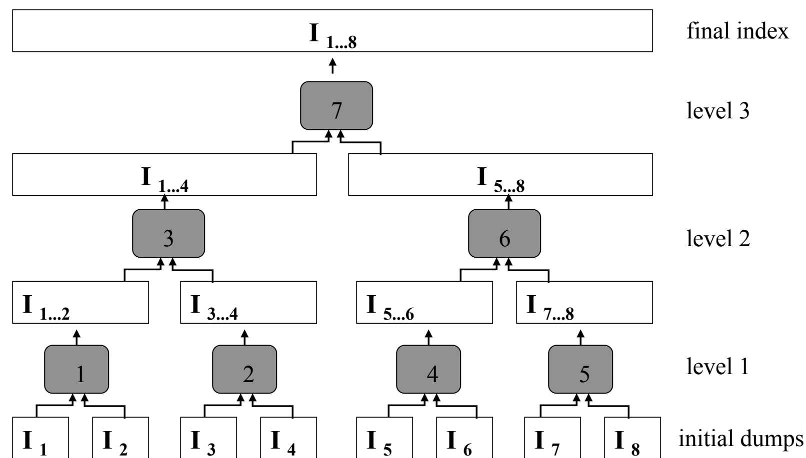
Algorithm B (sort-based inversion)

- **As soon as each term is seen, a tuple (or record) is written to a *temporary* file**
  - **<socrates, doc1, 1>**
  - **<men, doc2, 1>**
  - **<socrates, doc3, 2>**
- **These intermediate files are externally sorted in-place, on disk**
  - **term is primary key, docid is secondary key**
  - **Let's assume each temp. file is of size M**
- **Then the sorted files are merged together**
- **Not great**
  - **lots of I/O (including random access)**
  - **sorting C * O(t * log(t/M))   C is big**
  - **merging O(t)**

Alg C (memory-sorted inversion)

- An option is to fill main memory (M) with tuples, and then, sort the tuples and write the partial index I_i obtained up to now to disk. Then the current, partial index is erased from main memory and we continue with the rest of the text.
  - This nicely avoids sorting on disk
  - Described in IIR 4.2 (and in MG 5.2)
- Once the text is exhausted, a number of partial indices I_i exist on disk
- The partial indices are merged to obtain the final index

Alg C: Hierarchical Merge Step



Merging Partial Inverted Files

- **Doc 1: Socrates is a man**
- **Doc 2: All men are mortal**
- **Doc 3: Socrates is mortal, mortal**

Partial inverted file (1+2)

Term	Doc	times
socrates	1	1
is	1	1
a	1	1
man	1	1
all	2	1
men	2	1
are	2	1
mortal	2	1

Inverted File (3)

Term	Doc	times
socrates	3	1
is	3	1
mortal	3	2

Records sorted by termid (socrates is termid 0, mortal is 7)

Analysis of Algorithm C

- **The total time to scan the text is *O(t)***
- **The number of partial indices is *O(t/M)***
  - **Each is sorted at a cost of O(M log (M))**
- ***$log_2(t/M)$* merging levels are required to merge the *O(t/M)* partial indices**
  - ***Cost: O(t * log(t/M))***
- **The total cost of this algorithm is:**
  - ***O(t) + O (t/M * (M log M)) + O(t * log(t/M)) =***
  - ***The larger of: O(t * log(M)) or O(t * log(t/M))***
- ***Can we do better?***

Considered Pause...

- **We've seen two kinds of algorithms**
  - **Alg A avoided sorting, but was not robust in the face of memory constraints**
  - **Algs B,C sort sub-indexes and hierarchical merging to avoid memory limits**

- **We can make two improvements**
  - **Better merging**
  - **Avoid sorting sub-indexes at all**
    - **(This method is not found in the text)**

Algorithm D

- **Instead of performing a hierarchical merge, perform an *n-way merge***
- **Described in *IIR, page 65***
  - **"To do the merging, we open all block files simultaneously, and maintain small read buffers for the ten blocks we are reading and a write buffer for the [one] final merged index we are writing.**
- **Limitation is the number of open file handles**
  - **Some systems limit the number to 40-1024 or so open files (an operating system resource)**
- **Merging is more efficient**
  - **Performance is O(t * log(M))**

Algorithm E (IIR 4.3 is similar)

- Act like Alg A. When memory is exhausted, don't give up. Instead write a partial index out for the currently examined documents
- Keep the to-date lexicon in memory, but flush all of the postings out. Now start with the next set of documents
- Perform an n-way merge on the partial indexes
- Overall cost
  - Scan each word, add individual <docid,count> pair to variable length array associated with lexicon: O(t)
  - Write out each word to a partial index (once): O(t)
  - N-way merge: O(t)
  - Total: 3 * O(t) = O(t) *linear in length of corpus*