Plan

- Last lecture: Tolerant retrieval
  - Wildcards
  - Spell correction
  - Soundex
- This time:
  - Index construction

Matching trigrams

- Consider the query *lord* – we wish to identify words matching 2 of its 3 bigrams (*lo, or, rd*)


![](image)

Standard postings “merge” will enumerate ...

Adapt this to using Jaccard (or another) measure.

Index construction

- How do we construct an index?
- What strategies can we use with limited main memory?

Recall our corpus

- Number of docs = \( n = 1M \)
  - Each doc has 1K terms
- Number of distinct terms = \( m = 500K \)
- Use Zipf to estimate number of postings entries

Zipf estimation of postings

- Recall the blocks in the matrix of Lecture 3
- Each row corresponds to term
  - Rows ordered by diminishing term frequency
- Each column corresponds to a document
- We broke up the matrix into blocks.
- We are asking: how many 1’s in this matrix?
Rows by decreasing frequency

- $n$ docs
- $m$ terms
- $j$ most frequent terms
- $j$ next most frequent terms
- $j$ next most frequent terms
- etc.

Overestimate or underestimate, by how much?

How many postings?

- Number of 1’s in the $i$ th block = $nJ/i$
- Summing this over $m/J$ blocks, we have
  $$\sum_{i=1}^{m/J} \frac{nJ}{i} \approx nJ \ln m / J.$$
- For our numbers, this should be about 667 million postings.

Recall index construction

- Documents are parsed to extract words and these are saved with the Document ID.

Key step

- After all documents have been parsed the inverted file is sorted by terms.

Index construction

- As we build up the index, cannot exploit compression tricks
- Parse docs one at a time.
- Final postings for any term – incomplete until the end.
  - (actually you can exploit compression, but this becomes a lot more complex)
- At 10-12 bytes per postings entry, demands several temporary gigabytes

System parameters for design

- Disk seek ~ 10 milliseconds
- Block transfer from disk ~ 1 microsecond per byte (following a seek)
- All other ops ~ 10 microseconds
  - E.g., compare two postings entries and decide their merge order
Bottleneck

- Parse and build postings entries one doc at a time
- Now sort postings entries by term (then by doc within each term)
- Doing this with random disk seeks would be too slow - must sort \( n=667 \text{M} \) records

If every comparison took 2 disk seeks, and \( n \) items could be sorted with \( \log n \) comparisons, how long would this take?

Sorting with fewer disk seeks

- 12-byte (4+4+4) records (term, doc, freq).
- These are generated as we parse docs.
- Must now sort 667M such 12-byte records by term.
- Define a **Block ~ 10M** such records
  - can "easily" fit a couple into memory.
  - Will have 64 such blocks to start with.
- Will sort within blocks first, then merge the blocks into one long sorted order.

Sorting 64 blocks of 10M records

- First, read each block and sort within:
  - Quicksort takes \( 2n \ln n \) expected steps
  - In our case 2 x (10M ln 10M) steps
- **Exercise:** estimate total time to read each block from disk and and quicksort it.
- 64 times this estimate - gives us 64 sorted runs of 10M records each.
- Need 2 copies of data on disk, throughout.

Merging 64 sorted runs

- Merge tree of \( \log 64=6 \) layers.
- During each layer, read into memory runs in blocks of 10M, merge, write back.

Merging 64 runs

- Time estimate for disk transfer:
  - \( 6 \times (64 \text{runs} \times 120 \text{MB} \times 10^{-6} \text{sec}) \times 2 \sim 25 \text{hrs.} \)
- Work out how these transfers are staged, and the total time for merging.
Exercise - fill in this table

<table>
<thead>
<tr>
<th>Step</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 64 initial quicksorts of 10M records each</td>
<td></td>
</tr>
<tr>
<td>2 Read 2 sorted blocks for merging, write back</td>
<td></td>
</tr>
<tr>
<td>3 Merge 2 sorted blocks</td>
<td></td>
</tr>
<tr>
<td>4 Add (2) + (3) = time to read/merge/write 64 times (4) = total merge time</td>
<td></td>
</tr>
</tbody>
</table>

Large memory indexing

- Suppose instead that we had 16GB of memory for the above indexing task.
- Exercise: What initial block sizes would we choose? What index time does this yield?
- Repeat with a couple of values of \( n, m \).
- In practice, spidering often interlaced with indexing.
  - Spidering bottlenecked by WAN speed and many other factors - more on this later.

Improvements on basic merge

- Compressed temporary files
  - Compress terms in temporary dictionary runs
- How do we merge compressed runs to generate a compressed run?
  - Given two \( \gamma \)-encoded runs, merge them into a new \( \gamma \)-encoded run
  - To do this, first \( \gamma \)-decode a run into a sequence of gaps, then actual records:
    - \( 33,14,107,5... \rightarrow 33, 47, 154, 159 \)
    - \( 13,12,109,5... \rightarrow 13, 25, 134, 139 \)

Merging compressed runs

- Now merge:
  - \( 13, 25, 33, 47, 134, 139, 154, 159 \)
- Now generate new gap sequence
  - \( 13,12,8,14,87,5,15,5 \)
- Finish by \( \gamma \)-encoding the gap sequence
- But what was the point of all this?
  - If we were to uncompress the entire run in memory, we save no memory
  - How do we gain anything?

“Zipper” uncompress/decompress

- When merging two runs, bring their \( \gamma \)-encoded versions into memory
- Do NOT uncompress the entire gap sequence at once – only a small segment at a time
  - Merge the uncompressed segments
  - Compress merged segments again

Improving on binary merge tree

- Merge more than 2 runs at a time
  - Merge \( k>2 \) runs at a time for a shallower tree
  - Maintain heap of candidates from each run
Dynamic indexing
- Docs come in over time
  - postings updates for terms already in dictionary
  - new terms added to dictionary
- Docs get deleted

Simplest approach
- Maintain “big” main index
- New docs go into “small” auxiliary index
- Search across both, merge results
- Deletions
  - Invalidation bit-vector for deleted docs
  - Filter docs output on a search result by this invalidation bit-vector
  - Periodically, re-index into one main index

Issue with big and small indexes
- Corpus-wide statistics are hard to maintain
- E.g., when we spoke of spell-correction: which of several corrected alternatives do we present to the user?
  - We said, pick the one with the most hits
- How do we maintain the top ones with multiple indexes?
  - One possibility: ignore the small index for such ordering
- Will see more such statistics used in results ranking

More complex approach
- Fully dynamic updates
- Only one index at all times
  - No big and small indices
- Active management of a pool of space

Fully dynamic updates
- Inserting a (variable-length) record
  - e.g., a typical postings entry
- Maintain a pool of (say) 64KB chunks
- Chunk header maintains metadata on records in chunk, and its free space

Global tracking
- In memory, maintain a global record address table that says, for each record, the chunk it’s in.
- Define one chunk to be current.
- Insertion
  - if current chunk has enough free space
    - extend record and update metadata.
  - else look in other chunks for enough space.
  - else open new chunk.
Building positional indexes
- Still a sorting problem (but larger) — Why?
- Recall the harder exercise of Lecture 3 for estimating the number of positional index entries
- Exercise: given 1GB of memory, how would you adapt the block merge described above?

Building n-gram indexes
- As text is parsed, enumerate n-grams.
- For each n-gram, need pointers to all dictionary terms containing it — the “postings”.
- Note that the same “postings entry” can arise repeatedly in parsing the docs — need efficient “hash” to keep track of this.
  - E.g., that the trigram *uou* occurs in the term *deciduous* will be discovered on each text occurrence of *deciduous*

Building n-gram indexes
- Once all *(n-grams)* term pairs have been enumerated, must sort for inversion
- Recall average English dictionary term is ~8 characters
  - So about 6 trigrams per term on average
  - For a vocabulary of 500K terms, this is about 3 million pointers — can compress

Changes to dictionary
- New terms appear over time
  - cannot use a static perfect hash for dictionary
- OK to use term character string w/pointers from postings as in Lecture 3.

Index on disk vs. memory
- Most retrieval systems keep the dictionary in memory and the postings on disk
- Web search engines frequently keep both in memory
  - massive memory requirement
  - feasible for large web service installations
  - less so for commercial usage where query loads are lighter

Indexing in the real world
- Typically, don’t have all documents sitting on a local filesystem
  - Documents need to be spidered
  - Could be dispersed over a WAN with varying connectivity
- Must schedule distributed spiders/indexers
- Could be (secure content) in
  - Databases
  - Content management applications
  - Email applications
Content residing in applications

- Mail systems/groupware, content management contain the most "valuable" documents
- http often not the most efficient way of fetching these documents - native API fetching
  - Specialized, repository-specific connectors
  - These connectors also facilitate document viewing when a search result is selected for viewing

Secure documents

- Each document is accessible to a subset of users
  - Usually implemented through some form of Access Control Lists (ACLs)
- Search users are authenticated
- Query should retrieve a document only if user can access it
  - So if there are docs matching your search but you’re not privy to them, “Sorry no results found”
  - E.g., as a lowly employee in the company, I get “No results” for the query “salary roster”

Users in groups, docs from groups

- Index the ACLs and filter results by them

```
Users                                      Documents
0/1                                      0 if user can't read doc, 1 otherwise
```

- Often, user membership in an ACL group verified at query time – slowdown

Exercise

- Can spelling suggestion compromise such document-level security?
- Consider the case when there are documents matching my query, but I lack access to them.

Compound documents

- What if a doc consisted of components
  - Each component has its own ACL.
  - Your search should get a doc only if your query meets one of its components that you have access to.
- More generally: doc assembled from computations on components
  - e.g., in Lotus databases or in content management systems
- How do you index such docs?

No good answers…

“Rich” documents

- (How) Do we index images?
- Researchers have devised Query Based on Image Content (QBIC) systems
  - “show me a picture similar to this orange circle”
  - watch for lecture on vector space retrieval
- In practice, image search based on meta-data such as file name e.g., monalisa.jpg
Passage/sentence retrieval

- Suppose we want to retrieve not an entire document matching a query, but only a passage/sentence - say, in a very long document
- Can index passages/sentences as mini-documents – what should the index units be?
- More on this when discussing XML search

Next up – scoring/ranking

- Thus far, documents either match a query or do not.
- It’s time to become more discriminating - how well does a document match a query?
- Gives rise to ranking and scoring

Resources

- MG Chapter 5