Indexing and Query Processing 2

Index building

Basic problem

Have:

`d1:` $f_{d1,t3}$, $f_{d1,t6}$, $f_{d1,t19}$, …

`d2:` $f_{d2,t1}$, $f_{d2,t99}$, $f_{d2,t110}$, …

…

Need:

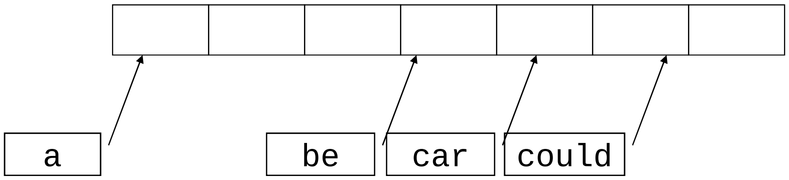
`t1:` $f_{d2,t1}$, $f_{d11,t1}$, $f_{d19,t1}$, …

`t2:` $f_{d7,t2}$, $f_{d8,t2}$, $f_{d55,t2}$, …

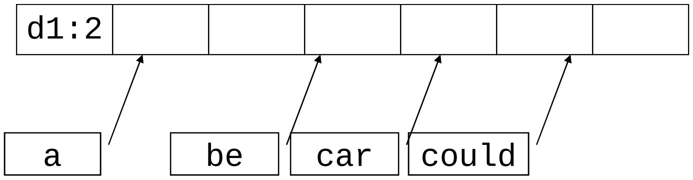
…

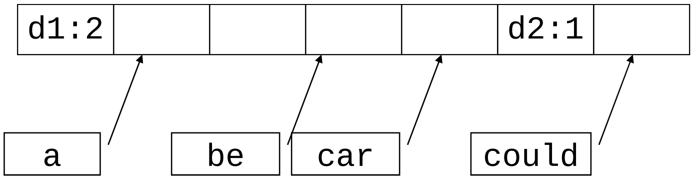
In-Memory Inversion

- Make a pass over all documents, figuring out f_t for each t.
- Then pre-allocate spans of memory for each inverted list.

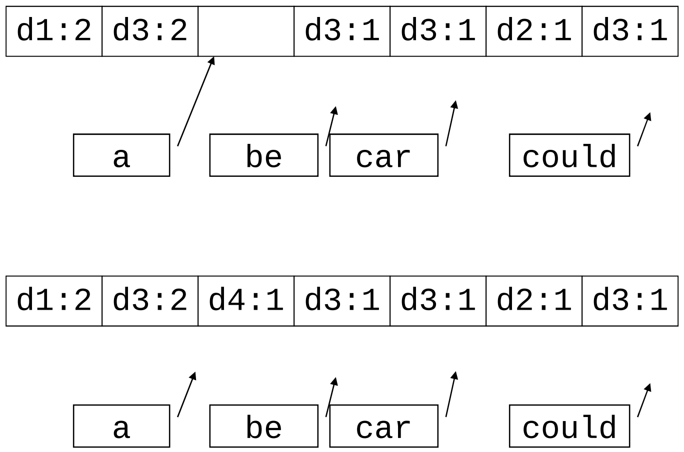


Example Continued





Example Continued



d1:2	d3:2		d3:1	d3:1	d2:1	d3:1

a be car could

d1:2	d3:2	d4:1	d3:1	d3:1	d2:1	d3:1

a be car could

Sort-based inversion

Have tuples (t, d, $f_{d,t}$)

  - Ordered on d
  - Can sort on t
  - Build inverted lists from sorted sequence

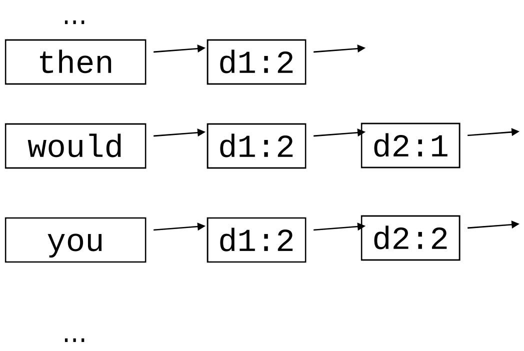
- Want a stable sort, such as merge sort
- Might benefit to compress blocks on disk
- Need to have vocabulary in memory

Merge small indexes

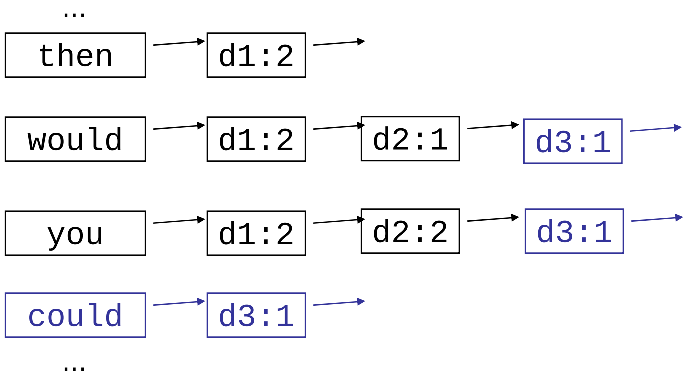
Build inverted indexes directly in memory

- Each term has an extensible list for entries
- Need a directory structure to get from each term to its list (hash table, balanced tree)
- Parse one document at a time, add entries to lists for terms in the document

Example index construction



Example index construction (2)



…

then → d1:2 →

would → d1:2 → d2:1 → d3:1 →

you → d1:2 → d2:2 → d3:1 →

could → d3:1 →

…

Combine small indexes

- When memory fills, flush vocabulary and index to disk (in order): a “run”
  - Use a space-efficient representation
  - Start over with next unindexed document
- When all documents processed, merge all runs into one big index

This process has sequential access to runs and to final vocabulary and terms lists

Updating an inverted index

Prohibitive to update disk-based index directly for each new document

- 100’s of term lists to update, each might be many pages long.
- Very poor locality of access

Update alternatives

1. Rebuild
   - Works if document set small or index can be out of date
   - Might need old and new index at same time for availability
2. Multiple indexes
   - New documents indexed in memory, less recent index on disk
   - Periodically merge the two
3. Incremental update
   - Same as previous, but only do merge term-by-term
   - (For example, when reading a term list)

How do you throw hardware at the problem?

Need to partition index, distribute work

Two main choices

Term partitioning (TP): one node indexes all documents for a subset of terms

Document partitioning (DP): one node indexes a subset of documents for all terms

Tradeoffs

- TP – every node needs access to all documents
- DP – every node needs full vocabulary
- Communications
  - TP moves term lists or partial $S_{q,d}$'s
  - DP moves ordered lists of doc-ids and $S_{q,d}$'s
- Balancing – probably easier in DP: don't have to worry about "hot" search terms

What does Google do?

Document partitioning

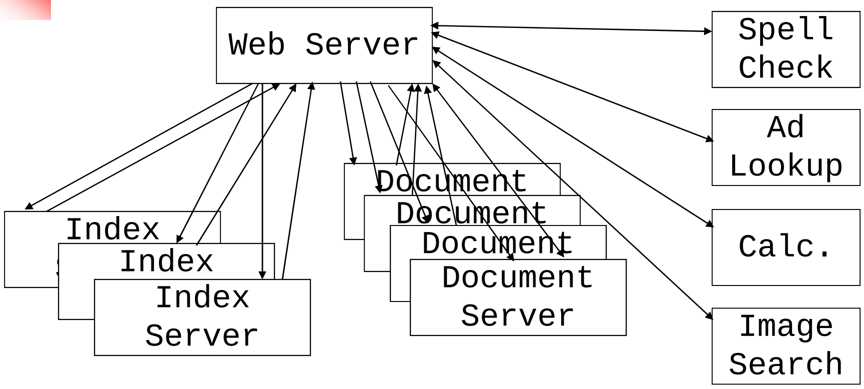
See: Barroso, Dean, Hoeltzle. “Web search for a planet: The Google cluster architecture.” *IEEE Micro* March-April 2003

Goal: High throughput per $

- Use commodity PCs (not the latest generation)
- Clusters of 1000s of nodes; software-based reliability

Main components

Ancillary services



Web Server

Index

Index

Index Server

Document

Document

Document

Document Server

Spell Check

Ad Lookup

Calc.

Image Search

etc.

Responsibilities

Web server: Receives user query, coordinates it, builds HTML result

Index is divided into *shards*, each with a random subset of documents

- Shard is support by a *pool* of machines
- To answer a query, need one machine in the pool of each shard
- Results get merged into ordered list of doc-ids

Responsibilities (2)

Doc servers also broken into shards, each supported by a pool

Retrieve snippets of documents for keyword-in-context part of result

Index is compressed. Index servers spend most of their cycles decompressing information.

Compressing an increasing list

Group exercise: You want to represent some initial sequence of primes. How many bits using small ints? What is a more space-efficient representation?

`2 3 5 7 11 13 17 19 23 29 31 37 41`

Compressing index structures

Doc-ids in a term list form an increasing sequence of integers

As do term positions in one document

Represent sequence of increasing integers by gaps

- Gives more smaller #'s than larger #'s
- Same is true of frequencies (more smaller than larger)

Want smaller #'s to take less space

Variable-length integer coding

Want shorter codes for smaller integers

Need to know where each codeword ends

One possibility is unary

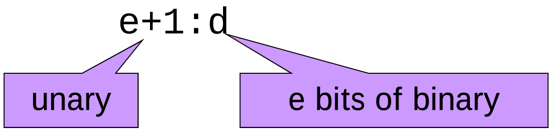
i is $1^{i-1}0$

3 is 110

Gamma code

Decompose I as $2^e + d$ (where e as large as possible)

Write as



e+1:d

unary

e bits of binary

19 is $16 + 3 = 2^4 + 3$

$\underline{11110}0011$

Exercise

- What is the code for

  74?

  2?
- What does 111111101001001 encode?
- Is every sequence of bits a gamma code for some integer?

Delta code

Again, consider i as $2^e + d$, but write e+1 in gamma code

For example

```
137 = 128 + 9 = 2⁷ + 9
e+1:0001001
7 = 2² + 3
full code is 110 11 0001001
```

Which code to use?

Zobel and Moffat give other codes

Which one to use (and with which parameters) depends on distribution of values you want represent

Byte-aligned codes

Might be expensive to manipulate individual bits; want codeword to be an even number of bytes

Use first bit of each byte to indicate if there are more bytes coming (0=no, 1=yes). Other bits are binary rep.

`10110 is 00010110`

`1001101 1110100 1101 is`

`11001101 11110100 00001101`

Entries in a term list

Coding a list of <d, $f_{d,t}$> entries

The d values are increasing, so compute "d-gaps", store as delta or gamma

Represent in $f_{d,t}$ (fixed) binary or gamma

Note that as doc-id grows, we expect the gaps to stay small

`<3,6><18,1><20,3><41,2> …`

`<2017,1><2019,3><2025,2> …`

Indexing common words

What is the indexing overhead for a common term?

I.e., does leaving out stopwords help?

Consider a word such as "and"

- It will appear in most documents
- Thus most gaps are 1 or 2
- Doc-ids in its inverted list take a small number of bits

Comparison to rare terms

Rare term has fewer document entries, but the gaps are much larger on average

So not necessarily a big difference in storage for common and rare terms

However, inverted lists for common terms require updating more often than those for rare terms

Term-position index

Recall: Entries have the form $<d;k;p_1,p_2,\ldots,p_k>$

- Have gaps between term positions, can use variable-length code for them
- Entries for common words do take more space
  - Index only the first m positions?
  - Store only in a document index?

+/- of compression

- Reduces space and transfer cost
- Can improve caching: Can keep more inverted lists in a given amount of main memory
- Incurs processing overhead
  - Decoding on use
  - Decoding & recoding on update
  - Generally compensated by lower disk costs