Plan

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

Index construction

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

Hardware basics

- Access to data in memory is much faster than access to data on disk.
- Disk seeks: No data is transferred from disk while the disk head is being positioned.
- Therefore: Transferring one large chunk of data from disk to memory is faster than transferring many small chunks.
- Disk I/O is block-based: Reading and writing of entire blocks (as opposed to smaller chunks).
- Block sizes: 8KB to 256 KB.

Hardware basics

- Servers used in IR systems now typically have several GB of main memory, sometimes tens of GB.
- Available disk space is several (2–3) orders of magnitude larger.
- Fault tolerance is very expensive: It’s much cheaper to use many regular machines rather than one fault tolerant machine.
Introduction to Information Retrieval

Hardware assumptions for this lecture

- **symbol** statistic value
  - s average seek time 5 ms = $5 \times 10^{-3}$ s
  - b transfer time per byte 0.02 µs = $2 \times 10^{-8}$ s
  - processor’ s clock rate 10⁶ s⁻¹
  - p low-level operation (e.g., compare & swap a word) 0.01 µs = $10^{-8}$ s
  - size of main memory several GB
  - size of disk space 1 TB or more

RCV1: Our collection for this lecture

- Shakespeare’ s collected works definitely aren’ t large enough for demonstrating many of the points in this course.
- The collection we’ ll use isn’ t really large enough either, but it’ s publicly available and is at least a more plausible example.
- As an example for applying scalable index construction algorithms, we will use the Reuters RCV1 collection.
- This is one year of Reuters newswire (part of 1995 and 1996)

A Reuters RCV1 document

**REUTERS**

*Extreme conditions create rare Antarctic clouds*

**Document ID:** 1

4.5 bytes per word token vs. 7.5 bytes per word type: why?

Reuters RCV1 statistics

- **symbol** statistic value
  - N documents 800,000
  - L avg. # tokens per doc 200
  - M terms (= word types) 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

4.5 bytes per word token vs. 7.5 bytes per word type: why?

**Document ID:** 2

Recall IIR 1 index construction

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

**Term** Doc #
- did 1
- killed 1
- the 1
- Capitol 1
- Brutus 1
- Caesar 1
- I 1
- it 1
- me 1
- so 1
- with 1
- Caesar 1
- Brutus 1
- killed 1

Key step

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

We focus on this sort step. We have 100M items to sort.
**Scaling index construction**

- In-memory index construction does not scale
  - Can’t stuff entire collection into memory, sort, then write back
  - How can we construct an index for very large collections?
  - Taking into account the hardware constraints we just learned about . . .
  - Memory, disk, speed, etc.

**Sort-based index construction**

- As we build the index, we parse docs one at a time.
  - While building the index, we cannot easily exploit compression tricks (you can, but much more complex)
  - The final postings for any term are incomplete until the end.
  - At 12 bytes per non-positional postings entry (term, doc, freq), demands a lot of space for large collections.
  - \( T = 100,000,000 \) in the case of RCV1
  - So … we can do this in memory in 2009, but typical collections are much larger. E.g., the New York Times provides an index of >150 years of newswire
  - Thus: We need to store intermediate results on disk.

**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.

**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 \( T=100M \) records

If every comparison took 2 disk seeks, and \( N \) items could be sorted with \( N \log_2 N \) comparisons, how long would this take?

**BSBI: Blocked sort-based Indexing**

(Sorting with fewer disk seeks)

- 12-byte (4+4+4) records (term, doc, freq).
  - These are generated as we parse docs.
  - Must now sort 100M such 12-byte records by term.
  - Define a block \( \approx 10M \) such records
    - Can easily fit a couple into memory.
    - Will have 10 such blocks to start with.
  - Basic idea of algorithm:
    - Accumulate postings for each block, sort, write to disk.
    - Then merge the blocks into one long sorted order.
Sorting 10 blocks of 10M records

- First, read each block and sort within:
  - Quicksort takes $2N \ln N$ expected steps
  - In our case $2 \times (10M \ln 10M)$ steps
- Exercise: estimate total time to read each block from disk and sort it.
- 10 times this estimate – gives us 10 sorted runs of 10M records each.
- Done straightforwardly, need 2 copies of data on disk
  - But can optimize this

BSIndexConstruction()

```
1 n ← 0
2 while (all documents have not been processed)
3   do n ← n + 1
4   block ← ParseNextBlock()
5   BSBI-Invert(block)
6   WriteBlockToDisk(block, f_n)
7   MergeBlocks(f_1, ..., f_n, f_merged)
```

How to merge the sorted runs?

- Can do binary merges, with a merge tree of $\log_2 10 = 4$ layers.
- During each layer, read into memory runs in blocks of 10M, merge, write back.

Remaining problem with sort-based algorithm

- Our assumption was: we can keep the dictionary in memory.
- We need the dictionary (which grows dynamically) in order to implement a term to termID mapping.
- Actually, we could work with term,docID postings instead of termID,docID postings . . .
- . . . but then intermediate files become very large.
  (We would end up with a scalable, but very slow index construction method.)

SPIMI: Single-pass in-memory indexing

- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.
- Key idea 2: Don’t sort. Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.
### Introduction to Information Retrieval

#### Sec. 4.3

**SPIMI-Invert**

```plaintext
SPIMI-INVERT(token_stream)
1  output_file = NEWFILE()
2  dictionary = NEWHASH()
3  while (free memory available)
4    do token = next(token_stream)
5      if term(token) ≠ dictionary
6        then postings_list = ADDTOLEXICON(dictionary, term(token))
7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
8      if full(postings_list)
9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10       ADDTOSTORAGE(postings_list, docID(token))
11      sorted_terms — SORTTERMS(dictionary)
12      WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

- Merging of blocks is analogous to BSBI.

#### Sec. 4.3

**SPIMI: Compression**

- Compression makes SPIMI even more efficient.
  - Compression of terms
  - Compression of postings
  - See next lecture

#### Sec. 4.4

**Distributed indexing**

- For web-scale indexing (don’t try this at home!):
  - must use a distributed computing cluster
- Individual machines are fault-prone
  - Can unpredictably slow down or fail
- How do we exploit such a pool of machines?

#### Sec. 4.4

**Web search engine data centers**

- Web search data centers (Google, Bing, Baidu) mainly contain commodity machines.
- Data centers are distributed around the world.
- Estimate: Google ~1 million servers, 3 million processors/cores (Gartner 2007)

#### Sec. 4.4

**Massive data centers**

- If in a non-fault-tolerant system with 1000 nodes, each node has 99.9% uptime, what is the uptime of the system?
- Answer: 63%
- Exercise: Calculate the number of servers failing per minute for an installation of 1 million servers.

#### Sec. 4.4

**Distributed indexing**

- Maintain a master machine directing the indexing job – considered “safe”.
- Break up indexing into sets of (parallel) tasks.
- Master machine assigns each task to an idle machine from a pool.
Parallel tasks
- We will use two sets of parallel tasks
  - Parsers
  - Inverters
- Break the input document collection into splits
- Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)

Parsers
- Master assigns a split to an idle parser machine
- Parser reads a document at a time and emits (term, doc) pairs
- Parser writes pairs into j par**ons
- Each par**on is for a range of terms’ first letters
  - (e.g., a-f, g-p, q-z) – here j = 3.
- Now to complete the index inversion

Inverters
- An inverter collects all (term,doc) pairs (= postings) for one term-partition.
- Sorts and writes to postings lists

MapReduce
- The index construction algorithm we just described is an instance of MapReduce.
- MapReduce (Dean and Ghemawat 2004) is a robust and conceptually simple framework for distributed computing ...
- ... without having to write code for the distribution part.
- They describe the Google indexing system (ca. 2002) as consisting of a number of phases, each implemented in MapReduce.

MapReduce
- Index construction was just one phase.
- Another phase: transforming a term-partitioned index into a document-partitioned index.
  - Term-partitioned: one machine handles a subrange of terms
  - Document-partitioned: one machine handles a subrange of documents
- As we’ll discuss in the web part of the course, most search engines use a document-partitioned index ... better load balancing, etc.
### Schema for index construction in MapReduce

- **Schema of map and reduce functions**
  - map: input $\rightarrow$ list(k, v) → reduce: (k, list(v)) → output

- **Instantiation of the schema for index construction**
  - map: collection $\rightarrow$ list(termID, docID)
  - reduce: (termID1, list(docID1), termID2, list(docID2), ...)$\rightarrow$(postings list1, postings list2, ...)

### Example for index construction

- **Map:**
  - d1 : C came, C c` ed.
  - d2 : C died.$\rightarrow$
    - <C,d1>, <came,d1>, <C,d1>, <c` ed,d1>, <C,d2>, <died,d2>

- **Reduce:**
  - (<C,(d1,d2,d1)>, <died,(d2)>, <came,(d1)>, <c` ed,(d1)>)$\rightarrow$(<C,(d1:2,d2:1)>, <died,(d2:1)>, <came,(d1:1)>, <c` ed,(d1:1)>)

### Dynamic indexing

- **Up to now, we have assumed that collections are static.**
- **They rarely are:**
  - Documents come in over time and need to be inserted.
  - Documents are deleted and modified.
- **This means that the dictionary and postings lists have to be modified:**
  - Postings updates for terms already in dictionary
  - New terms added to dictionary

### 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

### Issues with main and auxiliary indexes

- **Problem of frequent merges – you touch stuff a lot**
- **Poor performance during merge**
- **Actually:**
  - Merging of the auxiliary index into the main index is efficient if we keep a separate file for each postings list.
  - Merge is the same as a simple append.
  - But then we would need a lot of files – inefficient for OS.
- **Assumption for the rest of the lecture:** The index is one big file.
- **In reality:** Use a scheme somewhere in between (e.g., split very large postings lists, collect postings lists of length 1 in one file etc.)

### Logarithmic merge

- **Maintain a series of indexes, each twice as large as the previous one**
  - At any time, some of these powers of 2 are instantiated
  - Keep smallest ($Z_0$) in memory
  - Larger ones ($I_0, I_1, ...$) on disk
  - If $Z_0$ gets too big ($> n$), write to disk as $I_0$
  - or merge with $I_j$ (if $I_j$ already exists) as $Z_1$
  - Either write merge $Z_j$ to disk as $I_j$ (if no $I_j$)
  - Or merge with $I_j$ to form $Z_{j+1}$
Further issues with multiple indexes

- Collection-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 and invalidation bit vectors?
  - One possibility: ignore everything but the main index for such ordering
  - Will see more such statistics used in results ranking

Dynamic indexing at search engines

- All the large search engines now do dynamic indexing
  - Their indices have frequent incremental changes
    - News items, blogs, new topical web pages
    - Sarah Palin, …
  - But (sometimes/typically) they also periodically reconstruct the index from scratch
    - Query processing is then switched to the new index, and the old index is deleted

Other sorts of indexes

- Positional indexes
  - Same sort of sorting problem ... just larger
  - Building character 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 will arise repeatedly in parsing the docs — need efficient hashing 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
      - Only need to process each term once
Resources for today’s lecture

- Chapter 4 of IIR
- MG Chapter 5
- Original publication on MapReduce: Dean and Ghemawat (2004)