Overview

- Recap
- Introduction
- Index construction algorithms
- Distributed indexing
- Dynamic indexing

Recap → How to construct index?

- Computerese
  - term
  - document
  - docID
  - df (doc freq)
  - posting
  - posting list
  - inverted list
  - postings

Introduction

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

Index construction

- Step 1: Documents are parsed to extract words and saved with the Document ID.

  Doc 1: I did enact Julius Caesar I was killed in the Capitol, Brutus killed me.

  Doc 2: So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious.
Index construction

- Step 2: After all documents have been parsed, the inverted file is sorted by terms, and then written to disk.

We focus on this sort step. We have 100M items to sort.

Impact factors of index construction

- Many design decisions in information retrieval are based on the characteristics of hardware.
- We begin by reviewing hardware basics.

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 assumptions

<table>
<thead>
<tr>
<th>statistic</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>average seek time</td>
<td>5 ms</td>
</tr>
<tr>
<td>transfer time per byte</td>
<td>0.02 μs</td>
</tr>
<tr>
<td>processor's clock rate</td>
<td>$10^9$ s$^{-1}$</td>
</tr>
<tr>
<td>low-level operation</td>
<td>0.01 μs</td>
</tr>
<tr>
<td>e.g., compare &amp; swap a word</td>
<td></td>
</tr>
<tr>
<td>size of main memory</td>
<td>several GB</td>
</tr>
<tr>
<td>size of disk space</td>
<td>1 TB or more</td>
</tr>
</tbody>
</table>

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.

### 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 log₂N comparisons, how long would this take?

### Overview

- Recap
- Introduction
- Index construction algorithms
- Distributed indexing
- Dynamic indexing

### BSBI: Blocked sort-based Indexing

- 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 ~ 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.
Merge Blocks into One Block

postings

block

merged postings

disk

actually term should be termID

BSBI Index Construction Algorithm

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, fn)
7 MERGEBLOCKS(f1, ..., fn; f_merger)

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
  - 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

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.
- But it is more efficient to do a multi-way merge, where you are reading from all blocks simultaneously
- Providing you read decent-sized chunks of each block into memory and then write out a decent-sized output chunk, then you’re not killed by disk seeks

Problems 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.

SPIMI-Invert

```python
SPIMI-INVERT(token_stream):
  1. output_file = NEWFILE()
  2. dictionary = NEWHASH()
  3. while (free memory available)
    4. do token = next(token_stream)
    5. if not(token) if dictionary
    6. then postings.List = ADDToDictionary(dictionary, term(token))
    7. else postings.List = GETPOSTINGSLIST(dictionary, term(token))
    8. if full(postings.file)
    9. then postings.List = DOUBLEPOSTINGSLIST(dictionary, term(token))
  10. ADDToPOSTINGSLIST(postings.List, docID(token))
  11. sort_posts = SORTPOSTS(dictionary)
  12. WRITEBLOCKTOFILE(sort_posts, terms, dictionary, output_file)
  13. return output_file
```
- Merging of blocks is analogous to BSBI.

SPIMI: Compression

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

Overview

- Recap
- Introduction
- Index construction algorithms
- Distributed indexing
- Dynamic indexing

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?

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)
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* partitions
- Each partition 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.

Overview

- Recap
- Introduction
- Index construction algorithms
- Distributed indexing
- **Dynamic indexing**

Dynamic indexing

- 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

Our work

- Mimir
- SSD and information retrieval
- PCM and information retrieval

New challenge

- SSD VS HDD
- PCM VS main memory
- New storage devices will impacts on index construction and dynamic indexing

SSD PK HDD

SSD

HDD
SSD—NAND Flash

- SSD– NAND Flash Structure
  - page->block->plane->die (chip) ->package

1 page = (2k+64) B
1 block = 32 page
1 plane = 2048 block
1 die = 4 plane
1 package = 2 die

PCM

Read

Write

10ns | 100ns | 1us | 10us | 100us | 1ms