Index construction

Index construction

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

Corpus(语料库) for this lecture

- Number of docs n =1M = 1 000 000 (100万)
  - Each doc has 1000 terms
  - M : million
- Number of distinct terms m = 500K = 500 000 (50万)
- Postings entries N = 667M = 667 000 000 (667百万)

Recall index construction : Key step

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

Focus on this **sort** step

Have **667M** items to **sort**

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Index construction

- Build up the index
  - Cannot exploit compression(压缩) tricks
  - Parse docs one at a time.
  - Final postings for any term incomplete(不完整) until the end.

  - Actually, **can exploit compression** (压缩).
  - But this becomes a lot more complex.

自阅

- At 10-12 bytes per postings entry, demands several temporary gigabytes(千兆字节)

System parameters for design

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

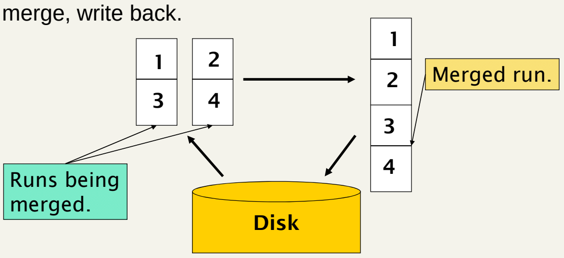
Bottleneck(瓶颈)

- 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*=667M records

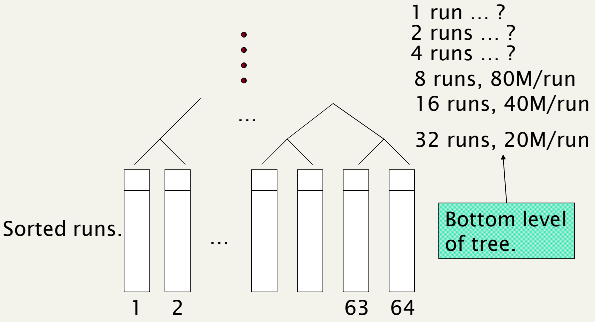
If every comparison took 2 disk seeks, and *N* items could be sorted with $N \log_2 N$ comparisons, how long would this take?

Merging 64 sorted runs

- Merge tree of $\log_2 64 = 6$ layers.
- During each layer, read into memory runs in blocks of 10M, merge, write back.



Merge tree



Merging 64 runs

- Time estimate for disk transfer:
- 6 x (64runs x 120MB x 10^{-6}sec) x 2 ~ 25hrs.

Disk block transfer time. Why is this an Overestimate?

Work out how these transfers are staged, and the total time for *merging*.

Layers in merge tree

Read + Write

Exercise - fill in this table

	Step	Time
1	64 initial quicksorts of 10M records each	
2	Read 2 sorted blocks for merging, write back	
3	Merge 2 sorted blocks	
4	Add (2) + (3) = time to read/merge/write	?
5	64 times (4) = total merge time	

Other indexing （研讨）

- **Large memory indexing**
- **Distributed indexing**
  - **Parallel tasks**
    - **Parsers**
    - **Inverters**
- **Dynamic indexing**

Large memory indexing

- Suppose
  - 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(蜘蛛) often interlaced(交错) with indexing.
  - Spidering bottlenecked by WAN speed and many other factors
    - more on this later.

Distributed indexing

- For web-scale indexing
  - must use a distributed computing cluster(分布的计算集群)
- Individual machines are fault-prone(故障)
  - Can unpredictably slow down or fail
- How do we exploit such a pool of machines?

Distributed indexing 2

- Maintain a *master* machine directing the indexing job
  - considered "safe".
- Break up (分解) indexing into sets of (parallel) tasks.
- Master machine assigns each task to an idle(空闲) machine from a pool.

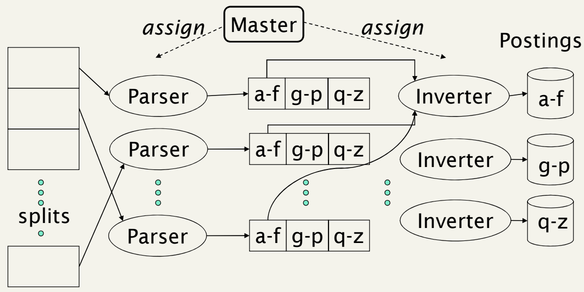
Parallel tasks

- Use two sets of parallel tasks
  - Parsers
  - Inverters
- Break the input document corpus into *splits*
  - Each split is a subset of documents
- Master
  - assigns a split to an idle parser machine

Parallel tasks 2

- Parser
  - reads a document at a time
  - emits (term, doc) pairs
- Parser
  - writes pairs into *j* partitions
- Each for a range of terms' first letters
  - (e.g., ***a-f, g-p, q-z***)
    - here *j*=3.
- Complete the index inversion

Parallel tasks : Data flow



Inverters

- Collect all (term, doc) pairs for a partition
- Sorts and writes to postings list
- Each partition contains a set of postings

Above process flow a special case of ***MapReduce***

Dynamic indexing

- Docs come in over time
  - postings updates for terms already in dictionary
  - new terms added to dictionary
- Docs get deleted

Simplest approach

- 1 Maintain "big" main index
- 2 New docs go into "small" auxiliary(辅助) index
- 3 Search across both, merge results
- 4 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
- How 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

Building positional(位置) indexes

- Still a sorting problem (but larger) ⇐ Why?
- Exercise:
  - given 1GB of memory
  - how adapt the block merge described earlier?

n-gram Indexes

Building *n*-gram indexes 1

- As text is parsed, enumerate(枚举) *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 2

- Once all (*n*-gram∈***term***) pairs have been enumerated
  - must sort for inversion(倒转).
- 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(压缩)

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 (海量存储需求)
  - feasible for large web service installations (大型网络服务设施)
  - less so for commercial usage (商业用途)
    - where query loads (查询负载) are lighter