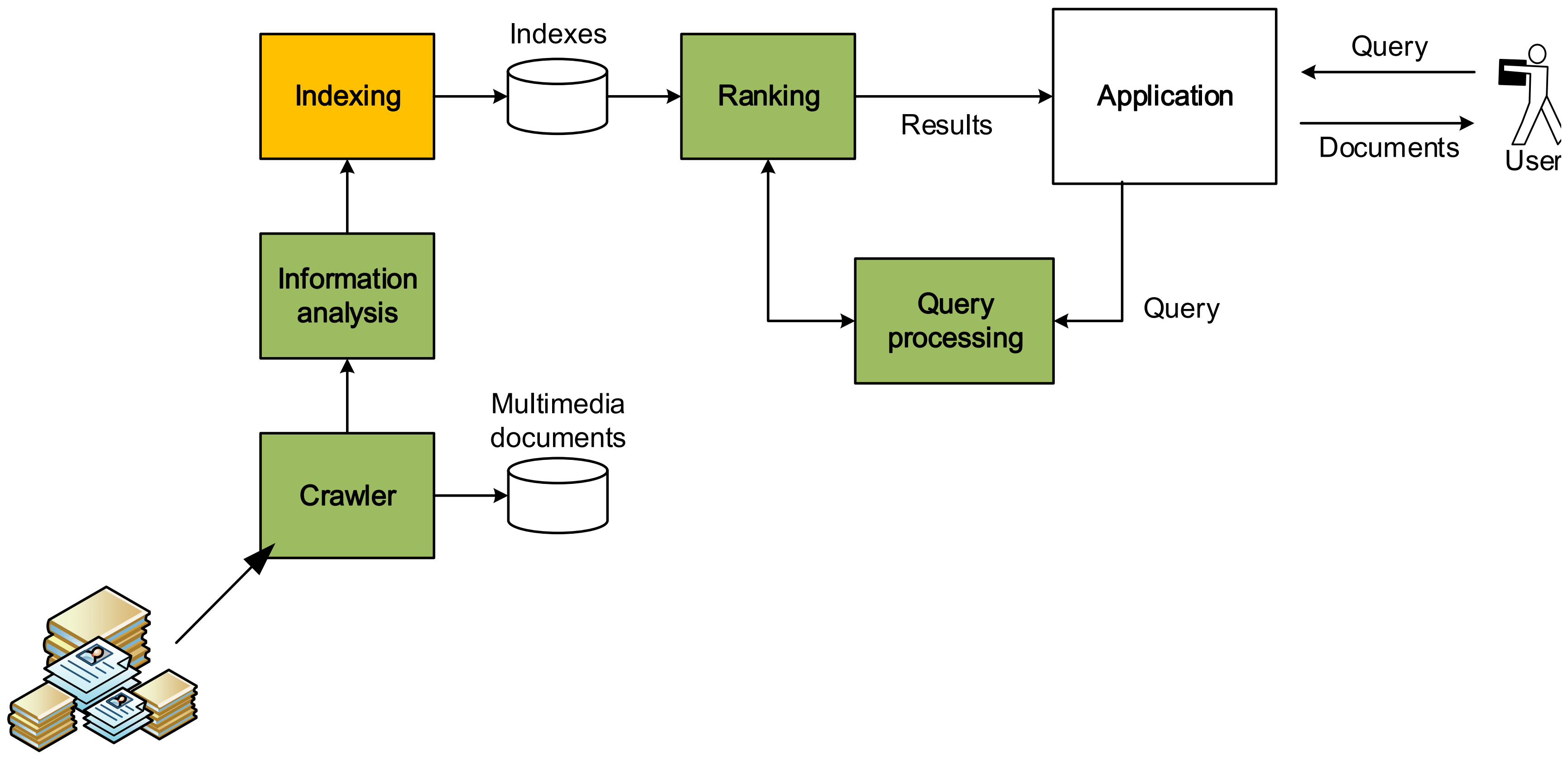
Index Construction

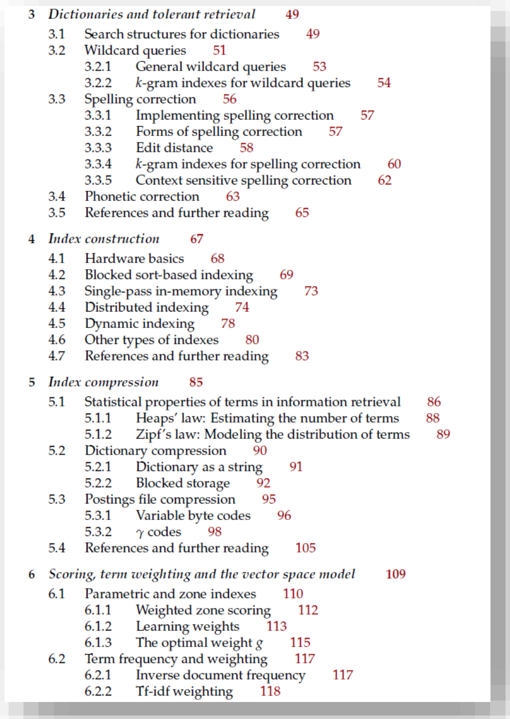
Dictionary, postings, scalable indexing, dynamic indexing

Web Search

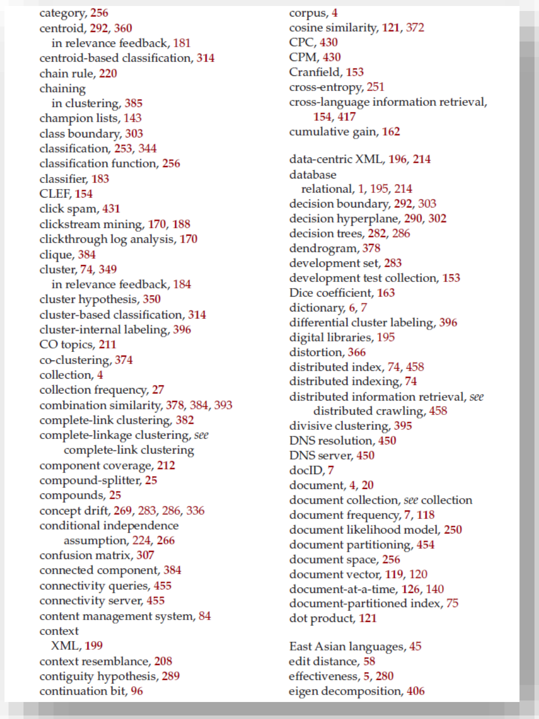
Overview


Indexing
Indexes
Ranking
Results
Application
Query
Documents
User
Information analysis
Query processing
Query
Multimedia documents
Crawler

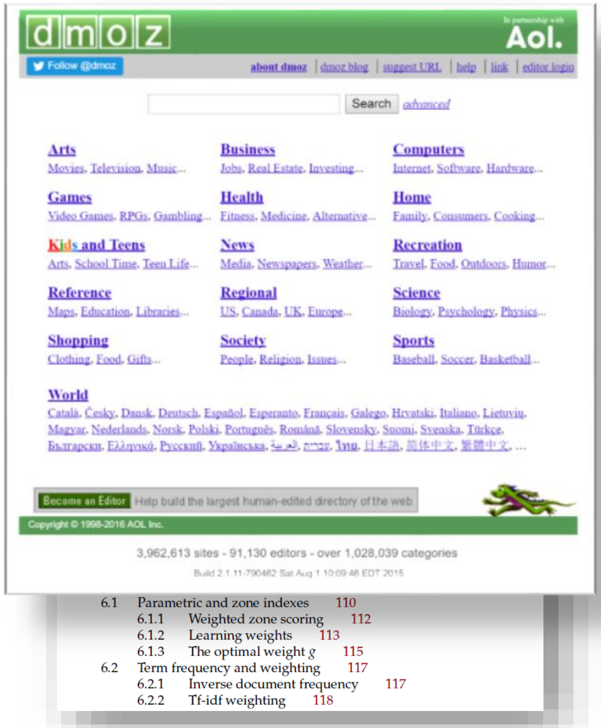
Indexing by similarity



Indexing by terms



Indexing by similarity


dmoz

In partnership with Aol.

Follow @dmoz

about dmoz | dmoz blog | suggest URL | help | link | editor login

Search advanced

Arts Movies, Television, Music...

Business Jobs, Real Estate, Investing...

Computers Internet, Software, Hardware...

Games Video Games, RPGs, Gambling...

Health Fitness, Medicine, Alternative...

Home Family, Consumers, Cooking...

Kids and Teens Arts, School Time, Teen Life...

News Media, Newspapers, Weather...

Recreation Travel, Food, Outdoors, Humor...

Reference Maps, Education, Libraries...

Regional US, Canada, UK, Europe...

Science Biology, Psychology, Physics...

Shopping Clothing, Food, Gifts...

Society People, Religion, Issues...

Sports Baseball, Soccer, Basketball...

World Català, Česky, Dansk, Deutsch, Español, Esperanto, Français, Galego, Hrvatski, Italiano, Lietuvių, Magyar, Nederlands, Norsk, Polski, Português, Română, Slovensky, Suomi, Svenska, Türkçe, Български, Ελληνικά, Русский, Українська, عربية, עברית, ไทย, 日本語, 简体中文, 繁體中文, ...

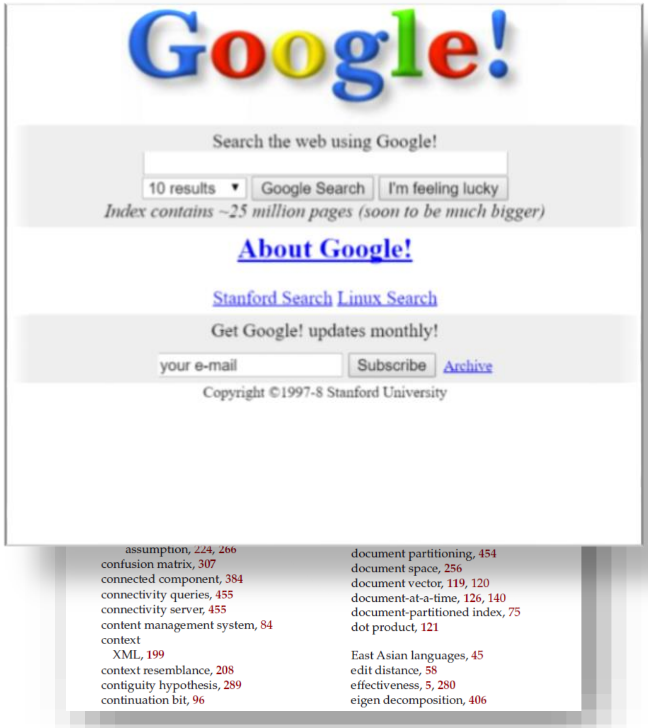
Become an Editor Help build the largest human-edited directory of the web

Copyright © 1998-2016 AOL Inc.

3,962,613 sites - 91,130 editors - over 1,028,039 categories

Build 2.1.11-790462 Sat Aug 1 10:09:48 EDT 2015

Indexing by terms


Google!

Search the web using Google!

10 results ▾ Google Search I'm feeling lucky

Index contains ~25 million pages (soon to be much bigger)

About Google!

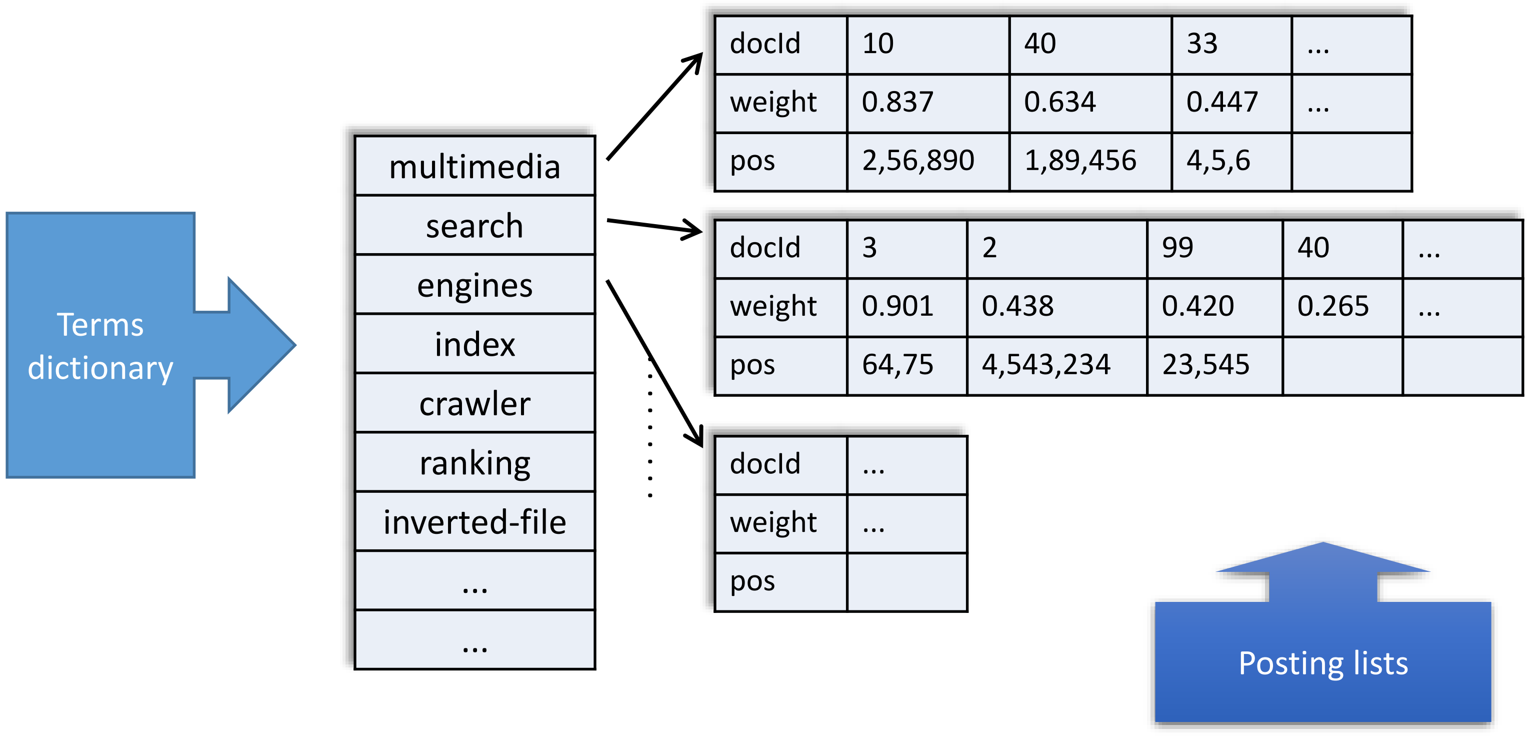
Stanford Search Linux Search

Get Google! updates monthly!

your e-mail Subscribe Archive

Copyright ©1997-8 Stanford University

Text based inverted file index



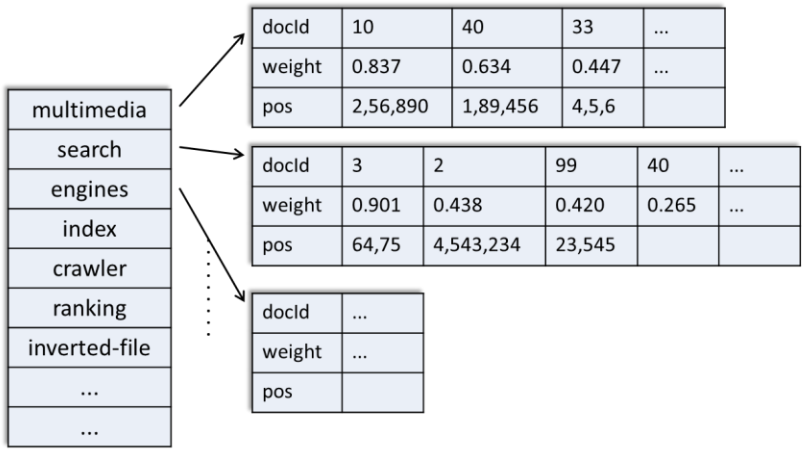
docId	10	40	33	...
weight	0.837	0.634	0.447	...
pos	2,56,890	1,89,456	4,5,6	

docId	3	2	99	40	...
weight	0.901	0.438	0.420	0.265	...
pos	64,75	4,543,234	23,545		

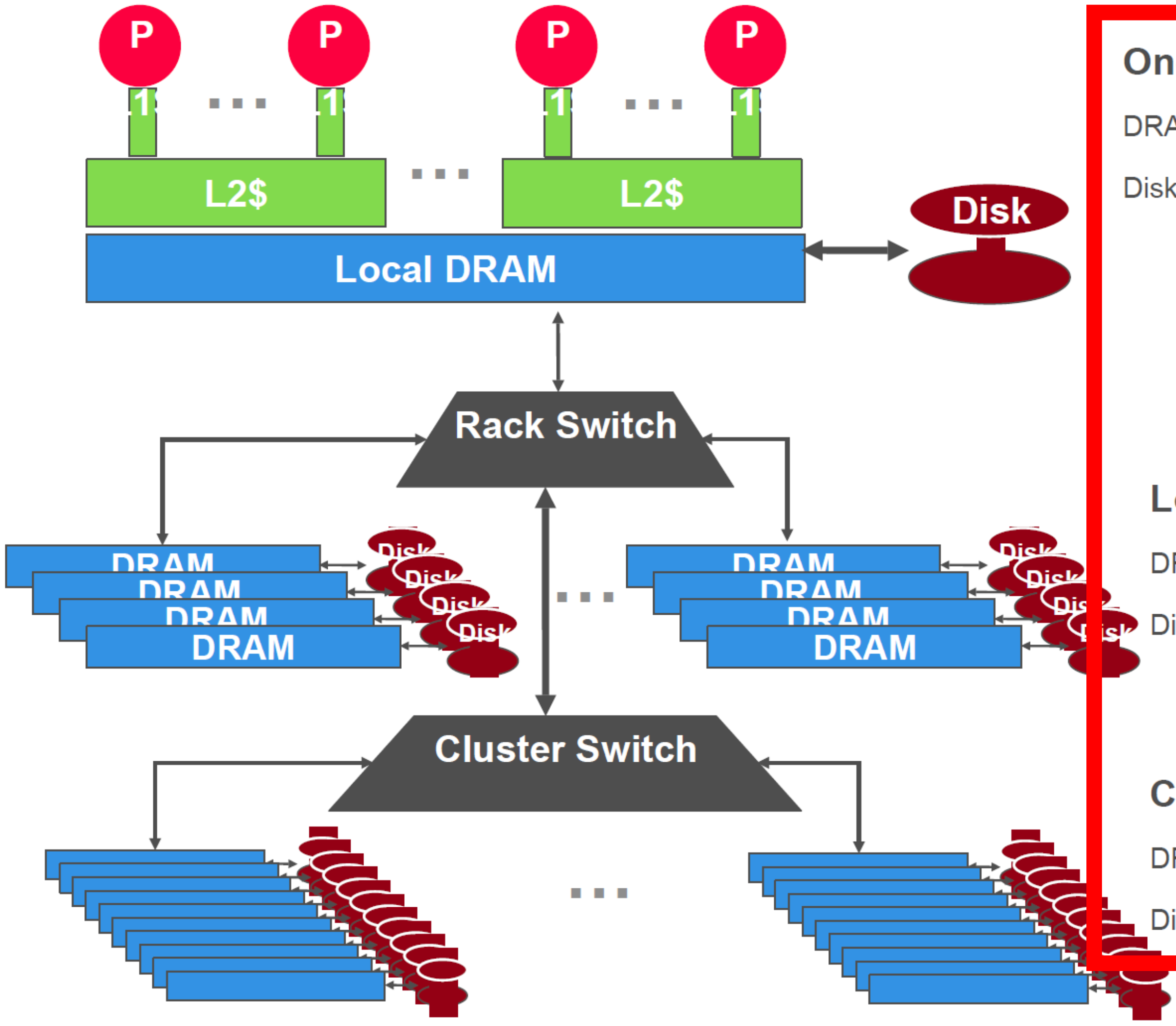
docId	...
weight	...
pos	

Index construction

- How to compute the dictionary?
- How to compute the posting lists?
- How to index billions of documents?



Architectural view of the storage hierarchy



One server

DRAM: 16GB, 100ns, 20GB/s

Disk: 2TB, 10ms, 200MB/s

Local rack (80 servers)

DRAM: 1TB, 300us, 100MB/s

Disk: 160TB, 11ms, 100MB/s

Cluster (30+ racks)

DRAM: 30TB, 500us, 10MB/s

Disk: 4.80PB, 12ms, 10MB/s

Google

Some numbers

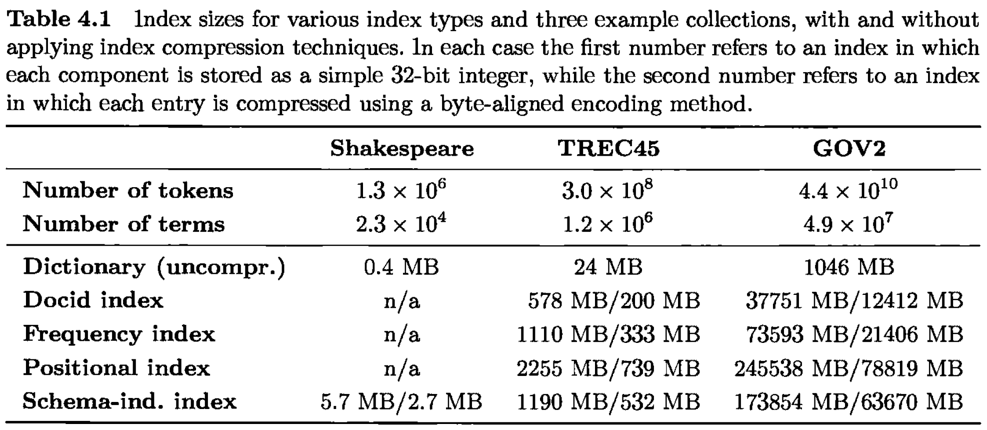


Table 4.1 Index sizes for various index types and three example collections, with and without applying index compression techniques. In each case the first number refers to an index in which each component is stored as a simple 32-bit integer, while the second number refers to an index in which each entry is compressed using a byte-aligned encoding method.

	Shakespeare	**TREC45**	**GOV2**
Number of tokens	1.3×10^6	3.0×10^8	4.4×10^{10}
Number of terms	2.3×10^4	1.2×10^6	4.9×10^7
Dictionary (uncompr.)	0.4 MB	24 MB	1046 MB
Docid index	n/a	578 MB/200 MB	37751 MB/12412 MB
Frequency index	n/a	1110 MB/333 MB	73593 MB/21406 MB
Positional index	n/a	2255 MB/739 MB	245538 MB/78819 MB
Schema-ind. index	5.7 MB/2.7 MB	1190 MB/532 MB	173854 MB/63670 MB

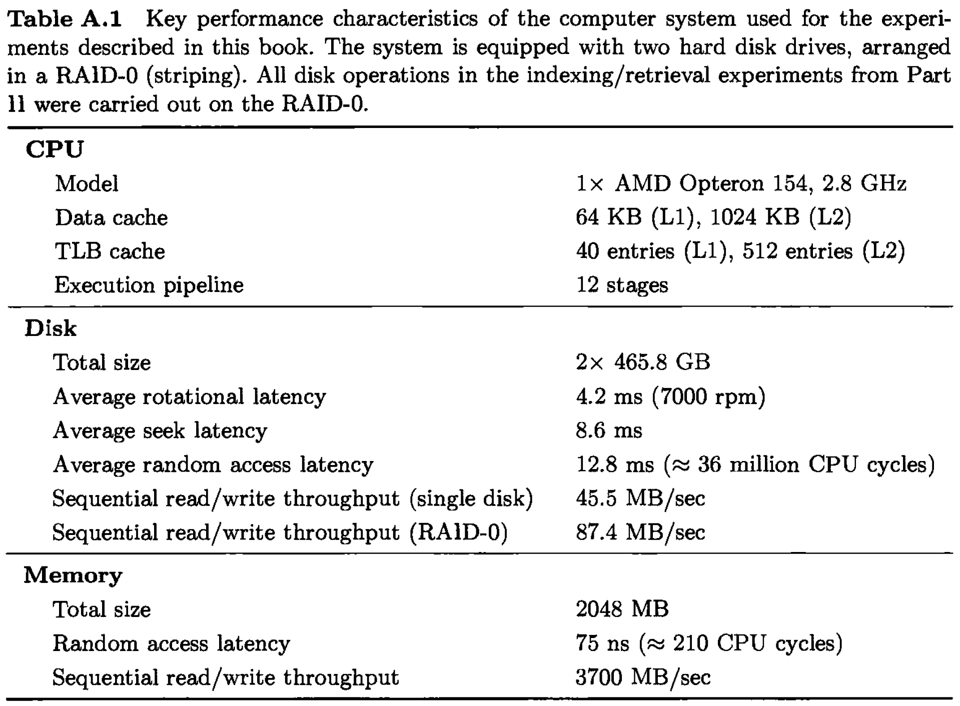
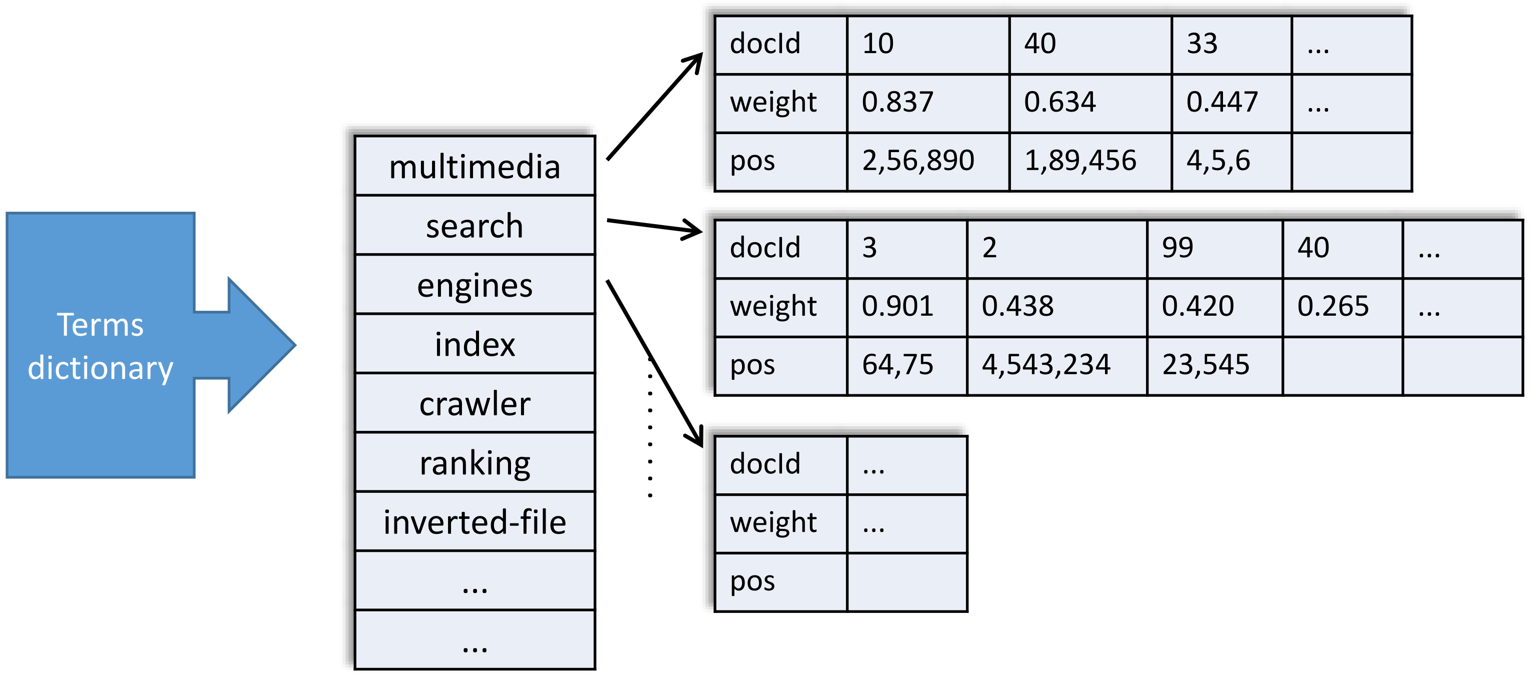


Table A.1 Key performance characteristics of the computer system used for the experiments described in this book. The system is equipped with two hard disk drives, arranged in a RAID-0 (striping). All disk operations in the indexing/retrieval experiments from Part II were carried out on the RAID-0.

CPU	
Model	1× AMD Opteron 154, 2.8 GHz
Data cache	64 KB (L1), 1024 KB (L2)
TLB cache	40 entries (L1), 512 entries (L2)
Execution pipeline	12 stages
Disk	
Total size	2× 465.8 GB
Average rotational latency	4.2 ms (7000 rpm)
Average seek latency	8.6 ms
Average random access latency	12.8 ms ($\approx$ 36 million CPU cycles)
Sequential read/write throughput (single disk)	45.5 MB/sec
Sequential read/write throughput (RAID-0)	87.4 MB/sec
Memory	
Total size	2048 MB
Random access latency	75 ns ($\approx$ 210 CPU cycles)
Sequential read/write throughput	3700 MB/sec

Text based inverted file index



multimedia
search
engines
index
crawler
ranking
inverted-file
...
...

docId	10	40	33	...
weight	0.837	0.634	0.447	...
pos	2,56,890	1,89,456	4,5,6	

docId	3	2	99	40	...
weight	0.901	0.438	0.420	0.265	...
pos	64,75	4,543,234	23,545		

docId	...
weight	...
pos	

Sort-based index construction

Sec. 4.2

- As we build the index, we parse docs one at a time.
- 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 now, 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.**

Use the same algorithm for disk? Sec. 4.2

- 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.

BSBI: Blocked sort-based Indexing

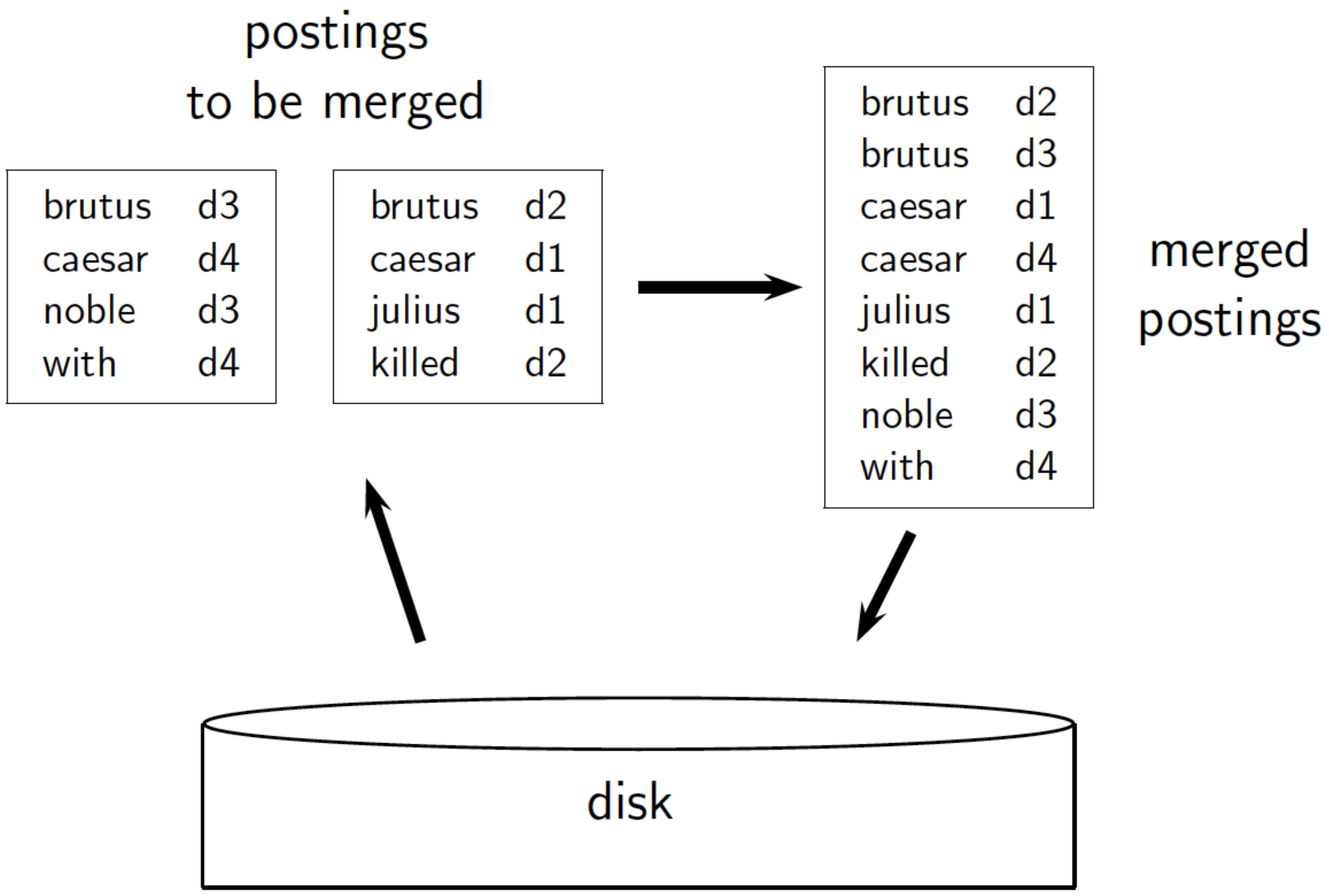
Sec. 4.2

- 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:
  - Compute postings dictionary
  - Accumulate postings for each block, sort, write to disk.
  - Then merge the blocks into one long sorted order.



brutus	d3
caesar	d4
noble	d3
with	d4

brutus	d2
caesar	d1
julius	d1
killed	d2

brutus	d2
brutus	d3
caesar	d1
caesar	d4
julius	d1
killed	d2
noble	d3
with	d4

Sorting 10 blocks of 10M records

Sec. 4.2

- First, read each block and sort within:
  - Quicksort takes 2*N* ln *N* expected steps
  - In our case 2 x (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

BSBI: Blocked sort-based Indexing

Sec. 4.2

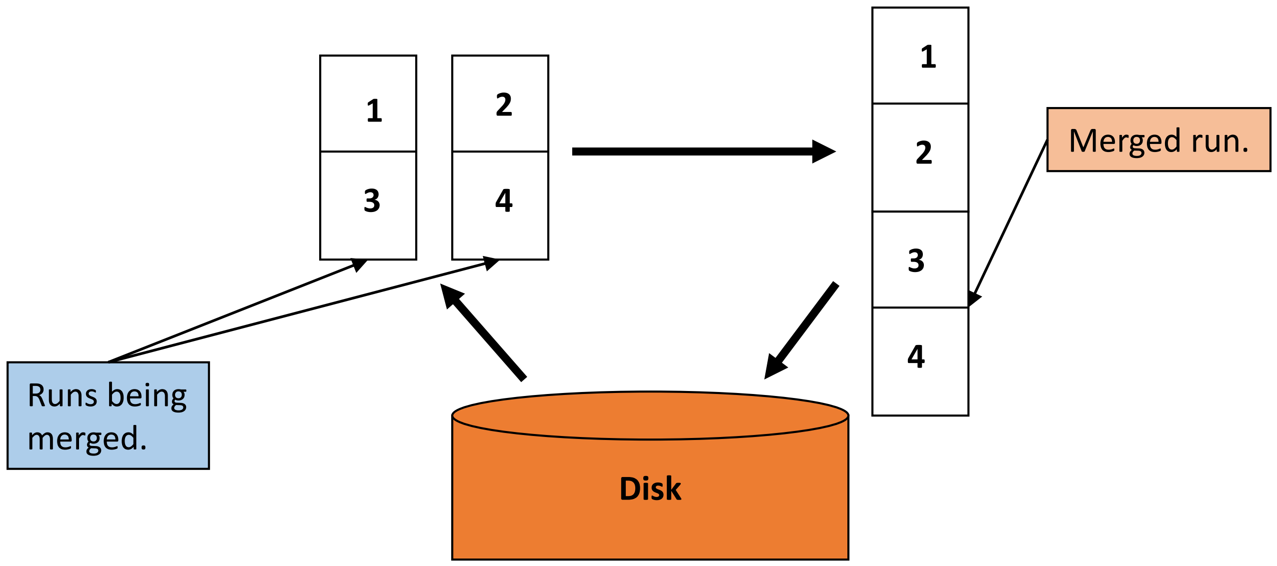
```
BSBIIndexConstruction()
1  $n \leftarrow 0$
2  while (all documents have not been processed)
3  do $n \leftarrow n + 1$
4      $block \leftarrow$ ParseNextBlock()
5      BSBI-Invert($block$)
6      WriteBlockToDisk($block, f_n$)
7  MergeBlocks($f_1, \ldots, f_n; f_{\text{merged}}$)
```

Notes:
4: Parse and accumulate all termID-docID pairs
5: Collect all termID-docID with the same termID into the same postings list
7: Opens all blocks and keep a small reading buffer for each block. Merge into the final file. (Avoid seeks, read/write sequentially)

How to merge the sorted runs?

Sec. 4.2

- 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.



Dictionary

- The size of document collections exposes many poor software designs
  - The distributed scale also exposes such design flaws

- The choice of the data-structures has great impact on overall system performance

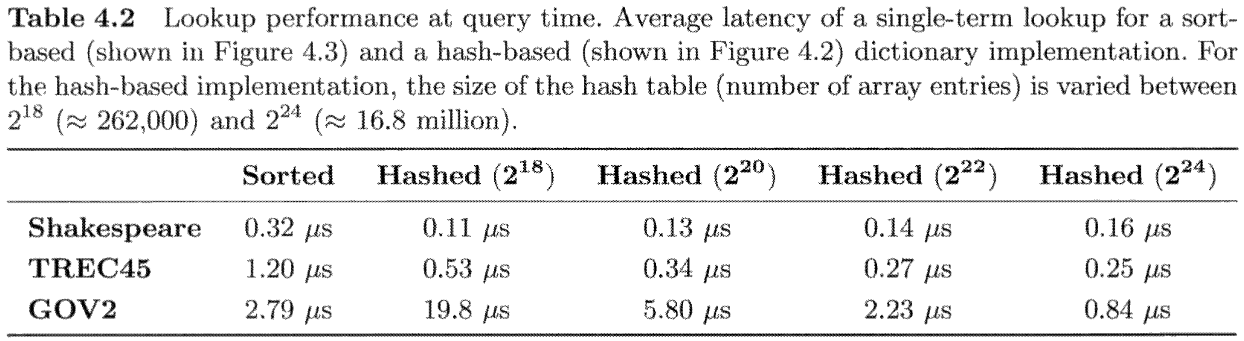


Table 4.2 Lookup performance at query time. Average latency of a single-term lookup for a sort-based (shown in Figure 4.3) and a hash-based (shown in Figure 4.2) dictionary implementation. For the hash-based implementation, the size of the hash table (number of array entries) is varied between 2^{18} ($\approx$ 262,000) and 2^{24} ($\approx$ 16.8 million).

	Sorted	Hashed (2^{18})	Hashed (2^{20})	Hashed (2^{22})	Hashed (2^{24})
Shakespeare	0.32 μs	0.11 μs	0.13 μs	0.14 μs	0.16 μs
TREC45	1.20 μs	0.53 μs	0.34 μs	0.27 μs	0.25 μs
GOV2	2.79 μs	19.8 μs	5.80 μs	2.23 μs	0.84 μs

To hash or not to hash?

The small look-up table of the Shakespeare collection is so small that it fits in the CPU cache.

What about wildcard queries?

Lookup table construction strategies

- Insight: 90% of terms occur only 1 time

- Insert at the back
  - Insert terms at the back of the chain as they occur in the collection, i.e., frequent terms occur first, hence they will be at the front of the chain

- Move to the front:
  - Move to the front of the chain the last acessed term.

Indexing time dictionary

- The bulk of the dictionary's lookup load stems from a rather small set of very frequent terms.
  - In a hashtable, those terms should be at the front of the chains

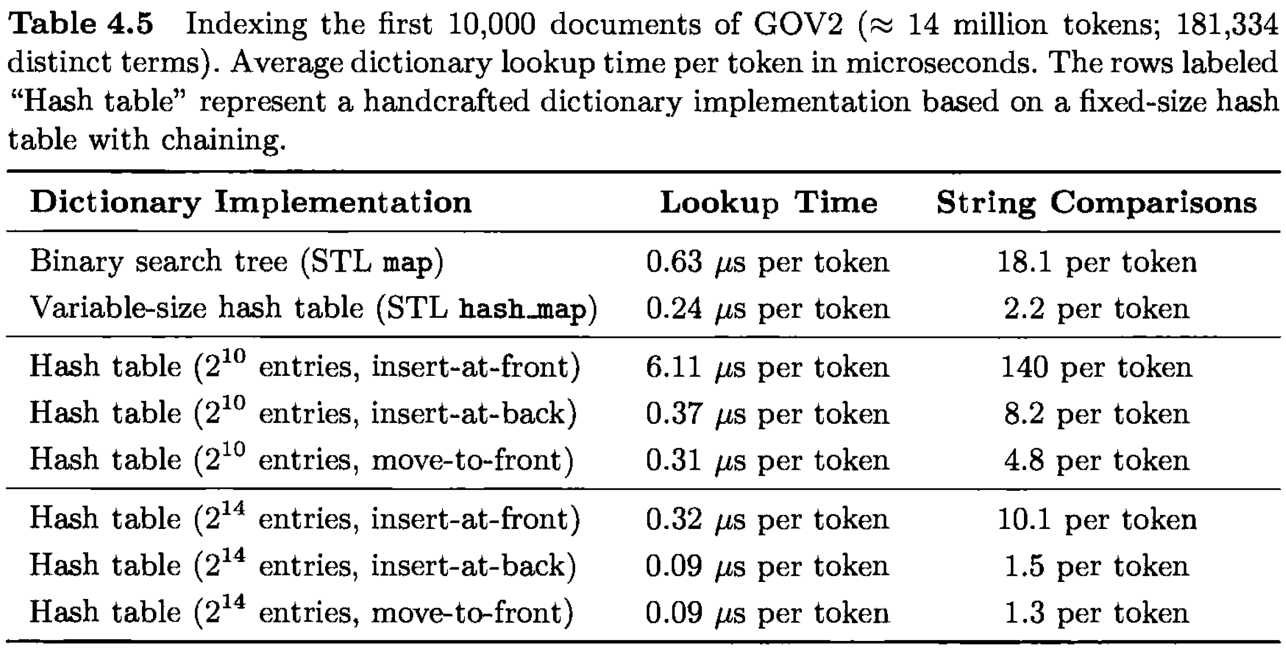


Table 4.5 Indexing the first 10,000 documents of GOV2 (≈ 14 million tokens; 181,334 distinct terms). Average dictionary lookup time per token in microseconds. The rows labeled "Hash table" represent a handcrafted dictionary implementation based on a fixed-size hash table with chaining.

Dictionary Implementation	**Lookup Time**	**String Comparisons**
Binary search tree (STL `map`)	0.63 μs per token	18.1 per token
Variable-size hash table (STL `hash_map`)	0.24 μs per token	2.2 per token
Hash table (2^{10} entries, insert-at-front)	6.11 μs per token	140 per token
Hash table (2^{10} entries, insert-at-back)	0.37 μs per token	8.2 per token
Hash table (2^{10} entries, move-to-front)	0.31 μs per token	4.8 per token
Hash table (2^{14} entries, insert-at-front)	0.32 μs per token	10.1 per token
Hash table (2^{14} entries, insert-at-back)	0.09 μs per token	1.5 per token
Hash table (2^{14} entries, move-to-front)	0.09 μs per token	1.3 per token

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.

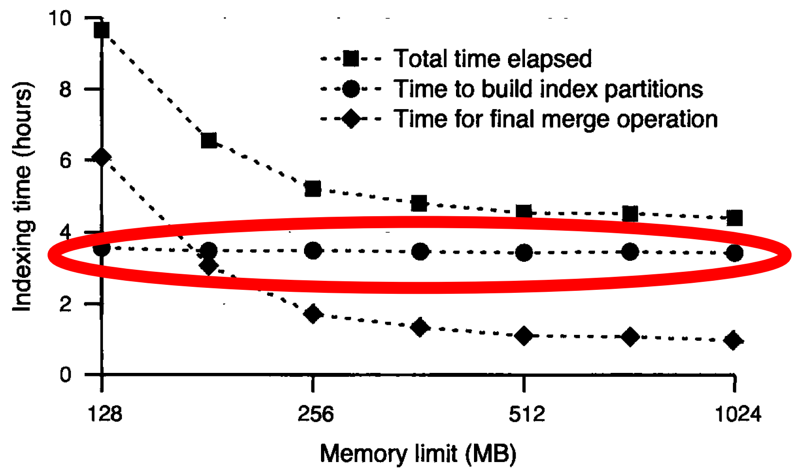
SPIMI-Invert

```
SPIMI-INVERT(token_stream)
output_file = NEWFILE()
dictionary = NEWHASH()
while (free memory available)
do token ← next(token_stream)
    if term(token) $\notin$ dictionary
      then postings_list = ADDTODICTIONARY(dictionary, term(token))
      else postings_list = GETPOSTINGSLIST(dictionary, term(token))
    if full(postings_list)
      then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
    ADDTOPOSTINGSLIST(postings_list, docID(token))
sorted_terms ← SORTTERMS(dictionary)
WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
return output_file
```

Experimental comparison

- The index construction is mainly influenced by the available memory

- Each part of the indexing process is affected differently
  - Parsing
  - Index inversion
  - Indexes merging



- For web-scale indexing must use a distributed computing cluster

How do we exploit such a pool of machines?

Distributed document parsing

Sec. 4.4

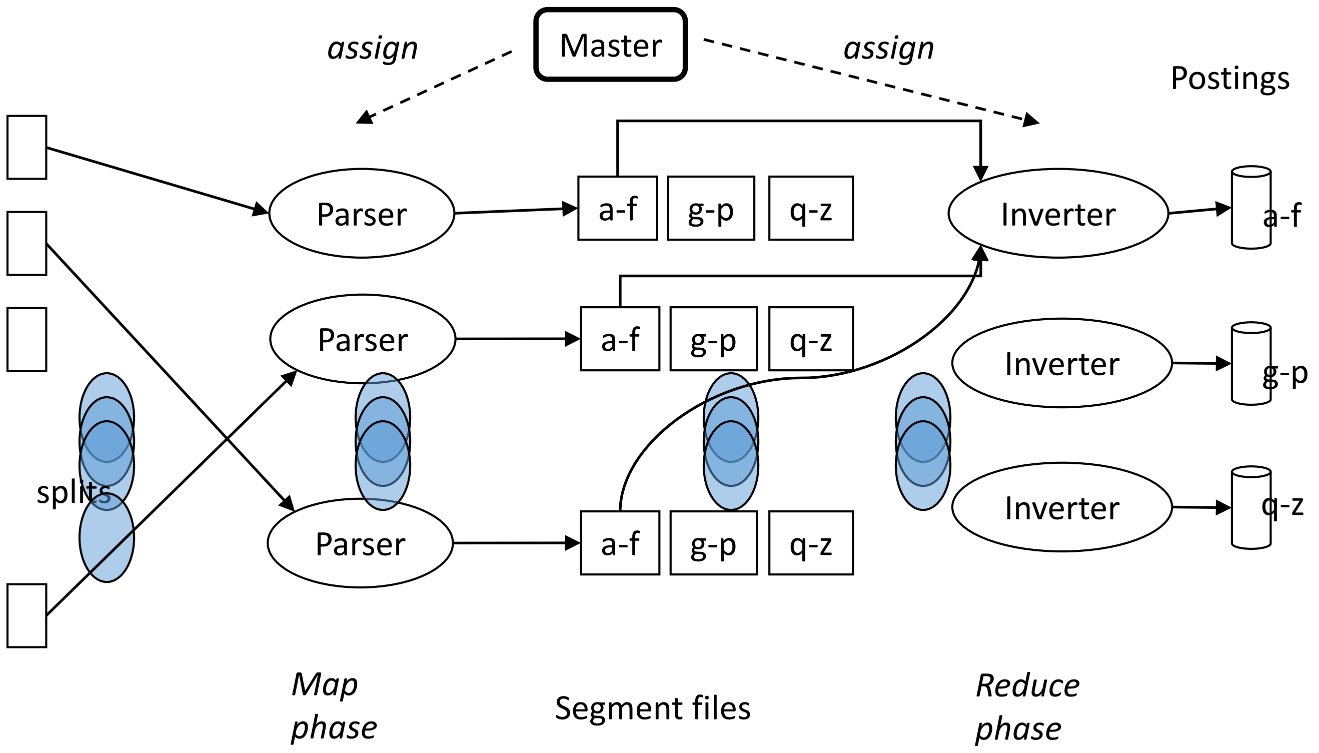
- Maintain a *master* machine directing the indexing job.
- Break up indexing into sets of parallel tasks:
  - Parsers
  - Inverters
- Break the input document collection into *splits*
  - Each split is a subset of documents (corresponding to blocks in BSBI/SPIMI)
- Master machine assigns each task to an idle machine from a pool.

Parallel tasks

- 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

Data flow


Master
assign
assign
Postings
Parser
Parser
Parser
a-f
g-p
q-z
a-f
g-p
q-z
a-f
g-p
q-z
Inverter
Inverter
Inverter
a-f
g-p
q-z
splits
Map phase
Segment files
Reduce phase

MapReduce

Sec. 4.4

- 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.

Google data centers

Sec. 4.4

https://www.youtube.com/watch?v=zRwPSFpLX8I

- Google data centers mainly contain commodity machines.
- Data centers are distributed around the world.
- Estimate: a total of 1 million servers, 3 million processors/cores (Gartner 2007)
- Estimate: Google installs 100,000 servers each quarter.
  - Based on expenditures of 200–250 million dollars per year
- This would be 10% of the computing capacity of the world!?!

The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)

~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours me back)

~1 rack-move (plenty of warning, ~500-1000 machines po hours)

~1 network rewiring (rolling ~5% of machines do

~20 rack failures (40-80 machines instan ck)

~5 racks go wonky (40-80 mach

~8 network maintenan ndom connectivity losses)

~12 router rel vips for a couple minutes)

~3 router failu pull traffic for an hour)

~dozens of mino blips for dns

~1000 individual chine failures

~thousands of hard drive failures

slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

Things will crash.
Deal with it!!

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

Sec. 4.5

- 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 O/S.
- 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

Sec. 4.5

- Maintain a series of indexes, each twice as large as the previous one.
- 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_0 (if I_0 already exists) as Z_1
- Either write merge Z_1 to disk as I_1 (if no I_1)
    - Or merge with I_1 to form Z_2
- etc.

```
LMergeAddToken($indexes, Z_0, token$)
Z_0 ← Merge($Z_0, \{token\}$)
if $|Z_0| = n$
  then for $i ← 0$ to $\infty$
         do if $I_i \in indexes$
              then $Z_{i+1}$ ← Merge($I_i, Z_i$)
                     ($Z_{i+1}$ is a temporary index on disk.)
                   $indexes ← indexes - \{I_i\}$
              else $I_i ← Z_i$     ($Z_i$ becomes the permanent index $I_i$.)
                   $indexes ← indexes \cup \{I_i\}$
                   Break
         $Z_0 ← \emptyset$

LogarithmicMerge()
$Z_0 ← \emptyset$     ($Z_0$ is the in-memory index.)
$indexes ← \emptyset$
while true
do LMergeAddToken($indexes, Z_0$, getNextToken())
```

Logarithmic merge

- Auxiliary and main index: index construction time is $O(T^2)$ as each posting is touched in each merge.

- Logarithmic merge:
    - Each posting is merged O(log T) times, so complexity is O(T log T)

- So logarithmic merge is much more efficient for index construction

- But query processing now requires the merging of O(log T) indexes
    - Whereas it is O(1) if you just have a main and auxiliary index

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

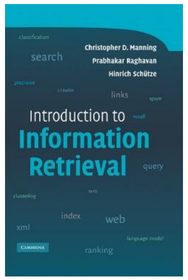
Dynamic indexing at search engines

Sec. 4.5

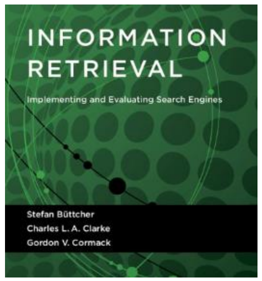
- All the large search engines now do dynamic indexing
- Their indices have frequent incremental changes
  - News items, blogs, new topical web pages
    - Sarah Palin, …
- But (sometimes/typically) they also periodically reconstruct the index from scratch
  - Query processing is then switched to the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

Summary

- Indexing
- Dictionary data structures
- Scalable indexing (BSBI, SPIMI)
- Distributed document parsing
- Dynamic indexing



Chapter 4



Chapter 4 (dictionary data structures)