

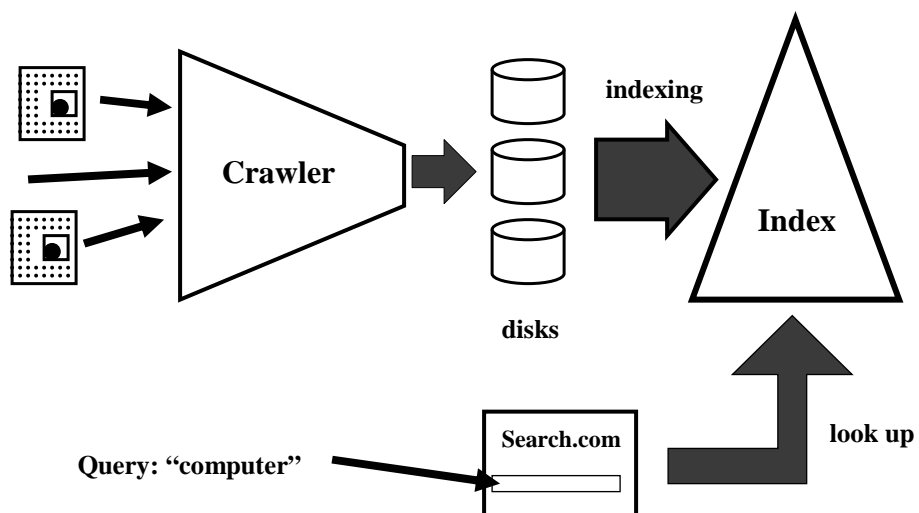
An Introduction of Full-Text Index for the Web

Xiaohui Long

xlong@cis.poly.edu

<http://cis.poly.edu/~xlong>

Search Engine Architecture:



Outline

- **Inverted index**
- **Building an inverted index**
- **Some additional issues**

Inverted Index

- **Inverted index**
 - A set of inverted lists
- **Inverted list**
 - One inverted list for each word
 - Consists of postings (doc id, pos)
 - Usually sorted by doc id and pos
- **Posting**
 - One occurrence of a word
 - Better to include positions
 - The pages have the search terms near each other is much more likely to relevant to the query
 - Phrase searching
 - Increase size of index significantly

Example:

doc1: "Bob reads a book"
doc2: "Alice likes Bob"
doc3: "book"

inverted index:

a: {(1, 3)}
alice: {(2, 1)}
bob: {(1, 1), (2, 3)}
book: {(1, 4), (3, 1)}
likes: {(2, 2)}
reads: {(1, 2)}

Inverted Index (Cont'd)

- **Lexicon**

- Set of all words in the documents collection
- Statistics for ranking
- Case folding
- Stemming: “run = runs = running”
 - Try to detect these words based on “rules”
 - Rules are language-dependant and complicate
- Stop words
 - common words, like “the”, “a”
 - ignore? save space, but maybe better not!
 - “to be or not to be”, “the who”, ...

Example:

doc1: “Bob reads a book”
doc2: “Alice likes Bob”
doc3: “book”

a: {(1, 3)}
alice: {(2, 1)}
bob: {(1, 1), (2, 3)}
book: {(1, 4), (3, 1)}
likes: {(2, 2)}
reads: {(1, 2)}

Context

- **Title**
- **Bold, italic**
- **Font size**
- **Anchor text**
 - ` search engine course `

Querying Inverted Index

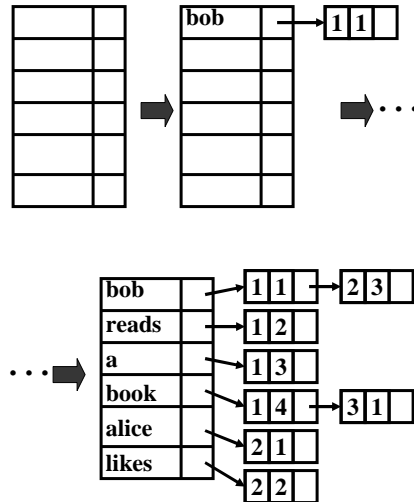
- Query: “hello world”
- Looking for pages contain both “hello” and “world”
- Look up the lexicon and get those two inverted lists: one for “hello”, one for “world”
- Merge these two inverted lists to find common documents
- Calculate the ranks for those documents and return the top k results to the user

Building an Inverted Index

A Naive Approach

- a) create an empty dynamic dictionary data structure (e.g. hash table) in main memory;
- b) scan through all the documents, for every word encountered:
 - i. create an entry in the dictionary, if the word does not exist;
 - ii. Insert (doc id, pos) into the inverted list corresponding to the word;
- c) traverse the dictionary and dump the inverted index on disks.

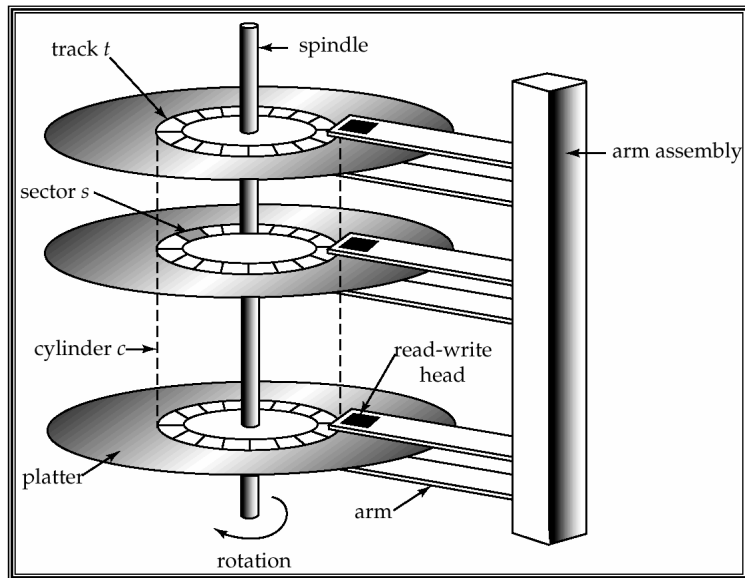
doc1: "Bob reads a book"
doc2: "Alice likes Bob"
doc3: "book"



Why It Doesn't Work

- With data size increasing, it will be too large to fit all the index in main memory!
- Some numbers from [Moffat&Bell]: for only 5 million pages, more than 500 million postings will be generated. That means more than 5GB main memory will be needed!
- Compression can help to reduce the space, but does NOT solve the problem.
- Have to use lower level storage device -- hard disk.

Hard Disk



Hard Disk (Cont'd)

- **Read-write head**
 - Positioned very close to the platter surface (almost touching it)
 - Reads or writes magnetically encoded information.
- **Surface of platter divided into circular tracks**
 - Over 16,000 tracks per platter on typical hard disks
- **Each track is divided into sectors.**
 - A sector is the smallest unit of data that can be read or written.
 - Sector size typically 512 bytes
 - Typical sectors per track: 200 (on inner tracks) to 400 (on outer tracks)

Hard Disk (Cont'd)

- **To read/write a sector**
 - Disk arm swings to position head on right track
 - Platter spins continually; Data is read/written as sector passes under head
- **Head-disk assemblies**
 - Multiple disk platters on a single spindle (typically 2 to 4)
 - One head per platter, mounted on a common arm
- **Cylinder i consists of i^{th} track of all the platters**

Performance Measure of Hard Disk

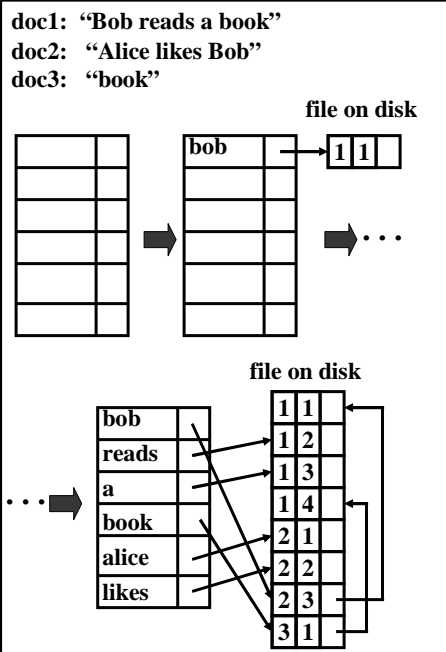
- **Access time – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:**
 - **Seek time – time it takes to reposition the arm over the correct track.**
 - Average seek time is 1/2 the worst case seek time.
 - 4 to 10 milliseconds on typical disks
 - **Rotational latency – time it takes for the sector to be accessed to appear under the head.**
 - Average latency is 1/2 of the worst case latency.
 - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
- **Data-transfer rate – the rate at which data can be retrieved from or stored to the disk.**
 - 4 to 8 MB per second is typical

Random Accesses

- Average seek time: 4.8ms
- Average rotational latency: 4.2ms (7200 RPM)
- Data-transfer rate: 8MB per second
- The time to (sequentially) read/write 4k data on this hard disk is:
 $4.8 + 4.2 + 4/8 = 9.5\text{ms}$
- The time to read/write 4MB data, which are separately stored on this hard disk as 1,000 4k blocks, is:
 $9.5 * 1000 = 9500\text{ms}$
- The time to (sequentially) read/write 4MB data on this hard disk is:
 $4.8 + 4.2 + 4000/8 = 509\text{ms}$
- Random accesses for small data on hard disks are expensive!

On-Disk Approach

- **Idea:** keep all the inverted lists (as a file) on disks instead of in main memory.
- **Step (b):** postings are appended to the end of a file, which stores all the postings.



On-Disk Approach (Cont'd)

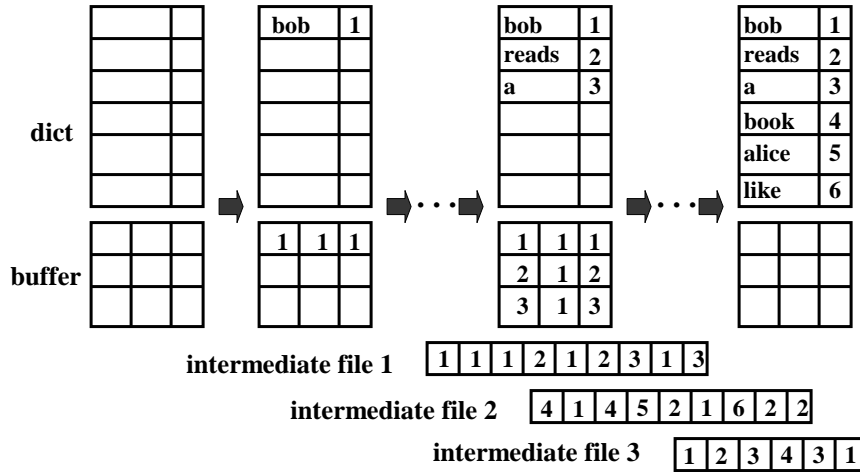
- **Step (c): since postings for different inverted lists are interleaved on disk, to traverse all the inverted lists, there is one random access to hard disk for every posting!**
- **Could take months!! ($30 \cdot 24 \cdot 60 \cdot 60 \cdot 1000 / 9 = 288$ million random accesses per month)**

Sort-Based Approach

- a) **create an empty dynamic dictionary data structure (e.g. hash table) in main memory;**
- b) **scan through all the documents, for every word encountered:**
 - i. **look up the dictionary for word id; create a new one, if it doesn't exist;**
 - ii. **keep (word id, doc id, pos) in a buffer in main memory;**
 - iii. **If buffer is full, sort postings in the buffer by (word id, doc id, pos) and write all the postings in the buffer onto disk;**
- c) **merge all the intermediate files into a final index file.**

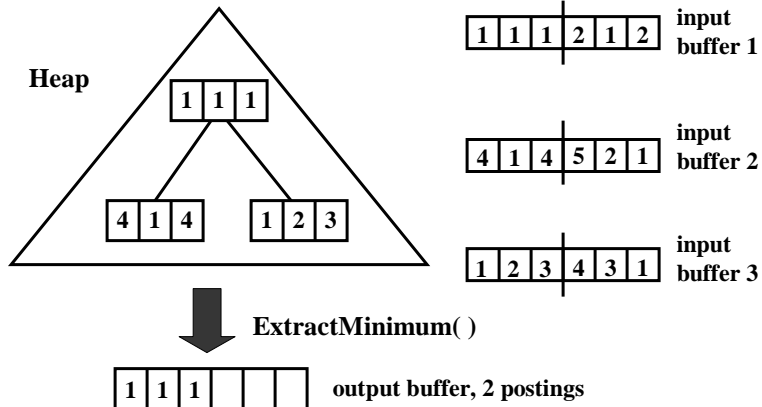
Sort-Based Approach (Cont'd)

doc1: "Bob reads a book", doc2: "Alice likes Bob", doc3: "book"
 buffer size: 3 postings

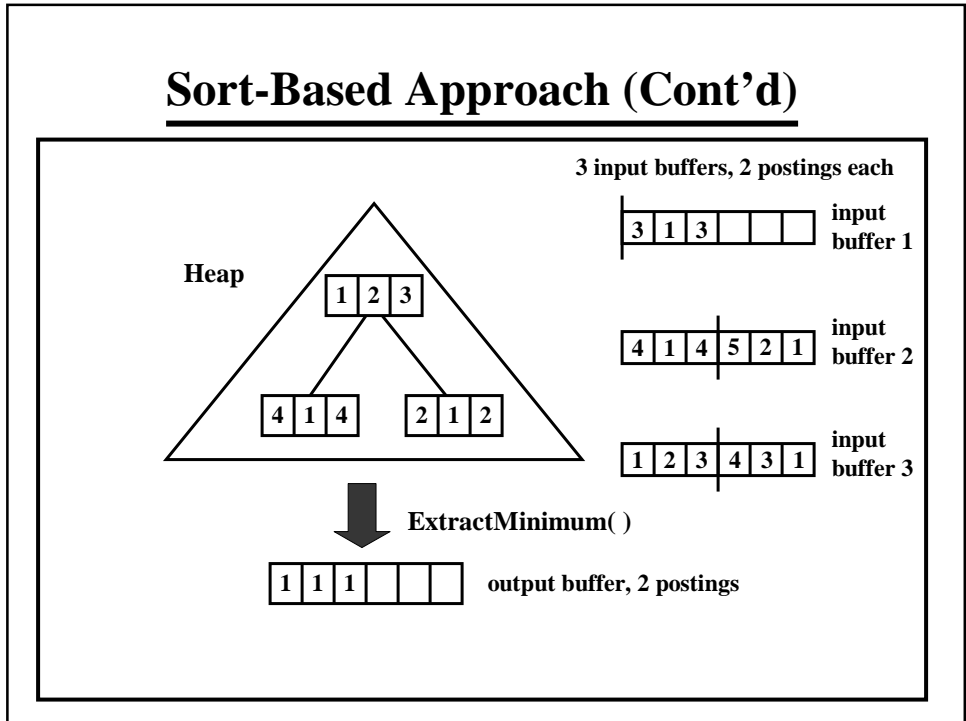


Sort-Based Approach (Cont'd)

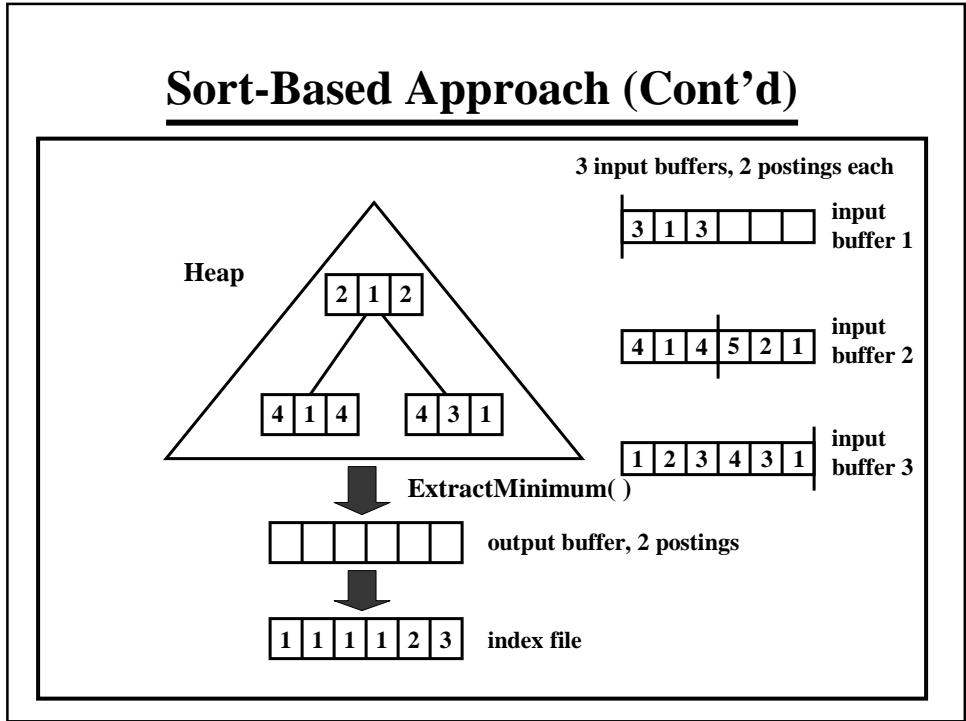
3 input buffers, 2 postings each



Sort-Based Approach (Cont'd)



Sort-Based Approach (Cont'd)



Performance

- 500 million postings
- 10 bytes per posting: 4 bytes each for word id and doc id, 2 bytes for pos
- average hard disk access time 9ms
- data transfer rate 8MB per second
- 10 million postings per intermediate file
- 50-way merge
- 50 input buffers (100,000 postings each)
- 1 output buffer (1 million postings)

Performance (Cont'd)

- In step (b), sort-based approach will take more time than on-disk approach, because of internal sorting. But not huge!
- In step (c), sorted-based approach will need read 1MB (100,000 postings) data 100*50 times and write 10MB data (1 million postings) 500 times:
$$(0.009+1/8)*5000 + (0.009+10/8)*500 = 1300s = 22mins$$
- However, on-disk approach will need read 10 bytes (1 postings) data randomly 500 million times:

$$0.009*5*10^8 = 4.5*10^6s = 52 \text{ days}$$

Optimizations

- **Compression**
 - Compressed index improves query performance.
 - Performance gain vs. decompression overhead.
- **In-Place**
 - The size of intermediate files could be large.
 - In step (c), part of the intermediate files will be read into input buffers first. And those space can be reused to store the index file.
 - No extra space.
 - More details in [Moffat&Bell].
- **Pipeline in step (b) [S. Melnik]**
 - Overlap CPU computations and I/O operations.

Some Indexing Numbers

- **120 million pages, about 1.8TB**
- **16 nodes: 1.6GHz P4 CPU with 1GB and 80GB*2**
- **7.5 million pages, 120GB uncompressed, 35GB compressed**
- **Only use one disk per node for index**
- **Indexing performance: 3MB/s per node**
- **11 hours per node**
- **Final index: 10GB (compressed) per node**

Some Additional Issues

- **Partitioning inverted index**
- **Updating inverted index**
- **Alternative index structures**

Partitioning Inverted Index

- **More than 3 billions pages indexed by Google**
- **Index could be more than 4TB**
- **Must be partitioned onto many nodes**
- **Horizontal vs. Vertical**

doc1: "Bob reads a book"
doc2: "Alice likes Bob"
doc3: "book"

horizontal partitioning:

index 1:
a: {(1, 3)}
bob: {(1, 1)}
book: {(1, 4)}
reads: {(1, 2)}

index2:

alice: {(2, 1)}
bob: {(2, 3)}
book: {(3, 1)}
likes: {(2, 2)}

vertical partitioning:

index 1:
a: {(1, 3)}
alice: {(2, 1)}
bob: {(1, 1), (2,3)}

index2:

book: {(1,4), (3, 1)}
likes: {(2, 2)}
reads: {(1, 2)}

Updating Inverted Index

- **Web pages are changing rapidly**
- **Updating can be challenging**
 - Assume you are adding one new page, which consists 500 words
 - 500 insertions into index on disk!!!
- **Many indexers don't support update (efficiently)**
- **Solutions:**
 - Semi-dynamic: build a separate index and merge
 - Buffer insertions in memory
 - ...
- **Need to decide if update performance is important**

Alternative Index Structures

- **Signature file**
 - Each word is hashed to a n-bit string
 - Signature of a page is computed by OR of n-bit strings of all the words in that page
 - False positive
 - “hello”: 0001, “world”: 0100, “web”: 0101
 - Signature of pages contain both “hello” and “world” will be “x1x1”, which will be considered to contain “web”
- **Bitmap**
 - One-to-one hash
 - Millions of words

Conclusion

- **Inverted index is the most used structure for full-text index**
- **Using I/O efficient (sorting) algorithm is important to build an inverted index**
- **Partitioning, updating, ...**