11-741:
Information Retrieval

Data Structures, Algorithms and Implementation Issues

Jamie Callan
Carnegie Mellon University
callan@cs.cmu.edu

Inverted List Indexes: Access Methods

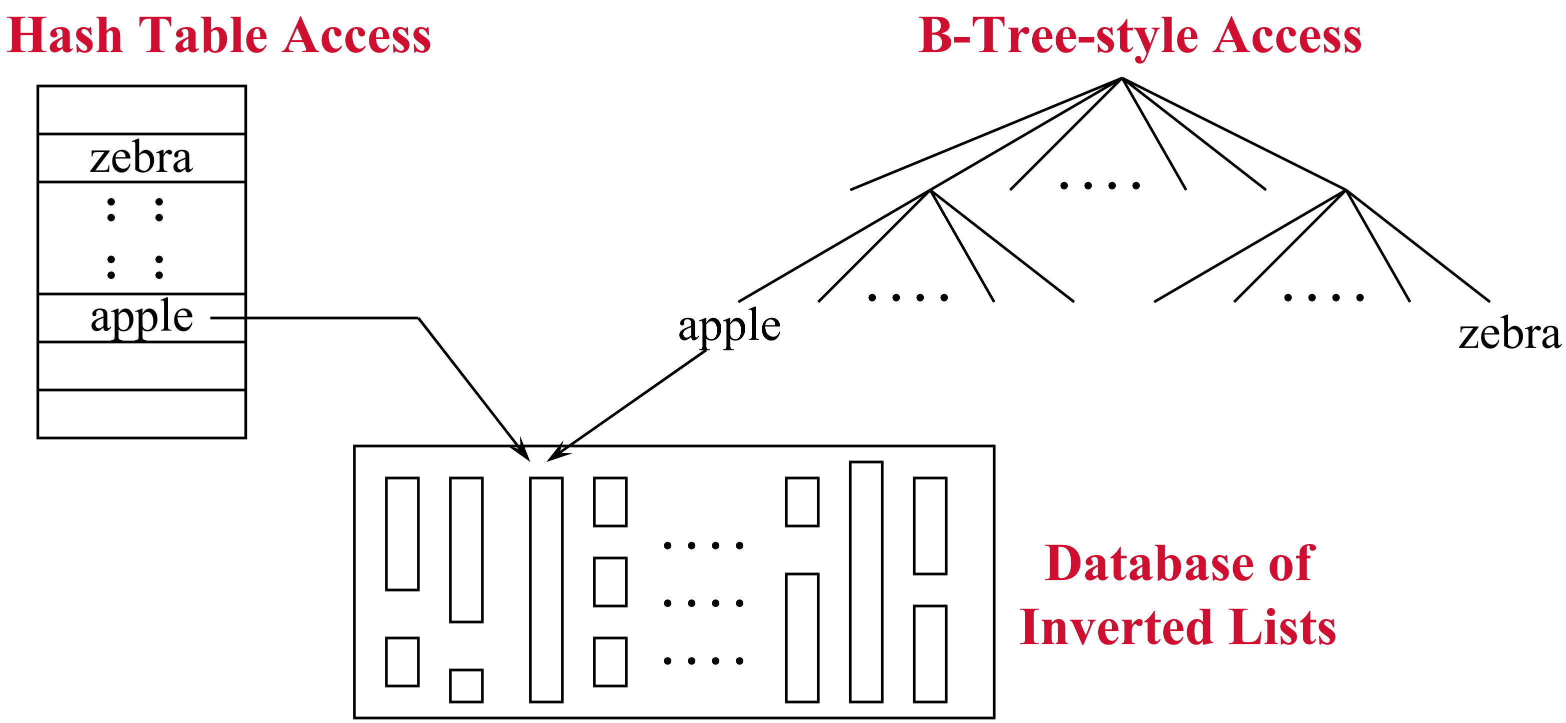
How is a file of inverted lists accessed?

- **B-Tree (B+ Tree, B* Tree, etc)**
  - Supports exact-match and range-based lookup
    - » "apple", "apple – apples", "appl*"
  - *O*(log n) lookups to find a list
  - Usually easy to expand
- **Hash table**
  - Supports exact-match lookup
    - » "apple"
  - O(1) lookups to find a list
  - May be complex to expand

Inverted List Indexes: Access Methods

Task: Enable accurate and *efficient* document retrieval

Solution: Database of inverted lists, different access methods

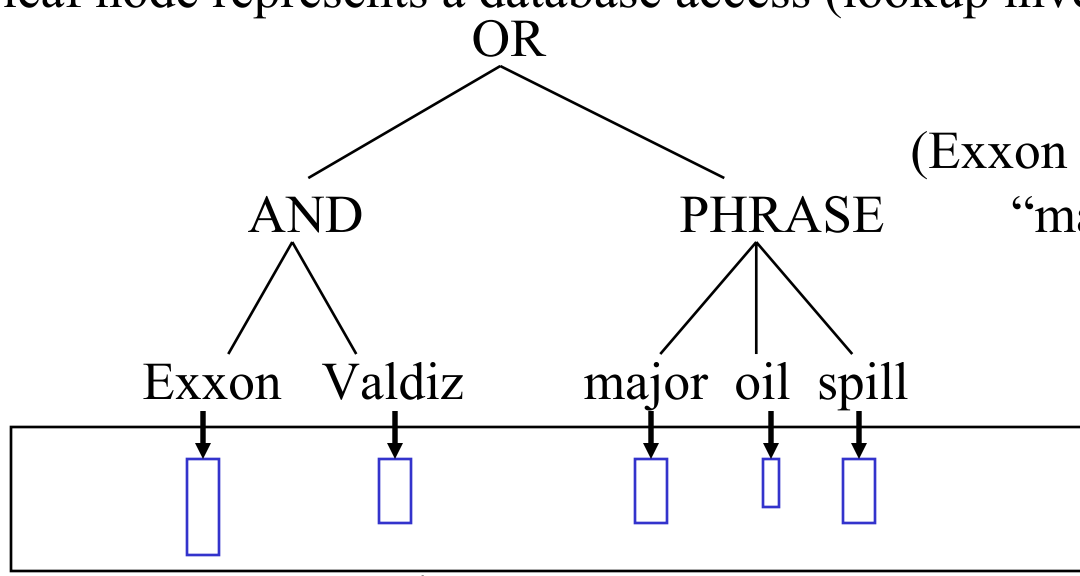


Document Retrieval Revisited

Task: Determine a score for each document (*quickly*)

Solution:

- Depth-first evaluation of query tree
- Each internal node represents query operators
- Each leaf node represents a database access (lookup inverted list)



Query

(Exxon AND Valdiz) OR
"major oil spill"

Database of Inverted Lists

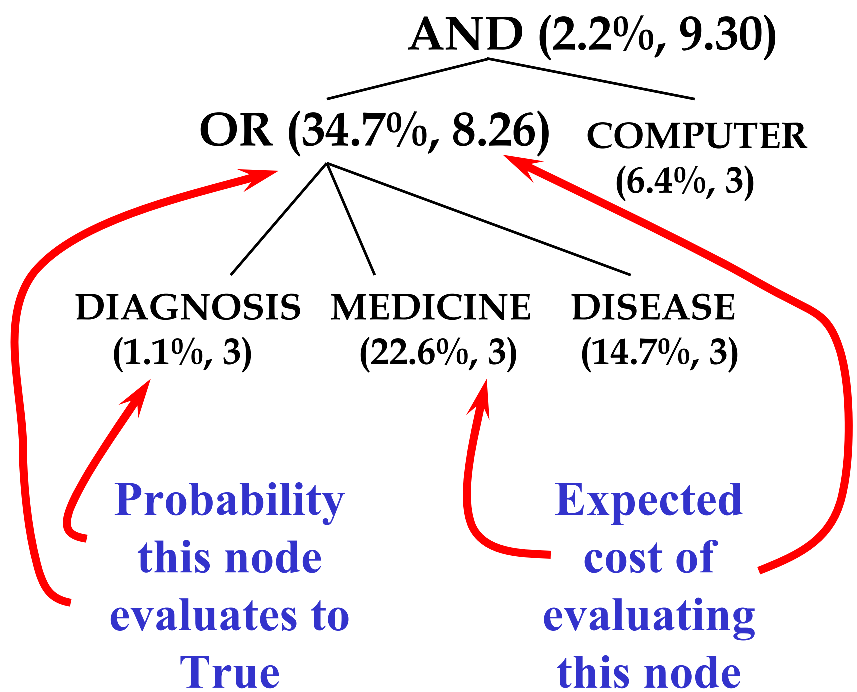
Optimizations

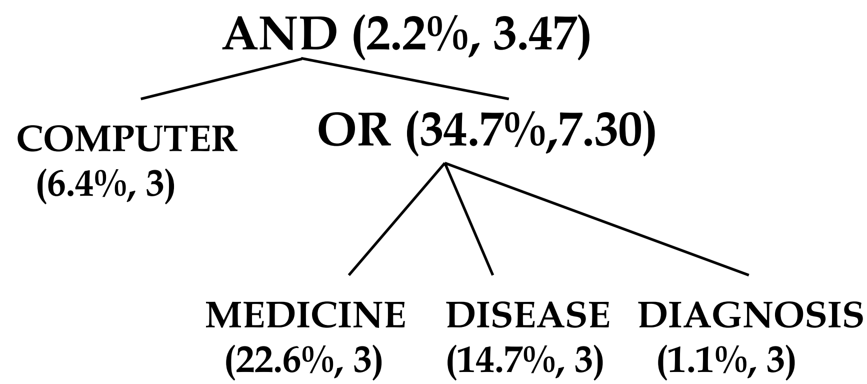
What happens when databases get large?

- **Boolean query processing**
- **Term-at-a-time vs. Document-at-a-time**
- **Skip lists**
- **Top-docs lists**
- **Separating data**

Implementation Details: Boolean Query Optimization

Goal: Lower average cost of evaluating query





- **For “intersection” query operators such as AND nodes the optimal strategy is “fail early”.**
- **For “union” query operators such as OR nodes the optimal strategy is “succeed early”.**

Implementation Details: Boolean Query Optimization

- **AND probability:** $p_1 p_2 p_3 \cdots p_n$

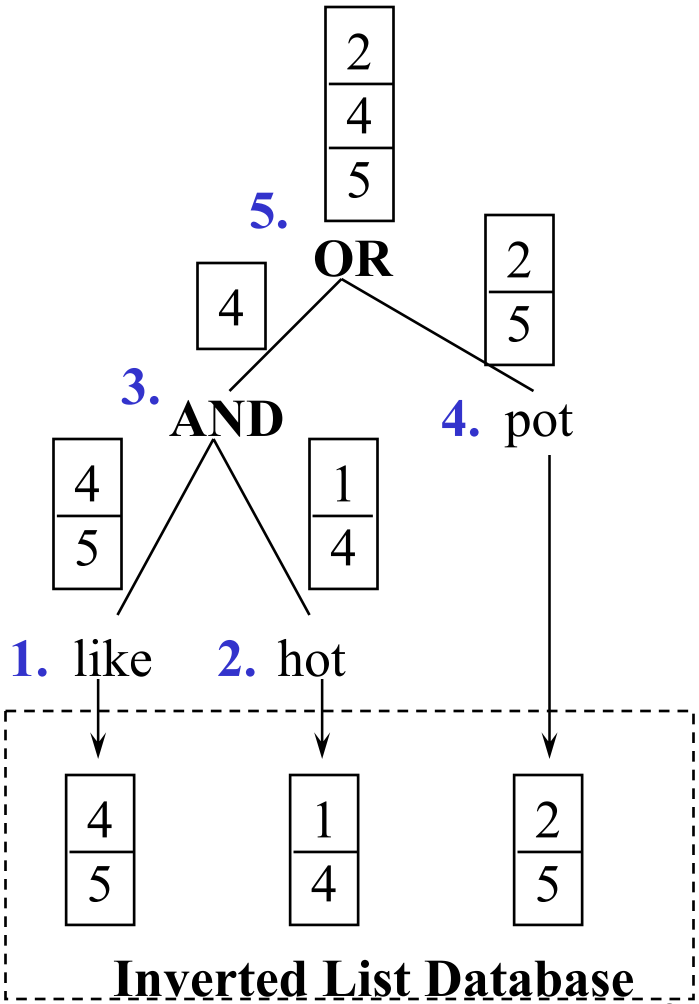
  AND cost: $c_1 + p_1(c_2 + p_2(c_3 + p_3(\ldots(c_n + p_n)\ldots)))$
- **OR probability:** $1-(1-p_1)(1-p_2)(1-p_3)\ldots(1-p_n)$

  OR cost: $c_1 + (1-p_1)(c_2 + (1-p_2)(c_3 + (1-p_3)(\ldots(c_n + (1-p_n))\ldots)))$

Minimize cost by reordering nodes, such that:

- **AND satisfies:** $\frac{c_1}{1-p_1} < \frac{c_2}{1-p_2} < \ldots < \frac{c_n}{1-p_n}$
- **OR satisfies:** $\frac{c_1}{p_1} < \frac{c_2}{p_2} < \ldots < \frac{c_n}{p_n}$

Ranking Documents: Term-at-a-Time Evaluation



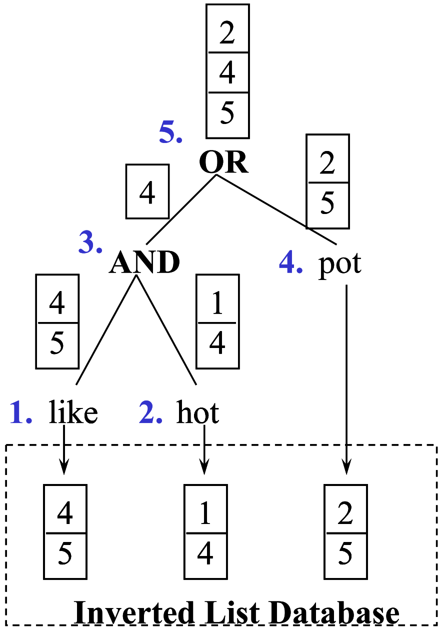
How should this query be evaluated?

- **Term-at-a-time**
  - Read inverted list for “like”
  - Read inverted list for “hot”
  - Merge them
  - Read the inverted list for “pot”
  - Merge it
- **This strategy is simple & popular for small systems**
  - It is not practical in big systems

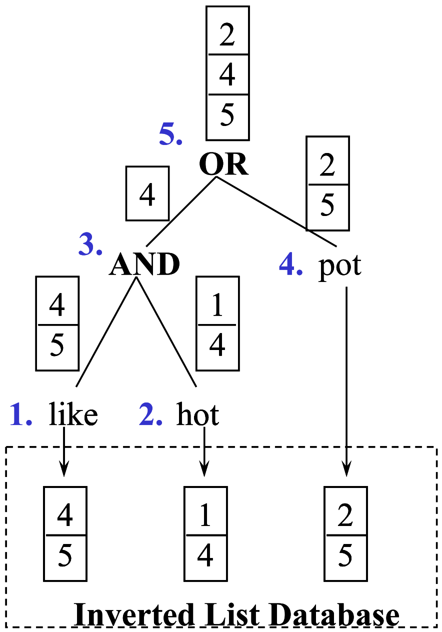
Ranking Documents: Term-at-a-Time Evaluation

Problems for the term-at-a-time approach

- **Inverted lists too big to fit in memory**
- **Deeply nested queries**
  - – Worst case is D+1 inverted lists kept in memory simultaneously
- **Systems that process queries in parallel**
  - – In some studies 3-4 query parallelism is optimal
    - » Some processes wait for I/O
    - » Some processes use the CPU
  - – But…the complexity of one query affects the amount of memory available to other queries



Ranking Documents: Document-at-a-Time Evaluation



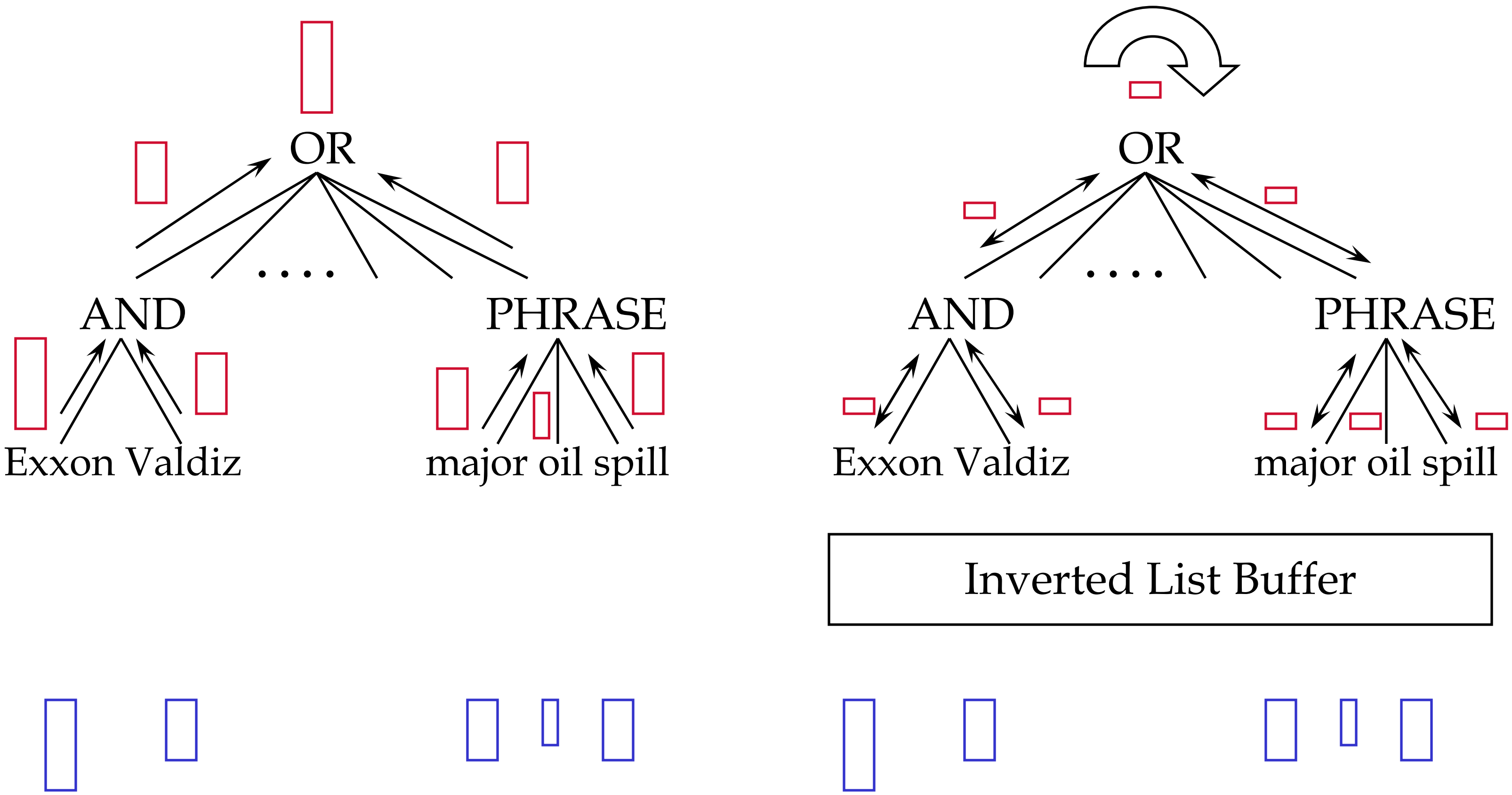
- **An alternative is to evaluate the query repeatedly**
  - Ask the tree what the next document is
    - » "like" returns 4, "hot" returns 1
    - » "AND" returns 4
    - » "pot" returns 2
    - » "OR" returns 2
  - Now ask the tree to:
    - » Evaluate the query only for document 2, and
    - » Return the next document id
- **This probably looks foolish, but**
  - It's faster than you think (because it allows some optimizations)
  - Now we don't need to keep entire inverted lists in memory

Ranking Documents: Document-at-a-Time Evaluation

Document-at-a-time optimizations

- **Next-doc processing allows large parts of inverted lists to be skipped**
  - E.g., AND operators apply a MAX function to the next-doc ids of their children
- **Typically processing isn't actually one document at a time**
  - More likely several thousand documents at a time
  - The chunk size depends on the memory available <u>right now</u>
    - » If the system is busy, process smaller chunks of documents
    - » If the system is idle and the query small, may default to term-at-a-time

Ranking Documents: Term-at-a-Time vs Doc-at-a-Time



Ranking Documents: Term at a Time vs. Document at a Time

Term at a Time:

- **Minimizes I/O**
  - 1 lookup per query term
- **Bookkeeping:**
  - List of candidate documents
  - A partial score per candidate document
- **Unpredictable memory usage**
  - Grows with query size

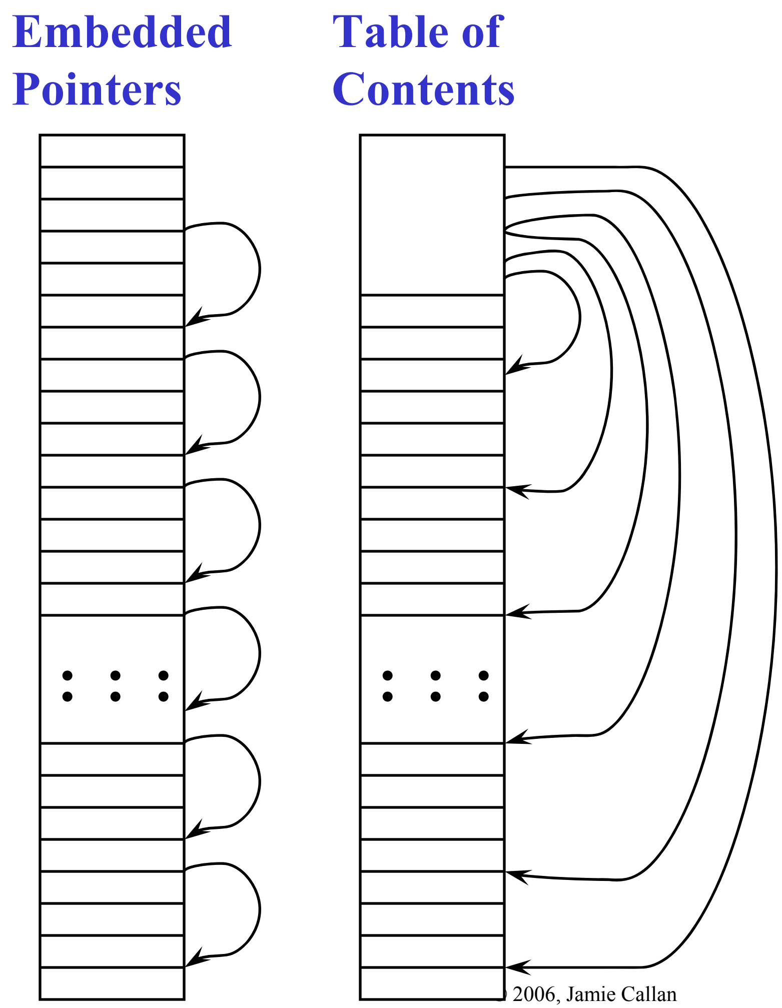
Both strategies are used
Hybrids are also used

Document at a Time:

- **Unpredictable I/O**
  - Grows with query length
  - Grows with query term *df*
- **Bookkeeping:**
  - Matched documents
  - A complete score matched document
  - Partial score for current document
  - Inverted list fragments
- **Constant memory usage**

Inverted List Optimizations: Skip Lists

- **Pointers that allow parts of the list to be skipped**
- **Gives no improvement unless tuned carefully**
- **Common problems**
  - No I/O improvement, because each block is examined anyway
  - Interferes with compression



Inverted List Optimizations: Top-Docs List

- **Most inverted lists point to many documents**
- **Most queries need to return 100 documents or less**
- **Why rank 500,000 documents if only 100 are needed?**
- **Top-docs lists:**
  - Points to only the best docs
  - Sorted by doc score
- **Much faster on long queries**
- **Lower recall**
  - Maybe lower precision
- **Can't use with some queries**

"apple" Inverted List

Doc 1 0.42
Doc 2 0.572
: : :
Doc 258392 0.44
Doc 258393 0.73
: : :
Doc 1025429 0.75
Doc 1025430 0.571

"apple" Top-Docs List

Doc 1025429 0.75
Doc 258393 0.73
: : :
Doc 2 0.572
Doc 1025430 0.571

1,000 documents

1,025,430 documents

Inverted List Optimizations: Separating Data

Separating presence information from location information

- **Many operators only need presence information**
- **Location information takes substantial space (I/O)**
- **If split,**
  – reduced I/O for presence operators
  – increased I/O for location operators (or larger index)
- **Common in CD-ROM implementations**

Inverted List Access: Strengths and Weaknesses

Strengths:

- **Simple to create**
- **Space efficient**
- **Given a term, very efficient access to postings**

Weaknesses:

- **It's not necessarily easy to update a document**
  – Many lists affected
  – Changes in the middle of lists means rewriting the entire list
- **Nearly impossible to find adjacent terms in a document**
  – "apple" is a location 12…what's at location 13?

Storing Structure

- **Many documents have internal structure**
  – Title, date, author, …
  – Chapter, section, subsection, paragraph, sentence, references, …
- **Research systems have tended to ignore structure**
  – Some studies suggest that people don't use it much
- **Commercial systems have tended to support some structure**
  – Especially title, date, author, sentence
- **Some current trends suggest the document structure will be increasingly important in the future**
  – XML
  – Use of IR systems as "back-ends" for other language processing tasks
  – Document markup such as POS, named-entity, syntax, …

Document Markup

- **XML markup**

  <newsitem itemid="100000" id="root" date="1996-10-07" xml:lang="en">

  <title>USA: NYCE cotton closes up on Tropical Storm Josephine.</title>

  <headline>NYCE cotton closes up on Tropical Storm Josephine.</headline>

  <dateline>NEW YORK 1996-10-07</dateline>

  <text>

  <p>Light speculative buying buoyed NYCE cotton futures to a higher close as Tropical Storm Josephine was poised to deluge an already soaked …

  <p>--Suzanne Rostler, New York Commodities 212-859-1646</p>

  </text>

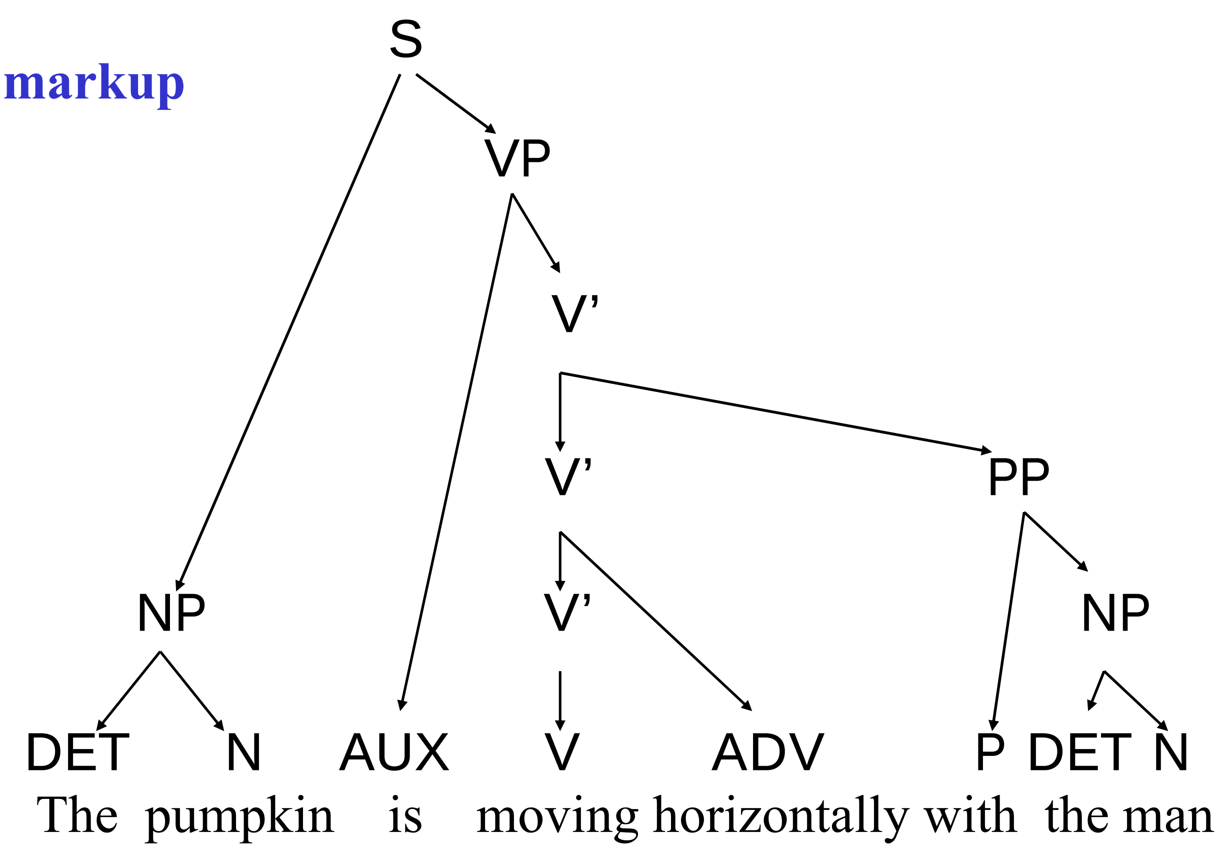
  <copyright>(c) Reuters Limited 1996</copyright>

Document Markup

- **Part of speech markup**
  - "The/AT chief/NN of/PRF these/DT spies/NN is/VBZ the/AT celebrated/AJ0 Belle/NP Boyd/NP ./."
- **Named-entity markup**
  - "The chief of these spies is the celebrated <ENAMEX TYPE=PERSON>Belle Boyd</ENAMEX>"

Document Markup

- **Syntactic markup**



(Carolyn Penstein Rose)

Storing Term Properties

- **One of the simplest approaches is to store meta-terms**
  - <Apple, Doc 23, Location 18>
  - <Property:Organization, Doc 23, Location 18>
- **The meta-term is stored just like any other indexing term**
  - Usual inverted list indexing
- **Query operators associate terms with properties**
  - The query operator "Property (Apple, Organization)" is implemented as "Apple Near/0 Property:Organization"
  - Seems crude, but very effective for single-term properties
- **For multi-term properties, use Begin/End tags and slightly more sophisticated query operator**
  - E.g., Property:Organization:Begin, Property:Organization:End

Storing Fields and Structure

- **Very easy to do for shallow, “hard-wired” fields**
  - E.g., treat Title:Apple and Text:Apple as different index terms
- **More “interesting” for deep, dynamic fields**
  - E.g., XML documents
  - Terms in “Subsection” should also appear in “Section”
    - » But, don’t want to store them twice
  - **Solution:** Include additional data structures to store document structure explicitly
    - » Essentially store the parse tree…

      …perhaps with additional information
      - E.g., the length of the field
      - E.g., pointers to its parent or children

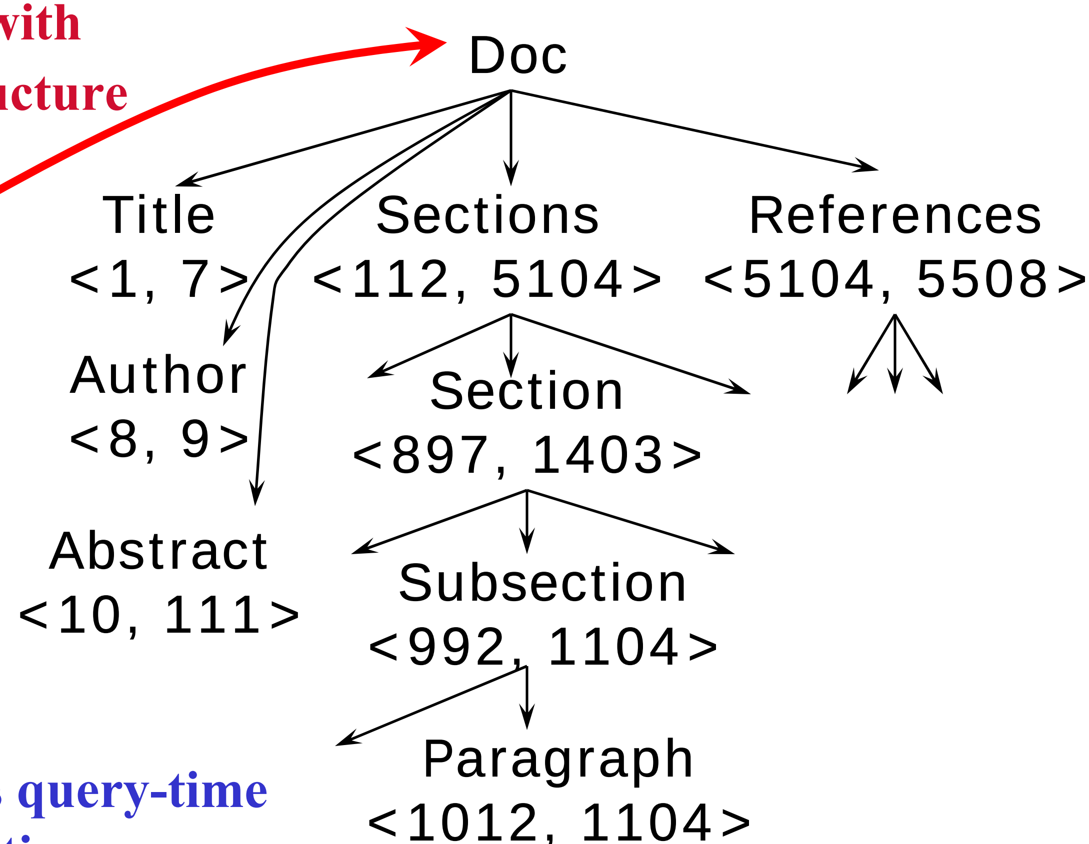
Storing Fields and Structure

Traditional inverted list

Doc: 28
TF: 3
Loc: 6
Loc: 27
Loc: 5442
Doc: 92
:
:

Inverted list with support for structure

Doc: 28
Struct:
TF: 3
Loc: 6
Loc: 27
Loc: 5442
Doc: 92
:
:



This looks messy, but it gives query-time control over which term locations are used for matching and scoring

Storing Fields and Structure: Issues

This is an active area of research

- **How to do it efficiently**
- **How to provide effective retrieval**
- **How to make it easy for people to use**
  - Or, maybe not primarily intended for use by people
- **What is necessary/useful to support other language tasks?**
- **How to support complex user models**

Document Term Lists

- **Sometimes the IR system needs to know what terms are in the document**
  - E.g., for query expansion, relevance feedback, …
- **How does it find out?**
  - Parse the document again?
    - » A little slow (although not as bad as you would think)
    - » Done when disk space is expensive
    - » But it's very hard to guarantee the exact same parse
  - Store the parsed document?
    - » Guaranteed to be accurate, but it takes a lot of space
    - » Done when disk space is cheap

Document Term Lists

Doc: 21
Length: 433
Postings: 492
<1, 41318>
<2, 39122>
<4, 55823>
<4, 28>
<5, 9975>
:   :

- **Length: Number of words in the surface text**
- **Postings: Number of elements in the document term list**
  - Includes annotations, e.g., named-entity, POS
- **Term 3 is missing…it was a stopword**
- **Term 4 was annotated**
  - Use small term ids for common terms
    - » Because they compress more effectively
- **A big data structure**
  - But locations compress very well

Other Indexes: Wildcard Matching

- **Historically wildcard matching was important**
  - Partly as a substitute for good morphological processing
    - » computer* matched “computer”, “computers”
  - As better morphological processing was introduced, wildcard matching mostly disappeared from text search systems
- **Wildcard matching remains important in some applications**
  - Names with inconsistent spellings
    - » Qaddafi, Kaddafi, …
  - Languages with very flexible morphology
    - » E.g., Arabic
  - Domains where spelling is still in flux
    - » E.g., bioinformatics

Other Indexes: Wildcard Matching

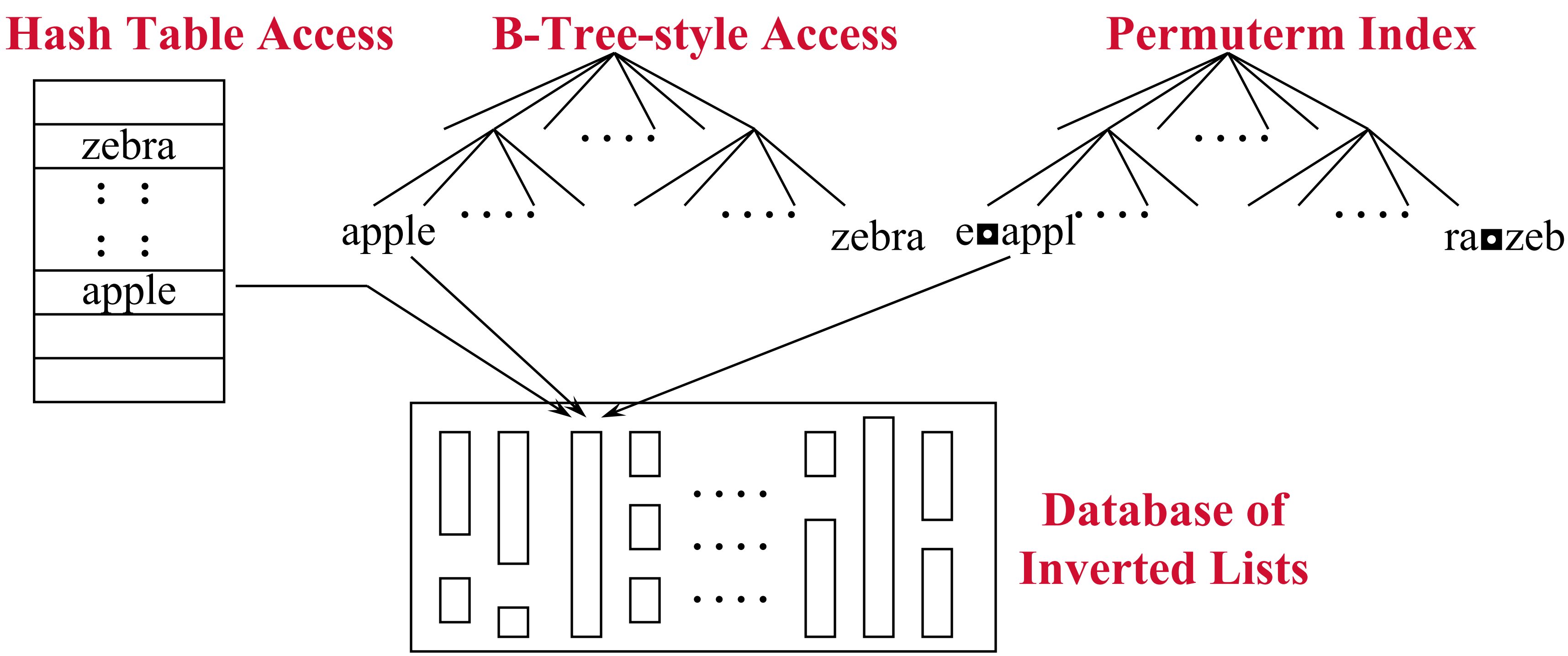
- **X* is easy, but *X, *X*, and X*Y are hard (why?)**
- **"Permuterm" Index on Inverted Lists:**
  - Prefix each term X with a "◘"
  - Rotate each augmented term cyclically (with wraparound) by one character, to produce n new terms
  - Insert all forms into the dictionary
    - » Any data structure supporting lexically-adjacent lookup, e.g., B+ Tree
- **Lookup:**

	Term	Key
– X: Match ◘X exactly	sled	◘sled
– X*: Match all terms beginning with ◘X	sled*	◘sled…
– *X: Match all terms beginning with X◘	*sled	sled◘ …
– *X*: Match all terms beginning with X	*sled*	sled…
– X*Y: Match all terms beginning with Y◘X	d*sled	sled◘d…

Inverted List Indexes: Access Methods Revisited

Task: Enable accurate and *efficient* document retrieval

Solution: Database of inverted lists, different access methods



Summary

- **Inverted lists for term-based access**
  - Exact match lookup via hash table
  - Range-based lookup via B-Tree
  - Wildcard lookup via Permuterm
- **Document term lists for access to parsed document**
  - Name varies…this is a relatively new feature in most systems

For More Information

- I.H. Witten, A. Moffat, and T.C. Bell. "Managing Gigabytes." Morgan Kaufmann. 1999.
- G. Salton. "Automatic Text Processing." Addison-Wesley. 1989.
- E. Brown. "Execution Performance Issues in Full-Text Information Retrieval." Ph.D. dissertation, University of Massachusetts. 1995.
  Available as technical report IR-73 at http://ciir.cs.umass.edu/.