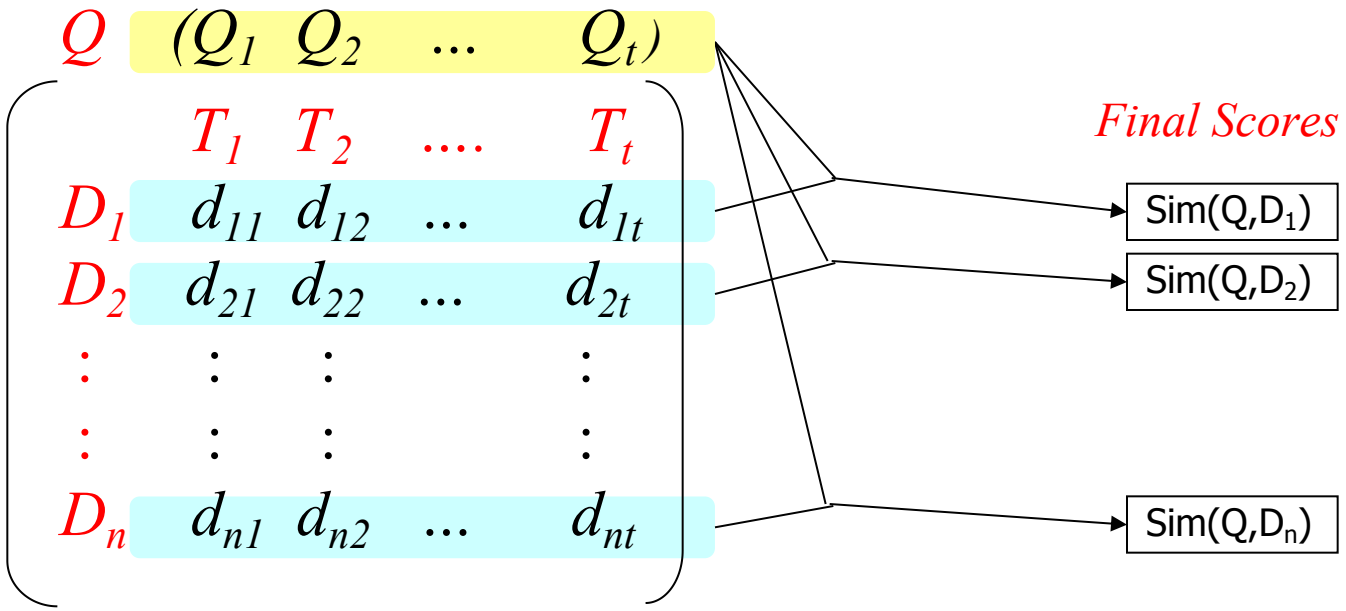
Implementation Issues

Straight Vector Multiplication (Theoretical)

- for all document vectors d
  do compute score(d) = Sim(d, q)
- end
- Sort documents according to scord(d)



Term (Column) Based Computation

- for i in Q_1, Q_2, ..., Q_m [totally m terms in the query]
- for j = 1 to n [totally n documents]

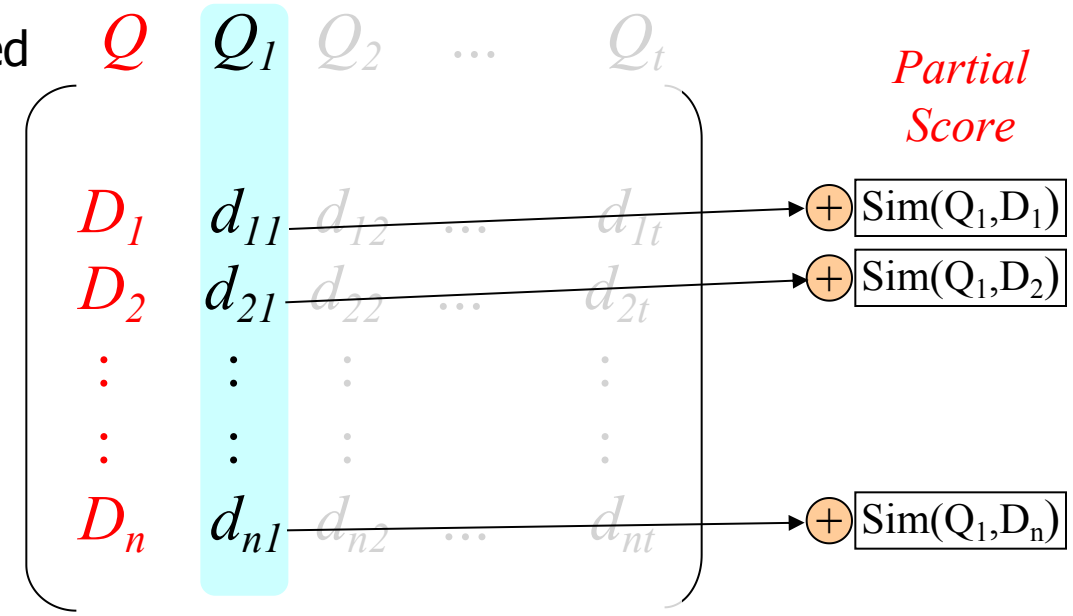
  Compute partial score between D_j and Q_i

  score (D_j, Q) += partial score (D_j , Q_i)

  end {for}

  end {for}
- Perform normalization if needed

Inner product = sum up score between each query term and each document



- Note: the partial score is the product of the term weight and query weight; query weights are assumed to be 1, so the partial score is an accumulation of all d_{ij} weights for all terms found in the query.

Term (Column) Based Computation

- for i = 1 to t [totally t terms in the vector space]
- for j = 1 to n [totally n documents]

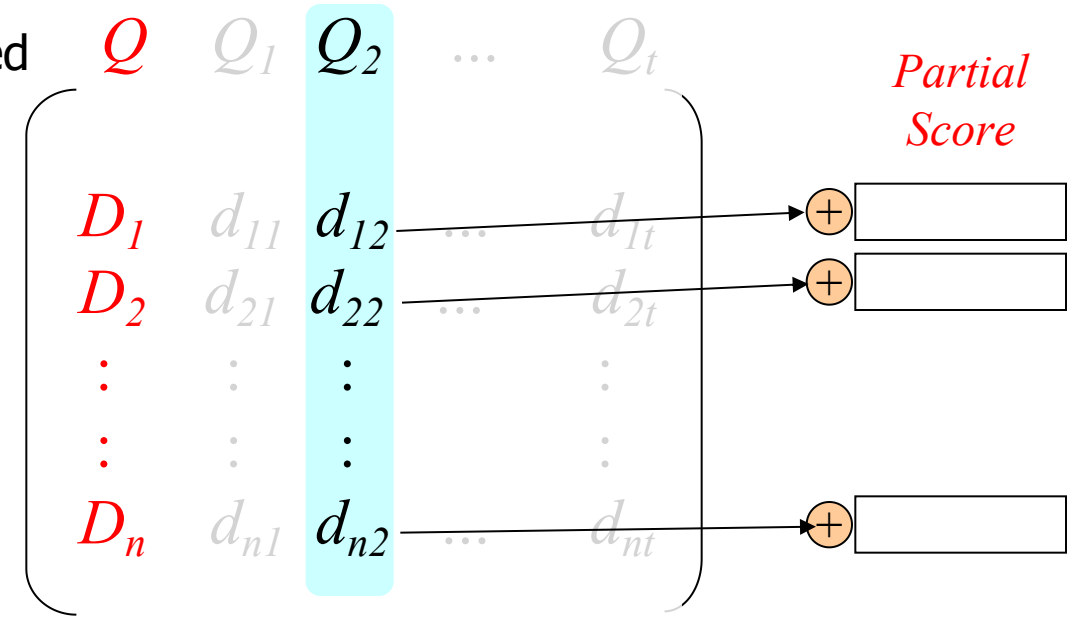
  Compute partial score between D_j and Q_i

  score (D_j, Q) += partial score (D_j , Q_i)

  end {for}

  end {for}
- Perform normalization if needed

Inner product = sum up score between each query term and each document



- Note: the partial score is the product of the term weight and query weight; query weights are assumed to be 1, so the partial score is an accumulation of all d_{ij} weights for all terms found in the query.

Term (Column) Based Computation (Theoretical)

- for i = 1 to t [totally t terms in the vector space]
- for j = 1 to n [totally n documents]

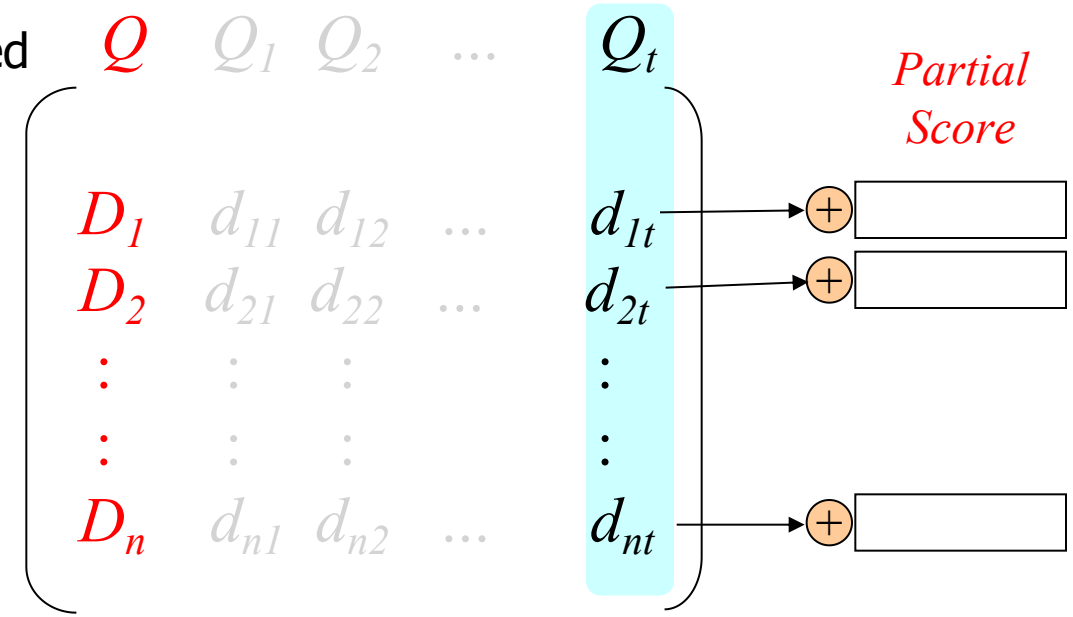
  Compute partial score between D_j and Q_i

  score (D_j, Q) += partial score (D_j , Q_i)

  end {for}

  end {for}
- Perform normalization if needed

Inner product = sum up score between each query term and each document



- The partial score is the product of the document weight and query weight; query weights are assumed to be 1, so the partial score is an accumulation of all d_{ij} weights for all terms in the query.

In reality, we do not store or process query and document vectors directly!

Algorithm based on Word List

- In practice, document vectors are not stored directly; more efficient to represent a document vector as a word list:
  D_j = <computer 5, database 6, …, science 3.5, system 1.2>
  Q = <database, text, information >

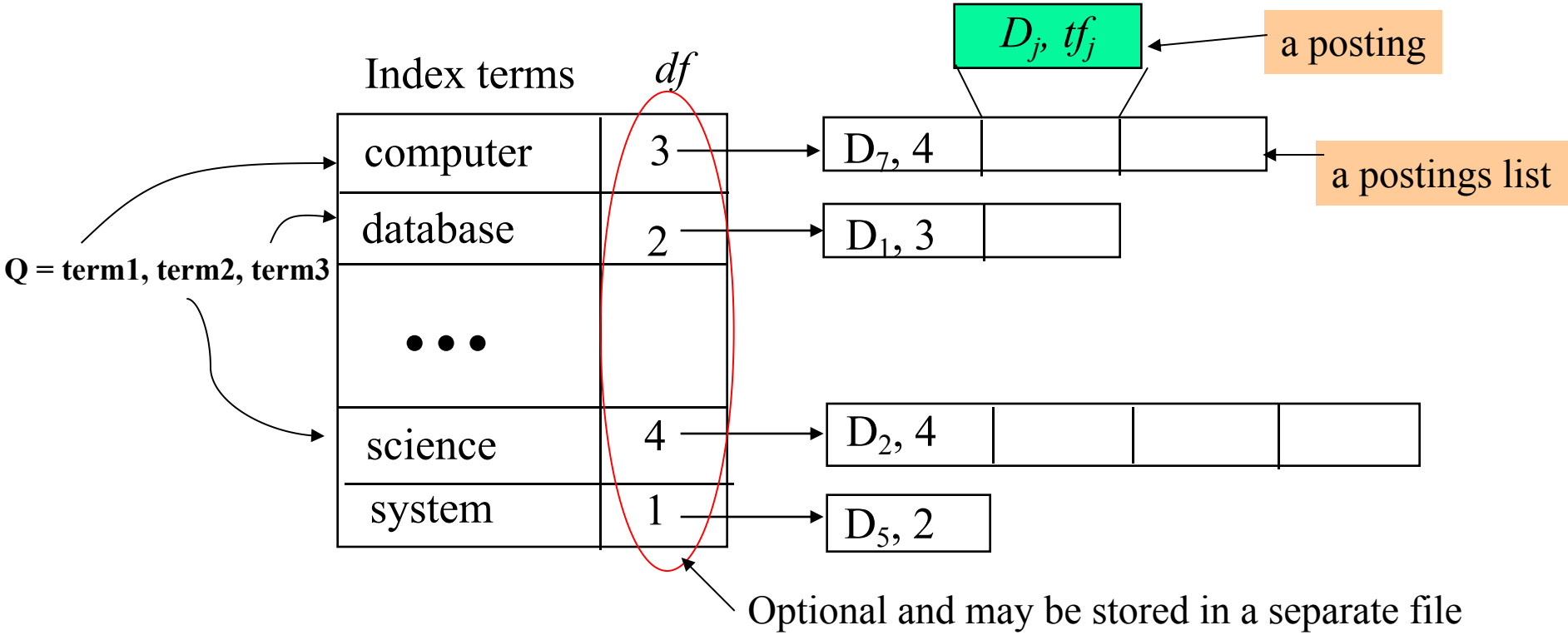
- for each term i in Q [Only considers terms specified in Q]
- for j = 1 to n [loop through all documents]
  Look up Q_i in D_j's word list [search half of the words on average]
  Compute partial score between D_j and Q_i
  score (D_j, Q) += partial score (D_j , Q_i)
  end {for}
  end {for}
- Perform normalization if needed

Searching through all documents one by one for each query term is still too slow!

Indexing (building an inverted file) comes to help!

Implementation based on Inverted Files

- Use an inverted file to speed up the lookup process (1st step in the inner loop)
- An inverted file can be implemented as a hash file, a sorted list, or a B-tree, etc.

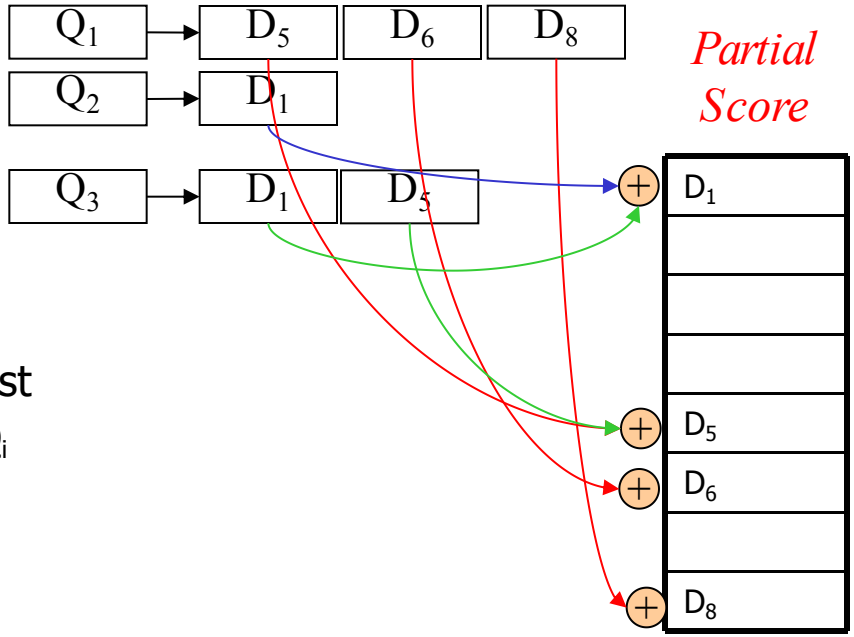


Algorithm based on Inverted Index

- For each term Q_i in Q
- Look up Q_i from inverted index
- If not found: continue
- If found: retrieve postings list for Q_i

  For each document D_j on the postings list
  - Compute partial score between D_j and Q_i
  - score (D_j, Q) += partial score (D_j , Q_i)

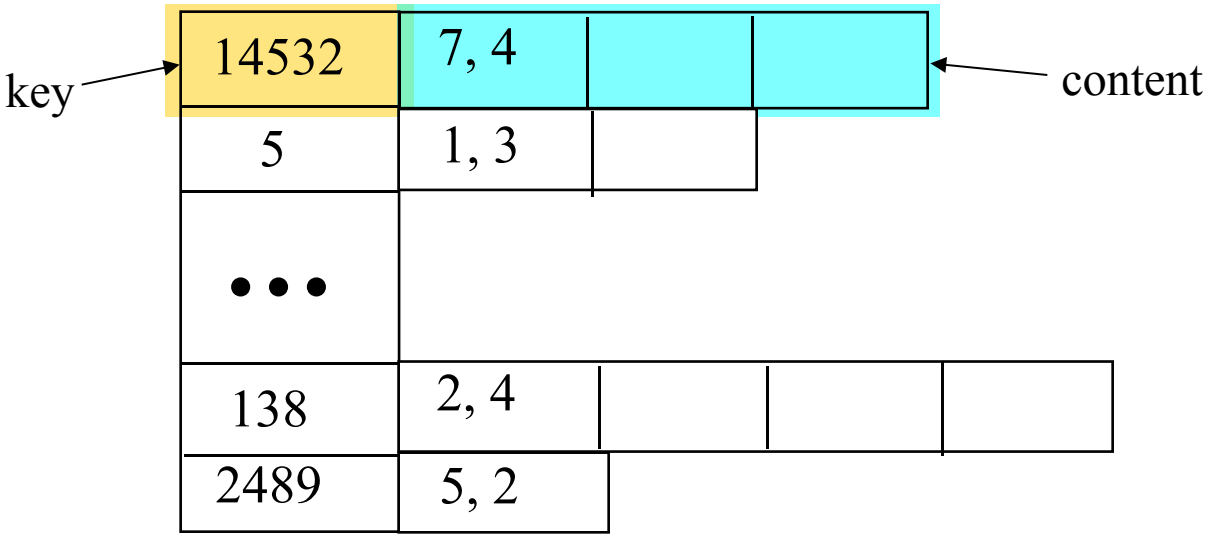
  end {for}
- end {for}
- Perform normalization if needed

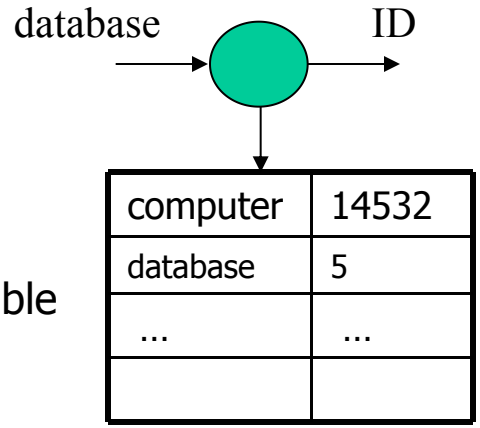


- Note: partial score depends on similarity formula used; for tf/tfmax * idf, tf is stored in the postings entry, tfmax must be stored somewhere, and idf is computed based on N and DF, the latter is obtained from the inverted file

Internal Representation

- Words and documents are mapped into 16-bit or 32-bit integer IDs
- Whole index implemented as one file:





if word is in table, return ID
if not, insert <word, count> into table
count++

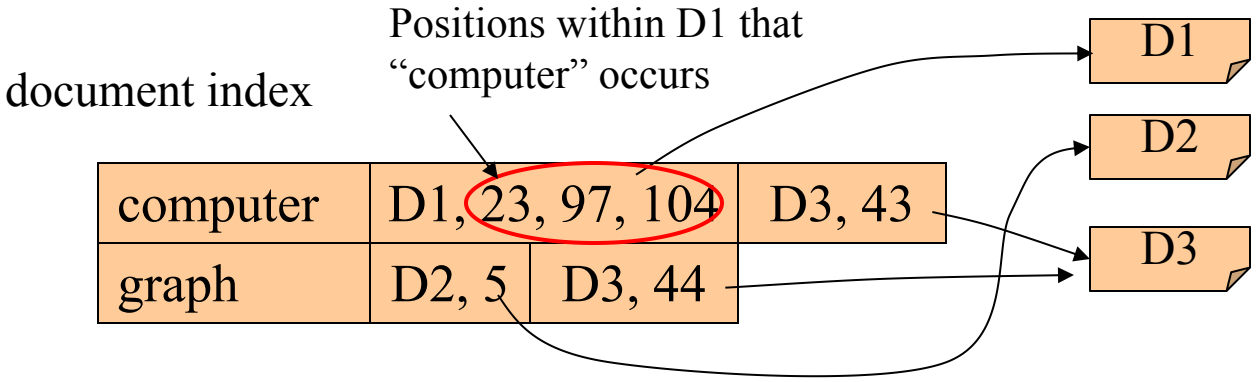
Representation of Global Statistics

- How to represent tf_{max}?
- How to handle deletion of a document?
- Why not store *idf* directly?

Need a "forward index": Given a document, return the terms it contains and perhaps it's highest tf value.

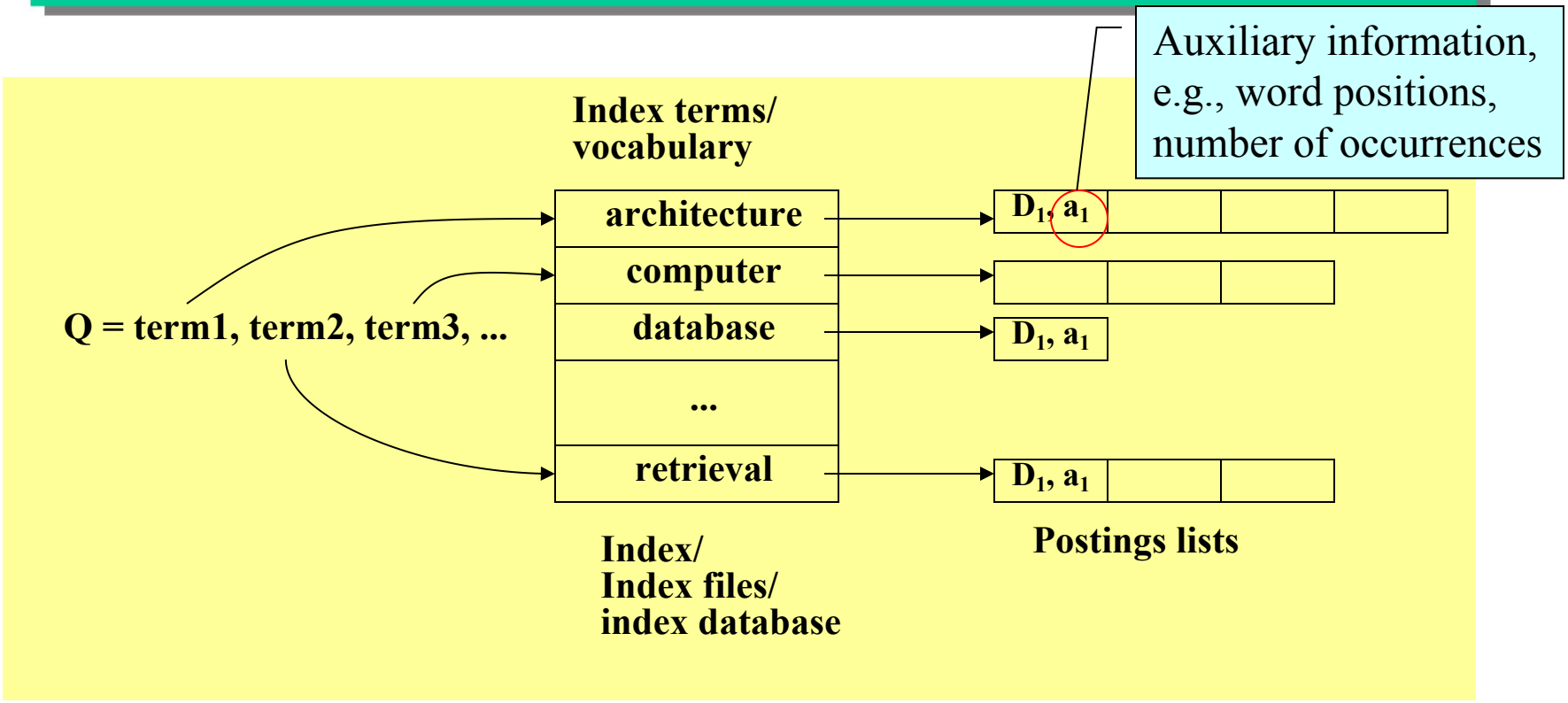
Indexing and Inverted Files

Indexing on Documents



- Index structures: *hashing, B+-trees, tries*.
  (Tries are good for dictionary-based searching.)
- Each word in a document is indexed
- If a document is considered as one single attribute, then indexing is done on *part* of an attribute value.
- Partial match: '%database% ', wildcards
- Phrase search: which document contains "computer graph"

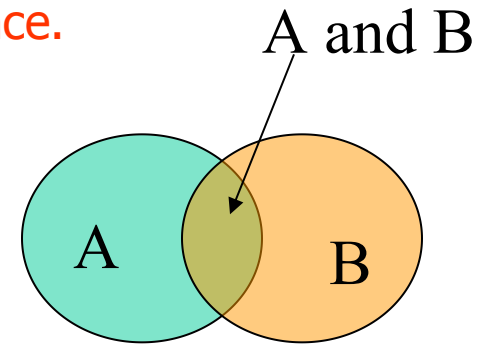
General Inverted Index



- Index file can be implemented with any file structure
- Terms in the index file is the *vocabulary* of the document collection

Boolean Retrieval on Inverted Indexes

- A Boolean query consists of ***n*** terms connected with Boolean operators, e.g., "computer AND news AND NOT newsgroup"
  - Parenthesis can be used to denote precedence.



- Each term returns a postings list from the inverted index
  - The postings list is empty if the term doesn't exist in any document
- Results are combined based on:
  - AND: set intersection
  - OR: set merge
  - NOT: set subtraction (NOT x is difficult to evaluate; x AND NOT y is OK)

Query Optimization on Inverted Indexes

- Standard optimization techniques apply: start with AND on the shortest lists; keep intermediate results as small as possible.
  - "Hong Kong" AND "Dik Lee" AND "HKUST"
  - which terms should be evaluated first?
  - Impact of optimization on queries that return no result
  - ("Hong Kong" or HKUST) AND "Dik Lee"

Keyword	Freq
Dik Lee	20
HKUST	1000
Hong Kong	500000

Advantages of Inverted Indexes

- Fast retrieval (long queries take more time)
- Flexible structure: different kinds of information can be stored in the postings list
- Support complex retrieval operations if enough information is stored
  - e.g., phrase and proximity queries can be supported if word positions are recorded

Disadvantages of Inverted Indexes

- Large *storage overhead* (50% - 150% - 300%)
- High maintenance costs on updates, insertions and deletions
- Processing cost increases with the number of Boolean operators
- Questions:
  - Why storage overhead could exceed 100%?
  - How could some system claim to have 2-3% storage overhead?
  - Why is insertion cost important (more and more so)?

*storage overhead = size of index / size of text * 100%*

Extensions on Inverted Indexes

- The inverted index is flexible enough to store more and more information about keywords to support more sophisticated retrieval

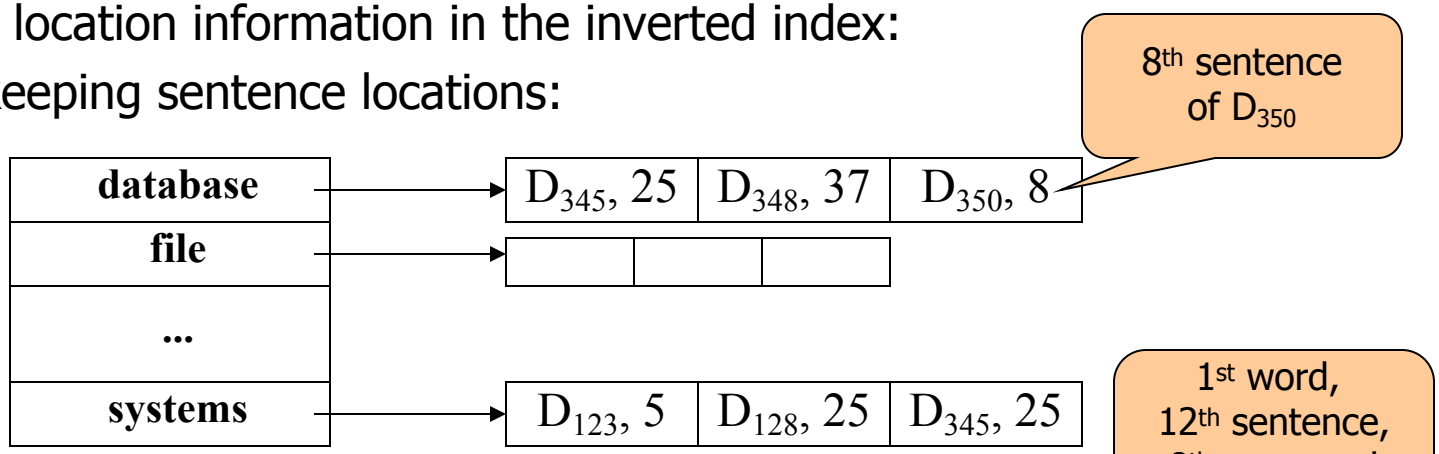
Extension – Distance Constraints

- Adjacency conditions are often needed. E.g.,
  - "database" immediately followed by "systems"
    - I.e., phrase search "database systems"
  - "database" and "systems" no more than 3 words apart
  - "database" and "architecture" in the same sentence.

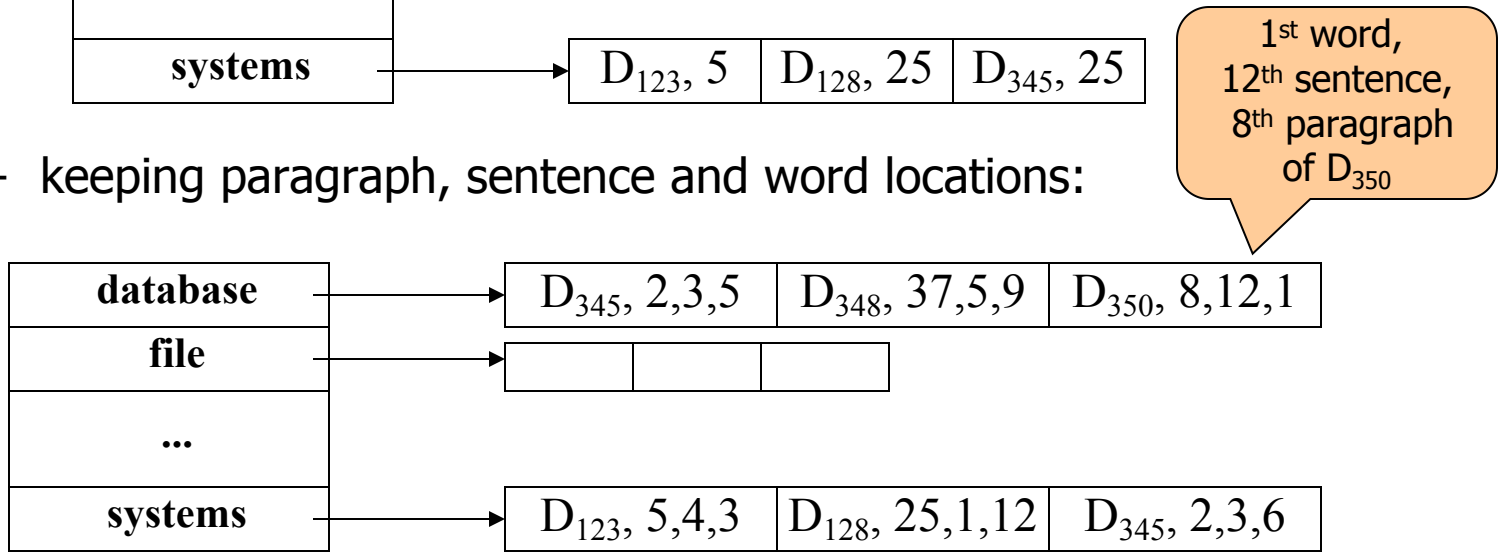
- Requires extension:
  - Inverted index keeps locations of keywords within documents, and document components (title, section heading, sentence breaks, etc.)
  - Retrieval algorithm has to correlate location information and check document components

Extension – Distance Constraints

- Keep location information in the inverted index:
  - keeping sentence locations:

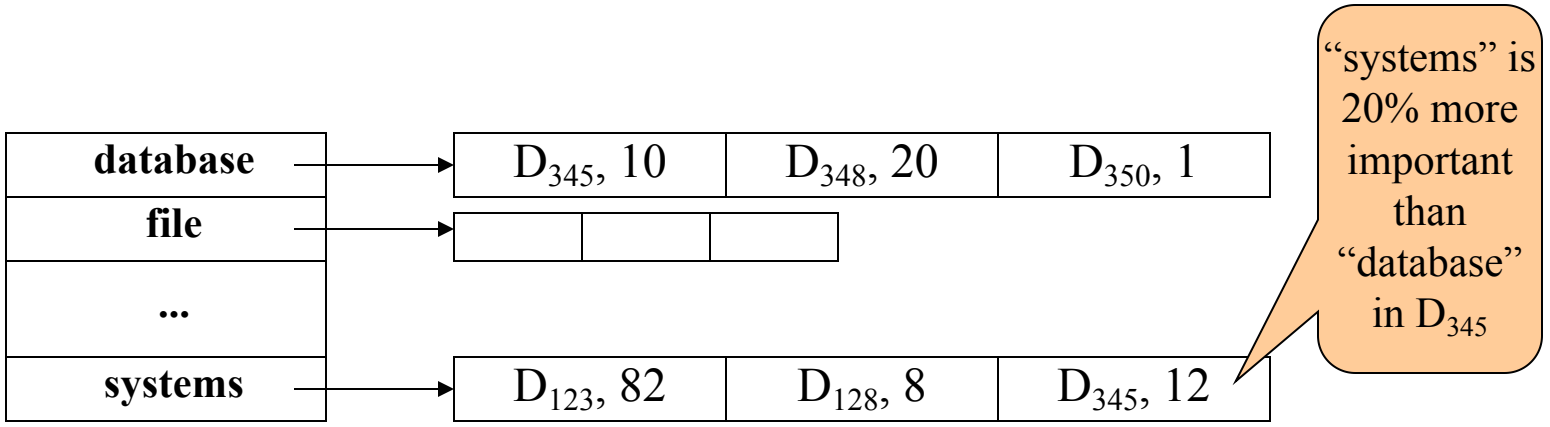


  - keeping paragraph, sentence and word locations:



- Note the increase in storage overhead
- NEAR operator supported in Altavista

Extension – Term Weights



- The second component of a postings could be a weight by itself (e.g., could be normalized to a number between 0 and 1) or occurrence frequency information based on which a weight can be computed
- If you record the positions where a word occurs in a document, you can obtain the *term frequency* by counting the positions
- The relative importance of a word/term in a document can be estimated from:
  - *Term frequency*: Number of times the term appears in a document
  - *Document frequency*: Number of documents containing the term

Extension – Synonyms

- Synonyms are important for increasing the coverage of a query.
- Synonyms can be added to the index with pointers pointing to the same postings list.

Extension – Term Truncation

- Suffix truncation is a simple form of stemming:
  - comput* : computer, computing, computation, etc.
  - Can be handled easily if the inverted index is implemented as a trie
  - Somewhat difficult to handle on a b+-tree (e.g, use a mapping table to map comput* to compute, computer, computing, etc)
  - Not feasible if a hash file is used
- Some system enforces a minimum length of the known prefix to limit the size of the result. On altavista:
  - compute* returns 107 million hits
  - comput* returns 114 million hits
  - compu* returns 117 million hits
  - comp* returns no results

33 million hits
28 million hits
39 million hits

2000 Google

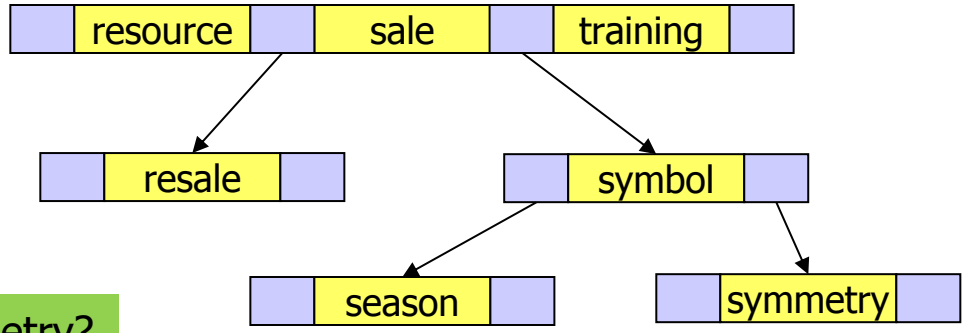
13 million hits
3.9 million hits
10.9 million hits
24 million hits

2016 Bing

Google (as of 2007) does not support wild cards; Bing still does (2016)

Extension – Prefix Truncation

- *symmetry matches symmetry, asymmetry, etc.
- Hash file requires knowing the exact key
  - e.g., symmetry -> hash(symmetry) -> location of symmetry in hash file
- Tree requires search key to "guide" the search down the tree
  - e.g. locate the subtree for all strings beginning with "s", then go to the child node for strings beginning with "sy", and so on
- It is difficult to match a word without knowing the prefix, since there is no "starting point" to guide a search.



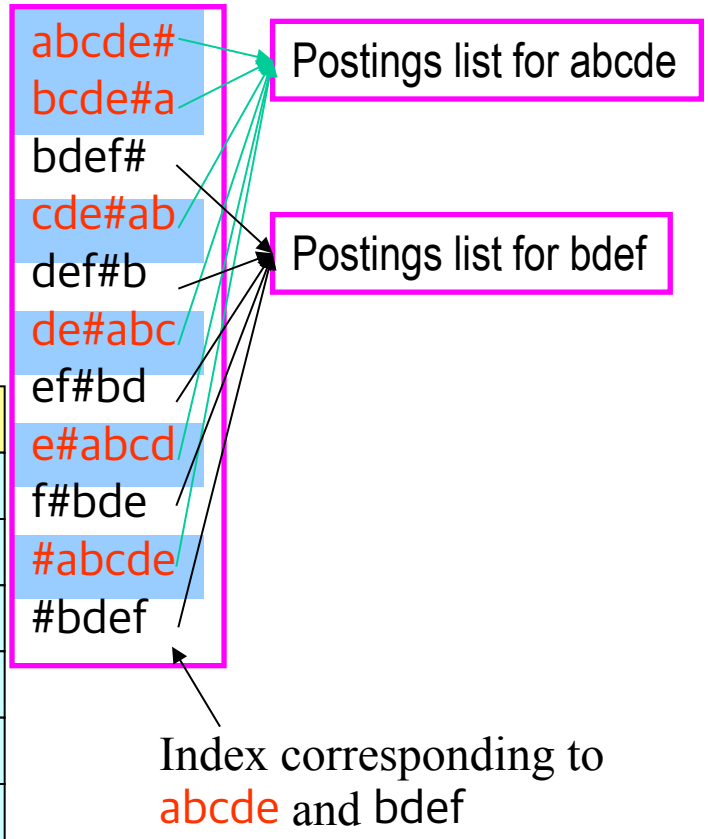
How to search *symmetry?

Extension – Prefix Truncation

- *Rotated index*: Given a term abcde to be indexed
  - append word terminator: abcde#
  - generate all possible rotations and insert them into the index

abcde#, #abcde, e#abcd, de#abc, cde#ab, bcde#a

	pattern	⇒	search	returns
prefix truncation	*abcde	Remove * Append #	abcde#	abcde
	*de		de#	abcde (not bdef)
suffix truncation	abcde*	Remove * Prepend #	#abcde	abcde
	ab*		#ab	abcde
both	*bcd*	Remove *	bcd	abcde
	de		de	abcde, bdef



- Note: *de means "de" must appear at the end of a keyword; *de* means anywhere in a keyword; de* means prefix of a keyword

Extension – Infix Truncation

- *Infix truncation*
  - wom*n: woman,women;
  - ≡ wom* then check if last character is "n"

- Can pattern matching be supported? i.e., can om match woman and women

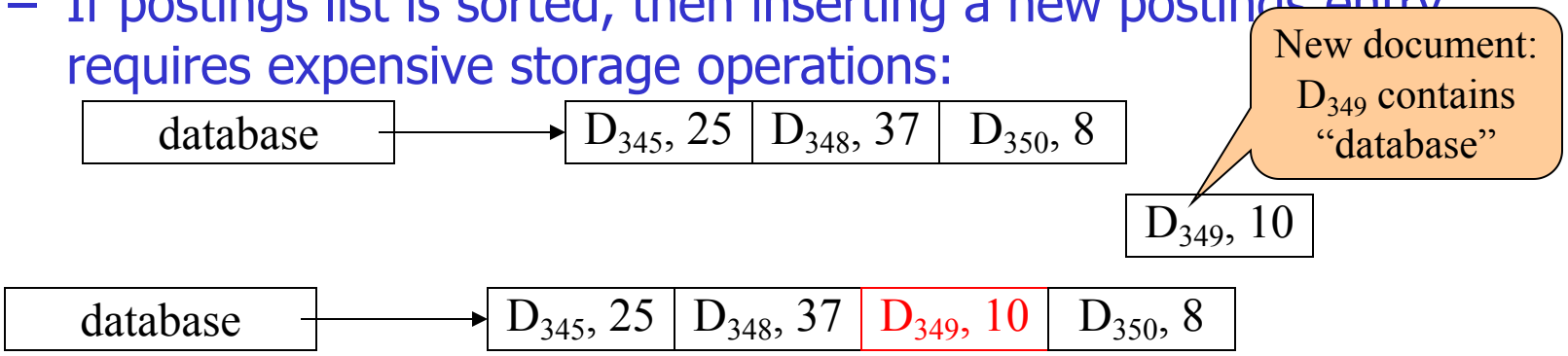
Conclusion

- The overall effect of including more and more information into the index is higher and higher storage overhead. It has been reported in the literature that the storage over head could reach 300%.
- Retrieval is more expensive since more postings are retrieved (e.g., when there are synonyms) and more processing is needed to check position information (e.g, matching of phrases)
- Inverted indexes are good for relatively static environment (few insertion and updates)

Insertion Overhead and Fast Insertion Method

Overheads for Inverted Indexing

- Worst case for the insertion of one document:
  - when every word in the document is unique; when the document contains *n* words, insertion needs to update *n* postings lists
- For each update on the postings list:
  - If postings list is not sorted, then new postings entry can be appended at the end; fast, but retrieval on unsorted postings list is very slow
  - If postings list is sorted, then inserting a new postings entry requires expensive storage operations:



Where is the time spent?

- Suppose we have 200,000 unique keywords, storing them in a b-tree of fan-out equal to 10 means that the b-tree has only 6 levels (Levels 0 to 5)
  - A search on the b-tree will take at most 6 disk accesses
  - Inserting a new keyword takes at most 6 disk accesses (assuming the insertion causes splitting all the way up to the root)

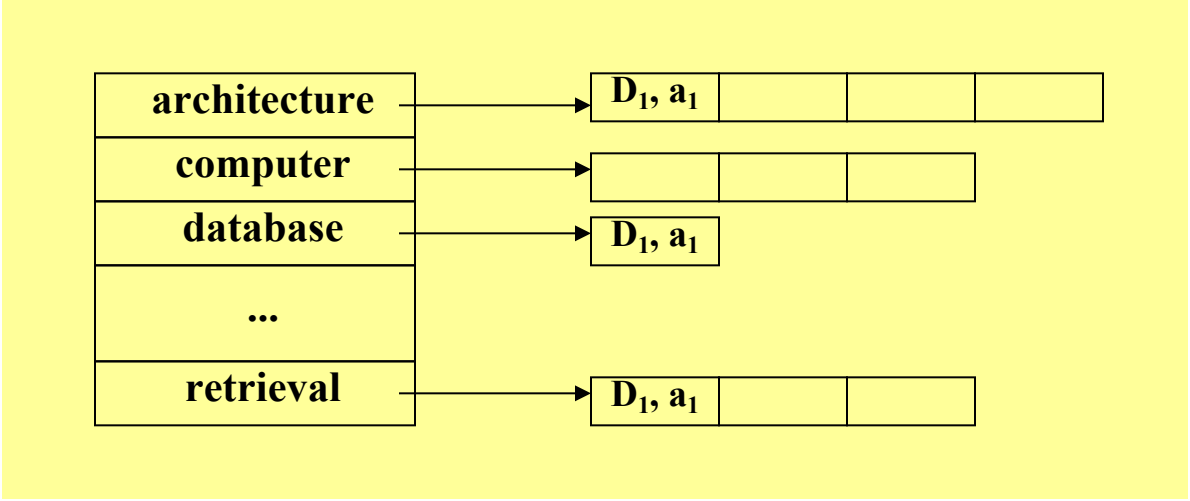
- Assume UNIX's page size is 512 bytes and each postings entry requires 8 bytes (a serious underestimate!), each page can hold 64 entries
  - It means if a keyword appears in 6,400 documents, the postings list would occupy 100 disk pages

Overheads for Inverted Indexing

- Insertion overhead depends on the frequency of update, requirement on the currency of the index, and workload of the system.
- For library applications, insertion overhead is not a problem; for newspaper and World Wide Web indexing, it is.
- When we design a new insertion method, we must consider:
  - size of main memory and temporary disk space available;
  - batch or online update of the index.

Deletion and Update

- We already studied how retrieval is done on an inverted file
- Update is a deletion followed by an insertion
- How do we update the index after a document is deleted?
  - E.g., what should be done after D_1 is deleted?



Deletion and Update

- To support deletion, a "forward index" is maintained to record words contained in a document

- Find the Doc ID of the document to be deleted
- Obtain the words contained in the document from the forward index
- Delete the postings entries for the Doc ID from the inverted file

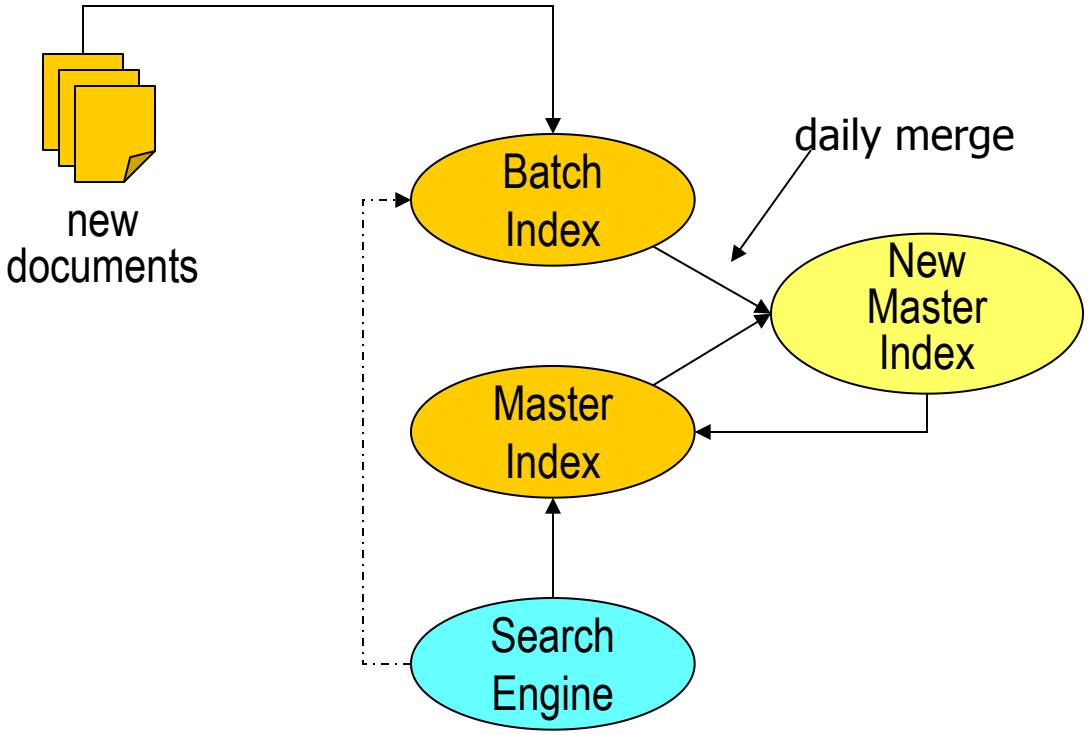
– Deletion is very expensive; to reduce deletion cost:
  – Keep a table containing document Ids of deleted documents
  – During retrieval, ignore deleted Doc Ids in the inverted file
  – Clean the inverted file periodically

Typical Indexes Needed for a Search Engine

- Regular keyword-based indexes:
  - Mapping indexes:
    - Word -> Word-ID
    - URL -> Page-ID
  - Inverted-file index
    - Word-ID -> {Page-ID, <word positions>}
  - Forward index
    - Page-ID -> {keywords}
  - Page properties
    - Page-ID -> title, URL, last-date-of-modification, size, etc.
- Link-based index (adjacency list representing the link structure):
  - Child-Page-ID -> Parent-Page-ID

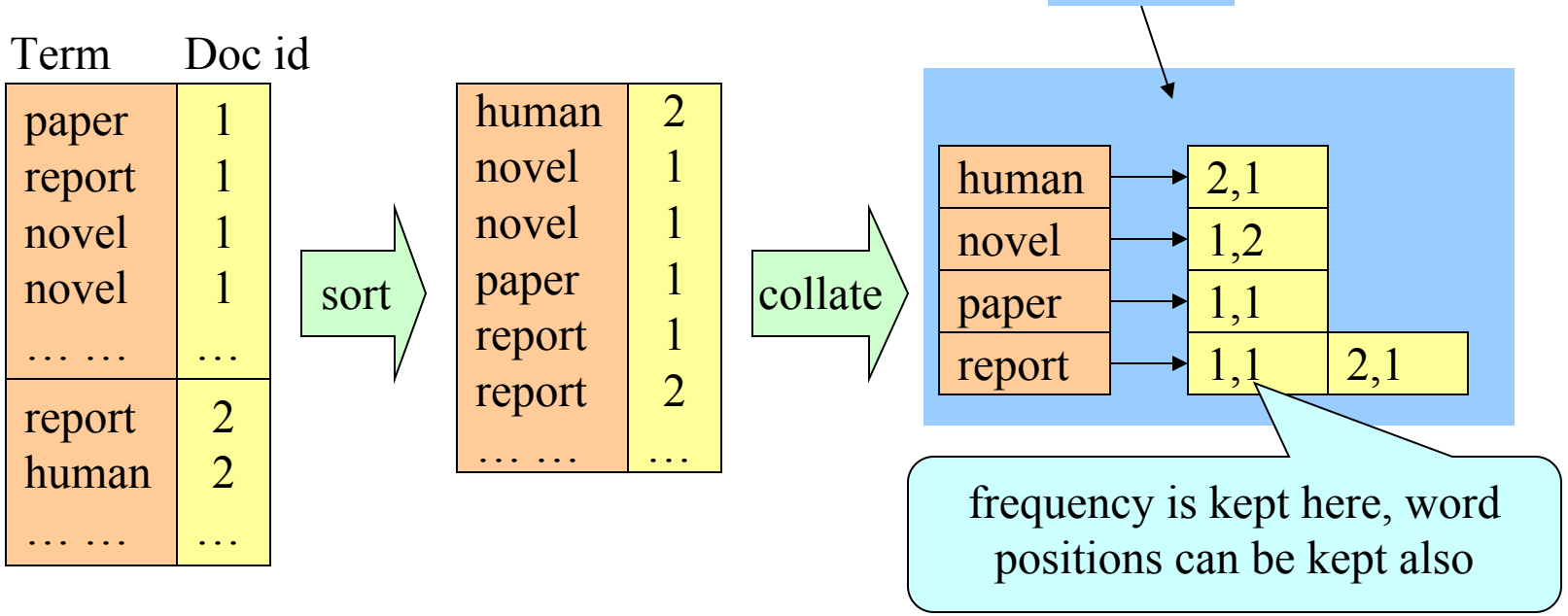
Batch/Bulk Insertion by Sorting

1. Insert new documents into an empty inverted index ("batch index")
2. Merge the batch with the "master" index periodically



Batch/Bulk Insertion by Sorting

- Extract the terms for each document and prepare a "batch" inverted file:



Batch inverted file is then inserted into the main inverted file – *why is batch insertion faster?*

Speed Improvement in Batch Insertion

- A postings list is retrieved from the inverted file, several postings entries are inserted, and the resulting list is put back into the inverted file
- Creation of the batch inverted file is inexpensive since it is rather small (if, say, only a few hundred insertions are batch together)
  - Maybe small enough to fit the entire batch index into main memory
- Batch size cannot be too small; otherwise, the chance of having more than one document containing the same word is lower
  - It can't be too large; otherwise inserting into the batch by itself is expensive
- Availability of new documents for search is delayed

Note that we assume so far that postings lists are stored in a single sequential file. It may be a bad idea from update point of view.

Bulk Insertion

- Bulk Insertion: create an inverted index from scratch for a large collection of documents
  - Inserting a document and updating the inverted index one document at a time is very inefficient

- Make use of main memory and sequential append to improve speed
  - Insert the documents batch by batch and choose a batch size that can match full use of the main memory available
  - All processing of a batch can be done in main memory
  - Size of a batch is confined by main memory size because of the need to sort the inverted index
- How to decrease "random" expansion of the inverted file?
  - Inserting records in the middle of a file is expensive
  - Pre-allocation of disk space to the inverted file

Fast Inversion Algorithm

Doc id	Word IDs
1	3, 5, 12, 14
2	1, 3, 4, 11, 12
3	2, 4, 5, 12, 13
4	1, 5, 11, 12, 14
5	3, 7, 13, 14

split

Load file 1

Doc id	Word IDs
1	3
2	1, 3, 4
3	2, 4
4	1
5	3

Load file 2

1	5
2	11
3	5
4	5, 11
5	7

Load file 3

1	12, 14
2	12
3	12, 13
4	12, 14
5	13, 14

insert

insert

insert

Word IDs	Doc id
1	2, 4
2	3
3	1, 2, 5
4	2, 3
5	1, 3, 4
7	5
11	2, 4
12	1, 2, 3, 4
13	3, 5
14	1, 4, 5

Final inverted file

Previous method requires sorting of 23 entries

Split into 3 "equal-size" load files:

	load files		
	1	2	3
no. of unique word IDs	4	3	3
no. of word-doc pairs	8	6	9
Range of Word IDs:	1-4	5-11	12-14

- Word Ids are always appended at the end
- Each partition has known size and storage can be pre-allocated

Algorithm of Fast-Inv

1 Given the document vector file, find out the number of document-word pairs in the documents.

2 Given the memory size, try to divide the document vector file into *j* load files so that:

- each load file, containing document-word pairs, can be load into the *main memory* for processing (inversion, sorting, etc)
- each load file has about the same number of unique words in it
- *j* is as small as possible.
- word Ids in load file *i* are greater than word Ids in load file *j*, for all $j < i$.

Algorithm of Fast-Inv

3 Build the inverted file starting from the first load file.

- Because of the way load files are created, the inverted file is built starting from the lowest word ID in ascending order.
- The information collected about the documents and words allows storage to be *pre-allocated* for the posting file. Postings can be inserted directly into the file without searching or causing storage to be allocated (as in the case of linked list).
- The exact amount of gain depends on the underlying OS d file

Requires reading three files
1) read input file to generate file statistics
2) read input file again to generate load files
3) read the load files to generate the final inverted file

and writing two files:
1) write the load files
2) write the final inverted file

Take Home Messages

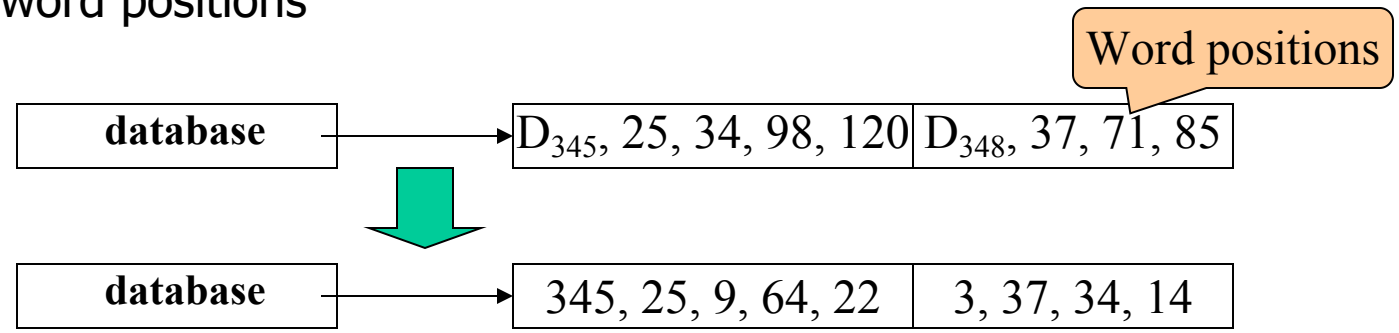
- Indexing is a MUST for any search engine with non-trivial size
- Inverted file which can efficiently look up documents that contain a word is by far the "standard" index method
- Index maintenance is VERY important for large-scale search engines
  - The larger the index, the harder it is to maintain
- Use of batch update and an index migration policy
- Make use of main memory as much as possible to reduce random accesses to hard disks
- Use of multiple indexes

Other Implementation Issues

Fine details: Compressing the inverted file
Scalability: Searching on many machines
Relational database: Indexing and searching text strings

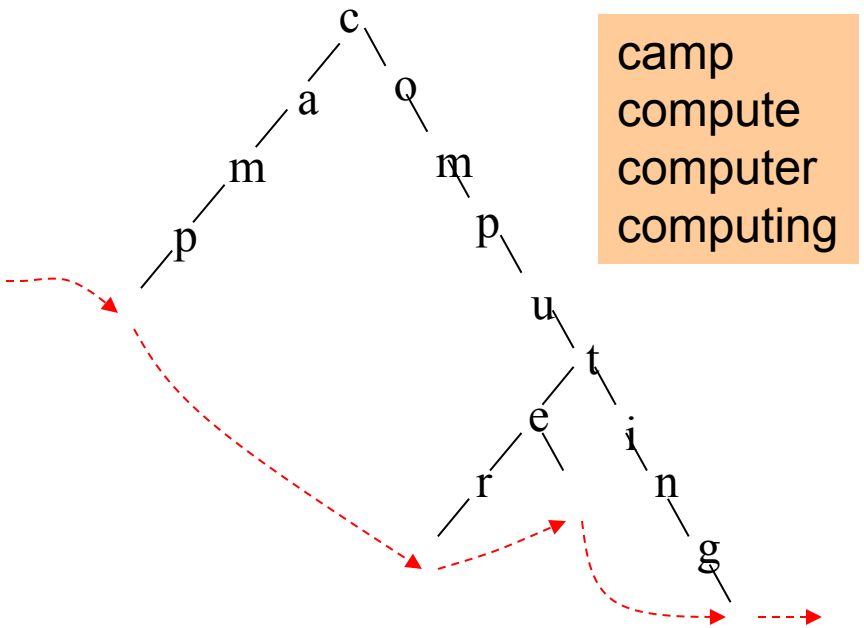
Further Optimization on Inverted Files

- Compression of index and postings files
  - number of bytes assigned to a document ID or word position
  - 16 bits are not enough; 32 bits take a lot of space
  - Just record the differences between successive document Ids or word positions



- Fast insertion methods
  - Batch updates: create an empty index for new updates and insertion and then merge index information into the main index

Trie



camp
compute
computer
computing

- Good for partial match within a word and sequential retrieval of a set of words
- Size of trie depends on:
  - amount of overlaps between words
  - size of alphabet

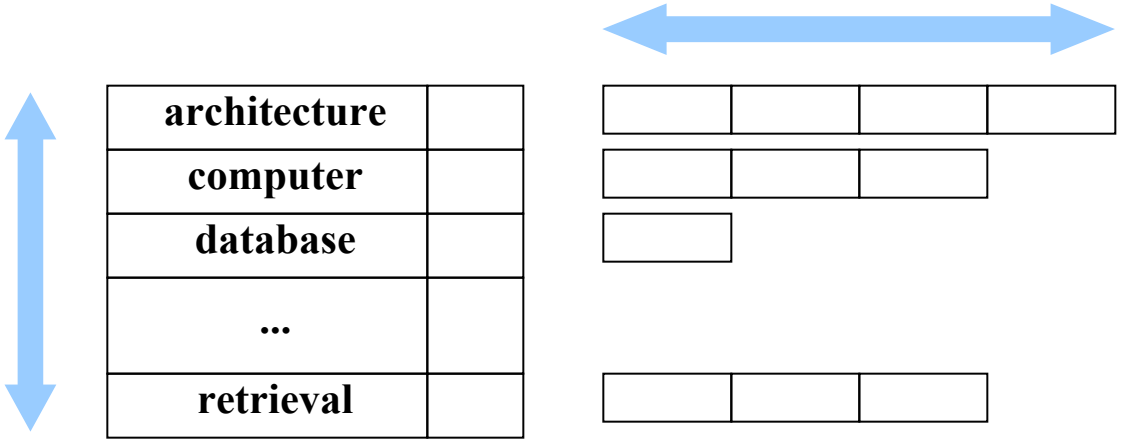
- Condense sequence of nodes into one node (e.g., "ompu" stored in one node)
- B+tree/trie is good for sequential retrieval of keywords, e.g., "get all words between camp and computing"
- Hash file is good for fast random retrieval of individual words

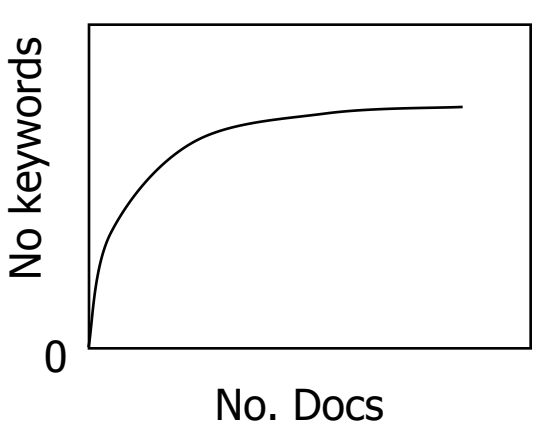
back

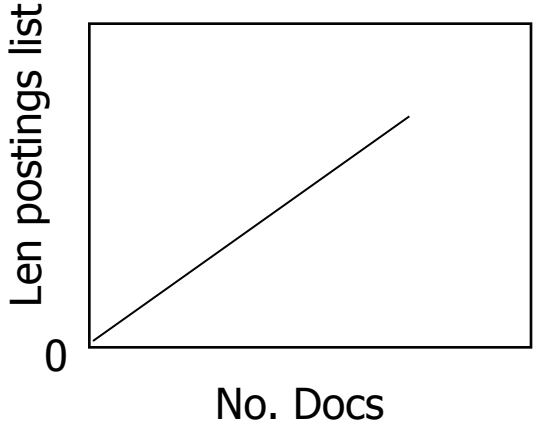
Scalability Issues

- As the number of documents increases, a single inverted file cannot provide fast-enough search (not to mention updates). Why?
- Inverted file speeds up searching the indexed keywords but the number of indexed keywords would grow slowly
- Length of a postings list is linear to the number of documents (in fact, to the total number of keywords in the collection)

Scalability Issues

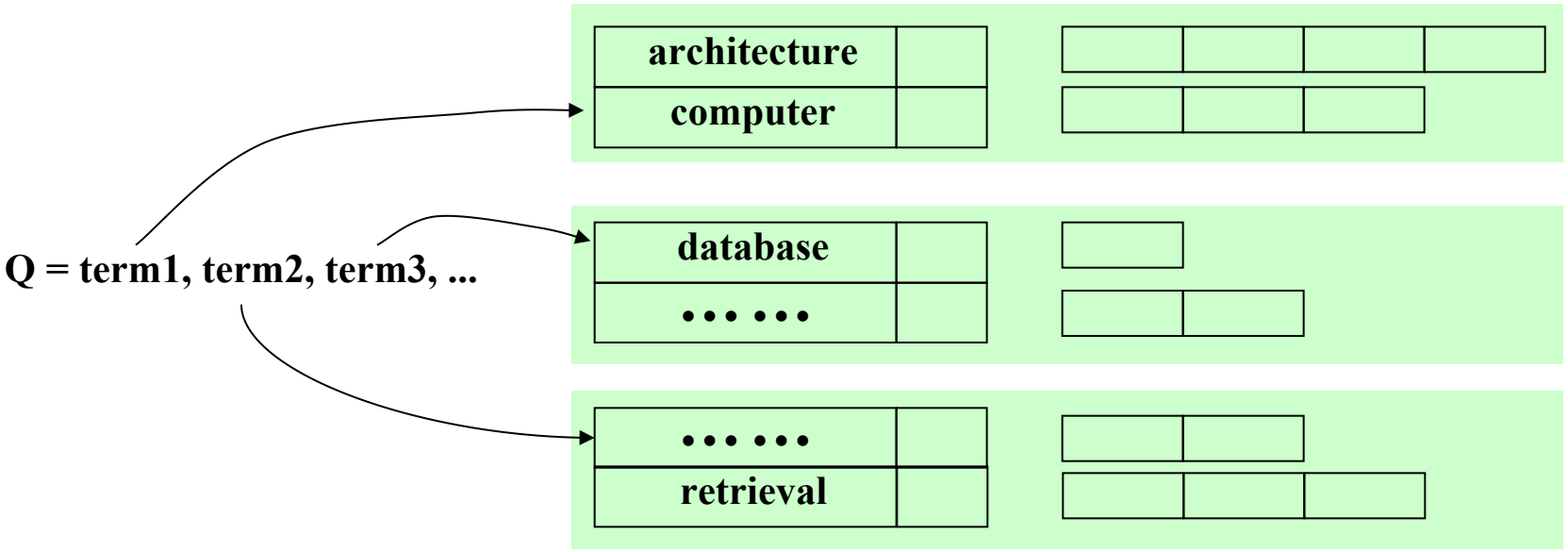






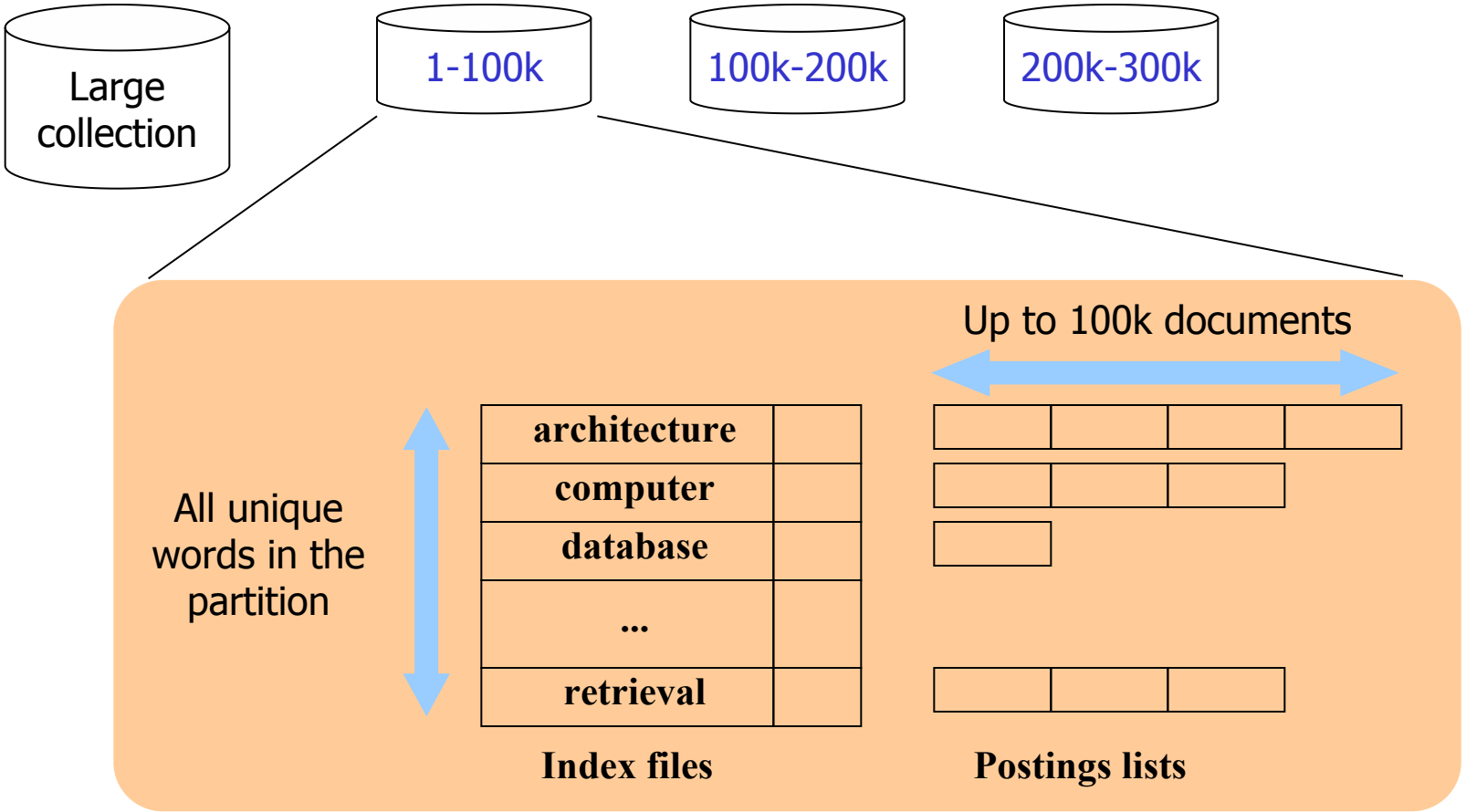
Partitioning and Parallel Processing

- The index can be partitioned on multiple servers which could speed up search
- Horizontal (index) partitioning does not work well because it speeds up search on the index but not the posting lists



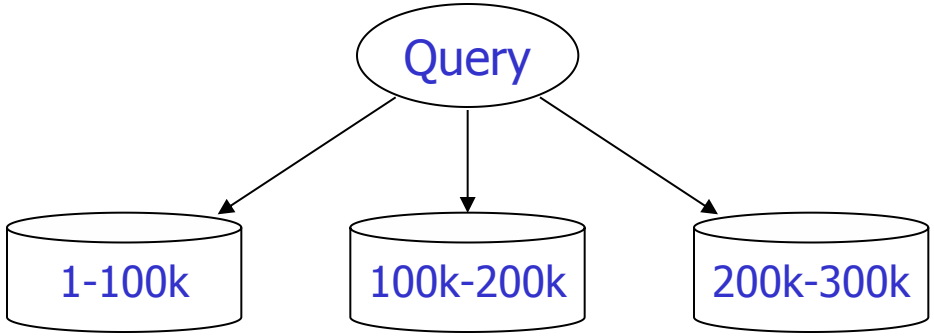
Collection Partitioning

- Partitioning a large collection into small ones and build an index for each of them: does not speed up index search but shorten postings lists

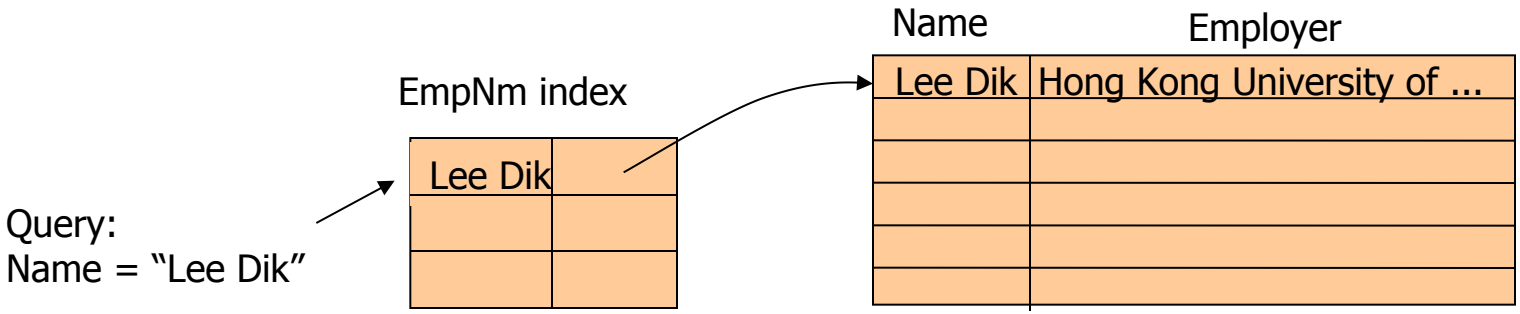


Searching Multiple Collections

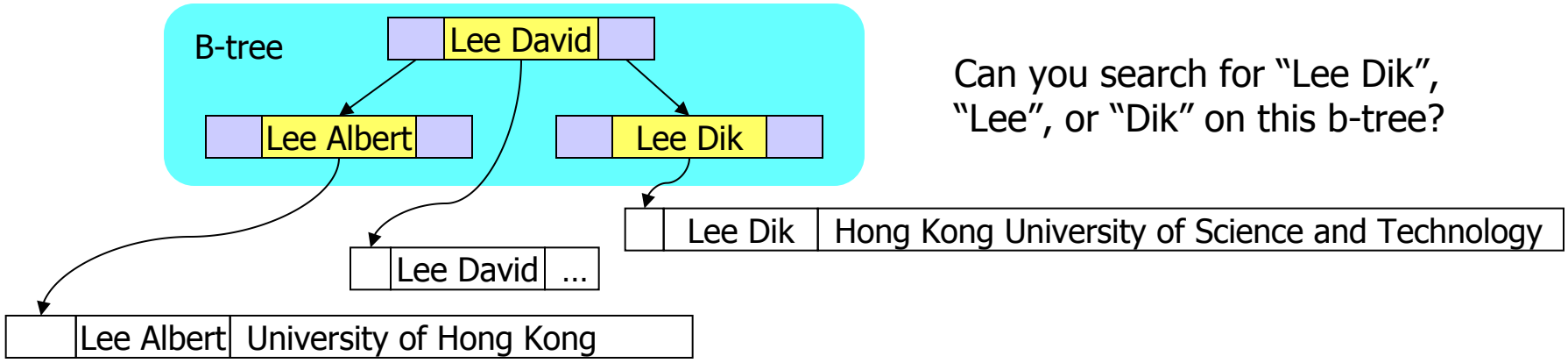
- A query must be searched on all partitions (i.e., multiple "tiny" collections are searched)
  - Is ranking performed on each collection locally?
  - How to merge the results set together?
  - Can we do collection ranking and then document ranking within the selected collections?



Indexing on Relational Databases

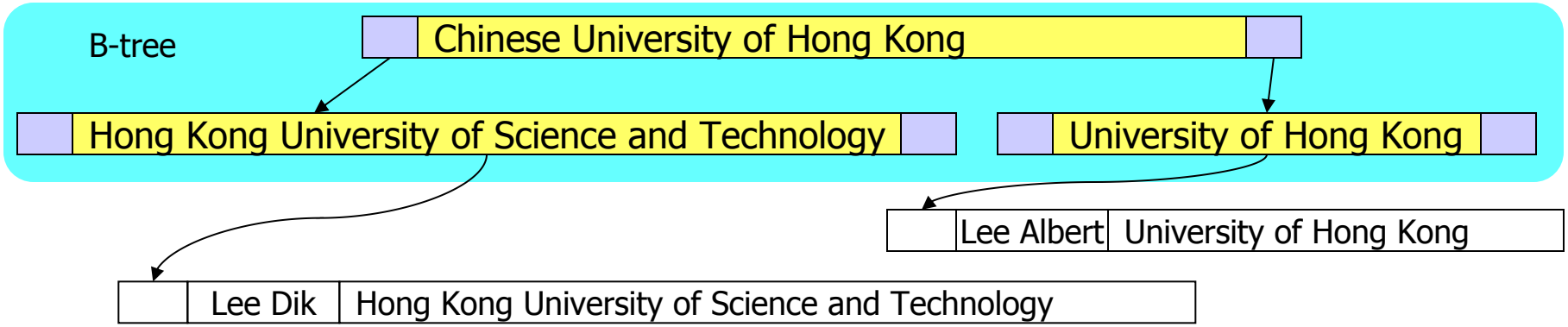


- SQL: create index EmpNm on Employee(Name)
- Index structures: hash file, B-tree, etc.
- Relation database indexes the whole value
  - Search can be performed on the whole value or a prefix of it



Can you search for "Lee Dik", "Lee", or "Dik" on this b-tree?

Worse with Long Text Values



- SQL: create index EmpEmployerName on Employee(Employer)
- Index and search on whole attribute value, e.g.,
  - "Hong Kong University of Science and Technology" will be stored as a single string in the EmpEmployerName index
  - search "Hong Kong University of Science and Technology" can make use of the index for a fast search, but not "Hong Kong" or "University"
- What is the storage overhead?
  - Whole values repeated in index; only frequently queried columns are indexed
- How to make the index good for search engines?
  - Index individual words as we have seen in previous inverted files

Take Home Message

- An inverted file consists of two parts:
  - A keyword index allows you to find keywords in the index
  - A postings list that tells you which documents contain the keyword, plus some additional information such as word positions, etc.
- Keyword index is typically stored as a b-tree or hash file, depending on which file package you use
- A postings list is stored as a string in the key-value pair
  - Simple but incur high insertion/update/deletion costs, because the string is a linear, contiguous data structure
  - We show that manipulating long postings lists are expensive – much more expensive than searching the index
  - In real life, a better design is needed (try to do it in your project)
- If DBMS (e.g., mySQL) is used to store and search documents, performance deteriorates quickly because a plain DBMS does not support efficient substring search