Information Processing Technology of Internet of Things

Chapter 3
Information Retrieval

Wu Liu

Beijing Key Lab of Intelligent Telecomm. Software and Multimedia
Beijing University of Posts and Telecommunications

3.6 Indexing and Searching

Introduction

- Although **efficiency** might seem a secondary issue compared to **effectiveness**, it can rarely be neglected in the design of an IR system
- Efficiency in IR systems**:** to process user queries with minimal requirements of computational resources
- As we move to larger-scale applications, efficiency becomes more and more important
  - For example, in Web search engines that index terabytes of data and serve hundreds or thousands of queries per second

Introduction

- Index: a data structure built from the text to speed up the searches
- In the context of an IR system that uses an index, the efficiency of the system can be measured by:
  - **Indexing time**: Time needed to build the index
  - **Indexing space**: Space used during the generation of the index
  - **Index storage**: Space required to store the index
  - **Query latency**: Time interval between the arrival of the query and the generation of the answer
  - **Query throughput**: Average number of queries processed per second

Introduction

- When a text is updated, any index built on it must be updated as well
- Current indexing technology is not well prepared to support very frequent changes to the text collection
- **Semi-static collections**: collections which are updated at reasonable regular intervals (say, daily)
- Most real text collections, including the Web, are indeed semi-static
  - For example, although the Web changes very fast, the crawls of a search engine are relatively slow
- For maintaining freshness, incremental indexing is used

3.6.1 Inverted Indexes

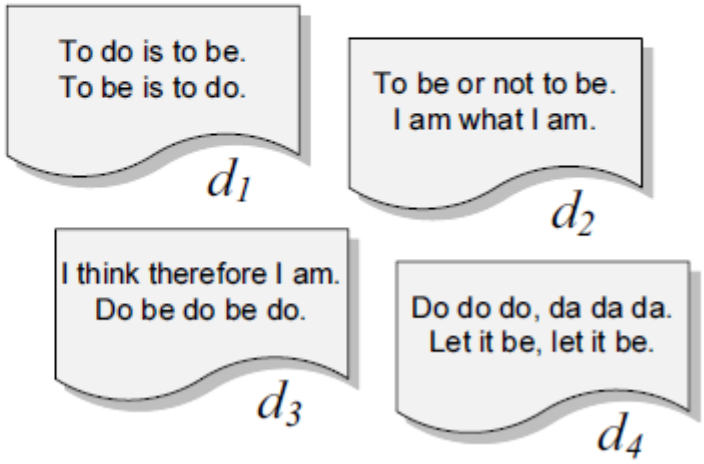
Basic Concepts

- Inverted index: a word-oriented mechanism for indexing a text collection to speed up the searching task
- The inverted index structure is composed of two elements: the **vocabulary** and the **occurrences**
- The vocabulary is the set of all different words in the text
- For each word in the vocabulary the index stores the documents which contain that word (inverted index)

Basic Concepts

- **Term-document matrix**: the simplest way to represent the documents that contain each word of the vocabulary

Vocabulary	n_i	d_1	d_2	d_3	d_4
to	2	4	2	-	-
do	3	2	-	3	3
is	1	2	-	-	-
be	4	2	2	2	2
or	1	-	1	-	-
not	1	-	1	-	-
I	2	-	2	2	-
am	2	-	2	1	-
what	1	-	1	-	-
think	1	-	-	1	-
therefore	1	-	-	1	-
da	1	-	-	-	3
let	1	-	-	-	2
it	1	-	-	-	2



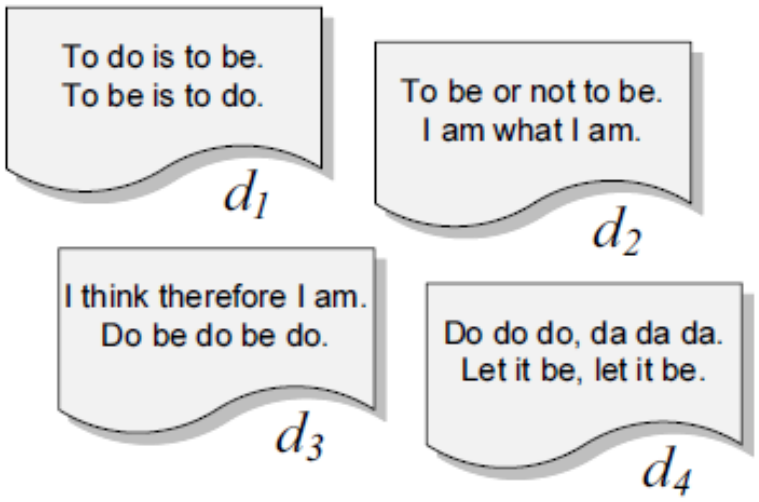
Basic Concepts

- The main problem of this simple solution is that it requires too much space
- As this is a sparse matrix, the solution is to associate a list of documents with each word
- The set of all those lists is called the **occurrences**

Basic Concepts

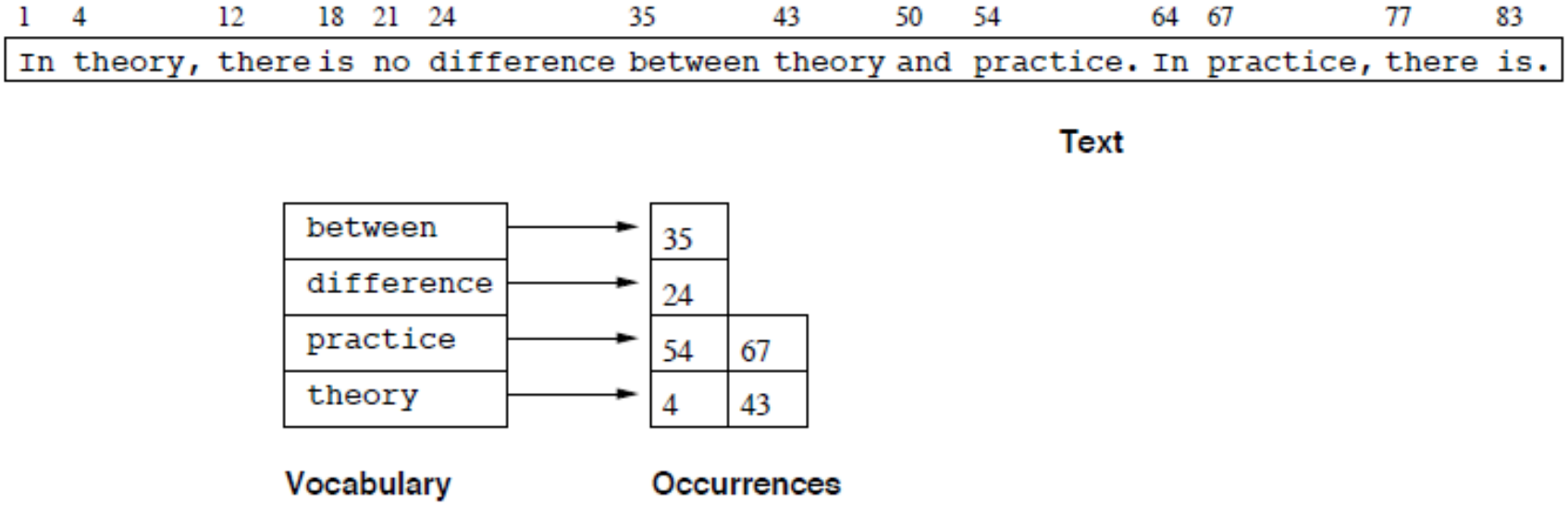
- Basic inverted index

Vocabulary	n_i	Occurrences as inverted lists
to	2	[1,4],[2,2]
do	3	[1,2],[3,3],[4,3]
is	1	[1,2]
be	4	[1,2],[2,2],[3,2],[4,2]
or	1	[2,1]
not	1	[2,1]
I	2	[2,2],[3,2]
am	2	[2,2],[3,1]
what	1	[2,1]
think	1	[3,1]
therefore	1	[3,1]
da	1	[4,3]
let	1	[4,2]
it	1	[4,2]



Full Inverted Indexes

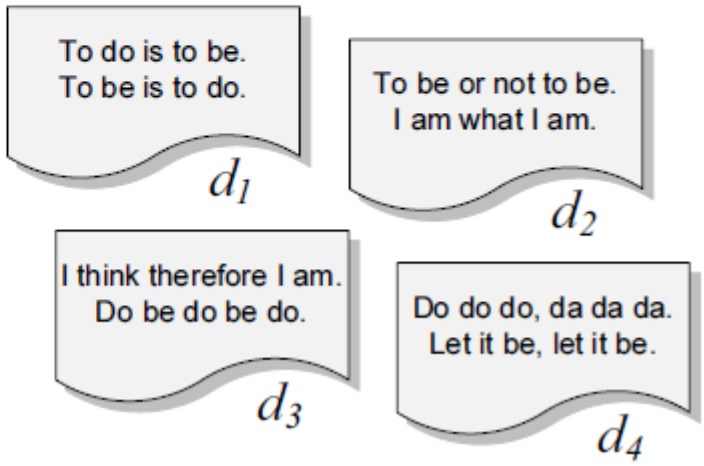
- The basic index is not suitable for answering phrase or proximity queries
- Hence, we need to add the positions of each word in each document to the index (full inverted index)



Full Inverted Indexes

- In the case of multiple documents, we need to store one occurrence list per term-document pair

Vocabulary	n_i	Occurrences as full inverted lists
to	2	[1,4,[1,4,6,9]],[2,2,[1,5]]
do	3	[1,2,[2,10]],[3,3,[6,8,10]],[4,3,[1,2,3]]
is	1	[1,2,[3,8]]
be	4	[1,2,[5,7]],[2,2,[2,6]],[3,2,[7,9]],[4,2,[9,12]]
or	1	[2,1,[3]]
not	1	[2,1,[4]]
I	2	[2,2,[7,10]],[3,2,[1,4]]
am	2	[2,2,[8,11]],[3,1,[5]]
what	1	[2,1,[9]]
think	1	[3,1,[2]]
therefore	1	[3,1,[3]]
da	1	[4,3,[4,5,6]]
let	1	[4,2,[7,10]]
it	1	[4,2,[8,11]]



Full Inverted Indexes

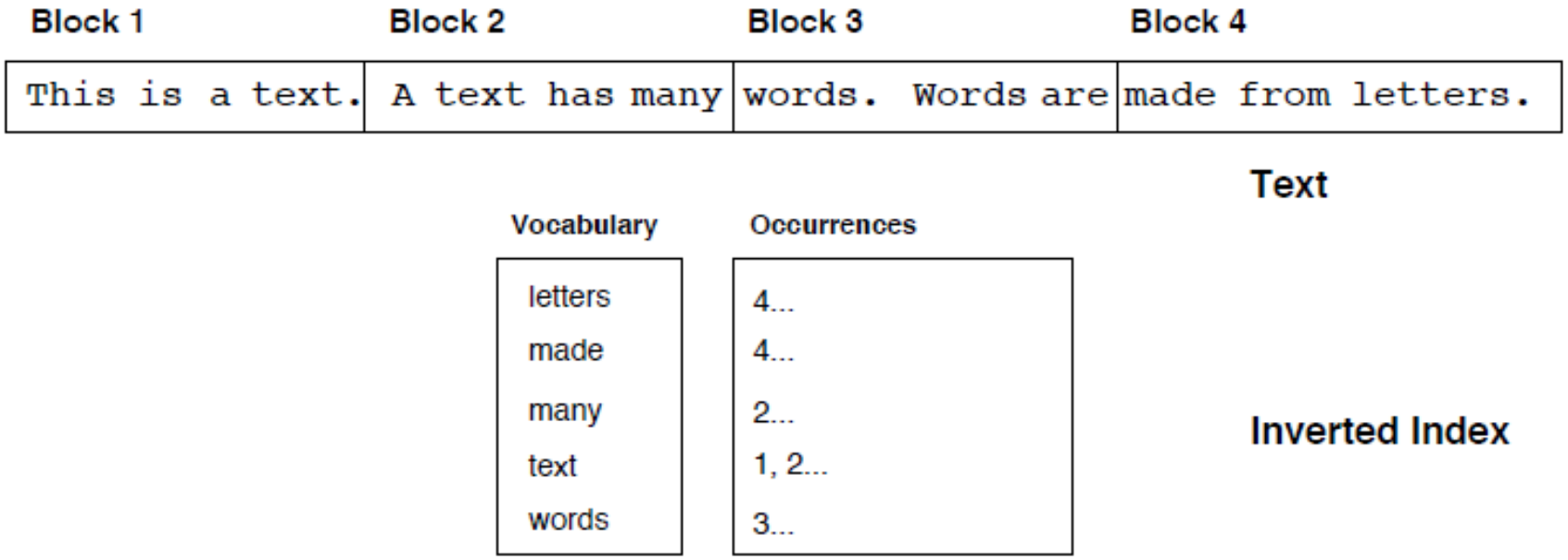
- The space required for the vocabulary is rather small
- Heaps' law: the vocabulary grows as $O(n^{\beta})$, where
  - n is the collection size
  - β is a collection-dependent constant between 0.4 and 0.6
- For instance, in the TREC-3 collection, the vocabulary of 1 gigabyte of text occupies only 5 megabytes
- This may be further reduced by stemming and other normalization techniques

Full Inverted Indexes

- The occurrences demand much more space
- The extra space will be O(n) and is around
  - 40% of the text size if stopwords are omitted
  - 80% when stopwords are indexed
- Document-addressing indexes are smaller, because only one occurrence per file must be recorded, for a given word
- Depending on the document (file) size, document-addressing indexes typically require 20% to 40% of the text size

Full Inverted Indexes

- To reduce space requirements, a technique called block addressing is used
- The documents are divided into blocks, and the occurrences point to the blocks where the word appears
  - The blocks can be of fixed size or they can be defined using the division of the text collection into documents
  - The division into blocks of fixed size improves efficiency at retrieval time



Full Inverted Indexes

- The Table below presents the projected space taken by inverted indexes for texts of different sizes

Index granularity	Single document (1 MB)		Small collection (200 MB)		Medium collection (2 GB)	
Addressing words	45%	73%	36%	64%	35%	63%
Addressing documents	19%	26%	18%	32%	26%	47%
Addressing 64K blocks	27%	41%	18%	32%	5%	9%
Addressing 256 blocks	18%	25%	1.7%	2.4%	0.5%	0.7%

Single Word Queries

- The simplest type of search is that for the occurrences of a single word
- The vocabulary search can be carried out using any suitable data structure
  - Ex: hashing, tries, or B-trees
- The first two provide O(m) search cost, where m is the length of the query
- We note that the vocabulary is in most cases sufficiently small so as to stay in main memory
- The occurrence lists, on the other hand, are usually fetched from disk

Multiple Word Queries

- If the query has more than one word, we have to consider two cases:
  - conjunctive (AND operator) queries
  - disjunctive (OR operator) queries
- **Conjunctive queries** imply to search for all the words in the query, obtaining one inverted list for each word
- Following, we have to **intersect** all the inverted lists to obtain the documents that contain all these words
- For **disjunctive queries** the lists must be merged

List Intersection

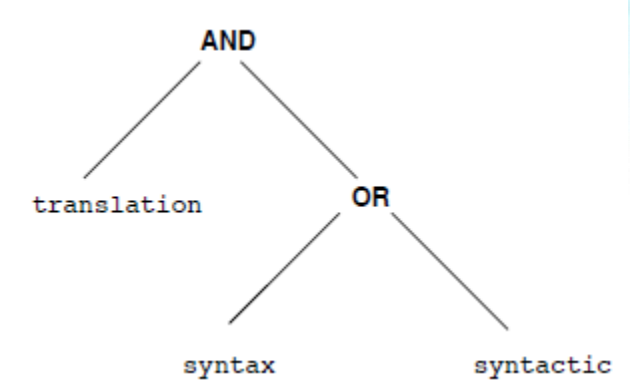
- The most time-demanding operation on inverted indexes is the merging of the lists of occurrences
  - Thus, it is important to optimize it
- Consider one pair of lists of sizes m and n respectively, stored in consecutive memory, that needs to be intersected
- If m is much smaller than n, it is better to do m binary searches in the larger list to do the intersection
- If m and n are comparable, Baeza-Yates devised a double binary search algorithm

Phrase and Proximity Queries

- Context queries are more difficult to solve with inverted indexes
- The lists of all elements must be traversed to find places where
  - all the words appear in sequence (for a phrase), or
  - appear close enough (for proximity)
  - these algorithms are similar to a list intersection algorithm
  - e.g., “a b c” at i, i+1, i+2 or |i-j|<=k+1

Boolean Queries

- In boolean queries, a **query syntax tree** is naturally defined



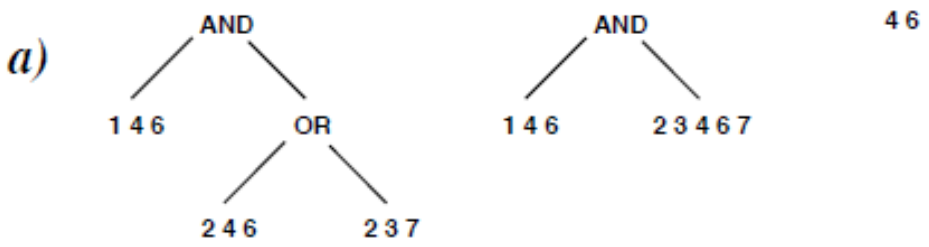
- Normally, for boolean queries, the search proceeds in three phases:
  - the first phase determines which documents to match
  - the second determines the likelihood of relevance of the documents matched
  - the final phase retrieves the exact positions of the matches to allow highlighting them during browsing, if required

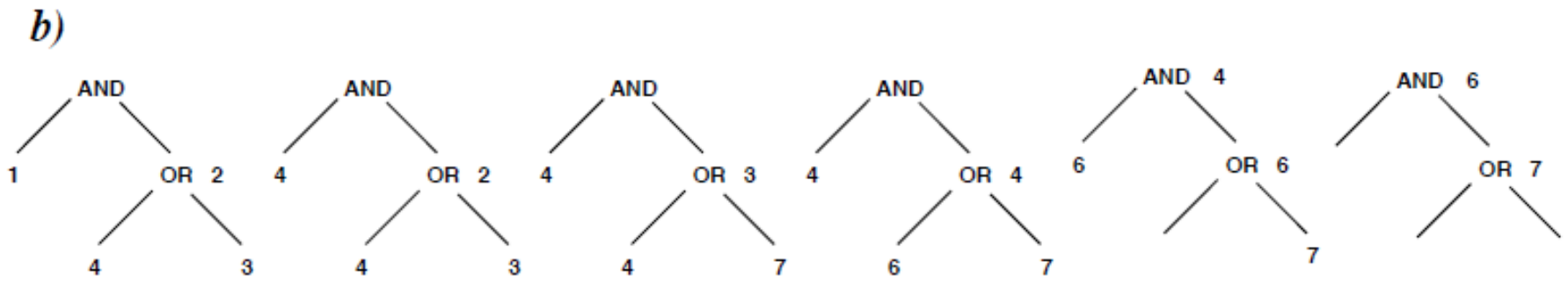
Boolean Queries

- Once the leaves of the query syntax tree find the classifying sets of documents, these sets are further operated by the internal nodes of the tree
- Under this scheme, it is possible to evaluate the syntax tree in **full or lazy form**
  - In the full evaluation form, both operands are first completely obtained and then the complete result is generated
  - In lazy evaluation, the partial results from operands are delivered only when required, and then the final result is recursively generated

Boolean Queries

- Processing the internal nodes of the query syntax tree
  - In (a) full evaluation is used
  - In (b) we show lazy evaluation in more detail





Ranking

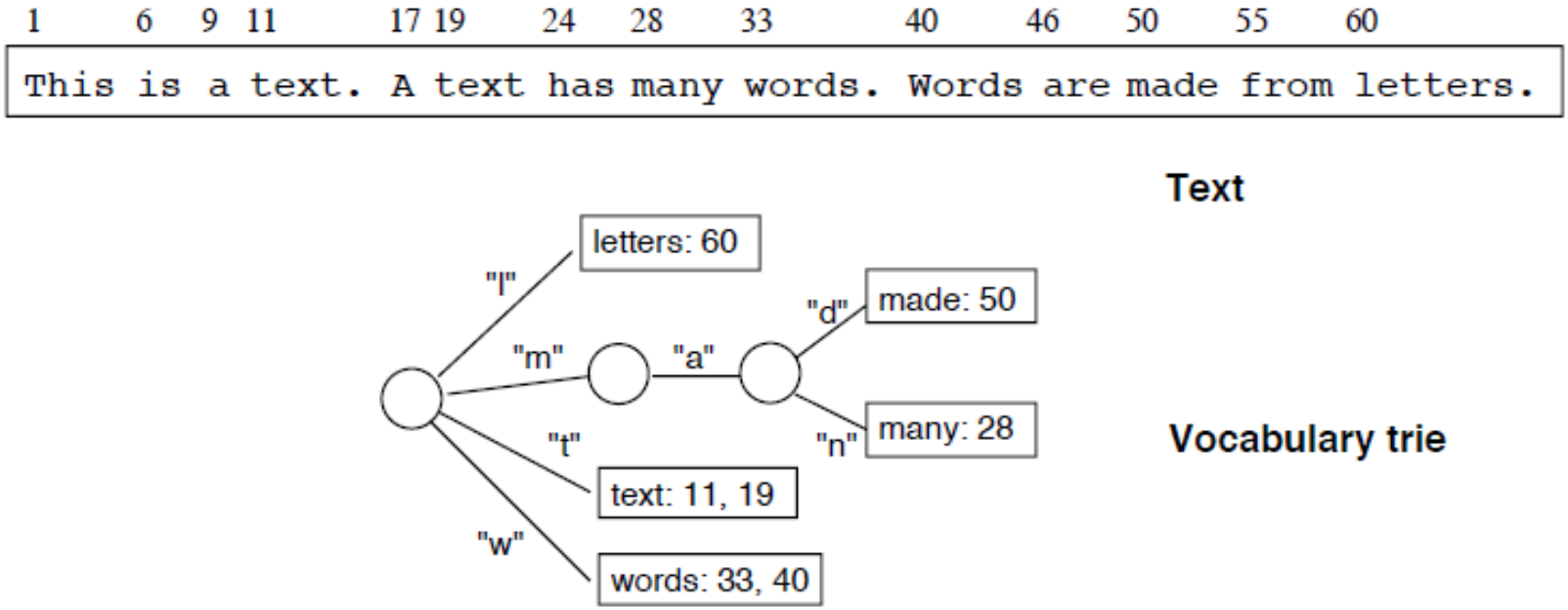
- How to find the top-k documents and return them to the user when we have weight-sorted inverted lists?
- If we have a single word query, the answer is trivial as the list can be already sorted by the desired ranking
- For other queries, we need to merge the lists

Internal Algorithms

- Building an index in internal memory is a relatively simple and low-cost task
- A dynamic data structure to hold the vocabulary (B-tree, hash table, etc.) is created empty
- Then, the text is scanned and each consecutive word is searched for in the vocabulary
- If it is a new word, it is inserted in the vocabulary before proceeding

Internal Algorithms

- A large array is allocated where the identifier of each consecutive text word is stored
- A full-text inverted index for a sample text with the incremental algorithm:



Internal Algorithms

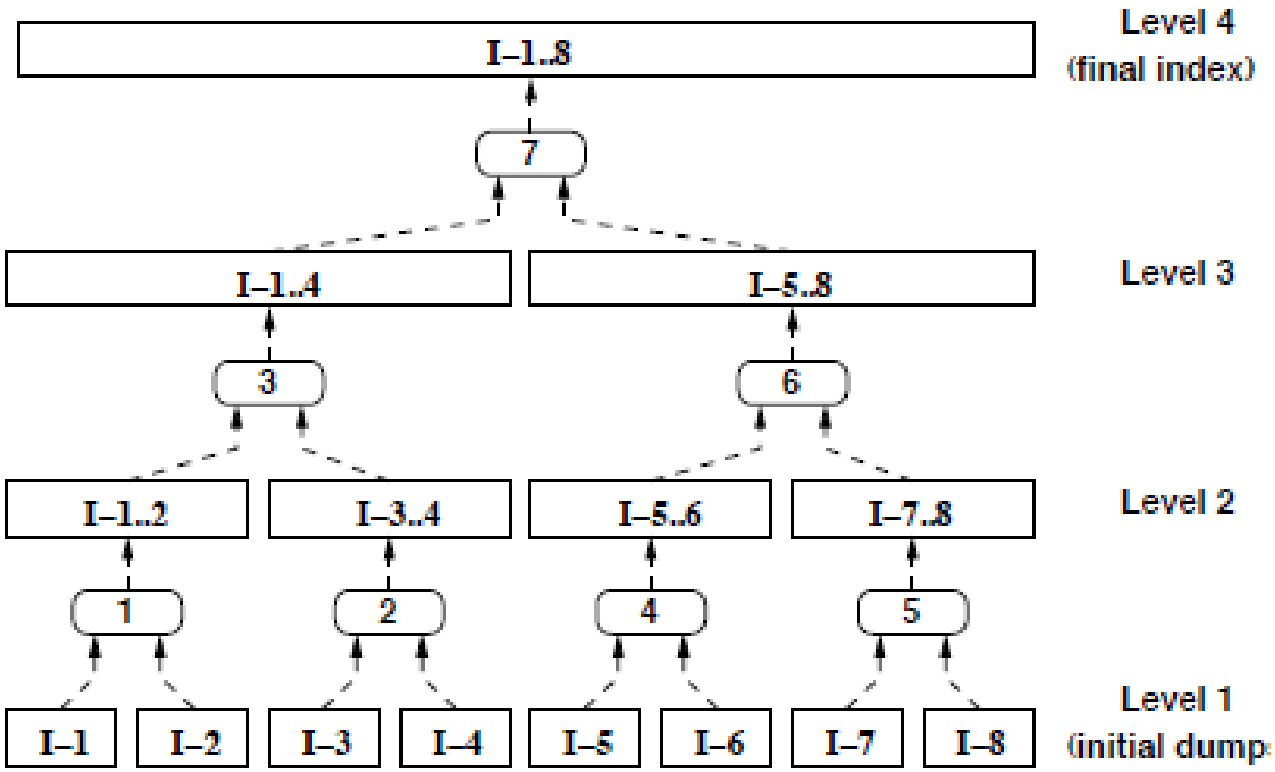
- The vocabulary and the lists of occurrences are written on two distinct disk files
- The vocabulary contains, for each word, a pointer to the position of the inverted list of the word
- This allows the vocabulary to be kept in main memory at search time in most cases

External Algorithms

- All the previous algorithms can be extended by using them until the main memory is exhausted
- At this point, the **partial** index Ii obtained up to now is written to disk and erased from main memory
- These indexes are then **merged** in a hierarchical fashion

External Algorithms

- Merging the partial indexes in a binary fashion
  - Rectangles represent partial indexes, while rounded rectangles represent merging operations



External Algorithms

- In general, maintaining an inverted index can be done in three different ways:
  - **Rebuild**
    - If the text is not that large, rebuilding the index is the simplest solution
  - **Incremental updates**
    - We can amortize the cost of updates while we search
    - That is, we only modify an inverted list when needed
  - **Intermittent merge**
    - New documents are indexed and the resultant partial index is merged with the large index
    - This in general is the best solution

Structural Queries

- Let us assume that the structure is marked in the text using **tags**
- The idea is to make the index take the tags as if they were words
- After this process, the inverted index contains all the information to answer structural queries

Structural Queries

- Consider the query:
  - select structural elements of type **A** that contain a structure of type **B**
- The query can be translated into finding <A> followed by <B> without </A> in between
- The positions of those tags are obtained with the full-text index
- Many queries can be translated into a search for tags plus validation of the sequence of occurrences
- In many cases this technique is efficient and its integration into an existing text database is simpler

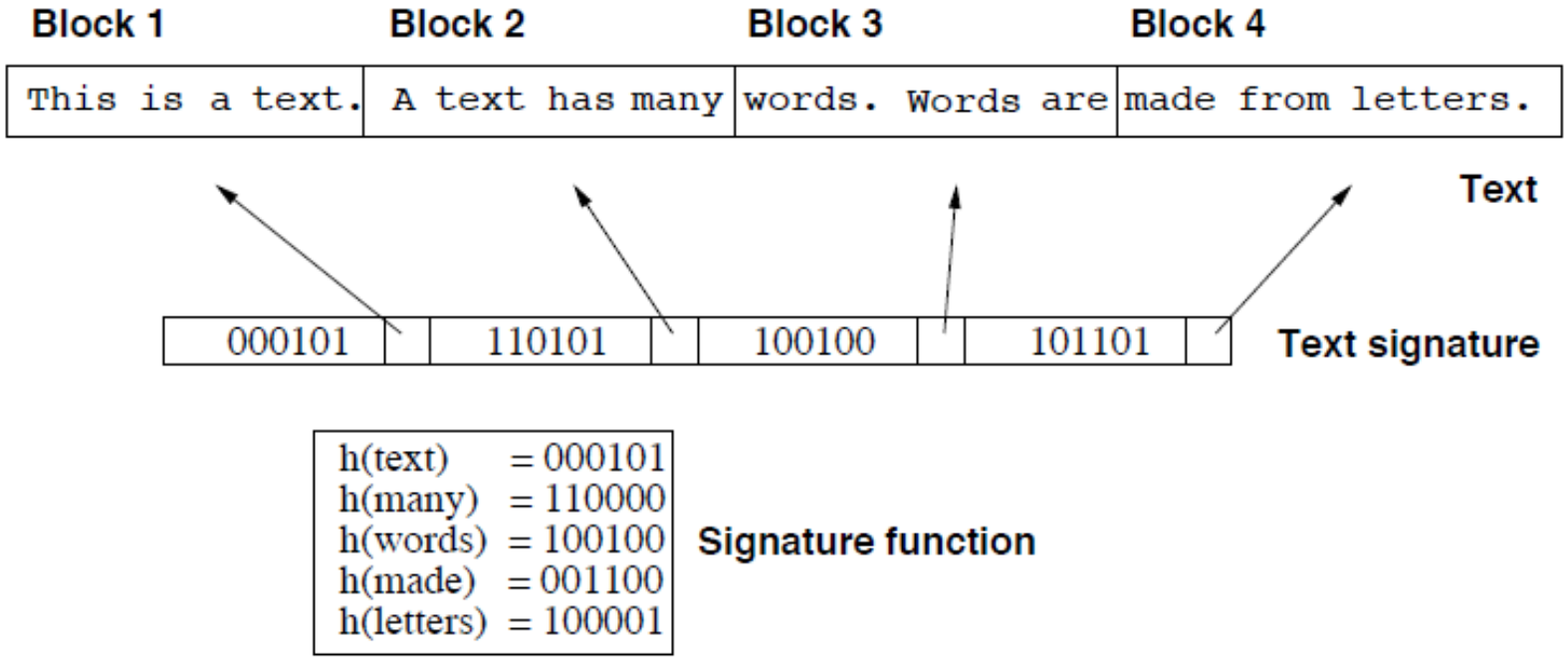
3.6.2 Signature Files

Signature Files

- **Signature files** are word-oriented index structures based on hashing
- They pose a low overhead, at the cost of forcing a sequential search over the index
- Since their search complexity is linear, it is suitable only for not very large texts
- Nevertheless, inverted indexes outperform signature files for most applications

Structure

- A signature divides the text in blocks of b words each, and maps words to bit masks of B bits
- This mask is obtained by bit-wise ORing the signatures of all the words in the text block



Structure

- If a word is present in a text block, then its signature is also set in the bit mask of the text block
- Hence, if a query signature is not in the mask of the text block, then the word is not present in the text block
- However, it is possible that all the corresponding bits are set even though the word is not there
  - This is called a **false drop**
- A delicate part of the design of a signature file is:
  - to ensure the probability of a false drop is low, and
  - to keep the signature file as short as possible

Structure

- The hash function is forced to deliver bit masks which have at least *l* bits set
- A good model assumes that *l* bits are randomly set in the mask (with possible repetition)

Searching

- Searching a single word is made by comparing its bit mask W with the bit masks Bi of all the text blocks
- Whenever (W & Bi = W), where & is the bit-wise AND, the text block **may** contain the word
- Hence, an online traversal must be performed to verify if the word is actually there
- This scheme is more efficient to search phrases and reasonable proximity queries
  - This is because **all** the words must be present in a block in order for that block to hold the phrase or the proximity query

Construction

- The construction of a signature file is rather easy:
  - The text is simply cut in blocks, and for each block an entry of the signature file is generated
- Adding text is also easy, since it is only necessary to keep adding records to the signature file
- Text deletion is carried out by deleting the appropriate bit masks

3.7 Parallel and Distributed IR

Introduction

- The volume of online content today is staggering and it has been growing at an exponential rate
- On at a slightly smaller scale, the largest corporate intranets now contain several million Web pages
- As document collections grow larger, they become more expensive to manage
- In this scenario, it is necessary to consider alternative IR architectures and algorithms
- The application of parallelism and distributed computing can greatly enhance the ability to scale IR algorithms

Data Partitioning

- IR tasks are typically characterized by a small amount of processing applied to a large amount of data
- How to partition the **document collection and the index?**

Data Partitioning

- Figure below presents a high level view of the data processed by typical search algorithms

		Indexing Items					
		k_1	k_2	...	k_i	...	k_t
Documents	d_1	$w_{1,1}$	$w_{2,1}$	...	$w_{i,1}$	...	$w_{t,1}$
	d_2	$w_{1,2}$	$w_{2,2}$	...	$w_{i,2}$	...	$w_{t,2}$
	...	...	...	...	...	...	...
	d_j	$w_{1,j}$	$w_{2,j}$	...	$w_{i,j}$	...	$w_{t,j}$
	...	...	...	...	...	...	...
	d_N	$w_{1,N}$	$w_{2,N}$	...	$w_{i,N}$	...	$w_{t,N}$

- Each row represents a document dj and each column represents an indexing item ki

Data Partitioning

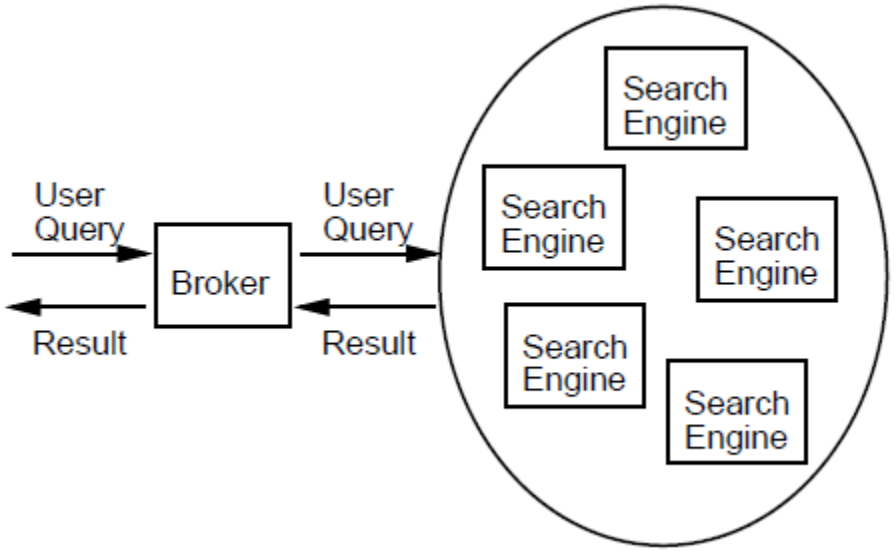
- **Document partitioning** slices the matrix horizontally, dividing the documents among the subtasks
- The N documents in the collection are distributed across the P processors in the system
- During query processing, each parallel process evaluates the query on N/P documents
- The results from each of the sub-collections are combined into a final result list

Data Partitioning

- In **term partitioning**, the matrix is sliced vertically
  - It divides the indexing items among the P processors
- In this way, the evaluation procedure for each document is spread over multiple processors
- Other possible partition strategies include divisions by **language or other intrinsic characteristics** of the data
- It may be the case that each independent search server is focused on a particular subject area

Collection Partitioning

- When the distributed system is **centrally administered, more** options are available
- The first option is just the replication of the collection across all search servers
- A **broker** routes queries to the search servers and balances the load on the servers:



Collection Partitioning

- The second option is random distribution of the documents
- This is appropriate when a large document collection must be distributed for performance reasons
- However, the documents will always be viewed and searched as if they are part of a single, logical collection
- The broker broadcasts every query to all search servers and combines the results for the user

Collection Partitioning

- The final option is explicit semantic partitioning of the documents
- Here the documents are either already organized into semantically meaningful collections

- How to partition a collection of documents to make each collection "well separated" from the others?
  - Well separated means that each query maps to a distinct collection containing the largest number of relevant documents

Inverted Index Partitioning

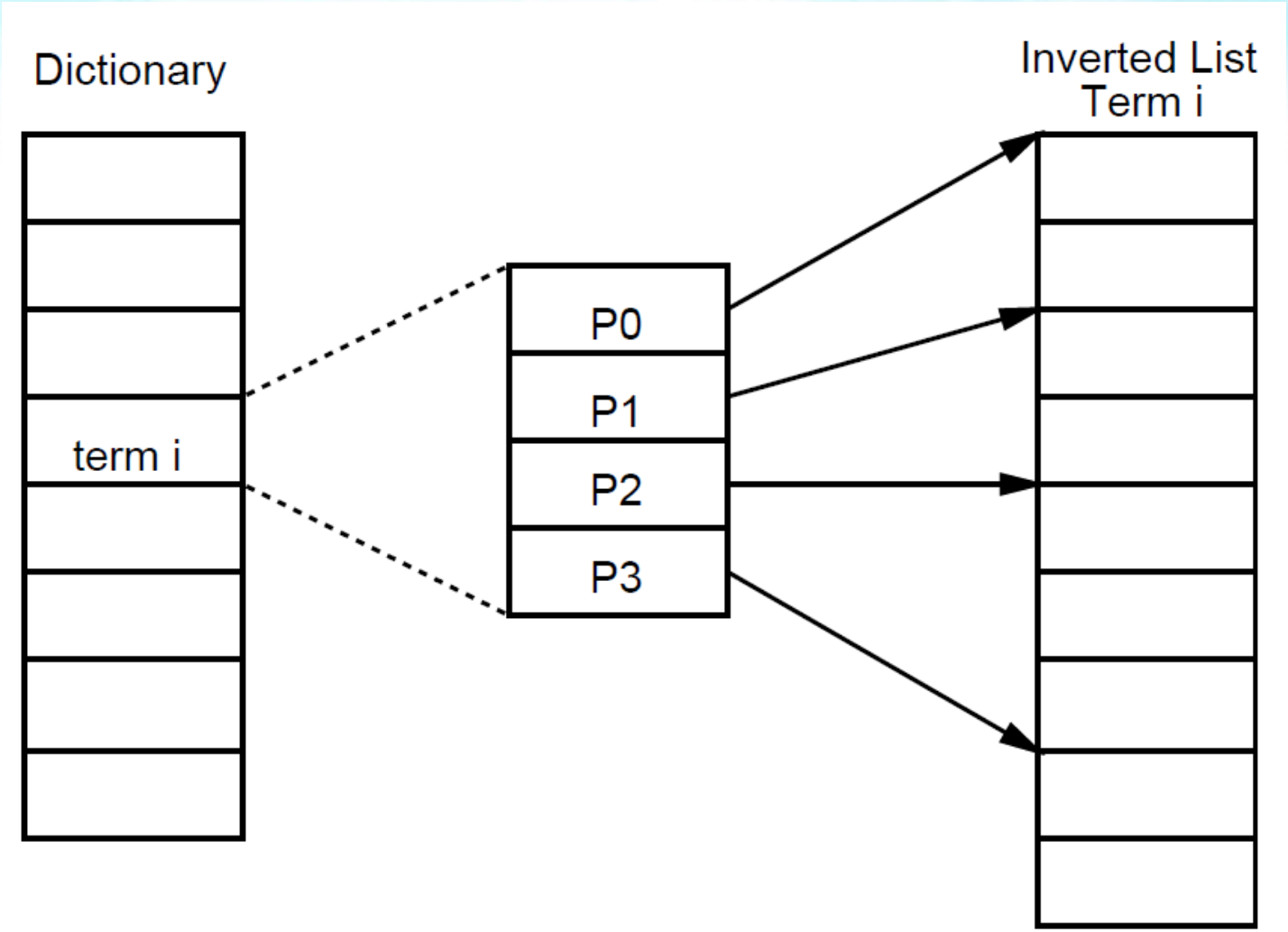
- We first discuss inverted indexes that employ document partitioning, and then we cover term partitioning
- In both cases we address the indexing and the basic query processing phase
- There are two approaches to document partitioning in systems that use inverted indexes
  - **Logical document partitioning**
  - **Physical document partitioning**

Logical Document Partitioning

- In this case, the data partitioning is done logically using the same inverted index as in the original algorithm
- The inverted index is extended to give each processor direct access to their portion of the index
- Each term dictionary entry is extended to include P pointers into the corresponding inverted list
- The j-th pointer indexes the block of document entries in the inverted list associated with the sub-collection in the j-th processor

Logical Document Partitioning

- Extended dictionary entry for document partitioning



Logical Document Partitioning

- When a query is submitted to the system, the broker initiates P parallel processes to evaluate the query
- Each process executes the same document scoring algorithm on its document sub-collection
- The search processes record document scores in a single shared array of document score accumulators
- Then, the broker sorts the array of document score accumulators and produces the final ranking

Logical Document Partitioning

- At inverted index construction time, the indexing process can exploit the parallel processors
- First, the indexer partitions the documents among the processors
- Next, it assigns document identifiers such that all identifiers in partition i are less than those in partition i + 1
- The indexer then runs a separate indexing process on each processor in parallel
- After all of the batches have been generated, a merge step is performed to create the final inverted index

Physical Document Partitioning

- In this second approach, the documents are physically partitioned into separate sub-collections
- Each sub-collection has its own inverted index and the processors share nothing during query evaluation
- When a query is submitted to the system, the broker distributes the query to all of the processors
- Each processor evaluates the query on its portion of the document collection, producing a intermediate hit-list
- The broker then collects the intermediate hit-lists from all processors and merges them into a final hit-list
- The P intermediate hit-lists can be merged efficiently using a binary heap-based priority queue

Physical Document Partitioning

- Each process may require global term statistics in order to produce globally consistent document scores
- There are two basic approaches to collect information on global term statistics
- The first approach is to compute global term statistics at indexing time and store these statistics with each of the sub-collections
- The second approach is to process the queries in two phases
  - 1. Term statistics from each of the processes are combined into global term statistics
  - 2. The broker distributes the query and global term statistics to the search processes

Physical Document Partitioning

- To build the inverted indexes for physically partitioned documents, each processor creates its own index
- In the case of replicated collections, indexing the documents is handled in one of two ways
  - In the first method, each search server separately indexes its replica of the documents
  - In the second method, each server is assigned a mutually exclusive subset of documents to index and the index subsets are replicated across the search servers
  - A merge of the subsets is required at each search server to create the final indexes
- In either case, document updates and deletions must be broadcast to all servers in the system

Comparison

- Logical document partitioning requires less communication than physical document partitioning
  - Thus, it is likely to provide better overall performance
- Physical document partitioning, on the other hand, offers more flexibility
  - E.g., document partitions may be searched individually
- The conversion of an existing IR system into a parallel system is simpler using physical document partitioning
- For either document partitioning scheme, threads provide a convenient programming paradigm for creating the search processes

Term Partitioning

- In term partitioning, the inverted lists are spread across the processors
- Each query is decomposed into items and each item is sent to the corresponding processor
- The processors create hit-lists with partial document scores and return them to the broker
- The broker then combines the hit-lists

Term Partitioning

- The queries can be processed concurrently, as each processor can answer different partial queries
- However, the query load is not necessarily balanced, and then part of the concurrency gains are lost
- Hence, the major goal is to partition the index such that:
  - The number of contacted processors/servers is minimal; and
  - Load is equally spread across all available processors/servers

Parallel IR

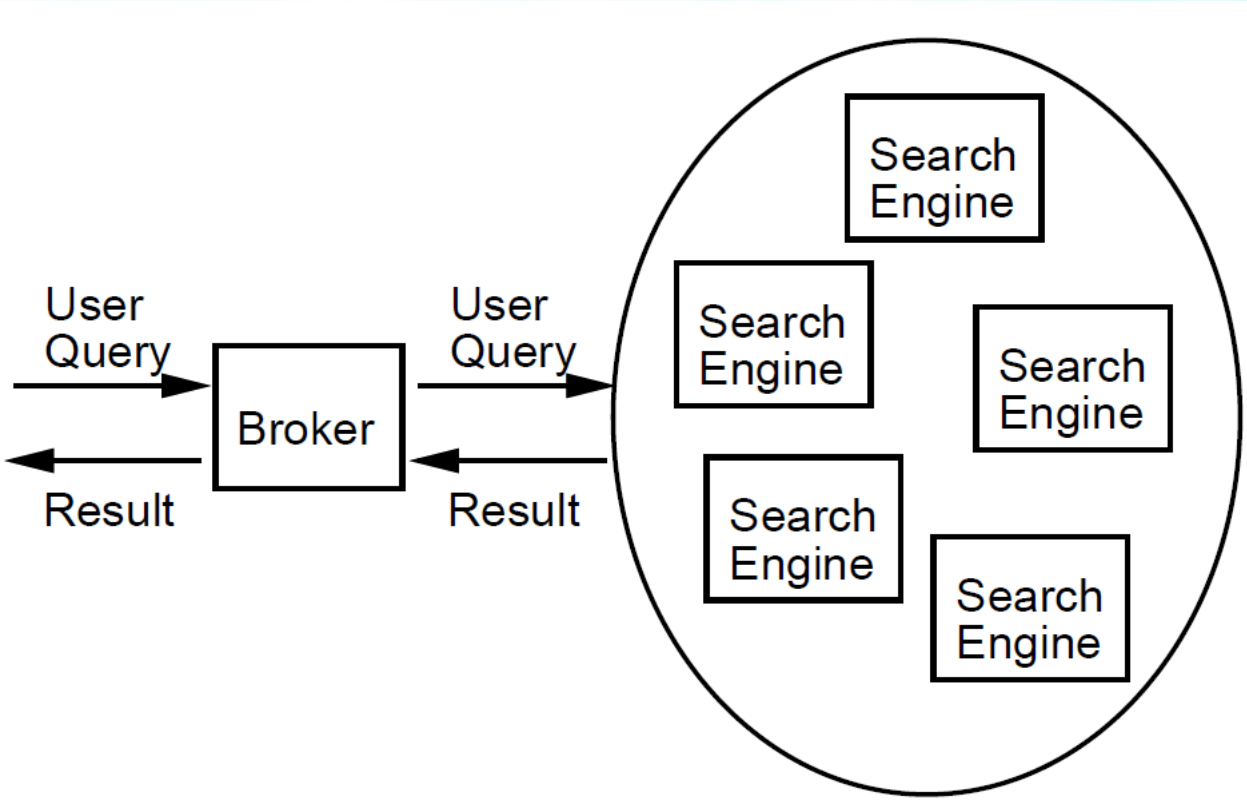
- We can approach the development of parallel IR algorithms from two different directions
- One possibility is to develop new retrieval strategies that directly lend themselves to parallel implementation
  - For example, a text search procedure can be built on top of a neural network
- The other possibility is to adapt existing, well studied IR algorithms to parallel processing
- Parallel computing is the simultaneous application of multiple processors to solve a single problem
- The overall time required to solve the problem can be reduced to the time required by the longest running part

Parallel IR on MIMD Architectures

- MIMD architectures offer a great deal of flexibility in how parallelism is defined and exploited to solve a problem
- The simplest way in which a retrieval system can exploit a MIMD computer is through the use of **multitasking**
- Each of the processors in the parallel computer runs a separate, independent search service
- The submission of user queries to the search services is managed by a broker
- The broker accepts search requests and distributes the requests among the available search services

Parallel IR on MIMD Architectures

- Parallel multitasking on a MIMD machine



Distributed IR

- Distributed computing uses multiple computers connected by a network to solve a single problem
- The ultimate goal of a distributed IR system is to answer queries well and fast in a large document collection

Query Processing in Distributed IR

- In a distributed IR system, it is important to determine which resources to allocate to process a given query
- The pool of available resources comprises components having one of the following roles:
  - Coordinator, cache, or query processor
- A coordinator makes decisions on how to route the queries to different parts of the system
- The query processors hold index or document information
- Cache servers can hold results for the most frequent or popular queries
  - They can reduce query latency and load on servers

Query Processing in Distributed IR

- Instance of a distributed query processing system