Chapter 10

Parallel and Distributed IR

with Eric Brown

Introduction
A Taxonomy of Distributed IR Systems
Data Partitioning
Parallel IR
Cluster-based IR
Distributed IR
Federated Search
Retrieval in Peer-to-Peer Networks

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

A Taxonomy of Distributed IR Systems

A Taxonomy

- Taxonomy of distributed and parallel IR systems

	One Processor	Multiple Processors	
	Same Software	Same Software	Various Software
Internal Communication	Standard Search	Parallel Search SIMD	Parallel Search MIMD
Local Area Communication	n/a	Cluster-based Search	Local Federated Search
Broadband Communication	n/a	Distributed Search (P2P)	Federated Search (P2P)

Data Partitioning

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

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

- Each row represents a document d_j and each column represents an indexing item k_i

Data Partitioning

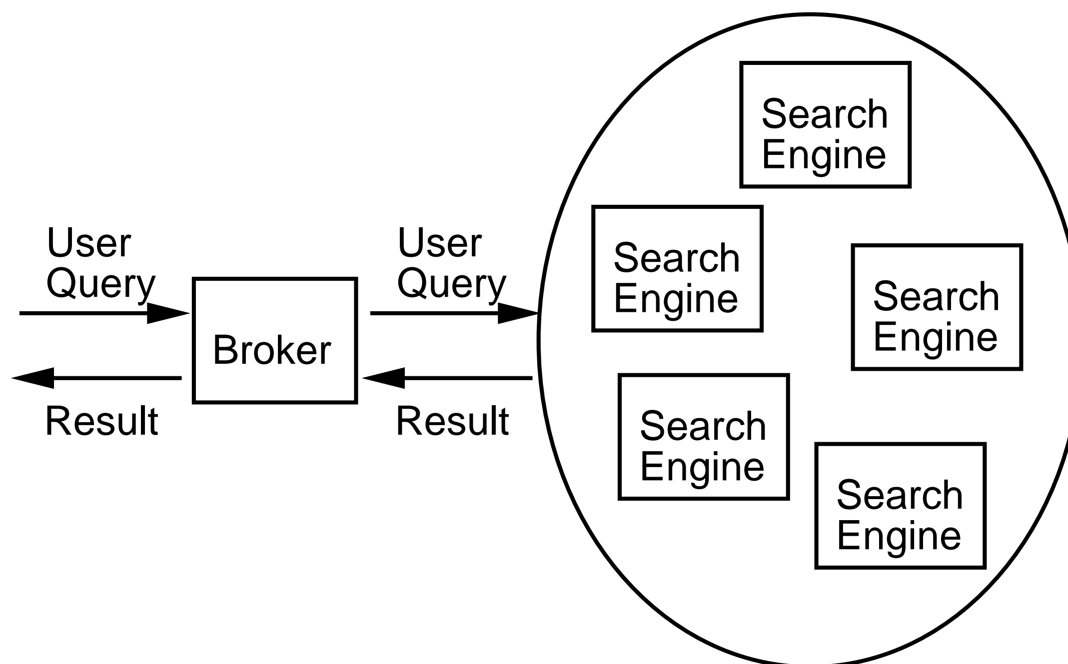
- **Document partitioning** slices the matrix horizontally, dividing the documents among the subtasks
- The N documents in the collection are distributed across the P processors in the system
- During query processing, each parallel process evaluates the query on N/P documents
- The results from each 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

Collection Partitioning

- To construct such a mapping, Puppin *et al* used query logs
- They represent each document with all the queries that return that document as an answer
- This representation enables to build clusters of queries and clusters of documents

Collection Selection

- In many cases, the collections are predetermined and cannot be changed
- In that case, collection selection is the process of determining which of the document collections are most likely to contain relevant documents for each query
- One approach is to always assume that every collection is equally likely to contain relevant documents
- When collections are semantically partitioned, the collections can be ranked according to their likelihood of containing relevant documents

Collection Selection

- The basic technique is to treat each collection as if it were a single large document
- Then, we can evaluate the query against the collections to produce a ranked listing of collections
- Let $w_{c,ij}$ refer to the weight of term k_i in collection C_j:

$$w_{c,ij} = f_{c,ij} \times IDF_{c,i}$$

where

  - $f_{c,ij}$ is the total frequency of occurrence of term k_i in all documents of collection C_j
  - $IDF_{c,i}$ is the inverse collection frequency

Collection Selection

- That is,

$$IDF_{c,i} = \log\left(\frac{N_c}{n_{c,i}}\right)$$

where

  - N_c is the number of collections
  - $n_{c,i}$ is the number of collections in which term k_i occurs

- These weights are then used to assemble query and collection vectors

Collection Selection

- A problem is that may happen that there are no relevant documents within a collection that receives a high relevance score
- Moffat and Zobel avoid this problem
  - They propose to index each collection as a series of blocks, where each block contains B documents
- Voorhees proposes to use training queries to build a content model for the distributed collections
- The GIOSS system ranks collections based on:
  - The number of documents containing a query term, and
  - The total weight of a term over all documents
- The CORI system ranks collections as if they were documents, using the Inquery inference network

Inverted Index Partitioning

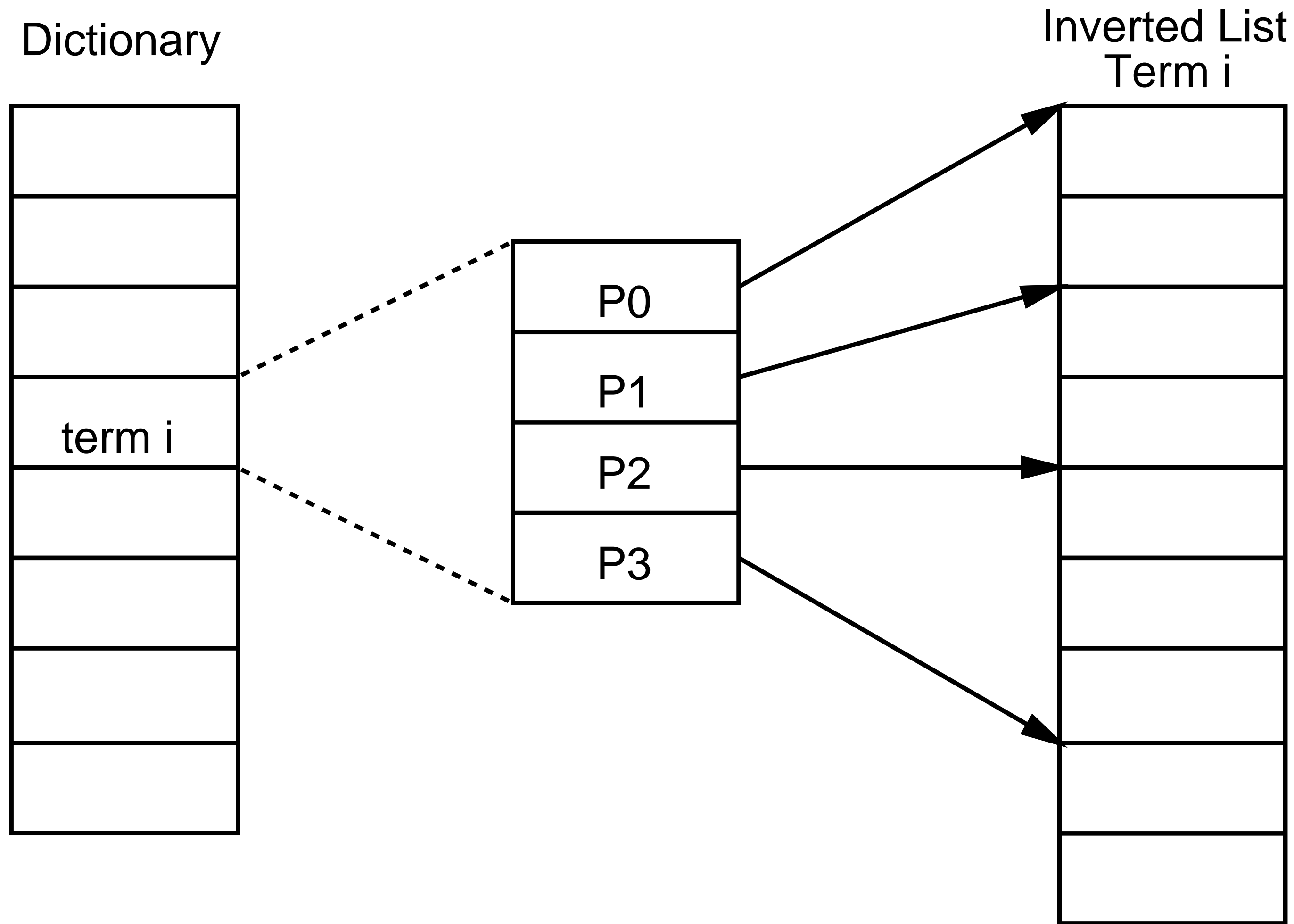
- We first discuss inverted indexes that employ document partitioning, and then we cover term partitioning
- In both cases we address the indexing and the basic query processing phase
- There are two approaches to document partitioning in systems that use inverted indexes
  - Logical document partitioning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

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
- We can use query logs to split the index vocabulary among the processors to achieve the goal above
- A complementary technique is to process the query using a pipeline of processors

Overall Comparison

- Document partitioning affords simpler inverted index construction and maintenance than term partitioning
- Assuming each processor has its own I/O channel and disks, document partitioning performs better
- When terms are uniformly distributed in user queries, term partitioning performs better
- In fact, Webber *et al* show that term partitioning results in lower utilization of resources
- More specifically, it significantly reduces the number of disk accesses and the volume of data exchanged

Overall Comparison

- The major drawback of document partitioned systems:
  - Many not needed operations are carried out to query sub-collections possibly containing few relevant documents
- The main disadvantage of term partitioning:
  - It have to build and maintain the entire global index, which limits its scalability
- In addition, term partitioning has a larger variance regarding answer time and fixing this needs more complicated balancing mechanisms

Suffix Arrays

- We can apply document partitioning to suffix arrays in a straight forward fashion
- As before, the document collection is divided among the P processors and each partition is treated as an independent collection
- The system can then apply the suffix array construction techniques to each of the partitions
- During search, the broker broadcasts the query to all of the search processes
- Then the intermediate results are merged into a final hit-list

Suffix Arrays

- If all of the documents will be kept in a single collection, we can still exploit the parallel processors to reduce indexing time
- In the suffix array construction algorithm for large texts, each of the merges of partial indexes is independent
- Therefore all of the $O((n/M)^2)$ merges may be run in parallel on separate processors
- After all merges are complete, the final index merge may be performed

Suffix Arrays

- In term partitioning for a suffix array, each processor is responsible for a lexicographical interval of the array
- During query processing, the broker distributes the query to the processors that contain the relevant portions of the suffix array and merges the results
- Note that when searching the suffix array, all of the processors require access to the entire text
- However, on a single parallel computer with shared memory, the text may be cached in shared memory

Signature Files

- To implement document partitioning in a system that uses signature files, the documents are divided among the processors as before
- Each processor generates signatures for its document partition
- At query time, the broker generates a signature for the query and distributes it to all of the parallel processors
- Each processor evaluates the query signature locally as if its partition was a separate collection
- Then the results are sent to the broker, which combines them into a final hit-list for the user

Signature Files

- To apply term partitioning, we would have to use a bit-sliced signature file and partition the bit slices across the processors
- The amount of sequential work required severely limits the speedup S available with this organization
- Accordingly, this organization is not recommended

Parallel IR

Parallel Computing

- Processors can be combined in a variety of ways to form parallel architectures
- Flynn has defined a commonly used taxonomy of parallel architectures that includes four classes:
  - SISD: single instruction stream, single data stream;
  - SIMD: single instruction stream, multiple data stream;
  - MISD: multiple instruction stream, single data stream;
  - MIMD: multiple instruction stream, multiple data stream
- The SISD class includes the traditional von Neumann computer running sequential programs,
  - **E.g.**, uniprocessor personal computers

Parallel Computing

- SIMD computers consist of N processors operating on N data streams, and are often computers with:
  - Many relatively simple processors running the same program
  - A communication network between the processors
  - A control unit that supervises the synchronous operation of the processors
- The processors may use shared memory, or each processor may have its own local memory
- Sequential programs require significant modification to make effective use of a SIMD architecture

Parallel Computing

- MISD computers use N processors operating on a single data stream in shared memory
  - MISD architectures are relatively rare and systolic arrays are the best known example
- MIMD is the most general and most popular class of parallel architectures
  - A MIMD computer contains N processors, N instruction streams, and N data streams
  - In this architecture, each processor has its own control unit, processing unit, and local memory

Parallel Computing

- The processors can work on separate, unrelated tasks, or they can cooperate to solve a single task
- **Tightly coupled**: MIMD systems with a high degree of processor interaction
- **Loosely coupled**: systems with a low degree of processor interaction
- MIMD can also characterize **distributed computing** architectures
  - In distributed computing, multiple computers connected by a local or wide area network cooperate to solve a single problem

Performance Measures

- When we employ parallel computing, we usually want to know what is the **performance improvement**
- A number of metrics are available to measure the performance of a parallel algorithm
- One such measure is the **speedup**, defined as:

$$S = \frac{\text{Running time of best available sequential algorithm}}{\text{Running time of parallel algorithm}}$$

- Ideally, when running a parallel algorithm on N processors, we would obtain perfect speedup, or $S = N$

Performance Measures

- In practice, perfect speedup is unattainable either because:
  - the problem cannot be decomposed into N equal independent subtasks
  - the parallel architecture imposes control and communication overheads, or
  - the problem contains an inherently sequential component
- Amdahl's law:

$$S \leq \frac{1}{f + (1 - f)/N} \leq \frac{1}{f}$$

where f is the fraction of the problem that must be computed sequentially

Performance Measures

- Another measure of parallel performance is **efficiency**, given by:

$$\phi = \frac{S}{N}$$

  where S is the speedup and N is the number of processors

- Ideal efficiency occurs when $\phi = 1$ and no processor is ever idle or performs unnecessary work

- As with perfect speedup, ideal efficiency is unattainable in practice

Performance Measures

- Ultimately, the performance improvement of a parallel program over a sequential program is viewed as the combination of:
  - the reduction in real time required to complete the task
  - the additional monetary cost associated with the parallel hardware required to run the parallel program
- This gives the best overall picture of parallel program performance and cost effectiveness

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
- This later approach is considered next

Parallel IR

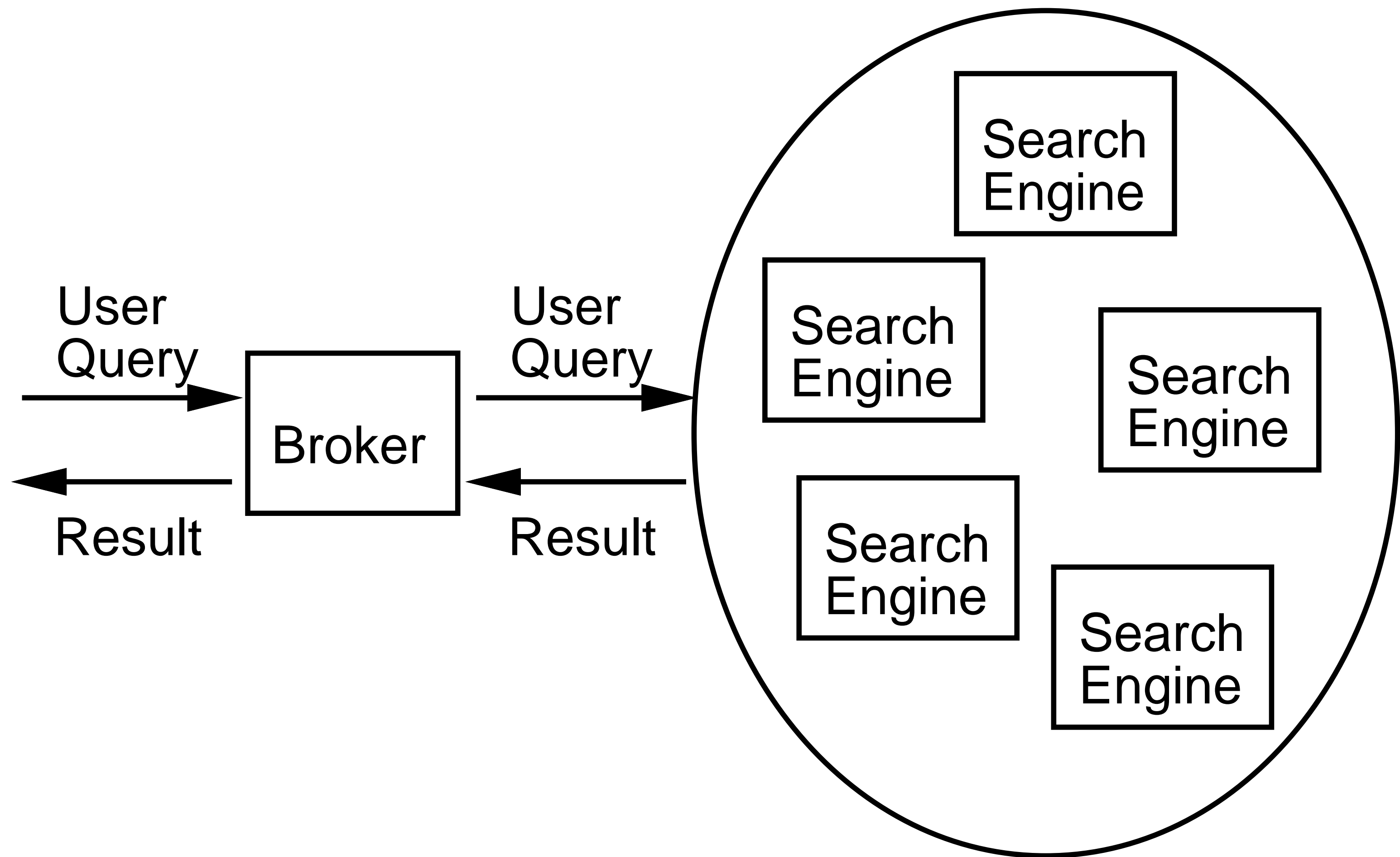
- The modifications required to adapt an existing algorithm depend on the target parallel platform
  - We investigate techniques for applying a number of retrieval algorithms to the MIMD and SIMD architectures
- Parallel computing is the simultaneous application of multiple processors to solve a single 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



Parallel IR on MIMD Architectures

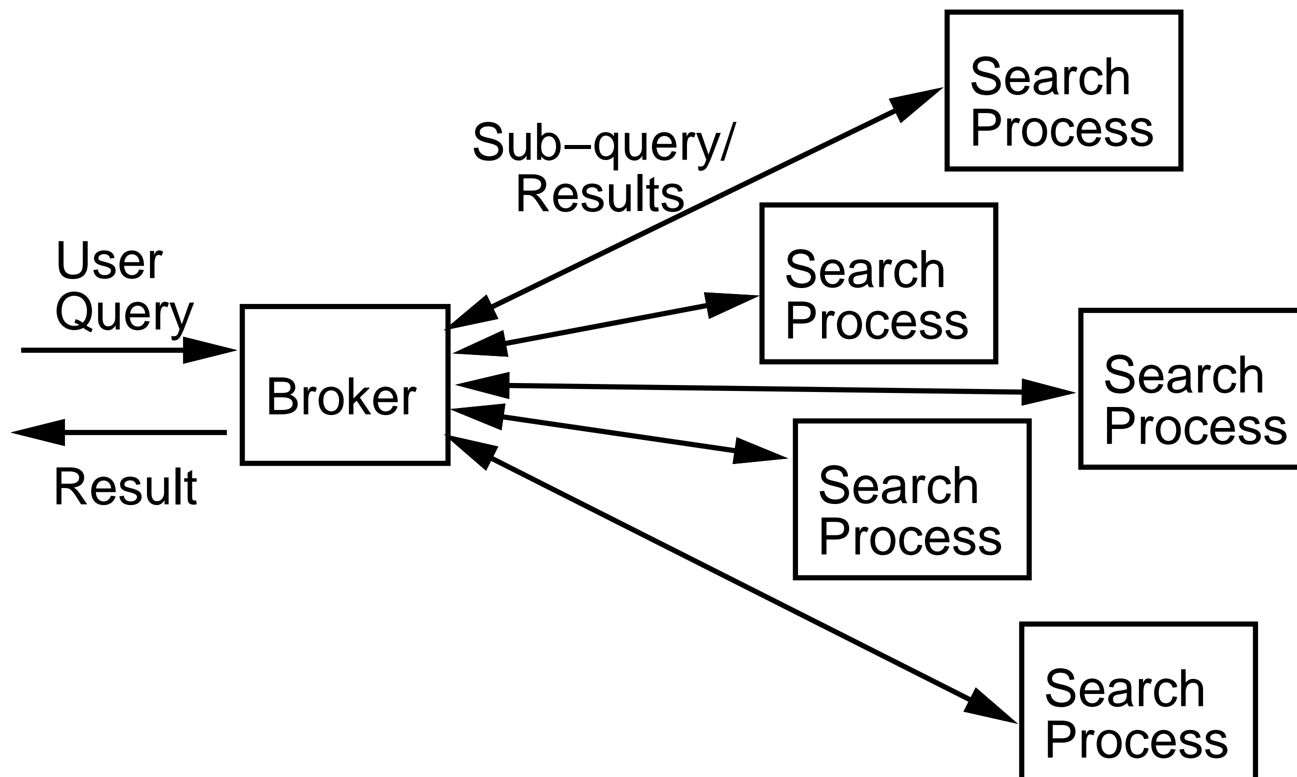
- Care must be taken to properly balance the hardware resources on the system
- Search processes running on the different processors can perform I/O and compete for disk access
- A bottleneck at the disk will be disastrous for performance and could eliminate the throughput gains
- In addition to adding more disks to the computer, the index data must be distributed over the disks
- At one extreme, replicating the entire index on each disk eliminates disk contention at the cost of increased storage requirements and update complexity

Parallel IR on MIMD Architectures

- Alternatively, heavily accessed data can be replicated and less frequently accessed data can be distributed
- Yet another approach is to install a **disk array** and let the operating system handle partitioning the index
- As in sequential systems, **caching** is another important technique that improves performance

Parallel IR on MIMD Architectures

- To improve query response time, the computation required to evaluate a single query must be:
  - partitioned into subtasks
  - distributed among the multiple processors



Parallel IR on SIMD Architectures

- SIMD architectures lend themselves to a more restricted domain of problems than MIMD architectures
- Perhaps the best known SIMD architecture is the Thinking Machines Connection Machine 2 (CM-2)
  - This computer was discontinued during the 1990's
- The CM-2 was used to support both signature file and inverted index based information retrieval algorithms

Inverted Indexes

- Inverted indexes are somewhat awkward to implement on SIMD machines
- Nevertheless, Stanfill have proposed two adaptations of inverted indexes for the CM-2
- In its simplest form, an inverted list contains a **posting** for each document in which a given term appears
- A posting is a tuple of the form (k_i, d_j), where k_i is a term identifier and d_j is a document identifier
- Depending on the retrieval model, postings may additionally contain weights or positional information
- If positional information is stored, then a posting is created for each occurrence of k_i in d_j

Inverted Indexes

- The first parallel inverted index for the CM-2 used the two standard structures: a **postings table** and an **index**
  - The postings table contains the document identifiers from the postings
  - The index (vocabulary) maps terms to their corresponding entries in the postings table

Inverted Indexes

- Parallel inverted index

Postings

beef	2
had	2
home	1
little	0
little	1
little	2
market	0
piggy	0
piggy	1
piggy	2
roast	2
stayed	1
this	0
this	1
this	2
to	0
went	0

Documents

This little piggy went to market.	This little piggy stayed home.	This little piggy had roast beef.

Index

Term	First		Last	
	Row	Pos	Row	Pos
beef	0	0	0	0
had	0	1	0	1
home	0	2	0	2
little	0	3	1	1
market	1	2	1	2
piggy	1	3	2	1
roast	2	2	2	2
stayed	2	3	2	3
this	3	0	3	2
to	3	3	3	3
went	4	0	4	0

Postings Table

2	2	1	0
1	2	0	0
1	2	2	1
0	1	2	0
0			

Inverted Indexes

- At search time these data structures are used to rank documents as follows
- First, the retrieval system loads the postings table onto the back-end processors
- For each query term, an index lookup returns the range of postings table entries that must be processed
- For each row of this range, the processors that contain entries for the current term are activated
- Then, the associated document identifiers are used to update the scores of the corresponding documents
- Document scores are built up in accumulators, which are allocated in a parallel array similar

Inverted Indexes

- The complete algorithm for scoring a weighted term is

```
score_term(P_float Doc_score[], P_posting Posting[],term_t term) {
 int     i, first_pos, last_pos;
 P_int   Doc_row, Doc_pos;
 P_float Weight;
 for (i = term.first_row; i <= term.last_row; i++) {
  first_pos = (i == term.first_row ? term.first_pos : 0);
  last_pos = (i == term.last_row ?
                term.last_pos : N_PROCS - 1);
  where (Position >= first_pos && Position <= last_pos) {
   Doc_row = Posting[i].row;
   Doc_pos = Posting[i].pos;
   Weight = term.weight * Posting[i].weight;
   [Doc_pos]Doc_score[Doc_row] += Weight;
  }
 }
}
```

Inverted Indexes

- It is expensive to send posting weights to accumulators on different processors
- To address this problem, Stanfill proposed the **partitioned postings file**
  - This structure stores the postings and accumulator for a given document on the same processor
  - This proposal eliminates the communication required in the previous algorithm

Inverted Indexes

- The Figure below shows how the postings can be loaded into a table for two processors
- In this Figure, documents 0 and 1 were assigned to processor 0 and document 2 was assigned to processor 1

home	1	beef	2
little	0	had	2
little	1	little	2
market	0	piggy	2
piggy	0	roast	2
piggy	1	this	2
stayed	1		
this	0		
this	1		
to	0		
went	0		

(a)

Inverted Indexes

- Notice that the postings for the term *this* are skewed and no longer span consecutive rows
- To handle this situation, we apply the second trick of the partitioned postings file: segment the postings such that every term in segment i is lexicographically less than or equal to every term in segment $i+1$

home	1	beef	2
little	0	had	2
little	1	little	2
market	0	piggy	2
piggy	0	roast	2
piggy	1		
stayed	1	this	2
this	0		
this	1		
to	0		
went	0		

(b)

Inverted Indexes

- The postings table and index undergo a few more modifications before reaching their final form:

Index

Term	First Partition	Last Partition	Tag
beef	0	0	0
had	0	0	1
home	0	0	2
little	0	0	3
market	1	1	0
piggy	1	1	1
roast	1	1	2
stayed	2	2	0
this	2	2	1
to	3	3	0
went	3	3	1

Postings Table

2	1	0	0
3	0	1	0
3	1	3	0
0	0	1	0
1	0	2	0
1	1		
0	1	1	0
1	0		
1	1		
0	0		
1	0		

Inverted Indexes

- The modified term scoring algorithm is below
  - Here `N_ROWS` is the number of rows per partition

```
ppf_score_term (P_float Doc_score[], P_posting Posting[],term_t term) {
 int     i;
 P_int   Doc_row;
 P_float Weight;
 for (i = term.first_part * N_ROWS;
      i < (term.last_part + 1) * N_ROWS; i++) {
  where (Posting[i].tag == term.tag) {
   Doc_row = Posting[i].row;
   Weight = term.weight * Posting[i].weight;
   Doc_score[Doc_row] += Weight;
  }
 }
}
```

Cluster-based IR

Cluster-based IR

- **Cluster computing** is an intermediate case between parallel and distributed computing
- A **cluster of servers** is a distributed system that has many computers, all physically close and usually connected through a fast local area network
- As local networks become faster, a cluster presents behavior that resembles that of a parallel machine
- One important problem in cluster computing is to balance the workload among the servers
- **Load balancers**: special nodes that balance the load among different machines

Cluster-based IR

- There are many different types of clusters
  - For instance, clusters targeted to high-availability have redundant nodes
- Other clusters are used primarily for computational purposes, as the following two cases:
  - **Beowulf Clusters** are clusters of homogeneous nodes that are run on a dedicated network
  - **Grid Computing** allows the allocation of jobs to computers that perform the work independently of the rest of the cluster

Cluster-based IR

- The same measures that we mentioned in parallel IR can be applied to cluster-based computing
- The equivalent to efficiency is called **load balancing**
- For example, we can measure the fraction of the highest deviation from the average load ℓ:

$$LB = \max_{i=1}^{n} \left(\frac{|load_i - \ell|}{\ell} \right)$$

where $\ell = \ sum_{j=1}^{n} load_j / n$

- Notice that LB can range from:
  - $LB = 0$ (perfect balance) to
  - $LB = n - 1$ (complete imbalance)

Cluster-based IR

- Load balance can be achieved by the combination of several techniques
- The simplest one is to have a special broker, a load balancer, which takes care of the job
- However in some cases that is not possible and specific load balancing algorithms are needed

Cluster-based IR

- To program a cluster, there are middleware software such as:
  - MPI (Message Passing Interface)
  - PVM (Parallel Virtual Machine)
- Another possibility is the map-reduce parallel computing paradigm introduced by Dean *et al*
- It is available as open source software in the Hadoop package
- Current research is focused on extending the power of the map-reduce paradigm

Distributed IR

Distributed Computing

- Distributed systems typically consist of:
  - a set of server processes, each running on a separate node, and
  - a designated broker process
- The broker:
  - accepts and distributes the requests to the servers,
  - collects intermediate results from the servers, and
  - combines the intermediate results into a final result for the client
- The communication between the subtasks is performed using a network protocol such as TCP/IP

Distributed Computing

- Distributed computing uses multiple computers connected by a network to solve a single problem
- A distributed computing system can employ a heterogeneous collection of processors in the system
  - In fact, a single processing node in the distributed system could be a parallel computer in its own right

Distributed Computing

- The cost of inter-processor communication is considerably higher in a distributed computing system
- As such, distributed programs are usually **coarse grained**
  - Granularity refers to the amount of computation relative to the amount of communication performed by the program
  - Coarse grained programs perform large amounts of computation relative to the communication cost
- Of course, an application may use different levels of granularity at different times to solve a given problem

Distributed Computing

- Further, in distributed computing each processor has its own local memory
- On the other hand, a distributed system is also in practice a parallel system
- In a distributed system, we have four elements that are crucial for scalability:
  - **Partitioning** deals with data scalability and, in a large IR system, implies partitioning the document collection and the index
  - **Communication** deals with processing scalability, which in our case is query processing
  - A system is **dependable** if its operation is free of failures
  - The **external factors** are the external constraints on the system

Goals and Key Issues

- Applications that lend themselves well to a distributed implementation usually involve:
  - Computation and data that can be split into coarse grained operations, and
  - Relatively little communication is required between the operations
- Parallel information retrieval based on document partitioning fits this profile well
  - Document partitioning can be used to divide the search task into multiple, self contained subtasks
  - Each subtask involves extensive computation and data processing with little communication among them

Goals and Key Issues

- The ultimate goal of a distributed IR system is to answer queries well and fast in a large document collection
- That implies three different goals that we detail next
  - **Scalability**: the IR system needs to cope with content growth and change
  - **Capacity**: the system must also provide high capacity
  - **Quality**: the system must not compromise quality of answers, as it is easy to output bad answers quickly
- These main goals are shared by all the modules of an IR system
- The goals above are crucial for Web retrieval

Goals and Key Issues

- Main modules of a distributed IR system, and key issues for each module

Module	Communication	Dependability (synchronization)	External factors
Indexing	Reindexing	Partial indexing Updating Merging	Content growth Content change Global statistics
Querying	Replication Caching	Rank aggregation Personalization	Changing user needs User base growth DNS

Dependability

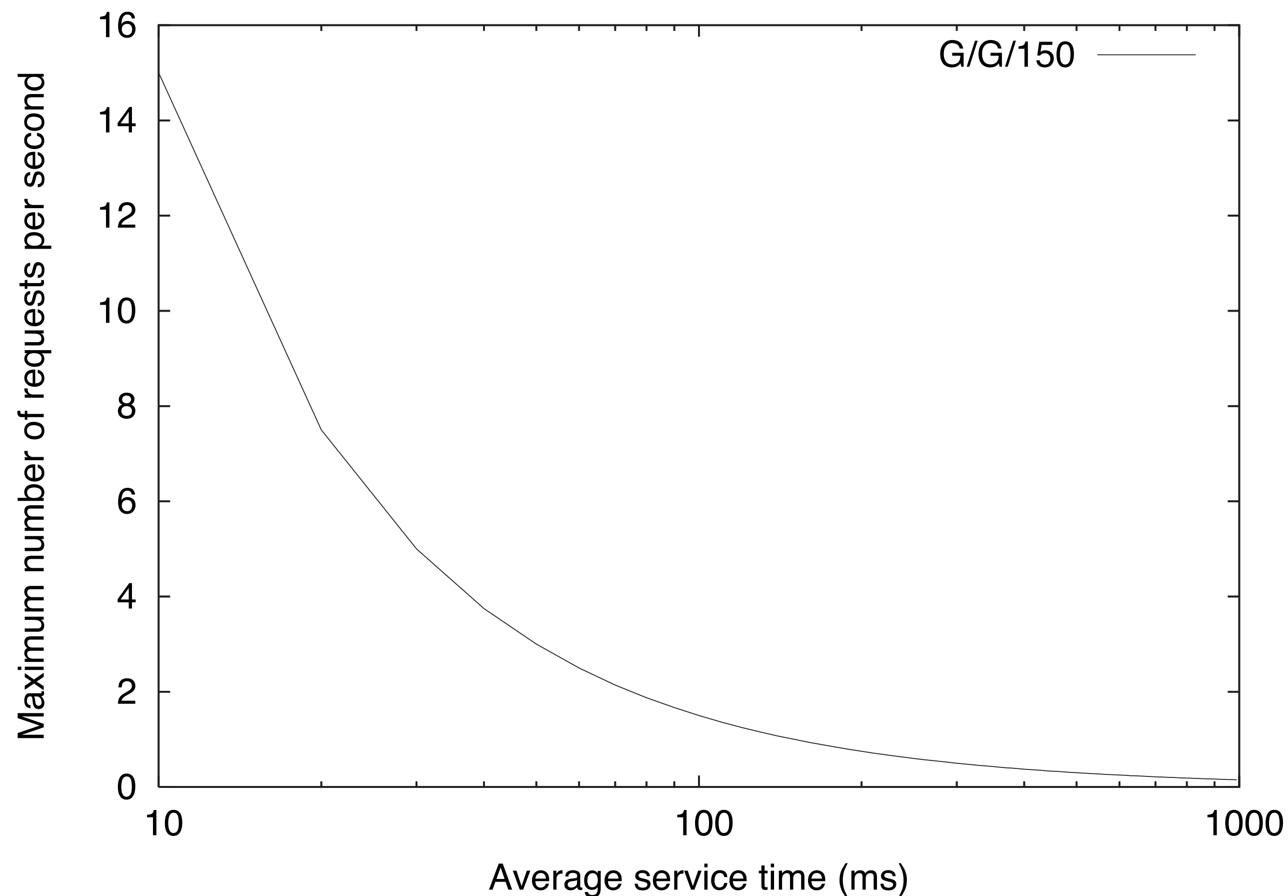
- A classic way of coping with faults is replication
- There are different aspects to replicate: network communication, functionality, and data
- To replicate network communication, we replicate the number of links, making sites multi-homed
- There are two possible levels of replication for functionality and data:
  - In a **single site**, if either functionality or data is not replicated, then a single fault can render a service unavailable
  - Using **multiple sites** increases the likelihood that there is always some server available to perform the request

Communication

- A major drawback that arises from the distributed system is that the servers have to communicate
- Network communication can be a bottleneck as bandwidth is often a scarce resource
- As a simple example, suppose we model a front-end server as a queueing system $G/G/c$
- In this model, the c servers correspond to the threads that serve requests on, for example, a Web server
- The response of each thread to a request depends upon the communication of this thread with other parts of the system
- In this case, bandwidth and message latency contribute to the response time

Communication

- Maximum capacity of a front-end server using a $G/G/150$ model



Indexing

- One way to partition the index across the query processors is to consider the topics of the documents
- Routing the queries according to their topic involves identifying the topics of both documents and queries
- However, topic distribution might have a negative effect on the performance of the distributed retrieval system
- Changes in the topic distribution can result in either:
  - the resources not being exploited to their full extent, or
  - allocation of fewer resources to popular topics
- A possible solution to this challenge is the automatic reconfiguration of the index partition, considering information from the query logs of the IR system

Indexing

- Partitioning the index according to the language of queries is also a suitable approach
- A challenge in routing queries using language is the presence of multilingual documents such as in the Web
  - For example, documents describing technical content can have a number of English terms
- In addition, queries can be multilingual, involving terms in different languages

Indexing

- Building an index in a distributed fashion is a challenging problem
- So far, very few papers suggest approaches to build an inverted index in a distributed fashion
- For example, a possible approach is to organize the servers in a pipeline

Dependability

- The distributed search system depends upon the existence of index structures that enable query resolution
  - For example, if enough index servers fail, then the service as a whole also fails
- Another issue with dependability is the update of the index
  - In some systems it is crucial to have the latest results for queries and content changes very often
  - In this case, it is important that the index data available at a given moment reflects all the changes in a timely fashion

Dependability

- If a server of the system fails, it is impossible to recover the content of that server unless it is replicated
- If this is not the case, then a possible inefficient way to recover is to rebuild the entire index
- Another possibility would be to make the partitions partially overlapping
- Document partitioned systems are more robust with respect to servers failures
- Suppose that a server fails
  - The system might still be able to answer queries possibly without losing too much effectiveness

Communication

- The distributed merge operations of the indexing process can impact the communication among servers
  - A practical approach for achieving this goal is a map-reduce approach
- Indexes are usually rebuilt from scratch after each update of the underlying document collection
- This update operation usually requires locking the index, jeopardizing the whole system performance
- Terms that require frequent updates might be spread across the servers, thus amplifying the lockout effect

External Factors

- In distributed IR systems there are several bottlenecks to deal with
- In a document partitioned IR system is necessary to compute values for some global parameters such as
  - the collection frequency, and
  - the inverse document frequency of a term
- There are two possible approaches:
  - One can compute the final global parameter by aggregating all the local statistics available after the indexing phase
  - The problem of computing global statistics can be moved to the system's broker

External Factors

- To compute such statistics, the broker usually resolves queries using a two-round protocol
  - In the first round the broker requests local statistics from each server
  - In the second, it requests results from each server, piggybacking global statistics onto the second message containing the query
- The question at this point is:
  - Given a *smart* partitioning strategy using local instead of global statistics, what is the impact on the final system effectiveness?

External Factors

- In a real world search engine, in fact, it is difficult to define what is a correct answer for a query
- Thus, it is difficult to understand whether using only local statistics makes a difference
- Furthermore, note that if we make use of a collection selection strategy, using the global statistics is not feasible

Query Processing

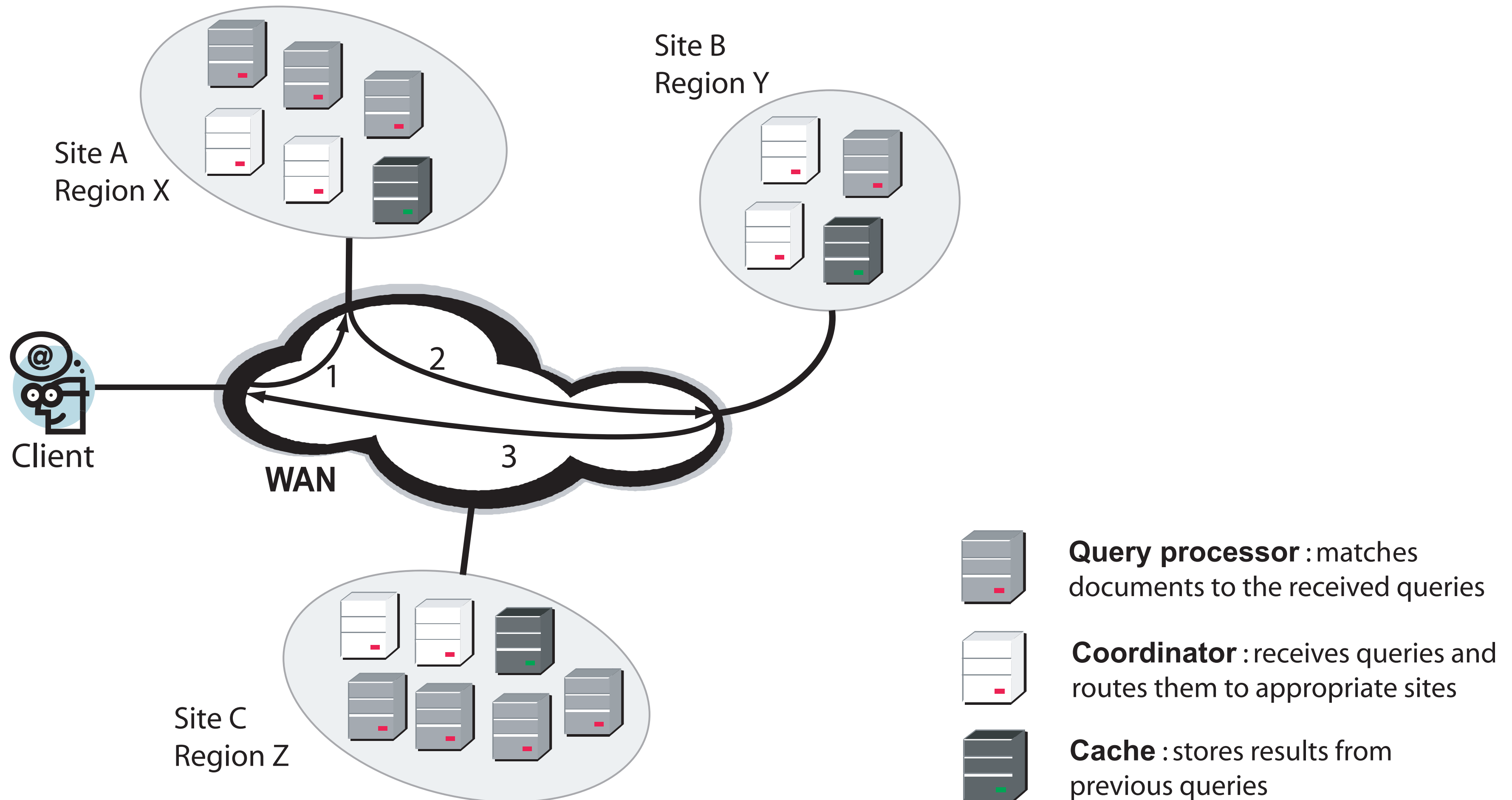
- In a distributed IR system, it is important to determine which resources to allocate to process a given query
- The pool of available resources comprises components having one of the following roles:
  - Coordinator, cache, or query processor
- A coordinator makes decisions on how to route the queries 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

- An important assumption is that one or more servers implement each of these components
  - This assumption is particularly important for large-scale systems
- Designing components in such a way that we can add more physical servers to increase the overall system capacity is fundamental for such large-scale systems
  - In fact, separating parts of the system into component roles is already an attempt to promote scalability as a single monolithic system cannot scale in an unrestricted way
- As these servers can be in different physical locations, we call *site* to each group of collocated servers

Query Processing

- Instance of a distributed query processing system



Query Processing

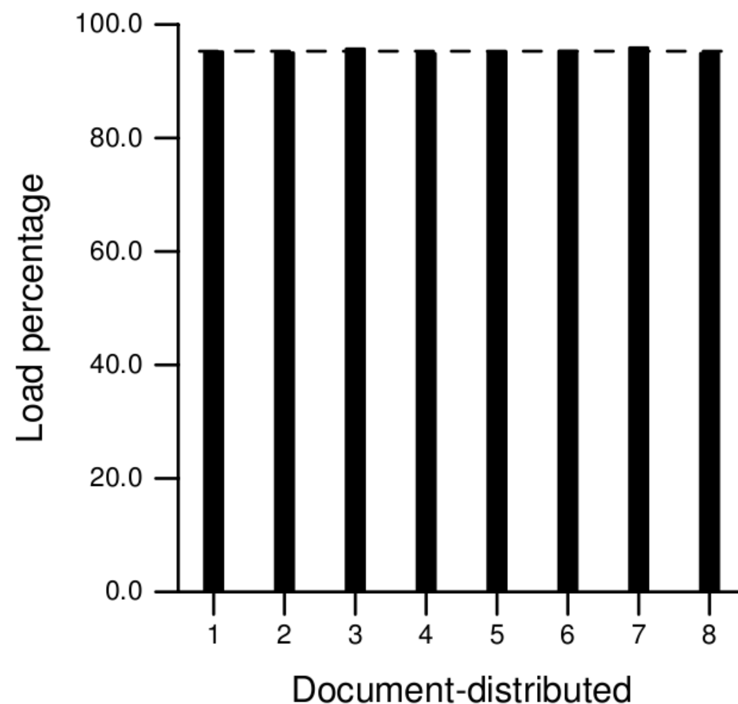
- We classify a distributed query processing system according to four attributes:
  - Number of components
  - Connectivity
  - Distinction of roles
  - Interaction
- The number of components determines the amount of resources available for processing queries
- The choices on the allocation of components change as different choices lead to different performance values
- In fact, minimizing the amount of resources per query is in general an important goal

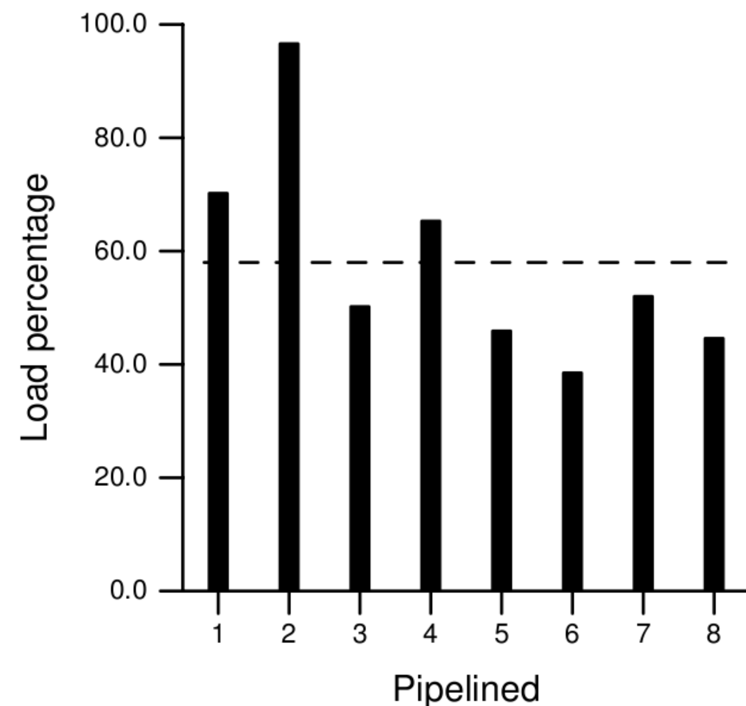
Query Processing

- In practice, partitioning the data potentially enhances the query throughput performance
  - In the case of document partitioning, we could select only a subset of the machines in the search server that ideally contain relevant results
- However, the ability of retrieving the largest number of relevant documents is, as we already discussed, the **collection selection** or **query routing** problem
- In the case of term partitioning, effective collection selection is not a hard problem
  - The solution in this case consists in selecting the server that holds the information on the particular terms of the query

Query Load Balancing

- The major issue for query throughput, in fact, is an uneven distribution of the load across the servers
- The Figure below illustrates the average busy load for each of the 8 servers of a document partitioned system (left) and a pipelined term partitioned system (right)





Query Load Balancing

- In the term partitioned system, there is an evident lack of balance in the distribution on the load of the servers
  - To overcome this issue, one could take into account estimates of the index access patterns to distribute the query load
- In the case of partitioning documents randomly across servers, all servers receive all queries
  - This in principle is a perfect load balance
  - However, the amount of work per server is not necessarily the same, and then a random partitioning does not guarantee an even query load balance

Query Load Balancing

- For a term partitioned system, Moffat *et al* show that it is possible to balance the load
  - Their approach exploit information on the frequencies of terms occurring in the queries and postings list replication
  - They abstract the problem of partitioning the vocabulary in a term partitioned system as a *bin-packing problem*
  - Each bin represents a partition and each term represents an object to put in the bin

Dependability

- Query processors cannot fulfill client requests without the processing capacity and the data they store
- Also, due to the large amount of data they handle, it is challenging to determine good replication schemes
- Having all query processors storing the same data, the system achieves the best availability level possible
- This is likely to impose a significant and unnecessary overhead, also reducing the total storage capacity
- Thus, an open question is how to replicate data with minimal storage overhead

Dependability

- Multiple query processors enable a more dependable system, as well as a more scalable solution
- Availability for caches can also refers to failures of query processors
  - If a query processor is temporarily unavailable, we can serve cached results during the period of the outage

Dependability

- Consistency is also often a very important goal for online systems
- There are techniques from distributed algorithms to implement fault-tolerant services
- The main challenge is to apply such techniques on large-scale systems
- It is also possible to use techniques that enable stale results thus implementing weaker consistency constraints

Dependability

- A system design can consider a caching system as either an alternative or a complement to replication
  - Upon query processor failures, the system returns cached results
- An important question is how to design such a cache system to be effective in coping with failures
- Of course, a good design has also to consider the primary goals of a cache system, which are:
  - reducing the average response time,
  - balance the load on the query processing servers, and
  - good bandwidth utilization

Communication

- Distributed systems need to consider the overheads imposed by the communication of its components
- A term partitioned system using pipelining routes partially resolved queries among servers
- If the index includes the position of terms, the communication overhead between servers increases
- In such a case, the position information needs to be compressed efficiently

Communication

- In the case of a document partitioned system, query processors send the query results to the coordinator
  - The coordinator may become a bottleneck while merging the results from a large number of query processors
- In such a case, it is possible to use a hierarchy of coordinators
- Furthermore, the response time depends on the response time of its slowest component
- This constraint depends on the disk caching mechanism, the amount of memory, and the number of servers

Communication

- When multiple processors participate in the resolution of a query, the communication latency can be significant
- One way to mitigate this problem is to adopt an incremental query processing approach
  - In this approach, the faster query processors provide an initial set of results
  - Other remote query processors provide additional results with a higher latency and users continuously obtain new results
- However, more relevant results may appear later due to latencies

Communication

- Sometimes, the query processing involves adaptation of the search results according to the interests of the user
- Each user profile represents a state, which must be the latest state and be consistent across replicas
- Alternatively, a system can implement personalization as a thin layer on the client-side
- However, this last approach restricts the user to always using the same terminal

Communication

- Since user behavior changes over time, one should be able to update the model accordingly
- A simple approach is to schedule updates of the model at fixed time intervals
- However, a higher update frequency implies a higher network traffic and a lower query processing capacity
- Ideally, the system communication adapts to variations of the underlying model whenever they occur

Communication

- In addition, in a large IR system, there are hundreds of thousands to millions of queries per day
- Logging these actions and using them effectively is challenging because the volume of data is extremely high
- In fact, moving this data from server to server is rarely a possibility due to bandwidth limitations

External Factors

- The design of large IR systems includes users in different ways
- Similarly, the design and analysis of caching policies require information on users, or a user model
- User behavior, however, is an external factor, which cannot be controlled by the IR system
    - For example, the topics the users search for have slowly changed in the past

External Factors

- A change in user behavior can also affect the performance of caching policies
  - Sometimes it is necessary to provide mechanisms that enable automatic reconfiguration of the system
- The challenge would then be to determine online when users change their behavior significantly

Web Issues

- The distributed techniques can be used directly in the Web, as if it were any other large document collection
  - This is the approach currently taken by most of the popular Web search services
- Alternatively, we can spread the work of collecting, organizing, and searching all of the documents
- This is the approach taken by the Harvest system and newer distributed Web search architectures
  - Harvest comprises a number of components for gathering, summarizing, replicating, distributing, and searching documents

Web Issues

- Queries are processed by **brokers**, which collect and refine information from **gatherers** and other brokers
- The information at a particular broker is typically related to a restricted set of topics
- This allows users to direct their queries to the most appropriate brokers
- A central broker registry helps users find the best brokers for their queries

Federated Search

Federated Search

- A federated search system relies on a collection of heterogeneous servers to answer user queries
- The critical engineering issues are basically three:
  - defining the search protocol for transmitting requests and results,
  - designing a server that can efficiently accept a request and initiate a thread to service, and the request
  - designing a broker that can submit asynchronous search requests to multiple servers and combine the results

Federated Search

- The algorithmic issues are also three:
  - how to distribute documents across the distributed search servers
  - how to select which servers should receive a particular query
  - how to process the queries and combine the results from the different servers
- The search protocol specifies:
  - the syntax and semantics of messages transmitted between clients and servers
  - the sequence of messages required to establish a connection and carry out a search operation
  - the underlying transport mechanism for sending messages

Federated Search

- At a minimum, the protocol should allow a client to:
  - obtain information about a search sever, e.g., a list of databases available for searching at the server
  - submit a search request for one or more databases using a well defined query language
  - receive search results in a well defined format
  - retrieve items identified in the search results

Federated Search

- For closed systems, a custom search protocol may be most appropriate
- Alternatively, a standard protocol may be used, allowing the system to interoperate with other search servers:
  - Z39.50 is the standard for client/server information retrieval
  - STARTS, Stanford Proposal for Internet Meta-Searching
- STARTS included features intended to solve the related algorithmic issues, such as merging results from heterogeneous sources

Query Processing

- Query processing in a federated IR system proceeds as follows:
  1. select the collections to search
  2. distribute the query to the selected collections
  3. process the query at each of the distributed collections in parallel
  4. combine the partial results into a final result
- Step 1 may be eliminated if the query is always broadcast to every document collection in the system
- The participating servers evaluates the query on the selected collections using its own local search algorithm

How to merge the results?

- First, if the query is Boolean and the servers return Boolean result sets, all of the sets are simply joined
- If the query involves free-text ranking, a number of techniques are available
- The simplest approach is to combine the ranked hit-lists using round robin interleaving
- An improvement on this process is to merge the hit-lists based on their relevance scores
  - Unless proper global term statistics are used to compute the document scores, we may get incorrect results
- If the distributed document collections are semantically partitioned, then re-ranking must be performed

How to merge the results?

- Callan proposes re-ranking documents by weighting document scores based on their collection similarity computed during the source selection step
- The weight for a collection is computed as

$$w = 1 + \mid C \mid \times \frac{s - \bar{s}}{\bar{s}}$$

where

  - $\mid C \mid$ is the number of collections searched
  - s is the collection's score
  - $\bar{s}$ is the mean of the collection scores

How to merge the results?

- The most accurate technique for merging ranked hit-lists is to use accurate global term statistics
- The broker can include these statistics in the query when it distributes the query to the remote search servers
- If a collection index is unavailable, query distribution can proceed in two rounds of communication:
  - In the first round, the broker distributes the query and gathers collection statistics from each of the search servers
  - Then, these statistics are combined by the broker and distributed back to the search servers

How to merge the results?

- The search protocol can require that search servers return global and per-document query term statistics
- The broker can then re-rank documents using the query term statistics and a ranking algorithm of its choice
- The end result is a hit-list that contain documents in the same order as if all of the documents had been indexed in a single collection

Retrieval in Peer-to-Peer Networks

Retrieval in Peer-to-Peer Networks

- A peer or node is an arbitrary computer which, when connected to the Internet, joins a **peer-to-peer network**, conforming a peer-to-peer (P2P) system
- IR algorithms can take advantage of resources distributed across Internet, in particular **file sharing**
- The first file sharing systems, such as Napster, Gnutella, and Freenet, differed in how the data of the peers was found
  - Napster was the most efficient system using a central index server, but was also the one most vulnerable to attacks
  - Gnutella, on the other hand, used a flooding query model that was inefficient, but highly fault tolerant
  - Freenet used a more efficient heuristic, but did not guarantee that an existing file would be found

Retrieval in Peer-to-Peer Networks

- The solution to this problem was a **distributed hash table** (DHT)
- DHTs are a middleware layer to provide the following characteristics:
  - **Decentralization**: the peers collectively form the system without any central coordination
  - **Scalability**: the system functions efficiently even with millions of peers, as is the case of Internet
  - **Fault tolerance**: the system is as reliable as possible, even with peers continuously joining, leaving, and failing

Retrieval in Peer-to-Peer Networks

- To achieve these goals, one of the peers can coordinate with only a few other peers in the network
  - commonly $\Theta(\log n)$ for a system with currently n peers
- This limits the amount of work needed when a peer joins or leaves the network
- In addition, DHTs must deal with problems such as load balancing, data integrity, and performance

Retrieval in Peer-to-Peer Networks

- DHTs are based in an abstract numerical key space, where the keys identify any resource
- Then, using a partitioning scheme, the ownership of the key space is divided among the participating peers
- An **overlay network** then connects the peers, allowing them to find the owner of any given key in the key space
- The first four DHTs were introduced more or less at the same time:
  - CAN (for content addressable network)
  - Chord
  - Pastry
  - Tapestry

Retrieval in Peer-to-Peer Networks

- The partitioning scheme usually employs some variant of consistent hashing
  - Consistent hashing defines a distance function δ among keys
- Then, a peer identified with the ID (key) i will own all the keys for which i is the closest key under δ
- Consistent hashing has the essential property that removing or adding a peer changes only the set of keys owned by the peers with adjacent IDs

Retrieval in Peer-to-Peer Networks

- The overlay network is based on a routing table where each peer maintains the set of neighbouring peers
- The main property is that each peer owns a particular key k or has a neighbor that is closer to the owner of k
- A simple greedy algorithm forwards messages to the neighbor peer whose ID is closest to k
- This algorithms guarantees finding k in time bounded by the diameter of the overlay network, but it is not necessarily optimal

Retrieval in Peer-to-Peer Networks

- To store a document with given file name, we produce a key k using a hashing function over the file name
- Then we send a message $(k, document)$ to the overall system that will be routed through neighbor peers
- This will be forwarded from peer to peer until it reaches the single peer responsible for the key k
- In this peer, which is specified by the partitioning of the key space, the tuple $(k, document)$ is finally stored

Retrieval in Peer-to-Peer Networks

- To retrieve the document we reverse the process:
  - The peer finds k by hashing the file name and sending a query to find the data associated with k in the network
- The message will again be routed through the overlay network to the peer responsible for k
- Then, the peer will send back directly the data stored associated with that key

Retrieval in Peer-to-Peer Networks

- The main retrieval drawback is that DHTs only support exact-match search, rather than keyword search
- P2P-IR, and in particular full text retrieval, has been investigated for various P2P network organizations
  - Search techniques in unstructured networks are usually based on broadcast, thus suffering from high bandwidth consumption
- Hence approaches based on random walks have been proposed to reduce the traffic in a P2P network

Retrieval in Peer-to-Peer Networks

- **Peer-level** document collection descriptions can be used to identify nodes that can process the query
- These descriptions guide the peer-selection process and the document retrieval from the selected peers
- Resources are ranked by their likelihood to return relevant documents and top-ranked resources are selected

Retrieval in Peer-to-Peer Networks

- The federated search system described by Lu *et al* uses a hierarchical P2P network organization
- Minerva maintains a global index with peer selection statistics
- Minerva∞ is a P2P-IR system that is based on an order preserving DHT
  - It relies on Term Index Networks (TINs) storing the global inverted list of a term on several peers
  - The query is processed by a parallel top-k algorithm involving nodes within TINs and across TINs

Retrieval in Peer-to-Peer Networks

- **Document-level** indexing approaches can potentially deliver higher retrieval quality
- On the other side, such approaches typically distribute the complete index in a structured P2P network
  - Thus, it requires higher index maintenance costs
- This approach faces significant scalability problems caused by the high traffic costs required for intersecting large posting lists
- Thus, a number of solutions have been suggested to resolve this issue

Retrieval in Peer-to-Peer Networks

- Chen *et al* report 73% traffic reduction by applying an optimal Bloom filter for DHT-based full text retrieval
- However, Zhang *et al* shows that single-term indexing is practically not scalable for Web sizes

Retrieval in Peer-to-Peer Networks

- Top-k query processing has been employed to solve the problem of extensive bandwidth consumption
- The main idea is to terminate the processing of a query when the top-k results obtained so far are correct
- Early termination is particularly beneficial for distributed intersections of posting lists
- Top-k query processing algorithms tailored for P2P networks include:
  - Distributed Pruning Protocol (DPP)
  - Three-Phase Uniform Threshold (TPUT) algorithm
  - A family of distributed threshold algorithms (DTA) with Bloom filter optimizations

Retrieval in Peer-to-Peer Networks

- Michel *et al* proposed a family of approximate top-k query processing algorithms called KLEE
  - With small penalties on the top-k result quality, KLEE algorithms significantly reduce bandwidth consumption
- The approach by Thau Loo *et al* suggests to complement index-based query processing with broadcasting
  - The authors suggest using flooding mechanisms to answer popular queries, and resort to indexing only for rare queries

Retrieval in Peer-to-Peer Networks

- A hybrid index partitioning scheme for keyword search is proposed in Shi *et al*
- All peers are clustered in groups and the indexing technique employs term partitioning within the groups
- Each query has to be broadcast to all the groups but only several nodes do the actual processing
- Since the document collection size within a group can be bounded, this solution reduces latency and
- Further, it efficiently distributes the bandwidth consumption

Retrieval in Peer-to-Peer Networks

- Nguyen *et al* suggest an adaptive scheme aiming at balancing the costs between indexing and query processing
- For an individual peer, groups of local documents are created and represented as term sets
- Thus, such a group-level indexing strategy is a generalization of both indexing techniques:
  - peer-level (one group per peer)
  - document-level (one document per group)
- The authors propose a probabilistic model to estimate the cost associated with a given number of groups