WISDOM

Programma di ricerca (cofinanziato dal MIUR, esercizio 2004)
Ricerca Intelligente su Web basata su Ontologie di Dominio
Web Intelligent Search based on DOMain ontologies

Critical analysis of query processing techniques for heterogeneous environments

Ilaria Bartolini, Paolo Ciaccia, Alessandro Linari, Marco Patella

D3.R2 **14 luglio 2005**

Sommario

This deliverable surveys the most important techniques for processing queries in heterogeneous environments. The accent is on those models and solutions that are suitable to be applied in peer-to-peer scenarios. In particular, solutions are reviewed by considering their capability to support one or more of the following query types: key-based, feature-based, schema-based, and relevance-based.

Tema	Tema 3: Elaborazione di interrogazioni
Codice	D3.R2
Data	14 luglio 2005
Tipo di prodotto	Rapporto tecnico
Numero di pagine	44
Unità responsabile	BO
Unità coinvolte	BO
Autori	Ilaria Bartolini, Paolo Ciaccia, Alessandro Linari, Marco Patella
Autore da contattare	Paolo Ciaccia Dipartimento di Elettronica, Informatica e Sistemistica, Università degli Studi di Bologna viale Risorgimento 2, 40136 - Bologna pciaccia@deis.unibo.it

Critical analysis of query processing techniques for heterogeneous environments

Ilaria Bartolini, Paolo Ciaccia, Alessandro Linari, Marco Patella

Abstract

This deliverable surveys the most important techniques for processing queries in heterogeneous environments. The accent is on those models and solutions that are suitable to be applied in peer-to-peer scenarios. In particular, solutions are reviewed by considering their capability to support one or more of the following query types: key-based, feature-based, schema-based, and relevance-based.

1 Introduction

This deliverable presents the state of the art of query processing techniques for heterogeneous environments, with a particular attention to models and solutions recently proposed for the peer-to-peer scenario. In particular, this deliverable will not consider "classical" aspects, such as problems of distributed query processing or mediator-based architectures (for a detailed survey on these aspects we refer the reader to [50]).

The ultimate goal of the WISDOM project is to develop innovative solutions, based on domain ontologies, for efficiently and effectively searching the Web. To this end, the WISDOM view is to have a network of peers among which suitable *semantic mappings* are established.

In order to better understand which can be an effective approach to process queries in such scenario, we conveniently distinguish the following basic types of queries that have been considered in the literature:

Key-Based Search: The user provides the key (identifier) of the object to be retrieved; the result consists of a single object and query processing for such queries involves searching for a peer storing the requested object.

Feature-Based Search: Each object is described by way of a set of features (e.g., keywords) and the user specifies the features that should be included in the resulting objects; the result is a set of objects and processing the query means to retrieve all/some of the objects containing/matching the requested features.

Schema-Based Search: Data have a *schema* and queries are posed against such schema, e.g., using an SQL-like syntax. Since different data sources can present different schemata for their data, the main problem is now the integration of such schemata, possibly using semantic information.

Relevance-Based Search: Each query induces an order on the data in the network, obtained by measuring the relevance of each object to the query. The user can, thus, request for the "best" objects with respect to the query. This is the most complex type of query we consider, since processing such requests might require, besides the integration of data coming from different peers, also the combination of different (*local*) ordering criteria.

In this report we survey:

- solutions based only on the structure of the underlying network (Section 2), that are able to process key-based and, to a limited extent, feature-based queries;
- techniques that take into account the actual content of data sources and/or the semantics of their relative schemata (Section 3), to fully support feature-based and schema-based queries;
- methods for computing the relevance of objects and for integrating data ordered according to different relevance criteria (Section 4).

2 Network-Based Query Processing

In this section, we survey query processing techniques that are based solely on the structure imposed on the underlying network organization. In particular, we distinguish among *unstructured* and *structured* overlay networks: in a structured network, it is always possible to reach a data instance if we know its identifier, whereas this might not be the case with unstructured networks. From the point of view of query processing, structured network provide mechanisms to discover the placement of a data item given the identifier, while unstructured network usually include heuristics to find the requested data item without *flooding* the query to all the peers. In this context, since the goal of query processing algorithms is to find the node(s) containing the requested object (document), they are equivalent to *routing* algorithms, thus the two terms will be interchangeably used in this section.

With respect to the logical organization of peers, both structured and unstructured networks can be *flat*, if peers are on the same level, or *hierarchical*, if the topology consists in more than one level.

2.1 Unstructured Networks

In unstructured networks the placement of a piece of data is completely uncorrelated to its identifier; albeit, from the query processing point of view, this might be a drawback, because we may have to search through the whole network for the node where the document we are searching is located, the replication of commonly-used data, which is usually not allowed in structured networks, can have a beneficial effect, because the search can be interrupted as soon as one copy of the requested document is found. Typically, unstructured networks are characterized by a high dynamicity and self-organization with respect to structured networks, since a node joining or leaving the network has a small impact on the system performance and there is almost no coordination in the way the topology is created. The complexity is slightly increased in hierarchical super-peer networks networks (see Section 2.1.2), but this allows to achieve a usually higher query recall. The advantages over structured networks include:

- in some networks, peers are extremely transient, thus the overhead needed to maintain the overlay structure can be too heavy;
- the frequency of queries is not uniformly distributed (the 80-20 rule is still valid for peer-to-peer systems), and very requested documents are usually well-distributed over the network: routing algorithms for unstructured networks have good performance for well-replicated documents;
- unstructured networks are not based on identifiers, thus they are able to process feature-based queries (see Section 1) without much extra-effort.

2.1.1 Flat Unstructured Networks

In flat networks, all peers have a limited knowledge of the global topology, since they only maintain a list of neighboring peers. Since documents are published on peers without any knowledge of the underlying network topology, the only possible search algorithm is to recursively request the document to neighbor nodes until a result is found. In particular, the neighboring graph can be traversed by the query in a breadth-first (Gnutella) or in a depth-first (Freenet) manner. Maintenance of the neighboring graph is usually very easy, since each peer discover its neighbors in an incremental way as it contacts other peers during the search.

Gnutella The Gnutella (protocol 0.4) network [1], for the simplicity of its communication protocol and for its widespread usage, is the most important unstructured network. When a query originates at a node p, it is forwarded to all the neighbors of p. To avoid flooding the whole network, Gnutella uses Time-To-Live (TTL) to control the maximum number of hops that a query can be propagated to: this, inevitably, generates an horizon which, from the peer's point of view, partitions the network. Moreover, choosing the appropriate TTL value θ might be a difficult problem: if θ is too low, the requested document might not be reached even if it exists somewhere in the network, while if θ is too high, the network is unnecessarily burdened for each query, since the request is forwarded to many nodes. One of the main drawbacks of the Gnutella routing algorithm is that, once the request has been forwarded to neighboring nodes, it cannot be stopped. For example, suppose that the neighboring list of node p include nodes n and n' (Figure 1): when a query q is posed at p, p forwards it to both n and n', which, in turn, will forward it to their neighboring nodes; if a result is found at n, there is nothing we can do to stop the flooding of requests starting from n', thus a lot of work is wasted.

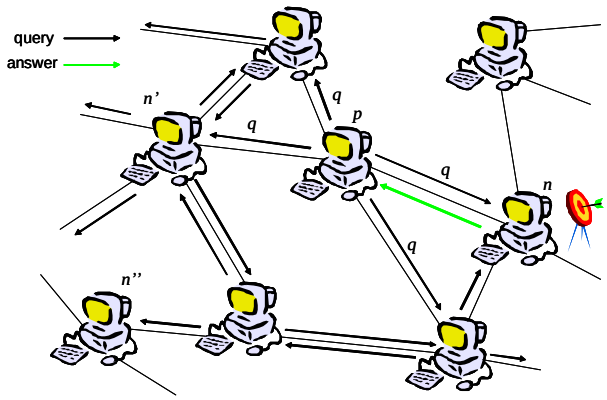


Figure 1: Forwarding the query in Gnutella: the query originated in p and $\theta = 3$, thus node n'' does not forward the query.

Freenet To avoid the high overhead requested by the Gnutella protocol (a peer has to answer all the requests of its θ-neighbors in the network), Freenet [21] uses a depth-first traversal. At every request, only a neighboring node is contacted and the query is forwarded to other nodes until an answer is found or a maximum number of hops θ is reached. In case the maximum length is reached, the search is backtracked and the next neighbor in the list is contacted, until either the requested document is found or the complete neighboring graph has been traversed. Of course, this depth-first traversal of the network lightens the burden on each peer, because not all nodes have to be contacted for every query, but response times might increase exponentially in θ and the query recall is lower with respect to the breadth-first case.

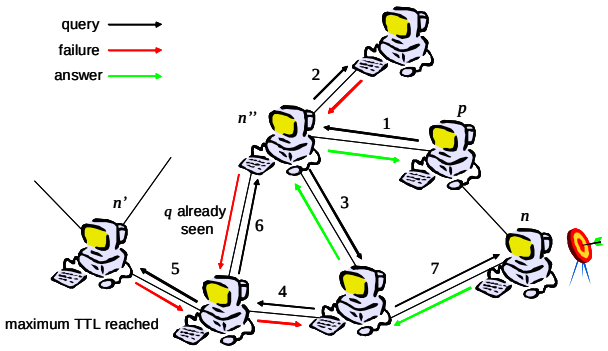


Figure 2: Forwarding the query in Freenet: the requested document is found in node n; the search fails at node n' because the maximum TTL has been reached ($\theta = 4$), and at node n'' because q was already processed by n''.

Expanding Ring The Expanding Ring technique applies to Gnutella-like networks and was presented in [55] and [82], where it is termed *Iterative Deepening*. Basically, it mimics the Gnutella routing algorithm by using an increasing value of TTL until either the requested document is found or the maximum TTL value θ is reached. This, of course, has the goal to improve searching performances, by reducing the number of messages for documents that can be found close to the peer originating the query, i.e., for highly-replicated documents, at the expense of a reduced speed for finding documents in farther nodes. In [55], it is experimentally demonstrated that, even with low values of TTL, it is possible to achieve high recall rates, thus incurring in a limited number of messages between network peers, particularly for common documents. On the other hand, it is also shown that the network topology highly influences performance, with random networks attaining the best results.

Directed Breadth-First Search The Directed Breadth-First Search (DBFS) technique [82] tries to improve over Gnutella flooding by forwarding the query only to a selected number of neighbors, that will then use the simple breadth-first strategy. This requires every peer to maintain, for each neighboring peer, simple statistics, like the number of results received through that neighbor for past queries. This aims at reducing the number of messages while keeping a high quality of the result, since only "good" nodes are contacted. Several heuristics are proposed in [82], like:

- select the neighbors that provided the highest number of results for previous queries;
- select the neighbors returning response messages whose path length is the shortest (thus, nodes that are close to useful data);
- select the neighbors that forwarded the largest number of messages (thus, nodes that are stable);
- select the neighbors having the shortest message queue (thus, nodes that are likely to route our request sooner).

Random Walk The random walk [55] is a well-known technique, which forwards each requests only to a set of m randomly chosen neighbors. The standard random walk ($m = 1$) is able to reduce the number of messages by an order of magnitude with respect to the expanding ring technique, for example, but response times are also increased by an order of magnitude. To reduce the delay, usually $m > 1$ "walkers" are used. Such walkers are forwarded until a positive answer is found or a maximum TTL has expired (see Figure 3). Other techniques for stopping the walkers include checking back with the node that originated the request, to see if a result has

been found [55], or "coloring" the nodes traversed by a query, so that a walker can be stopped when an already colored peer is reached [77]. This results in a lower number of messages, compared to both the flooding and the expanding ring strategies [55].

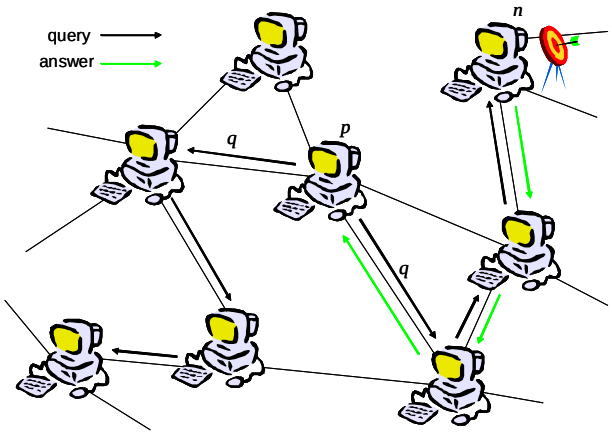


Figure 3: The random walk technique: the requested document is found in node n; 2 "walkers" are used and the maximum walker length is $\theta = 3$.

In [7], an adaptation of the random walk technique is proposed for power-law networks, where the choice of the neighbor(s) to be contacted is biased to prefer high-degree nodes, i.e., peers in high-concentrated areas. This, intuitively, should favor nodes connected to a high number of other peers, and thus of documents, so that the probability of finding an answer increases. However, this technique, as argued in [19], is likely to overload high-degree nodes, thus hindering their capability of quickly finding results for every query.

2.1.2 Hierarchical Unstructured Networks

Hierarchical unstructured networks are usually organized as multi-level structures where each level is an unstructured network. Nodes at level 0 of the hierarchy are regular peers, whereas nodes in upper levels are called *super-peers* [83]. Each super-peer is a node that acts as a server to a subset of client peers: such clients submit queries to their super-peer and receive results from it. Because of this double nature of super-peers, super-peer networks are more properly viewed as an hybrid between peer-to-peer networks and client-server systems.

Each super-peer, together with its associated client nodes, forms a *cluster*, and the maximum size of such clusters is usually a system parameter (when the size equals 1, the network degenerate to a pure peer-to-peer network). Super-peer nodes are connected to each other in a "regular" network and route messages over this "upper-level" overlay network (see Figure 4). In line of principle, the hierarchy can have more than 2 levels, thus we may have super-super-peers, although this is rarely done.

Upon receiving a request from one of its client nodes, a super-peer should decide whether the query can be resolved "within" the cluster or it has to be forwarded to other super-peers. For this purpose, the super-peer should know the content of all the peers in its cluster, by maintaining an index holding sufficient information to answer all possible queries (see Section 3).

The use of super-peers, which are usually fast and stable machines, aims to limiting flooding to the super-peer overlay network and to increase the stability and the computational power of the whole system. This comes at the price of an overhead needed for maintaining the index at each super-peer (which is, however, small in comparison to the saving achieved in query costs). Redundancy is also usually exploited to reduce the possible bottleneck deriving from a failure of a super-peer: if clusters have s super-peers, the network is said s-redundant. Redundancy reduces

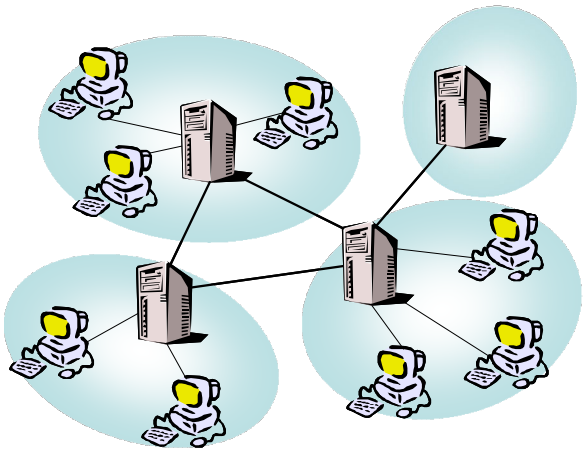


Figure 4: A super-peer network.

the load at each super-peer (queries can be served in a round-robin fashion), but increases the overhead for maintaining the replicated index at each super-peer.

Super-peers are used in Gnutella (protocol 0.6) [1], where the overlay network follows a power-law [7], in KaZaA [3] and iMesh [2], where the FastTrack protocol underlying both systems is still largely unknown, and in Edutella [61], where super-peers are organized in an HyperCuP structured network (see Section 2.2.1), making it a hybrid network [62].

2.2 Structured Overlay Networks

Structured networks are characterized by the placement of data being dependent on their identifier, i.e., the exact location of each data instance is stored by the peer that, according to a particular algorithm, is responsible for it.

2.2.1 Flat Structured Networks

In flat structured networks, peers have all the same role and occupy equivalent positions in the topology. Usually, the search space is partitioned among peers and each peer locally stores a list of neighboring peers, i.e., peers associated to zones of the search space that are close to its zone. The association between document identifiers and peers is usually performed using Distributed Hash Tables (DHTs), i.e., hash tables that are distributed over the peers, and specific algorithms are provided to modify the subdivision of the space over time, when peers join or leave the network. When searching for the peer responsible for the searched data instance, neighboring peers are contacted in an iterative way, until the goal peer is reached: the cost of searching is usually logarithmic in the number of nodes in the network, and is thus very efficient [37]. However, since hashing functions hardly preserve locality, other search paradigms, like searching for a key prefix or for keys similar to a given one, require additional techniques to be supported (see Section 2.3).

Content-Addressable Networks The Content-Addressable Network (CAN) approach [67] considers data keys as points in a d-dimensional Cartesian wrapped coordinate space, i.e., a d torus. Each key is mapped to a point v in the coordinate space by way of a hash function, and the space is partitioned among peers so that each peer stores the location of all documents whose hashed key is located within its corresponding zone. When a document is requested at a peer p, the corresponding point v is obtained by way of the hash function: if v falls within the zone associated to p, then p can immediately fulfill the request; if v is owned by one of the

neighbors of p, the request is routed towards the correct neighbor; if, however, this is not the case, the query should be forwarded through the CAN until the node storing the zone of v is found.

In [67], a simple greedy routing algorithm is proposed for CANs, that follows neighbors having coordinates closest to the coordinates of the zone containing the goal point v (see Figure 5 (a)). It is also demonstrated that, when the space is partitioned into N zones (i.e., N peers are in the CAN), the average routing path length is $(d/4)(N^{1/d})$ and zones store just $2d$ neighbors. To achieve a logarithmic routing length, which is clearly desirable in large networks, one should consider $d = \log N$ dimensions. In [67], it is also argued that more complicated routing algorithm should be adopted if one has to consider crashed neighboring nodes; such algorithms basically mimic those used by unstructured networks, by recurring to flooding limited by a TTL.

Building a CAN involves the splitting of zones of the search space among network peers. In particular, when a peer joins the CAN, four steps have to be performed:

1. the new peer should find a node already in the CAN;
2. using the CAN routing algorithm, a node should be found whose zone has to be split, e.g., by randomly choosing a point v in the d-dimensional space and contacting the peer whose zone contains v;
3. the zone of the so-found peer is split between the two peers (this is done by assuming a predefined ordering of the d dimensions, much as is done for k-d trees [12]);
4. finally, the neighbors of the split node should be informed that the new peer has to be inserted in their list of neighbors; this only affects a number of nodes which is linear in d (see Figure 5 (b)).

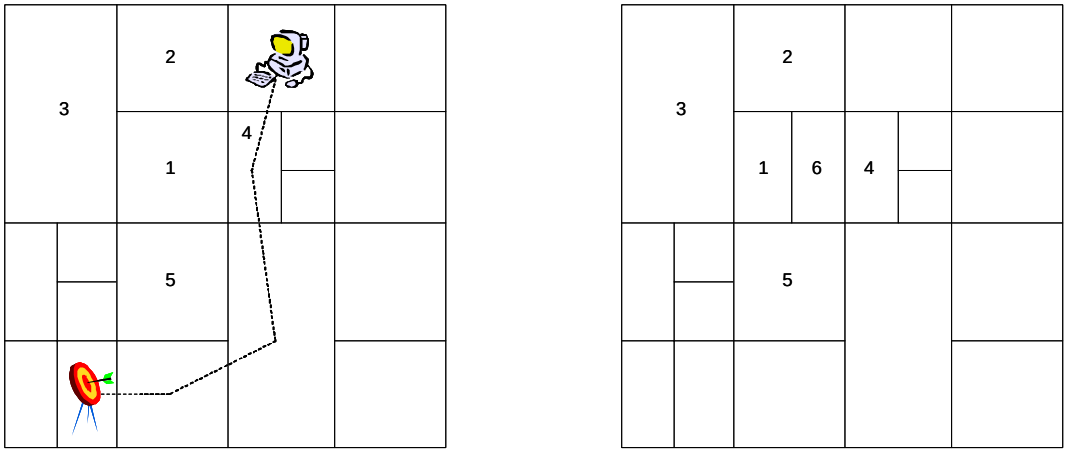


Figure 5: A CAN with $d = 2$: (a) the routing path towards a goal document; (b) splitting zone 1, neighbor lists of nodes 1, 2, 4, and 5 should be updated.

CAN also includes a mechanism for peers to take over the zones of nodes that are leaving the CAN: this is done by choosing one of the neighbors of the leaving peer and should also include a protocol for discovering the failure of a peer or its unexpected leaving/crash.

Chord The Chord [74] protocol also uses hash functions to map keys to nodes in the network. Each node is given an integer value, called the node identifier, and keys are mapped by way of a *consistent hashing* to node identifiers. In particular, a key v is assigned to the first node whose identifier is not lower than v (this is called the successor of v). The space of identifiers is

wrapped, so that all nodes are logically arranged in a ring of 2^m elements, where each connected peer is mapped to an element and the successor of v can be found as the first node encountered by visiting the ring clockwise and starting at v. The value of m is chosen so that 2^m is higher than the maximum number of hashing values, V.

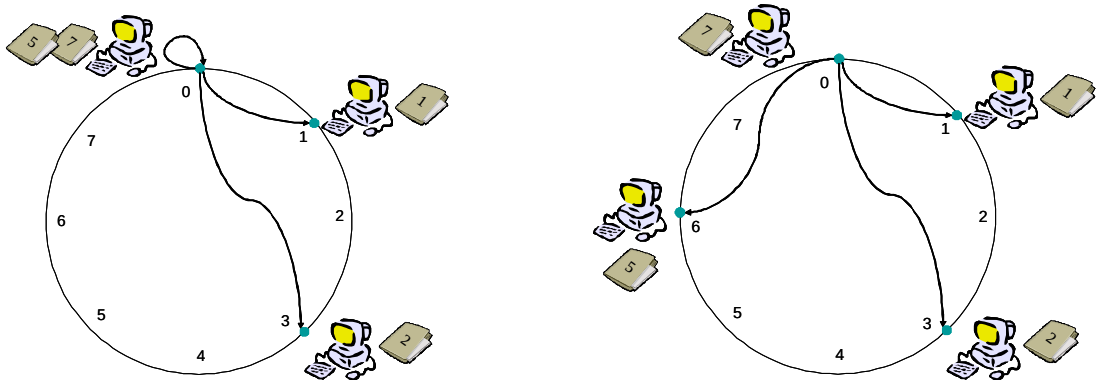


Figure 6: A Chord network with $m = 3$, $N = 3$ nodes and 4 documents (a), the routing table of node 0 is shown; the same network when a new node, with identifier $n = 6$, is added (b).

To find the node responsible for the hashed value of v, it is sufficient to maintain the successor of each node, in order to be able to discover the successor of v: in the worst case, however, this requires traversing all the nodes of the network. To avoid this linear search complexity, in Chord it is proposed to keep additional routing information. In particular, a *finger table* of size m is stored at each node such that the i-th entry in the table succeeds the node by at least 2^{i-1} steps. Formally, if n is the node identifier, then the i-th entry in the table, s_i, is computed as $s_i = successor((n + 2^{i-1}) \mod 2^m)$, $i \in [1, m]$. The basic rationale is that the distance between n and s_{i+1} doubles with respect to the distance between n and s_i. In order to find the key v, node n should find the first node, in its finger table, whose identifier is not higher than v and route the request to it. In this way, the search for the successor of v halves, at each step, the search space, leading to a complexity which is logarithmic in the number of network nodes, with high probability. For example, in Figure 6 (a), we have $m = 3$, thus the network can have a maximum of $2^3 = 8$ documents and nodes. Since only nodes 0, 1, and 3 are present, the finger tables are as in Figure 7.

$n = 0$

i	$n + 2^{i-1}$	s_i
1	1	1
2	2	3
3	4	0

$n = 1$

i	$n + 2^{i-1}$	s_i
1	2	3
2	3	3
3	5	0

$n = 3$

i	$n + 2^{i-1}$	s_i
1	4	0
2	5	0
3	7	0

Figure 7: Routing tables of nodes in Figure 6 (a).

When a node joins the network, the finger tables should be updated to ensure correctness of successors of all nodes and keys. In particular:

1. the finger table of the new node has to be initialized;
2. the finger tables of all existing nodes are (possibly) updated to reflect the addition of the new node;
3. the new node n should receive, from its successor in the network, the keys that n is now responsible for.

Referring to Figure 6 (b), we obtain the routing tables shown in Figure 8 (modified values with respect to Figure 7 are highlighted in bold).

$n = 0$

i	$n + 2^{i-1}$	s_i
1	1	1
2	2	3
3	4	**6**

$n = 1$

i	$n + 2^{i-1}$	s_i
1	2	3
2	3	3
3	5	**6**

$n = 3$

i	$n + 2^{i-1}$	s_i
1	4	**6**
2	5	**6**
3	7	0

$n = 6$

i	$n + 2^{i-1}$	s_i
1	7	**0**
2	0	**0**
3	3	**3**

Figure 8: Routing tables of nodes in Figure 6 (b).

In [74] it is demonstrated that this only require $O(\log^2 N)$ messages, where N is the number of nodes in the network, that each node is responsible for at most $O(\frac{V \log N}{N})$ keys, and that $O(V/N)$ keys are transferred when a node joins or leaves the network.

Pastry In the Pastry protocol [69], each network peer is associated to a unique numeric identifier and the protocol is able to find, in logarithmic time, the node having the identifier closest to a user-provided key. This allows, for example, to associate a hashed key to each document and to replicate the document in the n Pastry peers having a node identifier which is numerically closest to the document key, as in PAST [28].

To allow logarithmic search, each peer maintains a routing table consisting in $\log_{2^b} N$ rows with $2^b - 1$ entries each, where N is the number of nodes in the network and b is a configuration parameter. Each entry in the routing table of peer p contains the address of a node whose identifier shares with the p's identifier a prefix. In particular, the $2^b - 1$ entries at row i of the table, share with the identifier of p a prefix of length i, but differ in the $(i + 1)$-th digit. Among all the nodes in the network having such property, usually the closest nodes are chosen, so as to preserve locality during the search.

To increase the locality of the search algorithm, a leaf set is also maintained besides the routing table, containing the 2^b network nodes having identifiers which are numerically closest to the identifier of the current node. In this way, when a key v is requested at peer p, it is first searched within the leaf set and, possibly, forwarded to the closest peer therein. In case v is not covered by the leaf set, then the routing table should be used and the request is routed towards the node that shares a common prefix with v which is longer than the prefix that p and v have in common. In case such node is not found (or it has gone off-line), then a node is chosen such that the common prefix with v has the same length as p, but is numerically close to v (such node is always included in the leaf set). The complexity of this search is $O(\log_{2^b} N)$ with high probability, thus the choice of the value of b has to trade off between the complexity of the routing algorithm and the size of the routing table, which is $\log_{2^b} N \times (2^b - 1)$.

The self-organization of the routing tables is needed when nodes join/leave the network. In the case of a node joining Pastry, the new node is assigned an identifier v and it should locate, using the above-described routing algorithm, the node n having the identifier closest to v, by starting at a nearby node p. All the necessary information are then obtained by the nodes encountered along the route between p and n, and the overall cost, in terms of exchanged messages, is $O(\log_{2^b} N)$. When a node p leaves the network, all the nodes referencing p should update their leaf set or their routing table (possibly both): this is done by requesting the necessary information from a node in the leaf set or in the appropriate row of the routing table, respectively.

Kademlia The Kademlia [57] protocol is very similar to the one used in Pastry: there are $\log_{2^b} N$ rows in the routing table of each peer, and there are up to $2^b - 1$ nodes in each row. However, differently from Pastry, here nodes at level i in the routing table of peer p are within a XOR distance of $[2^i, 2^{i+1} - 1]$ from a. This allows the same routing algorithms of Pastry to

NodeId: 10233102			
Routing table			
02212102	**1**	**2**2301203	**3**1203203
0	*1***1**301233	*1***2**230203	*1***3**021022
*10***0**31203	*10***1**32102	**2**	*10***3**23302
*102***0**0230	*102***1**1302	*102***2**2302	**3**
*1023***0**322	*1023***1**123	*1023***2**301	**3**
*10233***0**02	**1**	*10233***2**31	
0		*102331***2**0	
		2	
Leaf set			
10233033	10233021	10233120	10233230

Figure 9: The routing table of a Pastry node with ID=10233102, $b = 2$. The IDs of nodes in the routing table have been highlighted to emphasize the common prefix (in italics) and the first different digit (in bold); single digits in each row represent the corresponding value of the node ID.

be used here, but when peer failures occur during the search, Kademlia is more efficient since the XOR distance to the destination can still be reduced, e.g., by changing a lower order bit. Multiple paths towards the destination can therefore exist and such paths have different length.

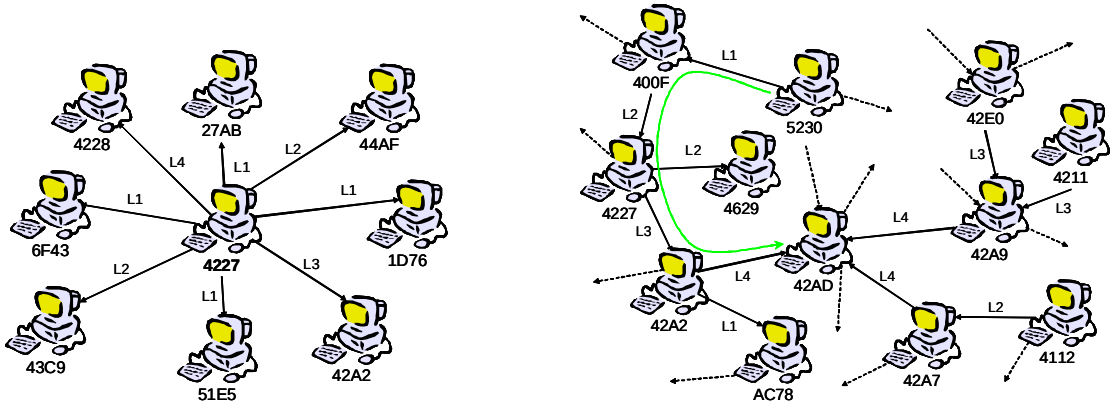
Tapestry In the Tapestry infrastructure [85], documents and peers are assigned identifiers drawn from the same space. Each document, with identifier v, is assigned to a unique on-line node, which is called the root of v: if a node exist whose identifier equals v, then this is the root node, otherwise the node whose identifier is numerically close to v is chosen. The root node of v is guaranteed to know the exact location of the document having key v. The routing mesh of Tapestry is a neighboring graph that allows nodes to locate the root node of a given document key v. The neighbor map of a peer p is a multi-level list of neighboring nodes: at the i-th level of the list we find peers whose identifiers share with p's identifier a prefix of length $i-1$; the j-th element of such list, if present, is the peer closest to p having the digit j in position i, to improve locality of the search algorithm (see Figure 10 (a)).

To search for a given key v, Tapestry looks for the root node of v. This equals to search, in the current peer p, for the peer sharing the longest prefix with v and to route the request towards such node. Eventually, the root node of v is reached, and the location of v is found in less than $\log_b N$ steps, where b is the number of different digits composing identifiers (see Figure 10 (b)). When a digit cannot be exactly matched within a peer multi-level list, the peer looks for the "closest" digit: this is called *surrogate* routing and still guarantees logarithmic search times.

Location of documents in Tapestry is facilitated by the publication algorithm, used by the server sharing a document with key v to find the root node of v (see Figure 11). All the nodes along the path between the server and the root of v store the address of the server so that subsequent requests for v can stop *before* the root of v is actually reached. For example, suppose that the peer s publish a document with key v, that peer n is on the route between s and the root of v, and that peer p requests for such a document. If, on the route between p and the root of v, the peer n is contacted, it can immediately route peer p to peer s, without the need of contacting the root of v. Since the elements in the multi-level lists are close to each other, this highly improve locality, because the closer a peer is to the peer sharing a document, the sooner it will likely find a peer in the document publishing path.

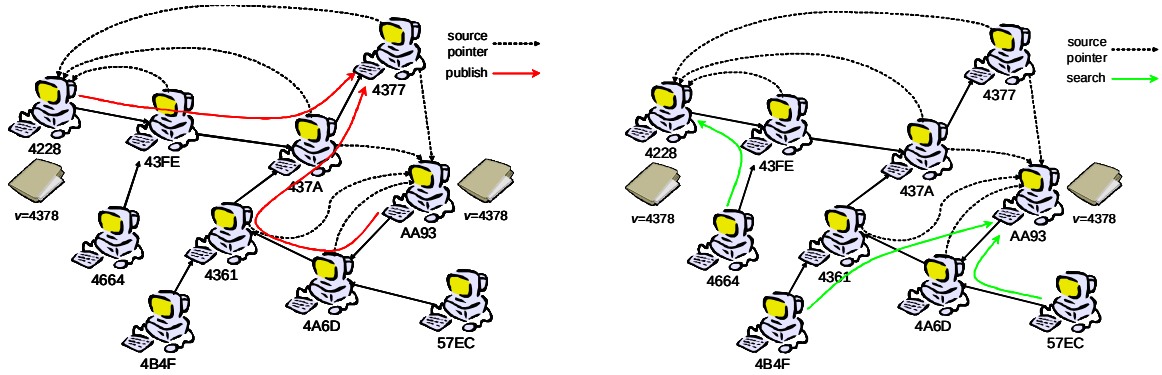
Building neighboring lists when a node joins the Tapestry consists of four steps:

1. the new node is assigned an identifier v and it contacts the root node of v; such node



(a) (b)

Figure 10: Tapestry routing mesh for node 4227 (a). The routing path for a message between node 5230 and node 42AD (b).



(a) (b)

Figure 11: Publishing/searching a document in Tapestry: two copies of a document with key v =4378 are published to the root node 4377 (a); requests for the same documents are routed towards the root document, and are redirected to the source when they intersect the publish path to node 4377 (b).

then contacts all those nodes sharing a prefix with v, and the new node is added to their neighboring lists;

2. nodes contacted in the previous step send references to documents rooted in the new node, to maintain their availability;

3. the new node builds its neighboring list by using contacted nodes and performing an iterative nearest neighbor search;

4. nodes contacted during the nearest neighbor search of the new node might insert it in their neighbor lists where appropriate.

Finally, backpointers and redundancy in neighboring lists are used to deal with voluntary/involuntary node deletion.

HyperCuP HyperCuP [71] builds and maintains network nodes into a hypercube topology. In particular, a hypercube graph of base b and d dimensions contains at most b^d nodes and each peer has $(b-1) \cdot d$ neighbors, $b-1$ for each dimension (see Figure 12 (a) for an example). The maximum distance between two peers in such a network is $\log_b N \leq d$ and, since connections are

redundant, i.e., multiple paths exist between nodes, its connectivity is optimal, i.e., the minimum number of nodes to remove in order to obtain a partitioned graph is maximal. Of course, because of the great symmetry of this structure, constructing and maintaining a hypercube network are both difficult tasks, even if their complexity is still $\log_b N$.

Searching in a hypercube network relies on the broadcasting of messages. Because of the highly-structured topology of the network, however, only $N-1$ messages are requested to reach all the peers and the maximum path length is $\log_b N$ (see Figure 12 (b)). Thus, even broadcasting messages represents an appealing solution, whereas this is not the case with unstructured networks (see Section 2.1). To reduce the complexity, however, searching in a hypercube might limit the path length of forwarded messages using a TTL threshold θ.

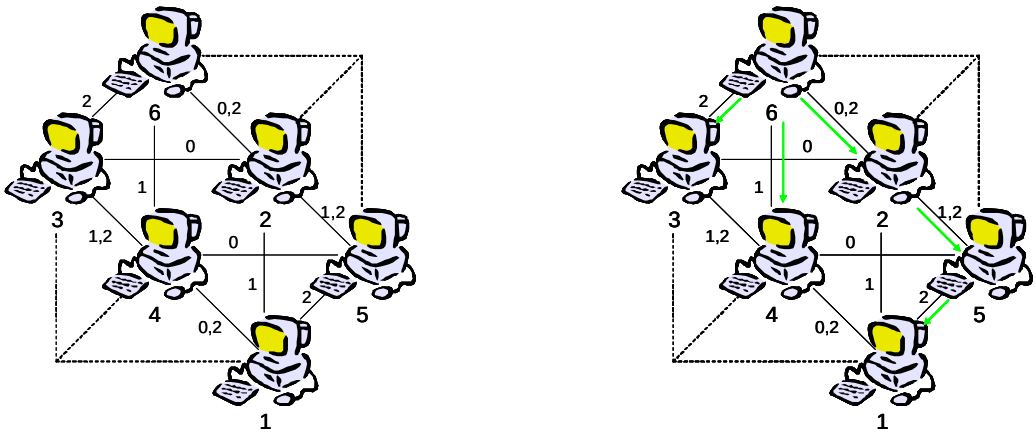


Figure 12: An HyperCup network with base $b = 1$, $d = 3$ dimensions, and $N = 6$ nodes (a), each link is lists the involved hypercube dimensions; broadcasting a message in HyperCup, only $N-1$ messages are required (b).

The structure of HyperCuP is logically equivalent to the one used by Pastry. However, in HyperCuP a new peer receives its identifier, i.e., its position within the hypercube, by using a handshaking protocol with other peers and not by recurring to a "global" hashing function. This slightly increase the complexity of joining the network, but provides a simpler method for increasing the address space, since a new hypercube dimension can be always added (compare this with the expansion of the hashing function should the number of peers exceed the addressable space).

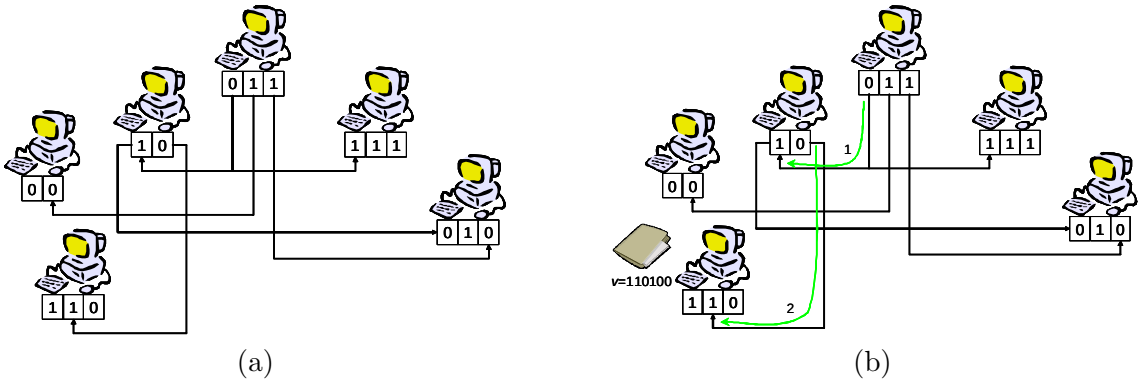
2.2.2 Hierarchical Structured Networks

A hierarchical network is organized in two or more layers, with higher peers having the responsibility for a partition of peers from the lower level. Different levels of the network partition the same data set with a different granularity. This is mainly intended to add flexibility to the search mechanism, using an indexing scheme typical of centralized environments.

P-Grid In P-Grid [4], a dynamic and unbalanced tree is maintained in a distributed way, such that the search space, consisting of binary strings, is partitioned among peers. In particular, each peer n corresponds to a *path*, $p(n)$, and stores the exact location of all documents whose key string starts with $p(n)$, i.e., $p(n)$ is a prefix of all the documents whose position is stored in the peer.

In order to retrieve documents whose prefix differs form its path, each peer also maintains a multi-level grid of references to other peers. For each peer n, the peers referenced by n at level i complete the search space of n with respect to the i-th bit of the path of n. This means that a

peer n' is included in the references of n at level i only if the paths $p(n)$ and $p(n')$ are equal for the first $i-1$ bits and differ in the i-th bit (see Figure 13 (a) for an example). In this way, if a query q is issued at n requesting for a string starting with $p(n)$, then n can process the request locally. Otherwise, the position i of the first bit where q and $p(n)$ differ is computed and one of the peers n' included in the n's references at level i is contacted: n' could solve the query locally (if $q = p(n')$) or re-route it to another peer at a level $i' > i$ (see Figure 13 (b)); this guarantees that the processing of q always terminates, since the length of the common prefix always increases.



(a) (b)

Figure 13: A sample P-Grid, where the multi-level grids of two peers are highlighted (a); searching in the P-Grid for the document with key $v = 110100$ (b).

The construction of a P-Grid is performed independently by the peers during their meetings with other peers, e.g., when one of the peers is providing the answer to the query originated in the other peer. In details, two peers can:

1. merge their lists of references at level i, if their paths agree up to the i-th bit;
2. split their paths, if they have an identical path whose length is lower than the maximum allowed length;
3. split the path of one peer (up to the maximum path length), if its path is a prefix of the other peer's path;
4. share their lists of references at level $i+1$ with the peers referenced by the other peer at level i, if their paths agree up to the i-th bit, and so on.

The maximum number of peers included in the references of a peer is a parameter that has to trade-off the amount of information stored at each peer with the availability of peers. In particular, if P is the probability that a peer is online, M is the number of peers referenced at each level, and l is the query length, in [4] it is demonstrated that the probability to perform a successful key search is given as

$$(1-(1-P)^M)^l.$$

As an example, this allows to reach a 99% probability of a successful search for $l = 10$ and $P = 30\%$ with just $M = 20$ references at each level.

P2PR-Tree The P2PR-Tree [59] search mechanism is based on the recursive partitioning scheme of the R-tree [39]. The search space is first statically partitioned into *blocks* and each block is statically divided into *groups*. Each peer then has to compute the Minimum Bounding Box (MBB) of the keys it indexes and is associated to all the groups whose region intersects with the peer MBB. The static decomposition of the space has an important advantage during both the search phase and the joining of a node to the network. During the search, first the R-tree

local to the node that originated the query is searched and results are (possibly) returned. If the result set is empty, the query is routed towards the relevant block and group: to do this, each node should only maintain the address of one peer for each block and of one peer for each group in its block; this way, the R-tree indices of all the peers are augmented of the static structure of blocks and groups, thus allowing an efficient search of the tree relevant for each query.

A peer joining the network requires that the R-tree corresponding to the new node is included in the appropriate group. This, of course, might require splitting the group node of the distributed P2PR-tree, thus the overall structure is not, in general, height-balanced. Moreover, since the MBB of a peer is not guaranteed to be contained in the statically chosen region of a group/block, the address of a peer can be replicated over the structure, thus the P2PR-tree is more similar to a R+-tree [72]. It has however to be noted that the hierarchical nature of the R-tree allows this structure to perform similarity searches, i.e., the user can also search for key values which are close to a given one, without recurring to external mechanisms (see Section 2.3).

2.2.3 Hybrid Approaches

Hybrid approaches usually consist in DHTs supported by a hierarchical structure that helps achieve the logarithmic search complexity.

P-Tree In the P-tree [23], nodes are mapped, by way of a hash function, to identifiers in a ring, as in Chord [74]. However, to speed-up the search for a key v, a distributed B+-tree is used. In particular, each peer only maintains the left-most root-to-leaf path of the overall B+-tree and relies on other trees to obtain the whole tree, i.e., right-most subtrees are not complete and only contain pointers to other peers. Searching for a key v is performed as follows: the node p where the request has originated looks within its tree to see if it is relevant for the query. If not, the request is routed towards the appropriate nodes by using the pointers stored within the right-most branches of the p's tree. This guarantees that, for a tree of order d, the search is performed in $O(\log_d N)$ time and the total storage requirement at each peer is $O(d \cdot \log_d N)$. In [23], algorithms are also provided to guarantee the maintenance of the P-tree in case of insertion/deletion of nodes in the network: the two main components are the *Ping* and the *Stabilization* processes, that are used to detect if peers are alive and in a consistent state, and to repair the inconsistent entries in the P-tree, respectively. It should also be noted that the use of a hierarchical structure, like the B+-tree, allows users to also perform range queries, i.e., to request for all documents whose key is included in an interval of values, thus enriching the expressivity of queries.

Viceroy Viceroy [56] is a hybrid solution since it uses wrapped peer identifiers, much as done in Chord [74], but maintains a distributed connection graph that emulates the behavior of a butterfly network. Each node in Viceroy is given an hash-based identifier and is assigned to a random level (there are $\log N$ levels). The routing table of each peer consists in three types of links (see Figure 14)):

1. a *general ring*, where each peer is connected to its successor and predecessor in the wrapped identifier space,
2. *level rings*, where peers at the same level are connected to each other in a ring, and
3. the *butterfly*, where each node at level i is connected to two "down" nodes at level $i + 1$ and to one "up" node at level $i - 1$.

The routing algorithm for searching the node corresponding to a key v consists in three phases:

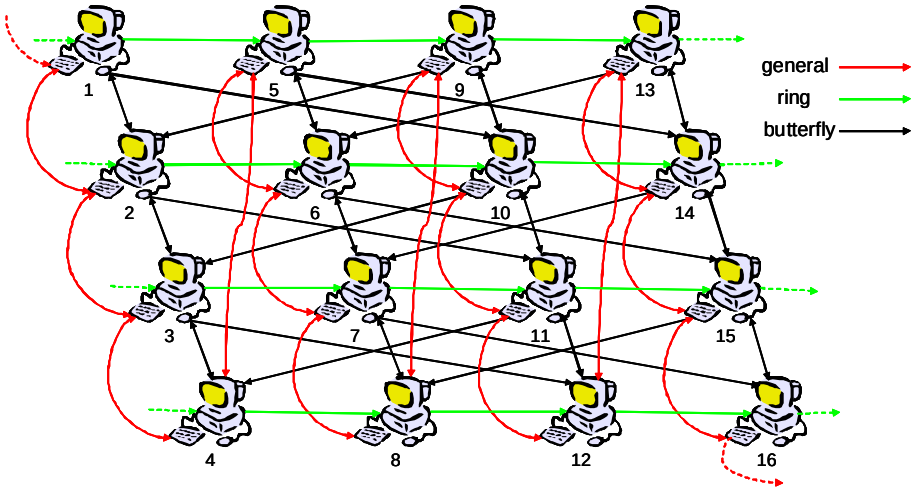


Figure 14: An ideal Viceroy network with 16 nodes: the three different types of links are highlighted.

1. first, at most $\log N$ steps are performed to navigate the butterfly up to level 1;
2. then, at most $\log N$ steps are used to navigate the butterfly down to the node whose identifier is closest to v;
3. finally, at most $\log N$ steps are needed to traverse the general ring links to find the peer containing the location of v.

To achieve logarithmic complexity in the first two phases, the butterfly links are chosen so as to mimic a Binary Search Tree in the identifier space.

The level rings links, which are not used during the search phase, are however necessary when a node joins/leaves the network. In particular, when a peer p joins the network, the construction of its routing table is as follows:

- p is assigned an identifier and is placed in the appropriate level i;
- p contacts its successor in the network and the general ring links are updated;
- p is placed in the appropriate position in the i-th level ring and links are updated;
- the butterfly links of p are computed by contacting the successor of p in the i-th level ring.

Similar operations are needed whenever a peer leaves the network.

2.3 Supporting Complex Query Paradigms with Structured Networks

As we saw in Section 2.2, structured networks are much more efficient with respect to unstructured networks, since they can provide logarithmic search times, and require less computational power to peers, since flooding is usually not contemplated. On the other hand, they require a higher overhead for maintaining the network structure, and are not immediately suited to requests different from the simple key search, whereas unstructured networks have a higher flexibility, since each peer can pose virtually any kind of query to neighboring nodes. This is indeed, the main drawback of structured networks [43], since DHTs usually only support exact match queries. In this section, we present some approaches proposed to extend the flexibility of structured networks.

In [38], Locality Sensitive Hashing (LSH) [46] is used to map data values in the identifier space and a Chord [74] network allows searching for node identifiers. The use of LSH guarantees

that similar values are mapped to the same node with high probability, thus a range search on the key space is likely to find all relevant data in a single peer, or at least in neighboring peers. This allows to avoid issuing a different query for each value in the search range, as it is the case when using hashing functions that do not preserve locality.

2.3.1 PIER

In previous sections, we have shown how DHTs can be used to efficiently perform key-based search. Recently, some approaches have been presented that use DHTs to map complex objects, e.g., fragments of relational tables, portions of inverted index, to peers in the network. Such techniques use DHTs not as a tool for performing efficient retrieval on a peer-to-peer network, but as a service to distribute indices, relations, and so on. They are, therefore, built "on top" of DHTs and can use any of the structured overlay networks described in Section 2.2.1. Examples of such techniques are the distribution of an inverted index in [11], that is surveyed in Section 3.1, and PIER.

PIER [44] is a query engine built on top of CAN [67] with the goal of providing the query processing facilities of existing relational DBMS to the scalable and flexible world of peer-to-peer networks. To this end, three basic principles are used to allow the engine to scale to a world-wide size:

Relaxed Consistency: not all the ACID properties of transactions are guaranteed. In particular, PIER focuses on providing a high availability to data, and sacrifices the tolerance to network partitions; thus, results provided by PIER might be incomplete, e.g., because a part of the network is unreachable.

Natural Habitats for Data: the data are stored locally and are accessible by way of wrappers, that provide to other peers only the requested information.

Standard Schemata: the semantics of data are shared by all peers in the network; this, as we will see in Section 3.3, seems to be a very strong requirement. However, as it is argued in [44], several applications exist where data have a standard schema, e.g., data provided by network monitoring tools like Snort and tcpdump.

Each DB relation in PIER is stored in a single node and simple operators, based on the use of the underlying DHT supplied by CAN, are provided for selection and projection. Moreover, two distributed join algorithms are presented, *symmetric hash* and *fetch matches*, that represent adaptations to the DHT case of already existing algorithms.

3 Content-Based Query Processing

Query processing algorithms presented so far exhibit some limitations that prevent the use of a rich search paradigm:

- structured networks mainly allow only key-based searches, since they are based on mapping document identifiers on network peers;
- unstructured networks, on the other hand, are able to provide keyword-based search, but network peers are searched blindly, because each peer does not know the content of other network peers, except when query results are returned.

The feasibility of peer-to-peer web indexing is analyzed in [52]. In particular, the partitioning over a number of peers of an inverted file built over all the documents in the web (around 3 billion web pages indexed by Google in 2003) is considered, and two approaches are proposed:

Partition by Document: Here, the "global" inverted index is *vertically* partitioned among peers, so that each peer only indexes its local documents. The limitation of this approach is that queries should necessarily be flooded to all peers to retrieve all results, thus the network bandwidth is a bottleneck.

Partition by Keyword: With this approach, the overall inverted index is *horizontally* partitioned and every peer is responsible for some keywords. The network bandwidth is, however, still a bottleneck due to the extra work required by multi-term queries, that need to transfer among peers posting lists that have to be intersected.

The overall conclusion drawn in [52] is that, for performing keyword-based searching using classical IR algorithms in conjunction with network-based query processing algorithms, the costs should be reduced by at least an order of magnitude of available resources. To overcome limitations of network-based query processing algorithms, content-based techniques have been proposed in recent years, providing efficient algorithms for keyword-based searching. Basically, such algorithms mimic those proposed for unstructured networks, i.e., the query is forwarded towards neighboring peers: however, now each peer collects and stores data about other peers and uses such information to route the query to the peer(s) that are likely to provide a result for the query. Information about peers are stored in a "profile", that provides a description of the data stored by each peer. Query processing algorithms differ in the way such data is collected, stored and used to select the promising peers for each specific query.

We concentrate on three kinds of profiles: content summaries, routing indices, and semantic mappings.

Content summaries (Section 3.1) are data structures that statistically characterize the local content of a peer. They were first introduced to solve the problem of source selection in heterogenous environments [35] and have been also applied to scenarios characterized by incomplete information, such as the Hidden Web [47], where content of data sources is not available to search engines and crawlers and can, therefore, only be known by dynamically querying a web-accessible interface. In the peer-to-peer scenario, neighboring peers can be considered as remote data sources storing unknown knowledge, thus content summaries could be in principle used to select the peers containing data relevant for each query.

Routing indices (Section 3.2) are data structures which exploit the knowledge that a peer has about its neighborhood, obtained by collecting past queries or by direct exchange of information between nodes. For each neighbor n, the profile consists of a short summary of the content provided by n and its neighbors. Each profile, thus, can estimate the content "that is reachable if the query is forwarded to that peer", and query processing algorithms can select the best neighbors according to their relevance with respect to the query.

Semantic mappings (Section 3.3) characterize schema-based environments, where the data in each peer is described by a schema, e.g., relational tables, a schema describing semi-structured data collections, or an ontology expressed using a description logic formalism. Typically, this scenario is characterized by a high locality, in the sense that there is no shared schema on which peers have agreed upon, thus queries expressed with respect to a peer schema are not even guaranteed to be understood by another peer. Such *semantic conflicts* arise whenever we have to deal with heterogeneous descriptions. The role of semantic mappings is to facilitate the translation process from one schema to another, with a particular focus on the preservation of the original meaning of the query. In this sense, they have a very different role during query processing with respect to the one assumed by routing indices: the latter, in fact, are usually presented as a tool to improve query efficiency maintaining a high recall in the answer, in the typical IR-style fashion; semantic mappings, on the other hand, concentrate on the completeness problem, which

means that the focus is on the creation of a translation that is as accurate as possible (of course, efficiency issues are still considered).

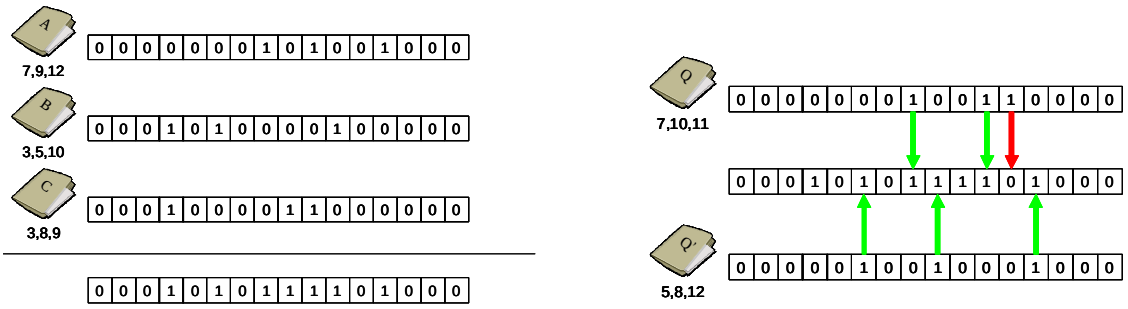
3.1 Content Summaries

A content summary (CS) is a statistical data structure associated to a database or a set of documents that can be used to estimate the relevance of the database/set of documents with respect to a given query. The two main approaches that can be found in the literature are based on collecting statistics from queries answered by the system or on directly crawling the knowledge base. The latter is, of course, a more appealing solution, but it is not applicable to many real scenarios, e.g., the Hidden Web; in such cases, the task of building a content summary is split in two parts:

1. the choice of a set of sample queries to send to the data source, and
2. the aggregation of all the answers into a coherent statistical description.

Since content summaries are only used to characterize the content of data sources, here we mainly detail algorithms used to build and update the profile for each site.

Bloom Filters A Bloom filter [13] is a compact and lossy representation of sets. In particular, a Bloom filter is a bit-vector of length l associated to a family of independent hash functions that maps set elements to integer values in $[0, l[$. To obtain the representation of a set, each element is hashed and the bits in the vector corresponding to hashed values are set. To search if a given element is contained in a set, the element is first hashed and bits in the Bloom filter corresponding to hashed values are checked; if any of the bits is not set, then the element is not contained in the set, otherwise it *may* be contained and a full scan is required to avoid *false positives*. False positives are due to collisions in hash functions and to the presence of several sets in the Bloom filter: if the number of represented sets becomes a significant fraction of the vector width l, then the filter is said *overloaded* and performance is likely to degrade, due to the high number of false positives. To build a Bloom filter over a set of documents, first each document is represented as a set of terms, by using stemming and stop-lists [70]; then, the Bloom filter of the documents is obtained by bitwise *or*-ing the hashed codes of terms in all documents.



(a) (b)

Figure 15: A Bloom filter of width $l = 16$ with 3 hash functions built on three documents (a); in (b), query Q is filtered out, since it differs in bit 11 with the filter, while query Q' is a false positive.

This description has the property of being compact and efficient to be looked up and to be kept up-to-date. However, it requires either a global keyword space or global hash functions [49].

GlOSS In [35], a number of solutions, collectively named *GlOSS* (Glossary-Of-Servers Server), is proposed for the selection of text-based data sources. In particular, two versions of summaries are presented: *vGlOSS*, based on the vector space model, and *bGlOSS*, based on the Boolean model. In both cases, summaries are obtained in a statistical way and are used to estimate the usefulness of each data source with respect to a query, expressed according to a particular model (vector/Boolean). In particular, in [35] this problem is solved requiring that each data source periodically extracts statistics about local term frequencies and sends them to a centralized *GlOSS* server (e.g., using the STARTS protocol [33]). In this way, each data source has a local index, that is always up-to-date, while only the central server collects information about all data sources, and is thus able to select the sources that are more relevant for a query. Moreover, the use of a central server allows the estimation of inverse frequencies for document terms in the vector space model; this is, as we will discuss also in Section 4.1, is the basic problem preventing the use of distributed indices for document retrieval.

To decrease the centrality of the *GlOSS* server, in [35] a hierarchical solution is also adopted, where a *hGlOSS* server keeps information about a number of *GlOSS* servers. Such information is, basically, the same that is kept by a *GlOSS* server about data sources: instead of collecting information about documents in data sources, the *hGlOSS* server maintains information about databases in servers.

CORI In [16], a source selection algorithm is proposed for distributed information retrieval, where data sources are characterized by way of classical $tf \times idf$ weights for each term (see also Section 4.1). In particular, the Bayesian Inference Network model is used to describe the data sources, so that each data source D is connected to the terms contained in D, and each query q is connected to terms included in q (see Figure 16). The probability that a data source D is relevant for a query q is computed by instantiating the D node and propagating probabilities through the Bayesian network towards the query node q. In particular, the probability $p(w|D)$ that a keyword w is observed at a data source D is estimated by way of a variation of the classical $tf \times idf$ formula [70]:

$$\begin{aligned} tf_{w,D} &= \frac{df_{w,D}}{df_{w,D} + 50 + 150 \cdot \frac{\Delta_i}{\Delta_{avg}}} \\ idf_w &= \frac{\log\left(\frac{N+0.5}{cf_w}\right)}{\log(N+1)} \end{aligned} \tag{1}$$

where $df_{w,D}$ is the number of documents in D containing term w, Δ_i is the number of documents in D, Δ_{avg} is the average number of documents in a data source, N is the number of data sources, and cf_w is the number of data sources containing term w. The only global values that have to be To obtain the overall score for a data source D with respect to a query q, the probabilities $p(w_i|D)$ of all terms w_i in q have to be combined in the Bayesian network by way of probabilistic operators corresponding to operators present in q, e.g., using the average, the product, a weighted average, and so on.

Equations 1, used to compute the relevance of a data source, are known as the CORI algorithm for ranking data sources, even if the name CORI was originally intended to apply to the broader use of inference networks for ranking databases [16].

Overlap-Aware Source Selection In [11], the CORI model is used to establish the *novelty* of a data source with respect to a given reference document collection. In this way, a peer (document source) might be able to rank other peers with respect to documents already seen, e.g., in the peer itself or in other peers previously contacted. A global directory is used to maintain term-peer pairs, and such index is distributed among peers using a DHT (see Section 2.2.1). Whenever a new document appears in a peer p, p *posts* such information in the global inverted

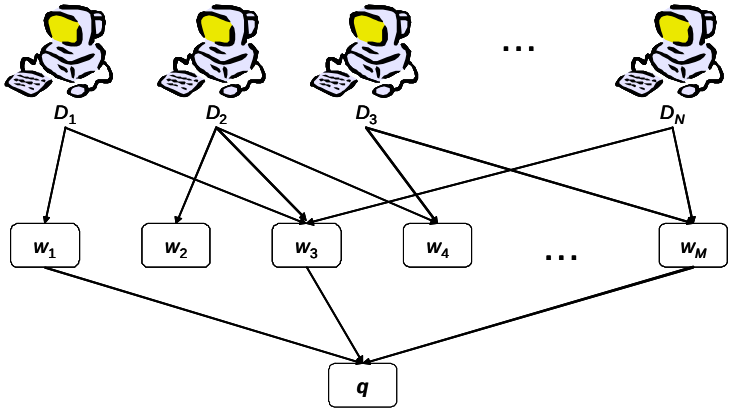


Figure 16: A sample Bayesian Inference Network: the upper resource network connects the N sources to the M terms, while the lower query network connects the query q to the terms it contains.

index by contacting all peers that are responsible for the terms contained in the document. Bloom filters are used locally at each peer p for storing the inverted index on terms included in documents in p.

To estimate the novelty of a peer p with respect to the collection of documents in peers already contacted (or selected), the Bloom filter of p is compared to the Bloom filter obtained by bitwise *or*-ing the Bloom filter of the already-seen peers. The novelty of peer p can therefore be estimated as the number of bits that are set in p's Bloom filter but not in the overall Bloom filter. Peers are then ultimately ranked according to a weighted combination of relevance and novelty.

Focused Probing In [47], *focused probing* is proposed to estimate the frequencies of documents for the classification of hidden textual databases. To this end, the document frequencies of words characterizing each category have to be extracted for every data source, i.e., for each D and for each w, the number of documents in D that contain w is estimated. So-computed document frequencies can then be used to estimate the number of documents in D matching a particular query, by using above described database selection algorithms, like *bGloss* [35] or CORI [16]. Furthermore, each database can be classified under a given hierarchy of categories, since keywords are associated to categories, e.g., "cancer" is representative of category health, whereas "jordan" and "lakers" are associated to categories basketball and sports.

As said, to build the content summary of a data source D we have to estimate the actual document frequencies of all the keywords in D. To this end, queries (*probes*) are adaptively sent to D to effectively produce a document sample that is representative of the topics contained in D. In particular, if we send a query using the keyword w to D, then we know exactly the document frequency of w, but we also know the relative document frequency of other keywords contained in the returned documents, i.e., the frequency of keywords in the sample. Actual frequencies of keywords that have not been probed can then be estimated by assuming a statistical dependency between the relative document frequency (this is called *SampleDF* in [47]) and the actual document frequency (*ActualDF* in [47]) and assuming a power law relationships in the document frequencies. Of course, since the goal of [47] is to classify each data source D under some categories, the probes to be sent to D are chosen in the set of keywords representing each category.

QProber QProber [36] is a technique for classifying text data sources by way of their content, which is statistically characterized by probing each source with an appropriate set of queries.

The classification scheme is based on the concepts of *coverage* and *specificity* and allows to measure the relevance of a data source D with respect to a category C_i. The coverage of D with respect to C_i is computed as the number of documents in D that are classified as relevant for C_i, and represents the "absolute" amount of information about C_i in D. On the other hand, the specificity of D with respect to C_i is computed as the coverage of D for C_i divided by the total number of documents in D, and represents how important is C_i within D. An ideal *coverage-based* classification would assign D to all categories for which D has enough documents; on the other hand, an ideal *specificity-based* classification assigns D to the categories that cover a significant fraction of documents in D.

A rule-based classifier is then used to estimate the coverage and the specificity of each data source according to a hierarchy of categories. In particular, for each category C_i, a set of rules $w_k \rightarrow C_i$ is created, where w_k is a conjunction of keywords. If the database D is queried using the w_ks, the coverage of D for C_i can then be approximated as the sum of documents obtained as result for each w_k, while the specificity can be calculated by taking into account the specificity of the parent category of C_i in the taxonomy and the coverage of sibling categories.

3.1.1 Content Summaries in Peer-to-Peer Networks

Content summaries have been extensively applied to peer-to-peer networks to improve the query recall for unstructured networks or to provide content-based search capabilities to structured networks.

Content-Based Leaf Node Selection In [53], a Gnutella-based super-peer network is considered and local inverted indices are maintained at each super-peer to represent the content of peers in its corresponding cluster. In this way, when a peer p requests for a query q, the local index of the super-peer s of the p's cluster is first searched for q and, in case an answer is not found, q is flooded by s in the super-peer network. Since, however, the resource selection based on exact term matching may lead to overload the network with messages, in [53] it is proposed to use a Kullback-Leibler (K-L) divergence resource selection method, so that the probability that a peer p will satisfy q is computed as the negative of the K-L divergence between q and the collection stored at p: such value can be computed by only considering local term frequencies, thus the relevance of each peer p with respect to q can be *independently* evaluated by its corresponding super-peer s, since s contains the inverted index of terms contained by sub-peers in its cluster. Upon receiving a query q, thus, each super-peer can autonomously choose the most relevant peer(s) for q and forward q to them, or can flood q to other super-peers if the best peer is not relevant enough. This, as shown in [53], allows to increase the query recall and, at the same time, to reduce the number of messages with respect to the simple TTL-limited flooding algorithm of Gnutella [1].

Hierarchical Summary Indexing In [73], a Hierarchical Summary Indexing technique is presented for efficient keyword search in a super-peer network. Three indexing levels are defined:

Unit-Level: at this level, units, as documents or images, are summarized;

Peer-Level: this level summarizes all information contained in a network node;

Super-Level: the higher level contains summaries for all information contained in a peer group, i.e., in all peers served by a same super-peer.

Indices at all levels contain documents represented as points in a vector space represented by keywords, and documents coordinates are obtained using $tf \times idf$ frequencies (see Section 4.1). To reduce the space needed by each document in the index, Latent Semantic Indexing (LSI) [26]

is applied as a dimensionality reduction technique, and points are indexed by means of a VA-file [79]. When a query q is received by a peer p, the peer can search its peer-level index to check if it can answer q locally; then, q is forwarded to the super-peer of p, s; s can check its super-level index to see if any of its sub-peers can provide results for q, and also forwards q to other super-peers by flooding. Updating of indices is, of course, an issue, since a lot of summaries have to be updated when a new document is inserted in the network or when a peer joins/leaves the network; to this end, the Accumulated Information Ratio is computed for the new document for each index: if such value is higher than a predefined threshold, the document significantly alter the LSI summary and the index is rebuilt, otherwise, the index is not updated and the document is just inserted in the VA file. The main limitation of this approach is the fact that *idf* values are used, thus each peer should estimate the term frequency over all documents in the network (see Section 4.1).

Cell Abstract Indices In [78], Cell Abstract Indices (cell AbIx) are proposed to support content-based approximate search. In particular, each document in the network is summarized by way of d features and the d-dimensional space is divided in cells, much as done in CAN [67]. Each peer is also mapped to this space, by computing a summary of all its documents (in [78], it is proposed to use the average of features of all documents), thus each cell contains a number of peers; one *head* peer is chosen among such nodes as a super-peer in charge of keeping track of super-peers of neighboring cells. When a query q is issued at peer p, q is forwarded to the head peer of the cell containing q, that will route q towards the cell containing q (the *end* cell), using the CAN routing algorithm (see Section 2.2.1). When the head s of the end cell is reached, s contacts all the peers in the cell asking for q; to improve query recall, s can also forward q to neighboring cells, or to their neighbors, according to a maximum number of hops defined by the user that issued the query. Of course, this is only an approximate algorithm, because there is no guarantee that the AbIx of a peer p is close to all the documents contained in p.

3.2 Routing Indices

A routing index (RI) is a data structure that, given a query q, returns the list of neighbors which probably store documents that are relevant for q. The importance of routing indices for query processing is twofold:

1. they improve search efficiency, because blind search strategies typical of unstructured networks (see Section 2.1) can be (partially or completely) avoided, and

2. they increase query recall, because search algorithms can be focused towards most promising sites, avoiding peers that are unlikely to provide results.

In general, a routing index is a list of entries constituted by a semantic description (e.g., a property name or value) and a neighboring peer, which represent the direction that has to be taken in order to reach one or more data instances characterized by that description. Sometimes, entries are also enriched with statistical information, e.g., the number of elements referring to the entry, or a frequency value.

In the query processing scenario, routing indices are used during the neighbors selection phase: each entry is matched against the query and sites associated to relevant entries are considered for routing the query, while other peers are pruned from the search space. The query is then forwarded to selected peers, where it can be locally solved and/or forwarded to other peers. With respect to content summaries, routing indices are usually more compact, because they don't have to provide a summary about *all* the content of neighboring peers; in addition, they can also take into account the content of peers reachable from each node, thus the problem of limiting the horizon is raised when maintaining the content of other peers.

3.2.1 Keyword-Based Routing Indices

The keyword-based scenario consists of a network where documents and queries are characterized by a set of attributes and properties. Entries stored in routing indices are lists of keywords and are usually collected by monitoring the network for queries answered successfully.

In [24], the problem of defining routing indexes is handled in a systematic way. In particular, the authors distinguish between three different classes of routing indices:

Compound RI: Compound routing indices store, for each outgoing link, the number of data instances that are globally reachable following that path and those that are reachable for each topic of interest (i.e., of interest for the peer). The query processing algorithm first identifies the overlap between the set of keywords in the query and the set of topics in the RI and, for each of them, the relative occurrence frequency, i.e., the ratio of the number of documents for each topic divided by the total number of documents, is computed; such values are then aggregated to provide a "goodness" score for each peer (for example, if the query is conjunctive, frequencies are multiplied). Peers can then be sorted with respect to the goodness score and the query might be forwarded only to the first few neighbors. Algorithms for building and maintaining the routing indices at each peer are extremely simple, from a logical point of view, since they only need to count documents relevant for each category; however, they require several messages to be exchanged between peers, particularly when the document set of a peer changes or when a node joins/leaves the network.

The major drawbacks of compound routing indices derive from the absence of an horizon: if, on one hand, it could be desirable that a routing index represents a condensed snapshot of the entire network, on the other hand, this approach suffer several limitations, since building and maintaining routing indices requires a high number of messages, particularly for dynamic collections of documents, and cycles eventually present in the network are not easily dealt with, requiring a particular cycle-discovering mechanism; finally, the indexing policy treats all documents and peers at the same level, giving no importance to their distance from the query originating node, while data instances close to the source node are intuitively better than farther ones.

Hop-Count RI: An hop-count routing index stores, for each number i of hops, statistics for peers in the network that are reachable in i hops, i.e., it gives information about the content that can be found in the network at increasing distances from the peer, up to a fixed number of hops (this is called the *horizon* of the routing index). For example, if the horizon is equal to two, the first level of the RI stores the number of objects one hop away from the peer, while the second level stores the number of objects contained in nodes that are one hop away from the current peer's neighbors. For each level l, the RI is obtained as a compound RI taking into account only nodes which are l hops away.

From a theoretical point of view, even compound routing indices may have an horizon. However, in [24], a significant difference between the two data structures is considered, i.e., the more sophisticated ranking strategy that can be adopted with hop-count RI. In particular, it is proposed to weight differently peers at different levels, assigning to each peer a value equal to the ratio between the number of documents available and the number of messages needed to get such documents: in this way, the number of results that will be returned per each sent message is estimated.

Exponential RI: Exponential routing indices are introduced to get the best of compound and hop-count routing indices. From compound RIs, the idea of not having an horizon is borrowed, thus no *a-priori* limitation to the possibility of describing the entire network exists. From hop-count RIs, the idea that closer neighbors are more relevant than distant

ones is used. From a practical point view, exponential RIs are similar to compound RIs, because they store the number of instances *globally* reachable following a given path along with the number of topic-related instances. During the construction process, however, such values are attenuated using a given factor, that takes into account the hop-count from the target peer and that increases with the distance to the target peer. In this way, the presence of cycles in the network is no longer a problem, because the feedback effect of a cycle is attenuated by the exponential factor and the fact that the values in the RI are not updated if the new computed value does not differ significantly from the older one.

In [24], the three solutions are evaluated on simulated networks having different topologies. The overall conclusion is that compound RIs are slightly better than hop-count and exponential RIs when query performance is taken into account, except when cycles are present. However, when the cost of updates is an issue, the very high number of messages required by compound RIs shows that their effective use in dynamic networks is not feasible.

Local Indices In [82], so-called *Local Indices* are introduced, among other solutions to alleviate the flooding problem in Gnutella-like networks, for routing queries towards nodes containing relevant documents. To this end, each peer n should maintain an index over the data contained in all nodes within distance r from n, i.e., all nodes that can be reached in no more than r hops from n. The *radius* r is a system-wide value that trades off the amount of data in the local index and the efficiency of the search algorithm. Local Indices are thus a particular case of hop-count RIs.

Local Indices are used in conjunction with a system-wide *policy*, specifying the depths at which each query has to be processed. For example, if the policy is $\{1, 5\}$, then the query is forwarded from the node originating it to all its neighbors (depth 1): since 1 is in the policy, then all such nodes will try to solve the query using their local indices *and* will forward the query to neighboring nodes (depth 2). Nodes at depth 2 will not process the query (2 is not in the policy) but will only forward it to other nodes (depth 3), and so on until nodes at depth 5 are contacted. Since 5 is the last value in the policy, such nodes will try to solve the query using local indices but will not forward it to other nodes, thus the flooding is limited.

Intelligent Search Mechanism In [48, 84], the *Intelligent Search Mechanism* is proposed. The authors state that a search strategy which uses routing indices is always characterized by four elements, namely:

Search Mechanism: this is the classical routing mechanism, used by a peer to exchange queries with its neighbors. The peer that receives a query (either obtained from an user or routed from another peer) only forwards it to most promising peers, according to the *peer ranking mechanism* (see below).

Profiling Structure: for each neighboring node of peer n, a *profile* storing some information on its content is maintained in n. In [48], for example, peers store the most recent queries that were successfully answered by their neighbors; later, such structure was enhanced by also including the number of results returned for each query [84].

Peer Ranking Mechanism: profiles can be used to select peers to which forward queries, disregarding the others. To this end, upon receiving a query, the *RelevanceRank* for each neighboring peer is computed, reflecting the similarity between entries in the profile for that peer and the actual query. The similarities between the query and past queries stored in the profile are computed according to a function (see below) and are aggregated to provide a single *RelevanceRank* value, according to which peers are ranked.

Query Similarity Function: this is used to match the query against entries in the profile. It is suggested to adopt the cosine similarity function [70], a well-known technique used in information retrieval to compare lists of keywords (see also Section 4.1).

In this case, maintaining the routing index only consists in counting the number of results obtained from each neighboring peer for past queries, and to estimate the number of results for the actual query according to its similarity to stored queries. To avoid the formation of cycles in the forwarding of queries, some random peers are always chosen in the set of best peers, so that a larger part of the network is explored for results.

Peer Indices In [32], the *Peer Index* (PI) is presented to improve search efficiency on collections of XML documents. Each PI is basically an inverted list that maps keywords to peers in the network; to avoid indexing the whole content of the network, each peer has an horizon, thus it only *sees* a fraction of nodes, and can autonomously decide which keywords to include in its PI and which ones to export for other peers' PIs. The query processing algorithm is very simple, since it just has to intersect peers lists obtained for each query keyword to obtain the list of peers that are likely to contain qualifying documents. The drawback of this approach, as also outlined in [32], is that, depending on the value of the horizon, each PI requires a lot of space on each peer and the update protocol is bandwidth consuming, since a content change at any peer involves the update of the PIs of peers in the horizon. To limit this second aspect, however, it is suggested to use an update policy that piggybacks updates inside queries, thus having an overall system which gradually evolve toward a globally stable state.

An indexing structure like the Peer Index suffers, however, from an intrinsic limitation: since when a peer n indexes keywords from nodes that are not directly connected with n, the IDs of such nodes are stored in n, this is equivalent to creating links from n to such nodes, thus obtaining a highly-connected network. This, of course, can have a detrimental effect on dynamic networks where the frequency of nodes joining/leaving the network is high, since it requires a high number of messages to update the PIs.

Attenuated Bloom Filters In [68], "attenuated" Bloom filters are presented. For each peer, a multi-level data structure that is logically equivalent to a hop-count routing index [24] is maintained for each neighboring peer. The l-level attenuated Bloom filter for peer p stored at peer n is a Bloom filter which summarizes all documents which can be found at a distance l on the way from node n to node p. Given a query and a routing index, if the query matches the Bloom filter at level l, the score for that given peer is increased by $\frac{1}{2^l}$, thus the importance of a peer is geometrically decreasing with the distance to the originating peer. Such score is computed for all neighboring peers, resulting in a ranked list, which gives the relative importance of each neighbor with respect to the query.

The updating of attenuated Bloom filters presents two major difficulties:

1. Due to collisions in keyword signatures, not all changes in a peer have to be reflected in all the RIs in its neighbors. Sending useless updates is prevented because each peer not only maintains the Bloom filter for its neighbors, but also a copy of the neighbors' view in the reverse direction, so that only compressed (differential) updates are needed.

2. The topology of the network can lead to different paths (of different length) connecting two peers. Thus, updates for a source peer n might reach a peer p more than once (precisely, once for each level l that has to be updated in the attenuated Bloom filter of p). This redundancy affects the network bandwidth, thus, in [68], two different *filtering* strategies are proposed: one performed only at p (p simply ignore subsequent arrivals of an update of n through different paths) and one performed at n (upon arrival of a duplicate update, p informs n's neighbors that no new updates are needed).

Multi-Level Bloom Filters In [49], Bloom filters are extended to Multi-Level Bloom Filters (MLBFs) to support querying of XML documents. Breadth BFs and Depth BFs are simple Bloom filters where only portions of XML trees are stored in a breadth-first and in a depth-first manner, respectively. By maintaining a MLBF on its XML documents (*local filter*), a peer p can therefore efficiently answer to path queries by using query processing techniques created for "regular" Bloom filters (see Section 3.1).

Exploiting MLBFs in a peer-to-peer environment relies on the use of a hierarchical network organization (although authors of [49] have also applied their techniques to non-hierarchical topologies). Each peer maintains its local filter and a *merged filter*, obtained by merging (through a bitwise *or*) MLBFs of nodes in its sub-tree. When a query q is originated at a node p, the local filter is first searched using q and local documents are possibly returned; then the merged filter of p is searched and, if a match is found, q is propagated to nodes in p's sub-tree, until either a merged filter indicates that q cannot be found or a leaf node (having no merged filter) is reached; finally, q is routed to the parent of p until a root node is found. When a root node n is reached, n should check its merged filter and also route q to other root nodes. Finally, updates to MLBFs are dealt with in a similar way as for regular BFs.

Histogram-Based Routing Indices Another variant of hop-count routing indices is presented in [65]: here, the particular case is considered where each node stores a single relation R with a numeric attribute x and users are interested in range selection queries on x. Each peer maintains, for every neighbor n, an histogram $H(n)$ on values of $R.x$, describing the content of the network up to an horizon: in particular, the i-th bin of the RI of peer n is computed as the fraction of values of $R.x = i$ that can be obtained in all nodes reachable from n within the horizon. In this way, given a query with radius r asking for particular values of attribute $R.x$, a peer can estimate the number of results that can be retrieved (within r hops) by forwarding the query to n. The peer, thus, is able to rank its neighbors with respect to a given query. The search strategy adopted is *depth-first*, i.e., a peer forwards a query to the most promising neighbor and waits until a result is returned. If no answer is obtained, the not-yet-visited most promising neighbor is contacted, and so on until the search can be considered terminated.

In [65], a clustering algorithm is also proposed to group together peers having similar histograms so that each query can be routed towards the most promising *group* of peers. To this end, similarity metrics are defined on histograms and, upon registration of a new peer p in the network, p can search for the k peers having the most similar histograms with respect to p and keep links to such peers. In this way, whenever a query is routed to a peer similar to p, p is also immediately contacted, thus increasing locality.

Adaptive Probabilistic Search In [77], the *Adaptive Probabilistic Search* algorithm (APS) is presented, that can be considered a variant of an exponential routing index. In APS, each node p keeps a list of entries representing objects (e.g., keywords) that were requested in the past and that were forwarded to neighbors of p. For each object, the value stored for a neighboring peer n represents the probability of n to answer successfully to a request for the same object.

The update and the search phases are merged into a single algorithm, which is explained in the following:

1. The search is based on the m-walkers paradigm, i.e., only the initiating peer forwards the query to m neighbors, while subsequent peers only forward the query to a single node.

2. When a peer p receives a query q, it chooses the neighbor(s) having the highest probability of obtaining a successful answer to q. Then, p adopts either the *optimistic* approach (p assumes that a result will be obtained from the chosen neighbor, thus it increases its probability for the given object) or the *pessimistic* approach (p pessimistically assumes the choice was wrong, and decreases the probability).

3. A walker may stop with a success, if the requested object is found, or with a failure, because the TTL has expired or an already searched node has been contacted. With the optimistic approach, a failure must be tracked back to the peer p, in order to decrease the probability of all peers on the way. With the pessimistic approach, the backtracking is only required for successful requests.

In APS, "referential" content-based paths for the flow of the information are built, avoiding the need of a complex update algorithm, but at the expense of maintaining a *soft state* for each query that is processed, in order to allow backtracking and updates.

A variant of this algorithm, proposed by the authors, is the *weighted-APS*: with the pessimistic approach, it is possible to weight the increment of the probability, along the backtracking path to the originating peer p, by a factor inversely proportional to the distance from the peer, in order to prefer nearer objects.

Adaptive Query Routing In [81], the *Adaptive Query Routing* (AQR) method is proposed as a probabilistic technique for efficient routing of keyword-based searching. To this end, each peer p maintains a routing table containing, for each neighboring peer n and each key w, the probability that a document containing keyword w appears in n (or can be reached through n). Such probabilities can be explicitly stored in p by keeping statistics on results of past queries about w that have been forwarded to n, or can be estimated using probabilities for other keywords known for n and probabilistic information about overlap between keywords, i.e., which is the probability that a document containing w' also contains w. At query time, the peer p originating the query q has to compute the probability of finding q (only single terms queries are considered in [81]) in nodes reachable from each neighbor n; then, q is forwarded only to most promising peers.

The AQR approach suffer a main limitation in the fact that overall statistic probabilities about overlap of keywords should be globally known, thus AQR is only appealing for stationary scenarios where the set of keywords is known in advance and overlap probabilities do not change over time.

3.2.2 Schema-Based Routing Indices

Routing indices analyzed so far are built upon a keyword-based knowledge representation model and do not consider more complex scenarios. In this section, we survey Edutella [62, 63], a framework which is able to process schema-based queries and relies on schema-based routing indices: the system's underlying topology is an HyperCuP-based super-peer network (see Sections 2.1.2 and 2.2.1).

Objects in the network are RDF instances (see [80] for details about the RDF data model) and the queries are schema-based. Routing indices are also schema-based and are only stored in super-peers. Indices can be of two different types:

- *SP/P routing indices* describe peers directly connected to the super-peer, while
- *SP/SP routing indices* refer to the content of other super-peers.

Routing indices are, however, homogeneous, in the sense that they hold and exchange the same kind of data. With respect to keyword-based RI's, where entries are lists of one or more keywords, in [62, 63] a data structure is described that can store four different types of elements for each piece of data:

Schema Namespace: the RDF data model [80] allows to use multiple namespaces, thus the search mechanism has to take into account that a query should be routed only toward those peers that are able to support it.

Property/Sets of Properties: peers may support only portions of a schema, thus it is important to know who owns a certain property or set of properties.

Property Value Range: some properties assume values from a predefined topic hierarchy or vocabulary; the ACM CSS taxonomy [8], for example, describe the area of interest of a computer science paper: if a node has only papers about programming languages, it does not need to be contacted when searching for data mining. This indexing technique can be very useful if peers can create summaries of different schemas, aggregating meta-data from several information sources.

Property Values: these are specific values that a property may assume when referring to a data instance. RIs built on property values coincide with feature-based RIs.

The representation model presented in [63] is more complex than the keyword-based one and has a great impact on query processing, which relies on a variety of attributes and properties which are not present in the former scenario. As suggested in [63], if a peer matches the query, this means that the query can be understood, but it cannot be guaranteed that an answer can also be provided. This model, however, greatly improves traditional query processing techniques, allowing them to work at different granularities and, possibly, to exploit schema-based query evaluation plans.

3.3 Semantic Mappings

A semantic mapping (SM) is used to solve semantic conflicts between heterogeneous descriptions that may be found in different peers and consists in a way of translating (part of) a schema into another schema. To this end, each peer should import a remote knowledge, e.g., a part of the schema describing the data of the other peer, and interpret it according to its own "local" description and language.

The problem of establishing mappings between different schemata comes from data integration systems [51], where the main focus is to provide a uniform interface to a heterogeneous set of data sources. Such an interface is called *mediated schema*, because it represents the result of the mediation process needed to ensure the interoperability of the systems. How the mediated schema is obtained depends on the approach followed during the construction process: according to the *global-as-view* (GAV) approach, the global schema is built as a view over its data sources; on the other hand, the *local-as-view* (LAV) approach expresses the schema of each local source as a view over the mediated schema (see [51] for a survey). Suppose, for example, that we want to build the following mediated schema:

$$Movie(title, dir, year, genre)$$
$$Schedule(cinema, title)$$

and that data sources provide the following schemata:

$$S_1(title, dir, year)$$
$$S_2(cinema, title, dir, genre)$$

With a GAV approach, the queries on the mediated schema can be unfolded using rules that maps local schemata to the global schema. For example, using a datalog formalism, we can obtain:

$$Movie(title, dir, year, genre) : -S_1(title, dir, year), S_2(\_, title, dir, genre)$$
$$Schedule(cinema, title) : -S_2(cinema, title, \_, \_)$$

GAV techniques are conceptually easy: they specify how to extract tuples from the sources using a query expressed on the mediated schema and rely on the generalization of local views, giving the further possibility to organize them in hierarchies. The most evident drawback, however, is that all available sources must be considered in defining the GAV, thus it is difficult to add a new source.

If, on the other hand, a LAV approach is used, queries on the global schema should be rewritten at each peer, using rules that, using the datalog formalism as above, look as follows:

$$S_1(title, dir, year) \subseteq Movie(title, dir, year, \_)$$
$$S_2(cinema, title, dir, genre) \subseteq Schedule(cinema, title), Movie(title, dir, \_, genre)$$

This approach is more flexible, because there is no need to know all data sources in advance to create the mediated schema. Moreover, the possibility is given to distinguish between contents of closely related sources: if, for example, two sources have similar content, they will have a similar translation with respect to the mediated schema. LAV techniques, however, require query rewriting, as explained in [40], thus the query cannot be simply unfolded using stored rules. The data sources can be interpreted as materialized views over the mediated schema: the query processing algorithm first chooses the best views that are able to answer the query; then, it translates the query with respect to the so-obtained views. This is a typical issue of database design and is encountered in different fields, like query optimization, data integration, and data warehousing. However, it is still of great interest in modern peer-to-peer systems, since peers can be modeled as remote data sources characterized by a local schema. Thus, accepting that a global schema is not obtainable due to the dynamic nature of the network, the problem of building a mediated schema at a peer which summarizes just a small portion of the network, i.e., its neighborhood, is still an issue. This approach leads to the definition of mappings between different peers, called *semantic mappings* [41]. As it is explained in [42, 75], such a scenario is different (more complex) from a traditional peer-to-peer system, since it assumes that peers have some kind of "semantic awareness" and that they share a common knowledge of certain semantic relations that can be established between them.

Query processing in this context consists of two tasks:

1. the peer where the query has been originated chooses the best schemata that can answer the query, thus ranking neighbors according to their relevance, and
2. the query is "reformulated" according to the remote schemata using the semantic mappings.

In this way, the problem of query processing "can be viewed as a search through a space of reformulations" [75].

In the literature, two main approaches to the construction of semantic mappings in peer-to-peer systems are present: the *instance-based* approach and the *schema-based* approach (for a more comprehensive survey on the problem of schema matching, see [66]):

- Instance-based mappings are created by considering the views offered by two peers on a common set of instances S; the measure of "overlap" of the peers' views can be used to assess the similarity between concepts in the two schemata.
- Schema-based mappings, on the other hand, consider a common knowledge that is used to "reconcile" the two schemata. Schema-based mappings, in fact, provide, for each concept of a schema, an associated meaning that comes from a common, shared knowledge on a particular domain, and is thus independent from the instances that may or may not be classified under it. This is a very strong hypothesis, since it implies a relation between two concepts being dictated not just by the fact that they identify the same objects (extensional description), but because they share structural and semantical analogies going beyond the simple observation (intensional description).

The main practical difference between instance-based and schema-based approaches is that, in the first case, mappings characterize the "likeness" of concepts without, however, specifying what kind of semantic relations can be established between them. From this point of view, instance-based mappings are considered semantically poor. We can further observe that in the former case links are bidirectional, while in the latter we cannot assume that, for any chain of translations that leads from a concept to another one, a corresponding back-path exists (this should be explicitly inferred by the algorithm and cannot be obtained as a simple "inverse function").

3.3.1 Instance-Based Mappings

Among *instance-based* mappings, the GLUE approach, presented in [27] tackles the problem of ontology matching: each concept in the ontology is modeled as the set of instances belonging to the data source to which the ontology is referred. In particular, for matching concepts obtained from two different ontologies, say O_1 and O_2, a *similarity matrix* is built, where each element represents the similarity between a concept $A \in O_1$ and a concept $B \in O_2$ (and vice-versa). Such similarity is assessed by way of the Jaccard Coefficient between the two concepts:

$$sim_J(A, B) = \frac{\Pr(A \cap B)}{\Pr(A \cup B)} = \frac{\Pr(A, B)}{\Pr(A, B) + \Pr(A, \overline{B}) + \Pr(\overline{A}, B)} \tag{2}$$

where $\Pr(A, B)$ represents the probability that an object randomly chosen among the universe of all data instances can be classified as belonging to both A and B, and $\overline{A}$ denotes the complement of concept A. It has, however, to be noted that in [27] only taxonomies are considered, and not general ontologies, thus the generality of this approach is quite limited.

3.3.2 Schema-Based Mappings

CtxMatch An example of schema-based semantic mappings can be found in [14]. Here, mappings represent semantic relations between concepts: the CtxMatch technique proposed in [14] uses WordNet as a common thesaurus and ontologies are expressed as hierarchical classifications of concepts, again limiting the applicability of this solution. The relations that are automatically inferred by CtxMatch between concepts of ontologies are: $\equiv$ (equivalence), $\subset$ (subset), $\supset$ (superset), $*$ (partial overlap), $\perp$ (exclusion). This is done using *semantic explicitation*, i.e., the explicit semantic meaning of each node in the hierarchies is provided by using a linguistic interpretation (that exploits the thesaurus) and a contextualization (that takes into account the domain and structural knowledge about the ontologies). Finally, the existence of relations between concepts of the ontologies is formulated as a problem of propositional satisfiability that is solved using standard algorithms.

Piazza In [42, 41], the difference is highlighted between *stored* and *peer* schemata: *stored schemata* identify the relations (or the documents) that are currently stored in each peer, while *peer schemata* can be used to express queries and to establish semantic mappings. This separation means that a peer might be able to understand a query even if it cannot answer it: this is, indeed, a very attractive feature in a decentralized system, since the most difficult problem is to discover a semantic path for translating the original query to the schema of the answering peer.

In [41], the Piazza framework is presented. Piazza is an example of a Peer Database Management System (PDMS), since it consists of a set of peers (data sources) linked by a network of semantic mappings. Each peer schema is described using the XML Schema notation and the query language is a modified version of XQuery [17]. Query processing in Piazza relies on query reformulation and is based on two different kinds of mappings: the first type includes relations of *inclusion* and *equality* between peer schemata, while the second type consists of *Datalog rules*,

composed by a head and a body, that are used to map relations of a peer to relations of another peer. Note that, since all mappings are available at a single location, e.g., a centralized server, reformulation can be done at the node generating the query, without the need of distributing this burden to other peers in the network. Intuitively, this reduces to the problem of answering queries using views [40], with an additional complexity due to the fact that peer schemata are different from stored schemata, thus the problem of which peers, among those constituting the semantic path, might be useful to answer the query is an issue.

In [75], the problem of query processing in Piazza is further investigated: the plan generated for a query by the peer where the query originated consists of a tree whose nodes correspond to reformulations. During the plan generation process, the most important issues consist in pruning redundant nodes and minimizing the number of reformulations. In the exploration phase, on the other hand, the problem is to choose the nodes that need to be evaluated first: it is observed that a depth-first search strategy is less efficient than a breadth-first visit, because nodes that are distant from the tree root are also semantically dissimilar from the original query and, thus, provide less significant answers.

PeerDB In [64], the PeerDB peer-to-peer distributed data management system is presented. Essentially, each PeerDB node stores some data in a local DBMS, and users are able to express SQL queries over such data. To solve a query, every peer can also ask data to other network peers. This is performed in two phases:

1. first, the query is forwarded (using *relation matching agents*) to neighboring peers in the network (using a TTL to reduce flooding) to ask for schemata that are similar to the query schema;

2. the peers contacted by the relation matching agents return their best matching relations to the peer originating the query, so that the user can choose which relations are to be queried; for each peer chosen by the user, a *data retrieval agent* is forwarded to that peer to retrieve the data.

It has to be noted that the two phases can be interleaved, so that data retrieving can be initiated as soon as a few relations have been selected as interesting by the user.

Schema mapping in PeerDB is based on metadata: for each relation name and each attribute in a given peer p, the administrator of p provides a number of keywords describing the semantics of that relation or attribute. In this way, the relation matching agent at peer p can establish the similarity between the schema of p and the schema of the original query, by selecting those relations and attributes that share keywords with the query (this might also involve the use of thesauri). In PeerDB, hence, the mappings between peers' schemata are not stored, but are computed upon request by the relation matching agents using keyword matching, and are manually evaluated by the user when she receives the results from each agent, thus the mappings are not generated in a (semi-)automatic way. This approach, of course, requires the mappings to be re-created for each query, thus it cannot reuse mappings already computed for past queries; moreover, since the rewriting is performed independently by each peer, the results obtained by different peers cannot be "joined" (this would require a rewriting to be performed at the peer originating the query, because nodes store no semantic information about other peers).

4 Relevance-Based Search

Solutions presented in previous sections only consider that the result of a query is a set of objects. Under this assumption, the problem of discovering an appropriate answer to a query is equivalent to the problem of finding the objects which "independently" satisfy the query constraints. In this section, we consider a different approach to the query processing problem,

where the relevance of data instances with respect to the query is evaluated by means of a *score* (i.e., a numeric value measuring how much each object "satisfies" the actual query), inducing a linear order on the underlying objects: the user, thus, expects to receive as answer a list of objects in decreasing order of relevance. We refer to this problem as *relevance*-based search, and, in the following, we analyze state-of-the-art solutions for this problem by classifying them in two main categories:

IR-style Approaches: Information retrieval (IR)-style techniques borrow tools and techniques from IR, extending them to a distributed scenario. The data collection is composed by "atomic" objects (i.e., objects with just one component) which are spread over the network (objects are thus *horizontally partitioned* among the peers). Given a query, each object comes with a (global) rank, computed according to some scoring function, and the problem is how to obtain the complete answer to the query in the most efficient way.

Top-k Approaches: Here, a single object can be found in different peers that only store separate properties or components of it (this is also called *vertical partitioning*), and the problem is to retrieve only the most relevant ("best") objects as fast as possible. Each object thus consists of different components and each component can be obtained from a data source along with a *local* rank of relevance with respect to a partial query (this is the fragment of query answered by that data source). By means of an integration process (named *object fusion*), for each object the corresponding *global* rank is assessed, starting from its local scores; in this way, it is then possible to compute which are the top-k objects existing in the network. The more general case where objects are replicated in different data sources and are ranked with respect to distinct criteria should be also contemplated.

4.1 IR-style Approaches

In the classical IR scenario, each object is a text document and is characterized by a set of keywords which are (usually) automatically extracted from the document itself. The set of all keywords of the collection defines a multi-dimensional vector space where each document is represented by a point, whose coordinate values depend on the frequency of each keyword in the document [70]. The usual approach to compute such coordinates is based on the *Term Frequency* and *Inverse Document Frequency* ($tf \times idf$). In details, for each document d, the frequencies ($tf_{w,d}$) of all terms w contained in d are computed, representing the document relevance for each keyword; for each keyword w, the fraction of documents in the collection containing w (idf_w) is also computed, representing the relative importance of w in the collection (unfrequent terms are given a higher weight). According to this model, two documents are similar when the corresponding vectors are close in the vector space defined by their keywords. To assess the similarity between vectors, usually the cosine of the angle between them is computed.

Though the $tf \times idf$ approach is effective when applied to a traditional scenario, it presents several problems for a decentralized environment such as a peer-to-peer network, since it relies on "global" properties (the *idf*s), that depend on all documents in the network and thus cannot be computed independently by single peers which only have a "local" knowledge. To solve this problem, a very simple solution is to consider external servers able to process all documents in the network and to compute all global properties [34]: at query time, each peer will retrieve such statistics directly from the servers. This strategy clearly imposes the use of centralized servers, thus inheriting all the limitations of a client/server architecture. A more efficient solution consists in computing the *idf* values on demand [25, 54, 60]: the main idea is to propagate local knowledge of each peer over the network (e.g., by means of a *gossiping* algorithm) in order to allow other peers to accurately estimate the actual *idf* values. It has to be noted that this technique presents some conservative properties, since the computed values are more accurate when the network has little dynamicity and they become exact only if the network is static.

Different solutions using above statistics have been proposed to compute the query result. In [60], a super-peer network with a HyperCuP topology is assumed (see Sections 2.1.2 and 2.2.1): given a query, the super-peers compute the updated *idf* values through gossiping, and such values are routed to individual peers; each peer is then able to compute a local ranking of documents and to return it to its super-peer; finally, results are merged and ranked again at a super-peer level and routed to the query originator.

PlanetP [25] assumes a completely unstructured network: to avoid storing the *idf* values in a global index, the inverse peer frequency ipf is computed for each term w, as the number of peers having at least a document containing w. Given a query requesting for the best k documents, peers are first ranked according to their likelihood of having relevant documents, e.g., by summing the ipfs for all query terms, and the number m of peers that need to be contacted to answer the query is established. The peers then compute the best documents locally, using ipf values in place of the idfs. The documents returned by selected peers are finally globally ranked to obtain the final answer. This only requires that each peer estimates the ipf values, and this is done by gossiping in a simple way, allowing a lower number of messages when compared to the amount necessary to keep updated *idf* values.

4.1.1 The Ranking Framework

As a relevant extension of the basic IR-style query processing approach, we refer to the work presented in [5, 6] where a ranking algebra as a formal framework for ranking computation is proposed. In particular, multiple ranking criteria for the global ranking process are provided. This allows to share different interpretations of the peers content by supporting personalized and context-dependent searches. In such scenario, a common architecture is assumed to enable the interoperability and the sharing of resources, whereas a common logical framework is used to allow peers to represent different ranking outputs and to apply composition operators to them. Ranking is considered as a first citizen class concept and it is characterized by means of different types of metadata which represent:

1. the type of ranking (e.g., text-based vs. link-based model),
2. the operations (along with their interfaces) applicable to it,
3. the scope of the ranking (complete ranking vs. local ranking), and
4. its quality (e.g., when the validity of the ranking expires).

The different rankings can be combined to satisfy different information needs, manipulated and compared by means of a ranking algebra providing both basic operations (e.g., selection and projection) and ranking operations (e.g., normalization and weighted combination of rankings). Such framework would allow to overcome the main limitations of global ranking algorithms, like scalability for the billions of web documents, instability of ranking algorithms, dynamicity of web content (which influences global ranking) connected to the latency of global indices, high computation costs, and so on.

4.2 Top-k Query Processing

The top-k retrieval paradigm was first introduced in the database domain with the goal of retrieving the best k database objects with respect to a query, by minimizing the number of objects to be accessed, i.e., compared with the query. In particular, "middleware" algorithms [29, 31, 30] have been proposed for top-k queries, for scenarios where the best k objects are retrieved given the (partial) descriptions provided for such object by m distinct data sources. Such algorithms are important in our context, since peer-to-peer networks involve different data sources (one for each peer). Indeed, in the peer-to-peer scenario, different features of the same object can be

stored in distinct peers, thus the problem of reconstructing the overall top-k objects starting from m ranked lists independently provided by individual peers is an issue.

Here, we summarize the usual assumptions made in the database domain on data sources to characterize the involved scenario:

Single Dataset: all data sources manage a same set of objects.

Object Identifiers: each object has a "global" ID across all data sources. This ensures that each object is uniformly identifiable by each data source and by the middleware algorithm that computes the global ranking.

Ranked List of Results: given a query, each data source is able to return a ranked list of results, sorted according to the "local" similarity criterion in non-increasing order of relevance with respect to the query. More precisely, each entry in the ranked list contains, at least, the object ID and a score that numerically assesses in which measure the object matches the query on the data source. Such score is also called *local* (or *partial*) score.

Sorted Access Interface: the ranked list is provided by each data source through a sorted access, that provides, at each call, the next best object according to the local score.

Random Access Interface: each source is able, upon request, to provide the local score of an object, given its ID.

A number of different algorithms, sharing above assumptions, have been proposed for databases [29, 31, 30], Web searches [15], and multimedia retrieval [18, 10]. Basically, they differ in considering distinct query predicates concerning specific characteristics of the objects, and in the aggregation modality used to obtain the global score from partial scores. In particular, early approaches [29, 31, 15, 30] use a numerical scoring function (e.g., avg, min, and max) to compute a global score for ordering the result list. Recently, more general solutions using qualitative preferences (instead of scoring functions) have been presented [10]. Here, arbitrary partial (rather than only linear) orders on the objects are defined, so that a large flexibility is gained in representing what the user is looking for.

Only recently, top-k queries have been investigated in the peer-to-peer scenario, where they assume a high relevance due to the potentially large number of peers that can be involved in the search process. However, up to now only a few peer-to-peer-based top-k query algorithms have been proposed. In the following section, we will survey available solutions and approaches that, even when proposed for other domains, could be useful for our purposes.

Distributed Top-k Retrieval Algorithm In [9], an algorithm for progressive distributed top-k retrieval in peer-to-peer networks is presented, assuming a super-peer HyperCuP topology of the network (see Section 2.2.1). The main goal of the proposed solution is to minimize the data traffic over the network using dynamically collected query statistics by locally evaluating as many parts of the query as possible. The main assumption of the authors is that all peers are collaborative and provide objects with normalized scores that can be compared to evaluate the quality between different objects. In particular, each data source contains a subset of documents in the network and the same document can be independently provided by several data sources; moreover, random access is not available.

In details, each super-peer stores a local index containing names of peers in its cluster and of neighboring super-peers that have contributed to results of recent queries. The internal index is used for routing optimization (i.e., to avoid non-relevant destinations). To solve a top-k query originated at peer p, 4 structures are maintained at the corresponding super-peer s:

- For each peer that received the query, *TopRes* maintains the object ID-score pair relative to the next best result obtained from that peer;

- *BestPeers* contains the identifiers of super-peers that have contributed to the result of the current query;
- *RequestResults* stores all peers that were asked to return a result;
- *Delivered* represents the result set, where top-k objects are added.

The main idea of the algorithm is inspired by the $\mathcal{B}_0$ middleware algorithm [29], that represents the pioneering solution for top-k queries based on scoring functions, and that uses the max function to aggregate local scores and produce the overall score for each object. The basic rationale is very simple and consists in executing k sorted accesses for each data source. Among the so obtained objects (which are no more than $m \times k$), the best k objects (i.e., the ones with maximum score) are in the overall result. In details, the super-peer s:

1. initializes the *TopRes* structure and executes a sorted access on all involved super-peers;
2. updates *RequestResults* with the super-peers for which a request has been delivered and inserts the results in *TopRes*, until all peers have provided an answer;
3. the best objects in *TopRes* are removed and inserted in *Delivered*;
4. the process is repeated until k results are in the result list;
5. the *BestPeers* list is used to update the local index of the super-peer.

The algorithm is also able to deliver objects in an incremental way, so that the user can explore some preliminary best results before all the top-k objects have been found.

In [9], the problem to be solved is the replication of objects over different data sources: this is simpler with respect to the general scenario described in Section 4.2. In this case, in fact, when the same object ID is seen at different peers, the copy with maximum score can be only considered; this is exactly the requirement for the simple $\mathcal{B}_0$ algorithm [29]. In order to deal with the more general scenario arising when integration of complex objects is needed, i.e., when distinct peers provide partial descriptions of a same object, $\mathcal{B}_0$ is however no longer appropriate, mainly because the global score of each object is computed by taking into account the contribution of only one component (the one with maximum score): if the object has a very high partial score at one peer and a very low score at another peer, it is still considered to be a very good object. On the other hand, in the general scenario we are interested in, scores of object's components have to be integrated to obtain a global score by considering that the "completeness" of object descriptions is no longer assured, due to the the possible existence of "null" values for both properties and instances (e.g., at query time, the peer that is in charge of answering the sub-query on a particular attribute could be down or could not be able to return the partial score for a particular object because the object itself is not included among its data). To solve this problem, more effective and flexible integration solutions are needed: to this end, advanced integration algorithms could be profitably considered, that are able to take into account all object properties, and rely either on numerical scoring functions (such as *avg* and *median*), e.g., $\mathcal{A}_0$ [29], TA [31], and MEDRANK [30], or on more general qualitative preference relations, e.g., iMPO [10].

Upper An interesting step towards the solution of the integration problem is proposed in [15], where an efficient algorithm (named Upper) is presented to process top-k queries over Web-accessible databases, where it is assumed that exactly one source (named S-Source) provides a sorted list of objects ranked according to a particular attribute, and multiple sources (R-Sources) are only able to return scoring information about individual objects upon request. At each step, the Upper algorithm chooses both the best object (from the S-Source) and the best attribute

(among the R-Sources) that have to be requested. The basic idea here is that, in general, it is not necessary to probe all attributes for each considered object, but only those needed to find the top-k answers.

Probabilistic Top-k In [76], a probabilistic model for top-k query evaluation is presented. The notion of top-k query is relaxed into the concept of approximation such that the query processor can tolerate errors with probabilistic guarantees with respect to the exact answer [20]. In details, the total score is predicted for objects for which a partial score is already known: by computing the probability that such global score exceeds a certain threshold that makes the object relevant to the final result, a drastic reduction of data accesses can be ensured. The flexible framework for distributed top-k queries presented in [58] has the same goal: in particular, here the problem is addressed by considering that index lists for query attributes (which are essential for the assumed TA-like top-k approach) are spread across peers. The integrated top-k algorithms guarantee efficient results with good-enough quality by applying probabilistic approximations to the exact result. Such relaxation are justified by the need for peer-to-peer systems to trade-off execution cost and search quality.

NRA-RJ In [45], the problem of incremental joins of multiple ranked data sets for the relational database domain is investigated by considering the case where the join condition is a list of user-defined predicates on input tuples. Although the reference scenario is that of centralized databases, this work is very relevant for top-k queries in peer-to-peer networks, since one of the proposed algorithms, namely J^*, considers the join of multiple sets of different objects under arbitrary join constraints (i.e., the mapping is of type $n - m$); this contrasts with the typical approach of joining multiple sets of same objects applied by traditional algorithms (like TA [31] and iMPO [10]), where the mapping is $1 - 1$. Thus, J^* represents a more general solution that includes, as special cases, algorithms proposed for the middleware scenario.

IncrementalFD In [22], an incremental algorithm for computing ranked full disjunctions is proposed. The full disjunction is a natural associative extension of the binary outer-join operator to an arbitrary number of relations. In particular, it maximally combines tuples from connected relations, while preserving all information in relations: this represents a very useful property for the information integration problem, that aims to combine data coming from different sources.

The basic algorithm (named IncrementalFD) computes the full disjunction (FD) of a set of relations, $R_1, \ldots, R_n$, in an incremental way, by calculating the subsets $FD_i \subseteq FD$, for each $i \in [1, n]$, that contain the tuples of FD for which a tuple from R_i exists. To compute each FD_i, IncrementalFD uses two linked lists (named *Complete* and *Incomplete*): *Complete* is initially empty and will contain the result set FD_i, whereas *Incomplete* contains the tuples considered at each step by the algorithm and is initialized with all tuples from R_i. At each step, IncrementalFD extracts a tuple t_i from *Incomplete* and checks if it is possible to join it with additional tuples obtained from other relations. Here, a new tuple t is added to *Incomplete* if t is "join consistent" with the tuple t_i previously considered, i.e., t and t_i should share some (not null) value on all common attributes; if a join consistent tuple is obtained that cannot be further extended, it is added to *Complete* and can be immediately returned. The algorithm stops as soon as *Incomplete* becomes empty.

Since IncrementalFD can return results in an incremental way, it is of interest to understand under which conditions the algorithm can be modified so as to efficiently compute the top-k tuples according to a specific ranking function. It is demonstrated that in the general case the problem is NP-hard (exponential in the number of relations to be joined), whereas it can be solved in polynomial time when the value of the ranking function can be determined by looking at most at a constant number c of tuple components.

5 Conclusions

The analysis of solutions available for processing queries in heterogeneous scenarios shows that none of them can be directly implemented in the WISDOM project. This basically follows from the observation that WISDOM needs novel query processing techniques able to satisfy *both* the following requirements:

1. They should work by exploiting semantic mappings and without having a global (shared) knowledge of all the available mappings in the network. This makes the WISDOM approach quite different from that of, say, Piazza (see Section 3.3.2).

2. They should take into account several aspects related to the *relevance* of both peers and objects.

Thus, neither solutions developed for schema-based queries (but with no attention to relevance issues) nor solutions proposed for relevance-based query processing (but with a minor emphasis on semantic issues) are appropriate.

Consequently, a successful solution should properly make use of (at least) the following basic "ingredients":

Relevance Assessment: a preliminary step towards query evaluation is to properly understand and model the various aspects that, in the WISDOM scenario, can influence the relevance of both peers and objects. In particular, it has to be understood what it means for a peer to "fit" better a query request, how several contrasting factors should be reconciled (e.g., "quality" vs. "completeness" of results, response time vs. "coverage", etc.) and which ranking criteria are appropriate in the WISDOM context.

Content Summaries: relevance assessment necessarily needs some form of content summaries in order to allow more accurate choices. Thus, it is important to understand how content summaries should be built and structured within the WISDOM scenario in order to select the most relevant peers and, by acting as routing indices, to direct the search to the most relevant part(s) of the network.

Starting from these ingredients, effort should then be put in deriving efficient query evaluation strategies, able to return (possibly incrementally) the "best" objects satisfying the query.

References

[1] The Gnutella project. `http://www.gnutella.com` (valid as in June 2005).

[2] The iMesh website. `http://imesh.com` (valid as in June 2005).

[3] The KaZaA website. `http://www.kazaa.com` (valid as in June 2005).

[4] Karl Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *Proceedings of the 9th International Conference on Cooperative Information Systems (CoopIS 2001)*, volume 2172 of *Lecture Notes in Computer Science*, pages 179–194, Trento, Italy, September 2001. Springer.

[5] Karl Aberer and Jie Wu. A framework for decentralized ranking in web information retrieval. In *Proceedings of the 5th Asia-Pacific Web Conference on Advanced Web Technologies and Applications (APWeb 2003)*, volume 2642 of *Lecture Notes in Computer Science*, pages 213–226, Xian, China, April 2003. Springer.

[6] Karl Aberer and Jie Wu. Towards a common framework for peer-to-peer web retrieval. In Matthias Hemmje, Claudia Niederée, and Thomas Risse, editors, *From Integrated Publication and Information Systems to Virtual Information and Knowledge Environments*, volume 3379 of *Lecture Notes in Computer Science*, pages 138–151. Springer, 2005.

[7] Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani, and Bernardo A. Huberman. Search in power-law networks. *Physical Review E*, 64(046135), October 2001.

[8] Association for Computer Machinery. ACM computing classification system. `http://www.acm.org/class/1998/` (valid as in June 2005).

[9] Wolf-Tilo Balke, Wolfgang Nejdl, Wolf Siberski, and Uwe Thaden. Progressive distributed top k retrieval in peer-to-peer networks. In *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, pages 174–185, Tokyo, Japan, April 2005. IEEE Computer Society.

[10] Ilaria Bartolini, Paolo Ciaccia, Vincent Oria, and M. Tamer Özsu. Flexible integration of multimedia sub-queries with qualitative preferences. To appear in Multimedia Tools and Applications Journal.

[11] Mathias Bender, Sebastian Michel, Peter Triantafillou, Gerhard Weikum, and Christian Zimmer. Improving collection selection with overlap-awareness. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Salvador, Brazil, August 2005. To appear.

[12] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[13] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *ACM Computing Surveys*, 13(7):422–426, July 1970.

[14] Paolo Bouquet, Luciano Serafini, and Stefano Zanobini. Peer-to-peer semantic coordination. *Journal of Web Semantics*, 22(1):81–97, December 2004.

[15] Nicolas Bruno, Luis Gravano, and Amélia Marian. Evaluating top-k queries over web-accessible databases. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, pages 369–382, San Jose, California, USA, February 2002. IEEE Computer Society.

[16] Jamie Callan. Distributed information retrieval. In W. Bruce Croft, editor, *Advances in Information Retrieval*, chapter 5, pages 127–150. Kluwer Academic, 2000.

[17] Donald D. Chamberlin, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery: A query language for XML, 2001. `http://www.w3.org/TR/xquery` (valid as in June 2005).

[18] Surajit Chaudhuri and Luis Gravano. Optimizing queries over multimedia repositories. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 91–102, Montreal, QC, Canada, June 1996. ACM Press.

[19] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P systems scalable. In *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 407–418, Karlsruhe, Germany, August 2003. ACM Press.

[20] Paolo Ciaccia and Marco Patella. PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Proceedings of the 16th International Conference on Data Engineering (ICDE 2000)*, pages 244–255, San Diego, CA, March 2000. IEEE Computer Society.

[21] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*, pages 46–66, Berkeley, CA, July 2000. Springer.

[22] Sara Cohen and Yehoshua Sagiv. An incremental algorithm for computing ranked full disjunctions. In *Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'05)*, Baltimore, MD, June 2005. To appear.

[23] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using P-trees. In *Proceedings of the 7th International Workshop on the Web and Databases (WebDB'04)*, Paris, France, June 2004.

[24] Arturo Crespo and Hector Garcia-Molina. Routing indices for peer-to-peer systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, pages 23–32, Vienna, Austria, July 2002. IEEE Computer Society. Online publication.

[25] Francisco Matias Cuenca-Acuna, Christopher Peery, Richard P. Martin, and Thu D. Nguyen. PlanetP: Using gossiping to build content addressable peer-to-peer information sharing communities. In *Proceedings of the 12th International Symposium on High-Performance Distributed Computing (HPDC-12 2003)*, pages 236–249, Seattle, WA, June 2003. IEEE Computer Society.

[26] Scott Deerwester, Susan Dumais, George Furnas, Thomas Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[27] AnHai Doan, Jayant Madhavan, Robin Dhamankar, Pedro Domingos, and Alon Y. Halevy. Learning to match ontologies on the semantic web. *The VLDB Journal*, 12(4):303–319, November 2003.

[28] Peter Druschel and Antony I. T. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 75–80, Elmau/Oberbayern, Germany, May 2001. IEEE Computer Society.

[29] Ronald Fagin. Combining fuzzy information from multiple systems. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'96)*, pages 216–226, Montreal, Canada, June 1996. ACM Press.

[30] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 301–312, San Diego, CA, June 2003. ACM Press.

[31] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'01)*, pages 216–226, Santa Barbara, CA, May 2001. ACM Press.

[32] Leonidas Galanis, Yuan Wang, Shawn R. Jeffery, and David J. DeWitt. Processing queries in a large peer-to-peer system. In *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE 2003)*, volume 2681 of *Lecture Notes in Computer Science*, pages 273–288, Klagenfurt, Austria, June 2003. Springer.

[33] Luis Gravano, Kevin Chen-Chuan Chang, Hector Garcia-Molina, and Andreas Paepcke. STARTS: Stanford proposal for internet meta-searching (experience paper). In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 207–218, Tucson, AZ, May 1997. ACM Press.

[34] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. The effectiveness of GlOSS for the text database discovery problem. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 126–137, Minneapolis, MI, May 1994. ACM Press.

[35] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. GlOSS: Text-source discovery over the internet. *ACM Transactions on Database Systems*, 24(2):229–264, June 1999.

[36] Luis Gravano, Panagiotis G. Ipeirotis, and Mehran Sahami. QProber: A system for automatic classification of hidden-web databases. *ACM Transactions on Information Systems*, 21(1):1–41, January 2003.

[37] P. Krishna Gummadi, Ramakrishna Gummadi, Steven D. Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 381–394, Karlsruhe, Germany, August 2003. ACM Press.

[38] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, CA, January 2003. Online proceedings.

[39] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, June 1984. ACM Press.

[40] Alon Y. Halevy. Answering queries using views: a survey. *The VLDB Journal*, 10(4):270–294, 2001.

[41] Alon Y. Halevy, Zachary G. Ives, Jayant Madhavan, Peter Mork, Dan Suciu, and Igor Tatarinov. The Piazza peer data management system. *IEEE Transactions on Knowledge and Data Engineering*, 18(7):787–798, July 2004.

[42] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *Proceedings of the 19th International Conference on Data Engineering (ICDE 2003)*, pages 505–516, Bangalore, India, March 2003. ACM Press.

[43] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, volume 2429 of *Lecture Notes in Computer Science*, pages 242–259, Cambridge, MA, March 2002. Springer.

[44] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the internet with PIER. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)*, pages 321–332, Berlin, Germany, September 2003. Morgan Kaufmann.

[45] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Joining ranked inputs in practice. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 950–961, Toronto, Canada, August 2002. Morgan Kaufmann.

[46] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC'98)*, pages 604–613, Dallas, TX, May 1998. ACM Press.

[47] Panagiotis G. Ipeirotis and Luis Gravano. Distributed search over the hidden web: Hierarchical database sampling and selection. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 394–405, Hong Kong, China, August 2002. Morgan Kaufmann.

[48] Vana Kalogeraki, Dimitios Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management*, pages 300–307, McLean, VA, November 2002. ACM Press.

[49] Georgia Koloniari and Evaggelia Pitoura. Content-based routing of path queries in peer-to-peer systems. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT 2004)*, pages 29–47, Heraklion, Greece, March 2004. Springer.

[50] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, December 2000.

[51] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'96)*, pages 233–246, Madison, WI, June 2002. ACM Press.

[52] Jinyang Li, Boon Thau Loo, Joseph M. Hellerstein, M. Frans Kaashoek, David R. Karger, and Robert Morris. On the feasibility of peer-to-peer web indexing and search. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS 2003)*, volume 2735 of *Lecture Notes in Computer Science*, pages 207–215, Berkeley, CA, February 2003. Springer.

[53] Jie Lu and Jamie Callan. Content-based retrieval in hybrid peer-to-peer networks. In *Proceedings of the 2003 ACM CIKM International Conference on Information and Knowledge Management*, pages 199–206, New Orleans, LA, November 2003. ACM Press.

[54] Zhiguo Lu, Bo Ling, Weining Qian, Wee Siong Ng, and Aoying Zhou. A distributed ranking strategy in peer-to-peer based information retrieval systems. In *Proceedings of the 6th Asia-Pacific Web Conference on Advanced Web Technologies and Applications (APWeb 2004)*, volume 3007 of *Lecture Notes in Computer Science*, pages 279–284, Hangzhou, China, April 2004. Springer.

[55] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 2002 International Conference on Supercomputing (ICS 2002)*, pages 84–95, New York, NY, June 2002. ACM Press.

[56] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 183–192, Monterey, CA, July 2002. ACM Press.

[57] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65, Cambridge, MA, March 2002. Springer.

[58] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. KLEE: A framework for distributed top-k query algorithms. In *Proceedings of the 31th International Conference on Very Large Data Bases (VLDB 2005)*, Trondheim, Norway, September 2005. To appear.

[59] Anirban Mondal, Yi Lifu, and Masaru Kitsuregawa. P2PR-tree: An R-tree-based spatial index for peer-to-peer environments. In *Proceedings of the EDBT 2004 Workshop on Peer-to-Peer Computing and Databases (P2P&DB)*, volume 3268 of *Lecture Notes in Computer Science*, pages 516–525, Berlin, Germany, March 2004. Springer.

[60] Wolfgang Nejdl, Wolf Siberski, Uwe Thaden, and Wolf-Tilo Balke. Top-k query evaluation for schema-based peer-to-peer networks. In *Proceedings of the 3rd International Semantic Web Conference (ISWC 2004)*, volume 3298 of *Lecture Notes in Computer Science*, pages 137–151, Hiroshima, Japan, November 2004. Springer.

[61] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In *Proceedings of the 11th International World Wide Web Conference (WWW 2002)*, pages 604–615, Honolulu, HI, May 2002. ACM Press.

[62] Wolfgang Nejdl, Martin Wolpers, Wolf Siberski, Christoph Schmitz, Mario T. Schlosser, Ingo Brunkhorst, and Alexander Löser. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In *Proceedings of the 12th International World Wide Web Conference (WWW 2003)*, pages 536–543, Budapest, Hungary, May 2003. ACM Press.

[63] Wolfgang Nejdl, Martin Wolpers, Wolf Siberski, Christoph Schmitz, Mario T. Schlosser, Ingo Brunkhorst, and Alexander Löser. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. *Journal of Web Semantics*, 1(2):177–186, February 2004.

[64] Wee Siong Ng, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proceedings of the 19th International Conference on Data Engineering (ICDE 2003)*, pages 633–644, Bangalore, India, March 2003. IEEE Computer Society.

[65] Yannis Petrakis, Georgia Koloniari, and Evaggelia Pitoura. On using histograms as routing indexes in peer-to-peer systems. In *Proceedings of the 2nd International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P 2004)*, volume 2992 of *Lecture Notes in Computer Science*, pages 16–30, Toronto, QC, August 2004. Springer.

[66] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[67] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 161–172, San Diego, CA, August 2001. ACM Press.

[68] Sean C. Rhea and John Kubiatowicz. Probabilistic location and routing. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, volume 3, pages 1248–1257, New York, NY, June 2002. IEEE.

[69] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01)*, pages 329–350, November 2001.

[70] Gerard Salton. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, Reading, MA, 1989.

[71] Mario T. Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. HyperCuP - hypercubes, ontologies, and efficient search on peer-to-peer networks. In *Proceedings of the 1st International Workshop on Agents and Peer-to-Peer Computing (AP2PC 2002)*, volume 3367 of *Lecture Notes in Computer Science*, pages 112–124, Bologna, Italy, July 2002. Springer.

[72] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB'87)*, pages 507–518, Brighton, England, September 1987. Morgan Kaufmann.

[73] Heng Tao Shen, Yanfeng Shu, and Bei Yu. Efficient semantic-based content search in P2P network. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):813–826, July 2004.

[74] Ian Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, San Diego, CA, August 2001. ACM Press.

[75] Igor Tatarinov and Alon Y. Halevy. Efficient query reformulation in peer data management systems. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 539–550, Paris, France, June 2004. ACM Press.

[76] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004)*, pages 648–659, Toronto, Canada, August 2004. Morgan Kaufmann.

[77] Dimitrios Tsoumakos and Nick Roussopoulos. Adaptive probabilistic search for peer-to-peer networks. In *Proceedings of the 3rd International Conference on Peer-to-Peer Computing (P2P 2003)*, pages 102–109, Linköping, Sweden, September 2003. IEEE Computer Society.

[78] Chaokun Wang, Jianzhong Li, and Shengfei Shi. Cell abstract indices for content-based approximate query processing in structured peer-to-peer data systems. In *Proceedings of the 6th Asia-Pacific Web Conference on Advanced Web Technologies and Applications (APWeb 2004)*, volume 3007 of *Lecture Notes in Computer Science*, pages 269–278, Hangzhou, China, April 2004. Springer.

[79] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB'98)*, pages 194–205, New York City, NY, August 1998. Morgan Kaufmann.

[80] World Wide Web Consortium. Resource description framework (RDF): Concepts and abstract syntax, February 2004. `http://www.w3.org/TR/rdf-concepts/` (valid as in June 2005).

[81] Linhao Xu, Chenyun Dai, Wenyuan Cai, Shuigeng Zhou, and Aoying Zhou. Towards adaptive probabilistic search in unstructured P2P systems. In *Proceedings of the 6th Asia-Pacific Web Conference on Advanced Web Technologies and Applications (APWeb 2004)*, volume 3007 of *Lecture Notes in Computer Science*, pages 258–268, Hangzhou, China, April 2004. Springer.

[82] Beverly Yang and Hector Garcia-Molina. Improving search in peer-to-peer networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, pages 5–14, Vienna, Austria, July 2002. IEEE Computer Society. Online publication.

[83] Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. In *Proceedings of the 19th International Conference on Data Engineering (ICDE 2003)*, pages 49–60, Bangalore, India, March 2003. IEEE Computer Society.

[84] D. Zeinalipour-Yazti, Vana Kalogeraki, and Dimitrios Gunopulos. Exploiting locality for scalable information retrieval in peer-to-peer networks. *Information Systems*, 30(4):277–298, June 2005.

[85] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.