Overlay Networks and Query Processing: A Survey[*]

Georgia Koloniari Nikos Kremmidas Kostas Lillis
Panos Skyvalidas and Evaggelia Pitoura
Department of Computer Science, University of Ioannina, Greece
{kgeorgia, nkremmid, klillis, skyval, pitoura}@cs.uoi.gr

Computer Science Department,
University of Ioannina,
Technical Report 2006-08,
October 2006.

[*] Work supported in part by the integrated project AEOLUS, IST-15964

Contents

1 Overlay Networks

In a peer-to-peer system, a large number of nodes share data with each other. The participation of nodes is highly dynamic, nodes may enter and leave the system at will. Since, it is not possible to maintain links with all nodes, for performance as well as for privacy and anonymity reasons, each node maintains links with a selected subset of other nodes, thus forming an *overlay network*. A message between any two nodes in the p2p system is routed through the overlay network. The overlay network is built on top of the physical one. Thus, two nodes that are neighbors in the overlay network may be many links apart in the physical network.

The overlay network is built to facilitate the operation of a p2p system. In data sharing p2p systems, a basic functionality is discovering the data of interest. A look-up query for data items may be posed at any node in the overlay. The query is then routed through the overlay to efficiently discover the nodes in the p2p system that hold the data items requested. Such query routing must be achieved by contacting as "small" a number of nodes in the overlay as possible and by maintaining as "little" state information at each overlay node as possible.

There are two basic types of overlays: structured and unstructured ones. To assist lookup, *structured overlays* map (keys of) data items to nodes. In *unstructured overlays*, there is no correlation between nodes and data items. In structured overlays, the mapping is usually done through hashing the key space of the data items to the id space of nodes. Thus, each node in the overlay maintains a partition of the data space. In structured overlays, lookup reduces to locating the node in the overlay that is responsible for the corresponding data partition. Most structured overlays guarantee lookup operations that are logarithmic in the number of nodes. Unstructured overlays usually provide no guarantees for the performance of lookup. In the lack of any additional information, search in unstructured overlays is done through some form of flooding, where each node forwards the lookup to its neighbors until the data item is located. To assist in selecting an appropriate neighbor for forwarding the query, often some additional index information is kept at each node.

Most structured overlays follow a regulated topology, such as a ring, tree or grid. Then, upon entering the p2p system, each node takes a specific position in the overlay network. In unstructured overlays, the topology is not a rigid one. Upon entering the system, each node usually connects to an existing node selected randomly from a set of known p2p entry points. Due to the high churn produced by nodes entering and leaving the system, the maintenance cost in structured overlays is high. Another important issue in building structured overlays is deriving mappings that produce a load balanced assignment of items to nodes. Load is also related to query load, so that each node in the overlay network receives the same amount of query and update messages.

To improve performance of lookup, caching and replication of either data, search paths or both is possible. Replication besides improving search may also assist in providing load balancing. Further, replication improves fault tolerance and the durability of data items.

2 Structured Overlays

There are a number of issues regarding the design of a structured p2p system. One design dimension refers to the geometry of the overlay, that is, its structural characteristics. Another design choice is how data are mapped to nodes. The mapping must be fair so that nodes receive similar loads even when the data sets or the operations are skewed. All designs aim at supporting efficiently the basic operations of the overlay, that is, its construction, its incremental maintenance when nodes enter or leave the system, and search. Efficiency must be achieved even in the case of high churn. For some designs, it is easier to achieve physical network awareness either during their constructions by selecting as overlay neighbors, nodes that are neighbors in the physical network or during search, by routing a query to nodes that are neighbors at the physical layer. Finally, overlays differ with respect to the range of different type of queries that they support.

In the following, we discuss each of the above issues for: DHTs (CAN, CHORD), Tree-topologies (BATON, Cornell, P-Grid), skip-lists and hypercubes.

2.1 Distributed Hash Tables (DHTs)

2.1.1 Content Addressable Network (CAN)

The overlay network in CAN [85] is structured as a d-dimensional Cartesian coordinate space (d-torus). Each node is assigned randomly a point in this space, when it enters the system. An instance of a CAN is shown in Fig. 1. Each node stores the coordinates of its zone as well as the coordinates of the neighbor zones. The key space is also dynamically partitioned into zones, so that each node "owns" one or more zone using hashing. To map a key-value, (K, V), pair to a specific node, CAN uses a hash function which maps key K to a particular point P in the virtual coordinate space. (K, V) is stored at the node which is responsible for the zone in which P lies. To retrieve a data item with key K, CAN uses the same hash function over K, thus K is mapped to point P. Next CAN retrieves the data from the node that owns P.

To route a query from a source to a specific destination, the routing algorithm of CAN follows the path from the source to the destination based on the Cartesian coordinates of the d-dimensional space in which CAN has been structured. In particular, a greedy algorithm is used to forward the query to the neighbor zone whose coordinates are closest to the coordinates of the destination. An example is shown in Fig. 2. Assume that the node that owns zone 1 asks for the data with coordinates (x, y). Node 1 will forward the query to its neighbor zone whose distance from (x, y) is the smallest. In our example, this zone is zone 4, thus the query is forwarded to the node that owns zone 4. Again, node 4 selects to forward the query to the neighbor with the smallest distance from (x, y). Routing continues until the node that owns the zone in which (x, y) belongs is reached.

For a *d-dimensional* space, partitioned into n equal zones (each node corresponds to one and only one node, so we have N nodes, in total), each node has *2d* neighbors and the average path length is *$(d/4)(N^{1/d})$*. Also, for a d-dimensional space, when we increase the number of nodes (consequently, increasing the number of zones), the average path length grows as *$O(N^{1/d})$*, while the amount of information that each node has to store stays the same ($2d$).

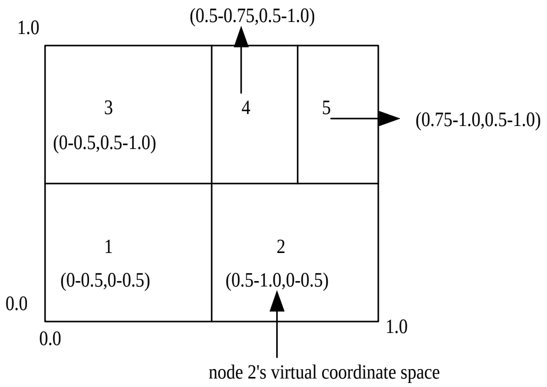
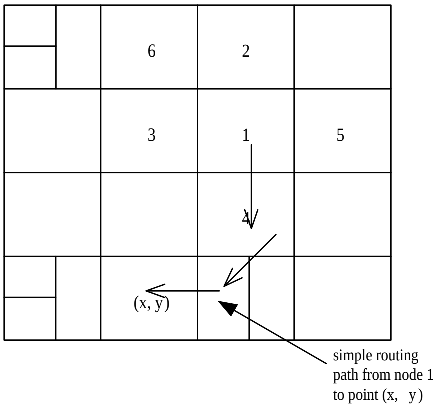


Figure 1: Structure of CAN

The virtual coordinate space is constructed whenever a new node arrives. Assume that a node wants to join to CAN. It chooses a random point P in the coordinate space and sends a join message to the node that owns the zone where P lies. When that message arrives to the node that owns P, the node divides its zone into two equal zones. For example, if we have a 2-dimensional space, then the node will divide its zone, initially on the first dimension (e.g. on X) and then on the second dimension (e.g. on Y). If the same node needs to divide its zone again, then it will divide it on dimension X again and the whole procedure is repeated by dividing its zone on dimension X, and Y, alternatively. Next, the (K, V) pairs that belong to the newly created zone are transferred to the new node and the neighbors of the old and the new node are informed of the arrival of the new node and the creation of the new zone. Finally, the new node is also informed of its neighbor zones. Fig. 3 provides an example, where node 7 joins the CAN system of Fig. 2. Node 7 is randomly assigned to a point, say on the zone owned by node 1. Node 1 divides its zone into two equal zones (on dimension Y). The left part of the zone is assigned to 1 and the other part to 7. In addition, all the (K, V) pairs of node 1 that now fall into the zone of 7 zone are transferred to 7. All the neighbors of 1 and 7 (nodes 2, 3, 4 and 5) are informed of the arrival of 7. Finally, 7 is informed of its neighbors (nodes 1, 2, 4 and 5).

When a node leaves CAN voluntarily, it hands over its zone and its (K, V) pairs to a neighbor node (with the restriction that valid zones, as described above, must be maintained). When a node fails, all the unreachable nodes initiate an *immediate takeover algorithm* [1], which assigns the zone of the failed node to a neighbor. A node failure can be traced through periodic messages that nodes send among them.

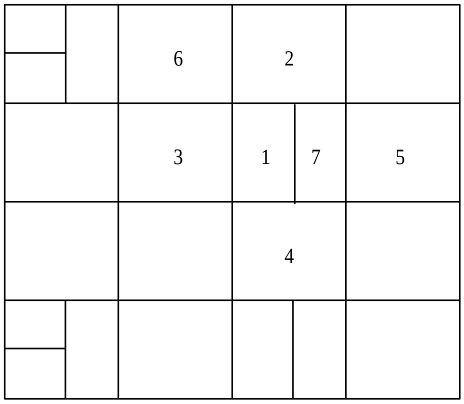
CAN uses several methods for achieving load balancing. First, CAN uses *uniform partitioning* so that nodes are assigned an equivalent space volume. When a node is about to split its zone after having received a request for an arrival of a new node, instead of splitting its own zone, it compares it with the zones of its neighbors. Then, the zone that occupies the largest space among them is selected to be split, so that one half of this zone is handed over to the new node. In this way, CAN manages to balance the data-load stored at each node. Other methods for load balancing exploit replication. One replication method for handling the query load is the following. When a node becomes overloaded with requests for a specific data key, the node replicates the key to its neighbors and



simple routing
path from node 1
to point (x,   y)

1's coordinate neighbor set = {2, 3, 4, 5}
7's coordinate neighbor set = {}

Figure 2: Routing example. Query is routed from 1 to (x, y)



1's coordinate neighbor set = {2, 3, 4, 7}
7's coordinate neighbor set = {1, 2, 4, 5}

Figure 3: An example of a node joining CAN. Node 7 joins the system

thus shares the load with them. Another replication technique is based on using multiple *realities*. Multiple coordinate zones are built so that each node can be assigned to a different zone in each reality. The contents of the hash table are replicated in each reality. This improves data availability as well as load balancing. Also data availability and load balancing can be improved by overloading coordinate zones by assigning each zone to more than one node. Finally, multiple hash functions can be used to map each key to k different points in the coordinate space and thus increase its availability.

Furthermore, each node can use caching so as to cache data keys it recently accessed. In that way when a node requests for a specific key, it can access, first, its own cache and if it finds it there it will not have to forward the request any further. As a result of caching, data becomes widely available and the search cost is drastically reduced.

2.1.2 CHORD

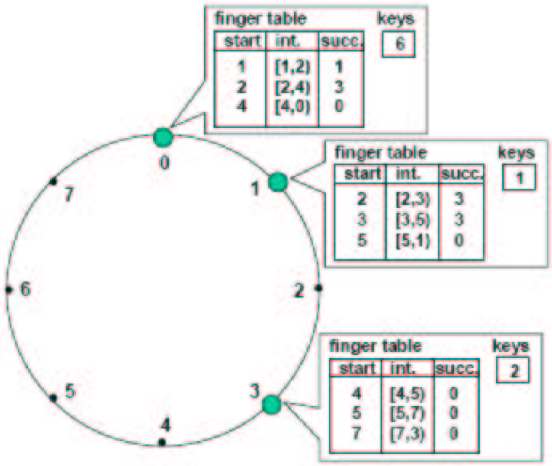
In Chord [75], nodes form a ring called identifier circle or Chord ring. Data files are also distributed over the same identifier space. In particular, Chord maps the node identifier key space and the data key space to an m-bits identifier using consistent hashing. Consistent hashing tends to balance load, as each node receives roughly the same number of keys. Nodes are placed in the ring according to the modulo of the key with the number 2^m. A data file with key k is stored on the first node whose identifier is equal to or follows k in the identifier space. This node is called the *successor* node of key k.

Each node needs only to be aware of its successor node in the circle. Specifically, each node is linked to the next one with a successor pointer. Queries for a given identifier can be passed around the circle via the successor pointers until they first encounter a node that succeeds the identifier; this is the node the query maps to. To accelerate this process, each node maintains a routing table with information for $O(\log N)$ other nodes. In this way, queries are executed more efficiently. In particular, each node knows all the other nodes carrying the nearest, largest key among all the keys that are at a distance of an increasing power of 2. This knowledge is stored in a table with m entries, called finger table. A finger table entry includes the Chord identifier, the IP address and the port number of the relevant node. The first entry is its immediate successor on the circle. The i^{th} entry of the finger table for a node n, called the i^{th} finger of n, contains the identity of the first node, s, that succeeds n by at least 2^{i-1} in the identifier circle, i.e., $s = successor((n + 2^{i-1})mod(m))$, where $1 \leq i \leq m$.

Fig. 5 shows the finger tables of three nodes in a Chord ring with $m = 3$. When a node n performs a lookup for a key, the first thing checked is whether the key is located between n and its successor. If this is true, then n's successor is the node that owns the key. Otherwise, n searches its finger table to find the node with the first largest key from the key that is being looked up. This procedure continues until the node that stores the key is found. In a system with N nodes, a lookup operation requires a total of $O(\log N)$ messages to be transmitted among nodes.

As nodes can join and leave the network dynamically, the Chord protocol needs to make sure that every lookup operation is performed successfully. For this reason, the two following invariants are preserved at any time.

- Each node's successor is correctly maintained.



finger table

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	0

keys 6

finger table

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	0

keys 1

finger table

start	int.	succ.
4	[4,5)	0
5	[5,7)	0
7	[7,3)	0

keys 2

Figure 4: Chord ring for m=3. Finger tables of nodes 0, 1, 3

- For every key k, node $successor(k)$ is responsible for k.

To simplify the join and leave operations, each node maintains also a pointer to its immediate predecessor in the circle. When a new node n joins the system three steps must be carried out. Initially, n must connect with an existing node p in the system and initialize its finger table using p's support. Secondly, the finger tables of the existing nodes in the system must be updated to include node n and finally the appropriate keys that n is responsible for, must be transferred to n from its successor.

Chord uses two mechanisms to deal with nodes joining the system concurrently with nodes that fail or leave the system. The first mechanism is a stabilization protocol that each node runs periodically to keep its successor pointers up to date, which is sufficient to guarantee correctness of lookups. These successor pointers are then used to verify and correct finger table entries, which allows the lookup operations to be fast as well as correct. When node n runs the stabilization protocol, it asks its successor q for q's predecessor p and decides whether p should be its successor instead. This would be the case if node p recently joined the system. The stabilization protocol also notifies node n's successor of n's existence, giving the successor the chance to change its predecessor to n. The successor does this only if it knows of no closer predecessor than n. The second mechanism is a replication mechanism. Each Chord node maintains a successor-list of its r nearest successors on the Chord ring. If node n notices that its successor has failed, it replaces it with the first live entry in its successor list. At that point, n can direct ordinary lookups for keys for which the failed node was the successor to the new successor. As time passes, the stabilization protocol will correct finger table entries and successor-list entries pointing to the failed node.

2.1.3 P-Ring

P-Ring [30] introduces a new content router called Hierarchical Ring (HR) that can efficiently route range queries and handle highly skewed data distributions. The basic idea behind HR is the creation of a hierarchy of rings. At each level of the hierarchy a list of the first d successors are maintained, where d is an integer called the *order* of HR. At the lowest level, level 1, node p maintains a list of the first d successors on the ring. Using these successors, a message can be forwarded to the last successor in the list that does not overshoot the target skipping up to d-1 nodes at a time. At level i, a list of d successors is also maintained. However, a successor at level i corresponds to the dth successor at level i - 1. Using these successors, a message can be routed to the last successor in the list that does not overshoot the target, skipping up to $d^(i-1)$ nodes at a time. The procedure of defining the successor at level $l + 1$ and creating a list of level $l + 1$ successors is iterated until no more levels can be created. The fact that positions of the ring are indexed instead of values, allows the structure to handle skewed data distributions. An HR of order d, has only $log_d(P)$ levels, and the space requirement for the HR component at each node is $O(dlog_d(P))$, where P is the number of nodes in the ring.

The routing procedure takes as input the lower-bound (lb) and the upper-bound (ub) of the range in the request, the message that needs to be routed and the address of the node where the request was originated. The routing procedure at each node selects the farthest away pointer that does not overshoot lb and forwards the request to that node. Once the algorithm reaches the lowest level of the HR, it traverses the successor list until the value of a node exceeds ub. In a consistent state, the routing procedure will go down one level in the HR every time a routing message is forwarded to a different node. Hence, each search process requires $O(dlog_d(P))$ steps.

As an example, consider the P-Ring of Fig. 5, which consists of an HR with three levels and a query with range (18, 25] that is issued at node 1. The routing algorithm first determines the highest HR level in node 1 that contains an entry whose value is between 5 (value stored in node 1) and 18 (the lower bound of the range query). This corresponds to the first entry at the second level of 1's HR successors, which points to node 3 with value 15. The query is hence forwarded to node 3. 3 forwards in a similar way the query to node 4. Since 4 is responsible for items that fall within the required range, 4 processes the routed message and returns the results to the originator node. Since the successor of 4, node 5, might store items in the (18, 25] range, the request is also forwarded to 5. 5 processes the request and sends the results to the originator node 1. The search terminates at node 5 as the value of its successor does not fall within the query range.

In a dynamic p2p environment with continuous node insertions and failures maintaining the HR consistent becomes a matter. P-Ring uses a simple Stabilization Process similar to that in Chord that runs periodically at each node and fixes the successor lists at all levels of the hierarchy. Each node needs only local information to compute its own successor at each level. The algorithm loops from the lowest level to the top-most level of the HR until the highest level is reached. When a node p stabilizes a level, it contacts its successor at that level and asks for its entries at the corresponding level. Node p replaces its own entries with the received entries and inserts its successor as the first entry in the index node.

P-Ring also uses a replication mechanism to deal with failures. Consider an item i stored in node p. The replication manager replicates i to k successors of p. In this way, even if p fails, i can be recovered from one of the successors of p. Larger values of k offer better fault-tolerance but introduce additional overhead.

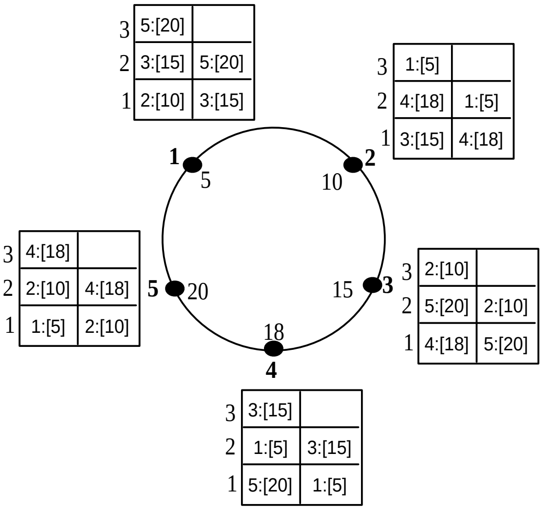


Figure 5: P-Ring with HR of 3 levels

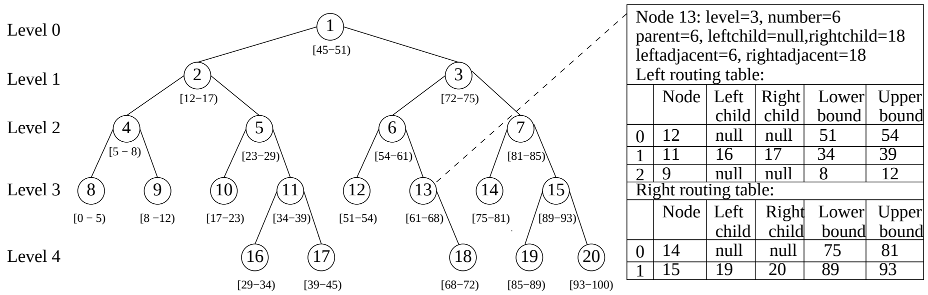
2.2 Tree-based Topologies

2.2.1 BATON

In BATON [56] a balanced binary search tree is maintained as shown in Fig. 6. In a typical search tree nodes near the root are much more frequently accessed than nodes near the leaves. The main contribution of BATON is a tree-structured overlay network that does not have a substantial skew in access load. Each node of the tree maps exactly one peer and is associated to its level and its position number at the level. The combination of the two precisely determines the location of a node in the tree and can be used to determine the structural relationship between a given pair of nodes. Thus, each node has a logical id in terms of its level and position number, and a physical id in terms of its IP address. Additionally, each node maintains links (the physical id) to its parent, to its children, to its left and right adjacent nodes (determined by the in-order traversal of the tree) and to selected neighbors at the same tree level. Links to selected neighbors are maintained by means of two routing tables, a left routing table and a right routing table, each of which has $O(\log N)$ entries where N is the number of peers in the network. The j^{th} entry in the left (right) routing table at node numbered n contains a link to the node at number n - 2^{j-1} (respectively n + 2^{j-1}) at the same level.

The overlay network described is used to partition data among participating peers and to build an effective distributed index structure. To achieve that, a range of values is assigned to each node. The range of values managed by a node is required to be to the right of the range managed by its left subtree and to the left of the range managed by its right subtree (for example [10 , 20] is to the right of [1 , 9] and to the left of [21 , 30]) the same way as in a binary search tree. Each node records, for each of its links, the range of values managed by the target node (Fig. 6).

Nodes can join and leave the network at any time. A node wanting to join must send a JOIN request to a node inside the network. A node receiving a JOIN request can accept the new node as its child if it does not already have two children and the tree remains



Node 13: level=3, number=6
parent=6, leftchild=null,rightchild=18
leftadjacent=6, rightadjacent=18
Left routing table:

	Node	Left child	Right child	Lower bound	Upper bound
0	12	null	null	51	54
1	11	16	17	34	39
2	9	null	null	8	12

Right routing table:

	Node	Left child	Right child	Lower bound	Upper bound
0	14	null	null	75	81
1	15	19	20	89	93

Figure 6: Binary balanced tree index architecture and data range assigned at each node

balanced. Otherwise, it forwards the request, by using its links, towards a node which has less then two children or is a leaf-node. When the node that will be the parent of the new node is located, it splits half of its range associated to it with its new child. The parent of the new node uses its own links to gather the information needed by the new node to create its links and to notify the nodes concerned by the join about the new nodes' existence.

If a leaf node wishes to leave the network and its departure does not upset the tree balance, it can simply leave after transferring its range of values to its parent and notifying all nodes in its routing tables and its adjacent nodes about the departure. All the above nodes update their links to reflect the changes provoked by the departure. If a node that wishes to leave can not do so without upsetting the tree balance, it has to find a leaf node, which can normally leave the network, to replace it. The leaf node follows the procedure mentioned above to leave the network and takes the place of the departing node. The nodes with links to the departed node are notified to replace its physical address with the address of the replacement node. In [56], it is shown that both node join and departure need $O(\log N)$ messages.

In case of a node failure, the parent of the failed node makes use of links maintained in its own routing tables to regenerate the links of the failed node by contacting the children of nodes in its own routing tables. This way the parent node can initiate a "graceful departure" for the failed node.

The tree structure of BATON allows it to efficiently support exact match and range queries. For an exact match query issued or received at a node, the node will first check its own range. If it is within the current range, the query can be answered locally otherwise it has to be routed to the node holding the range containing the query. Initially, if the nodes' upper (lower) bound is smaller (greater) than the query, the query is routed horizontally to the farthest neighboring node whose range upper (lower) bound is smaller (greater) than the query. Next the query is routed vertically by following right (left) adjacent or right (left) child links, until the node holding the range containing the query is located. At each step of the routing the search space is reduced by half, leading to a search cost of $O(\log N)$ steps. Observe that a request is forwarded at a higher level node only if the node contains the answer or the forwarding node does not have two children. This property helps the root to avoid receiving more requests than other nodes. A range query proceeds exactly in the same manner as an exact match query with the difference that the search is performed for a node whose range intersects with the query range. Then the query proceeds left and/or right to cover the remainder of the searched range. The cost of answering range queries is the cost of locating the first node, plus the cost of visiting the X nodes covering the remainder of the search range, i.e. $O(\log N + X)$ steps.

Skewed datasets and queries can lead some nodes of the network, holding the corresponding data range, to being overloaded. To avoid this from happening, nodes are allowed to split part of their range or acquire additional range from other nodes. Thus, if a node detects it is overloaded, it searches for a lightly loaded node to share its load. If the overloaded node is a non-leaf node, it does load balancing with adjacent nodes. Otherwise, if it is a leaf node and its adjacent nodes are also loaded, it finds a lightly loaded leaf node to do load balancing with. The lightly loaded leaf node leaves its position (the same way as if it leaves the network) and joins as a child of the overloaded node. This can cause the tree to be unbalanced and a network restructuring is needed to re-balance the tree. The restructuring is achieved by a number of rotations during which some nodes change their position in the tree and their links need to be updated. Thus, load balancing comes with a cost: for each node changing position, adjusting the links requires an $O(\log N)$ effort.

The only concern of the node joining procedure is keeping the tree balanced. Additionally during load balancing nodes are moved away from their initial positions with the single scope of relieving overloaded nodes. These observations indicate that it is particularly hard to achieve physical network awareness in the construction of the tree overlay.

2.2.2 BATON* and VBI-Tree

BATON provides search with cost $O(\log_2 N)$. For improving the search cost, a multi-way tree structure called BATON* [54] is derived from BATON by increasing the fanout of the search tree. Thus, in BATON* each node can have up to m children and in addition to storing links to its parent and children, it also keeps track of the range of values managed by its children. The range of values managed by a node is to the right of the ranges managed by its first $\lceil m/2 \rceil$ children and at the left of the ranges managed by its last $\lfloor m/2 \rfloor$ children. Additionally, different from BATON, in the routing tables of each node, links are maintained to neighbor nodes at the same level which are at distance $d \cdot m^i$, where $d = 1 \ldots m-1$ and $i \geq 0$. Thus, routing tables store $O(m \cdot \log_m N)$ entries.

The algorithms for nodes joining or leaving the network are the same as those for BATON and have a cost of $O(m \cdot \log_m N)$ for updating routing table entries at the nodes involved.

The search algorithm (for answering exact match queries) is modified from that in BATON to reflect the increased number of children at each node. Thus, when the query is forwarded vertically at a child node. The forwarding node needs to consider the information about the bounds of ranges maintained by its children to locate the appropriate child node to forward the query to. Obviously, since the fanout of the tree is m and routing tables store links to nodes at logarithmic (with base m) distance, the search cost in BATON* is $O(\log_m N)$. The cost for search is decreased logarithmically as the fanout is increased, but the cost for node insertions and deletions goes up. Thus, the choice of the fanout must consider the ratio of queries to node joining and leaving the network. Increased fanout of the tree also increases the number of links among nodes and the number of leaves in the network, thus achieving better fault tolerance and better load balancing. Therefore, fault tolerance and load balancing should be also considered in selecting the appropriate fanout. Additionally to speeding up search BATON* can also support queries over multiple attributes (described in Section 6.2).

The VBI-Tree [55] is another balanced binary tree structure which is based on BATON and supports multi-dimensional indexing schemes. In the VBI-Tree, nodes belong to two classes: data nodes that are leaf nodes actually storing data and routing nodes and internal nodes storing routing information. Each node, maintains parent, children, adjacent links same as BATON nodes, and information about heights of sub-trees rooted at its children. Additionally, each routing node maintains an upside table with information about regions covered by each of its ancestors and the left and right routing tables. Different from BATON, each peer in the network is assigned a pair of VBI-Tree nodes: a routing node and its right adjacent data node. Multi-dimensional data define data points in the attribute space, and regions containing them are partitioned among nodes. Each internal node has associated a region that covers all regions managed by its children.

The algorithms for peers joining and leaving the network are the same as those used in BATON, with the difference that now two nodes join/leave the network and data regions are divided/merged only between data nodes. The index constructed by nodes joining the network and hierarchically partitioning the attribute space among nodes, is used for efficiently answering queries on the multi-dimensional data.

A node n receiving a data point query checks if the region associated to it covers the data point. If the above is true and n is a data node it can answer the query itself otherwise it forwards it to one of its children. If the region associated to n does not contain the data point, using the upside table n finds the nearest ancestor x covering the data point query and forwards the query to a neighbor found in n's routing tables, situated at the other side of the sub-tree rooted at x. For answering range queries firstly a node associated to a region intersecting the query range is located and the query is forwarded to other appropriate nodes. Load balance is performed in the same manner as in BATON and as all other operations of VBI-Tree has the same cost as those in BATON.

2.2.3 P-Grid

In P-Grid [3], a dynamic binary search tree is maintained in a distributed way, such that the search space, consisting of binary strings, is partitioned between peers. The salient feature of P-Grid is the separation of concerns between peer identifier and peers position in the network. More specifically, each peer p is associated to a path $\pi(p)$ of the tree and is responsible for storing references to the peers that store data items indexed by keywords for which $\pi(p)$ is a prefix. Multiple peers can be associated to and thus, be responsible for the same path. Each peer maintains a routing table where for each prefix of $\pi(p)$ of length l it stores references to at least one other peer whose path has the same prefix of length l but differs at position $l + 1$. This means that at each level of the tree, each peer stores the address of one or more peers that are responsible for the other side of the tree at that level. An example of P-Grid is shown in Fig. 7

Paths of each peer are not determined a priori but are acquired and changed dynamically through negotiation with other peers. Initially, all peers are responsible for the whole search space (all binary keys). When two peers meet they decide, based on a local criterion, if they will divide the search space in half and each peer will become responsible for the corresponding half. Thus, peers extend their paths in opposite directions and store

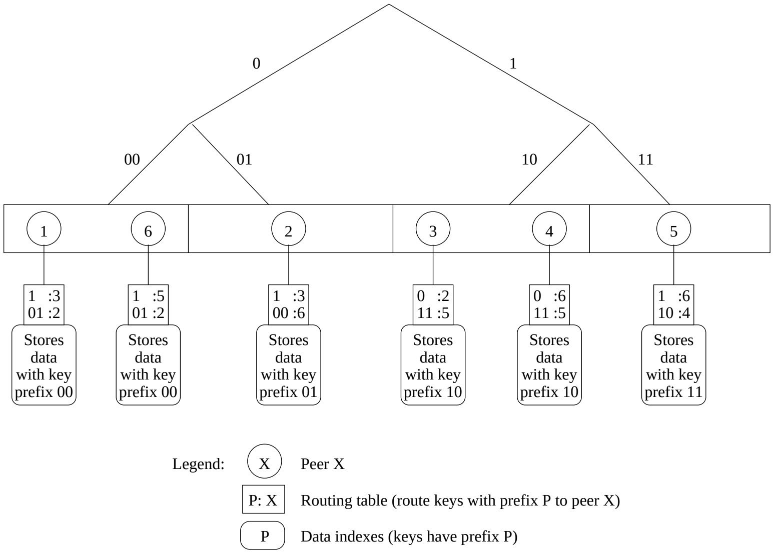

0
1
00
01
10
11
1
6
2
3
4
5
1 :3
01 :2
1 :5
01 :2
1 :3
00 :6
0 :2
11 :5
0 :6
11 :5
1 :6
10 :4
Stores data with key prefix 00
Stores data with key prefix 00
Stores data with key prefix 01
Stores data with key prefix 10
Stores data with key prefix 10
Stores data with key prefix 11
Legend:
X
Peer X
P: X
Routing table (route keys with prefix P to peer X)
P
Data indexes (keys have prefix P)

Figure 7: Example P-Grid. Each peer is responsible for part of the overall tree

references to each other. The same happens whenever two peers responsible for the same path meet. When two peers whose paths share a common prefix meet, they share their list of references at the level of the common prefix and initiate new meetings by forwarding each other to their referenced peers. Finally, when two peers meet and the path of one peer is a prefix of the other, the peer with the shorter path extends its path to the opposite direction from the other peer at that level.

If peers use as a splitting criterion their current storage load, then the resulting shape of the tree will adapt to the data key distribution, achieving an even distribution of storage load among peers. This way, skewed data sets don not influence the load balance but can lead to an unbalanced tree with height linear in network size.

The search space in P-Grid consists of binary strings, therefore, insertions and queries are based on binary keys extracted from the data. Queries can be issued at any peer and are routed through peers routing tables. A peer p issuing or receiving a query q checks if $\pi(p)$ is a prefix of q, therefore it can satisfy the query locally. Otherwise, it calculates the length l of the common prefix of $\pi(p)$ and q, and forwards the query to a randomly selected peer included in p's references for the level $l + 1$. This way the query is always routed downwards at the binary tree achieving a cost of $O(\log N)$ messages for a balanced tree. Theoretical analysis [4] shows that even for unbalanced trees, probabilistically the cost remains logarithmic in terms of messages due to the randomized selection of references to peers in the routing table.

As mentioned above, multiple peers can be responsible for the same path. Thus, all those peers store the same references to the peers that store data items indexed by keywords for which the path is a prefix. This way, data becomes replicated and the system more robust to failures. Additionally, peer failures don't influence the query routing algorithm. By storing multiple, alternative references in the routing table of each peer, in case of peer failures, there exists the possibility to use alternative routes to reach the desired destination.

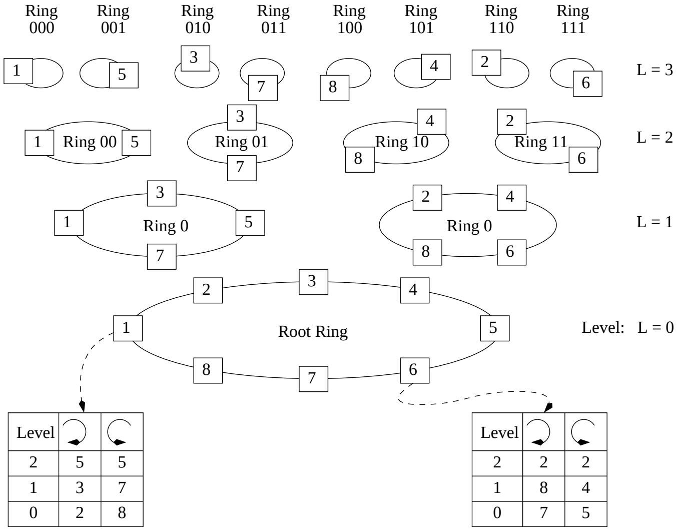
2.3 Skip-lists

SkipNet [51] organizes peers in a circular distributed data structure that provides controlled data placement and routing locality. It is a generalization of Skip Lists [83] which are sorted linked lists in which some nodes are supplemented with pointers that skip over many list elements. In SkipNet each peer is mapped to an element of the Skip List by using its string name ID (e.g. its DNS name). In a Skip List the head of the list is accessed at every search. Since peers should have equal roles and their processing overhead should be roughly the same, the list is transformed to a doubly-linked ring. Each peer maintains a routing table consisting of $2 \log N$ pointers to other peers, where N is the number of peers in the system. The pointers at level h of a peers routing table, point to peers that are roughly 2^h peers to the left and right of the given peer. All peers are connected by the root ring formed by each peer's pointers at level 0. Links at each level i of the routing tables form 2^i disjoint rings, which are obtained by splitting each level i - 1 ring into two disjoint rings, each ring containing every second peer of level i - 1 ring. At each level, each peer participates in exactly one ring which is encoded by a binary string. Therefore, to each peer is assigned a unique binary number, according to the ring it participates at each level, which represents the peer's numeric ID. The SkipNet infrastructure is shown in Fig. 8.

Routing in SkipNet can be performed in both the name and the numeric ID spaces. Routing by name ID is identical to searching in Skip Lists: At each peer, a search message will be routed along the highest level pointer of the left or right routing table that does not point past the destination value (peer name ID). The search terminates when the message arrives at a peer whose name ID is closest to the destination. If the sources name ID and the destination name ID share no common prefix, a message can be routed either left or right. In that case, a peer randomly picks a direction so that nodes whose name IDs are near the middle of the sorted ordering do not get a disproportionately large share of the traffic. If the source's name ID and the destination name ID share a common prefix, routing traverses only nodes whose name IDs share a non-decreasing prefix with the destination. This is an interesting property that can be exploited so that, (by using DNS names as name IDs) routing paths remain local within an administrative domain whenever possible.

Routing by numeric ID begins by examining peers in the root level until a peer is found whose numeric ID matches the destination numeric ID in the first digit. Next the search message is forwarded to that peer. Each peer that receives such a message and its numeric ID matches the destination numeric ID up to the i^{th} digit, forwards the message to a peer in the level i ring, whose numeric ID matches the destination numeric ID at least in the first $i + 1$ digits. This procedure is repeated until the destination is reached or the message can not be forwarded any further. In the second case the destination is chosen to be the peer whose numeric ID is numerically closest to the destination numeric ID among all peers in the ring where routing has stopped. Both routing by name ID and numeric ID have a cost of $O(\log N)$ messages [50].

To insert data in SkipNet, a search for its name ID is performed and data is stored at the peer reached by the search procedure. This makes the algorithm of answering exact match queries obvious. Additionally, since peers are ordered by their name IDs, range queries can be answered by locating the peer storing the lower bound of the query and



Level	↻	↺
2	5	5
1	3	7
0	2	8

Level	↻	↺
2	2	2
1	8	4
0	7	5

Figure 8: The SkipNet routing infrastructure for an 8 node system

following the neighbor pointers at the root ring until the peer storing the upper bound of the query is reached.

To join a SkipNet, a new peer first selects a random unique numeric ID which defines at which ring at each level the peer will participate (this way each node randomly at uniform chooses at each level to which of the two rings it will participate). Next, it must find the top level ring that corresponds to its numeric ID by performing a search by numeric ID. The new peer then finds its neighbors at the top level ring ring, using a search by name ID within this ring only. One of the neighbors takes over searching for the new peers name ID at the next lower level and thus finding its neighbors at that level. This procedure is repeated for each level until the root ring is reached and the new peer has joined the root ring. Finally the new peer notifies its neighbors at each level that it should be inserted next to them. When a peer wants to leave the system, it can proactively notify all its neighbors at each level to repair their pointers immediately.

The correct function of the overlay is ensured by the neighbor pointers in the root ring. The pointers in the next rings are just optimization hints. Thus, for the system to be fault tolerant is enough that the root ring neighbor pointers are maintained correctly. To achieve this each participating peer maintains a leaf set of 16 neighbors, thus, in case of peer failures the surviving peers can still reach their neighbors in the root ring. The pointers of the higher level rings are restored by a background process. Additionally, routing path locality ensures that failures along administration domains, allow peers to still point to live nodes within the same domain.

Routing by name ID makes it possible to control where data is stored within the network. For example, by appending the document name at the name ID of the desired peer, the search will reach that specific peer and the document will be stored there. Additionally, data can be uniformly distributed across the nodes within a certain domain by use of Constrained Load Balance. The document name ID is divided in two parts: the name ID of the domain where load balancing should be performed (the CLB domain) and the document name. To perform CLB, first the CLB domain is used as a name ID to

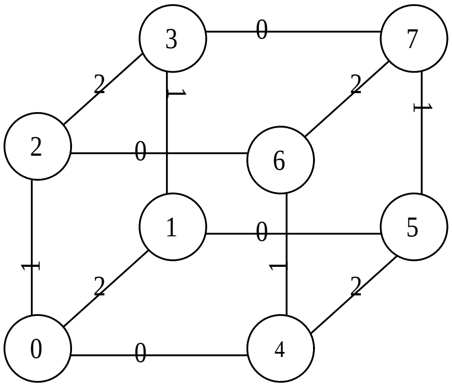


Figure 9: Hypercube topology

locate a peer inside the domain. Next a hash function is used to compute a numeric ID for the document name which is used for routing by numeric ID inside the domain. The peer where routing ends is the peer that will store the specified document. This way, by using a hash function that uniformly distributes keys, the load will be spread evenly among the peers within the domain.

Routing path locality ensures that routing between peers within the same domain will never leave the domain. To take into account the physical network topology when routing between peers in different domains, two additional routing tables are maintained: P-Table for routing by name ID and C-Table for routing by numeric ID. These two tables maintain routing in $O(\log N)$ messages, while also ensuring that each message has low cost in terms of network latency.

2.4 Hypercubes

HyperCup [92] is a p2p structure where nodes are placed in a hypercube topology. The advantage of a hypercube topology is that its diameter (that is the longer distance between two nodes) is O(log_bn), where b is the base of the hypercube and n is the number of nodes in the structure. Also, a hypercube topology is a symmetric structure, something that is crucial for load balancing in the network as the load will always be shared equally. In Fig 9, we can see a hypercube topology with base $b = 2$. As we can see the diameter of this hypercube is $log_2 8 = 3$ and the topology is symmetric (all nodes have two neighbors).

Routing in HyperCup is essentially a broadcast in the hypercube topology with a time-to-live (with limited scope). Broadcast in a hypercube works as follows: Each edge is labeled (node X is dubbed *i-neighbor* of node Y), as shown in Fig 9. When a node issues a broadcast, it sends the message to all its neighbors tagging it with the edge label on which the message was sent. Nodes receiving the message restrict the forwarding of the message to those links tagged with higher edge labels. For example, let us assume that node 0 in Fig. 9 sends a broadcast message. Node 4 receives this message which is tagged with the 0 label. Then node 4 will forward this message to both nodes 5 and 6, as the edges 4-5 and 4-6 are labeled with a higher label than 0. On the same step, node 1 receives the same message which is tagged with 2. In this case, node 1 will not forward the message to its neighbors as edges 1-5 and 1-3 are labeled with lower labels than 2. Finally, on the same

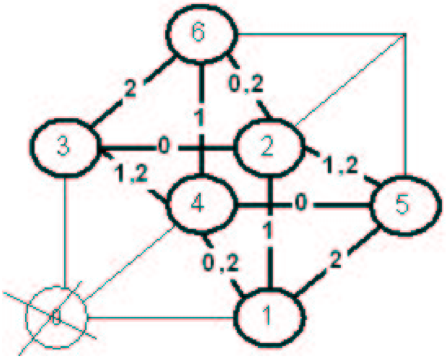


Figure 10: Node 7 has already left the structure. This is how the hypercube is "transformed" after the departure of node 0

step node 2 receives a message which is tagged with 1 and will forward it only to node 3, as edge 2-5 has lower label than 1 and edge 2-3 has higher label than 1. The broadcast algorithm continues with the same way until all the nodes get the message. In this way, it is guaranteed that exactly n-1 messages are required until the broadcast message reaches all nodes, while only $log_b n$ steps are required for the message to reach the furthest nodes.

Assume that a node wants to leave the network. Firstly, it must define the *buffering cluster*. The buffering cluster is the set of nodes that are responsible for the departing node and it is always the complementary *cluster* of the departing node. A cluster, in a hypercube topology is formed of two i-neighbors and all of the nodes of higher level. For example, in Fig. 9, one cluster is 0, 1, 2, 3 at level 0 (lower cluster). Secondly, the departing node must define the active nodes of the buffering cluster, which are the nodes that exist in the topology (in contradiction to the nodes of the buffering cluster that have left the hypercube). Next, the buffering nodes are defined. Each node of the buffering cluster takes over certain position in the departing node's cluster, so buffering nodes are those nodes of the buffering cluster where the links of the departing node are going to be transferred. Next, the links of the departing node are transferred to the buffering node. Finally, the departing node's neighbors are informed of the node's departure and they disconnect from it, while connecting to the buffering nodes.

In Fig. 10, we can see how the hypercube topology is "transformed" when a node disconnects from the hypercube. Let us assume that node 7 has already left the hypercube and now node 0 wants to leave it too. Firstly, node 0 defines the buffering cluster, which is the set of nodes 4, 5, 6, 7. Secondly, it defines the active buffering nodes, which are nodes 4, 5, 6 (since node 7 had previously left the structure). Then, the buffering node is determined, which, in our case, is node 4. Node 0 transfers its links (with node 3 and 1) to node 4, thus creating the links 1,2 and 0,2. Finally, nodes 3 and 1 are informed of node's 0 departure and they disconnect from node 0, while connecting to buffering node 4 (with the links 1,2 and 0,2 respectively).

Node arrivals essentially follow similar steps as node departure. The node that wants to join the network may contact any of the existing nodes of the hypercube. The contacted node chooses the *integration position*, which is one of the vacant positions of the hypercube where the arriving node will be placed. This position is the vacant position

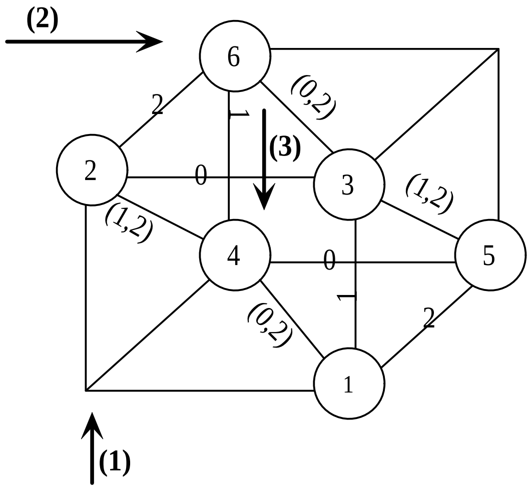


Figure 11: Node 8 arrives to the hypercube. Arrows (1), (2) and (3) show the steps of the arrival's algorithm

of the lower cluster. If the hypercube is complete (in the current level), as in Fig. 9, then the hypercube is extended creating a new (higher) level in the hypercube. Next, the contacted node chooses the *integration control node*, which is, in fact, the buffering node of the integration position. Then the integration request is forwarded to the integration control node. Finally, the links that were held by the buffering node are now transferred to the new node and the buffering cluster of the new node is informed of its presence.

In Fig 6, we can see an example of a node arrival. Let's assume that the current hypercube is as in Fig. 11 (continuing the example of Fig. 10) and node 8 arrives to the network. Firstly, let us assume that node 8 contacts node 6 (arrow (2) of Fig. 11). Node 6 defines that the integration position is position number 0 (arrow (1) of Fig. 11), as it is the vacant position of the lower cluster. Next, node 6 chooses node 4 as the integration control node, as it is the buffering node of position number 0 (arrow (3) of Fig. 11). Finally, node 4 transfers the responding links to position number 0 and thus to node 8 and the buffering cluster of node 8 is informed of the new node's presence with a broadcast message (arrows (4) and (3) of Fig. 12).

2.5 Comparison

Table 1 summarizes the characteristics of the structured systems presented.

3 Unstructured Overlays

Unstructured overlays are formed by the nodes as they join the system by either selecting randomly a node from a known list of other nodes in the overlay or by following some loose rules regarding this selection. The resulting topology may have certain properties however there is no assumption regarding the way the data space is mapped to the address space of the nodes in the overlay.

Table 1: Structured P2P Systems. N is the number of nodes, h the tree height, d the number of CAN dimensions, b the base of the hypercube, m the order of BATON*, R the average number of references stored at each level, and D the order of the hierarchical ring

	Search	**Join & Leave**	**Space**	**Load Balance**	**Query Type**	**Replication**	**Physical Proximity**	**Routing**

CAN	O($dN^{1/d}$)	Join:
		O($dN^{1/d}$)
		Leave:
		O(1)
			O(d)	Yes	Key-value	Yes
						Load balance
						Availability
						Performance
						Multiple realities
						Multiple hash functions
						Replicate at neighbors
						Caching
						Overload coord zones
							RTT	Iterative

Chord	O(log$N$)	O(log$^2 N$)	O(log$N$)	Yes Use of virtual nodes Consistent hashing	Key-value	Yes Load balance Availability successor-list of r successors Replication at r successors	No	Iterative
P-Ring	O(log$_D N$)	O(log$_D N$)	O(Dlog$_D N$)	Yes Overflow: split Underflow: merge or redistribute	Key-value Range	Availability Replication at r-successors	No	Iterative
BATON	O(log$N$)	O(log$N$)	O(log$N$)	Yes Loaded nodes split content with lightly loaded ones	Value Range	No	No	Recursive
BATON*	O(log$_m N$)	O($m$log$_m N$)	O($m$log$_m N$)	Yes Loaded nodes split content with lightly loaded ones	Single/multi attribute value/range	No	No	Recursive
VBI-Tree	O(log$N$)	O(log$N$)	O(log$N$)	Yes Loaded nodes split content with lightly loaded ones	Single/multi attribute value/range	No	No	Recursive
PGrid	O(logN)		O(hR)	Yes Storage Querying assuming uniform query distribution	Key-value Range	Yes Availability Performance Many peers responsible for the same part of the search space	No	Recursive
SkipNet	O(logN)	O(logN)	O(logN)	Yes Constrained load balancing, consistent hashing	Value Key Range	No	Yes, by construction nodes sharing common domains are neighbors	Recursive Iterative
HyperCup	O($log_b N$)	O($log_b N$)		Yes Assuming uniform join and leave		No	No	Recursive

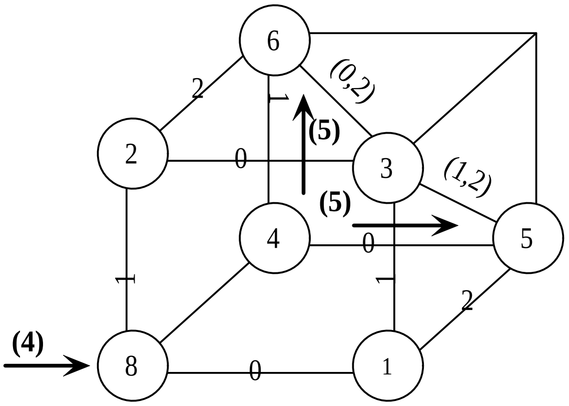


Figure 12: Node 8 has taken its place in the hypercube and the buffering cluster is informed of its presence

To locate data of interest, a node queries its neighbors in the overlay. The most typical query method is *flooding*. In flooding, the query is propagated to all neighbors within a certain radius. Unstructured overlays are resilient to nodes entering and leaving the system. However, they provide no guarantees on the complexity of search. It has been shown that the search cost is reduced if the square-root principle is satisfied. The square root principle states that each object is probed with probability proportional to the square root of its query popularity.

We can roughly distinguish search in blind and informed search [100]. In *blind* search, nodes maintain no information about the distribution of items to nodes. Their goal is to send the query to a sufficient number of other nodes so that the query is successful. In *informed* search, to make searching more efficient in terms of message overhead and response time, nodes in the overlay maintain information about the location of data. Such indexes are distributed in the overlay network.

The topology of an unstructured overlay is built up over time in a decentralized manner as peers join and leave the system. In many existing systems, upon joining the network, a peer selects to connect to another peer essentially at random. In these systems, topologies tend toward a power-law distribution, where some long-lived peers have many connections, while most other peers have a few connections. Formally, in a power-law network, the number of neighbors of the ith most connected peer is proportional to $\frac{\omega}{i^a}$, where ω is a constant that determines the degree of connectivity and a is a constant that determines the skew of the distribution. with larger values resulting in more skew.

3.1 Blind Search

With blind search, the only way to locate an item is to visit enough nodes so that, with high probability, one of the nodes has the item. Thus, the key to scalable blind searches in unstructured network is to *cover* the right number of nodes as quickly as possible and with as little overhead as possible.

The authors in [70] highlight three important issues in reaching the requested coverage. One is a low-overhead mechanism for adaptive termination of search. Flooding uses a fixed value for TTL to terminate search whose value is critical for both the probability of success and the associated cost. If the TTL value is too high, each query unnecessarily burdens the network. If the TTL is too low, a query may fail to locate a requested item, even if the item exists in the network. Another issue is to minimize message duplication. An important problem with flooding is that it introduces a very large number of duplicate messages, particularly in high connectivity overlays. *Duplicate messages* are multiple copies of a query that are sent to a node by its multiple neighbors. Duplicate messages are pure overhead, since they incur extra network interrupt processing at the nodes receiving them but do not increase the chance of locating an item. Usually, duplicate detection mechanisms are deployed in unstructured p2p systems, so that when a node receives a query more than once, it does not forward it any further. However, even in this case, the number of duplicate messages can still be excessive, and the problem worsens as the TTL increases. Finally, the granularity of the coverage should be small. Each additional step in the search should not increase the number of nodes visited significantly. In flooding, at each step, the number of nodes visited is multiplied resulting in an exponential growth.

3.1.1 Expanding Ring

To address the first issue, that of determining a good value of TTL, *expanding ring* initiates successive flood requests with increasing TTLs. A node starts with a flood having a small TTL. If the search is not successful, the node increases the TTL and starts another flood. The process is repeated until the object is found or some maximum TTL value is reached. This method is expected to work well when hot data items are replicated more widely than cold data items. In this case, most of the queries are satisfied with a small TTL, since hot items can be found within a very small number of steps. Although, expanding ring addresses the shortcoming of having a fixed TTL value, it still suffers from a high number of duplicate messages. More importantly, the granularity of coverage is still large.

A similar approach to expanding ring, called *iterative deepening* is proposed in [107] with a slightly different implementation. The requesting node waits for a protocol specified time period W. If during this period, it does not receive any replies, it proceeds with issuing a new flood with a larger TTL value. Assume two successive flood messages with TTL a and b, $b > a$. Queries are labeled with an ID, so that when at a subsequent flooding round, they receive the same query they do not process it, but simply forward it to their neighbors.

3.1.2 Random Walks

With *random walk*, at each step, a query message is forwarded to a single randomly chosen neighbor until the object is found. This message is called a *walker*. Random walks can reduce the message overhead by an order of magnitude compared to expanding ring. However, there is also an order of magnitude increase in user-perceived delay. To reduce this delay, the number of *walkers* may increase. With *k random walkers*, instead of just sending out one query message, a requesting node sends k query messages, and each query message takes its own random walk. The expectation is that k walkers after T steps should reach roughly the same number of nodes as 1 walker after kT steps. This is confirmed by simulation. The number of walkers is an important parameter. With more walkers, data items are located faster, but the message overhead is higher. Further, when the number of walkers is large enough, increasing it further yields only a very small reduction in the number of hops although significantly increasing the message traffic. The authors experimented with different number of walkers. Usually, 16 to 64 walkers give good results. Two methods are proposed for terminating the random walk search: TTL and checking. As in flooding, TTL means that, each random walk terminates after a certain number of hops is reached. With *checking*, a walker periodically checks with the original requester before walking to the next node. The checking method still uses a TTL, but the TTL is very large and mainly used to prevent cycles. Simulations shows that using checking is preferable than using TTL, since the TTL approach runs into the same TTL selection problem as flooding does. Further experiments show that checking once at every fourth step strikes a good balance between the overhead of the checking messages and the benefits of checking.

The authors also studied an improvement to random walkers that requires each node keep *state*. Each query has a unique ID and all its k walkers are tagged with that ID. For each ID, a node remembers the neighbors to which it has forwarded queries of that ID. When a new query with the same ID arrives, the node forwards it to a different neighbor randomly chosen. This state keeping accelerates the walks because walkers are less likely to cover the same route and hence they visit more nodes. Simulation results show that there is improvement, however this is very small in power-law style graphs.

3.1.3 Biased Random Walks

Although random walks result in a better coverage, this can be improved so that the selection of the next node to visit is not random but instead takes into account the likelihood that this node will have responses for the query. This form of search is not blind any more since it assumes that there is some information about the content of neighboring peers that would help the selection of which node to visit next. In particular, peers must know the content of their neighbor peers.

[7] proposes to bias the random walks towards *high-degree nodes.* The intuition behind this is that high-degree nodes have many neighbors and thus will have knowledge of a large number of items. Hence, a high-degree node is more likely to have an answer that matches the query.

Biasing the search towards high-degree nodes may overload them. The design of Gia [24] takes into account the *capacity constraints* associated with each node. The capacity of a node depends upon a number of factors including its processing power, disk latencies, and access bandwidth. Studies of p2p systems have shown that the nodes of a p2p exhibit high degrees of heterogeneity with regards to capacity. Gia proposes a dynamic topology adaptation protocol with the goal to ensure that high capacity nodes are indeed the ones with high degree and that low capacity nodes are within short reach of higher capacity ones. An active flow control scheme is used to avoid overloading hot-spots by assigning flow-control tokens to nodes based on available capacity. Random walks are biased towards high-capacity nodes, which are typically best able to answer the queries.

The authors of [112] propose *popularity-biased* random walks that realizes the square-root principle. At each step of the walk, the node to visit next is chosen from the neighbors of the current node with probabilities that are proportional to the square root of the content popularity of the neighbor.

3.1.4 Restricted Flooding

Another approach to restrict the traffic introduced by flooding is instead of forwarding a search request to all neighbors of a node, to forward it to a subset of its neighbors. In *Random Breadth First Search* [57] or *teeming* [36], the sets of neighbors to forward the query are selected randomly.

A variation of teeming, called *teeming with decay* was proposed in [67]. This works like teeming but the probability ϕ of contacting a neighbor decreases with the distance from the requesting node to avoid the high message count in later steps. If a node is in distance t from the requesting peer, then the probability is given by:

$$\phi_t = \phi(1-c)^t,$$

where $\phi_0 = \phi$ and c is the *decay* parameter. For $c = 0$ we have simple teeming, while if in addition $\phi = 1$, the strategy is pure flooding. The higher the value of c, the faster the probability decreases and the fewer the message transmissions become.

In the *Directed BFS (DBFS)* [107], each node maintains simple statistics about the query behavior of its neighbors and uses them to select those that are expected to return more "quality" results. Such statistics may include for example the number of results received through that neighbor for past queries, or the latency of the connections with that neighbor. These statistics are used for selecting s the best neighbor to send the query such as selecting the neighbor (i) that has returned the highest number of results for previous queries, (ii) that returns response messages that have taken the lowest average number of hops (iii) that has forwarded the largest number of messages for the node, potentially implying that this neighbor is stable and it can handle a large flow of messages, or . (iv) with the shortest message queue. In general DBFS has better response time than expanding ring but generates more message traffic.

3.1.5 Search in Square-Root Overlays

[29] introduce the *square-root topology*, where the degree of each peer is proportional to the square root of the popularity of the content at the peer. The content popularity of a peer is estimated as the number of queries satisfied by a peer divided by the total number of queries received by it. Analytical results based on random walks in Markov chains show that the square-root topology is not only better than power-law networks, it is in fact optimal in the number of hops needed to find content. Intuitively, the probability that a random walk quickly reaches a peer is proportional to the degree of the peer, and if peers with popular content have correspondingly high degrees, then most searches will quickly reach the right peers and find matching content. Simulation results confirm the analysis, showing that a random walk requires up to 45 percent fewer hops in a square-root topology than in a power-law topology. Simulation results also show that several other walk-based techniques perform better in a square-root topology than in a power-law topology. To construct the square-root topology, each peer uses purely local information to estimate the popularity of its content. Then, each peer adds or drops connections to other peers to achieve its optimal degree.

3.2 Using Indexes

In the *Local Indices* technique [107], each node n maintains an index over the data of all nodes within r hops of itself, called the radius of the index. When a node receives a query, it can process it on behalf of all nodes within distance r from it. This way, the data of many nodes can be searched by processing the query at few nodes. A system-wide policy specifies the depths at which the query should be processed. All nodes at depths not listed in the policy simply forward the query to the next depth. To minimize the overhead, the hop-distance between two consecutive depths must be $2r + 1$. In contrast to iterative deepening, where all nodes within the TTL process the query, with local indexes only nodes at the specified depth process it. Local indexes also require flood with TTL $= r$, whenever a node joins or leaves the network and whenever it updates its local data.

The objective of a *Routing Index* (RI) [31] is to allow a node to select the "best" neighbors to forward a query to. The notion of goodness may vary but in general it should reflect the number of documents that the neighbor is expected to provide. The *compound RI* (CRI) maintains for each neighbor n of a node an index of the number of documents that can be reached through n. A limitation of using CRIs is that they do not take into account the difference in cost due to the number of hops required to retrieve a document. The *hop-count RI* (HRI) stores aggregated RIs for each hop up to a maximum number of hops, called the *horizon* of the RI. The *exponentially aggregated* RI stores the number of documents weighted by the distance. The weight decreases exponentially with the distance. Routing indexes impose an extra overhead for updates. One additional problem with routing indexes is dealing with cycles. In the case of cycles in the p2p network, a document may be reached from a node by following more than one path.

Depending on the content shared by the peers, many different types of data structures can be used as routing indexes. For numerical data, [80] proposes using histograms as hop-count routing indexes. For each neighbor n of a node, the corresponding histogram is used to estimate the selectivity of the nodes on the path through n. Each query is then forward to the neighbors with the largest estimated selectivity.

[86] propose using a data structure called attenuated Bloom filter as a routing index. An *attenuated Bloom filter* of depth d is an array of d normal Bloom filters. Each neighbor of a node is associated with one attenuated Bloom filter. The first filter in the array summarizes documents available from that neighbor, that is, one hop along the link. The i-th Bloom filter is the merger of all Bloom filters for all of the nodes a distance i through any path starting with that neighbor link. To map from an attenuated Bloom filter to a potential value, one queries each level for a document's name. The levels are assigned geometrically decreasing potential values; the value of the potential function of a filter for a given document is the sum of all of the potential values for the levels of the filter which contain the document. To perform a query, the requesting node examines the 1st level of each of its neighbors' attenuated Bloom filters. If one of the filters matches, it is likely that the desired data item is only one hop away, and the query is forwarded to the matching neighbor closest to the current node in network latency. If no filter matches, the querying node looks for a match in the 2nd level of every filter and so on. By doing so, the query is forwarded to the nearest neighbor that potentially has a copy of the requested data item.

In [57], each node maintains a *profile* for each of its immediate neighbors. The profile contains the list of the most recent past queries that the specific peer provided an answer for. The node accumulates the list of past queries by two different mechanisms. In the first mechanism, the peer is continuously monitoring and recording the query and the corresponding QueryHit messages it receives. In addition, in the second mechanism, when replying to a query, each peer broadcasts this information to its neighbor peers. This operation increases the accuracy of the system, at the expense of $O(d)$ extra messages (where d is the average degree of the network) per answering node. Once the repository is full, the node uses a Least Recently Used (LRU) policy to keep the most recent queries in the repository. For each query it receives, the receiver peer uses the profiles of its peers to find which ones are most likely to have documents that are relevant to the query. To compute the ranking, the receiver peer compares the query to previously seen queries and finds the most similar ones in the repository. One problem of this technique is that it is possible for search messages to get locked into a cycle where each node forwards the query only to its neighbors. To address this problem, a small random subset of peers is picked and added to the set of best peers for each query. In this case, even if the best peers form a cycle, with high probability the mechanism will explore a larger part of the network and will learn about the contents of additional peers.

In Freenet [109], files are identified by binary file keys obtained by applying a hash function to a string that describes the content of the file. Each node maintains a routing table which is a set of (key, pointer) pairs, where pointer points to a node that has a file associated with the key. A steepest-ascent hill-climbing with backtracking is used to locate a document. Each node forwards a request for an item with key k to the node in its routing table whose key is the one most similar to k.

4 Other Overlays

4.1 Multi-Layer Overlays

Besides pure peer-to-peer systems in which all nodes in the system have equal roles, in hybrid peer-to-peer systems, the responsibilities assigned to peers differ. In super-peer p2p systems, some peers, called the *super peers*, are serving as servers to the rest of the peers. Super-peer networks strike a balance between the efficiency of centralized search, and the autonomy, load balancing and robustness to failures and attacks provided by fully distributed search. Furthermore, they take advantage of the heterogeneity of capabilities (such as bandwidth and processing power) across peers.

A popular topology is having a super-peer serving as a server for a subset of the peers. Peers submit queries to their super-peer and receive results from it. Super-peers are also connected to each other as peers in a pure system are, routing messages over this overlay network, and submitting and answering queries on behalf of their peers and themselves. This creates a two-layer topology, one among the super-peers and one between the super-peer and the peers assigned to it. Taking this design a step further, peers of a super-peer can also organize themselves in a hybrid overlay, where some peers become super-peers of the rest leading to multiple levels of peers with decreasing responsibilities.

Designing super-peer networks raises many issues. Some of them are discussed in [108]. One issue is determining the number of peers assigned to each super-peer, that is the size of the clusters. In terms of cluster size, there is a clear trade-off between aggregate and individual network and processing load. Increasing cluster size decreases aggregate load, but increases individual load at each super-peer. Further, the reliability and availability of p2p overlays with large clusters and thus only a few super-peers is small, since there are a few points of failure and should the few super-peers leave, fail or be attacked, processing at their clusters is affected. In general, introducing redundancy at the super-peer layer, that is having more than one peer serving a cluster of peers, results in better availability, reliability and load balance.

Work in [43] introduces *Canon*, a *hierarchical DHT*. The key idea behind Canon lies in its recursive routing structure. The internal nodes in the hierarchy are called domains to be distinguished from the actual system nodes. The nodes in a domain of a node n are all the system nodes at the subtree rooted at n. The design of Canon ensures that the nodes in any domain form a DHT routing structure by themselves. The DHT corresponding to any domain is synthesized by merging its children DHTs by the addition of some links. The challenge is to perform this merging in such a fashion that the total number of links per node remains the same as in flat DHT designs, and that global routing between any two nodes can still be achieved as efficiently as in flat designs. The Canon principle can be applied to transform many different DHT designs into their Canonical versions.

Another proposal for a hierarchical DHT design is Coral [41]. Coral also provides a distributed sloppy hash table abstraction (DSHT). Instead of storing actual data, the system stores weakly-consistent lists of pointers that index nodes at which the data resides. Instead of one global lookup system, Coral uses several levels of DSHTs called *clusters*. Coral nodes belong to one DSHT at each level. Clusters are build based on latency. To this end, Coral assigns round-trip-time thresholds to clusters to bound cluster diameter and ensure fast lookups. The current design supports three clusters. The goal is to establish many fast clusters with regional coverage, multiple clusters with continental coverage and one planet-wide cluster. To insert a key/value pair, a node performs a put on all levels of its clusters. This practice results in a loose hierarchical data cache, whereby a higher-level cluster contains nearly all data stored in the lower-level clusters to which its members also belong. To retrieve a key, a requesting node first performs a get on its lower level cluster to take advantage of locality and if the search is not successful the search continues at higher levels. Two conflicting criteria impact the effectiveness of a clustered DSHTs. First, clusters should be large in terms of membership. The more peers in a DSHT, the greater its capacity and the lower the miss rate. Second, clusters should have small network diameter to achieve fast lookup. That is, the expected latency between randomly-selected peers within a cluster should be below the cluster specified threshold. Coral provides an algorithm for self-organizing, merging and splitting clusters, to ensure acceptable cluster diameters.

4.2 Hybrid Structured and Unstructured Overlays

Motivated by the fact that popular items are easy to locate in unstructured p2p systems, while rare items require excessive flooding, the authors of [69] propose building hybrid overlays. The hybrid search infrastructure utilizes selective publishing techniques that identify and publish only rare items into the DHT while flooding is used for the popular items.

Different heuristics can be used to identify which items are rare. One simple heuristic is based on the observation that rare items are those that are seen in small result sets.

Although this scheme is simple, it suffers from the fact that many rare items may not have been previously queried and returned in any result sets. Consequently, these items will not be published to the DHT via such a caching scheme. Other techniques may be used for these items, for example, maintaining statistics. This hybrid infrastructure can easily be implemented: if not enough results are returned by flooding within a predetermined time, the query is reissued as a DHT query.

4.3 Semantic Overlays

In semantic overlay networks, peers are connected to each other based on the likelihood of them providing results for similar queries. There are several ways to capture such semantic relationships between peers. One approach is to explicitly identify such relations by using a content similarity clustering or classification algorithm to place in the same cluster or class the peers that have similar content. Another way of creating semantic overlays is by some form of caching. In this case, the overlay is built incrementally based on the results of the queries submitted to the system.

In semantic overlays, each peer is connected with the other peers in its cluster through a large number of links. To help route across semantic clusters, a few random links are usually added between peers at different clusters. This resembles the creation of small-worlds [104] which are networks having a large clustering coefficient and a small diameter.

4.3.1 Semantic Overlays through Content Clustering

In *Semantic Overlay Networks (SONs)* [32], nodes with semantically similar content are clustered together. Queries are processed by identifying which SON (or SONs) are better suited to answer them and each query is sent to a node in those SONs. Then, these nodes forward the query only to other members of their SON. In this way, a query for an item goes directly to the nodes that have similar content (which are likely to have answers for it), thus reducing the time that it takes to answer the query. Furthermore, nodes outside the cluster (and therefore unlikely to have answers) are not burdened by such queries. A classifier is used for determining which cluster or clusters each incoming node should join and which cluster should receive each query. The classification is based on the content of each node. An important issue is the level of granularity for the classification. A small granularity will not generate enough locality, while a large granularity will increase maintenance costs.

A similar approach is taken by SETS [12]. Nodes are arranged in a *topic-segmented overlay.* Each topic segment has a succinct description called the topic centroid. Topically focused sets of nodes are joined together into a cluster connected through short distance links. Long distance links connect pairs of nodes from different segments. When a search query is initiated, it is forwarded to other nodes using a topic-driven routing protocol. First, topic centroids are used to select a small set of relevant topic segments. Next, the selected segments are probed in sequence. A probe to a particular segment proceeds in two steps: First, the query is routed along long distance links to reach a random node belonging to the target segment. Next, the short distance links are used to propagate the query to an appropriate subset of the nodes within the segment.

Psearch [98] builds a semantic overlay so that the distance of two documents in the network is proportional to the their dissimilarity in semantics. The document semantics is produced using LSI. CAN is used as the semantic overlay by using the semantic vector (generated by LSI) of a document as the key to store the document index in the CAN. By doing so, the dimensionality of the CAN is set equal to that of LSI's semantic space, which typically ranges from 50 to 350. The actual dimensionality of the CAN, however, is much lower because there are not enough nodes to partition all the dimensions of a high-dimensional CAN. Along those unpartitioned dimensions, the search space is not reduced. Further, semantic vectors are not uniformly distributed in the semantic space. Due to the curse of dimensionality, it has been shown that limiting the search region in high-dimensional spaces is difficult. These problems are addressed by leveraging the properties of the semantic space and trading accuracy for efficiency and/or storage overhead when necessary. Taking advantage of the higher importance of low-dimensional elements of semantic vectors, a rolling index scheme partitions the semantic space along more dimensions by rotating the semantic vectors. A content-aware node bootstrapping helps distribute the indexes more evenly across nodes. Using samples of indexes and recently processed queries to guide the search, the content-directed search algorithm substantially reduces the search region in the high-dimensional semantic space.

A decentralized approach for building content-based overlays is proposed in [81, 80, 62]. The approach exploits the routing indexes maintained at each node. When a new node enters the p2p system, its local index is used to route the node towards the nodes that have similar routing indexes and thus similar content with it.

The authors in [63] propose *workload-aware content clustering*. The motivation is that clustering based solely on the content of the nodes does not take into account the query workload. For example, items that are rarely queried should affect the formation of content clusters less than popular items. Workload-aware clustering builds upon the approach in [80] that uses histogram-based routing indexes for constructing content-based overlays. Workload-awareness is achieved by using a weighted edit distance between the histograms serving as indexes of each peer to define the similarity of the peers' content. The weight is proportional to the estimated popularity of the corresponding items.

4.3.2 Semantic Overlays through Caching

Content Caching An example of a cached-based approach for building semantic overlays is caching in Freenet [109]. Freenet creates caches of similar content. This is achieved through the cache replacement policy used in Freenet. When a node n enters the systems, it chooses a seed, say a, randomly from the key space. By doing so, node n becomes responsible for building a cluster for a. When the cache of node n is full and a new item with key b arrives, b replaces the item in the cache that is the farthest apart from a. In addition, if b is closest to a than the item that was evicted from the cache, an entry for b is added to the routing table. An enhanced version, coined *enforced-cluster with random shortcuts*, adds a few extra links to the routing table to peers not similar to a.

Locality of Interest Caching The approach described in [96] builds an *interest-based shortcut overlay* of peers with similar interests based on the premise that there is *locality of interest* among peers, so that peers used in the past by a peer to provide results to one of its queries are likely to provide results to future queries as well. The shortcut overlay is built on top of the p2p overlay, so that if an item cannot be located via shortcuts, it can always be located via the underlying p2p overlay.

Shared interests are located based on the premise that peers that have content that a peer looks for share similar interests with it. Shortcut discovery is through search. Each peer creates a list of shortcuts incrementally: shortcuts are added and removed from the list based on their perceived utility. If shortcuts are useful, they are ranked at the top of the list. A peer locates content by sequentially asking all of the shortcuts on its list, starting from the top, until content is found. Rankings can be based on many metrics, such as probability of providing content, latency of the path to the shortcut, available bandwidth of the path, amount of content at the shortcut, and load at the shortcut. A combination of such metrics can also be used. The authors explore the possibility of providing content (success rate) as a ranking (utility) metric. Success rate is defined as the ratio between the number of times a shortcut was used to successfully locate content to the total number of times it was tried. The higher the ratio of a shortcut, the higher the shortcut is ranked. Shortcuts that have the lowest ranking are removed from the shortcut list, when the list becomes full.

There are several design alternatives for shortcut discovery. New shortcuts may be discovered through exchanging shortcut lists between peers, or through establishing more sophisticated link structures for each content category similar to structures used by search engines. In addition, multiple shortcuts, as opposed to just one, may be added to the list at the same time.

The authors in [48] evaluate a number of policies for ranking the entries in the shortcut cache using real traces collected from a peer-to-peer file sharing system [40]. In the *LRU-policy*, the peer that was used most recently is placed at the top of the cache, while the least recently used peer is the one removed from the cache, if needed. To locate an item, the x most recently used peers are contacted first. In the *history-based policy* [102], a counter is maintained along with each peer in the cache indicating how many times the peer was used over a time frame. To locate an item, the x most useful peers over the time frame are contacted first. Results show that both LRU and history-based shortcuts provide a hit ratio close to 30% even when only 5 peers are contacted, with the history-based approach having a slightly better hit ratio. The hit ratio is influenced by the existence of a few "generous" downloaders, that is, peers that provide the bulk of popular files. However, experiments showed that semantic clustering exists, since even when this generous peers were removed from the cache, there were still hits.

The authors in [48] also consider whether locality of interest is transitive. Specifically, first, each peer contacts the peers in its cache. If this fails, it uses their caches. Results show that the results attained when a peer contacts peers in the cache of its semantic neighbors are comparable to those attained when the peer just contacts peers in its own cache. This indicates that semantic links tend to automatically cluster semantically related peers, so that a one-level cache is sufficient. Finally, the authors show that maintaining a content cache, that is, a list of useful peers per file type, improves hit ratio.

4.3.3 Associative Overlays

Associative overlays [27] are based around the notion of a guide rule. A *guide rule* is a set of peers that satisfy some predicate. For each guide rule that a peer belongs to, it maintains a small list of other peers that belong to the same rule. A guided-search query is propagated only to the guide-rules specified by the originating peer, and thus yields more effective search probes. Inside a guide rule, a query performs a blind search.

Guide rules can be defined on properties extracted automatically or specified by the user. The authors focus on automatically extracted guide rules of a very particular form, which they call possession rules. Each *possession rule* has a corresponding data item, and its predicate is the presence of the item in the local index, thus, a peer can participate in a rule only if it shares the corresponding item. The premise is that on average, peers that share items (in particular rare items) are more likely to satisfy each others' queries than random peers. More precisely, search using possession-rules exploits presence of pairwise co-location associations between items.

For choosing which possession rules to use, two search algorithms are proposed: RAPIER and GAS. RAPIER (Random Possession Rule) selects a possession-rule uniformly at random from the list of previously-requested items by the querying peer. RAPIER works better than blind search, if there is correlation between items. The random choice of a guide rule is clearly non-optimal. Some guide rules may contribute much more to the probability of a successful search than others. GAS (Greedy Guide Rule) is a theoretically-grounded strategy, in which each peer prefers to invoke rules that would have been more effective on its past selections. GAS approximates the strategy that would have performed best on all previous queries.

4.4 Replication and Caching

Maintaining multiple copies of data items is a commonly used mechanism for improving the performance and fault-tolerance of any distributed system. By placing copies of data items closer to their requesters, the response time of queries can be improved. An additional reason for replication is load balancing. For instance, by allocating many copies to popular data items, the query load can be evenly distributed among the servers that hold these copies. Similarly, by eliminating hotspots, replication can lead to a better distribution of the communication load over the network links. Further, in the case of overlays, replication can be used to control the associated routing load. Besides performance related reasons, replication improves system availability, since the more the copies of an item, the more site failures can be tolerated. Maintaining copies cal also improve the durability of items.

There are many issues related to replication. One issue is what to replicate. In p2p systems, one may either replicate data items or index (catalog information). Another issue is the granularity of replication. Further, a replication protocol should specify where and when to replicate items.

The cost associated with replication includes the storage cost as well as the maintenance cost in the case of updates. Updates are either initiated by the owner of the copy that was updated (push-based updates) or by the holder of the copy (pull-based updates). Often entries are associated with a time-out value, whose expiration signals the removal of the entry, thus providing only soft-state consistency.

Replication depends on the query workload and the system availability. It also depends on the underlying topology.

Caching vs Replication The difference between caching and replication is in general subtle. Caching is usually initiated at the clients, in our case, the peers that made the request for an item. Typically, in a client-server setting, the client (requester) keeps a copy of the item, so that it can subsequently use it. In p2p systems, this copy may be used by others peers as well. Replication is a server-based decision, in the sense that, the server decides to make copies keeping statistics about requests received by its clients. Thus, replication is typically proactive. Copies in the case of caching are implicitly removed by some replacement policy, say LRU. In the case of replication, removing copies is a more global decision, in general involving a larger set of nodes. Finally, in the case of replication, information about the location of the replicas is maintained in catalogs, whereas usually cached copies are not indexed.

Erasure Coding An *erasure code* provides redundancy without the overhead of strict replication. Erasure codes divide an object into m fragments and recode them into n fragments, where $n > m$. The ratio $\frac{n}{m}$ is called the *rate* of encoding. A rate r code increases the storage cost by a factor of $\frac{1}{r}$. The key property of erasure codes is that the original object can be reconstructed from any m fragments.

The authors of [105] quantify the availability gained using erasure codes. Then, they show that erasure-resilient codes use an order of magnitude less bandwidth and storage than replication for systems with similar mean time to failure (MTTF). They also show that employing erasure-resilient codes increase the MTTF of the system by many orders of magnitude over simple replication with the same storage overhead and repair times.

The authors of [88] argue that while gains from coding exist, they are highly dependent on the characteristics of the nodes that comprise the DHT overlay. In fact, the benefits of coding are so limited in some cases that they can easily be outweighed by some disadvantages and the extra complexity of erasure codes.

4.4.1 Unstructured Overlays

The authors of [28] consider the general problem of what is the best way to replicate data in unstructured p2p systems given that the total amount of storage in the network is fixed. In particular, there are n nodes, m data items and a total budget of $R > m$ copies overall. Each data item i has a query frequency q_i. The replica allocation problem refers to determining the fraction p_i of the total budget R to be alloted to item i.

Two natural ways to perform replication is uniform and proportional replication. With *uniform replication*, the same number of copies is created for all items, that is p_i is the same for all items, while with *proportional replication*, the number of copies created for each item i is proportional to the item's popularity, i.e., to its query rate q_i. It is shown that both replication strategies have the same expected search size for successful queries. However, they differ in other aspects. Proportional replication distributes the load evenly to all copies, whereas in the case of uniform replication, copies receive load proportional to their query rates. Proportional replication also makes popular items easier to find, at the expense of making less popular ones harder to find. Thus, with proportional replication, a much higher limit (TTL value) is required for locating them. On the other hand, uniform allocation minimizes this limit.

It is also shown that in terms of search size, uniform and proportional replication lie at two extremes where the ratio of allocation of two items is between 1 and the ratio of their query rates. All replication strategies that lie between these extremes yield better search sizes for successful queries with square-root replication achieving the optimal such size. *Square root replication* allocates replicas to items proportional to the square root of their query rate, $(\sqrt(q_i))$. The gain attained with square root replication grows with the query skew.

To strike a balance between soluble and insoluble queries, assume that we fix the resources consumed by insoluble queries by fixing the set of locatable queries and the bound in the maximum search size. *Square-root* replication* lies between square root and uniform replication and allocates a sufficient minimum number of copies to locatable data item so that the search size is bounded. Since square-root* replication minimizes the search size while fixing the maximum search size, by sweeping the lower bound on the allocation of any one item, we obtain a range of optimal strategies for any given ratio of soluble and insoluble queries. The extremes of these ranges are uniform allocation which is optimal when insoluble queries completely dominate and square-root allocation which is optimal when there are relatively few insoluble queries.

The authors also consider ways of implementing square root replication using simple distributed protocols. They propose three such protocols: path replication, replication with sibling-number memory and replication with probe memory.

Path replication sets the number of copies to be the size of the search, that is, the number of nodes probed. A possible disadvantage of path replication is that the current number of copies may "overshoot" or "undershoot" the fixed point by a large factor. Thus, if queries arrive in large bursts or if the time between search and subsequent copy generation is large compared to the query rate then the number of copies can fluctuate and never reach the fix point. *Replication with sibling-number memory (SNM)* attempts to remedy this by additional bookkeeping. Under SNM, with each new copy, we record the number of sibling copies that were generated when it was generated and its generation time. The algorithm also assumes that each node knows the historic lifetime distribution as a function of the age of the copy. Thus, the age of the copy provides the sibling survival rate. With *replication with probe memory*, each node records for each item it had seen at least one probe for, the total number of probes it received and the combined size of the searches which these searches were part of. Then, when searching for item i, the total number of recent probes for i seen on nodes in the search path is accumulated. If this count is lower than the threshold, this suggests that item i is over-replicated with respect to square-root replication. Otherwise, it is under-replicated.

Steady state is achieved when the replica creation rate equals the deletion rate. To achieve this, the lifetime of replicas must be independent of object identity or query rate. Examples of deletion policies that have this property include assigning fixed lifetimes for each replica, random deletions and First In First Out (FIFO) replacement. However, popular policies such as Least Recently Used (LRU) or Least Frequently Used (LFU) do not have this independence property.

When compared with optimal replication, the SNM algorithm arrives more quickly to square-root replication and is not sensitive to delayed creation of followup copies.

If we fix the set of locatable items, that is, the items for which we want queries to be soluble, the optimal algorithm is a hybrid of uniform and square-root. For each locatable item, the uniform component assigns a number of permanent copies. The square-root component - which can be implemented using any of the three algorithms previously presented - generates transient copies as results of searches. The maximum search size (TTL) is set together with the number of permanent copies so that items with this minimum number of copies would be locatable. The value of these two dependent parameters is then tuned according to the mix of soluble and insoluble queries to obtain the optimal balance between the cost of soluble and insoluble queries.

The authors of [70] provide an evaluation of two easily implementable replication strategies, namely owner and path replication, under a realistic setting. Owner replication is used in system such as Gnutella, while path replication is used in systems such as Freenet. Under *owner replication*, when a search is successful, the object is replicated at the requester node only. *Path replication* is implemented by replicating the object on all nodes on the search path from the requester node to the provider node. The evaluation of these two strategies is done in conjunction with a k random walkers search strategy. In this case, the number of nodes between the requester node and the provider node is $1/k$ of the total number of nodes visited. Since path replication tends to replicate objects to nodes that are topologically along the same path, the authors also consider a third replication strategy called random replication. *Random replication* counts the number of nodes on the path between the requester and the provider, say p, then, randomly picks p of the nodes that the k walkers visited and replicates the object on them. The evaluation of the three allocation strategies is done on a random graph network topology. The replica allocation achieved by both path and random replication is as expected quite close to square-root replication. Both replication strategies reduce the overall traffic by a factor of three to four mainly by reducing the search size. Random replication improves over path replication for the cost of a more involved implementation and a higher cost for creating replicas. In general, path replication distributes query load for popular items across multiple nodes, reduces latency and alleviates hot spots.

In Freenet, each node maintains a routing table whose entries are pairs of the form (*key*, *pointer*) where *pointer* is the node that maintains a copy of the file associated with *key*. At each node, a request for a file with key k is routed to the node in the routing table whose key is the closest, i.e, most similar, to k. After a successful request, the associated routing table entry (k, *k_owner*) is cached at all nodes on the path from the owner of the file to the requester node, following path replication. In addition, each node on the path caches the file itself on its own datastore. When a file causes the datastore to exceed its size, the Least Recently Used (LRU) files are evicted to make room for it. Routing table entries are also replaced using an LRU policy. The size of the routing table is chosen with the intention that the entry for a file will be retained longer than the file itself.

It was shown that under low load the resulting network bares the characteristics of a small world. However, this does not hold for larger loads. An enhanced cache replacement policy to support the desired clustering was proposed in [109]. The goal of the enhanced-clustering cache replacement is to create caches of similar content. First, when a node n enters the systems, it chooses a seed $s(n)$ randomly from the key space. When the cache of node n is full and a new file with key u arrives, then, let v be the key that is the farthest from the seed from the keys of all files in the datastore, v is replaced by u. In addition, if $distance(u, s(n)) \leq distance(v, s(n))$, an entry for u is added in the routing table. A variation of the above scheme is also proposed, called enforced-clustered with random shortcuts. This method creates an entry for u in the routing table also in the case of $distance(u, s(n)) > distance(v, s(n))$ with probability p (randomness). A good value of p seems to be 0.03.

4.4.2 Structured Overlays

Replication in DHT for availability is achieved by replicating at the k servers immediately after the item's successor say on the Chord ring. Servers close to each other on the ID ring are not likely to be physically close to each other, since the ID is based on a hash of its IP address. This provides the desired independence of failures. Besides availability, these replicas can be used to improve query latency by allowing to choose the replica holder with the lowest reported latency. This works best when proximity in the underlying network is transitive. Fetching from the lowest-latency replica has the side-effect of spreading the load of serving a lookup over the replicas. Caching in DHTs is based on placing copies on the lookup path.

[34] discuss CFS a storage layer built on top of a DHT, namely Chord. CFS provides distinct mechanisms for replication and caching. Both caching and replication are performed at a file block level. CFS places a block's replicas at the r servers immediately after the block's successor on the Chord ring. The placement of block replicas makes it easy for a client to select the replica likely to be fastest to download. CFS also caches blocks to avoid overloading servers that hold popular data. A block is cached at all nodes on the search path after each successful look-up. Cached blocks are replaced in least-recently-used order. This has the effect of preserving the cached copies close to the successor, and expands and contracts the degree of caching for each block according to its popularity.

The work in [45] primarily addresses replication for load balance in DHTs. This is an alternative to attempts to balance load by using cryptographic hashes to randomize the mapping between data item names and locations. Under an assumption of uniform demand for all data items, the number of items retrieved from each server as well as the routing load incurred by servers in these systems will be balanced. However, if demand for individual data items is nonuniform, neither routing nor destination load will be balanced, and indeed may be arbitrarily bad. Instead of creating replicas on all nodes on a source-destination path, the protocol relies on individual server load measurements to precisely choose replication points. The routing process is augmented with lightweight hints that shortcut the original routing and direct queries towards new replicas.

Beehive [84] is a general replication framework that operates on top of any DHT that uses prefix-routing. The central observation behind Beehive is that the length of the average query path will be reduced by one hop when an object is proactively replicated at all nodes logically preceding that node on all query paths. For example, replicating the object at all nodes one hop prior to the home-node decreases the lookup latency by one hop. This can be applied iteratively to disseminate objects widely throughout the system. Replicating an object at all nodes k hops or lesser from the home node will reduce the lookup latency by k hops. Beehive controls the extent of replication in the system by assigning a replication level to each object. An object at level i is replicated on all nodes that have at least i matching prefixes with the object. Queries to objects replicated at level i incur a lookup latency of at most i hops. Objects stored only at their home nodes are at level $log_b(N)$, while objects replicated at level 0 are cached at all the nodes in the system. The goal of Beehive's replication strategy is to find the minimal replication level for each object such that the average lookup performance for the system is a constant number of hops. Naturally, the optimal strategy involves replicating more popular objects at lower levels (on more nodes) and less popular objects at higher levels.

An analytical model provides Beehive with closed-form optimal solutions indicating the appropriate levels of replication for each object. Second, a monitoring protocol based on local measurements and limited aggregation estimates relative object popularity, and the global properties of the query distribution. These estimates are used, independently and in a distributed fashion, as inputs to the analytical model which yields the locally desired level of replication for each object. Finally, a replication protocol proactively makes copies of the desired objects around the network.

[25] builds a multicast tree of replicas on top of a DHT, namely Tapestry. The goal is to place replicas so that to satisfy both capacity constraints and latency requirements.

4.4.3 Updates

Updating cache entries and replicas in a p2p system mainly aims at providing soft-state guarantees. Often each entry is associated with an expiration time. Further, flooding-style rumor spreading protocols have been used to propagate replica updates. Such protocols consider both push and pull based propagation. With push, the node where the update originates propagates the update. With pull, the node that holds a copy of an item, initiates a look-up for any updates.

A basic issue in these protocols is when should a peer pull. Pulling too often creates substantial message overhead. Pulling infrequently may result in missing important updates. Adaptive pull strategies try to minimize the communication overhead, while maintaining good consistency levels by having each replica holder pull at specific intervals. These intervals are determined by a time-to-refresh (TTR) parameter, which is adaptively adjusted depending on the previous pull results [95]. If after the last pull the item was found unchanged, TTR is increased so as to pull less frequently; otherwise, TTR is decreased so as to check for updates more often.

The work in [35] consider a p2p system where peers have low online probabilities and only local knowledge. The update strategy proposed for this environment is based on a hybrid push/pull rumor spreading algorithm which offers probabilistic guarantees rather than ensuring strict consistency. A generic analytical model is developed to investigate the utility of this hybrid update propagation scheme from the perspective of communication overhead.

The work in [67] also considers a hybrid update propagation method for metadata. The proposed method combines periodic pull requests and on-demand pull requests initiated when a copy is found to be invalid. For push based updates, various variations of flooding are studied including a novel variation of flooding, where updates of an item are temporarily stored in the neighborhood of the initiator of the update.

The authors of [89] consider how to maintain the cache entries in the case of path replication, where metadata are cached at intermediate nodes that lie on the path taken by a search query. This is called *Path Caching with Expiration* (PCX) because cached metadata entries typically have expiration times after which they are considered stale and require a new search. The cache maintenance problem is challenging because the global set of valid metadata changes constantly as peer nodes join and leave the network, content is added to and deleted from the network, and replicas of existing content are added to alleviate bandwidth congestion at nodes holding the content. Nodes that cache metadata to serve queries in a more timely fashion need to know about changes to the metadata to serve queries better. Keeping cached metadata up-to-date therefore requires tracking which metadata items need to be updated, as well as tracking when interest in updating particular items at each cache has subsided to avoid unnecessary update propagation for the maintenance of these items. The proposed *Controlled Update Propagation Protocol*, CUP, moderates update propagation, by allowing each node to receive and propagate updates only when it has an economic incentive to do so. This occurs when the investment return (or benefit) a node secures by propagation outweighs the cost of propagation and thus, all overhead is recovered. The first class of policies is probabilistic where a node computes the probability that a received update is justified using an estimate of the number of nodes that depend on this node for answers to queries for the item. The second class is history-based, where the node compares the ratio of query arrivals to update arrivals in a sliding window of update arrivals. These policies favor the receipt of updates for popular items since these items generate queries most often.

4.4.4 Other Issues

Proactive Replication A common way of maintaining replicas for durability is to detect node failures and respond by creating additional copies of objects that were stored on failed nodes and hence suffered a loss of redundancy. Reactive techniques can minimize total bytes sent since they only create replicas as needed; however, they can create spikes in network use after a failure. These spikes may overwhelm application traffic and can make it difficult to provision bandwidth. In [93], a proactive approach is proposed that creates additional copies not in response to failures, but periodically at a fixed low rate. A distributed hash table, called Tempo, is introduced that allows each user to specify a maximum maintenance bandwidth and uses it to perform proactive replication.

View Materialization When a query is not simply a keyword-one, we may cache the result of the query along with the query itself. This is similar to view materialization. There is not much research done in the context of p2p case but for the simple case of conjunctive and disjunctive keyword-value queries. For example, assume that indexes a_i, a_j, and a_k are located on distinct nodes in the network. Computing $a_j \wedge a_i \wedge a_k$ directly from these indexes is much more expensive than intersecting the result of a prior $a_i \wedge a_j$ operation together with a_k. The authors of [14] propose a data structure, the *view tree*, that can be used to store and retrieve such prior results.

4.4.5 Replication and Quality-of-Service

Replication and caching are two of the most important mechanisms for achieving *Quality of Service (QoS)* in a p2p environment. Quality of Service is a notion that is widely used in many settings. Here we focus on those aspects of quality of service that we consider relevant to p2p global computing. These quality of service requirements are defined in more detail in AEOLUS deliverable D3.1.1 [8]. In this section, we position them in terms of replication and indicate how the research proposals surveyed have addressed them.

Replication increases the number of copies for each shared piece of data in the system. By doing so, the probability that some or all the data is temporarily or permanently lost significantly decreases, thus the *dependability* of the system in terms of *reliability* and *availability* is increased. Additionally, by having more copies for popular data items, the load for routing and answering queries can be evenly distributed among the servers that hold the copies. This way, the *performance* of query processing is increased in terms of *throughput* and *response time*, since congestions in "hot" servers are avoided. 'Moving" data closer to their requesters as done by replication also improves the performance of query processing. Caching makes query processing even faster since cached queries can be locally answered further improving response time and consequently throughput. Finally, *performability* combines availability and performance by considering the impact of transient failures to system performance; it captures performance measures conditioned by the probability distribution of resource availability.

Replication and caching may also improve *data recall.* In structured overlays data recall is not an issue contrary to unstructured overlays. While unstructured overlays, which adopt flooding-based techniques, are effective for locating popular data, they are poorly suited for locating rare data. Thus, by replicating the rare data the probability of locating the data during query processing increases, consequently increasing data recall. However, replication may affect *datafreshness.* P2p systems consist of autonomous peers that can arbitrarily delete or update their content, thus the replicated or cached data can become stale if not updated properly. Updating replicas and cache entries in a p2p system mainly aims at providing soft-state guarantees. Hence, query processing might encounter out-of-date copies of data, thus failing to achieve result freshness.

We provide next a short summary of the replication approaches previously described from the perspective of quality of service. The technique proposed in [14] uses result-caching and manages to increase the performance of the system, in terms of throughput and response time.

The performance of a system can also be improved by using a smart cache replacement policy. [109] shows that an enhanced clustering cache replacement scheme can reduce the query latency in Freenet, since it maintains a small average number of hops per successful request. [25] builds a dynamic content distribution system, called the dissemination tree. The system achieves good load balance and small delay penalties. This way, the response time is reduced and the data freshness is ensured, since update propagation is fulfilled in short time.

[28] and [70] describe replication strategies that achieve square root replication. These strategies increase the dependability and the performance of a system, in terms of availability and response time respectively. The CFS storage system discussed in [34] provides mechanisms for replication and caching. The algorithms that are used increase efficiency, achieve load balance and low latencies and ensure the reliability and robustness of the system. In [45] an adaptive replication protocol (LAR) is described. LAR is efficient at redistributing load and improves query latency and reliability. In addition, it is robust in the face of widely varying input and underlying system organization.

Beehive [84] is a proactive replication framework that can provide constant lookup performance for common Zipf-like query distributions. As simulations show, Beehive can achieve good latencies and ensure robustness since it adapts efficiently to sudden changes in object popularity. Another proactive replication approach is described in [93]. The approach increases the availability by periodically replicating data, at a fixed low rate. This periodic replication also makes the system reliable and robust in the case of failures. [105] provides a comparison between erasure coding and replication. Systems that employ

Table 2: Replication techniques and QoS goals

				Response		Result	
Approach	Reliability	Availability	Throughput	Time	Performability	Quality	Robustness
[14]			√	√			√
[25]			√	√		√	
[28]		√	√	√			
[34]	√	√	√	√	√		√
[45]	√			√			√
[70]		√		√			
[84]			√	√		√	√
[93]	√	√					√
[105]		√			√		
[109]				√			

erasure-resilient codes can increase availability of data, by increasing the mean time to failure.

Table 2 summarizes the focus of each of the above replication and caching approaches in terms of QoS.

5 Query Processing

Most of the research work in p2p systems has focused on processing simple keyword-value queries. In this part of the survey, we shall discuss the processing of more advanced queries for data. Data may be numeric (multi-dimensional or single dimensional) and may have structure, such as in the case of XML documents. Data may also follow a schema or be schema-less.

There are many ways to define a p2p database system: In terms of distribution:

- There are may be a single storage or the data may be distributed at many nodes (using either structured or unstructured data partitioning).

- There may be one or more centralized catalog/index or a distributed (either structured or unstructured) one.

In terms of information disclosure, an important issue is whether we know which data exist or do we have to search both at the logical level (schema) and at the physical level (data).

Normally, query processing in centralized database systems proceeds in the following three phases: (1) parse (2) re-write and optimize, and (3) execute. In the first phase, the query is parsed and translated into an internal representation. During the query rewriting phase, a query is transformed by carrying out optimizations that are good independently of the physical state of the system, for example, independently of the size of the relations or the presence of indexes. The rewritten query is then optimized using information from the system catalog including various statistics. The output of the query optimizer is a query plan that specifies precisely how the query is to be executed. Finally, the query is executed by the query execution engine. Typically, the execution engine provides generic implementations of every basic operator.

In adaptive query processing, the optimization and execution steps of processing a query are interleaved, possibly many times, over the execution of a query. The goal of adaptive query processing is to make query processing more robust to optimizer mistakes, unknown statistics and changes in input characteristics and system conditions.

Analogously, the normal steps in distributed query processing are: (1) parsing, (2) localization, by replacing global relations by appropriate combinations of their local counterparts, (3) re-writing and global optimization and (4) local optimization and execution at each single site. A central question is how these steps should be performed in a p2p setting and especially localization, since there may be no globally available catalog information.

So far, the focus has been on processing each query independently. For example, PIER suggests using distinct routing trees for each query in the system, in order to balance the network load. There is also the possibility of *multiquery optimization* that considers optimizing the execution for a set of queries, for instance by sharing computation and communication cost among them.

6 Processing of Relational Operators

6.1 Relational Queries

A query engine for executing complex queries on top of DHT-based P2P systems is described in [49]. Although the approach introduces database facilities in P2P systems, the database storage semantics are relaxed since there is no need for data to be loaded into a database. Instead, sets of P2P files are manipulated as relations, allowing the query engine to query the natural attributes of data, such as the name, id and host, intelligently while providing the storage semantics that users are common with. The system consists of three layers. The local data store that provides mechanisms for scanning sets of objects and accessing their natural attributes, the DHT layer that is used as an indexing and routing mechanism and the query processing layer that provides two query interfaces, an SQL one and a graph scripting one. In order to manage multiple data structures, required for query processing, a hierarchical name space is implemented, on top of the flat identifier space provided by the DHT, by partitioning the identifiers in multiple fields and then have each field identify objects of the same granularity. The routing protocol is also modified in order to be able to route queries to a desired subset of nodes. A node forwards a query to all neighbors that make progress in the identifier space towards any of the identifiers covered by the query.

The basic idea of [49] is materialized in PIER [53]. PIER is a massively distributed query engine based on overlay networks, which is intended to bring database query processing facilities to widely distributed environments. It is built on top of a DHT, which is divided into 3 modules, the Routing Layer, the Storage Manager and the Provider. The architecture of PIER is shown in Fig. 13. An instance of each DHT and PIER component runs at each participating node. The Routing layer provides basic functions for executing lookup, join and leave operations and a *locationMapChange* function provided to notify higher levels asynchronously when the set of keys mapped locally has changed. The storage manager is responsible for the temporary storage of DHT-based data and provides

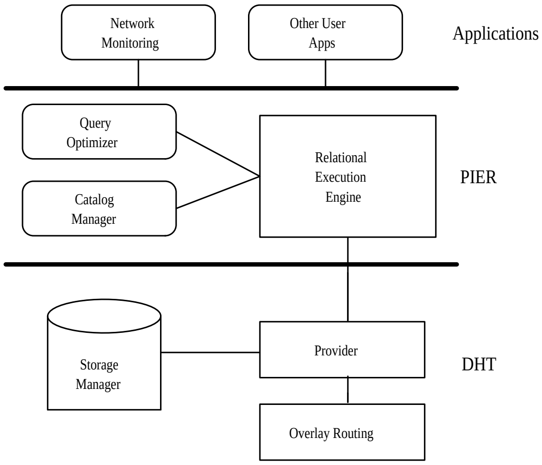


Figure 13: The architecture of PIER

functions for storing, retrieving and removing an item. Finally, the provider is responsible for tying the routing layer and storage manager together while providing a useful interface to applications.

Each object in the DHT has a namespace, resourceID, and instanceID. The namespace and resourceID are used to calculate the DHT key, via a hash function. The namespace identifies the application or group an object belongs to. For query processing, each namespace corresponds to a relation. Namespaces do not need to be predefined, they are created implicitly when the first item is put and destroyed when the last item expires. The resourceID is generally intended to be a value that carries some semantic meaning about the object. The query processor by default assigns the resourceID to be the value of the primary key for base tuples. Items with the same namespace and resourceID will have the same key and thus map to the same node. The instanceID is an integer randomly assigned by the user application, which serves to allow the storage manager to separate items with the same namespace and resourceID. During the query process, PIER attempts to contact the nodes that hold data in a particular namespace. A multicast communication primitive is used by the provider for this purpose. The provider supports scan access to all the data stored locally on the node through an *lscan* iterator. When run in parallel on all nodes serving a particular namespace, this serves the purpose of scanning a relation. Finally, the provider supports the *newData* function used to inform the application when a new data item has arrived in a particular namespace.

PIER is also used in [68], where a hybrid search infrastructure is described. In that approach flooding is used for highly replicated items, while *PIERSearch* engine processes queries for rare items. PIERSearch is a DHT-based search engine implemented using PIER and consists of two main components, the *publisher* and the *search engine*. For each item in the system, the publisher generates tuples conforming to a schema consisting of two tables, the *Item* table and the *Inverted* table. The Item table contains a tuple for each item that is being shared, having the *fileID* as a unique file identifier of the item which is also used as the index key for the DHT. For each keyword of an item, the Inverted table contains a tuple with the keyword and the *fileID* of the item. Inverted tuples with

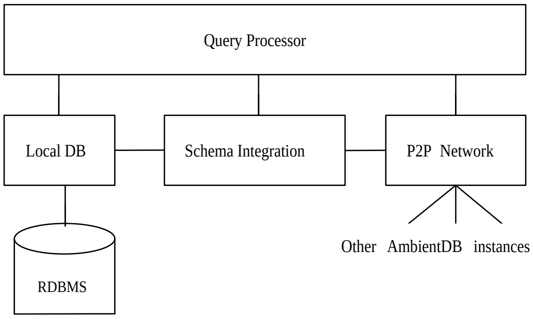


Figure 14: The architecture of AmbientDB

the same keyword are hosted by the same node. For a given keyword query, the Search Engine forms a relational query which is executed by its local PIER engine on its behalf. In particular, for a two-keyword query, the corresponding relational plan retrieves two sets of Inverted tuples, one for each keyword and executes a join of the two sets of tuples by *fileIDs*. Item tuples with the resulting *fileIDs* form the answer set. This query plan can be extended for queries with more than two search keywords simply by adding an extra self-join with the Inverted relation for each additional keyword. When such a query is executed, PIER routes the query plan via the DHT to all sites that host a keyword in the query and executes a distributed join. The node that hosts the first keyword in the query plan sends the matching Inverted tuples to the node that hosts the next keyword. The receiving node performs a symmetric hash join (SHJ) between the incoming tuples and its local matching tuples and sends the results to the next node. On the node hosting the last keyword in the query plan, the matching fileIDs are streamed back to the query node, which fetches the Item tuples from the DHT based on the incoming *fileIDs*.

AmbientDB [16] is a query processing architecture for executing queries in a declarative, optimizable language, over an ad-hoc P2P network. The system consists of the *distributed query processor*, allowing the execution of queries on all connected nodes, the *P2P protocol* used to connect all AmbinetDB nodes, the *local DB component*, where each node may store its local tables and the *schema integration engine* that couples different data delivered by different nodes. The architecture of AmbientDB is shown in Fig. 14. AmbientDB provides a standard relational data model and a standard relational algebra as query language. Query execution is performed by a three level translation. Initially, a user query is posed in an abstract global algebra, which is a relational algebra, providing the operators for selection, join, aggregation and sort. During the second phase, the abstract table types of an abstract query plan are translated to concrete types, starting from the leaves and continuing to the root of the query graph, which is required to yield a local result table. The third phase consists of translating the concrete query plan into wave plans, where each individual concrete operator maps onto one or more waves. *Waves* are tuple streams that can move either upward or downward in the routing tree used to connect all nodes through TCP/IP connections. Each node may receive tuples from its parent, process them with regards to local data and propagate data to its children. It may also receive data from its children, merge them with each other as well as with local data and pass the resulting tuples back to the parent. Each wave, in turn consists of a graph

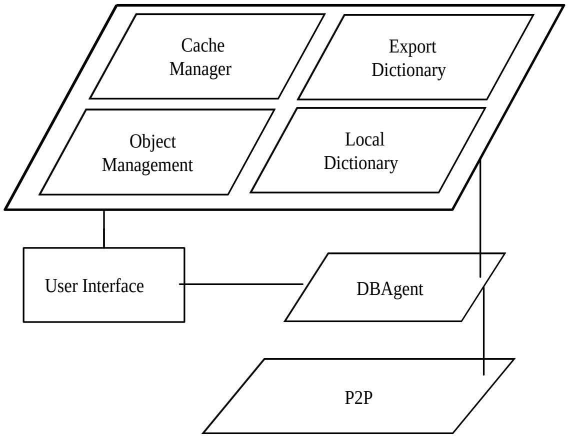


Figure 15: The architecture of PeerDB

of dataflow algebra operators. Dataflow operators may read multiple tuple streams but always produce just one output stream, such that each wave-plan can be executed by one separate thread.

PeerDB [78] is a prototype peer-based data management system that allows relational data sharing. It employs an IR approach to discover matching relations and agents to assist query processing. The system consists of three layers, the *P2P layer* that provides P2P capabilities, such as data exchanging and resource discovery, *the agent layer* that exploits agents and the *data management layer* that provides the data storage, processing capabilities and an SQL interface for users to pose queries. The architecture of PeerDB is shown in Fig. 15. For each relation created by the user, meta-data are maintained for each relation name and attributes in a *Local Dictionary.* These are essentially keywords/descriptions provided by the users upon creation of the table, and serve as a kind of synonymous names of relation names and attributes. There is also an *Export Dictionary* that reflects the meta-data of relations that are sharable to other nodes. The query processing strategy consists of two phases that are completely assisted by agents. In the first phase, agents are sent out to the peers to locate relations that are potentially similar to the query relations. These relations (meta-data, database name, and location) are then returned to the query node. In phase two, the query is directed to the nodes containing the selected relations and the answers are finally returned and cached.

6.2 Range Queries

Range queries involve queries for single or multi attribute data whose attribute values are contained within a range defined by the query. To support such queries in P2P systems data must be indexed in such way that the load is distributed equally between peers and that queries are processed efficiently. Thus, data is partitioned among peers so that each peer is responsible for a part of the whole data range. Single attribute/dimension data is assigned to the peer responsible for the data range containing the attribute value ([46], [111]). For multi attribute/dimension data different techniques are applied: (i) each attribute is indexed and queried separately ([13], [21], [82]), (ii) multiple attributes/dimensions are mapped to one attribute/dimension and are indexed and queried based on the mapping ([44], [23]), (iii) frequently queried attributes/dimensions are indexed separately and rarely queried attributes/dimensions are grouped and mapped to one attribute/dimension ([54]), (iv) all the attributes/values are indexed and queried together at the peer responsible for the multi dimensional region containing the region defined by the attributes/dimensions of the queried data ([55], [44], [90]). Since the goal is to efficiently answer range queries, data ranges/regions as defined above are mapped to peers in such a way that neighboring ranges/regions are indexed at related peers (neighbors, parent, children, etc.). The above mapping is achieved by use of overlay structures that support such mapping (trees, CAN, Skip Lists) or by modifying the hash function of some DHT (CHORD using Locality/Order Preserving Hash Functions). In general, query processing locates a peer holding a range/region that contains or intersects the queried range/region and proceeds by following links to related peers until the whole range is covered. Another approach is splitting the range defined by the query into sub-ranges covered by each peer and creating sub-queries for the sub-ranges. The answer of the range query is the union of the answers provided by the sub-queries.

The Distributed Segment Tree (DST) [111] incorporates a distributed tree structure above a DHT overlay for supporting single attribute range queries. The segment tree data structure is a full binary tree that represents a range of length L. Each node represents an interval$[s_l,k,t_l,k]$, ($l \in [0, logL]$ and $k \in [0, 2l - 1]$). Clearly, the root node interval equals to the segment tree range and the leaf node interval is one. Each non-leaf node has two two children whose ranges union covers the range of the parent node. All nodes in the same level span the whole range of the tree. Using a range splitting algorithm, the range of the segment tree is split into a minimum of node intervals and the DST is constructed. In DST, the segment tree structure is distributed onto DHT in a way similar to that adopted in PHT [23], the node interval [s, t] is assigned to the DHT node associated with the key Hash([s, t]). Thus, information about any node of the segment tree can be efficiently located through a DHT lookup operation. To insert a key into a DST, the given key is inserted to a specific leaf node and all the ancestors of that leaf node, because the node interval of any ancestor covers that specific key. A *downward load stripping* mechanism [111] is utilized so as to remove some keys from nodes that are overloaded, usually at the upper levels of the tree.

Given a range query [s, t], a node splits the range into a union of minimum node intervals of segment tree, using the range splitting algorithm. It then uses DHT lookup to retrieve the keys maintained on the corresponding DST nodes. The final query result is the union of the keys returned. Since the lookups are performed in parallel, the latency complexity is O(1). As an effect of the downward load splitting mechanism, a query may need to access more nodes in the lower levels of the tree so as to retrieve keys that have been deleted from overloaded nodes. Since this operation is done in parallel the performance loss is not very significant.

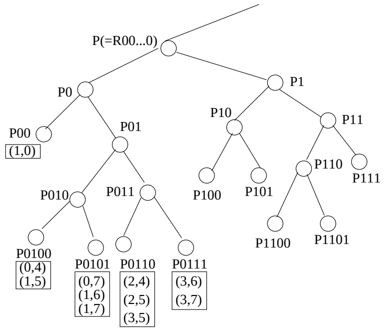
The work reported in [46] addresses the problem of answering range selection queries on single attribute data in P2P systems. The approach assumes that peers cache horizontal partitions of various relations and that a global schema is known to all the peers in the system. A peer can submit a query in the form of an SQL statement with the restriction that the selects on a relation can be only on one attribute at a time. The approach is based on CHORD and on Locality Sensitive Hashing. In particular, the peer nodes are hashed using a hash function, such as SHA-1 over their IP address into the identifier space. The range specifying a data partition is also hashed into the same identifier space using locality sensitive hashing. From the properties of locality sensitive hashing (LSH) similar ranges are hashed to the same identifier with high probability. Each data partition identifier i is mapped to the peer node with the least identifier greater than or equal to i in the circular identifier space. A peer is thus responsible for all hash buckets corresponding to identifiers from the identifier of its predecessor node (excluding it) to itself. To locate a given identifier, each peer in Chord also maintains information about other peers in the identifier ring that are at logarithmically increasing distances. Using this information, the peer holding a requested identifier is located and the bucket corresponding to the requested identifier is searched for the most similar range.

Mercury [13] is a scalable routing protocol for supporting multiattribute range queries. Mercury creates a routing hub for each attribute of the database schema. Each hub is a logical set of nodes, and so a physical node can participate in multiple logical hubs. Each new data file, according to its attributes, is sent to all the corresponding hubs. In reverse, each query that is referred to a set of attributes is propagated to only one of the corresponding hubs. A hub in Mercury organizes its nodes into a Chordlike ring. Each node in the hub is responsible for a range of values for the specific attribute. So, when a query is posed in a hub, it is routed to the node that is responsible for the first value of the range and via successor pointers that each node maintains, the query is spread along the ring, until it arrives to the node that is responsible for the last value of the range. Also, a node maintains pointers to its predecessor to keep consistent the ring with lower cost when a node's join or leave occurs.

MAAN (*Multi-Attribute Addressable Network*) [21] is an extension of Chord (see section 2.1.1). MAAN uses the Chord structure and the SHA1 hashing to assign an m-bit identifier to a node and to the attribute value with string type, but for attributes with numerical value, MAAN uses uniform locality preserving hash functions. MAAN uses the following routing algorithm in order to route single-attribute range queries. Suppose a node n submits a range query (l, v), with l and v being the lower and the upper bound, respectively. First, the query will be forwarded to node n_l, where l has been hashed to, as if n was querying for l in the Chord ring. Then, all the resources of n_l that satisfy the range query are gathered. Next, the query is forwarded to the immediate successor of node n_l in the ring and so on, until the query reaches node n_v, where v has been hashed to. Thus, routing a query in a Chord ring with n nodes requires $O(\log n)$ hops for forwarding the query to n_l and $O(k)$ hops for routing the query from n_l to n_v (assuming that there are k nodes between n_l and n_v), which gives a total routing cost of $O(\log n + k)$. MAAN also supports multi-attribute range queries. In the multi-attribute setting, data consists of a set of attributes and their respective values. MAAN registers attribute-value pairs and the resource information of that pair, by hashing each attribute value. When a node queries for interested resources, it composes a multi-attribute range query which is the combination of sub-queries on each attribute dimension. Next, all sub-queries are intersected at the query originator, in order to have a final answer.

In [82], an approach is presented that tries to ensure both access load balance and efficient range query processing in a DHT-based peer-to-peer network. HotRoD is introduced, a locality-preserving load-balancing DHT-based architecture that incorporates a novel locality-preserving hash function, used for data placement and a replication mechanism for popular values/ranges, aiming at distributing access load fairly among peers. Data objects are database tuples of a k-attribute relation R, placed in range partitions over an m-bit identifier space in an order-preserving way. Each peer keeps track of the number of times it was accessed during a time interval and the average low (avgLow) and high (avgHigh) bounds of the ranges of the queries it processed, at this time interval. A peer is considered to be overloaded or *hot* when its access count exceeds the upper limit of its resource capacity. In this case, the corresponding arc of peers (i.e. successive peers on the ring, which correspond to the range [avgLow, avgHigh]) is considered hot as well and is replicated and rotated over the identifier space, in order to reduce costly jumps during range query processing. Thus, the identifier space can be visualized as a number of replicated, rotated, and overlapping rings. A HotRoD instance consists of a regular CHORD ring and a number of virtual rings where values are addressed using a multi-rotation hash function, *mrhf()*. For every value v of attribute A, the multi-rotation hash function is defined as: $mrhf(v,\delta) = hash(v) + \mathrm{random}([\delta]\cdot s)\mathrm{mod}2^m$. $hash(v)$ is the hash function of the underlying DHT, δ is an index variable used to distinguish the different instances of A's values, $s = (1/p_{max(A)})\cdot 2m$, called the rotation unit, random$[1p_{max(A)}+1]$ is a pseudo-random permutation of the integers in $[1, p_{max(A)}]$ and $p_{max(A)}$ refers to the maximum number of instances that each value of attribute A can have. Thus, when the δ^{th} instance of a value v is created, or else the $(\delta-1)^{th}$ replica of v, it is placed on the peer whose identifier is closer to $mrhf(v,1)$ shifted by $\delta \cdot s$ clockwise. The algorithm used for the query processing of a range query [vlow(A), vhigh(A)], on attribute A (case of single-attribute relation R), initiated at peer p_{init} is the following. Initially, peer p_{init} randomly selects a number, r from 1 to $p(vlow(A))$, the current number of replicas of $vlow(A)$. Then, it forwards the query to peer p_l: $succ(mrhf(vlow(A),\ r))$. Peer p_l searches for matching tuples and forwards the query to its successor, p_2. Peer p_2 repeats similarly as long as it finds replicas of values of R at the current ring. Otherwise, p_2 forwards the range query to a (randomly selected) lower-level ring and repeats. Processing is completed when all values of R have been looked up.

SCRAP (*Space-filling Curves with Range Partitioning*) is another approach, proposed in [44], for supporting multidimensional range queries. SCRAP uses a two-step solution in order two partition data: (1) data are mapped down into a single dimension by using space-filling curve and (2) this single-dimensional data is range partitioned across a dynamic set of participant nodes. For instance, if there is a two-dimensional datum (i.e., a datum with two attributes), e.g. $\prec$x, y$\succ$, with x = 01001 and y = 10110, then one way to map it down into one dimension is to interleave the bits in x and y alternatively (z-ordering), thus $\prec$01001, 10110$\succ$ will become 0110010110. In the second step, data are partitioned among the participating nodes as follows. Every node is responsible for a continuous range of one-dimensional values. When a node is inserted into the system, the range of an existing node must be divided so that one half of this range will be appointed to the new node. When a node leaves the system, one of its neighbors takes over its range. Routing of multidimensional range queries in SCRAP is also done in two steps: (1) the multidimensional query is turned into a single-dimensional query and (2) each single-dimensional query is routed to nodes which are responsible for the relevant range that appears in the message. The first step is done by the use of well-known algorithms, such us the space-filling curves algorithm. The second step is done by using a well-known structured P2P topologies, such as skip-lists (see section 2.1.3). First, the node that is



PHT node label	(Lat,Lon)	(Lat,Lon) binary	Z−curve key
P00	(1,0)	(001,000)	000010
P0100	(0,4)	(000, 100)	010000
	(1,5)	(001,101)	010011
P0101	(0,7)	(000,111)	010101
	(1,6)	(001,110)	010110
	(1,7)	(001,111)	010111

Figure 16: (left) A sample two-dimensional PHT. (right) A table with the data items (and their z-curve keys)

responsible for the minimum value of the query's range is reached. Then, the nodes that are responsible for the rest of the range are found through that node's skip pointers.

The Place Lab [23] application supports queries over a two-dimensional latitude-longitude coordinate domain. To index this domain using PHTs, space-filling curves that map multi-dimensional data into a single dimension are used. All latitudes and longitudes are normalized and represented by a simple binary format. The z-curve linearization technique is used to map each two-dimensional data point into an one-dimensional key space. Z-curve linearization is performed by interleaving the bits of the binary representation of the latitude and longitude. This z-curve keys and their prefixes are used as the node labels.

For a one-dimensional PHT, given two keys L and H (L $\leq$ H), a range query L $\leq$K $\leq$ H is evaluated by locating the PHT node corresponding to the longest common prefix of L and H and then performing a parallel traversal of its subtree to retrieve all the desired items.

A multi-dimensional query for all matching data within a rectangular region defined by (latMin, lonMin) and (latMax, lonMax) is performed as follows. The linearized prefix that minimally encompasses the entire query region is determined. This is done by computing the z-curve keys zMin and zMax for the two end-points of the query, and the longest common prefix of these keys: zPrefix. Then, the PHT node corresponding to zPrefix is looked up and a parallel traversal of its sub-tree is performed. Unlike the simpler case of one-dimensional queries, not all nodes between the leaf for the minimum key and the leaf for the maximum key contribute to the query result. Hence, starting at the PHT node corresponding to zPrefix, if this node is a leaf node, then the range query is applied to all items within the node. If the node is an interior node, both its subtrees are check in parallel to evaluate whether they contribute any results to the query. This is done by determining whether there is any overlap in the rectangular region defined by the subtree's prefix and the range of the original query. If an overlap exists, the query is propagated recursively down the subtree. Thus the query algorithm requires no more than d sequential steps, where d is the depth of the tree.

In BATON* [54] a balanced search tree is maintained where each node has at most m children. Data is partitioned among peers by assigning at each node a range of values so that the range of values managed by a node is to the right of the ranges managed by its first $\lceil m/2 \rceil$ children and to the left of the ranges managed by its last $\lfloor m/2 \rfloor$ children. Each node maintains a set of links that includes a pair of links to adjacent nodes with ranges to the left and to the right of the range associated to the node. To answer a single attribute range query, initially the search is performed for a node whose range intersects with the query range. Then the query proceeds left and/or right to cover the remainder of the searched range, by following adjacent links until the whole query range is covered.

BATON* can also support queries over multiple attributes. The method for answering queries over multiple attributes is based on the observation that such queries, most of the time, involve only a small number of attributes. The range of data indexed by BATON* is divided into several sections used to separately index attributes that frequently appear in queries and groups of attributes that rarely appear in queries. To index a group of attributes their values are converted into one-dimensional values by use of Hilbert Space Filling Curve. Grouping rarely used attributes reduces the number of indexes created for the same data if each attribute is indexed separately. A multi-attribute query is defined as a set of subqueries, in which each subquery involves one attribute. The final result of the query is computed by answering one of the subqueries and using the rest of the subqueries as filters. If the selected subquery involves a separately indexed attribute answering it is straightforward. If the subquery involves an attribute indexed together with other attributes, the subquery is mapped into smaller subqueries (by using the Hilbert Space Filling Curve), which can be answered in parallel.

In VBI-Tree [55], a balanced binary tree is maintained for indexing multi-dimensional data. The attribute space is hierarchically partitioned among nodes so that each node is associated with a data region that contains the region associated to its children. A range query on multi-dimensional data defines a region in the attribute space. The algorithm for answering range queries in VBI-Tree seeks nodes with regions that intersect the query region. Thus, the query is firstly forwarded to a node whose region intersects with the query region. This node executes the query locally and forwards the query to other nodes whose region possibly intersects the query range by using the routing information it maintains. The same procedure is repeated to every node receiving the query avoiding forwarding a query to the node from which it initially received the query.

Another approach for supporting multidimensional range queries, proposed in [44], is MURK (*MUlti-dimensional Rectangulation with Kd-trees*). MURK partitions data by dividing the data space into "rectangles" and each node takes over a single rectangle. To achieve this, kd-trees are used, where each rectangle corresponds to a leaf in the kd-tree. Initially, there is only one node in the system, which is responsible for all data. Each time that a new node is inserted, the data space is partitioned in one of the dimensions in turn (as in CAN -see section 2.1.1, when a node joins the system). This kind of partitioning is done (1) for preserving the data locality in all dimensions and (2) so that the load is shared among all nodes equally. When a node leaves the system, then its space is taken over by a neighboring node and the kd-tree is reformulated. Routing in MURK is done as in CAN (see section 2.1.1). Every node is aware of the boundaries of its neighbors and it is connected with them via links. In that way, a grid-like structure is created. The query is forwarded to the node that reduces the Manhattan distance to the point where the data is located, by the largest amount. In that way, the routing cost for a uniform 2d grid of n nodes is $\Theta(\sqrt{n})$.

A method to evaluate range queries based on the multidimensional CAN system is

Table 3: Range queries in P2P

	Single or multi attribute/dimension	Attribute indexing	Overlay
[111]	Single attribute		Tree (DST)
[46]	Single attribute		CHORD
[13]	Multi attribute	Each attribute separately	CHORD
[21]	Multi attribute	Each attribute separately	CHORD
[82]	Multi attribute	Each attribute separately	CHORD
SCRAP [44]	Multi dimensional	Map to one dimension	SkipNet
[23]	Two dimensional	Map to one dimension	Tree (PHT)
[54]	Multi dimensional	Frequently queried attributes separately, rarely queried mapped to one dimension	Tree (BATON*)
[55]	Multi attribute	Region defined by all attributed	Tree (VBI)
MURK [44]	Multi dimensional	Region defined by all dimensions	Tree (kd-tree)
[90]	Multi dimensional	Zone defined by all dimensions	CAN

used in [90]. Nodes cache the results of range queries and use them to answer future range queries. The system maintains a global database schema that is known from all the nodes. Nodes store range partitions of the data files and cooperate with each other to answer queries, instead of asking direct the database. The system uses a 2d virtual space; two dimensions for each CAN dimension. The virtual space is partitioned among the nodes. A node is responsible for a part of the virtual space, accordingly to the range of the data files that stores. This part is called *zone*. A data file is referred to a range. This range is assigned to a point in the virtual space. This specific point belongs to a zone, and so the data file is stored to the node that is responsible for that zone. A query for a particular range is assigned to a point in the virtual space. The query is routed to the node that is responsible for the zone that includes this point. The routing is executed such in CAN. The query is first propagated to the neighbor that is closest to the point, and so on. When a node gets the result for its query, it caches the result to use it in the future, for itself or for another node.

Table 3 summarizes the approaches for processing range queries in p2p systems.

6.3 Top-k and Skyline

Top-k Queries In [76] a distributed top-k algorithm for peer-to-peer infrastructures is presented. The algorithm retrieves the k most relevant results for queries on RDF metadata, in a Super-peer (SP) network where Super-peers are arranged in the HyperCup topology. Each peer computes local rankings for a given query, results are merged and ranked again at the SPs and routed back to the query originator. On this way back each involved SP again merges results from local peers and neighboring SPs and forwards only the top-k results, until the aggregated top-k results reach the peer that issued the corresponding query. While results are routed through the SPs, the SPs maintain statistics which peers/SPs returned the best results. This information is subsequently used to directly route queries that were answered before mainly to those peers able to provide top answers. For local ranking, score aggregation and merging, topic similarity in taxonomy and TFxIDF methods are used.

The algorithm in [11]is similar to [76] with the difference that each peer sends only its top-1 result to its SP. The SPs do not merge the results from local peers and neighboring SPs but only forward the top-1 result. After the SP that issued the query has computed the current best result, repeats the procedure by routing the query to the peers that have contributed to the result produced so far, until the top-k results are computed. Statistics are maintained and used in the same manner as in [76].

Skyline Queries In [106] a distributed skyline query processing algorithm is presented. Data space is normalized and directly mapped to a d-dimensional CAN overlay, by using the normalized attribute values as coordinates. A constrained skyline query (Queries for skyline within a subset of records that satisfies multiple hard constrains. For example, find the cheapest hotel within the price range 100 to 200 which lies at the closest distance from the sea within distance 500m - 1000m) defines a query region in CAN, which is recursively partitioned so that each partition is covered by one CAN zone. The way data is mapped to CAN and a dynamic zone encoding, define a skyline partial order over the set of zones covering the partitions. With the aid of the partial order each zone (peer) in the query region is able to determine its final result based only on its own data and on the data from the zones that precede it in the order. This way participating peers are pipelined during query execution and as the query propagates progressively results are generated.

6.4 Similarity and Partial-match Queries

Similarity Queries A similarity preserving hash function is proposed [15] for indexing vector data into a DHT which, with high probability, maps vectors that are at a small distance from each other to keys that also are at a small distance from each other. A peer that has a query computes a set of keys which are at a distance r from the key of the query and queries all the nodes which own these keys using the lookup primitive supported by the underlying DHT. These nodes return objects that are within the similarity threshold defined by the query. The distance r is affected by the similarity threshold and the desired search accuracy.

Partial-match Queries In [110] a distributed index for efficiently answering full-text partial-match queries on peer-to-peer networks is proposed. The index, named Distributed Pattern Tree (DPTree), manages a tree hierarchy of popular query patterns. Every node of the DPTree is associated with a pattern, maintains an index to the list of documents matching the pattern and is represented by a cluster of strongly connected peers responsible for the pattern. All the keywords of the pattern of a parent node are contained in the pattern of the child node. The root of the DPTree is a pattern of a single keyword. The root nodes are positioned in the overlay using a DHT, while the single-word patterns that they manage serve as the key. During a series of query sessions, every DPTree node collects its query history and mines the frequent patterns periodically. The DPTrees, initially consisting of only roots, are then expanded dynamically as new frequent patterns are discovered. A keyword-based search starts from the roots whose pattern matches one of the keywords of the query and is propagated along the pattern trees until the patterns that best match the query are reached.

6.5 Aggregation Queries

A new information management service, called *Astrolable*, is described in [101]. Astrolable is used to monitor the dynamically changing state of a collection of distributed resources, stored in a P2P system, by continuously computing summaries of the data in the system using on-the-fly aggregation. Astrolable agents run on each peer and communicate with each other through an epidemic peer-to-peer protocol known as gossip. Periodically, each agent selects some other agent and exchanges state information with it. The resources are organized into a hierarchy of domains, called *zones*. The structure of Astrolabe's zones can be viewed as a tree. The leaves of this tree represent the peers, while the root contains all peers. Associated with each zone is a unique identifier, a pathname, and an attribute list (MIB) which contains the information associated with the zone. The Astrolabe attributes are not directly writable, but generated by aggregation functions. Each zone has a set of aggregation functions that calculate the attributes for the zone's MIB. An aggregation function for a zone is an SQL program that takes a list of the MIBs of the zone's child zones, and produces a summary of their attributes. Applications invoke Astrolabe interfaces through calls to a library. The library has an SQL interface that allows applications to view each node in the zone tree as a relational database table, with a row for each child zone and a column for each attribute. The programmer can then simply invoke SQL operators to retrieve data from the tables.

[9] presents a sampling-based technique for approximate answering of ad-hoc aggregation queries in unstructured P2P systems. The approach followed here is based on the skewed sampling of relational data and the assumption that certain aspects of the P2P network are known to all peers, such as the average degree of the nodes, a good estimate of the number of peers in the system, and certain topological characteristics of the network structure. The query processing strategy consists of two phases. In the first phase, a random walk is initiated from the query node, long enough to ensure that the visited peers represent a close sample from the underlying stationary distribution. The information retrieved from the visited peers, such as the number and aggregate of tuples (e.g., SUM/COUNT/AVG, etc.) that satisfy the selection condition is sent back to the query node. Then an analysis is followed to determine the skewed nature of the data that is distributed across the network, such as the variance of the aggregates of the data at nodes, the amount of correlation between tuples that exists within the same nodes and the variance in the degrees of individual nodes in the P2P network. Once this data has been analyzed, an estimation is made on how much more samples are required and in

what way should these samples be collected so that the original query can be optimally answered within the desired accuracy with high probability. The second phase is then straightforward, a random walk is re-initiated and tuples are collected according to the recommendations made by the first phase.

[39] presents a model for multidimensional data distributed in a P2P network and a query rewriting technique, that allows a local peer to propagate aggregation queries. The model supports P2P OLAP queries, a class of queries involving aggregate functions, defined over generalizations of fact tables and dimensions that are distributed in a P2P network. Each node involved in the system defines a context where it becomes the local peer and all its dimensions and fact tables are considered local henceforth. The rest of the nodes that connect with the local peer, are considered acquaintances of the local peer within this context. Because a multidimensional database normally consists in a collection of views of aggregated data, a careful translation process is needed in this case, in order to transform any summary concept that appears in a peer acquaintance into a summary concept meaningful to the requesting peer. For this reason, a LAV (Local As View) integration approach is used, which does not rely on mapping exclusively. Instead, a systematic revise and map strategy is used for defining how an instance of a dimension in an acquaintance is viewed from the local peer. Whenever a query is submitted to a node, that node fixes a context where all its fact tables and rollup functions become local. The posed query is then rewritten for propagation, introducing references to the appropriate mapping tables and revised rollup functions. Thus, the query result depends on the node where the query is posed and may change as changes in the revision and mapping process occur. Moreover, if the mapping is incomplete, the rewriting technique employs a bottom-up homomorphism preserving completion approach that allows the system to always produce a query result.

7 Processing of XML and RDF Queries

7.1 Processing RDF

We shall first present a short introduction to the RDF model and the query languages for RDF data and metadata. Then, we discuss issues related to storing and querying RDF data in peer-to-peer systems.

7.1.1 RDF Data Model and Query Languages

RDF (Resource Description Framework) is a framework for describing metadata. It enables interoperability between machines by interchanging information about information resources. Resources are identified by RDF with Uniform Resource Identifiers (URIs) as in [1]. The base element of RDF is the *triple*: a resource (subject) is linked to another resource (object) through an arc labeled with another resource (predicate). Alternatively, the subject is called a resource, the predicate is called property (of that resource) and the object is called value (of that property). The arc always goes from a resource to a value, thus creating a directed labeled graph (if more than one triples are linked together). The RDF Model Specification defines syntax for expressing RDF statements using an XML encoding i.e., an RDF document is created using syntax rules for XML documents. Any

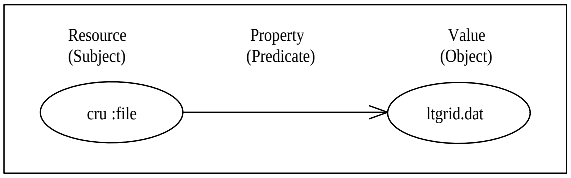
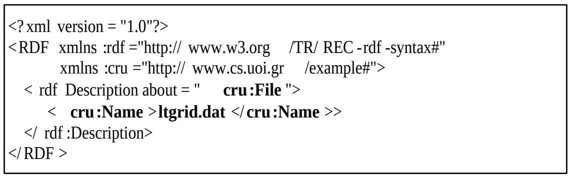


Figure 17: Graph represented RDF statement



```
<? xml  version = "1.0"?>
<RDF  xmlns :rdf ="http://  www.w3.org    /TR/ REC -rdf -syntax#"
         xmlns :cru ="http://  www.cs.uoi.gr    /example#">
  < rdf  Description about = "     cru :File ">
      <   cru :Name >ltgrid.dat </ cru :Name >>
  </  rdf :Description>
</ RDF >
```

Figure 18: XML-encoded RDF statement

XML parser can check well-formedness and validity of RDF files but due to its semantics orientated constructs and graph structure, specific RDF-aware XML parsers are required. For example, Figs. 17 and 18 show both the graph representation and the XML encoding of the sentence *"The name of the file is ltgrid.dat"*. The graph in Fig. 17 indicates that the *Resource* node has the value *file*, the *Property* node has the value *name* and the *Value* node has the value *ltgrid.dat*.

In addition to plain RDF data, RDF schema vocabularies are used to define the labels of nodes (classes) and edges (properties) that can be used to describe and query resources in specific user communities. These labels can be organized into appropriate taxonomies, carrying inclusion semantics. RDF schemata are used to declare classes and property-types, typically authored for a specific community or domain. The upper part of Fig. 19 illustrates such a schema for a cultural application. The scope of the declarations is determined by the namespace of the schema, e.g., *ns1* (*http://www.culture.gr/schema.rdf*).

Namespaces are used in order to avoid conflicts between similar names of classes or properties that have different semantic meaning (as different users may create two semantically different classes and label them with the same name). Classes and property types are uniquely identified by prefixing their names with their schema namespace, as for example, *ns1Artist* or *ns1creates*. To simplify our presentation, we henceforth omit the namespace prefixes. Moreover, classes can be organized into a taxonomy through simple or multiple specialization. The root of this hierarchy, is a built-in class called *Resource*. For instance, *Painter* and *Painting* are subclasses of *Artist* and *Artifact* respectively, both specializing *Resource*. RDF classes do not impose any structure to their objects and class hierarchies simply carry inclusion semantics. RDF property types serve to represent attributes of resources as well as relationships (or roles) between resources. For example, *creates* defines a relationship between the resource classes *Artist* (its domain) and *Artifact* (its range), while *fname* is an attribute of *Artist* with type *Literal*. As RDF literals we can have any primitive datatype defined in XML as well as XML markup which is not further interpreted by an RDF processor. As we can see in Fig. 19, property types may also be refined: *paints* is a specialization of *creates*, with its domain and range restricted to the classes *Painter* and *Painting*, respectively.

Several RDF query languages have been developed for querying RDF data and RDF schemata. Some of them are:

- *RDQL* [74]: RDQL is a query language for RDF proposed by the developers of the Jena Java RDF toolkit [73]. RDQL operates at the RDF triple level, without taking RDF Schema information into account (like RQL [60] does) and without providing inferencing capabilities.

- *RQL* [60]: RQL is a typed language, following a functional approach. It is defined by a set of basic queries and iterators. These basic queries are the building blocks of the query language. Then, such queries and iterators can be used to build more complex queries through functional composition, by preserving type integrity constraints which are specific for each operation, allowing arbitrary nesting in a query. RQL supports generalized path expressions featuring variables on labels for both nodes (i.e., classes) and edges (i.e., properties).

- *SeRQL* [18]: SeRQL ("Sesame RDF Query Language", pronounced "circle") is being developed by Aduna as part of Sesame [19].

- *RVL* [71]: RVL is a view definition language capable of creating not only virtual resource descriptions, but also virtual RDF/S schemata from (meta)classes, properties, as well as, resource descriptions available on the Semantic Web. RVL exploits the functional nature and type system of the RQL query language in order to navigate, filter and restructure complex RDF/S schema and resource description graphs.

7.1.2 RDF Data in P2P

As we discussed above, RDF data items are essentially (subject, predicate, object) triples. Thus, to store RDF data in a P2P system, we have to store these three elements. The problem is to find an efficient way for storing and retrieving this kind of data. One way is to store all three elements in one place. However, in this way we may not take advantage of the full capabilities of the system. Another way is to store each element at a different place and create links among them.

It is even a greater challenge to store RDF schemata in a P2P system. We have to take care of the subsumptions of schemata (both horizontal and vertical) as well as to preserve the semantic meanings between each class and each property. These two necessities make RDFS storage and retrieval a very challenging and difficult goal.

The system in [97] has been developed on top of an unstructured network. It is used for querying RDF data using SeRQL [20] as the query language. Queries pass, via a parser, from the query generator to an RDF API (one for each different RDF warehouse). Intermediate SAIL defines the parts of the query that will be forwarded to the relevant information sources. Queries are forwarded only to resources that are expected to provide answers. To index complex queries, join indexes are used so as to forward them to the destination resource. Additional database tables are created that contain results of a join over a specific property. At runtime, the system accesses the database, rather than computing a join, which is less expensive. The resulting indexing structure is a join index hierarchy. The most general element in the hierarchy is an index table for elements

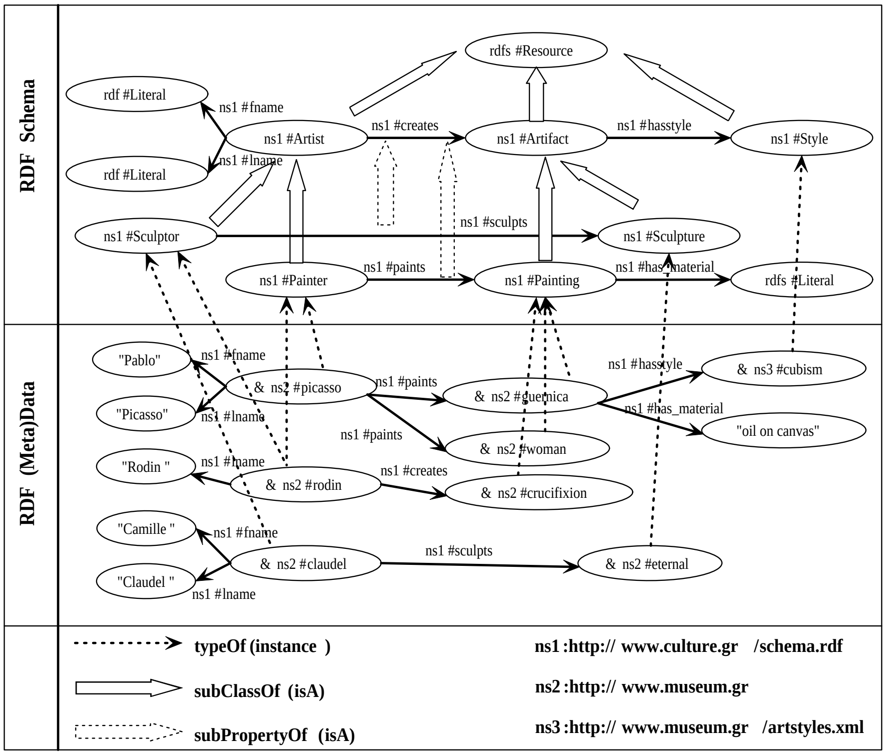


Figure 19: Graph represented RDF statement

connected by a certain path of length n. Every following level contains all the paths of a particular length, from 2 paths of length n - 1 at the second level of the hierarchy, to n paths of length 1 to the bottom level of the hierarchy. Join index hierarchies are also used for defining resources that contain results to a sub-query. The resulting expression is decomposed to a set of expressions, which describe simpler paths. Then, these paths are forwarded to resources that contain relevant information, by using a hierarchy of join indexes. To get the final results, the answers acquired from the individual resources ate joined. To answer a query, all possible sub-paths of the query path need to be found. For each one of these sub-paths, the resources that contain answers are located, the results from each resource are retrieved and joined to a single result, to answer the whole path.

The system in [22] has been developed for querying RDF and range data. It uses a DHT structure: MAAN [21], an extension of chord. The query language used is RDQL. Each RDF triple is stored at three places based on the MAAN storage mechanism. First, a triple is stored based on its subject, second on its predicate and third on its object. Every triplet has an expiration time. The peer that inserts a triplet must update it before the expiration time of the triplet runs out. If the peer that has stored the triplet does not accept any updates concerning the specific triplet, it will delete it. Replication is supported by maintaining copies of triplets at neighbor peers. Atomic queries can be answered based on the exact value of an element of the triple, that is, at least an element of the triple must have an exact value. Disjunctive range queries can be answered by using a list of constraints that limit the value range of the variables based on the support provided by

MAAN for processing range queries. Conjunctive multi-predicate queries can be solved by using a recursive algorithm that searches for candidate subjects for each predicate and calculates the intersection of these subjects before returning the result of the query.

The system in [77] uses a hypercube structure to store and query RDF/S data and is compatible with any query that is in line with RDF/XML conventions. Super-peers are placed in a hypercube topology and they are responsible for the efficient search of RDF metadata and the maintenance of the hypercube structure. Simple peers are are responsible for storing the RDF metadata. They are connected to each super-peer forming a star topology. Super-peers maintained a data structure, called SP/P routing table, with information about the schema of the peer, the properties of the schema and the value of the properties. The SP/P table is used for the insertion of a peer and its connection with a super-peer, as well as for the efficient routing of the queries, only to nodes which are relevant with these queries. Each super-peer also keeps an SP/SP table which is the sum of its SP/P indexes for routing queries among super-peers. When a peer leaves the network, all the respective registrations of the SP/P table of the leaving node of the super-peer that is responsible for the specific peer are removed. When a peer registers the network, the metadata of the peer are compared against the current registrations of the SP/P table of the super-peer with which the peer connects to. If the metadata already exist in the super-peer, then only the id of the peer is inserted into the SP/P table, else all the metadata that are not present at the super-peer are inserted. An update in the SP/P table causes the update of the SP/SP tables of the hypercube, in a way, similar to the way with which an efficient broadcast can take place in a hypercube. When a new super-peer registers the network, the super-peer, that is responsible for the position in the hypercube, in which the new super-peer is inserted, transfers one half of its SP/SP table to the new super-peer, while the neighbors of these super-peers are updated appropriately. When a super-peer leaves the network, it transfers the SP/SP and SP/P tables to the super-peer that is responsible for the vacant position in the hypercube, that is about to be created and all the neighbors of the two node are appropriately updated. For routing a query, the elements of the query are compared against the SP/SP and SP/P tables and the query is forwarded to the peers that can answer these elements.

The system in [26] uses, also, a hypercube structure for storing and querying RDF/S data. It uses the QEL language. It extends [77] to support publish/subscribe systems. Peers can subscribe queries, advertise their content and notify the network when new resources become available. The routing of advertisements is based on the SP/SP and the SP/P tables. To route subscriptions, super-peers store a hierarchy of subscriptions. Subscriptions that are contained in parent peers are placed at the children peers of the hierarchy. The rest of the routing procedure is based on the SP/SP and SP/P tables as the routing of advertisements. As for the routing of notifications, when a notification reaches a super-peer, it is compared with the root subscription of the local hierarchy subscription of the specific super-peer. If there is a match, then the notification is compared against the children of the root subscription and so on. For every such comparison where the notification matches the compared subscription, the notification is sent to the super-peer, from which the compared subscription came from, thus following the path of the subscription in the opposite direction. Update of advertisements can be done in a way similar to the way for updating the SP/SP and SP/P tables described above.

Piazza [47] is a system for querying semantic (RDF/S, OWL, XML) and non-semantic relational data. It can be implemented on top of any type of overlay network. The query language used in Piazza is XQuery. When a new peer is added to the system, it is semantically related to some portion of the existing network. Queries are always posed from the perspective of a given peer's schema, which defines the preferred terminology of the user. When a query is posed, Piazza provides answers that utilize all semantically related XML data within the system. To exploit data from other peers, there must be semantic mappings between the peers. Mappings are specified between small numbers of peers, usually pairs. There are three main cases, depending on whether the mapping are between pairs of OWL/RDF nodes, between pairs of XML/XML Schema nodes, or between nodes of different types. Mappings play two roles; the first role is as storage descriptions that specify which data is actually stored at a peer and the second role is as schema mappings, which describe how the terminology and structure of one peer correspond to those in a second peer. This allows to separate between the intended domain and the actual data stored at the peer. A query at a particular peer must be expanded and translated into appropriate queries over semantically related Query answering may require that we follow semantic mappings in both directions. In one direction, composing semantic mappings is simply query composition for an XQuery-like language. In the other direction, composing mappings requires using mappings in the reverse direction, which is known as the problem of answering queries using views.

The project in [61] considers the efficient query processing of RDF/S data either on top of a DHT (e.g. chord) structure or on top of a hybrid network (unstructured network with super-peers). The query language for RDF/S data is RQL and RVL. Two ways of structuring peers are described. The first one uses a hybrid P2P semantic network, where peers are organized in a hierarchy, under specific super-peers. Children peers contain RDF schemata that are subsumed in the RDF schemata of their parent peers. Peers that are described by the same schema are placed under the same super-peer, so that each super-peer knows which schemata are in its hierarchy. When a peer issues a query that can not be answered by the same peer, the query is forwarded to the super-peer that the specific peer belongs to. If this super-peer does not contain the schema of the query, then the super-peer forwards the query to one of the other super-peers, randomly. This procedure continues until the super-peer that has the specific schema is found. Then, the query is forwarded to the hierarchy of thus super-peer. The second way of structuring peers is a semantic network based on DHT P2P system, like Chord. In this case, keys of the hash functions are assigned to whole sub-schemata. In order to preserve the semantic network, there is a need to derive appropriate hash functions for the assignment of the keys, so that peers that have similar schemata can be found in successive steps. In this case, to answer a sub-query, all peers (of the entire network) must be asked in successive steps. The results of these queries must be joined, so that we can get a complete answer of the query.

The project in [33] studies the problem of integrating RDF and XML data on an unstructured super-peer network. The query languages used, for this purpose are XQuery and RDQL. Super-peers contain global RDF ontologies, while simple peers contain local schemata and local data sources. Peers are connected to super-peers through P2P mappings. The architecture has four main components: (a) an XML to RDF wrapper used to transform the XML schema into a local RDF schema, (b) local XML and RDF schemata (residing at peers), (c) global RDF ontologies residing at super-peers and (d) mapping tables used to store mappings between local schemata and the global ontology. Whenever a new peer joins the network, the peer gets registered and indexed in the super peer by establishing mappings from its local schema to the global ontology. The mappings are established through a process of schema matching and stored in the mapping table of the peer. During the process of schema matching, the global ontology is extended by integrating the local schemata. The domain structure and document structure of local schemata are encoded in the mappings. The current architecture provides two query processing modes. In the data-integration mode, the user poses a query (source query) on the global ontology in the super peer, which is then reformulated into multiple subqueries (target queries) over the XML and RDF sources in the peers (one subquery for each source). By executing the target queries and integrating their results, the system returns an answer to the user at the site of the super-peer. In the hybrid P2P mode, the user can pose a source query on the local XML or RDF source in some peer. Locally, the query will be executed on the local source to get a local answer. Meanwhile, the source query is reformulated into a target query over every other peer through transitive mappings, that is, compositions of mappings from the original peer to the super-peer and mappings from the super-peer to the other target peers. By executing the target query, each peer returns an answer to the original peer, called the remote answer. The local and remote answers are integrated and returned to the user at the site of the originating peer. Query translation is achieved by using the mappings in conjunction with a collection of query rewriting algorithms.

GridVine [5] uses a two-layer approach to storing and retrieving RDF data and RDF Schemata: the *logical layer* that supports operations for maintaining the semantic overlay network (e.g. schema management, attribute-based search and schema mapping) and the underlying *physical layer* which is a structured network, namely P-Grid. In GridVine, statements are stored as RDF triples and refer to data items shared in the P-Grid infrastructure. Simple RDF data are stored three times by hashing subject, predicate and object separately. For storing RDF Schemata, GridVine defines two new meta-schema classes: $P - GridDataItem$ and $P - GridDataItemProperty$. All classes of the schemata have to be sub-classes of the meta-class $P - GridDataItem$ and all properties have to be sub-properties of the meta-class $P - GridDataItemProperty$. This way, the user can introduce new schemata and reconciliate them with the existing schemata. To answer a query, where not all of the three attributes of a triple are known, P-Grid hashes the known attribute in order to find the peer that is responsible for this attribute. Then the peer search its local database and reports with all the findings. Note that GridVine does not allow queries in which none of the attributes of a triple is known. GridVine uses semantic gossiping as a reconciliation method among two different schemata. All peers that describe their data according to the same schema are said to be in the same semantic neighborhood. Each peer has the possibility to create a mapping (link) between two schemata, thus creating a network that can be seen as a direct graph.

Remindin' [99] is an algorithm for forwarding and answering RDF queries, built on top of the SWAP platform. Remindin' uses SeRQL queries. The algorithm works as follows. Assume a user of a peer issues a query. First, the query is evaluated locally, against the local repository of the peer. The query is also forwarded to other peers to be evaluated across the network. For forwarding a query, Remindin' selects a set of peers that appear more promising than the others to answer the given query. There are some cases where no such peer can be selected. In such cases, the query is broadened so that all peers can be

Table 4: Taxonomy of RDF systems

	Overlay Type	Support Schema	Load Balance	Query Planning	Range Queries	Routing through	Replication
[97]	Unstructured	√		√		Index	
RDFPeers [22]	Structured		√		√		√
[77]	Structured	√	√			Index	
[26]	Structured	√	√			Index	
Piazza [47]	Unstructured Structured	√		√		Semantic Mapping	
SQPeers [61]	Unstructured Structured	√		√			
PERSINT [33]	Unstructured	√		√		Semantic Mapping	
GridVine [5]	Structured	√		√	√	Semantic Mapping and Index	√
Remindin' [99]	Unstructured	√		√		Index	

selected. Next, the original query will be forwarded to a subset of the selected peers. To avoid cycles, the message containing the query has a unique id and stores the id's of visited peers. When a peer receives a query it will try to answer it and it will store an index pointing at the originator of the query. The answer is returned directly to the querying peer (only if it is not empty). If the number of maximum hops is not reached yet, the query will be forwarded to a selected set of peers (using the same peer selection described before). On the arrival of answers at the querying peer, relevant answers are selected and included into the repository.

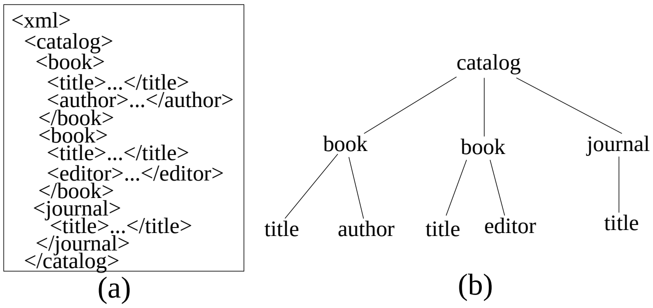
A summary of the approaches to processing RDF data in p2p systems described above is provided in Table 4.

7.2 Processing XML

We shall first present a short introduction to XML documents and languages for querying then. Then, we survey the various approaches that have been proposed for storing and querying XML documents in peer-to-peer systems.

7.2.1 XML Data Model and Query Languages

XML [2] has evolved as the new standard for data representation and exchange in the Internet. XML is a self-descriptive language that was designed so as to focus on the description and the structure of the data. Its flexibility in describing heterogeneous data makes it very popular for distributed applications and systems where the data are either native XML documents or XML descriptions of data or services that are represented in different formats in the underlying sources (i.e. in relational databases). Furthermore, XML supports more expressive query languages that address both the structure and the content of the data and therefore, they convey more semantics to the queries.



```
<xml>
  <catalog>
    <book>
      <title>...</title>
      <author>...</author>
    </book>
    <book>
      <title>...</title>
      <editor>...</editor>
    </book>
    <journal>
      <title>...</title>
    </journal>
  </catalog>
```

Figure 20: Example of (a) an XML document and (b) its corresponding XML tree

XML employs a tree-structured model for representing data. The main building blocks of an XML document are the elements that may contain other elements, attributes and content. Elements are hierarchically structured and provide information about the data they describe. Attributes are used to provide additional information about the elements. Typically, an XML document is modeled as a node-labeled tree $T(V, E)$ (Fig. 20). Each node $e_i \in V$ corresponds to an XML element with a label assigned from some some string literals alphabet that captures element's semantics. Edges $(e_i, e_j) \in E$ are used to capture the containment of element e_j under e_i. Leaf elements in T typically contain values.

Queries in XML query languages typically specify patterns of selection predicates on multiple elements that have some specified tree structured relationships. XPath is a language for specifying these tree-structured relationships within an XML document. It uses path expressions for addressing different parts of the XML tree. The simplest and most commonly used path expressions involve the parent-child and descendant-or-self axes (i.e."/" and "//" operators) and the wildcard operator ("*"). These expressions are represented as a sequence of element tags and describe a navigation through the XML tree. An XPath expression $//t_1/t_2/.../t_n$ is evaluated sequentially by finding an element t_1 anywhere in the document, and nested in it an element t_2, and so on until we find t_n. The result of the XPath expression on a given XML document is the set of t_n nodes found in the tree.

7.2.2 Querying XML Documents in P2P Systems

There many approaches to processing XML documents in a p2p setting including building structured, unstructured and clustered overlays [65]. In the case of structured overlays, an important issue is providing an appropriate mapping of XML documents to nodes. In unstructured overlays, the focus is on building efficient data structures for routing indexes for XML documents. Clustered overlays create groups of nodes that hold similar documents.

Structured Overlays. XP2P [17] maps XML fragments along with ancestor and children fragments onto the Chord ring by using the fingerprinting hashing technique. Queries are fingerprinted and looked up by the Chord functionality; if all the child fragments are unfolded full match is attained. Otherwise, gradual pruning is used for a partial match. For //, a bottom-up technique is followed and hints through children fragments are ex-

Table 5: Querying XML data in P2P systems

Name	**Data Type**	**Query Type**	**Overlay Type**
XP2P [17]	XML	Partial & full XPath [/,//] and positional filters	Structured (Chord)
[42]	XML	XPath [/,//,*] and value predicates	Structured (Chord)
[94]	XML	XPath [/,//,*]	Structured (P-Grid)
Multi-level Bloom filters [64]	XML	Path queries [/,//,?]	Unstructured (interconnected hierarchies)
Mutant Query Plans [79]	XML	SQL (σ, π, join)	Structured (CAN)
XPeer [91]	XML	XQuery FLWR	Unstructured (hierarchies of superpeers)
[66]	XML	XPath [/,//]	Unstructured (routing indexes)
[103]	XML	XPath [/,//]	Structured (CAN)
INS/Twine [10]	Hierarchical/ orthogonal attribute-value pairs	Partial/ complete tree pattern queries	Structured (Chord or any DHT)
KADOP [6]	AXML	Tree-pattern queries	Structured (Pastry)
BRICKS [87]	XML	XPath, XQuery	Structured (DHT-based)
MediaPeer [38, 37]	XML	XQuery	Unstructured hybrid
[52, 59, 58]	XML	XPath	Unstructured (routing indexes)
XMIDDLE [72]	XML & DOM	XPath	Ad-hoc networks of mobile devices

ploited to avoid exhaustive search.

In [42], elements and attributes are the keys hashed into Chord along with structural (all possible paths leading from the root to key) and value summaries. All simple paths are extracted from the query and looked up in Chord starting from their last tag to the first, retrieving at each step the set of candidate peers from the catalog. Their intersection yields the final result set.

In [94], for each path expression inserted, all of its possible suffixes are hashed into P-Grid, using an order-preserving hash function. Each data item stores the key, the original fragment and its URI to the source XML document/fragment. For query processing the longest simple path is hashed and looked up in P-Grid. Broadcasting in sub-tries is deployed to retrieve all the results. The final results are checked against the original query before being returned.

In [103], the system presumes that all peers know the schema of the data. Each dimension in the virtual multi-dimensional space corresponds to either a path level or a unique attribute on a specific path level. The virtual space is viewed as a hyper-rectangle and each piece of XML is mapped to a logical node according to its coordinates that are derived by hashing each element name and attribute that corresponds to each of the dimensions. When the queries consist of absolute location paths with only parent-child axis, is done by CAN, which is enhanced in order to find the closest hyper-rectangle. If the query is more complex, it is transformed into one or more absolute location paths and the same mechanism is deployed.

INS/TWINE [10] transforms data into a canonical form of attribute-value trees. Each data item is split into all possible prefixes that are hashed into Chord along with the whole data item. A data item matches a query if the AVTree formed by the query is the same as the AVTree of the original description, with zero or more truncated attribute-value pairs. A query is resolved by searching for its longest prefix.

In KADOP [6], each resource published in the system is stored locally in the ActiveXML repository within the ActiveXML layer of a peer. The peer offers web services to access and retrieve the resource when its identifier is known. KADOP indexes its resources with a DHT using tag names, precise words, concept names etc, as keys, while resource identifiers make up the associated values.

BRICKS [87] uses a datastore component that follows the DOM model, splitting the XML documents into sets of nodes. A query engine and an optional index manager are located on top of the datastore. The sets of nodes are associated with a hash key and inserted into the DHT. A distributed UDDI is used for resource discovery. A peer sends the relevant parts of the queries to peers that might have relevant results. If some nodes are matched, then the references to the nodes will be sent to the originator of the query.

Unstructured Overlays In [52, 59, 58], the system uses routing indexes that are incrementally updated based on query feedback so as to adapt to changing workloads and topologies. Statistical information maintained at the routing indexes is used to optimize query execution based on a cost model. Path trees are used as routing indexes, which are pruned to accommodate lower memory constraints. Two dimensional histograms are used for the efficient evaluation of value predicates, where the first dimension is the classical division into buckets and the second is a time axis of expected returned results within a number of hops.

XMIDDLE [72] defines a set of possible access points for the private tree on each device, so that other devices can link to these points to gain access to this information; essentially, the access points address branches of trees that can be modified and read by peers. In order to share data, a host needs to explicitly link to another host's tree, by using XLink. The applications on the devices are enabled to manipulate the XML information through the DOM API, which provides primitives for traversing, adding and deleting nodes to an XML tree. XMIDDLE provides an approach to sharing that allows on-line collaboration, off-line data manipulation, synchronization and application dependent reconciliation.

In [66], path indexes are used as the internal organization of the routing indexes, which maintain pointers from each path to the corresponding peers that contain it. Since the information included in a path index can grow excessively, aggregating paths with common prefixes is proposed to accommodate the routing index in a given space overhead. The indexes are distributed either by slowing each peer to potentially know and communicate with every other peer or by letting peers enter into bilateral agreements with some other peers called its neighbors.

Clustered Content-based Overlays In [64], each peer in the hierarchies, stores a local index summarizing its own content and one routing indexes that summarize the contents of the peers forming its subtree. The roots additionally store the other roots routing indexes. Multi-level Bloom filters that preserve the hierarchical relationships of XML data by inserting the elements of the XML tree to a different level of the filter, are used as indexes. Queries are routed through the hierarchies by consulting the routing indexes of each peer.

In [79], the data of the peers are assumed to belong to some categorization hierarchies relevant to a domain, called a multihierarchic namespace. Groups of peers choose what data to host based on their own interests, thus defining their interest areas. The interest areas describe index coverage of other groups' data and are encoded into URNs. The associated index servers to each interest area are contacted to find relevant data. A mutant query plan is an algebraic query plan graph, encoded in XML that may also include verbatim XML-encoded data, references to resource locations (URLs), and references to abstract resource names (URNs). An MQP starts as a regular query operator tree at the client peer and is then passed around among peers accumulating partial results, until it is fully evaluated into a constant piece of XML data.

XPeer [91] logically organizes peers into clusters that are formed on a schema-similarity basis, whenever this is possible. Superpeers are organized into trees, where each peer hosts schema information about its children. Peers export a tree-shaped DataGuide description of their data, which is automatically inferred by a tree search algorithm. Queries are translated it into location-free algebraic expressions and sent to the superpeer network for the compilation of a location assignment. After the location assignment, the query is passed back to the peer that issued it for execution: the query is split into single-location subqueries that are sent to the corresponding peers.

In MediaPeer [38, 37], peers are clustered to superpeers according to their metadata information and superpeers may be in turn clustered forming overlay trees. Every peer provides an XML Schema abstraction (set of tree structures) of the local data. Every XQuery is translated to a collection of path-sets routed by each SP to peers that contain relevant data. The forwarding decisions are taken based on the index structures maintained at each SP in the overlay network. Each SP index is organized as an adaptable trie with a more reduced size for the SPs at upper levels.

A summary of the approaches to processing XML documents in p2p systems described above is provided in Table 5.

References

[1] http://www.w3.org/RDF.

[2] World wide web consortium. extensible markup language (xml) 1.0 (second edition). *http://www.w3.org/TR/REC-xml.*

[3] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. *Sixth International Conference on Cooperative Information Systems (CoopIS 2001), Lecture Notes in Computer Science*, 2172:179–194, 2001.

[4] K. Aberer. Scalable Data Access in P2P Systems Using Unbalanced Search Trees. *Workshop on Distributed Data and Structures (WDAS-2002)*, 2002.

[5] K. Aberer, P. Cudré-Mauroux, M. Hauswirth, and T. Van Pelt. Gridvine: Building internet-scale semantic overlay networks. In *International Semantic Web Conference (ISWC)*, pages 107–121, 2004.

[6] S. Abiteboul, I. Manolescu, and N. Preda. Constructing and Querying Peer-to-Peer Warehouses of XML Resources. *21st International Conference on Data Engineering (ICDE 2005)*, 2005.

[7] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman. Search in power-law networks. *Physical Review*, E 64, 2001.

[8] AEOLUS Project. Deliverable D3.1.1. Distributed Data Management: State-of-the-art Survey, 2006.

[9] B. Arai, G. Das, D. Gunopulos, and V. Kalogeraki. Approximating Aggregation Queries in Peer-to-Peer Networks. *ICDE 2006*, 2006.

[10] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. *Pervasive 2002*, 2002.

[11] W.T. Balke, W. Nejdl, W. Siberski, and U. Thaden. Progressive Distributed Top-k Retrieval in Peer-to-Peer Networks. *21st International Conference on Data Engineering*, pages 174–185, 2005.

[12] M. Bawa, G. S. Manku, and P. Raghavan. Sets: search enhanced by topic segmentation. In *SIGIR*, pages 306–313, 2003.

[13] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute RangeQueries. *SIGCOMM '04*, 2004.

[14] B. Bhattacharjee, S. S. Chawathe, V. Gopalakrishnan, P. J. Keleher, and B. D. Silaghi. Efficient Peer-To-Peer Searches Using Result-Caching. In *IPTPS*, pages 225–236, 2003.

[15] I. Bhattacharya, S. R. Kashyap, and S. Parthasarathy. Similarity Searching in Peer-to-Peer Databases. *25th IEEE International Conference on Distributed Computing Systems*, pages 329–338, 2005.

[16] P. Boncz and C. Treijtel. AmbientDB : Relational Query Processing in a P2P Network. *International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P)*, 2003.

[17] A. Bonifati, U. Matrangolo, A. Cuzzocrea, and M. Jain. XPath Lookup Queries in P2P networks. *ACM WIDM '04*, 2004.

[18] J. Broekstra and A. Kampman. SeRQL: An RDF Query and Transformation Language. *International Semantic Web Conference (ISWC 2004)*, 2004.

[19] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. *International Semantic Web Conference (ISWC 2002)*, 2002.

[20] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. *The Semantic Web - ISWC 2002*, 2342:54–68, 2002.

[21] M. Cai, M. Frank, J. Chen, and P. Szekely. MAAN: a multi-attribute addressable network for grid information services. *Proceedings of the 4th International Workshop on Grid Computing*, 2003.

[22] M. Cai, M. Frank, B. Yan, and R. MacGregor. A subscribable peer-to-peer RDF repository for distributed metadata management. *Journal of Web Semantics*, 2(2):109–130, 2004.

[23] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, J. Hellerstein, and S. Shenker. A Case Study in Building Layered DHT Applications. In *SIGCOMM*, 2005.

[24] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *SIGCOMM*, pages 407–418, 2003.

[25] Y. Chen, R. H. Katz, and J. Kubiatowicz. Dynamic Replica Placement for Scalable Content Delivery. In *IPTPS*, pages 306–318, 2002.

[26] P. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl. Publish/Subscribe for RDF-based P2P Networks. *1st European Semantic Web Symposium*, 2004.

[27] E. Cohen, A. Fiat, and H. Kaplan. Associative search in peer to peer networks: Harnessing latent semantics. In *INFOCOM*, 2003.

[28] E. Cohen and S. Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. In *SIGCOMM*, pages 177–190, 2002.

[29] B. F. Cooper. An optimal overlay topology for routing peer-to-peer searches. In *Middleware*, pages 82–101, 2005.

[30] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-Ring: An Index Structure for Peer-to-Peer Systems. *Cornell University Technical Report*, 2004.

[31] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS*, 2002.

[32] A. Crespo and H. Garcia-Molina. Semantic overlay networks for p2p systems. In *Stanford Univ. TR 2003-75*, 2003.

[33] I. F. Cruz, H. Xiao, and F. Hsu. Peer-to-Peer Semantic Integration of XML and RDF Data Sources. *Third International Workshop on Agents and Peer-to-Peer Computing (AP2PC 2004)*, 2004.

[34] F. Dabek, M. F. Kaashoek, D. R. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *SOSP*, pages 202–215, 2001.

[35] A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *ICDCS*, 2003.

[36] V. V. Dimakopoulos and E. Pitoura. Performance analysis of distributed search in open agent systems. In *IPDPS*, page 20, 2003.

[37] F. Dragan, G. Gardarin, and L. Yeh. MediaPeer: a Safe, Scalable P2P Architecture for XML Query Processing. *16th International Workshop on Database and Expert Systems Applications (DEXA 2005)*, 2005.

[38] F. Dragan, G. Gardarin, and L. Yeh. Routing XQuery in A P2P Network Using Adaptable Trie-Indexes. *IADIS International Conference WWW/Internet*, 2005.

[39] M. M. Espil and A. A. Vaisman. Aggregate Queries in Peer to Peer OLAP. *DOLAP'04*, 2004.

[40] F. Le Fessant, S. Handurukande, A.-M. Kermarrec, and Laurent Massoulié. Clustering in peer-to-peer file sharing workloads. In *3rd International Workshop on Peer-to-peer systems (IPTPS)*, 2004.

[41] M. J. Freedman and D. Mazières. Sloppy hashing and self-organizing clusters. In *IPTPS*, pages 45–55, 2003.

[42] L. Galanis, Y. Wang, S.R. Jeffery, and D.J. DeWitt. Locating Data Sources in Large Distributed Systems. *29th International Conference on Very Large Data Bases (VLDB)*, 2003.

[43] P. Ganesan, P. Krishna Gummadi, and H. Garcia-Molina. Canon in g major: Designing dhts with hierarchical structure. In *ICDCS*, 2004.

[44] P. Ganesan, B. Yang, and H. Garcia-Molina. One Torus to Rule Them All: Multi-dimensional Queries in P2P Systems. In *WebDB*, pages 19–24, 2004.

[45] V. Gopalakrishnan, B. D. Silaghi, B. Bhattacharjee, and P. J. Keleher. Adaptive Replication in Peer-to-Peer Systems. In *ICDCS*, pages 360–369, 2004.

[46] A. Gupta, D. Agrawal, and A. E. Abbadi. Approximate Range Selection Queries in Peer-to-Peer Systems. *CIDR 2003*, 2003.

[47] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: Data Management Infrastructure for Semantic Web Applications. *WWW 2003*, 2003.

[48] S. Handurukande, A.-M. Kermarrec, F. Le Fessant, and Laurent Massoulié. Exploiting semantic clustering in the edonkey p2p network. In *SIGOPS European Workshop*, 2004.

[49] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. *First International Workshop on Peer-to-Peer Systems*, 2002.

[50] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. *Technical Report MSR-TR-2002-92, Microsoft Research*, 2002.

[51] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. *Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

[52] K. Hose, M. Karnstedt, K-U. Sattler, and E-A. Stehr. Adaptive Routing Filters for Robust Query Processing in Schema-Based P2P Systems. *9th International Database Engineering and Application Symposium (IDEAS 05)*, 2005.

[53] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. *Proceedings of the 29th VLDB Conference*, 2003.

[54] H. V. Jagadish, B. C. Ooi, Q. H. Vu, K. L. Tan, Q. H. Vu, and R. Zhang. Speeding up Search in Peer-to-Peer Networks with A Multi-way Tree Structure. In *SIGMOD*, 2006.

[55] H. V. Jagadish, B. C. Ooi, Q. H. Vu, R. Zhang, and A. Zhou. VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes. In *ICDE*, 2006.

[56] H.V. Jagadish, B. C. Ooi, and Q.H. Vu. BATON: a balanced tree structure for peer-to-peer networks. *VLDB 2005*, pages 661–672, 2005.

[57] V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *CIKM*, pages 300–307, 2002.

[58] M. Karnstedt, K. Hose, and K-U. Sattler. Distributed Query Processing in P2P Systems with Incomplete Schema Information. *3rd International Workshop on Data Integration over the Web (DIWeb2004)*, 2004.

[59] M. Karnstedt, K. Hose, and K-U. Sattler. Routing and Processing in Schema-Based P2P Systems. *DEXA Workshops*, 2004.

[60] G. Karvounarakis, V. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. *11th International World Wide Web Conference (WWW)*, 2002.

[61] G. Kokkinidis, L. Sidirourgos, and V. Christophides. Query Processing in RDF/S-based P2P Database Systems. *Semantic Web and Peer-to-Peer Springer-Verlag*, 2006.

[62] G. Koloniari, Y. Petrakis, and E. Pitoura. Content-based overlay networks for xml peers based on multi-level bloom filters. In *DBISP2P*, pages 232–247, 2003.

[63] G. Koloniari, Y. Petrakis, E. Pitoura, and T. Tsotsos. Query workload-aware overlay construction using histograms. In *CIKM*, pages 640–647, 2005.

[64] G. Koloniari and E. Pitoura. Content-based Routing of Path Queries in Peer-to-Peer Systems. *International Conference on Extending Database Technology (EDBT)*, 2004.

[65] G. Koloniari and E. Pitoura. Peer-to-peer management of xml data: issues and research challenges. *SIGMOD Record*, 34(2):6–17, 2005.

[66] N. Koudas, M. Rabinovich, D. Srivastava, and T. Yu. Routing XML Queries. *20th International Conference on Data Engineering (ICDE'04) (poster)*, 2004.

[67] E. Leontiadis, V. V. Dimakopoulos, and E. Pitoura. Cache updates in a peer-to-peer network of mobile agents. In *Peer-to-Peer Computing*, pages 10–17, 2004.

[68] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P File-Sharing with an Internet-Scale Query Processor. *VLDB*, 2004.

[69] B. T. Loo, R. Huebsch, I. Stoica, and J. M. Hellerstein. The case for a hybrid p2p search infrastructure. In *IPTPS*, pages 141–150, 2004.

[70] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *ICS*, pages 84–95, 2002.

[71] A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the Semantic Web Through RVL Lenses. *2nd International Semantic Web Conference (ISWC)*, 2003.

[72] C. Mascolo, L. Capra, and W. Emmerich. An XML-based Middleware for Peer-to-Peer Computing. *International Conference on Peer-to-Peer Computing*, 2001.

[73] B. McBride. Jena: Implementing the RDF Model and Syntax specification. *2nd Int'l Semantic Web Workshop*, 2001.

[74] L. Miller, A. Seaborne, and A. Reggiori. Three implementations of SquishQL, a simple RDF query language. *In First Int'l Semantic Web Conference*, 2002.

[75] R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *ACM SIGCOMM 2001*, pages 149–160, 2001.

[76] W. Nejdl, W. Siberski, U. Thaden, and W.T. Balke. Top-k Query Evaluation for Schema-Based Peer-to-Peer Networks. *International Semantic Web Conference*, pages 137–151, 2004.

[77] W. Nejdl, M. Wolpers, W.Siberski, C. Schmitz, M. Schlosser, and I. Brunkhorst. Super-Peer-Based Routing Strategies for RDF-Based Peer-to-Peer Networks. *Journal of Web Semantics*, 1(2):177–186, 2004.

[78] B. C. Ooi, Y. Shu, and K.-L. Tan. Relational Data Sharing in Peer-based Data Management Systems. *ACM SIGMOD*, 23 3, 2003.

[79] V. Papadimos, D. Maier, and K. Tufte. Distributed Query Processing and Catalogs for Peer-to-Peer Systems. *ICIDR Conference*, 2003.

[80] Y. Petrakis, G. Koloniari, and E. Pitoura. On using histograms as routing indexes in peer-to-peer systems. In *DBISP2P*, pages 16–30, 2004.

[81] Y. Petrakis and E. Pitoura. On constructing small worlds in unstructured peer-to-peer systems. In *EDBT Workshops*, pages 415–424, 2004.

[82] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, Load Balancing and Efficient Range Query Processing in DHTs. In *EDBT*, pages 131–148, 2006.

[83] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Workshop on Algorithms and Data Structures*, 1989.

[84] V. Ramasubramanian and E. Gün Sirer. Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays. In *NSDI*, pages 99–112, 2004.

[85] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. *ACM SIGCOMM'01*, pages 161–172, 2001.

[86] S. C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *INFOCOM*, 2002.

[87] T. Risse and P. Knezevic. A Self-organizing Data Store for Large Scale Distributed Infrastructures. *International Workshop on Self-Managing Database Systems (SMDB '05)*, 2005.

[88] R. Rodrigues and B. Liskov. High Availability in DHTs: Erasure Coding vs. Replication. In *IPTPS*, pages 226–239, 2005.

[89] M. Roussopoulos and M. Baker. CUP: Controlled Update Propagation in Peer-to-Peer Networks. In *USENIX Annual Technical Conference, General Track*, pages 167–180, 2003.

[90] O. D. Sahin, A. Gupta, D. Agrawal, and A. El Abbadi. A Peer-to-peer Framework for Caching Range Queries. *20th International Conference on Data Engineering (ICDE'04)*, 2004.

[91] C. Sartiani, P. Manghi, G. Ghelli, and G. Conforti. XPeer: A Self-organizing XML P2P Database System. *First International Workshop on P2P and DB*, 2004.

[92] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. HyperCuP-Hypercubes, Ontologies and Efficient Search on P2P Networks. *International Workshop on Agents and Peer-to-Peer Computing*, 2002.

[93] E. Sit, A. Haeberlen, F. Dabek, B. Chun, H. Weatherspoon, R. Morris, M. Frans Kaashoek, and John Kubiatowicz. Proactive Replication for Data Durability. In *IPTPS*, 2006.

[94] G. Skobeltsyn, M. Hayswirth, and K. Aberer. Efficient Processing of XPath Queries with Structured Overlay Network. *4th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE)*, 2005.

[95] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining temporal coherency of virtual data warehouses. In *IEEE Real-Time Systems Symposium*, 1998.

[96] K. Sripanidkulchai, B. M. Maggs, and H. Zhang. Efficient content location using interest-based locality in peer-to-peer systems. In *INFOCOM*, 2003.

[97] Heiner Stuckenschmidt, Richard Vdovjak, Geert-Jan Houben, and Jeen Broekstra. Index Structures and Algorithms for Querying Distributed RDF Repositories. *WWW 2004*, 2004.

[98] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *SIGCOMM*, pages 175–186, 2003.

[99] C. Tempich, S. Staab, and A. Wranik. Remindin': Semantic query routing in peer-to-peer networks based on social metaphors. In *Proceedings of the 13th international conference on World Wide Web, (WWW 2004), New York, NY, USA, May 17-20*, pages 640–649, 2004.

[100] D. Tsoumakos and N. Roussopoulos. A comparison of peer-to-peer search methods. In *WebDB*, pages 61–66, 2003.

[101] R. van Renesse and K. Birman. Scalable Management and Data Mining using Astrolabe. *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[102] S. Voulgaris, A.-M. Kermarrec, L. Massoulié, and M. van Steen. Exploiting semantic proximity in peer-to-peer content searching. In *10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2004)*, 2004.

[103] Q. Wang and M. T. Ozsu. A Data Locating Mechanism for Distributed XML Data over P2P Networks. *Technical report CS-2004-45, University of Waterloo*, 2004.

[104] D. J. Watts and S. H. Strogatz. Collective Dynamics of Small-World Networks. *Nature*, 393:440–442, 1998.

[105] H. Weatherspoon and J. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *IPTPS*, pages 328–338, 2002.

[106] P. Wu, C. Zhang, Y. Feng, B. Y. Zhao, D. Agrawal, and A. El Abbadi. Parallelizing Skyline Queries for Scalable Distribution. *EDBT 2006*, 2006.

[107] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *ICDCS*, pages 5–14, 2002.

[108] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *ICDCS*, pages 5–14, 2002.

[109] H. Zhang, A. Goel, and R. Govindan. Using the Small-World Model to Improve Freenet Performance. In *INFOCOM*, 2002.

[110] D. J. Zhao, D. L. Lee, and Q. Luo. DPTree: A Distributed Pattern Tree Index for Partial-Match Queries in Peer-to-Peer Networks. *EDBT 2005*, pages 515–532, 2005.

[111] C. Zheng, G. Shen, S. Lind, and S. Shenker. Distributed Segment Tree: Support of Range Query and Cover Query over DHT. In *IPTPS*, 2006.

[112] M. Zhong and K. Shen. Popularity-Biased Random Walks for Peer-to-Peer Search under the Square-Root Principle. In *IPTPS*, 2006.