Chapter 16
Peer-to-Peer Data Management

In this chapter, we discuss the data management issues in the "modern" peer-to-peer (P2P) data management systems. We intentionally use the phrase "modern" to differentiate these from the early P2P systems that were common prior to client/server computing. As indicated in Chapter 1, early work on distributed DBMSs had primarily focused on P2P architectures where there was no differentiation between the functionality of each site in the system. So, in one sense, P2P data management is quite old – if one simply interprets P2P to mean that there are no identifiable "servers" and "clients" in the system. However, the "modern" P2P systems go beyond this simple characterization and differ from the old systems that are referred to by the same name in a number of important ways, as mentioned in Chapter 1.

The first difference is the massive distribution in current systems. While the early systems focused on a few (perhaps at most tens of) sites, current systems consider thousands of sites. Furthermore, these sites are geographically very distributed, with possible clusters forming at certain locations.

The second is the inherent heterogeneity of every aspect of the sites and their autonomy. While this has always been a concern of distributed databases, coupled with massive distribution, site heterogeneity and autonomy take on added significance, disallowing some of the approaches from consideration.

The third major difference is the considerable volatility of these systems. Distributed DBMSs are well-controlled environments, where additions of new sites or the removal of existing sites is done very carefully and rarely. In modern P2P systems, the sites are (quite often) people's individual machines and they join and leave the P2P system at will, creating considerable hardship in the management of data.

In this chapter, we focus on this modern incarnation of P2P systems. In these systems, the following requirements are typically cited [Daswani et al., 2003]:

- **Autonomy.** An autonomous peer should be able to join or leave the system at any time without restriction. It should also be able to control the data it stores and which other peers can store its data (e.g., some other trusted peers).
- **Query expressiveness.** The query language should allow the user to describe the desired data at the appropriate level of detail. The simplest form of query

M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems: Third Edition*, DOI 10.1007/978-1-4419-8834-8_16, © Springer Science+Business Media, LLC 2011

is key look-up, which is only appropriate for finding files. Keyword search with ranking of results is appropriate for searching documents, but for more structured data, an SQL-like query language is necessary.

- **Efficiency.** The efficient use of the P2P system resources (bandwidth, computing power, storage) should result in lower cost, and, thus, higher throughput of queries, i.e., a higher number of queries can be processed by the P2P system in a given time interval.
- **Quality of service.** This refers to the user-perceived efficiency of the system, such as completeness of query results, data consistency, data availability, query response time, etc.
- **Fault-tolerance.** Efficiency and quality of service should be maintained despite the failures of peers. Given the dynamic nature of peers that may leave or fail at any time, it is important to properly exploit data replication.
- **Security.** The open nature of a P2P system gives rise to serious security challenges since one cannot rely on trusted servers. With respect to data management, the main security issue is access control which includes enforcing intellectual property rights on data contents.

A number of different uses of P2P systems have been developed [Valduriez and Pacitti, 2004]: they have been successfully used for sharing computation (e.g., SETI@home – http://www.setiathome.ssl.berkeley.edu), communication (e.g., ICQ – http://www.icq.com), or data sharing (e.g., Gnutella – http://www.gnutelliums.com – and Kazaa – http://www.kazaa.com). Our interest, naturally, is on data sharing systems. The commercial systems (such as Gnutella, Kazaa and others) are quite limited when viewed from the perspective of database functionality. Two important limitations are that they provide only file level sharing with no sophisticated content-based search/query facilities, and they are single-application systems that focus on performing one task, and it is not straightforward to extend them for other applications/functions [Ooi et al., 2003b]. In this chapter, we discuss the research activities towards providing proper database functionality over P2P infrastructures. Within this context, data management issues that must be addressed include the following:

- Data location: peers must be able to refer to and locate data stored in other peers.
- Query processing: given a query, the system must be able to discover the peers that contribute relevant data and efficiently execute the query.
- Data integration: when shared data sources in the system follow different schemas or representations, peers should still be able to access that data, ideally using the data representation used to model their own data.
- Data consistency: if data are replicated or cached in the system, a key issue is to maintain the consistency between these duplicates.

Figure 16.1 shows a reference architecture for a peer participating in a data sharing P2P system. Depending on the functionality of the P2P system, one or more

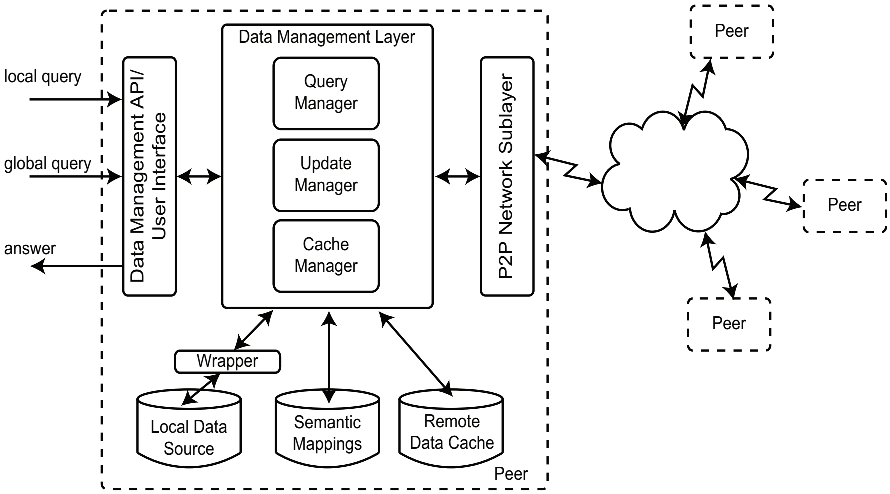


Fig. 16.1 Peer Reference Architecture

of the components in the reference architecture may not exist, may be combined together, or may be implemented by specialized peers. The key aspect of the proposed architecture is the separation of the functionality into three main components: (1) an interface used for submitting the queries; (2) a data management layer that handles query processing and metadata information (e.g., catalogue services); and (3) a P2P infrastructure, which is composed of the P2P network sublayer and P2P network. In this chapter, we focus on the P2P data management layer and P2P infrastructure.

Queries are submitted using a user interface or data management API and handled by the data management layer. Queries may refer to data stored locally or globally in the system. The query request is processed by a query manager module that retrieves semantic mapping information from a repository when the system integrates heterogeneous data sources. This semantic mapping repository contains meta-information that allows the query manager to identify peers in the system with data relevant to the query and to reformulate the original query in terms that other peers can understand. Some P2P systems may store the semantic mapping in specialized peers. In this case, the query manager will need to contact these specialized peers or transmit the query to them for execution. If all data sources in the system follow the same schema, neither the semantic mapping repository nor its associated query reformulation functionality are required.

Assuming a semantic mapping repository, the query manager invokes services implemented by the P2P network sublayer to communicate with the peers that will be involved in the execution of the query. The actual execution of the query is influenced by the implementation of the P2P infrastructure. In some systems, data are sent to the peer where the query was initiated and then combined at this peer. Other systems provide specialized peers for query execution and coordination. In either case, result data returned by the peers involved in the execution of the query may be cached locally to speed up future executions of similar queries. The cache manager maintains the local cache of each peer. Alternatively, caching may occur only at specialized peers.

The query manager is also responsible for executing the local portion of a global query when data are requested by a remote peer. A wrapper may hide data, query language, or any other incompatibilities between the local data source and the data management layer. When data are updated, the update manager coordinates the execution of the update between the peers storing replicas of the data being updated.

The P2P network infrastructure, which can be implemented as either structured or unstructured network topology, provides communication services to the data management layer.

In the remainder of this chapter, we will address each component of this reference architecture, starting with infrastructure issues in Section 16.1. The problems of data mapping and the approaches to address them are the topics of Section 16.2. Query processing is discussed in Section 16.3. Data consistency and replication issues are discussed in Section 16.4.

16.1 Infrastructure

The infrastructure of all P2P systems is a P2P network, which is built on top of a physical network (usually the Internet); thus it is commonly referred to as the *overlay network*. The overlay network may (and usually does) have a different topology than the physical network and all the algorithms focus on optimizing communication over the overlay network (usually in terms of minimizing the number of "hops" that a message needs to go through from a source node to a destination node – both in the overlay network). The possible disconnect between the overlay network and the physical network may be a problem in that two nodes that are neighbors in the overlay network may, in some cases, be considerably far apart in the physical network. Therefore, the cost of communication within the overlay network may not reflect the actual cost of communication in the physical network. We address this issue at the appropriate points during the infrastructure discussion.

Overlay networks can be of two general types: pure and hybrid. *Pure overlay networks* (more commonly referred to as *pure P2P networks*) are those where there is no differentiation between any of the network nodes – they are all equal. In *hybrid P2P networks*, on the other hand, some nodes are given special tasks to perform. Hybrid networks are commonly known as *super-peer systems*, since some of the peers are responsible for "controlling" a set of other peers in their domain. The pure networks can be further divided into structured and unstructured networks. *Structured networks* tightly control the topology and message routing, whereas in *unstructured networks* each node can directly communicate with its neighbors and can join the network by attaching themselves to any node.

16.1.1 Unstructured P2P Networks

Unstructured P2P networks refer to those with no restriction on data placement in the overlay topology. The overlay network is created in a nondeterministic (ad hoc) manner and the data placement is completely unrelated to the overlay topology. Each peer knows its neighbors, but does not know the resources that they have. Figure 16.2 shows an example unstructured P2P network.

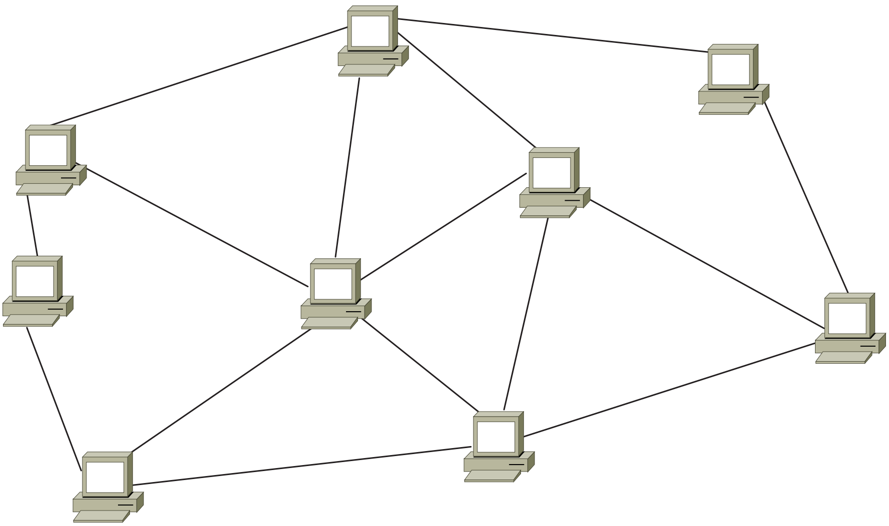


Fig. 16.2 Unstructured P2P Network

Unstructured networks are the earliest examples of P2P systems whose core functionality was (and remains) file sharing. In these systems replicated copies of popular files are shared among peers, without the need to download them from a centralized server. Examples of these systems are Napster (http://www.napster.com), Gnutella, Freenet [Clarke et al., 2000, 2002], Kazaa, and BitTorrent (http://www.bittorrent.com).

A fundamental issue in all P2P networks is the type of index to the resources that each peer holds, since this determines how resources are searched. Note that what is called "index management" in the context of P2P systems is very similar to catalog management that we studied in Chapter 3. Indexes are stored metadata that the system maintains. The exact content of the metadata differs in different P2P systems. In general, it includes, at a minimum, information on the resources and sizes.

There are two alternatives to maintaining indices: centralized, where one peer stores the metadata for the entire P2P system, and distributed, where each peer maintains metadata for resources that it holds. Again, the alternatives are identical to those for directory management. Napster is an example of a system that maintains a centralized index, while Gnutella maintains a distributed one.

The type of index supported by a P2P system (centralized or distributed) impacts how resources are searched. Note that we are not, at this point, referring to running queries; we are merely discussing how, given a resource identifier, the underlying P2P infrastructure can locate the relevant resource. In systems that maintain a centralized index, the process involves consulting the central peer to find the location of the resource, followed by directly contacting the peer where the resource is located (Figure 16.3). Thus, the system operates similar to a client/server one up to the point of obtaining the necessary index information (i.e., the metadata), but from that point on, the communication is only between the two peers. Note that the central peer may return a set of peers who hold the resource and the requesting peer may choose one among them, or the central peer may make the choice (taking into account loads and network conditions, perhaps) and return only a single recommended peer.

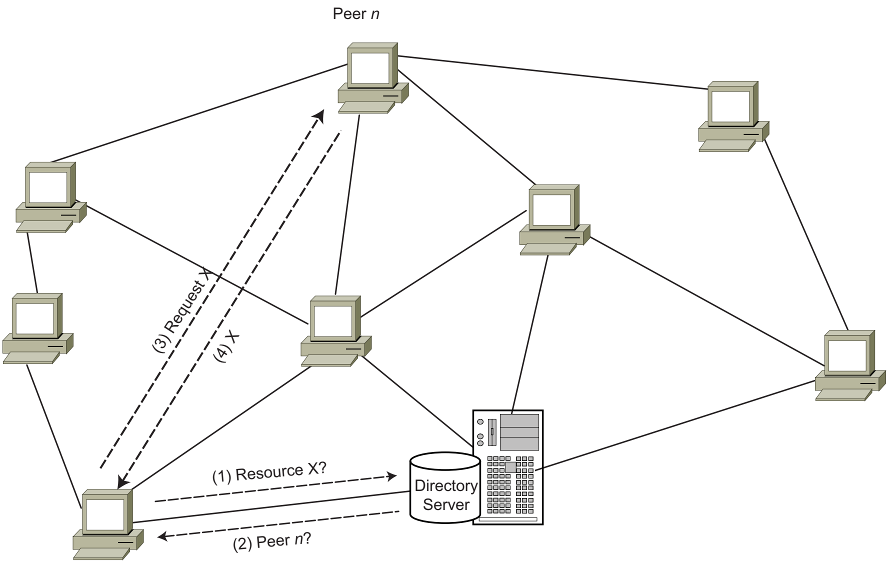


Fig. 16.3 Search over a Centralized Index. (1) A peer asks the central index manager for resource, (2) The response identifies the peer with the resource, (3) The peer is asked for the resource, (4) It is transferred.

In systems that maintain a distributed index, there are a number of search alternatives. The most popular one is flooding, where the peer looking for a resource sends the search request to all of its neighbors on the overlay network. If any of these neighbors have the resource, they respond; otherwise, each of them forwards the request to its neighbors until the resource is found or the overlay network is fully spanned (Figure 16.4).

Naturally, flooding puts very heavy demands on network resources and is not scalable – as the overlay network gets larger, more communication is initiated. This has been addressed by establishing a Time-to-Live (TTL) limit that restricts the

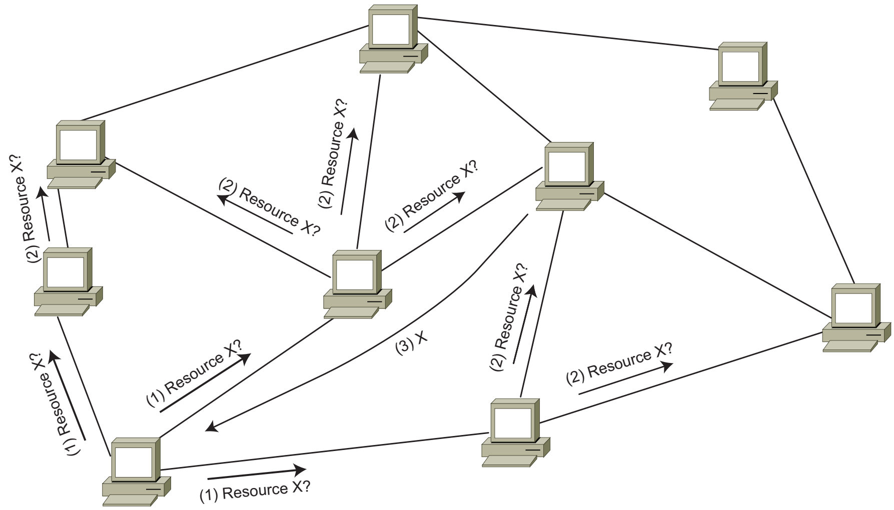


Fig. 16.4 Search over a Decentralized Index. (1) A peer sends the request for resource to all its neighbors, (2) Each neighbor propagates to its neighbors if it doesn't have the resource, (3) The peer who has the resource responds by sending the resource.

number of hops that a request message makes before it is dropped from the network. However, TTL also restricts the number of nodes that are reachable.

There have been other approaches to address this problem. A straightforward method is for each peer to choose a subset of its neighbors and forward the request only to those [Kalogeraki et al., 2002]. How this subset can be determined may vary. For example, the concept of random walks can be used [Lv et al., 2002] where each peer chooses a neighbor at random and propagates the request only to it. Alternatively, each neighbor can maintain not only indices for local resources, but also for resources that are on peers within a radius of itself and use the historical information about their performance in routing queries [Yang and Garcia-Molina, 2002]. Still another alternative is to use similar indices based on resources at each node to provide a list of neighbors that are most likely to be in the direction of the peer holding the requested resources [Crespo and Garcia-Molina, 2002]. These are referred to as routing indices and are used more commonly in structured networks, where we discuss them in more detail.

Another approach is to exploit *gossip protocols*, also known as *epidemic protocols* [Kermarrec and van Steen, 2007]. Gossiping has been initially proposed to maintain the mutual consistency of replicated data by spreading replica updates to all nodes over the network [Demers et al., 1987]. It has since been successfully used in P2P networks for data dissemination. Basic gossiping is simple. Each node in the network has a complete view of the network (i.e., a list of all nodes' addresses) and chooses a node at random to spread the request. The main advantage of gossiping is robustness over node failures since, with very high probability, the request is eventually propagated to all the nodes in the network. In large P2P networks, however, the basic gossiping model does not scale as maintaining the complete view of the network at each node would generate very heavy communication traffic. A solution to scalable gossiping is to maintain at each node only a partial view of the network, e.g., a list of tens of neighbour nodes [Voulgaris et al., 2003]. To gossip a request, a node chooses, at random, a node in its partial view and sends it the request. In addition, the nodes involved in a gossip exchange their partial views to reflect network changes in their own views. Thus, by continuously refreshing their partial views, nodes can self-organize into randomized overlays that scale up very well.

The final issue that we would like to discuss with respect to unstructured networks is how peers join and leave the network. The process is different for centralized versus distributed index approaches. In a centralized index system, a peer that wishes to join simply notifies the central index peer and informs it of the resources that it wishes to contribute to the P2P system. In the case of a distributed index, the joining peer needs to know one other peer in the system to which it "attaches" itself by notifying it and receiving information about its neighbors. At that point, the peer is part of the system and starts building its own neighbors. Peers that leave the system do not need to take any special action, they simply disappear. Their disappearance will be detected in time, and the overlay network will adjust itself.

16.1.2 Structured P2P Networks

Structured P2P networks have emerged to address the scalability issues faced by unstructured P2P networks [Ritter, 2001; Ratnasamy et al., 2001b; Stoica et al., 2001a]. They achieve this goal by tightly controlling the overlay topology and the placement of resources. Thus, they achieve higher scalability at the expense of lower autonomy as each peer that joins the network allows its resources to be placed on the network based on the particular control method that is used.

As with unstructured P2P networks, there are two fundamental issues to be addressed: how are the resources indexed, and how are they searched. The most popular indexing and data location mechanism that is used in structured P2P networks is *dynamic hash table* (DHT). DHT-based systems provide two API's: `put(key, data)` and `get(key)`, where key is an object identifier. The key is hashed to generate a peer id, which stores the data corresponding to object contents (Figure 16.5). Dynamic hashing has also been successfully used to address the scalability issues of very large distributed file structures [Devine, 1993; Litwin et al., 1993].

A straightforward approach could be to use the URI of the resource as the IP address of the peer that would hold the resource [Harvey et al., 2003]. However, one of the important design requirements is to provide a uniform distribution of resources over the overlay network and URIs/IP addresses do not provide sufficient flexibility. Consequently, *consistent hashing* techniques that provide uniform hashing of values are used to evenly place the data on the overlay. Although many hash functions may be employed for generating *virtual address mappings* for the resource, SHA-1 has

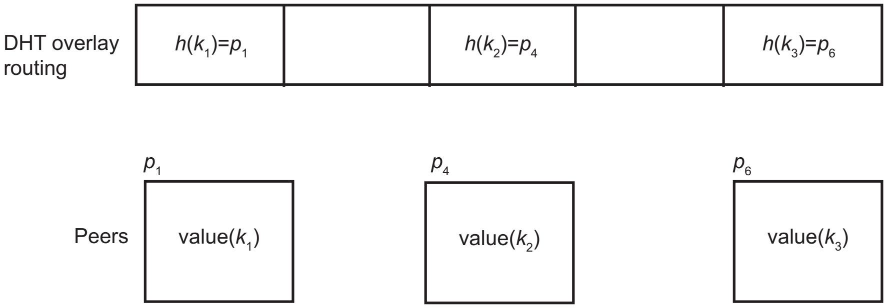


Fig. 16.5 DHT-based P2P Network

become the most widely accepted *base*[1] hash function that supports both uniformity as well as security (by supporting data-integrity for the keys). The actual design of the hash function may be implementation dependent and we won't discuss that issue any further.

Search (commonly called "lookup") over a DHT-based structured P2P network also involves the hash function: the key of the resource is hashed to get the id of the peer in the overlay network that is responsible for that key. The lookup is then initiated on the overlay network to locate the target node in question. This is referred to as the *routing protocol*, and it differs between different implementations and is closely associated with the overlay structure used. We will discuss one example approach shortly.

While all routing protocols aim to provide efficient lookups, they also try to minimize the *routing information* (also called *routing state*) that needs to be maintained in a routing table at each peer in the overlay. This information differs between various routing protocols and overlay structures, but it needs to provide sufficient directory-type information to route the `put` and `get` requests to the appropriate peer on the overlay. All routing table implementations require the use of maintenance algorithms in order to keep the routing state up-to-date and consistent. In contrast to routers on the Internet that also maintain routing databases, P2P systems pose a greater challenge since they are characterized by high node volatility and undependable network links. Since DHTs also need to support perfect recall (i.e., all the resources that are accessible through a given key have to be found), routing state consistency becomes a key challenge. Therefore, the maintenance of consistent routing state in the face of concurrent lookups and during periods of high network volatility is essential.

Many DHT-based overlays have been proposed. These can be categorized according to their *routing geometry* and *routing algorithm* [Gummadi et al., 2003]. Routing geometry essentially defines the manner in which neighbors and routes are arranged. The routing algorithm corresponds to the routing protocol discussed above

[1] A base hash function is defined as a function that is used as a basis for the design of another hash function.

and is defined as the manner in which next-hops/routes are chosen on a given routing geometry. The more important existing DHT-based overlays can be categorized as follows:

- **Tree.** In the tree approach, the leaf nodes correspond to the node identifiers that store the keys to be searched. The height of the tree is $\log(n)$, where n is the number of nodes in the tree. The search proceeds from the root to the leaves by doing a longest prefix match at each of the intermediate nodes until the target node is found. Therefore, in this case, matching can be thought of as correcting bit values from left-to-right at each successive hop in the tree. A popular DHT implementation that falls into this category is Tapestry [Zhao et al., 2004], which uses *surrogate routing* in order to forward requests at each node to the closest digit in the routing table. Surrogate routing is defined as routing to the *closest* digit when an exact match in the longest prefix cannot be found. In Tapestry, each unique identifier is associated with a node that is the root of a unique spanning tree used to route messages for the given identifier. Therefore, lookups proceed from the base of the spanning tree all the way to the root node of the identifier. Although this is somewhat different from traditional tree structures, Tapestry routing geometry is very closely associated to a tree structure and we classify it as such.
  In tree structures, a node in the system has 2^{i-1} nodes to choose from as its neighbor from the subtree with whom it has $\log(n-i)$ prefix bits in common. The number of potential neighbors increases exponentially as we proceed further up in the tree. Thus, in total there are $n^{\log(n)/2}$ possible routing tables per node (note, however that, only one such routing table can be selected for a node). Therefore, the tree geometry has good neighbor selection characteristics that would provide it with fault tolerance. However, routing can only be done through one neighboring node when sending to a particular destination. Consequently, the tree-structured DHTs do not provide any flexibility in the selection of routes.
- **Hypercube.** The hypercube routing geometry is based on d-dimensional Cartesian coordinate space that is partitioned into an individual set of zones such that each node maintains a separate zone of the coordinate space. An example of hypercube-based DHT is the Content Addressable Network (CAN) [Ratnasamy et al., 2001a]. The number of neighbors that a node may have in a d-dimensional coordinate space is $2d$ (for the sake of discussion, we consider $d = \log(n)$). If we consider each coordinate to represent a set of bits, then each node identifier can be represented as a bit string of length $\log(n)$. In this way, the hypercube geometry is very similar to the tree since it also simply *fixes* the bits at each hop to reach the destination. However, in the hypercube, since the bits of neighboring nodes only differ in *exactly* one bit, each forwarding node needs to modify only a single bit in the bit string, which can be done in any order. Thus, if we consider the correction of the bit string, the first correction can be applied to any $\log(n)$ nodes, the next correction can be applied to any $\log(n)-1$ nodes, etc. Therefore, we have $\log(n)!$ possible routes between

nodes which provides high route flexibility in the hypercube routing geometry. However, a node in the coordinate space does not have any choice over its neighbors' coordinates since adjacent coordinate zones in the coordinate space can't change. Therefore, hypercubes have poor neighbor selection flexibility.

- **Ring.** The ring geometry is represented as a one-dimensional circular identifier space where the nodes are placed at different locations on the circle. The distance between any two nodes on the circle is the numeric identifier difference (clockwise) around the circle. Since the circle is one-dimensional, the data identifiers can be represented as single decimal digits (represented as binary bit strings) that map to a node that is closest in the identifier space to the given decimal digit. Chord [Stoica et al., 2001b] is a popular example of the ring geometry. Specifically, in Chord, a node whose identifier is a maintains information about $\log(n)$ other neighbors on the ring where the i^{th} neighbor is the node closest to $a + 2^{i-1}$ on the circle. Using these links (called *fingers*), Chord is able to route to any other node in $\log(n)$ hops.
  A careful analysis of Chord's structure reveals that a node does not necessarily need to maintain the node closest to $a + 2^{i-1}$ as its neighbor. In fact, it can still maintain the $\log(n)$ lookup upper bound if any node from the range $[(a + 2^{i-1}), (a + 2^i)]$ is chosen. Therefore, in terms of route flexibility, it is able to select between $n^{\log(n)/2}$ routing tables for each node. This provides a great deal of neighbor selection flexibility. Moreover, for routing to any node, the first hop has $\log(n)$ neighbors that can route the search to the destination and the next node has $\log(n) - 1$ nodes, and so on. Therefore, there are typically $\log(n)!$ possible routes to the destination. Consequently, ring geometry also provides good route selection flexibility.

In addition to these most popular geometries, there have been many other DHT-based structured overlays that have been proposed that use different topologies. Some of these are Viceroy [Malkhi et al., 2002], Kademlia [Maymounkov and Mazières, 2002], and Pastry [Rowstron and Druschel, 2001].

DHT-based overlays are efficient in that they guarantee finding the node on which to place or find the data in $\log(n)$ hops where n is the number of nodes in the system. However, they have a number of problems, in particular when viewed from the data management perspective. One of the issues with DHTs that employ consistent hashing functions for better distribution of resources is that two peers that are "neighbors" in the overlay network because of the proximity of their hash values may be geographically quite apart in the actual network. Thus, communicating with a neighbor in the overlay network may incur high transmission delays in the actual network. There have been studies to overcome this difficulty by designing *proximity-aware* or *locality-aware* hash functions. Another difficulty is that they do not provide any flexibility in the placement of data – a data item has to be placed on the node that is determined by the hash function. Thus, if there are P2P nodes that contribute their own data, they need to be willing to have data moved to other nodes. This is problematic from the perspective of node autonomy. The third difficulty is in that it is hard to run range queries over DHT-based architectures since, as is well-known, it is hard to run range queries over hash indices. There have been studies to overcome this difficulty that we discuss later.

These concerns have caused the development of structured overlays that do not use DHT for routing. In these systems, peers are mapped into the data space rather than the hash key space. There are multiple ways to partition the data space among multiple peers.

- **Hierarchical structure.** Many systems employ hierarchical overlay structures, including trie, balanced trees, randomized balance trees (e.g., skip list [Pugh, 1989]), and others. Specifically PHT [Ramabhadran et al., 2004] and P-Grid [Aberer, 2001; Aberer et al., 2003a] employ a binary trie structure, where peers whose data share common prefixes cluster under common branches. Balanced trees are also widely used due to their guaranteed routing efficiency (the expected "hop length" between arbitrary peers is proportional to the tree height). For instance, BATON [Jagadish et al., 2005], VBI-tree [Jagadish et al., 2005], and BATON* [Jagadish et al., 2006] employ k-way balanced tree structure to manage peers, and data are evenly partitioned among peers at the leaf-level. In comparison, P-Tree [Crainiceanu et al., 2004] uses a B-tree structure with better flexibility on tree structural changes. SkipNet [Harvey et al., 2003] and Skip Graph [Aspnes and Shah, 2003] are based on the skip list, and they link peers according to a randomized balanced tree structure where the node order is determined by each node's data values.
- **Space-filling curve.** This architecture is usually used to linearize sort data in multi-dimensional data space. Peers are arranged along the space-filling curve (e.g., Hilbert curve) so that sorted traversal of peers according to data order is possible [Schmidt and Parashar, 2004].
- **Hyper-rectangle structure.** In these systems, each dimension of the hyper-rectangle corresponds to one attribute of the data according to which an organization is desired. Peers are distributed in the data space either uniformly or based on data locality (e.g., through data intersection relationship). The hyper-rectangle space is then partitioned by peers based on their geometric positions in the space, and neighboring peers are interconnected to form the overlay network [Ganesan et al., 2004].

16.1.3 Super-peer P2P Networks

Super-peer P2P systems are hybrid between pure P2P systems and the traditional client-server architectures. They are similar to client-server architectures in that not all peers are equal; some peers (called *super-peers*) act as dedicated serves for some other peers and can perform complex functions such as indexing, query processing, access control, and meta-data management. If there is only one super-peer in the system, then this reduces to the client-server architecture. They are considered P2P systems, however, since the organization of the super-peers follow P2P organization, and super-peers can communicate with each other in sophisticated ways. Thus, unlike client-server systems, global information is not necessarily centralized and can be partitioned or replicated across super-peers.

In a super-peer network, a requesting peer sends the request, which can be expressed in a high-level language, to its responsible super-peer. The super-peer can then find the relevant peers either directly through its index or indirectly using its neighbor super-peers. More precisely, the search for a resource proceeds as follows (see Figure 16.6):

1. A peer, say Peer 1, asks for a resource by sending a request to its super-peer.
2. If the resource exists at one of the peers controlled by this super-peer, it notifies Peer 1, and the two peers then communicate to retrieve the resource. Otherwise, the super-peer sends the request to the other super-peers.
3. If the resource does not exist at one of the peers controlled by this super-peer, the super-peer asks the other super-peers. The super-peer of the node that contains the resource (say Peer *n*) responds to the requesting super-peer.
4. Peer *n*'s identity is sent to Peer 1, after which the two peers can communicate directly to retrieve the resource.

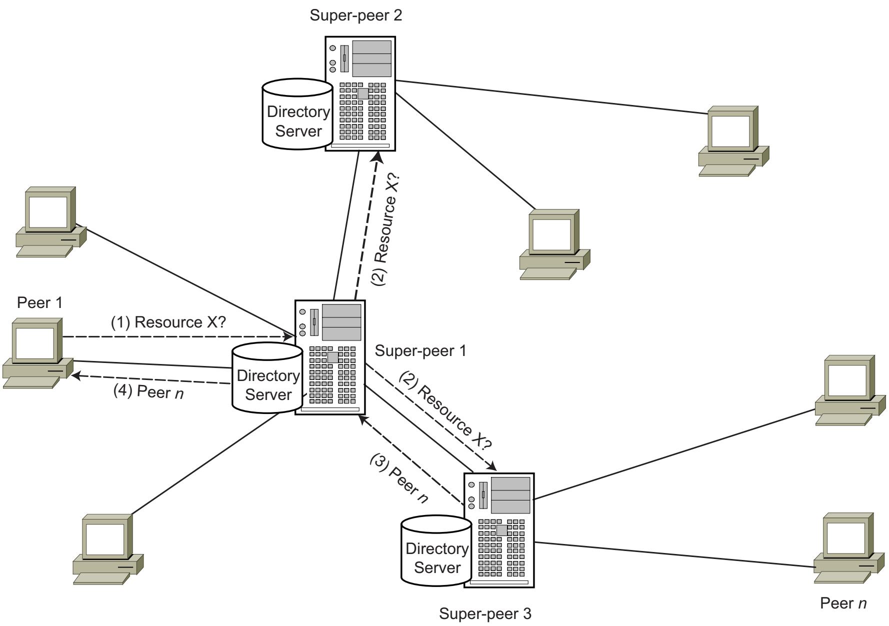


Fig. 16.6 Search over a Super-peer System. (1) A peer sends the request for resource to all its super-peer, (2) The super-peer sends the request to other super-peers if necessary, (3) The super-peer one of whose peers has the resource responds by indicating that peer, (4) The super-peer notifies the original peer.

Requirements	Unstructured	Structured	Super-peer
Autonomy	Low	Low	Moderate
Query expressiveness	High	Low	High
Efficiency	Low	High	High
QoS	Low	High	High
Fault-tolerance	High	High	Low
Security	Low	Low	High

Fig. 16.7 Comparison of Approaches.

The main advantages of super-peer networks are efficiency and quality of service (e.g., completeness of query results, query response time, etc.). The time needed to find data by directly accessing indices in a super-peer is very small compared with flooding. In addition, super-peer networks exploit and take advantage of peers' different capabilities in terms of CPU power, bandwidth, or storage capacity as super-peers take on a large portion of the entire network load. Access control can also be better enforced since directory and security information can be maintained at the super-peers. However, autonomy is restricted since peers cannot log in freely to any super-peer. Fault-tolerance is typically lower since super-peers are single points of failure for their sub-peers (dynamic replacement of super-peers can alleviate this problem).

Examples of super-peer networks include Edutella [Nejdl et al., 2003] and JXTA (http://www.jxta.org).

16.1.4 Comparison of P2P Networks

Figure 16.7 summarizes how the requirements for data management (autonomy, query expressiveness, efficiency, quality of service, fault-tolerance, and security) are possibly attained by the three main classes of P2P networks. This is a rough comparison to understand the respective merits of each class. Obviously, there is room for improvement in each class of P2P networks. For instance, fault-tolerance can be improved in super-peer systems by relying on replication and fail-over techniques. Query expressiveness can be improved by supporting more complex queries on top of structured networks.

16.2 Schema Mapping in P2P Systems

We discussed the importance of, and the techniques for, designing database integration systems in Chapter 4. Similar issues arise in data sharing P2P systems.

Due to specific characteristics of P2P systems, e.g., the dynamic and autonomous nature of peers, the approaches that rely on centralized global schemas no longer apply. The main problem is to support decentralized schema mapping so that a query expressed on one peer's schema can be reformulated to a query on another peer's schema. The approaches which are used by P2P systems for defining and creating the mappings between peers' schemas can be classified as follows: pairwise schema mapping, mapping based on machine learning techniques, common agreement mapping, and schema mapping using information retrieval (IR) techniques.

16.2.1 Pairwise Schema Mapping

In this approach, each user defines the mapping between the local schema and the schema of any other peer that contains data that are of interest. Relying on the transitivity of the defined mappings, the system tries to extract mappings between schemas that have no defined mapping.

Piazza [Tatarinov et al., 2003] follows this approach (see Figure 16.8). The data are shared as XML documents, and each peer has a schema that defines the terminology and the structural constraints of the peer. When a new peer (with a new schema) joins the system for the first time, it maps its schema to the schema of some other peers in the system. Each mapping definition begins with an XML template that matches some path or subtree of an instance of the target schema. Elements in the template may be annotated with query expressions that bind variables to XML nodes in the source. Active XML [Abiteboul et al., 2002, 2008b] also relies on XML documents for data sharing. The main innovation is that XML documents are active in the sense that they can include Web service calls. Therefore, data and queries can be seamlessly integrated. We discuss this further in Chapter 17.

The Local Relational Model (LRM) [Bernstein et al., 2002] is another example that follows this approach. LRM assumes that the peers hold relational databases, and each peer knows a set of peers with which it can exchange data and services. This set of peers is called peer's *acquaintances*. Each peer must define semantic dependencies and translation rules between its data and the data shared by each of its acquaintances. The defined mappings form a semantic network, which is used for query reformulation in the P2P system. Hyperion [Kementsietsidis et al., 2003] generalizes this approach to deal with autonomous peers that form acquaintances at run-time, using mapping tables to define value correspondences among heterogeneous databases. Peers perform local querying and update processing, and also propagate queries and updates to their acquainted peers.

PGrid [Aberer et al., 2003b] also assumes the existence of pairwise mappings between peers, initially constructed by skilled experts. Relying on the transitivity of these mappings and using a gossip algorithm, PGrid extracts new mappings that relate the schemas of the peers between which there is no predefined schema mapping.

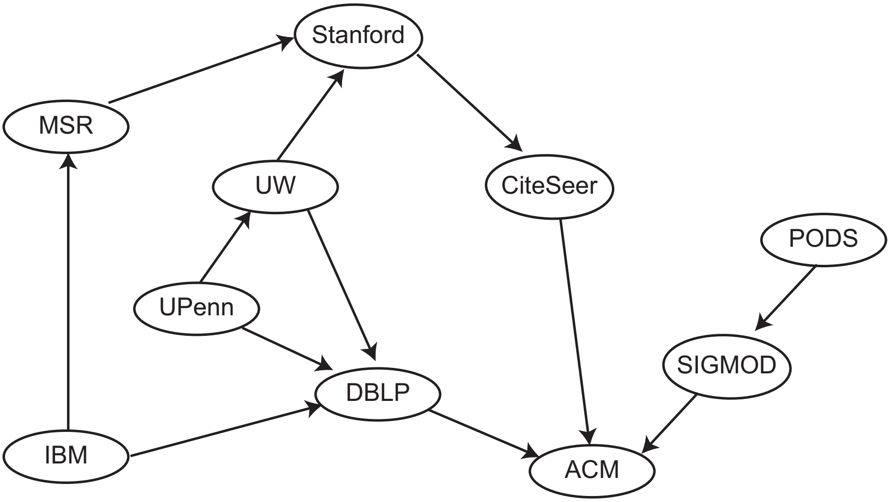


Fig. 16.8 An Example of Pairwise Schema Mapping in Piazza

16.2.2 Mapping based on Machine Learning Techniques

This approach is generally used when the shared data are defined based on ontologies and taxonomies as proposed for the semantic web. It uses machine learning techniques to automatically extract the mappings between the shared schemas. The extracted mappings are stored over the network, in order to be used for processing future queries. GLUE [Doan et al., 2003b] uses this approach. Given two ontologies, for each concept in one, GLUE finds the most similar concept in the other. It gives well founded probabilistic definitions to several practical similarity measures, and uses multiple learning strategies, each of which exploits a different type of information either in the data instances or in the taxonomic structure of the ontologies. To further improve mapping accuracy, GLUE incorporates commonsense knowledge and domain constraints into the schema mapping process. The basic idea is to provide classifiers for the concepts. To decide the similarity between two concepts A and B, the data of concept B are classified using A's classifier and vice versa. The amount of values that can be successfully classified into A and B represent the similarity between A and B.

16.2.3 Common Agreement Mapping

In this approach, the peers that have a common interest agree on a common schema description for data sharing. The common schema is usually prepared and maintained by expert users. APPA [Akbarinia et al., 2006a; Akbarinia and Martins, 2007] makes the assumption that peers wishing to cooperate, e.g., for the duration of an experiment, agree on a Common Schema Description (CSD). Given a CSD, a peer schema can be specified using views. This is similar to the LAV approach in data integration systems, except that queries at a peer are expressed in terms of the local views, not the CSD. Another difference between this approach and LAV is that the CSD is not a global schema, i.e., it is common to a limited set of peers with a common interest (see Figure 16.9). Thus, the CSD does not pose scalability challenges. When a peer decides to share data, it needs to map its local schema to the CSD.

Example 16.1. Given two CSD relation definitions r_1 and r_2, an example of peer mapping at peer p is:

$$p : r(A,B,D) \subseteq csd : r_1(A,B,C), csd : r_2(C,D,E)$$

In this example, the relation $r(A,B,D)$ that is shared by peer p is mapped to relations $r_1(A,B,C)$, $r_2(C,D,E)$ both of which are involved in the CSD. In APPA, the mappings between the CSD and each peer's local schema are stored locally at the peer. Given a query Q on the local schema, the peer reformulates Q to a query on the CSD using locally stored mappings. ♦

AutoMed [McBrien and Poulovassilis, 2003] is another system that relies on common agreements for schema mapping. It defines the mappings by using primitive bidirectional transformations defined in terms of a low-level data model.

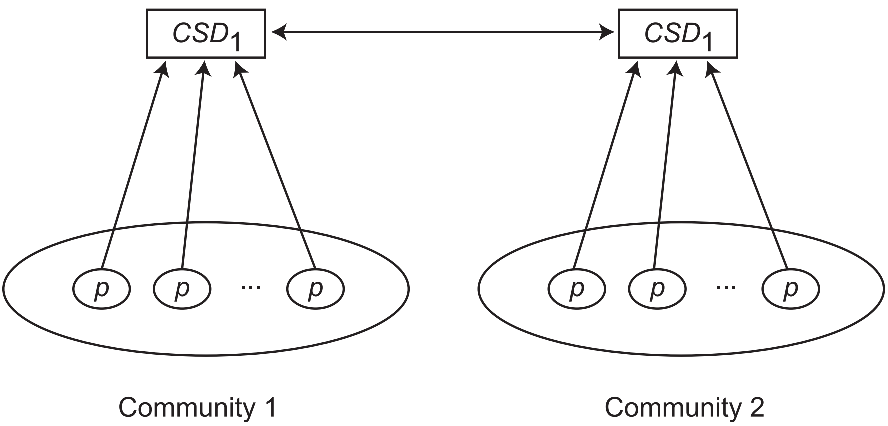


Fig. 16.9 Common Agreement Schema Mapping in APPA

16.2.4 Schema Mapping using IR Techniques

This approach extracts the schema mappings at query execution time using IR techniques by exploring the schema descriptions provided by users. PeerDB [Ooi et al., 2003a] follows this approach for query processing in unstructured P2P networks. For each relation that is shared by a peer, the description of the relation and its attributes is maintained at that peer. The descriptions are provided by users upon creation of relations, and serve as a kind of synonymous names of relation names and attributes. When a query is issued, a request to find out potential matches is produced and flooded to the peers that return the corresponding metadata. By matching keywords from the metadata of the relations, PeerDB is able to find relations that are potentially similar to the query relations. The relations that are found are presented to the issuer of the query who decides whether or not to proceed with the execution of the query at the remote peer that owns the relations.

Edutella [Nejdl et al., 2003] also follows this approach for schema mapping in super-peer networks. Resources in Edutella are described using the RDF metadata model, and the descriptions are stored at super-peers. When a user issues a query at a peer p, the query is sent to p's super-peer where the stored schema descriptions are explored and the addresses of the relevant peers are returned to the user. If the super-peer does not find relevant peers, it sends the query to other super-peers such that they search relevant peers by exploring their stored schema descriptions. In order to explore stored schemas, super-peers use the RDF-QEL query language, which is based on Datalog semantics and thus compatible with all existing query languages, supporting query functionalities that extend the usual relational query languages.

16.3 Querying Over P2P Systems

P2P networks provide basic techniques for routing queries to relevant peers and this is sufficient for supporting simple, exact-match queries. For instance, as noted earlier, a DHT provides a basic mechanism to efficiently look up data based on a key value. However, supporting more complex queries in P2P systems, particularly in DHTs, is difficult and has been the subject of much recent research. The main types of complex queries which are useful in P2P systems are top-k queries, join queries, and range queries. In this section, we discuss the techniques for processing them.

16.3.1 Top-k Queries

Top-k queries have been used in many domains such as network and system monitoring, information retrieval, and multimedia databases [Ilyas et al., 2008]. With a top-k query, the user requests k most relevant answers to be returned by the system. The degree of relevance (score) of the answers to the query is determined by a scoring function. Top-k queries are very useful for data management in P2P systems, in particular when the number of all the answers is very large [Akbarinia et al., 2006b].

Example 16.2. Consider a P2P system with medical doctors who want to share some (restricted) patient data for an epidemiological study. Assume that all doctors agreed on a common Patient description in relational format. Then, one doctor may want to submit the following query to obtain the top 10 answers ranked by a scoring function over height and weight:

```
SELECT      *
FROM        Patient P
WHERE       P.disease = ``diabetes''
AND         P.height < 170
AND         P.weight > 160
ORDER BY    scoring-function(height,weight)
STOP AFTER 10
```

The scoring function specifies how closely each data item matches the conditions. For instance, in the query above, the scoring function could compute the ten most overweight people. ♦

Efficient execution of top-k queries in large-scale P2P systems is difficult. In this section, we first discuss the most efficient techniques proposed for top-k query processing in distributed systems. Then, we present the techniques proposed for P2P systems.

16.3.1.1 Basic Techniques

An efficient algorithm for top-k query processing in centralized and distributed systems is the Threshold Algorithm (TA) [Nepal and Ramakrishna, 1999; Güntzer et al., 2000; Fagin et al., 2003]. TA is applicable for queries where the scoring function is monotonic, i.e., any increase in the value of the input does not decrease the value of the output. Many of the popular aggregation functions such as Min, Max, and Average are monotonic. TA has been the basis for several algorithms, and we discuss these in this section.

Threshold Algorithm (TA).

TA assumes a model based on lists of data items sorted by their local scores [Fagin, 1999]. The model is as follows. Suppose we have m lists of n data items such that each data item has a local score in each list and the lists are sorted according to the local scores of their data items. Furthermore, each data item has an overall score that is computed based on its local scores in all lists using a given scoring function. For example, consider the database (i.e., three sorted lists) in Figure 16.10. Assuming the scoring function computes the sum of the local scores of the same data item in all lists, the overall score of item d_1 is $30+21+14=65$.

Then the problem of top-k query processing is to find the k data items whose overall scores are the highest. This problem model is simple and general. Suppose we want to find the top-k tuples in a relational table according to some scoring function over its attributes. To answer this query, it is sufficient to have a sorted (indexed) list of the values of each attribute involved in the scoring function, and return the k tuples whose overall scores in the lists are the highest. As another example, suppose we want to find the top-k documents whose aggregate rank is the highest with respect to some given set of keywords. To answer this query, the solution is to have, for each keyword, a ranked list of documents, and return the k documents whose aggregate rank over all lists are the highest.

TA considers two modes of access to a sorted list. The first mode is sorted (or sequential) access that accesses each data item in their order of appearance in the list. The second mode is random access by which a given data item in the list is directly looked up, for example, by using an index on item id.

Given the m sorted lists of n data items, TA (see Algorithm 16.1), goes down the sorted lists in parallel, and, for each data item, retrieves its local scores in all lists through random access and computes the overall score. It also maintains in a set Y the k data items whose overall scores are the highest so far. The stopping mechanism of TA uses a threshold that is computed using the last local scores seen under sorted access in the lists. For example, consider the database in Figure 16.10. At position 1 for all lists (i.e., when only the first data items have been seen under sorted access) assuming that the scoring function is the sum of the scores, the threshold is $30+28+30=88$. At position 2, it is 84. Since data items are sorted in the lists in decreasing order of local score, the threshold decreases as one moves down the list. This process continues until k data items are found whose overall scores are greater than a threshold.

Example 16.3. Consider again the database (i.e., three sorted lists) shown in Figure 16.10. Assume a top-3 query Q (i.e., $k=3$), and suppose the scoring function computes the sum of the local scores of the data item in all lists. TA first looks at the data items which are at position 1 in all lists, i.e., d_1, d_2, and d_3. It looks up the local scores of these data items in other lists using random access and computes their overall scores (which are 65, 63 and 70, respectively). However, none of them has an overall score that is as high as the threshold of position 1 (which is 88). Thus, at position 1, TA does not stop. At this position, we have $Y=\{d_1,d_2,d_3\}$, i.e., the k highest scored data items seen so far. At positions 2 and 3, Y is set to $\{d_3,d_4,d_5\}$ and $\{d_3,d_5,d_8\}$ respectively. Before position 6, none of the data items involved in Y has an overall score higher than or equal to the threshold value. At position 6, the threshold value is 63, which is less than the overall score of the three data items involved in Y, i.e., $Y=\{d_3,d_5,d_8\}$. Thus, TA stops. Note that the contents of Y at position 6 is exactly the same as at position 3. In other words, at position 3, Y already contains all top-k answers. In this example, TA does three additional sorted accesses in each list that do not contribute to the final result. This is a characteristic of TA algorithm in that it has a conservative stopping condition that causes it to stop later than necessary – in this example, it performs 9 sorted accesses and $18=(9*2)$ random accesses that do not contribute to the final result. ♦

Algorithm 16.1: Threshold Algorithm (TA)

```
Input: L1, L2, ..., Lm: m sorted lists of n data items ;
f: scoring function
Output: Y: list of top-k data items
begin
    j ← 1 ;
    threshold ← 1 ;
    min_overall_score ← 0 ;
    while j ≠ n+1 and min_overall_score < threshold do
        {Do sorted access in parallel to each of the m sorted lists}
        for i from 1 to m  in parallel do
            {Process each data item at position j}
            for each data item d at position j in Li do
                {access the local scores of d in the other lists through random
                access}
                overall_score(d) ← f(scores of d in each Li)
        Y ← k data items with highest score so far ;
        min_overall_score ← smallest overall score of data items in Y ;
        threshold ← f(local scores at position j in each Li) ;
        j ← j+1
end
```

TA-Style Algorithms.

Several TA-style algorithms, i.e., extensions of TA, have been proposed for distributed top-k query processing. We illustrate these by means of the Three Phase Uniform Threshold (TPUT) algorithm that executes top-k queries in three round trips [Cao and Wang, 2004], assuming that each list is held by one node (which we call the *list holder*) and that the scoring function is sum. The TPUT algorithm (see Algorithm 16.2 executed by the query originator) works as follows.

1. The query originator first gets from each list holder its k top data items. Let f be the scoring function, d be a received data item, and $s_i(d)$ be the local score of d in list L_i. Then the partial sum of d is defined as $psum(d) = \sum_{i=1}^{m} s_i'(d)$ where $s_i'(d) = s_i(d)$ if d has been sent to the coordinator by the holder of L_i, else $s_i'(d) = 0$. The query originator computes the partial sums for all received data items and identifies the items with the k highest partial sums. The partial sum of the $k-$th data item (called *phase-1 bottom*) is denoted by λ_1.
2. The query originator sends a threshold value $\tau = \lambda_1/m$ to every list holder. In response, each list holder sends back all its data items whose local scores are not less than τ. The intuition is that if a data item is not reported by any node in this phase, its score must be less than λ_1, so it cannot be one of the

Position	List 1		List 2		List 3	
	Data Item	Local score s_1	Data Item	Local score s_2	Data Item	Local score s_3
1	d_1	30	d_2	28	d_3	30
2	d_4	28	d_6	27	d_5	29
3	d_9	27	d_7	25	d_8	28
4	d_3	26	d_5	24	d_4	25
5	d_7	25	d_9	23	d_2	24
6	d_8	23	d_1	21	d_6	19
7	d_5	17	d_8	20	d_{13}	15
8	d_6	14	d_3	14	d_1	14
9	d_2	11	d_4	13	d_9	12
10	d_{11}	10	d_{14}	12	d_7	11
...	...	...	...	...	...	...

Fig. 16.10 Example database with 3 sorted lists

top-k data items. Let Y be the set of data items received from list holders. The query originator computes the new partial sums for the data items in Y, and identifies the items with the k highest partial sums. The partial sum of the k-th data item (called phase-2 bottom) is denoted by λ_2. Let the upper bound score of a data item d be defined as $u(d) = \sum_{i=1}^{m} u_i(d)$ where $u_i(d) = s_i(d)$ if d has been received, else $u_i(d) = \tau$. For each data item $d \in D$, if $u(d)$ is less than λ_2, it is removed from Y. The data items that remain in Y are called top-k candidates because there may be some data items in Y that have not been obtained from all list holders. A third phase is necessary to retrieve those.

3. The query originator sends the set of top-k candidate data items to each list holder that returns their scores. Then, it computes the overall score, extracts the k data items with highest scores, and returns the answer to the user.

Example 16.4. Consider the first two sorted lists (List 1 and List 2) in Figure 16.10. Assume a top-2 query Q, i.e., $k = 2$, where the scoring function is sum. Phase 1 produces the sets $Y = \{d_1, d_2, d_4, d_6\}$ and $Z = \{d_1, d_2\}$. Thus we get $\lambda_1/2 = 28/2 = 14$. Let us now denote each data item d in Y as $(d, scoreinList1, scoreinList2)$. Phase 2 produces $Y = \{(d_1, 30, 21), (d_2, 0, 28), (d_3, 26, 14), (d_4, 28, 0), (d_5, 17, 24), (d_6, 14, 27), (d_7, 25, 25), (d_8, 23, 20), (d_9, 27, 23)\}$ and $Z = \{(d_1, 30, 21), (d_7, 25, 25)\}$. Note that d_9 could also have been picked instead of d_7 because it has same partial sum. Thus we get $\lambda_2/2$=50. The upper bound scores of the data items in Y are obtained as:

$$u(d_1) = 30 + 21 = 51$$
$$u(d_2) = 14 + 28 = 42$$
$$u(d_3) = 26 + 14 = 40$$

Algorithm 16.2: Three Phase Uniform Threshold(TPUT)

Input: $L_1, L_2, \ldots, L_m$: m sorted lists of n data items, each at a different list holder;
f: scoring function
Output: Y: list of top-k data items
begin
    {Phase 1}
    for *i from* 1 *to m in parallel* **do**
        $Y \leftarrow$ receive top-k data items from L_i holder
    $Z \leftarrow$ data items with the k highest partial sum in Y ;
    $\lambda_1 \leftarrow$partial sum of k-th data item in Z ;
    {Phase 2}
    for *i from* 1 *to m in parallel* **do**
        send λ_1/m to L_i's holder ;
        $Y \leftarrow$ all data items from L_i's holder whose local scores are not less than λ_1/m
    $Z \leftarrow$ data items with the k highest partial sum in Y ;
    $\lambda_2 \leftarrow$ partial sum of k-th data item in Z ;
    $Y \leftarrow Y - \{$data items in Y whose upper bound score is less than $\lambda_2\}$;
    {Phase 3}
    for *i from* 1 *to m in parallel* **do**
        send Y to L_i holder ;
        $Z \leftarrow$ data items from L_i's holder that are in both Y and L_i
    $Y \leftarrow k$ data items with highest overall score in Z
end

$u(d_4) = 28 + 14 = 42$
$u(d_5) = 17 + 24 = 41$
$u(d_6) = 14 + 27 = 41$
$u(d_7) = 25 + 25 = 50$
$u(d_8) = 23 + 20 = 43$
$u(d_9) = 27 + 23 = 50$

After removal of the data items in Y whose upper bound score is less than λ_2, we have $Y = \{d_1, d_7, d_9\}$. The third phase is not necessary in this case as all data items have all their local scores. Thus the final result is $Y = \{d_1, d_7\}$ or $Y = \{d_1, d_9\}$. ♦

When the number of lists (i.e., m) is high, the response time of TPUT is much better than that of the basic TA algorithm [Cao and Wang, 2004].

Best Position Algorithm (BPA).

There are many database instances over which TA keeps scanning the lists although it has seen all top-k answers (as in Example 16.3). Thus, it is possible to stop much sooner. Based on this observation, best position algorithms (BPA) that execute top-k queries much more efficiently than TA have been proposed [Akbarinia et al., 2007a]. The key idea of BPA is that the stopping mechanism takes into account special seen positions in the lists, called the *best positions*. Intuitively, the best position in a list is the highest position such that any position before it has also been seen. The stopping condition is based on the overall score computed using the best positions in all lists.

The basic version of BPA (see Algorithm 16.3) works like TA, except that it keeps track of all positions that are seen under sorted or random access, computes best positions, and has a different stopping condition. For each list L_i, let P_i be the set of positions that are seen under sorted or random access in L_i. Let bp_i, the best position in L_i, be the highest position in P_i such that any position of L_i between 1 and bp_i is also in P_i. In other words, bp_i is best because we are sure that all positions of L_i between 1 and bpi have been seen under sorted or random access. Let $s_i(bp_i)$ be the local score of the data item that is at position bp_i in list L_i. Then, BPA's threshold is $f(s_1(bp_1), s_2(bp_2), \ldots, s_m(bp_m))$ for some function f.

Example 16.5. To illustrate basic BPA, consider again the three sorted lists shown in Figure 16.10 and the query Q in Example 16.3.

1. At position 1, BPA sees the data items d_1, d_2, and d_3. For each seen data item, it does random access and obtains its local score and position in all the lists. Therefore, at this step, the positions that are seen in list L_1 are positions 1, 4, and 9, which are respectively the positions of d_1, d_3 and d_2. Thus, we have $P_1 = \{1,4,9\}$ and the best position in L_1 is $bp_1 = 1$ (since the next position is 4 meaning that positions 2 and 3 have not been seen). For L_2 and L_3 we have $P_2 = \{1,6,8\}$ and $P_3 = \{1,5,8\}$, so $bp_2 = 1$ and $bp_3 = 1$. Therefore, the best positions overall score is $\lambda = f(s_1(1), s_2(1), s_3(1)) = 30+28+30 = 88$. At position 1, the set of the three highest scored data items is $Y = \{d_1, d_2, d_3\}$, and since the overall score of these data items is less than λ , BPA cannot stop.
2. At position 2, BPA sees d_4, d_5, and d_6. Thus, we have $P_1 = \{1,2,4,7,8,9\}$, $P_2 = \{1,2,4,6,8,9\}$ and $P_3 = \{1,2,4,5,6,8\}$. Therefore, we have $bp_1 = 2$, $bp_2 = 2$ and $bp_3 = 2$, so $\lambda = f(s_1(2), s_2(2), s_3(2)) = 28+27+29 = 84$. The overall score of the data items involved in $Y = \{d_3, d_4, d_5\}$ is less than 84, so BPA does not stop.
3. At position 3, BPA sees d_7, d_8, and d_9. Thus, we have $P_1 = P_2 = \{1,2,3,4,5,6,7,8,9\}$, and $P_3 = \{1,2,3,4,5,6,7,8,10\}$. Thus, we have $bp_1 = 9$, $bp_2 = 9$ and $bp_3 = 8$. The best positions overall score is $\lambda = f(s_1(9), s_2(9), s_3(8)) = 11+13+14 = 38$. At this position, we have $Y = \{d_3, d_5, d_8\}$. Since the score of all data items involved in Y is higher than λ, BPA stops, i.e., exactly at the first position where BPA has all top-k answers.

Algorithm 16.3: Best Position Algorithm (BPA)

```
Input: L_1, L_2, ..., L_m: m sorted lists of n data items ;
f: scoring function
Output: Y: list of top-k data items
begin
    j ← 1 ;
    threshold ← 1 ;
    min_overall_score ← 0 ;
    for i from 1 to m in parallel do
        P_i ← ∅
    while j ≠ n+1 and min_overall_score < threshold do
        {Do sorted access in parallel to each of the m sorted lists}
        for i from 1 to m in parallel do
            {Process each data item at position j}
            for each data item d at position j in L_i do
                {access the local scores of d in the other lists through random
                access}
                overall_score(d) ← f(scores of d in each L_i)
            P_i ← P_i ∪ {positions seen under sorted or random access} ;
            bp_i ← best position in L_i
        Y ← k data items with highest score so far ;
        min_overall_score ← smallest overall score of data items in Y ;
        threshold ← f(local scores at position bp_i in each L_i) ;
        j ← j+1
end
```

Recall that over this database, TA stops at position 6. ♦

It has been proven that, for any set of sorted lists, BPA stops as early as TA, and its execution cost is never higher than TA [Akbarinia et al., 2007a]. It has also been shown that the execution cost of BPA can be $(m-1)$ times lower than that of TA. Although BPA is quite efficient, it still does redundant work. One of the redundancies with BPA (and also TA) is that it may access some data items several times under sorted access in different lists. For example, a data item that is accessed at a position in a list through sorted access and thus accessed in other lists via random access, may be accessed again in the other lists by sorted access at the next positions. An improved algorithm, BPA2 [Akbarinia et al., 2007a], avoids this and is therefore much more efficient than BPA. It does not transfer the seen positions from list owners to the query originator. Thus, the query originator does not need to maintain the seen positions and their local scores. It also accesses each position in a list at most once. The number of accesses to the lists done by BPA2 can be about $(m-1)$ times lower than that of BPA.

16.3.1.2 Top-k Queries in Unstructured Systems

One possible approach for processing top-k queries in unstructured systems is to route the query to all the peers, retrieve all available answers, score them using the scoring function, and return to the user the k highest scored answers. However, this approach is not efficient in terms of response time and communication cost.

The first efficient solution that has been proposed is that of PlanetP [Cuenca-Acuna et al., 2003], which is an unstructured P2P system. In PlanetP, a content-addressable publish/subscribe service replicates data across P2P communities of up to ten thousand peers. The top-k query processing algorithm works as follows. Given a query Q, the query originator computes a relevance ranking of peers with respect to Q, contacts them one by one in decreasing rank order and asks them to return a set of their top-scored data items together with their scores. To compute the relevance of peers, a global fully replicated index is used that contains term-to-peer mappings. This algorithm has very good performance in moderate-scale systems. However, in a large P2P system, keeping the replicated index up-to-date may hurt scalability.

We describe another solution that was developed within the context of APPA, which is a P2P network-independent data management system [Akbarinia et al., 2006a]. A fully distributed framework to execute top-k queries has been proposed that also addresses the volatility of peers during query execution, and deals with situations where some peers leave the system before finishing query processing. Given a top-k query Q with a specified TTL, the basic algorithm called Fully Decentralized Top-k (FD) proceeds as follows (see Algorithm 16.4).

1. **Query forward.** The query originator forwards Q to the accessible peers whose hop-distance from the query originator is less than TTL.
2. **Local query execution and wait.** Each peer p that receives Q executes it locally: it accesses the local data items that match the query predicate, scores them using a scoring function, selects the k top data items and saves them as well as their scores locally. Then p waits to receive its neighbors' results. However, since some of the neighbors may leave the P2P system and never send a score-list to p, the wait time has a limit that is computed for each peer based on the received TTL, network parameters and peer's local processing parameters.
3. **Merge-and-backward.** In this phase, the top scores are bubbled up to the query originator using a tree-based algorithm as follows. After its wait time has expired, p merges its k local top scores with those received from its neighbors and sends the result to its parent (the peer from which it received Q) in the form of a score-list. In order to minimize network traffic, FD does not bubble up the top data items (which could be large), only their scores and addresses. A score-list is simply a list of k pairs (a, s) where a is the address of the peer owning the data item and s its score.
4. **Data retrieval.** After receiving the score-lists from its neighbors, the query originator forms the final score-list by merging its k local top scores with the

merged score-lists received from its neighbors. Then it directly retrieves the k top data items from the peers that hold them.

Algorithm 16.4: Fully Decentralized Top-k (FD)

```
Input: Q: top-k query ;
f: scoring function;
TTL: time to live;
w: wait time
Output: Y: list of top-k data items
begin
    At query originator peer
    begin
        send Q to neighbors ;
        Final_score_list ← merge local score lists received from neighbors
        for each peer p in Final_score_list do
            Y ← retrieve top-k data items in p
    end
    for each peer that receives Q from a peer p do
        TTL ← TTL − 1 ;
        if TTL > 0 then
            send Q to neighbors
        Local_score_list ← extract top-k local scores;
        Wait a time w;
        Local_score_list ← Local_score_list ∪ top-k received scores;
        Send Local_score_list to p
end
```

The algorithm is completely distributed and does not depend on the existence of certain peers, and this makes it possible to address the volatility of peers during query execution. In particular, the following problems are addressed: peers becoming inaccessible in the merge-and-backward phase; peers that hold top data items becoming inaccessible in the data retrieval phase; late reception of score-lists by a peer after its wait time has expired. The performance evaluation of FD shows that it can achieve major performance gains in terms of communication cost and response time [Akbarinia et al., 2006b].

16.3.1.3 Top-k Queries in DHTs

As we discussed earlier, the main functionality of a DHT is to map a set of keys to the peers of the P2P system and lookup efficiently the peer that is responsible for a given key. This offers efficient and scalable support for exact-match queries.

However, supporting top-k queries on top of DHTs is not easy. A simple solution is to retrieve all tuples of the relations involved in the query, compute the score of each retrieved tuple, and finally return the k tuples whose scores are the highest. However, this solution cannot scale up to a large number of stored tuples. Another solution is to store all tuples of each relation using the same key (e.g., relation's name), so that all tuples are stored at the same peer. Then, top-k query processing can be performed at that central peer using well-known centralized algorithms. However, the peer becomes a bottleneck and a single point of failure.

A solution has been proposed as part of APPA project that is based on TA (see Section 16.3.1.1) and a mechanism that stores the shared data in the DHT in a fully distributed fashion [Akbarinia et al., 2007c]. In APPA, peers can store their tuples in the DHT using two complementary methods: tuple storage and attribute-value storage. With tuple storage, each tuple is stored in the DHT using its identifier (e.g., its primary key) as the storage key. This enables looking up a tuple by its identifier similar to a primary index. Attribute value storage individually stores in the DHT the attributes that may appear in a query's equality predicate or in a query's scoring function. Thus, as in secondary indices, it allows looking up the tuples using their attribute values. Attribute value storage has two important properties: (1) after retrieving an attribute value from the DHT, peers can retrieve easily the corresponding tuple of the attribute value; (2) attribute values that are relatively "close" are stored at the same peer. To provide the first property, the key, which is used for storing the entire tuple, is stored along with the attribute value. The second property is provided using the concept of domain partitioning as follows. Consider an attribute a and let D_a be its domain of values. Assume that there is a total order $<$ on D_a (e.g., D_a is numeric). D_a is partitioned into n non-empty sub-domains $d_1, d_2, \ldots, d_n$ such that their union is equal to D_a, the intersection of any two different sub-domains is empty, and for each $v_1 \in d_i$ and $v_2 \in d_j$, if $i < j$ then we have $v_1 < v_2$. The hash function is applied on the sub-domain of the attribute value. Thus, for the attribute values that fall in the same sub-domain, the storage key is the same and they are stored at the same peer. To avoid attribute storage skew (i.e., skewed distribution of attribute values within sub-domains), domain partitioning is done in such a way that attribute values are uniformly distributed in sub-domains. This technique uses histogram-based information that describes the distribution of values of the attribute.

Using this storage model, the top-k query processing algorithm, called DHTop (see Algorithm 16.5), works as follows. Let Q be a given top-k query, f be its scoring function, and p_0 be the peer at which Q is issued. For simplicity, let us assume that f is a monotonic scoring function. Let scoring attributes be the set of attributes that are passed to the scoring function as arguments. DHTop starts at p_0 and proceeds in two phases: first it prepares ordered lists of candidate sub-domains, and then it continuously retrieves candidate attribute values and their tuples until it finds k top tuples. The details of the two steps are as follows:

1. For each scoring attribute a, p_0 prepares the list of sub-domains and sorts them in descending order of their positive impact on the scoring function. For each list, p_0 removes from the list the sub-domains in which no member can satisfy Q's conditions. For instance, if there is a condition that enforces the scoring attribute to be equal to a constant, (e.g., $a = 10$), then p_0 removes from the list all the sub-domains except the sub-domain to which the constant value belongs. Let us denote by L_a the list prepared in this phase for a scoring attribute a.

2. For each scoring attribute a, in parallel, p_0 proceeds as follows. It sends Q and a to the peer, say p, that is responsible for storing the values of the first sub-domain of L_a, and requests it to return the values of a at p. The values are returned to p_0 in order of their positive impact on the scoring function. After receiving each attribute value, p_0 retrieves its corresponding tuple, computes its score, and keeps it if the score is one of the k highest scores yet computed. This process continues until k tuples are obtained whose scores are higher than a threshold that is computed based on the attribute values retrieved so far. If the attribute values that p returns to p_0 are not sufficient for determining the k top tuples, p_0 sends Q and a to the site that is responsible for the second sub-domain of L_a and so on until k top tuples are found.

Let $a_1, a_2, \ldots, a_m$ be the scoring attributes and $v_1, v_2, \ldots, v_m$ be the last values retrieved respectively for each of them. The threshold is defined to be $\tau = f(v_1, v_2, \ldots, v_m)$. A main feature of DHTop is that after retrieving each new attribute value, the value of the threshold decreases. Thus, after retrieving a certain number of attribute values and their tuples, the threshold becomes less than k of the retrieved data items and the algorithm stops. It has been analytically proven that DHTop works correctly for monotonic scoring functions and also for a large group of non-monotonic functions.

16.3.1.4 Top-k Queries in Super-peer Systems

A typical algorithm for top-k query processing in super-peer systems is that of Edutella [Balke et al., 2005]. In Edutella, a small percentage of nodes are super-peers and are assumed to be highly available with very good computing capacity. The super-peers are responsible for top-k query processing and the other peers only execute the queries locally and score their resources. The algorithm is quite simple and works as follows. Given a query Q, the query originator sends Q to its super-peer, which then sends it to the other super-peers. The super-peers forward Q to the relevant peers connected to them. Each peer that has some data items relevant to Q scores them and sends its maximum scored data item to its super-peer. Each super-peer chooses the overall maximum scored item from all received data items. For determining the second best item, it only asks one peer, one that has returned the first top item, to return its second top scored item. The super-peer selects the overall second top item from the previously received items and the newly received item. Then, it asks the peer which has returned the second top item and so on until all k top items are retrieved. Finally the super-peers send their top items to the super-peer of the query originator, to extract the overall k top items, and send them to the query originator.

Algorithm 16.5: DHT Top-k (DHTop)

```
Input: Q: top-k query;
f: scoring function;
A: set of m attributes used in f
Output: Y: list of top-k tuples
begin
    {Phase 1: prepare lists of attributes' subdomains}
    for each scoring attribute a in A do
        L_a ← all sub-domains of a;
        L_a ← L_a − sub-domains which do not satisfy Q's condition;
        Sort L_a in descending order of its sub-domains
    {Phase 2: continuously retrieve attribute values and their tuples until finding
    k top tuples}
    Done ← false;
    for each scoring attribute a in A in parallel do
        i ← 1
        while (i < number of sub-domains of a) and not Done do
            send Q to peer p that maintains the attribute values of sub-domain i
            in L_a;
            Z ← a values (in descending order) from p that satisfy Q's
            condition, along with their corresponding data storage keys ;
            for each received value v do
                get the tuple of v;
                Y ← k tuples with highest score so far;
                threshold ← f(v_1, v_2, ..., v_m) such that v_i is the last value
                received for attribute a_i in A;
                min_overall_score ← smallest overall score of tuples in Y;
                if min_overall_score ≤ threshold then
                    Done ← true
                i ← i + 1
end
```

This algorithm minimizes communication between peers and super-peers since, after having received the maximum scored data items from each peer connected to it, each super-peer asks only one peer for the next top item.

16.3.2 Join Queries

The most efficient join algorithms in distributed and parallel databases are hash-based. Thus, the fact that a DHT relies on hashing to store and locate data can be naturally exploited to support join queries efficiently. A basic solution has been proposed in the context of the PIER P2P system [Huebsch et al., 2003] that provides support for complex queries on top of DHTs. The solution is a variation of the parallel hash join algorithm (PHJ) (see Section 14.3.2) which we call PIERjoin. As in the PHJ algorithm, PIERjoin assumes that the joined relations and the result relations have a home (called *namespace* in PIER), which are the nodes that store horizontal fragments of the relation. Then it makes use of the `put` method for distributing tuples onto a set of peers based on their join attribute so that tuples with the same join attribute values are stored at the same peers. To perform joins locally, PIER implements a version of the symmetric hash join algorithm [Wilschut and Apers, 1991] that provides efficient support for pipelined parallelism. In symmetric hash join, with two joining relations, each node that receives tuples to be joined maintains two hash tables, one per relation. Thus, upon receiving a new tuple from either relation, the node adds the tuple into the corresponding hash table and probes it against the opposite hash table based on the tuples received so far. PIER also relies on the DHT to deal with the dynamic behavior of peers (joining or leaving the network during query execution) and thus does not give guarantees on result completeness.

For a binary join query Q (which may include select predicates), PIERjoin works in three phases (see Algorithm 16.6): multicast, hash and probe/join.

1. **Multicast phase.** The query originator peer multicasts Q to all peers that store tuples of the join relations R and S, i.e., their homes.
2. **Hash phase.** Each peer that receives Q scans its local relation, searching for the tuples that satisfy the select predicate (if any). Then, it sends the selected tuples to the home of the result relation, using `put` operations. The DHT key used in the put operation is calculated using the home of the result relation and the join attribute.
3. **Probe/join phase.** Each peer in the home of the result relation, upon receiving a new tuple, inserts it in the corresponding hash table, probes the opposite hash table to find tuples that match the join predicate (and a select predicate if any) and constructs the result joined tuples. Recall that the "home" of a (horizontally partitioned) relation was defined in Chapter 8 as a set of peers where each peer has a different partition. In this case, the partitioning is by hashing on the join attribute. The home of the result relation is also a partitioned relation (using `put` operations) so it is also at multiple peers.

This basic algorithm can be improved in several ways. For instance, if one of the relations is already hashed on the join attributes, we may use its home as result home, using a variation of the parallel associative join algorithm (PAJ) (see Section 14.3.2), where only one relation needs to be hashed and sent over the DHT.

To avoid multicasting the query to large numbers of peers, another approach is to allocate a limited number of special powerful peers, called *range guards*, for the task of join query processing [Triantafillou and Pitoura, 2003]. The domains of the join attributes are divided, and each partition is dedicated to a range guard. Then, join queries are sent only to range guards, where the query is executed.

Algorithm 16.6: PIERjoin

Input: Q: join query over relations R and S on attribute A;
h: hash function;
H_R, H_S: homes of R and S
Output: T: join result relation;
H_T: home of T
begin
  {Multicast phase}
  At query originator peer send Q to all peers in H_R and H_S ;
  {Hash phase}
  for *each peer p in H_R that received Q in parallel* **do**
    for *each tuple r in R_p that satisfies the select predicate* **do**
      place r using $h(H_T, A)$
  for *each peer p in H_S that received Q in parallel* **do**
    for *each tuple s in S_p that satisfies the select predicate* **do**
      place s using $h(H_T, A)$
  {Probe/join phase}
  for *each peer p in H_T in parallel* **do**
    if *a new tuple i has arrived* **then**
      if *i is an r tuple* **then**
        probe s tuples in S_p using $h(A)$
      else
        probe r tuples in R_p using $h(A)$
      $T_p \leftarrow r \bowtie s$
end

16.3.3 Range Queries

Recall that range queries have a WHERE clause of the form "attribute A in range $[a,b]$", with a and b being numerical values. Structured P2P systems, in particular, DHTs are very efficient at supporting exact-match queries (of the form "$A = a$") but have difficuties with range queries. The main reason is that hashing tends to destroy the ordering of data that is useful in finding ranges quickly.

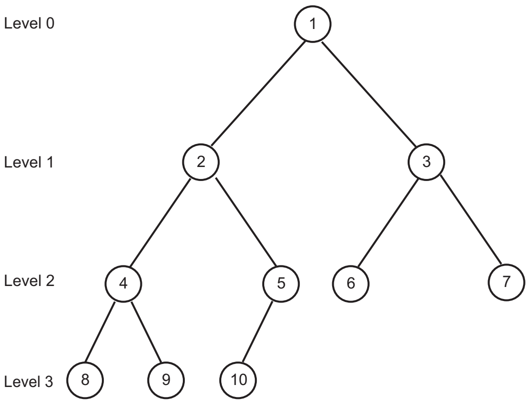
There are two main approaches for supporting range queries in structured P2P systems: extend a DHT with proximity or order-preserving properties, or maintain the key ordering with a tree-based structure. The first approach has been used in several systems. Locality sentitive hashing [Gupta et al., 2003] is an extension to DHTs that hashes similar ranges to the same DHT node with high probability. However, this method can only obtain approximate answers and may cause unbalanced loads in large networks. SkipNet [Harvey et al., 2003] is a lexicographic order-preserving DHT that allows data items with similar values to be placed on contiguous peers. It uses names rather than hashed identifiers to order peers in the overlay network, and each peer is responsible for a range of strings. This facilitates the execution of range queries. However, the number of peers to be visited is linear in the query range.

The Prefix Hash Tree (PHT) [Ramabhadran et al., 2004] is a trie-based distributed data structure that supports range queries over a DHT, by simply using the DHT lookup operation. The data being indexed are binary strings of length D. Each node has either 0 or 2 children, and a key k is stored at a leaf node whose label is a prefix of k. Furthermore, leaf nodes are linked to their neighbors. PHT's lookup operation on key k must return the unique leaf node $leaf(k)$ whose label is a prefix of k. Given a key k of length D, there are $D+1$ distinct prefixes of k. Obtaining $leaf(k)$ can be performed by a linear scan of these potential $D+1$ nodes. However, since a PHT is a binary trie, the linear scan can be improved using a binary search on prefix length. This reduces the number of DHT lookups from $(D+1)$ to $(\log D)$. Given two keys a and b such as $a \leq b$, two algorithms for range queries are supported, using PHT's lookup. The first one is sequential: it searches $leaf(a)$ and then scans sequentially the linked list of leaf nodes until the node $leaf(b)$ is reached. The second algorithm is parallel: it first identifies the node which corresponds to the smallest prefix range that completely covers the range $[a,b]$. To reach this node, a simple DHT lookup is used and the query is forwarded recursively to those children that overlap with the range $[a,b]$.

As in all hashing schemes, the first approach suffers from data skew that can result in peers with unbalanced ranges, which hurts load balancing. To overcome this problem, the second approach exploits tree-based structures to maintain balanced ranges of keys. The first attempt to build a P2P network based on a balanced tree structure is BATON (BAlanced Tree Overlay Network) [Jagadish et al., 2005]. We now present BATON and its support for range queries in more detail.

BATON organizes peers as a balanced binary tree (each node of the tree is maintained by a peer). The position of a node in BATON is determined by a (level,number) tuple, with level starting from 0 at the root, number starting from 1 at the root and sequentially assigned using in-order traversal. Each tree node stores links to its parent, children, adjacent nodes and selected neighbor nodes that are nodes at the same level. Two routing tables: a *left routing table* and a *right routing table* store links to the selected neighbor nodes. For a node numbered i, these routing tables contain links to nodes located at the same level with numbers that are less (left routing table) and greater (right routing table) than i by a power of 2. The j^{th} element in the left (right) routing table at node i contains a link to the node numbered $i-2^{j-1}$ (respectively $i+2^{j-1}$) at the same level in the tree. Figure 16.11 shows the routing table of node 6.

In BATON, each leaf and internal node (or peer) is assigned a range of values. For each link this range is stored at the routing table and when its range changes, the link is modified to record the change. The range of values managed by a peer is required to be to the right of the range managed by its left subtree and less than the range managed by its right subtree (see Figure 16.12). Thus, BATON builds an effective distributed index structure. The joining and departure of peers are processed such that the tree remains balanced by forwarding the request upward in the tree for joins



Node 6: level 2, number=3
parent=3, leftchild=null, rightchild=null
leftadjacent=1, rightadjacent=3

Left routing table

	Node	Left Child	Right Child	Lower Bound	Upper Bound
0	5	10	null	LB5	UB5
1	4	8	9	LB4	UB4

Right routing table

	Node	Left Child	Right Child	Lower Bound	Upper Bound
0	7	null	null	LB7	UB7

Fig. 16.11 BATON structure-tree index and routing table of node 6

and downward in the tree for leaves, thus with no more than $O(\log\ n)$ steps for a tree of n nodes.

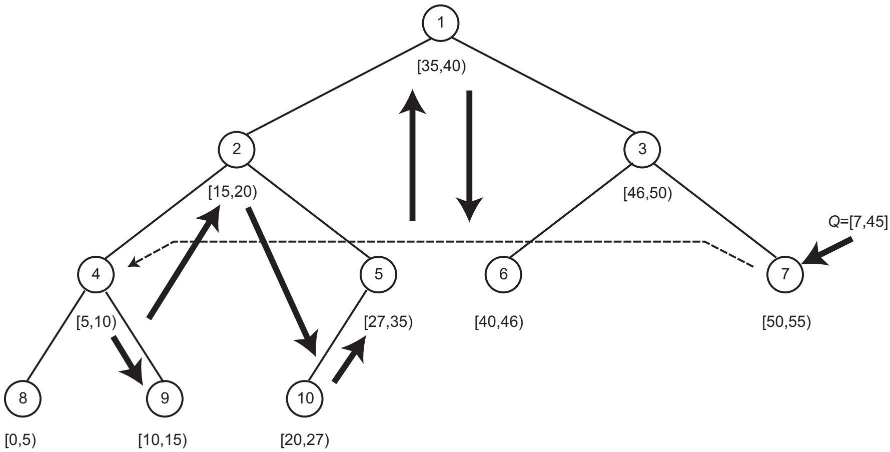


Fig. 16.12 Range query processing in BATON

A range query is processed as follows (Algorithm 16.7). For a range query Q with range $[a,b]$ submitted by node i, it looks for a node that intersects with the lower bound of the searched range. The peer that stores the lower bound of the range checks locally for tuples belonging to the range and forwards the query to its right adjacent node. In general, each node receiving the query checks for local tuples and contacts its right adjacent node until the node containing the upper bound of the range is reached. Partial answers obtained when an intersection is found are sent to the node that submits the query. The first intersection is found in $O(\log\ n)$ steps using an algorithm for exact match queries. Therefore, a range query with X nodes covering the range is answered in $O(\log n + X)$ steps.

Algorithm 16.7: BatonRange

```
Input: Q: a range query in the form [a,b]
Output: T: result relation
begin
    {Search for the peer storing the lower bound of the range}
    At query originator peer
    begin
        find peer p that holds value a ;
        send Q to p;
    end
    for each peer p that receives Q do
        T_p ← Range(p) ∩ [a,b];
        send T_p to query originator ;
        if Range(RightAdjacent(p)) ∩ [a,b] ≠ ∅ then
            let p be right adjacent peer of p ;
            send Q to p
end
```

Example 16.6. Consider the query Q with range $[7,45]$ issued at node 7 in Figure 16.12. First, BATON executes an exact match query looking for a node containing the lower bound of the range (see dashed line in the figure). Since the lower bound is in the range assigned to node 4, it checks locally for tuples belonging to the range and forwards the query to its adjacent right node (node 9). Node 9 checks for local tuples belonging to the range and forwards the query to node 2. Nodes 10, 5, 1 and 6 receive the query, they check for local tuples and contact their respective right adjacent node until the node containing the upper bound of the range is reached. ♦

16.4 Replica Consistency

To increase data availability and access performance, P2P systems replicate data. However, different P2P systems provide very different levels of replica consistency. The earlier, simple P2P systems such as Gnutella and Kazaa deal only with static data (e.g., music files) and replication is "passive" as it occurs naturally as peers request and copy files from one another (basically, caching data). In more advanced P2P systems where replicas can be updated, there is a need for proper replica management techniques. Unfortunately, most of the work on replica consistency has been done only in the context of DHTs. We can distinguish three approaches to deal with replica consistency: basic support in DHTs, data currency in DHTs, and replica reconciliation. In this section, we introduce the main techniques used in these approaches.

16.4.1 Basic Support in DHTs

To improve data availability, most DHTs rely on data replication by storing $(key, data)$ pairs at several peers by, for example, using several hash functions. If one peer is unavailable, its data can still be retrieved from the other peers that hold a replica. Some DHTs provide basic support for the application to deal with replica consistency. In this section, we describe the techniques used in two popular DHTs: CAN and Tapestry.

CAN provides two approaches for supporting replication [Ratnasamy et al., 2001a]. The first one is to use m hash functions to map a single key onto m points in the coordinate space, and, accordingly, replicate a single $(key, data)$ pair at m distinct nodes in the network. The second approach is an optimization over the basic design of CAN that consists of a node proactively pushing out popular keys towards its neighbors when it finds it is being overloaded by requests for these keys. In this approach, replicated keys should have an associated TTL field to automatically undo the effect of replication at the end of the overloaded period. In addition, the technique assumes immutable (read-only) data.

Tapestry [Zhao et al., 2004] is an extensible P2P system that provides decentralized object location and routing on top of a structured overlay network. It routes messages to logical end-points (i.e., endpoints whose identifiers are not associated with physical location), such as nodes or object replicas. This enables message delivery to mobile or replicated endpoints in the presence of instability of the underlying infrastructure. In addition, Tapestry takes latency into account to establish each node's neighborhood. The location and routing mechanisms of Tapestry work as follows. Let o be an object identified by $id(o)$; the insertion of o in the P2P network involves two nodes: the server node (noted n_s) that holds o and the root node (noted n_r) that holds a mapping in the form $(id(o), n_s)$ indicating that the object identified by $id(o)$ is stored at node n_s. The root node is dynamically determined by a globally consistent deterministic algorithm. Figure 16.13a shows that when o is inserted into n_s, n_s publishes $id(o)$ at its root node by routing a message from n_s to n_r containing the mapping $(id(o), n_s)$. This mapping is stored at all nodes along the message path. During a location query (e.g., "$id(o)$?" in Figure 16.13a, the message that looks for $id(o)$ is initially routed towards n_r, but it may be stopped before reaching it once a node containing the mapping $(id(o), n_s)$ is found. For routing a message to $id(o)$'s root, each node forwards this message to its neighbor whose logical identifier is the most similar to $id(o)$ [Plaxton et al., 1997].

Tapestry offers the entire infrastructure needed to take advantage of replicas, as shown in Figure 16.13b. Each node in the graph represents a peer in the P2P network and contains the peer's logical identifier in hexadecimal format. In this example, two replicas O_1 and O_2 of object O (e.g., a book file) are inserted into distinct peers

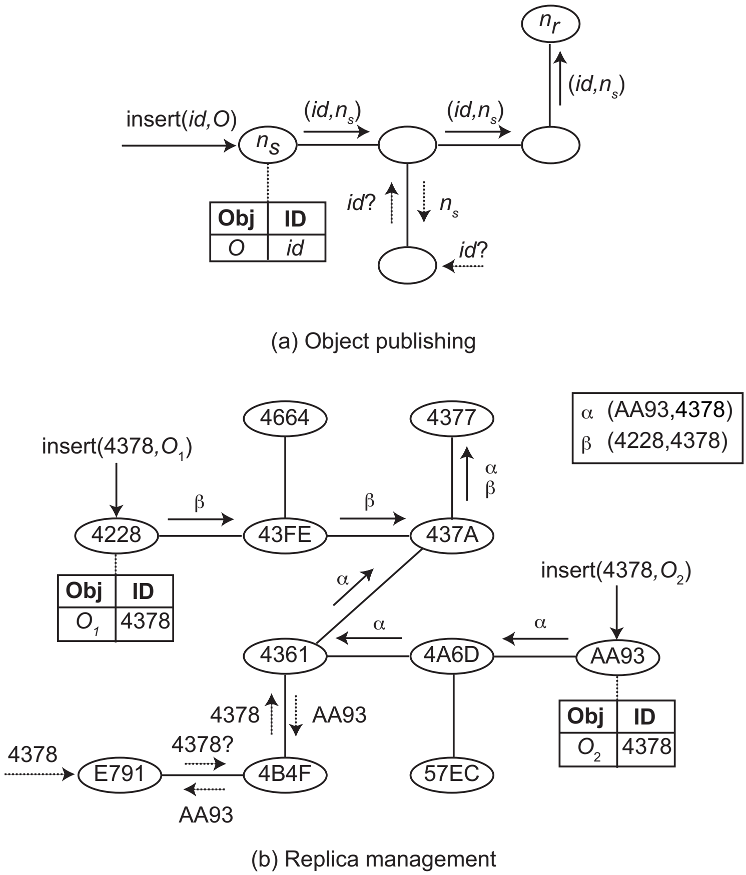


Fig. 16.13 Tapestry (a) Object publishing (b) Replica management.

($O_1 \rightarrow$ peer 4228 and $O_2 \rightarrow$ peer $AA93$). The identifier of O_1 is equal to that of O_2 (i.e., 4378 in hexadecimal) as O_1 and O_2 are replicas of the same object O. When O_1 is inserted into its server node (peer 4228), the mapping $(4378, 4228)$ is routed from peer 4228 to peer 4377 (the root node for O_1's identifier). As the message approaches the root node, the object and the node identifiers become increasingly similar. In addition, the mapping $(4378, 4228)$ is stored at all peers along the message path. The insertion of O_2 follows the same procedure. In Figure 16.13b, if peer E791 looks for a replica of O, the associated message routing stops at peer 4361. Therefore, applications can replicate data across multiple server nodes and rely on Tapestry to direct requests to nearby replicas.

16.4.2 Data Currency in DHTs

Although DHTs provide basic support for replication, the mutual consistency of the replicas after updates can be compromised as a result of peers leaving the network or concurrent updates. Let us illustrate the problem with a simple update scenario in a typical DHT.

Example 16.7. Let us assume that the operation `put` (k, d_0) (issued by some peer) maps onto peers p_1 and p_2 both of which get to store data d_0. Now consider an update (from the same or another peer) with the operation `put` (k, d_1) that also maps onto peers p_1 and p_2. Assuming that p_2 cannot be reached (e.g., because it has left the network), only p_1 gets updated to store d_1. When p_2 rejoins the network later on, the replicas are not consistent: p_1 holds the current state of the data associated with k while p_2 holds a stale state.

Concurrent updates also cause problems. Consider now two updates `put` (k, d_2) and `put` (k, d_3) (issued by two different peers) that are sent to p_1 and p_2 in reverse order, so that p_1's last state is d_2 while p_2's last state is d_3. Thus, a subsequent `get` (k) operation will return either stale or current data depending on which peer is looked up, and there is no way to tell whether it is current or not. ♦

For some applications (e.g., agenda management, bulletin boards, cooperative auction management, reservation management, etc.) that could take advantage of a DHT, the ability to get the current data are very important. Supporting data currency in replicated DHTs requires the ability to return a current replica despite peers leaving the network or concurrent updates. Of course, replica consistency is a more general problem, as discussed in Chapter 13, but the issue is particularly difficult and important in P2P systems, since there is considerable dynamism in the peers joining and leaving the system. The problem can be partially addressed by using data versioning [Knezevic et al., 2005]. Each replica has a version number that is increased after each update. To return a current replica, all replicas need to be retrieved in order to select the latest version. However, because of concurrent updates, it may happen that two different replicas have the same version number, thus making it impossible to decide which one is the current replica.

A more complete solution has been proposed that considers both data availability and data currency [Akbarinia et al., 2007b]. To provide high data availability, data are replicated in the DHT using a set of independent hash functions H_r, called *replication hash functions*. The peer that is responsible for key k with respect to hash function h at the current time is denoted by $rsp(k, h)$. To be able to retrieve a current replica, each pair $(k, data)$ is stamped with a logical timestamp, and for each $h \in H_r$, the pair $(k, newData)$ is replicated at $rsp(k, h)$ where $newData = \{data, timestamp\}$, i.e., newdata is composed of the initial data and the timestamp. Upon a request for the data associated with a key, we can return one of the replicas that are stamped with the latest timestamp. The number of replication hash functions, i.e., H_r, can be different for different DHTs. For instance, if in a DHT the availability of peers is low, a high value of H_r (e.g., 30) can be used to increase data availability.

This solution is the basis for a service called *Update Management Service* (UMS) that deals with efficient insertion and retrieval of current replicas based on timestamping. Experimental validation has shown that UMS incurs very little overhead in terms of communication cost. After retrieving a replica, UMS detects whether it is current or not, i.e., without having to compare with the other replicas, and returns it as output. Thus, UMS does not need to retrieve all replicas to find a current one; it only requires the DHT's lookup service with `put` and `get` operations.

To generate timestamps, UMS uses a distributed service called *Key-based Timestamping Service* (KTS). The main operation of KTS is `gen_ts(`k`)`, which, given a key k, generates a real number as a timestamp for k. The timestamps generated by KTS are *monotonic* such that if ts_i and ts_j are two timestamps generated for the same key at times t_i and t_j, respectively, $ts_j > ts_i$ if t_j is later than t_i. This property allows ordering the timestamps generated for the same key according to the time at which they have been generated. KTS has another operation denoted by *last_ts(k)*, which, given a key k, returns the last timestamp generated for k by KTS. At anytime, `gen_ts(`k`)` generates at most one timestamp for k, and different timestamps for k are monotonic. Thus, in the case of concurrent calls to insert a pair $(k, data)$, i.e., from different peers, only the one that obtains the latest timestamp will succeed to store its data in the DHT.

16.4.3 Replica Reconciliation

Replica reconciliation goes one step further than data currency by enforcing mutual consistency of replicas. Since a P2P network is typically very dynamic, with peers joining or leaving the network at will, eager replication solutions (see Chapter 13) are not appropriate; lazy replication is preferred. In this section, we describe the reconciliation techniques used in OceanStore, P-Grid and APPA to provide a spectrum of proposed solutions.

16.4.3.1 OceanStore

OceanStore [Kubiatowicz et al., 2000] is a data management system designed to provide continuous access to persistent information. It relies on Tapestry and assumes an infrastructure composed of untrusted powerful servers that are connected by high-speed links. For security reasons, data are protected through redundancy and cryptographic techniques. To improve performance, data are allowed to be cached anywhere, anytime.

OceanStore allows concurrent updates on replicated objects; it relies on reconciliation to assure data consistency. Figure 16.14 illustrates update management in OceanStore. In this example, R is a replicated object whereas R_i and r_i denote, respectively, a primary and a secondary copy of R. Nodes n_1 and n_2 are concurrently updating R. Such updates are managed as follows. Nodes that hold primary copies of

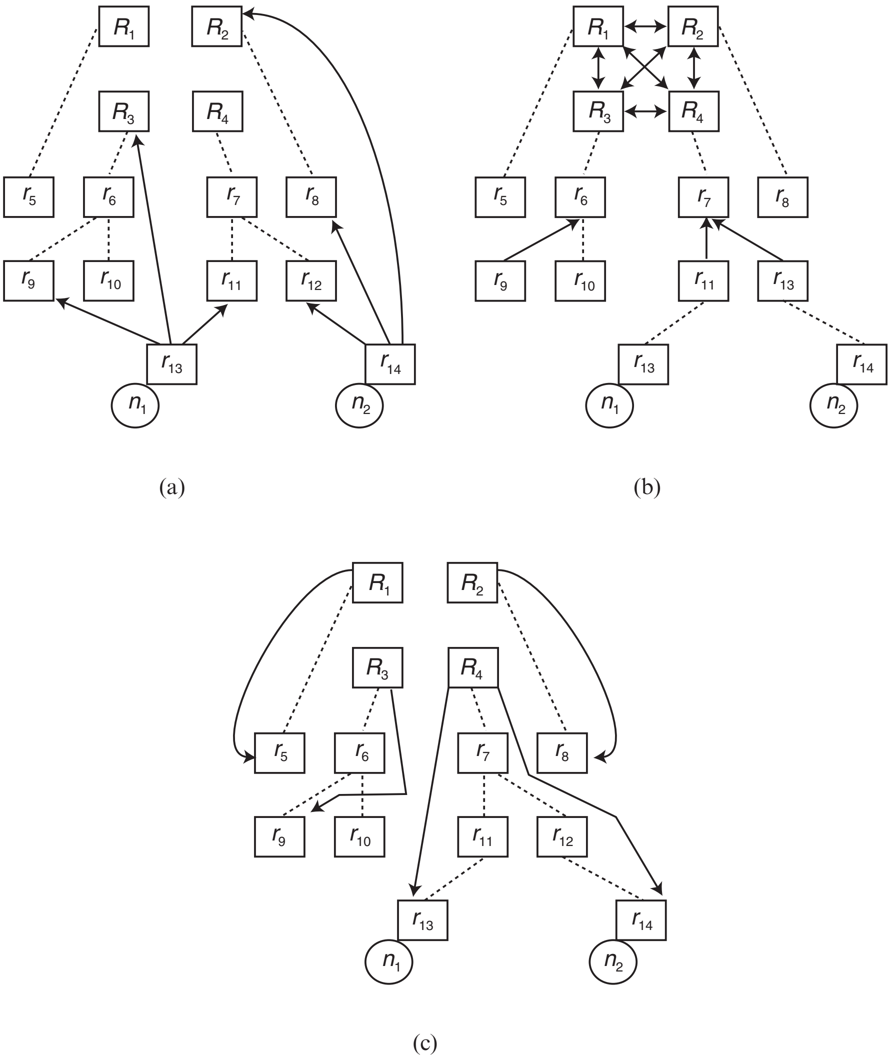


Fig. 16.14 OceanStore reconciliation. (a) Nodes n_1 and n_2 send updates to the master group of R and to several random secondary replicas. (b) The master group of R orders updates while secondary replicas propagate them epidemically. (c) After the master group agreement, the result of updates is multicast to secondary replicas.

R, called the *master group of* R, are responsible for ordering updates. So, n_1 and n_2 perform tentative updates on their local secondary replicas and send these updates to the master group of R as well as to other random secondary replicas (see Figure 16.14a). The tentative updates are ordered by the master group based on timestamps assigned by n_1 and n_2; at the same time, these updates are epidemically propagated among secondary replicas (Figure 16.14b). Once the master group obtains an agreement, the result of updates is multicast to secondary replicas (Figure 16.14c), which contain both tentative[2] and committed data.

Replica management adjusts the number and location of replicas in order to service requests more efficiently. By monitoring the system load, OceanStore detects when a replica is overwhelmed and creates additional replicas on nearby nodes to alleviate load. Conversely, these additional replicas are eliminated when they are no longer needed.

16.4.3.2 P-Grid

P-Grid [Aberer et al., 2003a] is a structured P2P network based on a binary trie structure. A decentralized and self-organizing process builds P-Grid's routing infrastructure which is adapted to a given distribution of data keys stored by peers. This process addresses uniform load distribution of data storage and uniform replication of data to support availability.

To address updates of replicated objects, P-Grid employs gossiping, without strong consistency guarantees. P-Grid assumes that quasi-consistency of replicas (instead of full consistency which is too hard to provide in a dynamic environment) is enough.

The update propagation scheme has a push phase and a pull phase. When a peer p receives a new update to a replicated object R, it pushes the update to a subset of peers that hold replicas of R, which, in turn, propagate it to other peers holding replicas of R, and so on. Peers that have been disconnected and get connected again, peers that do not receive updates for a long time, or peers that receive a pull request but are not sure whether they have the latest update, enter the pull phase to reconcile. In this phase, multiple peers are contacted and the most up-to-date among them is chosen to provide the object content.

16.4.3.3 APPA

APPA provides a general lazy distributed replication solution that assures eventual consistency of replicas [Martins et al., 2006a; Martins and Pacitti, 2006; Martins et al., 2008]. It uses the action-constraint framework [Kermarrec et al., 2001] to capture the application semantics and resolve update conflicts.

The application semantics is described by means of constraints between update actions. An *action* is defined by the application programmer and represents an application-specific operation (e.g., a write operation on a file or document, or a database transaction). A *constraint* is the formal representation of an application invariant. For instance, the *predSucc*(a_1, a_2) constraint establishes causal ordering between actions (i.e., action a_2 executes only after a_1 has succeeded); the *mutuallyExclusive*(a_1, a_2) constraint states that either a_1 or a_2 can be executed. The aim of reconciliation is to take a set of actions with the associated constraints and produce

[2] Tentative data are data that the primary replicas have not yet committed.

a *schedule*, i.e., a list of ordered actions that do not violate constraints. In order to reduce the schedule production complexity, the set of actions to be ordered is divided into subsets called *clusters*. A cluster is a subset of actions related by constraints that can be ordered independently of other clusters. Therefore, the *global schedule* is composed by the concatenation of clusters' ordered actions.

Data managed by the APPA reconciliation algorithm are stored in data structures called *reconciliation objects*. Each reconciliation object has a unique identifier in order to enable its storage and retrieval in the DHT. Data replication proceeds as follows. First, nodes execute local actions to update a replica of an object while respecting user-defined constraints. Then, these actions (with the associated constraints) are stored in the DHT based on the object's identifier. Finally, reconciler nodes retrieve actions and constraints from the DHT and produce the global schedule, by reconciling conflicting actions based on the application semantics. This schedule is locally executed at every node, thereby assuring eventual consistency.

Any connected node can try to start reconciliation by inviting other available nodes to engage with it. Only one reconciliation can run at-a-time. The reconciliation of update actions is performed in 6 distributed steps as follows. Nodes at step 2 start reconciliation. The outputs produced at each step become the input to the next one.

- **Step 1 - node allocation:** a subset of connected replica nodes is selected to proceed as reconcilers based on communication costs.
- **Step 2 - action grouping:** reconcilers take actions from the action logs and put actions that try to update common objects into the same group since these actions are potentially in conflict. Groups of actions that try to update object R are stored in the *action log R* reconciliation object (L_R).
- **Step 3 - cluster creation:** reconcilers take action groups from the action logs and split them into clusters of semantically dependent conflicting actions (two actions a_1 and a_2 are semantically independent if the application judges it safe to execute them together, in any order, even if they update a common object; otherwise, a_1 and a_2 are semantically dependent. Clusters produced in this step are stored in the cluster set reconciliation object.
- **Step 4 - clusters extension:** user-defined constraints are not taken into account in cluster creation. Thus, in this step, reconcilers extend clusters by adding to them new conflicting actions, according to user-defined constraints.
- **Step 5 - cluster integration:** cluster extensions lead to cluster overlapping (an overlap occurs when the intersection of two clusters results in a non-null set of actions). In this step, reconcilers bring together overlapping clusters. At this point, clusters become mutually-independent, i.e., there are no constraints involving actions of distinct clusters.
- **Step 6 - cluster ordering:** in this step, reconcilers take each cluster from the cluster set and order the cluster's actions. The ordered actions associated with each cluster are stored in the *schedule* reconciliation object. The concatenation of all clusters' ordered actions makes up the global schedule that is executed by all replica nodes.

At every step, the reconciliation algorithm takes advantage of data parallelism, i.e., several nodes per-form simultaneously independent activities on a distinct subset of actions (e.g., ordering of different clusters).

16.5 Conclusion

By distributing data storage and processing across autonomous peers in the network, "modern" P2P systems can scale without the need for powerful servers. Advanced P2P applications such as scientific cooperation must deal with semantically rich data (e.g., XML documents, relational tables, etc.). Supporting such applications requires significant revisiting of distributed database techniques (schema management, access control, query processing, transaction management, consistency management, reliability and replication). When considering data management, the main requirements of a P2P system are autonomy, query expressiveness, efficiency, quality of service, and fault-tolerance. Depending on the P2P network architecture (unstructured, structured DHT, or hybrid super-peer), these requirements can be achieved to varying degrees. Unstructured networks have better fault-tolerance but can be quite inefficient because they rely on flooding for query routing. Hybrid systems have better potential to satisfy high-level data management requirements. However, DHT systems are best for key-based search and could be combined with super-peer networks for more complex searching.

Most of the work on sharing semantically rich data in P2P systems has focused on schema management and query processing. However, there has been very little work on update management, replication, transactions and access control. Much more work is needed to revisit distributed database techniques for large-scale P2P systems. The main issues that have to be dealt with include schema management, complex query processing, transaction support and replication, and privacy. Furthermore, it is unlikely that all kinds of data management applications are suited for P2P systems. Typical applications that can take advantage of P2P systems are probably light-weight and involve some sort of cooperation. Characterizing carefully these applications is important and will be useful to produce performance benchmarks.

16.6 Bibliographic Notes

Data management in "modern" P2P systems, those characterized by massive distribution, inherent heterogeneity, and high volatility, has become an important research topic. The topic is fully covered in a recent book [Vu et al., 2009]. A shorter survey can be found in [Ulusoy, 2007]. Discussions on the requirements, architectures, and issues faced by P2P data management systems are provided in [Bernstein et al., 2002; Daswani et al., 2003; Valduriez and Pacitti, 2004]. A number of P2P data management systems are presented in [Aberer, 2003].

An extensive survey of query processing in P2P systems is provided in [Akbarinia et al., 2007d] and has been the basis for writing Sections 16.2 and 16.3. A good discussion of the issues of schema mapping in P2P systems can be found in [Tatarinov et al., 2003]. An important kind of query in P2P systems is top-k queries. A survey of top-k query processing techniques in relational database systems is provided in [Ilyas et al., 2008]. An efficient algorithm for top-k query processing is the Threshold Algorithm (TA) which was proposed independently by several researchers [Nepal and Ramakrishna, 1999; Güntzer et al., 2000; Fagin et al., 2003]. TA has been the basis for several algorithms in P2P systems, in particular in DHTs [Akbarinia et al., 2007c]. A more efficient algorithm than TA is the Best Position Algorithm [Akbarinia et al., 2007a]. A survey of ranking algorithms in databases (not necessarily in P2P systems) is given in [Ilyas et al., 2008].

The survey of replication in P2P systems by Martins et al. [2006b] has been the basis for Section 16.4. A complete solution to data currency in replicated DHTs, i.e., providing the ability to find the most current replica, is given in [Akbarinia et al., 2007b]. Reconciliation of replicated data are addressed in OceanStore [Kubiatowicz et al., 2000], P-Grid [Aberer et al., 2003a] and APPA [Martins et al., 2006a; Martins and Pacitti, 2006].

P2P techniques have recently received attention to help scaling up data management in the context of Grid Computing. This triggered open problems and new issues which are discussed in [Pacitti et al., 2007a].

Exercises

Problem 16.1. What is the fundamental difference between P2P and client-server architectures? Is a P2P system with a centralized index equivalent to a client-server system? List the main advantages and drawbacks of P2P file sharing systems from different points of view:

- end-users;
- file owners;
- network administrators.

Problem 16.2 (**). A P2P overlay network is built as a layer on top of a physical network, typically the Internet. Thus, they have different topologies and two nodes that are neighbors in the P2P network may be far apart in the physical network. What are the advantages and drawbacks of this layering? What is the impact of this layering on the design of the three main types of P2P networks (unstructured, structured and superpeer)?

Problem 16.3 (*). Consider the unstructured P2P network in Figure 16.4 and the bottom-left peer that sends a request for resource. Illustrate and discuss the two following search strategies in terms of result completeness:

- flooding with TTL=3;
- gossiping with each peer has a partial view of at most 3 neighbours.

Problem 16.4 (*). Consider Figure 16.7, focusing on structured networks. Refine the comparison using the scale 1-5 (instead of low - moderate - high) by considering the three main types of DHTs: tree, hypercube and ring.

Problem 16.5 (**). The objective is to design a P2P social network application, on top of a DHT. The application should provide basic functions of social networks: register a new user with her profile; invite or retrieve friends; create lists of friends; post a message to friends; read friends' messages; post a comment on a message. Assume a generic DHT with put and get operations, where each user is a peer in the DHT.

Problem 16.6 (**). Propose a P2P architecture of the social network application, with the (key, data) pairs for the different entities which need be distributed. Describe how the following operations: create or remove a user; create or remove a friendship; read messages from a list of friends. Discuss the advantages and drawbacks of the design.

Problem 16.7 (**). Same question, but with the additional requirement that private data (e.g., user profile) must be stored at the user peer.

Problem 16.8. Discuss the commonalities and differences of schema mapping in multidatabase systems and P2P systems. In particular, compare the local-as-view approach presented in Chapter 4 with the pairwise schema mapping approach in Section 16.2.1.

Problem 16.9 (*). The FD algorithm for top-k query processing in unstructured P2P networks (see Algorithm 16.4) relies on flooding. Propose a variation of FD where, instead of flooding, random walk or gossiping is used. What are the advantages and drawbacks?

Problem 16.10 (*). Apply the TPUT algorithm (Algorithm 16.2) to the three lists of the database in Figure 16.10 witk k=3. For each step of the algorithm, show the intermediate results.

Problem 16.11 (*). Same question applied to Algorithm DHTop (see Algorithm 16.5.

Problem 16.12 (*). Algorithm 16.6 assumes that the input relations to be joined are placed arbitrarily in the DHT. Assuming that one of the relations is already hashed on the join attributes, propose an improvement of Algorithm 16.6.

Problem 16.13 (*). To improve data availability in DHTs, a common solution is to replicate $(k, data)$ pairs at several peers using several hash functions. This produces the problem illustrated in Example 16.7. An alternative solution is to use a non-replicated DHT (with a single hash function) and have the nodes replicating (k, data) pairs at some of their neighbors. What is the effect on the scenario in Example 16.7? What are the advantages and drawbacks of this approach, in terms of availability and load balancing?