database statistics and query processing time estimates. In practice, these estimates are hard to obtain with black-box DBMSs.

Adaptive virtual partitioning (AVP) solves this problem by dynamically tuning partition sizes, thus without requiring these estimates. AVP runs independently at each participating cluster node, avoiding internode communication (for partition size determination). Initially, each node receives an interval of values to work with. These intervals are determined exactly as for SVP. Then, each node performs the following steps:

1. Start with a very small partition size beginning with the first value of the received interval.
2. Execute a subquery with this interval.
3. Increase the partition size and execute the corresponding subquery while the increase in execution time is proportionally smaller than the increase in partition size.
4. Stop increasing. A stable size has been found.
5. If there is performance degradation, i.e., there were consecutive worse executions, decrease size and go to Step 2.

Starting with a very small partition size avoids full table scans at the very beginning of the process. This also avoids having to know the threshold after which the DBMS does not use clustered indices and starts performing full table scans. When partition size increases, query execution time is monitored allowing determination of the point after which the query processing steps that are data size independent do not influence too much total query execution time. For example, if doubling the partition size yields an execution time that is twice the previous one, this means that such a point has been found. Thus the algorithm stops increasing the size. System performance can deteriorate due to DBMS data cache misses or overall system load increase. It may happen that the size being used is too large and has benefited from previous data cache hits. In this case, it may be better to shrink partition size. That is precisely what step 5 does. It gives a chance to go back and inspect smaller partition sizes. On the other hand, if performance deterioration was due to a casual and temporary increase of system load or data cache misses, keeping a small partition size can lead to poor performance. To avoid such a situation, the algorithm goes back to Step 2 and restarts increasing sizes.

AVP and other variants of virtual partitioning have several advantages: flexibility for node allocation, high availability because of full replication, and opportunities for dynamic load balancing. But full replication can lead to high cost in disk usage. To support partial replication, hybrid solutions have been proposed to combine physical and virtual partitioning. The hybrid design uses physical partitioning for the largest and most important relations and fully replicates the small tables. Thus, intraquery parallelism can be achieved with lesser disk space requirements. The hybrid solution combines AVP with physical partitioning. It solves the problem of disk usage while keeping the advantages of AVP, i.e., full table scan avoidance and dynamic load balancing.

8.8 Conclusion

Parallel database systems have been exploiting multiprocessor architectures to provide high-performance, high-availability, extensibility, and scalability with a good cost/performance ratio. Furthermore, parallelism is the only viable solution for supporting very large databases and applications within a single system.

Parallel database system architectures can be classified as shared-memory, shared-disk, and shared-nothing. Each architecture has its advantages and limitations. Shared-memory is used in tightly coupled NUMA multiprocessors or multicore processors, and can provide the highest performance because of fast memory access and great load balancing. However, it has limited extensibility and scalability. Shared-disk and shared-nothing are used in computer clusters, typically using multicore processors. With low latency networks (e.g., Infiniband and Myrinet), they can provide high performance and scale up to very large configurations (with thousands of nodes). Furthermore, the RDMA capability of those networks can be exploited to make cost-effective NUMA clusters. Shared-disk is typically used for OLTP workloads as it is simpler and has good load balancing. However, shared-nothing remains the only choice for highly scalable systems, as need in OLAP or big data, with the best cost/performance ratio.

Parallel data management techniques extend distributed database techniques. However, the critical issues for such architectures are data partitioning, replication, parallel query processing, load balancing, and fault-tolerance. The solutions to these issues are more involved than in distributed DBMS because they must scale to high numbers of nodes. Furthermore, recent advances in hardware/software such as low latency interconnect, multicore processor nodes, large main memory, and RDMA provide new opportunities for optimization. In particular, parallel algorithms for the most demanding operators such as join and sort need be made NUMA-aware.

A database cluster is an important kind of parallel database system that uses a black-box DBMS at each node. Much research has been devoted to take full advantage of the cluster stable environment in order to improve performance and availability by exploiting data replication. The main results of this research are new techniques for replication, load balancing, and query processing.

8.9 Bibliographic Notes

The earlier proposal of a database machine dates back to [Canaday et al. 1974], mainly to address the "I/O bottleneck" [Boral and DeWitt 1983], induced by high disk access time with respect to main memory access time. The main idea was to push database functions closer to disk. CAFS-ISP is an early example of hardware-based filtering device [Babb 1979] that was bundled within disk controllers for fast associative search. The introduction of general-purpose microprocessors in disk controllers also led to intelligent disks [Keeton et al. 1998].

The first parallel database system products were Teradata and Tandem NonStopSQL in the early 1980s. Since then, all major DBMS players have delivered a parallel version of their product. Today, the field is still the subject of intensive research to deal with big data and exploit new hardware capabilities, e.g., low latency interconnects, multicore processor nodes, and large main memories.

Comprehensive surveys of parallel database systems are provided in [DeWitt and Gray 1992, Valduriez 1993, Graefe 1993]. Parallel database system architectures are discussed in [Bergsten et al. 1993, Stonebraker 1986, Pirahesh et al. 1990], and compared using a simple simulation model in [Breitbart and Silberschatz 1988]. The first NUMA architectures are described in [Lenoski et al. 1992, Goodman and Woest 1988]. A more recent approach based on Remote Direct Memory Access (RDMA) is discussed in [Novakovic et al. 2014, Leis et al. 2014, Barthels et al. 2015].

Examples of parallel database system prototypes are Bubba [Boral et al. 1990], DBS3 [Bergsten et al. 1991], Gamma [DeWitt et al. 1986], Grace [Fushimi et al. 1986], Prisma/DB [Apers et al. 1992], Volcano [Graefe 1990], and XPRS [Hong 1992].

Data placement, including replication, in a parallel database system is treated in [Livny et al. 1987, Copeland et al. 1988, Hsiao and DeWitt 1991]. A scalable solution is Gamma's *chained partitioning* [Hsiao and DeWitt 1991], which stores the primary and backup copy on two adjacent nodes. Associative access to a partitioned relation using a global index is proposed in [Khoshafian and Valduriez 1987].

Parallel query optimization is treated in [Shekita et al. 1993], [Ziane et al. 1993], and [Lanzelotte et al. 1994]. Our discussion of cost model in Sect. 8.4.2.2 is based on [Lanzelotte et al. 1994]. Randomized search strategies are proposed in [Swami 1989, Ioannidis and Wong 1987]. XPRS uses a two phase optimization strategy [Hong and Stonebraker 1993]. The exchange operator, which is the basis for parallel repartitioning in parallel query processing, was proposed in the context of the Volcano query evaluation system [Graefe 1990].

There is an extensive literature on parallel algorithms for database operators, in particular sort and join. The objective of these algorithms is to maximize the degree of parallelism, following Amdahl's law [Amdahl 1967] that states that only part of an algorithm can be parallelized. The seminal paper by [Bitton et al. 1983] proposes and compares parallel versions of merge sort, nested loop join, and sort-merge join algorithms. Valduriez and Gardarin [1984] propose the use of hashing for parallel join and semijoin algorithms. A survey of parallel sort algorithms can be found in [Bitton et al. 1984]. The specification of two main phases, *build* and *probe*, [DeWitt and Gerber 1985] has been useful to understand parallel hash join algorithms. The Grace hash join [Kitsuregawa et al. 1983], the hybrid hash join algorithm [DeWitt et al. 1984, Shatdal et al. 1994], and the radix hash join [Manegold et al. 2002] have been the basis for many variations in particular to exploit multicore processors and NUMA [Barthels et al. 2015]. Other important join algorithms are the symmetric hash join [Wilschut and Apers 1991] and the Ripple join [Haas and Hellerstein 1999b]. In [Barthels et al. 2015], the authors show that a radix hash join can perform very well in large-scale shared-nothing clusters using RDMA.

The parallel sort-merge join algorithm is gaining renewed interest in the context of multicore and NUMA systems [Albutiu et al. 2012, Pasetto and Akhriev 2011].

Load balancing in parallel database systems has been extensively studied both in the context of shared-memory and shared-disk [Lu et al. 1991, Shekita et al. 1993] and shared-nothing [Kitsuregawa and Ogawa 1990, Walton et al. 1991, DeWitt et al. 1992, Shatdal and Naughton 1993, Rahm and Marek 1995, Mehta and DeWitt 1995, Garofalakis and Ioannidis 1996]. The presentation of the Dynamic Processing execution model in Sect. 8.5 is based on [Bouganim et al. 1996, 1999]. The rate match algorithm is described in [Mehta and DeWitt 1995].

The effects of skewed data distribution on a parallel execution are introduced in [Walton et al. 1991]. A general adaptive approach to dynamically adjust the degree of parallelism using control operators is proposed in [Biscondi et al. 1996]. A good approach to deal with data skew is to use multiple join algorithms, each specialized for a different degree of skew, and to determine, at execution time, which algorithm is best [DeWitt et al. 1992].

The content of Sect. 8.6 on fault-tolerance is based on [Kemme et al. 2001, Jiménez-Peris et al. 2002, Perez-Sorrosal et al. 2006].

The concept of database cluster is defined in [Röhm et al. 2000, 2001]. Several protocols for scalable eager replication in database clusters using group communication are proposed in [Kemme and Alonso 2000b,a, Patiño-Martínez et al. 2000, Jiménez-Peris et al. 2002]. Their scalability has been studied analytically in [Jiménez-Peris et al. 2003]. Partial replication is studied in [Sousa et al. 2001]. The presentation of preventive replication in Sect. 8.7.2 is based on [Pacitti et al. 2005]. Load balancing in database clusters is addressed in [Milán-Franco et al. 2004, Gançarski et al. 2007].

Most of the content of Sect. 8.7.4 is based on the work on adaptive virtual partitioning [Lima et al. 2004] and hybrid partitioning [Furtado et al. 2008]. Physical partitioning in database clusters for decision-support is addressed by [Stöhr et al. 2000], using small grain partitions. Akal et al. [2002] propose a classification of OLAP queries such that queries of the same class have similar parallelization properties.

Exercises

Problem 8.1 (*) Consider a shared-disk cluster and very big relations that need to be partitioned across several disk units. How you would adapt the various partitioning and replication techniques in Sect. 8.3 to take advantage of shared-disk? Discuss the impact on query performance and fault-tolerance.

Problem 8.2 (**) Order-preserving hashing [Knuth 1973] could be used to partition a relation on an attribute A, so that the tuples in any partition $i+1$ have A values higher than those of the tuples in partition i. Propose a parallel sort algorithm that exploits order-preserving hashing. Discuss it advantages and limitations, compared with the b-way merge sort algorithm in Sect. 8.4.1.1.

Problem 8.3 Consider the parallel hash join algorithm in Sect. 8.4.1.2. Explain what the build phase and probe phase are. Is the algorithm symmetric with respect to its input relations?

Problem 8.4 (*) Consider the join of two relations `R` and `S` in a shared-nothing cluster. Assume that `S` is partitioned by hashing on the join attribute. Modify the parallel hash join algorithm in Sect. 8.4.1.2 to take advantage of this case. Discuss the execution cost of this algorithm.

Problem 8.5 (**) Consider a simple cost model to compare the performance of the three basic parallel join algorithms (nested loop join, sort-merge join, and hash join). It is defined in terms of total communication cost (C_{COM}) and processing cost (C_{PRO}). The total cost of each algorithm is therefore

$$Cost(Alg.) = C_{COM}(Alg.) + C_{PRO}(Alg.)$$

For simplicity, C_{COM} does not include control messages, which are necessary to initiate and terminate local tasks. We denote by $msg(\#tup)$ the cost of transferring a message of $\#tup$ tuples from one node to another. Processing costs (that include total I/O and CPU cost) are based on the function $C_{LOC}(m, n)$ that computes the local processing cost for joining two relations with cardinalities m and n. Assume that the local join algorithm is the same for all three parallel join algorithms. Finally, assume that the amount of work done in parallel is uniformly distributed over all nodes allocated to the operator. Give the formulas for the total cost of each algorithm, assuming that the input relations are arbitrary partitioned. Identify the conditions under which an algorithm should be used.

Problem 8.6 Consider the following SQL query:

```
SELECT ENAME, DUR
FROM   EMP, ASG, PROJ
WHERE  EMP.ENO=ASG.ENO
AND    ASG.PNO=PROJ.PNO
AND    RESP="Manager"
AND    PNAME="Instrumentation"
```

Give four possible operator trees: right-deep, left-deep, zigzag, and bushy. For each one, discuss the opportunities for parallelism.

Problem 8.7 Consider a nine way join (ten relations are to be joined), calculate the number of possible right-deep, left-deep, and bushy trees, assuming that each relation can be joined with anyone else. What do you conclude about parallel optimization?

Problem 8.8 (**) Propose a data placement strategy for a NUMA cluster (using RDMA) that maximizes a combination of *intranode* parallelism (intraoperator parallelism within shared-memory nodes) and *internode* parallelism (interoperator parallelism across shared-memory nodes).

Problem 8.9 (**) How should the DP execution model presented in Sect. 8.5.4 be changed to deal with interquery parallelism?

Problem 8.10 (**) Consider a multiuser centralized database system. Describe the main change to allow interquery parallelism from the database system developer and administrator's points of view. What are the implications for the end-user in terms of interface and performance?

Problem 8.11 (*) Consider the database cluster architecture in Fig. 8.19. Assuming that each cluster node can accept incoming transactions, make precise the database cluster middleware box by describing the different software layers, and their components and relationships in terms of data and control flow. What kind of information need be shared between the cluster nodes? how?

Problem 8.12 (**) Discuss the issues of fault-tolerance for the preventive replication protocol (see Sect. 8.7.2).

Problem 8.13 (**) Compare the preventive replication protocol with the eager replication protocol (see Chap. 6) in the context of a database cluster in terms of: replication configurations supported, network requirements, consistency, performance, fault-tolerance.

Problem 8.14 (**) Consider two relations R(A,B,C,D,E) and S(A,F,G,H). Assume there is a clustered index on attribute A for each relation. Assuming a database cluster with full replication, for each of the following queries, determine whether Virtual Partitioning can be used to obtain intraquery parallelism and, if so, write the corresponding subquery and the final result composition query.

(a)
```
SELECT B, COUNT(C)
FROM   R
GROUP BYB
```

(b)
```
SELECT C, SUM(D), AVG(E)
FROM   R
WHERE  B=:v1
GROUP BY C
```

(c)
```
SELECT B, SUM(E)
FROM.  R, S
WHERE. R.A=S.A
GROUP BY B
HAVING COUNT(*) > 50
```

(d)
```
SELECT B, MAX(D)
FROM+.  R, S
WHERE C = (SELECT SUM(G) FROM S WHERE S.A=R.A)
GROUP BY  B
```

(e)
```
SELECT B, MIN(E)
FROM.  R
WHERE  D > (SELECT MAX(H) FROM S WHERE G >= :v1)
GROUP BY B
```

Chapter 9
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 Chap. 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 Chap. 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 are 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:

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_9

- **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 is key lookup, 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, and query response time.
- **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 for sharing computation (e.g., SETI@home), communication (e.g., ICQ), or data (e.g., BitTorrent, Gnutella, and Kazaa). Our interest, naturally, is on data sharing systems. Popular systems such as BitTorrent, Gnutella, and Kazaa are quite limited when viewed from the perspective of database functionality. First, they provide only file level sharing with no sophisticated content-based search/query facilities. Second, they are single-application systems that focus on performing one task, and it is not straightforward to extend them for other applications/functions. 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 is replicated or cached in the system, a key issue is to maintain the consistency between these duplicates.

Figure 9.1 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 of the

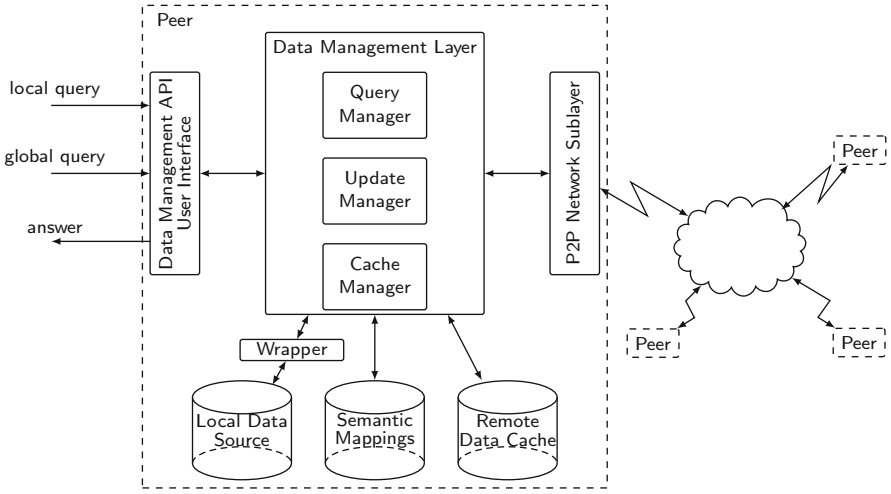


Fig. 9.1 Peer reference architecture

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. They 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 metainformation 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 is 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 is 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 is 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 is 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 Sect. 9.1. The problems of data mapping and the approaches to address them are the topics of Sect. 9.2. Query processing is discussed in Sect. 9.3. Data consistency and replication issues are discussed in Sect. 9.4. In Sect. 9.5, we introduce Blockchain, a P2P infrastructure for recording transactions efficiently, safely, and permanently.

9.1 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 distinction 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 *superpeer 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.

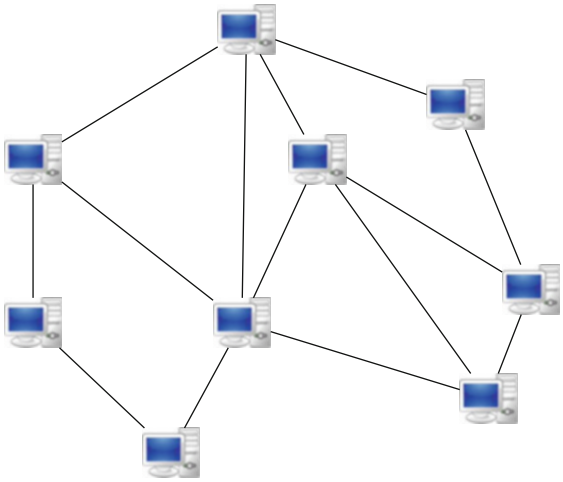


Fig. 9.2 Unstructured P2P network

9.1.1 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 9.2 shows an example unstructured P2P network.

Unstructured networks are the earliest examples of P2P systems whose core functionality 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 Gnutella, Freenet, Kazaa, and BitTorrent.

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 Chap. 2. 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.

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

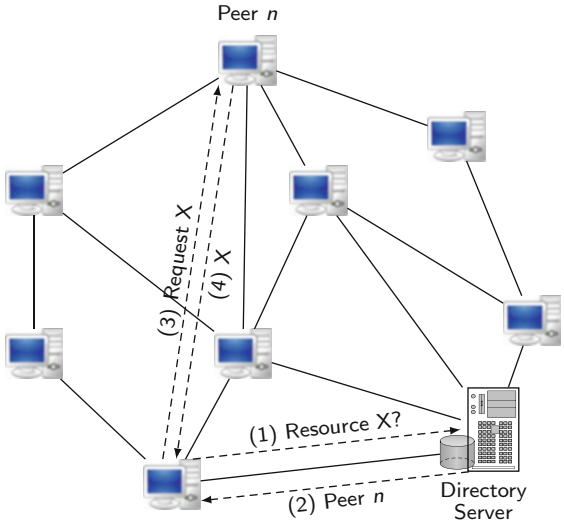


Fig. 9.3 Search over a centralized index. (1) A peer asks the central index manager for resource, (2) The response identifies the peer with the resource, (3) The peer is asked for the resource, (4) It is transferred

of the resource, followed by directly contacting the peer where the resource is located (Fig. 9.3). 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.

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 (Fig. 9.4).

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 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. There are different ways to determine this subset. For example, the concept of random walks can be used where each peer chooses a neighbor at random

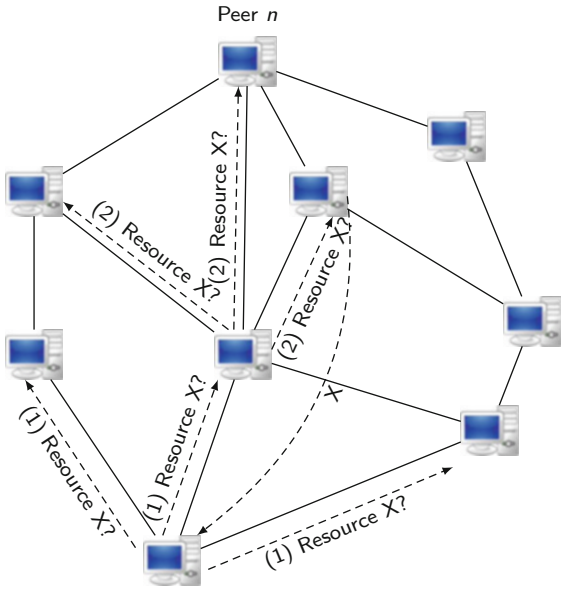


Fig. 9.4 Search over a Decentralized Index. (1) A peer sends the request for resource to all its neighbors, (2) Each neighbor propagates to its neighbors if it does not have the resource, (3) The peer who has the resource responds by sending the resource

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. 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. 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*. Gossiping has been initially proposed to maintain the mutual consistency of replicated data by spreading replica updates to all nodes over the network. 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 neighbor nodes. 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.

9.1.2 Structured P2P Networks

Structured P2P networks have emerged to address the scalability issues faced by unstructured P2P networks. 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 a *distributed hash table* (DHT). DHT-based systems provide two APIs: put(key, data) and get(key), where key is an object identifier. Each key (k_i) is hashed to generate a peer id (p_i), which stores the data corresponding to object contents (Fig. 9.5).

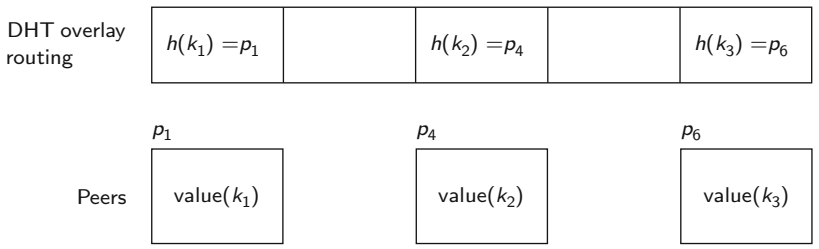


Fig. 9.5 DHT network

A straightforward approach could be to use the URI of the resource as the IP address of the peer that would hold the resource. 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 become the most widely accepted *base*[1] hash function that supports both uniformity and security (by supporting data integrity for the keys). The actual design of the hash function may be implementation dependent and we will not 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*. Routing geometry essentially defines the manner in which neighbors and routes are arranged. The routing algorithm corresponds to the routing protocol discussed above 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 trie approach, the leaf nodes correspond to the node identifiers that store the keys to be searched. The height of the trie is $\log n$, where n is the number of nodes in the trie. The search proceeds from the root to the leaves

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

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 trie. A popular DHT implementation that falls into this category is Tapestry, 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 trie used to route messages for the given identifier. Therefore, lookups proceed from the base of the spanning trie all the way to the root node of the identifier. Although this is somewhat different from traditional trie structures, Tapestry routing geometry is very closely associated with a trie structure and we classify it as such.

In trie 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 trie. 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 trie 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 trie-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). 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 trie 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 cannot 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 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 use different topologies.

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 several 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), and others. Specifically PHT and P-Grid 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 trie height). For

instance, BATON, VBI-tree, and BATON* employ k-way balanced trie structure to manage peers, and data is evenly partitioned among peers at the leaf-level. In comparison, P-Tree uses a B-tree structure with better flexibility on trie structural changes. SkipNet and Skip Graph are based on the skip list, and they link peers according to a randomized balanced trie 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 multidimensional 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.
- **Hyperrectangle structure.** In these systems, each dimension of the hyperrectangle 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 hyperrectangle space is then partitioned by peers based on their geometric positions in the space, and neighboring peers are interconnected to form the overlay network.

9.1.3 Superpeer P2P Networks

Superpeer 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 *superpeers*) act as dedicated serves for some other peers and can perform complex functions such as indexing, query processing, access control, and metadata management. If there is only one superpeer in the system, then this reduces to the client–server architecture. They are considered P2P systems, however, since the organization of the superpeers follows a P2P organization, and superpeers 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 superpeers.

In a superpeer network, a requesting peer sends the request, which can be expressed in a high-level language, to its responsible superpeer. The superpeer can then find the relevant peers either directly through its index or indirectly using its neighbor superpeers. More precisely, the search for a resource proceeds as follows (see Fig. 9.6):

1. A peer, say Peer 1, asks for a resource by sending a request to its superpeer.
2. If the resource exists at one of the peers controlled by this superpeer, it notifies Peer 1, and the two peers then communicate to retrieve the resource. Otherwise, the superpeer sends the request to the other superpeers.
3. If the resource does not exist at one of the peers controlled by this superpeer, the superpeer asks the other superpeers. The superpeer of the node that contains the resource (say Peer n) responds to the requesting superpeer.

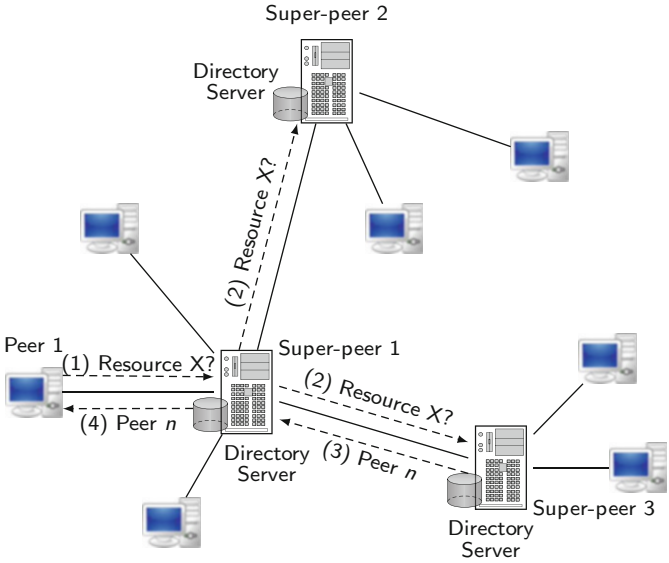


Fig. 9.6 Search over a superpeer system. (1) A peer sends the request for resource to all its superpeer, (2) The superpeer sends the request to other superpeers if necessary, (3) The superpeer one of whose peers has the resource responds by indicating that peer, (4) The superpeer notifies the original peer

4. Peer *n*'s identity is sent to Peer 1, after which the two peers can communicate directly to retrieve the resource.

The main advantages of superpeer networks are efficiency and quality of service (e.g., completeness of query results, query response time). The time needed to find data by directly accessing indices in a superpeer is very small compared with flooding. In addition, superpeer networks exploit and take advantage of peers' different capabilities in terms of CPU power, bandwidth, or storage capacity as superpeers 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 superpeers. However, autonomy is restricted since peers cannot log in freely to any superpeer. Fault-tolerance is typically lower since superpeers are single points of failure for their subpeers (dynamic replacement of superpeers can alleviate this problem).

Examples of superpeer networks include Edutella and JXTA.

Requirements	Unstructured	Structured	Superpeer
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. 9.7 Comparison of approaches

9.1.4 Comparison of P2P Networks

Figure 9.7 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 superpeer systems by relying on replication and fail-over techniques. Query expressiveness can be improved by supporting more complex queries on top of structured networks.

9.2 Schema Mapping in P2P Systems

We discussed the importance of, and the techniques for, designing database integration systems in Chap. 7. 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.

9.2.1 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.

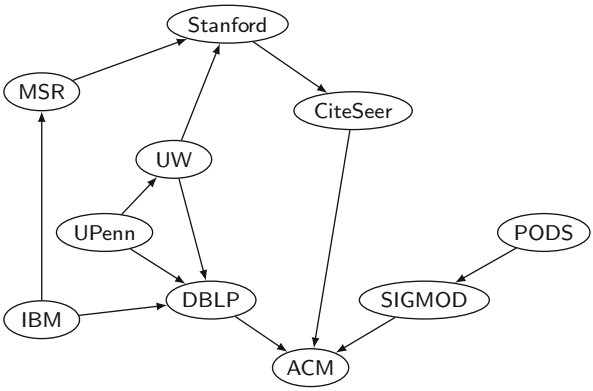


Fig. 9.8 An example of pairwise schema mapping in piazza

Piazza follows this approach (see Fig. 9.8). The data is 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.

The Local Relational Model (LRM) 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 generalizes this approach to deal with autonomous peers that form acquaintances at runtime, 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 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.

9.2.2 Mapping Based on Machine Learning Techniques

This approach is generally used when the shared data is 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 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 X and Y, the data of concept Y is classified using X's classifier and vice versa. The number of values that can be successfully classified into X and Y represent the similarity between X and Y.

9.2.3 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. The APPA P2P system 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 Fig. 9.9). 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 9.1 Given two CSD relation definitions $\mathtt{R}_1$ and $\mathtt{R}_2$, an example of peer mapping at peer p is

$$p : \mathtt{R}(\mathtt{A}, \mathtt{B}, \mathtt{D}) \subseteq csd : \mathtt{R}_1(\mathtt{A}, \mathtt{B}, \mathtt{C}), csd : \mathtt{R}_2(\mathtt{C}, \mathtt{D}, \mathtt{E})$$

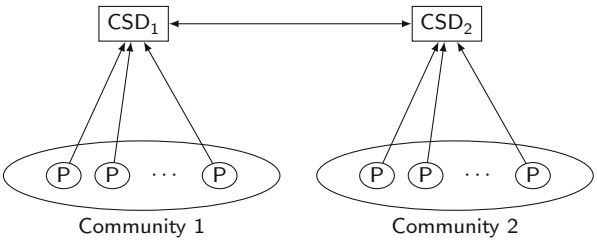


Fig. 9.9 Common agreement schema mapping in APPA

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. ♦

9.2.4 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 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 also follows this approach for schema mapping in superpeer networks. Resources in Edutella are described using the RDF metadata model, and the descriptions are stored at superpeers. When a user issues a query at a peer p, the query is sent to p's superpeer where the stored schema descriptions are explored and the addresses of the relevant peers are returned to the user. If the superpeer does not find relevant peers, it sends the query to other superpeers such that they search relevant peers by exploring their stored schema descriptions. In order to explore stored schemas, superpeers 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.

9.3 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.

9.3.1 Top-k Queries

Top-k queries have been used in many domains such as network and system monitoring, information retrieval, and multimedia databases. 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 complete answer set is very large.

Example 9.2 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 P2P systems is difficult because of the scale of the network. 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.

9.3.1.1 Basic Techniques

An efficient algorithm for top-k query processing in centralized and distributed systems is the Threshold Algorithm (TA) . 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. 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 Fig. 9.10. 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 m sorted lists of n data items, TA (see Algorithm 9.1) 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

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. 9.10 Example database with 3 sorted lists

Algorithm 9.1: Threshold Algorithm (TA)

```
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
    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)
            end for
        end for
        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 L_i)
        j ← j + 1
    end while
end
```

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 Fig. 9.10. 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 $30 + 28 + 30$. 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 9.3 Consider again the database (i.e., three sorted lists) shown in Fig. 9.10. 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 are 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. ♦

TA-Style Algorithms

Several TA-style algorithms, i.e., extensions of TAThreshold Algorithm, 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, 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 executed by the query originator is detailed in Algorithm 9.2.

TPUT 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 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 9.4 Consider the first two sorted lists (List 1 and List 2) in Fig. 9.10. Assume a top-2 query Q, i.e., $k = 2$, where the scoring function is sum. Phase 1

Algorithm 9.2: 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
  end for
  $Z \leftarrow$ 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
  end for
  $Z \leftarrow$ 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 λ_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
  end for
  $Y \leftarrow k$ data items with highest overall score in Z
end

produces the sets $Y = \{d_1, d_2, d_4, d_6\}$ and $Z = \{d_1, d_2\}$. The $k-$th (i.e., second) data item is d_2, whose partial sum is 28. Thus we get $\lambda_1/2 = 28/2 = 14$. Let us now denote each data item d in Y as (d, score in List 1, score in List 2). 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:

$$
\begin{aligned}
u(d_1) &= 30 + 21 = 51 \\
u(d_2) &= 14 + 28 = 42 \\
u(d_3) &= 26 + 14 = 40 \\
u(d_4) &= 28 + 14 = 42 \\
u(d_5) &= 17 + 24 = 41 \\
u(d_6) &= 14 + 27 = 41 \\
u(d_7) &= 25 + 25 = 50 \\
u(d_8) &= 23 + 20 = 43 \\
u(d_9) &= 27 + 23 = 50
\end{aligned}
$$

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.

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 9.3). 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. The key idea of BPA is that the stopping mechanism takes into account special 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 9.3) 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 9.5 To illustrate basic BPA, consider again the three sorted lists shown in Fig. 9.10 and the query Q in Example 9.3.

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.

Algorithm 9.3: 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 ← ∅
    end for
    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)
            end for
            P_i ← P_i ∪ {positions seen under sorted or random access}
            bp_i ← best position in L_i
        end for
        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 while
end
```

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.

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. It has also been shown that the execution cost of BPA can be $(m - 1)$ times (where m is the number of sorted lists) 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, 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.

9.3.1.2 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, 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. 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 9.4):

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 trie-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

Algorithm 9.4: 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
    end
    for each peer that receives Q from a peer p do
        TTL ← TTL − 1
        if TTL > 0 then
            send Q to neighbors
        end if
        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 for
end
```

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.

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.

9.3.1.3 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 Sect. 9.3.1.1) and a mechanism that stores the shared data in the DHT in a fully distributed fashion. 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 $\prec$ on D_a (e.g., D_a is numeric). D_a is partitioned into n nonempty subdomains $d_1, d_2, \ldots, d_n$ such that their union is equal to D_a, the intersection of any two different subdomains is empty, and for each $v_1 \in d_i$ and $v_2 \in d_j$, if $i \prec j$, then we have $v_1 \prec v_2$. The hash function is applied on the subdomain of the attribute value. Thus, for the attribute values that fall in the same subdomain, 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 subdomains), domain partitioning is done in such a way that attribute values are uniformly distributed in subdomains. 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 9.5), 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 subdomains, and then it

Algorithm 9.5: 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_i in A do
        L_{A_i} ← all subdomains of A_i
        L_{A_i} ← L_{A_i} − subdomains which do not satisfy Q's condition
        Sort L_{A_i} in descending order of its subdomains
    end for
    {Phase 2: continuously retrieve attribute values and their tuples until finding k top
     tuples}
    Done ← false
    for each scoring attribute A_i in A in parallel do
        i ← 1
        while (i < number of subdomains of A) and not Done do
            send Q to peer p that maintains the attribute values of subdomain i in L_{A_i}
            Z ← A_i 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
                end if
                i ← i + 1
            end for
        end while
    end for
end
```

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 $\mathtt{A}_i$, p_0 prepares the list of subdomains and sorts them in descending order of their positive impact on the scoring function. For each list, p_0 removes from the list the subdomains 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., $\mathtt{A}_i = 10$), then p_0 removes from the list all the subdomains except the subdomain to which the constant value belongs. Let us denote by $L_{\mathtt{A}_i}$ the list prepared in this phase for a scoring attribute $\mathtt{A}_i$.
2. For each scoring attribute $\mathtt{A}_i$, in parallel, p_0 proceeds as follows. It sends Q and $\mathtt{A}_i$ to the peer, say p, that is responsible for storing the values of the first subdomain of $L_{\mathtt{A}_i}$, and requests it to return the values of $\mathtt{A}_i$ 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 $\mathtt{A}_i$ to the site that is responsible for the second subdomain of $L_{\mathtt{A}_i}$ and so on until k top tuples are found.

Let $\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_m$ 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 nonmonotonic functions.

9.3.1.4 Top-k Queries in Superpeer Systems

A typical algorithm for top-k query processing in superpeer systems is that of Edutella. In Edutella, a small percentage of nodes are superpeers and are assumed to be highly available with very good computing capacity. The superpeers 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 superpeer, which then sends it to the other superpeers. The superpeers 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 superpeer. Each superpeer chooses the overall maximum scored item from all received data items. For determining the second best item, it only asks one peer, the one that has returned the first top item, to return its second top-scored item. The superpeer 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 superpeers send their top items to the superpeer of the query originator, to extract the overall k top items, and send them to the query originator. This algorithm minimizes communication between peers and superpeers since, after having received the maximum scored data items from each peer connected to it, each superpeer asks only one peer for the next top item.

9.3.2 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 that provides support for complex queries on top of DHTs. The solution is a variation of the parallel hash join algorithm (PHJ) (see Sect. 8.4.1) 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 (see Sect. 8.4.1.2) 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 9.6): 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 Chap. 4 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 Sect. 8.4.1), where only one relation needs to be hashed and sent over the DHT.

Algorithm 9.6: 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 $h(H_{\mathrm{T}}, \mathrm{A})$
    end for
  end for
  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 $h(H_{\mathrm{T}}, \mathrm{A})$
    end for
  end for
  {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 $h(\mathrm{A})$
      else
        probe r tuples in R_p using $h(\mathrm{A})$
      end if
      $\mathrm{T}_p \leftarrow \mathrm{r} \bowtie \mathrm{s}$
    end if
  end for
end

9.3.3 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 "$\mathrm{A} = a$") but have difficulties with range queries. The main reason is that hashing tends to destroy the ordering of data that is useful in finding ranges quickly.

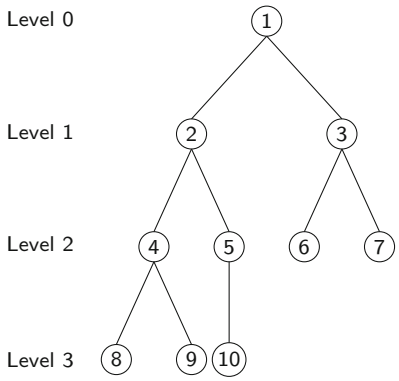
There are two main approaches for supporting range queries in structured P2P systems: extend a DHT with proximity or order-preserving properties, or maintain the key ordering with a trie-based structure. The first approach has been used in several systems. Locality sensitive hashing is an extension to 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.

The Prefix Hash Tree (PHT) 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 trie-based structures to maintain balanced ranges of keys. The first attempt to build a P2P network based on a balanced trie structure is BATON (BAlanced Tree Overlay Network). We now present BATON and its support for range queries in more detail.

BATON organizes peers as a balanced binary trie (each node of the trie 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 trie 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 trie. Figure 9.11 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 Fig. 9.12). Thus, BATON builds an effective distributed index structure. The joining and departure of peers are processed such that the trie remains balanced by forwarding the request upward in the trie 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. 9.11 BATON structure-tree index and routing table of node 6

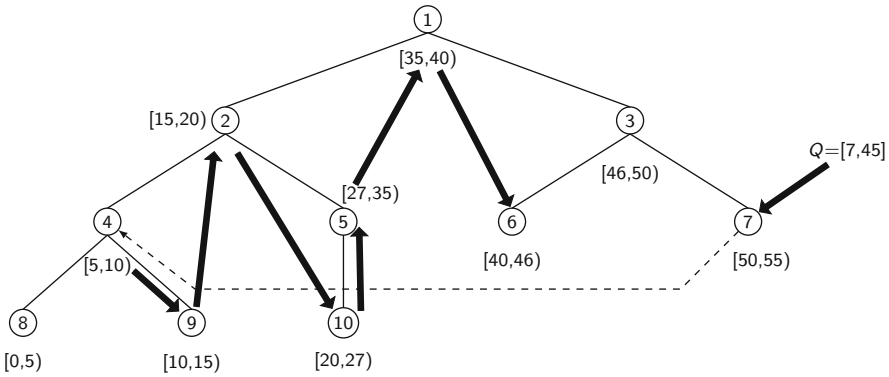


Fig. 9.12 Range query processing in BATON

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

A range query is processed as follows (Algorithm 9.7). 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 9.7: 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 if
    end for
end
```

Example 9.6 Consider the query Q with range [7, 45] issued at node 7 in Fig. 9.12. 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. ♦

9.4 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.

9.4.1 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. 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 is an extensible P2P system that provides decentralized object location and routing on top of a structured overlay network. It routes messages to logical endpoints (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 9.13a 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 Fig. 9.13a, 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 $id(o)$.

Tapestry offers the entire infrastructure needed to take advantage of replicas, as shown in Fig. 9.13b. 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 ($O_1 \rightarrow$ 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

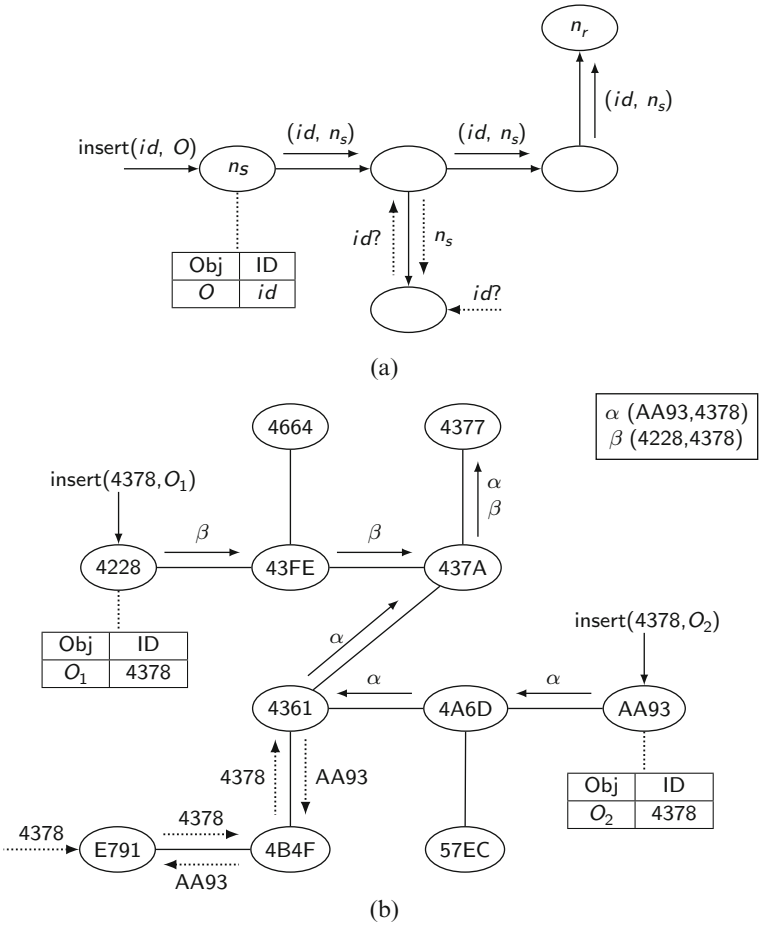


Fig. 9.13 Tapestry. (**a**) Object publishing. (**b**) Replica management

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 Fig. 9.13b, 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.

9.4.2 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 9.7 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 is 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 Chap. 6, but the issue is particularly difficult and important in P2P systems, since there is considerable dynamism in the peers joining and leaving the system.

A solution has been proposed that considers both data availability and data currency. To provide high data availability, data is 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 any time, 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.

9.4.3 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 Chap. 6) 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.

9.4.3.1 OceanStore

OceanStore 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 is protected through redundancy and cryptographic techniques. To improve performance, data is allowed to be cached anywhere in the network.

OceanStore allows concurrent updates on replicated objects and relies on reconciliation to assure data consistency. A replicated object can have multiple primary replicas and secondary replicas at different nodes. The primary replicas are all linked and cooperate among themselves to achieve replica mutual consistency by ordering updates. Secondary replicas provide a lesser degree of consistency in order to gain performance and availability. Thus, secondary replicas may be less up-to-date and can be in higher numbers than primary replicas. Secondary replicas communicate among themselves and primary replicase via an epidemic algorithm.

Figure 9.14 illustrates update management in OceanStore. In this example, R is the (only) replicated object, whereas R and R_{sec} denote, respectively, a primary and

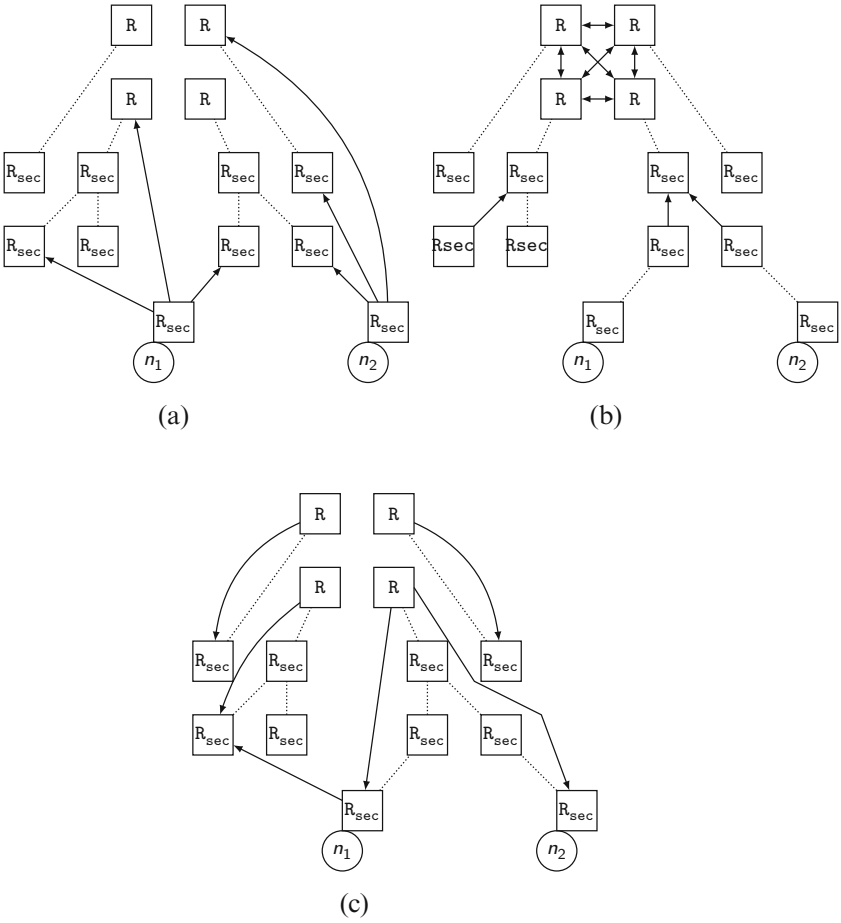


Fig. 9.14 OceanStore reconciliation. (**a**) Nodes n_1 and n_2 send updates to the master group of R and to several random secondary replicas. (**b**) The master group of R orders updates while secondary replicas propagate them epidemically. (**c**) After the master group agreement, the result of updates is multicast to secondary replicas

a secondary copy of R. The four nodes holding a primary copy are linked to each other (not shown in the figure). Dotted lines represent links between nodes holding primary or secondary replicas. Nodes n_1 and n_2 are concurrently updating R. Such updates are managed as follows. Nodes that hold primary copies of 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 Fig. 9.14a). 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 (Fig. 9.14b). Once the master group obtains an agreement, the

result of updates is multicast to secondary replicas (Fig. 9.14c), which contain both tentative[2] and committed data.

Replica management adjusts the number and location of replicas in order to serve 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.

9.4.3.2 P-Grid

P-Grid 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 quasiconsistency 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.

9.4.3.3 APPA

APPA provides a general lazy distributed replication solution that assures eventual consistency of replicas. It uses the IceCube action-constraint framework 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

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

reconciliation is to take a set of actions with the associated constraints and produce 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 nonnull 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 perform simultaneously independent activities on a distinct subset of actions (e.g., ordering of different clusters).

9.5 Blockchain

Popularized by bitcoin and other cryptocurrencies, blockchain is a recent P2P infrastructure that can record transactions between two parties efficiently and safely. It has become a hot topic, subject to much hype and controversy. On the one hand, we find enthusiastic proponents such as Ito, Narula, and Ali claiming in 2017 that blockchain is a disruptive technology that "will do to the financial system what the Internet did to media." On the other hand, we find strong opponents, e.g., famous economist N. Roubini who calls blockchain in 2018 the most "overhyped and least useful technology in human history." As always, the truth is probably somewhere in between.

Blockchain was invented for bitcoin to solve the double spending problem of previous digital currencies without the need of a trusted, central authority. On January 3, 2009, Satoshi Nakamoto[3] created the first source block with a unique transaction of 50 bitcoins to himself. Since then, there have been many other blockchains such as Ethereum in 2013 and Ripple in 2014. The success has been significant and cryptocurrencies have been used a lot for money transfer or high-risk investment, e.g., initial coin offerings (ICOs) as an alternative to initial public offerings (IPOs). The potential advantages of using a blockchain-based cryptocurrency are the following:

- Low transaction fee (set by the sender to speed up processing), which is independent of the amount of money transferred;
- Fewer risks for merchants (no fraudulent chargebacks);
- Security and control (e.g., protection from identity theft);
- Trust through the blockchain, without any central authority.

However, cryptocurrencies have also been used a lot for scams and illegal activities (purchases on the dark web, money laundering, theft, etc.), which has triggered warnings from market authorities and beginning of regulation in some countries. Other problems are that it is:

- unstable: as there is no backing by a state or federal bank (unlike strong currencies like Dollar or Euro);

[3] Pseudo for the person or people who developed bitcoin, which generated much speculation about their true identity.

- unrelated to real economy, which fosters speculation;
- highly volatile, e.g., the exchange rate with a real currency (as set by cryptocurrency marketplaces) can greatly vary in a few hours;
- subject to severe crypto-bubble bursts, as in 2017.

Thus, there are pros and cons to blockchain-based cryptocurrencies. However, we should avoid restricting the blockchain to cryptocurrency, as there are many other useful applications. The original blockchain is a public, distributed ledger that can record and share transactions among a number of computers in a secure and permanent way. It is a complex distributed database infrastructure, combining several technologies such as P2P, data replication, consensus, and public key encryption. The term Blockchain 2.0 refers to new applications that can be programmed into the blockchain to go beyond transactions and enable exchange of assets without powerful intermediaries. Examples of such applications are smart contracts, persistent digital ids, intellectual property rights, blogging, voting, reputation, etc.

9.5.1 Blockchain Definition

Recording financial transactions between two parties has been traditionally done using an intermediary centralized ledger, i.e., a database of all transactions, controlled by a trusted authority, e.g., a clearing house. In a digital world, this centralized approach has several problems. First, it creates single points of failure and makes it an attractive target for attackers. Second, it favors concentration of actors such as big financial institutions. Third, complex transactions that require multiple intermediaries, typically with heterogeneous systems and rules, may be difficult and take time to execute.

A blockchain is essentially a distributed ledger shared among a number of participant nodes in a P2P network. It is organized as an append-only, replicated database of blocks. Blocks are digital containers for transactions and are secured through public key encryption. The code of each new block is built on that of the preceding block, which guarantees that it cannot be tampered with. The blockchain is viewed by all participants that maintain database copies in multimaster mode (see Chap. 6) and collaborate through consensus in validating the transactions in the blocks. Once validated and recorded in a block, a transaction cannot be modified or deleted, making the blockchain tamper-proof. The participant nodes may not fully trust each other and some may even behave in malicious (Byzantine) manner, i.e., give different values to different observer nodes. Thus, in the general case, i.e., public blockchain as in bitcoin, the blockchain must tolerate Byzantine failures.

Note that the objective of a typical P2P data structure such as a DHT is to provide fast and scalable lookup. The purpose of a blockchain is quite different, i.e., to manage a continuously growing list of blocks in a secure and tamper-proof manner. But scalability is not an objective as the blockchain is not partitioned across P2P nodes.

Compared with the centralized ledger approach, the blockchain can bring the following advantages:

- Increased trust in transactions and value exchange, by trusting the data, not the participants.
- Increased reliability (no single point of failure) through replication.
- Built-in security through chaining of blocks and public key encryption.
- Efficient and cheaper transactions between participants, in particular, compared with relying on a long chain of intermediaries.

Blockchains can be used in two different kinds of markets: public, e.g., cryptocurrency, public auction, where anybody can join in, and private, e.g., supply chain management, healthcare, where participants are known. Thus, an important distinction to make is between public and private (also called permissioned) blockchains.

A public blockchain (like bitcoin) is an open P2P nonpermissioned network and can be very large scale. Participants are unknown and untrusted, and can join and leave the network without notification. They are typically pseudonymized which makes it possible to track a participant's entire transaction history and sometimes even to identify the participant.

A private blockchain is a closed permissioned network, so its scale is typically much smaller than a public blockchain. Control is regulated to ensure that only identified, approved participants can validate transactions. Access to blockchain transactions can be restricted to authorized participants, which increases data protection. Although the underlying infrastructure can be the same, the main difference between public and private blockchain is who (person, group, or company) is allowed to participate in the network and who controls it.

9.5.2 Blockchain Infrastructure

In this section, we introduce the blockchain infrastructure as originally proposed for bitcoin, focusing on the process of transaction processing. Participant nodes are called *full nodes* to distinguish from other nodes, e.g., lightweight client nodes that handle digital wallets. When a new full node joins the network, it synchronizes with known nodes using Domain Name System (DNS) to obtain a copy of the blockchain. Then, it can create transactions and become a "miner," i.e., participate in the validation of blocks called "mining" process.

Transaction processing is done in three main steps:

1. Creating a transaction after two users have agreed on transaction information exchange: wallet addresses, public keys, etc.
2. Grouping of transactions in a block and linking with a previous block.
3. Validation of the block (and of the transactions) using "mining," addition of the validated block in the blockchain and replication in the network.

In the rest of this section, we present each step in more detail.

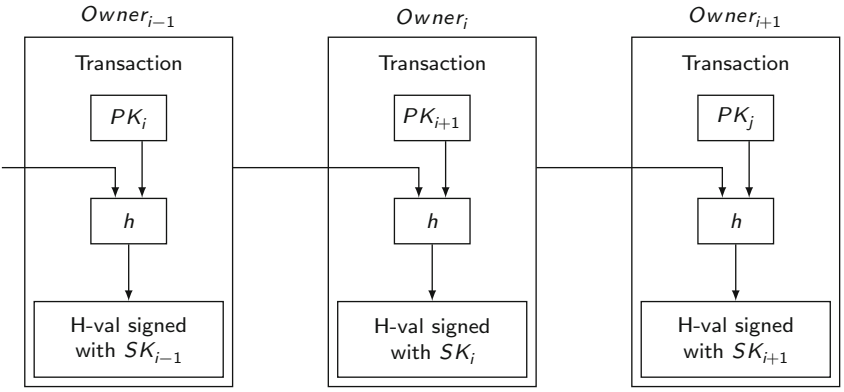


Fig. 9.15 Chaining of transactions

9.5.2.1 Creating a Transaction

Let us consider a bitcoin transaction between a coin owner and a coin recipient that receives the money. The transaction is secured with public key encryption and digital signature. Each owner has a public and private key. The coin owner signs the transaction by

- creating a hash digest of a combination of the previous transaction (with which it receives the coins) and of the public key of the next owner;
- signing the hash digest with its private key.

This signature is then appended to the end of the transaction, thus making a chain of transactions between all owners (see Fig. 9.15). Then, the coin owner publishes the transaction in the network by multicasting it to all other nodes. Given the public key of the coin owner who created the transaction, any node in the network can verify the transaction's signature.

9.5.2.2 Grouping Transactions into Blocks

Double spending is a potential flaw in a digital cash scheme in which the same single digital token can be spent more than once. Unlike physical cash, a digital token consists of a digital file that can be duplicated or falsified.

Each miner node (which maintains a copy of the blockchain) receives the transactions that get published, validates them, and groups them into blocks. To accept a transaction and include it in a block, the miners follow some rules such as checking that the inputs are valid and that a coin is not double-spent (spent more than once) as a result of an attack (see 51% attack next). It may be possible that a malicious miner tries to accept a transaction that violates some rules and include it in a block. In this case, the block will not obtain the consensus of other miners

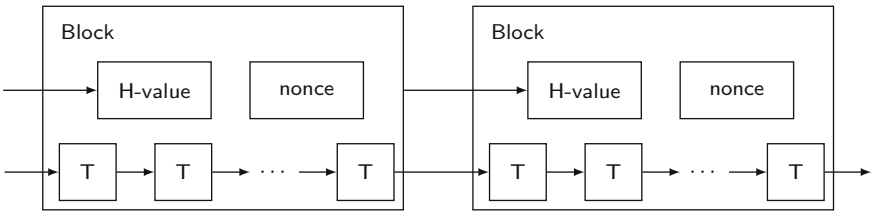


Fig. 9.16 Chaining of blocks

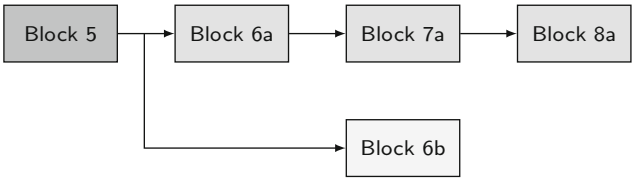


Fig. 9.17 Longest chain rule

that follow the rules and will not be accepted and included in the blockchain. Thus, if a majority of miners follow the rule, the system works. As shown in Fig. 9.16, each new block is built on a previous block of the chain by producing a hash digest ($h-$value) of the previous block's address, thus protecting the block from tampering or change. The current size of a bitcoin block is 1 Megabyte, reflecting a compromise between efficiency and security.

A problem that can arise is an accidental or intentional fork. As different blocks are validated in parallel by different nodes, one node can see several candidate chains at any time. For instance, in Fig. 9.17, a node may see blocks 7a and 6b, both originated from block 5. The solution is to apply the longest chain rule, i.e., choose the block which is in the longest chain. In the example of Fig. 9.17, the block 7a will be chosen to build the next block 7b. The rationale for this rule is to minimize the number of transactions that need to be resubmitted. For instance, transactions in Block 6b have to be resubmitted by the client (who will see that the block has not been validated). Thus, transactions in a validated block are only provisionally validated and confirmation must be awaited. Each new block accepted in the chain after the validation of the transaction is considered as a confirmation. Bitcoin considers a transaction mature after 6 confirmations (1 hour on average). In Fig. 9.17, transaction maturity is illustrated by the darkness of the boxes (Block 6b is lighter because its transactions will not be confirmed).

In addition to accidental forks, there are also intentional forks, which are useful to add new features to the blockchain code base (protocol changes) or to reverse the effects of hacking or catastrophic bugs. Two kinds of fork are possible: *soft fork* versus *hard fork*. A soft fork is backward compatible: the old software recognizes blocks created with new rules as valid. However, it makes it easy for attackers. A famous occurrence of a hard fork is that of the Ethereum blockchain in 2016, after

an attack against a complex smart contract for venture capital. Ethereum forked but without momentum from the community managing the software, thus leading to two blockchains: (new) Ethereum and (old) Ethereum Classic. Note that the battle has been more philosophical and ethical than technical.

9.5.2.3 Block Validation by Consensus

Since blocks are being produced in parallel by competing nodes, a consensus is needed to validate and add them to the blockchain. Note that in the general case of the public blockchain where participants are unknown, traditional consensus protocols such as Paxos (see Sect. 5.4.5) are not applicable. The consensus protocol of the bitcoin blockchain is based on *mining*.[4] We can summarize the consensus protocol as follows:

1. Miner nodes compete (as in a lottery) to produce new blocks. Using much computing power, each miner tries to produce a nonce (number used once) for the block (see Fig. 9.16).
2. Once a miner has found the nonce, it adds the block to the blockchain and multicasts it to all network nodes.
3. Other nodes verify the new block, by checking the nonce (which is easy).
4. Since many nodes try to be the first to add a block to the blockchain, a lottery-based reward system selects one of the competing blocks, based on some probability, and the winner gets paid, e.g., 12.5 bitcoins today (originally 50). This increases the money supply.

Mining is designed to be difficult. The more mining power the network has, the harder it is to compute the nonce. This allows controlling the injection of new blocks ("inflation") in the system, on average 1 block every 10 minutes. The mining difficulty consists in producing a Proof of Work (PoW), i.e., a piece of data that is difficult to calculate but easy to verify, to calculate the nonce. PoW was first proposed to prevent DoS attacks. The bitcoin blockchain uses the Hashcash PoW, which is based on the SHA-256 hash function. The goal is to produce a value v such that $h(f(block, v)) < T$, where

1. h is the SHA-256 hash function;
2. f is a function that combines v with information in the block, so the nonce cannot be precomputed;
3. T is a target value shared by all nodes and reflects the size of the network;
4. v is a 256-bit number starting with n zero bits.

The average effort to produce the PoW is exponential in the number of zero bits required, i.e., the probability of success is low and can be approximated as $1/2^n$.

[4] The term is used by analogy to gold mining as the process of bringing out coins that exist in the protocol's design.

This advantages powerful nodes, which now use big clusters of GPUs. However, verification is very simple and can be done by executing a single hash function.

A potential problem with PoW based mining is the *51% attack*, which enables the attacker to invalidate valid transactions and double spend funds. To do so, the attacker (a miner or miner coalition) must hold more than 50% of the total computing power for mining. It then becomes possible to modify a received chain (e.g., by removing a transaction) and produce a longer chain that will be selected by the majority according to the longest chain rule.

9.5.3 Blockchain 2.0

The first generation blockchain, pioneered by bitcoin, enables recording of transactions and exchange of cryptocurrencies without powerful intermediaries. Blockchain 2.0 is a major evolution of the paradigm to go beyond transactions and enable exchange of all kinds of assets. Pioneered by Etherum, it makes blockchain programmable, allowing application developers to build APIs and services directly on the blockchain.

Critical characteristics of the applications are that asset and value are exchanged (through transactions), there are multiple participants, possibly unknown to each other, and trust (in the data) is critical. There are many applications of Blockchain 2.0 in many industries, e.g., financial services and micropayments, digital rights, supply chain management, healthcare record keeping, Internet of Things (IoT), food provenance. Most of these applications can be supported by a private blockchain. In this case, the major advantages are increased privacy and control, and more efficient transaction validation since participants are trusted and there is no need to produce a PoW.

An important capability that can be supported in Blockchain 2.0 is smart contracts. A smart contract is a self-executing contract, with code that embeds the terms and conditions of a contract. An example of simple smart contract is a service contract between two parties, one that requests the service with an associated payment, and the other that fulfills the service and once executed gets the payment. In a blockchain, contracts can be partially or fully executed without human interaction and involve many participants, e.g., IoT devices. A major advantage of having smart contracts in the blockchain is that the code, which implements the contract, becomes visible to all for verification. However, once on a blockchain the contract cannot be changed. From a technical point of view, the main challenge is to produce bug-free code, which would best be done using code verification.

An important collaborative initiative to produce open source blockchains and related tools is the Hyperledger project of the Linux Foundation that was started in 2015 by IBM, Intel, Cisco, and others. The major frameworks are:

- Hyperledger Fabric (IBM, digital Asset): a permissioned blockchain infrastructure with smart contracts, configurable consensus, and membership services.

- Sawtooth (Intel): a novel consensus mechanism, "Proof of Elapsed Time," that builds on trusted execution environments.
- Hyperledger Iroha (Soramitsu): based on Hyperledger Fabric, with a focus on mobile applications.

9.5.4 Issues

Blockchain is often advertised as a disruptive technology for recording transactions and verifying records, with much impact on the finance industry. In particular, the ability to program applications and business logic in the blockchain opens up many possibilities for developers, e.g., smart contracts. Some proponents, e.g., cypherpunk activists, even consider it as a potential disruptive power that will establish a sense of democracy and equality, where individuals and small businesses will be able to compete with corporate powers.

However, there are important limitations, in particular in the case of the public blockchain, as is the most general infrastructure. The limitations are:

- Complexity and scalability, in particular, difficult evolution of operating rules that require forking the blockchain.
- Ever increasing chain size and high energy consumption (with PoW).
- Potential for a 51% attack.
- Low privacy as users are only pseudonymized. For instance, making a transaction with a user may reveal all its other transactions.
- Unpredictable duration of transactions, from a few minutes to days.
- Lack of control and regulation, which makes it hard for states to watch and tax transactions.
- Security concerns: if a private key is lost or stolen, an individual has no recourse.

To address these limitations, several research issues in distributed systems, software engineering, and data management can be identified:

- Scalability and security of the public blockchain. This issue has triggered renewed interest on consensus protocols, with more efficient alternatives to PoW: proof-of-stake, proof-of-hold, proof-of-use, proof-of-stake/time. Furthermore, there are other performance bottlenecks beside consensus. However, a major issue remains the trade-off between performance and security. Bitcoin-NG is a new generation blockchain with two types of blocks: key blocks that include PoW, a reference to previous block, and mining reward, which makes PoW computing more efficient; and microblocks that include transactions, but no PoW.
- Smart contract management, including code certification and verification, contract evolution (change propagation), optimization, and execution control.
- Blockchain and data management. As a blockchain is merely a distributed database structure, it can be improved by drawing from design principles of database systems. For instance, a declarative language could make it easier

to define, verify, and optimize complex smart contracts. BigchainDB is a new DBMS that applies distributed database concepts, in particular, a rich transaction model, role-based access control, and queries, to support a scalable blockchain. Understanding the performance bottlenecks also requires benchmarking. BLOCKBENCH is a benchmarking framework for understanding the performance of private blockchains against data processing workloads.

- Blockchain interoperability. There are many blockchains, each with different protocols and APIs. The Blockchain Interoperability Alliance (BIA) has been established to define standards in order to promote cross-blockchain transactions.

9.6 Conclusion

By distributing data storage and processing across autonomous peers in the network, P2P systems can scale without the need for powerful servers. Today, major data sharing applications such as BitTorrent, eDonkey, or Gnutella are used daily by millions of users. P2P has also been successfully used to scale data management in the cloud, e.g., DynamoDB key-value store (see Sect. 11.2.1). However, these applications remain limited in terms of database functionality.

Advanced P2P applications such as collaborative consumption (e.g., car sharing) must deal with semantically rich data (e.g., XML or RDF 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 data management system are autonomy, query expressiveness, efficiency, quality of service, and fault-tolerance. Depending on the P2P network architecture (unstructured, structured DHT, or superpeer), 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 superpeer networks for more complex searching.

Most of the work on data sharing in P2P systems has initially focused on schema management and query processing, in particular to deal with semantically rich data. However, more recently with blockchain, there has been much more work on update management, replication, transactions, and access control, yet over relatively simple data. P2P techniques have also received some attention to help scaling up data management in the context of Grid Computing or to help protecting data privacy in the context of information retrieval or data analytics.

Research on P2P data management is having renewed interest in two major contexts: blockchain and edge computing. In the context of blockchain, the major research issues, which we discussed at length at the end of Sect. 9.5, have to do with scalability and security of the public blockchain (e.g., consensus protocols), smart

contract management, in particular, using declarative query languages, benchmarking, and blockchain interoperability. In the context of edge computing, typically with IoT devices, mobile edge servers could be organized as a P2P network to offload data management tasks. Then, the issues are at the crossroads of mobile and P2P computing.

9.7 Bibliographic Notes

Data management in "modern" P2P systems is characterized by massive distribution, inherent heterogeneity, and high volatility. The topic is fully covered in several books including [Vu et al. 2009, Pacitti et al. 2012]. 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].

In unstructured P2P networks, the problem of flooding is handled using one of two methods as noted. Selecting a subset of neighbors to forward requests is due to Kalogeraki et al. [2002]. The use of random walks to choose the neighbor set is proposed by Lv et al. [2002], using a neighborhood index within a radius is due to Yang and Garcia-Molina [2002], and maintaining a resource index to determine the list of neighbors most likely to be in the direction of the searched peer is proposed by Crespo and Garcia-Molina [2002]. The alternative proposal to use epidemic protocol is discussed in [Kermarrec and van Steen 2007] based on gossiping that is discussed in [Demers et al. 1987]. Approaches to scaling gossiping are given in [Voulgaris et al. 2003].

Structured P2P networks are discussed in [Ritter 2001, Ratnasamy et al. 2001, Stoica et al. 2001]. Similar to DHTs, dynamic hashing has also been successfully used to address the scalability issues of very large distributed file structures [Devine 1993, Litwin et al. 1993]. DHT-based overlays can be categorized according to their routing geometry and routing algorithm [Gummadi et al. 2003]. We introduced in more details the following DHTs: Tapestry[Zhao et al. 2004], CAN [Ratnasamy et al. 2001], and Chord [Stoica et al. 2003]. Hierarchical structured P2P networks that we discussed and their source publications are the following: PHT [Ramabhadran et al. 2004], P-Grid [Aberer 2001, Aberer et al. 2003a], BATON [Jagadish et al. 2005], BATON* [Jagadish et al. 2006], VBI-tree [Jagadish et al. 2005], P-Tree [Crainiceanu et al. 2004], SkipNet [Harvey et al. 2003], and Skip Graph [Aspnes and Shah 2003]. Schmidt and Parashar [2004] describe a system that uses space-filling curves for defining structure, and Ganesan et al. [2004] propose one based on hyperrectangle structure.

Examples of superpeer networks include Edutella [Nejdl et al. 2003] and JXTA.

A good discussion of the issues of schema mapping in P2P systems can be found in [Tatarinov et al. 2003]. Pairwise schema mapping is used in Piazza [Tatarinov et al. 2003], LRM [Bernstein et al. 2002], Hyperion [Kementsietsidis et al. 2003],

and PGrid [Aberer et al. 2003b]. Mapping based on machine learning techniques is used in GLUE [Doan et al. 2003b]. Common agreement mapping is used in APPA [Akbarinia et al. 2006, Akbarinia and Martins 2007] and AutoMed [McBrien and Poulovassilis 2003]. Schema mapping using IR techniques is used in PeerDB [Ooi et al. 2003] and Edutella [Nejdl et al. 2003]. Semantic query reformulation using pairwise schema mappings in social P2P systems is addressed in [Bonifati et al. 2014].

An extensive survey of query processing in P2P systems is provided in [Akbarinia et al. 2007b] and has been the basis for writing Sections 9.2 and 9.3. 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. 2007a]. A more efficient algorithm than TA is the Best Position Algorithm [Akbarinia et al. 2007c]. Several TA-style algorithms have been proposed for distributed top-k query processing, e.g., TPUT[Cao and Wang 2004].

Top-k query processing in P2P systems has received much attention: in unstructured systems, e.g., PlanetP [Cuenca-Acuna et al. 2003] and APPA [Akbarinia et al. 2006]; in DHTs, e.g., APPA [Akbarinia et al. 2007a]; and in superpeer systems, e.g., Edutella [Balke et al. 2005]. Solutions to P2P join query processing are proposed in PIER [Huebsch et al. 2003]. Solutions to P2P range query processing are proposed in locality sensitive hashing [Gupta et al. 2003], PHT [Ramabhadran et al. 2004], and BATON [Jagadish et al. 2005].

The survey of replication in P2P systems by Martins et al. [2006b] has been the basis for Sect. 9.4. 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. 2007d]. Reconciliation of replicated data is addressed in OceanStore [Kubiatowicz et al. 2000], P-Grid [Aberer et al. 2003a], and APPA [Martins et al. 2006a, Martins and Pacitti 2006, Martins et al. 2008]. The action-constraint framework has been proposed for IceCube [Kermarrec et al. 2001].

P2P techniques have also received attention to help scaling up data management in the context of Grid Computing [Pacitti et al. 2007] or edge/mobile computing [Tang et al. 2019], or to help protecting data privacy in data analytics [Allard et al. 2015].

Blockchain is a relatively recent, polemical topic, featuring enthusiastic proponents [Ito et al. 2017] and strong opponents, e.g., famous economist N. Roubini [Roubini 2018]. The concepts are defined in the pioneering paper on the bitcoin blockchain [Nakamoto 2008]. Since then, many other blockchains for other cryptocurrencies have been proposed, e.g., Etherum and Ripple. Most of the initial contributions have been made by developers, outside the academic world. Thus, the main source of information is on web sites, white papers, and blogs. Academic research on blockchain has recently started. In 2016, Ledger, the first academic journal dedicated to various aspects (computer science, engineering, law,

economics, and philosophy) related to blockchain technology was launched. In the distributed system community, the focus has been on improving the security or performance of the protocols, e.g., Bitcoin-NG [Eyal et al. 2016]. In the data management community, we can find useful tutorials in major conferences, e.g., [Maiyya et al. 2018], survey papers, e.g., [Dinh et al. 2018], and system designs such as BigchainDB. Understanding the performance bottlenecks also requires benchmarking, as shown in BLOCKBENCH [Dinh et al. 2018].

Exercises

Problem 9.1 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 9.2 (**) 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 9.3 (*) Consider the unstructured P2P network in Fig. 9.4 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 neighbors.

Problem 9.4 (*) Consider Fig. 9.7, 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: trie, hypercube, and ring.

Problem 9.5 (**) 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 9.6 (**) 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 9.7 (**) Same question, but with the additional requirement that private data (e.g., user profile) must be stored at the user peer.

Problem 9.8 Discuss the commonalities and differences of schema mapping in multidatabase systems and P2P systems. In particular, compare the local-as-view approach presented in Chap. 7 with the pairwise schema mapping approach in Sect. 9.2.1.

Problem 9.9 (*) The FD algorithm for top-k query processing in unstructured P2P networks (see Algorithm 9.4) 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 9.10 (*) Apply the TPUT algorithm (Algorithm 9.2) to the three lists of the database in Fig. 9.10 with k=3. For each step of the algorithm, show the intermediate results.

Problem 9.11 (*) Same question applied to Algorithm DHTop (see Algorithm 9.5).

Problem 9.12 (*) Algorithm 9.6 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 9.6.

Problem 9.13 (*) 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 9.7. An alternative solution is to use a nonreplicated 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 9.7? What are the advantages and drawbacks of this approach, in terms of availability and load balancing?

Problem 9.14 (*) Discuss the commonalities and differences of public versus private (permissioned) blockchain. In particular, analyze the properties that need be provided by the transaction validation protocol.

Chapter 10
Big Data Processing

The past decade has seen an explosion of "data-intensive" or "data-centric" applications where the analysis of large volumes of heterogeneous data is the basis of solving problems. These are commonly known as *big data applications* and special systems have been investigated to support the management and processing of this data—commonly referred to as *big data processing systems*. These applications arise in many domains, from health sciences to social media to environmental studies and many others. Big data is a major aspect of data science, which combines various disciplines such as data management, data analysis and statistics, machine learning, and others to produce new knowledge from data. The more the data, the better the results of data science can be with the attendant challenges in managing and processing these data.

There is no precise definition of big data applications or systems, but they are typically characterized by the "four Vs" (although others have also been specified, such as value, validity, etc.):

1. **Volume.** The datasets that are used in these applications are very large, typically in the petabyte (PB; $10^{15} bytes$) range and with the growth of Internet-of-Things applications soon to reach zettabytes (ZB; $10^{21} bytes$). To put this in perspective, Google has reported that in 2016, user uploads to YouTube required 1PB of *new* storage capacity *per day*. They expect this to grow exponentially, with $10\times$ increase every five years (so by the time you read this book, their daily storage addition may be 10PB). Facebook stores about 250 billion images (as of 2018) requiring exabytes of storage. Alibaba has reported that during a heavy period in 2017, 320 PB of log data was generated in a six hour period as a result of customer purchase activity.
2. **Variety.** Traditional (usually meaning relational) DBMSs are designed to work on well-structured data—that is what the schema describes. In big data applications, this is no longer the case, and multimodal data has to be managed and processed. In addition to structured, the data may include images, text, audio, and video. It has been claimed that 90% of generated data today is unstructured. The

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_10

big data systems need to be able to manage and process all of these data types seamlessly.

3. **Velocity.** An important aspect of big data applications is that they sometimes deal with data that is arriving at the system at high-speed requiring systems to be able to process the data as they arrive. Following the examples we gave above for volume, Facebook has to process 900 million photos that users upload per day; Alibaba has reported that during a peak period, they had to process 470 million event logs per second. These numbers do not normally allow systems to store the data before processing, requiring real-time capabilities.
4. **Veracity.** The data used by big data applications comes from many sources, each of which may not be entirely reliable or trustworthy—there could be noise, bias, inconsistencies among the different copies and deliberate misinformation. This is commonly referred to as "dirty data" and it is unavoidable as the data sources grow along with the volume. It is claimed that dirty data costs upwards of $3 billion per year in US economy alone. Big data systems need to "clean" the data and maintain their provenance in order to reason about their trustworthiness. Another important dimension of veracity is "truthfulness" of the data to ensure that the data is not altered by noise, biases, or intentional manipulation. The fundamental point is that the data needs to be trustable.

These characteristics are quite different than the data that traditional DBMSs (which we have focused on up to this point) have to deal with—they require new systems, methodologies, and approaches. Perhaps it can be argued that parallel DBMSs (Chap. 8) handle volume reasonably well as there are very large datasets managed by these systems; however, the systems that can address **all** of the dimensions highlighted above require attention. These are topics of active research and development and our objective in this chapter and the next is to highlight the system infrastructure approaches that are currently being considered to address the first three points; veracity can be considered orthogonal to our discussion and is a complete topic in itself, and we will not consider it further. In the Bibliographic Notes, we will point to some of the literature in that area. Readers will recall that we briefly discussed it in Chap. 7 (specifically in Sect. 7.1.5); we will also address the issue in the context of web data management in Chap. 12 (specifically, Sect. 12.6.3).

Compared to traditional DBMSs, big data management uses a different software stack with the following layers (see Fig. 10.1). Big data management relies on a distributed storage layer, whereby data is typically stored in files or objects distributed over the nodes of a shared-nothing cluster. Data stored in distributed files is accessed directly by a data processing framework that enables programmers to express parallel processing code without an intervening DBMS. There could be scripting and declarative (SQL-like) querying tools on top of the data processing frameworks. For the management of multimodal data, typically NoSQL systems are deployed as part of the data access layer, or a streaming engine may be used, or even search engines can be employed. Finally, at the top various tools are provided that can be used to build more complex big data analytics, including machine learning (ML) tools. This software stack, as exemplified by Hadoop that we discuss shortly,

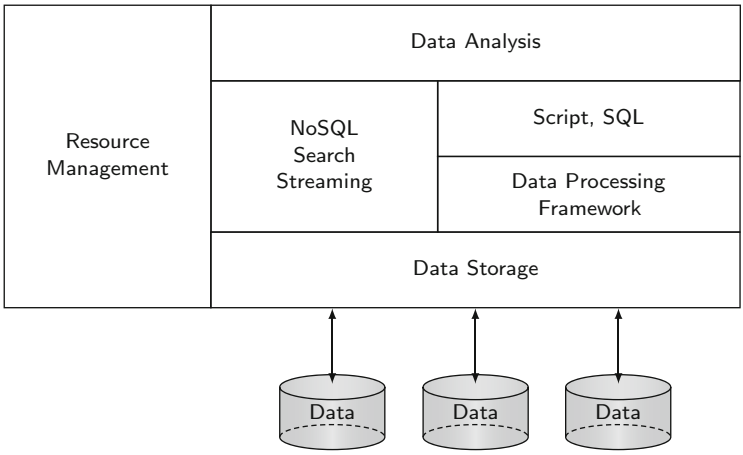


Fig. 10.1 Big data management software stack

fosters the integration of loosely-coupled (typically open source) components. For instance, a NoSQL DBMS typically supports different storage systems (e.g., HDFS, etc.). These systems are commonly deployed in a public or private cloud computing environment. This software stack architecture will guide our discussion in this and the following chapter.

The rest of this chapter is focused on components of this architecture going bottom-up, and, in the process, we address two of the V's that characterize big data systems. Section (10.1) focuses on distributed storage systems. Section 10.2, covers two important big data processing frameworks, focusing on MapReduce and Spark. Together with Sect. 10.1, this section addresses scalability concerns, i.e., the "volume" dimension. In Sect. 10.3 we discuss data processing for stream data—this addresses the "velocity" dimension. In Sect. 10.4 we cover graph systems focusing on graph analytics, addressing some of the "variety" issues. Variety issues are also addressed in Sect. 10.5 where we discuss the emerging field of data lakes. Data lakes integrate data from many sources that may or may not be structured. We leave the NoSQL side of this architecture to the next chapter (Chap. 11).

10.1 Distributed Storage Systems

Big data management relies on a distributed storage layer, whereby data is typically stored in files or objects distributed over the nodes of a shared-nothing cluster. This is one major difference with the software stack of current DBMSs that relies on block storage. The history of DBMSs is interesting to understand the evolution of this software stack. The very first DBMSs, based on the hierarchical or network models, were built as extensions of a file system, such as COBOL, with inter-file

links, and the first relational DBMSs too were built on top of a file system. For instance, the famous INGRES DBMS was implemented atop the Unix file system. But using a general-purpose file system was making data access quite inefficient, as the DBMS could have no control over data clustering on disk or cache management in main memory. The main criticism for this file-based approach was the lack of operating system support for database management (at that time). As a result, the architecture of relational DBMSs evolved from file-based to block-based, using a raw disk interface provided by the operating system. A block-based interface provides direct, efficient access to disk blocks (the unit of storage allocation on disks). Today all relational DBMSs are block-based, and thus have full control over disk management. The evolution towards parallel DBMSs kept the same approach, primarily to ease the transition from centralized systems. A primary reason for the return to the use of a file system is that the distributed storage can be made fault-tolerant and scalable, which makes it easier to build the upper data management layers.

Within this context, the distributed storage layer typically provides two solutions to store data, objects, or files, distributed over cluster nodes. These two solutions are complementary, as they have different purposes and can be combined.

Object storage manages data as objects. An object includes its data along with a variable amount of metadata, and a unique identifier (oid) in a flat object space. Thus, an object can be represented as a triple ⟨oid, data, metadata⟩, and once created, it can be directly accessed by its oid. The fact that data and metadata are bundled within each object makes it easy to move objects between distributed locations. Unlike in file systems where the type of metadata is the same for all files, objects can have variable amounts of metadata. This allows much user flexibility to express how objects are protected, how they can be replicated, when they can be deleted, etc. Using a flat object space allows managing massive amounts (e.g., billions or trillions) of unstructured data objects. Finally, objects can be easily accessed with a simple REST-based API with put and get commands easy to use on Internet protocols. Object stores are particularly useful to store a very high number of relatively small data objects, such as photos, mail attachments, etc. Therefore, this approach has been popular with most cloud providers who serve these applications.

File storage manages data within unstructured files (i.e., sequences of bytes) on top of which data can be organized as fixed-length or variable-length records. A file system organizes files in a directory hierarchy and maintains for each file its metadata (file name, folder position, owner, length of the content, creation time, last update time, access permissions, etc.), separate from the content of the file. Thus, the file metadata must first be read to locate the file's content. Because of such metadata management, file storage is appropriate for sharing files locally within a data center and when the number of files are limited (e.g., in the hundreds of thousands). To deal with big files that may contain high numbers of records, files need to be split and distributed on multiple cluster nodes, using a distributed file system. One of the most influential distributed file systems is Google File System (GFS). In the rest of this section, we describe GFS. We also discuss the combination of object storage and file storage, which is typically useful in the cloud.

10.1.1 Google File System

GFS has been developed by Google for its internal use and is used by many Google applications and systems, such as Bigtable.

Similar to other distributed file systems, GFS aims at providing performance, scalability, fault-tolerance, and availability. However, the targeted systems, shared-nothing clusters, are challenging as they are made of many (e.g., thousands of) servers built from inexpensive hardware. Thus, the probability that any server fails at a given time is high, which makes fault-tolerance difficult. GFS addresses this problem through replication and failover as we discuss later. It is also optimized for Google data-intensive applications such as search engine or data analysis. These applications have the following characteristics. First, their files are very large, typically several gigabytes, containing many objects such as web documents. Second, workloads consist mainly of read and append operations, while random updates are rare. Read operations consist of large reads of bulk data (e.g., 1 MB) and small random reads (e.g., a few KBs). The append operations are also large and there may be many concurrent clients that append the same file. Third, because workloads consist mainly of large read and append operations, high throughput is more important than low latency.

GFS organizes files as a trie of directories and identifies them by pathnames. It provides a file system interface with traditional file operations (create, open, read, write, close, and delete file) and two additional operations: snapshot, which allows creating a copy of a file or of a directory trie, and record append, which allows appending data (the "record") to a file by concurrent clients in an efficient way.

A record is appended atomically, i.e., as a continuous byte string, at a byte location determined by GFS. This avoids the need for distributed lock management that would be necessary with the traditional write operation (which could be used to append data).

The architecture of GFS is illustrated in Fig. 10.2. Files are divided into fixed-size partitions, called *chunks*, of large size, i.e., 64 MB. The cluster nodes consist of GFS clients that provide the GFS interface to applications, chunk servers that store chunks, and a single GFS master that maintains file metadata such as namespace,

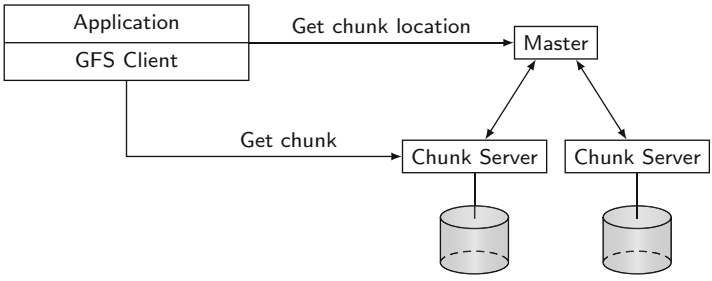


Fig. 10.2 GFS Architecture

access control information, and chunk placement information. Each chunk has a unique id assigned by the master at creation time and, for reliability reasons, is replicated on at least three chunk servers. To access chunk data, a client must first ask the master for the chunk locations, needed to answer the application file access. Then, using the information returned by the master, the client can request the chunk data to one of the replicas.

This architecture using single master is simple, and since the master is mostly used for locating chunks and does not hold chunk data, it is not a bottleneck. Furthermore, there is no data caching at either clients or chunk servers, since it would not benefit large reads. Another simplification is a relaxed consistency model for concurrent writes and record appends. Thus, the applications must deal with relaxed consistency using techniques such as checkpointing and writing self-validating records. Finally, to keep the system highly available in the face of frequent node failures, GFS relies on replication and automatic failover. Each chunk is replicated at several servers (by default, GFS stores three replicas). The master periodically sends each chunk server heartbeat messages. Then, upon a chunk server's failure, the master performs automatic failover, by redirecting all file accesses to an alive server that holds a replica. GFS also replicates all the master's data to a shadow master, so that in case of a master's failure, the shadow master automatically takes over.

There are open source implementations of GFS, such as Hadoop Distributed File System (HDFS), which we discuss in Sect. 10.2.1. There are other important open source distributed file systems for cluster systems, such as GlusterFS for shared-nothing and Global File System 2 (GFS2) for shared-disk, both being now developed by Red Hat for Linux.

10.1.2 Combining Object Storage and File Storage

An important trend is to combine object and file storage in a single system, in order to support both high numbers of objects and large files. The first system that combined object and file storage is Ceph. Ceph is an open source software storage platform, now developed by Red Hat in a shared-nothing cluster at exabyte scale. Ceph decouples data and metadata operations by eliminating file allocation tables and replacing them with data distribution functions designed for heterogeneous and dynamic clusters of unreliable object storage devices (OSDs). This allows Ceph to leverage the intelligence present in OSDs to distribute the complexity surrounding data access, update serialization, replication and reliability, failure detection, and recovery. Ceph and GlusterFS are now the two major storage platforms offered by Red Hat for shared-nothing clusters.

HDFS, on the other hand, has become the de facto standard for scalable and reliable file system management for big data. Thus, there is much incentive to add object storage capabilities to HDFS, in order to make data storage easier for cloud providers and users. In Azure HDInsight, Microsoft's Hadoop-based solution for

big data management in the cloud, HDFS is integrated with Azure Blob Storage, the object storage manager, to operate directly on structured or unstructured data. Blob storage containers store data as key-value pairs, and there is no directory hierarchy.

10.2 Big Data Processing Frameworks

An important class of big data applications requires data management without the overhead of full database management, and cloud services require scalability for applications that are easy to partition into a number of parallel but smaller tasks—the so-called embarrassingly parallelizable applications. For these cases where scalability is more important than declarative querying, transaction support, and database consistency, a parallel processing platform called MapReduce has been proposed. The fundamental idea is to simplify parallel processing using a distributed computing platform that offers only two interfaces: map and reduce. Programmers implement their own map and reduce functions, while the system is responsible for scheduling and synchronizing the map and reduce tasks. This architecture is further optimized in Spark, so much of the following discussion applies to both frameworks. We start discussing basic MapReduce (Sect. 10.2.1), and then introduce Spark optimizations (Sect. 10.2.2).

The commonly cited advantages of this type of processing framework are as follows:

1. **Flexibility**. Since the code for map and reduce functions is written by the user, there is considerable flexibility in specifying the exact processing that is required over the data rather than specifying it using SQL. Programmers can write simple map and reduce functions to process large volumes of data on many machines (or nodes, as is commonly used in parallel DBMSs)[1] without the knowledge of how to parallelize the processing of a MapReduce job.
2. **Scalability**. A major challenge in many existing applications is to be able to scale with increasing data volumes. In particular, in cloud applications *elastic scalability* is desired, which requires the system to be able to scale its performance up and down dynamically as the computation requirements change. Such a "pay-as-you-go" service model is now widely adopted by the cloud computing service providers, and MapReduce can support it seamlessly through data parallel execution.
3. **Efficiency**. MapReduce does not need to load data into a database, avoiding the high cost of data ingest. It is, therefore, very efficient for applications that require processing the data only once (or only a few times).

[1] In MapReduce literature, these are commonly referred as *workers*, while we use the term *node* in our discussions in the parallel DBMS chapter and the following chapter on NoSQL. The reader should note that we use the terms interchangeably.

4. **Fault-tolerance**. In MapReduce, each job is divided into many small tasks that are assigned to different machines. Failure of a task or a machine is compensated by assigning the task to a machine that is able to handle the load. The input of a job is stored in a distributed file system where multiple replicas are kept to ensure high availability. Thus, the failed map task can be repeated correctly by reloading the replica. The failed reduce task can also be repeated by repulling the data from the completed map tasks.

The criticisms of MapReduce center on its reduced functionality, requiring considerable amount of programming effort, and its unsuitability for certain types of applications (e.g., those that require iterative computations). MapReduce does not require the existence of a schema and does not provide a high-level language such as SQL. The flexibility advantage mentioned above comes at the expense of considerable (and usually sophisticated) programming on the part of the user. Consequently, a job that can be performed using relatively simple SQL commands may require considerable amount of programming in MapReduce, and this code is generally not reusable. Moreover, MapReduce does not have built-in indexing and query optimization support, always resorting to scans (this is highlighted both as an advantage and as a disadvantage depending on the viewpoint).

10.2.1 MapReduce Data Processing

As noted above, MapReduce is a simplified parallel data processing approach for execution on a computer cluster. It enables programmers to express in a simple, functional style their computations on large datasets and hides the details of parallel data processing, load balancing, and fault-tolerance. Its programming model consists of two user-defined functions, map() and reduce() with the following semantics:

map	$(k1, v1) \rightarrow list(k2, v2)$
reduce	$(k2, list(v2)) \rightarrow list(v3)$

The map function is applied to each record in the input dataset to compute zero or more intermediate (key,value) pairs. The reduce function is applied to all the values that share the same unique key in order to compute a combined result. Since they work on independent inputs, map and reduce can be automatically processed in parallel, on different data partitions using many cluster nodes.

Figure 10.3 gives an overview of MapReduce execution in a cluster. The inputs of the map function are a set of key/value pairs. When a MapReduce job is submitted to the system, the map tasks (which are processes that are referred to as *mappers*) are started on the compute nodes and each map task applies the map function to every key/value pair $(k1, v1)$ that is allocated to it. Zero or more intermediate

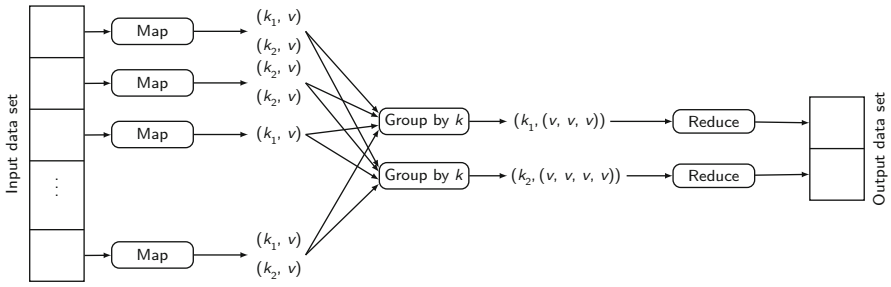


Fig. 10.3 Overview of MapReduce Execution

key/value pairs ($list(k2, v2)$) can be generated for the same input key/value pair. These intermediate results are stored in the local file system and sorted by the keys. After all the map tasks complete, the MapReduce engine notifies the reduce tasks (which are also processes that are referred to as *reducers*) to start their processing. The reducers will pull the output files from the map tasks in parallel, and merge-sort the files obtained from the map tasks to combine the key/value pairs into a set of new key/value pair ($k2, list(v2)$), where all values with the same key $k2$ are grouped into a list and used as the input for the reduce function—this is commonly known as the *shuffle* process, which is, in effect, a parallel sort. The reduce function applies the user-defined processing logic to process the data. The results, normally a list of values, are written back to the storage system.

In addition to writing the map and reduce functions, programmers can exert further control (e.g., input/output formats and partitioning function) by means of user-defined functions (UDFs) that these systems provide.

Example 10.1 Let us consider relation EMP(ENO, ENAME, TITLE, CITY) and the following SQL query that returns for each city, the number of employees whose name contains "Smith."

```
SELECT CITY, COUNT(*)
FROM   EMP
WHERE  ENAME LIKE "%Smith"
GROUP BY CITY
```

Processing this query with MapReduce can be done with the following Map and Reduce functions (which we give in pseudo code).

```
Map (Input: (TID,EMP), Output: (CITY,1))
    if EMP.ENAME like ``\%Smith'' return (CITY,1)
Reduce (Input: (CITY,list(1)), Output: (CITY, SUM(list(1)))
    return (CITY,SUM(1))
```

map is applied in parallel to every tuple in EMP. It takes one pair (TID,EMP), where the key is the EMP tuple identifier (TID) and the value being the EMP tuple, and,

if applicable, returns one pair (CITY,1). Note that the parsing of the tuple format to extract attributes needs to be done by the map function. Then all (CITY,1) pairs with the same CITY are grouped together and a pair (CITY,list(1)) is created for each CITY. reduce is then applied in parallel to compute the count for each CITY and produce the result of the query. ♦

10.2.1.1 MapReduce Architecture

In discussing specifics of MapReduce, we will focus on one particular implementation: Hadoop. The Hadoop stack is shown in Fig. 10.4, which is a particular realization of the big data architecture depicted in Fig. 10.1. Hadoop uses Hadoop Distributed File System (HDFS) as its storage, although it can be deployed on different storage systems. HDFS and the Hadoop processing engine are loosely connected; they can either share the same set of compute nodes, or be deployed on different nodes. In HDFS, two types of nodes are created: *name node* and *data node*. The name node records how data is partitioned, and monitors the status of data nodes in HDFS. Data imported into HDFS is split into equal-size chunks and the name node distributes the data chunks to different data nodes that store and manage the chunks assigned to them. The name node also acts as the dictionary server, providing partitioning information to applications that search for a specific chunk of data.

The decoupling of the Hadoop processing engine from the underlying storage system allows the processing and the storage layers to scale up and down independently as needed. In Sect. 10.1, we discussed different approaches to distributed storage system design and gave examples. Each data chunk that is stored at each machine in the cluster is an input to one mapper. Therefore, if the dataset is partitioned into k chunks, Hadoop will create k mappers to process the data (or vice versa).

Hadoop processing engine has two types of nodes, the *master* node and the *worker* nodes, as shown in Fig. 10.5. The master node controls the execution flow

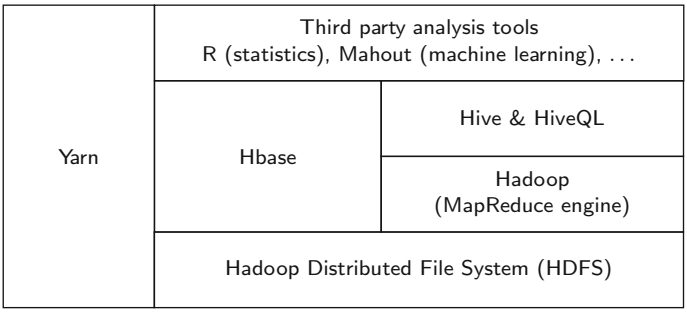


Fig. 10.4 Hadoop stack

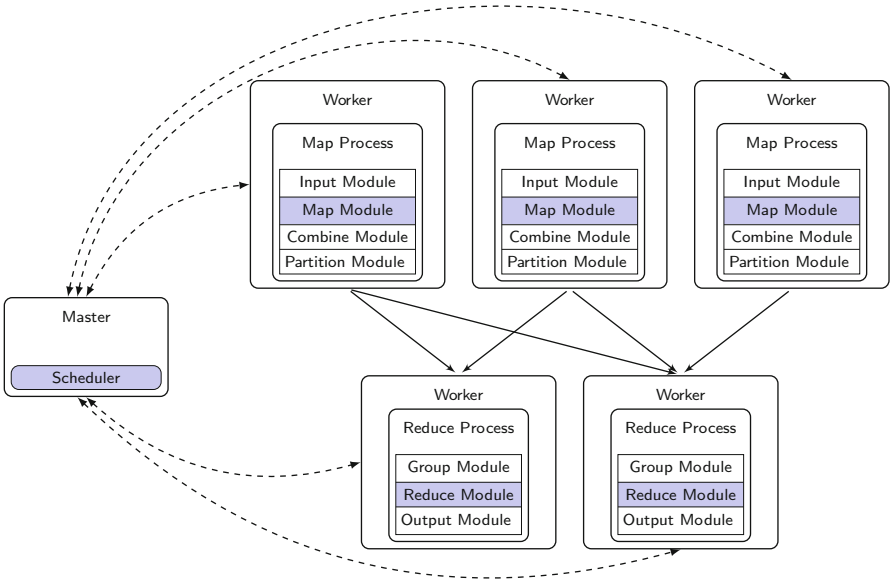


Fig. 10.5 Master-Worker Architecture of MapReduce

of the tasks at the worker nodes via the *scheduler* module (in Hadoop, this is known as the *job tracker*). Each worker node is responsible for a map or reduce task. The basic implementation of MapReduce engine needs to include the following modules the first three of which are essential modules; the remaining ones are extensions:

1. *Scheduler*. The *scheduler* is responsible for assigning the map and reduce tasks to the worker nodes based on data locality, network state, and other statistics of the worker nodes. It also controls fault-tolerance by rescheduling a failed process to other worker nodes (if possible). The design of the *scheduler* significantly affects the performance of the MapReduce system.
2. *Map module*. The *map module* scans a data chunk and invokes the user-defined map() function to process the input data. After generating the intermediate results (a set of key/value pairs), it groups the results based on the partition keys, sorts the tuples in each partition, and notifies the master node about the positions of the results.
3. *Reduce module*. The *reduce module* pulls data from the mappers after receiving the notification from the master. Once all intermediate results are obtained from the mappers, the reducer merges the data by keys and all values with the same key are grouped together. Finally, the user-defined function is applied to each key/value pair, and the results are output to distributed storage.
4. *Input and Output modules*. The *input module* is responsible for recognizing the input data with different input formats, and splitting the input data into key/value pairs. This module allows the processing engine to work with different storage systems by allowing different input formats to be used to parse different data

sources, such as text files, binary files, and even database files. The *output module* similarly specifies the output format of mappers and reducers.

5. *Combine module*. The purpose of this module is to reduce the shuffling cost by performing a local reduce process for the key/value pairs generated by the mapper.
6. *Partition module*. This is used to specify how to shuffle the key/value pairs from mappers to reducers. The default partition function is defined as $f(key) = h(key)\%numOfReducer$, where % indicates the mod operator and $h(key)$ is the hash value of the key. A key/value pair (k, v) is sent to the $f(k)$-th reducer. Users can define different partition functions to support more sophisticated behavior.
7. *Group module*. *Group module* specifies how to merge the data received from different map processes into one sorted run in the reduce phase. By specifying the group function, which is a function of the map output key, the data can be merged more flexibly. For example, if the map output key is a composition of several attributes (sourceIP,destURL), the group function can only compare a subset of the attributes (sourceIP). As a result, in the reducer module, the reduce function is applied to the key/value pairs with the same sourceIP.

Given its stated purpose of scaling over a large number of processing nodes, a MapReduce system needs to support fault-tolerance efficiently. When a map or reduce task fails, another task on a different machine is created to reexecute the failed task. Since the mapper stores the results locally, even a completed map task needs to be reexecuted in case of a node failure. In contrast, since the reducer stores the results in the distributed storage, a completed reduce task does not need to be reexecuted when a node failure occurs.

10.2.1.2 High-Level Languages for MapReduce

The design philosophy of MapReduce is to provide a flexible framework that can be exploited to solve different problems. Therefore, MapReduce does not provide a query language, expecting the users to implement their customized `map()` and `reduce()` functions. While this provides considerate flexibility, it adds to the complexity of application development. To make MapReduce easier to use, a number of high-level languages have been developed, some of which are declarative (HiveQL, Tenzing, JAQL), others are data flow languages (Pig Latin), procedural languages (Sawzall), Java library (FlumeJava), and still others are declarative machine learning languages (SystemML). From a database system perspective, perhaps the declarative languages are of more interest. Although these languages are different, they generally follow a similar architecture, as shown in Fig. 10.6. The upper level consists of multiple query interfaces such as command line interface, web interface, or JDBC/ODBC server. Currently, only Hive supports all these query interfaces. After a query is issued from one of the interfaces, the query compiler parses this query to generate a logical plan using the metadata. Then, the rule

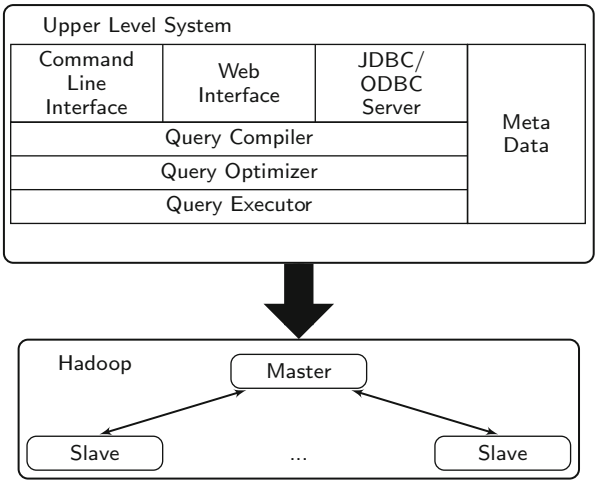


Fig. 10.6 Architecture of Declarative Query Implementations

based optimization, such as pushing projection down, is applied to optimize the logical plan. Finally, the plan is transformed into a directed acyclic graph (DAG) of MapReduce jobs, which are subsequently submitted to the execution engine one-by-one.

10.2.1.3 MapReduce Implementation of Database Operators

If MapReduce implementations such as Hadoop are to be used for data management going beyond the "embarrassingly parallelizable" applications, it is important to implement typical database operators in these systems, and this has been the subject of some research. Simple operators such as select and project can be easily supported in the map function, while complex ones, such as theta-join, equijoin, multiway join require significant effort. In this section, we discuss these implementations.

The projection and selection can be easily implemented by adding a few conditions to the map function to filter the unnecessary columns and tuples. The implementation of aggregation can be easily achieved using the map() and reduce() functions; Fig. 10.7 illustrates the data flow of the MapReduce job for the aggregation operator. The mapper extracts an aggregation key (Aid) for each incoming tuple (transformed into key/value pair). The tuples with the same aggregation key are shuffled to the same reducers, and the aggregation function (e.g., sum, min) is applied to these tuples.

Join operator implementations have attracted by far the most attention, as it is one of the more expensive operators, and a better implementation may potentially lead to

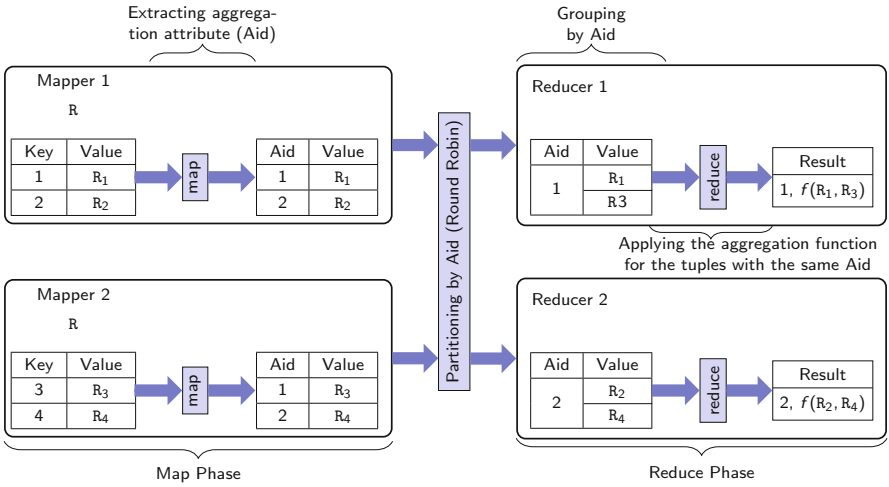


Fig. 10.7 Data flow of aggregation

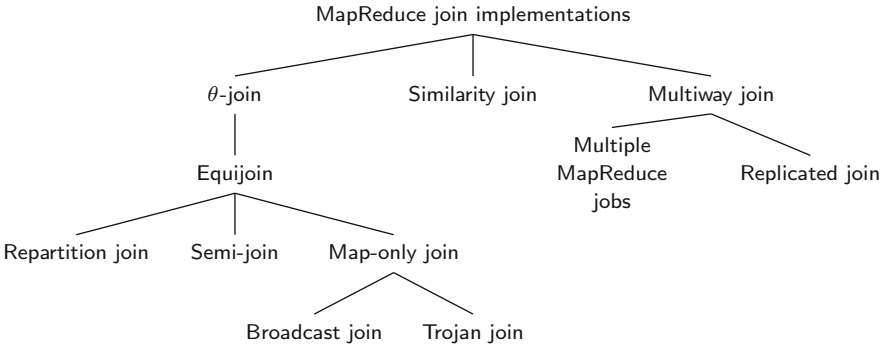


Fig. 10.8 Join implementations on MapReduce

significant performance improvement. The existing join algorithms are summarized in Fig. 10.8. We will describe theta-join and equijoin implementations as examples.

Recall that theta-join (θ-join) is a join operator where the join condition θ is one of $\{<, \leq, =, \geq, >, \neq\}$. A binary (natural) join of relations R(A, B) and S(B, C) can be performed using MapReduce in the following manner. Relation R is partitioned and each partition is assigned to a set of mappers. Each mapper takes tuples $\langle a, b\rangle$ and converts them to a list of key/value pairs of the form $(b, \langle a, R\rangle)$, where the key is the join attribute and the value includes the relation name R. These key/value pairs are shuffled and sent to the reducers so that all pairs with the same join key value are collected at the same reducer. The same process is applied to S. Each reducer then joins tuples of R with tuples of S (the inclusion of relation name in the value ensures that tuples of R or S are not joined with each other).

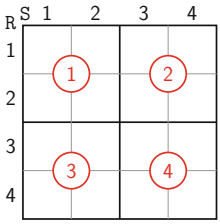


Fig. 10.9 Matrix-to-Reducer Mapping for Cross Product

To efficiently implement theta-join on MapReduce, the $|\mathtt{R}| \times |\mathtt{S}|$ tuples should be evenly distributed on the R reducers, so that each reducer generates about the same number of results: $\frac{|\mathtt{R}| \times |\mathtt{S}|}{r}$. 1-Bucket-Theta algorithm achieves this by evenly partitioning the join matrix into buckets (Fig. 10.9) and assigning each bucket to only one reducer to eliminate duplicate computation. This algorithm, at the same time, ensures that all the reducers are assigned the same number of buckets to balance the load. In Fig. 10.9, both tables R and S are evenly partitioned into 4 parts, resulting in a matrix with 16 buckets that are grouped into 4 regions. Each region is assigned to a reducer.

Figure 10.10 illustrates the data flow of the theta-join when θ equals "$\neq$" for the case depicted in Fig. 10.9. The map and reduce phases are implemented as follows:

1. *Map*. On the map side, for each tuple from R or S, a row id or column id (call it *Bid*) between 1 and the number of regions (4 in the above example) is randomly selected as the map output key, and the tuple is concatenated with a tag indicating the origin of the tuple as the map output value. The *Bid* specifies which row or column in the matrix (of Fig. 10.9) the tuple belongs to, and the output tuples of the map() function are shuffled to all the reducers (each reducer corresponds to one region) that intersect with the row or column.
2. *Reduce*. On the reduce side, the tuples from the same table are grouped together based on the tags. The local theta-join computation is then applied to the two partitions. The qualified results ($\mathtt{R}.key \neq \mathtt{S}.key$) are output to storage. Since each bucket is assigned to only one reducer, no redundant results are generated.

   In Fig. 10.9 there are 16 buckets organized into 4 regions; there are 4 reducers in Fig. 10.10, each responsible for one region. Since Reducer 1 is in charge of region 1, all R tuples where Bid = 1 or 2 and S tuples with Bid = 1 or 2 are sent to it. Similarly, Reducer 2 gets R tuples with Bid = 1 or 2 and S tuples with Bid = 3 or 4. Each reducer partitions the tuples it receives into two parts based on the origins, and joins these parts.

Let us now consider equijoin, which is a special case of θ-join where θ is "=". There are three variations of equijoin implementations: repartition join, semijoin-based join, and map-only join. We discuss repartition join in some detail below. Semijoin-based implementation consists of three MapReduce jobs: The first is a full

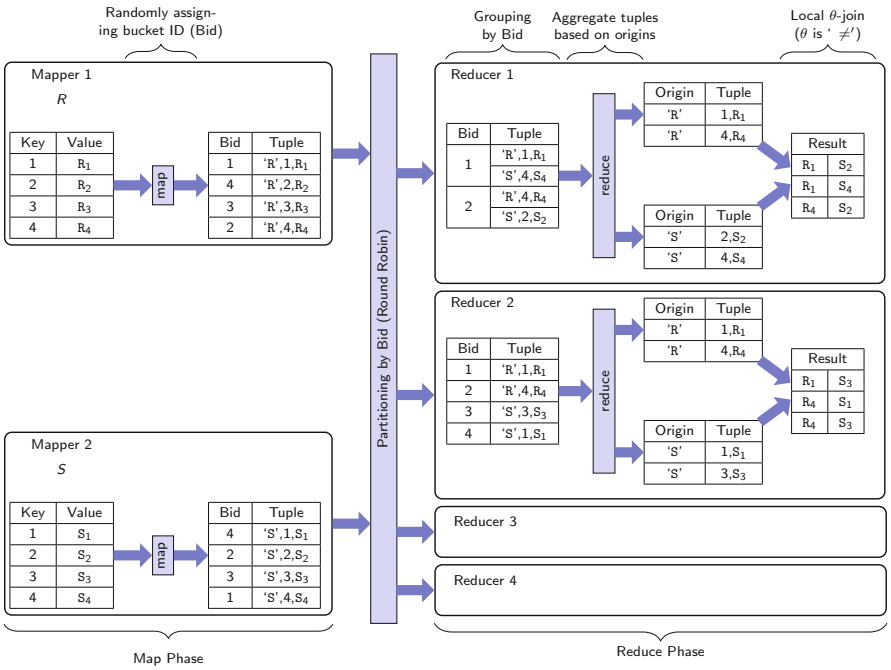


Fig. 10.10 Data flow of theta-join (theta equals "≠")

MapReduce job that extracts the unique join keys from one of the relations, say R, where the map task extracts the join key of each tuple and shuffles the identical keys to the same reducer, and the reduce task eliminates the duplicate keys and stores the results in DFS as a set of files $(u_0, u_1, \ldots, u_k)$. The second job is a map-only job that produces the semijoin results $S' = S \ltimes R$. In this job, since the files that store the unique keys of R are small, they are broadcast to each mapper and locally joined with the part of S (called *data chunk*) assigned to that mapper. The third job is also a map-only job where S' is broadcast to all the mappers and locally joined with R. Map-only join requires only map side processing. If the inner relation is much smaller than the outer relation, then shuffling can be avoided (as proposed in broadcast join) by using a map task similar to the third job of semijoin-based algorithm. Assuming S is the inner and R is the outer relation, each mapper loads the full S table to build an in-memory hash and scans its assigned data chunk of R (i.e., R_i). The local hash join is performed between S and R_i.

Repartition join is the default join algorithm for MapReduce in Hadoop. The two tables are partitioned in the map phase, followed by shuffling the tuples with the same key to the same reducer that joins the tuples. As shown in Fig. 10.11, repartition join can be implemented as one MapReduce job.

1. *Map*. Two types of mappers are created in the map phase, each of which is responsible for processing one of the tables. For each tuple of the table, the

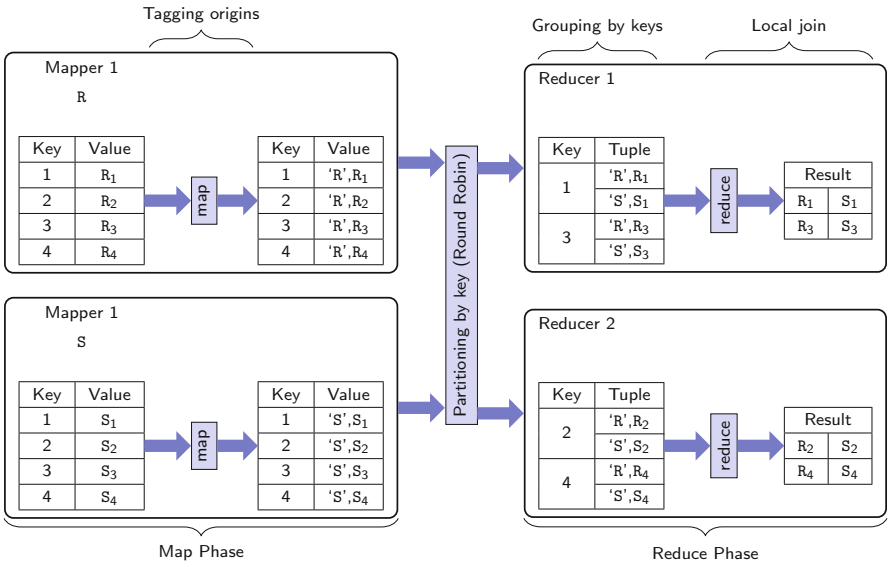


Fig. 10.11 Data flow of repartition join

mapper outputs a key/value pair $(k, \langle t, v \rangle)$, where k is the join attribute value, v is the entire tuple, and t is the tag indicating the source relation of the key/value pair. More specifically, the map phase consists of the following steps:

(a) Scanning the data from HDFS and generating the key/value pair.
(b) Sorting the map output (i.e., set of key/value pairs). On the map side, the output of each mapper needs to be sorted before being shuffled to the reducers.

2. *Shuffle*. After the map tasks are finished, the generated data is shuffled to the reduce tasks.
3. *Reduce*. The reduce phase includes the following steps:

(a) Merge. Each reducer merges the data that it receives using the sort-merge algorithm. Assume that the memory is sufficient for processing all sorted runs together. Then the reducer only needs to read and write data into local file systems once.
(b) Join. After the sorted runs are merged, the reducer needs two phases to complete the join. First, the tuples with the same key are split into two parts based on the tag indicating its source relation. Second, the two parts are joined locally. Assuming that the number of tuples for the same key are small and can fit in memory, this step only needs to scan the sorted run once.
(c) Write to HDFS. Finally, the results generated by the reducer should be written back to the HDFS.

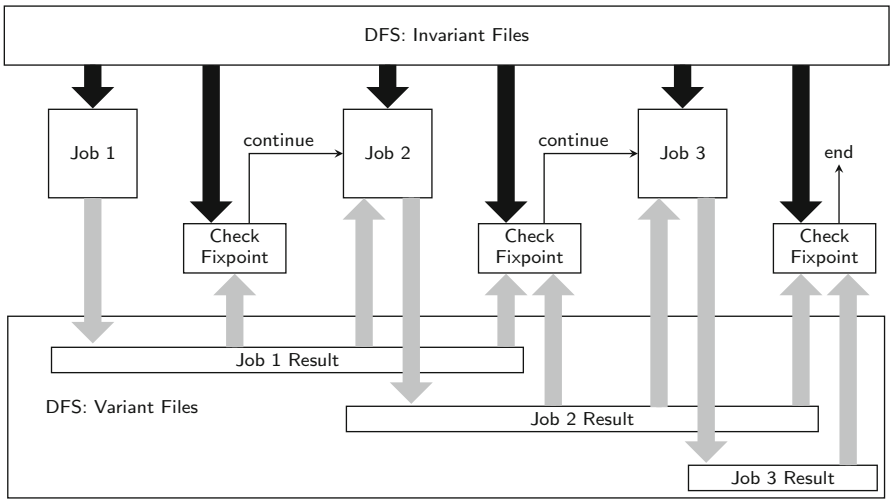


Fig. 10.12 MapReduce Processing for Iterative Computation

10.2.2 Data Processing Using Spark

Basic MapReduce, as discussed in the previous section, is not well suited for a class of data-intensive applications that are characterized by iterative computation requiring a chain of (i.e., multiple) MapReduce jobs (e.g., data mining) or online aggregation. In this section, we discuss an important extension of MapReduce to deal with this class of applications—this extension is the Spark system. We first start by discussing how iterative computing can be performed in a basic MapReduce system and why this is problematic.

Figure 10.12 shows an iterative job with three iterations that have two features: (1) the data source of each iteration consists of a variant part and an invariant part—the variant part consists of the files generated from the previous MapReduce jobs (the gray arrows in Fig. 10.12), and the invariant part is the original input file (the black arrows in Fig. 10.12); (2) a progress check might be needed at the end of each iteration to detect whether a fixpoint has been reached. The fixpoint has different meanings in different applications; in the k-means clustering algorithm that we discuss in Example 10.2, it may reflect whether the within-cluster sum-of-squares is minimized, or in the PageRank computation discussed in Example 10.4 it might reflect that the rank computation of each vertex has converged. This figure identifies three important issues in using MapReduce for these types of tasks. The first is that after each job (i.e., iteration), the intermediate results have to be written to the distributed file system (e.g., HDFS) and read again at the start of the next job (iteration). The second point is that there are no guarantees for subsequent jobs to be assigned to the same machines. Consequently, invariant data that do not change between iterations cannot be kept at the worker nodes and may have to be reread.

The third point is that an additional job is needed at the end of each iteration to compare the results generated between the current job and the previous one (i.e., check for convergence). All of these have high overhead, making it inefficient to use MapReduce for these applications. There have been a number of approaches to address these problems, some of which are task-specific, such as graph analytics that we discuss in the next section, while others, such as Spark, are more general.

An example of a workload that is problematic in MapReduce is the k-means clustering algorithm that is used quite frequently in big data analysis. We present this workload in Example 10.2 and discuss the difficulties in accomplishing it using MapReduce.

Example 10.2 The k-means algorithm takes a set X of values and partitions them in k clusters by placing each value $x_i \in X$ in cluster C_j whose centroid has the lowest distance to x_i. The centroid of a cluster is the mean of the values in that cluster. The distance computation is the within-cluster-sum-of-squares, i.e., $\sum_{j=1}^{k} \sum_{x_i \in C_j} (x_i - \mu_j)^2$ where μ_j is the centroid of cluster C_j. So, we are trying to find the allocation that minimizes this function for each x_i.

The standard k-means algorithm takes as input the value set $X = \{x_1, x_2, \ldots, x_r\}$, and an initial set of centroids $M = \{\mu_1, \mu_2, \ldots, \mu_m\}$ (usually $r \gg m$) and iteratively performs the following three steps:

1. Compute the distance of each $x_i \in X$ to every centroid $\mu_j \in M$ and assign x_i to cluster C_z if μ_z minimizes the above function.
2. Calculate a new set of centroids M according to the new value assignment to clusters.
3. For each of the clusters in C, check if the new and old centroid values are the same; if they are, convergence has been reached and the algorithms stops. Otherwise another iteration is needed with the new centroid values.

The implementation of this algorithm in MapReduce is straightforward: the first step is performed during map phase with each worker (mapper) performing the computation on a subset of X, while the second step is performed during reduce phase. The third step, checking for convergence, is another job as discussed above. One thing to note is that all of the mappers need the full set of centroids M; therefore, the convergence-checking job (step 3) needs to broadcast the new centroids if convergence has not been reached.

The problems with implementing iterative jobs using MapReduce are exhibited in this example: the results of computation at the end of each iteration (i.e., the newly computed centroids M and the current configuration of the clusters C) have to be written to HDFS so that they can be read by the mappers and reducers in the next iteration; since the assignment of the subsequent iteration job can go to any machine, the invariant data (i.e., X) has to be repartitioned and read again; and there is an extra convergence-check job at the end of each iteration. ♦

Spark addresses this shortcoming of MapReduce by providing an abstraction for sharing data across multiple stages of an iterative computation. The abstraction is

called *resilient distributed dataset* (RDD). It accomplishes efficient sharing in two ways: the first is that it ensures that the partitions that are assigned to each worker node are maintained between iterations to avoid shuffling data; the second is that it avoids writing and reading from HDFS in between iteration jobs by keeping the RDDs in memory—since the assignment to workers is maintained from one iteration to the next, this is feasible.

An RDD is a data structure that users can create, decide how it is partitioned among the worker nodes of a cluster, and explicitly decide whether it is stored on disk or kept in memory. If it is kept in memory, it serves as the working set cache of the application. An RDD is immutable (i.e., read-only) collection of data records; performing an update over an RDD is accomplished by a *transformation* (e.g., map(), filter(), groupByKey()) that results in the generation of a new RDD. Thus, an RDD can be created either from the data read from the file system or from another RDD through a transformation.

Example 10.3 Let us consider the implementation of k-means clustering (Example 10.2) in Spark. We will not give the full algorithm here[2] but will highlight how it addresses the issues:

1. Create an RDD for the invariant data (set X), and cache it in memory to bypass the I/O that has to be performed in between each iteration.
2. Create an RDD for variant data (chosen centroids M).
3. Compute the distance between $x_i \in X$ and each $\mu_j \in M$; save these distances as an RDD D.
4. Create a new RDD that includes each x_i and the μ_j with minimum distance from D.
5. Create an RDD M_{new} that includes the mean value of x_i assigned to each μ_j.
6. Compare M and M_{new}, and decide if convergence has been achieved (check fixpoint).
7. If not converged yet, then $M \leftarrow M_{new}$. There is no need to reload invariant data again, Spark can proceed to step 10.3.

♦

An important aspect of an RDD is whether it persists across iterations (or MapReduce jobs). If this is desired, then one of the two transforms have to be applied to the RDD: cache or persist. If it is desired for the RDD to remain in main memory across jobs, then cache is used; if flexibility is needed in specifying the "storage level" (e.g., disk only, disk and memory, etc.), then persist is used with the appropriate options with the default being persistence in memory.

Computation of RDDs is done lazily, when the program requires an *action*. Actions (e.g., collect(), count()) are different than transforms in that they materialize

[2]The full implementation of a variant of the algorithm we discussed above can be found at https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/clustering/KMeans.scala (accessed January 2018).

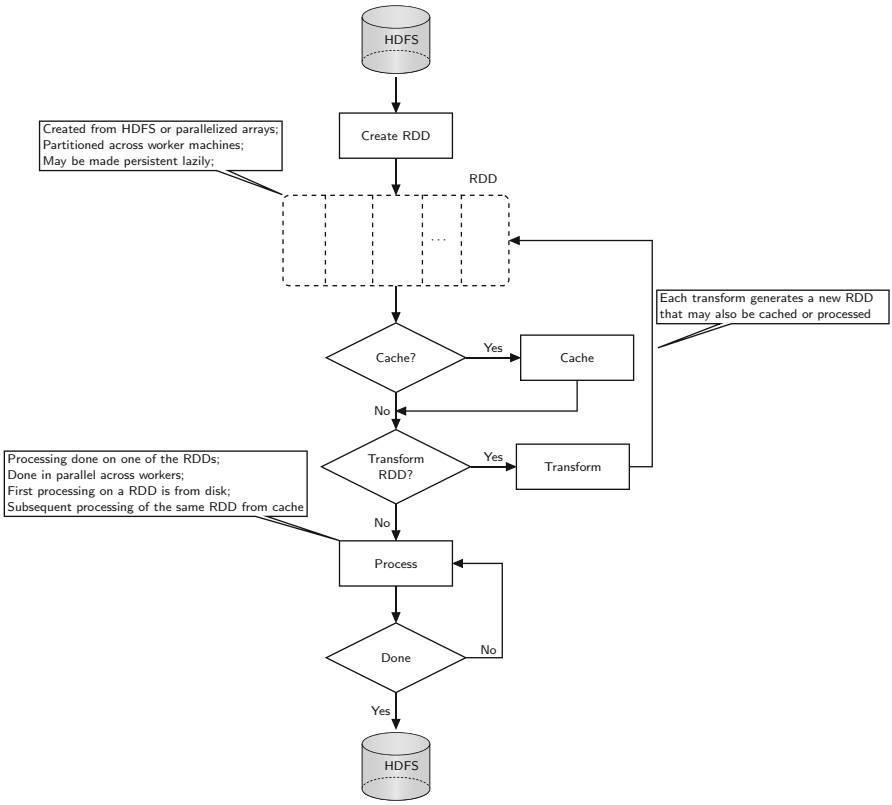


Fig. 10.13 Spark Program Flow

the RDD when the first action is performed, and the specified action is executed on the RDD at all the nodes where it is partitioned. We will discuss the execution aspect later.

Let us now look at the workflow of a Spark program execution—this is depicted in Fig. 10.13. The first thing that the program does is to create an RDD from the raw data on HDFS. Then, based on the user decision as to whether to cache/persist the RDD, the system takes appropriate preparations. Then, there may be additional transforms to generate other RDDs and, for each, cache/persist decision is specified. Finally, the processing starts with the actions indicated in the program. As noted above, the first action on an RDD materializes it, and then applies the action. The processing iterates over multiple actions and jobs.

Let us now discuss Spark support for executing programs written using the RDD concept. Spark expects to have a controller machine that executes the driver software. The driver generates the RDDs indicated in the program, and upon the first action on an RDD, materializes it, partitions it across the worker nodes, and then executes the action on the workers. The controller corresponds to the master

node in MapReduce while the driver performs the schedule function. Based on the cache/persist decision for each RDD, the driver instructs the workers to take appropriate action. When the workers indicate that the execution of the action is completed, the driver initiates the subsequent action. There are usual optimizations with respect to how RDDs are managed, how to deal with straggler workers, etc., but these are outside our scope.

Spark adds to standard MapReduce fault-tolerance by maintaining the lineage of the RDDs. In other words, it maintains a graph of how each RDD is generated from other RDDs. The lineage is constructed as an object and stored persistently for recovery. When a failure occurs and an RDD is lost, it can be recomputed based on the lineage. Furthermore, as discussed below, each RDD is partitioned across worker machines, so it is likely that the loss is restricted to some partitions of an RDD, and the recomputation can be restricted to those.

An important objective of Spark is to implement the reference architecture we discussed in a uniform way but providing a common ecosystem. This has resulted in the development of a relational DBMS on top of Spark (Spark SQL), a data stream system (Spark Streaming), and a graph processing system (GraphX). We discuss Spark Stream and GraphX in subsequent sections.

10.3 Stream Data Management

The traditional data management systems that we have been considering until now consist of a set of unordered objects that are relatively static, with insertions, updates, and deletions occurring less frequently than queries. They are sometimes called *snapshot databases* since they show a snapshot of the values of data objects at a given point in time.[3] Queries over these systems are executed when posed and the answer reflects the current state of the database. The typical paradigm is executing *transient* queries over *persistent* data.

A class of applications has emerged that does not fit this data model and querying paradigm. These include, among others, sensor networks, network traffic analysis, Internet-of-Things (IoT), financial tickers, on-line shopping and auctions, and applications that analyze transaction logs (such as web usage logs and telephone call records). In these applications, data is generated in real time, taking the form of an unbounded sequence (stream) of values. These are referred to as the *data stream* applications. In this section, we discuss systems that support these applications. Data stream applications are reflective of the velocity characteristic of big data.

Systems that process data streams usually come in two flavors: *data stream management systems* (DSMSs) that provide the functionalities of a typical DBMS

[3]Recall from our earlier discussion that data warehouses typically store historical data to allow analysis over time. Most systems we have been considering are OLTP ones, which deal with snapshots.

including a query language (declarative or data flow-based), and *data stream processing systems* (DSPSs) that do not claim to embody full DBMS functionality. Early systems were typically DSMSs, some of which had declarative languages (e.g., STREAM, Gigascope, TelegraphCQ) whereas others (e.g., Aurora and its distributed version Borealis) had a data flow language. More recent ones are typically in the DSPS class (e.g., Apache Storm, Heron, Spark Streaming, Flink, MillWheel, TimeStream). Many of the early DSMSs were single machine systems (except Borealis), while the more recent DSPSs are all distributed/parallel systems.

A fundamental assumption of the data stream model is that new data is generated continually and in a fixed order, although the arrival rates may vary across applications from millions of items per second (e.g., Internet traffic monitoring) down to several items per hour (e.g., temperature and humidity readings from a weather monitoring station). The ordering of streaming data may be implicit (by arrival time at the processing site) or explicit (by generation time, as indicated by a *timestamp* appended to each data item by the source). As a result of these assumptions, data stream systems (DSSs)[4] face the following requirements.

1. Much of the computation performed by a DSS is push-based, or data-driven. Newly arrived stream items are continually (or periodically) pushed into the system for processing. On the other hand, a traditional DBMS employs a mostly pull-based, or query-driven computation model, where processing is initiated when a query is posed.
2. As a consequence of the above, DSS queries and workloads are usually *persistent* (also referred to as *continuous*, *long-running*, or *standing* queries) in that they are issued once, but remain active in the system for possibly a long period of time. This means that a stream of updated results must be produced over time. These systems may, of course, accept and run transient ad-hoc queries as a traditional DBMS, but the persistent queries are their identifying characteristic.
3. A data stream is assumed to have unbounded, or at least unknown, length. Therefore, it is not possible to follow the usual approach and store the data completely before executing the queries; queries need to be executed as the data arrives at the system. Some systems employ *continuous processing model* where each new data item is processed as soon as it arrives in the system (e.g., Apache Storm, Heron). Others employ *windowed processing model* where incoming data items are batched and processed as a batch (e.g., , STREAM, Spark Streaming). From the user's point of view, recently arrived data may be more interesting and useful, leading to window definitions at the application level. Systems that follow a continuous processing model may (and usually do) provide windowing in their API. Therefore, from a user's perspective, they do both. Systems may also implement windows internally to overcome blocking operations as we discuss shortly.

[4]We will use this more general term when the separation between DSMS and DSPS is not important for the discussion.

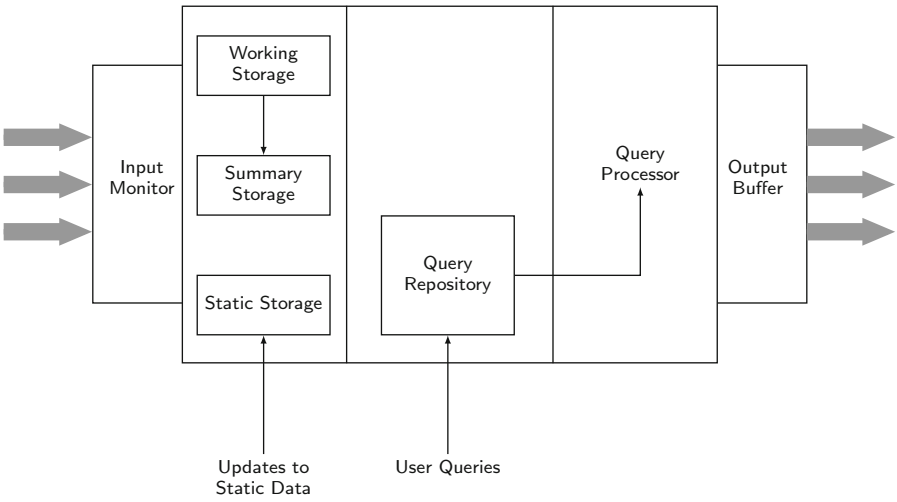


Fig. 10.14 Abstract reference architecture for a data stream management system

4. The system conditions may not be stable during the *lifetime* of a persistent query. For example, the stream arrival rates may fluctuate and the query workload may change.

An abstract single-node reference architecture for DSSs is shown in Fig. 10.14. Data arrives from one or more external sources. An input monitor regulates the input rates, perhaps by dropping items if the system is unable to keep up data is typically stored in three partitions: temporary working storage (e.g., for window queries that will be discussed shortly), summary storage for stream synopses (which is optional since some systems do not expose stream state to applications and therefore do not need this), and static storage for metadata (e.g., physical location of each source). Long-running queries are registered in the query repository and placed into groups for shared processing, though one-time queries over the current state of the stream may also be posed. The query processor communicates with the input monitor and may reoptimize the query plans in response to changing input rates. Results are streamed to the users or temporarily buffered. Users may then refine their queries based on the latest results. In a distributed/parallel DSS, this architecture would be replicated at each node and additional components are added for communication and distributed data management.

10.3.1 Stream Models, Languages, and Operators

We now focus on the fundamental model issues of stream systems. There is rich literature on this subject that we will point to in the Bibliographic Notes; our

objective at this point is to highlight and explain the fundamental concepts to understand the subsequent discussion.

10.3.1.1 Data Models

A data stream is an append-only sequence of timestamped items that arrive in some order. While this is the commonly accepted definition, there are more relaxed versions; for example, *revision tuples*, which are understood to replace previously reported (presumably erroneous) data, may be considered so that the sequence is not append-only. In publish/subscribe systems, where data is produced by some sources and consumed by those who subscribe to those data feeds, a data stream may be thought of as a sequence of events that are being reported continually. Since items may arrive in bursts, a stream may instead be modeled as a sequence of sets (or bags) of elements, with each set storing elements that have arrived during the same unit of time (no order is specified among data items that have arrived at the same time). In relation-based stream models (e.g., STREAM), individual items take the form of relational tuples such that all tuples arriving on the same stream have the same schema. In object-based models (e.g., COUGAR and Tribeca), sources and item types may be instantiations of (hierarchical) data types with associated methods. In more recent systems such as Apache Storm, Spark Streaming and others, data items can be any application-specific data—so, sometimes the generic term *payload* is used. Stream items may contain explicit source-assigned timestamps or implicit timestamps assigned by the DSMS upon arrival, so each data item is a tuple ⟨timestamp, payload⟩. In either case, the timestamp attribute may or may not be part of the stream schema, and therefore may or may not be visible to users. Stream items may arrive out-of-order (if explicit timestamps are used) and/or in preprocessed form. For instance, rather than propagating the header of each IP packet, one value (or several partially preaggregated values) may be produced to summarize the length of a connection between two IP addresses and the number of bytes transmitted.

A number of different classifications of window models have been defined, but two criteria are the most important and prevalent:

1. *Direction of movement of the endpoints:* Two fixed endpoints define a *fixed window*, two sliding endpoints (either forward or backward, replacing old items as new items arrive) define a *sliding window*, and one fixed endpoint and one moving endpoint (forward or backward) define a *landmark window*,.
2. *Definition of window size:* Logical or *time-based* windows are defined in terms of a time interval, whereas physical (also known as *count-based*) windows are defined in terms of the number of data items. Moreover, *partitioned windows* may be defined by splitting a window into groups and defining a separate count-based window on each group. The most general type is a *predicate window*, in which an arbitrary predicate specifies the contents of the window; e.g., all the packets from TCP connections that are currently open. A predicate window is

analogous to a materialized view and are also called *session windows* or *user-defined windows*.

Using this classification, the more important window models are time-based and count-based sliding windows. These have attracted the most attention and most of our discussions will focus on them.

10.3.1.2 Stream Query Models and Languages

An important issue is what the semantics of persistent (continuous) queries are, i.e., how do they generate answers. Persistent queries may be monotonic or non-monotonic. A *monotonic query* is one whose results can be updated incrementally. That is, it is sufficient to reevaluate the query over newly arrived items and append qualifying tuples to the result. Consequently, the answer of a monotonic persistent query is a continuous, append-only stream of results. Optionally, the output may be periodically updated by appending a batch of new results. *Non-monotonic queries* may produce results that cease to be valid as new data is added and existing data changed (or deleted). Consequently, they may need to be recomputed from scratch during every reevaluation.

As noted earlier, DSMSs provide a query language for access. Two fundamental querying paradigms can be identified: declarative and procedural. *Declarative languages* have SQL-like syntax, but stream-specific semantics. The languages in this class include CQL, GSQL, and StreaQuel. *Procedural languages* construct queries by defining an acyclic graph of operators (e.g., Aurora).

Languages that support windowed execution provide two language primitives: `size` and `slide`. The first specifies the length of the window and the second specifies how frequently the window moves. For example, for a time-based sliding window query, `size=10min, slide=5sec` would mean that we are interested in operating on data in a window that is 10 minutes long, and the window "moves" every 5 seconds. These have an impact on the way the content of the window is managed and we discuss the issue in Sect. 10.3.2.1.

10.3.1.3 Streaming Operators and Their Implementation

The applications that generate data streams also have similarities in the type of operations they perform. We list below a set of fundamental operations over streaming data.

- **Selection:** All streaming applications require support for complex filtering.
- **Complex aggregation:** Complex aggregates, including nested aggregates (e.g., comparing a minimum with a running average), frequent item queries, etc. are needed to compute trends in the data.

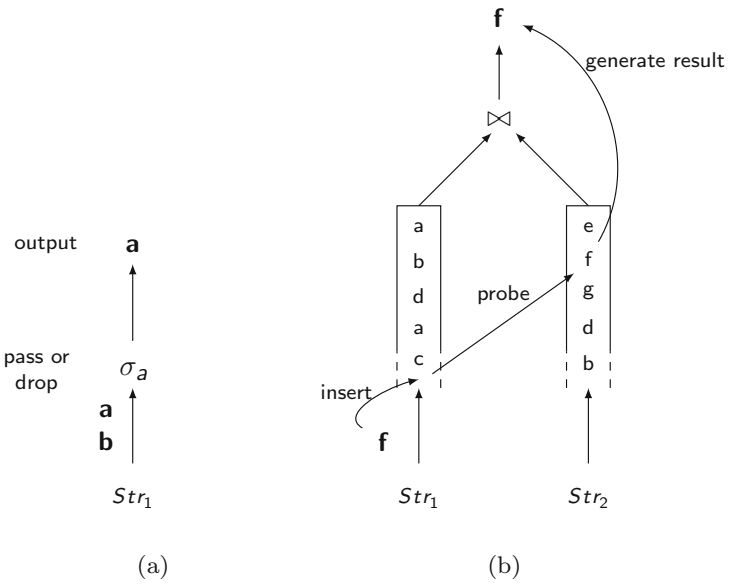


Fig. 10.15 Continuous query operators: (a) Selection, (b) Join

- **Multiplexing and demultiplexing:** Physical streams may need to be decomposed into a series of logical streams and conversely logical streams may need to be fused into one physical stream (similar to group-by and union, respectively).
- **Stream mining:** Operations such as pattern matching, similarity searching, and forecasting are needed for on-line mining of streaming data.
- **Joins:** Support should be included for multistream joins and joins of streams with static metadata.
- **Windowed queries:** All of the above query types may be constrained to return results inside a window (e.g., the last 24 hours or the last one hundred packets).

While these look, by and large, as ordinary relational query operators, their implementation and optimization present novel challenges that we discuss below.

Some of these operators are stateless (e.g., projection and selection) and their relational implementations may be used in streaming queries without significant modifications. Figure 10.15(a) depicts the implementation of selection operator as an example. Incoming tuples are simply filtered based on the select condition.

However, stateful operators (e.g., joins) have blocking behavior in their relational implementations that is not suitable in DSSs. For instance, prior to returning the next tuple, the Nested Loops Join (NLJ) may potentially scan the entire inner relation and compare each tuple therein with the current outer tuple. Given the unbounded nature of streaming data, such blocking is problematic. It has been proven that a query is monotonic if and only if it is *non-blocking*, which means that it does not need to wait until the end-of-input marker before producing results. Some operators

have non-blocking counterparts, such as joins and simple aggregates. For example, a non-blocking pipelined symmetric hash join (of two character streams, Str_1 and Str_2) builds hash tables on the fly for each of Str_1 and Str_2 (see Fig. 10.15(b)). Hash tables are stored in main memory and when a tuple from one of the relations arrives, it is inserted into its table and the other tables are probed for matches to generate results involving the new tuple, if any. Joins of more than two streams and joins of streams with a static relation are straightforward extensions. In the former, for each arrival on one input, the states of all the other inputs are probed in some order. In the latter, new arrivals on the stream trigger the probing of the relation. Since maintaining hash tables on unbounded streams is not practical, most DSMSs only support window joins, where the windows over each input stream are defined and joins are computed over the data in these windows based on the specific window semantics.

Unblocking a query operator may be accomplished by reimplementing it in an incremental form, restricting it to operate over a window, and exploiting stream constraints such as *punctuations*, which are constraints (encoded as data items) that specify conditions for all future items. We return to punctuations soon. Sliding window operators process two types of events: arrivals of new data and expirations of old data. We discuss this at length in the next section when we discuss query processing issues.

10.3.2 Query Processing over Data Streams

With some modifications, the query processing methodology over streaming data is similar to its relational counterpart: declarative queries are translated into execution plans that map logical operators specified in the query into physical implementations. However, a number of differences arise in the details.

An important difference is the introduction of persistent queries and the fact that operations consume data pushed into the plan by the sources, rather than pulling data from sources as in a traditional DBMS. Furthermore, as discussed above, the operations may be (and often are) more complicated than relational operators and involve UDFs. Queues allow sources to push data into the query plan and operations to retrieve data as needed. A simple scheduling strategy allocates a time slice to each operation, during which it extracts tuples from its input queue(s), processes them in timestamp order, and deposits output tuples into the next operation's input queue (Fig. 10.16).

As noted earlier, DSSs can follow either the continuous processing model or the windowed execution model. A fundamental concern in the latter case is how the windows are managed—specifically, the addition and deletion of data items to/from the current window. This represents another distinction from relational DBMSs and we discuss it in Sect. 10.3.2.1. Two other issues arise in stream systems that we discuss: load management when data arrival rate exceeds the system's processing capacity (Sect. 10.3.2.2), and dealing with out-of-order data

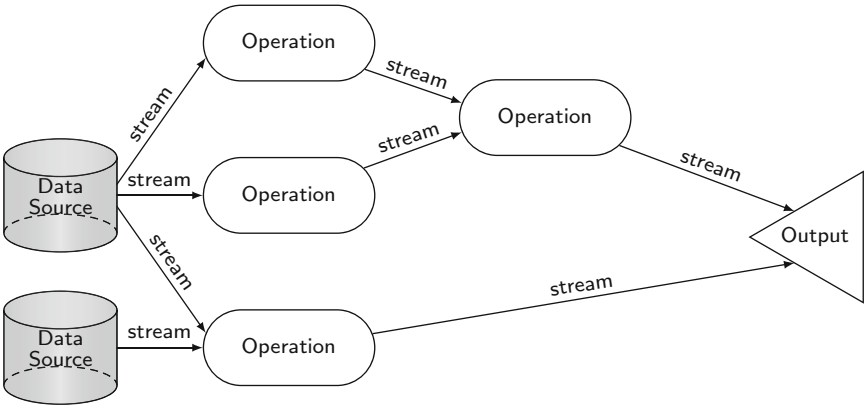


Fig. 10.16 Stream query plan example

items (Sect. 10.3.2.3). Finally, persistent queries provide additional opportunities for multiquery processing, and we discuss this topic in Sect. 10.3.2.4.

Distributed and parallel DSSs follow different paths, similar to their relational counterparts. In the distributed case, the fundamental technique is partitioning the query plan across multiple processing nodes on data that reside on those nodes. Partitioning the query plan involves assigning query operators to nodes and may require rebalancing over time. The issues here are analogous to distributed DBMSs that we have discussed at length earlier in the book. In parallel systems, data-parallel processing is typically followed where the stream data is partitioned and each processing node executes the same query on a subset of the data. Most modern systems follow the latter approach so we discuss those further in Sect. 10.3.2.5.

10.3.2.1 Windowed Query Execution

We noted earlier that in windowed execution, the system has to deal with the arrival of new data and expiration of old data. The actions taken upon arrival and expiration vary across operators. A new data item may generate new results (e.g., join) or remove previously generated results (e.g., negation). Furthermore, an expired data item may cause removal of data items from the result (e.g., aggregation) or addition of new data to the result (e.g., duplicate elimination and negation). Note that we are not discussing here the case where an application deletes a data item. This discussion is about removal of data items from query results as a consequence of window operations.

Consider, for example, the sliding window join: a newly arrived data on one of the inputs probes the state of the other input, as in a join of unbounded streams. Additionally, expired data is removed from the state.

Expiration from an individual time-based window is simple: a data item expires if its timestamp falls out of the range of the window.

In a count-based window, the number of data items remains constant over time. Therefore, expiration can be implemented by overwriting the oldest data item with a newly arrived one. However, if an operator stores state corresponding to the output of a count-based window join, then the number of data items in the state may change, depending upon the join attribute values of new tuples.

In general, there are two techniques for sliding window query processing and state maintenance: the negative tuple approach and the direct approach. In the negative tuple approach, each window referenced in the query is assigned an operator that explicitly generates a negative tuple for every expiration, in addition to pushing newly arrived tuples into the query plan. Thus, each window must be materialized so that the appropriate negative tuples are produced. Negative tuples propagate through the query plan and are processed by operators in a similar way as regular tuples, but they also cause operators to remove corresponding "real" tuples from their state. The negative tuple approach can be implemented efficiently using hash tables as operator state so that expired tuples can be looked up quickly in response to negative tuples. The downside is that twice as many tuples must be processed by the query because every tuple eventually expires from its window and generates a corresponding negative tuple. Furthermore, additional operators must be present in the plan to generate negative tuples as the window slides forward.

Direct approach handles negation-free queries over time-based windows. These queries have the property that the expiration times of base tuples and intermediate results can be determined via their expiry timestamps, which is the arrival time plus the window length. Hence, operators can access their state directly and find expired tuples without the need for negative tuples. The direct approach does not incur the overhead of negative tuples and does not have to store the base windows referenced in the query. However, it may be slower than the negative tuple approach for queries over multiple windows since state buffers may require a sequential scan during insertions or deletions.

10.3.2.2 Load Management

The stream arrival rates may be so high that not all tuples can be processed, regardless of the (static or run-time) optimization techniques used. In this case, two types of load shedding may be applied: random or semantic, with the latter making use of stream properties or quality-of-service parameters to drop tuples believed to be less significant than others. For an example of semantic load shedding, consider performing an approximate sliding window join with the objective of attaining the maximum result size. The idea is that tuples that are about to expire or tuples that are not expected to produce many join results should be dropped (in case of memory limitations), or inserted into the join state but ignored during the probing step (in case of CPU limitations). Note that other objectives are possible, such as obtaining a random sample of the join result.

In general, it is desirable to shed load in such a way as to minimize the drop in accuracy. This problem becomes more difficult when multiple queries with many operators are involved, as it must be decided where in the query plan the tuples should be dropped. Clearly, dropping tuples early in the plan is effective because all of the subsequent operators enjoy reduced load. However, this strategy may adversely affect the accuracy of many queries if parts of the plan are shared. On the other hand, load shedding later in the plan, after the shared subplans have been evaluated and the only remaining operators are specific to individual queries, may have little or no effect in reducing the overall system load.

One issue that arises in the context of load shedding and query plan generation is whether an optimal plan chosen without load shedding is still optimal if load shedding is used. It has been shown that this is indeed the case for sliding window aggregates, but not for queries involving sliding window joins.

Note that instead of dropping tuples during periods of high load, it is also possible to put them aside (e.g., spill to disk) and process them when the load has subsided. Finally, note that in the case of periodic reexecution of persistent queries, increasing the reexecution interval may be thought of as a form of load shedding.

10.3.2.3 Out-of-Order Processing

Our discussion so far has assumed that the DSS processes incoming data items in-order, usually in timestamp order. However, this is not always realistic. Since the data arrives from external sources, some items may arrive late, or out-of-order with respect to their generation time. Furthermore, no data may be received from a source (e.g., a remote sensor or a router) for some time, which could mean that there are no new data to report, or that the source is down. Particularly in distributed systems, this should be considered the normal operating condition due to network disconnections, length of recovery, etc. Consequently, out-of-order processing needs to be considered.

One early approach to deal with this issue is to build in a "slack" that establishes an upper bound on how much unordered the data can be. This is followed in Aurora that has, for example, a buffered sort operator where the incoming stream is buffered for this slack time before being processed. The operator outputs the stream in sorted order on some attribute. Data that arrives later than the slack time units are dropped. Truviso introduces the concept of "drift" to accommodate the case where streams from the same data source are ordered within themselves, but delays may occur in data feed from some sources. When the input monitor detects this, it starts a drift period during which it buffers data from other sources. The difference between the two is that Aurora can define slack on a per-operator basis whereas Truviso has drift management at the input monitor.

Another solution is to use punctuations introduced previously. In this case, a punctuation is a special tuple that contains a predicate that is guaranteed to be satisfied by the remainder of the data stream. For instance, a punctuation with the predicate `timestamp > 1262304000` guarantees that no more tuples will

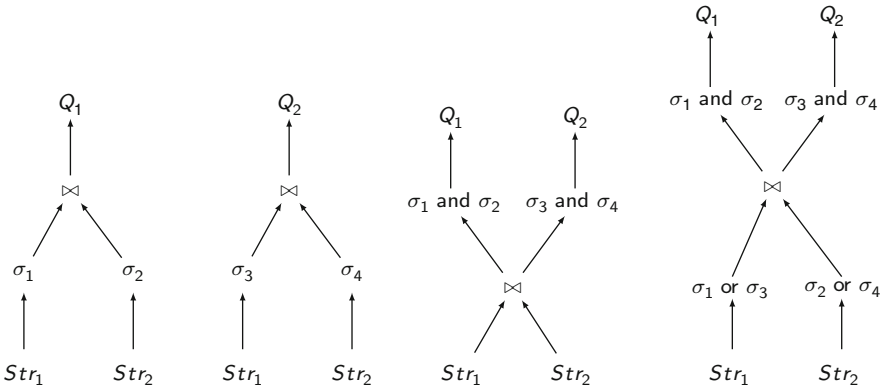


Fig. 10.17 Separate and shared query plans for Q_1 and Q_2

arrive with timestamps below the given Unix time; of course, if this punctuation is generated by the source, then it is useful only if tuples arrive in timestamp order. Punctuations that govern the timestamps of future tuples are typically referred to as *heartbeats*.

10.3.2.4 Multiquery Optimization

Database queries may share parts that are identical and techniques for optimizing a batch of queries have long been of interest and this is referred to as *multiquery optimization*. In streaming systems that support persistent queries, there is more opportunity to detect and exploit shared components and state in processing them. For example, aggregate queries over different window lengths and possibly different `SLIDE` intervals may share state and data structures. Similarly, state and computation may be shared across similar predicates and joins. Therefore, a DSS may group similar queries and run a single query plan per group.

Figure 10.17 shows some of the issues involved in shared query plans. The first two plans correspond to executing queries Q_1 and Q_2 separately, in which selections are evaluated before joins. The third plan executes both queries and evaluates the join first, then the selections (note that the join operator effectively creates two copies of its output stream). Despite sharing work across two queries, the third plan may be less efficient than separate execution if only a small fraction of the join result satisfies the selection predicates σ_1 through σ_4. If so, then the join operator will perform a great deal of unnecessary work over time. The fourth plan addresses this problem by "prefiltering" the streams before they are joined.

10.3.2.5 Parallel Data Stream Processing

Most of the modern DSSs run on large-scale parallel clusters; so, these are parallel data stream processing systems (PDSPS). These systems have significant similarities to parallel databases we discussed in Chap. 8 and, perhaps more importantly to the big data processing frameworks in Sect. 10.2 of this chapter. Therefore in the following discussion we rely on those discussions and appeal to the specific characteristics of data stream systems discussed earlier.

The typical execution environment in these systems can be characterized as parallel execution of continuous operators. Referring to Fig. 10.16, each vertex is a different continuous operation that is assigned to a number of worker nodes. To simplify the discussion, let us assume that each worker only executes one operation. In this context, each worker machine executes the operation assigned to it on a partition of the data stream and produces results that are streamed to the workers that execute the subsequent operation in the query plan. The important point to note here is that partitioning of the stream happens in between each pair of operations. So, the execution of each operation follows three steps:

1. Partitioning of the incoming stream;
2. Execution of the operation on the partition; and
3. (Optionally) aggregation of the results from the workers.

Stream Partitioning

As with all parallel systems, a particular objective of partitioning is to obtain a balanced load across the workers to avoid stragglers. The differentiating characteristic here is that the dataset that is assigned to each worker arrives in a streaming fashion; therefore the partitioning of the data (according to a key attribute) to multiple workers needs to be done on the fly rather than as an offline process as in the systems discussed in Sect. 10.2.

The simplest load balancing approach in distributed systems is to randomly distribute the load among workers. *Shuffle partitioning* routes incoming data items among the workers in a round-robin fashion (hence, it is also referred to as *round-robin partitioning*). Such partitioning results in a perfectly balanced workload. For stateless applications, this works well, but stateful ones require more care. Since data items with the same key may be assigned to different workers, an aggregation step is required for stateful operations to combine the partial results of each worker for each key at each step of the execution (more on this in Sect. 10.3.2.5). The aggregation is expensive and needs to be taken into account. Additionally, shuffle partitioning also has high space demands for stateful operations as each worker has to maintain the state of each key.

The other extreme is hash partitioning—a technique we have seen a number of times. Hashing ensures that data items with the same key are assigned to one worker, eliminating the expensive aggregation step, and minimizing space requirements

since only one worker maintains the state for each key value. However, it may result in heavily imbalanced load distribution, particularly for skewed (in terms of key values) data streams.

For stateful applications, shuffle and hash partitioning constitute upper-bounds for key splitting cost and load imbalance, respectively. Most recent work has focused on finding partitioning algorithms within these extremes. An approach that is promising is key-splitting, whereby hash partitioning is followed by splitting each key among a small number of workers to reduce the load imbalance. The objective is to reduce the overhead of aggregation while also reducing imbalance, particularly in skewed data streams. Partial Key Grouping (PKG) algorithm aims to reduce the load imbalance of hash partitioning by adapting key splitting. PKG utilizes the "power of two choices" by allowing each key to be split between two workers. That enables PKG to achieve significantly better load balance compared to hash partitioning and bounds the replication factor and the aggregation cost. For heavily skewed data, PKG has been extended to use more than two choices for the head of the distribution. Although it is shown to further improve load balance, its replication factor is upper bounded by the number of worker nodes in the worst case. Another approach to deal with heavily skewed data is to use a hybrid partitioning technique where the tuples in the head (frequent) and tail (less frequent) of the key distribution are treated differently, perhaps with a preference for well-balanced assignment of the heavy hitters from the head.

Parallel Stream Workload Execution

Let us first focus on the execution of individual operations. For stateless operations there are no specific issues raised as a consequence of streaming and the aggregation step is unnecessary. Stateful operations require more care and that is what we discuss below.

If shuffle partitioning is used for a stateful operation, as discussed above, data items with the same key may be located on different workers each of which will store only partial results. Therefore, shuffle partitioning requires an aggregation step to produce the final results. As an example, Fig. 10.18 depicts a counting operation over three workers where different colors indicate different keys. As can be observed, data items with the same keys may go to different workers, each of which maintain the count for each key (state) that are aggregated at the end.

If hash partitioning is used for a stateful operation, all data items with identical keys are assigned to the same worker, so there is no aggregation step. Figure 10.19 demonstrates hash partitioning for the same counting example we used earlier.

When these points are incorporated into the query plan, each operation is typically treated individually, with partitioning decisions performed for each as in Apache Storm and Heron. Consequently, the example query plan given in Fig. 10.16 takes the shape of Fig. 10.20.

An issue that arises is in terms of highly skewed data streams. As we discussed in Sect. 10.3.2.5, the commonly accepted current approach is to use key-splitting

possibly with certain optimizations for highly skewed data. Another approach is stream repartitioning in between operations in the query plan. The fundamental issue here is to reroute the dataflow between operations in the query plan, which also requires state migration (since another worker will be taking over parts of the stream). A number of alternative strategies have been developed, but the seminal work on this is Flux, and we use that as an exemplar to discuss how this repartitioning works. Flux is a dataflow operator that is placed between two operations in the query plan; it monitors the worker loads and dynamically reroutes data and migrates state from one worker to another. This process has two phases: rerouting the data and migrating the state. Rerouting requires update to internal routing tables. State migration requires more care since the state that is maintained at the "old" worker with respect to that partition has to be marshaled and moved to the "new" worker. This involves the following steps: stopping new tuples from being accepted into the partition; marshaling the state at the "old" worker, which involves extracting this information from internal data structures; moving the state to the "new" worker; unmarshaling the state and installing it by populating the data structures of the "new" machine; and restarting the receipt of data to the stream. This state migration needs to be fast, obviously, but it is a heavyweight task that involves complex synchronization protocols. That is one reason modern systems do not typically provide built-in support for state migration.

10.3.3 DSS Fault-Tolerance

Distributed/parallel DSS reliability has similarity to the relational DBMSs, but the issues are exacerbated by the fact that it is necessary to deal with streams flowing

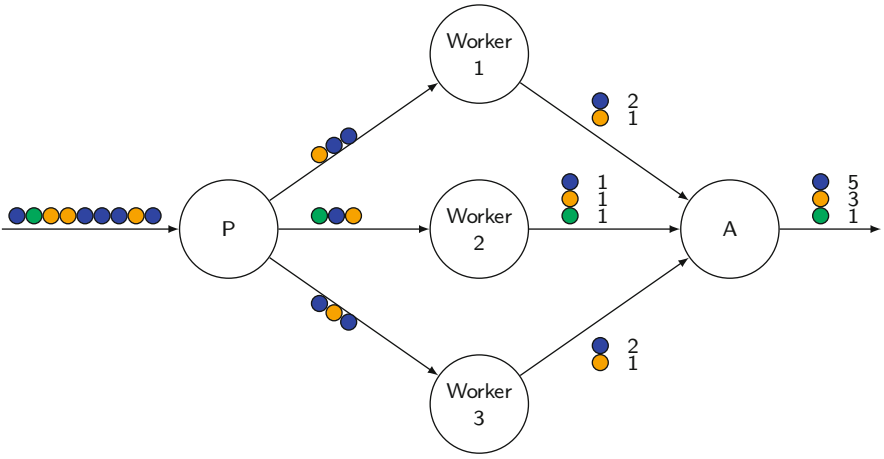


Fig. 10.18 Round-robin stream partitioning

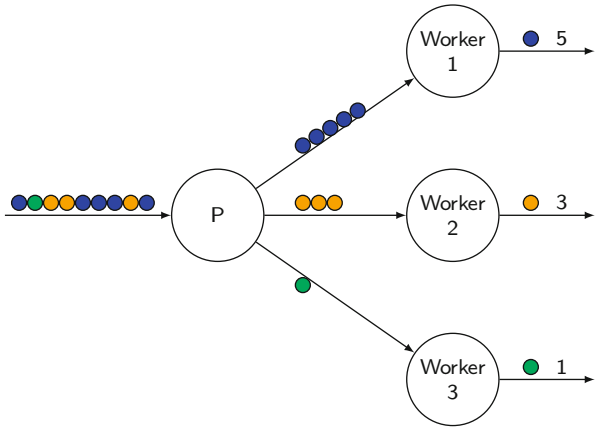


Fig. 10.19 Hash-based stream partitioning

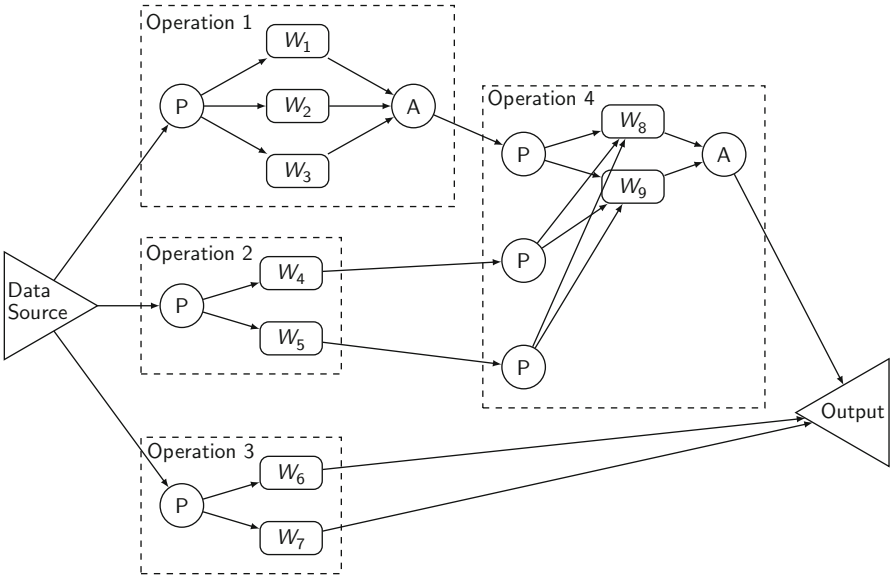


Fig. 10.20 Parallel stream query plan example

through the query plans. Let us first revisit the distinction that we made earlier between systems that partition the query plan and execute each part on a different server node and those that partition the data and replicate the query plan at each node (i.e., the data parallel execution). In the latter, failures can be handled by worker replication techniques that we discussed in Sect. 10.2.1. However, in the former case, the servers are "connected" as they execute parts of the query plan and data flows from upstream servers to downstream ones. A failure of a node can, therefore,

disrupt query execution due to the loss of significant (transient) state information and halting the downstream servers that no longer receive data. Consequently, these systems need to implement strong availability techniques.

An important issue is the query execution semantics a system provides as data flows through the network of servers (or through the query plan). There are three alternatives: at-least-once, at-most-once, and exactly-once. *At-least-once* semantics (also called *rollback recovery*) indicates that the system guarantees to process each data item at least once, but makes no guarantees about duplicates. So, if, a data item is forwarded again by a failed node after recovery, that data item may be processed again and produce a duplicate output. In contrast, if *at-most-once* semantics (also called *gap recovery*) is adopted, the system guarantees that duplicate data items would be detected and not processed, but certain data items may not be executed at all. This can be due to load management as discussed earlier, or as a result of the failure of a node that ignores, upon recovery, all the data items that it might have received while it was down. Finally, *exactly-once* semantics (also called *precise recovery*) means that the system executes each data item exactly once—so nothing gets dropped and nothing gets executed more than once. Each of these require different system functionality obviously. Each of these are supported in the existing systems: Apache Storm and Heron provide applications with a choice of at-least-once and at-most-once semantics, whereas Spark Streaming, Apache Flink, and MillWheel enforce exactly-once semantics.

The types of recovery techniques for DSSs can be classified into two main approaches: replication and upstream backup. In the case of *replication*, for each node that executes a portion of the query plan there is a replica node that is also responsible for that portion of the query plan. This is a primary-secondary arrangement where the primary node services the query plan as long as it is operational, and the secondary node picks up the work if the primary fails. The arrangement between the two could be active standby where both the primary and the secondary get data items from upstream nodes and process them at the same time with only the primary sending outputs downstream, or passive standby where the primary periodically sends the delta difference in its state to the secondary and the secondary updates its state accordingly. Both can be supported by checkpointing to speed up recovery. This approach has been proposed as part of the Flux operator discussed above and used in Borealis. The other alternative is *upstream backup* where upstream nodes buffer the data items that they flow to the downstream nodes until they are processed. If a downstream node fails and recovers, it obtains the buffered data items from its upstream node and reprocesses them. A design difficulty is determining how big these buffers should be to accommodate data that is gathered during failure and recovery. This is complicated because it is affected by data arrival rate as well as other considerations. Systems such as Apache Storm and TimeStream adopt this approach.

10.4 Graph Analytics Platforms

Graph data is of growing importance in many applications. In this section, we discuss *property graph*, which are graphs that have attributes associated with vertices and edges. Another type of graph is Resource Description Framework graph (RDF graph) that we discuss in Chap. 12. Property graphs are used to model entities and relationships in many domains such as bioinformatics, software engineering, e-commerce, finance, trading, and social networks. A graph $G = (V, E, D_v, D_E)$ is defined by a set of vertices V and a set of edges E,[5] D_V and D_E are defined below. The distinguishing characteristics of property graphs are the following:

- Each vertex in the graph represents an entity, and each edge between a pair of vertices represents a relationship between those two entities. For example, in a social network graph representing Facebook, each vertex might represent a user and each edge might represent the "friendship" relationship.
- It is possible to have multiple edges between a pair of vertices, each representing a different relationship; these graphs are commonly called *multigraphs*.
- Edges may have weights attached to them (*weighted graphs*), where the weight of an edge might have different semantics in different graphs.
- The graphs can be *directed* or *undirected*. For example Facebook graph is normally undirected, representing the symmetric friendship relationship between two users: if user A is friend of user B, then user B is a friend of user A. However, a Twitter graph, where the edges represent "follows" relationship, is directed representing that user A is following user B, but the inverse may not necessarily be true. As you will recall, RDF graphs are directed by definition.
- As noted, each vertex and each edge may have a set of attributes (properties) to encode the properties of the entity (in case of vertex) or the relationship (in case of edge). If edges have properties, these graphs are usually called *edge-labeled graphs*. D_V and D_E in the graph definition given above represent the set of vertex and edge properties, respectively. Each vertex/edge may have different properties, and when we refer to the properties of the graph in general, we will write D or D_G.

Real-life graphs that are the subject of graph analytics (such as social network graphs, road network graphs, as well as web graphs we discussed earlier) have a number of properties that are important and affect many aspects of system design:

1. These graphs are very large, some with billions of vertices and edges. Processing graphs with this number of vertices and, especially, edges, requires care.
2. Many of these graphs are known as the power-law or scale-free graphs in which there is significant variation in vertex degrees (known as *degree distribution*

[5] In this section, when necessary, we will write V_G and E_G to specifically refer to the vertices and edges, respectively, of graph G, but, we will omit the subscripts when it is obvious.

skew). For example, while the average vertex degree in Twitter graph is 35, the "supernodes" in that graph have maximum degree of 2.9 million.[6]

3. Following the point above, the average vertex degree in many real-world graphs is quite high with high-density cores. For example, the average vertex degree in Friendster graph is about 55 and in Facebook graph is 190.
4. Some of the real-world graphs have very large diameters (i.e., the number of hops between two farthest vertices). These include the spatial graphs (e.g., the road network graphs) and web graphs: the web graph diameters can be in the hundreds, while some road networks are much larger. The graph diameter affects graph analytics algorithms that depend on visiting and doing computation on each vertex iteratively (we discuss this further below).

Efficiently running workloads on these graphs is an essential part of big data platforms. As with many big data frameworks, these are mostly parallel/distributed (scale-out) platforms that rely on the data graph being partitioned across nodes of a cluster or sites of a distributed system.

Graph workloads are typically separated into two classes. The first is *analytical queries* (or *analytical workloads*) whose evaluation typically requires processing each vertex in the graph over multiple iterations until a fixpoint is reached. Examples of analytical workloads include PageRank computation (see Example 10.4), clustering, finding connected components (see Example 10.5), and many machine learning algorithms that utilize graph data (e.g., belief propagation). We focus on the different computational approaches that have been developed for these tasks, and the systems that have been built to support them. These are the specialized iterative computation platforms that we referred to when we discussed Spark in the previous section. They are our focus in this section. The second class of workloads is *online queries* (or *online workloads*), which are not iterative and usually require access to a portion of the graph and whose execution can be assisted by properly designed auxiliary data structures such as indexes. Examples of online workloads are reachability queries (e.g., whether a target vertex is reachable from a given source vertex), single-source shortest path (finding the shortest path between two vertices), and subgraph matching (graph isomorphism). We postpone the treatment of these workloads to Chap. 11 where we discuss graph DBMSs.

Example 10.4 PageRank is a well-known algorithm for computing the importance of web pages. It is based on the principle that the importance of a page is determined by the number and quality of other pages pointing to it. Quality, in this case, is measured as the PageRank of a page (hence the recursive definition). Each web page is represented as a vertex in the web graph (see Fig. 10.21), with each directed edge representing a "pointing to" relationship. Thus, the PageRank of a web page P_i, denoted $PR(P_i)$ is the summation of the PageRank of all the pages P_j pointing to it, normalized by the number of pages that each P_j points to. The idea is that if

[6] We caution that these values change as the graphs evolve over time. They should be taken as indicative of the point we are making rather than as definitive values.

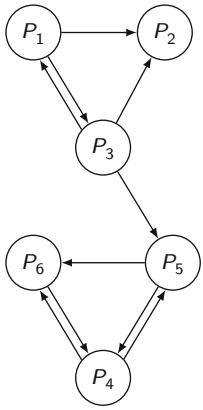


Fig. 10.21 Web Graph Representation for PageRank Computation

a page P_i points to n pages (one of which is P_i), its PageRank contributes to the PageRank computation of n pages equally. The PageRank formula also includes a damping factor based on the theory of random walks: if a user starts from a web page and continues clicking on the links to reach other web pages, this "walk" will eventually stop. So, when the user is at page P_i, there is a probability d that the user will continue clicking and $(1 - d)$ that the walk will stop; the typical value for d is 0.85 determined as a result of empirical studies. Therefore, if the set of in-neighbors of P_i is B_{P_i} (these are P_i's backward links), and the set of out-neighbors is F_{P_i} (forward links), the PageRank formula is

$$PR(P_I) = (1 - d) + d \sum_{P_j \in B_{P_i}} \frac{PR(P_j)}{|F_{P_j}|}.$$

We discuss PageRank in more detail in Chap. 12 when we consider web data management. For the time being, we will simply focus on the computation of the values using Fig. 10.21 as an example. Let us consider page P_2; the PageRank of this page is $PR(P_2) = 0.15 + 0.85(\frac{PR(P_1)}{2} + \frac{PR(P_3)}{3})$. Clearly, this is a recursive formula since it depends on the computation of the PageRank values for P_1 and P_3. The computation typically starts with assigning each vertex equal PageRank values (in this case 1/6 since there are 6 vertices), and iterates to compute the values of each node until a fixpoint is reached (i.e., the values no longer change). Therefore, PageRank computation exhibits both of the properties we identified for analytical workloads: iterative computation, and involvement of each vertex at each iteration.[7]
♦

[7]There are various optimizations that have been developed for PageRank computation, but we ignore them in this discussion.

Example 10.5 As a second example, let us consider the computation of connected components of a graph. First some basics. A graph is said to be *connected* if there is a path between every pair of vertices. A maximal connected subgraph of the graph is called a *connected component*—every vertex in this component is reachable from every other vertex. Finding the set of connected components in a graph is an important graph analytics problem that can be used for a number of applications such as clustering. If the graph is directed, then a subgraph where there is a directed path from every pair of vertices in both directions (i.e., a path from vertex v to u and a path from u to v) is called a *strongly connected component*. For example, in Fig. 10.21, $\{P_1, P_3\}$ and $\{P_4, P_5, P_6\}$ are two strongly connected components. If all directed edges in this graph are replaced by undirected edges, and then the maximal connected components are determined, this produces the set of *weakly connected components*. The entire graph in Fig. 10.21 is one weakly connected component (in this case, the graph is said to be *weakly connected*).

Finding weakly connected component is an iterative algorithm that uses depth-first search (DFS). Given a graph $G = (V, E)$ for each $v \in V$, one conducts DFS to determine the component in which v is in. ♦

10.4.1 Graph Partitioning

As noted earlier, most graph analytics systems are parallel, requiring the data graph to be partitioned and assigned to worker nodes. We have dealt with data partitioning earlier, considering it within the context of distributed relational systems in Chap. 2, and in the context of parallel database systems in Chap. 8. Partitioning graphs is different, because of the connections among the vertices; this is, in some sense, similar to worrying about cross-fragment integrity constraints in distribution design, but graphs require more care due to the heavy communication between vertices, as we will discuss in the upcoming sections. Thus, special algorithms have been designed for graph partitioning, and the literature on this topic is very rich.

Graph partitioning can follow either the *edge-cut* approach (also known as *vertex-disjoint*) or the *vertex-cut* approach (also called *edge-disjoint*). In the former, each vertex is assigned to a partition, but edges may be replicated among partitions if they connect boundary vertices. In the latter, each edge is assigned to a partition, but vertices may be replicated among partitions if they are incident to edges that are allocated to different partitions. In both of these approaches, three objectives are pursued: (1) allocate each vertex or edge to partitions such that the partitions are mutually exclusive, (2) ensure that the partitions are balanced, and (3) minimize the cuts (either edge-cuts or vertex-cuts) so as to minimize the communication between machines to which each partition is assigned. Balancing these requirements is the difficult part; if, for example, we only had to worry about balancing the workload while getting mutually exclusive partitions, a round-robin allocation of vertices (or edges) to machines might be sufficient. However, there is no guarantee that this would not cause large number of cuts.

Partitioning can be formulated as an optimization problem as follows. Given a graph $G(V, E)$ (ignoring properties for the time being), we wish to obtain a partitioning $P = \{P_1, \ldots, P_k\}$ of G into k partitions where the sizes of P_i are balanced, which can be formulated as the following optimization problem:

$$\text{minimize } C(P)$$

$$\text{subject to:}$$

$$w(P_i) \leq \beta * \frac{\sum_{j=1}^{k} w(P_j)}{k}, \forall i \in \{1 \ldots k\}.$$

where $C(P)$ represents the total communication cost of partitioning, and $w(P_i)$ is the abstract overhead of processing partition P_i. The two approaches (vertex-cut and edge-cut) differ in the definition of $C(P)$ and $w(P_i)$, as we discuss below. In the above formulation β is introduced as a slackness parameter to allow partitioning that is not exactly balanced; if $\beta = 1$, then the solution is an exactly balanced partitioning, and the problem is known as k-balanced graph partitioning optimization problem; if $\beta > 1$, then some deviation from exact balance is allowed, and this is known as the (k, β)-balanced graph partitioning optimization problem. This problem has been proven to be NP-hard, and researchers have proposed heuristics methods to achieve an approximate solution.

The heuristics for edge-cut (vertex-disjoint) approach attempt to achieve a balanced allocation of vertices to partitions while minimizing edge-cuts; thus each P_i contains a set of vertices. In these approaches, $w(P_i)$ is defined in terms of the number of vertices per partition (i.e., $w(P_i) = |P_i|$) while the communication cost is computed as a fraction of edge-cuts:

$$C(P) = \frac{\sum_{i=1}^{k} |e(P_i, V \setminus P_i)|}{|E|}.$$

where $|e(P_i, P_j)|$ is the number of edges between partitions P_i and P_j.

The best-known vertex-disjoint heuristic algorithm is METIS, which provides near-optimal partitioning. It consists of three steps:

1. Given a graph $G_0 = (V, E)$ a hierarchy of successively coarsened graphs $G_1, \ldots, G_n$ are produced such that $|V(G_i)| > |V(G_j)|$ for $i < j$. There are a number of possible ways of coarsening, but the most popular is what is called *contraction* where a set of vertices in G_i are replaced by a single vertex in G_j $(i < j)$. The coarsening usually stops when G_n is sufficiently small that a high-cost partitioning algorithm can still be applied. A graph G_i is coarsened to G_{i+1} by finding the *maximal match*, which is the set of edges where no two edges share a vertex. Then the endpoints of each of these edges are represented by a vertex in G_{i+1}.

2. G_n is partitioned using some partitioning algorithm—as noted above, G_n should by now be sufficiently small to use any desired partitioning algorithm regardless of its computational cost.
3. G_n is iteratively uncoarsened to G_0, and at each step:
   (a) the partitioning solution on graph G_j is projected to graph G_{j-1} (note that the smaller subscript indicates finer granularity of the graph) and
   (b) the partitioning of G_{j-1} is improved by various techniques.

Although METIS and related algorithms improve graph analytics workload processing times considerably, they are not practical for even medium-sized graphs because of their high computation cost. The partitioning overhead is an important consideration in graph analytic systems, as the loading and partitioning time of a graph can account for a large portion of the processing time.

A simple vertex-disjoint partitioning heuristic based on hashing is incorporated in the repertoire of most graph analytics systems we discuss below. In this case, a vertex is assigned to the partition to which its identifier hashes. This is simple, and very fast, and would work reasonably well, in terms of balancing the load, in graphs with uniform degree distribution. However, in real-life graphs that have degree skew as discussed above, the result may be unbalanced workload. Edge-cut partitioning models distribute the load in terms of vertices, but for some algorithms the load is proportional to the number of edges, which would not be balanced for skewed graphs. For these cases more sophisticated heuristics that pay attention to the structure of the graph would be appropriate.

One such approach is *label propagation*, where one starts with each vertex having its own label that it iteratively exchanges with its neighbors. At each iteration, each vertex assumes the most frequent label of its "neighborhood"; when the frequencies are identical, a method is used to select the label. This iterative process stops when vertex labels no longer change. This technique is sensitive to the graph structure, but is not guaranteed to produce a balanced partitioning. One way to achieve balance is to start with a non-balanced partitioning, and then use a greedy label propagation algorithm to relocate vertices to achieve balance (or near balance). The greedy algorithm moves vertices to maximize a relocation utility function subject to constraints for balance. The utility function might be, for example, the number of neighbors of the graph that are going to be in the same partition. It is possible to combine METIS with label propagation by incorporating the latter in the coarsening phase. Again, the problem is modeled as a constrained partitioning problem that maximizes a utility function that pays attention to the vertex neighborhood to minimize edge-cuts.

Example 10.6 Consider the graph in Fig. 10.22a. A vertex-disjoint partitioning of this graph is shown in Fig. 10.22(b), where the edge-cuts are shown by dashed lines. This partitioning was achieved by hashing as described above. Notice that this causes 10 out of 12 total edges to be cut. This example demonstrates the difficulty of partitioning graphs with high-degree vertices (in this graph vertices v_3, and in particular v_4 are high degree), leading to high edge-cuts. METIS does better on this graph, but it cannot generate three partitions, and instead produces

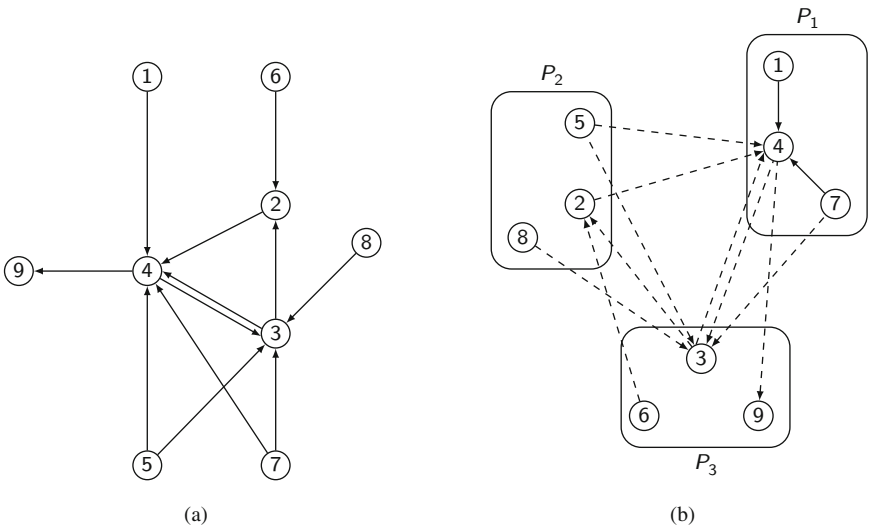


Fig. 10.22 Partitioning example. (**a**) Example graph. (**b**) Vertex-disjoint (edge-cut) partitioning

two: $\{v_1, v_3, v_4, v_7, v_9\}$ and $\{v_2, v_5, v_6, v_8\}$ resulting in five edge-cuts (a two-way partitioning based on hashing results in 8 edge-cuts). ♦

It has been demonstrated that edge-cut heuristics perform well for graphs with low-degree vertices, but perform poorly on power-law graphs causing high number of edge-cuts. METIS has been modified to deal with this particular problem, but its performance in dealing with very large graphs remains an issue. It is generally accepted that vertex-cut approaches that allocate edges to individual partitions while partitioning (replicating) the vertices incident to these edges handle power-law graphs more easily (i.e., each P_i contains a set of edges). The definition of $w(P_i)$, in this case, is the number of edges in partition P_i, i.e., $w_(P_i) = |e(P_i)|$. For these heuristics, the communication $C(P)$ is going to be affected by the replication factor of each vertex (defined as the number of partitions to which that vertex is assigned); this can be formulated as follows:

$$C(P) = \frac{\sum_{v \in V} |A(v)|}{|V|}.$$

where $A(v) \subseteq \{P_1, \ldots, P_k\}$ represent the set of partitions in which vertex v is assigned.

Hashing is also an option as a vertex-cut heuristic: in this case the hashing is done on the ids of the two vertices incident on an edge. It is simple, fast (since it can be easily parallelized), and would provide a good balanced partitioning. However, it may lead to high vertex replication. It is possible to use hashing, but control the

replication factor. One approach that has been proposed is to define, for edge $e_{u,v}$, *constraint sets* C_u and C_v, respectively, for its incident vertices u and v. These are the sets of partitions over which u and v can be replicated. Obviously, edge $e_{u,v}$ has to be assigned to a partition that is common to the constraint sets of both u and v, i.e., $C_u \cap C_u$. The constraint sets a limit on the number of partitions to which a vertex can be assigned, thereby controlling the upper bound of the replication factor. One way to generate these constraint sets is to define a square matrix of partitions, and assign as S_u (similarly S_v) by hashing u (similarly v) to one of the partitions (say P_i), and taking the partitions that lie on the same row and column as P_i.

Vertex-cut heuristics that are cognizant of the graph characteristics have also been designed. A greedy algorithm decides how to allocate $(I + 1)$-st edge to a partition such that the replication factor is minimized. Of course, the allocation of $(I + 1)$-st edge is dependent on the allocation of the first R edges, so past history is important. The location of edge $e_{u,v}$ is decided using the following heuristic rules:

1. If the intersection of $A(u)$ and $A(v)$ is not empty (i.e., there are some partitions that contain both u and v), then assign $e_{u,v}$ to one of the partitions in the intersection.
2. If the intersection of $A(u)$ and $A(v)$ is empty, but if $A(u)$ and $A(v)$ are not individually empty, then assign $e_{u,v}$ to one of the partitions in $A(u) \cup A(v)$ with the most unassigned edges.
3. If only one of $A(u)$ and $A(v)$ is not empty (i.e., only one of u or v has been assigned to partitions), then assign $e_{u,v}$ to one of the partitions of the assigned vertex.
4. If both $A(u)$ and $A(v)$ are empty, then assign $e_{u,v}$ to the smallest partition.

This algorithm takes the graph structure into account, but is hard to parallelize for high performance, since it relies on past history. Parallelizing requires either the maintenance of a global state that is periodically updated, or an approximation where each machine only considers its local state history without maintaining a global one.

It is also possible to use both vertex-cut and edge-cut approaches within one partitioning algorithm. PowerLyra, for example, uses an edge-cut algorithm for lower-degree vertices, and a vertex-cut algorithm for high-degree ones. Specifically, given a directed edge $e_{u,v}$, if the degree of v is low, then it hashes on v, and if it is high, then it hashes on u.

Example 10.7 Again consider the graph in Fig. 10.22a. An edge-disjoint partitioning of this graph is shown in Fig. 10.23, where the replicated vertices are shown by dotted circles. This partitioning was achieved by hashing as described above. Notice that this causes 6 out of 9 vertices to be replicated. ♦

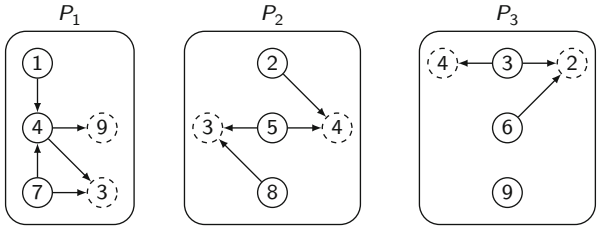


Fig. 10.23 Edge-disjoint (vertex-cut) partitioning

10.4.2 MapReduce and Graph Analytics

It is possible to use a MapReduce system such as Hadoop for graph processing and analytics. However, as noted in Sect. 10.2.1, MapReduce systems do not deal with iterative computation particularly well and we discussed the main issues in that section. In graph systems, there is the additional problem of balanced assignment of vertices across the workers due to the skewed degree distribution in many real-life graphs, as discussed in Sect. 10.4.1, that leads to variations in the communication overhead among worker nodes. All of these cause significant overhead that negatively impacts the performance of MapReduce systems for graph analytics. However, most of the special-purpose graph analytics systems that we discuss in the next section require the entire graph to be maintained in memory; when this is not possible, MapReduce might be a reasonable alternative, and there are studies that look at its use for various workloads, as well as modifications that would allow better scalability. There are also systems that modify MapReduce to better fit iterative graph analytics workloads. As noted earlier, Spark is an improvement of MapReduce to deal with iteration, and GraphX has been developed as a graph processing system on top of Spark. Another MapReduce variant for graph processing is exemplified by the HaLoop system. Both of these separate the state that changes over iterations from the invariant data that does not and cache the invariant data to avoid unnecessary I/O. They also modify the scheduler to ensure that the same data gets mapped to the same workers in multiple iterations. The approaches to implementing these obviously differ with HaLoop modifying the Hadoop task scheduler at the master and task tracker in workers and by implementing a loop control in the master to check for fixpoint. GraphX, on the other hand, uses modifications incorporated into Spark to better deal with iterative workloads. It performs an edge-disjoint partitioning of the graph and creates vertex tables and edge tables at each worker. Each entry in the vertex table includes the vertex identifier along with the vertex properties, while each entry in the edge table includes the endpoints of each edge as well as the edge properties. These tables are realized as Spark RDDs. Any graph computation involves a two-step process (with iteration): join vertex and edge tables, and performs an aggregation. The join involves moving vertex tables to the workers that hold the appropriate edge tables, since the number of vertices are smaller than the number of edges. In order to avoid

broadcasting each vertex table to all of the edge table worker nodes, GraphX creates a routing table that specifies, for each vertex, the edge tables worker nodes where it exists; this is implemented as an RDD as well.

10.4.3 Special-Purpose Graph Analytics Systems

We now turn our attention to systems that have been specifically developed for graph analytics. These systems can be characterized along their *programming models* and their *computation models*. Programming models specify how an application developer would write the algorithms to execute on a system, while computation models indicate how the underlying system would execute these algorithms.

There are three fundamental programming models: vertex-centric, partition-centric, and edge-centric:

- **Vertex-centric model**
  Vertex-centric approach requires the programmer to focus on the computation to be performed on each vertex. Therefore, this is commonly referred to as "think-like-a-vertex" approach. A vertex v bases its computation only on its own state and the states of its neighbor vertices. For example, in the computation of PageRank, each vertex is programmed to receive the rank computation from its neighbors, and to calculate its own rank based on these. The results of the state computation is then available to neighbor vertices so that they can perform their computation.
- **Partition-centric model**
  In systems that follow this programming model, the programmer is expected to specify the computation that is to be performed on an entire partition rather than on each vertex. This is also referred to as *block-centric*, since computation is over blocks of vertices. The approach is also known as "think-like-a-block" or "think-like-a-graph."

  Partition-centric approach typically uses a serial algorithm within each block and only relies on the states of entire neighbor blocks rather than states of individual vertices. This characteristic of using separate computation algorithms within a partition and across partitions (for the vertices at the boundaries) may result in more complicated algorithms, but reduces dependence on neighbor states and communication overhead.
- **Edge-centric model**
  A third approach is the dual of vertex-centric model in that the operations are specified for each edge rather than each vertex. In this case, the principle object of attention is an edge rather than a vertex. Following the same naming scheme, this can be called "think-like-an-edge."

The computation models are bulk synchronous parallel (BSP), asynchronous parallel (AP), and gather-apply-scatter (GAS):

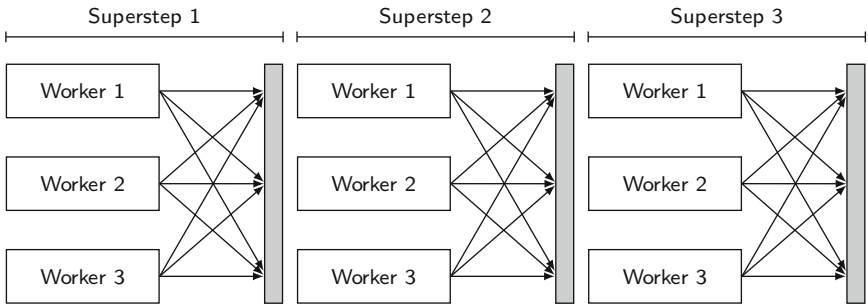


Fig. 10.24 BSP Computation Model

- **Bulk Synchronous Parallel**
  Bulk synchronous parallel (BSP) is a parallel computation model in which a computation is divided into a series of supersteps separated by global barriers. At each superstep, all of the processing nodes (i.e., the worker machines) perform the computation in parallel, and at the end of the superstep, they synchronize before starting the next superstep. Synchronization involves sharing state computed in that superstep with others so that this state can be used by all the worker nodes in the following superstep. The computation proceeds in multiple supersteps until a fixpoint is reached. Figure 10.24 shows an example BSP computation that reaches fixpoint in three supersteps involving three processor nodes.

  Since most of these systems run on parallel clusters, the communication is usually in terms of message passing (this is true for all computational models that we discuss). The BSP model implements a push-based communication approach: messages are pushed by the sender and buffered at the receiver. The receipt of a message at the end of one superstep causes the receiver to be automatically scheduled for execution in the next superstep. The BSP model simplifies parallel computation, but it requires care in task partitioning so that the worker machines are reasonably balanced to avoid stragglers. It also incurs synchronization overhead at the end of each superstep.
- **Asynchronous Parallel**
  The asynchronous parallel (AP) model removes the restriction of the BSP model that requires synchronization between worker machines at the global barrier—states computed at superstep k are available to be used in computation in superstep $k + 1$ even if they arrive at the destination within superstep k. The AP model retains global barriers to separate supersteps, but allows the received states to be seen and used immediately. Therefore, computation in state k may be based on the states of neighbors that were computed in superstep $k - 1$ but were delayed and not received until the end of superstep $k - 1$, or in superstep k.

  The fact that computation of state in one superstep may overlap with the receipt of states from neighbors gives rise to consistency concerns: state changes and state reads require careful control. This is usually handled by the application of locks on states while they are being read or written; given that the states are

distributed over multiple worker nodes, there is a need to have distributed locking solutions.

An important objective of the AP model is to improve performance by allowing faster' processing units to continue their processing without having to wait until the synchronization barrier.[8] This may result in fewer number of supersteps. However, since it still retains the synchronization barriers, the synchronization and communication overheads are not completely eliminated.

- **Gather-Apply-Scatter**
  As its name suggests, the gather-apply-scatter (GAS) model consists of three phases: In the gather phase, a graph element (vertex, block, or edge) receives (or pulls) information about its neighborhood; in the apply phase, it uses the gathered data to compute its own state; and in the scatter phase, it updates the states of its neighborhood. An important differentiating characteristic of GAS is the separation of state update from activation. In both BSP and AP, when a state update is communicated to neighbors, they are automatically activated (i.e., scheduled for execution), whereas in GAS, the two actions are separate. This separation is important as it allows the scheduler to make its own decisions as to which graph elements to execute next (perhaps based on priorities).

  GAS can be synchronous or asynchronous. Synchronous GAS is similar to the BSP model in that the global barriers are maintained, but there is one important difference: in BSP, each graph element pushes its state to its neighbors at the end of a superstep, while in GAS, a graph element that is activated pulls the state of its neighbors at the beginning of a superstep.

  The asynchronous GAS removes the global barriers, and therefore has the same consistency issue as the AP model that is addressed by distributed locking. However, it is different than the AP model, since it does not have any notion of supersteps in the same sense as the BSP model. It executes by iteratively scheduling a graph element for execution, gathering its neighbor states (called *scope*), computing its state, and updating the scope and the list of graph elements that require scheduling. The computation ends when there are no more graph elements that await scheduling.

Combination of programming and computation models define the design space with nine alternatives. However, systems have not been built, as of now, for each of the design alternatives. Most of the research and development efforts have focused on vertex-centric BSP systems (Sect. 10.4.4); consequently our discussion of this class will be more in-depth than the others. For those cases where there are no known actual systems, we briefly indicate how one might look like.

[8] We use the terms global barrier, global synchronization barrier, and synchronization barrier interchangeably.

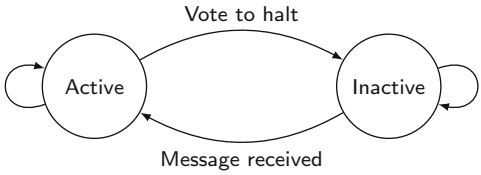


Fig. 10.25 Vertex States in Vertex-Centric Systems

10.4.4 Vertex-Centric Block Synchronous

As noted above, vertex-centric systems require the programmer to focus on the computation that is to be done on each vertex; edges are not first-class objects in these systems since there is no computation performed on them. When coupled with BSP computation model, these systems perform the computation iteratively (i.e., in supersteps) such that at each iteration, each vertex v accesses the state contained in the messages sent to it in the previous iteration, computes its new state based on these messages, and communicates its state to its neighbors (who will read the state in the subsequent superstep). The system then waits until all of the worker machines complete computation in that iteration (the global barrier) before starting the next iteration.

Each vertex is in either an "active" or "inactive" state. The computation starts with all vertices in active state and continues until all vertices reach fixpoint and enter inactive state and there are no pending messages in the system (Fig. 10.25). As each vertex reaches fixpoint it sends a "vote halt" message before it enters inactive state; once inactive, the vertex stays in that state unless it receives an external message to become active again.

This category has been the most popular one for system builders, as we noted earlier. The classical systems are Pregel and its open source counterpart Apache Giraph. Some of the others are GPS, Mizan, LFGraph, Pregelix, and Trinity. We focus on Pregel as an exemplar of this class of systems in discussing a number of details (these systems are commonly referred to as "Pregel-like").

To facilitate vertex-centric computation, a Compute() function is provided for each vertex, and the programmer needs to specify the computation that needs to be performed based on the application semantics. The system provides built-in functions such as GetValue() and WriteValue() to read the state associated with a vertex and modify the state of a vertex, as well as a SendMsg() function to push the vertex state updates to neighbor vertices. These are provided as the basic functionality and the programmer can focus on the computation that needs to be performed at each vertex. In this sense, the approach is similar to MapReduce where the programmer is expected to supply the specific codes for the map() and reduce() functions while the underlying system provides the execution and communication mechanism.

The Compute() function is quite general; in addition to computing a new state for the vertex, it can cause changes to the graph topology (called *mutations*) if the system supports this. For example, a clustering algorithm may replace a set of vertices with a single vertex. The mutations that are performed in one superstep are effective at the beginning of the following superstep. Naturally, conflicts can arise when multiple vertices require the same mutation, such as the addition of the same node with different values. These conflicts are resolved by partial ordering of the operations and by implementing user-defined handlers. The partial ordering of operations imposes the following order: edge removals are performed first, followed by vertex removals, followed by vertex additions, and, finally, edge additions. All of these mutations precede the call to the Compute() function.

Example 10.8 To demonstrate the vertex-centric BSP computation approach, we will compute the connected components of the graph given in Fig. 10.26a. For this example, we choose a simpler graph than the one we used for partitioning (Fig. 10.22a) to demonstrate the computation steps easily.

Note that, since this graph is directed, computing the connected component reduces to computing weakly connected component (WCC), where the directions are ignored (see Example 10.5). Furthermore, since this graph is fully connected, all of the vertices should be in one group, so we use that fact to check the correctness of the computation.

The vertex-centric BSP version of the WCC algorithm is as follows. Each vertex keeps information about the group it is in, and, in each superstep, it shares this information with its neighbors. At the beginning of the subsequent superstep each vertex gets these group ids from its neighbors and selects the smallest group id as its new group id—the Compute() = min{neighbor group ids, selfgroup id}. If its group id has not changed from the previous superstep, then the vertex enters inactive state (recall that inactive state represents that the vertex value has reached fixpoint). Otherwise, it pushes its new group id to its neighbors. When a vertex enters inactive state, it does not send any further messages to its neighbors, but it will receive messages from its active neighbors to determine if it should become active again. The computation continues in this fashion over multiple supersteps.

This execution is depicted in Fig. 10.26b. In the initialization step, the algorithm initializes by assigning each vertex to its own group identified by the vertex id (e.g., vertex v_1 is in group 1). Each vertex's value is the state at the end of the labeled superset. Then this group id is pushed to its neighbors. Each arrow shows when a message gets consumed by the recipient worker—so pointing to the next superset means the message is not accessed until then regardless of when it is delivered or received. In superstep 1, notice that vertices v_4, v_7, v_5, v_8, v_6, and v_9 change their group ids, while vertices v_1, v_2, and v_3 do not change their values and enter inactive state. The entire computation takes 9 supersteps in this example.

Notice that in some cases, vertices that are in inactive state become active as a result of messages they receive from neighbors. For example, vertex v_2 that becomes inactive in superstep 2 becomes active in superstep 4 when it receives a group id 1 from v_7 that causes it to update its own group id. This is a characteristic of this class of computation as discussed above. ♦

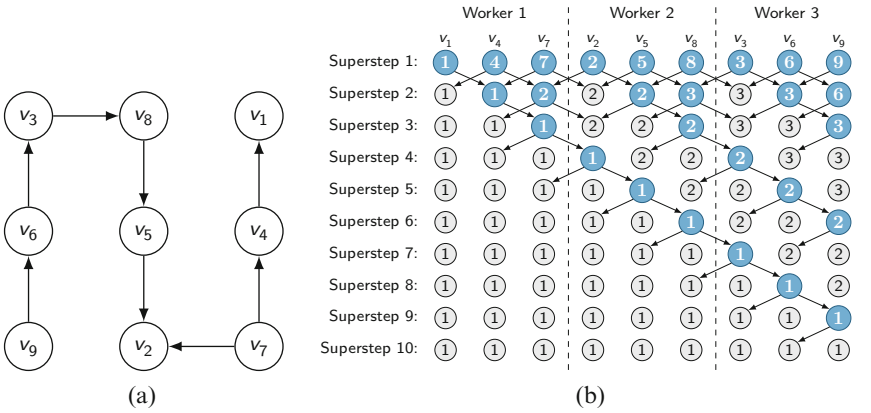


Fig. 10.26 Vertex-centric BSP example. (**a**) Example graph. (**b**) Vertex-centric BSP computation of WCC (gray vertices are inactive, blue vertices are active)

Recall that these systems perform parallel computations over a cluster where there is a master node and a number of worker nodes, with each worker hosting a set of graph vertices and implementing the Compute() function. In some systems (e.g., GPS and Giraph), there is an additional Master.Compute() function that allows for some parts of the algorithm to be executed serially at the master. The existence of these functions provides further flexibility for algorithm implementation and some optimizations (as we discuss below).

For some algorithms, it is important to capture the global state of the graph. To facilitate this, an *aggregator* can be implemented. Each vertex contributes a value to the aggregator, and the result of the aggregation is made available to all the vertices at the following superstep. Systems typically provide a number of basic aggregators such as min, max, and sum.

The performance of systems in this category is affected by two factors: communication cost and the number of supersteps. The properties of real-life graphs that we have discussed earlier impact the two cost factors mentioned above:

1. Power-law graphs with degree distribution skew: The problem with degree distribution skew is that the workers that hold these high-degree vertices receive and have to process far more messages than others, leading to load imbalances across workers that cause the straggler problem discussed earlier.
2. High average vertex degree: This results in each vertex having to deal with a high number of incoming messages, and having to communicate with a high number of neighbor vertices, leading to heavy communication overhead.
3. Large diameters: If, in the BSP computation, each superstep corresponds to one hop (i.e., one message) between vertices, these computations are going to take a large number of supersteps— proportional to the graph diameter. Although a few hundred hops as a graph diameter may not seem excessive, keep in mind that many of the analytics workloads require multiple passes over all the vertices in

these graphs, leading to high algorithmic cost. It has been reported, for example, that running strongly connected component algorithm over a graph of diameter 20 requires over 4,500 supersteps (without optimization).

A system optimization that can be implemented to reduce communication overhead between worker nodes is a *combiner* that combines the messages that are destined to vertex v based on application-defined semantics (e.g., if v only needs the sum of values from neighbors). This cannot be done automatically, since it is not possible for the system to determine when and how this aggregation is appropriate; instead, the system provides a Combine() function whose code the programmer needs to specify.

System-level optimizations to deal with skew include the implementation of graph partitioning algorithms that are sensitive to this skew. The partition-based systems we discuss in Sections 10.4.7–10.4.9 also address these issues by taking a dramatically different system design.

There have been proposals for algorithmic optimizations to deal with these problems as well, but these require modifications to the implementation of the workload algorithms. Although we will not get into these in any detail, we highlight one as an example. Some analytics workload algorithms may result in a small portion of the graph vertices to remain active after the others have become inactive, leading to many more supersteps before convergence. In this optimization, if the "active" part of the graph is sufficiently small, the computation is moved to the master node and executed serially using the Master.Compute() function. It has been shown experimentally that this can reduce the number of supersteps between 20% and 60%.

10.4.5 Vertex-Centric Asynchronous

These systems follow the same programming model as the previous case, but relax the synchronous execution model while maintaining synchronization barriers at the end of each superstep. Consequently, the Compute() function for each vertex is executed at each superstep as above, and the results are pushed to the neighbor vertices, but the messages available as input to the function are not restricted to those that were sent in the previous superstep; a vertex may see the messages it receives within the same superstep that it was sent. Messages that are not available at the time Compute() is executed are picked up at the beginning of the subsequent superstep as in BSP. This approach addresses an important problem with the BSP model, while maintaining the ease of vertex-centric programming: a vertex may see fresher messages that are not delayed until the subsequent superstep. This usually results in faster convergence than in BSP-based systems. GRACE and GiraphUC follow this approach.

Example 10.9 To demonstrate vertex-centric AP systems, we use the weakly connected component example from the previous section (Example 10.8). To

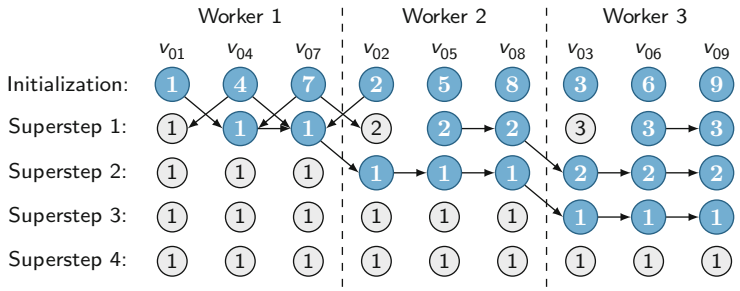


Fig. 10.27 Vertex-centric AP example

simplify matters, we assume that all the messages between vertices arrive to their destinations within the same superstep, and that the computation at each worker is also completed in the same superstep. Furthermore, we assume a single-threaded execution where each worker runs the Compute() on the vertices one-at-a-time. Under these assumptions, the calculation of the WCC over the graph in Fig. 10.26a is shown in Fig. 10.27. Note, for example, that during initialization step, v_1 pushes its group id (1) to v_4, and v_4 pushes its group id (4) to v_1 and v_7. Since we assume a single threaded execution, v_4.Compute is executed to change v_4's group id to 1, which is also pushed to v_7 in the same superstep (superset 1). So, when v_7.Compute() is executed, v_7's group id is set to 1. The access to the vertex states within the same superstep is the distinguishing feature of AP model. ♦

As noted earlier, asynchronous execution requires consistency control by distributed locking. This exhibits itself in this class of systems in the following way. The state of a vertex may be accessed by other vertices while they execute their Compute() functions—vertex v_i may be executing its Compute() function while its neighbor vertex v_j is pushing its updates to v_i. To avoid this, locks are placed on each vertex. Since vertices are distributed across worker nodes, this requires a distributed locking mechanism to ensure state consistency.

Another issue with the AP approach is that it breaks a different type of consistent execution guarantee provided by BSP. Each vertex executes Compute() with all the message it has received since the last execution, and this will be a mix of old messages (from the previous superstep) and new messages (from the current superstep). Therefore, it is not possible to argue that, at each superstep, each vertex consistently computes a new state based on the states of the neighbor vertices at the end of the previous superstep. However, this relaxation allows a vertex to execute its Compute() function as soon as it receives a high priority message, for example.

Although the AP model addresses the message staleness problem of BSP, it still has performance bottlenecks due to the existence of global synchronization barriers, namely it has high communication overhead and it has to deal with stragglers. These can be overcome by eliminating some of the barriers as proposed in the *barrierless asynchronous parallel* (BAP) model of GiraphUC. BAP maintains *global barriers* between *global supersteps* when the worker nodes globally synchronize, but it

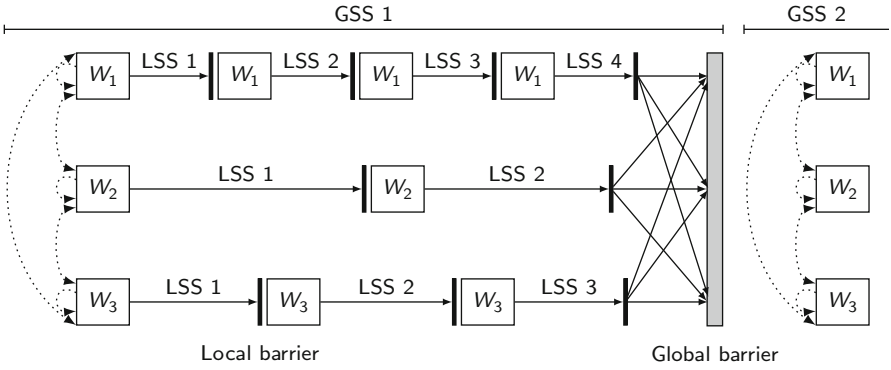


Fig. 10.28 BAP Model Over Three Worker Nodes

divides each global superstep into *logical supersteps* separated by very lightweight *local barriers*. This allows fast workers to execute Compute() function multiple times (once in each logical superstep) before it needs to globally synchronize with slower worker nodes. As in AP, vertices can immediately read the local and remote messages they have received, and this reduces the message staleness. Figure 10.28 demonstrates BAP over three worker nodes; the first worker (W_1) has four logical supersteps (LSS) within the first global superstep (GSS), the second worker has two, and the third worker has three. The dotted arrows represent messages received and processed in the same logical superstep, while solid arrows are messages that are picked up in the subsequent global superstep.

The BAP model requires care in determining termination. Recall that in both the BSP and AP models, the termination is checked at each synchronization barrier by checking that all the vertices are in inactive state and there are no messages in transmission. Since there is now a separation of local and global barriers, this check needs to occur at both the local and the global barriers. A simple approach is to check, at a local barrier, whether there are any more local or remote messages to process; if there are not, then there is no more work to do, and the vertices at this worker node arrive at the global barrier. When all the vertices in the graph arrive at the global barrier, a second check is performed, which is the same as in BSP and AP: the computation globally terminates if all vertices are inactive and there are no more messages.

10.4.6 Vertex-Centric Gather-Apply-Scatter

This category is characterized by GraphLab that combines vertex-centric programming model with the pull-based GAS computation model. As noted earlier, there can be a synchronous version of this approach (as implemented in GraphLab Sync) that is practically the same as vertex-centric BSP (except the pull aspect), so we will

not discuss that further. The asynchronous version is different: in the gather phase, a vertex pulls data from its neighbors rather than the neighbors pushing their data.[9] For each vertex v a *scope* is defined $[Scope(v)]$ that consists of the data stored in all adjacent edges and vertices as well as the data in v. The Compute() function (in GraphLab this is called the Update function) takes as input v and $Scope(v)$ and returns the updated $Scope'(v)$ as well as a set of vertices V' whose states have changed, and are candidates for scheduling. The execution follows three steps, taking as input a graph G and an initial set of vertices V':

1. Remove a vertex from V' according to the scheduling decision,
2. Execute the Compute() function and compute $Scope'(v)$ and V',
3. $V' \leftarrow V \cup V'$.

These three steps are executed iteratively until there are no more vertices in V. The separation of state updates in $Scope'(v)$ (i.e., states of neighbors) from the scheduling of vertex computations is a major distinction of GAS from the AP approach where the messages that update vertex states also schedule those vertices for computation. This separation allows flexibility in choosing the order of vertex computations, e.g., based on priorities or on load balancing. Furthermore, note that there is no explicit SendMsg() function in GAS execution; sharing state changes is done in the gather phase.

Since a vertex v can directly read data from its $Scope(v)$, inconsistency can arise as multiple vertex computations may cause conflicting updates of state, as noted earlier, requiring the deployment of distributed locking mechanisms. When vertex v is executing Compute, obtains locks over its $Scope(v)$, performs its computation, updates $Scope(v)$ and then releases its locks. In GraphLab, this is referred to as *full consistency*. In that particular system, two more relaxed consistency levels are provided to better accommodate applications whose semantics may not require full mutual exclusion: *edge consistency* and *vertex consistency*. Edge consistency ensures that v has read/write access to its own data and the data of its adjacent edges, but only read access to its neighboring vertices. For example, PageRank computation would only require edge consistency, since it only reads the ranks of the neighboring vertices. Vertex consistency simply ensures that while v is executing its Compute() function, no other vertex will be accessing it. Edge and vertex consistency allow application semantics to be taken into account for consistency.

10.4.7 Partition-Centric Block Synchronous Processing

As we noted in Sect. 10.4.4, many real-life graphs exhibit properties that are challenging for vertex-centric systems, and a number of optimizations have been

[9] GraphLab also differentiates itself by its distributed shared memory implementation, but that is not important in this discussion.

developed to deal with the issues that are raised. Partition-centric BSP systems constitute a different approach to deal with these problems. These systems exploit the partitioning of the graph over worker nodes so that, instead of each vertex communicating with others using message passing as in the vertex-centric approach, communication is limited to messages across blocks (partitions) with a simpler, serial algorithm implemented within each partition. The computation follows BSP, so multiple iterations are performed as supersteps until the system converges. This approach is exemplified by Blogel and Giraph++.

The critical point of these systems is they execute a serial algorithm within blocks, and only communicate between blocks. One way to reason about this is that given a graph $G = (V, E)$, after partitioning, we have a graph $G' = (B, E')$ where B is the set of blocks, and E' is the set of edges between blocks. In partition-centric algorithms, the communication overhead is bounded by $|E'|$, which is significantly smaller than $|E|$; therefore, graphs with high density have low communication overhead. For example, an experiment conducted on the Friendster graph to compute connected components show that a vertex-centric system takes 372 times more messages and 48 times longer computation time than a partition-centric system. Partition-based systems also reduce the diameter of the graphs since each block is represented by one vertex in G', and this results in a significantly reduced number of supersteps in BSP computation. A similar experiment for computing connected components over the USA road network graph (which has a diameter of about 9,000) demonstrates that the number of supersteps are reduced from over 6,000 to 22. Finally, dealing with skew in degree distribution is handled by the graph partitioning algorithm that ensures a balanced number of vertices in each block. Since a serial algorithm is executed within each block, higher degree vertices do not necessarily cause high number of messages—again the argument is that vertex-centric systems work on G while partition-centric systems work on the significantly smaller G'.

Example 10.10 Let us consider how the WCC computation we have been discussing would be performed in a partition-centric BSP system. The computation steps are shown in Fig. 10.29 where the graph segment at each worker is a partition denoted by shading. Since a serial algorithm is used within each partition, the algorithm we have been discussing, namely where each vertex starts out in its own group is not necessarily the one that might be used, but for comparison to previous approaches, we will assume the same algorithm. In superstep 1, each worker node performs a serial computation to determine the group ids for the vertices in its partition—for worker 1 the smallest group id is 1, so that becomes the group id of vertices v_1, v_4, and v_7. Similar computation takes places at other workers. At the end of the superstep, each worker pushes its group id to the other workers, and the computation repeats.

Notice that the number of supersteps in this case is the same as the vertex-centric AP (Example 10.9); in general, it could be lower, but this is not the main point. The savings is in the number of messages that are exchanged: partition-centric approach exchanges only 6 messages whereas vertex-centric AP exchanges 20 messages. The

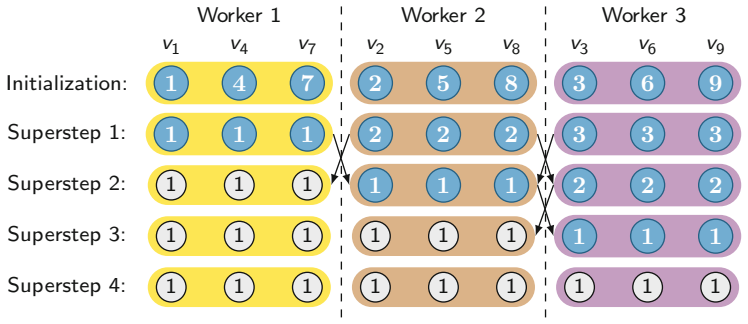


Fig. 10.29 Partition-centric BSP example

example graph we are using in these examples are not densely connected; if it were, the savings would have been more substantial. ♦

10.4.8 Partition-Centric Asynchronous

Systems in this category would partition the graph among worker nodes as in Sect. 10.4.7 and execute a serial algorithm within each partition, but communicate among workers asynchronously when inter-partition messages are sent. As noted earlier, the asynchronous communication is usually implemented using distributed locking. In that sense, these systems would be very similar to distributed DBMSs that we have been discussing in this book, if one treats each worker partition as a data fragment. At this point in time, there are no systems that have been developed in this category.

10.4.9 Partition-Centric Gather-Apply-Scatter

The only difference in this case to the partition-centric BSP described in Sect. 10.4.7 is the use of pull-based GAS rather than push-based BSP. Again, this class of systems are very similar to distributed DBMSs with appropriate changes to the data transfer operations in the query plans. So far, there are no systems that have been developed in this category.

10.4.10 Edge-Centric Block Synchronous Processing

Edge-centric systems focus on each edge as the primary object of interest; in this sense they are duals of vertex-centric approaches. The computation is done on each edge and iterates using supersteps until the fixpoint is reached. Note, however, that an edge in a graph is identified with its two incident vertices. Therefore, executing Compute() on an edge requires executing on that edge's incident vertices. The real difference, therefore, is that the system handles one edge at-a-time, rather than one vertex at-a-time as in the vertex-centric approach.

A natural question is why this would be preferable given that most graphs have far more edges than vertices. At first glance, it may seem that edge-centric approach would cause more computation. However, recall that the larger number of edges result in high messaging cost in vertex-centric systems. Furthermore, in vertex-centric systems, the edges are typically sorted by their originating vertex and an index is built upon them for easier access. When state updates are propagated to neighbor vertices, this index is then randomly accessed to locate edges incident to that vertex; this random access is expensive. Edge-centric systems aim to counter these problems by operating on unsorted sequence of edges where each edge identifies its source and target vertices—there is no random access to an index and when messages are pushed following computation on an edge, there is only one target vertex. Since we are considering BSP computation model, the updates that are computed at one superstep are available at the start of the subsequent superstep. X-Stream follows this approach over a shared memory parallel system and implements optimizations for both in-memory and disk-based graph processing.

10.4.11 Edge-Centric Asynchronous

The systems in this category would be modifications of vertex-centric asynchronous ones (Sect. 10.4.5) where the concern is on the consistent execution over each edge rather than over each vertex. Therefore, locks would be implemented on edges rather than on vertices. Currently, there are no known systems in this category.

10.4.12 Edge-Centric Gather-Apply-Scatter

Combining edge-centric programming with GAS computation model would entail changes to edge-centric BSP systems (Sect. 10.4.10) in the same way that vertex-centric GAS systems modify vertex-centric BSP systems. This would mean that the gather, apply, and scatter functions are implemented on edges with pull-based state read performed during the gather phase. At this point, there are no known systems that follow this approach.

10.5 Data Lakes

Big data technologies enable the storage and analysis of many different kinds of data, which can be structured, semistructured, or unstructured, in its natural format. This provides the opportunity to address in a new way the old problem of physical data integration (see Chap. 7, which has been typically solved using data warehouses. A data warehouse integrates data from different enterprises' data sources using a common format, usually the relational model, which requires data transformation. By contrast, a data lake stores data in its native format, using a big data store, such as Hadoop HDFS. Each data element can be directly stored, with a unique identifier and associated metadata (e.g., data source, format, etc.). Thus, there is no need for data transformation. The promise is that, for each business question, one can quickly find the relevant dataset to analyze it.

The term data lake was introduced to contrast with data warehouses and data marts. It is often associated with the Hadoop software ecosystem. It has become a hot, yet controversial, topic, in particular, in comparison with data warehouse.

In the rest of this section, we discuss in more details data lake versus data warehouse. Then, we introduce the principles and architecture of a data lake. Finally, we end with open issues.

10.5.1 Data Lake Versus Data Warehouse

As discussed in Chap. 7, a data warehouse follows physical database integration. It is central to OLAP and business analytics applications. Thus, it is generally at the heart of an enterprise's data-oriented strategy. When data warehousing started (in the 1980s), such enterprise data was located in OLTP operational databases. Today, more and more useful data is coming from many other big data sources, such as web logs, social networks, and emails. As the result, the traditional data warehouse suffers from several problems:

- **Long development process.** Developing a data warehouse is typically a long, multiyear process. The main reason is that it requires upfront a precise definition and modeling of the data that is needed. Once the needed data is found, often in enterprise information silos, a global schema and associated metadata need be carefully defined and data cleaning and transformation procedures must be designed.
- **Schema-on-write.** A data warehouse typically relies on a relational DBMS to manage the data, which is structured according to a relational schema. Relational DBMSs adopt what has been recently called a *schema-on-write* approach, to contrast with *schema-on-read* (discussed shortly). With schema-on-write, the data is written to the database with a fixed format, as defined by the schema. This helps enforce database consistency. Then, users can express queries based on the

schema to retrieve the data, which is already in the right format. Query processing is efficient in this case, as there is no need to parse the data at runtime. However, this is at the expense of difficult and costly evolution to adapt to changes in the business environment. For instance, introducing new data may imply schema modifications (e.g., adding new columns), which in turn may imply changes in applications and predefined queries.

- **OLAP workload data processing.** A data warehouse is typically optimized for OLAP workloads only, where data analysts can interactively query the data across different dimensions, e.g., through data cubes. Most OLAP applications, such as trend analysis and forecasting, need to analyze historical data and do not need the most current versions of the data. However, more recent OLAP applications may need real-time access to the operational data, which is hard to support in a data warehouse.
- **Complex development with ETL.** The integration of heterogeneous data sources through a global schema requires developing complex ETL programs, to deal with data cleaning, data transformation and manage data refreshment. With more and more diverse data sources to integrate, ETL development becomes even more difficult.

A data lake is a central repository of all the enterprise data in its natural format. Like a data warehouse, it can be used for OLAP and business analytics applications, as well as for batch or realtime data analysis using big data technologies. Compared with data warehouse, a data lake can provide the following advantages.

- **Schema-on-read.** Schema-on-read refers to the "load-first" approach to big data analysis, as exemplified by Hadoop. With schema-on-read, the data is loaded as is, in its native format, e.g., in Hadoop HDFS. Then, when the data is read, a schema is applied to identify the data fields of interest. Thus, the data can be queried in its native format. This provides much flexibility as new data can be added at any time in the data lake. However, more work is necessary to write the code that applies the schema to the data, for instance, as part of the Map function in MapReduce. Furthermore, data parsing is performed during query execution.
- **Multiworkload data processing.** The big data management software stack (see Fig. 10.1) provides support for multiple access methods to the same data, e.g., batch analysis with a framework like MapReduce, interactive OLAP, or business analytics with a framework like Spark, realtime analysis with a data streaming framework. Thus, by assembling these different frameworks, a data lake can support multiworkload data processing.
- **Cost-effective architecture.** By relying on open source technologies to implement the big data management software stack and shared-nothing clusters, a data lake provides excellent cost/performance ratio and return on investment.

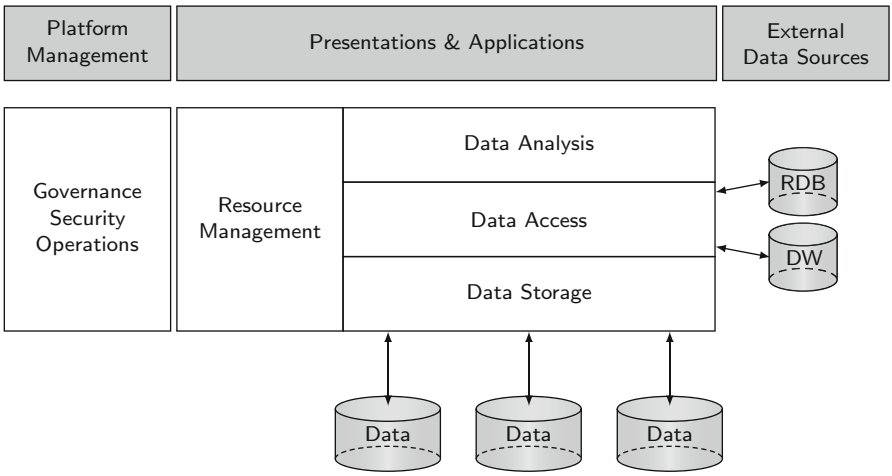


Fig. 10.30 Data Lake Architecture

10.5.2 Architecture

A data lake should provide the following main capabilities:

- Collect all useful data: raw data, transformed data, data coming from external data sources, etc.;
- Allow users from different business units to explore the data and enrich it with metadata;
- Access shared data through different methods: batch, interactive, realtime, etc.
- Govern, secure, and manage data and tasks.

Figure 10.30 shows the data lake architecture, with its main components.

At the center of the architecture is the big data management components (data storage, data access, data analysis, and resource management), on top of which different presentations and applications can be built. These components are those of the big data management software stack and can be found as Apache open source software. Note that many BI tools are now available to work with Hadoop, where we can distinguish new tools or extensions of traditional BI tools for RDBMSs. We can also distinguish between two approaches (which can be combined in one tool):

1. SQL-on-Hadoop, i.e., using an Hadoop SQL driver such as HiveQL or Spark SQL. Examples of tools are Tableau, Platfora, Pentaho, Power BI, and DB2 BigSQL.
2. Function library that provides HDFS access through high-level operators. Examples of tools are Datameer, Power BI, and DB2 BigSQL.

At the left-hand side of the architecture is platform management, which includes data governance, data security, and task operations. These components supplement

big data management with functions that are critical to share data at the enterprise's scale (across multiple business units). Data governance has growing importance in a data lake as it is necessary to manage the data according to the enterprise's policy, with special attention to data privacy laws, e.g., the famous General Data Protection Regulation (GDPR) adopted by the European Union in May 2018). This policy is typically supervised by a data governance committee and implemented by data stewards, who are in charge of organizing data for business needs. Data security includes user authentication, access control, and data protection. Task operations include provisioning, monitoring, and scheduling of tasks (typically in a SN cluster). Like for the BI tools for big data, one can now find Apache tools for data governance, e.g., Falcon, data security, e.g., Ranger and Sentry, and task operations, e.g., Ambari and Zookeeper.

Finally, the right-hand side of the architecture shows that different kinds of external data sources, e.g., SQL, NoSQL, etc., can be integrated, typically using the wrappers of the tools for data access, e.g., Spark connectors.

10.5.3 Challenges

Building and operating a data lake remain challenging, both for methodological and technical reasons. The methodology for a data warehouse is now well-understood. It consists of a combination of prescriptive data modeling (schema-on-write), metadata management, and data governance, which altogether yield strong data consistency. Then, with powerful OLAP or business analytics tools, different users, even with limited data analysis skills can get value out of the data. In particular, a data mart will make it easier to analyze data that is specific to a business need.

In contrast, a data lake lacks data consistency, which makes data analysis at the enterprise's scale much more difficult. This is the main reason skilled data scientists and data stewards are needed. Another reason is that the big data technology landscape is complex and keeps changing. Therefore, the following methodology and best practices to build a data lake should be considered:

- Set a list of priorities and business added values, in comparison with the enterprise's data warehouse. This should include the definition of precise business objectives, and the corresponding data requirements for the data lake.
- Have a global vision of the data lake architecture which should be extensible (to accommodate technical evolutions) and include data governance and metadata management.
- Define a security and privacy policy, which is critical if data is shared between business lines.
- Define a compute/storage model that supports the global vision. In particular, the extensibility and scalability aspects will drive the technical choices.
- Define an operational plan with service level agreements (SLAs) in terms of uptime, volume, variety, velocity, and veracity.

The technical reasons that make a data lake challenging are due to the issues of data integration and data quality. Traditional data integration (see Chap. 7) focuses on the problem of schema integration, including schema matching and mapping, in order to produce a global schema. In the context of big data integration, the problem of schema integration, with many heterogeneous data sources, gets more difficult. The data lake approach simply avoids the problem of schema integration, by managing schemaless data. However, as data lakes mature the issue of schema integration may arise, in order to improve data consistency. Then, an interesting solution is the automatic extraction of metadata and schema information from many related data items, e.g., within the same dataset. One way to do this is by combining techniques from machine learning, matching, and clustering.

10.6 Conclusion

Big data and its role in data science have become important topics in data management, although they are hard to define precisely. Consequently, there is no unifying framework within which developments in this area can be presented—perhaps the reference architecture we provided in Fig. 10.1 is the best that can be done. Therefore, in this chapter, we focused on the properties that characterize these systems and focused on the fundamental platforms that have been proposed to address them. To summarize, we discussed distributed storage systems and MapReduce and Spark processing platforms that address the issues with managing and processing large volumes of data; we discussed data streams that deal with the velocity property associated with big data applications; and we discussed graph analytics that, together with data streams, highlight the issues of variety. Data lakes discussion addresses the variety and scale issues in terms of data integration, and it highlights the data quality problems (veracity) and the need for data cleaning when the source data is not well-curated to begin with.

It is unavoidable that the topics covered in this chapter will continue to evolve and change—this is an area where technology moves fast. We have covered the fundamentals and have pointed to the foundational publications. We give more references in the following section, but the reader would be well-advised to monitor publications for developments.

10.7 Bibliographic Notes

Statistics about big data are scattered and there is no single publication that reports comprehensive statistics. YouTube statistics are from [Brewer et al. 2016] while Alibaba statistics are from personal correspondence. Many more numbers are reported in various blogs and web publications.

A good early discussion of problems with operating system support for DBMSs is Stonebraker [1981] which also explains why early DBMSs moved from file system based storage to block storage. Among the more recent storage systems that we discussed, Google File System is described in [Ghemawat et al. 2003] and Ceph in [Weil et al. 2006].

Our discussion on big data processing platforms (in particular MapReduce) is based primarily on Li et al. [2014]. Sakr et al. [2013] and Lee et al. [2012] also provide overviews of the topic. The original MapReduce proposal is in [Dean and Ghemawat 2004, 2010]. Criticism of MapReduce are discussed in [DeWitt et al. 2008, Dewitt and Stonebraker 2009, Pavlo et al. 2009, Stonebraker et al. 2010]. Sources for MapReduce languages are as follows: HiveQL [Thusoo et al. 2009], Tenzing [Chattopadhyay et al. 2011], JAQL [Beyer et al. 2009], Pig Latin [Olston et al. 2008], Sawzall [Pike et al. 2005]), FlumeJava [Chambers et al. 2010], and SystemML [Ghoting et al. 2011]. The MapReduce join implementation 1-Bucket-Theta algorithm is due to Okcan and Riedewald [2011], broadcast join is due to Blanas et al. [2010], and repartition join is proposed by Blanas et al. [2010].

Spark is proposed in [Zaharia et al. 2010, Zaharia 2016]. As part of the Spark ecosystem, Spark SQL is discussed in [Armbrust et al. 2015], Spark Streaming in [Zaharia et al. 2013], and GraphX in [Gonzalez et al. 2014].

On data stream systems, the rich literature is covered in a number of books. Golab and Özsu [2010] mostly focus on the earlier systems and data and query modeling issues. The book also discusses stream warehouses. Aggarwal [2007] contains a wide range of topics (including stream mining that we omitted) focusing on earlier work. More discussion on stream mining can be found in [Bifet et al. 2018]. Muthukrishnan [2005] focuses on the theoretical foundations of these systems. Generalization of data stream systems to event processing is another direction that has been followed; although we have not discussed this issue in this chapter, Etzion and Niblett [2010] is a good starting point to investigate this direction. In addition to the systems that we discuss, there are DSS deployments in the cloud, as exemplified by StreamCloud[Gulisano et al. 2010, 2012].

The definition of a data stream as an append-only sequence of timestamped items that arrive in some order [Guha and McGregor 2006]. Other definitions of a data stream are given in [Wu et al. 2006, Tucker et al. 2003]. The concept of revision tuples are introduced in data streams by Ryvkina et al. [2006]. Streaming query semantics are discussed by Arasu et al. [2006] within the context of CQL language and more generally by Law et al. [2004]. These languages are classified as either declarative (QL [Arasu et al. 2006, Arasu and Widom 2004], GSQL [Cranor et al. 2003], and StreaQuel [Chandrasekaran et al. 2003]) or procedural (Aurora [Abadi et al. 2003]). Operator executions in stream systems are important due to their non-blocking requirements. Non-blocking joins are topics of [Haas and Hellerstein 1999a, Urhan and Franklin 2000, Viglas et al. 2003, Wilschut and Apers 1991] and aggregation of [Hellerstein et al. 1997, Wang et al. 2003]. Joins of more than two streams (multistream joins) are discussed in [Golab and Özsu 2003, Viglas et al. 2003] and joins of streams with static data is the topic of [Balazinska et al. 2007]. The topic of punctuations as a means of unblocking is presented by

Tucker et al. [2003]. Punctuations can also be used to reduce the amount of state that operators need to support [Ding and Rundensteiner 2004, Ding et al. 2004, Fernández-Moctezuma et al. 2009, Li et al. 2006, 2005]. Heartbeats which are punctuations that govern the timestamps of future tuples are discussed in [Johnson et al. 2005, Srivastava and Widom 2004a].

Query processing over data streams is the topic of [Abadi et al. 2003, Adamic and Huberman 2000, Arasu et al. 2006, Madden and Franklin 2002, Madden et al. 2002a]. Windowed query processing is discussed in [Golab and Özsu 2003, Hammad et al. 2003a, 2005, Kang et al. 2003, Wang et al. 2004, Arasu et al. 2006, Hammad et al. 2003b, 2004]. Load management approaches when stream rate exceeds processing capacity are presented in [Tatbul et al. 2003, Srivastava and Widom 2004b, Ayad and Naughton 2004, Liu et al. 2006, Reiss and Hellerstein 2005, Babcock et al. 2002, Cammert et al. 2006, Wu et al. 2005].

There have been a number of stream processing systems proposed and developed as prototypes and production systems. We classified some of these as Data Stream Management Systems (DSMS): STREAM [Arasu et al. 2006], Gigascope [Cranor et al. 2003], TelegraphCQ [Chandrasekaran et al. 2003], COUGAR [Bonnet et al. 2001], Tribeca [Sullivan and Heybey 1998], Aurora [Abadi et al. 2003], Borealis [Abadi et al. 2005]. We classified others as Data Stream Processing Systems (DSPS): Apache Storm [Toshniwal et al. 2014], Heron [Kulkarni et al. 2015], Spark Streaming [Zaharia et al. 2013], Flink [Carbone et al. 2015], MillWheel [Akidau et al. 2013], and TimeStream [Qian et al. 2013]. As noted, all DSMSs except Borealis were single machine systems, while all of the DSPSs are distributed/parallel.

Partitioning streaming data in parallel/distributed systems is discussed in [Xing et al. 2006] and [Johnson et al. 2008]. Key-splitting is proposed by Azar et al. [1999], Partial Key Grouping (PKG) by Nasir et al. [2015] (the "power of two choices" that PKG is based on is discussed in [Mitzenmacher 2001]). PKG has been extended to use more than two choices for the head of the distribution [Nasir et al. 2016]. Hybrid partitioning to deal with skewed data is given in [Gedik 2014, Pacaci and Özsu 2018]. Repartitioning in between operations in the query plan is discussed in [Zhu et al. 2004, Elseidy et al. 2014, Heinze et al. 2015, Fernandez et al. 2013, Heinze et al. 2014]. In this context, the seminal work flux is proposed by Shah et al. [2003].

Recovery semantics of parallel/distributed stream systems is the topic of [Hwang et al. 2005].

There are many books that focus on the specific aspects of graph analytics platforms, and these typically address how to perform analytics using one of the platforms that we discuss. For a more general book on graph processing, [Deshpande and Gupta 2018] is a good source. Graph analytics is discussed in an extensive survey [Yan et al. 2017]. The survey by Larriba-Pey et al. [2014] is also a very good reference. The real challenges in graph processing is discussed by Lumsdaine et al. [2007]. McCune et al. [2015] provide a good survey of vertex-centric systems.

Graph characteristics, in particular skew in degree distribution, plays a significant role in graph processing. This is discussed in [Newman et al. 2002]. An important first step in parallel/distributed graph processing is graph partitioning, which takes

up a large portion of processing time [Verma et al. 2017] and is computationally expensive [Andreev and Racke 2006]. Graph partitioning techniques can be of two classes: vertex-disjoint (edge-cut) and edge-disjoint (vertex-cut). The main algorithm in the first class is METIS [Karypis and Kumar 1995] whose computation cost is analyzed by McCune et al. [2015].Hashing is another possibility when vertices are hashed to different partitions. These techniques distribute vertices in a balanced way, but they do not deal well with power-law graphs. Extensions to METIS have developed for this case [Abou-Rjeili and Karypis 2006]. Alternatively, label propagation Ugander and Backstrom [2013] is an alternative approach to deal with this problem. Combining METIS with label propagation by incorporating the latter in the coarsening phase of METIS has also been proposed [Wang et al. 2014]. Another alternative is to start from an unbalanced partitioning and incrementally achieving balance [Ugander and Backstrom 2013]. For edge-disjoint partitioning, hashing can be possible. It is also possible to combine both vertex-disjoint and edge-disjoint approaches as in PowerLyra [Chen et al. 2015].

MapReduce has been proposed as a possible approach to process graphs [Cohen 2009, Kiveris et al. 2014, Rastogi et al. 2013, Zhu et al. 2017] as well as modifications that would allow better scalability [Qin et al. 2014]. HaLoop system [Bu et al. 2010, 2012] is a specially tailored MapReduce approach to graph analytics. GraphX [Gonzalez et al. 2014] is a Spark-based system that follows the MapReduce approach.

In native graph analytics systems, the classification we discussed in Sect. 10.4.3 is based on [Han 2015, Corbett et al. 2013]. Bulk synchronous parallel (BSP) computation model is due to Valiant [1990]. Vertex-centric BSP systems include Pregel [Malewicz et al. 2010] and its open source counterpart Apache Giraph [Apache], GPS [Salihoglu and Widom 2013], Mizan [Khayyat et al. 2013], LFGraph [Hoque and Gupta 2013], Pregelix [Bu et al. 2014], and Trinity [Shao et al. 2013]. System-level optimizations to deal with skew are discussed in [Lugowski et al. 2012, Salihoglu and Widom 2013, Gonzalez et al. 2012]. Some algorithmic optimizations are presented in [Salihoglu and Widom 2014]. Vertex-centric asynchronous systems include GRACE [Wang et al. 2013] and GiraphUC [Han and Daudjee 2015]. The primary example of vertex-centric gather-apply-scatter systems is GraphLab [Low et al. 2012, 2010]. Blogel [Yan et al. 2014] and Giraph++ [Tian et al. 2013] follow the partition-centric BSP approach. X-Stream [Roy et al. 2013] is the only edge-centric BSP system that has been developed so far.

Data lake is a new topic and, thus, it is too early to see many technical books on the topic. Pasupuleti and Purra [2015] provide a good introduction to data lake architectures, with a focus on data governance, security, and data quality. One can also find useful information in white papers, e.g., [Hortonworks 2014] from companies that provide data lake components and services. Some of the challenges facing data lakes are in big data integration. Dong and Srivastava Dong and Srivastava [2015] provide an excellent survey of the recent techniques for big data integration. The broader issue of big data integration, which includes the web, is the topic of [Dong and Srivastava 2015]. Within this context, [Coletta et al. 2012]

suggest combining techniques from machine learning, matching, and clustering to address integration issues in data lakes.

Exercises

Problem 10.1 Compare and contrast the different approaches to storage system design in terms of scalability, ease of use (ingesting data, etc.), architecture (shared, shared-nothing, etc.), consistency scheme, fault-tolerance, meta-data management. Try to generate a comparison table in addition to a short discussion.

Problem 10.2 (*) Consider the big data management software stack (Fig. 10.1) and compare it with the traditional relational DBMS software stack, for instance, based on that in Fig. 1.9. In particular, discuss the main differences in terms of storage management.

Problem 10.3 (*) In the distributed storage layer of the big data management software stack, data is typically stored in files or objects. Discuss when to use object storage versus object storage based on the characteristics of the data, e.g., large size or small objects, high number of objects, similar records, and application requirements, e.g., easy to move across machines, scalability, fault-tolerance.

Problem 10.4 (*) In a distributed file system like GFS or HDFS, the files are divided into fixed-size partitions, called chunks. Explain the differences between the concept of chunk and horizontal partition as defined in Chap. 2.

Problem 10.5 Consider the various MapReduce implementations for equijoin given in Sect. 10.2.1.3. Compare broadcast join and repartition join in terms of generality and shuffling cost.

Problem 10.6 (*) Section 10.2.1.1 describes how the combine module is used to reduce the shuffling cost.

(a) Provide pseudo-code of such combiner function for the SQL query example given in Example 10.1.
(b) Describe how it would reduce the shuffling cost.

Problem 10.7 Consider the MapReduce implementation of the theta-join operator and its dataflow given in Fig. 10.10. Discuss the performance implication of such dataflow for equijoin (where θ is $=$).

Problem 10.8 Consider the Spark implementation of the k-means clustering algorithm presented in Example 10.3. Describe which steps of the algorithms result in shuffling of data across multiple workers.

Problem 10.9 We discussed PageRank computation in Example 10.4. A version, called personalized PageRank, computes the value of a page around a user-selected set of pages by assigning more importance to edges in the neighborhood of certain

pages that the user has identified. In this question, assume that the set of pages that the user has identified is a single page, called the *source*. It is with respect to this source page that the computation is conducted. The differences from usual PageRank are as follows:

- Recall that in PageRank when the random walk lands on a page, with probably d the walk jumps to a random page in the graph. In personalized PageRank, the jump is not to a random page, but always to the source page, i.e., with probability d, the walk jumps back to the *source*.
- When computation is initialized, instead of assigning equal PageRank values to all of the vertices of the graph, *source* is assigned a rank of 1, the rest of the pages are assigned a rank of 0.

Compute (by hand) the personalized PageRank of the web graph shown in Fig. 10.21.

Problem 10.10 (**) Implement personalized PageRank as defined in Problem 10.9 in MapReduce (use Hadoop).

Problem 10.11 (**) Implement personalized PageRank as defined in Problem 10.9 in Spark.

Problem 10.12 (**) Consider a DSPS as described in Sect. 10.3. Describe an algorithm to implement at-least-once delivery semantics.

Problem 10.13 (**) Consider the DSS as described in Sect. 10.3.

(a) Design an intra-operator parallel version of the filter streaming operator.
(b) Design an intra-operator parallel version of the aggregate operator. Hint: Unlike the filter operator above, the aggregate operator is stateful. What do you need to take into account that was not necessary for the (stateless) filter operator? How is data split across instances of the operator? What shall be done at the output of the previous operator to guarantee each streaming tuple goes to the right instance?

Problem 10.14 (**) Design a sliding window join operator for two streams. Is it deterministic? Why not? Can you propose an alternative design of the operator that guarantees determinism independently of the relative speed/interleaving of the input streams?

Problem 10.15 Consider the vertex-centric programming model for graph processing as described in Sect. 10.4.3. Compare BSP and GAS computation models in terms of

(a) generality and expressiveness of graph algorithms and
(b) performance optimizations

Problem 10.16 (**) Give an algorithm for personalized PageRank as defined in Problem 10.9 using vertex-centric BSP model.

Problem 10.17 (*) Example 10.8 describes an iterative label-propagation based algorithm for finding connected components of the input graph in the vertex-centric programming model. Consider a streaming application where each incoming tuple in the stream represents an undirected edge of the input graph. Design an incremental algorithm that finds connected components of the graph that is formed by the edges in the stream.

Problem 10.18 (*) Consider the greedy vertex-cut edge placement heuristics defined in Sect. 10.4.10. Such greedy partitioning heuristics are known to suffer from load-imbalance if the edge steam is presented in some adversarial order.

(a) Create an ordering of the vertices in Fig. 10.23 such that the described heuristics results in highly imbalanced partitioning, i.e., entire set of edges is assigned to a single partition.
(b) Propose a strategy to mitigate the load imbalance in case of such adversarial stream ordering.

Problem 10.19 (*) A data lake resembles a data warehouse (see Chap. 7), but for unstructured schemaless data, e.g., stored in HDFS. Consider the Spark big data system, which provides SQL access to HDFS data (through SparkSQL) and many other data sources. Is Spark sufficient to build a data lake? What functionality would be missing?

Problem 10.20 (**) Recent parallel DBMSs used in modern data warehousing have added support for external tables (see, for instance, Polybase in Chap. 11), which make the correspondence with the HDFS files and can be manipulated together with native relational tables using SQL queries. On the other hand, data lakes provide access to external data sources, e.g., SQL, NoSQL, etc., using wrappers, e.g., Spark connectors. Compare the two approaches (data lake and modern data warehouse) from the point of view of data integration. What is similar? What is different?

Chapter 11
NoSQL, NewSQL, and Polystores

For managing data in the cloud, one can always rely on a relational DBMS. All relational DBMSs have a distributed version, and most of them operate in the cloud. However, these systems have been criticized for their "one size fits all" approach. Although they have been able to integrate support for all kinds of data (e.g., multimedia objects, documents) and new functions, this has resulted in a loss of performance, simplicity, and flexibility for applications with specific, tight performance requirements. Therefore, it has been argued that more specialized DBMS engines are needed. For instance, column-oriented DBMSs which store column data together rather than rows in traditional row-oriented relational DBMSs have been shown to perform more than an order of magnitude better on OLAP workloads. Similarly, data stream management systems (see Sect. 10.3) are specifically architected to deal efficiently with data streams.

Thus, many different data management solutions have been proposed, specialized for different kinds of data and tasks, and able to perform orders of magnitude better than traditional relational DBMSs. Examples of new data management technologies include distributed file systems and parallel data processing frameworks for big data (see Chap. 10).

An important kind of new data management technology is NoSQL, meaning "Not Only SQL" to contrast with the "one size fits all" approach of relational DBMS. NoSQL systems are specialized data stores that address the requirements of web and cloud data management. The term data store is often used as it is quite general, including not only DBMSs but also simpler file systems or directories. As an alternative to relational DBMSs, NoSQL systems support different data models and different languages other than standard SQL. They also emphasize scalability, fault-tolerance, and availability, sometimes at the expense of consistency. There are different types of NoSQL systems, including key-value, document, wide column, and graph, as well as hybrid (multimodel or NewSQL).

These new data management technologies have led to a rich offering of services that can be used to build cloud data-intensive applications that can scale and exhibit

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_11

high performance. However, this has also led to a wide diversification of data store interfaces and the loss of a common programming paradigm. Thus, this makes it very hard for a user to build applications that use multiple data stores, e.g., distributed file system, relational DBMS, and NoSQL DBMS. This has motivated the design of *polystores*, also called multistore systems, that provide integrated access to a number of cloud data stores through one or more query languages.

This chapter is organized as follows: Section 11.1 discusses the motivations for NoSQL systems, in particular, the CAP theorem that helps to understand the trade-off between different properties. We then introduce the different types of NoSQL systems: key-value in Sect. 11.2, document in Sect. 11.3, wide column in Sect. 11.4, graph in Sect. 11.5. Section 11.6 presents the hybrid systems, i.e., multimodel NoSQL systems and NewSQL DBMSs. Section 11.7 discusses the polystores.

11.1 Motivations for NoSQL

There are several (complementary) reasons that have motivated the need for NoSQL systems. The first obvious one is the "one size fits all" limitation of relational DBMSs, which we discussed above.

A second reason is the limited scalability and availability of the early database architecture that has been used in the cloud. This architecture is a traditional 3-tier architecture with web clients accessing a data center that features a load balancer, web/application servers, and database servers. The data center typically uses a shared-nothing cluster, which is the most cost-effective solution for the cloud. For a given application, there is one database server, typically a relational DBMS, which provides fault-tolerance and data availability through replication. As the number of web clients increases, it is relatively easy to add web/application servers, typically using virtual machines, to absorb the incoming load and scale up. However, the database server becomes the bottleneck, and adding new database servers would require to replicate the entire database, which would take much time. In a shared-nothing cluster, a solution could be to use a parallel relational DBMS to provide scalability. However, this solution would be appropriate only for OLAP (read-intensive) workloads (see Sect. 8.2) and not cost-effective as parallel relational DBMSs are high-end products.

A third reason that has been used to motivate the need for NoSQL systems is that supporting strong database consistency as relational DBMSs do, i.e., through ACID transactions, hurts scalability. Therefore, some NoSQL systems have relaxed strong database consistency in favor of scalability. An argument to support this approach has been the famous CAP theorem from distributed systems theory. However, the argument is simply wrong as the CAP theorem has nothing to do with database scalability: it is related to replication consistency in the presence of network partitioning. Furthermore, it is quite possible to provide both strong database consistency and scalability, as some NewSQL systems do (see Sect. 11.6.2).

The CAP theorem states that a distributed data store with replication can only provide two out of the following three properties: (C) consistency, (A) availability, and (P) partition tolerance. Note there is no (S) scalability. These properties are defined as follows:

- Consistency: all nodes see the same data values at the same time, i.e., each read request returns the last written value. This property corresponds to linearizability (consistency over individual operations) and not serializability (consistency over groups of operations).
- Availability: any replica has to reply to any received request.
- Partition tolerance: the system continues to operate despite a partitioning of the network due to a failure.

A common misunderstanding of the CAP theorem is that one of these properties needs to be abandoned. However, only in the case of a network partitioning does one have to choose between consistency and availability.

NoSQL (Not Only SQL) is an overloaded term, which leaves much room for interpretation and definition. For instance, it can be applied to the early hierarchical and network DBMS, or the object or XML DBMSs. However, the term first appeared in the late 1990s for the new data stores built to address the requirements of web and cloud data management. As an alternative to relational databases, they support different data models and languages other than standard SQL. These systems typically emphasize scalability, fault-tolerance, and availability, sometimes at the expense of consistency.

In this chapter, we introduce the four main categories of NoSQL systems based on the underlying data model, i.e., key-value, wide column, document, and graph. We also consider the hybrid data stores: multimodel, to combine multiple data models in one system, and NewSQL, to combine the scalability of NoSQL with the strong consistency of relational DBMS. For each category, we illustrate with a representative system.

11.2 Key-Value Stores

In the key-value data model, all data is represented as key-value pairs, where the key uniquely identifies the value. Key-values stores are schemaless, which yields great flexibility and scalability. They typically provide a simple interface such as put (key, value), value=get (key), delete (key).

An extended form of key-value store is able to store records, as lists of attribute-value pairs. The first attribute is called major key or primary key, e.g., a social security number, and uniquely identifies the record among a collection of records, e.g., people. The keys are usually sorted, which enables range queries as well as ordered processing of keys.

A popular key-value store is Amazon DynamoDB, which we introduce below.

11.2.1 DynamoDB

DynamoDB is used by some of Amazon's core services that need high availability and key-based data access. Examples of services are those that provide shopping carts, seller lists, customer preferences, and product catalogs. To achieve scalability and availability, Dynamo sacrifices consistency under some failure scenarios and uses a synthesis of well-known P2P techniques (see Chap. 9) in a shared-nothing cluster.

DynamoDB stores data as database tables, which are collections of individual items. Each item is a list of attribute-value pairs. An attribute value can be of type scalar, set, or JSON. The items are analogous to rows in a relational table, and the attributes are analogous to columns. However, since attributes are self-describing, there is no need for a relational schema. Furthermore, items may be heterogeneous, i.e., with different attributes.

The original design of DynamoDB provides the P2P distributed hash table (DHT) abstraction (see Sect. 9.1.2). The primary key (the first attribute) is hashed over the different partitions, which allows efficient key-based read and write operations to an item as well as load balancing. More recently, DynamoDB has been extended to support composite primary keys, which are made of two attributes. The first attribute is the hash key and is not necessarily unique. The second attribute is the range key and allows range operations within the hash partition corresponding to the hash key. To access database tables, DynamoDB provides a Java API with the following operations:

- PutItem, UpdateItem, DeleteItem: adds, updates, or deletes an item in a table based on its primary key (either a hash primary key or a composite primary key).
- GetItem: returns an item based on its primary key in a table.
- BatchGetItem: returns all items that have the same primary key, but in several tables.
- Scan: returns all items in a table.
- Range query: returns all items based on a hash key and a range on the range key.
- Indexed query: returns all items based on an indexed attribute.

Example 11.1 Consider table Forum_Thread in Fig. 11.1. This table is made of homogeneous items that have four attributes: Forum, Subject, Date of last post, and Tags. It has a composite key, made of a hash key (Forum) and a range key (Subject). An example of primary key access is

GetItem(Forum="EC2," Subject="xyz")

which returns the last item. An example of range query is

Query(Forum="S3," Subject >"ac")

which returns the second and third items. ♦

DynamoDB builds an unordered hash index on the hash key, i.e., a DHT, and a sorted range index on the range (ordered) key. Furthermore, DynamoDB provides

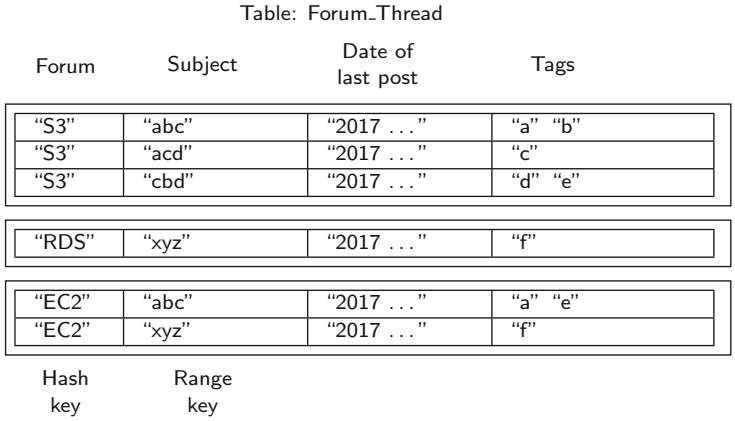


Fig. 11.1 DynamoDB table example

two kinds of secondary indexes to allow fast access to items based on nonkey attributes: local secondary indexes, to retrieve items within a hash partition, i.e., items that have the same value in their hash key, and global secondary indexes, to retrieve items in the whole DynamoDB table.

Data is partitioned and replicated across multiple cluster nodes in several data centers, which provides both load balancing and high availability. Data partitioning relies on consistent hashing , a popular hashing scheme that has been used in DHTs with ring geometry, e.g., Chord (see Sect. 9.1.2). The DHT is represented as a one-dimensional circular identifier space, i.e., a "ring," where each node in the system is assigned a random value within this space which represents its position on the ring. Each item is assigned to a node by hashing the item's key to yield its position on the ring, and then finding the first node clockwise with a position higher than the item position. Thus, each node becomes responsible for the interval in the ring between its predecessor node and itself. The main advantage of consistent hashing is that the addition (joins) and removal (leaves/failures) of nodes only affect the nodes' immediate neighbor, with no impact on other nodes.

DynamoDB also exploits consistent hashing to provide high availability, by replicating each item at n nodes, n being a system-configured parameter. Each item is assigned a coordinator node, as described above, and is replicated on the $n-1$ clockwise successor nodes. Thus, each node is responsible for the interval of the ring between its nth predecessor and itself.

Example 11.2 Figure 11.2 shows a ring with 6 nodes, each named by its position (hash value). For instance, node B is responsible for the hash value interval (A,B] and node A for the interval (F,A]. The put(c,v) operation yields a hash value for key c between A and B, so node B becomes responsible for the item. In addition, assuming replication parameter $n = 3$, the item would be replicated at nodes C and

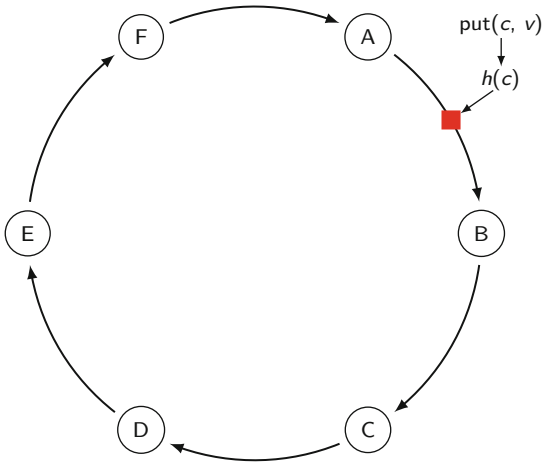


Fig. 11.2 DynamoDB consistent hashing. Node B is responsible for the hash value interval (A,B]. Thus, item (c,v) is assigned to node B

D. Thus, node D will store the items whose keys fall in the intervals (A, B], (B, C], and (C, D]. ♦

DynamoDB trades strong data consistency for scalability and availability, but with different ways of controlling consistency. It provides eventual consistency of replicas (see Sect. 6.1.1), which is achieved by an asynchronous update propagation protocol and a gossip-based distributed failure detection protocol.

Write consistency in a concurrent multiuser environment can be controlled through conditional writes. By default, the write operations (PutItem, UpdateItem, DeleteItem) will overwrite an existing item that has the given primary key. A conditional write specifies a condition over the item's attributes to succeed. For instance, a condition for a PutItem to succeed is that there is not already an item with the same primary key. Thus, conditional writes are useful in case of concurrent updates.

DynamoDB supports eventually consistent and strongly consistent reads. By default, reads are eventually consistent, i.e., may not return the latest data which is being asynchronously replicated. Strongly consistent reads return the most up-to-date data, which may not be possible in the case of network failures, because not all replica updates have been propagated.

11.2.2 Other Key-Value Stores

Other popular key-value stores are Cassandra, Memcached, Riak, Redis, Amazon SimpleDB, and Oracle NoSQL Database. Many systems provide further extensions

so that we can see a smooth transition to wide column store and document stores, which we discuss next.

11.3 Document Stores

Document stores are advanced key-value stores, where keys are mapped to values of document type, such as JSON, YAML, or XML. Documents are typically grouped into collections, which play a role similar to relational tables. However, documents are different from relational tuples. Documents are self-describing, storing data and metadata (e.g., markups in XML, field names in JSON objects) altogether and can be different from one another within a collection. Furthermore, the document structures are hierarchical, using nested constructs, e.g., nested objects and arrays in JSON. Thus, modeling a database using documents requires fewer collections than with (flat) relational tables, and also avoids expensive join operations.

In addition to the simple key-value interface to retrieve documents, document stores offer an API or query language that retrieve documents based on their contents. Document stores make it easier to deal with change and optional values, and to map into program objects. This makes them attractive for modern web applications, which are subject to continual change, and where speed of deployment is important.

A popular NoSQL document store is MongoDB, which we introduce below.

11.3.1 MongoDB

MongoDB is an open source system written in C++. It provides a JSON-based data model for documents, schema flexibility, high availability, fault-tolerance, and scalability in shared-nothing clusters.

MongoDB stores data as documents in BSON (Binary JSON), a binary encoded serialization of JSON to include additional types such as binary, int, long, and floating point. BSON documents contain one or more fields, and each field has a name and contains a value of a specific data type, including arrays, binary data, and subdocuments. Each document is a BSON object, i.e., with multiple fields, and uniquely identified by its first field of type ObjectId, whose value is automatically generated by MongoDB.

Documents that have a similar structure are organized as collections, like relational tables, document fields being similar to columns. However, documents in the same collection can have different structures, since there is no imposed schema.

MongoDB provides a rich query language to update and retrieve BSON data using functions expressed in JSON. Representing queries as JSON allows unifying both the way data is stored and manipulated. The query language can be used with APIs in various programming languages, such as Java, PHP, JavaScript, and Scala.

Since queries are implemented as methods or functions within the API of a specific programming language, the integration within application programs is natural and simple for developers.

MongoDB supports many kinds of queries to insert, update, delete, and retrieve documents. A query may return documents, subsets of specific fields within documents, or complex aggregations of values from many documents. Queries can also include user-defined JavaScript functions. The general form of a query is

db.collection.function (JSON expression)

where *db* is a global variable corresponding to the database connection, and *function* is the database operation applied to the collection. The JSON expression can be arbitrary and used as a criterion to select data. The different kinds of queries are

- Insert, delete, and update operations on documents. For delete and update operations, the JSON expression specifies the criteria to select the documents.
- Exact-match queries return results based on the equality of a value for a field in the documents, typically the primary key.
- Range queries return results based on values of a field in a given range.
- Geospatial queries return results based on proximity, intersection, and inclusion of geographical objects, such as point, line, circle, or polygon in GeoJSON format.
- Text search queries return results in relevance order based on text arguments using Boolean operators.
- Aggregation queries return aggregated values from a collection with operators such as count, min, max, and average. Furthermore, documents from two collections can be combined using a left outer join operation.

Example 11.3 Consider collection Posts in Fig. 11.3. Each post item in the collection is uniquely identified by its key of type ObjectId (generated by MongoDB) and its value is a JSON object, with nested arrays such as tags and comments. Examples of update operations are

db.posts.insert(author:“alex,” title:“No Free Lunch”)
db.posts.update(author:“alex,” $set:age:30)
db.posts.update(author:“alex,” $push:tags:“music”)

_id: ObjectId(“abc”)	author: “alex”, title: “No Free Lunch”, text: “This is . . .”, tags: [“business”,“ramblings”], comments: [{who:“jane”, what:“I agree.”},{who:“joe”, what:“No. . .”}]
_id: ObjectId(“abd”)	A post by X
_id: ObjectId(“acd”)	A post by Y

Unique key generated by MongoDB

Value = JSON object with nested arrays

Fig. 11.3 MongoDB posts collection example

where $set (sets a specified value for a field) and $push (appends a specified value to an array) are MongoDB instructions within JSON.

The following queries:

```
db.posts.find(author:“alex”)
db.posts.find(comments.who:“jane”)
```

are exact-match queries. The first one returns all the posts from Alex, while the second one returns all the posts for which Jane made a comment ♦

To provide efficient access to data, MongoDB includes support for many kinds of secondary indexes that can be declared on any field in the document, including fields within arrays. The different kinds of indexes are:

- Unique indexes, where the value of the indexed field is enforced to be unique.
- Multikey (compound) indexes on multiple fields.
- Array indexes for array fields, with a separate index entry for each array value.
- TTL indexes that should expire automatically after a certain Time-to-Live (TTL).
- Geospatial indexes to optimize queries based on proximity, intersection, and inclusion of geographical objects, such as point, line, circle, or polygon.
- Partial indexes that are created for a subset of documents that satisfy a condition specified by the user.
- Sparse indexes that index only documents that contain the specified field.
- Text search indexes use language-specific linguistic rules to optimize text search queries.

To scale out in shared-nothing clusters, MongoDB supports different kinds of data partitioning (or sharding) schemes: hash-based, range-based, and location-aware (whereby the user specifies key-ranges and associated nodes). High availability is provided through a variation of primary-copy replication, called replica sets, with asynchronous update propagation. If a master goes down, one of the replicas becomes the new master and continues to accept update operations. A MongoDB cluster consists of: shards (data partitions) where each shard can be a unit of replication (a replica set); mongos that act as query processors between client applications and the cluster; and configuration servers that store metadata and configuration settings for the cluster. Applications can optionally read from secondary replicas, where data is eventually consistent.

MongoDB has recently introduced support for ACID transactions on multiple documents, in addition to single document transactions. This is achieved through snapshot isolation. One or more fields in a document may be written in a single transaction, including updates to multiple subdocuments and elements of an array. Multidocument transactions can be used across multiple collections, databases, documents, and shards.

MongoDB also provides an option called *write concern* that allows users to specify a guarantee level for reporting the success of a write operation, i.e., with a desired trade-off between level of persistence and performance, on database replicas. There are four guarantee levels for clients to adjust the control of a

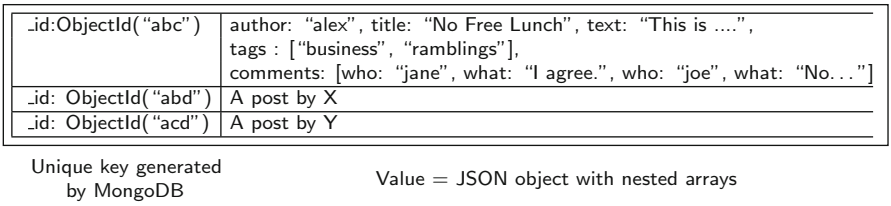


Fig. 11.4 MongoDB architecture

write operation, in order from weakest to strongest. These are unacknowledged (no guarantee), acknowledged (the write to disk has been done), journaled (the write has been recorded in the log), and replica acknowledged (the write has been propagated to replicas).

MongoDB's architecture fits in the big data management software stack (see Fig. 10.1). It supports pluggable storage engines, e.g., HDFS, or in-memory, for dealing with unique application demands, interfaces with big data frameworks like MapReduce and Spark, and supports third-party tools for analytics, IoT, mobile applications, etc. (see Fig. 11.4). It makes extensive use of main memory to speed up database operations and native compression, using its storage engine (WiredTiger).

11.3.2 Other Document Stores

Other popular document stores are AsterixDB, Couchbase, CouchDB, and RavenDB , which all support the JSON data model in a scalable shared-nothing cluster architecture. However, AsterixDB and Couchbase support a dialect of SQL++, an elegant extension of SQL with a few simple features to query JSON data. Couchbase's SQL++ dialect is called N1QL (Non-first normal form Query Language, pronounced "nickel"). Couchbase supports restricted transactions (atomic document writes) and allows to trade some properties for performance, e.g., relaxing durability by acknowledging write operations done in memory and then asynchronously writing to disk. In addition to the Couchbase server, which is used for queries and transactions, the Couchbase platform has an analytics service that allows online data analytics, without hurting transaction performance, and also without requiring a different data model or (as a result) ETL. This is kind of Hybrid Transaction and Analytics Processing (HTAP) for NoSQL (see the introduction of HTAP in Sect. 11.6.2). The implementation of this analytics service is based on AsterixDB's storage engine and parallel query processor. AsterixDB is a high-performance JSON document store with a combination of techniques from parallel databases and document databases.

11.4 Wide Column Stores

Wide column stores combine some of the nice properties of relational databases (e.g., representing data as tables) with the flexibility of key-value stores (e.g., schemaless data within columns). Each row in a wide column table is uniquely identified by a key and has a number of named columns. But unlike in a relational table, where columns can only contain atomic values, a column can be wide and contain multiple key-value pairs.

Wide column stores extend the key-value store interface with more declarative constructs that allow scans, exact-match and range queries over column families. They typically provide an API for these constructs to be used in a programming language. Some systems also provide an SQL-like query language, e.g., Cassandra Query Language (CQL).

At the origin of the wide column stores is Google Bigtable, which we introduce below.

11.4.1 Bigtable

Bigtable is a wide column store for shared-nothing clusters. Bigtable uses Google File System (GFS) for storing structured data in distributed files, which provides fault-tolerance and availability (see Sect. 10.1 on block-based distributed file systems). It also uses a form of dynamic data partitioning for scalability. Like GFS, it is used by popular Google applications, such as Google Earth, Google Analytics, and Google+.

Bigtable supports a simple data model that resembles the relational model, with multivalued, timestamped attributes. We briefly describe this model as it is the basis for Bigtable implementation that combines aspects of row-store and column-store DBMS. For consistency with the concepts we have used so far, we present the Bigtable data model as a slightly extended relational model.[1]

A Bigtable instance is a collection of (key, value) pairs where the key identifies a row and the value is the set of columns, organized as column families. Bigtable sorts its data by keys, which helps clustering rows of the same range in the same cluster node.

Each row in a Bigtable is uniquely identified by a *row key*, which is an arbitrary string (of up to 64KB in the original system). Thus, a row key is like a single attribute key in a relation. A Bigtable is always sorted by row keys, A row can have multiple *column families*, which form the unit of access control and storage. A column family is a set of columns of the same type. To create a Bigtable, only the table name and the column family names need to be specified. However, within a column family, arbitrary columns (of same type) can be dynamically added.

[1] In the original proposal, a Bigtable is defined as a multidimensional map, indexed by a row key, a column key, and a timestamp, each cell of the map being a single value (a string).

Row key	Name	Email	Web page
100	"Prefix": "Dr." "Last": "Dobb"	"email: gmail.com": "dobb@gmail.com"	<!DOCTYPE html PUBLIC... >
101	"First": "Alice" "Last": "Martin"	"email: gmail.com": "amartin@gmail.com" "email: free.fr": "amartin@free.fr"	<!DOCTYPE html PUBLIC... >

Fig. 11.5 A Bigtable with 3 column families and 2 rows

To access data in a Bigtable, it is necessary to identify columns within column families using *column keys*. A column key is a fully qualified name of the form **column-family-name:column-name**. The column family name is like a relation attribute name. The column name is like a relation attribute value, but used as a name as part of the column key to represent a single item. This allows the equivalent of multivalued attributes within a relation. In addition, the data identified by a column key within a row can have multiple versions, each identified by a timestamp (a 64 bit integer).

Example 11.4 Figure 11.5 shows an example of a Bigtable with 3 column families and 2 rows, as a relational style representation. The Name and EMail column families have heterogeneous columns. To access a column value, the row key and column key must be specified, e.g., row key = "111" and column key = "Email:gmail.com" which yields "am@gmail.com." ♦

Bigtable provides a basic API for defining and manipulating tables, within a programming language such as C++. It also provides functions for changing table, and column family metadata, such as access control rights. The API offers various operators to write and update values, and to iterate over subsets of data, produced by a scan operator. There are various ways to restrict the rows, columns, and timestamps produced by a scan, as in a relational select operator. However, there are no complex operators such as join or union, which need to be programmed using the scan operator.

Transactional atomicity is supported for single row updates only. Thus, for more complex multiple row updates, it is up to the programmer to write the appropriate code to control atomicity using an interface for batching writes across row keys at the clients.

To store a table in GFS, Bigtable uses range partitioning on the row key. Each table is divided into partitions called *tablets*, each corresponding to a row range. Partitioning is dynamic, starting with one tablet (the entire table range) that is subsequently split into multiple tablets as the table grows. To locate the (user) tablets in GFS, Bigtable uses a metadata table, which is itself partitioned in metadata tablets, with a single root tablet stored at a master server, similar to GFS's master. In addition to exploiting GFS for scalability and availability, Bigtable uses various techniques to optimize data access and minimize the number of disk accesses, such as compression of column families, grouping of column families with high locality of access, and aggressive caching of metadata information by clients.

Bigtable relies on a highly available and persistent distributed lock service called Chubby . A Chubby service consists of five active replicas, one of which is elected to be the master and actively serves requests. Bigtable uses Chubby for several tasks: to ensure that there is at most one active master at any time; to store the bootstrap location of Bigtable data; to discover tablet servers and finalize tablet server removals; and to store Bigtable schemas. If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable.

11.4.2 Other Wide Column Stores

There are popular open source implementations of Bigtable, such as Hadoop Hbase, a popular Java implementation that runs on top of HDFS, and Cassandra that combines techniques from Bigtable and DynamoDB.

11.5 Graph DBMSs

We introduced graph analytics in Chap. 10, where the entire graph can be processed multiple time until a fixpoint is reached. In contrast, graph DBMSs support queries that are not iterative and might access only a portion of the graph, allowing indexes to be effective. Graph databases represent and store data directly as graphs which allows easy expression and fast processing of graph-like queries, e.g., computing the shortest path between two elements in the graph. This is much more efficient than with a relational database where graph data need be stored as separated tables and graph-like queries require repeated, expensive join operations. Graph DBMSs typically provide a powerful graph query language. They have become popular with data-intensive web-based applications such as social networks and recommender systems.

Graph DBMSs can provide a flexible schema by specifying vertex and edge types with their properties. This facilitates the definition of indexes to provide fast access to vertices, based on some property value, e.g., a city name, in addition to structural indexes. Graph queries can be expressed using graph operators through a specific API or a declarative query language.

As we have seen above, in order to scale up to very large databases, key-value, document, and wide column stores partition data across a number of cluster nodes. The reason that such data partitioning works well is that it deals with individual items. However, graph data partitioning is much more difficult since the problem of optimally partitioning a graph is NP-complete (see Sect. 10.1). In particular, recall that we may get items in different partitions that are connected by edges. Traversing such interpartition edges incurs communication overhead which hurts the performance of graph traversal operations. Thus, the number of interpartition edges should be minimized. But, at the same time, this may produce unbalanced partitions.

In the following, we illustrate graph DBMSs with Neo4j, which is a popular one with many deployments.

11.5.1 Neo4j

Neo4j is a commercial open source system introduced as a scalable and high-performance graph DBMS with native graph storage and processing in shared-nothing clusters. It provides a rich graph data model with integrity constraints, a powerful query language, called Cypher, with indexes, ACID transactions, and support for high availability and load balancing.

The data model is based on directed graphs, with separated storage for edges (called relationships), vertices (called nodes), or attributes (called properties). Each node can have any number of properties, in the form of (attribute, value) pairs. A relationship must have a type, which gives semantics, and a direction from one node to another (or to itself). An important capability of Neo4j is that relationships can be traversed in both directions with the same performance. This simplifies the modeling of graph DBMSs since there is no need to create two different relationships between nodes, if one implies the other, e.g., a mutual relationship such as friend (whose reverse is also friend) or a 1-1 relationship such as owns (whose reverse is owned-by).

Updating a graph involves updating nodes, relationships, and properties, which needs to be done in a consistent manner. This is done using ACID transactions.

Example 11.5 Figure 11.6 shows an example of a simple graph for a social network. The friend relationship from Bob to Mary (meaning "Bob is a friend of Mary") is sufficient to represent the mutual relationship (hopefully, Mary is also a friend of Bob). It could have also been represented the other way (from Mary to Bob). This makes the graph model simple.

The following transaction, using the Java API, creates nodes Bob and Mary, their properties, and the friend relationship from Bob to Mary.

```
Transaction tx = neo.beginTx();
Node n1 = neo.CreateNode();
n1.setProperty(''name'', ''Bob'');
n1.setProperty(''age'', 35);
Node n2 = neo.createNode();
n2.setProperty(''name'', ''Mary'');
n1.setProperty(''age'', 29);
n1.setProperty(''job'', ''engineer'');
n1.createRelationshipTo(n2, RelTypes.friend);
tx.Commit();
```

♦

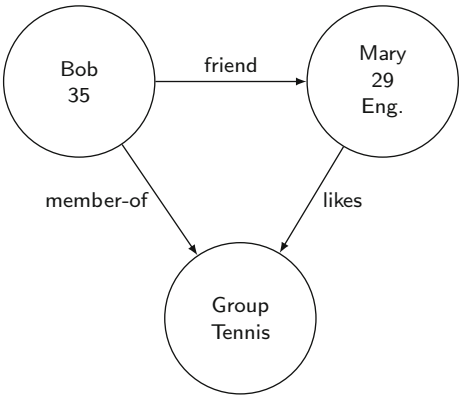


Fig. 11.6 Example of Neo4j graph

Neo4j imposes no schema on the graph, which provides much flexibility in allowing data to be created without having to fully understand upfront the way data will be used. However, schemas in other data models (relational, object, XML, etc.) have proved to be useful for database consistency and efficient query processing. Therefore, Neo4j introduces an optional schema based on the concept of labels and a data definition language to manipulate it. A label is like a tag, useful to group similar nodes. A node may be assigned any number of labels, e.g., person, student, and user. This allows Neo4j to query only some subset of the graph, e.g., the students in a given city. Labels are used when defining integrity constraints and indexes. Integrity constraints can be defined on nodes and relationships, e.g., unique node property, node property existence, or relationship property existence.

Indexes can be created based on labels and combinations of properties. They provide efficient node lookup, which is an important operation used to start graph traversals at specific nodes based on predicates that involve labels and properties. The Neo4j-spatial library also provides n-dimensional polygon indexes to optimize geospatial queries.

To query and manipulate graph data, Neo4j provides a Java API and a query language, called Cypher. The Java API gives the Java programmer access to the graph operations of nodes, relationships, properties, and labels, with ACID transactions. This API provides tight integration with the programming language.

Cypher is a powerful graph DBMS query language with SQL flavor. It can be used to manipulate graph data. For instance, the transaction in Example 11.5 could be simply written by the following **CREATE** statement:

```
CREATE (:Person {name:''Bob'', age:35}) <- [:FRIEND]
   -(:Person {name:''Mary'', age:29, job:''engineer''})
```

Cypher is easy to use through graph pattern matching: the user specifies a graph pattern as when drawing a diagram and queries the database to

find the data that matches the pattern. Cypher provides clauses such as **MATCH**, **MERGE**, **WHERE**, **RETURN**, which can manipulate node variables (like tuple variables in SQL) and a few others. **MATCH** does graph pattern matching. Nodes are expressed with parentheses, and relationships using pairs of dashes with greater-than or less-than signs to indicate relationship direction. Node and relationship property key-value pairs are then specified within curly braces. **MERGE** is useful to create or match graphs. WHERE specifies a predicate on nodes and **RETURN** the result nodes, relationships, and properties to be returned.

Example 11.6 The following Cypher query returns all direct and indirect friends of Bob whose name starts with "M." The **MATCH** expression defines a recursive pattern for the friend-of-friend relationship with the node variables bob and follower. The **WHERE** expression looks up the node whose name is "Bob," which may be done through an index, and selects its follower nodes of whose name starts with "M." The **RETURN** expressions return all pairs of nodes bound to the bob and follower variables.

```
MATCH bob-[:FRIEND]-> ()-[:FRIEND] -> follower
WHERE bob.name = ''Bob'' AND follower.name =~ ''M.*''
RETURN bob, follower.name
```

♦

Neo4j provides a cost-based query compiler that produces optimized query plans for Cypher queries, both read-only and update queries. The compiler first performs logical rewriting of the query plan using unnesting, merging, and simplification of various parts of the query. Then, based on statistical information on index and label selectivity, it chooses the best access methods for operators and produces the query execution plan, using nested iterators to be executed in a pipelined, top-down fashion.

Neo4j provides extensive support for high availability through full replication both at the cluster level and across data centers. Within a cluster, a variation of multimaster replication (see Chap. 6), called causal clustering, is used to scale out to large configurations. Causal clustering supports *causal consistency*, a consistency model that guarantees that causally related operations are seen in the same order by all client applications. Thus, a client application is guaranteed to read its own writes. Causal clustering architecture is shown in Fig. 11.7, with three kinds of cluster nodes: application server, core server, and read server. Application servers execute application code, and issue write transactions to core servers and read queries to read servers. The core servers replicate all transactions asynchronously using the Raft protocol. Raft ensures transaction durability using a majority of the core servers to acknowledge a write before it is safely committed. Core servers replicate transactions to read servers by shipping transaction logs. The Raft protocol is also used to implement various replication architectures across data centers to support disaster recovery.

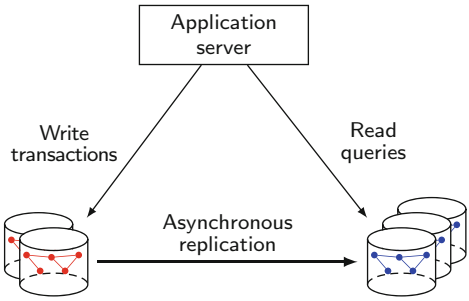


Fig. 11.7 Neo4j causal clustering architecture

In addition to high availability, causal clustering allows scaling out graph queries using many read servers. To optimize RAM memory utilization, Neo4j employs cache-based sharding, which mandates that all queries from the same user always be sent to the same read server. This naturally improves locality of reference at each read server, and allows scaling to very large graphs.

From the discussion above, it follows that the maximum size of a database graph is constrained to that of a core server disk. This constraint makes it possible to provide linear performance for path traversals while at the same time requiring to push compact graph storage to the limits. Neo4j uses dynamic pointer compression to expand the available address space as needed while allowing locating a node's adjacent nodes and relationships via a pointer hop. Finally, Neo4j's separation of storage for nodes, relationships, and properties allows further optimization. This allows the first two stores to keep only basic information and have fixed size node and relationship records, which yields efficient $O(1)$ path traversals. The property store allows for dynamic length records.

11.5.2 Other Graph Databases

Other popular graph DBMSs are Infinite Graph, Titan, GraphBase, Trinity, and Sparksee.

11.6 Hybrid Data Stores

Hybrid data stores combine capabilities typically found in different data stores and DBMS. We distinguish between multimodel NoSQL systems and NewSQL DBMSs.

11.6.1 Multimodel NoSQL Stores

Multimodel NoSQL systems are designed to reduce the need to deal with multiple systems when building complex applications. We illustrate multimodel NoSQL stores with OrientDB, a popular NoSQL data store that combines concepts from object-oriented and NoSQL document and graph data models. Other popular multimodel systems are ArangoDB and Microsoft Azure Cosmos DB.

OrientDB originated as a Java implementation of the storage layer of the Orient Object-Oriented DBMS (initially written in C++) for shared-nothing clusters. It provides a rich data model with schemas, a powerful SQL-based query language, optimistic ACID transactions, and support for high availability and load balancing.

The data model is a graph data model, with direct connections between records. There are four types of records: Document, RecordBytes (binary data), Vertex, and Edge. When OrientDB generates a record, which is the smallest unit of storage, it assigns it a unique identifier, called Record ID.

The query language is an extension of SQL with graph path traversals. It supports different kinds of indexes: SB-Tree, which is the default index; hashed index for efficient exact-match queries; Lucene full-text index for text-based search; and Lucene spatial index for spatial queries.

Schema management follows from the object-orientation with class inheritance. A class defines a set of similar records and can be schemaless, schema-full (as in object-oriented databases), or schema-hybrid. The schema-hybrid mode enables classes to define some attributes, but some records can have specific attributes. Class inheritance is based on structure, i.e., a subclass extends a parent class, inheriting all of its attributes.

Classes are the basis for clustering and partitioning records on multiple nodes. Each class can have one or more partitions, called clusters. When inserting a new record in a class, OrientDB selects the cluster to store it in using one of the following preconfigured strategies:

- default: selects the cluster using a default cluster identifier specified in the class;
- round-robin: arranges the clusters for the class into sequence and assigns each new record to the next cluster in order;
- balanced: checks the number of records in the clusters for the class and assigns the new record to the smallest cluster;
- local: when the database is replicated, it selects the master cluster on the current node (that is processing the insertion).

OrientDB supports multimaster replication, i.e., all nodes of the shared-nothing cluster can write to the database in parallel. Transactions are processed using optimistic multiversion concurrency control, based on the assumption that there are few update conflicts. So transactions proceed without any looking until commit time. When a transaction commits, each record version is checked to see if there are conflicting updates from another transaction, which may yield to aborting some transactions.

11.6.2 NewSQL DBMSs

NewSQL is a recent class of DBMS that seeks to combine the scalability of NoSQL systems with the strong consistency and usability of relational DBMSs. The main objective is to address the requirements of enterprise information systems, which have been supported by traditional relational DBMS, but also need to be able to scale. NoSQL systems provide scalability, as well as availability, flexible schemas, and practical APIs for programming complex data-intensive applications. As we have seen in the previous sections, this is typically achieved by exploiting data partitioning in shared-nothing clusters of commodity servers and relaxing database consistency. On the other hand, relational DBMSs provide strong database consistency with ACID transactions and make it easy for tools and applications to use with standard SQL. They can also scale, using their parallel version, but typically at a high price, even using a shared-nothing cluster.

An important class of NewSQL is Hybrid Transaction and Analytics Processing (HTAP) whose objective is to perform OLAP and OLTP on the same data. HTAP allows performing real-time analysis on operational data, thus avoiding the traditional separation between operational database and data warehouse and the complexity of dealing with ETL.

NewSQL systems are recent and have different architectures. However, we can identify the following common features: relational data model and standard SQL; ACID transactions; scalability using data partitioning in shared-nothing clusters; and availability using data replication.

In the rest of this section, we illustrate NewSQL with Google F1 and LeanXcale. Other kinds of NewSQL systems are Apache Ignite, CockroachDB, Esgyn, GridGain, MemSQL, NuoDB, Splice Machine, VoltDB, and SAP HANA.

11.6.2.1 F1

F1 is a NewSQL system from Google that combines the scalability of Bigtable and the consistency and usability of relational DBMSs. It has been built to support the AdWords application, which is a very large-scale update-intensive application. F1 provides a relational data model with some extensions, full SQL query support, with indexes and ad hoc querying, and optimistic transactions. It is built on top of Spanner, a scalable data storage system for shared-nothing clusters (see Sect. 5.5.1).

The F1 data model is the relational model, with a hierarchical implementation inspired from Bigtable. Several relational tables, with foreign key dependencies, can be organized as a nested relation, where the rows of each child table are clustered with the rows from their parent table based on the join key. This makes updates of multiple rows of same foreign key efficient and speeds up join processing. F1 also supports table columns with structured data types using protocol buffers, which is Google's language-neutral extensible mechanism for serializing structured data.

Using protocol buffers makes it easy to write transformations between database rows and in-memory data structures.

The primary interface is SQL, which is used for both OLTP transactions and large OLAP queries. It extends standard SQL with constructs for accessing data stored in Protocol Buffer columns. F1 also provides support for joining Spanner data with other data sources including Bigtable and CSV files. F1 supports a NoSQL key/value interface with fast access to rows, through exact-match and range queries, and updates based on primary key. Secondary indexes are stored in Spanner tables, keyed by a concatenation of the index key and the indexed table's primary key. F1 indexes can be local or global. Local indexes are local to a table hierarchy and include in their index keys the root row primary key as a prefix. Their index entries are colocated with the rows they index, which makes index updates efficient. In contrast, global indexes are global to multiple tables and do not include the root row primary key as a prefix. Thus, they cannot be colocated with the rows they index.

F1 supports both centralized and distributed query execution. Centralized execution is used for short OLTP queries, where an entire query runs on one F1 server node. Distributed execution is used for OLAP queries, with a high degree of parallelism, using hash-based repartitioning and streaming techniques.

In Sect. 5.5.1, we introduced Spanner's approach to scale out transaction management. Spanner also provides fault-tolerance, data partitioning within data centers, geographical synchronous replication across data centers, and ACID transactions. In Spanner, every transaction is assigned a commit timestamp that is used for the global total ordering of commits. F1 supports three types of transactions, on top of Spanner's strong transaction support:

- Snapshot transactions, for read-only transactions with snapshot isolation semantics, using Spanner snapshot timestamps.
- Pessimistic transactions, using ACID locking-based transactions provided by Spanner.
- Optimistic transactions, with a read phase that does not take locks, and then a validation phase that detects row-level conflicts, using rows' last modification timestamps, to decide whether to commit or abort.

11.6.2.2 LeanXcale

LeanXcale is a NewSQL/HTAP system with full SQL and polystore support in a shared-nothing cluster. It has three main subsystems: storage engine, query engine, and transactional engine, all three distributed and highly scalable (i.e., to 100s of nodes).

LeanXcale provides full SQL functionality over relational tables with JSON columns. Clients can access LeanXcale with any analytics tool using a JDBC driver. An important capability of LeanXcale is polystore access using the scripting mechanism of the CloudMdsQL query language (see Sect. 11.7.3.2). The data stores that can be accessed range from distributed raw data files (e.g., HDFS) through

parallel SQL databases, to NoSQL databases (e.g., MongoDB, where queries can be expressed as JavaScript programs).

The storage engine is a proprietary relational key-value store, KiVi, which allows for efficient horizontal partitioning of tables and indexes, based on the primary key or index key. Each table is stored as a KiVi table, where the key corresponds to the primary key of the LeanXcale table and all the columns are stored as they are in KiVi columns. Indexes are also stored as KiVi tables, where the index keys are mapped to the corresponding primary keys. This model enables high scalability of the storage layer by partitioning tables and indexes across KiVi data nodes. KiVi provides the typical put and get operations of key-value stores as well as all single table operations such as predicate-based selection, aggregation, grouping, and sorting, i.e., any algebraic operator but join. Multitable operations, i.e., joins, are performed by the query engine and any algebraic operator above the join in the query plan. Thus, all algebraic operators below a join are pushed down to the KiVi storage engine.

The query engine processes OLAP workloads over operational data, so that analytical queries are answered over real-time data. The parallel implementation of the query engine follows the single-program multiple data (SPMD) approach, which combines interquery and intraoperator parallelism. With SPMD, multiple symmetric workers (threads) on different query instances execute the same query/operator, but each of them deals with different portions of the data.

The query engine optimizes queries using two-step optimization. As queries are received, query plans are broadcast and processed by all workers. For parallel execution, an optimization step is added, which transforms a generated sequential query plan into a parallel one. This transformation involves replacing table scans with parallel table scans, and adding shuffle operators to make sure that, in stateful operators (such as group by or join), related rows are handled by the same worker. Parallel table scans divide the rows from the base tables among all workers, i.e., each worker will retrieve a disjoint subset of the rows during table scan. This is done by dividing the rows and scheduling the obtained subsets to the different query engine instances. Each worker then processes the rows obtained from subsets scheduled to its query engine instance, exchanging rows with other workers as determined by the shuffle operators added to the query plan. To process joins, the query engine supports two strategies for data exchange (shuffle and broadcast) and various join methods (hash, nested loop, etc.), performed locally at each worker after the data exchange takes place.

The query engine is designed to integrate with arbitrary data stores, where data resides in its natural format and can be retrieved (in parallel) by running specific scripts or declarative queries. This makes it a powerful polystore that can process data from its original format, taking full advantage of both expressive scripting and massive parallelism. Moreover, joins across any native datasets, such as HDFS or MongoDB, including LeanXcale tables, can be applied, exploiting efficient parallel join algorithms. To enable ad hoc querying of an arbitrary dataset, the query engine processes queries in the CloudMdsQL query language, where scripts are wrapped as native subqueries (Sect. 11.7.3.2).

In Sect. 5.5.2, we introduced LeanXcale's approach to scale out transaction management. LeanXcale scales out transactional management by decomposing the ACID properties and scaling each of them independently but in a composable manner. The transactional engine provides strong consistency with snapshot isolation. Thus, reads are not blocked by writes, using multiversion concurrency control. It supports timestamp-based ordering and conflict detection just before commit. The distributed algorithm for providing transactional consistency is able to commit transactions fully in parallel without any coordination by making a smart separation of concerns. Thus, the visibility of the committed data is separated from the commit processing. In this way, commit processing can adopt a fully parallel approach without compromising consistency that is regulated by the visibility of the committed updates. Thus, commits happen in parallel, and whenever there is a longer prefix of committed transactions without gaps the current snapshot is advanced to that point.

11.7 Polystores

Polystores provide integrated access to multiple cloud data stores such as NoSQL, relational DBMS, or HDFS. They typically support only read-only queries, as supporting distributed transactions across heterogeneous data stores is a hard problem. We can divide polystores based on the level of coupling with the underlying data stores: loosely coupled, tightly coupled, and hybrid. In this section, we introduce for each class a set of representative systems, with their architecture and query processing. We end the section with some remarks.

11.7.1 Loosely Coupled Polystores

Loosely coupled polystores are reminiscent of multidatabase systems in that they can deal with autonomous data stores, which can be accessed through the polystore common interface as well as separately through their local API. They follow the mediator-wrapper architecture with several data stores (e.g., NoSQL and relational DBMS) as depicted in Fig. 11.8. Each data store is autonomous, i.e., locally controlled, and can be accessed by other applications. The mediator-wrapper architecture, which has been used in data integration systems, can scale to a high number of data stores.

There are two main modules: one query processor and one wrapper per data store. The query processor has a catalog of data stores, and each wrapper has a local catalog of its data store. After the catalogs and wrappers have been built, the query processor can start processing input queries from the users, by interacting with wrappers. The typical query processing is as follows:

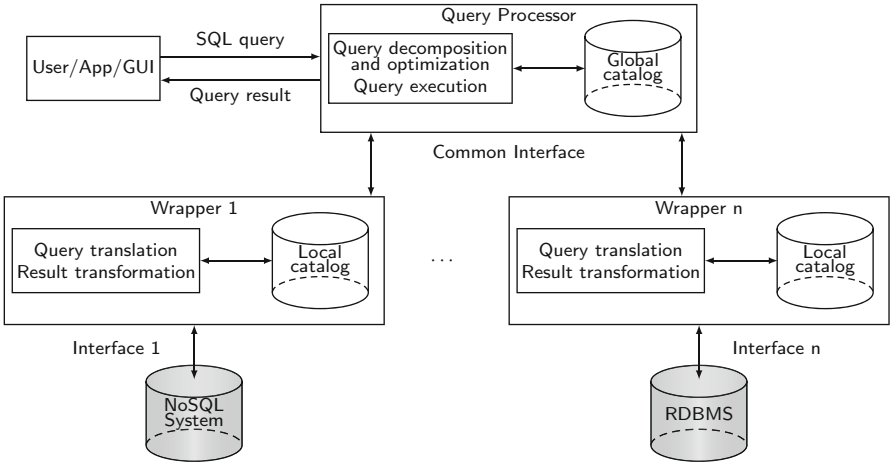


Fig. 11.8 Loosely coupled polystore architecture

1. Analyze the input query and translate it into subqueries (one per data store), each expressed in a common language, and an integration subquery.
2. Send the subqueries to the relevant wrappers, which trigger execution at the corresponding data stores and translate the results into the common language format.
3. Integrate the results from the wrappers (which may involve executing operators such as union and join), and return the results to the user. We describe below three loosely coupled polystores: BigIntegrator, Forward, and QoX.

11.7.1.1 BigIntegrator

BigIntegrator supports SQL-like queries and combines data in Bigtable data stores in the cloud with data in relational DBMS (not necessarily in the cloud). Bigtable is accessed through the Google Query Language (GQL), which has very limited query expressions, e.g., no join and only basic select predicates. To capture GQL's limited capabilities, BigIntegrator provides a query processing mechanism based on plugins, called absorber and finalizer, which enable to pre and postprocess those operations that cannot be done by Bigtable. For instance, a "LIKE" select predicate on a Bigtable or a join of two Bigtables will be processed through operations in BigIntegrator's query processor.

BigIntegrator uses the Local-As-View (LAV) approach (see Sect. 7.1.1) for defining the global schema of the Bigtable and relational data sources as flat relational tables. Each Bigtable or relational data source can contain several collections, each represented as a source table of the form "table-name_source-name," where table-name is the name of the table in the global schema and source-name is the name

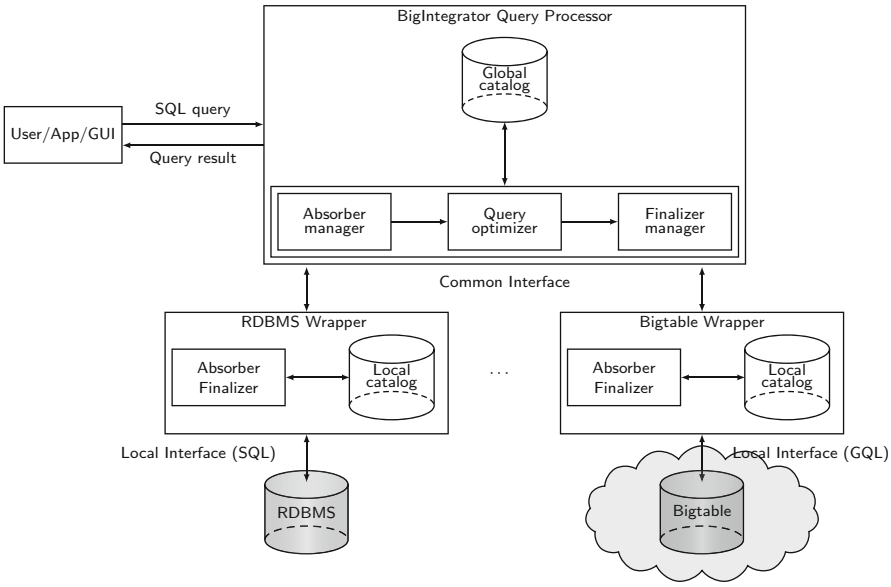


Fig. 11.9 BigIntegrator architecture

of the data source. For instance, "Employees_A" represents an Employees table at source A, i.e., a local view of Employees. The source tables are referenced as tables in the SQL queries.

Figure 11.9 illustrates the architecture of BigIntegrator with two data sources, one relational database and one Bigtable data store. Each wrapper has an importer module and absorber and finalizer plugins. The importer creates the source tables and stores them in the local catalog. The absorber extracts a subquery, called access filter, from a user query that selects data from a particular source table, based on the capabilities of the source. It translates each access filter (produced by the absorber) into an operator called interface function, specific for each kind of source. The interface function is used to send a query to the data source (i.e., a GQL or SQL query).

Query processing is performed in three steps, using an absorber manager, a query optimizer, and a finalizer manager. The absorber manager takes the (parsed) user query and, for each source table referenced in the query, calls the corresponding absorber of its wrapper. In order to replace the source table with an access filter, the absorber collects from the query the source tables and the possible other predicates, based on the capabilities of the data source. The query optimizer reorders the access filters and other predicates to produce an algebra expression that contains calls to both access filters and other relational operators. It also performs traditional transformations such as select push down and bind join. The finalizer manager takes the algebra expression and, for each access filter operator in the algebra expression,

calls the corresponding finalizer of its wrapper. The finalizer transforms the access filters into interface function calls.

Finally, query execution is performed by the query processor that interprets the algebra expression, by calling the interface functions to access the different data sources and executing the subsequent relational operations, using in-memory techniques.

11.7.1.2 Forward

Forward supports SQL++, an SQL-like language designed to unify the data model and query language capabilities of NoSQL and relational databases. SQL++ has a powerful, semistructured data model that extends both the JSON and relational data models. Forward also provides a rich web development framework, which exploits its JSON compatibility to integrate visualization components (e.g., Google Maps).

The design of SQL++ is based on the observation that the concepts are similar across both data models, e.g., a JSON array is similar to an SQL table with order, and an SQL tuple to a JSON object literal. Thus, an SQL++ collection is an array or a bag, which may contain duplicate elements. An array is ordered (similar to a JSON array) and each element is accessible by its ordinal position while a bag is unordered (similar to an SQL table). Furthermore, SQL++ extends the relational model with arbitrary composition of complex values and element heterogeneity. As in nested data models, a complex value can be either a tuple or collection. Nested collections can be accessed by nesting `SELECT` expressions in the SQL `FROM` clause or composed using the `GROUP BY` operator. They can also be unnested using the `FLATTEN` operator. And unlike an SQL table that requires all tuples to have the same attributes, an SQL++ collection may also contain heterogeneous elements comprising a mix of tuples, scalars, and nested collections.

Forward uses the Global-As-View (GAV) approach (see Sect. 7.1.1), where each data source (SQL or NoSQL) appears to the user as an SQL++ virtual view, defined over SQL++ collections. Thus, the user can issue SQL++ queries involving multiple virtual views. The Forward architecture is that of Fig. 11.8, with a query processor and one wrapper per data source. The query processor performs SQL++ query decomposition, by exploiting the underlying data store capabilities as much as possible. However, given an SQL++ query that is not directly supported by the underlying data source, Forward will decompose it into one or more native queries that are supported and combine the native query results in order to compensate for the semantics or capabilities gap between SQL++ and the underlying data source. Cost-based optimization of SQL++ queries is possible, by reusing techniques from multidatabase systems when dealing with flat collections. However, it would be much harder considering the nesting and element heterogeneity capabilities of SQL++.

11.7.1.3 QoX

QoX is a special kind of loosely coupled polystore, where queries are analytical data-driven workflows (or data flows) that integrate data from relational databases, and various execution engines such as MapReduce or ETL tools. A typical data flow may combine unstructured data (e.g., tweets) with structured data and use both generic data flow operations like filtering, join, aggregation, and user-defined functions like sentiment analysis and product identification. A novel approach to ETL design incorporates a suite of quality metrics, termed QoX, at all stages of the design process. The QoX Optimizer deals with the QoX performance metrics, with the objective of optimizing the execution of dataflows that integrate both the back-end ETL integration pipeline and the front-end query operations into a single analytics pipeline.

The QoX Optimizer uses xLM, a proprietary XML-based language to represent data flows, typically created with some ETL tool. xLM allows capturing the flow structure, with nodes showing operations and data stores and edges interconnecting these nodes, and important operation properties such as operation type, schema, statistics, and parameters. Using appropriate wrappers to translate xLM to a tool-specific XML format and vice versa, the QoX Optimizer may connect to external ETL engines and import or export dataflows to and from these engines.

Given a data flow for multiple data stores and execution engines, the QoX Optimizer evaluates alternative execution plans, estimates their costs, and generates a physical plan (executable code). The search space of equivalent execution plans is defined by data flow transformations that model data shipping (moving the data to where the operation will be executed), function shipping (moving the operation to where the data is), and operation decomposition (into smaller operations). The cost of each operation is estimated based on statistics (e.g., cardinalities, selectivities). Finally, the QoX Optimizer produces SQL code for relational database engines, Pig and Hive code for MapReduce engines, and creates Unix shell scripts as the necessary glue code for orchestrating different subflows running on different engines. This approach could be extended to access NoSQL engines as well, provided the availability of SQL-like interfaces and wrappers.

11.7.2 Tightly Coupled Polystores

Tightly coupled polystores aim at efficient querying of structured and unstructured data for (big) data analytics. They may also have a specific objective, such as self-tuning or integration of HDFS and relational DBMS data. However, they all trade autonomy for performance, typically in a shared-nothing cluster, so that data stores can only be accessed through the polystore.

Like loosely coupled systems, they provide a single language for querying of structured and unstructured data. However, the query processor directly uses the data store local interfaces (see Fig. 11.10), or in the case of HDFS, can interface

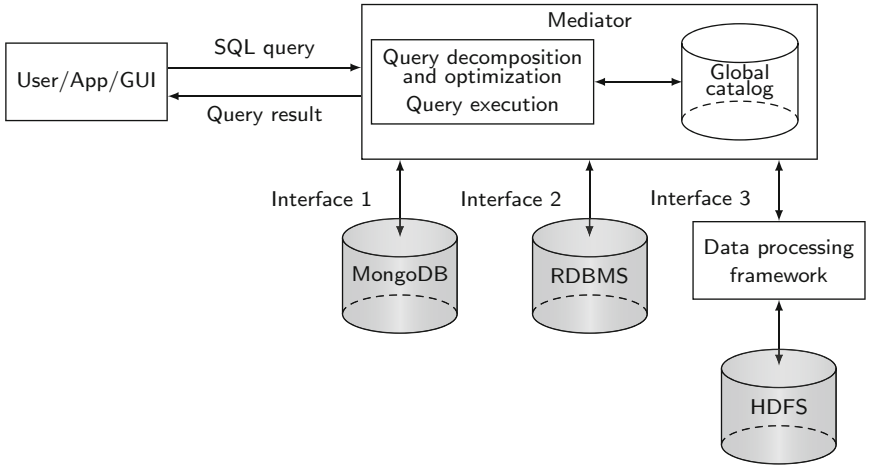


Fig. 11.10 Tightly coupled polystore architecture

a data processing framework such as MapReduce or Spark. Thus, during query execution, the query processor directly accesses the data stores. This allows efficient data movement across data stores. However, the number of data stores that can be interfaced is typically very limited.

In the rest of this section, we describe three representative tightly coupled polystores: Polybase, HadoopDB, and Estocada. Three other interesting systems are Redshift Spectrum, Odyssey, and JEN. Amazon Redshift Spectrum is a feature of Amazon's Redshift data warehouse product in the cloud platform Amazon Web Services (AWS). This feature enables running SQL queries against big unstructured data residing in Amazon Simple Storage Service (S3). Odyssey is a polystore that can work with different analytic engines, such as parallel OLAP system or Hadoop. It enables storing and querying data within HDFS and relational DBMS, using opportunistic materialized views, based on MISO, a method for tuning the physical design of a polystore (Hive/HDFS and relational DBMS), i.e., deciding in which data store the data should reside, in order to improve the performance of big data query processing. The intermediate results of query execution are treated as opportunistic materialized views, which can then be placed in the underlying stores to optimize the evaluation of subsequent queries. JEN is a component on top of HDFS to provide tight-coupling with a parallel relational DBMS. It allows joining data from two data stores, HDFS and relational DBMS, with parallel join algorithms, in particular, an efficient zigzag join algorithm, and techniques to minimize data movement. As the data size grows, executing the join on the HDFS side appears to be more efficient.

11.7.2.1 Polybase

Polybase is a feature of Microsoft SQL Server Parallel Data Warehouse (PDW), which allows users to query unstructured (HDFS) data stored in a Hadoop cluster using SQL and integrate them with relational data in PDW. The HDFS data can be referenced in Polybase as external tables, which make the correspondence with the HDFS file on the Hadoop cluster, and thus be manipulated together with PDW native tables using SQL queries. Polybase leverages the capabilities of PDW, a shared-nothing parallel DBMS. Using the PDW query optimizer, SQL operators on HDFS data are translated into MapReduce jobs to be executed directly on the Hadoop cluster. Furthermore, the HDFS data can be imported/exported to/from PDW in parallel, using the same PDW service that allows shuffling PDW data among compute nodes.

The architecture of Polybase, which is integrated within PDW, is shown in Fig. 11.11. Polybase takes advantage of PDW's Data Movement Service (DMS), which is responsible for shuffling intermediate data across PDW nodes, e.g., to repartition tuples, so that any matching tuples of an equijoin be collocated at the same computing node that performs the join. DMS is extended with an HDFS Bridge component, which is responsible for all communications with HDFS. The

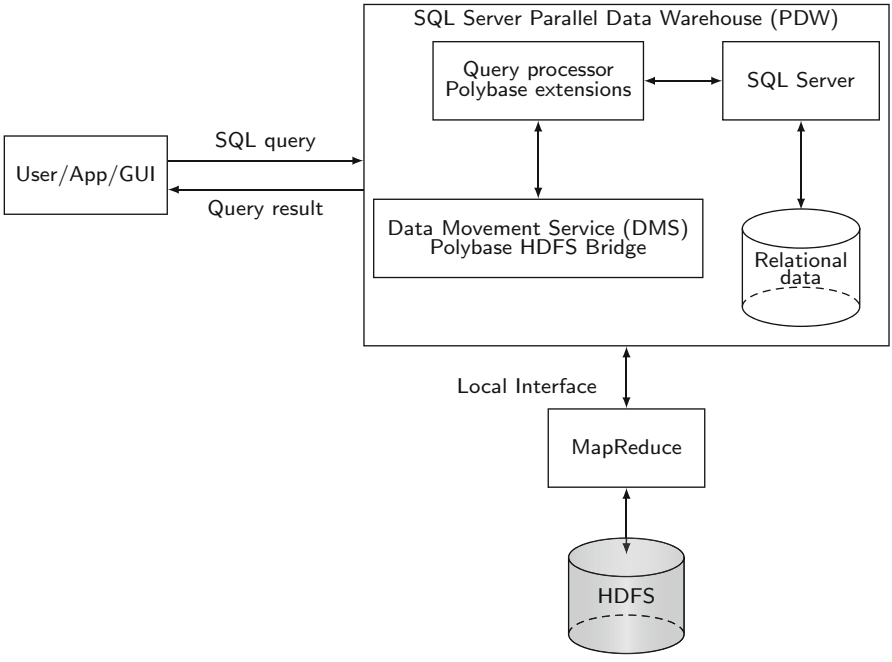


Fig. 11.11 Polybase architecture

HDFS Bridge enables DMS instances to also exchange data with HDFS in parallel (by directly accessing HDFS splits).

Polybase relies on the PDW cost-based query optimizer to determine when it is advantageous to push SQL operations on HDFS data to the Hadoop cluster for execution. Thus, it requires detailed statistics on external tables, which are obtained by exploring statistically significant samples of HDFS tables. The query optimizer enumerates the equivalent QEPs and selects the one with the least cost. The search space is obtained by considering the different decompositions of the query into two parts: one to be executed as MapReduce jobs at the Hadoop cluster and the other as regular relational operators at the PDW side. MapReduce jobs can be used to perform select and project operations on external tables, as well as joins of two external tables. The data produced by the MapReduce jobs can then be exported to PDW to be joined with relational data, using parallel hash-based join algorithms.

One strong limitation of pushing operations on HDFS data as MapReduce jobs is that even simple lookup queries have long latencies. A solution proposed for Polybase is to exploit an index built on the external HDFS data using a B+-tree that is stored inside PDW. This method leverages the robust and efficient indexing code in PDW without forcing a dramatic increase in the space that is required to store or cache the entire (large) HDFS data inside PDW. Thus, the index can be used as a prefilter by the query optimizer to reduce the amount of work that is carried out as MapReduce jobs. To keep the index synchronized with the data that is stored in HDFS, an incremental approach is used which records that the index is out-of-date, and lazily rebuilds it. Queries posed against the index before the rebuilding process is completed can be answered using a method that carefully executes parts of the query using the index in PDW, and the remaining part of the query is executed as a MapReduce job on just the changed data in HDFS. Apache AsterixDB uses a similar approach to accessing and indexing external data that lives in HDFS and allowing users' queries to span data that AsterixDB manages as well as external data in HDFS.

11.7.2.2 HadoopDB

The objective of HadoopDB is to provide the best of both parallel DBMS (high-performance data analysis over structured data) and MapReduce-based systems (scalability, fault-tolerance, and flexibility to handle unstructured data) with an SQL-like language (HiveQL) and a relational data model. To do so, HadoopDB tightly couples the Hadoop framework, including MapReduce and HDFS, with multiple single-node relational DBMS deployed across a cluster, as in a shared-nothing parallel DBMS.

HadoopDB extends the Hadoop architecture with four components: database connector, catalog, data loader, and SQL-MapReduce-SQL (SMS) planner. The database connector provides the wrappers to the underlying relational DBMS, using JDBC drivers. The catalog maintains information about the databases as an XML file in HDFS, and is used for query processing. The data loader is responsible for

(re)partitioning (key, value) data collections using hashing on a key and loading the single-node databases with the partitions (or chunks). The SMS planner extends Hive, a Hadoop component that transforms HiveQL into MapReduce jobs that connect to tables stored as files in HDFS. This architecture yields a cost-effective parallel relational DBMS, where data is partitioned both in relational DBMS tables and in HDFS files, and the partitions can be collocated at cluster nodes for efficient parallel processing.

Query processing is simple, relying on the SMS planner for translation and optimization, and MapReduce for execution. The optimization consists in pushing as much work as possible into the single-node databases, and repartitioning data collections whenever needed. The SMS planner decomposes a HiveQL query to a QEP of relational operators. Then the operators are translated to MapReduce jobs, while the leaf nodes are again transformed into SQL to query the underlying relational DBMS instances. In MapReduce, repartitioning should take place before the reduce phase. However, if the optimizer detects that an input table is partitioned on a column used as aggregation key for Reduce, it will simplify the QEP by turning it to a single Map-only job, leaving all the aggregation to be done by the relational DBMS nodes. Similarly, repartitioning is avoided for equijoins as well, if both sides of the join are partitioned on the join key.

11.7.2.3 Estocada

Estocada is a self-tuning polystore with the goal of optimizing the performance of applications that must deal with data in multiple data models, including relational, key-value, document, and graph. To obtain the best possible performance from the available data stores, Estocada automatically distributes and partitions the data across the different data stores, which are entirely under its control and hence not autonomous. Hence, it is a tightly coupled polystore.

Data distribution is dynamic and decided based on a combination of heuristics and cost-based decisions, taking into account data access patterns as they become available. Each data collection is stored as a set of partitions, whose content may overlap, and each partition may be stored in any of the underlying data stores. Thus, it may happen that a partition is stored in a data store that has a different data model than its native one. To make Estocada applications independent of the data stores, each data partition is internally described as a materialized view over one or several data collections. Thus, query processing involves view-based query rewriting.

Estocada supports two kinds of requests, for storing data and querying, with four main modules: storage advisor, catalog, query processor and execution engine. These components can directly access the data stores through their local interface. The query processor deals with single model queries only, each expressed in the query language of the corresponding data source. However, to integrate various data sources, one would need a common data model and language on top of Estocada. The storage advisor is responsible for partitioning data collections and delegating the storage of partitions to the data stores. For self-tuning the applications, it may

also recommend repartitioning or moving data from one data store to the other, based on access patterns. Each partition is defined as a materialized view expressed as a query over the collection in its native language. The catalog keeps track of information about partitions, including some cost information about data access operations by means of binding patterns which are specific to the data stores.

Using the catalog, the query processor transforms a query on a data collection into a logical QEP on possibly multiple data stores (if there are partitions of the collection in different stores). This is done by rewriting the initial query using the materialized views associated with the data collection, and selecting the best rewriting, based on the estimated execution costs. The execution engine translates the logical QEP into a physical QEP which can be directly executed by dividing the work between the data stores and Estocada's runtime engine, which provides its own operators (select, join, aggregate, etc.).

11.7.3 Hybrid Systems

Hybrid systems try to combine the advantages of loosely coupled systems (e.g., accessing many different data stores) and tightly coupled systems, e.g., accessing efficiently some data stores directly through their local interfaces. Therefore, the architecture (see Fig. 11.12) follows the mediator-wrapper architecture, while the query processor can also directly access some data stores, e.g., HDFS through MapReduce or Spark.

We describe below three hybrid polystores: Spark SQL, CloudMdsQL, and BigDAWG.

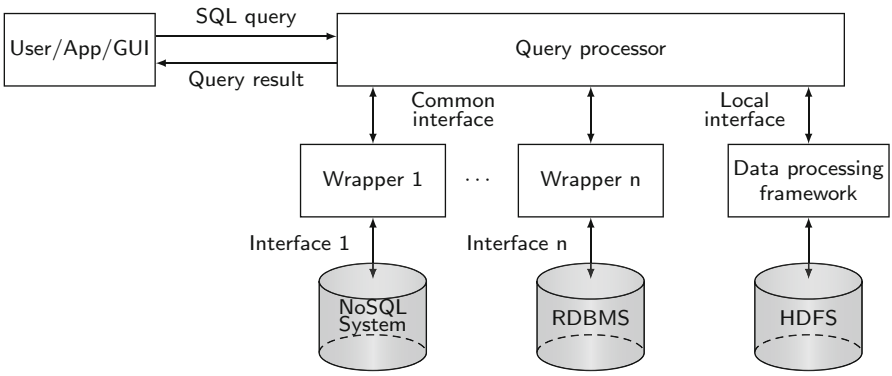


Fig. 11.12 Hybrid polystore architecture

11.7.3.1 Spark SQL

Spark SQL is a module in Apache Spark that integrates relational data processing with Spark's functional programming API. It supports SQL-like queries that can integrate HDFS data accessed through Spark and external data sources (e.g., relational databases) accessed through a wrapper. Thus, it is a hybrid polystore with tight-coupling of Spark/HDFS and loose-coupling of external data sources.

Spark SQL has a nested relational data model. It supports all major SQL data types, as well as user-defined types and complex data types (structs, arrays, maps, and unions), which can be nested together. It also supports DataFrames, which are distributed collections of rows with the same schema, like a relational table. A DataFrame can be constructed from a table in an external data source or from an existing Spark Resilient Distributed Dataset (RDD) of native Java or Python objects. Once constructed, DataFrames can be manipulated with various relational operators, such as WHERE and GROUPBY, which take expressions in procedural Spark code.

Figure 11.13 shows the architecture of Spark SQL, which runs as a library on top of Spark. The query processor directly accesses the Spark engine through the Spark Java interface, while it accesses external data sources (e.g., a relational DBMS or a key-value store) through the Spark SQL common interface supported by wrappers (JDBC drivers). The query processor includes two main components: the DataFrame API and the Catalyst query optimizer. The DataFrame API offers tight integration between relational and procedural processing, allowing relational operations to be performed on both external data sources and RDDs. It is integrated into Spark's supported programming languages (Java, Scala, Python) and supports easy inline

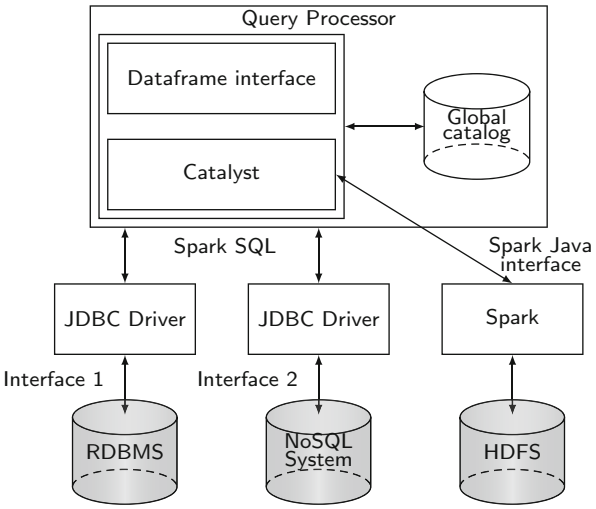


Fig. 11.13 Spark SQL architecture

definition of user-defined functions, without the complicated registration process typically found in other database systems. Thus, the DataFrame API lets developers seamlessly mix relational and procedural programming, e.g., to perform advanced analytics (which is cumbersome to express in SQL) on large data collections (accessed through relational operations).

Catalyst is an extensible query optimizer that supports both rule-based and cost-based optimization. The motivation for an extensible design is to make it easy to add new optimization techniques, e.g., to support new features of Spark SQL, as well as to enable developers to extend the optimizer to deal with external data sources, e.g., by adding data source specific rules to push down select predicates. Although extensible query optimizers have been proposed in the past, they have typically required a complex language to specify rules and a specific compiler to translate the rules into executable code. In contrast, Catalyst uses standard features of the Scala functional programming language, such as pattern matching, to make it easy for developers to specify rules, which can be compiled to Java code.

Catalyst provides a general transformation framework for representing query trees and applying rules to manipulate them. This framework is used in four phases: (1) query analysis, (2) logical optimization, (3) physical optimization, and (4) code generation. Query analysis resolves name references using a catalog (with schema information) and produces a logical plan. Logical optimization applies standard rule-based optimizations to the logical plan, such as predicate pushdown, null propagation, and Boolean expression simplification. Physical optimization takes a logical plan and enumerates a search space of equivalent physical plans, using physical operators implemented in the Spark execution engine or in the external data sources. It then selects a plan using a simple cost model, in particular, to select the join algorithms. Code generation relies on the Scala language, in particular, to ease the construction of abstract syntax trees (ASTs) in the Scala language. ASTs can then be fed to the Scala compiler at runtime to generate Java bytecode to be directly executed by compute nodes.

To speed up query execution, Spark SQL exploits in-memory caching of hot data using a column-based storage (i.e., storing data collections as sections of columns of data rather than as rows of data). Compared with Spark's native cache, which simply stores data as Java native objects, this column-based cache can reduce memory footprint by an order of magnitude by applying column compression schemes (e.g., dictionary encoding and run-length encoding). Caching is particularly useful for interactive queries and for the iterative algorithms common in machine learning.

11.7.3.2 CloudMdsQL

CloudMdsQL supports a powerful functional SQL-like language, designed for querying multiple heterogeneous data sources (e.g., relational and NoSQL). A CloudMdsQL query may contain nested subqueries, and each subquery addresses directly a particular data store and may contain embedded invocations to the data store native query interface. Thus, the major innovation is that a CloudMdsQL query

can exploit the full power of local data stores, by simply allowing some local data store native queries (e.g., a breadth-first search query against a graph database) to be called as functions. CloudMdsQL has been extended to address distributed processing frameworks such as Apache Spark by enabling the ad hoc usage of user-defined map/filter/reduce operators as subqueries.

The CloudMdsQL language is SQL-based with extended capabilities for embedding subqueries expressed in terms of each data store's native query interface. The common data model is table-based, with support of rich data types that can capture a wide range of the underlying data store data types, such as arrays and JSON objects, in order to handle nonflat and nested data, with basic operators over such composite data types. CloudMdsQL allows named table expressions to be defined as Python functions, which is useful for querying data stores that have only API-based query interface. A CloudMdsQL query is executed in the context of an ad hoc schema, formed by all named table expressions within the query. This approach fills the gap produced by the lack of a global schema and allows the query compiler to perform semantic analysis of the query.

The design of the CloudMdsQL query engine takes advantage of the fact that it operates in a cloud platform, with full control over where the system components can be installed. The architecture of the query engine is fully distributed, so that query engine nodes can directly communicate with each other, by exchanging code (query plans) and data. This distributed architecture yields important optimization opportunities, e.g., minimizing data transfers by moving the smallest intermediate data for subsequent processing by one particular node. Each query engine node consists of two parts (master and worker) and is collocated at each data store node in a computer cluster. Each master or worker has a communication processor that supports send and receive operators to exchange data and commands between nodes. A master takes as input a query and produces, using a query planner and catalog (with metadata and cost information on data sources) a query plan, which it sends to one chosen query engine node for execution. Each worker acts as a lightweight runtime database processor atop a data store and is composed of three generic modules (i.e., same code library)—query execution controller, operator engine, and table storage—and one wrapper module that is specific to a data store.

The query planner performs cost-based optimization. To compare alternative rewritings of a query, the optimizer uses a simple catalog, which provides basic information about data store collections such as cardinalities, attribute selectivities and indexes, and a simple cost model. Such information can be exposed by the wrappers in the form of cost functions or database statistics. The query language also provides a possibility for the user to define cost and selectivity functions whenever they cannot be derived from the catalog, mostly in the case of using native subqueries. The search space of alternative plans is obtained using traditional transformations, e.g., by pushing down select predicates, using bind join, performing join ordering, or planning intermediate data shipping.

11.7.3.3 BigDAWG

Like multidatabase systems, all the polystores we have seen so far provide transparent access across multiple data stores with the same data model and language. BigDAWG (Big Data Analytics Working Group) takes a different path, with the goal of unifying querying over a variety of data models and languages, Thus, there is no common data model and language. A key user abstraction in BigDAWG is an island of information, which is a collection of data stores accessed with a single query language. And there can be a variety of islands, including relational DBMSs, array DBMS, NoSQL, and data stream systems (DSSs). Within an island, there is loose-coupling of the data stores, which need to provide a wrapper (called a shim) to map the island language to their native one. When a query accesses more than one data store, objects may have to be copied between local databases, using a **`CAST`** operation, which provides a form of tight-coupling. This is why BigDAWG can be viewed as a hybrid polystore.

The architecture of BigDAWG is highly distributed, with a thin layer that interfaces the tools (e.g., visualization) and applications, with the islands of information. Since there is no common data model and language, there is no common query processor either. Instead, each island has its specific query processor. Query processing within an island is similar to that in multidatabase systems: most of the processing is pushed to the data stores and the query processor only integrates the results. The query optimizer does not use a cost model, but heuristics and some knowledge of the high performance of some data stores. For simple queries, e.g., select-project-join, the optimizer will use function shipping, in order to minimize data movement and network traffic among data stores. For complex queries, e.g., analytics, the optimizer may consider data shipping, to move the data to a data store that provides a high-performance implementation.

A query submitted to an island may involve multiple islands. In this case, the query must be expressed as multiple subqueries, each in a specific island language. To specify the island for which a subquery is intended, the user encloses the subquery in a `SCOPE` specification. Thus, a multiisland query will have multiple scopes to indicate the expected behavior of its subqueries. Furthermore, the user may insert **`CAST`** operations to move intermediate datasets between islands in an efficient way. Thus, the multiisland query processing is dictated by the way the subqueries, `SCOPE`, and **`CAST`** operations are specified by the user.

11.7.4 Concluding Remarks

Although all polystores share the same overall goal of querying multiple data stores, there are many different paths towards this goal, depending on the functional objective to be achieved. And this objective has important impact on the design choices. The major trend that dominates is the ability to integrate relational data (stored in relational DBMS) with other kinds of data in different data stores, such

as HDFS (Polybase, HadoopDB, Spark SQL, JEN) or NoSQL (Bigtable only for BigIntegrator, document stores for Forward). Thus, an important difference lies in the kind of data stores that are supported. For instance, Estocada, BigDAWG, and CloudMdsQL can support a wide variety of data stores, while Polybase and JEN target the integration of relational DBMS with HDFS only. We can also note the growing importance of accessing HDFS within Hadoop, in particular, with MapReduce or Spark, which corresponds to major use cases in structured/unstructured data integration.

Another trend is the emergence of self-tuning polystores, such as Estocada and Odyssey, with the objective of leveraging the available data stores for performance. In terms of data model and query language, most systems provide a relational SQL-like abstraction. However, QoX has a more general graph abstraction to capture analytic data flows. And both Estocada and BigDAWG allow the data stores to be directly accessed with their native (or island) languages. CloudMdsQL also allows native queries, but as subqueries within an SQL-like language.

Most polystores provide some support for managing a global schema, using either the GAV or LAV approaches, with some variations, e.g., BigDAWG uses GAV within (single model) islands of information. However, QoX, Estocada, Spark SQL, and CloudMdsQL do not support global schemas, although they provide some way to deal with the data stores' local schemas.

The query processing techniques are extensions of known techniques from distributed database systems, e.g., data/function shipping, query decomposition (based on the data stores' capabilities, bind join, select pushdown). Query optimization is also generally supported, with either a (simple) cost model or heuristics.

11.8 Conclusion

Compared with traditional relational DBMSs, these new technologies promise better scalability, performance, and ease of use. They are also complementary to the new data management technologies for big data (see Chap. 10).

The main motivation for NoSQL is to address three major limitations of relational DBMSs: "one size fits all" approach for all kinds of data and applications; limited scalability and availability of the database architecture in the cloud; and, as shown by the CAP theorem, the trade-off between strong database consistency and service availability. The four main categories of NoSQL systems are based on their underlying data model, i.e., key-value, wide column, document, and graph. For each category, we illustrated with a representative system: DynamoDB (key-value), Bigtable (wide column), MongoDB (document), and Neo4j (graph). We also illustrated multimodel NoSQL systems with OrientDB, which combines concepts from object-oriented and NoSQL document and graph data models.

NoSQL systems provide scalability, as well as availability, flexible schemas, and practical APIs, but this is generally achieved by relaxing strong database consistency. NewSQL is a recent class of DBMS that seeks to combine the

scalability of NoSQL systems with the strong consistency and usability of relational DBMS. The goal is to address the requirements of enterprise information systems, which have been supported by traditional relational DBMS, but also need to be able to scale. We illustrated NewSQL with the Google F1 and LeanXcale DBMSs.

Building cloud data-intensive applications often requires using multiple data stores (NoSQL, HDFS, relational DBMS, NewSQL), each optimized for one kind of data and tasks. In particular, many use cases exhibit the need to combine loosely structured data (e.g., log files, tweets, web pages) which are best supported by HDFS or NoSQL with more structured data in relational DBMS. Polystores provide integrated or transparent access to a number of cloud data stores through one or more query languages. We divided polystores based on the level of coupling with the underlying data stores, i.e., loosely coupled, tightly coupled, and hybrid. Then, we presented three representative polystores for each class: BigIntegrator, Forward, and QoX (loosely coupled); Polybase, HadoopDB, and Estocada (tightly coupled); Spark SQL, CloudMdsQL, and BigDAWG (hybrid).

The major trend that dominates is the ability to integrate relational data (stored in relational DBMS) with other kinds of data in different data stores, such as HDFS or NoSQL. However, an important difference between polystores lies in the kind of data stores that are supported. We also note the growing importance of accessing HDFS within Hadoop, in particular, with big data processing frameworks like MapReduce or Spark. Another trend is the emergence of self-tuning polystores, with the objective of leveraging the available data stores for performance. In terms of data model and query language, most systems provide a relational/ SQL-like abstraction. However, QoX has a more general graph abstraction to capture analytic data flows. And both Estocada and BigDAWG allow the data stores to be directly accessed with their native languages. The query processing techniques are extensions of known techniques from distributed database systems (see Chap. 4).

11.9 Bibliographic Notes

The landscape in NoSQL, NewSQL, and polystores keeps changing and lacks standards, which makes it difficult to come up with a good, up-to-date bibliography. There are many books and research papers on the topic but they become quickly outdated. Additional, up-to-date information can be found in systems' web sites and blogs. In this chapter, we focused on the systems' principles and architectures, rather than implementation details that may change over time.

An often cited motivation for NoSQL is the CAP theorem which helps understanding the trade-off between (C) consistency, (A) availability, and (P) partition tolerance. It started as a conjecture by [Brewer 2000], and was made a theorem by [Gilbert and Lynch 2002]. Although the CAP theorem says nothing about scalability, some NoSQL have used it to justify the lack of support for ACID transactions.

There are several books that introduce the NoSQL movement, in particular [Strauch 2011, Redmond and Wilson 2012], which illustrate well the topic with several representative systems presented in this chapter. There are also good books on particular systems. The presentation of the MongoDB document store is based on the book [Plugge et al. 2010] and other information on the MongoDB web site. AsterixDB [Alsubaiee et al. 2014] and Couchbase [Borkar et al. 2016] are JSON document stores that support a dialect of SQL++, initially proposed in [Ong et al. 2014]. There is also an excellent, practical book on SQL++ [Chamberlin 2018] written by Don Chamberlin, the coinventor of the original SQL language. AsterixDB's external data access and indexing mechanism is in [Alamoudi et al. 2015]. There are also good descriptions of DynamoDB [DeCandia et al. 2007] and Bigtable [Chang et al. 2008]. For an introduction to graph databases and Neo4j, there is the excellent book from the Neo4j team [Robinson et al. 2015]. Neo4j uses the causal consistency model [Elbushra and Lindström 2015] for multimaster replication and the Raft protocol for transaction durability [Ongaro and Ousterhout 2014].

The section on NewSQL systems is based on the description of the F1 DBMS [Shute et al. 2013] and the LeanXcale HTAP DBMS [Jimenez-Peris and Patiño Martinez 2011, Kolev et al. 2018].

A good motivation for polystores, or multistore systems, can be found in [Duggan et al. 2015, Kolev et al. 2016b]. The section on polystores is based on our survey paper on query processing in multistore systems [Bondiombouy and Valduriez 2016]. This paper identifies three classes of systems (1) loosely coupled, (2) tightly coupled, and (3) hybrid and illustrates each class with three representative systems: (1) BigIntegrator [Zhu and Risch 2011], Forward [Fu et al. 2014], and QoX [Simitsis et al. 2009, 2012]; (2) Polybase [DeWitt et al. 2013, Gankidi et al. 2014], HadoopDB [Abouzeid et al. 2009], and Estocada [Bugiotti et al. 2015]; (3) Spark SQL [Armbrust et al. 2015], BigDAWG [Gadepally et al. 2016], and CloudMdsQL [Bondiombouy et al. 2016, Kolev et al. 2016b,a]. Other important polystores are Amazon Redshift Spectrum, AsterixDB, AWESOME [Dasgupta et al. 2016], Odyssey [Hacigümüs et al. 2013], and JEN [Tian et al. 2016]. Odyssey uses opportunistic materialized views, based on MISO [LeFevre et al. 2014], a method for tuning the physical design of a polystore.

Exercises

Problem 11.1 Recall and discuss the motivations for NoSQL, in particular compared with relational DBMS.

Problem 11.2 (*) Explain why the CAP theorem is important. Consider a distributed architecture with multimaster replication (see Chap. 6). Assume a network partitioning in with asynchronous replication.

(a) Which of the CAP properties are preserved?
(b) What kind of consistency is achieved?

Same questions assuming synchronous replication.

Problem 11.3 (**) In this chapter, we divided NoSQL systems into four categories, i.e., key-value, wide column, document, and graph.

(a) Discuss the main similarities and differences in terms of data model, query language and interfaces, architectures, and implementation techniques.
(b) Identify the best use cases for each category of system.

Problem 11.4 (**) Consider the following simplified order-entry database schema (in nested relational format), with primary key attributes underlined:

```
CUSTOMERS(CID, NAME, ADDRESS (STREET, CITY, STATE,
  COUNTRY), PHONES)
ORDERS(OID, CID, O-DATE, O-TOTAL)
ORDER-ITEMS(OID, LINE-ID, PID, QTY)
PRODUCTS(PID, P-NAME, PRICE)
```

(a) Give the corresponding schemas in the four kinds of NoSQL systems (key-value, wide column, document, and graph). Discuss the respective advantages and disadvantages of each design in terms of ease of use, database administration, query complexity, and update performance.
(b) Consider now that a product can be made of several products, e.g., a six pack beer. Reflect this on the database schemas, and discuss the implications using the four kinds of NoSQL systems.

Problem 11.5 (**) As discussed in Chap. 10, there is no optimal solution for graph database partitioning. Elaborate on the impact on the scalability of graph databases? Propose ways around.

Problem 11.6 (**) Compare the F1 NewSQL system with a standard parallel relational DBMS, e.g., MySQL Cluster, in terms of data model, query language and interfaces, consistency, scalability, and availability.

Problem 11.7 Polystores provide integrated access through queries to multiple data stores such as NoSQL, relational DBMS, or HDFS. Compare polystores with the data integration systems we presented in Chap. 7.

Problem 11.8 (***) Polystores typically support only read-only queries, which satisfies the requirements of analytics. However, as more and more complex cloud data-intensive are built, the need for updating data across data stores will become important. Thus, the need for distributed transactions will arise. However, the transaction models of the data stores may be very different. In particular, most NoSQL systems do not provide ACID transaction support. Discuss the issue and propose directions for solutions.

Chapter 12
Web Data Management

The World Wide Web ("WWW" or "web" for short) has become a major repository of data and documents. Although measurements differ and change, the web has grown at a phenomenal rate.[1] Besides its size, the web is very dynamic and changes rapidly. For all practical purposes, the web represents a very large, dynamic, and distributed data store and there are the obvious distributed data management issues in accessing web data.

The web, in its present form, can be viewed as two distinct yet related components. The first of these components is what is known as the *publicly indexable web* (PIW) that is composed of all static (and cross-linked) web pages that exist on web servers. These can be easily searched and indexed. The other component, which is known as the *deep web* (or the *hidden web*), is composed of a huge number of databases that encapsulate the data, hiding it from the outside world. The data in the hidden web are usually retrieved by means of search interfaces where the user enters a query that is passed to the database server, and the results are returned to the user as a dynamically generated web page. A portion of the deep web has come to be known as the "dark web," which consists of encrypted data and requires a particular browser such as Tor to access.

The difference between the PIW and the hidden web is basically in the way they are handled for searching and/or querying. Searching the PIW depends mainly on crawling its pages using the link structure between them, indexing the crawled pages, and then searching the indexed data (as we discuss at length in Sect. 12.2). This may be either through the well-known keyword search or via question answering (QA) systems (Sect. 12.4). It is not possible to apply this approach to the hidden web directly since it is not possible to crawl and index those data (the techniques for searching the hidden web are discussed in Sect. 12.5).

Research on web data management has followed different threads in two separate but overlapping communities. Most of the earlier work in the web search and

[1] See http://www.worldwidewebsize.com/.

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_12

information retrieval community focused on keyword search and search engines. Subsequent work in this community focused on QA systems. The work in the database community focused on declarative querying of web data. There is an emerging trend that combines search/browse mode of access with declarative querying, but this work has not yet reached its full potential. In the 2000s, XML emerged as an important data format for representing and integrating data on the web. Thus, XML data management was a topic of significant interest. Although XML is still important in a number of application areas, its use in web data management has waned, mostly due to its perceived complexity. More recently, RDF has emerged as a common representation for Web data representation and integration.

The result of these different threads of development is that there is little in the way of a unifying architecture or framework for discussing web data management, and the different lines of research have to be considered somewhat separately. Furthermore, the full coverage of all the web-related topics requires far deeper and far more extensive treatment than is possible within a chapter. Therefore, we focus on issues that are directly related to data management.

We start by discussing how web data can be modeled as a graph. Both the structure of this graph and its management are important. This is discussed in Sect. 12.1. Web search is discussed in Sect. 12.2 and web querying is covered in Sect. 12.3. Section 12.4 summarizes question answering systems, and searching and querying the deep/hidden web is covered in Sect. 12.5. We then discuss web data integration in Sect. 12.6, focusing both on the fundamental problems and some of the representation approaches (e.g., web tables, XML, and RDF) that can assist with the task.

12.1 Web Graph Management

The web consists of "pages" that are connected by hyperlinks, and this structure can be modeled as a directed graph that reflects the hyperlink structure. In this graph, commonly referred to as the *web graph*, static HTML web pages are the vertices and the links between the pages are represented as directed edges. The characteristics of the web graph is important for studying data management issues since the graph structure is exploited in web search, categorization and classification of web content, and other web-related tasks. In addition, RDF representation that we discuss in Sect. 12.6.2.2 formalizes the web graph using a particular notation. The important characteristics of the web graph are the following:

(a) It is quite volatile. We already discussed the speed with which the graph is growing. In addition, a significant proportion of the web pages experience frequent updates.

(b) It is sparse. A graph is considered sparse if its average degree (i.e., the average of the degrees of all of its vertices) is less than the number of vertices. This

means that the each vertex of the graph has a limited number of neighbors, even if the vertices are in general connected. The sparseness of the web graph implies an interesting graph structure that we discuss shortly.

(c) It is "self-organizing." The web contains a number of communities, each of which consists of a set of pages that focus on a particular topic. These communities get organized on their own without any "centralized control," and give rise to the particular subgraphs in the web graph.

(d) It is a "small-world graph." This property is related to sparseness—each node in the graph may not have many neighbors (i.e., its degree may be small), but many nodes are connected through intermediaries. Small-world networks were first identified in social sciences where it was noted that many people who are strangers to each other are connected by intermediaries. This holds true in web graphs as well in terms of the connectedness of the graph.

(e) It is a power law graph. The in- and out-degree distributions of the web graph follow power law distributions. This means that the probability that a vertex has in- (out-) degree i is proportional to $1/i^{\alpha}$ for some $\alpha > 1$. The value of α is about 2.1 for in-degree and about 7.2 for out-degree.

This brings us to a discussion of the structure of the web graph, which has a "bowtie" shape (Fig. 12.1). It has a strongly connected component (the knot in the middle) in which there is a path between each pair of pages. The numbers we give below are from a study in 2000; while these numbers have possibly changed, the structure depicted in the figure has persisted. Readers should treat numbers as indicative of relative size and not as absolute values. The strongly connected component (SCC) accounts for about 28% of the web pages. A further 21% of the pages constitute the "IN" component from which there are paths to pages in SCC, but to which no paths exist from pages in SCC. Symmetrically, "OUT" component has pages to which paths exist from pages in SCC but not vice versa, and these also constitute 21% of the pages. What is referred to as "tendrils" consist of pages that cannot be reached from SCC and from which SCC pages cannot be reached

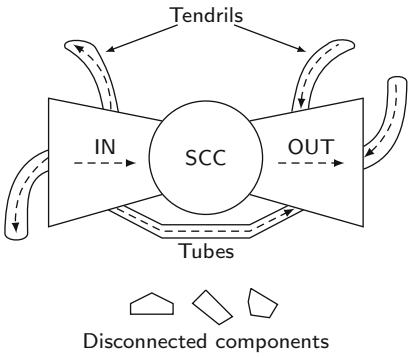


Fig. 12.1 The structure of the web as a bowtie (based on [Kumar et al. 2000])

either. These constitute about 22% of the web pages. These are pages that have not yet been "discovered" and have not yet been connected to the better known parts of the web. Finally, there are disconnected components that have no links to/from anything except their own small communities. This makes up about 8% of the web. This structure is interesting in that it determines the results that one gets from web searches and from querying the web. Furthermore, this graph structure is different than many other graphs that are normally studied, requiring special algorithms and techniques for its management.

12.2 Web Search

Web search involves finding "all" the web pages that are relevant (i.e., have content related) to keyword(s) that a user specifies. Naturally, it is not possible to find all the pages, or even to know if one has retrieved all the pages; thus the search is performed on a database of web pages that have been collected and indexed. Since there are usually multiple pages that are relevant to a query, these pages are presented to the user in ranked order of relevance as determined by the search engine.

The abstract architecture of a generic search engine is shown in Fig. 12.2. We discuss the components of this architecture in some detail.

In every search engine the *crawler* plays one of the most crucial roles. A crawler is a program used by a search engine to scan the web on its behalf and collect data about web pages. A crawler is given a starting set of pages—more accurately, it is given a set of Uniform Resource Locators (URLs) that identify these pages. The crawler retrieves and parses the page corresponding to that URL, extracts any

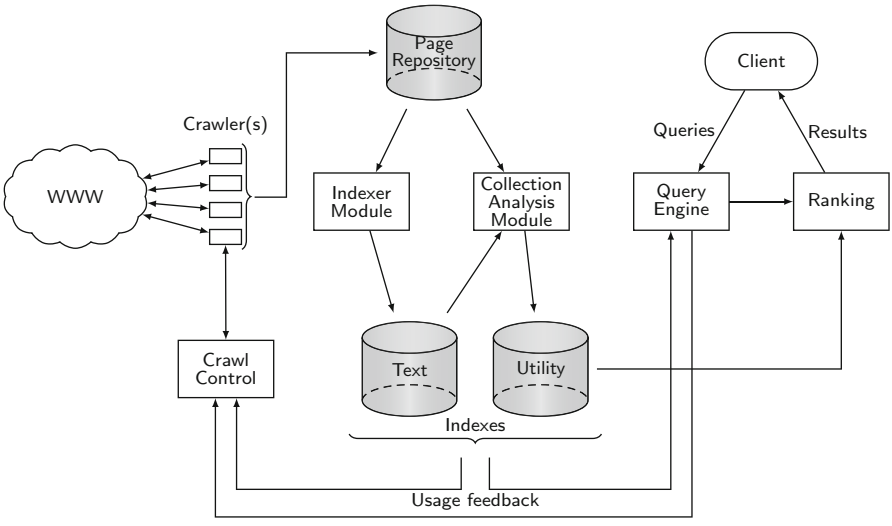


Fig. 12.2 Search engine architecture (based on [Arasu et al. 2001])

URLs in it, and adds these URLs to a queue. In the next cycle, the crawler extracts a URL from the queue (based on some order) and retrieves the corresponding page. This process is repeated until the crawler stops. A control module is responsible for deciding which URLs should be visited next. The retrieved pages are stored in a page repository. Section 12.2.1 examines crawling operations in more detail.

The *indexer module* is responsible for constructing indexes on the pages that have been downloaded by the crawler. While many different indexes can be built, the two most common ones are *text indexes* and *link indexes*. In order to construct a text index, the indexer module constructs a large "lookup table" that can provide all the URLs that point to the pages where a given word occurs. A link index describes the link structure of the web and provides information on the in-link and out-link state of pages. Section 12.2.2 explains current indexing technology and concentrates on ways indexes can be efficiently stored.

The *ranking module* is responsible for sorting a large number of results so that those that are considered to be most relevant to the user's search are presented first. The problem of ranking has drawn increased interest in order to go beyond traditional information retrieval (IR) techniques to address the special characteristics of the web—web queries are usually small and they are executed over a vast amount of data. Section 12.2.3 introduces algorithms for ranking and describes approaches that exploit the link structure of the web to obtain improved ranking results.

12.2.1 Web Crawling

As indicated above, a crawler scans the web on behalf of a search engine to extract information about the visited web pages. Given the size of the web, the changing nature of web pages, and the limited computing and storage capabilities of crawlers, it is impossible to crawl the entire web. Thus, a crawler must be designed to visit "most important" pages before others. The issue, then, is to visit the pages in some ranked order of importance.

There are a number of issues that need to be addressed in designing a crawler. Since the primary goal is to access more important pages before others, there needs to be some way of determining the importance of a page. This can be done by means of a measure that reflects the importance of a given page. These measures can be static, such that the importance of a page is determined independent of retrieval queries that will run against it, or dynamic in that they take the queries into consideration. Examples of static measures are those that determine the importance of a page P_i with respect to the number of pages that point to P_i (referred to as *backlink*), or those that additionally take into account the importance of the backlink pages as is done in the popular PageRank metric that is used by Google and others. A possible dynamic measure may be one that calculates the importance of a page P_i with respect its textual similarity to the query that is being evaluated using some of the well-known information retrieval similarity measures.

We had introduced PageRank in the Chap. 10 (Example 10.4). Recall that the PageRank of a page P_i, denoted $PR(P_i)$, is simply the normalized sum of the PageRank of all P_i's backlink pages (denoted as B_{P_i}) where the normalization for each $P_j \in B_{P_i}$ is over all of P_j's forward links F_{P_j}:

$$PR(P_i) = \sum_{P_j \in B_{P_i}} \frac{PR(P_j)}{|F_{P_j}|}$$

Recall also that this formula calculates the rank of a page based on the backlinks, but normalizes the contribution of each backlinking page P_j using the number of forward links that P_j has. The idea here is that it is more important to be pointed at by pages conservatively link to other pages than by those who link to others indiscriminately, but the "contribution" of a link from such a page needs to be normalized over all the pages that it points to.

A second issue is how the crawler chooses the next page to visit once it has crawled a particular page. As noted earlier, the crawler maintains a queue in which it stores the URLs for the pages that it discovers as it analyzes each page. Thus, the issue is one of ordering the URLs in this queue. A number of strategies are possible. One possibility is to visit the URLs in the order in which they were discovered; this is referred to as the *breadth-first approach*. Another alternative is to use random ordering whereby the crawler chooses a URL randomly from among those that are in its queue of unvisited pages. Other alternatives are to use metrics that combine ordering with importance ranking discussed above, such as backlink counts or PageRank.

Let us discuss how PageRank can be used for this purpose. A slight revision is required to the PageRank formula given above. We are now modeling a random surfer: when landed on a page P, a random surfer is likely to choose one of the URLs on this page as the next one to visit with some (equal) probability d or will jump to a random page with probability $1-d$. Then the above formula for PageRank is revised as follows:

$$PR(P_i) = (1-d) + d \sum_{P_j \in B_{P_i}} \frac{PR(P_j)}{|F_{P_j}|}$$

The ordering of the URLs according to this formula allows the importance of a page to be incorporated into the order in which the corresponding page is visited. In some formulations, the first term is normalized with respect to the total number of pages in the web.

Example 12.1 Consider the web graph in Fig. 12.3 where each web page P_i is a vertex and there is a directed edge from P_i to P_j if P_i has a link to P_j. Assuming the commonly accepted value of $d = 0.85$, the PageRank of P_2 is $PR(P_2) = 0.15 + 0.85(\frac{PR(P_1)}{2} + \frac{PR(P_3)}{3})$. This is a recursive formula that is evaluated by initially assigning to each page equal PageRank values (in this case $\frac{1}{6}$ since there are 6

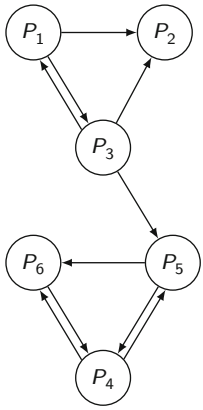


Fig. 12.3 Web graph representation for PageRank computation

pages) and iterating to compute each $PR(P_i)$ until a fixpoint is reached (i.e., the values no longer change). ♦

Since many web pages change over time, crawling is a continuous activity and pages need to be revisited. Instead of restarting from scratch each time, it is preferable to selectively revisit web pages and update the gathered information. Crawlers that follow this approach are called *incremental crawlers*. They ensure that the information in their repositories are as fresh as possible. Incremental crawlers can determine the pages that they revisit based on the change frequency of the pages or by sampling a number of pages. *Change frequency-based* approaches use an estimate of the change frequency of a page to determine how frequently it should be revisited. One might intuitively assume that pages with high change frequency should be visited more often, but this is not always true—any information extracted from a page that changes frequently is likely to become obsolete quickly, and it may be better to increase revisit interval to that page. It is also possible to develop an adaptive incremental crawler such that the crawling in one cycle is affected by the information collected in the previous cycle. *Sampling-based approaches* focus on web sites rather than individual web pages. A small number of pages from a web site are sampled to estimate how much change has happened at the site. Based on this sampling estimate, the crawler determines how frequently it should visit that site.

Some search engines specialize in searching pages belonging to a particular topic. These engines use crawlers optimized for the target topic, and are referred to as *focused crawlers*. A focused crawler ranks pages based on their relevance to the target topic, and uses them to determine which pages it should visit next. Classification techniques that are widely used in information retrieval are used in evaluating relevance; learning techniques are used to identify the topic of a given page. These techniques are beyond our scope, but a number of them have been developed for this purpose, such as naïve Bayes classifier, and its extensions, reinforcement learning, and others.

To achieve reasonable scale-up, crawling can be parallelized by running *parallel crawlers*. Any design for parallel crawlers must use schemes to minimize the overhead of parallelization. For instance, two crawlers running in parallel may download the same set of pages. Clearly, such overlap needs to be prevented through coordination of the crawlers' actions. One method of coordination uses a *central coordinator* to dynamically assign each crawler a set of pages to download. Another coordination scheme is to logically partition the web. Each crawler knows its partition, and there is no need for central coordination. This scheme is referred to as the *static assignment*.

12.2.2 Indexing

In order to efficiently search the crawled pages and the gathered information, a number of indexes are built as shown in Fig. 12.2. The two more important indexes are the *structure* (or *link*) *index* and a *text* (or *content*) *index*.

12.2.2.1 Structure Index

The structure index is based on the graph model that we discussed in Sect. 12.1, with the graph representing the structure of the crawled portion of the web. The efficient storage and retrieval of these pages is important and two techniques to address these issues were discussed in Sect. 12.1. The structure index can be used to obtain important information about the linkage of web pages such as information regarding the *neighborhood* of a page and the siblings of a page.

12.2.2.2 Text Index

The most important and mostly used index is the *text index*. Indexes to support text-based retrieval can be implemented using any of the access methods traditionally used to search over text document collections. Examples include *suffix arrays*, *inverted files* or *inverted indexes*, and *signature files*. Although a full treatment of all of these indexes is beyond our scope, we will discuss how inverted indexes are used in this context since these are the most popular text indexes.

An inverted index is a collection of inverted lists, where each list is associated with a particular word. In general, an inverted list for a given word is a list of document identifiers in which that particular word occurs. The location of the word on a particular page can also be saved as part of the inverted list. This information is usually needed in proximity queries and query result ranking. Search algorithms also often make use of additional information about the occurrence of terms in a web page. For example, terms occurring in bold face (within ⟨`strong`⟩ tags), in section headings (within ⟨`H1`⟩ or ⟨`H2`⟩ tags in HTML), or as anchor text might be weighted differently in the ranking algorithms.

In addition to the inverted list, many text indexes also keep a *lexicon*, which is a list of all terms that occur in the index. The lexicon can also contain some term-level statistics that can be used by ranking algorithms.

Constructing and maintaining an inverted index has three major difficulties:

1. In general, building an inverted index involves processing each page, reading all the words and storing the location of each word. In the end, the inverted files are written to disk. This process, while trivial for small and static collections, becomes hard to manage when dealing with a vast and nonstatic collection like the web.
2. The rapid change of the web poses a challenge for maintaining the "freshness" of the index. Although we argued in the previous section that incremental crawlers should be deployed to ensure freshness, periodic index rebuilding is still necessary because most incremental update techniques do not perform well when dealing with the large changes often observed between successive crawls.
3. Storage formats of inverted indexes must be carefully designed. There is a tradeoff between a performance gain through a compressed index that allows portions of the index to be cached in memory and the overhead of decompression at query time. Achieving the right balance becomes a major concern when dealing with web-scale collections.

Addressing these challenges and developing a highly scalable text index can be achieved by distributing the index by either building a *local inverted index* at each machine where the search engine runs or building a *global inverted index* that is then shared. We do not discuss these further, as the issues are similar to the distributed data and directory management issues we have already covered in previous chapters.

12.2.3 Ranking and Link Analysis

A typical search engine returns a large number of web pages that are expected to be relevant to a user query. However, these pages are likely to be different in terms of their quality and relevance. The user is not expected to browse through this large collection to find a high-quality page. Clearly, there is a need for algorithms to rank these pages such that higher quality web pages appear as part of the top results.

Link-based algorithms can be used to rank a collection of pages. To repeat what we discussed earlier, the intuition is that if a page P_j contains a link to page P_i, then it is likely that the authors of page P_j think that page P_i is of good quality. Thus, a page that has a large number of incoming links is likely of high quality, and hence the number of incoming links to a page can be used as a ranking criterion. This intuition is the basis of ranking algorithms, but, of course, each specific algorithm implements this intuition in a different way. We already discussed the PageRank algorithm, and it is used for ranking of results in addition to crawling. We will discuss an alternative algorithm called HITS to highlight different ways of approaching the issue.

HITS is also a link-based algorithm. It is based on identifying "authorities" and "hubs." A good authority page receives a high rank. Hubs and authorities have a mutually reinforcing relationship: a good authority is a page that is linked to by many good hubs, and a good hub is a document that links to many authorities. Thus, a page pointed to by many hubs (a good authority page) is likely to be of high quality.

Let us start with a web graph, $G = (V, E)$, where V is the set of pages and E is the set of links among them. Each page P_i in V has a pair of nonnegative weights (a_{P_i}, h_{P_i}) that represent the authoritative and hub values of P_i respectively.

The authoritative and hub values are updated as follows. If a page P_i is pointed to by many good hubs, then a_{P_i} is increased to reflect all pages P_j that link to it (the notation $P_j \rightarrow P_i$ means that page P_j has a link to page P_i):

$$a_{P_i} = \sum_{\{P_j | P_j \rightarrow P_i\}} h_{P_j}$$

$$h_{P_i} = \sum_{\{P_j | P_j \rightarrow P_i\}} a_{P_j}$$

Thus, the authoritative value (hub value) of page P_i, is the sum of the hub values (authority values) of all the backlink pages to P_i.

12.2.4 Evaluation of Keyword Search

Keyword-based search engines are the most popular tools to search information on the web. They are simple, and one can specify fuzzy queries that may not have an exact answer, but may only be answered approximately by finding facts that are "similar" to the keywords. However, there are obvious limitations as to how much one can do by simple keyword search. The obvious limitation is that keyword search is not sufficiently powerful to express complex queries. This can be (partially) addressed by employing iterative queries where previous queries by the same user can be used as the context for the subsequent queries. A second limitation is that keyword search does not offer support for a global view of information on the web the way that database querying exploits database schema information. It can, of course, be argued that a schema is meaningless for web data, but the lack of an overall view of the data is an issue nevertheless. A third problem is that it is difficult to capture user's intent by simple keyword search—errors in the choice of keywords may result in retrieving many irrelevant answers.

Category search addresses one of the problems of using keyword search, namely the lack of a global view of the web. Category search is also known as web directory, catalogs, yellow pages, and subject directories. There are a number of public web

directories available such as World Wide Web Virtual Library (http://vlib.org).[2] The web directory is a hierarchical taxonomy that classifies human knowledge. Although, the taxonomy is typically displayed as a trie, it is actually a directed acyclic graph since some categories are cross referenced.

If a category is identified as the target, then the web directory is a useful tool. However, not all web pages can be classified, so the user can use the directory for searching. Moreover, natural language processing cannot be 100% effective for categorizing web pages. We need to depend on human resource for judging the submitted pages, which may not be efficient or scalable. Finally, some pages change over time, so keeping the directory up-to-date involves significant overhead.

There have also been some attempts to involve multiple search engines in answering a query to improve recall and precision. A metasearcher is a web service that takes a given query from the user and sends it to multiple heterogeneous search engines. The metasearcher then collects the answers and returns a unified result to the user. It has the ability to sort the result by different attributes such as host, keyword, date, and popularity. Examples include Dogpile (http://www.dogpile.com/), MetaCrawler (http://www.metacrawler.com/), and lxQuick (http://www.ixquick.com/). Different metasearchers have different ways to unify results and translate the user query to the specific query languages of each search engines. The user can access a metasearcher through client software or a web page. Each search engine covers a smaller percentage of the web. The goal of a metasearcher is to cover more web pages than a single search engine by combining different search engines together.

12.3 Web Querying

Declarative querying and efficient execution of queries have been a major focus of database technology. It would be beneficial if the database techniques can be applied to the web. In this way, accessing the web can be treated, to a certain extent, similar to accessing a large database. We will discuss a number of the proposed approaches in this section.

There are difficulties in carrying over traditional database querying concepts to web data. Perhaps the most important difficulty is that database querying assumes the existence of a strict schema. As noted above, it is hard to argue that there is a schema for web data similar to databases.[3] At best, the web data are *semistructured*—data may have some structure, but this may not be as rigid, regular, or complete as that of databases, so that different instances of the data may be similar

[2] A list of these libraries is given in https://en.wikipedia.org/wiki/List_of_web_directories.

[3] We are focusing on the "open" web here; deep web data may have a schema, but it is usually not accessible to users.

but not identical (there may be missing or additional attributes or differences in structure). There are, obviously, inherent difficulties in querying schema-less data.

A second issue is that the web is more than the semistructured data (and documents). The links that exist between web data entities (e.g., pages) are important and need to be considered. Similar to search that we discussed in the previous section, links may need to be followed and exploited in executing web queries. This requires links to be treated as first-class objects.

A third major difficulty is that there is no commonly accepted language, similar to SQL, for querying web data. As we noted in the previous section, keyword search has a very simple language, but this is not sufficient for richer querying of web data. Some consensus on the basic constructs of such a language has emerged (e.g., path expressions), but there is no standard language. However, standardized languages for data models such as XML and RDF have emerged (XQuery for XML and SPARQL for RDF). We postpone discussion of these to Sect. 12.6 where we focus on web data integration

12.3.1 Semistructured Data Approach

One way to approach querying the web data is to treat it as a collection of semistructured data. Then, models and languages that have been developed for this purpose can be used to query the data. Semistructured data models and languages were not originally developed to deal with web data; rather they addressed the requirements of growing data collections that did not have as strict a schema as their relational counterparts. However, since these characteristics are also common to web data, later studies explored their applicability in this domain. We demonstrate this approach using a particular model (OEM) and a language (Lorel), but other approaches such as UnQL are similar.

OEM (Object Exchange Model) is a self-describing semistructured data model. Self-describing means that each object specifies the schema that it follows.

An OEM object is defined as a four-tuple ⟨`label, type, value, oid`⟩, where `label` is a character string describing what the object represents, `type` specifies the type of the object's value, `value` is obvious, and `oid` is the object identifier that distinguishes it from other objects. The type of an object can be `atomic`, in which case the object is called an *atomic object*, or `complex`, in which case the object is called a *complex object*. An atomic object contains a primitive value such as an integer, a real, or a string, while a complex object contains a set of other objects, which can themselves be atomic or complex. The value of a complex object is a set of oids.

Example 12.2 Let us consider a bibliographic database that consists of a number of documents. A snapshot of an OEM representation of such a database is given in Fig. 12.4. Each line shows one OEM object and the indentation is provided to simplify the display of the object structure. For example, the second line

```
<bib, complex, {&o2, &o22, &034}, &o1>
   <doc, complex, {&o3, &o6, &o7, &o20, &o22}, &o2>
      <authors, complex, {&o4, &o5}, &o3>
         <author, string, "M. Tamer Ozsu", &o4>
         <author, string, "Patrick Valduriez", &o5>
      <title, string, "Principles of Distributed ...", &o6>
      <chapters, complex, {&o8, &o11, &o14, &o9}, &o7>
         <chapter, complex, {&o9, &o10}, &o8>
            <heading, string, "...", &o9>
            <body, string, "...", &o10>
            ...
         <chapter, complex, {&o18, &o19}, &9>
            <heading, string, "...", &o18>
            <body, string, "...", &o19>
      <what, string, "Book", &o20>
      <price, float, 98.50, &o21>
   <doc, complex, {&o23, &o25, &o26, &o27, &o28}, &o22>
      <authors, complex, {&o24, &o4}, &o23>
         <author, string, "Yingying Tao", &o24>
      <title, string, "Mining data streams ...", &o25>
      <venue, string, "CIKM", &o26>
      <year, integer, 2009, &o27>
      <sections, complex, {&o29, &o30, &o31, &o32, &o33}, &28>
         <section, string, "...", &o29>
           ...
         <section, string, "...", &o33>
   <doc, complex, {&o16,&o9,&o7,&o18,&o19,&o20,&o21},&o34>
      <author, string, "Anthony Bonato", &o35>
      <title, string, "A Course on the Web Graph", &o36>
      <what, string, "Book", &o20>
      <ISBN, string, "TK5105.888.B667", &o37>
      <chapters, complex, {&o39, &o42, &o45}, &o38>
         <chapter, complex, {&o40, &o41}, &o39>
            <heading, string, "...", &o40>
            <body, string, "...", &o41>
         <chapter, complex, {&o43, &o44}, &o42>
            <heading, string, "...", &o43>
            <body, string, "...", &o44>
         <chapter, complex, {&o46, &o47}, &45>
            <heading, string, "...", &o46>
            <body, string, "...", &o47>
      <publisher, string, "AMS", &o48>
```

Fig. 12.4 An example OEM specification

`<doc, complex, &o3, &o6, &o7, &o20, &o21, &o2>` defines an object whose label is `doc`, type is `complex`, oid is `&o2`, and whose value consists of objects whose oids are `&o3, &o6, &o7, &o20`, and `&o21`.

This database contains three documents (`&o2, &o22, &o34`); the first and third are books and the second is an article. There are commonalities among the two books (and even the article), but there are differences as well. For example, `&o2` has

the price information that &o34 does not have, while &o34 has ISBN and publisher information that t&o2 does not have. . ♦

As noted earlier, OEM data are self-describing, where each object identifies itself through its type and its label. It is easy to see that the OEM data can be represented as a vertex-labeled graph where the vertices correspond to OEM objects and the edges correspond to the subobject relationship. The label of a vertex is the oid and the label of the corresponding object vertex. However, it is quite common in literature to model the data as an edge-labeled graph: if object o_j is a subobject of object o_i, then o_j's label is assigned to the edge connecting o_i to o_j, and the oids are omitted as vertex labels. In Example 12.3, we use a vertex and edge-labeled representation that shows oids as vertex labels and assigns edge labels as described above.

Example 12.3 Figure 12.5 depicts the vertex and edge-labeled graph representation of the example OEM database given in Example 12.2. Normally, each terminal vertex (i.e., no outgoing edges) also contains the value of that object. To simplify exposition of the idea, we do not show the values. ♦

The semistructured approach fits reasonably well for modeling web data since it can be represented as a graph. Furthermore, it accepts that data may have some structure, but this may not be as rigid, regular, or complete as that of traditional databases. The users do not need to be aware of the complete structure when they query the data. Therefore, expressing a query should not require full knowledge of the structure. These graph representations of data at each data source are generated by wrappers that we discussed in Sect. 7.2.

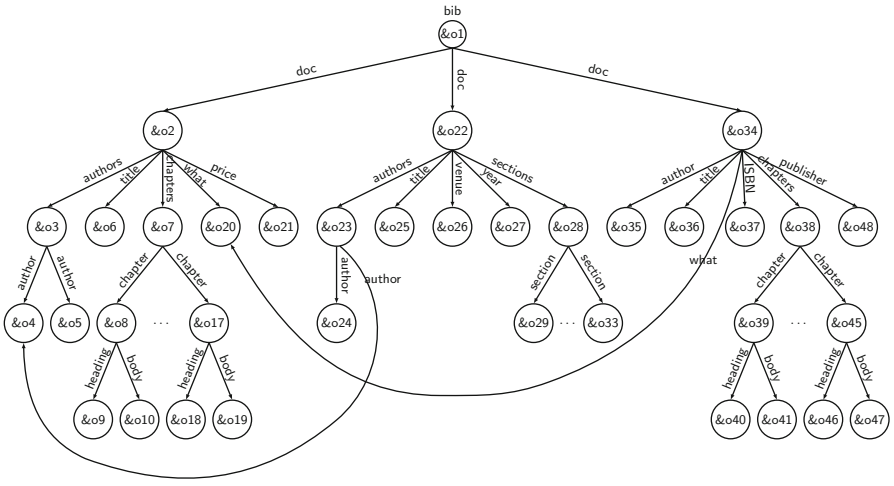


Fig. 12.5 The corresponding OEM graph for the OEM database of Example 12.2

A number of languages have been developed to query semistructured data. As noted above, we will focus our discussion by considering a particular language, Lorel, but other languages are similar in their basic approaches.

Lorel has the familiar `SELECT-FROM-WHERE` structure, but allows path expressions in the `SELECT`, `FROM` and `WHERE` clauses. The fundamental construct in forming Lorel queries is, therefore, a *path expression*. In its simplest form, a path expression in Lorel is a sequence of labels starting with an object name or a variable denoting an object. For example, `bib.doc.title` is a path expression whose interpretation is to start at bib and follow the edge-labeled doc and then follow the edge-labeled title. Note that there are three paths in Fig. 12.5 that would satisfy this expression: (i) `&o1.doc:&o2.title:&o6`, (ii) `&o1.doc:&o22.title:&o25`, and (iii) `&o1.doc:&o34.title:&o36`. Each of these is called a *data path*. In Lorel, path expressions can be more complex regular expressions such that what follows the object name or variable is not only a label, but more general expressions that can be constructed using conjunction, disjunction (|), iteration (? to mean 0 or 1 occurrences, + to mean 1 or more, and $*$ to mean 0 or more), and wildcards (#).

Example 12.4 The following are examples of acceptable path expressions in Lorel:

(a) `bib.doc(.authors)?.author` : start from `bib`, follow `doc` edge and the `author` edge with an optional `authors` edge in between.
(b) `bib.doc.#.author` : start from `bib`, follow `doc` edge, then an arbitrary number of edges with unspecified labels (using the wildcard #), and follow the `author` edge.
(c) `bib.doc.%price` : start from `bib`, follow `doc` edge, then an edge whose label has the string "price" preceded by some characters.

♦

Example 12.5 The following are example Lorel queries that use some of the path expressions given in Example 12.4:

(a) Find the titles of documents written by Patrick Valduriez.

```
SELECT D.title
FROM   bib.doc D
WHERE  bib.doc(.authors)?.author =
       "Patrick Valduriez"
```

In this query, the **`FROM`** clause restricts the scope to documents (`doc`), and the **`SELECT`** clause specifies the nodes reachable from documents by following the `title` label. We could have specified the **`WHERE`** predicate as

```
D(.authors)?.author = "Patrick Valduriez".
```

(b) Find the authors of all books whose price is under $100.

```
SELECT D(.authors)?.author
FROM   bib.doc D
WHERE  D.what = "Books" AND D.price < 100
```

♦

Semistructured data approach to modeling and querying web data is simple and flexible. It also provides a natural way to deal with containment structure of web objects, thereby supporting, to some extent, the link structure of web pages. However, there are also deficiencies of this approach. The data model is too simple—it does not include a record structure (each vertex is a simple entity) nor does it support ordering as there is no imposed ordering among the vertices of an OEM graph. Furthermore, the support for links is also relatively rudimentary, since the model or the languages do not differentiate between different types of links. The links may show either subpart relationships among objects or connections between different entities that correspond to vertices. These cannot be separately modeled, nor can they be easily queried.

Finally, the graph structure can get quite complicated, making it difficult to query. Although Lorel provides a number of features (such as wildcards) to make querying easier, the examples above indicate that a user still needs to know the general structure of the semistructured data. The OEM graphs for large databases can become quite complicated, and it is hard for users to form the path expressions. The issue, then, is how to "summarize" the graph so that there might be a reasonably small schema-like description that might aid querying. For this purpose, a construct called a DataGuide has been proposed. A DataGuide is a graph where each path in the corresponding OEM graph occurs only once. It is dynamic in that as the OEM graph changes, the corresponding DataGuide is updated. Thus, it provides concise and accurate structural summaries of semistructured databases and can be used as a lightweight schema, which is useful for browsing the database structure, formulating queries, storing statistical information, and enabling query optimization.

Example 12.6 The DataGuide corresponding to the OEM graph in Example 12.3 is given in Fig. 12.6. ♦

12.3.2 Web Query Language Approach

The approaches in this category are aimed to directly address the characteristics of web data, particularly focusing on handling *links* properly. Their starting point is to overcome the shortcomings of keyword search by providing proper abstractions for capturing the content structure of documents (as in semistructured data approaches)

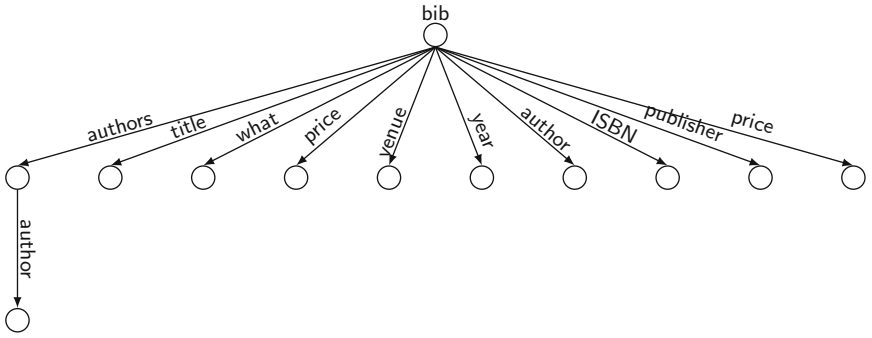


Fig. 12.6 The DataGuide corresponding to the OEM graph of Example 12.3

as well as the external links. They combine the content-based queries (e.g., keyword expressions) and structure-based queries (e.g., path expressions).

A number of languages have been proposed specifically to deal with web data, and these can be categorized as first generation and second generation. The first generation languages model the web as interconnected collection of *atomic* objects. Consequently, these languages can express queries that search the link structure among web objects and their textual content, but they cannot express queries that exploit the document structure of these web objects. The second generation languages model the web as a linked collection of *structured* objects, allowing them to express queries that exploit the document structure similar to semistructured languages. First generation approaches include WebSQL , W3QL, and WebLog, while second generation approaches include WebOQL and StruQL. We will demonstrate the general ideas by considering one first generation language (WebSQL) and one second generation language (WebOQL).

WebSQL is one of the early query languages that combines searching and browsing. It directly addresses web data as captured by web documents (usually in HTML format) that have some content and may include links to other pages or other objects (e.g., PDF files or images). It treats links as first-class objects, and identifies a number of different types of links that we will discuss shortly. As before, the structure can be represented as a graph, but WebSQL captures the information about web objects in two *virtual* relations:

```
DOCUMENT(URL, TITLE, TEXT, TYPE, LENGTH, MODIF)
LINK(BASE, HREF, LABEL)
```

`DOCUMENT` relation holds information about each web document where `URL` identifies the web object and is the primary key of the relation, `TITLE` is the title of the web page, `TEXT` is its text content of the web page, `TYPE` is the type of the web object (HTML document, image, etc.), `LENGTH` is self-explanatory, and `MODIF` is the last modification date of the object. Except `URL`, all other attributes can have null values. `LINK` relation captures the information about links where `BASE` is the URL

of the HTML document that contains the link, HREF is the URL of the document that is referenced, and LABEL is the label of the link as defined earlier.

WebSQL defines a query language that consists of SQL plus path expressions. The path expressions are more powerful than their counterparts in Lorel; in particular, they identify different types of links:

(a) *interior link* that exists within the same document (#>)
(b) *local link* that is between documents on the same server (->)
(c) *global link* that refers to a document on another server (=>)
(d) *null path* (=)

These link types form the alphabet of the path expressions. Using them, and the usual constructors of regular expressions, different paths can be specified as in Example 12.7.

Example 12.7 The following are examples of possible path expressions that can be specified in WebSQL.

(a) -> | =>: a path of length one, either local or global
(b) ->*: local path of any length
(c) =>->*: as above, but in other servers
(d) (-> |=>)*: the reachable portion of the web

♦

In addition to path expressions that can appear in queries, WebSQL allows scoping within the **FROM** clause in the following way:

```
FROM Relation SUCH THAT domain-condition
```

where domain-condition can be either a path expression, or can specify a text search using **MENTIONS**, or can specify that an attribute (in the **SELECT** clause) is equal to a web object. Of course, following each relation specification, there could be a variable ranging over the relation—this is standard SQL. The following example queries (taken from with minor modifications) demonstrate the features of WebSQL.

Example 12.8 Following are some examples of WebSQL:

(a) The first example we consider simply searches for all documents about "hypertext" and demonstrates the use of MENTIONS to scope the query.

```
SELECT D.URL, D.TITLE
FROM   DOCUMENT D
       SUCH THAT D MENTIONS "hypertext"
WHERE  D.TYPE = "text/html"
```

(b) The second example demonstrates two scoping methods as well as a search for links. The query is to find all links to applets from documents about "Java."

```
SELECT A.LABEL, A.HREF
FROM   DOCUMENT D SUCH THAT D MENTIONS "Java"
       ANCHOR A SUCH THAT BASE=X
WHERE  A.LABEL = "applet"
```

(c) The third example demonstrates the use of different link types. It searches for documents that have the string "database" in their title that are reachable from the ACM Digital Library home page through paths of length two or less containing only local links.

```
SELECT D.URL, D.TITLE
FROM   DOCUMENT D SUCH THAT
        "http://www.acm.org/dl"=|->|->-> D
WHERE  D.TITLE CONTAINS "database"
```

(d) The final example demonstrates the combination of content and structure specifications in a query. It finds all documents mentioning "Computer Science" and all documents that are linked to them through paths of length two or less containing only local links.

```
SELECT D1.URL, D1.TITLE, D2.URL, D2.TITLE
FROM   DOCUMENT D1 SUCH THAT
         D1 MENTIONS "Computer Science",
       DOCUMENT D2 SUCH THAT D1=|->|->-> D2
```

♦

WebSQL can query web data based on the links and the textual content of web documents, but it cannot query the documents based on their structure. This limitation is the consequence of its data model that treats the web as a collection of atomic objects.

The second generation languages, such as WebOQL, address this shortcoming by modeling the web as a graph of structured objects. In a way, they combine some features of semistructured data approaches with those of first generation web query models.

WebOQL's main data structure is a *hypertree*, which is an ordered edge-labeled trie with two types of edges: internal and external. An *internal edge* represents the internal structure of a web document, while an *external edge* represents a reference (i.e., hyperlink) among objects. Each edge is labeled with a record that consists of a number of attributes (fields). An external edge has to have a URL attribute in its record and cannot have descendants (i.e., they are the leaves of the hypertree).

Example 12.9 Let us revisit Example 12.2 and assume that instead of modeling the documents in a bibliography, it models the collection of documents about data management over the web. A possible (partial) hypertree for this example is given

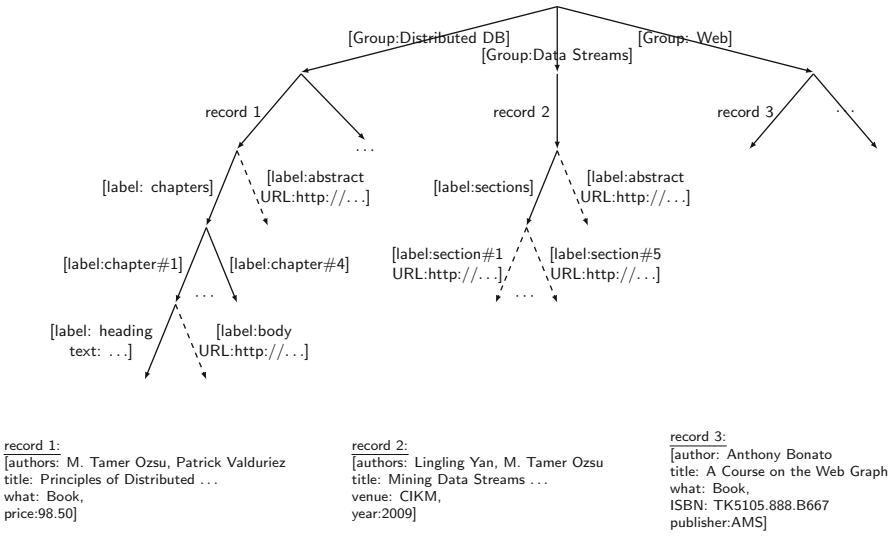


Fig. 12.7 The hypertree example

in Fig. 12.7. Note that we have made one revision to facilitate some of the queries to be discussed later: we added an abstract to each document.

In Fig. 12.7, the documents are first grouped along a number of topics as indicated in the records attached to the edges from the root. In this representation, the internal links are shown as solid edges and external links as dashed edges. Recall that in OEM (Fig. 12.5), the edges represent both attributes (e.g., author) and document structure (e.g., chapter). In the WebOQL model, the attributes are captured in the records that are associated with each edge, while the (internal) edges represent the document structure. ♦

Using this model, WebOQL defines a number of operators over trees:

Prime: returns the first subtree of its argument (denoted ′).

Peek: extracts a field from the record that labels the first outgoing edges of its document. For example, if x points to the root of the subtree reached from the "Groups = Distributed DB" edge, x.authors would retrieve "M. Tamer Ozsu, Patrick Valduriez."

Hang: builds an edge-labeled trie with a record formed with the arguments (denoted as []).

Example 12.10 Let us assume that the trie depicted in Fig. 12.8a is retrieved as a result of a query (call it Q1). Then the expression ["Label: "Papers by Ozsu" / Q1] results in the trie depicted in Fig. 12.8b. ♦

Concatenate: combines two trees (denoted +).

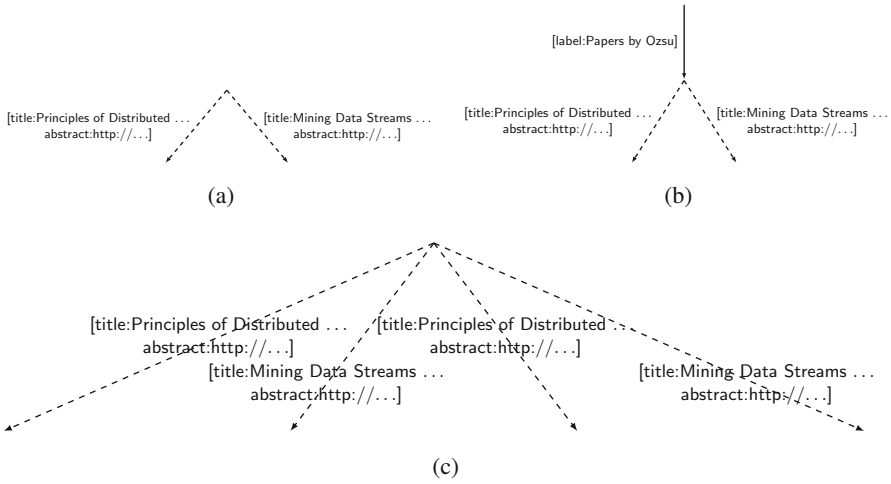


Fig. 12.8 Examples of Hang and Concatenate operators

Example 12.11 Again, assuming that the trie depicted in Fig. 12.8a is retrieved as a result of query Q1, Q1+Q2 produces trie in Fig. 12.8c. ♦

Head: returns the first simple trie of a trie (denoted &). A simple trie of a trie *t* are the trees composed of one edge followed by a (possibly null) trie that originates from *t*'s root.

Tail: returns all but the first simple trie of a trie (denoted !).

In addition to these, WebOQL introduces a string pattern matching operator (denoted ∼) whose left argument is a string and right argument is a string pattern. Since the only data type supported by the language is string, this is an important operator.

WebOQL is a functional language, so complex queries can be composed by combining these operators. In addition, it allows these operators to be embedded in the usual SQL (or OQL) style queries as demonstrated by the following example.

Example 12.12 Let `dbDocuments` denote the documents in the database shown in Fig. 12.7. Then the following query finds the titles and abstracts of all documents authored by "Ozsu" producing the result depicted in Fig. 12.8a.

```
SELECT y.title, y'.URL
FROM   x IN dbDocuments, y IN x'
WHERE  y.authors ~ "Ozsu"
```

The semantics of this query is as follows: The variable `x` ranges over the simple trees of `dbDocuments`, and, for a given `x` value, `y` iterates over the simple trees of the single subtree of `x`. It peeks into the record of the edge and if the `authors` value matches "Ozsu" (using the string matching operator ∼), then it constructs a

trie whose label is the `title` attribute of the record that `y` points to and the `URL` attribute value of the subtree. ♦

The web query languages discussed in this section adopt a more powerful data model than the semistructured approaches. The model can capture both the document structure and the connectedness of web documents. The languages can then exploit these different edge semantics. Furthermore, as we have seen from the WebOQL examples, the queries can construct new structures as a result. However, formation of these queries still requires some knowledge about the graph structure.

12.4 Question Answering Systems

In this section, we discuss an interesting and unusual (from a database perspective) approach to accessing web data: question answering (QA) systems. These systems accept natural language questions that are then analyzed to determine the specific query that is being posed. They then conduct a search to find the appropriate answer.

Question answering systems have grown within the context of IR systems where the objective is to determine the answer to posed queries within a well-defined corpus of documents. These are usually referred to as *closed domain* systems. They extend the capabilities of keyword search queries in two fundamental ways. First, they allow users to specify complex queries in natural language that may be difficult to specify as simple keyword search requests. In the context of web querying, they also enable asking questions without a full knowledge of the data organization. Sophisticated natural language processing (NLP) techniques are then applied to these queries to understand the specific query. Second, they search the corpus of documents and return explicit answers rather than links to documents that may be relevant to the query. This does not mean that they return exact answers as traditional DBMSs do, but they may return a (ranked) list of explicit responses to the query, rather than a set of web pages. For example, a keyword search for "President of USA" using a search engine would return the (partial) result in Fig. 12.9. The user is expected to find the answer within the pages whose URLs and short descriptions (called snippets) are included on this page (and several more). On the other hand, a similar search using a natural language question "Who is the president of USA?" might return a ranked list of presidents' names (the exact type of answer differs among different systems).

Question answering systems have been extended to operate on the web. In these systems, the web is used as the corpus (hence they are called *open domain* systems). The web data sources are accessed using wrappers that are developed for them to obtain answers to questions. A number of question answering systems have been developed with different objectives and functionalities, such as Mulder, WebQA, Start, and Tritus. There are also commercial systems with varying capabilities (e.g., Wolfram Alpha http://www.wolframalpha.com/).

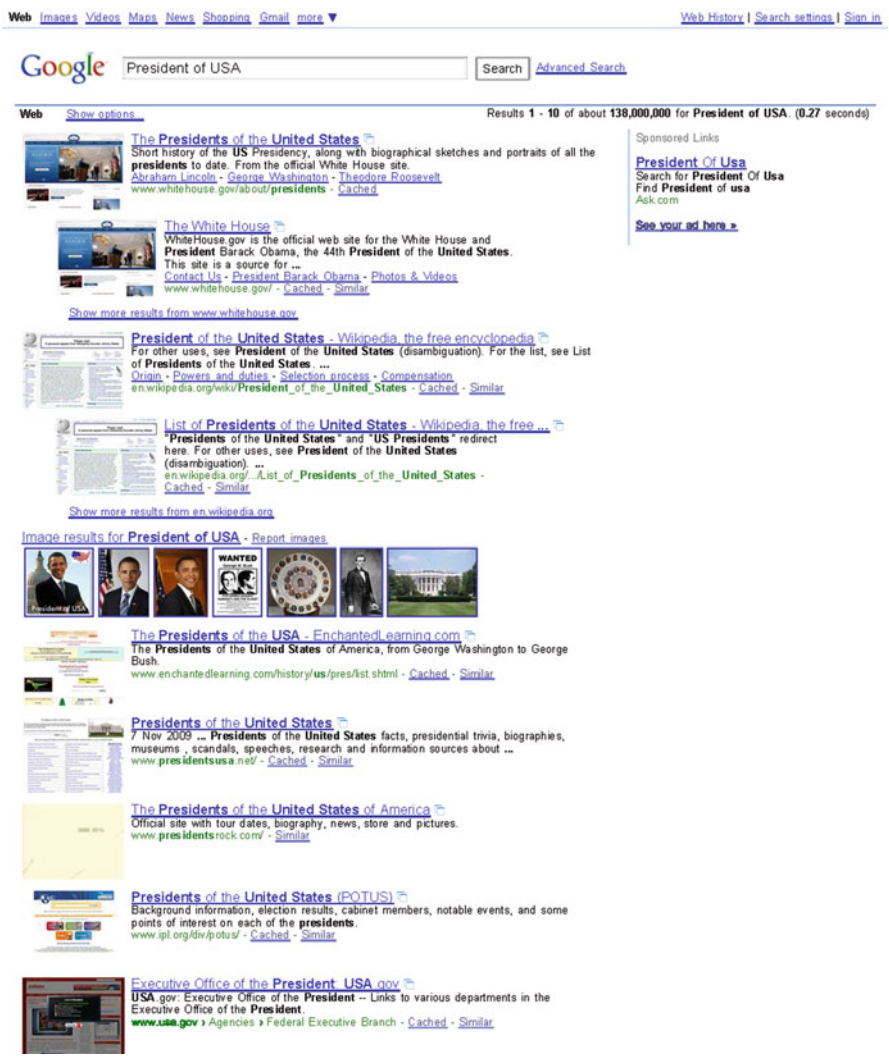


Fig. 12.9 Keyword search example

We describe the general functionality of these systems using the reference architecture given in Fig. 12.10. Preprocessing, which is not employed in all systems, is an offline process to extract and enhance the rules that are used by the systems. In many cases, these are analyses of documents extracted from the web or returned as answers to previously asked questions in order to determine the most effective query structures into which a user question can be transformed. These transformation rules are stored in order to use them at runtime while answering the user questions. For example, Tritus employs a learning-based approach that uses

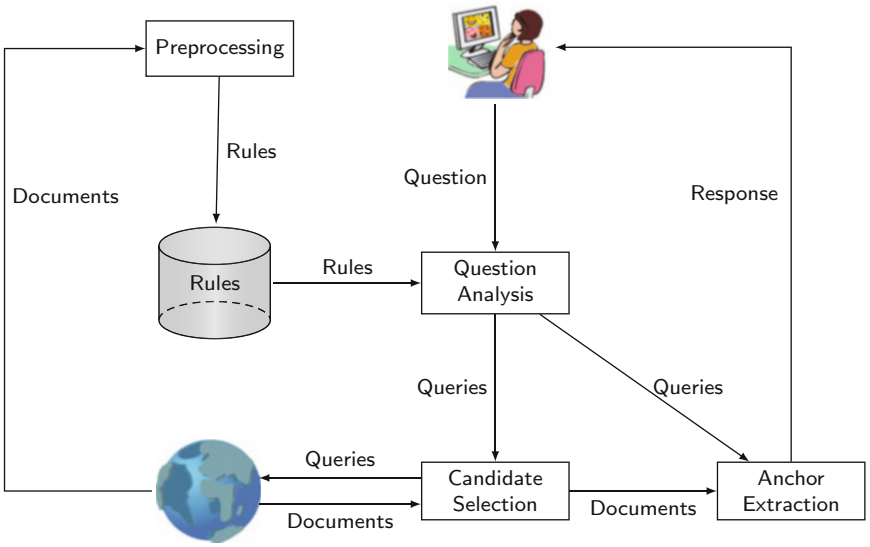


Fig. 12.10 General architecture of QA systems

a collection of frequently asked questions and their correct answers as a training dataset. In a three-stage process, it attempts to guess the structure of the answer by analyzing the question and searching for the answer in the collection. In the first stage, the question is analyzed to extract the *question phrase* (e.g., in the question "What is a hard disk?," "What is a" is a question phrase). This is used to classify the question. In the second phase, it analyzes the question-answer pairs in the training data and generates *candidate transforms* for each question phrase (e.g., for the question phrase "What is a," it generates "refers to," "stands for," etc.). In the third stage, each candidate transform is applied to the questions in the training dataset, and the resulting transformed queries are sent to different search engines. The similarities of the returned answers with the actual answers in the training data are calculated, and, based on these, a ranking is done for candidate transforms. The ranked transformation rules are stored for later use during runtime execution of questions.

The natural language question that is posed by a user first goes through the question analysis process. The objective is to understand the question issued by the user. Most of the systems try to guess the type of the answer in order to categorize the question, which is used in translating the question into queries and also in answer extraction. If preprocessing has been done, the transformation rules that have been generated are used to assist the process. Although the general goals are the same, the approaches used by different systems vary considerably depending on the sophistication of the NLP techniques employed by the systems (this phase is usually all about NLP). For example, question analysis in Mulder incorporates three phases: question parsing, question classification, and query generation. Query

parsing generates a parse trie that is used in query generation and in answer extraction. Question classification, as its name implies, categorizes the question in one of a number of classes: e.g., *nominal* is for nouns, *numerical* is for numbers, and *temporal* is for dates. This type of categorization is done in most of the QA systems because it eases the answer extraction. Finally, query generation phase uses the previously generated parse trie to construct one or more queries that can be executed to obtain the answers to the question. Mulder uses four different methods in this phase.

- Verb conversion: Auxiliary and main verb is replaced by the conjugated verb (e.g., "When did Nixon visit China?" is converted to "Nixon visited China").
- Query expansion: Adjective in the question phrase is replaced by its attribute noun (e.g., "How tall is Mt. Everest?" is converted to "The height of Everest is").
- Noun phrase formation: Some noun phrases are quoted in order to give them together to the search engine in the next stage.
- Transformation: Structure of the question is transformed into the structure of the expected answer type ("Who was the first American in space?" is converted to "The first American in space was").

Mulder is an example of a system that uses a sophisticated NLP approach to question analysis. At the other end of the spectrum is WebQA, which follows a lightweight approach in question parsing.

Once the question is analyzed and one or more queries are generated, the next step is to generate candidate answers. The queries that were generated at question analysis stage are used at this step to perform keyword search for relevant documents. Many of the systems simply use the general-purpose search engines in this step, while others also consider additional data sources that are available on the web. For example, CIA's World Factbook (https://www.cia.gov/library/publications/the-world-factbook/) is a very popular source for reliable factual data about countries. Similarly, weather information may be obtained very reliably from a number of weather data sources such as the Weather Network (http://www.theweathernetwork.com/) or Weather Underground (http://www.wunderground.com/). These additional data sources may provide better answers in some cases and different systems take advantage of these to differing degrees. Since different queries can be better answered by different data sources (and, sometimes, even by different search engines), an important aspect of this processing stage is the choice of the appropriate search engine(s)/data source(s) to consult for a given query. The naive alternative of submitting the queries to all search engines and data sources is not a wise decision, since these operations are quite costly over the web. Usually, the category information is used to assist the choice of the appropriate sources, along with a ranked listing of sources and engines for different categories. For each search engine and data source, wrappers need to be written to convert the query into the format of that data source/search engine and convert the returned result documents into a common format for further analysis.

In response to queries, search engines return links to the documents together with short snippets, while other data sources return results in a variety of formats.

The returned results are normalized into "records." The direct answers need to be extracted from these records, which is the function of the answer extraction phase. Various text processing techniques can be used to match the keywords to (possibly parts of) the returned records. Subsequently, these results need to be ranked using various information retrieval techniques (e.g., word frequencies, inverse document frequency). In this process, the category information that is generated during question analysis is used. Different systems employ different notions of the appropriate answer. Some return a ranked list of direct answers (e.g., if the question is "Who invented the telephone," they would return "Alexander Graham Bell" or "Graham Bell" or "Bell," or all of them in ranked order[4]), while others return a ranked order of the portion of the records that contain the keywords in the query (i.e., a summary of the relevant portion of the document).

Question answering systems are very different than the other web querying approaches we have discussed in previous sections. They are more flexible in what they offer users in terms of querying without any knowledge of the organization of web data. On the other hand, they are constrained by idiosynchrocies of natural language, and the difficulties of natural language processing.

12.5 Searching and Querying the Hidden Web

Currently, most general-purpose search engines only operate on the PIW while a considerable amount of the valuable data are kept in hidden databases, either as relational data, as embedded documents, or in many other forms. The current trend in web search is to find ways to search the hidden web as well as the PIW, for two main reasons. First is the size—the size of the hidden web (in terms of generated HTML pages) is considerably larger than the PIW, therefore the probability of finding answers to users' queries is much higher if the hidden web can also be searched. The second is in data quality—the data stored in the hidden web are usually of much higher quality than those found on public web pages since they are properly curated. If they can be accessed, the quality of answers can be improved.

However, searching the hidden web faces many challenges, the most important of which are the following:

1. Ordinary crawlers cannot be used to search the hidden web, since there are neither HTML pages, nor hyperlinks to crawl.
2. Usually, the data in hidden databases can be only accessed through a search interface or a special interface, requiring access to this interface.
3. In most (if not all) cases, the underlying structure of the database is unknown, and the data providers are usually reluctant to provide any information about their data that might help in the search process (possibly due to the overhead

[4]The inventor of the telephone is a subject of controversy, with multiple claims to the invention. We'll go with Bell in this example since he was the first one to patent the device.

of collecting this information and maintaining it). One has to work through the interfaces provided by these data sources.

In the remainder of this section, we discuss these issues as well as some proposed solutions.

12.5.1 Crawling the Hidden Web

One approach to address the issue of searching the hidden web is to try crawling in a manner similar to that of the PIW. As already mentioned, the only way to deal with hidden web databases is through their search interfaces. A hidden web crawler should be able to perform two tasks: (a) submit queries to the search interface of the database, and (b) analyze the returned result pages and extract relevant information from them.

12.5.1.1 Querying the Search Interface

One approach is to analyze the search interface of the database, and build an internal representation for it. This internal representation specifies the fields used in the interface, their types (e.g., text boxes, lists, checkboxes, etc.), their domains (e.g., specific values as in lists, or just free text strings as in text boxes), and also the labels associated with these fields. Extracting these labels requires an exhaustive analysis of the HTML structure of the page.

Next, this representation is matched with the system's task-specific database. The matching is based on the labels of the fields. When a label is matched, the field is then populated with the available values for this field. The process is repeated for all possible values of all fields in the search form, and the form is submitted with every combination of values and the results are retrieved.

Another approach is to use agent technology. In this case, *hidden web agents* are developed that interact with the search forms and retrieve the result pages. This involves three steps: (a) finding the forms, (b) learning to fill the forms, and (c) identifying and fetching the target (result) pages.

The first step is accomplished by starting from a URL (an entry point), traversing links, and using some heuristics to identify HTML pages that contain forms, excluding those that contain password fields (e.g., login, registration, purchase pages). The form filling task depends on identifying labels and associating them with form fields. This is achieved using some heuristics about the location of the label relative to the field (on the left or above it). Given the identified labels, the agent determines the application domain that the form belongs to, and fills the fields with values from that domain in accordance with the labels (the values are stored in a repository accessible to the agent).

12.5.1.2 Analyzing the Result Pages

Once the form is submitted, the returned page has to be analyzed, for example, to see if it is a data page or a search-refining page. This can be achieved by matching values in this page with values in the agent's repository. Once a data page is found, it is traversed, as well as all pages that it links to (especially pages that have more results), until no more pages can be found that belong to the same domain.

However, the returned pages usually contain a lot of irrelevant data, in addition to the actual results, since most of the result pages follow some template that has a considerable amount of text used only for presentation purposes. A method to identify web page templates is to analyze the textual contents and the adjacent tag structures of a document in order to extract query-related data. A web page is represented as a sequence of text segments, where a text segment is a piece of tag encapsulated between two tags. The mechanism to detect templates is as follows:

1. Text segments of documents are analyzed based on textual contents and their adjacent tag segments.
2. An initial template is identified by examining the first two sample documents.
3. The template is then generated if matched text segments along with their adjacent tag segments are found from both documents.
4. Subsequent retrieved documents are compared with the generated template. Text segments that are not found in the template are extracted for each document to be further processed.
5. When no matches are found from the existing template, document contents are extracted for the generation of future templates.

12.5.2 Metasearching

Metasearching is another approach for querying the hidden web. Given a user query, a metasearcher performs the following tasks:

1. Database selection: selecting the databases(s) that are most relevant to the user's query. This requires collecting some information about each database. This information is known as a *content summary*, which is statistical information, usually including the *document frequencies* of the words that appear in the database.
2. Query translation: translating the query to a suitable form for each database (e.g., by filling certain fields in the database's search interface).
3. Result merging: collecting the results from the various databases, merging them (and most probably, ordering them), and returning them to the user.

We discuss the important phases of metasearching in more detail below.

12.5.2.1 Content Summary Extraction

The first step in metasearching is to compute content summaries. In most of the cases, the data providers are not willing to go through the trouble of providing this information. Therefore, the metasearcher itself extracts this information.

A possible approach is to extract a document sample set from a given database D and compute the frequency of each observed word w in the sample, $SampleDF(w)$. The technique works as follows:

1. Start with an empty content summary where $SampleDF(w) = 0$ for each word w, and a general (i.e., not specific to D), comprehensive word dictionary.
2. Pick a word and send it as a query to database D.
3. Retrieve the top-k documents from among the returned documents.
4. If the number of retrieved documents exceeds a prespecified threshold, stop. Otherwise continue the sampling process by returning to Step 2.

There are two main versions of this algorithm that differ in how Step 2 is executed. One of the algorithms picks a random word from the dictionary. The second algorithm selects the next query from among the words that have been already discovered during sampling. The first constructs better profiles, but is more expensive.

An alternative is to use a focused probing technique that can actually classify the databases into a hierarchical categorization. The idea is to preclassify a set of training documents into some categories, and then extract different terms from these documents and use them as query probes for the database. The single-word probes are used to determine the *actual* document frequencies of these words, while only *sample* document frequencies are computed for other words that appear in longer probes. These are used to estimate the actual document frequencies for these words.

Yet another approach is to start by randomly selecting a term from the search interface itself, assuming that, most probably, this term will be related to the contents of the database. The database is queried for this term, and the top-k documents are retrieved. A subsequent term is then randomly selected from terms extracted from the retrieved documents. The process is repeated until a predefined number of documents are retrieved, and then statistics are calculated based on the retrieved documents.

12.5.2.2 Database Categorization

A good approach that can help the database selection process is to categorize the databases into several categories (for example, as Yahoo directory). Categorization facilitates locating a database given a user's query, and makes most of the returned results relevant to the query.

If the focused probing technique is used for generating content summaries, then the same algorithm can probe each database with queries from some category and

count the number of matches. If the number of matches exceeds a certain threshold, the database is said to belong to this category.

Database Selection

Database selection is a crucial task in the metasearching process, since it has a critical impact on the efficiency and effectiveness of query processing over multiple databases. A database selection algorithm attempts to find the best set of databases, based on information about the database contents, on which a given query should be executed. Usually, this information includes the number of different documents that contain each word (known as the document frequency), as well as some other simple related statistics, such as the number of documents stored in the database. Given these summaries, a database selection algorithm estimates how relevant each database is for a given query (e.g., in terms of the number of matches that each database is expected to produce for the query).

GlOSS is a simple database selection algorithm that assumes that query words are independently distributed over database documents to estimate the number of documents that match a given query. GlOSS is an example of a large family of database selection algorithms that rely on content summaries. Furthermore, database selection algorithms expect such content summaries to be accurate and up-to-date.

The focused probing algorithm discussed above exploits the database categorization and content summaries for database selection. This algorithm consists of two basic steps: (1) propagate the database content summaries to the categories of the hierarchical classification scheme, and (2) use the content summaries of categories and databases to perform database selection hierarchically by zooming in on the most relevant portions of the topic hierarchy. This results in more relevant answers to the user's query since they only come from databases that belong to the same category as the query itself.

Once the relevant databases are selected, each database is queried, and the returned results are merged and sent back to the user.

12.6 Web Data Integration

In Chap. 7 we discussed the integration of databases, each of which have well-defined schemas. The techniques discussed in that chapter are mostly appropriate when enterprise data is considered. When we wish to provide integrated access to web data sources, the problem becomes more complex—all the characteristics of "big data" play a role. In particular, the data may not have a proper schema, and if it does, the data sources are so varied that the schemas are widely different, making schema matching a real challenge. In addition, the amount of data and even the number of data sources are significantly higher than in an enterprise environment, making manual curation all but impossible. The quality of the data on the web is also

far more suspect than the enterprise data collections that we considered previously, and this increases the importance of data cleaning solutions.

An appropriate approach to web data integration is what is called *pay-as-you-go integration* where the up-front investment to integrate data is significantly reduced, eliminating some of the stages discussed in Chap. 7. Instead, a framework and basic infrastructure is provided for data owners to easily integrate their datasets into a federation. One proposal for the pay-as-you-go approach to web data integration is *data spaces*, which advocates that there should be lightweight integration platform with perhaps rudimentary access opportunities (e.g., keyword search) to start with, and ways to improve the value of the integration over time by providing the opportunity to develop tools for more sophisticated use. Perhaps the *data lakes* that have started to receive attention and that we discussed in Chap. 10, are more advanced versions of the data space proposal.

In this section, we cover some of the approaches that have been developed to address these challenges. In particular, we will look at web tables and fusion tables (Sect. 12.6.1) as a low-overhead integration approach for tabular structured data. We then look at the semantic web and the Linked Open Data (LOD) approach to web data integration (Sect. 12.6.2.3). Finally, we discuss the issues of data cleaning and the use of machine learning techniques in data integration and cleaning at web-scale integration in Sect. 12.6.3.

12.6.1 Web Tables/Fusion Tables

Two popular approaches to lightweight web data integration are *web portals* and *mashups* that aggregate web and other data on specific topics such as travel, hotel bookings, etc. The two differ in the technologies that they use, but that is not important for our discussion. These are examples of "vertically integrated" systems where each mashup or portal targets one domain.

A first question that comes up in developing a mashup is how to find the relevant web data. Web tables project is an early attempt at finding data on the web that has relational table structure and provide access over these tables (the so-called "database-like" tables). The focus is on the open web, and tables in the deep web are not considered as their discovery is a much more difficult problem. Even finding the database-like tables in the open web is not easy since the usual relational table structures (i.e., attribute names) may not exist. Web tables employ a classifier that can group HTML tables as relational and nonrelational. It then provides tools to extract a schema and maintain statistics about the schemas that can be used in search over these tables. Join opportunities across tables are introduced to allow more sophisticated navigation across the discovered tables. Web tables can be viewed as a method to retrieve and query web data, but they also serve as a virtual integration framework for web data with global schema information.

Fusion tables project at Google takes web tables a step further by allowing users to upload their own tables (in various formats) in addition to the discovered