13

Distributed and Parallel Database Systems

13.1 Introduction

The maturation of database management system (DBMS) technology [1] has coincided with significant developments in distributed computing and parallel processing technologies. The end result is the development of distributed DBMSs and parallel DBMSs that are now the dominant data management tools for highly data-intensive applications. With the emergence of cloud computing, distributed and parallel database systems have started to converge.

A parallel computer, or multiprocessor, is itself a distributed system made of a number of nodes (with processors, memories, and disks) connected by a high-speed network within a cabinet. Distributed database technology can be naturally extended to implement parallel database systems, that is, database systems on parallel computers [2,3]. Parallel database systems exploit the parallelism in data management in order to deliver high-performance and high-availability database servers.

In this chapter, we present an overview of the distributed DBMS and parallel DBMS technologies, highlight the unique characteristics of each, and indicate the similarities between them. We also discuss the new challenges and emerging solutions.

13.2 Underlying Principles

The fundamental principle behind data management is data independence, which enables applications and users to share data at a high conceptual level while ignoring implementation details. This principle has been achieved by database systems that provide advanced capabilities such as schema management, high-level query languages, access control, automatic query processing and optimization, transactions, and data structures for supporting complex objects.

A distributed database [1] is a collection of multiple, logically interrelated databases distributed over a computer network. A distributed database system is defined as the software system that permits
the management of the distributed database and makes the distribution transparent to the users. Distribution transparency extends the principle of data independence so that distribution is not visible to users.

These definitions assume that each site logically consists of a single, independent computer. Therefore, each site has the capability to execute applications on its own. The sites are interconnected by a computer network with loose connection between sites that operate independently. Applications can then issue queries and transactions to the distributed database system that transforms them into local queries and local transactions and integrates the results.

The database is physically distributed across the data sites by fragmenting and replicating the data. Given a relational database schema, for instance, fragmentation subdivides each relation into partitions based on some function applied to some tuples’ attributes. Based on the user access patterns, each of the fragments may also be replicated to improve locality of reference (and thus performance) and availability. The use of a set-oriented data model (like relational) has been crucial to define fragmentation, based on data subsets.

The functions provided by a distributed database system could be those of a database system (schema management, access control, query processing, transaction support, etc.). But since they must deal with distribution, they are more complex to implement. Therefore, many systems support only a subset of these functions. When the data and the databases already exist, one is faced with the problem of providing integrated access to heterogeneous data. This process is known as data integration, which consists of defining a global schema over the existing data and mappings between the global schema and the local database schemas. Data integration systems have received, over time, several other names such as federated database systems, multidatabase systems (MDBMS), and mediator systems. Standard protocols such as Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC) ease data integration using standard query language (SQL). In the context of the web, mediator systems allow general access to autonomous data sources (such as files, databases, and documents) in read-only mode. Thus, they typically do not support all database functions such as transactions and replication.

When the architectural assumption of each site being a (logically) single, independent computer is relaxed, one gets a parallel database system, that is, a database system implemented on a tightly coupled multiprocessor or a cluster. The main difference with a distributed database system is that there is a single operating system which facilitates implementation, and the network is typically faster and more reliable. The objective of parallel database systems is high performance and high availability. High performance (i.e., improving transaction throughput or query response time) is obtained by exploiting data partitioning and query parallelism, while high availability is obtained by exploiting replication. Again, this has been made possible by the use of a set-oriented data model, which eases parallelism—in particular, independent parallelism between data subsets.

There are three forms of parallelism inherent in data-intensive application workloads. Inter-query parallelism enables the parallel execution of multiple queries generated by concurrent transactions. Intra-query parallelism makes the parallel execution of multiple, independent operations (e.g., select operations) possible within the same query. Both inter-query and intra-query parallelism can be obtained by using data partitioning, which is similar to horizontal fragmentation. Finally, with intra-operation parallelism, the same operation can be executed as many sub-operations using function partitioning in addition to data partitioning. The set-oriented mode of database languages (e.g., SQL) provides many opportunities for intra-operation parallelism.

In a distributed or parallel environment, it should be easier to accommodate increasing database sizes or increasing performance demands. Major system overhauls are seldom necessary; expansion can usually be handled by adding more processing and storage power to the system.

Ideally, a parallel DBMS (and to a lesser degree a distributed DBMS) should demonstrate two advantages: linear scaleup and linear speedup. Linear scaleup refers to a sustained performance for a linear increase in both database size and processing and storage power. Linear speedup refers to a linear increase
Distributed and Parallel Database Systems

in performance for a constant database size and a linear increase in processing and storage power. Furthermore, extending the system should require minimal reorganization of the existing database.

There are a number of possible ways in which a distributed DBMS may be architected. We use a classification (Figure 13.1) that organizes the systems as characterized with respect to three dimensions: (1) the autonomy of local systems, (2) their distribution, and (3) their heterogeneity. Autonomy, in this context, refers to the distribution of control, not of data. It indicates the degree to which individual DBMSs can operate independently. Whereas autonomy refers to the distribution (or decentralization) of control, the distribution dimension of the taxonomy deals with the physical distribution of data over multiple sites (or nodes in a parallel system). There are a number of ways DBMSs have been distributed. We distinguish between client/server (C/S) distribution and peer-to-peer (P2P) distribution (or full distribution). With C/S DBMS, sites may be clients or servers, thus with different functionality, whereas with homogeneous P2P DBMS (DDBMS in Figure 13.1), all sites provide the same functionality. Note that DDBMS came before C/S DBMS in the late 1970s. P2P data management struck back in the 2000 with modern variations to deal with very-large-scale autonomy and decentralized control. Heterogeneity refers to data models, query languages, and transaction management protocols. MDBMS deal with heterogeneity, in addition to autonomy and distribution.

13.3 Distributed and Parallel Database Technology

Distributed and parallel DBMSs provide the same functionality as centralized DBMSs except in an environment where data are distributed across the sites on a computer network or across the nodes of a parallel system. As discussed earlier, the users are unaware of data distribution. Thus, these systems provide the users with a logically integrated view of the physically distributed database. Maintaining this view places significant challenges on system functions. We provide an overview of these new challenges in this section. We assume familiarity with basic database management techniques.

13.3.1 Query Processing and Optimization

Query processing is the process by which a declarative query is translated into low-level data manipulation operations. SQL is the standard query language that is supported in current DBMSs.
Query optimization refers to the process by which the “best” execution strategy for a given query is found from among a set of alternatives.

In centralized DBMSs, the process typically involves two steps [1]: query decomposition and query optimization. Query decomposition takes an SQL query and translates it into one expressed in relational algebra. In the process, the query is analyzed semantically so that incorrect queries are detected and rejected as easily as possible, and correct queries are simplified. Simplification involves the elimination of redundant predicates, which may be introduced as a result of query modification to deal with views, security enforcement, and semantic integrity control. The simplified query is then restructured as an algebraic query.

For a given SQL query, there are usually more than one possible algebraic equivalents. Some of these algebraic queries are “better” than others. The quality of an algebraic query is defined in terms of expected performance. The traditional procedure is to obtain an initial algebraic query by translating the predicates and the target statement into relational operations as they appear in the query. This initial algebraic query is then transformed, using algebraic transformation rules, into other algebraic queries until the “best” one is found.* The query optimizer is usually seen as three components: a search space, a cost model, and a search strategy. The search space is the set of alternative execution plans to represent the input query. These plans are equivalent, in the sense that they yield the same result but they differ on the execution order of operations and the way these operations are implemented. The cost model predicts the cost of a given execution plan. To be accurate, the cost model must have accurate knowledge about the parallel execution environment. The search strategy explores the search space and selects the best plan. It defines which plans are examined and in which order.

In distributed DBMSs, two additional steps are involved between query decomposition and query optimization: data localization and global query optimization [1].

The input to data localization is the initial algebraic query generated by the query decomposition step. The initial algebraic query is specified on global relations irrespective of their fragmentation or distribution. The main role of data localization is to localize the query’s data using data distribution information. In this step, the fragments that are involved in the query are determined, and the query is transformed into one that operates on fragments rather than on global relations. As indicated earlier, fragmentation is defined through fragmentation rules that can be expressed as relational operations (horizontal fragmentation by selection, vertical fragmentation by projection). A distributed relation can be reconstructed by applying the inverse of the fragmentation rules. This is called a localization program. The localization program for a horizontally (vertically) fragmented query is the union (join) of the fragments. Thus, during the data localization step, each global relation is first replaced by its localization program, and then the resulting fragment query is simplified and restructured to produce another “good” query. Simplification and restructuring may be done according to the same rules used in the decomposition step. As in the decomposition step, the final fragment query is generally far from optimal; the process has only eliminated “bad” algebraic queries.

The input to global query optimization step is a fragment query, that is, an algebraic query on fragments. The goal of global query optimization is to find an execution strategy for the query which is close to optimal. Remember that finding the optimal solution is computationally intractable. An execution strategy for a distributed query can be described with relational algebra operations and communication primitives (send/receive operations) for transferring data between sites. The previous layers have already performed some optimizations—for example, by eliminating redundant expressions. However, this optimization is independent of fragment characteristics such as cardinalities. In addition, communication operations are not yet specified. By permuting the ordering of operations within one fragment query, many equivalent query execution plans may be found.

* The difference between an optimal plan and the best plan is that the optimizer does not, because of computational intractability, examine all of the possible plans.
In a distributed environment, the cost function, often defined in terms of time units, refers to computing resources such as disk space, disk I/Os, buffer space, central processing unit (CPU) cost, and communication cost. Generally, it is a weighted combination of I/O, CPU, and communication costs. To select the ordering of operations, it is necessary to predict execution costs of alternative candidate orderings. Determining execution costs before query execution (i.e., static optimization) is based on fragment statistics and the formulas for estimating the cardinalities of results of relational operations. Thus the optimization decisions depend on the available statistics on fragments. An important aspect of query optimization is join ordering, since permutations of the joins within the query may lead to improvements of several orders of magnitude. One basic technique for optimizing a sequence of distributed join operations is through the use of the semijoin operator. The main value of the semijoin in a distributed system is to reduce the size of the join operands and thus the communication cost. However, more recent techniques, which consider local processing costs as well as communication costs, do not use semijoins because they might increase local processing costs. The output of the query optimization layer is an optimized algebraic query with communication operations included on fragments.

Parallel query optimization exhibits similarities with distributed query processing [4,5]. It takes advantage of both intra-operation parallelism, which was discussed earlier, and inter-operation parallelism.

Intra-operation parallelism is achieved by executing an operation on several nodes of a multiprocessor machine. This requires that the operands have been previously partitioned, that is, horizontally fragmented, across the nodes [6]. The way in which a base relation is partitioned is a matter of physical design. Typically, partitioning is performed by applying a hash function on an attribute of the relation, which will often be the join attribute. The set of nodes where a relation is stored is called its home. The home of an operation is the set of nodes where it is executed. The home of an operation must also be the home of its operand relations in order for the operation to access its operands. For binary operations such as join, this might imply repartitioning one of the operands [7]. The optimizer might even sometimes find that repartitioning both operands is useful. Parallel optimization to exploit intra-operation parallelism can make use of some of the techniques devised for distributed databases.

Inter-operation parallelism occurs when two or more operations are executed in parallel, either as a dataflow or independently. We designate as dataflow the form of parallelism induced by pipelining. Independent parallelism occurs when operations are executed at the same time or in an arbitrary order. Independent parallelism is possible only when the operations do not involve the same data.

### 13.3.2 Concurrency Control

Whenever multiple users access (read and write) a shared database, these accesses need to be synchronized to ensure database consistency. The synchronization is achieved by means of concurrency control algorithms, which enforce a correctness criterion such as serializability. User accesses are encapsulated as transactions [8], whose operations at the lowest level are a set of read and write operations to the database. Transactions typically have four properties: atomicity, consistency, isolation, and durability, which are collectively known as ACID properties. Concurrency control algorithms enforce the isolation property of transaction execution, which states that the effects of one transaction on the database are isolated from other transactions until the first completes its execution.

The most popular concurrency control algorithms are locking-based. In such schemes, a lock, in either shared or exclusive mode, is placed on some unit of storage (usually a page) whenever a transaction attempts to access it. These locks are placed according to lock compatibility rules such that read–write, write–read, and write–write conflicts are avoided. It is a well-known theorem that if lock actions on behalf of concurrent transactions obey the following simple rule, then it is possible to ensure the serializability of these transactions: “No lock on behalf of a transaction should be set once a lock previously held by the transaction is released.” This is known as two-phase locking [9], since transactions go through a growing phase when they obtain locks and a shrinking phase when they release locks.
In general, releasing of locks prior to the end of a transaction is problematic. Thus, most of the locking-based concurrency control algorithms are strict in that they hold on to their locks until the end of the transaction.

In distributed DBMSs, the challenge is to extend both the serializability argument and the concurrency control algorithms to the distributed execution environment. In these systems, the operations of a given transaction may execute at multiple sites where they access data. In such a case, the serializability argument is more difficult to specify and enforce. The complication is due to the fact that the serialization order of the same set of transactions may be different at different sites. Therefore, the execution of a set of distributed transactions is serializable if and only if

1. The execution of the set of transactions at each site is serializable.
2. The serialization orders of these transactions at all these sites are identical.

Distributed concurrency control algorithms enforce this notion of global serializability. In locking-based algorithms, there are three alternative ways of enforcing global serializability: centralized locking, primary copy locking, and distributed locking.

In centralized locking, there is a single lock table for the entire distributed database. This lock table is placed under the control of a single lock manager at one of the sites. The lock manager is responsible for setting and releasing locks on behalf of transactions. Since all locks are managed at one site, this is similar to centralized concurrency control, and it is straightforward to enforce the global serializability rule. These algorithms are simple to implement, but suffer from two problems. The central site may become a bottleneck, both because of the amount of work it is expected to perform and because of the traffic that is generated around it; and the system may be less reliable since the failure or inaccessibility of the central site would cause system unavailability.

Primary copy locking is a concurrency control algorithm that is useful in replicated databases where there may be multiple copies of a data item stored at different sites. One of the copies is designated as a primary copy, and it is this copy that has to be locked in order to access that item. The set of primary copies for each data item is known to all the sites in the distributed system, and the lock requests on behalf of transactions are directed to the appropriate primary copy. If the distributed database is not replicated, primary copy locking degenerates into a distributed locking algorithm.

In distributed (or decentralized) locking, the lock management duty is shared by all the sites in the system. The execution of a transaction involves the participation and coordination of lock managers at more than one site. Locks are obtained at each site where the transaction accesses a data item. Distributed locking algorithms do not have the overhead of centralized locking ones. However, both the communication overhead to obtain all the locks and the complexity of the algorithm are greater.

One side effect of all locking-based concurrency control algorithms is that they cause deadlocks. The detection and management of deadlocks in a distributed system is difficult. Nevertheless, the relative simplicity and better performance of locking algorithms make them more popular than alternatives such as timestamp-based algorithms or optimistic concurrency control. Timestamp-based algorithms execute the conflicting operations of transactions according to their timestamps, which are assigned when the transactions are accepted. Optimistic concurrency control algorithms work from the premise that conflicts among transactions are rare and proceed with executing the transactions up to their termination at which point a validation is performed. If the validation indicates that serializability would be compromised by the successful completion of that particular transaction, then it is aborted and restarted.

13.3.3 Reliability Protocols

We indicated earlier that distributed DBMSs are potentially more reliable because there are multiples of each system component, which eliminates single point of failure. This requires careful system design and the implementation of a number of protocols to deal with system failures.
In a distributed DBMS, four types of failures are possible: transaction failures, site (system) failures, media (disk) failures, and communication line failures. Transactions can fail for a number of reasons. Failure can be due to an error in the transaction caused by input data, as well as the detection of a present or potential deadlock. The usual approach that is followed in cases of transaction failure is to abort the transaction, resetting the database to its state prior to the start of the database.

Site (or system) failures are due to a hardware failure (e.g., processor, main memory, and power supply) or a software failure (bugs in system or application code). The effect of system failures is the loss of main memory contents. Therefore, any update to the parts of the database that are in the main memory buffers (also called volatile database) is lost as a result of system failures. However, the database that is stored in secondary storage (also called stable database) is safe and correct. To achieve this, DBMSs typically employ logging protocols, such as Write-Ahead Logging [9], which record changes to the database in system logs and move these log records and the volatile database pages to stable storage at appropriate times. From the perspective of distributed transaction execution, site failures are important since the failed sites cannot participate in the execution of any transaction.

Media failures refer to the failure of secondary storage devices that store the stable database. Typically, these failures are addressed by duplexing storage devices and maintaining archival copies of the database. Media failures are frequently treated as problems local to one site and therefore are not specifically addressed in the reliability mechanisms of distributed DBMSs.

The three types of failures described earlier are common to both centralized and distributed DBMSs. Communication failures, on the other hand, are unique to distributed systems. There are a number of types of communication failures. The most common ones are errors in the messages, improperly ordered messages, lost (or undelivered) messages, and line failures. Generally, the first two of these are considered to be the responsibility of the computer network protocols and are not addressed by the distributed DBMS. The last two, on the other hand, have an impact on the distributed DBMS protocols and, therefore, need to be considered in the design of these protocols. If one site is expecting a message from another site and this message never arrives, this may be because (a) the message is lost, (b) the line(s) connecting the two sites may be broken, or (c) the site which is supposed to send the message may have failed. Thus, it is not always possible to distinguish between site failures and communication failures. The waiting site simply timeouts and has to assume that the other site is unable to communicate. Distributed DBMS protocols have to deal with this uncertainty. One drastic result of line failures may be network partitioning in which the sites form groups where communication within each group is possible but communication across groups is not. This is difficult to deal with in the sense that it may not be possible to make the database available for access while at the same time guaranteeing its consistency.

Two properties of transactions are maintained by reliability protocols: atomicity and durability. Atomicity requires that either all the operations of a transaction are executed or none of them are (all-or-nothing). Thus, the set of operations contained in a transaction is treated as one atomic unit. Atomicity is maintained even in the face of failures. Durability requires that the effects of successfully completed (i.e., committed) transactions endure subsequent failures.

The enforcement of atomicity and durability requires the implementation of atomic commitment protocols and distributed recovery protocols. The most popular atomic commitment protocol is two-phase commit (2PC). The recoverability protocols are built on top of the local recovery protocols, which are dependent upon the supported mode of interaction (of the DBMS) with the operating system [10,11].

2PC is a very simple and elegant protocol that ensures the atomic commitment of distributed transactions. It extends the effects of local atomic commit actions to distributed transactions by insisting that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent (i.e., all sites terminate the transaction in the same manner). If all the sites agree to commit a transaction, then all the actions of the distributed transaction take effect; if one of the
sites declines to commit the operations at that site, then all of the other sites are required to abort the transaction. Thus, the fundamental 2PC rule states

1. If even one site rejects to commit (which means it votes to abort) the transaction, the distributed transaction has to be aborted at each site where it executes.
2. If all the sites vote to commit the transaction, the distributed transaction is committed at each site where it executes.

The simple execution of the 2PC protocol is as follows. There is a coordinator process at the site where the distributed transaction originates and participant processes at all the other sites where the transaction executes. There are two rounds of message exchanges between the coordinator and the participants (hence the name 2PC protocol): in the first phase, the coordinator asks for votes regarding the transaction from the participants and determines the fate of the transaction according to the 2PC rule, and, in the second phase, it informs the participants about the decision. In the end, all of the participants and the coordinator reach the same decision. Two important variants of 2PC are the presumed abort 2PC and presumed commit 2PC [12]. These are important because they reduce the message and I/O overhead of the protocols. Presumed abort protocol is included in the X/Open XA standard and has been adopted as part of the International Standards Organization (ISO) standard for Open Distributed Processing.

One important characteristic of 2PC protocol is its blocking nature. Failures can occur during the commit process. As discussed earlier, the only way to detect these failures is by means of a time-out of the process waiting for a message. When this happens, the process (coordinator or participant) that timeouts follows a termination protocol to determine what to do with the transaction that was in the middle of the commit process. A non-blocking commit protocol is one whose termination protocol can determine what to do with a transaction in case of failures under any circumstance. In the case of 2PC, if a site failure occurs at the coordinator site and one participant site while the coordinator is collecting votes from the participants, the remaining participants cannot determine the fate of the transaction among themselves, and they have to remain blocked until the coordinator or the failed participant recovers. During this period, the locks that are held by the transaction cannot be released, which reduces the availability of the database.

### 13.3.4 Replication Protocols

In replicated distributed databases,* each logical data item has a number of physical instances. For example, the salary of an employee (logical data item) may be stored at three sites (physical copies). The issue in this type of a database system is to maintain some notion of consistency among the copies. The most discussed consistency criterion is one copy equivalence, which asserts that the values of all copies of a logical data item should be identical when the transaction that updates it terminates.

If replication transparency is maintained, transactions will issue read and write operations on a logical data item x. The replica control protocol is responsible for mapping operations on x to operations on physical copies of x (x₁, ..., xₙ). A typical replica control protocol that enforces one copy serializability is known as Read-Once/Write-All (ROWA) protocol. ROWA maps each read on x [Read(x)] to a read on one of the physical copies xᵢ [Read(xᵢ)]. The copy that is read is insignificant from the perspective of the replica control protocol and may be determined by performance considerations. On the other hand, each write on logical data item x is mapped to a set of writes on all copies of x.

ROWA protocol is simple and straightforward, but it requires that all copies of all logical data items that are updated by a transaction be accessible for the transaction to terminate. Failure of one site may block a transaction, reducing database availability.

---

* Replication is not a significant concern in parallel DBMSs because the data are normally not replicated across multiple processors. Replication may occur as a result of data shipping during query optimization, but this is not managed by the replica control protocols.
A number of alternative algorithms have been proposed, which reduce the requirement that all copies of a logical data item be updated before the transaction can terminate. They relax ROWA by mapping each write to only a subset of the physical copies.

This idea of possibly updating only a subset of the copies, but nevertheless successfully terminating the transaction, has formed the basis of quorum-based voting for replica control protocols. The majority consensus algorithm can be viewed from a slightly different perspective: It assigns equal votes to each copy and a transaction that updates that logical data item can successfully complete as long as it has a majority of the votes. Based on this idea, an early quorum-based voting algorithm [13] assigns a (possibly unequal) vote to each copy of a replicated data item. Each operation then has to obtain a read quorum ($V_r$) or a write quorum ($V_w$) to read or write a data item, respectively. If a given data item has a total of $V$ votes, the quorums have to obey the following rules:

1. $V_r + V_w > V$ (a data item is not read and written by two transactions concurrently, avoiding the read–write conflict)
2. $V_w > V/2$ (two write operations from two transactions cannot occur concurrently on the same data item, avoiding write–write conflict)

The difficulty with this approach is that transactions are required to obtain a quorum even to read data. This significantly and unnecessarily slows down read access to the database. An alternative quorum-based voting protocol that overcomes this serious performance drawback [14] has also been proposed. However, this protocol makes unrealistic assumptions about the underlying communication system. It requires that failures that change the network’s topology are detected by all sites instantaneously, and that each site has a view of the network consisting of all the sites with which it can communicate. In general, communication networks cannot guarantee to meet these requirements. The single copy equivalence replica control protocols are generally considered to be restrictive in terms of the availability they provide. Voting-based protocols, on the other hand, are considered too complicated with high overheads. Therefore, these techniques are not used in current distributed DBMS products. More flexible replication schemes have been investigated where the type of consistency between copies is under user control. A number of replication servers have been developed or are being developed with this principle.

### 13.4 New Challenges and Emerging Solutions

The pervasiveness of the web has spurred all kinds of data-intensive applications and introduced significant challenges for distributed data management [15]. New data-intensive applications such as social networks, web data analytics, and scientific applications have requirements that are not met by the traditional distributed database systems in Figure 13.1. What has changed and made the problems much harder is the scale of the dimensions: very-large-scale distribution, very high heterogeneity, and high autonomy. In this section, we discuss some of these challenges and the emerging solutions to these challenges.

#### 13.4.1 Cloud Data Management

Cloud computing is the latest trend in distributed computing and has been the subject of much debate. The vision encompasses on demand, reliable services provided over the Internet (typically represented as a cloud) with easy access to virtually infinite computing, storage, and networking resources. Through very simple web interfaces and at small incremental cost, users can outsource complex tasks, such as data storage, system administration, or application deployment, to very large data centers operated by cloud providers. Thus, the complexity of managing the software/hardware infrastructure gets shifted from the users’ organization to the cloud provider. From a technical point of view, the grand challenge is to support in a cost-effective way the very large scale of the infrastructure, which has to manage lots of users and resources with high quality of service.
However, not all data-intensive applications are good candidates for being supported in a cloud [16]. To simplify, we can classify between the two main classes of data-intensive applications: Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). OLTP deals with operational databases of average sizes (up to a few terabytes), is write-intensive, and requires complete ACID transactional properties, strong data protection, and response time guarantees. OLAP, on the other hand, deals with historical databases of very large sizes (up to petabytes), is read-intensive, and thus can accept relaxed ACID properties. Furthermore, since OLAP data are typically extracted from operational OLTP databases, sensitive data can be simply hidden for analysis (e.g., using anonymization) so that data protection is not as crucial as in OLTP.

OLAP is more suitable than OLTP for cloud primarily because of two cloud characteristics [16]: elasticity and security. To support elasticity in a cost-effective way, the best solution that most cloud providers adopt is a shared-nothing cluster, which has a fully distributed architecture, where each node is made of processor, main memory, and disk and communicates with other nodes through message passing. Shared-nothing provides high scalability but requires careful data partitioning. Since OLAP databases are very large and mostly read-only, data partitioning and parallel query processing are effective [4]. However, it is much harder to support OLTP on shared-nothing systems because of ACID guarantees that require complex concurrency control. For these reasons and because OLTP databases are not very large, shared disk, where any processor has direct access to any disk unit through the interconnection network, is the preferred architecture for OLTP.

The second reason that OLTP is not so suitable for cloud is that highly sensitive data get stored at an untrusted host (the provider site). Storing corporate data at an untrusted third-party, even with a carefully negotiated Service Level Agreement with a reliable provider, creates resistance from some customers because of security issues. However, this resistance is much reduced for historical data, with anonymized sensitive data.

There is much more variety in cloud data than in scientific data since there are many different kinds of customers (individuals, small- and medium-size enterprises, large corporations, etc.). However, we can identify common features. Cloud data can be very large, unstructured (e.g., text-based) or semi-structured, and typically append-only (with rare updates). Cloud users and application developers may be in high numbers, but not DBMS experts.

Generic data management solutions (e.g., relational DBMS) that have proven effective in many application domains (e.g., business transactions) are not efficient at dealing with emerging cloud applications, thereby forcing developers to build ad hoc solutions that are labor-intensive and that cannot scale. In particular, relational DBMSs have been lately 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, XML documents, and new functions), this has resulted in a loss of performance and flexibility for applications with specific requirements because they provide both too much and too little. 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 as in traditional row-oriented relational DBMSs, have been shown to perform more than an order of magnitude better on OLAP workloads. The “one-size-does-not-fit-all” counterargument generally applies to cloud data management as well.

Therefore, current data management solutions for the cloud have traded consistency for scalability, simplicity, and flexibility. As alternative to relational DBMS (which use the standard SQL language), these solutions have been recently quoted as “Not Only SQL” (NoSQL) by the database research community. Many of these solutions exploit large-scale parallelism, typically with shared-nothing clusters. Distributed data management for cloud applications emphasizes scalability, fault-tolerance, and availability, sometimes at the expense of consistency or ease of development. We illustrate this approach with three popular solutions: Bigtable, PNUTS, and MapReduce.

Bigtable is a database storage system initially proposed by Google for a shared-nothing cluster [17]. It uses the Google File System (GFS) [18] for storing structured data in distributed files, with fault-tolerance and availability. It also uses a form of dynamic data partitioning for scalability. There are also...
open source implementations of Bigtable, such as Hadoop Hbase, as part of the Hadoop project of the Apache foundation (http://hadoop.apache.org), which runs on top of Hadoop Distributed File System, an open source implementation of GFS. Bigtable supports a simple data model that resembles the relational model, with multi-valued, timestamped attributes. It provides a basic application programming interface (API) for defining and manipulating tables, within a programming language such as C++, and 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. 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.

PNUTS is a parallel and distributed database system for cloud applications at Yahoo! [19]. It is designed for serving web applications, which typically do not need complex queries, but require good response time, scalability, and high availability and can tolerate relaxed consistency guarantees for replicated data. PNUTS supports the relational data model, with arbitrary structures allowed within attributes of Blob type. Schemas are flexible as new attributes can be added at any time even though the table is being queried or updated, and records need not have values for all attributes. PNUTS provides a simple query language with selection and projection on a single relation. Updates and deletes must specify the primary key. PNUTS provides a replica consistency model that is between strong consistency and eventual consistency, with several API operations with different guarantees. Database tables are horizontally partitioned into tablets, through either range partitioning or hashing, which are distributed across many servers in a cluster (at a site).

Both Bigtable and PNUTS provide some variation of the relational model, a simple API or language for manipulating data, and relaxed consistency guarantees. They also rely on fragmentation (partitioning) and replication for fault-tolerance. Thus, they capitalize on the well-known principles of distributed data management.

MapReduce [20] is a good example of generic parallel data processing framework, on top of a distributed file system (GFS). It supports a simple data model (sets of (key, value) pairs), which allows user-defined functions (map and reduce). MapReduce was initially developed by Google as a proprietary product to process large amounts of unstructured or semistructured data, such as web documents and logs of web page requests, on large shared-nothing clusters of commodity nodes and produce various kinds of data such as inverted indices or uniform resource locators (URLs) access frequencies.

MapReduce enables programmers to express in a simple, functional style their computations on large data sets and hides the details of parallel data processing, load balancing, and fault-tolerance. The programming model includes only two operations, map and reduce, which we can find in many functional programming languages such as Lisp and ML. The Map operation is applied to each record in the input data set to compute one or more intermediate (key, value) pairs. The Reduce operation 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.

Different implementations of MapReduce are now available such as Amazon MapReduce (as a cloud service) or Hadoop MapReduce (as open source software). There is also much research going on improving the performance of the MapReduce framework, which performs full scans of data sets. For instance, Hadoop++ [21] introduces noninvasive, DBMS-independent indexing and join techniques to boost the performance of Hadoop MapReduce.

13.4.2 Scientific Data Management

Scientific data management has become a major challenge for the database and data management research community [22]. Modern science such as agronomy, bio-informatics, physics, and
environmental science must deal with overwhelming amounts of experimental data produced through empirical observation and simulation. Such data must be processed (cleaned, transformed, and analyzed) in all kinds of ways in order to draw new conclusions, prove scientific theories, and produce knowledge. However, constant progress in scientific observational instruments (e.g., satellites, sensors, and Large Hadron Collider) and simulation tools (that foster in silico experimentation, as opposed to traditional in situ or in vivo experimentation) creates a huge data overload. For example, climate modeling data are growing so fast that they will lead to collections of hundreds of exabytes expected by 2020.

Scientific data are also very complex, in particular, because of heterogeneous methods used for producing data, the uncertainty of captured data, the inherently multi-scale nature (spatial scale and temporal scale) of many sciences, and the growing use of imaging (e.g., satellite images), resulting in data with hundreds of attributes, dimensions, or descriptors. Processing and analyzing such massive sets of complex scientific data are, therefore, a major challenge since solutions must combine new data management techniques with large-scale parallelism in cluster, grid, or cloud environments [23].

Furthermore, modern science research is a highly collaborative process, involving scientists from different disciplines (e.g., biologists, soil scientists, and geologists working on an environmental project), in some cases from different organizations distributed in different countries. Since each discipline or organization tends to produce and manage its own data, in specific formats, with its own processes, integrating distributed data and processes gets difficult as the amounts of heterogeneous data grow.

Another major difficulty that is inherent with science is that knowledge and understanding may keep evolving, making it sometimes very hard to model data. Thus, we need to better understand the fundamental aspects of the scientific data management problem, in relationship with the main users, that is, scientists.

Despite their variety, we can identify common features of scientific data [24]: massive scale; manipulated through complex, distributed workflows; typically complex, for example, multidimensional or graph-based; with uncertainty in the data values, for example, to reflect data capture or observation; important metadata about experiments and their provenance; heavy floating-point computation; and mostly append-only (with rare updates). For reasons similar to those discussed for cloud data, relational DBMS are not efficient for dealing with most scientific data and NoSQL can be useful for scientific data.

The goal of scientific data management is to make scientific data easier to access, reproduce, and share by scientists of different disciplines and institutions. In recent international interdisciplinary workshops and conferences (e.g., http://www-conf.slac.stanford.edu/xldb), the following key requirements for scientific data management (that cannot be supported by current technology) have been identified [25]:

- Built-in support for managing and processing uncertain data (e.g., inaccurate data generated by faulty sensors or by imprecise observations) in distributed environments
- Rich representation of scientific data with an extensible data model that features multidimensional arrays, graph structures, sequences, etc.
- Distributed and parallel workflow execution involving large numbers of distributed processes and large amounts of heterogeneous data, with support of data provenance (lineage) to understand result data
- Scalability to hundreds of petabytes and thousands of nodes in high-performance computing environments (e.g., very large clusters), with high degrees of tolerance to failures
- Efficient data and metadata management, in particular, with semantics (ontologies), in order to help integrating data coming from different sources, with different formats and semantics
- Open source software in order to foster a community of contributors (from many different laboratories and institutes) and to insure data independence from proprietary systems

Addressing the aforementioned challenges and requirements is now on the agenda of a very active research community composed of scientists from different disciplines and data management researchers. For instance, the SciDB organization (http://www.scidb.org) is building an open source database system for scientific data analytics. SciDB supports an array data model, which generalizes the relational
model, with array operators. User-defined functions also allow for more specific data processing. SciDB will be certainly effective for similar applications for which the data is well understood (with well-defined models). However, to avoid that the one-size-fits-all argument applies to SciDB as well, the key question is: How generic should scientific data management be, without hampering application-specific optimizations? For instance, to perform scientific data analysis efficiently, scientists typically resort to dedicated indexes, compression techniques, and specific algorithms. Thus, generic techniques, inspired from the database (DB) research community should be able to cope with these specific techniques.

Genericity in data management encompasses two dimensions: data model, which provides data structures (captured by the data model), and data processing (inferred by the query language). Relational DBMS have initially provided genericity through the relational data model (that subsumes earlier data models) and a high-level query language (SQL). However, successive object extensions to include new data structures such as lists and arrays and support user-defined functions in a programming language have resulted in a yet generic, but more complex data model and language for the developers. Therefore, emerging NoSQL solutions tend to rely on a more specific data model (e.g., to deal with graphs, arrays, or sequences) with a simple set of operators easy to use from a programming language with a simple API.

In emerging solutions, it is interesting to witness the development of algebras to raise the level of abstraction for the programmer and provide automatic optimization. For example, Pig Latin [26] is an alternative data management solution to MapReduce with an algebraic query language. Another example is the approach proposed in [27] to deal with the efficient processing of scientific workflows that are computational and data-intensive, that is, manipulating huge amounts of data through specific programs and files and requiring execution in large-scale parallel computers. The authors propose an algebraic approach (inspired by relational algebra) and a parallel execution model that enable automatic optimization of scientific workflows. With the algebra, data are uniformly represented by relations and workflow activities are mapped to operators that have data-aware semantics. The execution model is based on the concept of activity activation, inspired from data activations proposed for parallel DBMS [28], which enables transparent distribution and parallelization of activities.

13.4.3 Stream Applications and Stream Data Management

The database systems that we have discussed 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. Queries over these systems are executed when posed, and the answer reflects the current state of the database. In these systems, typically, the data are persistent and queries are transient.

However, the past few years have witnessed an emergence of applications that do not fit this data model and querying paradigm. These applications include network traffic analysis, financial tickers, online auctions, and applications that analyze transaction logs (such as web usage logs and telephone call records). In these applications, data are generated in real time, taking the form of an unbounded sequence (stream) of values. These are referred to as the data stream applications, and the systems that handle these types of data are known as data stream management systems (DSMS) (or stream data management systems) [29,30].

A data stream is commonly modeled as an append-only sequence of timestamped items that arrive in some order. Stream items may contain explicit source-assigned timestamps or implicit timestamps assigned by the DSMS upon arrival. 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 pre-aggregated values) may be produced to summarize the length of a connection between two IP addresses and the number of bytes transmitted.
A fundamental assumption of the data stream model is that new data are generated continually and in 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, DSMSs face the following novel requirements:

1. Much of the computation performed by a DSMS 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 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 preceding text, DSMS queries are persistent (also referred to as continuous, long-running, or standing queries) in that they are issued once, but remain active in the system for a possibly long period of time. This means that a stream of updated results must be produced as time goes on. In contrast, a DBMS deals with one-time queries (issued once and then “forgotten”), whose results are computed over the current state of the database.

3. 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.

4. A data stream is assumed to have unbounded, or at least unknown, length. From the system’s point of view, it is infeasible to store an entire stream in a DSMS. From the user’s point of view, recently arrived data are likely to be more accurate or useful.

5. New data models, query semantics, and query languages are needed for DSMSs in order to reflect the facts that streams are ordered and queries are persistent.

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

- **Selection**: All streaming applications require support for complex filtering.
- **Nested aggregation**: Complex aggregates, including nested aggregates (e.g., comparing a minimum with a running average) are needed to compute trends in the data.
- **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).
- **Frequent item queries**: These are also known as top-k or threshold queries, depending on the cutoff condition.
- **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 multi-stream joins and joins of streams with static metadata.
- **Windowed queries**: All of the aforementioned query types may be constrained to return results inside a window (e.g., the last 24 h or the last one hundred packets).

Proposed data stream systems resemble the abstract architecture shown in Figure 13.2. An input monitor regulates the input rates, perhaps by dropping items if the system is unable to keep up. Data are typically stored in three partitions: temporary working storage (e.g., for window queries that will be discussed shortly), summary storage for stream synopses, 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 re-optimize 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.
Unbounded streams cannot be stored locally in a DSMS, and only a recent excerpt of a stream is usually of interest at any given time. In general, this may be accomplished using a time-decay model, and the most common way to achieve this is by means of window models where items within the window are given full consideration and items outside the window are ignored. There are a number of ways to classify windows, but the most common are the logical, or time-based windows, which are defined in terms of a time interval, and physical, or count-based windows, which are defined in terms of the number of tuples in the window.

Query languages developed for DSMSs fall into three classes: declarative, object-based, and procedural. Declarative languages have SQL-like syntax, but stream-specific semantics, as described earlier. Similarly, object-based languages resemble SQL in syntax, but employ DSMS-specific constructs and semantics, and may include support for streaming abstract data types and associated methods. Finally, procedural languages construct queries by defining data flow through various operators.

While the streaming languages discussed earlier may resemble standard SQL, their implementation, processing, and optimization present novel challenges. Some relational operators are blocking. For instance, prior to returning the next tuple, a nested loop join algorithm may potentially scan the entire inner relation and compare each tuple therein with the current outer tuple. Since data streams are unbounded, it is not possible to scan them, requiring nonblocking implementations. Some operators have non-blocking counterparts; in general, unblocking a query operator may be accomplished by re-implementing it in an incremental form, restricting it to operate over a window (more on this shortly), and exploiting stream constraints. However, there may be cases where an incremental version of an operator does not exist or is inefficient to evaluate, where even a sliding window is too large to fit in main memory, or where no suitable stream constraints are present. In these cases, compact stream summaries may be stored, and approximate queries may be posed over the summaries. This implies a trade-off between accuracy and the amount of memory used to store the summaries. An additional restriction is that the processing time per item should be kept small, especially if the inputs arrive at a fast rate.

Since windowed processing is crucial in DSMSs, we briefly discuss window operators. Sliding window operators process two types of events: arrivals of new tuples and expirations of old tuples. The actions taken upon arrival and expiration vary across operators. A new tuple may generate new results (e.g., join) or remove previously generated results (e.g., negation). Furthermore, an expired tuple may cause a removal of one or more tuples from the result (e.g., aggregation) or an addition of new tuples to the result (e.g., duplicate elimination and negation). Moreover, operators that must explicitly react to expired tuples (by producing new results or invalidating existing results) perform state purging eagerly (e.g., duplicate elimination, aggregation, and negation), whereas others may do so eagerly or lazily (e.g., join).
There is also the orthogonal problem of determining when tuples expire; sliding window operators must remove old tuples from their state buffers and possibly update their answers. Expiration from an individual time-based window is simple: A tuple expires if its timestamp falls out of the range of the window. That is, when a new tuple with timestamp $ts$ arrives, it receives another timestamp, call it $exp$, that denotes its expiration time as $ts$ plus the window length. In effect, every tuple in the window may be associated with a lifetime interval of length equal to the window size. Now, if this tuple joins with a tuple from another window, whose insertion and expiration timestamps are $ts'$ and $exp'$, respectively, then the expiration timestamp of the result tuple is set to $\min(exp, exp')$. That is, a composite result tuple expires if at least one of its constituent tuples expires from its windows. This means that various join results may have different lifetime lengths and furthermore, the lifetime of a join result may have a lifetime that is shorter than the window size. Moreover, as discussed earlier, the negation operator may force some result tuples to expire earlier than their $exp$ timestamps by generating negative tuples. Finally, if a stream is not bounded by a sliding window, then the expiration time of each tuple is infinity.

In a count-based window, the number of tuples remains constant over time. Therefore, expiration can be implemented by overwriting the oldest tuple with a newly arrived tuple. However, if an operator stores state corresponding to the output of a count-based window join, then the number of tuples in the state may change, depending upon the join attribute values of new tuples. In this case, expirations must be signaled explicitly using negative tuples.

Window sizes can be set system-wide or by each application (or a combination of both). It is possible to use P2P techniques to dynamically increase the size of the sliding window [31].

In addition to the implementation of operators, there is the larger question of optimizing and processing queries, specifically continuous queries. A DSMS query optimizer performs tasks that are similar to a DBMS, but it must use an appropriate cost model and rewrite rules. Additionally, DSMS query optimization involves adaptivity, load shedding, and resource sharing among similar queries running in parallel, as summarized in the subsequent sections.

Traditional DBMSs use selectivity information and available indices to choose efficient query plans (e.g., those which require the fewest disk accesses). However, this cost metric does not apply to (possibly approximate) persistent queries, where processing cost per unit time is more appropriate. Alternatively, if the stream arrival rates and output rates of query operators are known, it may be possible to optimize for the highest output rate or to find a plan that takes the least time to output a given number of tuples. Finally, quality-of-service metrics such as response time may also be used in DSMS query optimization.

A DSMS query optimizer rewrites queries as usual, but with particular attention to the commutativity rules that arise in data stream systems. For example, selections and time-based sliding windows commute, but not selections and count-based windows. Furthermore, the notion of adaptivity is important in query rewriting; operators may need to be re-ordered on-the-fly in response to changes in system conditions. In particular, the cost of a query plan may change for three reasons: change in the processing time of an operator, change in the selectivity of a predicate, and change in the arrival rate of a stream. Initial efforts on adaptive query plans include mid-query re-optimization and query scrambling, where the objective is to preempt any operators that become blocked and schedule other operators instead. To further increase adaptivity, instead of maintaining a rigid tree-structured query plan, the Eddy approach [32] performs scheduling of each tuple separately by routing it through the operators that make up the query plan. In effect, the query plan is dynamically re-ordered to match current system conditions. This is accomplished by tuple routing policies that attempt to discover which operators are fast and selective, and those operators are scheduled first. A recent extension adds queue length as the third factor for tuple routing strategies in the presence of multiple distributed Eddies [33]. There is, however, an important trade-off between the resulting adaptivity and the overhead required to route each tuple separately. Adaptivity involves online reordering of a query plan and may therefore require that the internal state stored by some operators be migrated over to the new query plan consisting of a different arrangement of operators.
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 sub-plans 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 re-execution of persistent queries, increasing the re-execution interval may be thought of as a form of load shedding.

### 13.4.4 Web Data Management

The World Wide Web (“WWW” or “web” for short) has become a major repository of data and documents. 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 [34]. 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 nodes, and the links between them are represented as directed edges. Studying the web graph is obviously of interest to theoretical computer scientists, because it exhibits a number of interesting characteristics, but it is also important for studying data management issues since the graph structure is exploited in web search [35–37], categorization and classification of web content, and other web-related tasks. The important characteristics of the web graph are the following [38]:

- 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.
- It is sparse. A graph is considered sparse if its average degree is less than the number of vertices. This means that the each node of the graph has a limited number of neighbors, even if the nodes are in general connected. The sparseness of the web graph implies an interesting graph structure that we discuss shortly.
- 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.
- It is a “small-world network.” 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.

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

The management of the very large, dynamic, and volatile web graph is an important issue. Two alternatives have been proposed for this purpose. The first is compressing the web graph, and the second is to develop special storage structures that allow efficient storage and querying (e.g., S-nodes [39]).

Web search is arguably the most common method for accessing the web. It 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.

A search engine consists of a number of components. One component is the crawler, which 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 URLs that identify these pages. The crawler retrieves and parses the page corresponding to that URL, extracts any 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.

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.

The ranking module is responsible for sorting the 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.

There have also been attempts to perform declarative querying of web data and the efficient execution of these queries. Although these technologies are well developed for relational DBMSs, 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 earlier, it is hard to argue that there is a schema for web data similar to databases.* 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 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

---
* We are focusing on the "open" web here; deep web data may have a schema, but it is usually not accessible to users.
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, a standardized language
for XML has emerged (XQuery), and as XML becomes more prevalent on the web, this language is likely
to become dominant and more widely used.

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.

There have also been special purpose web query language developments. These 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 con-
tent structure of documents (as in semistructured data approaches) as well as the external links. They
combine the content-based queries (e.g., keyword expressions) and structure-based queries (e.g., path
expressions).

13.4.5 P2P Data Management

In contrast to C/S DBMS, P2P systems [40] adopt a completely decentralized approach to distributed
data sharing. By distributing data storage and processing across autonomous peers in the network, they
can scale without the need for powerful servers. Popular examples of P2P systems such as BitTorrent and
eMule have millions of users sharing petabytes of data over the Internet. Although very useful, these
systems are quite simple (e.g., file sharing), support-limited functions (e.g., keyword search) and use
simple techniques (e.g., resource location by flooding), which have performance problems. To deal with
the dynamic behavior of peers that can join and leave the system at any time, they rely on the fact that
popular data get massively duplicated. Furthermore, they are single-application systems and focus on
performing one task, and it is not straightforward to extend them for other applications/functions [41].

To provide proper database functionality (schema, queries, replication, consistency, availability, etc.)
over P2P infrastructures, the following data management issues must be addressed:

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

Data location depends on the underlying infrastructure. P2P networks can be of two general types:
pure and hybrid. Pure P2P networks are those where there is no differentiation between any of the network
nodes—they are all equal. In hybrid P2P networks, on the other hand, some nodes are given special
tasks to perform. Hybrid networks are commonly known as super-peer systems, since some of the
peers are responsible for “controlling” a set of other peers in their domain. The pure P2P networks can
be further divided into structured and unstructured networks. Structured networks tightly control the
topology and message routing, for instance using a distributed hash table (DHT) or a tree structure,
whereas in unstructured networks, each node can directly communicate with its neighbors and can join
the network by attaching themselves to any node.

Structured P2P networks have emerged to address the scalability issues faced by unstructured P2P
networks, which rely on flooding to locate data. They achieve this goal by tightly controlling the network
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.

Data integration raises issues that are more difficult than for designing database integration systems. Due to specific characteristics of P2P systems, for example, 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 on one peer’s schema can be reformulated in a query on another peer’s schema. The approaches, which are used by P2P data management systems for defining and creating the mappings between peers’ schemas, can be classified as follows: pairwise schema mapping, common agreement mapping, and schema mapping using IR techniques. With pairwise schema mapping [42], each user defines the mapping between the local schema and the schema of any other peer that contains data that is of interest. Relying on the transitivity of the defined mappings, the system tries to extract mappings between schemas, which have no defined mapping. In common agreement mapping [43], the peers that have a common interest agree on a common schema description (CSD) for data sharing. Given a CSD, a peer schema can be specified using views. This is similar to the local-as-view approach in data integration systems, except that queries at a peer are expressed in terms of the local views, not the CSD. Another difference is that the CSD is not a global schema, that is, it is common to a limited set of peers with common interest. The last approach extracts the schema mappings at query execution time using IR techniques by exploring the schema descriptions provided by users. PeerDB [41] follows this approach for query processing in unstructured P2P networks.

P2P networks provide basic techniques for routing queries to relevant peers and this is sufficient for supporting simple, exact-match queries. For instance, a DHT provides a basic mechanism for efficient data lookup 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.

With a top-k query, the user can specify a number \( k \) of the 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. An efficient algorithm for top-k query processing in centralized and distributed systems is the threshold algorithm (TA) [44]. TA assumes a general model based on lists of data items sorted by their local scores and is applicable for queries where the scoring function is monotonic, that is, any increase in the value of the input does not decrease the value of the output. The stopping mechanism of TA uses a threshold which is computed using the last local scores seen under sorted access in the lists. Many of the popular aggregation functions such as min, max, and average are monotonic. TA has been the basis for several TA-style algorithms in P2P systems. There are many database instances over which TA keeps scanning the lists although it has seen all top-k answers. Thus, it is possible to stop much sooner. Based on this observation, best position algorithms that execute top-k queries much more efficiently than TA have been proposed [45].

Structured P2P systems, in particular, DHTs are very efficient at supporting exact-match queries (i.e., queries of the form “\( A = value \)”) but have difficulties with range queries. The main reason is that hashing tends to destroy the ordering of data which is useful to find ranges quickly. There are two main approaches for supporting range queries in structured P2P systems: extend a DHT with proximity or order-preserving properties [46], or maintain the key ordering with a tree-based structure [47].

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 [48] is a variation of the traditional parallel hash join algorithm. The joined relations and the result relations have a namespace that indicates the nodes, which store horizontal fragments of the relation. Then it makes use of the DHT put method for distributing tuples onto a set of peers based on their join attribute so that tuples with same join attribute values are stored at the same peers. Then, joins can be performed locally at each peer, for example, using a hash join algorithm.
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. Most of the work on replica consistency has been done only in the context of DHTs.

To improve data availability, most DHTs rely on data replication by storing data items 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, for example, Tapestry [49].

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. For some applications (e.g., agenda management), 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. The problem can be partially addressed by using data versioning, where each replica has a version number that is increased after each update. To return a current replica, all replicas need to be retrieved in order to select the latest version. However, because of concurrent updates, it may happen that two different replicas have the same version number, thus making it impossible to decide which one is the current replica. A more complete solution considers both data availability and data currency in DHTs, using distributed timestamping [50].

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 are not appropriate. The major solution is lazy distributed replication as used in OceanStore [51] and P-Grid [52]. APPA provides a general lazy distributed replication solution, which assures eventual consistency of replicas [53].

13.5 Summary

The impact of distributed and parallel data management technology on practice has been very high as all commercial DBMSs today have distributed and parallel versions. Parallel database systems are used to support very large databases (e.g., hundreds of terabytes or petabytes). Examples of applications that deal with very large databases are e-commerce, data warehousing, and data mining. Very large databases are typically accessed through high numbers of concurrent transactions (e.g., performing online orders on an electronic store) or complex queries (e.g., decision-support queries). The first kind of access is representative of OLTP applications while the second is representative of OLAP applications. Although both OLTP and OLAP can be supported by the same parallel database system (on the same multiprocessor), they are typically separated and supported by different systems to avoid any interference and ease database operation.

To address the requirements of new data-intensive applications (e.g., scientific applications or cloud), emerging solutions tend to rely on a more specific data model (e.g., Bigtable, which is a variant of the nested relational model) with a simple set of operators easy to use from a programming language. For instance, to address the requirements of social network applications, new solutions rely on a graph data model and graph-based operators. To address the requirements of scientific applications, SciDB [54] supports an array data model, which generalizes the relational model, with array operators. User-defined functions also allow for more specific data processing. In emerging solutions, it is interesting to witness the development of algebras, with specific operators, to raise the level of abstraction in a way that enables optimization, while making it easy to manipulate data from a programming language with a simple API.

Distributed and parallel database technology can now be used in two complementary large-scale distributed contexts: P2P and cloud. A P2P solution is appropriate to support the collaborative nature
of many applications, for example, scientific applications, as it provides scalability, dynamicity, autonomy, and decentralized control. Peers can be the participants or organizations involved in collaboration and may share data and applications while keeping full control over their (local) data sources. But for very-large-scale data analysis or to execute very large workflow activities, cloud computing is the right approach as it can provide virtually infinite computing, storage, and networking resources.

References