# Chapter 1
# Introduction

The current computing environment is largely distributed—computers are connected to Internet to form a worldwide distributed system. Organizations have geographically distributed and interconnected data centers, each with hundreds or thousands of computers connected with high-speed networks, forming mixture of distributed and parallel systems (Fig. 1.1). Within this environment, the amount of data that is captured has increased dramatically. Not all of this data is stored in database systems (in fact a small portion is) but there is a desire to provide some sort of data management capability on these widely distributed data. This is the scope of distributed and parallel database systems, which have moved from a small part of the worldwide computing environment a few decades ago to mainstream. In this chapter, we provide an overview of this technology, before we examine the details in subsequent chapters.

## 1.1 What Is a Distributed Database System?

We define a *distributed database* as *a collection of multiple, logically interrelated databases located at the nodes of a distributed system*. A *distributed database management system* (distributed DBMS) is then defined as *the software system that permits the management of the distributed database and makes the distribution transparent to the users*. Sometimes "distributed database system" (distributed DBMS) is used to refer jointly to the distributed database and the distributed DBMS. The two important characteristics are that data is logically interrelated and that it resides on a distributed system.

The original version of this chapter was revised. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-26253-2_13

M. T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*,
https://doi.org/10.1007/978-3-030-26253-2_1

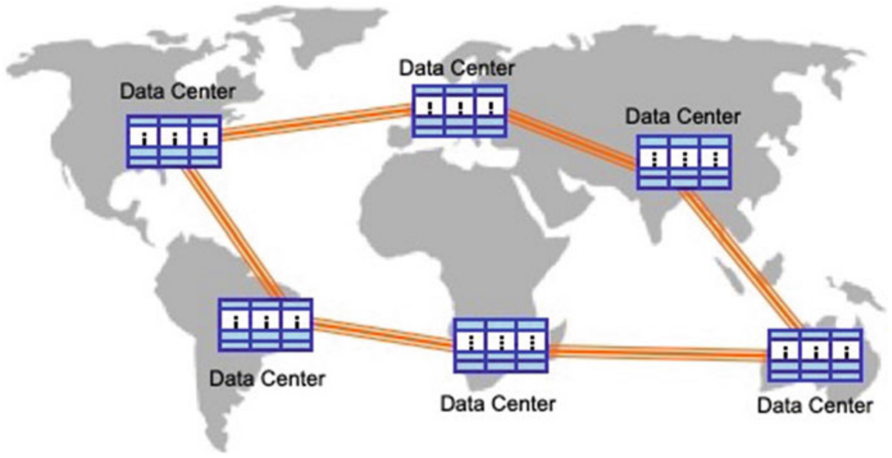

**Fig. 1.1** Geographically distributed data centers

The existence of a distributed system is an important characteristic. In this context, we define a *distributed computing system* as a number of interconnected autonomous processing elements (PEs). The capabilities of these processing elements may differ, they may be heterogeneous, and the interconnections might be different, but the important aspect is that PEs do not have access to each other's state, which they can only learn by exchanging messages that incur a communication cost. Therefore, when data is distributed, its management and access in a logically integrated manner requires special care from the distributed DBMS software.

A distributed DBMS is not a "collection of files" that can be individually stored at each PE of a distributed system (usually called "site" of a distributed DBMS); data in a distributed DBMS is interrelated. We will not try to be very specific with what we mean by interrelated, because the requirements differ depending on the type of data. For example, in the case of relational data, different relations or their partitions might be stored at different sites (more on this in Chap. 2), requiring join or union operations to answer queries that are typically expressed in SQL. One can usually define a *schema* of this distributed data. At the other extreme, data in NoSQL systems (discussed further in Chap. 11) may have a much looser definition of interrelatedness; for example, it may be vertices of a graph that might be stored at different sites.

The upshot of this discussion is that a distributed DBMS is *logically integrated* but *physically distributed*. What this means is that a distributed DBMS gives the users the view of a unified database, while the underlying data is physically distributed.

As noted above, we typically consider two types of distributed DBMSs: geographically distributed (commonly referred to as *geo-distributed*) and single location (or single site) . In the former, the sites are interconnected by wide area networks that are characterized by long message latencies and higher error rates. The latter consist of systems where the PEs are located in close proximity allowing much faster exchanges leading to shorter (even negligible with new technologies) message latencies and very low error rates. Single location distributed DBMSs are typically characterized by computer clusters in one data center, and are commonly known as parallel DBMSs (and the PEs are referred to as "nodes" to distinguish from "sites"). As noted above, it is now quite common to find distributed DBMSs that have multiple single site clusters interconnected by wide area networks, leading to hybrid, multisite systems. For most of this book, we will focus on the problems of data management among the sites of a geo-distributed DBMS; we will focus on the problems of single site systems in Chaps. 8, 10, and 11 where we discuss parallel DBMSs, big data systems, and NoSQL/NewSQL systems.

## 1.2 History of Distributed DBMS

Before the advent of database systems in the 1960s, the prevalent mode of computation was one where each application defined and maintained its own data (Fig. 1.2). In this mode, each application defined the data that it used, its structure and access methods, and managed the file in the storage system. The end result was significant uncontrolled redundancy in the data, and high overhead for the programmers to manage this data within their applications.

Database systems allow data to be defined and administered centrally (Fig. 1.3). This new orientation results in *data independence*, whereby the application programs are immune to changes in the logical or physical organization of the data and vice versa. Consequently, programmers are freed from the task of managing and maintaining the data that they need, and the redundancy of the data can be eliminated (or reduced).

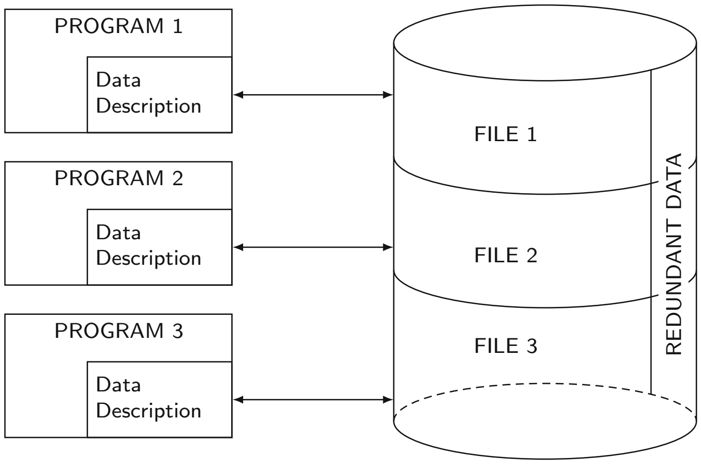

**Fig. 1.2** Traditional file processing

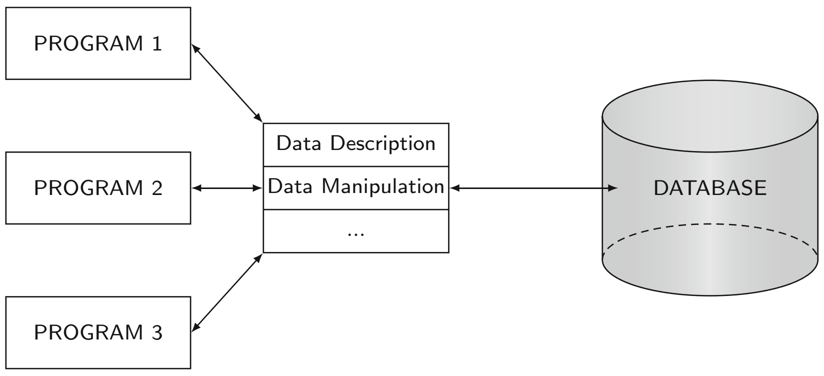

**Fig. 1.3** Database processing

One of the original motivations behind the use of database systems was the desire to integrate the operational data of an enterprise and to provide integrated, thus controlled access to that data. We carefully use the term "integrated" rather than "centralized" because, as discussed earlier, the data can physically be located on different machines that might be geographically distributed. This is what the distributed database technology provides. As noted earlier, this physical distribution can be concentrated at one geographic location or it can be at multiple locations. Therefore, each of the locations in Fig. 1.5 might be a data center that is connected to other data centers over a communication network. These are the types of distributed environments that are now common and that we study in this book.

Over the years, distributed database system architectures have undergone significant changes. The original distributed database systems such as Distributed INGRES and SDD-1 were designed as geographically distributed systems with very slow network connections; consequently they tried to optimize operations to reduce network communication. They were early *peer-to-peer systems* (P2P) in the sense that each site had similar functionality with respect to data management. With the development of personal computers and workstations, the prevailing distribution model shifted to *client/server* where data operations were moved to a back-end server, while user applications ran on the front-end workstations. These systems became dominant in particular for distribution at one particular location where the network speeds would be higher, enabling frequent communication between the clients and the server(s). There was a reemergence of P2P systems in the 2000s, where there is no distinction of client machines versus servers. These modern P2P systems have important differences from the earlier systems that we discuss later in this chapter. All of these architectures can still be found today and we discuss them in subsequent chapters.

The emergence of the World Wide Web (usually called the web) as a major collaboration and sharing platform had a profound impact on distributed data management research. Significantly more data was opened up for access, but this was not the well-structured, well-defined data DBMSs typically handle; instead, it is unstructured or semistructured (i.e., it has some structure but not at the level of a database schema), with uncertain provenance (so it might be "dirty" or unreliable), and conflicting. Furthermore, a lot of the data is stored in systems that are not easily accessible (what is called the *dark web*). Consequently, distributed data management efforts focus on accessing this data in meaningful ways.

This development added particular impetus to one thread of research that existed since the beginning of distributed database efforts, namely *database integration*. Originally, these efforts focused on finding ways to access data in separate databases (thus the terms *federated database* and *multidatabase*), but with the emergence of web data, these efforts shifted to virtual integration of different data types (and the term *data integration* became more popular). The term that is in vogue right now is *data lake* which implies that all of the data is captured in a logically single store, from which relevant data is extracted for each application. We discuss the former in Chap. 7 and the latter in Chaps. 10 and 12.

A significant development in the last ten years has been the emergence of cloud computing. Cloud computing refers to a computing model where a number of service providers make available shared and geo-distributed computing resources such that users can rent some of these resources based on their needs. Clients can rent the basic computing infrastructure on which they could develop their own software, but then decide on the operating system they wish to use and create virtual machines (VMs) to create the environment in which they wish to work—the so-called *Infrastructure-as-a-Service* (IaaS) approach. A more sophisticated cloud environment involves renting, in addition to basic infrastructure, the full computing platform leading to *Platform-as-a-Service* (PaaS) on which clients can develop their own software. The most sophisticated version is where service providers make available specific software that the clients can then rent; this is called *Software-as-a-Service* (SaaS). There has been a trend in providing distributed database management services on the cloud as part of SaaS offering, and this has been one of the more recent developments.

In addition to the specific chapters where we discuss these architectures in depth, we provide an overview of all of them in Sect. 1.6.1.2.

## 1.3 Data Delivery Alternatives

In distributed databases, data delivery occurs between sites—either from server sites to client sites in answer to queries or between multiple servers. We characterize the data delivery alternatives along three orthogonal dimensions: *delivery modes*, *frequency*, and *communication methods*. The combinations of alternatives along each of these dimensions provide a rich design space.

The alternative delivery modes are pull-only, push-only, and hybrid. In the *pull-only* mode of data delivery, the transfer of data is initiated by a pull (i.e., request) from one site to a data provider—this may be a client requesting data from a server or a server requesting data from another server. In the following we use the terms "receiver" and "provider" to refer to the machine that received the data and the machine that sends the data, respectively. When the request is received by the provider, the data is located and transferred. The main characteristic of pull-based delivery is that receivers become aware of new data items or updates at the provider only when they explicitly poll. Also, in pull-based mode, providers must be interrupted continuously to deal with requests. Furthermore, the data that receivers can obtain from a provider is limited to when and what clients know to ask for. Conventional DBMSs offer primarily pull-based data delivery.

In the *push-only* mode of data delivery, the transfer of data from providers is initiated by a push without a specific request. The main difficulty of the push-based approach is in deciding which data would be of common interest, and when to send it to potentially interested receivers—alternatives are periodic, irregular, or conditional. Thus, the usefulness of push depends heavily upon the accuracy of a provider to predict the needs of receivers. In push-based mode, providers disseminate information to either an unbounded set of receivers (random broadcast) who can listen to a medium or selective set of receivers (multicast), who belong to some categories of recipients.

The *hybrid* mode of data delivery combines the pull and push mechanisms. The persistent query approach (see Sect. 10.3) presents one possible way of combining the pull and push modes, namely: the transfer of data from providers to receivers is first initiated by a pull (by posing the query), and the subsequent transfer of updated data is initiated by a push by the provider.

There are three typical frequency measurements that can be used to classify the regularity of data delivery. They are periodic, conditional, and ad hoc (or irregular).

In *periodic delivery*, data is sent from the providers at regular intervals. The intervals can be defined by system default or by receivers in their profiles. Both pull and push can be performed in periodic fashion. Periodic delivery is carried out on a regular and prespecified repeating schedule. A request for a company's stock price every week is an example of a periodic pull. An example of periodic push is when an application can send out stock price listing on a regular basis, say every morning. Periodic push is particularly useful for situations in which receivers might not be available at all times, or might be unable to react to what has been sent, such as in the mobile setting where clients can become disconnected.

In *conditional delivery*, data is sent by providers whenever certain conditions specified by receivers in their profiles are satisfied. Such conditions can be as simple as a given time span or as complicated as event-condition-action rules. Conditional delivery is mostly used in the hybrid or push-only delivery systems. Using conditional push, data is sent out according to a prespecified condition, rather than any particular repeating schedule. An application that sends out stock prices only when they change is an example of conditional push. An application that sends out a balance statement only when the total balance is 5% below the predefined balance threshold is an example of hybrid conditional push. Conditional push assumes that changes are critical to the receivers who are always listening and need to respond to what is being sent. Hybrid conditional push further assumes that missing some update information is not crucial to the receivers.

*Ad hoc delivery* is irregular and is performed mostly in a pure pull-based system. Data is pulled from providers in an ad hoc fashion in response to requests. In contrast, periodic pull arises when a requestor uses polling to obtain data from providers based on a regular period (schedule).

The third component of the design space of information delivery alternatives is the communication method. These methods determine the various ways in which providers and receivers communicate for delivering information to clients. The alternatives are unicast and one-to-many. In *unicast*, the communication from a provider to a receiver is one-to-one: the provider sends data to one receiver using a particular delivery mode with some frequency. In *one-to-many*, as the name implies, the provider sends data to a number of receivers. Note that we are not referring here to a specific protocol; one-to-many communication may use a multicast or broadcast protocol.

We should note that this characterization is subject to considerable debate. It is not clear that every point in the design space is meaningful. Furthermore, specification of alternatives such as conditional **and** periodic (which may make sense) is difficult. However, it serves as a first-order characterization of the complexity of emerging distributed data management systems. For the most part, in this book, we are concerned with pull-only, ad hoc data delivery systems, and discuss push-based and hybrid modes under streaming systems in Sect. 10.3.

## 1.4 Promises of Distributed DBMSs

Many advantages of distributed DBMSs can be cited; these can be distilled to four fundamentals that may also be viewed as promises of distributed DBMS technology: transparent management of distributed and replicated data, reliable access to data through distributed transactions, improved performance, and easier system expansion. In this section, we discuss these promises and, in the process, introduce many of the concepts that we will study in subsequent chapters.

### *1.4.1 Transparent Management of Distributed and Replicated Data*

Transparency refers to separation of the higher-level semantics of a system from lower-level implementation issues. In other words, a transparent system "hides" the implementation details from users. The advantage of a fully transparent DBMS is the high level of support that it provides for the development of complex applications. Transparency in distributed DBMS can be viewed as an extension of the data independence concept in centralized DBMS (more on this below).

$\texttt{EMP}(\underline{\texttt{ENO}}, \texttt{ENAME}, \texttt{TITLE})$

$\texttt{PROJ}(\underline{\texttt{PNO}}, \texttt{PNAME}, \texttt{BUDGET}, \texttt{LOC})$

$\texttt{ASG}(\underline{\texttt{ENO}, \texttt{PNO}}, \texttt{RESP}, \texttt{DUR})$

$\texttt{PAY}(\underline{\texttt{TITLE}}, \texttt{SAL})$

**Fig. 1.4** Example engineering database

Let us start our discussion with an example. Consider an engineering firm that has offices in Boston, Waterloo, Paris, and San Francisco. They run projects at each of these sites and would like to maintain a database of their employees, the projects, and other related data. Assuming that the database is relational, we can store this information in a number of relations (Fig. 1.4): `EMP` stores employee information with employee number, name, and title[1]; `PROJ` holds project information where `LOC` records where the project is located. The salary information is stored in `PAY` (assuming everyone with the same title gets the same salary) and the assignment of people to projects is recorded in `ASG` where `DUR` indicates the duration of the assignment and the person's responsibility on that project is maintained in `RESP`. If all of this data were stored in a centralized DBMS, and we wanted to find out the names and employees who worked on a project for more than 12 months, we would specify this using the following SQL query:

```
SELECT ENAME, AMT
FROM EMP NATURAL JOIN ASG, EMP NATURAL JOIN PAY
WHERE ASG.DUR > 12
```

However, given the distributed nature of this firm's business, it is preferable, under these circumstances, to localize data such that data about the employees in Waterloo office is stored in Waterloo, those in the Boston office is stored in Boston, and so forth. The same applies to the project and salary information. Thus, what we are engaged in is a process where we partition each of the relations and store each partition at a different site. This is known as *data partitioning* or *data fragmentation* and we discuss it further below and in detail in Chap. 2.

Furthermore, it may be preferable to duplicate some of this data at other sites for performance and reliability reasons. The result is a distributed database which is fragmented and replicated (Fig. 1.5). Fully transparent access means that the users can still pose the query as specified above, without paying any attention to the fragmentation, location, or replication of data, and let the system worry about resolving these issues. For a system to adequately deal with this type of query over a distributed, fragmented, and replicated database, it needs to be able to deal with a number of different types of transparencies as discussed below.

**Data Independence.** This notion carries over from centralized DBMSs and refers to the immunity of user applications to changes in the definition and organization of data, and vice versa.

[1]Primary key attributes are underlined.

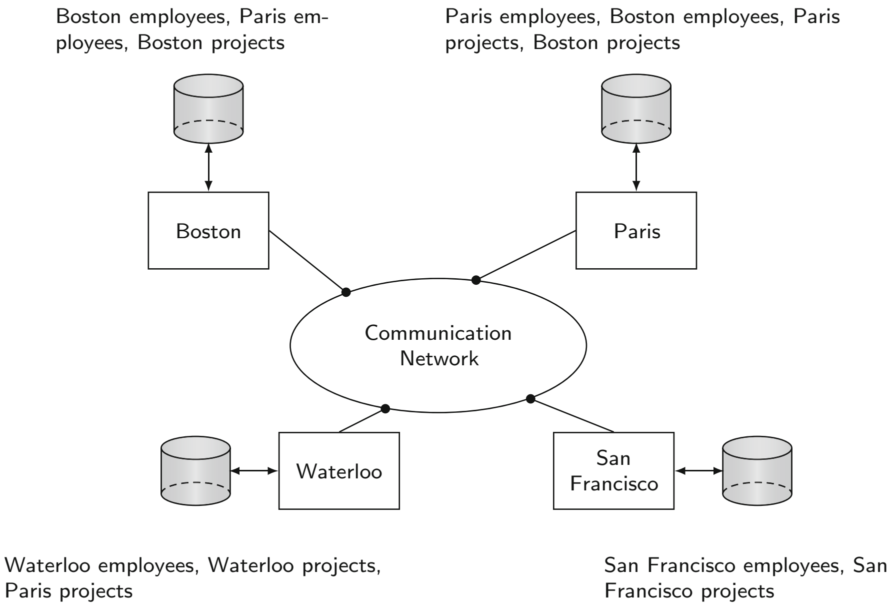
Boston employees, Paris employees, Boston projects
Paris employees, Boston employees, Paris projects, Boston projects
Boston
Paris
Communication Network
Waterloo
San Francisco
Waterloo employees, Waterloo projects, Paris projects
San Francisco employees, San Francisco projects

**Fig. 1.5** Distributed database

Two types of data independence are usually cited: logical data independence and physical data independence. *Logical data independence* refers to the immunity of user applications to changes in the logical structure (i.e., schema) of the database. *Physical data independence*, on the other hand, deals with hiding the details of the storage structure from user applications. When a user application is written, it should not be concerned with the details of physical data organization. Therefore, the user application should not need to be modified when data organization changes occur due to performance considerations.

**Network Transparency.** Preferably, users should be protected from the operational details of the communication network that connects the sites; possibly even hiding the existence of the network. Then there would be no difference between database applications that would run on a centralized database and those that would run on a distributed database. This type of transparency is referred to as *network transparency* or *distribution transparency*.

Sometimes two types of distribution transparency are identified: location transparency and naming transparency. *Location transparency* refers to the fact that the command used to perform a task is independent of both the location of the data and the system on which an operation is carried out. *Naming transparency* means that a unique name is provided for each object in the database. In the absence of naming transparency, users are required to embed the location name (or an identifier) as part of the object name.

**Fragmentation Transparency.** As discussed above, it is commonly desirable to divide each database relation into smaller fragments and treat each fragment as a separate database object (i.e., another relation). This is commonly done for reasons of performance, availability, and reliability—a more in-depth discussion is in Chap. 2. It would be preferable for the users not to be aware of data fragmentation in specifying queries, and let the system deal with the problem of mapping a user query that is specified on full relations as specified in the schema to a set of queries executed on subrelations. In other words, the issue is one of finding a query processing strategy based on the fragments rather than the relations, even though the queries are specified on the latter.

**Replication Transparency.** For performance, reliability, and availability reasons, it is usually desirable to be able to distribute data in a replicated fashion across the machines on a network. Assuming that data is replicated, the transparency issue is whether the users should be aware of the existence of copies or whether the system should handle the management of copies and the user should act as if there is a single copy of the data (note that we are not referring to the placement of copies, only their existence). From a user's perspective it is preferable not to be involved with handling copies and having to specify the fact that a certain action can and/or should be taken on multiple copies. The issue of replicating data within a distributed database is introduced in Chap. 2 and discussed in detail in Chap. 6.

### 1.4.2 Reliability Through Distributed Transactions

Distributed DBMSs are intended to improve reliability since they have replicated components and thereby eliminate single points of failure. The failure of a single site, or the failure of a communication link which makes one or more sites unreachable, is not sufficient to bring down the entire system. In the case of a distributed database, this means that some of the data may be unreachable, but with proper care, users may be permitted to access other parts of the distributed database. The "proper care" comes mainly in the form of support for distributed transactions.

A DBMS that provides full transaction support guarantees that concurrent execution of user transactions will not violate database consistency, i.e., each user thinks their query is the only one executing on the database (called *concurrency transparency*) even in the face of system failures (called *failure transparency*) as long as each transaction is correct, i.e., obeys the integrity rules specified on the database.

Providing transaction support requires the implementation of distributed concurrency control and distributed reliability protocols—in particular, two-phase commit (2PC) and distributed recovery protocols—which are significantly more complicated than their centralized counterparts. These are discussed in Chap. 5. Supporting replicas requires the implementation of replica control protocols that enforce a specified semantics of accessing them. These are discussed in Chap. 6.

### 1.4.3 Improved Performance

The case for the improved performance of distributed DBMSs is typically made based on two points. First, a distributed DBMS fragments the database, enabling data to be stored in close proximity to its points of use (also called *data locality*). This has two potential advantages:

1. Since each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for centralized databases.
2. Locality reduces remote access delays that are usually involved in wide area networks.

This point relates to the overhead of distributed computing if the data resides at remote sites and one has to access it by remote communication. The argument is that it is better, in these circumstances, to distribute the data management functionality to where the data is located rather than moving large amounts of data. This is sometimes a topic of contention. Some argue that with the widespread use of high-speed, high-capacity networks, distributing data and data management functions no longer makes sense and that it may be much simpler to store data at a central site using a very large machine and access it over high-speed networks. This is commonly referred to as *scale-up* architecture. It is an appealing argument, but misses an important point of distributed databases. First, in most of today's applications, data is distributed; what may be open for debate is how and where we process it. Second, and more important, point is that this argument does not distinguish between bandwidth (the capacity of the computer links) and latency (how long it takes for data to be transmitted). Latency is inherent in distributed environments and there are physical limits to how fast we can send data over computer networks. Remotely accessing data may incur latencies that might not be acceptable for many applications.

The second point is that the inherent parallelism of distributed systems may be exploited for interquery and intraquery parallelism. *Interquery parallelism* enables the parallel execution of multiple queries generated by concurrent transactions, in order to increase the transactional throughput. The definition of intraquery parallelism is different in distributed versus parallel DBMSs. In the former, intraquery parallelism is achieved by breaking up a single query into a number of subqueries, each of which is executed at a different site, accessing a different part of the distributed database. In parallel DBMSs, it is achieved by *interoperator* and *intraoperator* parallelism. Interoperator parallelism is obtained by executing in parallel different operators of the query tree on different processors, while with intraoperator parallelism, the same operator is executed by many processors, each one working on a subset of the data. Note that these two forms of parallelism also exist in distributed query processing.

Intraoperator parallelism is based on the decomposition of one operator in a set of independent suboperators, called *operator instances*. This decomposition is done using partitioning of relations. Each operator instance will then process one relation partition. The operator decomposition frequently benefits from the initial partitioning of the data (e.g., the data is partitioned on the join attribute). To illustrate intraoperator parallelism, let us consider a simple select-join query. The select operator can be directly decomposed into several select operators, each on a different partition, and no redistribution is required (Fig. 1.6). Note that if the relation is partitioned on the select attribute, partitioning properties can be used to eliminate some select instances. For example, in an exact-match select, only one select instance will be executed if the relation was partitioned by hashing (or range) on the select attribute. It is more complex to decompose the join operator. In order to have independent joins, each partition of one relation R may be joined to the entire other relation S. Such a join will be very inefficient (unless S is very small) because it will imply a broadcast of S on each participating processor. A more efficient way is to use partitioning properties. For example, if R and S are partitioned by hashing on the join attribute and if the join is an equijoin, then we can partition the join into independent joins. This is the ideal case that cannot be always used, because it depends on the initial partitioning of R and S. In the other cases, one or two operands may be repartitioned. Finally, we may notice that the partitioning function (hash, range, round robin—discussed in Sect. 2.3.1) is independent of the local algorithm (e.g., nested loop, hash, sort merge) used to process the join operator (i.e., on each processor). For instance, a hash join using a hash partitioning needs two hash functions. The first one, $h_1$, is used to partition the two base relations on the join attribute. The second one, $h_2$, which can be different for each processor, is used to process the join on each processor.

Two forms of interoperator parallelism can be exploited. With *pipeline parallelism*, several operators with a producer–consumer link are executed in parallel. For instance, the two select operators in Fig. 1.7 will be executed in parallel with the join operator. The advantage of such execution is that the intermediate result does not need to be entirely materialized, thus saving memory and disk accesses. *Independent*

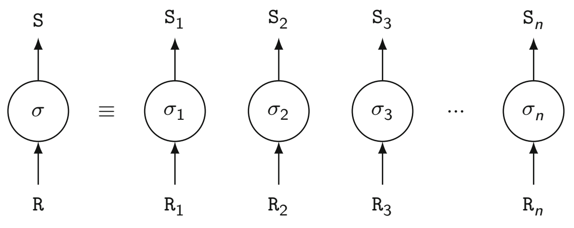

**Fig. 1.6** Intraoperator parallelism. $\sigma_i$ is instance $i$ of the operator; $n$ is the degree of parallelism

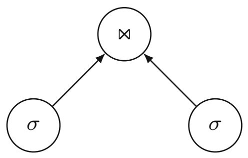

**Fig. 1.7** Interoperator parallelism

*parallelism* is achieved when there is no dependency between the operators that are executed in parallel. For instance, the two select operators of Fig. 1.7 can be executed in parallel. This form of parallelism is very attractive because there is no interference between the processors.

### *1.4.4 Scalability*

In a distributed environment, it is much easier to accommodate increasing database sizes and bigger workloads. System expansion can usually be handled by adding processing and storage power to the network. Obviously, it may not be possible to obtain a linear increase in "power," since this also depends on the overhead of distribution. However, significant improvements are still possible. That is why distributed DBMSs have gained much interest in *scale-out* architectures in the context of cluster and cloud computing. Scale-out (also called *horizontal scaling*) refers to adding more servers, called "scale-out servers" in a loosely coupled fashion, to scale almost infinitely. By making it easy to add new component database servers, a distributed DBMS can provide scale-out.

## 1.5 Design Issues

In the previous section, we discussed the promises of distributed DBMS technology, highlighting the challenges that need to be overcome in order to realize them. In this section, we build on this discussion by presenting the design issues that arise in building a distributed DBMS. These issues will occupy much of the remainder of this book.

### *1.5.1 Distributed Database Design*

The question that is being addressed is how the data is placed across the sites. The starting point is one global database and the end result is a distribution of the data across the sites. This is referred to as *top-down design*. There are two basic alternatives to placing data: *partitioned* (or *nonreplicated*) and *replicated*. In the partitioned scheme the database is divided into a number of disjoint partitions each of which is placed at a different site. Replicated designs can be either *fully replicated* (also called *fully duplicated*) where the entire database is stored at each site, or *partially replicated* (or *partially duplicated*) where each partition of the database is stored at more than one site, but not at all the sites. The two fundamental design issues are *fragmentation*, the separation of the database into partitions called *fragments*, and *distribution*, the optimum distribution of fragments.

A related problem is the design and management of system directory. In centralized DBMSs, the *catalog* contains metainformation (i.e., description) about the data.

In a distributed system, we have a directory that contains additional information such as where data is located. Problems related to directory management are similar in nature to the database placement problem discussed in the preceding section. A directory may be global to the entire distributed DBMS or local to each site; it can be centralized at one site or distributed over several sites; there can be a single copy or multiple copies. Distributed database design and directory management are topics of Chap. 2.

### *1.5.2 Distributed Data Control*

An important requirement of a DBMS is to maintain data consistency by controlling how data is accessed. This is called *data control* and involves view management, access control, and integrity enforcement. Distribution imposes additional challenges since data that is required to check rules is distributed to different sites requiring distributed rule checking and enforcement. The topic is covered in Chap. 3.

### *1.5.3 Distributed Query Processing*

Query processing deals with designing algorithms that analyze queries and convert them into a series of data manipulation operations. The problem is how to decide on a strategy for executing each query over the network in the most cost-effective way, however, cost is defined. The factors to be considered are the distribution of data, communication costs, and lack of sufficient locally available information. The objective is to optimize where the inherent parallelism is used to improve the performance of executing the transaction, subject to the above-mentioned constraints. The problem is NP-hard in nature, and the approaches are usually heuristic. Distributed query processing is discussed in detail in Chap. 4.

### *1.5.4 Distributed Concurrency Control*

Concurrency control involves the synchronization of accesses to the distributed database, such that the integrity of the database is maintained. The concurrency control problem in a distributed context is somewhat different than in a centralized framework. One not only has to worry about the integrity of a single database, but also about the consistency of multiple copies of the database. The condition that requires all the values of multiple copies of every data item to converge to the same value is called *mutual consistency*.

The two general classes of solutions are *pessimistic*, synchronizing the execution of user requests before the execution starts, and *optimistic*, executing the requests and then checking if the execution has compromised the consistency of the database. Two fundamental primitives that can be used with both approaches are *locking*, which is based on the mutual exclusion of accesses to data items, and *timestamping*, where the transaction executions are ordered based on timestamps. There are variations of these schemes as well as hybrid algorithms that attempt to combine the two basic mechanisms.

In locking-based approaches deadlocks are possible since there is mutually exclusive access to data by different transactions. The well-known alternatives of prevention, avoidance, and detection/recovery also apply to distributed DBMSs. Distributed concurrency control is covered in Chap. 5.

### 1.5.5 Reliability of Distributed DBMS

We mentioned earlier that one of the potential advantages of distributed systems is improved reliability and availability. This, however, is not a feature that comes automatically. It is important that mechanisms be provided to ensure the consistency of the database as well as to detect failures and recover from them. The implication for distributed DBMSs is that when a failure occurs and various sites become either inoperable or inaccessible, the databases at the operational sites remain consistent and up-to-date. Furthermore, when the computer system or network recovers from the failure, the distributed DBMSs should be able to recover and bring the databases at the failed sites up-to-date. This may be especially difficult in the case of network partitioning, where the sites are divided into two or more groups with no communication among them. Distributed reliability protocols are the topic of Chap. 5.

### 1.5.6 Replication

If the distributed database is (partially or fully) replicated, it is necessary to implement protocols that ensure the consistency of the replicas, i.e., copies of the same data item have the same value. These protocols can be *eager* in that they force the updates to be applied to all the replicas before the transaction completes, or they may be *lazy* so that the transaction updates one copy (called the *master*) from which updates are propagated to the others after the transaction completes. We discuss replication protocols in Chap. 6.

### *1.5.7 Parallel DBMSs*

As earlier noted, there is a strong relationship between distributed databases and parallel databases. Although the former assumes each site to be a single logical computer, most of these installations are, in fact, parallel clusters. This is the distinction that we highlighted earlier between single site distribution as in data center clusters and geo-distribution. Parallel DBMS objectives are somewhat different from distributed DBMSs in that the main objectives are high scalability and performance. While most of the book focuses on issues that arise in managing data in geo-distributed databases, interesting data management issues exist within a single site distribution as a parallel system. We discuss these issues in Chap. 8.

### *1.5.8 Database Integration*

One of the important developments has been the move towards “looser” federation among data sources, which may also be heterogeneous. As we discuss in the next section, this has given rise to the development of multidatabase systems (also called *federated database systems*) that require reinvestigation of some of the fundamental database techniques. The input here is a set of already distributed databases and the objective is to provide easy access by (physically or logically) integrating them. This involves *bottom-up design*. These systems constitute an important part of today’s distributed environment. We discuss multidatabase systems, or as more commonly termed now *database integration*, including design issues and query processing challenges in Chap. 7.

### *1.5.9 Alternative Distribution Approaches*

The growth of the Internet as a fundamental networking platform has raised important questions about the assumptions underlying distributed database systems. Two issues are of particular concern to us. One is the re-emergence of peer-to-peer computing, and the other is the development and growth of the World Wide Web. Both of these aim at improving data sharing, but take different approaches and pose different data management challenges. We discuss peer-to-peer data management in Chap. 9 and web data management in Chap. 12.

### *1.5.10 Big Data Processing and NoSQL*

The last decade has seen the explosion of “big data” processing. The exact definition of big data is elusive, but they are typically accepted to have four characteristics dubbed the “four V’s”: data is very high *volume*, is multimodal (*variety*), usually comes at very high speed as data streams (*velocity*), and may have quality concerns due to uncertain sources and conflicts (*veracity*). There have been significant efforts to develop systems to deal with "big data," all spurred by the perceived unsuitability of relational DBMSs for a number of new applications. These efforts typically take two forms: one thread has developed general purpose computing platforms (almost always scale-out) for processing, and the other special DBMSs that do not have the full relational functionality, with more flexible data management capabilities (the so-called NoSQL systems). We discuss the big data platforms in Chap. 10 and NoSQL systems in Chap. 11.

## 1.6 Distributed DBMS Architectures

The architecture of a system defines its structure. This means that the components of the system are identified, the function of each component is specified, and the interrelationships and interactions among these components are defined. The specification of the architecture of a system requires identification of the various modules, with their interfaces and interrelationships, in terms of the data and control flow through the system.

In this section, we develop four "reference" architectures[2] for a distributed DBMS: client/server, peer-to-peer, multidatabase, and cloud. These are "idealized" views of a DBMS in that many of the commercially available systems may deviate from them; however, the architectures will serve as a reasonable framework within which the issues related to distributed DBMS can be discussed.

We start with a discussion of the design space to better position the architectures that will be presented.

### *1.6.1 Architectural Models for Distributed DBMSs*

We use a classification (Fig. 1.8) that recognizes three dimensions according to which distributed DBMSs may be architected: (1) the autonomy of local systems, (2) their distribution, and (3) their heterogeneity. These dimensions are orthogonal as we discuss shortly and in each dimension we identify a number of alternatives. Consequently, there are 18 possible architectures in the design space; not all of these architectural alternatives are meaningful, and most are not relevant from the perspective of this book. The three on which we focus are identified in Fig. 1.8.

[2] A reference architecture is commonly created by standards developers to clearly define the interfaces that need to be standardized.

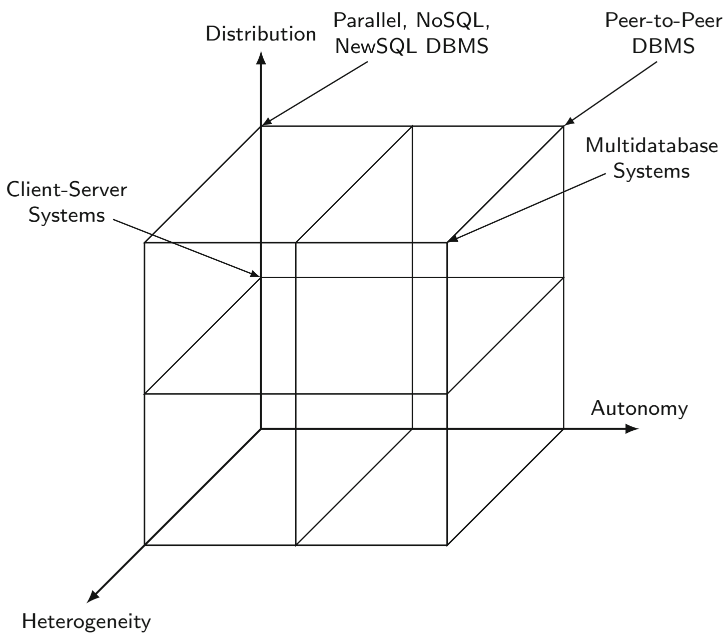

**Fig. 1.8** DBMS implementation alternatives

#### 1.6.1.1 Autonomy

*Autonomy*, in this context, refers to the distribution of control, not of data. It indicates the degree to which individual DBMSs can operate independently. Autonomy is a function of a number of factors such as whether the component systems (i.e., individual DBMSs) exchange information, whether they can independently execute transactions, and whether one is allowed to modify them.

We will use a classification that covers the important aspects of these features. This classification highlights three alternatives. One alternative is *tight integration*, where a single-image of the entire database is available to any user who wants to share the data that may reside in multiple databases. From the users' perspective, the data is logically integrated in one database. In these tightly integrated systems, the data managers are implemented so that one of them is in control of the processing of each user request even if that request is serviced by more than one data manager. The data managers do not typically operate as independent DBMSs even though they usually have the functionality to do so.

Next, we identify *semiautonomous* systems that consist of DBMSs that can (and usually do) operate independently, but have decided to participate in a federation to make their local data sharable. Each of these DBMSs determines what parts of their own database they will make accessible to users of other DBMSs. They are not fully autonomous systems because they need to be modified to enable them to exchange information with one another.

The last alternative that we consider is *total isolation*, where the individual systems are stand-alone DBMSs that know neither of the existence of other DBMSs nor how to communicate with them. In such systems, the processing of user transactions that access multiple databases is especially difficult since there is no global control over the execution of individual DBMSs.

#### 1.6.1.2 Distribution

Whereas autonomy refers to the distribution (or decentralization) of control, the distribution dimension of the taxonomy deals with data. Of course, we are considering the physical distribution of data over multiple sites; as we discussed earlier, the user sees the data as one logical pool. There are a number of ways DBMSs have been distributed. We abstract these alternatives into two classes: *client/server* distribution and *peer-to-peer* distribution (or *full* distribution). Together with the nondistributed option, the taxonomy identifies three alternative architectures.

The client/server distribution concentrates data management duties at servers, while the clients focus on providing the application environment including the user interface. The communication duties are shared between the client machines and servers. Client/server DBMSs represent a practical compromise to distributing functionality. There are a variety of ways of structuring them, each providing a different level of distribution. We leave detailed discussion to Sect. 1.6.2.

In *peer-to-peer systems*, there is no distinction of client machines versus servers. Each machine has full DBMS functionality and can communicate with other machines to execute queries and transactions. Most of the very early work on distributed database systems have assumed peer-to-peer architecture. Therefore, our main focus in this book is on peer-to-peer systems (also called *fully distributed*), even though many of the techniques carry over to client/server systems as well.

#### 1.6.1.3 Heterogeneity

Heterogeneity may occur in various forms in distributed systems, ranging from hardware heterogeneity and differences in networking protocols to variations in data managers. The important ones from the perspective of this book relate to data models, query languages, and transaction management protocols. Representing data with different modeling tools creates heterogeneity because of the inherent expressive powers and limitations of individual data models. Heterogeneity in query languages not only involves the use of completely different data access paradigms in different data models (set-at-a-time access in relational systems versus record-at-a-time access in some object-oriented systems), but also covers differences in languages even when the individual systems use the same data model. Although SQL is now the standard relational query language, there are many different implementations and every vendor's language has a slightly different flavor (sometimes even different semantics, producing different results). Furthermore, big data platforms and NoSQL systems have significantly variable access languages and mechanisms.

### 1.6.2 Client/Server Systems

Client/server entered the computing scene at the beginning of 1990s and has made a significant impact on the DBMS technology. The general idea is very simple and elegant: distinguish the functionality that needs to be provided on a server machine from those that need to be provided on a client. This provides a *two-level architecture* which makes it easier to manage the complexity of modern DBMSs and the complexity of distribution.

In relational client/server DBMSs, the server does most of the data management work. This means that all of query processing and optimization, transaction management, and storage management are done at the server. The client, in addition to the application and the user interface, has a *DBMS client* module that is responsible for managing the data that is cached to the client and (sometimes) managing the transaction locks that may have been cached as well. It is also possible to place consistency checking of user queries at the client side, but this is not common since it requires the replication of the system catalog at the client machines. This architecture, depicted in Fig. 1.9, is quite common in relational systems where the communication between the clients and the server(s) is at the level of SQL

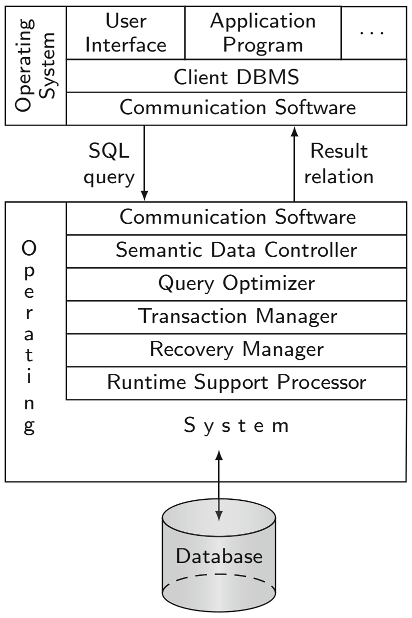

**Fig. 1.9** Client/server reference architecture

statements. In other words, the client passes SQL queries to the server without trying to understand or optimize them. The server does most of the work and returns the result relation to the client.

There are a number of different realizations of the client/server architecture. The simplest is the case where there is only one server which is accessed by multiple clients. We call this *multiple client/single server*. From a data management perspective, this is not much different from centralized databases since the database is stored on only one machine (the server) that also hosts the software to manage it. However, there are important differences from centralized systems in the way transactions are executed and caches are managed—since data is cached at the client, it is necessary to deploy cache coherence protocols.

A more sophisticated client/server architecture is one where there are multiple servers in the system (the so-called *multiple client/multiple server* approach). In this case, two alternative management strategies are possible: either each client manages its own connection to the appropriate server or each client knows of only its "home server" which then communicates with other servers as required. The former approach simplifies server code, but loads the client machines with additional responsibilities. This leads to what has been called "heavy client" systems. The latter approach, on the other hand, concentrates the data management functionality at the servers. Thus, the transparency of data access is provided at the server interface, leading to "light clients."

In the multiple server systems, data is partitioned and may be replicated across the servers. This is transparent to the clients in the case of light client approach, and servers may communicate among themselves to answer a user query. This approach is implemented in parallel DBMS to improve performance through parallel processing.

Client/server can be naturally extended to provide for a more efficient function distribution on different kinds of servers: *clients* run the user interface (e.g., web servers), *application servers* run application programs, and *database servers* run database management functions. This leads to the three-tier distributed system architecture.

The application server approach (indeed, an n-tier distributed approach) can be extended by the introduction of multiple database servers and multiple application servers (Fig. 1.10), as can be done in classical client/server architectures. In this case, it is typically the case that each application server is dedicated to one or a few applications, while database servers operate in the multiple server fashion discussed above. Furthermore, the interface to the application is typically through a load balancer that routes the client requests to the appropriate servers.

The database server approach, as an extension of the classical client/server architecture, has several potential advantages. First, the single focus on data management makes possible the development of specific techniques for increasing data reliability and availability, e.g., using parallelism. Second, the overall performance of database management can be significantly enhanced by the tight integration of the database system and a dedicated database operating system. Finally, database servers can also exploit advanced hardware assists such as GPUs and FPGAs to enhance both performance and data availability.

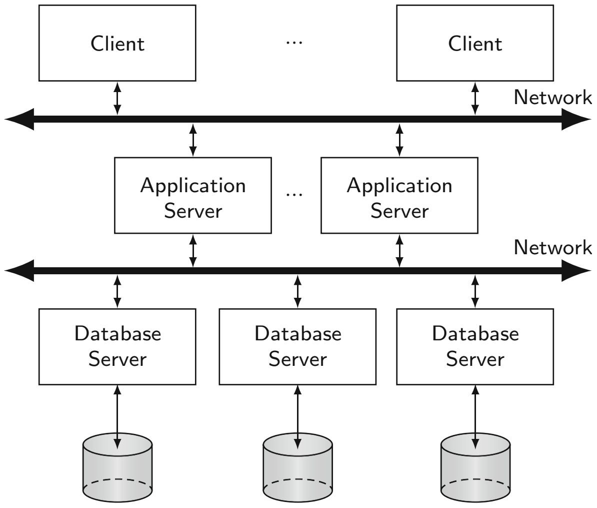

**Fig. 1.10** Distributed database servers

Although these advantages are significant, there is the additional overhead introduced by another layer of communication between the application and the data servers. The communication cost can be amortized if the server interface is sufficiently high level to allow the expression of complex queries involving intensive data processing.

## *1.6.3 Peer-to-Peer Systems*

The early works on distributed DBMSs all focused on peer-to-peer architectures where there was no differentiation between the functionality of each site in the system. Modern peer-to-peer systems have two important differences from their earlier relatives. The first is the massive distribution in more recent systems. While in the early days the focus was on a few (perhaps at most tens of) sites, current systems consider thousands of sites. 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, as discussed earlier, coupled with massive distribution, site heterogeneity and autonomy take on an added significance, disallowing some of the approaches from consideration. In this book we initially focus on the classical meaning of peer-to-peer (the same functionality at each site), since the principles and fundamental techniques of these systems are very similar to those of client/server systems, and discuss the modern peer-to-peer database issues in a separate chapter (Chap. 9).

In these systems, the database design follows a top-down design as discussed earlier. So, the input is a (centralized) database with its own schema definition (*global conceptual schema*—GCS). This database is partitioned and allocated to sites of the distributed DBMS. Thus, at each site, there is a local database with its own schema (called the *local conceptual schema*—LCS). The user formulates queries according to the GCS, irrespective of its location. The distributed DBMS translates global queries into a group of local queries, which are executed by distributed DBMS components at different sites that communicate with one another. From a querying perspective, peer-to-peer systems and client/server DBMSs provide the same view of data. That is, they give the user the appearance of a logically single database, while at the physical level data is distributed.

The detailed components of a distributed DBMS are shown in Fig. 1.11. One component handles the interaction with users, and another deals with the storage.

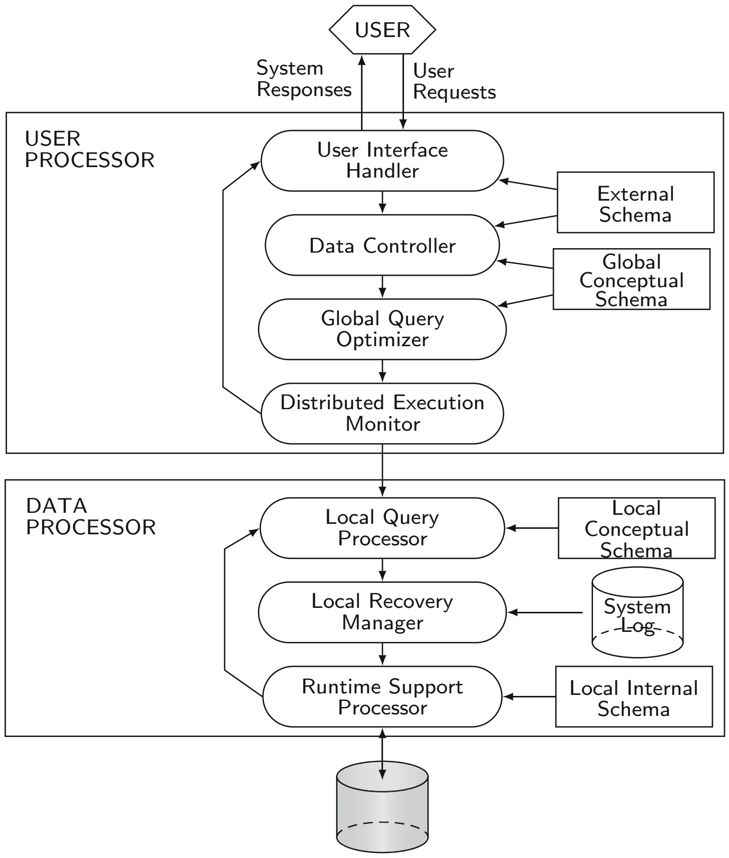

**Fig. 1.11** Components of a distributed DBMS

The first major component, which we call the *user processor*, consists of four elements:

1. The *user interface handler* is responsible for interpreting user commands as they come in, and formatting the result data as it is sent to the user.
2. The *data controller* uses the integrity constraints and authorizations that are defined as part of the global conceptual schema to check if the user query can be processed. This component, which is studied in detail in Chap. 3, is also responsible for authorization and other functions.
3. The *global query optimizer and decomposer* determines an execution strategy to minimize a cost function, and translates the global queries into local ones using the global and local conceptual schemas as well as the global directory. The global query optimizer is responsible, among other things, for generating the best strategy to execute distributed join operations. These issues are discussed in Chap. 4.
4. The *distributed execution monitor* coordinates the distributed execution of the user request. The execution monitor is also called the *distributed transaction manager*. In executing queries in a distributed fashion, the execution monitors at various sites may, and usually do, communicate with one another. Distributed transaction manager functionality is covered in Chap. 5.

The second major component of a distributed DBMS is the *data processor* and consists of the following three elements. These are all issues that centralized DBMSs deal with, so we do not focus on them in this book.

1. The *local query optimizer*, which actually acts as the *access path selector*, is responsible for choosing the best access path[3] to access any data item.
2. The *local recovery manager* is responsible for making sure that the local database remains consistent even when failures occur.
3. The *runtime support processor* physically accesses the database according to the physical commands in the schedule generated by the query optimizer. The runtime support processor is the interface to the operating system and contains the *database buffer* (or *cache*) *manager*, which is responsible for maintaining the main memory buffers and managing the data accesses.

It is important to note that our use of the terms “user processor” and “data processor” does not imply a functional division similar to client/server systems. These divisions are merely organizational and there is no suggestion that they should be placed on different machines. In peer-to-peer systems, one expects to find both the user processor modules and the data processor modules on each machine. However, there can be “query-only sites” that only have the user processor.

[3]The term *access path* refers to the data structures and the algorithms that are used to access the data. A typical access path, for example, is an index on one or more attributes of a relation.

### *1.6.4 Multidatabase Systems*

Multidatabase systems (MDBSs) represent the case where individual DBMSs are fully autonomous and have no concept of cooperation; they may not even "know" of each other's existence or how to talk to each other. Our focus is, naturally, on distributed MDBSs, which refers to the MDBS where participating DBMSs are located on different sites. Many of the issues that we discussed are common to both single-node and distributed MDBSs; in those cases we will simply use the term MDBS without qualifying it as single node or distributed. In most current literature, one finds the term *database integration* used instead. We discuss these systems further in Chap. 7. We note, however, that there is considerable variability in the use of the term "multidatabase" in literature. In this book, we use it consistently as defined above, which may deviate from its use in some of the existing literature.

The differences in the level of autonomy between the MDBSs and distributed DBMSs are also reflected in their architectural models. The fundamental difference relates to the definition of the global conceptual schema. In the case of logically integrated distributed DBMSs, the global conceptual schema defines the conceptual view of the *entire* database, while in the case of MDBSs, it represents only the collection of *some* of the local databases that each local DBMS wants to share. The individual DBMSs may choose to make some of their data available for access by others. Thus the definition of a *global database* is different in MDBSs than in distributed DBMSs. In the latter, the global database is equal to the union of local databases, whereas in the former it is only a (possibly proper) subset of the same union. In an MDBS, the GCS (which is also called a *mediated schema*) is defined by integrating (possibly parts of) local conceptual schemas.

The component-based architectural model of a distributed MDBS is significantly different from a distributed DBMS, because each site is a full-fledged DBMS that manages a different database. The MDBS provides a layer of software that runs on top of these individual DBMSs and provides users with the facilities of accessing various databases (Fig. 1.12). Note that in a distributed MDBS, the MDBS layer may run on multiple sites or there may be central site where those services are offered. Also note that as far as the individual DBMSs are concerned, the MDBS layer is simply another application that submits requests and receives answers.

A popular implementation architecture for MDBSs is the mediator/wrapper approach (Fig. 1.13). A *mediator* "is a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications" [Wiederhold 1992]. Thus, each mediator performs a particular function with clearly defined interfaces. Using this architecture to implement an MDBS, each module in the MDBS layer of Fig. 1.12 is realized as a mediator. Since mediators can be built on top of other mediators, it is possible to construct a layered implementation. The mediator level implements the GCS. It is this level that handles user queries over the GCS and performs the MDBS functionality.

The mediators typically operate using a common data model and interface language. To deal with potential heterogeneities of the source DBMSs, *wrappers*

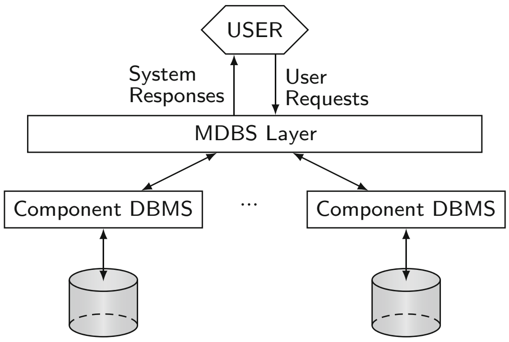

**Fig. 1.12** Components of an MDBS

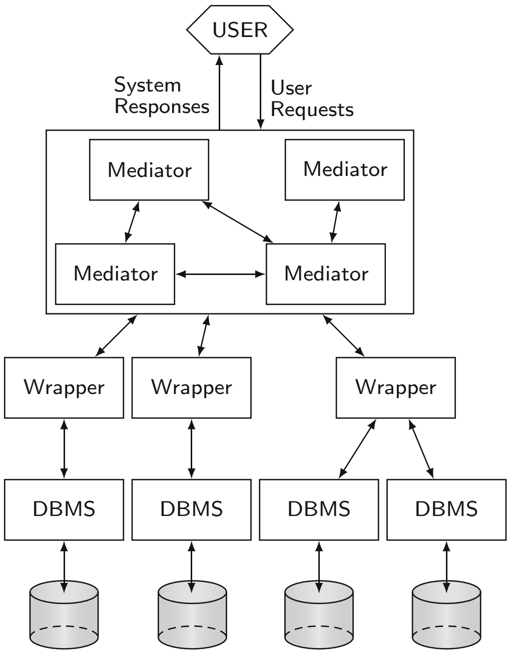

**Fig. 1.13** Mediator/wrapper architecture

are implemented whose task is to provide a mapping between a source DBMSs view and the mediators' view. For example, if the source DBMS is a relational one, but the mediator implementations are object-oriented, the required mappings are established by the wrappers. The exact role and function of mediators differ from one implementation to another. In some cases, mediators do nothing more than translation; these are called "thin" mediators. In other cases, wrappers take over the execution of some of the query functionality.

One can view the collection of mediators as a middleware layer that provides services above the source systems. Middleware is a topic that has been the subject of significant study in the past decade and very sophisticated middleware systems have been developed that provide advanced services for development of distributed applications. The mediators that we discuss only represent a subset of the functionality provided by these systems.

### *1.6.5 Cloud Computing*

Cloud computing has caused a significant shift in how users and organizations deploy scalable applications, in particular, data management applications. 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, database management, 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.

Cloud computing is a natural evolution, and combination, of different computing models proposed for supporting applications over the web: service-oriented architectures (SOA) for high-level communication of applications through web services, utility computing for packaging computing and storage resources as services, cluster and virtualization technologies to manage lots of computing and storage resources, and autonomous computing to enable self-management of complex infrastructure. The cloud provides various levels of functionality such as:

- Infrastructure-as-a-Service (IaaS): the delivery of a computing infrastructure (i.e., computing, networking, and storage resources) as a service;
- Platform-as-a-Service (PaaS): the delivery of a computing platform with development tools and APIs as a service;
- Software-as-a-Service (SaaS): the delivery of application software as a service; or
- Database-as-a-Service (DaaS): the delivery of database as a service.

What makes cloud computing unique is its ability to provide and combine all kinds of services to best fit the users' requirements. From a technical point of view, the grand challenge is to support in a cost-effective way, the very large scale of the infrastructure that has to manage lots of users and resources with high quality of service.

Agreeing on a precise definition of cloud computing is difficult as there are many different perspectives (business, market, technical, research, etc.). However, a good working definition is that a "cloud provides on demand resources and services over the Internet, usually at the scale and with the reliability of a data center" [Grossman and Gu 2009]. This definition captures well the main objective (providing on-demand resources and services over the Internet) and the main requirements for supporting them (at the scale and with the reliability of a data center). Since the resources are accessed through services, everything gets delivered as a service. Thus, as in the services industry, this enables cloud providers to propose a pay-as-you-go pricing model, whereby users only pay for the resources they consume.

The main functions provided by clouds are: security, directory management, resource management (provisioning, allocation, monitoring), and data management (storage, file management, database management, data replication). In addition, clouds provide support for pricing, accounting, and service level agreement management.

The typical advantages of cloud computing are the following:

- **Cost.** The cost for the customer can be greatly reduced since the infrastructure does not need to be owned and managed; billing is only based on resource consumption. As for the cloud provider, using a consolidated infrastructure and sharing costs for multiple customers reduces the cost of ownership and operation.
- **Ease of access and use.** The cloud hides the complexity of the IT infrastructure and makes location and distribution transparent. Thus, customers can have access to IT services anytime, and from anywhere with an Internet connection.
- **Quality of service.** The operation of the IT infrastructure by a specialized provider that has extensive experience in running very large infrastructures (including its own infrastructure) increases quality of service and operational efficiency.
- **Innovation.** Using state-of-the-art tools and applications provided by the cloud encourages modern practice, thus increasing the innovation capabilities of the customers.
- **Elasticity.** The ability to scale resources out, up and down dynamically to accommodate changing conditions is a major advantage. This is typically achieved through server virtualization, a technology that enables multiple applications to run on the same physical computer as virtual machines (VMs), i.e., as if they would run on distinct physical computers. Customers can then require computing instances as VMs and attach storage resources as needed.

However, there are also disadvantages that must be well-understood before moving to the cloud. These disadvantages are similar to when outsourcing applications and data to an external company.

- **Provider dependency.** Cloud providers tend to lock in customers, through proprietary software, proprietary format, or high outbound data transfer costs, thus making cloud service migration difficult.
- **Loss of control.** Customers may lose managerial control over critical operations such as system downtime, e.g., to perform a software upgrade.
- **Security.** Since a customer's cloud data is accessible from anywhere on the Internet, security attacks can compromise business's data. Cloud security can be improved using advanced capabilities, e.g., virtual private cloud, but may be complex to integrate with a company's security policy.

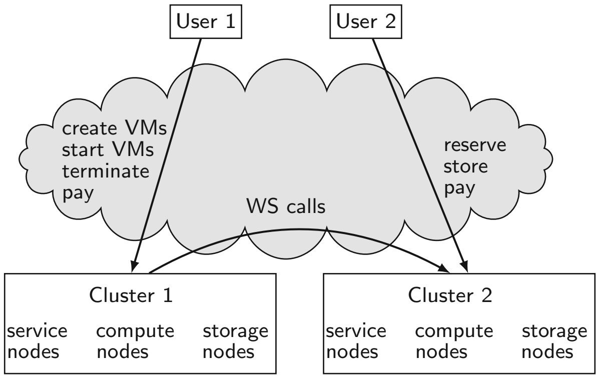

**Fig. 1.14** Simplified cloud architecture

- **Hidden costs.** Customizing applications to make them cloud-ready using SaaS/-PaaS may incur significant development costs.

There is no standard cloud architecture and there will probably never be one, since different cloud providers provide different cloud services (IaaS, PaaS, SaaS, etc.) in different ways (public, private, virtual private, etc.) depending on their business models. Thus, in this section, we discuss a simplified cloud architecture with emphasis on database management.

A cloud is typically multisite (Fig. 1.14), i.e., made of several geographically distributed sites (or data centers), each with its own resources and data. Major cloud providers divide the world in several regions, each with several sites. There are three major reasons for this. First, there is low latency access in a user's region since user requests can be directed to the closest site. Second, using data replication across sites in different regions provides high availability, in particular, resistance from catastrophic (site) failures. Third, some national regulations that protect citizen's data privacy force cloud providers to locate data centers in their region (e.g., Europe). Multisite transparency is generally a default option, so the cloud appears "centralized" and the cloud provider can optimize resource allocation to users. However, some cloud providers (e.g., Amazon and Microsoft) make their sites visible to users (or application developers). This allows choosing a particular data center to install an application with its database, or to deploy a very large application across multiple sites communicating through web services (WS). For instance, in Fig. 1.14, we could imagine that Client 1 first connects to an application at Data Center 1, which would call an application at Data Center 2 using WS.

The architecture of a cloud site (data center) is typically 3-tier. The first tier consists of web clients that access cloud web servers, typically via a router or load balancer at the cloud site. The second tier consists of web/application servers that support the clients and provide business logic. The third tier consists of database servers. There can be other kinds of servers, e.g., cache servers between the application servers and database servers. Thus, the cloud architecture provides two levels of distribution: geographical distribution across sites using a WAN and within a site, distribution across the servers, typically in a computer cluster. The techniques used at the first level are those of geographically distributed DBMS, while the techniques used at the second level are those of parallel DBMS.

Cloud computing has been originally designed by web giants to run their very large scale applications on data centers with thousands of servers. Big data systems (Chap. 10) and NoSQL/NewSQL systems (Chap. 11) specifically address the requirements of such applications in the cloud, using distributed data management techniques. With the advent of SaaS and PaaS solutions, cloud providers also need to serve small applications for very high numbers of customers, called *tenants*, each with its own (small) database accessed by its users. Dedicating a server for each tenant is wasteful in terms of hardware resources. To reduce resource wasting and operation cost, cloud providers typically share resources among tenants using a "multitenancy" architecture in which a single server can accommodate multiple tenants. Different multitenant models yield different trade-offs between performance, isolation (both security and performance isolation), and design complexity. A straightforward model used in IaaS is hardware sharing, which is typically achieved through server virtualization, with a VM for each tenant database and operating system. This model provides strong security isolation. However, resource utilization is limited because of redundant DBMS instances (one per VM) that do not cooperate and perform independent resource management. In the context of SaaS, PaaS, or DaaS, we can distinguish three main multitenant database models with increasing resource sharing and performance at the expense of less isolation and increased complexity.

- **Shared DBMS server.** In this model, tenants share a server with one DBMS instance, but each tenant has a different database. Most DBMSs provide support for multiple databases in a single DBMS instance. Thus, this model can be easily supported using a DBMS. It provides strong isolation at the database level and is more efficient than shared hardware as the DBMS instance has full control over hardware resources. However, managing each of these databases separately may still lead to inefficient resource management.
- **Shared database.** In this model, tenants share a database, but each tenant has its own schema and tables. Database consolidation is typically provided by an additional abstraction layer in the DBMS. This model is implemented by some DBMS (e.g., Oracle) using a single container database hosting multiple databases. It provides good resource usage and isolation at schema level. However, with lots (thousands) of tenants per server, there is a high number of small tables, which induces much overhead.
- **Shared tables.** In this model, tenants share a database, schema, and tables. To distinguish the rows of different tenants in a table, there is usually an additional column tenant_id. Although there is better resource sharing (e.g., cache memory), there is less isolation, both in security and performance. For instance, bigger customers will have more rows in shared tables, thus hurting the performance for smaller customers.

## 1.7 Bibliographic Notes

There are not many books on distributed DBMSs. The two early ones by Ceri and Pelagatti [1983] and Bell and Grimson [1992] are now out of print. A more recent book by Rahimi and Haug [2010] covers some of the classical topics that are also covered in this book. In addition, almost every database book now has a chapter on distributed DBMSs.

The pioneering systems Distributed INGRES and SDD-1 are discussed in [Stonebraker and Neuhold 1977] and [Wong 1977], respectively.

Database design is discussed in an introductory manner in [Levin and Morgan 1975] and more comprehensively in [Ceri et al. 1987]. A survey of the file distribution algorithms is given in [Dowdy and Foster 1982]. Directory management has not been considered in detail in the research community, but general techniques can be found in [Chu and Nahouraii 1975] and [Chu 1976]. A survey of query processing techniques can be found in [Sacco and Yao 1982]. Concurrency control algorithms are reviewed in [Bernstein and Goodman 1981] and [Bernstein et al. 1987]. Deadlock management has also been the subject of extensive research; an introductory paper is [Isloor and Marsland 1980] and a widely quoted paper is [Obermack 1982]. For deadlock detection, good surveys are [Knapp 1987] and [Elmagarmid 1986]. Reliability is one of the issues discussed in [Gray 1979], which is one of the landmark papers in the field. Other important papers on this topic are [Verhofstadt 1978] and [Härder and Reuter 1983]. [Gray 1979] is also the first paper discussing the issues of operating system support for distributed databases; the same topic is addressed in [Stonebraker 1981]. Unfortunately, both papers emphasize centralized database systems. A very good early survey of multidatabase systems is by Sheth and Larson [1990]; Wiederhold [1992] proposes the mediator/wrapper approach to MDBSs. Cloud computing has been the topic of quite a number of recent books; perhaps [Agrawal et al. 2012] is a good starting point and [Cusumano 2010] is a good short overview. The architecture we used in Sect. 1.6.5 is from [Agrawal et al. 2012]. Different multitenant models in cloud environments are discussed in [Curino et al. 2011] and [Agrawal et al. 2012].

There have been a number of architectural framework proposals. Some of the interesting ones include Schreiber's quite detailed extension of the ANSI/SPARC framework which attempts to accommodate heterogeneity of the data models [Schreiber 1977], and the proposal by Mohan and Yeh [1978]. As expected, these date back to the early days of the introduction of distributed DBMS technology. The detailed component-wise system architecture given in Fig. 1.11 derives from [Rahimi 1987]. An alternative to the classification that we provide in Fig. 1.8 can be found in [Sheth and Larson 1990].

The book by Agrawal et al. [2012] gives a very good presentation of the challenges and concepts of data management in the cloud, including distributed transactions, big data systems, and multitenant databases.