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.

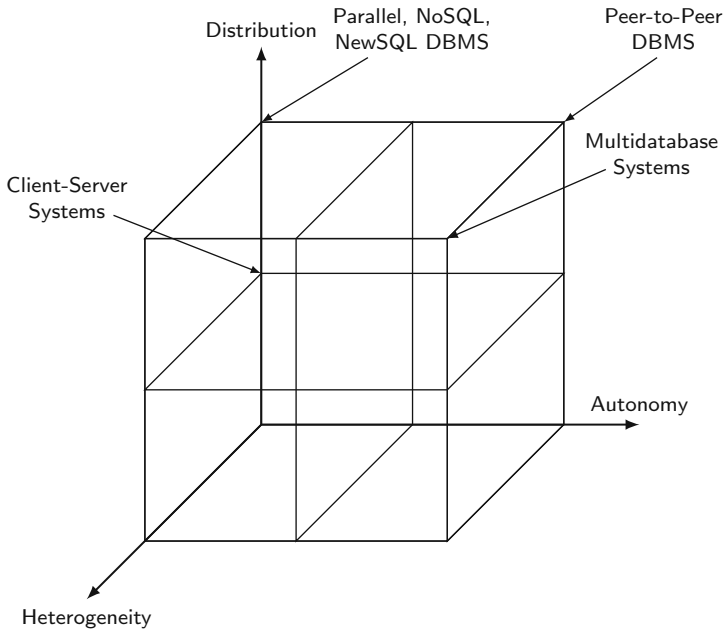

Fig. 1.8 DBMS implementation alternatives

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

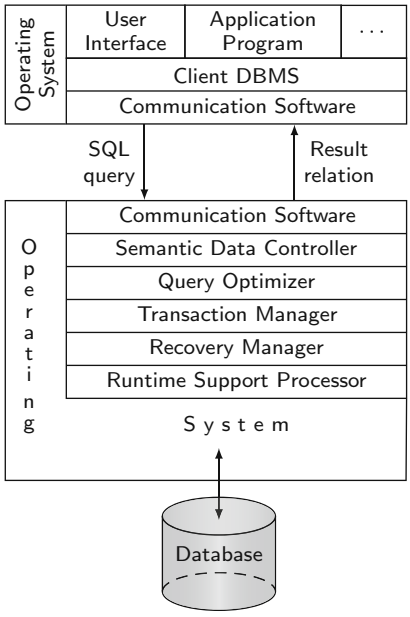

Fig. 1.9 Client/server reference architecture

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.

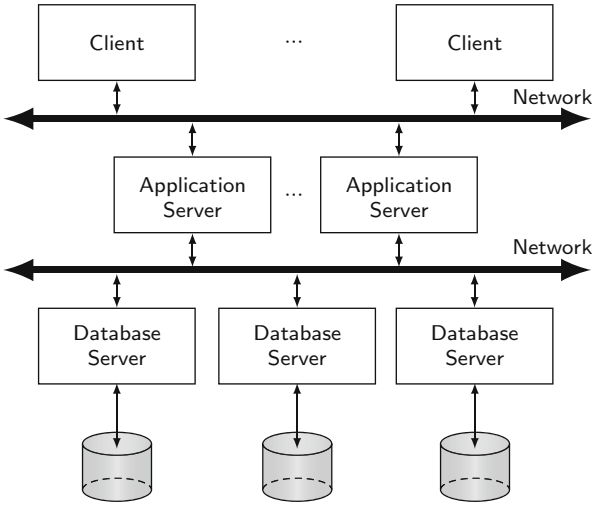

Fig. 1.10 Distributed database servers

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.

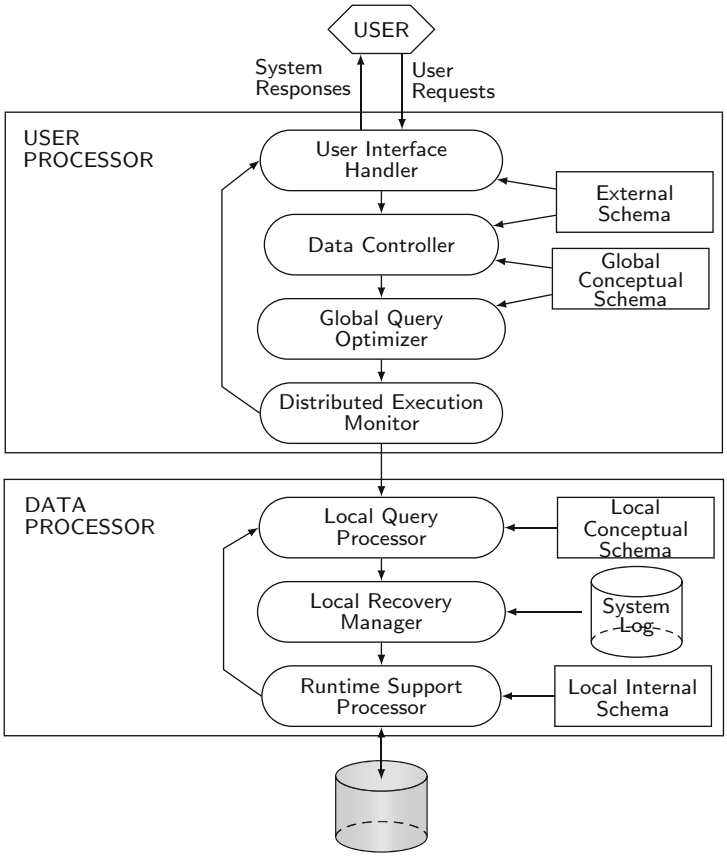

Fig. 1.11 Components of a distributed DBMS

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*

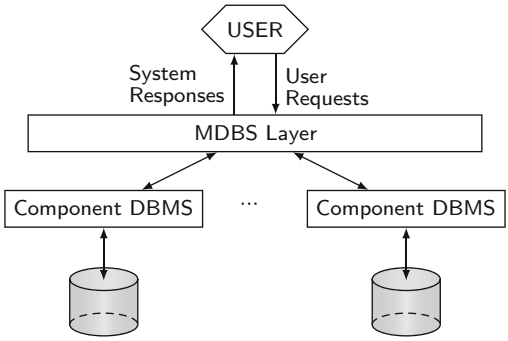

Fig. 1.12 Components of an MDBS

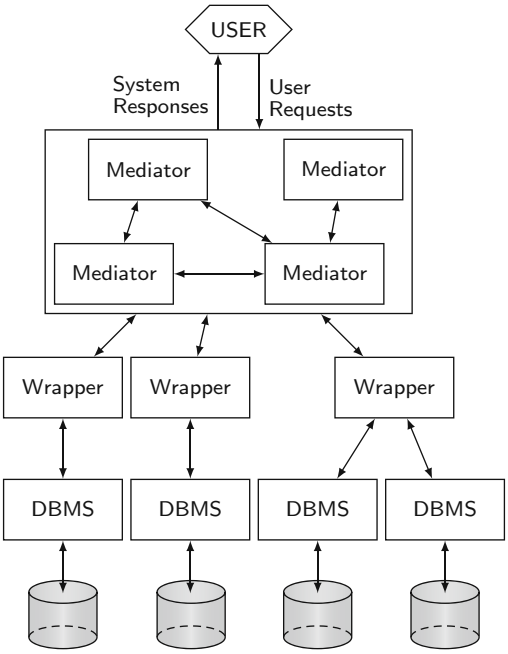

Fig. 1.13 Mediator/wrapper architecture

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.

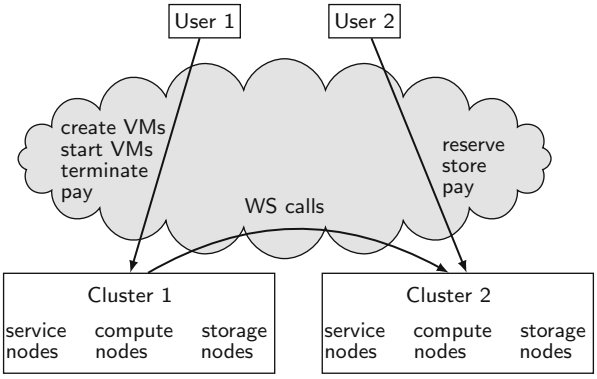

Fig. 1.14 Simplified cloud architecture

- **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.