DBMS - Architecture

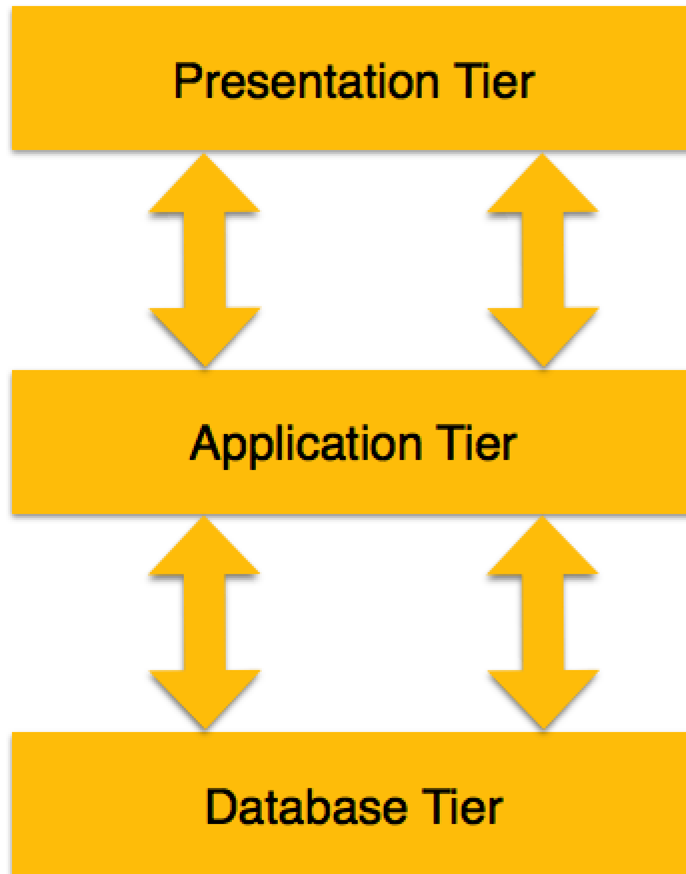
The design of a DBMS depends on its architecture. It can be centralized or decentralized or hierarchical. The architecture of a DBMS can be seen as either single tier or multi-tier. An n-tier architecture divides the whole system into related but independent **n** modules, which can be independently modified, altered, changed, or replaced.

In 1-tier architecture, the DBMS is the only entity where the user directly sits on the DBMS and uses it. Any changes done here will directly be done on the DBMS itself. It does not provide handy tools for end-users. Database designers and programmers normally prefer to use single-tier architecture.

If the architecture of DBMS is 2-tier, then it must have an application through which the DBMS can be accessed. Programmers use 2-tier architecture where they access the DBMS by means of an application. Here the application tier is entirely independent of the database in terms of operation, design, and programming.

3-tier Architecture

A 3-tier architecture separates its tiers from each other based on the complexity of the users and how they use the data present in the database. It is the most widely used architecture to design a DBMS.



- **Database (Data) Tier** − At this tier, the database resides along with its query processing languages. We also have the relations that define the data and their constraints at this level.

- **Application (Middle) Tier** − At this tier reside the application server and the programs that access the database. For a user, this application tier presents an abstracted view of the database. End-users are unaware of any existence of the database beyond the application. At the other end, the database tier is not aware of any other user beyond the application tier. Hence, the application layer sits in the middle and acts as a mediator between the end-user and the database.

- **User (Presentation) Tier** − End-users operate on this tier and they know nothing about any existence of the database beyond this layer. At this layer,

multiple views of the database can be provided by the application. All views are generated by applications that reside in the application tier.

Multiple-tier database architecture is highly modifiable, as almost all its components are independent and can be changed independently.

Centralized and Client/Server Architectures for DBMSs

1. Centralized DBMSs Architecture

Architectures for DBMSs have followed trends similar to those for general computer system architectures. Earlier architectures used mainframe computers to provide the main processing for all system functions, including user application programs and user interface programs, as well as all the DBMS functionality. The reason was that most users accessed such systems via computer terminals that did not have processing power and only provided display capabilities. Therefore, all processing was performed remotely on the computer system, and only display information and controls were sent from the computer to the display terminals, which were connected to the central computer via various types of communications networks.

As prices of hardware declined, most users replaced their terminals with PCs and workstations. At first, database systems used these computers similarly to how they had used display terminals, so that the DBMS itself was still a **centralized** DBMS

in which all the DBMS functionality, application program execution, and user inter-face processing were carried out on one machine. Figure 2.4 illustrates the physical components in a centralized architecture. Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.

2. Basic Client/Server Architectures

First, we discuss client/server architecture in general, then we see how it is applied to DBMSs. The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers,

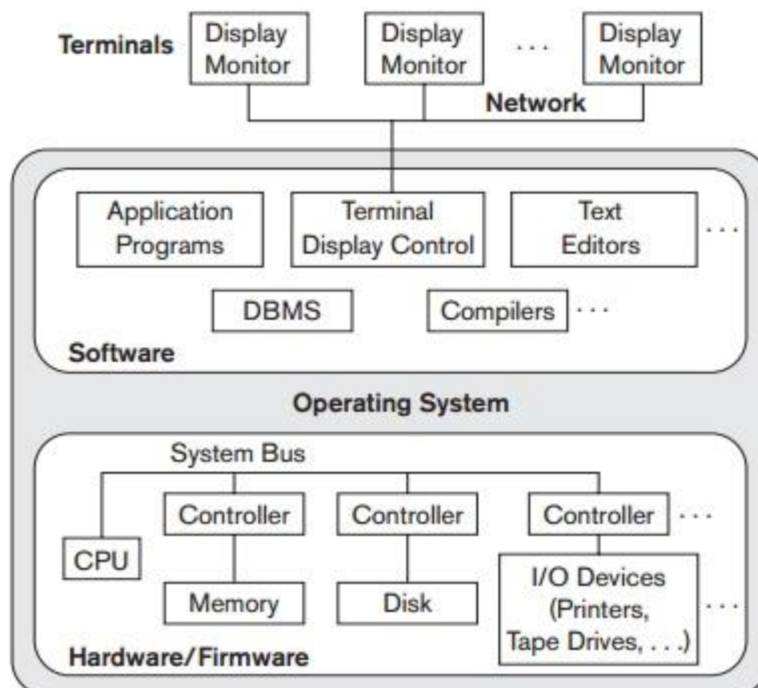


Figure 2.4
A physical centralized architecture.

Web servers, e-mail servers, and other software and equipment are connected via a network. The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client machines. Another machine can be designated as a **printer server** by being connected to various printers; all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** also fall into the specialized server category. The resources provided by specialized servers can be accessed by many client machines. The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications. This concept can be carried over to other software packages, with specialized programs—such as a CAD (computer-aided design) package—being stored on specific server machines and being made accessible to multiple clients. Figure 2.5 illustrates client/server architecture at the logical level; Figure 2.6 is a simplified diagram that shows the physical architecture. Some machines would be client sites only (for example, diskless work-stations or workstations/PCs with disks that have only client software installed).

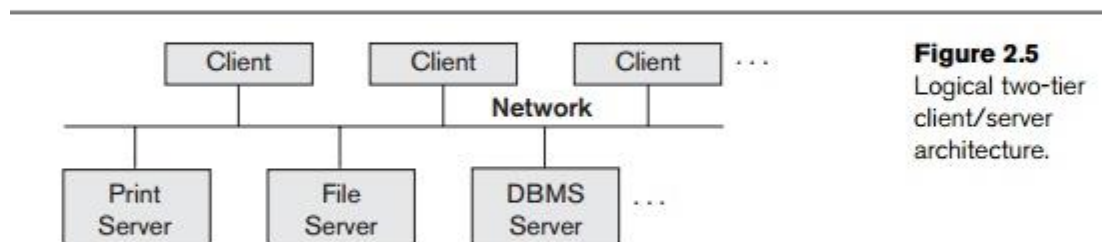


Figure 2.5
Logical two-tier client/server architecture.

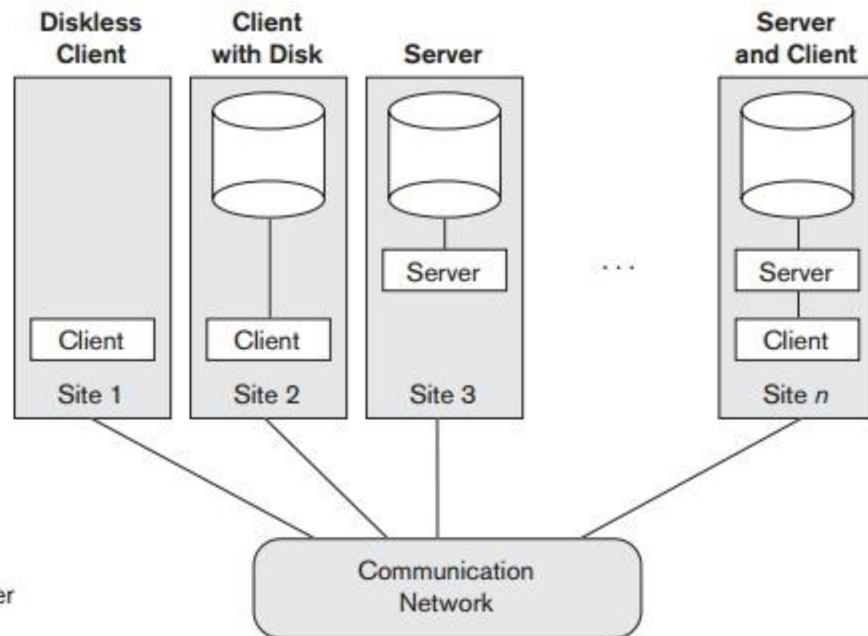


Figure 2.6
Physical two-tier client/server architecture.

Other machines would be dedicated servers, and others would have both client and server functionality.

The concept of client/server architecture assumes an underlying framework that consists of many PCs and workstations as well as a smaller number of mainframe machines, connected via LANs and other types of computer networks. A **client** in this framework is typically a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality—such as database access—that does not exist at that machine, it connects to a server that provides the needed functionality. A **server** is a system containing both hardware and software that can provide services to the client machines, such as file access, printing, archiving, or database access. In general, some machines install only client software, others only server software, and still others may include both client and server software, as illustrated in Figure 2.6. However, it is more

common that client and server software usually run on separate machines. Two main types of basic DBMS architectures were created on this underlying client/server framework: **two-tier** and **three-tier**.

3. Two-Tier Client/Server Architectures for DBMSs

In relational database management systems (RDBMSs), many of which started as centralized systems, the system components that were first moved to the client side were the user interface and application programs. Because SQL (see Chapters 4 and 5) provided a standard language for RDBMSs, this created a logical dividing point between client and server. Hence, the query and transaction functionality related to SQL processing remained on the server side. In such an architecture, the server is often called a **query server** or **transaction server** because it provides these two functionalities. In an RDBMS, the server is also often called an **SQL server**.

The user interface programs and application programs can run on the client side. When DBMS access is required, the program establishes a connection to the DBMS (which is on the server side); once the connection is created, the client program can communicate with the DBMS. A standard called **Open Database Connectivity**

(**ODBC**) provides an **application programming interface** (**API**), which allows client-side programs to call the DBMS, as long as both client and server machines

have the necessary software installed. Most DBMS vendors provide ODBC drivers for their systems. A client program can actually connect to several RDBMSs and send query and transaction requests using the ODBC API, which are then processed at the server sites. Any query results are sent back to the client program, which can process and display the results as needed. A related standard for the Java programming language, called **JDBC**, has also been defined. This allows Java client programs to access one or more DBMSs through a standard interface.

The different approach to two-tier client/server architecture was taken by some object-oriented DBMSs, where the software modules of the DBMS were divided between client and server in a more integrated way. For example, the **server level** may include the part of the DBMS software responsible for handling data storage on disk pages, local concurrency control and recovery, buffering and caching of disk pages, and other such functions. Meanwhile, the **client level** may handle the user interface; data dictionary functions; DBMS interactions with programming language compilers; global query optimization, concurrency control, and recovery across multiple servers; structuring of complex objects from the data in the buffers; and other such functions. In this approach, the client/server interaction is more tightly coupled and is done internally by the DBMS modules—some of which reside on the client and some on the server—rather than by the users/programmers. The exact division of functionality can vary from system to system. In such a client/server architecture, the server has been called a **data server** because it provides data in disk pages to the client. This data can then be structured into objects for the client programs by the client-side DBMS software.

The architectures described here are called **two-tier architectures** because the soft-ware components are distributed over two systems: client and server. The advan-tages of this architecture are its simplicity and seamless compatibility with existing systems. The emergence of the Web changed the roles of clients and servers, leading to the three-tier architecture.

4. Three-Tier and n-Tier Architectures for Web Applications

Many Web applications use an architecture called the **three-tier architecture**, which adds an intermediate layer between the client and the database server, as illustrated in Figure 2.7(a).

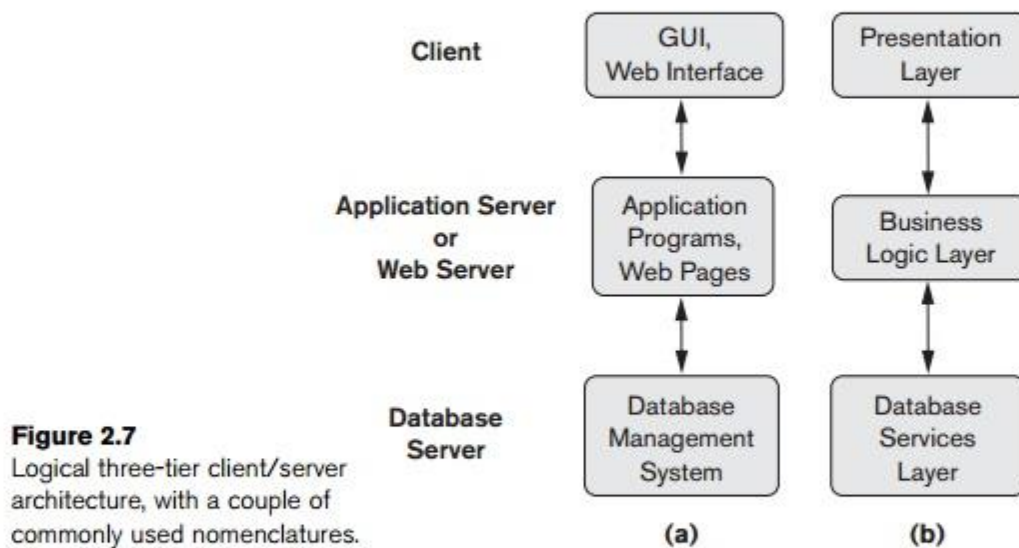


Figure 2.7
Logical three-tier client/server architecture, with a couple of commonly used nomenclatures.

This intermediate layer or **middle tier** is called the **application server** or the **Web server**, depending on the application. This server plays an intermediary role by run-ning application programs and storing business rules (procedures or constraints) that are used to access data from the database server. It can also

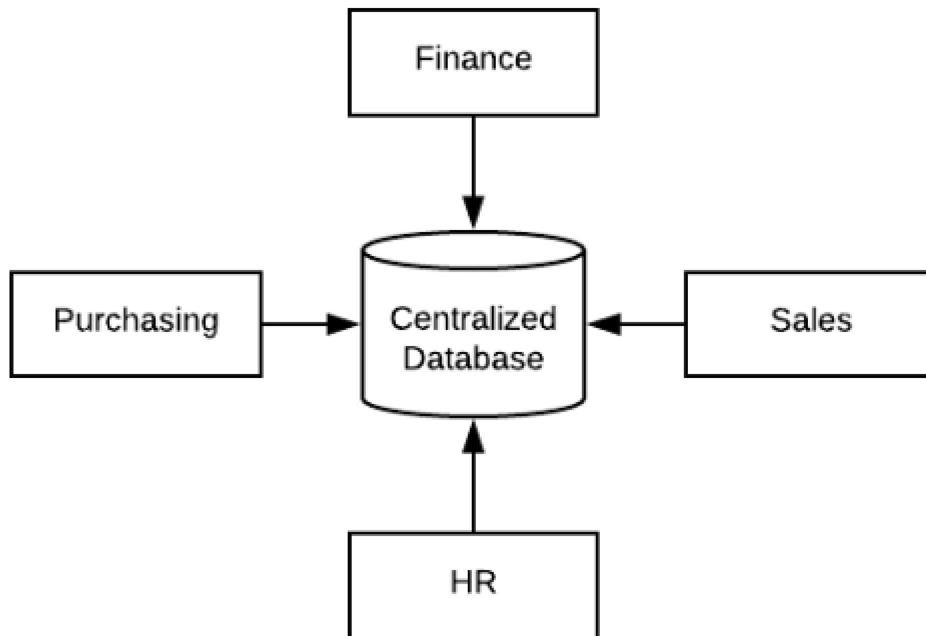
improve database security by checking a client's credentials before forwarding a request to the data-base server. Clients contain GUI interfaces and some additional application-specific business rules. The intermediate server accepts requests from the client, processes the request and sends database queries and commands to the database server, and then acts as a conduit for passing (partially) processed data from the database server to the clients, where it may be processed further and filtered to be presented to users in GUI format. Thus, the *user interface, application rules,* and *data access* act as the three tiers. Figure 2.7(b) shows another architecture used by database and other application package vendors. The presentation layer displays information to the user and allows data entry. The business logic layer handles intermediate rules and constraints before data is passed up to the user or down to the DBMS. The bottom layer includes all data management services. The middle layer can also act as a Web server, which retrieves query results from the database server and formats them into dynamic Web pages that are viewed by the Web browser at the client side.

Other architectures have also been proposed. It is possible to divide the layers between the user and the stored data further into finer components, thereby giving rise to *n*-tier architectures; where *n* may be four or five tiers. Typically, the business logic layer is divided into multiple layers. Besides distributing programming and data throughout a network, *n*-tier applications afford the advantage that any one tier can run on an appropriate processor or operating system platform and can be handled independently. Vendors of ERP (enterprise resource planning) and CRM (customer relationship management) packages often use a *middleware layer,* which accounts for the front-end modules (clients) communicating with a number of back-end databases (servers).

Advances in encryption and decryption technology make it safer to transfer sensitive data from server to client in encrypted form, where it will be decrypted. The latter can be done by the hardware or by advanced software. This technology gives higher levels of data security, but the network security issues remain a major concern. Various technologies for data compression also help to transfer large amounts of data from servers to clients over wired and wireless networks.

Centralized Database Management System

A centralized database is stored at a single location such as a mainframe computer. It is maintained and modified from that location only and usually accessed using an internet connection such as a LAN or WAN. The centralized database is used by organizations such as colleges, companies, banks etc.



As can be seen from the above diagram, all the information for the organisation is stored in a single database. This database is known as the centralized database.

Advantages

Some advantages of Centralized Database Management System are −

- The data integrity is maximised as the whole database is stored at a single physical location. This means that it is easier to coordinate the data and it is as accurate and consistent as possible.
- The data redundancy is minimal in the centralised database. All the data is stored together and not scattered across different locations. So, it is easier to make sure there is no redundant data available.
- Since all the data is in one place, there can be stronger security measures around it. So, the centralised database is much more secure.
- Data is easily portable because it is stored at the same place.

- The centralized database is cheaper than other types of databases as it requires less power and maintenance.
- All the information in the centralized database can be easily accessed from the same location and at the same time.

Disadvantages

Some disadvantages of Centralized Database Management System are −

- Since all the data is at one location, it takes more time to search and access it. If the network is slow, this process takes even more time.
- There is a lot of data access traffic for the centralized database. This may create a bottleneck situation.
- Since all the data is at the same location, if multiple users try to access it simultaneously it creates a problem. This may reduce the efficiency of the system.
- If there are no database recovery measures in place and a system failure occurs, then all the data in the database will be destroyed.

Client Server System

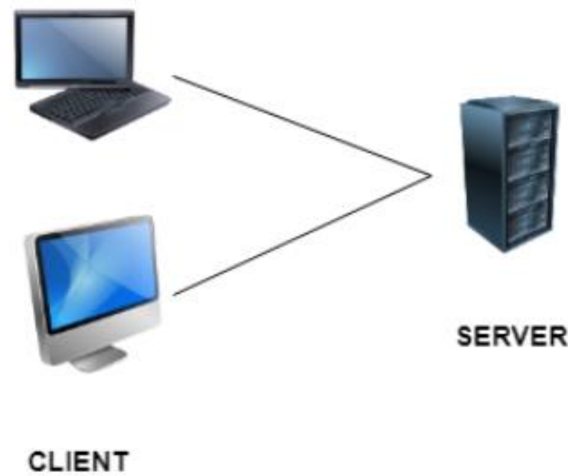
In client server computing, the clients request a resource and the server provides that resource. A server may serve multiple clients at the same time while a client is in contact with only one server.

The different structures for two tier and three tier are given as follows −

Two - Tier Client/Server Structure

The two tier architecture primarily has two parts, a client tier and a server tier.The client tier sends a request to the server tier and the server tier responds with the desired information.

An example of a two tier client/server structure is a web server. It returns the required web pages to the clients that requested them.



An illustration of the two-tier client/server structure is as above

Advantages of Two - Tier Client/Server Structure

Some of the advantages of the two-tier client/server structure are −

- This structure is quite easy to maintain and modify.
- The communication between the client and server in the form of request response messages is quite fast.

Disadvantages of Two - Tier Client/Server Structure

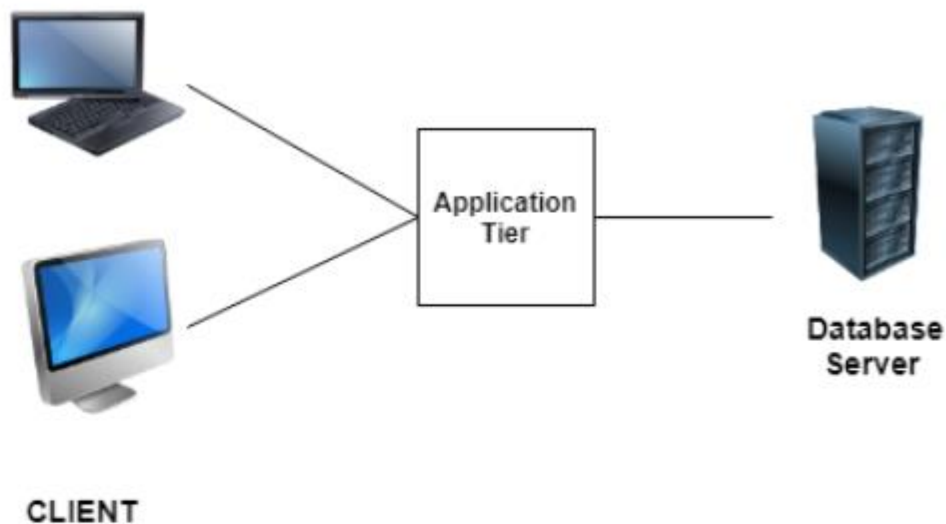
A major disadvantage of the two-tier client/server structure is −

- If the client nodes are increased beyond capacity in the structure, then the server is not able to handle the request overflow and performance of the system degrades.

Three - Tier Client/Server Structure

The three tier architecture has three layers namely client, application and data layer. The client layer is the one that requests the information. In this case it could be the GUI, web interface etc. The application layer acts as an interface between the client and data layer. It helps in communication and also provides security. The data layer is the one that actually contains the required data.

An illustration of the three-tier client/server structure is as follows −



Advantages of Three - Tier Client/Server Structure

Some of the advantages of the three-tier client/server structure are −

- The three tier structure provides much better service and fast performance.
- The structure can be scaled according to requirements without any problem.

- Data security is much improved in the three tier structure.

Disadvantages of Three - Tier Client/Server Structure

A major disadvantage of the three-tier client/server structure is −

- Three - tier client/server structure is quite complex due to advanced features.

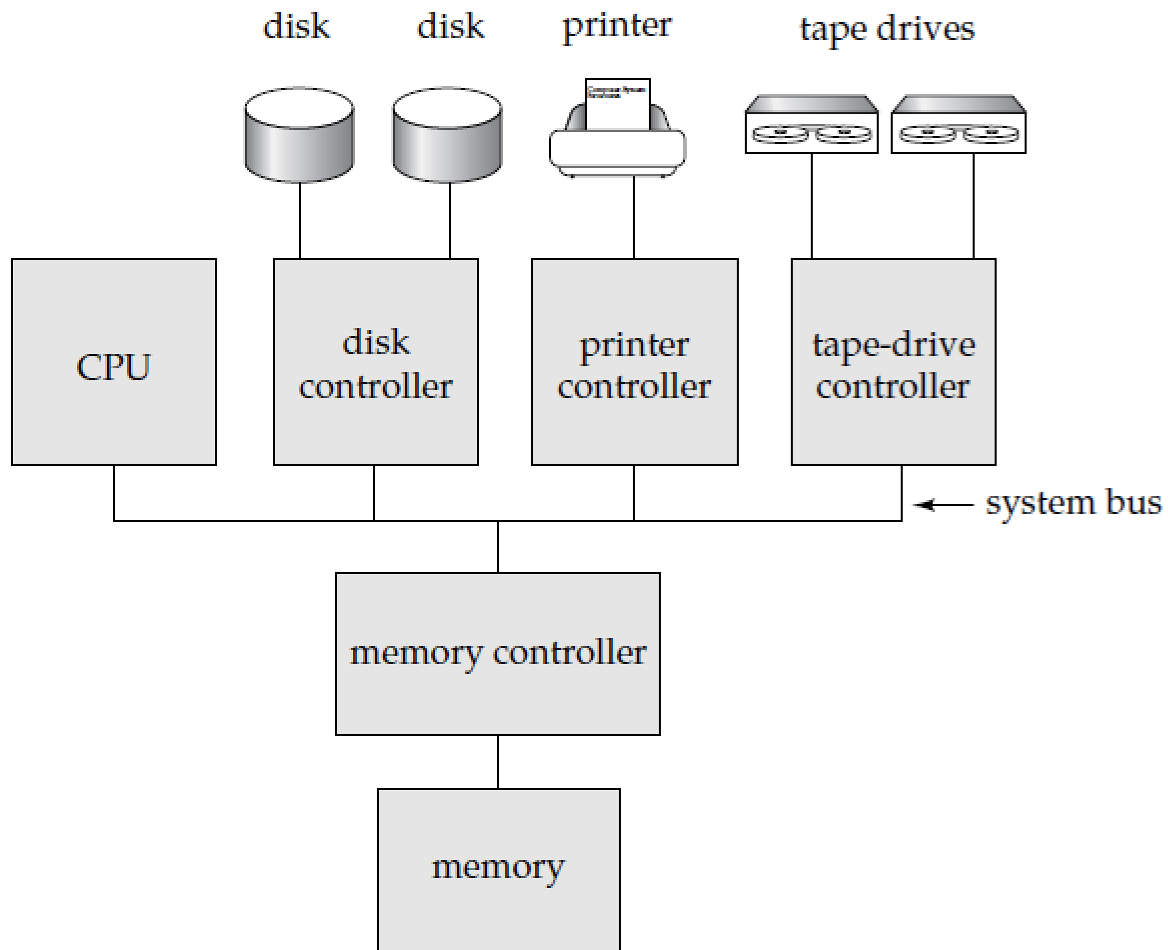
Centralized and Client–Server Architectures

Centralized database systems are those that run on a single computer system and do not interact with other computer systems. Such database systems span a range from single-user database systems running on personal computers to high-performance database systems running on high-end server systems. Client–server systems, on the other hand, have functionality split between a server system, and multiple client systems.

Centralized Systems

A modern, general-purpose computer system consists of one to a few CPUs and a number of device controllers that are connected through a common bus that provides access to shared memory . The CPUs have local cache memories that store local copies of parts of the memory, to speed up access to data. Each device controller is in charge of a specific type of device (for example, a disk drive, an audio device, or a video display). The CPUs and the device controllers can execute concurrently, competing for memory access. Cache memory reduces the contention for memory access, since it reduces the number of times that the CPU needs to access the shared memory.

We distinguish two ways in which computers are used: as single-user systems and as multiuser systems. Personal computers and workstations fall into the first category. A typical single-user system is a desktop unit used by a single person, usually with only one CPU and one or two hard disks, and usually only one person using the machine at a time. A typical multiuser system, on the other hand, has more disks and more memory, may have multiple CPUs and has a multiuser operating system. It serves a large number of users who are connected to the system via terminals.



A centralized computer system.

Database systems designed for use by single users usually do not provide many of the facilities that a multiuser database provides. In particular, they may not support concurrency control, which is not required when only a single user can generate updates. Provisions for crash-recovery in such systems are either absent or primitive for example, they may consist of simply making a backup of the database before any update. Many such systems do not support SQL, and provide a simpler query language, such as a variant of QBE. In contrast, database systems designed for multi user systems support the full transactional features that we have studied earlier.

Although general-purpose computer systems today have multiple processors, they have coarse-granularity parallelism, with only a few processors (about two to four, typically), all sharing the main memory. Databases running on such machines usually do not attempt to partition a single query among the processors; instead, they run each query on a single processor, allowing multiple queries to run concurrently. Thus, such systems support a higher throughput; that is, they allow a greater number of transactions to run per second, although individual transactions do not run any faster.

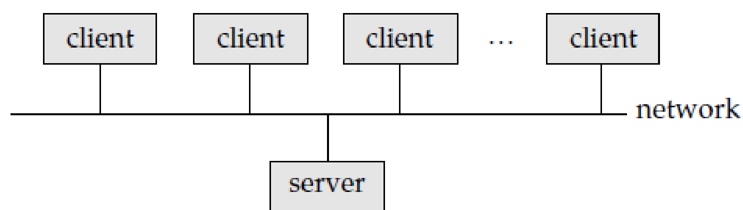
Databases designed for single-processor machines already provide multitasking, allowing multiple processes to run on the same processor in a time-shared manner, giving a view to the user of multiple processes running in parallel. Thus, coarse granularity parallel machines logically appear to be identical to single-processor machines, and database systems designed for time-shared machines can be easily adapted to run on them.

In contrast, machines with fine-granularity parallelism have a large number of processors, and database systems running on such machines attempt to parallelize

single tasks (queries, for example) submitted by users. We study the architecture of parallel database systems.

Client–Server Systems

As personal computers became faster, more powerful, and cheaper, there was a shift away from the centralized system architecture. Personal computers supplanted terminals connected to centralized systems. Correspondingly, personal computers assumed the user-interface functionality that used to be handled directly by the centralized systems. As a result, centralized systems today act as server systems that satisfy requests generated by client systems. Figure shows the general structure of a client–server system. Database functionality can be broadly divided into two parts the front end and the back end as in Figure . The back end manages access structures, query evaluation and optimization, concurrency control, and recovery. The front end of a database system consists of tools such as forms, report writers, and graphical user interface facilities. The interface between the front end and the back end is through SQL, or through an application program.

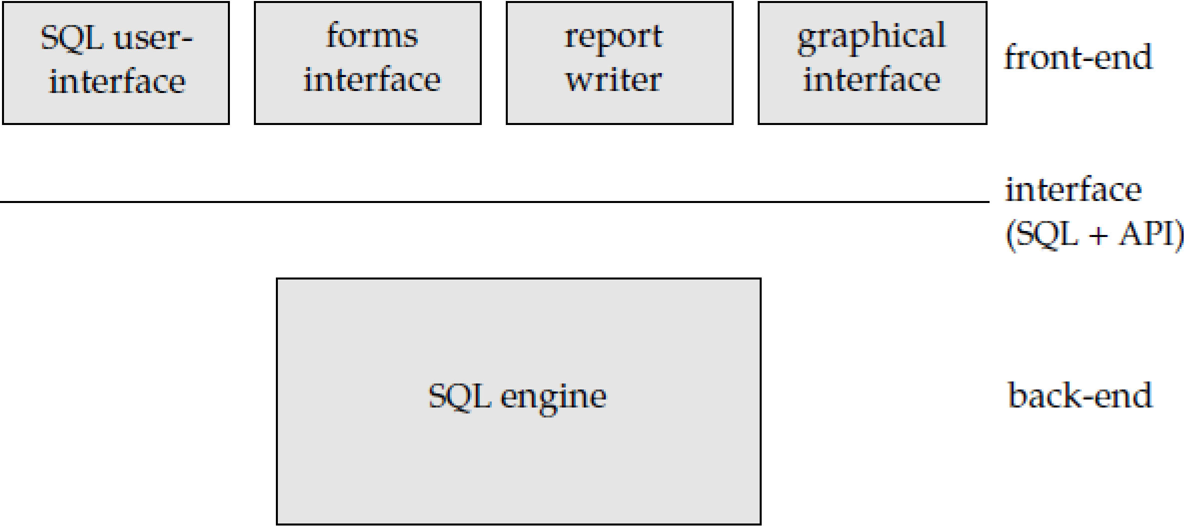


General structure of a client–server system.

Standards such as ODBC and JDBC, were developed to interface clients with servers. Any client that uses the ODBC or JDBC interfaces can connect to any server that provides the interface. In earlier-generation database systems, the lack of such standards necessitated that the front end and the back end be provided by the same software vendor. With the growth of interface standards, the front-end

user interface and the back-end server are often provided by different vendors. Application development tools are used to construct user interfaces; they provide graphical tools that can be used to construct interfaces without any programming. Some of the popular application development tools are PowerBuilder, Magic, and Borland Delphi; Visual Basic is also widely used for application development.

Further, certain application programs, such as spreadsheets and statistical-analysis packages, use the client–server interface directly to access data from a back-end server. In effect, they provide front ends specialized for particular tasks. Some transaction-processing systems provide a transactional remote procedure call interface to connect clients with a server. These calls appear like ordinary procedure calls to the programmer, but all the remote procedure calls from a client are enclosed in a single transaction at the server end. Thus, if the transaction aborts, the server can undo the effects of the individual remote procedure calls.



Front-end and back-end functionality.

Server System Architectures

Server systems can be broadly categorized as transaction servers and data servers.

- Transaction-server systems, also called query-server systems, provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client. Usually, client machines ship transactions to the server systems, where those transactions are executed, and results are shipped back to clients that are in charge of displaying the data. Requests may be specified by using SQL, or through a specialized application program interface.
- Data-server systems allow clients to interact with the servers by making requests to read or update data, in units such as files or pages. For example, file servers provide a file-system interface where clients can create, update, read, and delete files. Data servers for database systems offer much more functionality; they support units of data such as pages, tuples, or objects that are smaller than a file. They provide indexing facilities for data, and provide transaction facilities so that the data are never left in an inconsistent state if a client machine or process fails.

Of these, the transaction-server architecture is by far the more widely used architecture. We shall elaborate on the transaction-server and data-server architectures.

Parallel Systems

Parallel systems improve processing and I/O speeds by using multiple CPUs and disks in parallel. Parallel machines are becoming increasingly common, making the study of parallel database systems correspondingly more important. The driving force behind parallel database systems is the demands of applications that have to query extremely large databases (of the order of terabytes that is, 1012 bytes) or that have to process an extremely large number of transactions per second

(of the order of thousands of transactions per second). Centralized and client–server database systems are not powerful enough to handle such applications.

In parallel processing, many operations are performed simultaneously, as opposed to serial processing, in which the computational steps are performed sequentially. A coarse-grain parallel machine consists of a small number of powerful processors; a massively parallel or fine-grain parallel machine uses thousands of smaller processors.

Most high-end machines today offer some degree of coarse-grain parallelism: Two or four processor machines are common. Massively parallel computers can be distinguished from the coarse-grain parallel machines by the much larger degree of parallelism that they support. Parallel computers with hundreds of CPUs and disks are available commercially.

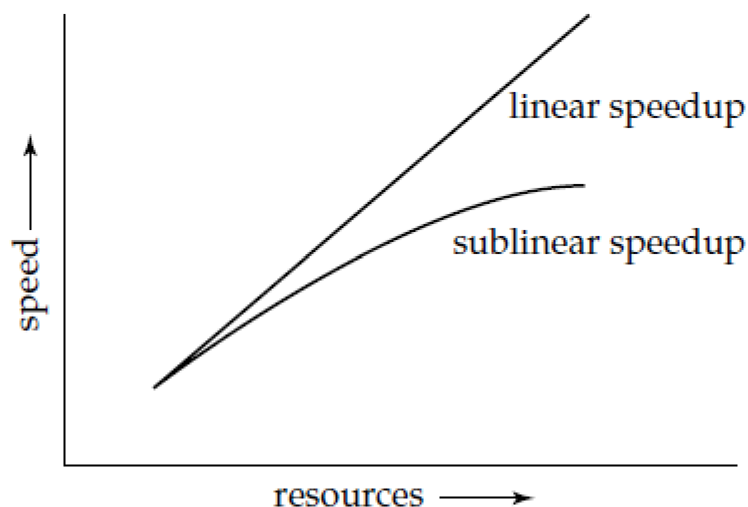
There are two main measures of performance of a database system: (1) throughput, the number of tasks that can be completed in a given time interval, and (2) response time, the amount of time it takes to complete a single task from the time it is submitted. A system that processes a large number of small transactions can improve throughput by processing many transactions in parallel. A system that processes large transactions can improve response time as well as throughput by performing subtasks of each transaction in parallel.

Speedup and Scale up

Two important issues in studying parallelism are speedup and scale up. Running a given task in less time by increasing the degree of parallelism is called speedup. Handling larger tasks by increasing the degree of parallelism is called scale up. Consider a database application running on a parallel system with a certain number of processors and disks. Now suppose that we increase the size of the system by

increasing the number or processors, disks, and other components of the system. The goal is to process the task in time inversely proportional to the number of processors and disks allocated. Suppose that the execution time of a task on the larger machine is TL, and that the execution time of the same task on the smaller machine is TS.

The speedup due to parallelism is defined as TS/TL. The parallel system is said to demonstrate linear speedup if the speedup is N when the larger system has N times the resources (CPU, disk, and so on) of the smaller system. If the speedup is less than N, the system is said to demonstrate sub linear speedup. Figure illustrates linear and sub linear speedup.



Speedup with increasing resources.

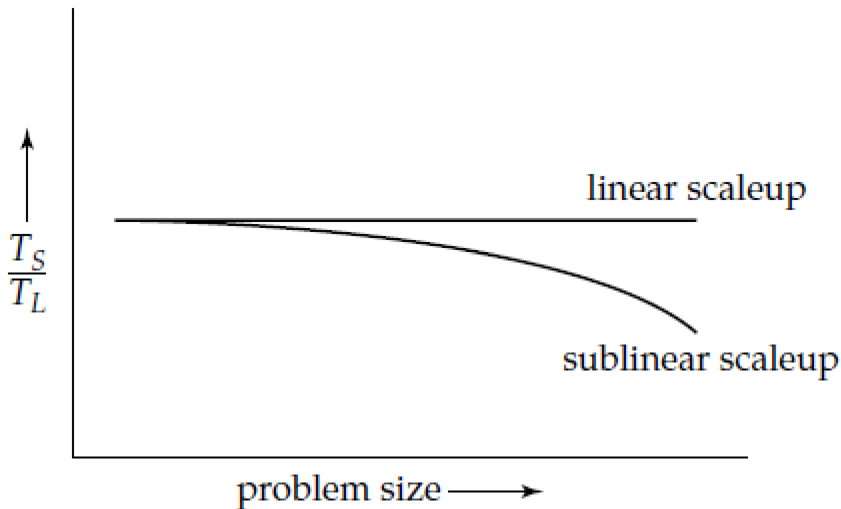
Scale up relates to the ability to process larger tasks in the same amount of time by providing more resources. Let Q be a task, and let QN be a task that is N times bigger than Q. Suppose that the execution time of task Q on a given machine MS is TS, and the execution time of task QN on a parallel machine ML, which is N times larger than MS, is TL. The scale up is then defined as TS/TL. The parallel system ML is said to demonstrate linear scale up on task Q if TL = TS. If TL > TS, the

system is said to demonstrate sub linear scale up. Figure illustrates linear and sub linear scale ups (where the resources increase proportional to problem size). There are two kinds of scale up that are relevant in parallel database systems, depending on how the size of the task is measured:

- In batch scale up, the size of the database increases, and the tasks are large jobs whose runtime depends on the size of the database. An example of such a task is a scan of a relation whose size is proportional to the size of the database. Thus, the size of the database is the measure of the size of the problem. Batch scale up also applies in scientific applications, such as executing a query at an N-times finer resolution or performing an N-times longer simulation.
- In transaction scale up, the rate at which transactions are submitted to the database increases and the size of the database increases proportionally to the transaction rate. This kind of scale up is what is relevant in transaction processing systems where the transactions are small updates for example, a deposit or withdrawal from an account and transaction rates grow as more accounts are created. Such transaction processing is especially well adapted for parallel execution, since transactions can run concurrently and independently on separate processors, and each transaction takes roughly the same amount of time, even if the database grows.

Scale up is usually the more important metric for measuring efficiency of parallel database systems. The goal of parallelism in database systems is usually to make sure that the database system can continue to perform at an acceptable speed, even as the size of the database and the number of transactions increases. Increasing the capacity of the system by increasing the parallelism provides a smoother path for growth for an enterprise than does replacing a centralized system by a faster machine (even assuming that such a machine exists). However, we must also look

at absolute performance numbers when using scale up measures; a machine that scales up linearly may perform worse than a machine that scales less than linearly, simply because the latter machine is much faster to start off with



Scale up with increasing problem size and resources.

A number of factors work against efficient parallel operation and can diminish both speedup and scale up.

- Startup costs. There is a startup cost associated with initiating a single process. In a parallel operation consisting of thousands of processes, the startup time may overshadow the actual processing time, affecting speedup adversely.
- Interference. Since processes executing in a parallel system often access shared resources, a slowdown may result from the interference of each new process as it competes with existing processes for commonly held resources, such as a system bus, or shared disks, or even locks. Both speedup and scale up are affected by this phenomenon.
- Skew. By breaking down a single task into a number of parallel steps, we reduce the size of the average step. Nonetheless, the service time for the single slowest step will determine the service time for the task as a whole. It is often difficult to

divide a task into exactly equal-sized parts, and the way that the sizes are distributed is therefore skewed. For example, if a task of size 100 is divided into 10 parts, and the division is skewed, there may be some tasks of size less than 10 and some tasks of size more than 10; if even one task happens to be of size 20, the speedup obtained by running the tasks in parallel is only five, instead of ten as we would have hoped.

Distributed Systems

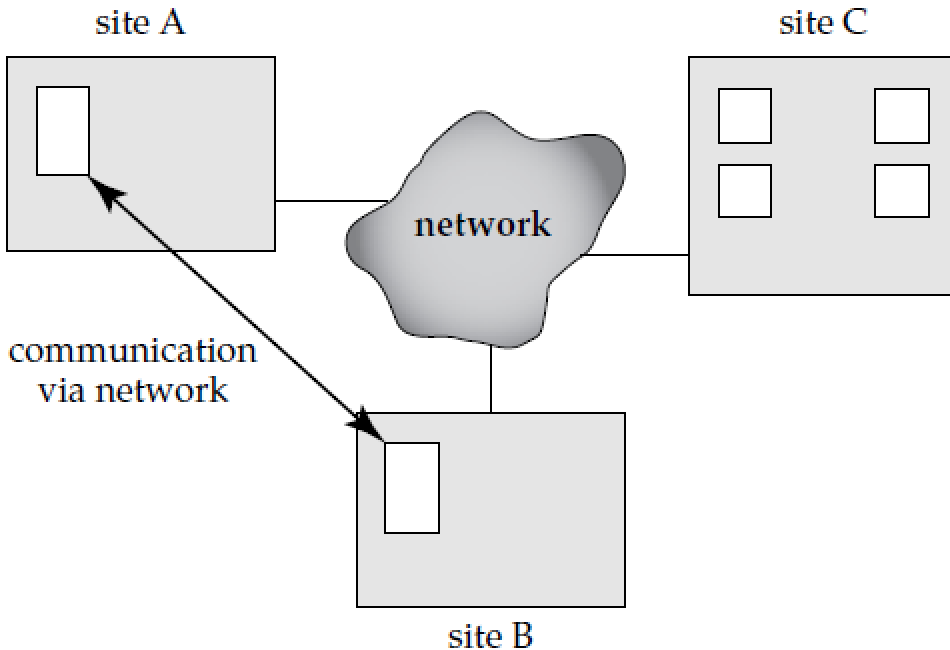
In a distributed database system, the database is stored on several computers. The computers in a distributed system communicate with one another through various communication media, such as high-speed networks or telephone lines. They do not share main memory or disks. The computers in a distributed system may vary in size and function, ranging from workstations up to mainframe systems.

The computers in a distributed system are referred to by a number of different names, such as sites or nodes, depending on the context in which they are mentioned. We mainly use the term site, to emphasize the physical distribution of these systems. The general structure of a distributed system appears in Figure .

The main differences between shared-nothing parallel databases and distributed databases are that distributed databases are typically geographically separated, are separately administered, and have a slower interconnection. Another major difference is that, in a distributed database system, we differentiate between local and global transactions. A local transaction is one that accesses data only from sites where the transaction was initiated. A global transaction, on the other hand, is one that either accesses data in a site different from the one at which the transaction was initiated, or accesses data in several different sites.

There are several reasons for building distributed database systems, including sharing of data, autonomy, and availability.

- Sharing data. The major advantage in building a distributed database system is the provision of an environment where users at one site may be able to access the data residing at other sites. For instance, in a distributed banking system, where each branch stores data related to that branch, it is possible for a user in one branch to access data in another branch. Without this capability, a user wishing to transfer funds from one branch to another would have to resort to some external mechanism that would couple existing systems.
- Autonomy. The primary advantage of sharing data by means of data distribution is that each site is able to retain a degree of control over data that are stored locally. In a centralized system, the database administrator of the central site controls the database. In a distributed system, there is a global database administrator responsible for the entire system. A part of these responsibilities is delegated to the local database administrator for each site. Depending on the design of the distributed database system, each administrator may have a different degree of local autonomy. The possibility of local autonomy is often a major advantage of distributed databases.



A distributed system.

- Availability. If one site fails in a distributed system, the remaining sites may be able to continue operating. In particular, if data items are replicated in several sites, a transaction needing a particular data item may find that item in any of several sites. Thus, the failure of a site does not necessarily imply the shutdown of the system.

The failure of one site must be detected by the system, and appropriate action may be needed to recover from the failure. The system must no longer use the services of the failed site. Finally, when the failed site recovers or is repaired, mechanisms must be available to integrate it smoothly back into the system.

Although recovery from failure is more complex in distributed systems than in centralized systems, the ability of most of the system to continue to operate despite the failure of one site results in increased availability. Availability is crucial for database systems used for real-time applications. Loss of access to data by, for example, an airline may result in the loss of potential ticket buyers to competitors.

An Example of a Distributed Database

Consider a banking system consisting of four branches in four different cities. Each branch has its own computer, with a database of all the accounts maintained at that branch. Each such installation is thus a site. There also exists one single site that maintains information about all the branches of the bank. Each branch maintains (among others) a relation account(Account-schema),

where Account-schema = (account-number, branch-name, balance)

The site containing information about all the branches of the bank maintains the relation branch(Branch-schema), where

Branch-schema = (branch-name, branch-city, assets)

There are other relations maintained at the various sites; we ignore them for the purpose of our example.

To illustrate the difference between the two types of transactions local and global at the sites, consider a transaction to add $50 to account number A-177 located at the Valley view branch. If the transaction was initiated at the Valley view branch, then it is considered local; otherwise, it is considered global. A transaction to transfer $50 from account A-177 to account A-305, which is located at the Hillside branch, is a global transaction, since accounts in two different sites are accessed as a result of its execution.

In an ideal distributed database system, the sites would share a common global schema (although some relations may be stored only at some sites), all sites would run the same distributed database-management software, and the sites would be aware of each other's existence. If a distributed database is built from scratch, it would indeed be possible to achieve the above goals. However, in reality a distributed database has to be constructed by linking together multiple already-

existing database systems, each with its own schema and possibly running different database management software. Such systems are sometimes called multi database systems or heterogeneous distributed database systems. where we show how to achieve a degree of global control despite the heterogeneity of the component systems.

Implementation Issues

Atomicity of transactions is an important issue in building a distributed database system. If a transaction runs across two sites, unless the system designers are careful, it may commit at one site and abort at another, leading to an inconsistent state. Transaction commit protocols ensure such a situation cannot arise. The two-phase commit protocol (2PC) is the most widely used of these protocols.

The basic idea behind 2PC is for each site to execute the transaction till just before commit, and then leave the commit decision to a single coordinator site; the transaction is said to be in the ready state at a site at this point. The coordinator decides to commit the transaction only if the transaction reaches the ready state at every site where it executed; otherwise (for example, if the transaction aborts at any site), the coordinator decides to abort the transaction. Every site where the transaction executed must follow the decision of the coordinator. If a site fails when a transaction is in ready state, when the site recovers from failure it should be in a position to either commit or abort the transaction, depending on the decision of the coordinator.

Concurrency control is another issue in a distributed database. Since a transaction may access data items at several sites, transaction managers at several sites may need to coordinate to implement concurrency control. If locking is used (as is almost always the case in practice), locking can be performed locally at the sites containing accessed data items, but there is also a possibility of deadlock involving

transactions originating at multiple sites. Therefore deadlock detection needs to be carried out across multiple sites. Failures are more common in distributed systems since not only may computers fail, but communication links may also fail. Replication of data items, which is the key to the continued functioning of distributed databases when failures occur, further complicates concurrency control.

The standard transaction models, based on multiple actions carried out by a single program unit, are often inappropriate for carrying out tasks that cross the boundaries of databases that cannot or will not cooperate to implement protocols such as 2PC.

Alternative approaches, based on persistent messaging for communication, are generally used for such tasks. When the tasks to be carried out are complex, involving multiple databases and/or multiple interactions with humans, coordination of the tasks and ensuring transaction properties for the tasks become more complicated. Workflow management systems are systems designed to help with carrying out such tasks.

In case an organization has to choose between a distributed architecture and a centralized architecture for implementing an application, the system architect must balance the advantages against the disadvantages of distribution of data.We have already seen the advantages of using distributed databases. The primary disadvantage of distributed database systems is the added complexity required to ensure proper coordination among the sites. This increased complexity takes various forms:

- Software-development cost. It is more difficult to implement a distributed database system; thus, it is more costly.

- Greater potential for bugs. Since the sites that constitute the distributed system operate in parallel, it is harder to ensure the correctness of algorithms, especially operation during failures of part of the system, and recovery from failures. The potential exists for extremely subtle bugs.
- Increased processing overhead. The exchange of messages and the additional computation required to achieve inter site coordination are a form of overhead that does not arise in centralized systems.

There are several approaches to distributed database design, ranging from fully distributed designs to ones that include a large degree of centralization.

Network systems

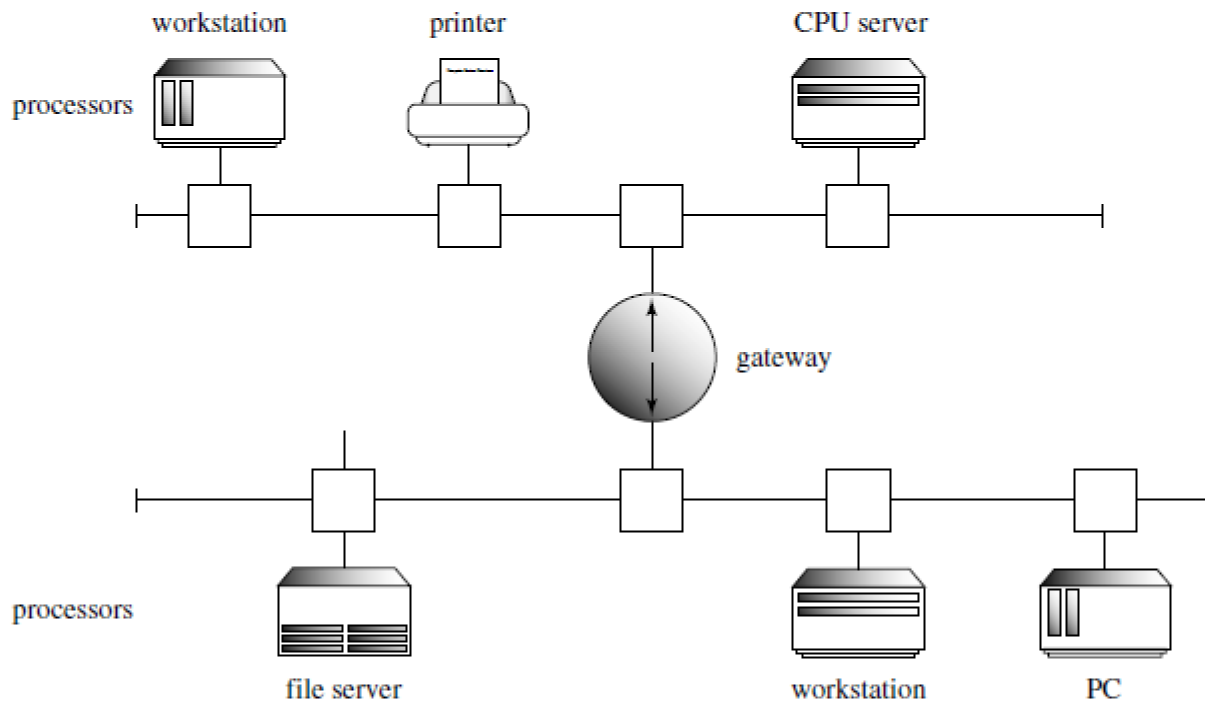
Distributed databases and client–server systems are built around communication networks. There are basically two types of networks: local-area networks and wide area networks. The main difference between the two is the way in which they are distributed geographically. In local-area networks, processors are distributed over small geographical areas, such as a single building or a number of adjacent buildings.

In wide-area networks, on the other hand, a number of autonomous processors are distributed over a large geographical area (such as the United States or the entire world). These differences imply major variations in the speed and reliability of the communication network, and are reflected in the distributed operating-system design.

Local-Area Networks

Local-area networks (LANs) (Figure) emerged in the early 1970s as a way for computers to communicate and to share data with one another. People recognized

that, for many enterprises, numerous small computers, each with its own self contained applications, are more economical than a single large system. Because each small computer is likely to need access to a full complement of peripheral devices (such as disks and printers), and because some form of data sharing is likely to occur in a single enterprise, it was a natural step to connect these small systems into a network.



Local-area network.

LANs are generally used in an office environment. All the sites in such systems are close to one another, so the communication links tend to have a higher speed and lower error rate than do their counterparts in wide-area networks. The most common links in a local-area network are twisted pair, coaxial cable, fiber optics, and, increasingly, wireless connections. Communication speeds range from a few megabits per second (for wireless local-area networks), to 1 gigabit per second for Gigabit Ethernet.

Standard Ethernet runs at 10 megabits per second, while Fast Ethernet run at 100 megabits per second. A storage-area network (SAN) is a special type of high-speed local-area network designed to connect large banks of storage devices (disks) to computers that use the data. Thus storage-area networks help build large-scale shared-disk systems. The motivation for using storage-area networks to connect multiple computers to large banks of storage devices is essentially the same as that for shared-disk databases, namely

- Scalability by adding more computers
- High availability, since data is still accessible even if a computer fails RAID organizations are used in the storage devices to ensure high availability of the data, permitting processing to continue even if individual disks fail. Storage area networks are usually built with redundancy, such as multiple paths between nodes, so if a component such as a link or a connection to the network fails, the network continues to function.

Wide-Area Networks

Wide-area networks (WANs) emerged in the late 1960s, mainly as an academic research project to provide efficient communication among sites, allowing hardware and software to be shared conveniently and economically by a wide community of users. Systems that allowed remote terminals to be connected to a central computer via telephone lines were developed in the early 1960s, but they were not true WANs.

The first WAN to be designed and developed was the Arpanet. Work on the Arpanet began in 1968. The Arpanet has grown from a four-site experimental network to a worldwide network of networks, the Internet, comprising hundreds of millions of computer systems. Typical links on the Internet are fiber-optic lines and, sometimes, satellite channels. Data rates for wide-area links typically range

from a few megabits per second to hundreds of gigabits per second. The last link, to end user sites, is often based on digital subscriber loop (DSL) technology supporting a few megabits per second), or cable modem (supporting 10 megabits per second), or dial-up modem connections over phone lines (supporting up to 56 kilobits per second).

WANs can be classified into two types:

- In discontinuous connection WANs, such as those based on wireless connections, hosts are connected to the network only part of the time.
- In continuous connection WANs, such as the wired Internet, hosts are connected to the network at all times.