Dr. N. SHAKEELA

Guest Lecturer

School of Computer Science, Engineering & Applications

Bharathidasan University

Trichy-24

Database-System Architectures

The architecture of a database system is greatly influenced by the underlying computer system on which it runs, in particular by such aspects of computer architecture as networking, parallelism, and distribution:

- Networking of computers allows some tasks to be executed on a server system and some tasks to be executed on client systems. This division of work has led to *client–server database systems*.
- Parallel processing within a computer system allows database-system activities to be speeded up, allowing faster response to transactions, as well as more transactions per second. The need for parallel query processing has led to *parallel database systems*.
- Distributing data across sites in an organization allows those data to reside where they are generated or most needed, but still to be accessible from other sites and from other departments. Keeping multiple copies of the database across different sites also allows large organizations to continue their database operations even when one site is affected by a natural disaster, such as flood, fire, or earthquake. *Distributed database systems* handle geographically or administratively distributed data spread across multiple database systems.

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

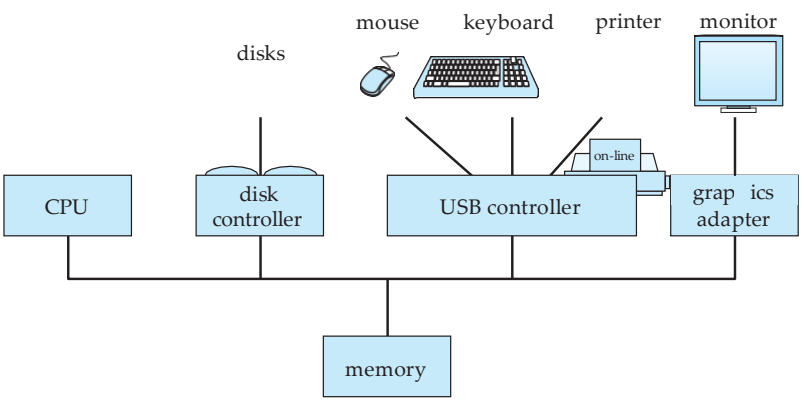
– server systems, on the other hand, have functionality split between a server system and multiple client systems.

Centralized Systems

A modern, general-purpose computer system consists of one to a few processors and a number of device controllers that are connected through a common bus that provides access to shared memory (Figure 17.1). The processors have local cache memories that store local copies of parts of the memory, to speed up access to data. . Cache memory reduces the contention for memory access, since it reduces the number of times that the processor 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 processor 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 and may have multiple processors. It serves a large number of users who are connected to the system remotely.

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. In contrast,



A centralized computer system.

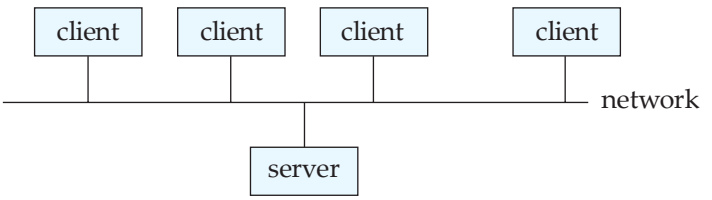
database systems designed for multiuser systems support the full transactional features that we have studied earlier.

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.

Client–Server Systems

As personal computers became faster, more powerful, and cheaper, there was a shift away from the centralized system architecture. 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 17.2 shows the general structure of a client–server system.

Functionality provided by database systems can be broadly divided into two parts — the front end and the back end. 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 the SQL user interface, forms interfaces, report generation tools, and data mining and analysis tools (see Figure 17.3). 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.

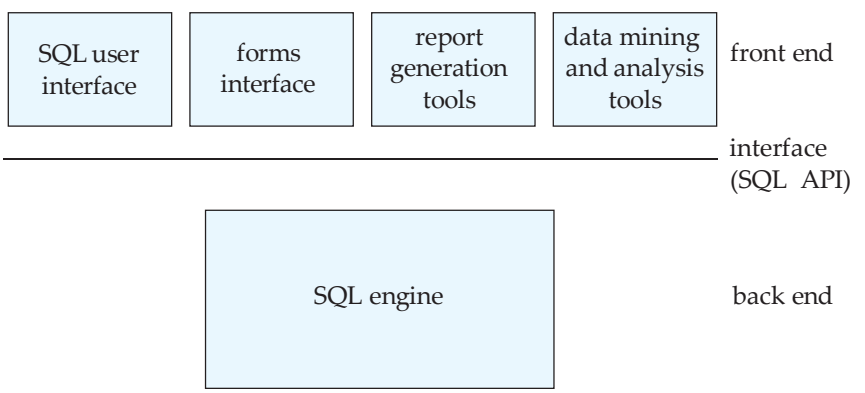
Standards such as *ODBC* and *JDBC*, were developed to interface clients with servers. Any client that uses the ODBC or JDBC interface can connect to any server that provides the interface.

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.

Server System Architectures

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



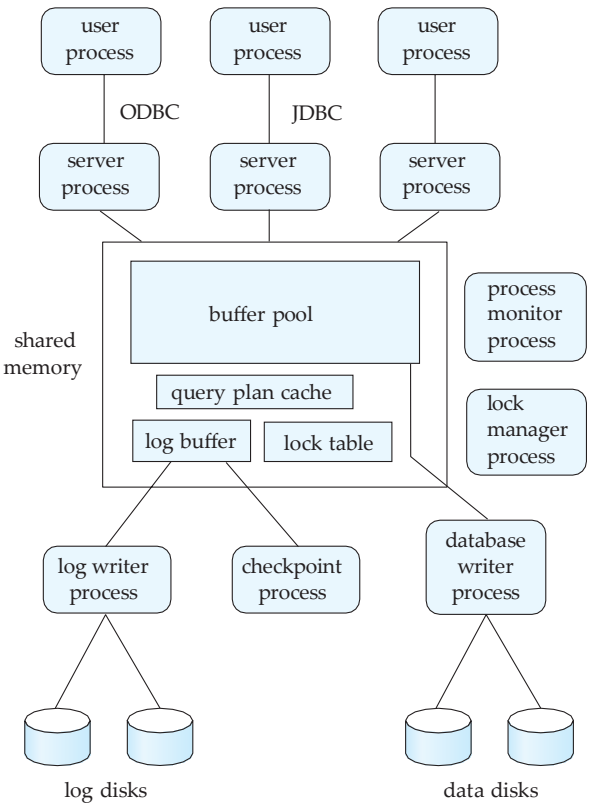
Front-end and back-end functionality.

- **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.

Transaction Servers

A typical transaction-server system today consists of multiple processes accessing data in shared memory, as in Figure 17.4. The processes that form part of the database system include:

- **Server processes**: These are processes that receive user queries (transactions), execute them, and send the results back. The queries may be submitted to the server processes from a user interface, or from a user process running embedded SQL, or via JDBC, ODBC, or similar protocols. Some database systems use a separate process for each user session, and a few use a single database process for all user sessions, but with multiple threads so that multiple queries can execute concurrently. (A **thread** is like a process, but multiple threads execute as part of the same process, and all threads within a process run in the same virtual-memory space. Multiple threads within a process can execute concurrently.).
- **Lock manager process**: This process implements lock manager functionality, which includes lock grant, lock release, and deadlock detection.
- **Database writer process**: There are one or more processes that output modified buffer blocks back to disk on a continuous basis.
- **Log writer process**: This process outputs log records from the log record buffer to stable storage. Server processes simply add log records to the log



Shared memory and process structure.

record buffer in shared memory, and if a log force is required, they request the log writer process to output log records.

- **Checkpoint process**: This process performs periodic checkpoints.
- **Process monitor process**: This process monitors other processes, and if any of them fails, it takes recovery actions for the process, such as aborting any transaction being executed by the failed process, and then restarting the process.

The shared memory contains all shared data, such as:

- Buffer pool.
- Lock table.
- Log buffer, containing log records waiting to be output to the log on stable storage.

- Cached query plans, which can be reused if the same query is submitted again.

All database processes can access the data in shared memory. Since multiple processes may read or perform updates on data structures in shared memory, there must be a mechanism to ensure that a data structure is modified by at most one process at a time, and no process is reading a data structure while it is being written by others. Such **mutual exclusion** can be implemented by means of operating system functions called semaphores.

To avoid the overhead of message passing, in many database systems, server processes implement locking by directly updating the lock table (which is in shared memory), instead of sending lock request messages to a lock manager process.

- Since multiple server processes may access shared memory, mutual exclusion must be ensured on the lock table.
- If a lock cannot be obtained immediately because of a lock conflict, the lock request code may monitor the lock table to check when the lock has been granted. The lock release code updates the lock table to note which process has been granted the lock.

  To avoid repeated checks on the lock table, operating system semaphores can be used by the lock request code to wait for a lock grant notification. The lock release code must then use the semaphore mechanism to notify waiting transactions that their locks have been granted.

Even if the system handles lock requests through shared memory, it still uses the lock manager process for deadlock detection.

Data Servers

Data-server systems are used in local-area networks, where there is a high-speed connection between the clients and the server, the client machines are comparable in processing power to the server machine, and the tasks to be executed are computation intensive. In such an environment, it makes sense to ship data to client machines, to perform all processing at the client machine (which may take a while), and then to ship the data back to the server machine. Note that this architecture requires full back-end functionality at the clients. Data-server architectures have been particularly popular in object-oriented database systems
.

Interesting issues arise in such an architecture, since the time cost of communication between the client and the server is high compared to that of a local memory reference :

- **Page shipping** versus **item shipping**. The unit of communication for data can be of coarse granularity, such as a page, or fine granularity, such as a tuple (or an object, in the context of object-oriented database systems). We use the term **item** to refer to both tuples and objects.

  If the unit of communication is a single item, the overhead of message passing is high compared to the amount of data transmitted. Instead, when an item is requested, it makes sense also to send back other items that are likely to be used in the near future. Fetching items even before they are requested is called **prefetching**. Page shipping can be considered a form of prefetching if multiple items reside on a page, since all the items in the page are shipped when a process desires to access a single item in the page.

- **Adaptive lock granularity**. Locks are usually granted by the server for the data items that it ships to the client machines. A disadvantage of page shipping is that client machines may be granted locks of too coarse a granularity — a lock on a page implicitly locks all items contained in the page. Even if the client is not accessing some items in the page, it has implicitly acquired locks on all pre fetched items. Other client machines that require locks on those items may be blocked unnecessarily. Techniques for lock **de-escalation** have been proposed where the server can request its clients to transfer back locks on pre fetched items. If the client machine does not need a pre fetched item, it can transfer locks on the item back to the server, and the locks can then be allocated to other clients.

- **Data caching**. Data that are shipped to a client on behalf of a transaction can be **cached** at the client, even after the transaction completes, if sufficient storage space is available. Successive transactions at the same client may be able to make use of the cached data. However, **cache coherency** is an issue: Even if a transaction finds cached data, it must make sure that those data are up to date, since they may have been updated by a different client after they were cached. Thus, a message must still be exchanged with the server to check validity of the data, and to acquire a lock on the data.

- **Lock caching**. If the use of data is mostly partitioned among the clients, with clients rarely requesting data that are also requested by other clients, locks can also be cached at the client machine. Suppose that a client finds a data item in the cache, and that it also finds the lock required for an access to the data item in the cache. Then, the access can proceed without any communication with the server. However, the server must keep track of cached locks; if a client requests a lock from the server, the server must **call back** all conflicting locks on the data item from any other client machines that have cached the locks. The task becomes more complicated when machine failures are taken into account.

Cloud-Based Servers

Servers are usually owned by the enterprise providing the service, but there is an increasing trend for service providers to rely at least in part upon servers that are owned by a "third party" that is neither the client nor the service provider.

One model for using third-party servers is to outsource the entire service to another company that hosts the service on its own computers using its own software. This allows the service provider to ignore most details of technology and focus on the marketing of the service.

Another model for using third-party servers is **cloud computing**, in which the service provider runs its own software, but runs it on computers provided by another company. Under this model, the third party does not provide any of the application software; it provides only a collection of machines. These machines are not "real" machines, but rather simulated by software that allows a single real computer to simulate several independent computers. Such simulated machines are called **virtual machines**. The service provider runs its software (possibly including a database system) on these virtual machines. A major advantage of cloud computing is that the service provider can add machines as needed to meet demand and release them at times of light load. This can prove to be highly cost-effective in terms of both money and energy.

A third model uses a cloud computing service as a data server; Database applications using cloud-based storage may run on the same cloud (that is, the same set of machines), or on another cloud.

Parallel Systems

Parallel systems improve processing and I/O speeds by using multiple processors and disks in parallel. The driving force behind parallel database systems is the demands of applications that have to query extremely large databases or that have to process an extremely large number 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. Virtually all high-end machines today offer some degree of coarse-grain parallelism: at least two or four processors. 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 processors 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 Scaleup

Two important issues in studying parallelism are speedup and scaleup. 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 **scaleup**.

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 of 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 T_L, and that the execution time of the same task on the smaller machine is T_S. The speedup due to parallelism is defined as T_S / T_L. The parallel system is said to demonstrate **linear speedup** if the speedup is N when the larger system has N times the resources (processors, 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 17.5 illustrates linear and sub linear speedup.

Scaleup 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 Q_N be a task that is N times bigger than Q. Suppose that the execution time of task Q on a given machine

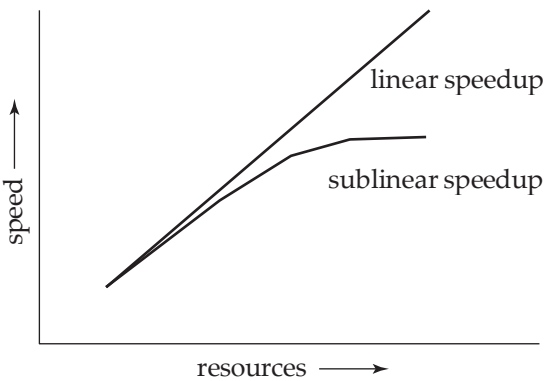


Figure 17.5 Speedup with increasing resources.

Scaleup with increasing problem size and resources.

M_S is T_S, and the execution time of task Q_N on a parallel machine M_L, which is N times larger than M_S, is T_L. The scaleup is then defined as T_S / T_L. The parallel system M_L is said to demonstrate **linear scaleup** on task Q if $T_L = T_S$. If $T_L > T_S$, the system is said to demonstrate **sublinear scaleup**. There are two kinds of scaleup that are relevant in parallel database systems, depending on how the size of the task is measured:

- In **batch scaleup**, 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.
- In **transaction scaleup**, This kind of scaleup is what is relevant in transaction- 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.

Scaleup 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 .

How- ever, we must also look at absolute performance numbers when using scaleup measures; a machine that scales up linearly may perform worse than a machine that scales less than linearly.

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

- **Start-up costs**. There is a start-up cost associated with initiating a single process. In a parallel operation consisting of thousands of processes, the *start-up 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 scaleup 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*..

Parallel Database Architectures

There are several architectural models for parallel machines. Among the most prominent ones are those in Figure 17.8 (in the figure, M denotes memory, P denotes a processor, and disks are shown as cylinders):

- **Shared memory**. All the processors share a common memory (Figure 17.8a).
- **Shared disk**. All the processors share a common set of disks (Figure 17.8b). Shared-disk systems are sometimes called **clusters**.
- **Shared nothing**. The processors share neither a common memory nor common disk (Figure 17.8c).
- **Hierarchical**. This model is a hybrid of the preceding three architectures (Figure 17.8d).

.

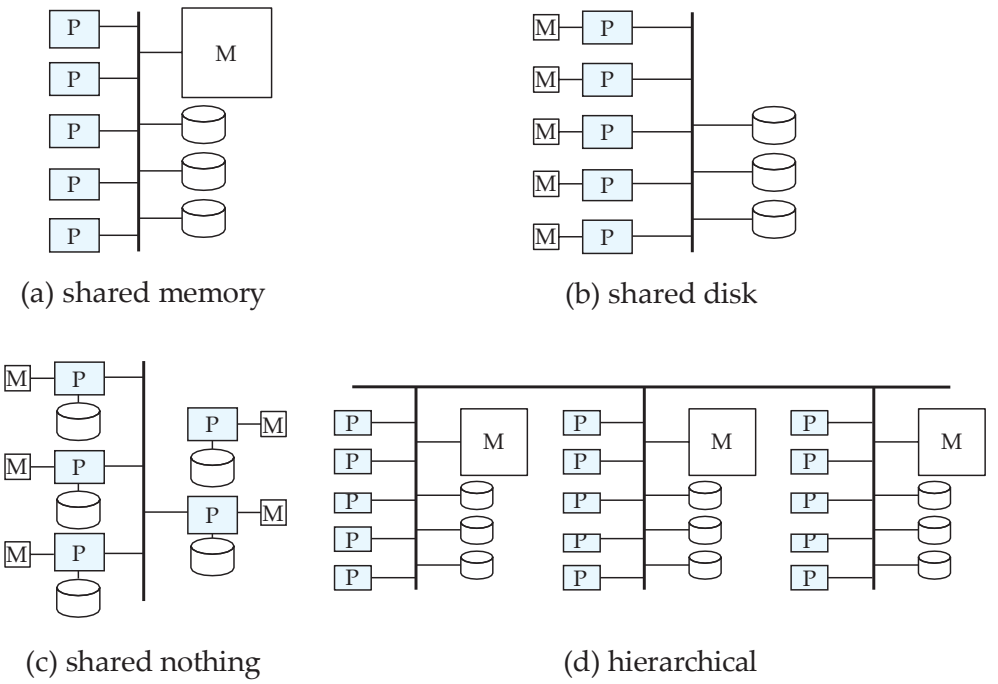


Figure 17.8 Parallel database architectures.

Shared Memory

In a **shared-memory** architecture, the processors and disks have access to a common memory, typically via a bus or through an interconnection network. The benefit of shared memory is extremely efficient communication between processors — data in shared memory can be accessed by any processor without being moved with software. A processor can send messages to other processors much faster by using memory writes (which usually take less than a microsecond) than by sending a message through a communication mechanism. The downside of shared-memory machines is that the architecture is not scalable .Adding more processors does not help after a point, since the processors will spend most of their time waiting for their turn on the bus to access memory.

Shared-memory architectures usually have large memory caches at each processor, so that referencing of the shared memory is avoided whenever possible.

However, at least some of the data will not be in the cache, and accesses will have to go to the shared memory. Moreover, the caches need to be kept coherent; that is, if a processor performs a write to a memory location, the data in that memory location should be either updated at or removed from any processor where the data are cached. Maintaining cache coherency becomes an increasing overhead with increasing numbers of processors. Consequently, shared-memory machines are not capable of scaling up beyond a point;

Shared Disk

In the **shared-disk** model, all processors can access all disks directly via an inter connection network, but the processors have private memories. There are two advantages of this architecture over a shared-memory architecture. First, since each processor has its own memory, the memory bus is not a bottleneck. Second, it offers a cheap way to provide a degree of **fault tolerance**: If a processor (or its memory) fails, the other processors can take over its tasks, since the database is resident on disks that are accessible from all processors. We can make the disk subsystem itself fault tolerant by using a RAID architecture. The shared-disk architecture has found acceptance in many applications.

The main problem with a shared-disk system is again scalability. Although the memory bus is no longer a bottleneck, the interconnection to the disk subsystem is now a bottleneck; it is particularly so in a situation where the database makes a large number of accesses to disks. Compared to shared-memory systems, shared-disk systems can scale to a somewhat larger number of processors, but communication across processors is slower

Shared Nothing

In a **shared-nothing** system, each node of the machine consists of a processor, memory, and one or more disks. The processors at one node may communicate with another processor at another node by a high-speed interconnection network. A node functions as the server for the data on the disk or disks that the node owns. Since local disk references are serviced by local disks at each processor, the shared-nothing model overcomes the disadvantage of requiring all I/O to go through a single interconnection network. Moreover, the interconnection networks for shared-nothing systems are usually designed to be scalable, so that their transmission capacity increases as more nodes are added. Consequently, shared-nothing architectures are more scalable and can easily support a large number of processors. The main drawbacks of shared-nothing systems are the costs of communication and of nonlocal disk access, which are higher than in a shared-memory or shared-disk architecture since sending data involves software interaction at both ends.

Hierarchical

The **hierarchical architecture** combines the characteristics of shared-memory, shared-disk, and shared-nothing architectures. At the top level, the system consists of nodes that are connected by an interconnection network and do not share disks or memory with one another. Thus, the top level is a shared-nothing architecture. Each node of the system could actually be a shared-memory system with a few processors. Alternatively, each node could be a shared-disk system, and each of the systems sharing a set of disks could be a shared-memory system. Thus, a system could be built as a hierarchy, with shared-memory architecture with a few processors at the base, and a shared-nothing architecture at the top, with possibly a shared-disk architecture in the middle. Figure 17.8d illustrates a hierarchical architecture with shared-memory nodes connected together in a shared-nothing architecture. Commercial parallel database systems today run on several of these architectures.

Attempts to reduce the complexity of programming such systems have yielded **distributed virtual-memory** architectures, where logically there is a single shared memory, but physically there are multiple disjoint memory systems; the virtual-memory-mapping hardware, coupled with system software, allows each processor to view the disjoint memories as a single virtual memory. Since access speeds differ, depending on whether the page is available locally or not, such an architecture is also referred to as a **non uniform memory architecture** (NUMA).

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 private networks or the Internet. 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.

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. An**other 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.

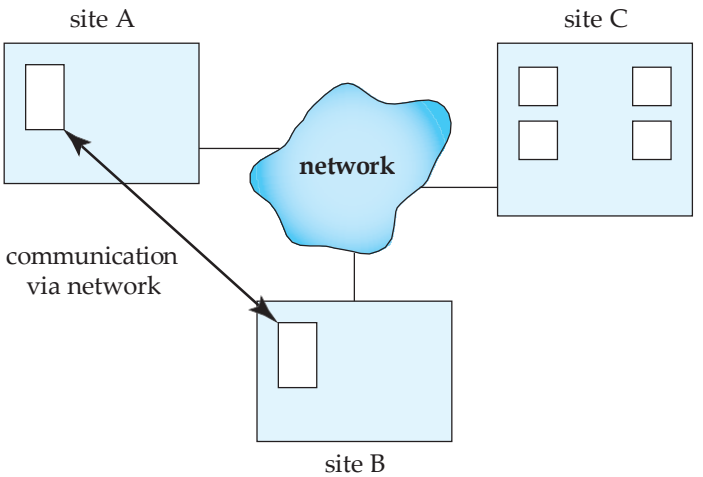


Figure 17.9 A distributed system.

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 **university system, where each campus stores data related to that campus**, it is possible for a user in one campus to access data in another campus. Without this capability, the transfer of student records from one campus to another campus 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.
- **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.

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. There also exists one single site that maintains information about all the branches of the bank.

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 Valleyview branch. If the transaction was initiated at the Valleyview 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 **multidatabase systems** or **heterogeneous distributed database 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 until it enters the partially committed state, 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, 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.

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,

- **Increased processing overhead**. The exchange of messages and the additional computation required to achieve intersite coordination are a form of overhead that does not arise in centralized systems.

Network Types

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

Local-Area Networks

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.

A **storage-area network** (**SAN**) is a special type of high-speed local-area net- work 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 are still accessible even if a computer fails.

Wide-Area Networks

In addition to limits on data rates, communication in a WAN must also contend with significant **latency**: a message may take up to a few hundred milliseconds to be delivered across the world, both due to speed of light delays, and due to queuing delays at a number of routers in the path of the message. Applications whose data and computing resources are distributed geographically have to be carefully designed to ensure latency does not affect system performance excessively.

WANs can be classified into two types:

- In **discontinuous connection** WANs, such as those based on mobile 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.

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 private networks or the Internet. 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.

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. An**other 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.

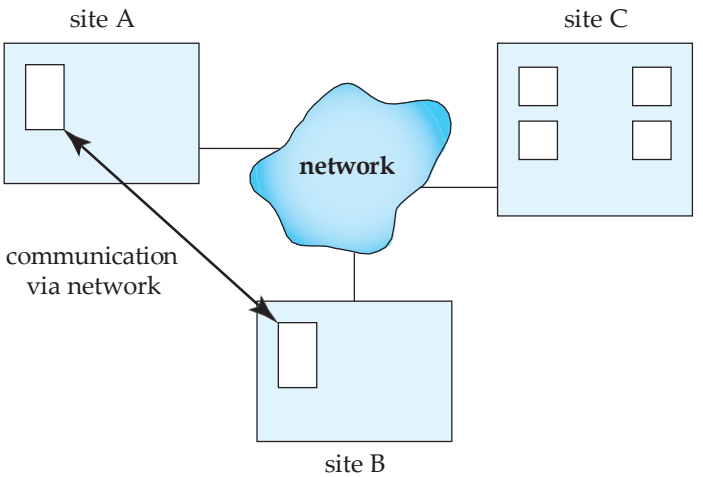


Figure 17.9 A distributed system.

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 **university system, where each campus stores data related to that campus**, it is possible for a user in one campus to access data in another campus. Without this capability, the transfer of student records from one campus to another campus 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.
- **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.

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. There also exists one single site that maintains information about all the branches of the bank.

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 Valleyview branch. If the transaction was initiated at the Valleyview 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 **multidatabase systems** or **heterogeneous distributed database 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 until it enters the partially committed state, 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, 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.

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,

- **Increased processing overhead**. The exchange of messages and the additional computation required to achieve intersite coordination are a form of overhead that does not arise in centralized systems.

Network Types

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

Local-Area Networks

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.

A **storage-area network** (**SAN**) is a special type of high-speed local-area net- work 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.

Wide-Area Networks

In addition to limits on data rates, communication in a WAN must also contend with significant **latency**: a message may take up to a few hundred milliseconds to be delivered across the world, both due to speed of light delays, and due to queuing delays at a number of routers in the path of the message. Applications whose data and computing resources are distributed geographically have to be carefully designed to ensure latency does not affect system performance excessively.

WANs can be classified into two types:

- In **discontinuous connection** WANs, such as those based on mobile 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.