COP 4710: Database Systems
Spring 2006

CHAPTER 22 – Parallel and Distributed Database Systems – Part 1

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790
http://www.cs.ucf.edu/courses/cop4710/spr2006

School of Electrical Engineering and Computer Science
University of Central Florida

Introduction to Parallel and Distributed Database Systems

- So far in this course, we have considered centralized DBMSs in which all of the data is maintained at a single site. We further assumed that processing individual transactions was essentially sequential.
- One of the most important trends in databases is the increased use of parallel evaluation techniques (parallel DBMS) and data distribution (distributed DBMS).
- We will focus primarily on distributed database management systems, but we will examine some parallel query execution strategies.

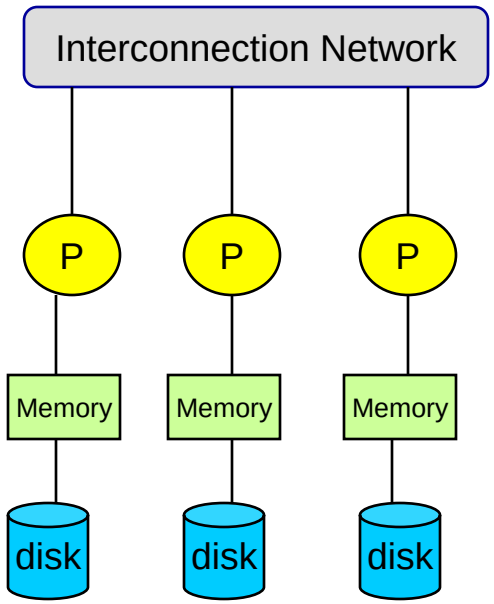
Parallel Database Systems

- A parallel database system seeks to improve performance of the database through the parallelization of various operations of the DBMS.
- Parallelization can occur:
  - in the loading of data
  - building/searching indices
  - query evaluation
- Although it is common for data to be distributed in such a system, the distribution is governed solely by performance considerations.

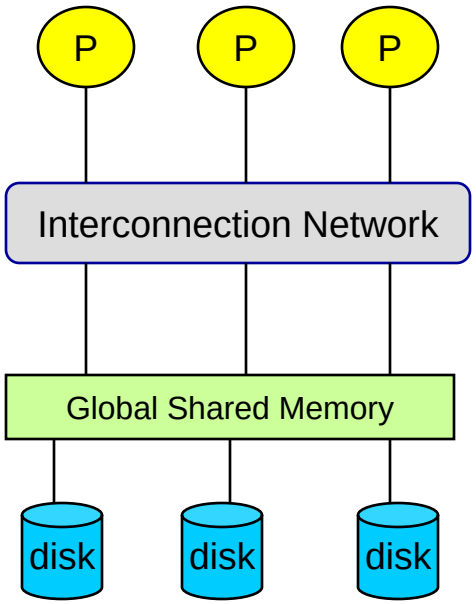
Parallel Database System Architectures

- Three main architectures have been proposed for building parallel DBMSs.
- In a shared-memory system, multiple CPUs are attached to an interconnection network and can access a common region of main memory.
- In a shared-disk system, each CPU has a private memory and direct access to all disks through an interconnection network.
- In a shared-nothing system, each CPU has local main memory and disk space, but no two CPUs can access the same storage area; all communication between CPUs is through a network connection.

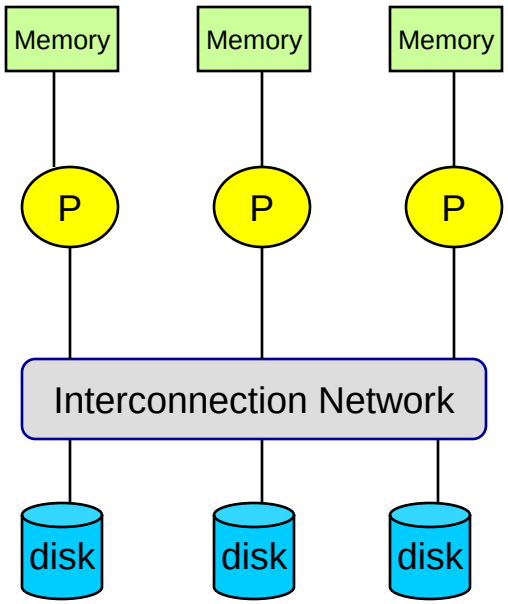
Parallel Database System Architectures (cont.)



Shared Nothing



Shared Memory



Shared Disk

The best architecture for parallel DBMSs

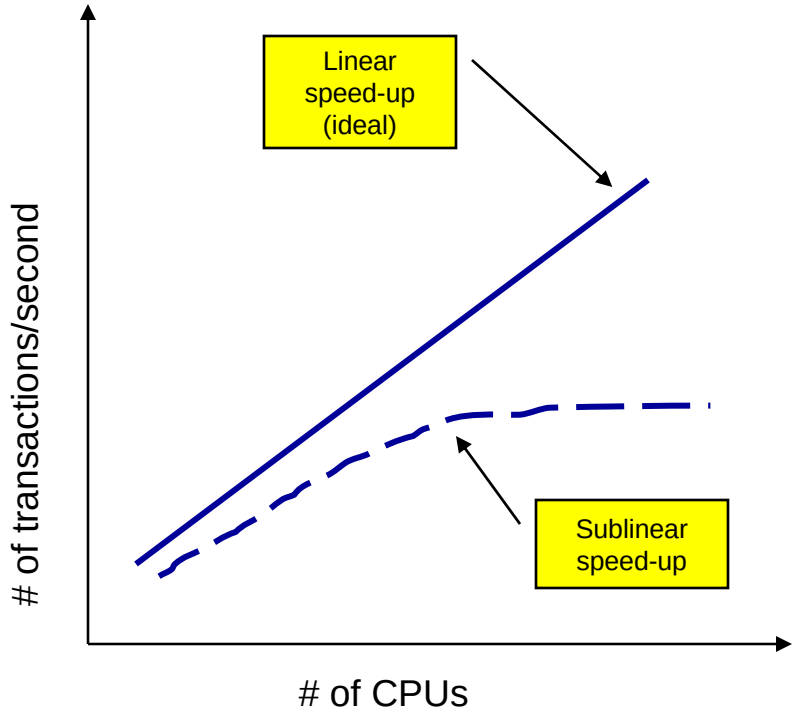
Parallel Database System Architectures (cont.)

- The basic problem with the shared-memory and shared-disk architectures is interference.
- As more CPUs are added, existing CPUs are slowed down because of the increased contention for memory accesses and network bandwidth.
- It has been shown that:
  - An average of 1% slowdown per additional CPU limits the maximum speed-up to a factor of 37.
  - Adding additional CPUs actually slows down the system.
  - A system with 1000 CPUs is only 4% as effective as a single CPU.
- These observations motivated the development of the shared-nothing architecture for large parallel database systems.

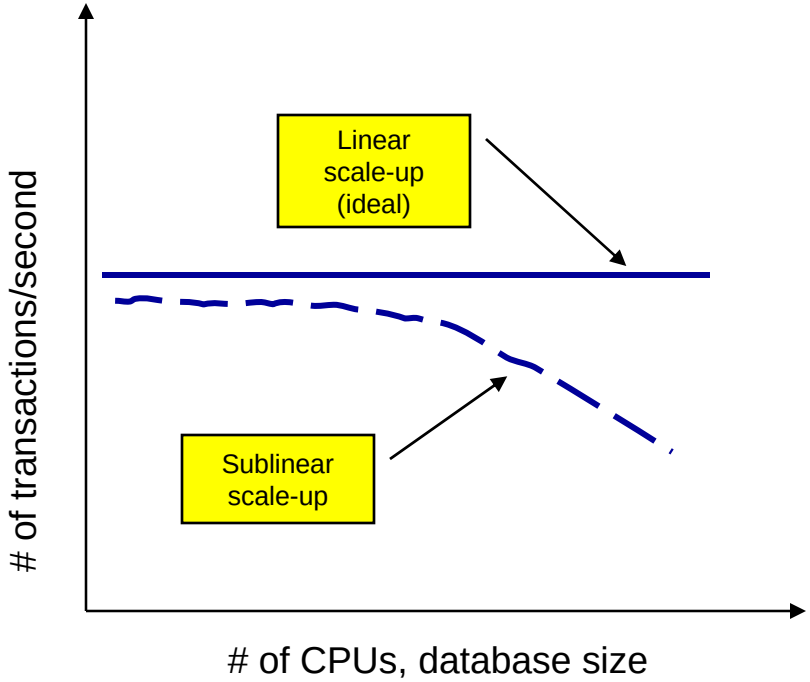
Parallel Database System Architectures (cont.)

- The shared-nothing architecture requires more extensive reorganization of the DBMS code, but it has been shown to provide a linear speed-up and linear scale-up.
- Linear speed-up occurs when the time required by an operation decreases in proportion to the increase in the number of CPUs and disks.
- Linear scale-up occurs when the performance level is sustained if the number of CPUs and disks are increased in proportion to the amount of data.
- As a result, ever-more-powerful parallel database systems can be constructed by taking advantage of the rapidly improving performance for single-CPU systems and connecting as many CPUs as desired.

Parallel Database System Architectures (cont.)



Speed-up



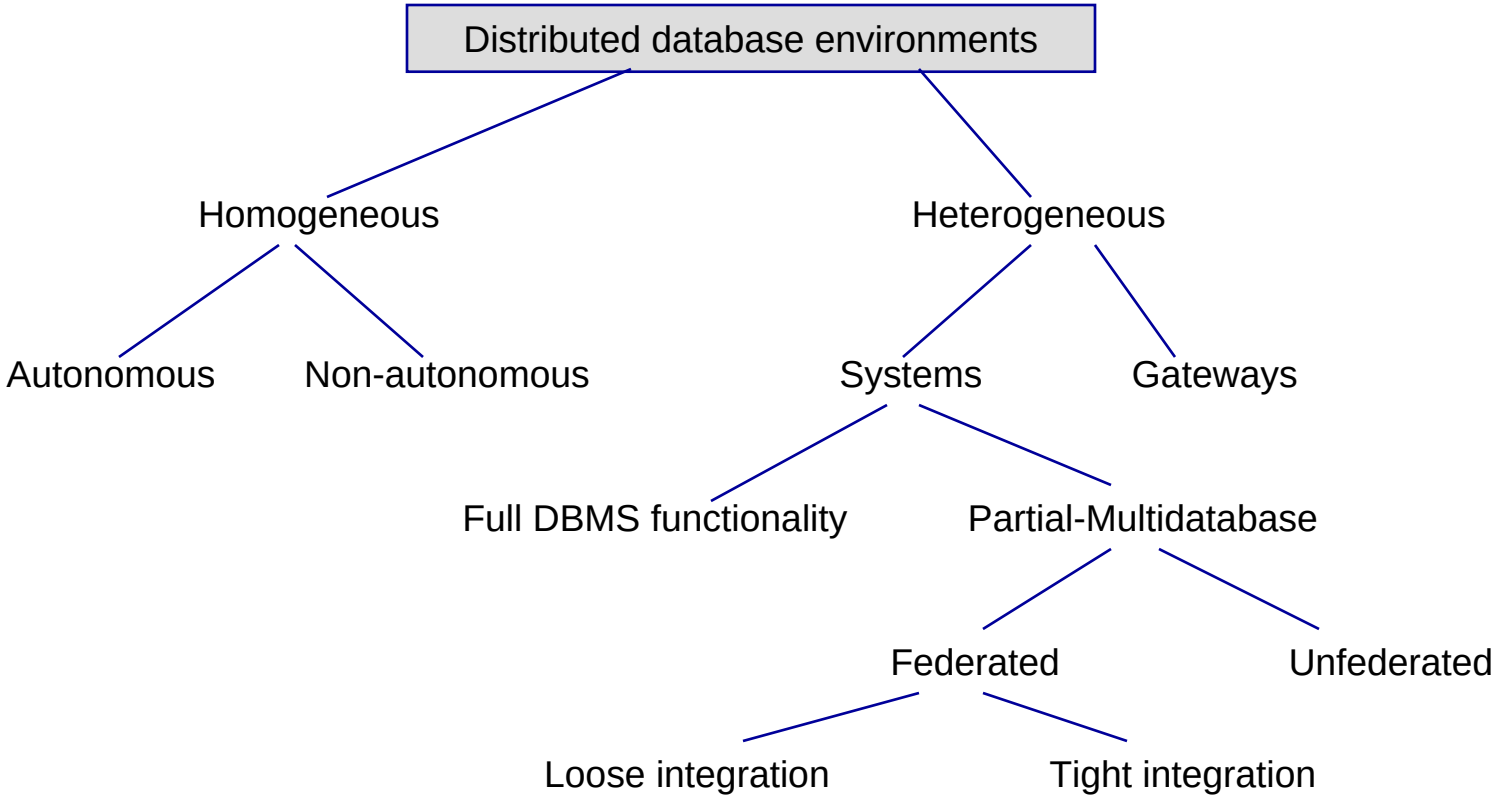
Scale-up

Distributed Database Systems

- In a distributed database system, data is physically stored across several sites, and each site is typically managed by a DBMS capable of running independent of the other sites.
- The location of the data items and the degree of autonomy of the individual sites have a significant impact on all aspects of the system, including query processing and optimization, concurrency control, and recovery.
- In contrast to parallel database systems, the distribution of data is governed by factors such as local ownership and increased availability, in addition to performance related issues.

Distributed Database Systems (cont.)

- Distributed database systems have been around since the mid-1980s. As you might expect, a variety of distributed database options exist. The diagram below shows the basic distributed database environments.



Distributed Database Systems (cont.)

Homogeneous – same DBMS is used at each site.

- Autonomous – each DBMS works independently, passing messages back and forth to share data updates.
- Nonautonomous – a central, or master, DBMS coordinates database access and updates across the sites.

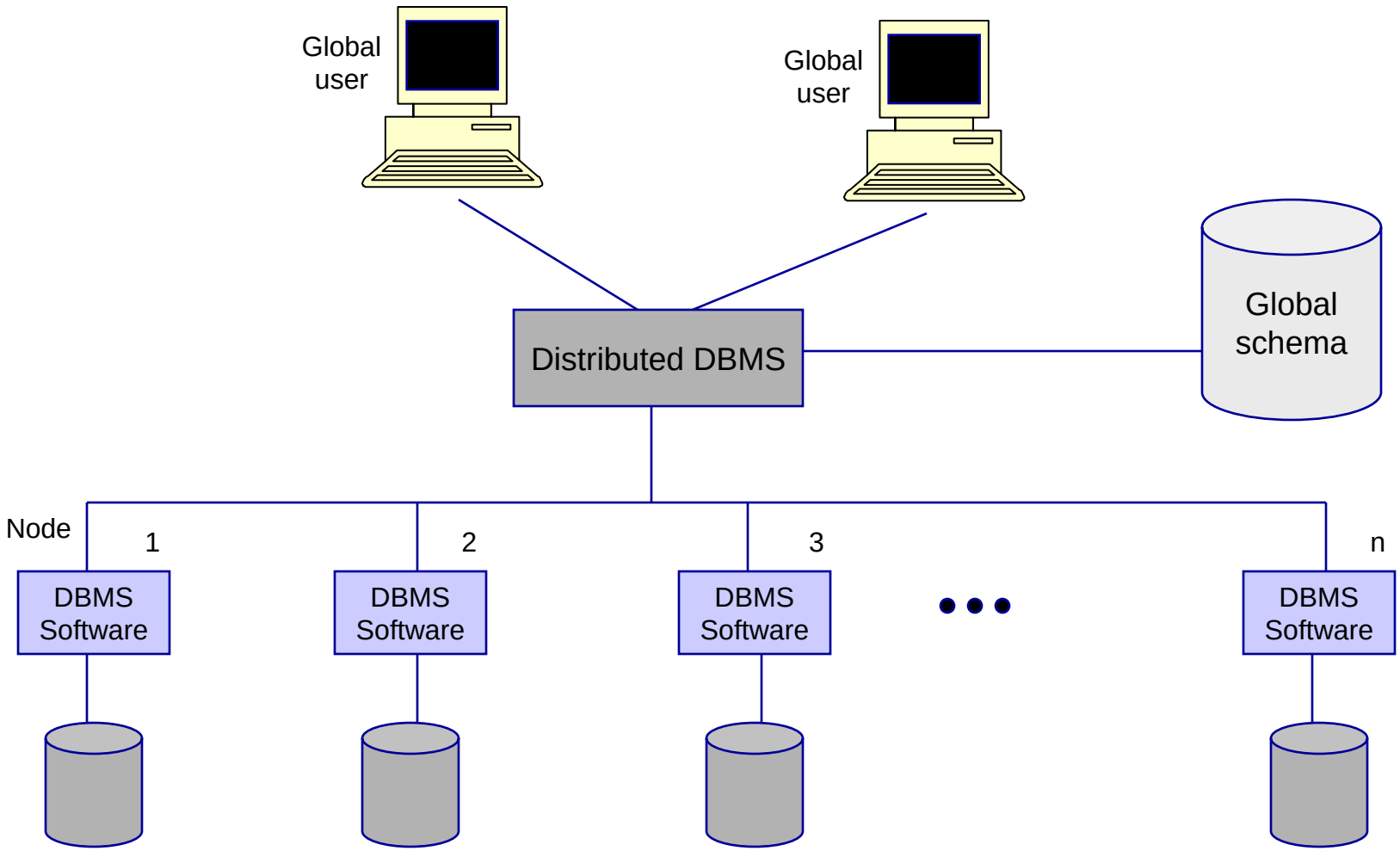
Heterogeneous – potentially different DBMSs are used at each site.

- Systems – support some or all of the functionality of one logical database.
  - Full DBMS functionality – supports all of the functionality of a distributed database.
  - Partial-Multidatabase – supports some of the features of a distributed database.
    - Federated – supports local databases for unique data requests.
      - » Loose integration – many schemas exist: each local database and each local DBMS must communicate with all local schemas.
      - » Tight integration – one global schema exists that defines all the data across all local databases.
    - Unfederated – requires all access to go through a central coordinating module.
- Gateways – simple paths are created to other databases, without the benefits of one logical database.

A Homogeneous Distributed Database

- A typical homogeneous distributed database environment is illustrated on the following page.
- This environment is typically defined by the following characteristics:
  - Data are distributed across all the nodes.
  - The same DBMS is used at each location.
  - All data are managed by the distributed DBMS. There is no exclusively local data.
  - All users access the database through one global schema or database definition.
  - The global schema is simply the union of all the local database schemas.

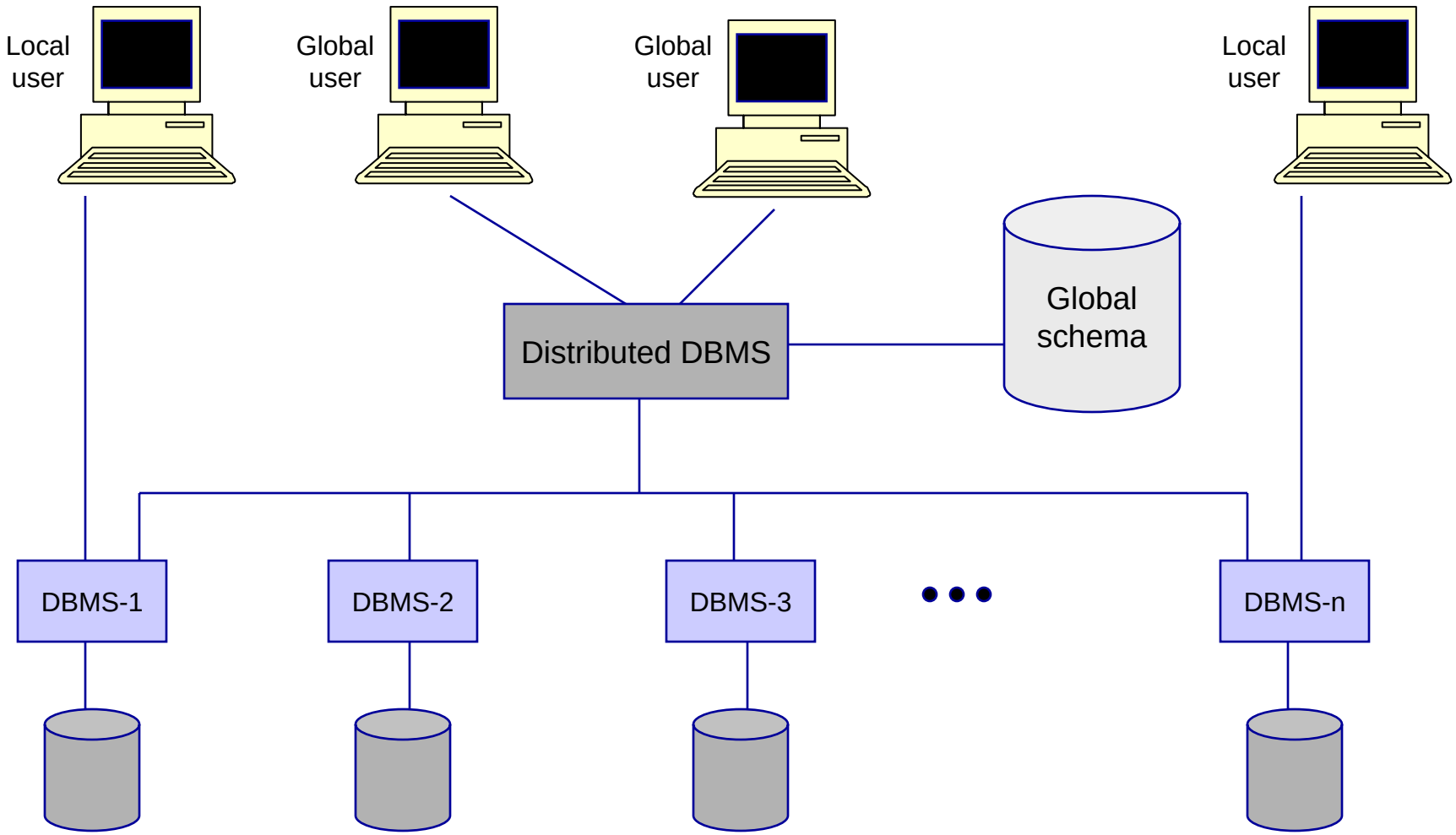
A Homogeneous Distributed Database System



A Heterogeneous Distributed Database

- It is difficult in most organizations to force a homogeneous environment, yet heterogeneous environments are much more difficult to manage.
- As the diagram on page 10 illustrates, there are many variations of heterogeneous distributed database environments, however; a typical heterogeneous distributed database environment is defined by the following characteristics:
  - Data are distributed across all the nodes.
  - Different DBMSs may used at each location.
  - Some users require only local access to databases, which can be accomplished using only the local DBMS and schema.
  - A global schema exists, which allows local users to access remote data.

A Heterogeneous Distributed Database System



Objectives of Distributed Database Systems

The fundamental principle of distributed database

To the user, a distributed database system should look exactly like a nondistributed database system.

- The fundamental principle of distributed databases gives rise to a set of twelve fundamental objectives. These objectives were defined by C.J. Date in 1990.

1. Local autonomy
2. No reliance on a central site
3. Continuous operation
4. Location transparency
5. Fragmentation transparency
6. Replication transparency
7. Distributed query processing
8. Distributed transaction management
9. Hardware independence
10. Operating system independence
11. Network transparency
12. DBMS independence

1. Local Autonomy

- The sites in a distributed system should be autonomous to the maximum extent possible (some situations arise where site X must relinquish some control to some other site Y).
- Local autonomy means that all operations at a given site X are controlled by that site: no site X should depend on some site Y for its successful operation, otherwise if site Y is down, then X cannot run even if there is nothing wrong with site X itself.
- Local autonomy implies that local data is locally owned and managed, with local accountability.

2. No Reliance on a Central Site

- Local autonomy implies that all sites must be treated as equals.
- There must not be any reliance on a central "master" site for some central service, such as transaction management or query processing.
- Central "master" sites represent a potential bottleneck but more importantly, if the central site goes down, the whole system would be down.
- Note: if local autonomy is achieved, this objective is automatically satisfied.

3. Continuous Operation

- An advantage of distributed systems in general is that they can provide greater reliability and greater availability.
  - Reliability is the probability that the system is up and running at any given moment. Reliability is improved in distributed systems because they can continue to operate (possibly at some reduced level of performance) when faced with the failure of some individual component, such as an individual site.
  - Availability is the probability that the system is up and running throughout a specified period. As with reliability, distributed systems improve availability partly for the same reason, but also because of data replication.

4. Location Transparency

- The basic idea of location transparency is that users should not have to know where the data is physically stored, but should be able to behave – at least from a logical standpoint – as if the data were all stored at their own local site.
- Location transparency is desirable because it simplifies application programs and end-user activities; in particular, it allows data to migrate from site to site without invalidating any of those programs or activities.
- Location transparency allows data to migrate around the network in response to changing performance requirements.

5. Fragmentation Transparency

- We'll examine fragmentation more closely later, but for now we'll assume that a system that supports data fragmentation allows a database (or its components) to be divided into pieces or fragments for physical storage purposes and that these fragments can be stored at physically different sites.
- Fragmentation transparency allows users to behave – from a logical standpoint – as if the data were not fragmented.
- Fragmentation transparency allows data to be refragmented at any time (and fragments to be redistributed at any time) in response to changing performance requirements.

6. Replication Transparency

- We'll examine replication more closely later, but for now we'll assume that a system that supports data replication allows a database (or its components) to be represented in storage by many distinct copies or replicas, stored at physically different sites.
- Replication transparency allows users to behave – from a logical standpoint – as if the data were not replicated.
- Replication transparency allows replicas to be created or destroyed at any time in response to changing performance requirements.

7. Distributed Query Processing

- In a distributed database system, query processing can involve both local as well as global queries.
- Local queries are executed against only local data while global queries will involve non-local data.
- Query optimization is even more important in a distributed environment than it is in a centralized environment. Since many different possibilities exist for moving data around a network in response to processing a query, it is crucially important that an efficient execution strategy be found.

8. Distributed Transaction Management

- There are two major aspects to transaction management, recovery and concurrency, and both require extended treatment in a distributed environment.
- In a distributed system, a single transaction can involve the execution of code at many sites; in particular it can involve updates at many sites. Each transaction is therefore said to consist of several agents, where an agent is the process performed on behalf of a given transaction at a given site.
- The system must know when two agents are part of the same transactions; for example, two agents of the same transaction must obviously not be allowed to deadlock with each other!

8. Distributed Transaction Management (cont.)

On the recovery side

- In order to ensure that a given transaction is atomic (all or nothing) in the distributed environment, the system must ensure that the set of agents for a given transaction either all commit in unison or all roll back in unison.
  - Note: A two-phase commit protocol (which is similar to the two-phase locking protocol we saw under centralized transaction management) works in a centralized environment, but is not applicable in a distributed environment.

On the concurrency side

- Concurrency control in most distributed systems is based on locking, just as it is in nondistributed systems. Some systems use multi-version controls, but locking is the most popular technique.

9. Hardware Independence

- The heading pretty much says it all for this objective.
- Real-world computer installations typically involve a multiplicity of different machines and hardware which must be configured to integrate the data on all of the systems to present the user with a “single-system image”.
- The same DBMS must run on different hardware platforms, and furthermore to have those different machines all participate as equal partners in the distributed system.

10. Operating System Independence

- This objective is a corollary of the previous one.
- It is obviously desirable to be able to run the same DBMS on different operating systems on either different or the same hardware.

11. Network Independence

- The system should be able to support many disparate sites, with disparate hardware and disparate operating systems.
- It is also desirable to support a variety of disparate communication networks.

12. DBMS Independence

- The system should be able to relax any requirements for strict homogeneity amongst the DBMSs.
- Realize that all this requirement really dictates is that the DBMS instances at different sites all support the same interface. They do not all need to be copies of the same DBMS software,

Synchronous v. Asynchronous DDB

- A significant trade-off in designing a DDB is whether to use synchronous or asynchronous distributed technology.
- In synchronous DDBs, all data across the network are continuously kept up-to-date so that a user at any site can access data anywhere on the network at any time and get the same answer.
- In asynchronous DDBs, replicated copies at different sites are not updated continuously but commonly at set intervals in time and thus there is some propagation delay when replicas may not be synchronized. More sophisticated strategies are required to ensure the correct level of data integrity and consistency across the sites.

Options for Distributing A Database

- There are four basic strategies that can be employed for distributing a database:

1. Data replication
   - Full
   - Partial
2. Horizontal fragmentation
3. Vertical fragmentation
4. Combinations of those above.
   - Replicated horizontal fragments
   - Replicated vertical fragments
   - Horizontal/vertical fragments

Data Replication

- Data replication has become an increasingly popular option for data distribution. This is in part due to the fault tolerance this technique provides.
- Data replication can use either synchronous or asynchronous technologies, although asynchronous technologies are more common in replication only environments.
- Full replication places a replica at each site in the network.
- Partial replication places a replica at some of the sites (at least two sites maintain replicas) in the network.

Data Replication (cont.)

- Data replication has 5 main advantages:

1. Reliability – A replica is available at another site if one site containing a replica should fail.
2. Fast response – Each site with a replica can process queries locally.
3. Avoid complicated distributed transaction integrity routines – Replicas are typically refreshed at periodic intervals, so most forms of replication are used when some relaxation of synchronization across the replicas is acceptable.
4. Node decoupling – Each transaction can proceed without coordination across the network. In place of real-time synchronization of updates, a behind-the-scenes process coordinates all replicas.
5. Reduce network traffic at prime time – Updating typically happens during prime business hours, when network traffic is highest and demands for rapid response greatest. Replication, with delayed updating, shifts this traffic to non-prime time.

Data Replication (cont.)

- Data replication has 2 primary disadvantages:

  1. Storage requirements – Each site that has a full replica must have the same storage capacity that would be required if the data were stored centrally. Each replica requires storage space as well as processing time when updates to the replicas are processed.

  2. Complexity and cost of updating – Whenever a base relation is updated, it must (eventually) be updated at each site that holds a replica. Synchronizing updating in near real-time requires careful coordination (as we'll see later).

Data Replication (cont.)

- Because of the advantages and disadvantages just outlined, data replication is favored where most process requests are read-only (queries) and where the data are relatively static, as in catalogs, telephone directories, train schedules, and so on.
- Replication is used for "noncollaborative data", where one location does not need a real-time update of data maintained by other locations.
- In these applications, the replicas need eventually to be synchronized, as quickly as practical, but real-time or near real-time constraints do not apply.
- Replication is not a viable approach for online applications such as airline reservation systems, ATM transactions, or applications where each user needs data about the same, nonsharable resource.

Updating Replicas – Snapshot Replication

- Several different schemes exist for updating replicas.
- Application such as data warehousing/data mining, or decision support systems – which do not require current up-to-the minute data – are typically supported by simple table copying or periodic snapshots.
- Assuming that multiple sites are updating the same data, this basically works as follows:
  - First, updates from all replicated sites are periodically collected at a master or primary site, where all the updates are made to form a consolidated record of all changes. This snapshot log, is a table of row identifiers for the records to go into the snapshot.
  - Then a read-only snapshot is sent to each site where there is a copy (it is often said that these other sites "subscribe" to the data owned at the primary site).
  - This is called a full refresh of the database.

Updating Replicas – Snapshot Replication (cont.)

- An alternative method is that only those pages that have changes since the last snapshot are sent. In this case, a snapshot log for each replicated table is joined with the associated base table to form the set of changed rows which are sent to the replicated sites.
- This is called a differential or incremental refresh.
- A more advanced form of snapshot replication allows shared ownership of the data. Shared updates introduces significant issues for managing update conflicts across sites.
  - For example, what if tellers at two branch banks try to update a customer's address simultaneously? Asynchronous technology would allow such a conflict to exist temporarily, which is fine as long as the update is not critical to business operations, provided that such a conflict can be detected and resolved before a business problem arises.

Updating Replicas – Snapshot Replication (cont.)

- The cost to perform a snapshot refresh depends on whether the snapshot is simple or complex.
- A simple snapshot is one that references either a portion or all of a single table.
- A complex snapshot involves multiple tables, usually from transactions that involve joins.
- Typically, DDS a simple snapshot can be refreshed using differential refresh whereas complex snapshots require more time-consuming full refresh.

Updating Replicas – Near Real-Time Replication (cont.)

- For situations that require near real-time refresh of replicas, store and forward messages for each completed transaction can be broadcast across the network informing all sites to update data as soon as possible, without forcing a confirmation to the originating site (confirmations are required in coordinated commit protocols), before the database at the originating site is updated.
- A common method for generating these messages is through the use of triggers. A trigger is stored at each site so that when a piece of replicated data is updated, the trigger executes corresponding update commands against remote replicas.
- Triggers allow each update event to be handled individually and transparently to programs and users.
- If a site is off-line or busy, the update message is held in a queue.

Updating Replicas – Push – Pull Strategies

- The mechanisms we've seen so far for updating replicas are all examples of push strategies.
- Push strategies for updating replicas always originate at the site where the original update occurred (the source). The update is then "pushed" out onto the network for other sites (the targets) to update their replicas.
- In pull strategies, the target, not the source, controls when a local replica is updated.
- With pull strategies, the local database determines when it needs to be refreshed, and requests a snapshot or the emptying of a message queue.
- Pull strategies have the advantage that the local site controls when it needs and can handle the updates. Thus, synchronization is less disruptive and occurs only when needed by each site, not when a central master site thinks it is best to update.

Database Integrity With Replication

- For both periodic and near real-time replication, consistency across the distributed, replicated database is compromised.
- Whether delayed or near real-time, the DBMS managing replicated database still must ensure the integrity of the database.
- Decision support systems permit synchronization on a table-by-table basis, whereas near real-time application require transaction-by-transaction synchronization.
- One of the main difficulties of handling updates with replicated databases depends on the number of sites at which updates may occur.
  - In a single-updater environments, updates are usually handled by periodically sending read-only snapshots to the non-updater sites. This effectively batches multiple updates together.
  - In multiple-updater environments, the main issue is data collision. Data collisions arise when independent updating sites each attempt to update the same data at the same time.

When To Use Replication

- Whether replication is a viable design for a distributed database system depends on several factors:

1. Data timeliness – Applications that can tolerate out-of-date data (whether this be a few seconds or a few hours) are better candidates for replication.

2. DBMS capabilities – An important DBMS capability is whether it will support a query that references data from more than one site. If not, then replication is a better candidate than the partitioning schemes (we're going to look at these next).

3. Performance implications – Replication means that each site must be periodically refreshed. During refreshment, the distributed site may be very busy handling a large volume of updates. If refreshment occurs via triggers, refreshment could come at an inopportune time for a given site, i.e., it is busy doing local work.

4. Heterogeneity in the network – Replication can be complicated if different site use different OSs and DBMSs. Mapping changes from one site to *n* other sites may imply *n* different routines to translate changes from the source to the *n* target sites.

5. Communications network capabilities – Transmission speeds and capacity in the network may prohibit frequent, complete refresh of large tables.