CGS 2545: Database Concepts
Summer 2007

Chapter 13 – Distributed Database Systems

Instructor : Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cgs2545/sum2007

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.

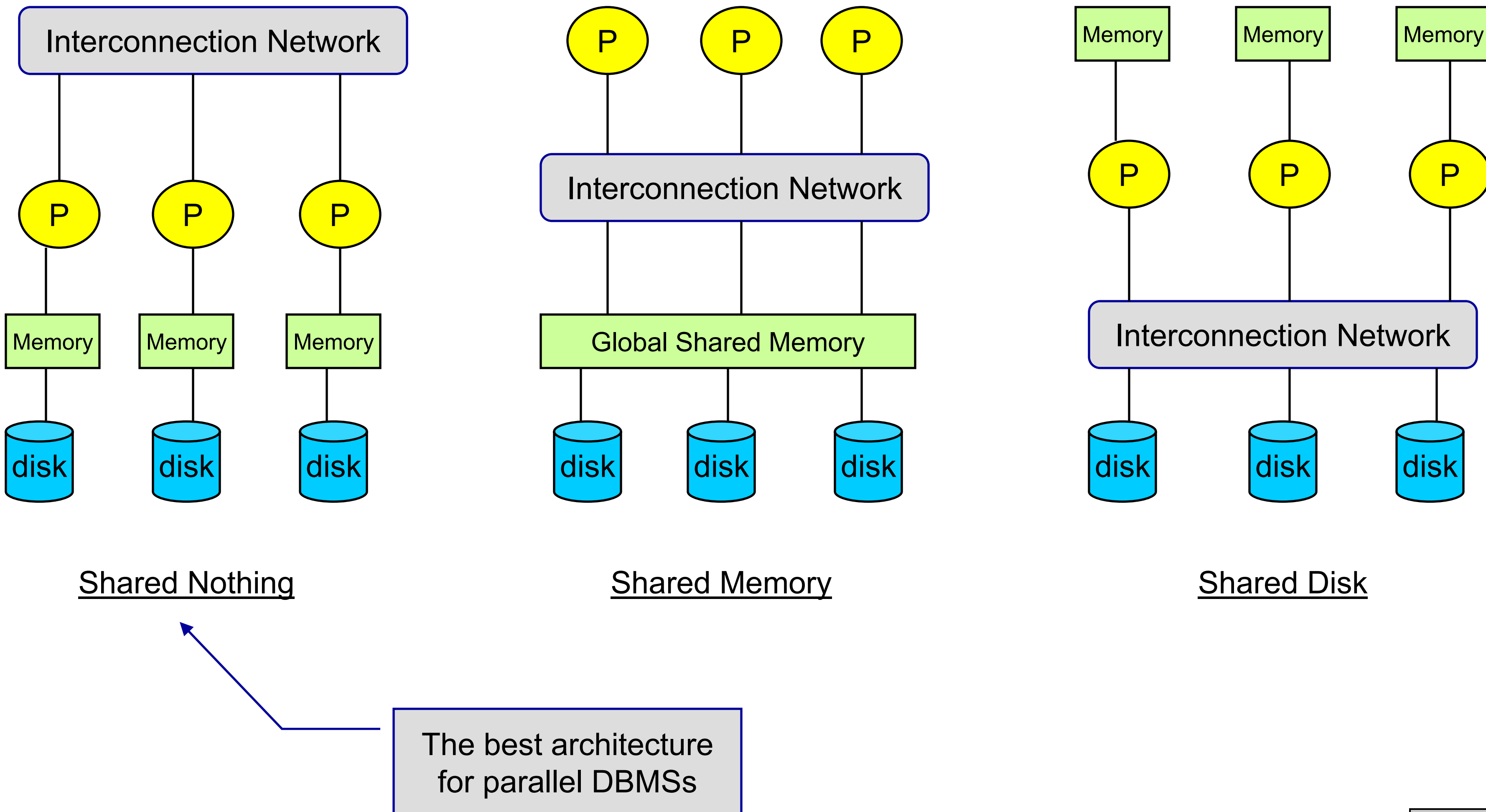
Parallel Database Systems

- 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.)



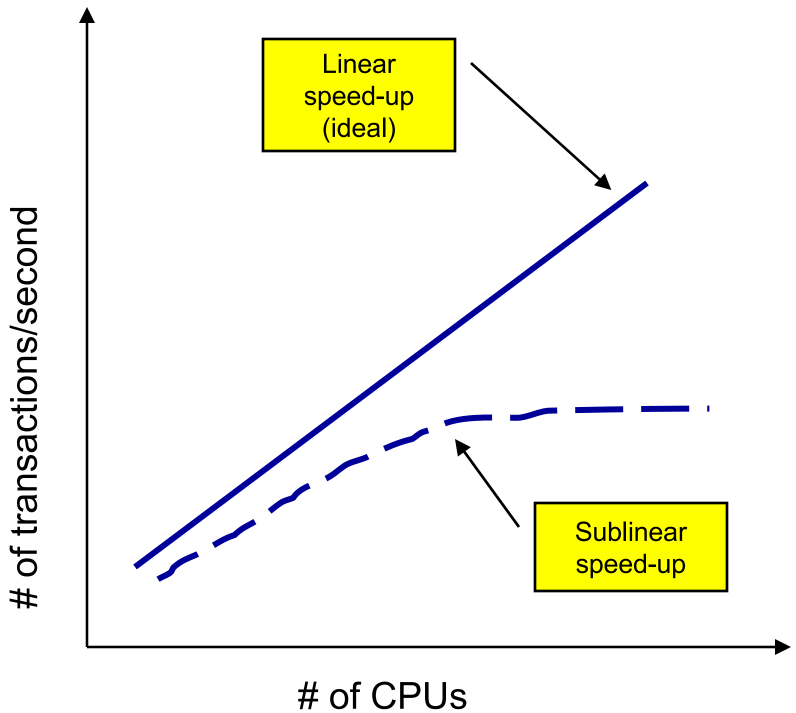
Parallel Database System Architectures (cont.)

- 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.

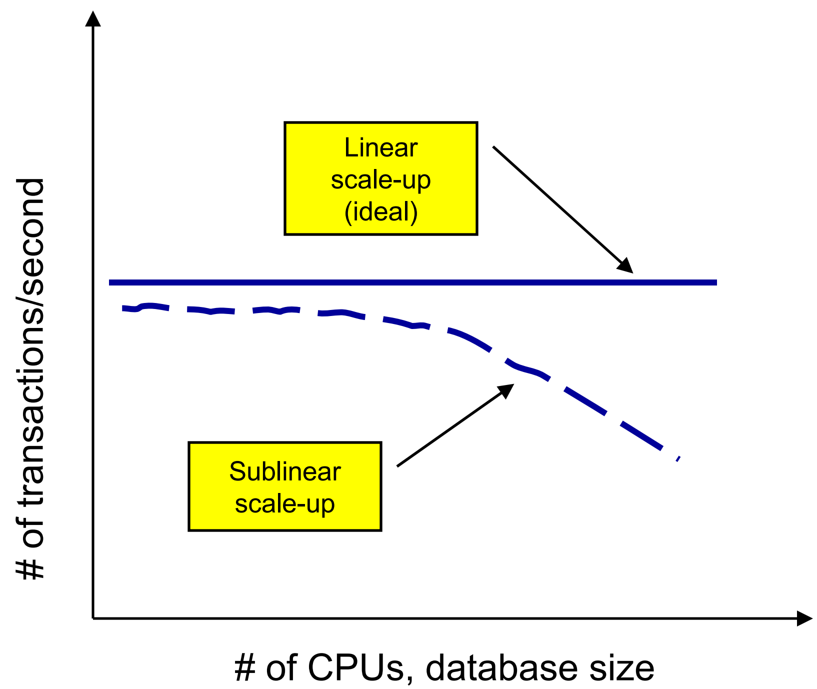
Parallel Database System Architectures (cont.)

- 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



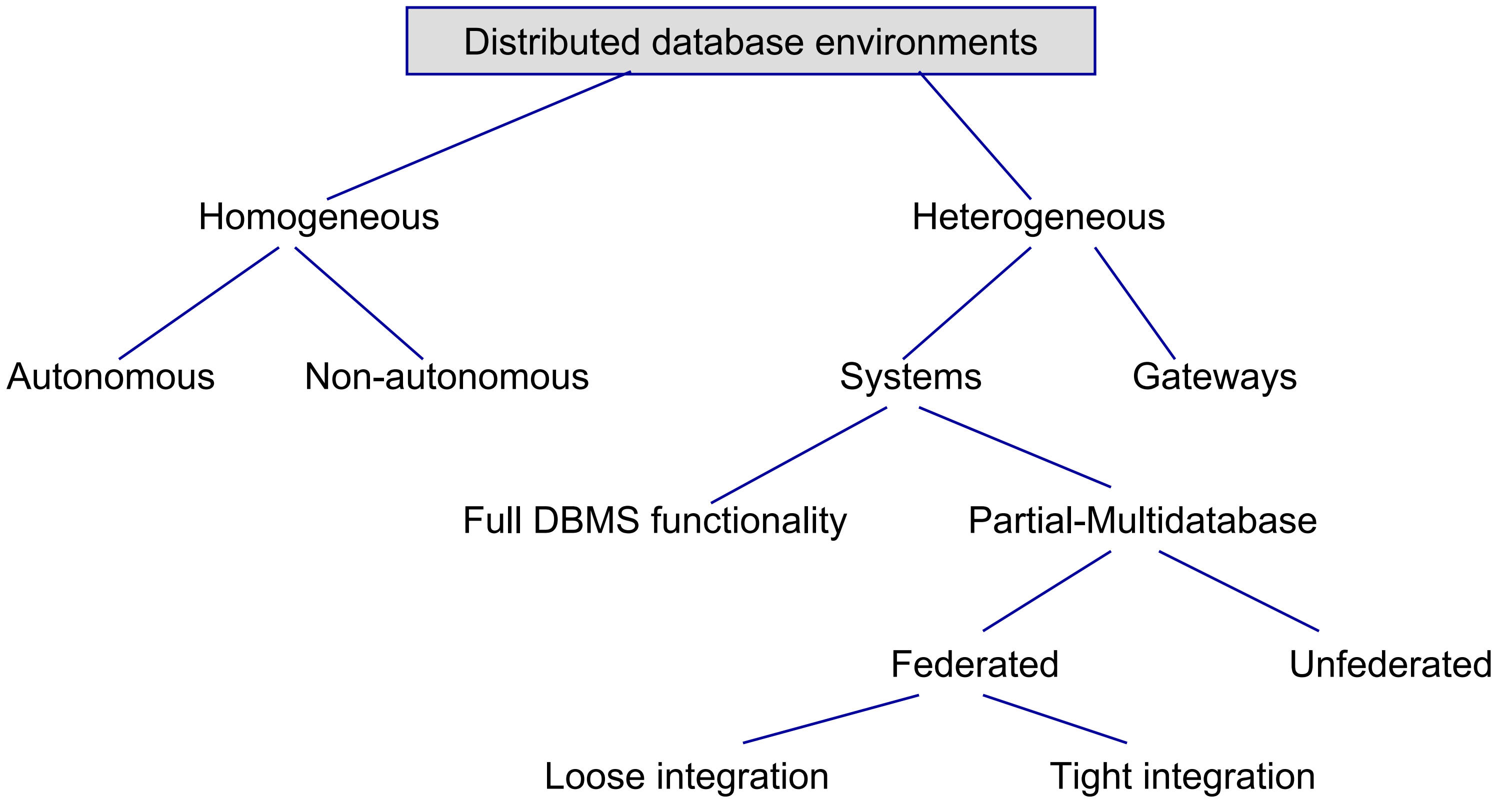
Scale-up

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.

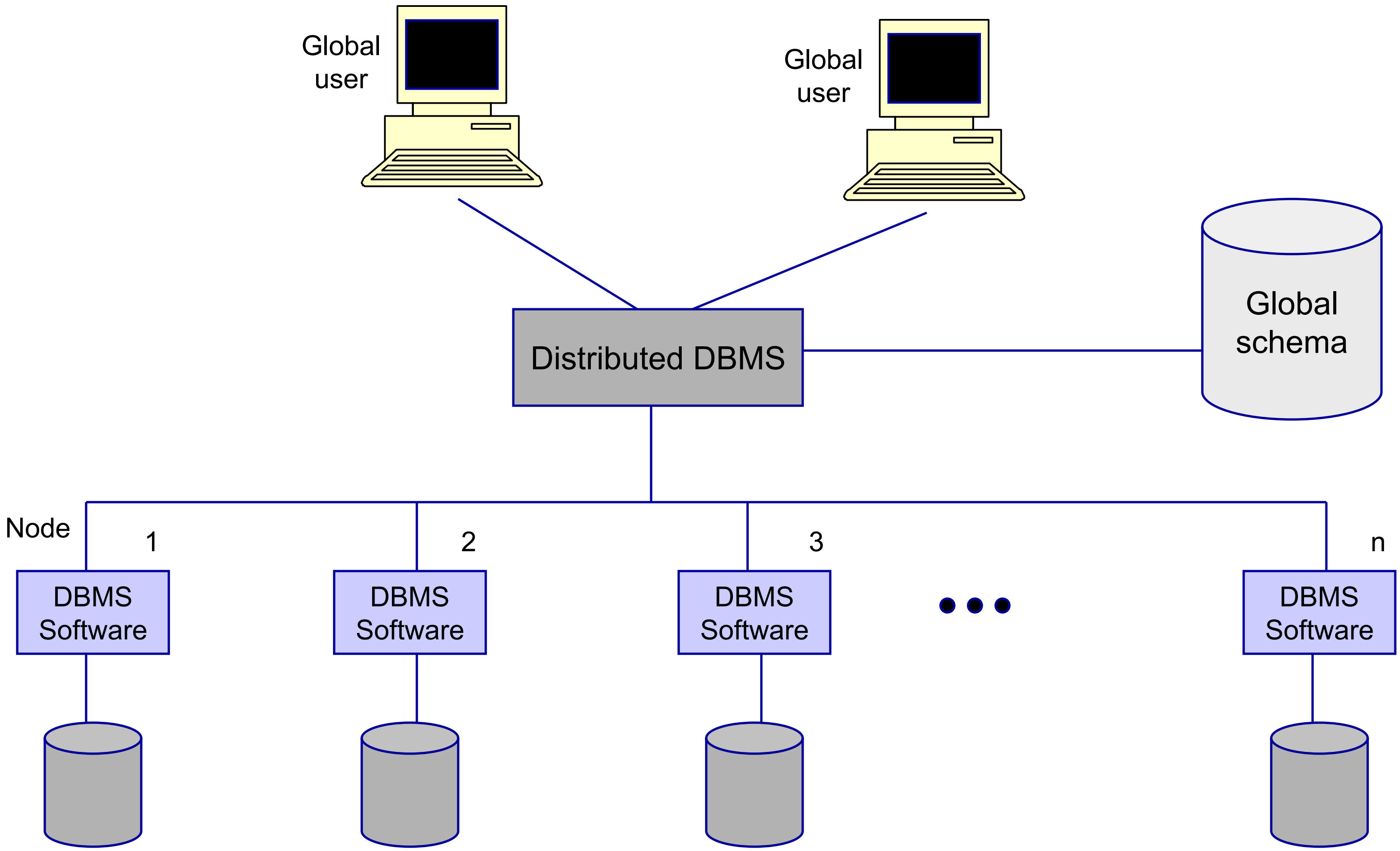
Heterogeneous – potentially different DBMSs are used at each site.

- 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.

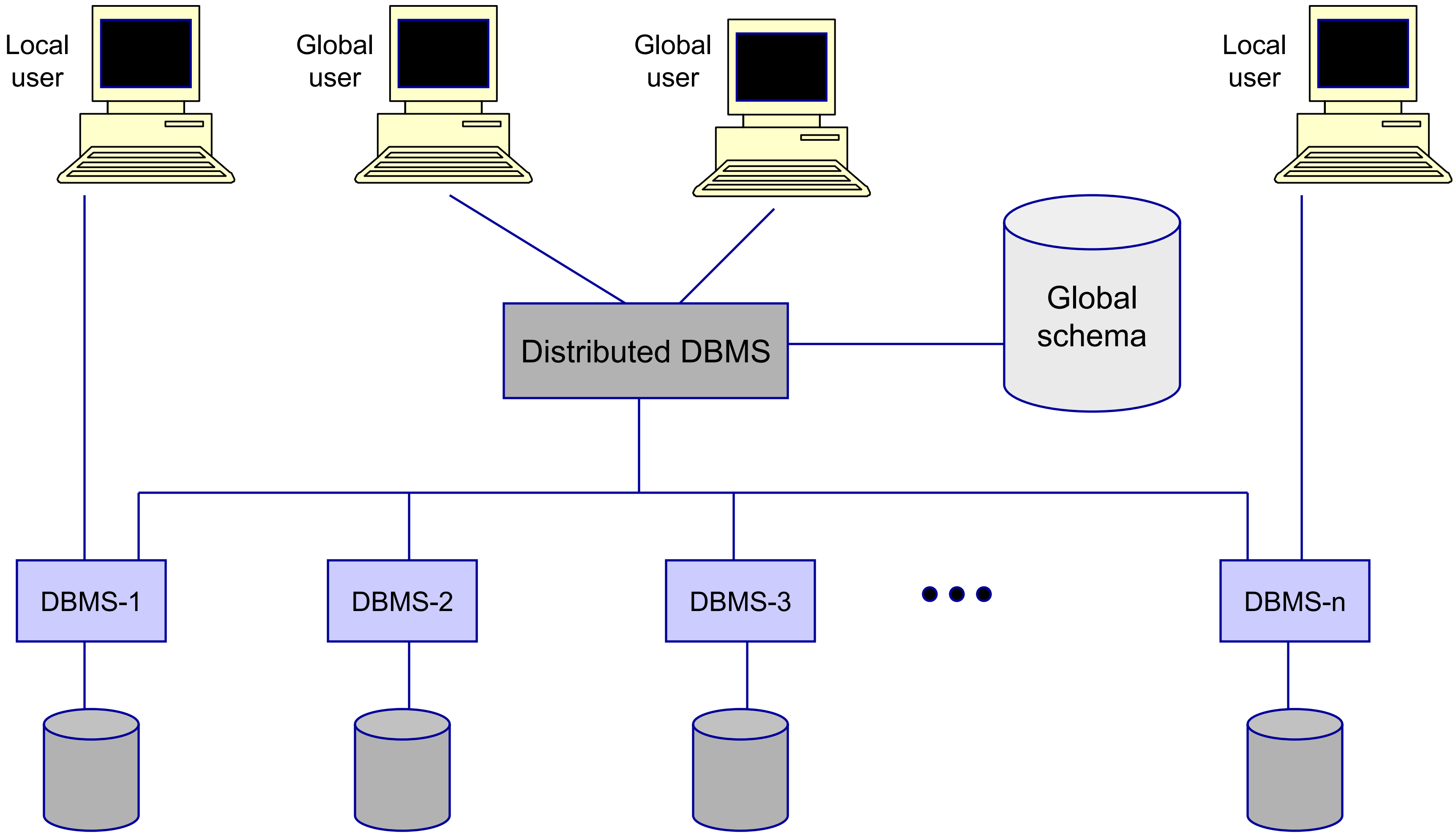
A Homogeneous Distributed Database System



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.

Options for Distributing A Database

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

1. Data replication
   - Full
   - Partial

   Covered in previous section of notes

2. Horizontal fragmentation
3. Vertical fragmentation
4. Combinations of those above.
   - Replicated horizontal fragments
   - Replicated vertical fragments
   - Horizontal/vertical fragments

Horizontal Fragmentation

- With horizontal fragmentation, some of the rows of a relation (table) are put into a base relation at one site, and other rows of the relation are put into a base relation at another site.
  - Note: there is no overlapping of the rows across the various sites – this is pure fragmentation, if there were overlapping rows, we would also have replication, which falls into the last category of distributed options. This would be a more general approach, although it is also quite common.
- Horizontal fragmentation results from applying selection conditions (relational algebra selections) to relations.

Horizontal Fragmentation (cont.)

Customer Name	Branch
Kristi	Oviedo
Debbie	Maitland
Michael	Longwood
Didi	Oviedo
Tawni	Oviedo

Initial table R

Horizontal fragments based on:

$\delta_{(Branch\ =\ \text{'Oviedo'})}(R)$

Customer Name	Branch
Kristi	Oviedo
Didi	Oviedo
Tawni	Oviedo

Fragment #1

Customer Name	Branch
Debbie	Maitland
Michael	Longwood

Fragment #2

Horizontal Fragmentation (cont.)

- Horizontal fragmentation has four major advantages:

1. Efficiency – Data can be stored close to where they are used and separate from other data used by other users or applications.

2. Local optimization – Data can be stored to optimize performance for local access rather than global access.

3. Security – Data not relevant to usage at a particular site is not made available at that site.

4. Ease of querying – Combining data across horizontal fragments is easy because the rows are simply merged by unions across the fragments.

Horizontal Fragmentation (cont.)

- Horizontal fragmentation has two primary disadvantages:

  1. Inconsistent access speed – When data from several fragments are required, the access time can be significantly different from local-only data access.

  2. Backup vulnerability – Since the data is not replicated, when data at one site becomes inaccessible or damaged, usage cannot switch to another site where a copy exists; data may be lost if proper backup is not performed at each site.

- Horizontal fragmentation is typically used when an organizational function is distributed, but each site is concerned with only a subset of the entity instances (often geographically based).

Vertical Fragmentation

- With vertical fragmentation, some of the columns of a relation (table) are put into a base relation at one site, and other columns of the relation are put into a base relation at another site.
  - Note: there must be a common domain stored at each site so that the original table can be reconstructed.
- Vertical fragmentation results from applying projection operations (relational algebra projection) to relations.

Vertical Fragmentation (cont.)

Customer Name	Branch	Balance
Kristi	Oviedo	15,000
Debbie	Maitland	23,000
Michael	Longwood	4,000
Didi	Oviedo	50,000
Tawni	Oviedo	18,000

Initial table R

Vertical fragment based on: $\pi_{(name,\ branch)}(R)$

Customer Name	Branch
Kristi	Oviedo
Debbie	Maitland
Michael	Longwood
Didi	Oviedo
Tawni	Oviedo

Vertical fragment based on: $\pi_{(name,\ balance)}(R)$

Customer Name	Balance
Kristi	15,000
Debbie	23,000
Michael	4,000
Didi	50,000
Tawni	18,000

Combinations of Distribution Strategies

- To complicate matters even further, there are an almost unlimited number of combinations of distribution strategies based upon the previous cases.
- Some data may be stored centrally while others are replicated. Both horizontal and vertical fragments can be replicated.

Horizontal/Vertical Fragmentation

Customer Name	Branch	Balance
Kristi	Oviedo	15,000
Debbie	Maitland	23,000
Michael	Longwood	4,000
Didi	Oviedo	50,000
Tawni	Oviedo	18,000

Initial table R

Fragment based on: $\delta_{(branch\ =\ \text{'Oviedo'})}(\ \pi_{(name,\ branch)}(R))$

Customer Name	Branch
Kristi	Oviedo
Didi	Oviedo
Tawni	Oviedo

Fragment based on: $\delta_{(branch\ <>\ \text{'Oviedo'})}(\ \pi_{(name,\ branch)}(R))$

Customer Name	Branch
Debbie	Maitland
Michael	Longwood

Fragment based on: $\delta_{(balance\ >15000)}(R)$

Customer Name	Branch	Balance
Debbie	Maitland	23,000
Didi	Oviedo	50,000
Tawni	Oviedo	18,000

Fragment based on: $\delta_{(name\ =\ \text{'Krisit'})}(\ \pi_{(name,\ balance)}(R))$

Customer Name	Balance
Kristi	15,000

Selecting a Distribution Strategy

- A distributed database can be organized in five unique ways:
  1. Totally centralized – all data resides at one location accessed from many geographically distributed sites.
  2. Partially or totally replicated (snapshot) – data is either partially or totally replicated across geographically distributed sites, with each replica periodically updated with snapshots.
  3. Partially or totally replicated (real-time synchronization) – data is either partially or totally replicated across geographically distributed sites, with near real-time synchronization.
  4. Fragmented (integrated) – data is into segments at different geographically distributed sites, but still within one logical database and one distributed DBMS.
  5. Fragmented (nonintegrated) – data is fragmented into independent, non integrated segments spanning multiple computer systems and database software.

Summary of Distributed Design Strategies

Strategy	*Reliability*	*Expandability*	*Communications Overhead*	*Manageability*	*Data Consistency*
Centralized	**POOR:** Highly dependent on central server	**POOR:** Limitations are barriers to performance	**VERY HIGH:** High traffic to one site	**VERY GOOD:** One, monolithic site requires little coordination	**EXCELLENT:** All users always have same data
Replicated with snapshots	**GOOD:** Redundancy and tolerated delays	**VERY GOOD:** Cost of additional copies may be less than linear	**LOW to MEDIUM:** Not constant, but periodic snapshots can cause bursts of network traffic	**VERY GOOD:** Each copy is like every other one	**MEDIUM:** Fine as long as delays are tolerated by business needs
Synchronized replication	**EXCELLENT:** Redundancy and minimal delays	**VERY GOOD:** Cost of additional copies may be low and synchronization work only linear	**MEDIUM:** Messages are constant, but some delays are tolerated	**MEDIUM:** Collisions add some complexity to manageability	**MEDIUM to VERY GOOD:** Close to precise consistency
Integrated partitions	**VERY GOOD:** Effective use of partitioning and redundancy	**VERY GOOD:** New nodes get only data they need without changes in overall database design	**LOW to MEDIUM:** Most queries are local but queries which require data from multiple sites can cause a temporary load	**DIFFICULT:** Especially difficult for queries that need data from distributed tables, and updates must be tightly coordinated	**VERY POOR:** Considerable effort, and inconsistencies not tolerated
Decentralized with independent partitions	**GOOD:** Depends on only local database availability	**GOOD:** New sites independent of existing ones	**LOW:** Little if any need to pass data or queries across the network (if one exists)	**VERY GOOD:** Easy for each site, until there is a need to share data across sites	**LOW:** No guarantees of consistency, in fact pretty sure of inconsistency

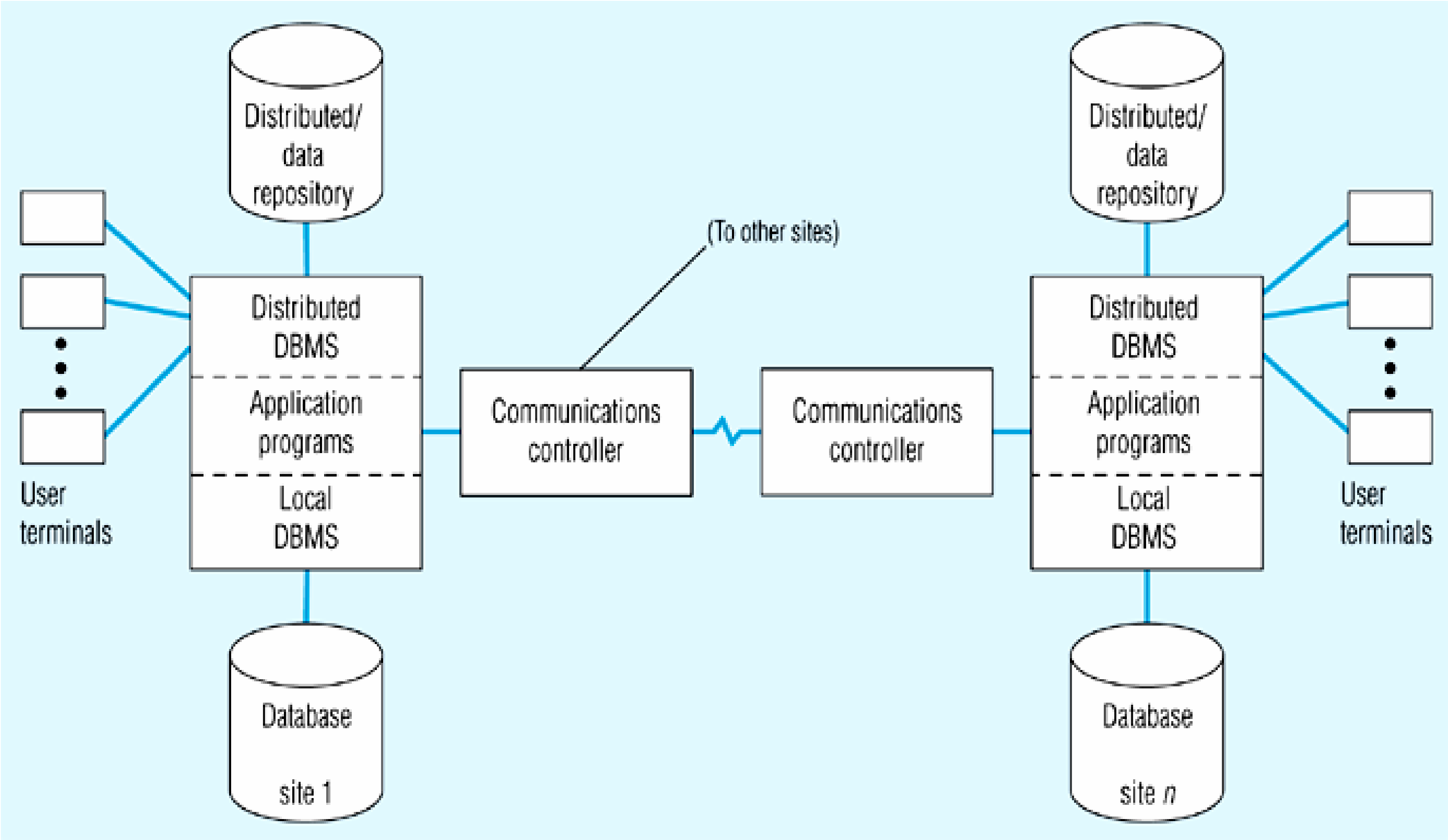
Distributed DBMS

- To have a distributed database, there must be a database management system that coordinates the access to the data at the various sites.
- Such a system is called a distributed DBMS.
- Although each site may have a DBMS managing the local database at that site, a distributed DBMS must perform the following functions for the distributed database.

Functions of a Distributed DBMS

- Locate data with a distributed data dictionary.
- Determine location from which to retrieve data and process query components.
- DBMS translation between nodes with different local DBMSs (using middleware).
- Data consistency (via multiphase commit protocols).
- Global primary key control.
- Provide scalability.
- Security, concurrency, query optimization, failure recovery.

Distributed DBMS Architecture



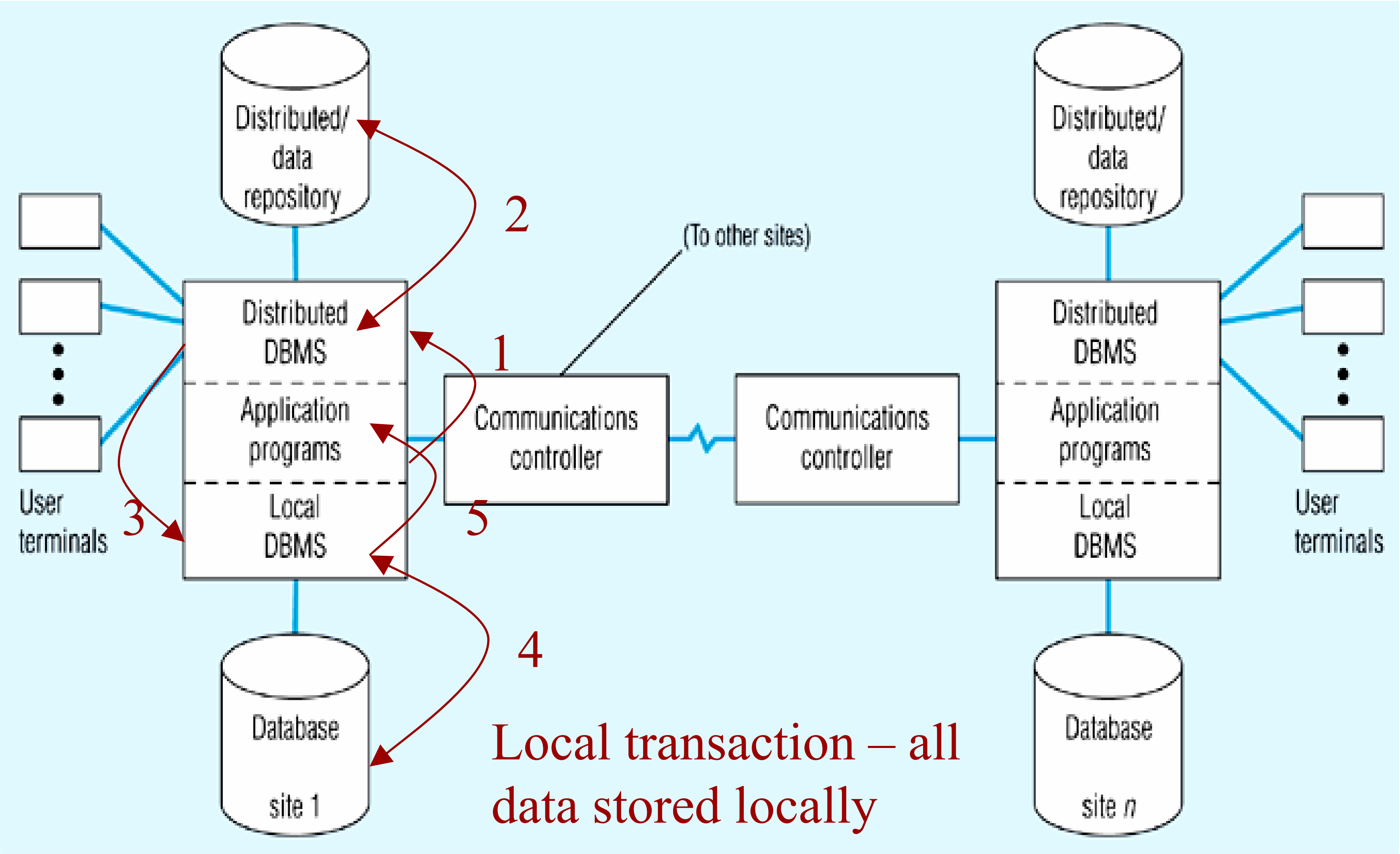
Local vs. Global Transactions

- A local transaction is one for which the required data are stored entirely at the local site.
  - The distributed DBMS passes the request to the local DBMS.
- A global transaction references data at one or more non-local sites.
  - The distributed DBMS routes the request to other sites as necessary. The distributed DBMSs at the participating sites exchange messages as needed to coordinate the processing of the transaction until it is completed (or aborted, if necessary).

Steps to Process a Local Transaction

1. Application makes request to distributed DBMS
2. Distributed DBMS checks distributed data repository for location of data. Finds that it is local.
3. Distributed DBMS sends request to local DBMS
4. Local DBMS processes request
5. Local DBMS sends results to application

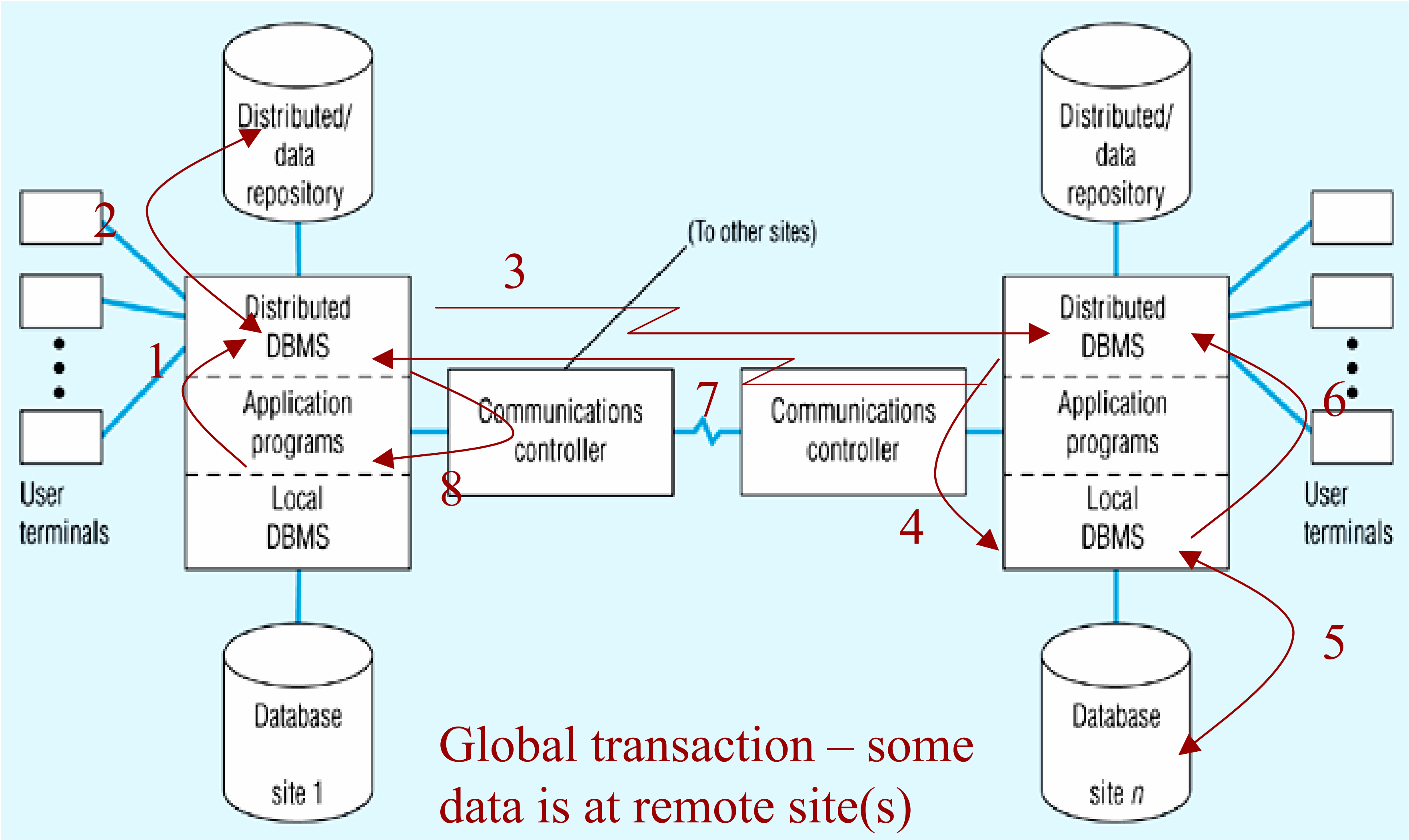
Processing a Local Transaction



Steps to Process a Global Transaction

1. Application makes request to distributed DBMS
2. Distributed DBMS checks distributed data repository for location of data. Finds that it is **remote**
3. Distributed DBMS routes request to remote site
4. Distributed DBMS at remote site translates request for its local DBMS if necessary, and sends request to local DBMS
5. Local DBMS at remote site processes request
6. Local DBMS at remote site sends results to distributed DBMS at remote site
7. Remote distributed DBMS sends results back to originating site
8. Distributed DBMS at originating site sends results to application

Processing a Global Transaction



Global transaction – some data is at remote site(s)

Distributed DBMS Transparency Objectives

- Location Transparency
  - User/application does not need to know where data resides
- Replication Transparency
  - User/application does not need to know about duplication
- Failure Transparency
  - Either all or none of the actions of a transaction are committed
  - Each site has a transaction manager
    - Logs transactions and before and after images
    - Concurrency control scheme to ensure data integrity
  - Requires special *commit protocol*

Two-Phase Commit

- **Prepare Phase**
  - Coordinator receives a commit request
  - Coordinator instructs all resource managers to get ready to "go either way" on the transaction. Each resource manager writes all updates from that transaction to its own physical log
  - Coordinator receives replies from all resource managers. If all are ok, it writes commit to its own log; if not then it writes rollback to its log

Two-Phase Commit (cont.)

- **Commit Phase**
  - Coordinator then informs each resource manager of its decision and broadcasts a message to either commit or rollback (abort). If the message is commit, then each resource manager transfers the update from its log to its database
  - A failure during the commit phase puts a transaction "in limbo." This has to be tested for and handled with timeouts or polling

Concurrency Control

Concurrency Transparency

- Design goal for distributed database

- Timestamping
  - Concurrency control mechanism
  - Alternative to locks in distributed databases

Query Optimization

- In a query involving a multi-site join and, possibly, a distributed database with replicated files, the distributed DBMS must decide where to access the data and how to proceed with the join. Three step process:

  1 Query decomposition - rewritten and simplified

  2 Data localization - query fragmented so that fragments reference data at only one site

  3 Global optimization -

    - Order in which to execute query fragments
    - Data movement between sites
    - Where parts of the query will be executed

Distributed Query Processing

- As we've just seen (global vs. local transactions), with distributed databases, the response to a query may require the DDBMS to assemble data from several different sites (remember though that location transparency will make the user unaware of this fact).

- A major decision for the DDBMS is how to process a query. How the query will be processed is affected primarily by two factors:

  1. How the user formulates the query (as we saw in the centralized case) and how it can be transformed by the DDBMS.

  2. Intelligence of the DDBMS in developing a sensible plan of execution (distributed optimization).

Distributed Query Processing – Example

- Consider the simplified version of our supplier/parts database as shown below:

  suppliers (s#, city) [located at site A, contains 10,000 tuples]

  parts (p#, color) [located at site B, contains 100,000 tuples]

  shipments (s#, p#, qty) [located at site A, contains 1,000,000 tuples]

Assumptions

- Each tuple is 100 bytes.
- There are exactly 10 red parts.
- The query is: List the supplier numbers for suppliers in Orlando who ship a red part.
- There are 100,000 tuples in the shipments relation that involve shipments from suppliers in Orlando.
- Computation time at any site is negligible compared to communication time.
- Network transfer rate is 10,000 bytes/sec.
- Access delay = 1 second (time to send a message – not a tuple from one site to another).
- T = total communication time = total access delay + (total data volume / data rate)

  = (# messages sent x 1 sec/message) + (total # of bytes sent / 10,000)

Distributed Query Processing – Example (cont.)

Strategy #1

- Move entire parts relation to site A and process query at site A.
  - $T_1 = 1 + (100{,}000 \times 100)/10{,}000 \approx 1000$ sec = 16.7 minutes

Strategy #2

- Move supplier and shipment relations to site B and process the query at site B.
  - $T_2 = 2 + ((10{,}000 + 1{,}000{,}000) \times 100)/10{,}0000 = 10{,}100$ sec = 2.8 hours

Distributed Query Processing – Example (cont.)

Strategy #3

- Join suppliers and shipments relations at site A, select tuples from the join for which the city is Orlando, and then, for each of those tuples in turn, check site B to see if the indicated part is red. Each check requires 2 messages, a query, and a response. Transmission time for these messages is small compared to the access delay. There will be 100,000 tuples in the join for which the supplier is located in Orlando.
  - T_3 = (100,000 tuples to check) x (2) x (1 sec/message) = 200,000 sec ≈ 55 hours = 2.3 days

Strategy #4

- Select tuples from the parts relation at site B for which the color is red, and then, for each of these tuples in turn, check at site A to see if there exists a shipment of the part from an Orlando supplier. Again, each check requires two messages.
  - T_4 = (10 red parts) x (2 messages each) x (1 sec/message) = 20 sec

Distributed Query Processing – Example (cont.)

Strategy #5

- Join suppliers and shipments relations at site A, select tuples from the join for which the city is Orlando, and then, project only the s# and p# attributes and move this "qualified" relation to site B where the query processing will be completed.
  - T_5 = (1 + (100,000 tuples for Orlando) x (100 bytes/tuple)/10,000 bytes/second ≈ 1000 sec = 16.7 minutes

Strategy #6

- Select tuples from the parts relation at site B for which the color is red, then move this result to site A to complete the query processing.
  - T_4 = 1 + (10 red parts x (100 bytes/tuple) / 10,000 ≈ 1 sec

Distributed Query Processing – Example (cont.)

Summary

Strategy		Time
1	Move parts table to site A, process query at site A.	16.7 minutes
2	Move suppliers and shipments tables to site B, process query at site B.	2.8 hours
3	Join suppliers and shipments at site A, check selected rows at site B.	2.3 days
4	Select red parts from parts tables at site B, for these tuples check at site A for a shipment of this part.	20 seconds
5	Join suppliers and parts at site A, move "qualified" rows to site B for processing.	16.7 minutes
6	Select red parts from parts table at site B, move these tuples to site A for processing.	≈1 second

Distributed Query Transformation

Horizontal fragmentation example

- Suppose we have the `shipments` table horizontally fragmented as follows:
  - shipments = SPJ1 ∪ SPJ2 where

    $$SPJ1 = \sigma_{(p\# = \text{'P1'})}(\text{shipments}) \text{ and } SPJ2 = \sigma_{(p\# \neq \text{'P1'})}(\text{shipments})$$

  - assume that SPJ1 is located at site1 and SPJ2 is located at site 2.
- A user at some site (assume its is neither site 1 or site 2) wants the answer to the query "list the supplier numbers for those suppliers who ship part P1" and issues the query expression: $\pi_{s\#}(\sigma_{(p\#=\text{'P1'})}(\text{shipments}))$ to determine the results.
- Remember that the user is unaware of the fragmentation of the shipments relation.

Distributed Query Transformation (cont.)

Horizontal fragmentation example (cont.)

- Since shipments is defined as shipments = SPJ1 ∪ SPJ2 the query will be transformed into: $\pi_{s\#}(\sigma_{(p\#='P1')}(SPJ1 \cup SPJ2))$.
- The query optimizer will initially transform the expression above into: $[\pi_{s\#}(\sigma_{(p\#='P1')}(SPJ1)] \cup [\pi_{s\#}(\sigma_{(p\#='P1')}(SPJ2)]$.
- Further optimization can be done since the system can determine that SPJ2 is defined as: $SPJ2 = \sigma_{(p\# \neq 'P1')}(shipments)$. Due to this definition, the sub-expression involving SPJ2 does not need to be evaluated as it will not contribute any values to the result set.
- Further since SPJ1 is defined as: $SPJ1 = \sigma_{(p\# = 'P1')}(shipments)$, the query can be further simplified to: $\pi_{s\#}(SPJ1)$.

Distributed Query Transformation (cont.)

Customer Name	Branch
Kristi	Oviedo
Debbie	Maitland
Michael	Longwood
Didi	Oviedo
Tawni	Oviedo

Initial table R

Horizontal fragments based on:

$\sigma_{(Branch\ =\ 'Oviedo')}(R)$

Customer Name	Branch
Kristi	Oviedo
Didi	Oviedo
Tawni	Oviedo

Fragment #1

Customer Name	Branch
Debbie	Maitland
Michael	Longwood

Fragment #2

Consider queries such as:

(1) List customer names at branch in Oviedo.

(2) List customer names at branches not in Oviedo.

(3) List customer names at any branch.

Distributed Query Transformation

Vertical fragmentation example

- Suppose we have the `shipments` table horizontally fragmented as follows:
  - shipments = SPJ1 ∪ SPJ2 where

    SPJ1 = $\sigma_{(p\# = \text{'P1'})}$(shipments) and SPJ2 = $\sigma_{(p\# \neq \text{'P1'})}$(shipments)
  - assume that SPJ1 is located at site1 and SPJ2 is located at site 2.
- A user at some site (assume its is neither site 1 or site 2) wants the answer to the query "list the supplier numbers for those suppliers who ship part P1" and issues the query expression: $\pi_{s\#}(\sigma_{(p\#=\text{'P1'})}$(shipments) to determine the results.
- Remember that the user is unaware of the fragmentation of the shipments relation.

Distributed Query Transformation (cont.)

Vertical fragmentation example

Customer Name	Branch	Balance
Kristi	Oviedo	15,000
Debbie	Maitland	23,000
Michael	Longwood	4,000
Didi	Oviedo	50,000
Tawni	Oviedo	18,000

Initial table R

VF1: $\pi_{(name,\ branch)}(R)$

Customer Name	Branch
Kristi	Oviedo
Debbie	Maitland
Michael	Longwood
Didi	Oviedo
Tawni	Oviedo

VF2: $\pi_{(name,\ balance)}(R)$

Customer Name	Balance
Kristi	15,000
Debbie	23,000
Michael	4,000
Didi	50,000
Tawni	18,000

Query: List customer names in Oviedo with balances >= 15,000

Initial query expression: $\pi_{customer\ name}(\sigma_{(balance\ >=\ 15000\ and\ branc\ =\ 'Oviedo')}(R))$

Query will be transformed into:

$\pi_{customer\ name}[(\sigma_{(balance\ >=\ 15000)}(VF2)) \bowtie (\sigma_{(branch\ =\ 'Oviedo')}(VF1))]$

Semi Join Strategy

- In general, join operations are costly. This is especially true in a distributed environment where shipping large join tables around the network can be extremely costly.
- One technique that is commonly employed is the semi join (See Chapter 4 notes, pages 14-15).
- In a semi join, only the joining attribute is sent from one site to another, and then only the required rows are returned. If only a small percentage of the rows participate in the join, then the amount of data being transferred is minimized.
- $R1 \rhd R2 \equiv \pi_{R1}(R1 \bowtie R2)$ (recall that $R1 \rhd R2 \neq R2 \rhd R1$)

Semi Join Strategy - Example

- Consider the following distributed database.

Site 1

Customer table

Customer_num	10 bytes
Customer_name	50 bytes
Zipcode	10 bytes
SIC	5 bytes

Current instance contains 10,000 rows

Site 2

Order table

Order_num	10 bytes
Customer_num	10 bytes
Order_Date	4 bytes
Order_amount	6 bytes

Current instance contains 400,000 rows

Semi Join Strategy – Example (cont.)

- Assume that a query originates at site 1 to display the Customer_name, SIC, and Order_date for all customers in a particular Zipcode range and an Order_amount above a specified value.
- Further assume that 10% of the customers fall into the particular zipcode range and 2% of the orders are above the specified value.
- Given these conditions, a semi join will work as follows:
  - A query is executed at site 1 to create a list of the Customer_num values in the desired Zipcode range. So, 1,000 rows satisfy the zipcode condition (since 10% of 10,000 = 1000) and each of these rows involves a 10-byte Customer_num field, so in total, 10,000 bytes will be sent from site 1 to site 2.

Semi Join Strategy – Example (cont.)

- A query is executed at site 2 to create a list of the Customer_num and Order_date values to be sent back to site 1 to compose the final result. If we assume roughly the same number of orders for each customer, then 40,000 rows of the order table will match with Customer_num values sent from site1. Assuming that any order is equally likely to be above the amount limit, then 800 rows (2% of 40,000) apply to this query. This means that 11,200 bytes (14 bytes/row x 800 rows) will be sent to site 1.

- The total amount of data transferred is only 21,200 bytes using the semi join strategy.
- The total data transferred that would result from simply sending the subset of each table needed to the other site would be:

Semi Join Strategy – Example (cont.)

- To send data from site 1 to site 2 requires sending the Customer_num, Customer_name, and SIC: total of 65 bytes/row for 1000 rows of the Customer table = 65,000 bytes from site 1 to site 2.
- To send data from site 2 to site 1 requires sending the Customer_num and Order_date: total of 14 bytes for 8000 rows of the Order table = 112,000 bytes.
- The semi join strategy required only 21,200 bytes to be transferred.