DISTRIBUTED DATABASES

DEFINITIONS

A distributed database (DDB) is a collection of multiple, *logically interrelated* databases distributed over a *computer network*.

A distributed database management system (D–DBMS) is the software that manages the DDB and provides an access mechanism that makes this distribution *transparent* to the users.

Distributed database system (DDBS) = DB + Communication

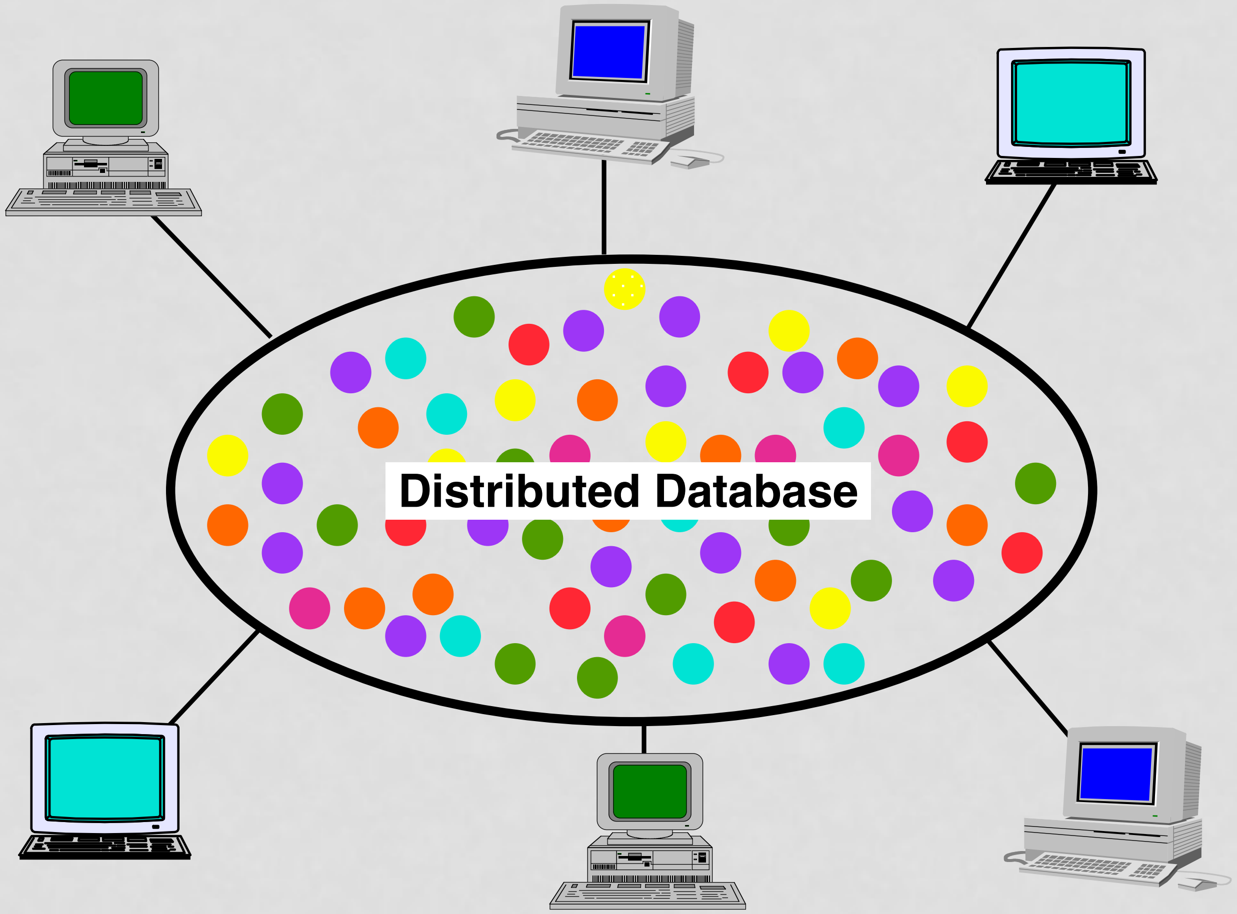
DISTRIBUTED DATABASES MAIN CONCEPTS

- **Data are stored at several locations**
  - Each managed by a DBMS that can run autonomously
- **Ideally, location of data is unknown to client**
  - ***Distributed Data Independence***
- **Distributed Transactions**
  - Clients can write Transactions regardless of where the affected data are located
  - ***Big question:*** *How to ensure the ACID properties Distributed Transactions???*

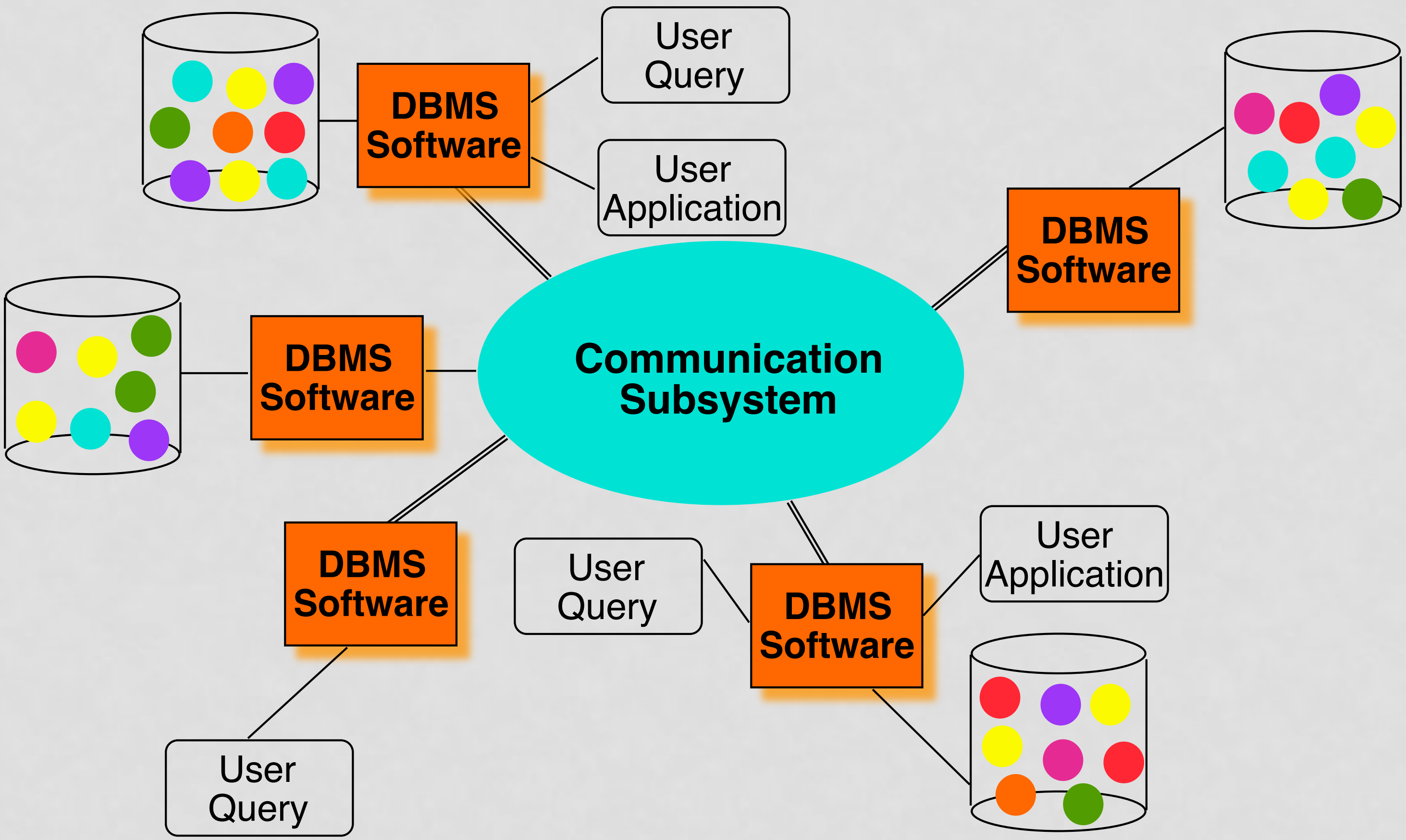
DISTRIBUTED DBMS PROMISES

- *Transparent management of distributed, fragmented, and replicated data*
- *Improved reliability/availability through distributed transactions*
- *Improved performance*
- *Easier and more economical system expansion*

DISTRIBUTED DATABASE - USER VIEW



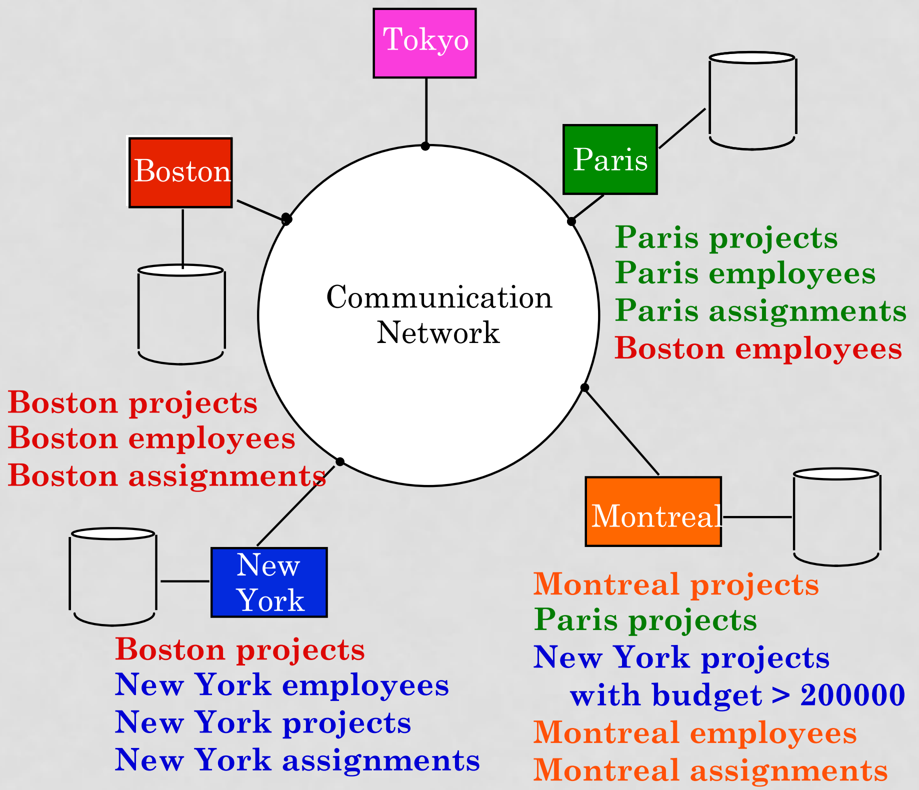
DISTRIBUTED DBMS - REALITY



TRANSPARENCY & DATA INDEPENDENCE

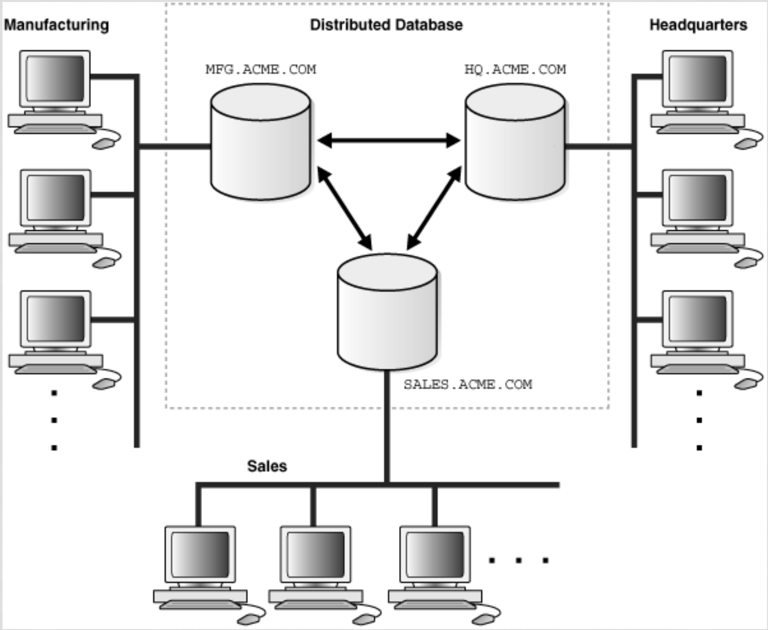
- Data distributed (with some replication)
- Transparently ask query:

```
SELECT ENAME,SAL
FROM   EMP,ASG,PAY
WHERE  DUR > 12
AND    EMP.ENO = ASG.ENO
AND    PAY.TITLE = EMP.TITLE
```

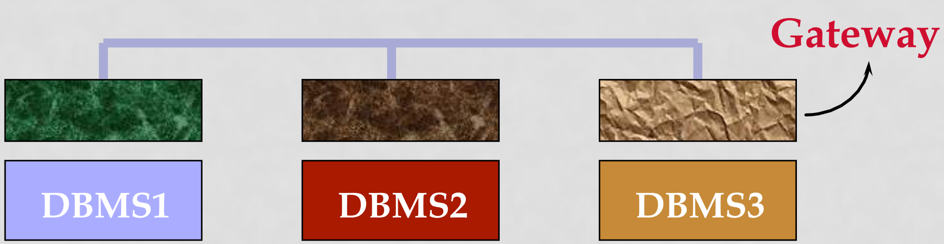


TYPES OF DISTRIBUTED DATABASES

- **Homogeneous**
  - Every site runs the same type of DBMS
- **Heterogeneous:**
  - Different sites run different DBMS (maybe even RDBMS and ODBMS)



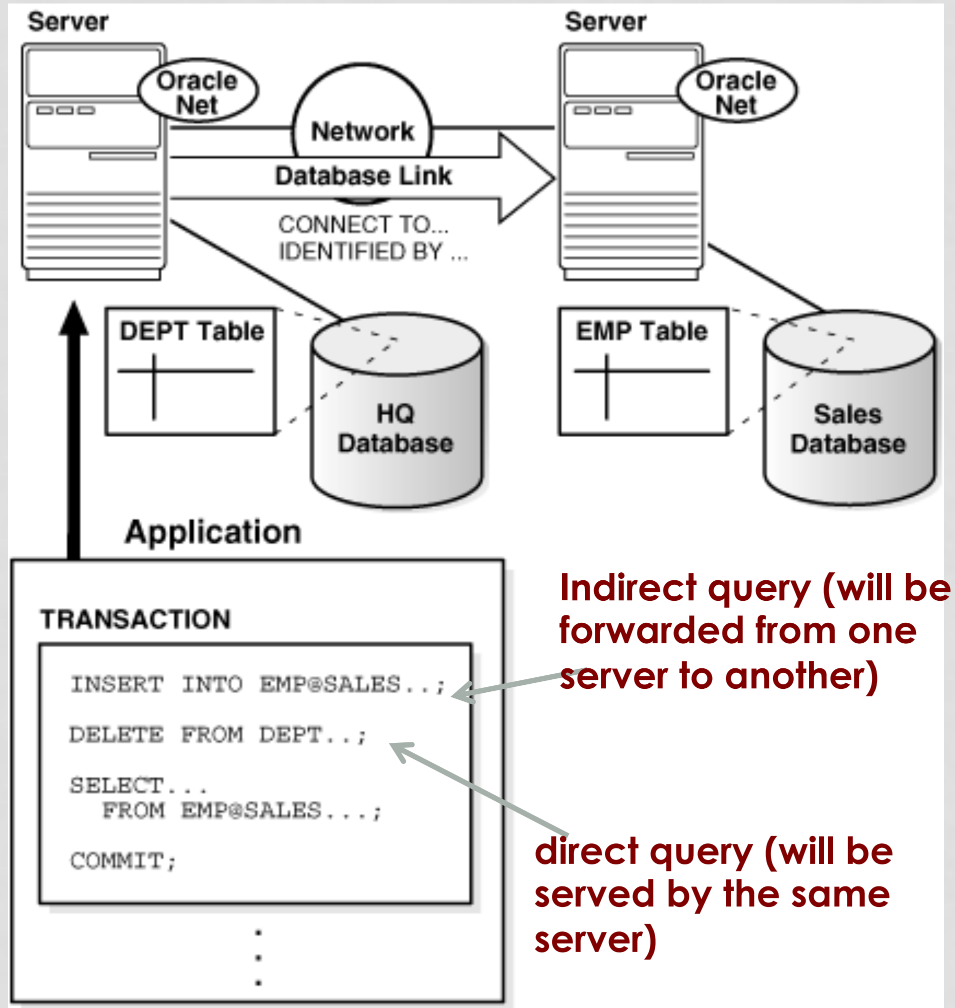
Homogeneous DBs can communicate directly with each other



Heterogeneous DBs communicate through gateway interfaces

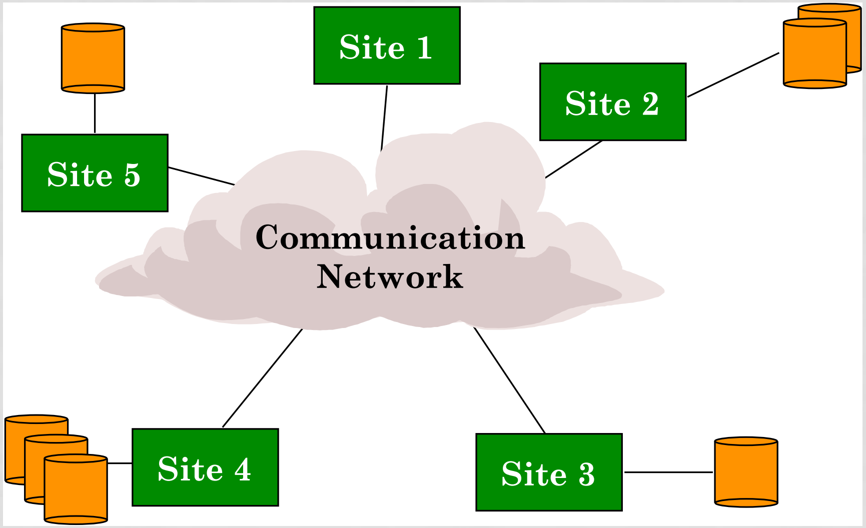
DISTRIBUTED DATABASE ARCHITECTURE

- **Client-Server**
  - Client connects directly to specific server(s) and access only their data
  - Direct queries only
- **Collaborative Servers**
  - Servers can serve queries or be clients and query other servers
  - Support indirect queries



DISTRIBUTED DATABASE ARCHITECTURE (CONT'D)

- **Peer-to-Peer Architecture**
  - Scalability and flexibility in growing and shrinking
  - All nodes have the same role and functionality
  - Harder to manage because all machines are ***autonomous*** and ***loosely coupled***



MAIN ISSUES

- **Data Layout Issues**
  - Data partitioning and fragmentation
  - Data replication

- **Query Processing and Distributed Transactions**
  - Distributed join
  - Transaction atomicity using two-phase commit
  - Transaction serializability using distributed locking

MAIN ISSUES

- **Data Layout Issues**
  - Data partitioning and fragmentation
  - Data replication
- **Query Processing and Distributed Transactions**
  - Distributed join
  - Transaction atomicity using two-phase commit
  - Transaction serializability using distributed locking

FRAGMENTATION

- **How to divide the data? Can't we just distribute relations?**

- **What is a reasonable unit of distribution?**
  - **relation**
    - views are subsets of relations
    - extra communication
    - Less parallelism
  - **fragments of relations (sub-relations)**
    - concurrent execution of a number of transactions that access different portions of a relation
    - views that cannot be defined on a single fragment will require extra processing
    - semantic data control (especially integrity enforcement) more difficult

FRAGMENTATION ALTERNATIVES – HORIZONTAL

$PROJ_1$: projects with budgets less than $200,000

$PROJ_2$: projects with budgets greater than or equal to $200,000

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris
P5	CAD/CAM	500000	Boston

$PROJ_1$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York

Stored in London

$PROJ_2$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris
P5	CAD/CAM	500000	Boston

Stored in Boston

FRAGMENTATION ALTERNATIVES – VERTICAL

$PROJ_1$: information about project budgets

$PROJ_2$: information about project names and locations

PROJ

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal
P2	Database Develop.	135000	New York
P3	CAD/CAM	250000	New York
P4	Maintenance	310000	Paris
P5	CAD/CAM	500000	Boston

Horizontal partitioning is more common

$PROJ_1$

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000
P5	500000

Stored in London

$PROJ_2$

PNO	PNAME	LOC
P1	Instrumentation	Montreal
P2	Database Develop.	New York
P3	CAD/CAM	New York
P4	Maintenance	Paris
P5	CAD/CAM	Boston

Stored in Boston

CORRECTNESS OF FRAGMENTATION

- **Completeness**
  - Decomposition of relation R into fragments $R_1, R_2, \ldots, R_n$ is complete if and only if each data item in R can also be found in some R_i

- **Reconstruction (Lossless)**
  - If relation R is decomposed into fragments $R_1, R_2, \ldots, R_n$, then there should exist some relational operator ∇ such that

$$R = \nabla_{1 \leq i \leq n} R_i$$

- **Disjointness (Non-overlapping) – not mandatory**
  - If relation R is decomposed into fragments $R_1, R_2, \ldots, R_n$, and data item d_i is in R_j, then d_i should not be in any other fragment R_k $(k \neq j)$.

REPLICATION ALTERNATIVES

- Non-replicated
  - partitioned : each fragment resides at only one site
- Replicated
  - fully replicated : each fragment at each site
  - partially replicated : each fragment at some of the sites
- Rule of thumb:

If $\frac{\text{read - only queries}}{\text{update queries}} \geq 1$ replication is advantageous,

otherwise replication may cause problems

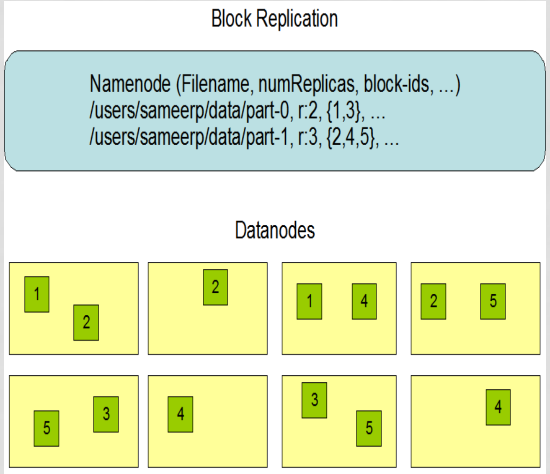
DATA REPLICATION

- **Pros:**
  - Improves availability
  - Distributes load
  - Reads are cheaper
- **Cons:**
  - N times more updates
  - N times more storage

Catalog Management

- Catalog is needed to keep track of the location of each fragment & replica
- Catalog itself can be centralized or distributed

Similar to NameNode in Hadoop



UPDATING REPLICAS

- Synchronous Replication: All copies of modified relation (fragment) must be updated before modifying Xact commits.
  - Data distribution is made transparent to users.
- Asynchronous Replication: Copies of modified relation only periodically updated; different copies may get out of synch in meantime.
  - Users must be aware of data distribution.
- **Current products tend to follow later approach.**

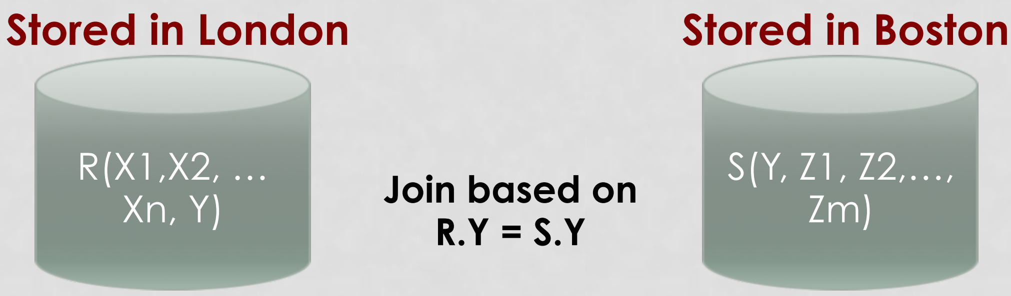
COMPARISON OF REPLICATION ALTERNATIVES

	Full-replication	Partial-replication	Partitioning
QUERY PROCESSING	Easy	← Same Difficulty →	
DIRECTORY MANAGEMENT	Easy or Non-existant	← Same Difficulty →	
CONCURRENCY CONTROL	Moderate	Difficult	Easy
RELIABILITY	Very high	High	Low
REALITY	Possible application	Realistic	Possible application

MAIN ISSUES

- **Data Layout Issues**
  - Data partitioning and fragmentation
  - Data replication
- **Query Processing and Distributed Transactions**
  - Distributed join
  - Transaction atomicity using two-phase commit
  - Transaction serializability using distributed locking

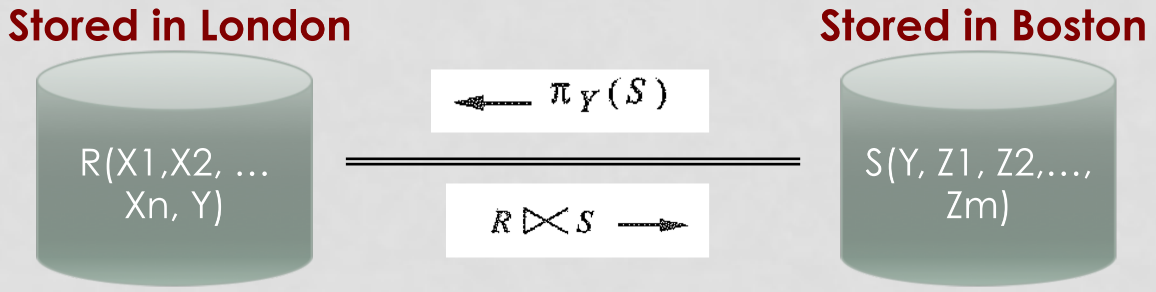
DISTRIBUTED JOIN R(X,Y) $\bowtie$ S(Y,Z)



- **Option 1:** Send R to S's location and join their
- **Option 2:** Send S to R's location and join their
- ***Communication cost is expensive, too much data to send***

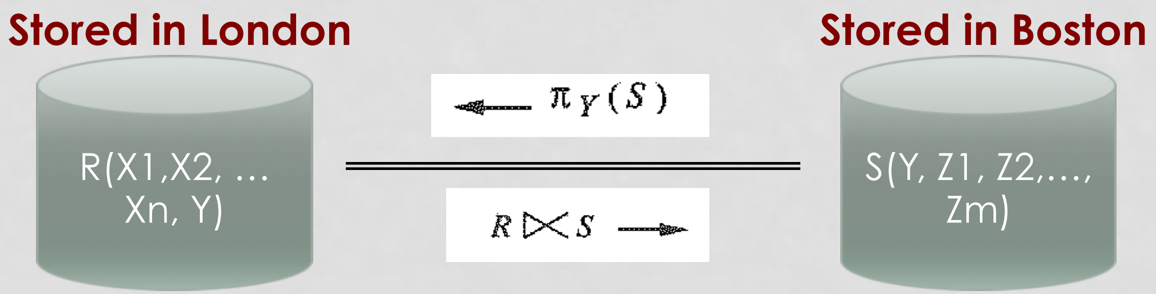
- **Is there a better option ???**
  - **Semi Join**
  - **Bloom Join**

SEMI-JOIN



- Send only S.Y column to R’s location
- Do the join based on Y columns in R’s location ***(Semi Join)***
- Send the records of R that will join (without duplicates) to S’s location
- Perform the final join in S’s location

IS SEMI-JOIN EFFECTIVE



Depends on many factors:

- If the size of Y attribute is small compared to the remaining attributes in R and S
- If the join selectivity is high → $R \ltimes S$ is small
- If there are many duplicates that can be eliminated

BLOOM JOIN

- Build a bit vector of size K in R's location (all 0's)

0	0	1	1	…	0	0	1

- **For every record in R, use a hash function(s) based on Y value (return from 1 to K)**
  - Each function hashes Y to a bit in the bit vector. Set this bit to 1

- Send the bit vector to S's location

- **S will use the same hash function(s) to hash its Y values**
  - If the hashing matched with 1's in all its hashing positions, then this Y is candidate for Join
  - Otherwise, not candidate for join
  - Send S's records having candidate Y's to R's location for join

SELECTING ALTERNATIVES

```
SELECT    ENAME
FROM      EMP,ASG
WHERE     EMP.ENO = ASG.ENO
AND       DUR > 37
```

Π Project
σ Select
$\times$ Join

Strategy 1

$$\Pi_{ENAME}(\sigma_{DUR>37 \wedge EMP.ENO=ASG.ENO}\ (EMP \times ASG))$$

Strategy 2

$$\Pi_{ENAME}(EMP \bowtie_{ENO}\ (\sigma_{DUR>37}\ (ASG)))$$

Strategy 2 avoids Cartesian product, so is “better”

MAIN ISSUES

- **Data Layout Issues**
  - Data partitioning and fragmentation
  - Data replication

- **Query Processing and Distributed Transactions**
  - Distributed join
  - **Transaction atomicity using two-phase commit**
  - Transaction serializability using distributed locking

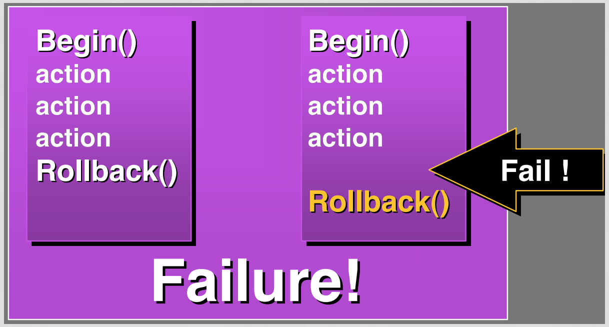
TRANSACTIONS

- **A Transaction is an atomic sequence of actions in the Database (reads and writes)**

- **Each Transaction has to be executed *completely*, and must leave the Database in a consistent state**

- **If the Transaction fails or aborts midway, then the Database is "rolled back" to its initial consistent state (before the Transaction began)**

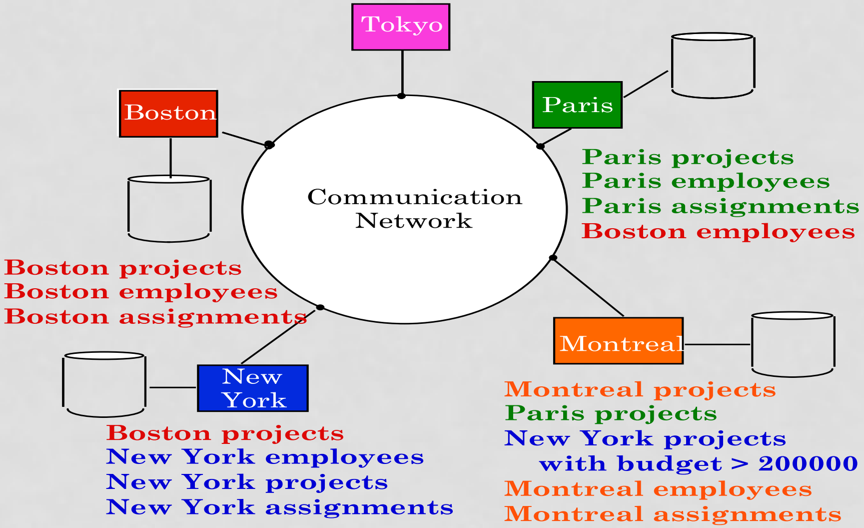
ACID Properties of Transactions



ATOMICITY IN DISTRIBUTED DBS

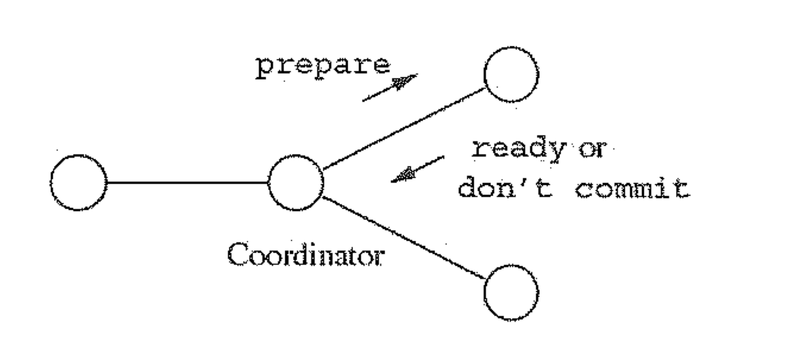
- **One transaction *T* may touch many sites**
  - T has several components T1, T2, …Tm
  - Each Tk is running (reading and writing) at site k
  - **How to make *T* is atomic ????**
    - Either T1, T2, …, Tm complete or None of them is executed

- **Two-Phase Commit techniques is used**



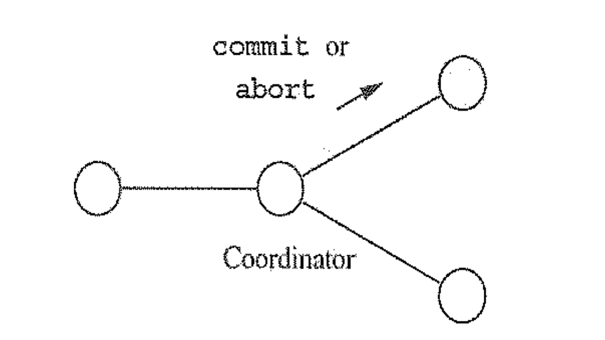
TWO-PHASE COMMIT

- **Phase 1**
  - Site that initiates T is the ***coordinator***
  - When coordinator wants to commit (complete T), it sends a *"prepare T"* msg to all participant sites
  - Every other site receiving *"prepare T"*, either sends *"ready T"* or *"don't commit T"*
    - A site can wait for a while until it reaches a decision (Coordinator will wait reasonable time to hear from the others)
  - **These msgs are written to local logs**



TWO-PHASE COMMIT (CONT'D)

- **Phase 2**
  - ***IF coordinator received all "ready T"***
    - Remember no one committed yet
    - Coordinator sends *"commit T"* to all participant sites
    - Every site receiving *"commit T"* commits its transaction
  - ***IF coordinator received any "don't commit T"***
    - *Coordinator sends "abort T"* to all participant sites
    - Every site receiving *"abort T"* commits its transaction



  - **These msgs are written to local logs**

- Straightforward if no failures happen
- In case of failure logs are used to ensure **ALL** are done or **NONE**

Example 1: What if one sites in Phase 1 replied "don't commit T", and then crashed???

Example 2: What if all sites in Phase 1 replied "ready T", then one site crashed???

MAIN ISSUES

- **Data Layout Issues**
  - Data partitioning and fragmentation
  - Data replication

- **Query Processing and Distributed Transactions**
  - Distributed join
  - Transaction atomicity using two-phase commit
  - **Transaction serializability using distributed locking**

DATABASE LOCKING

- *Locking mechanisms are used to prevent concurrent transactions from updating the same data at the same time*

- *Reading(x) → shared lock on x*
- *Writing(x) → exclusive lock on x*
- **More types of locks exist for efficiency**

What you have

What you request

	Shared lock	Exclusive lock
Shared lock	Yes	No
Exclusive lock	No	No

In Distributed DBs:

- x may be replicated in multiple sites (not one place)
- The transactions reading or writing x may be running at different sites

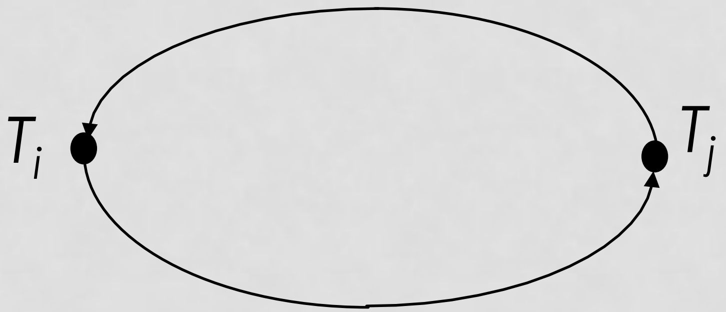
DISTRIBUTED LOCKING

- **Centralized approach**
  - One dedicated site managing all locks
  - Cons: bottleneck, not scalable, single point of failure
- **Primary-Copy approach**
  - Every item in the database, say x, has a primary site, say Px
  - Any transaction running any where, will ask Px for lock on x
- **Fully Distributed approach**
  - To read, lock any copy of x
  - To write, lock all copies of x
  - Variations exists to balance the cots of read and write op.

Deadlocks are very possible. How to resolve them???
Using timeout: After waiting for a while for a lock, abort and start again

DEADLOCK

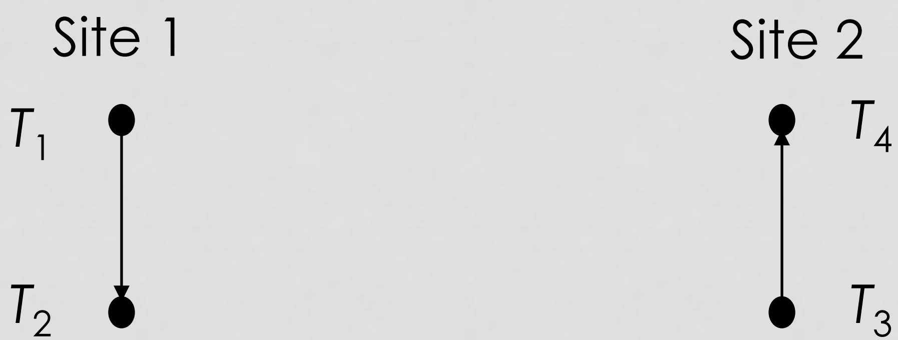
- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- TO-based algorithms that involve waiting may cause deadlocks.
- Wait-for graph
  - If transaction T_i waits for another transaction T_j to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG.



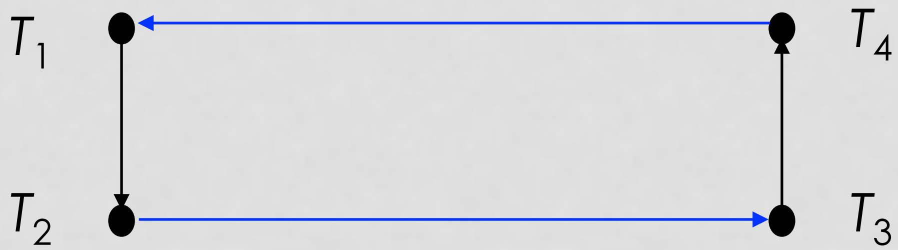
LOCAL VS. GLOBAL WFG

Assume T_1 and T_2 run at site 1, T_3 and T_4 run at site 2. Also assume T_3 waits for a lock held by T_4 which waits for a lock held by T_1 which waits for a lock held by T_2 which, in turn, waits for a lock held by T_3.

Local WFG



Global WFG



DEADLOCK MANAGEMENT

- **Ignore**
  - Let the application programmer deal with it, or restart the system
- **Prevention**
  - Guaranteeing that deadlocks can never occur in the first place. Check transaction when it is initiated. Requires no run time support.
- **Avoidance**
  - Detecting potential deadlocks in advance and taking action to insure that deadlock will not occur. Requires run time support.
- **Detection and Recovery**
  - Allowing deadlocks to form and then finding and breaking them. As in the avoidance scheme, this requires run time support.

DEADLOCK PREVENTION

- All resources which may be needed by a transaction must be predeclared.
  - The system must guarantee that none of the resources will be needed by an ongoing transaction.
  - Resources must only be reserved, but not necessarily allocated a priori
  - Unsuitability of the scheme in database environment
  - Suitable for systems that have no provisions for undoing processes.

DEADLOCK AVOIDANCE

- Transactions are not required to request resources a priori.
- Transactions are allowed to proceed unless a requested resource is unavailable.
- In case of conflict, transactions may be allowed to wait for a fixed time interval.
- Order either the data items or the sites and always request locks in that order.
- More attractive than prevention in a database environment.

DEADLOCK DETECTION

- Transactions are allowed to wait freely.
- Wait-for graphs and cycles.
- Topologies for deadlock detection algorithms
  - Centralized
  - Distributed
  - Hierarchical

SUMMARY OF DISTRIBUTED DBS

- ***Promises of DDBMSs***
  - Transparent management of distributed, fragmented, and replicated data
  - Improved reliability/availability through distributed transactions
  - Improved performance
  - Easier and more economical system expansion

- ***Classification of DDBMS***
  - Homogeneous vs. Heterogeneous
  - Client-Sever vs. Collaborative Servers vs. Peer-to-Peer

SUMMARY OF DISTRIBUTED DBS (CONT'D)

- **Data Layout Issues**
  - Data partitioning and fragmentation
  - Data replication

- **Query Processing and Distributed Transactions**
  - Distributed join
  - Transaction atomicity using two-phase commit
  - Transaction serializability using distributed locking