# Distributed Database Management Systems

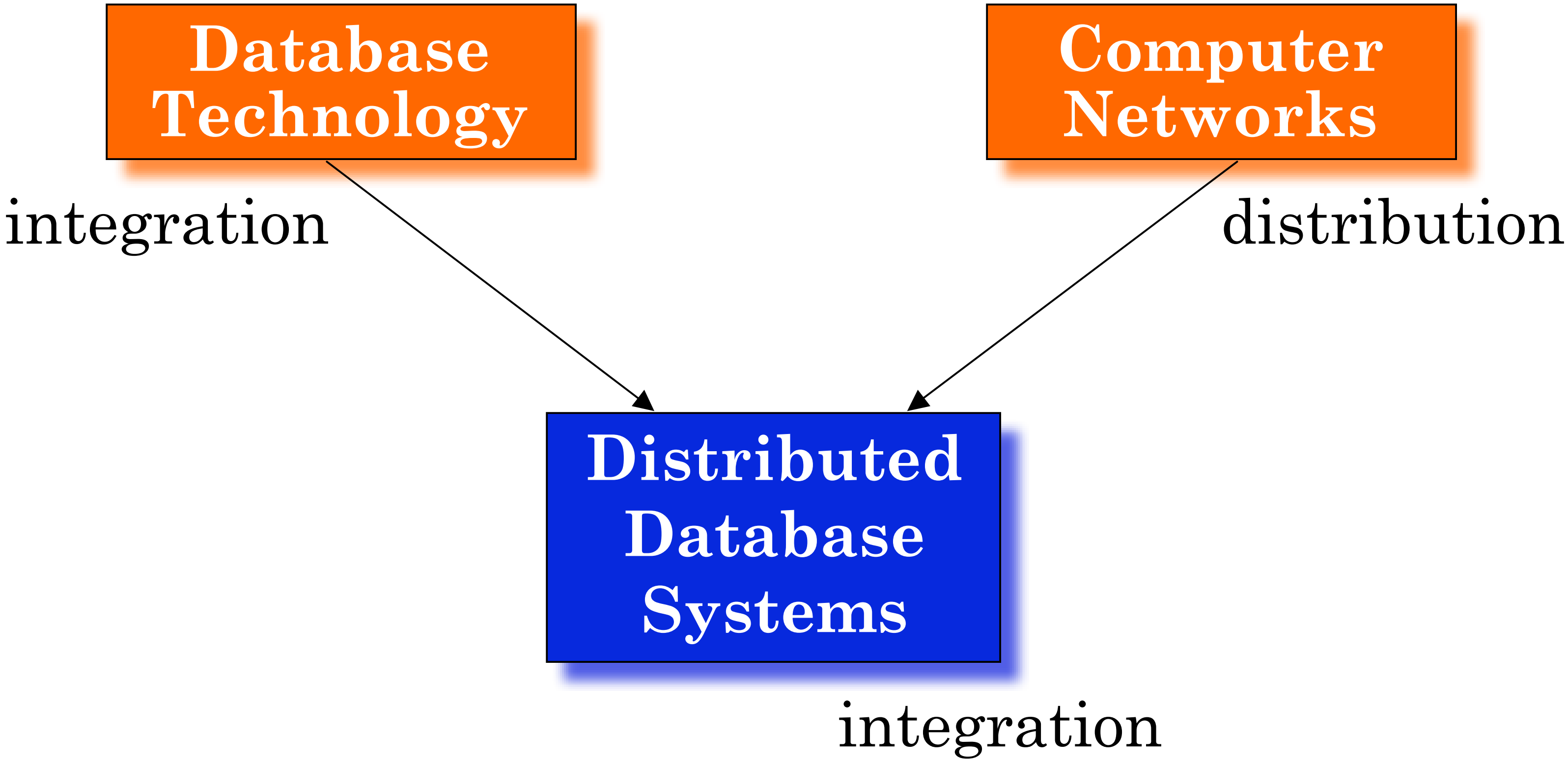
# Outline

- Introduction
- Distributed DBMS Architecture
- Distributed Database Design
- Distributed Query Processing
- Distributed Concurrency Control
- Distributed Reliability Protocols

# Outline

- Introduction
  - What is a distributed DBMS
  - Problems
  - Current state-of-affairs
- Distributed DBMS Architecture
- Distributed Database Design
- Distributed Query Processing
- Distributed Concurrency Control
- Distributed Reliability Protocols

# Motivation



Database Technology

Computer Networks

integration

distribution

Distributed Database Systems

integration

**integration ≠ centralization**

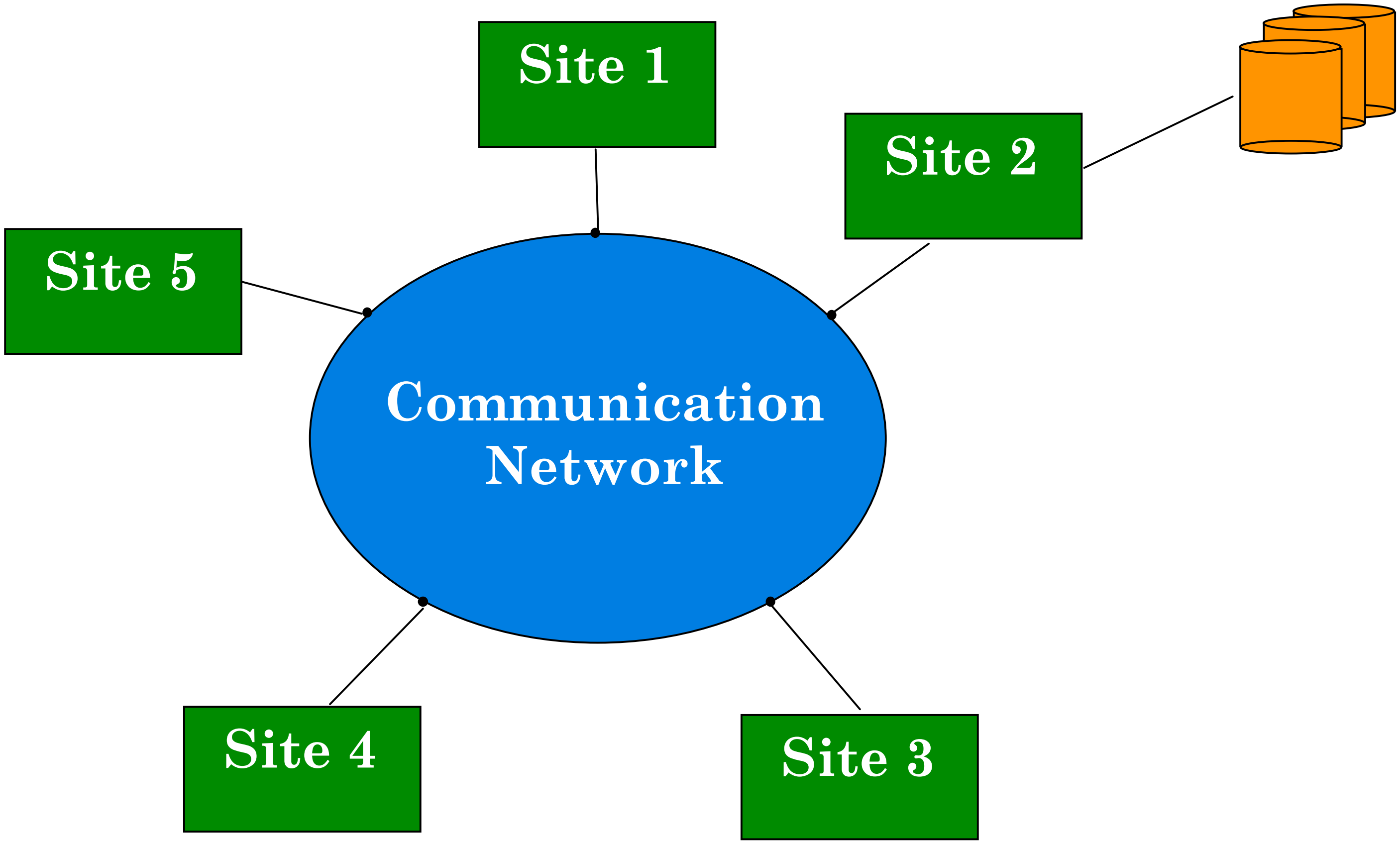
# What is a Distributed Database System?

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

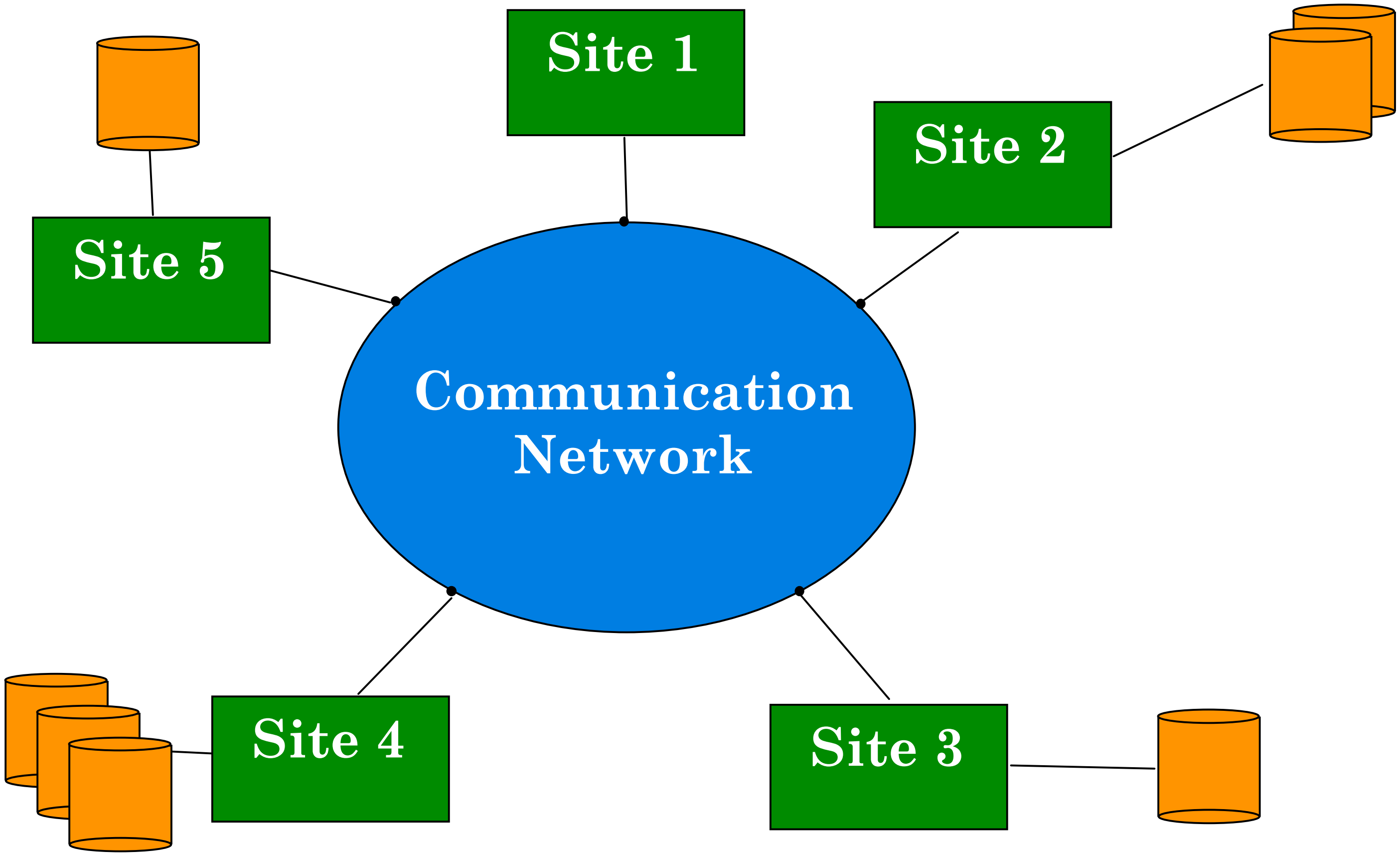
A distributed database management system (D–DBMS) is the software that manages the DDB and provides an access mechanism that makes this distribution transparent to the users.

Distributed database system (DDBS) = DDB + D–DBMS

# Centralized DBMS on Network



# Distributed DBMS Environment



# Implicit Assumptions

- Data stored at a number of sites ⇨ each site *logically* consists of a single processor.
- Processors at different sites are interconnected by a computer network ⇨ no multiprocessors
  - parallel database systems
- Distributed database is a database, not a collection of files ⇨ data logically related as exhibited in the users' access patterns
  - relational data model
- D-DBMS is a full-fledged DBMS
  - not remote file system, not a TP system

# Distributed DBMS Promises

❶ Transparent management of distributed, fragmented, and replicated data

❷ Improved reliability/availability through distributed transactions

❸ Improved performance

❹ Easier and more economical system expansion

# Transparency

- Transparency is the separation of the higher level semantics of a system from the lower level implementation issues.
- Fundamental issue is to provide

  **data independence**

  in the distributed environment

  - Network (distribution) transparency
  - Replication transparency
  - Fragmentation transparency
    - horizontal fragmentation: selection
    - vertical fragmentation: projection
    - hybrid

# Example

EMP

| ENO | ENAME | TITLE |
|---|---|---|
| E1 | J. Doe | Elect. Eng. |
| E2 | M. Smith | Syst. Anal. |
| E3 | A. Lee | Mech. Eng. |
| E4 | J. Miller | Programmer |
| E5 | B. Casey | Syst. Anal. |
| E6 | L. Chu | Elect. Eng. |
| E7 | R. Davis | Mech. Eng. |
| E8 | J. Jones | Syst. Anal. |

ASG

| ENO | PNO | RESP | DUR |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P3 | Consultant | 10 |
| E3 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P5 | Engineer | 23 |
| E8 | P3 | Manager | 40 |

PROJ

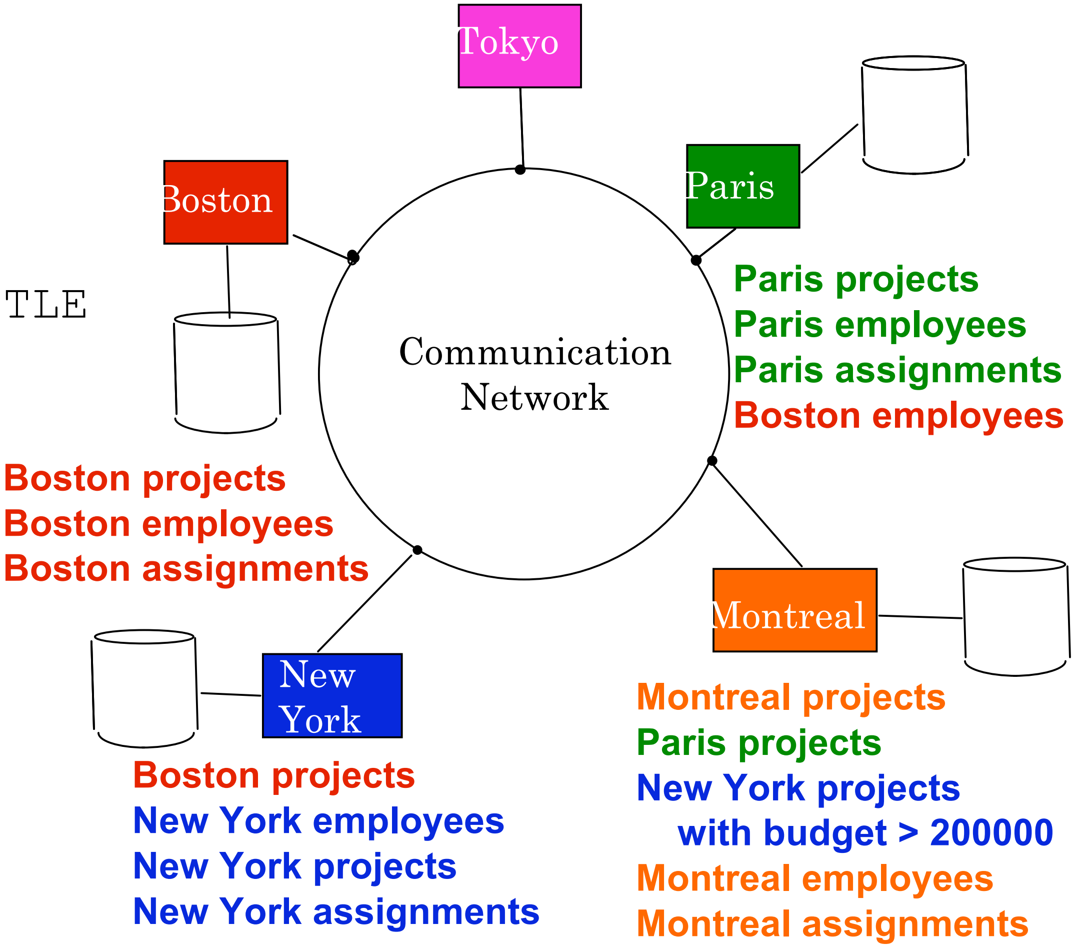
| PNO | PNAME | BUDGET |
|---|---|---|
| P1 | Instrumentation | 150000 |
| P2 | Database Develop. | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |

PAY

| TITLE | SAL |
|---|---|
| Elect. Eng. | 40000 |
| Syst. Anal. | 34000 |
| Mech. Eng. | 27000 |
| Programmer | 24000 |

# Transparent Access

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



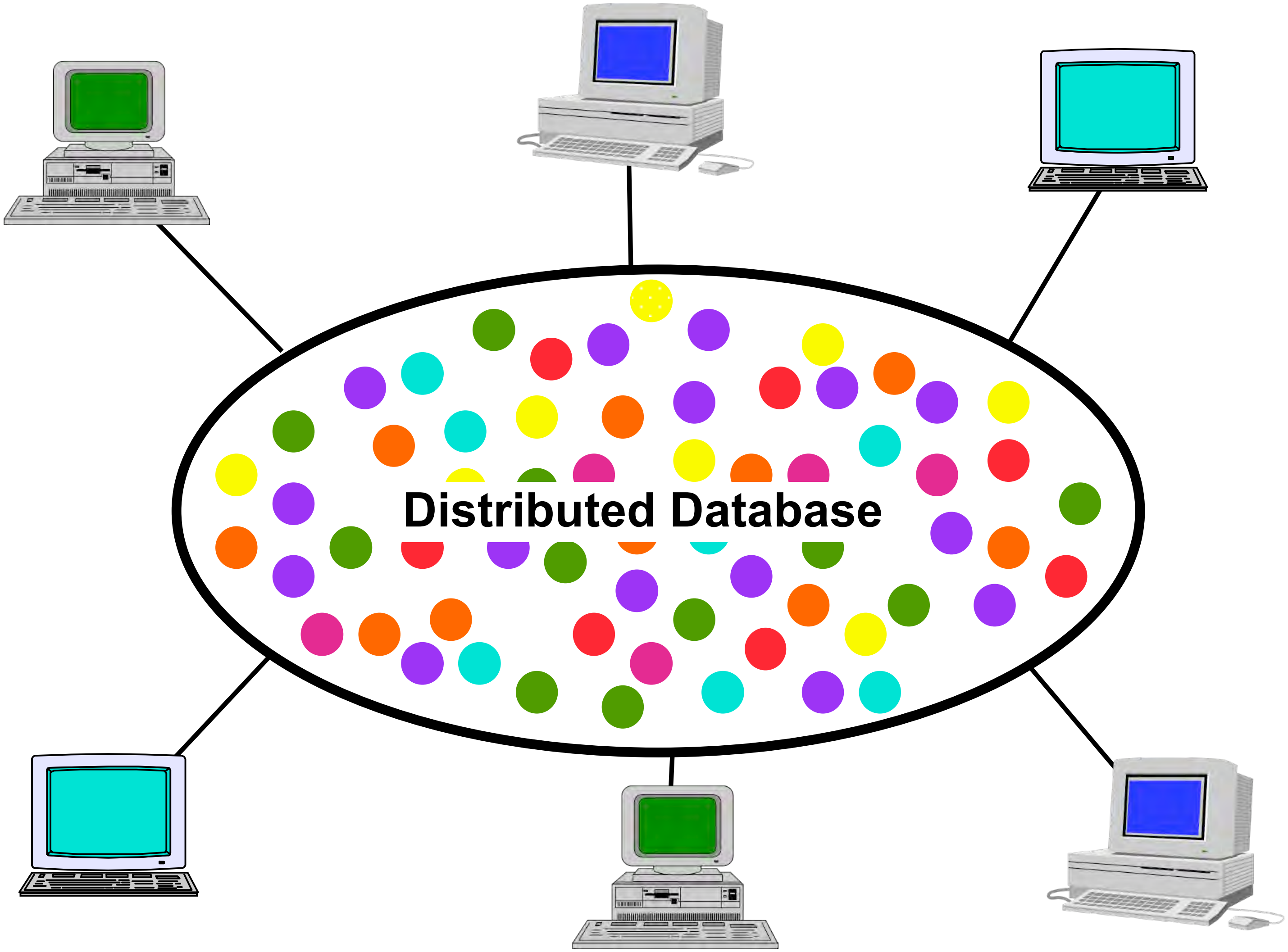
Paris projects
Paris employees
Paris assignments
Boston employees

Boston projects
Boston employees
Boston assignments

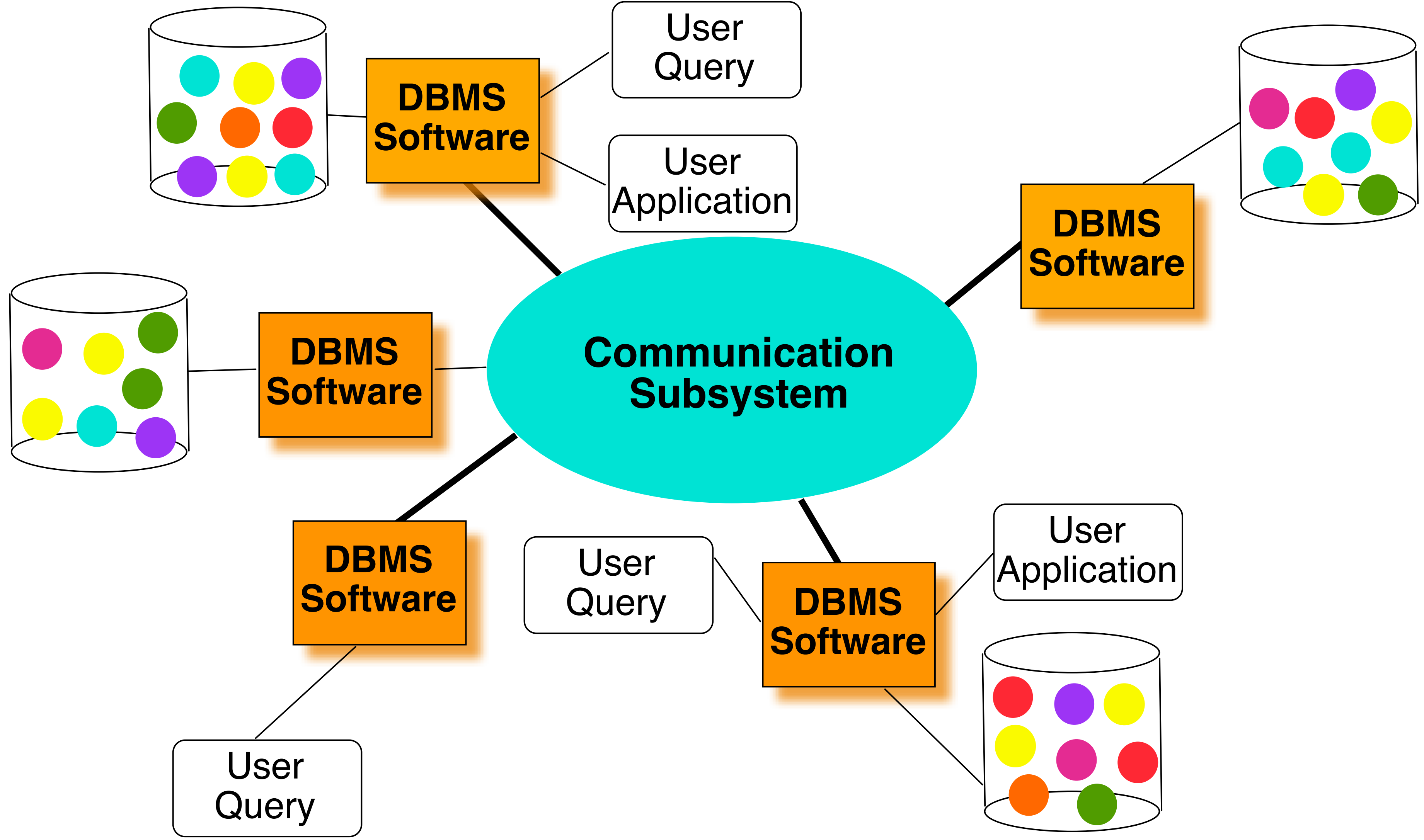
Boston projects
New York employees
New York projects
New York assignments

Montreal projects
Paris projects
New York projects
with budget > 200000
Montreal employees
Montreal assignments

# Distributed Database – User View



# Distributed DBMS - Reality



# Potentially Improved Performance

- Proximity of data to its points of use
  - Requires some support for fragmentation and replication
- Parallelism in execution
  - Inter-query parallelism
  - Intra-query parallelism

# Parallelism Requirements

- Have as much of the data required by *each* application at the site where the application executes
  - Full replication
- How about updates?
  - Updates to replicated data requires implementation of distributed concurrency control and commit protocols

# System Expansion

- Issue is database scaling
- Emergence of microprocessor and workstation technologies
  - Demise of Grosh's law
  - Client-server model of computing
- Data communication cost vs telecommunication cost

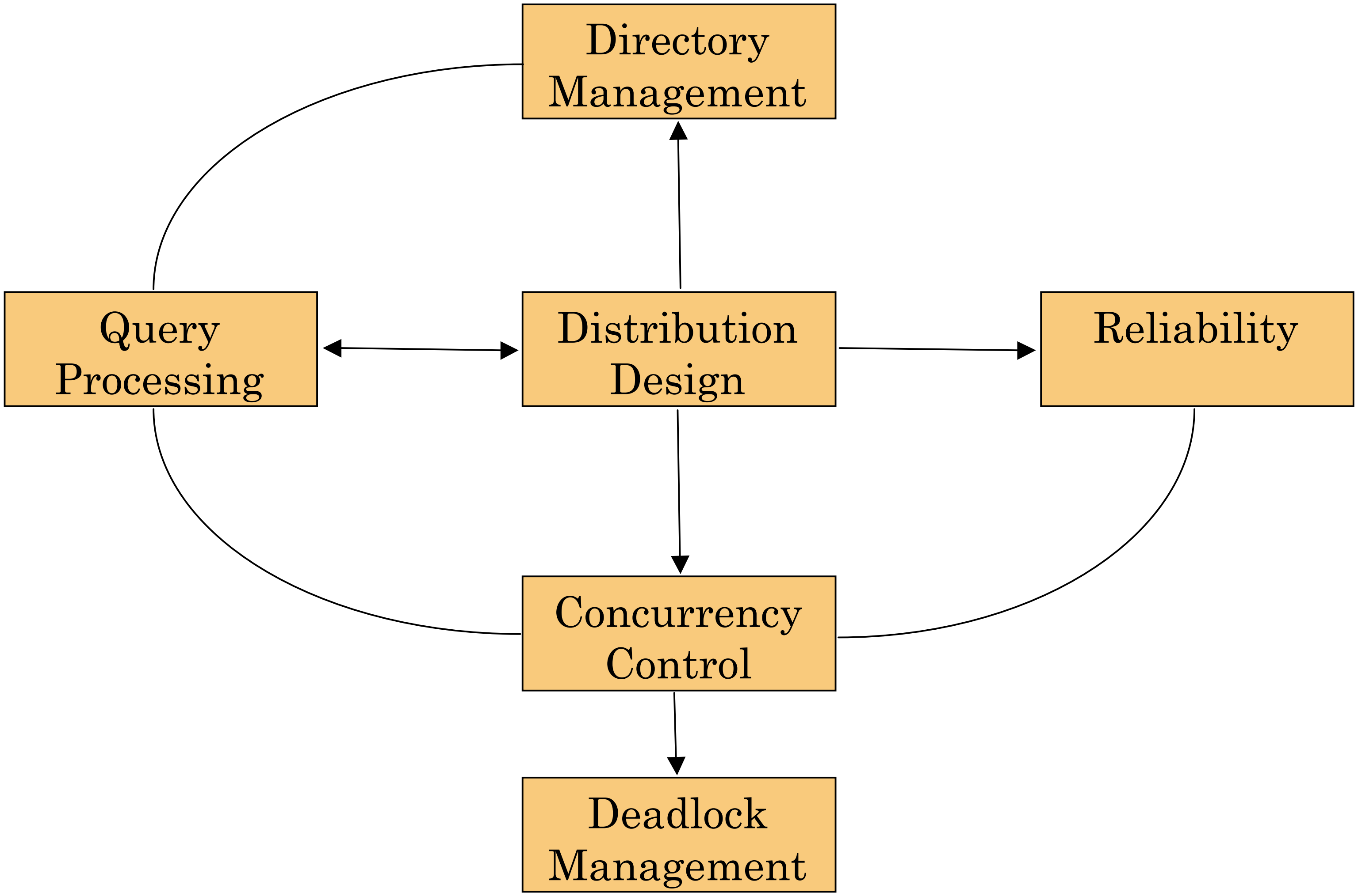
# Distributed DBMS Issues

- **Distributed Database Design**
  - how to distribute the database
  - replicated & non-replicated database distribution
  - a related problem in directory management
- **Query Processing**
  - convert user transactions to data manipulation instructions
  - optimization problem
  - min{cost = data transmission + local processing}
  - general formulation is NP-hard

# Distributed DBMS Issues

- **Concurrency Control**
  - synchronization of concurrent accesses
  - consistency and isolation of transactions' effects
  - deadlock management
- **Reliability**
  - how to make the system resilient to failures
  - atomicity and durability

# Relationship Between Issues



Directory Management

Query Processing

Distribution Design

Reliability

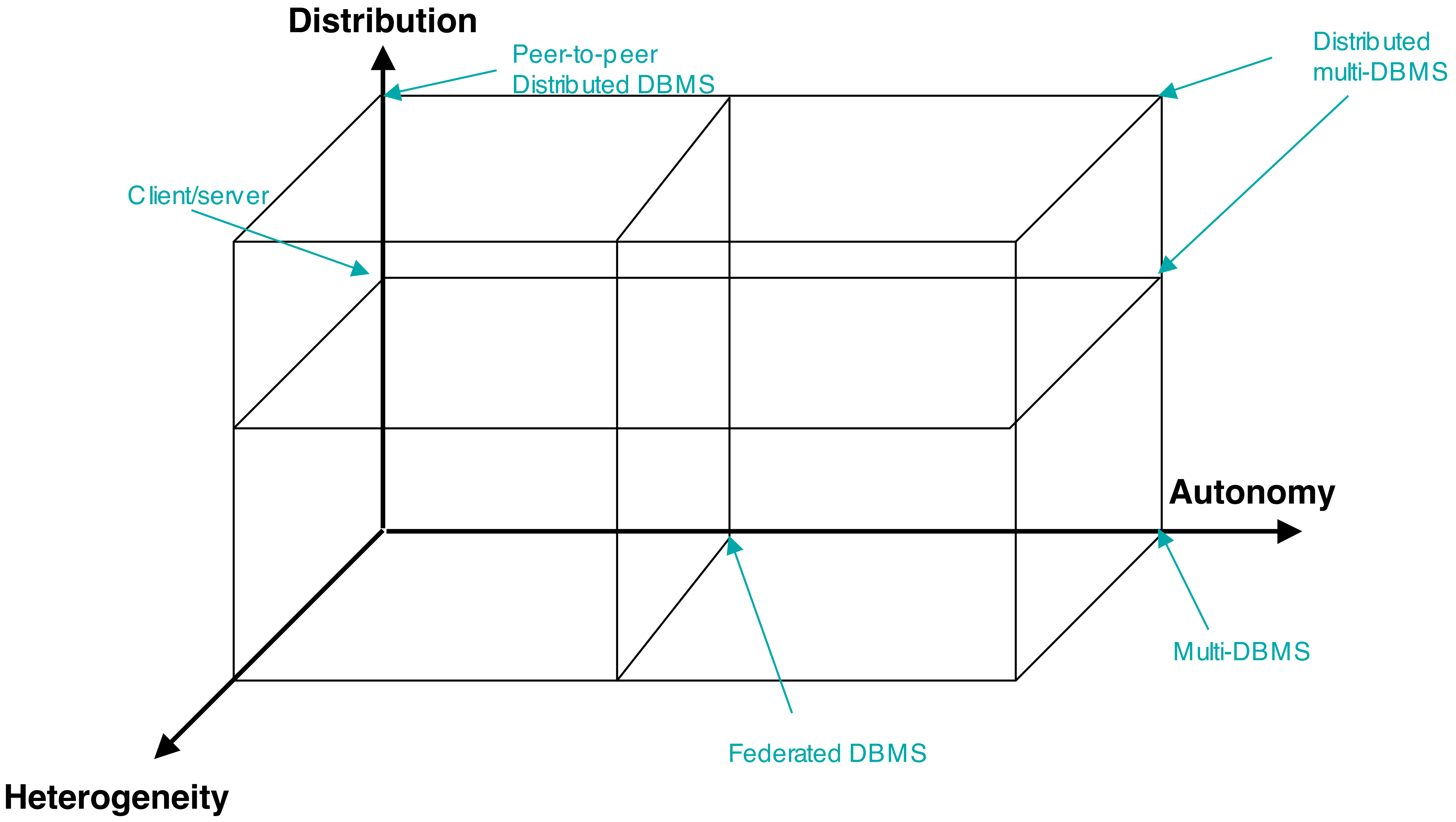
Concurrency Control

Deadlock Management

# Outline

- Introduction
- Distributed DBMS Architecture
  - Implementation Alternatives
  - Component Architecture
- Distributed Database Design
- Distributed Query Processing
- Distributed Concurrency Control
- Distributed Reliability Protocols

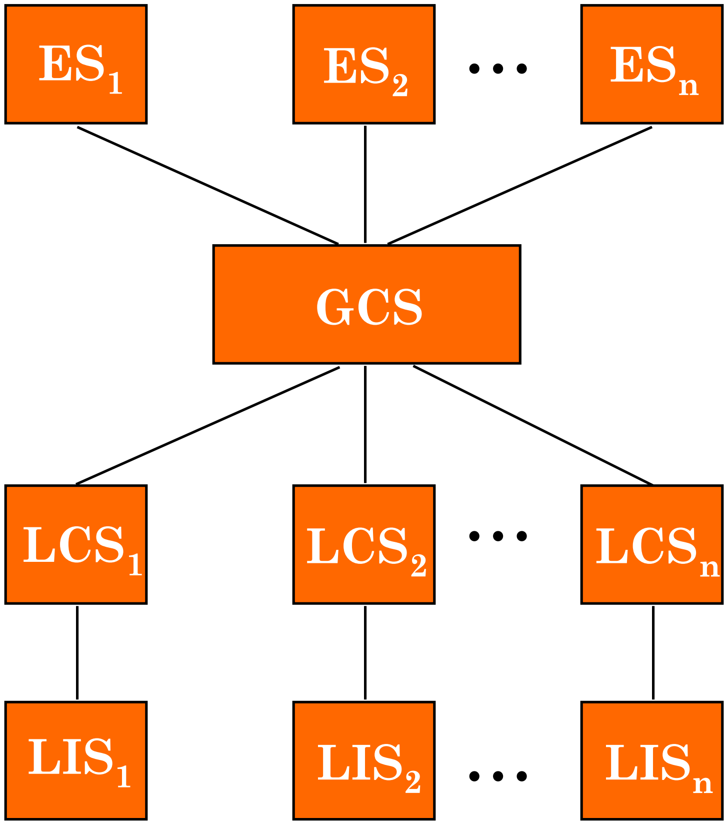
# DBMS Implementation Alternatives



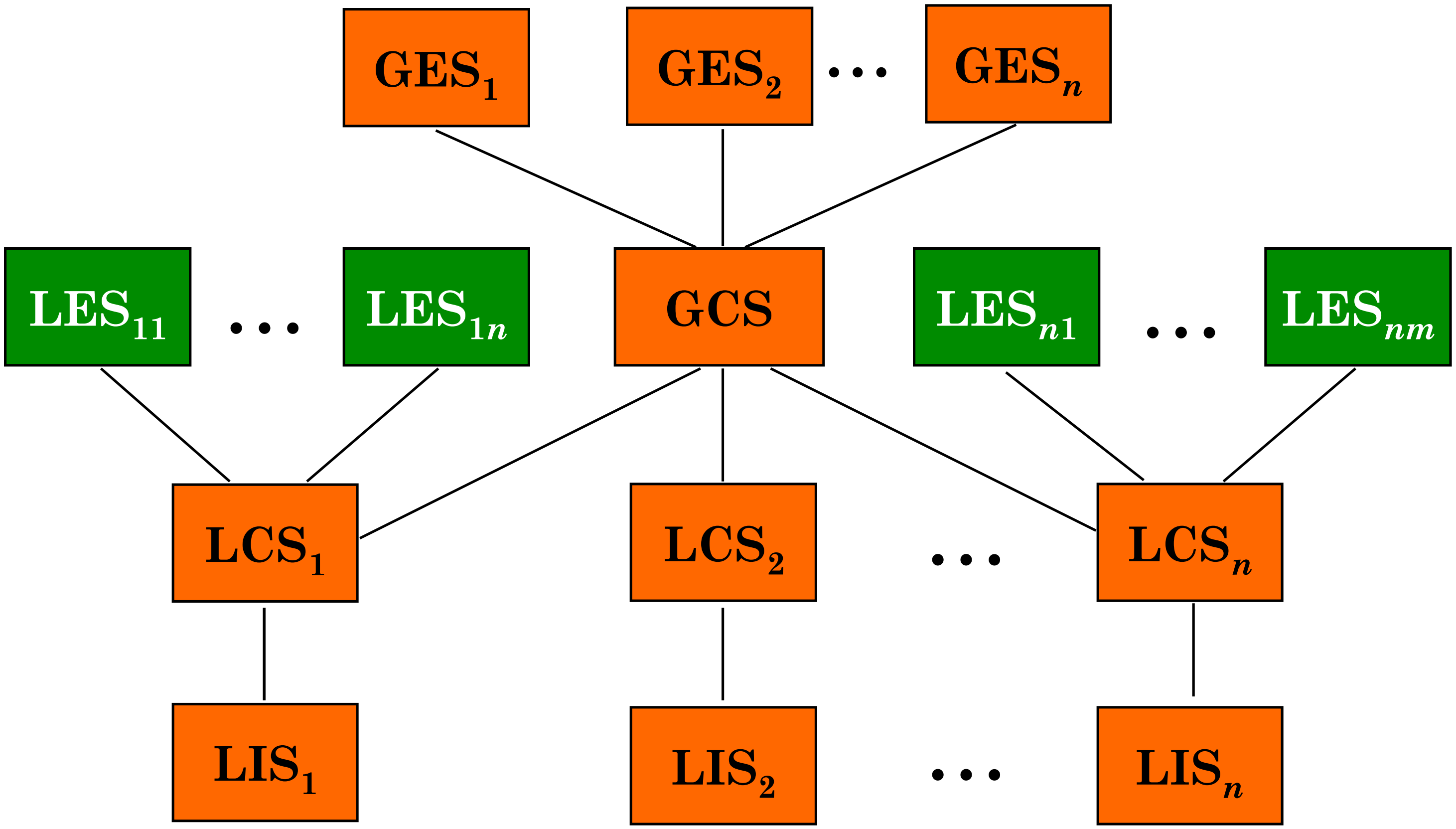
# Dimensions of the Problem

- Distribution
  - Whether the components of the system are located on the same machine or not
- Heterogeneity
  - Various levels (hardware, communications, operating system)
  - DBMS important one
    - data model, query language,transaction management algorithms
- Autonomy
  - Not well understood and most troublesome
  - Various versions
    - Design autonomy: Ability of a component DBMS to decide on issues related to its own design.
    - Communication autonomy: Ability of a component DBMS to decide whether and how to communicate with other DBMSs.
    - Execution autonomy: Ability of a component DBMS to execute local operations in any manner it wants to.

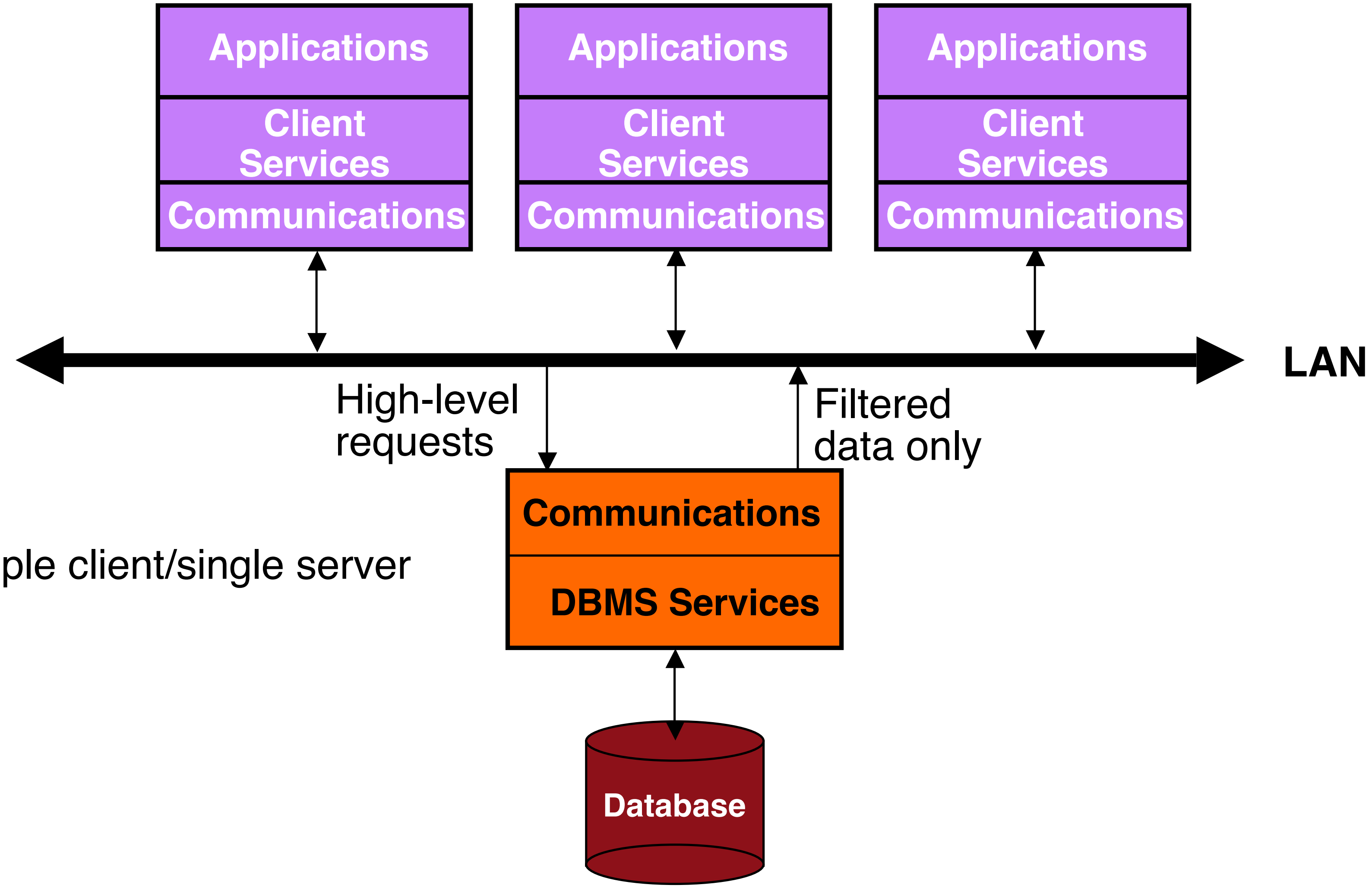
# Datalogical Distributed DBMS Architecture



# Datalogical Multi-DBMS Architecture

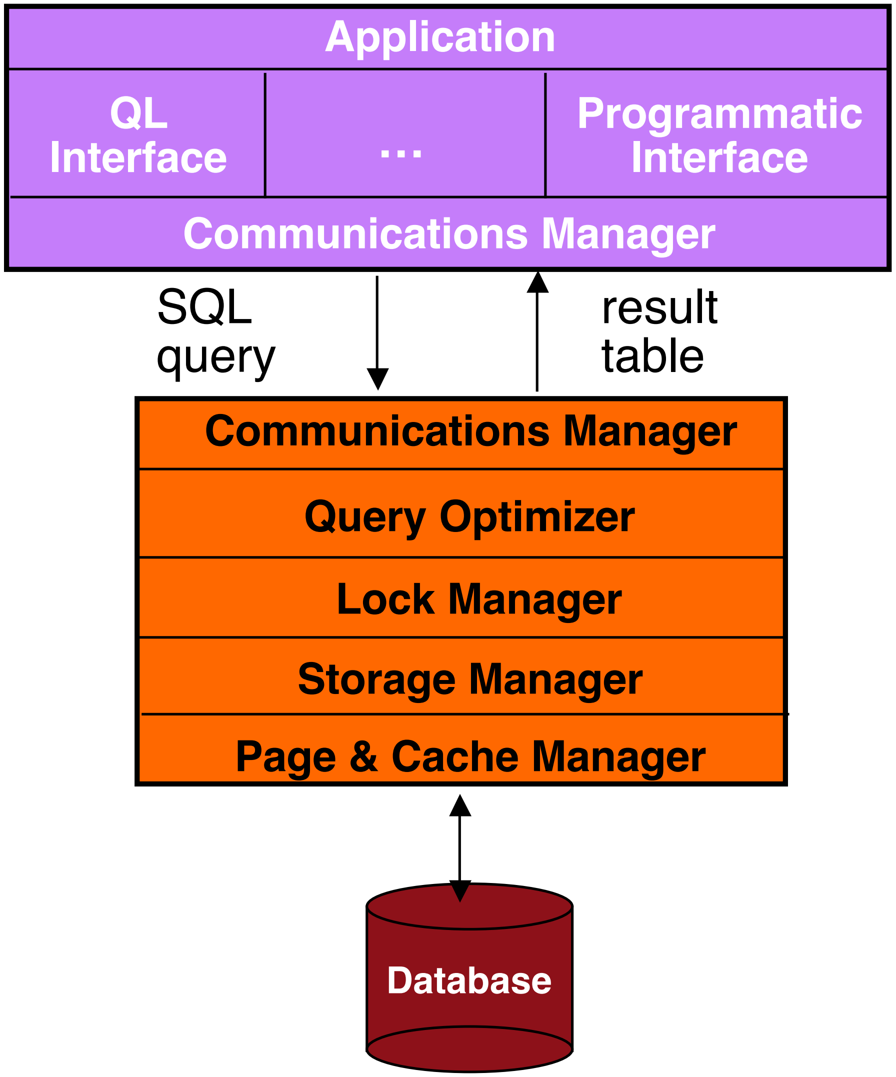


# Clients/Server



Multiple client/single server

# Task Distribution



# Advantages of Client-Server Architectures

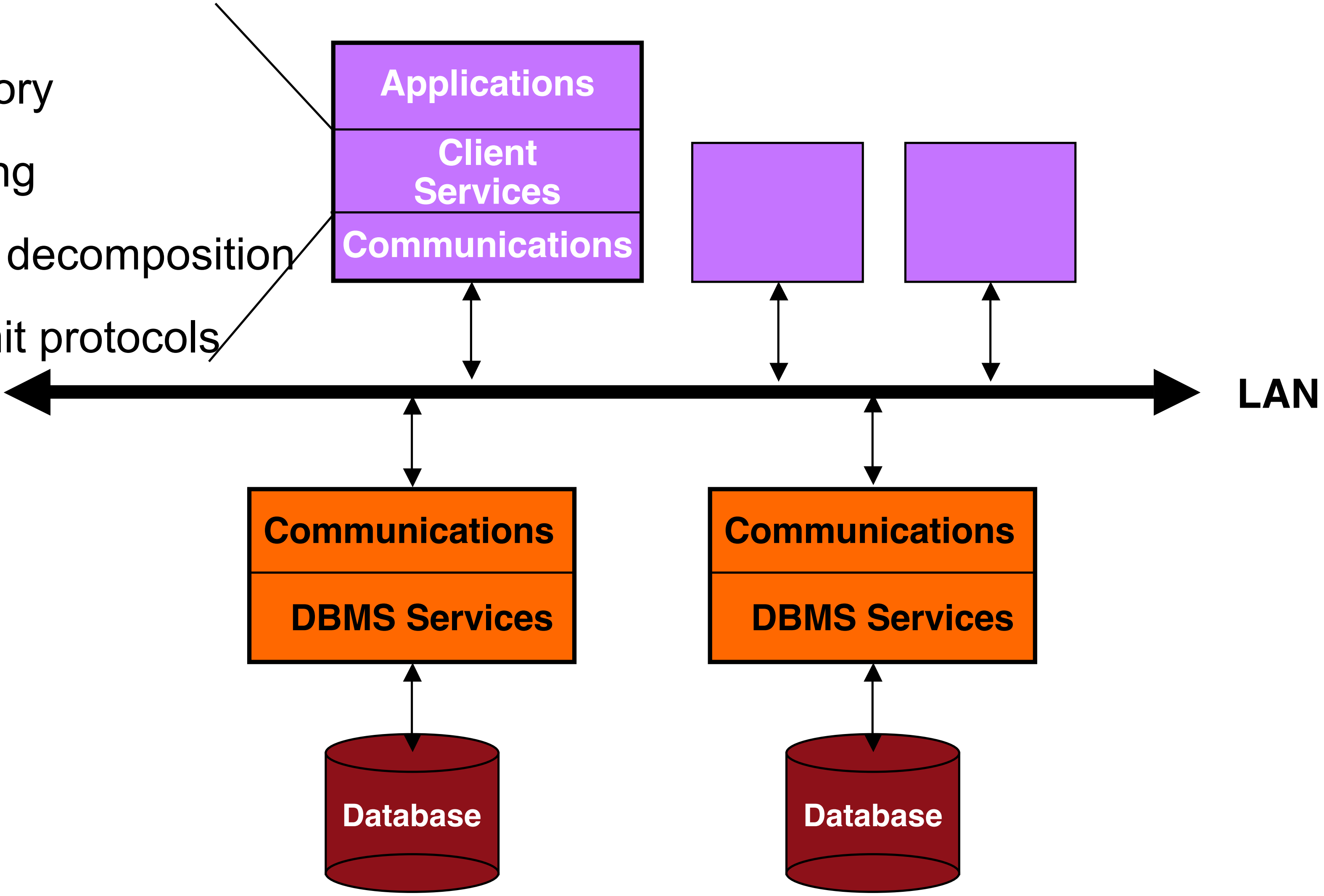
- More efficient division of labor
- Horizontal and vertical scaling of resources
- Better price/performance on client machines
- Ability to use familiar tools on client machines
- Client access to remote data (via standards)
- Full DBMS functionality provided to client workstations
- Overall better system price/performance

# Problems With Multiple-Client/Single Server

- Server forms bottleneck
- Server forms single point of failure
- Database scaling difficult

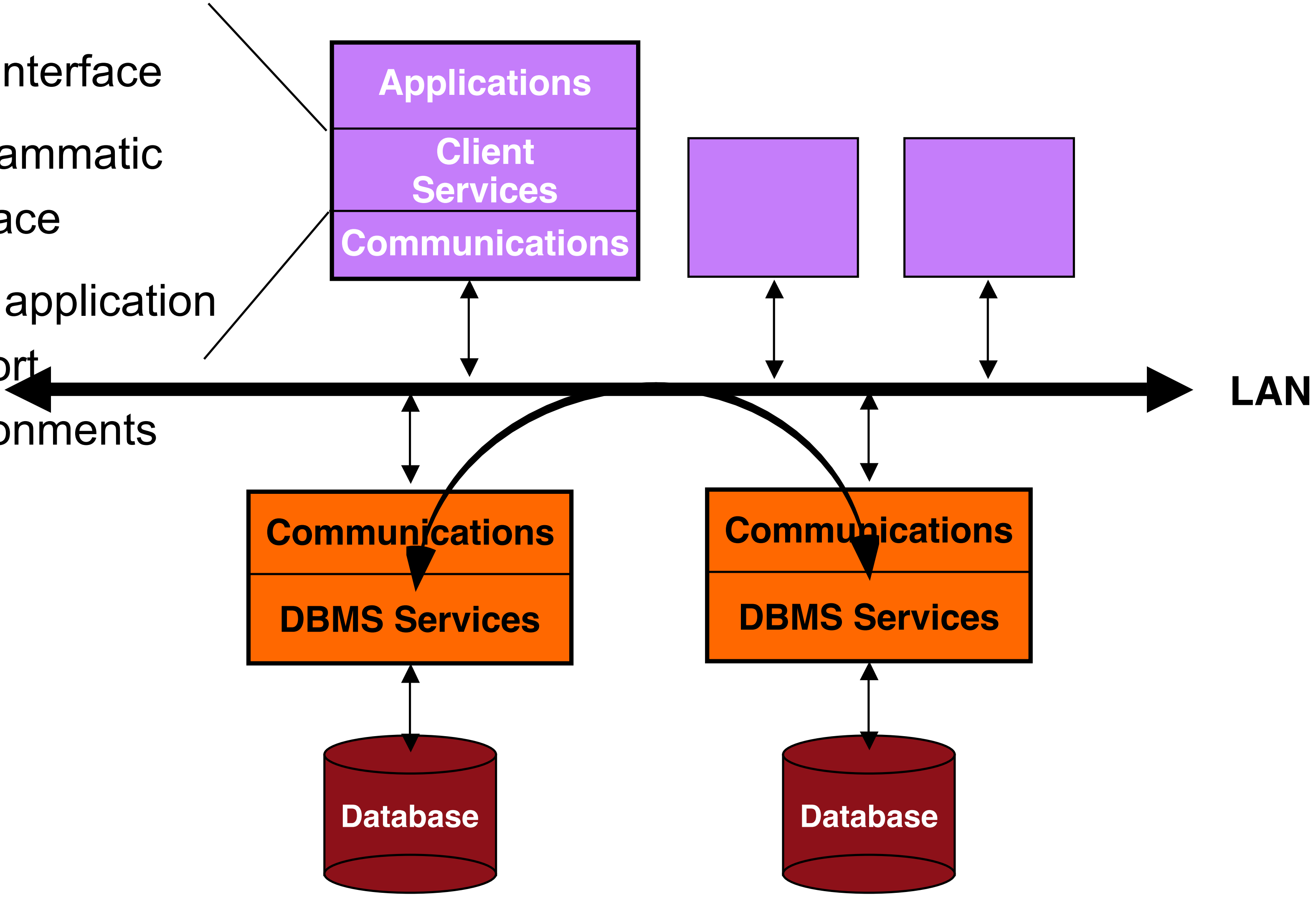
# Multiple Clients/Multiple Servers

- directory
- caching
- query decomposition
- commit protocols

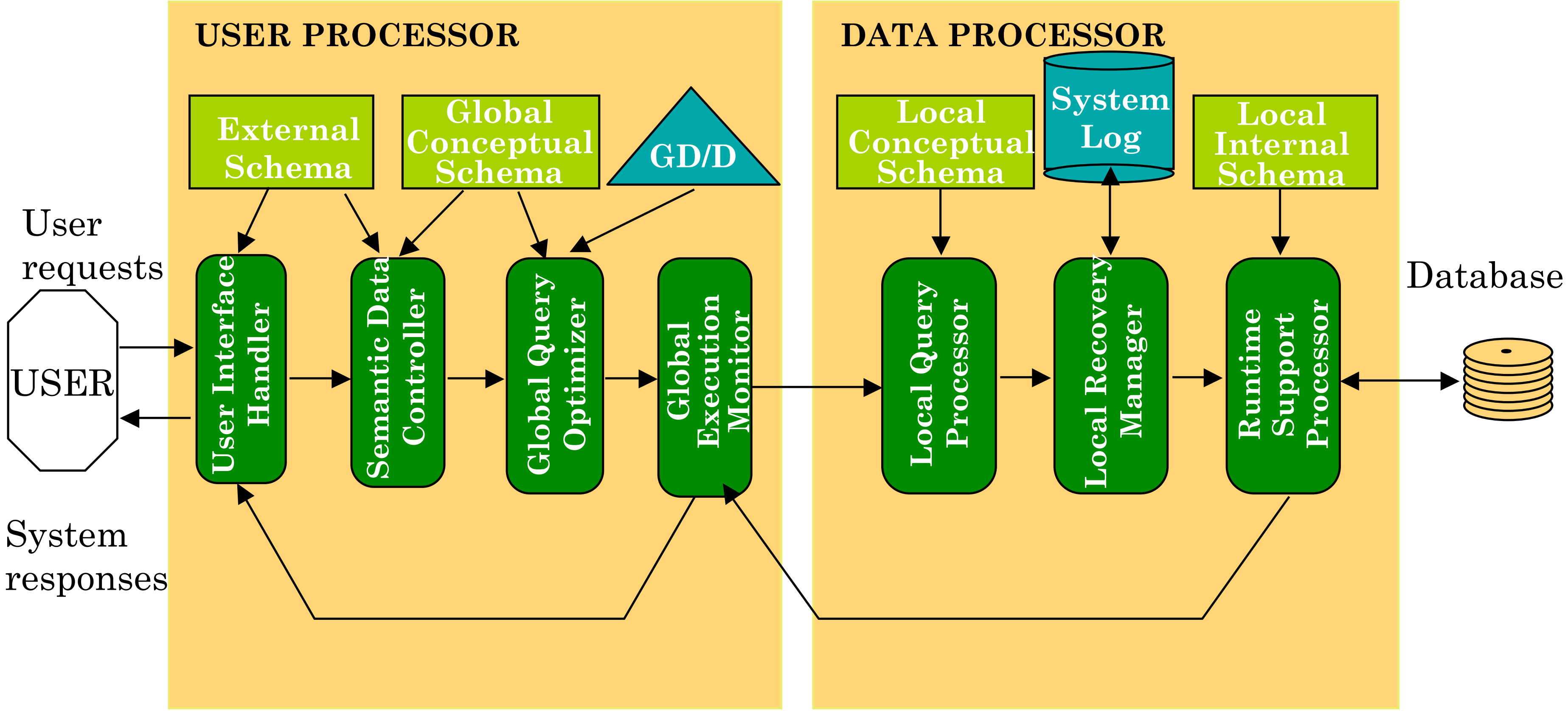


# Server-to-Server

- SQL interface
- programmatic interface
- other application support environments



# Peer-to-Peer Component Architecture



# Outline

- Introduction
- Distributed DBMS Architecture
- Distributed Database Design
  - Fragmentation
  - Data Placement
- Distributed Query Processing
- Distributed Concurrency Control
- Distributed Reliability Protocols

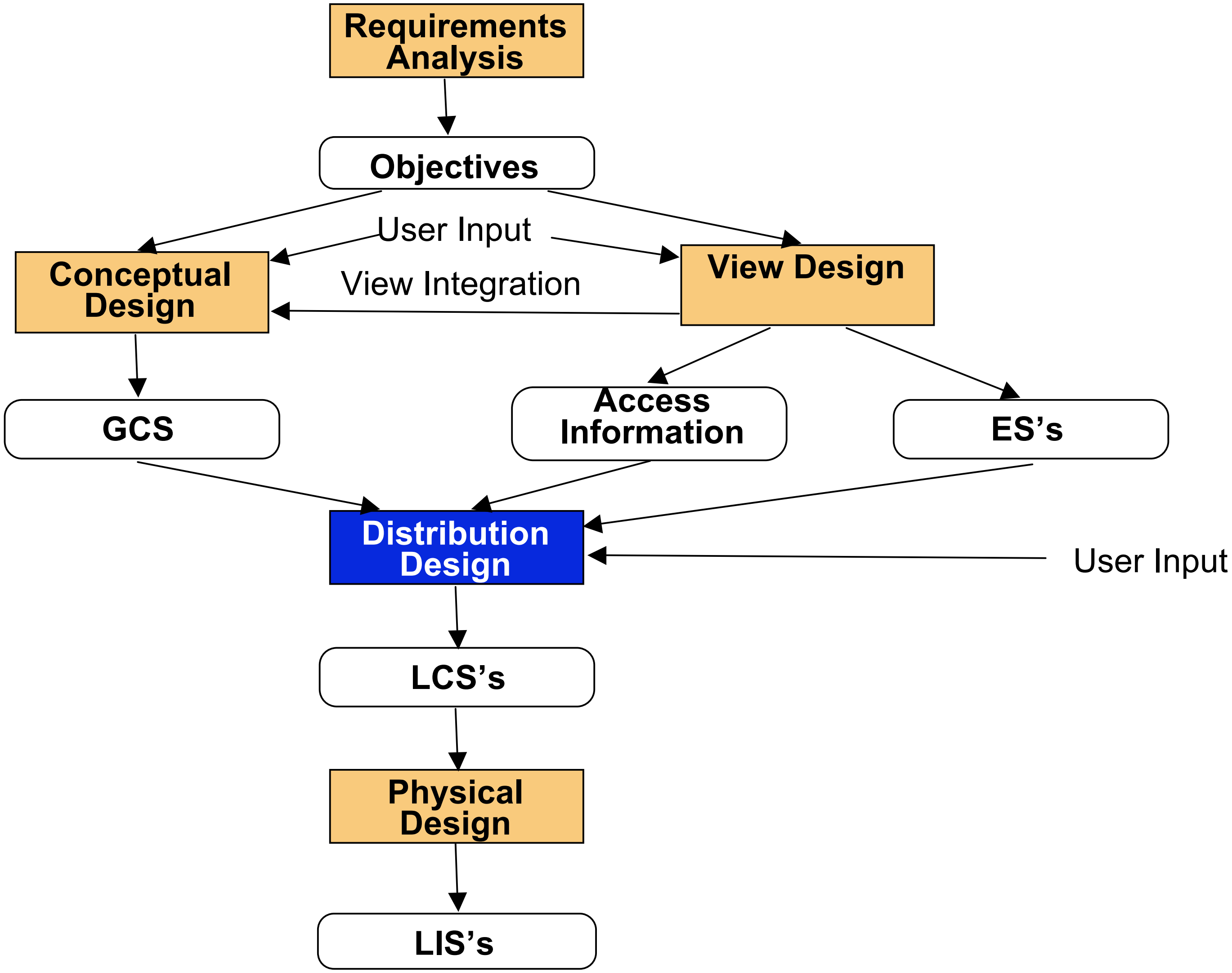
# Design Problem

- In the general setting :
  - Making decisions about the placement of data and programs across the sites of a computer network as well as possibly designing the network itself.
- In Distributed DBMS, the placement of applications entails
  - placement of the distributed DBMS software; and
  - placement of the applications that run on the database

# Distribution Design

- Top-down
  - mostly in designing systems from scratch
  - mostly in homogeneous systems
- Bottom-up
  - when the databases already exist at a number of sites

# Top-Down Design



# Distribution Design

- Fragmentation
  - Localize access
  - Horizontal fragmentation
  - Vertical fragmentation
  - Hybrid fragmentation
- Distribution
  - Placement of fragments on nodes of a network

# Horizontal Fragmentation

$PROJ_1$ : 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 |

$PROJ_2$

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

# Vertical Fragmentation

$PROJ_1$: 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 |

$PROJ_1$

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

$PROJ_2$

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

# Correctness of Fragmentation

- Completeness
  - Decomposition of relation $R$ into fragments $R_1$, $R_2$, ..., $R_n$ is complete iff each data item in $R$ can also be found in some $R_i$
- Reconstruction
  - If relation $R$ is decomposed into fragments $R_1$, $R_2$, ..., $R_n$, then there should exist some relational operator ∇ such that

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

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

# Allocation 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

# Fragment Allocation

- Problem Statement
  - Given
    - F = {F1, F2, …, Fn} fragments
    - S ={S1, S2, …, Sm} network sites
    - Q = {q1, q2,…, qq} applications
  - Find the "optimal" distribution of F to S.
- Optimality
  - Minimal cost
    - Communication + storage + processing (read & update)
    - Cost in terms of time (usually)
  - Performance
    - Response time and/or throughput
  - Constraints
    - Per site constraints (storage & processing)

# Allocation Model

**General Form**

min(Total Cost)

subject to

response time constraint

storage constraint

processing constraint

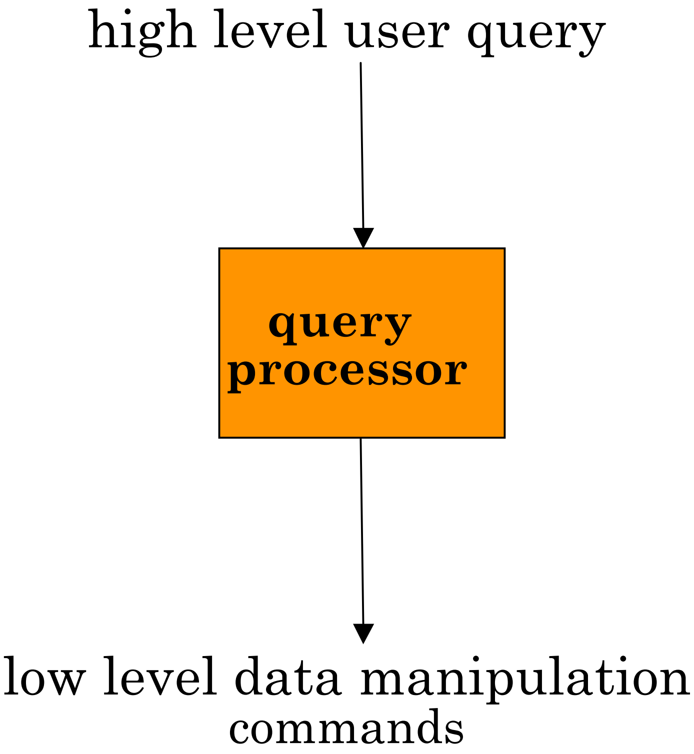
Decision Variable

$$x_{ij} = \begin{cases} 1 \text{ if fragment } F_i \text{ is stored at site } S_j \\ 0 \text{ otherwise} \end{cases}$$

# Outline

- Introduction
- Distributed DBMS Architecture
- Distributed Database Design
- Distributed Query Processing
  - Query Processing Methodology
  - Distributed Query Optimization
- Distributed Concurrency Control
- Distributed Reliability Protocols

# Query Processing



high level user query

**query processor**

low level data manipulation commands

# Query Processing Components

- Query language that is used
    - SQL: “intergalactic dataspeak”
- Query execution methodology
    - The steps that one goes through in executing high-level (declarative) user queries.
- Query optimization
    - How do we determine the “best” execution plan?

# Selecting Alternatives

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

Strategy 1

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

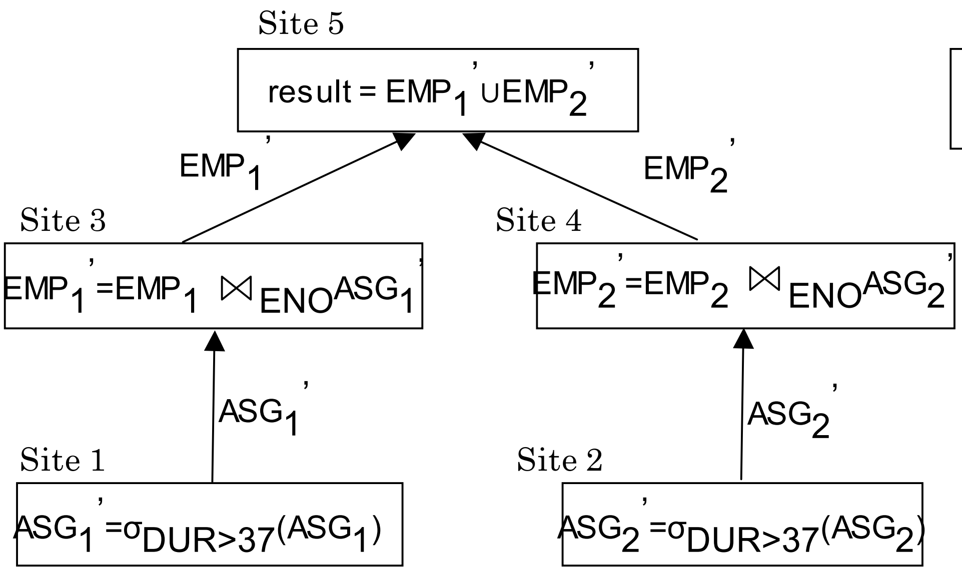
Strategy 2

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

Strategy 2 avoids Cartesian product, so is “better”

# What is the Problem?

| Site 1 | Site 2 | Site 3 | Site 4 | Site 5 |
|---|---|---|---|---|
| $ASG_1=\sigma_{ENO\leq\text{“}E3\text{”}}(ASG)$ | $ASG_2=\sigma_{ENO>\text{“}E3\text{”}}(ASG)$ | $EMP_1=\sigma_{ENO\leq\text{“}E3\text{”}}(EMP)$ | $EMP_2=\sigma_{ENO>\text{“}E3\text{”}}(EMP)$ | Result |



Site 5

$result = EMP_1' \cup EMP_2'$

$EMP_1'$

$EMP_2'$

Site 3

$EMP_1' = EMP_1 \bowtie_{ENO} ASG_1'$

Site 4

$EMP_2' = EMP_2 \bowtie_{ENO} ASG_2'$

$ASG_1'$

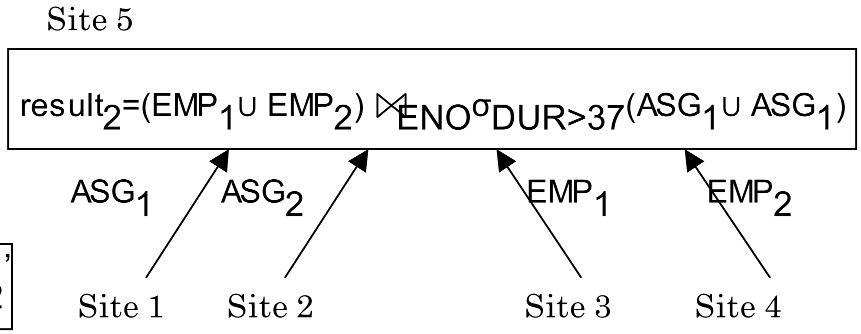
$ASG_2'$

Site 1

$ASG_1' = \sigma_{DUR>37}(ASG_1)$

Site 2

$ASG_2' = \sigma_{DUR>37}(ASG_2)$



Site 5

$result_2 = (EMP_1 \cup EMP_2) \bowtie_{ENO} \sigma_{DUR>37}(ASG_1 \cup ASG_1)$

$ASG_1$

$ASG_2$

$EMP_1$

$EMP_2$

Site 1

Site 2

Site 3

Site 4

# Cost of Alternatives

- Assume:
  - *size*(EMP) = 400, *size*(ASG) = 1000
  - tuple access cost = 1 unit; tuple transfer cost = 10 units
- Strategy 1

| | | |
|---|---|---|
| ❶ | produce ASG': (10+10)∗tuple access cost | 20 |
| ❷ | transfer ASG' to the sites of EMP: (10+10)∗tuple transfer cost | 200 |
| ❸ | produce EMP': (10+10) ∗tuple access cost∗2 | 40 |
| ❹ | transfer EMP' to result site: (10+10) ∗tuple transfer cost | 200 |
| | Total cost | 460 |

- Strategy 2

| | | |
|---|---|---|
| ❶ | transfer EMP to site 5:400∗tuple transfer cost | 4,000 |
| ❷ | transfer ASG to site 5 :1000∗tuple transfer cost | 10,000 |
| ❸ | produce ASG':1000∗tuple access cost | 1,000 |
| ❹ | join EMP and ASG':400∗20∗tuple access cost | 8,000 |
| | Total cost | 23,000 |

# Query Optimization Objectives

Minimize a cost function

I/O cost + CPU cost + communication cost

These might have different weights in different distributed environments

Wide area networks

- communication cost will dominate
    - low bandwidth
    - low speed
    - high protocol overhead
- most algorithms ignore all other cost components

Local area networks

- communication cost not that dominant
- total cost function should be considered

Can also maximize throughput

# Query Optimization Issues – Types of Optimizers

- Exhaustive search
  - cost-based
  - optimal
  - combinatorial complexity in the number of relations
- Heuristics
  - not optimal
  - regroup common sub-expressions
  - perform selection, projection first
  - replace a join by a series of semijoins
  - reorder operations to reduce intermediate relation size
  - optimize individual operations

# Query Optimization Issues – Optimization Granularity

- Single query at a time
  - cannot use common intermediate results
- Multiple queries at a time
  - efficient if many similar queries
  - decision space is much larger

# Query Optimization Issues – Optimization Timing

- Static
  - compilation ⇒ optimize prior to the execution
  - difficult to estimate the size of the intermediate results ⇒ error propagation
  - can amortize over many executions
  - R*
- Dynamic
  - run time optimization
  - exact information on the intermediate relation sizes
  - have to reoptimize for multiple executions
  - Distributed INGRES
- Hybrid
  - compile using a static algorithm
  - if the error in estimate sizes > threshold, reoptimize at run time
  - MERMAID

# Query Optimization Issues – Statistics

- Relation
  - cardinality
  - size of a tuple
  - fraction of tuples participating in a join with another relation
- Attribute
  - cardinality of domain
  - actual number of distinct values
- Common assumptions
  - independence between different attribute values
  - uniform distribution of attribute values within their domain

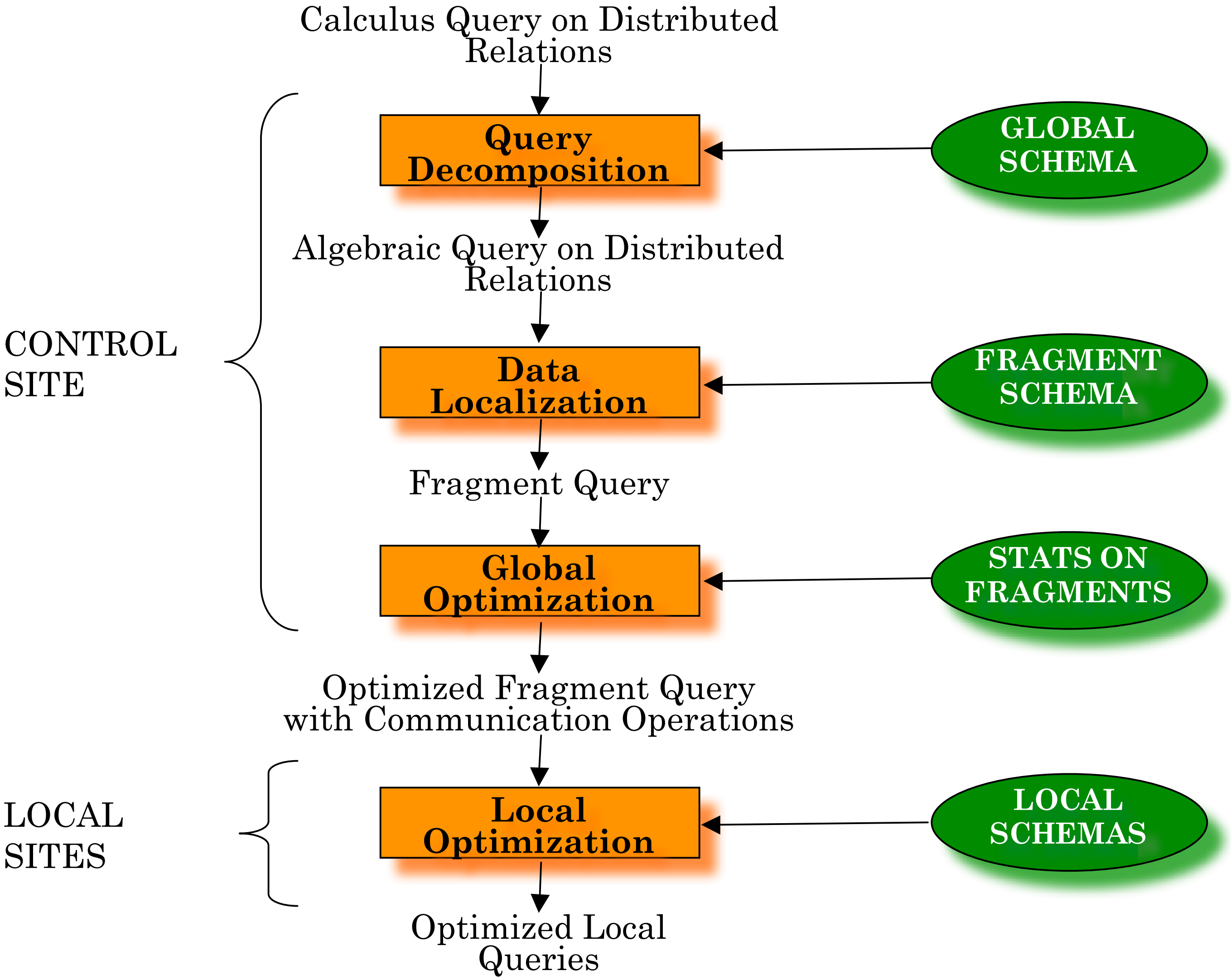
# Query Optimization Issues – Decision Sites

- Centralized
  - single site determines the “best” schedule
  - simple
  - need knowledge about the entire distributed database
- Distributed
  - cooperation among sites to determine the schedule
  - need only local information
  - cost of cooperation
- Hybrid
  - one site determines the global schedule
  - each site optimizes the local subqueries

# Query Optimization Issues – Network Topology

- Wide area networks (WAN) – point-to-point
  - characteristics
    - low bandwidth
    - low speed
    - high protocol overhead
  - communication cost will dominate; ignore all other cost factors
  - global schedule to minimize communication cost
  - local schedules according to centralized query optimization
- Local area networks (LAN)
  - communication cost not that dominant
  - total cost function should be considered
  - broadcasting can be exploited (joins)
  - special algorithms exist for star networks

# Distributed Query Processing Methodology



Calculus Query on Distributed Relations

**Query Decomposition**

GLOBAL SCHEMA

Algebraic Query on Distributed Relations

**Data Localization**

FRAGMENT SCHEMA

Fragment Query

**Global Optimization**

STATS ON FRAGMENTS

Optimized Fragment Query with Communication Operations

**Local Optimization**

LOCAL SCHEMAS

Optimized Local Queries

CONTROL SITE

LOCAL SITES

# Step 1 – Query Decomposition

Input : Calculus query on global relations

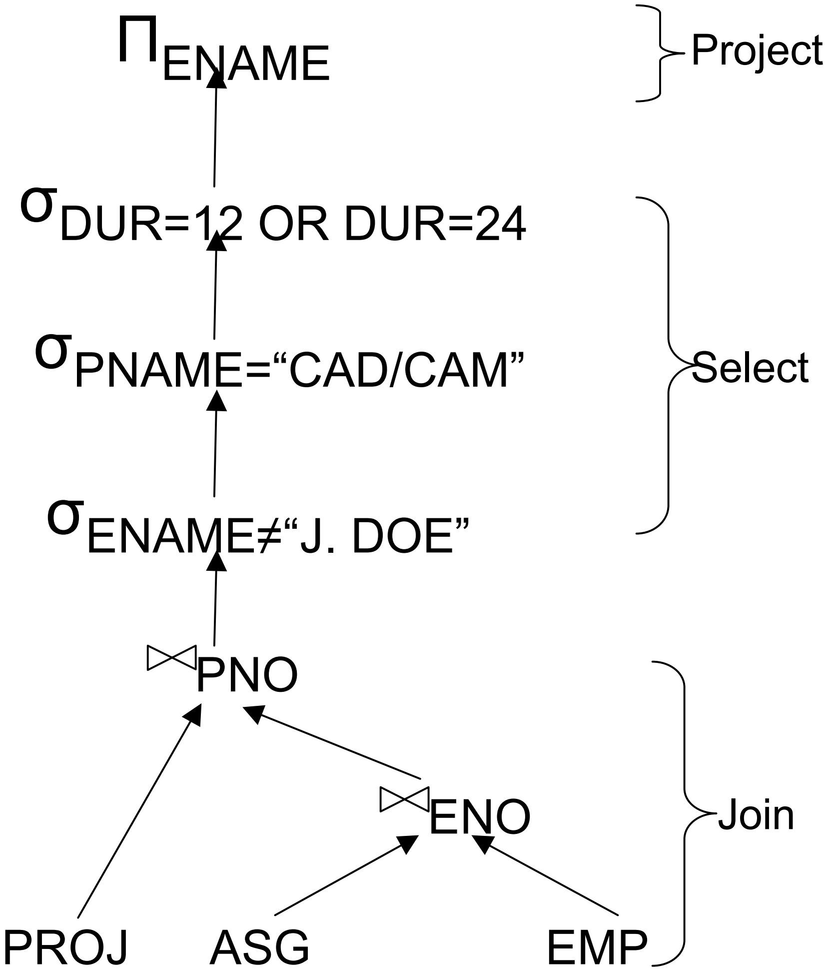
- Normalization
  - manipulate query quantifiers and qualification
- Analysis
  - detect and reject “incorrect” queries
  - possible for only a subset of relational calculus
- Simplification
  - eliminate redundant predicates
- Restructuring
  - calculus query ⇒ algebraic query
  - more than one translation is possible
  - use transformation rules

# Restructuring

- Convert relational calculus to relational algebra
- Make use of query trees
- Example

Find the names of employees other than J. Doe who worked on the CAD/CAM project for either 1 or 2 years.

```
SELECT ENAME
FROM EMP, ASG, PROJ
WHERE  EMP.ENO = ASG.ENO
AND  ASG.PNO = PROJ.PNO
AND  ENAME ≠ “J. Doe”
AND  PNAME = “CAD/CAM”
AND  (DUR = 12 OR DUR = 24)
```



# Restructuring –Transformation Rules (Examples)

- Commutativity of binary operations
  - $R \times S \Leftrightarrow S \times R$
  - $R \bowtie S \Leftrightarrow S \bowtie R$
  - $R \cup S \Leftrightarrow S \cup R$
- Associativity of binary operations
  - $( R \times S ) \times T \Leftrightarrow R \times (S \times T)$
  - $( R \bowtie S ) \bowtie T \Leftrightarrow R \bowtie (S \bowtie T )$
- Idempotence of unary operations
  - $\Pi_{A'}(\Pi_{A'}(R)) \Leftrightarrow \Pi_{A'}(R)$
  - $\sigma_{p_1(A_1)}(\sigma_{p_2(A_2)}(R)) = \sigma_{p_1(A_1) \wedge p_2(A_2)}(R)$

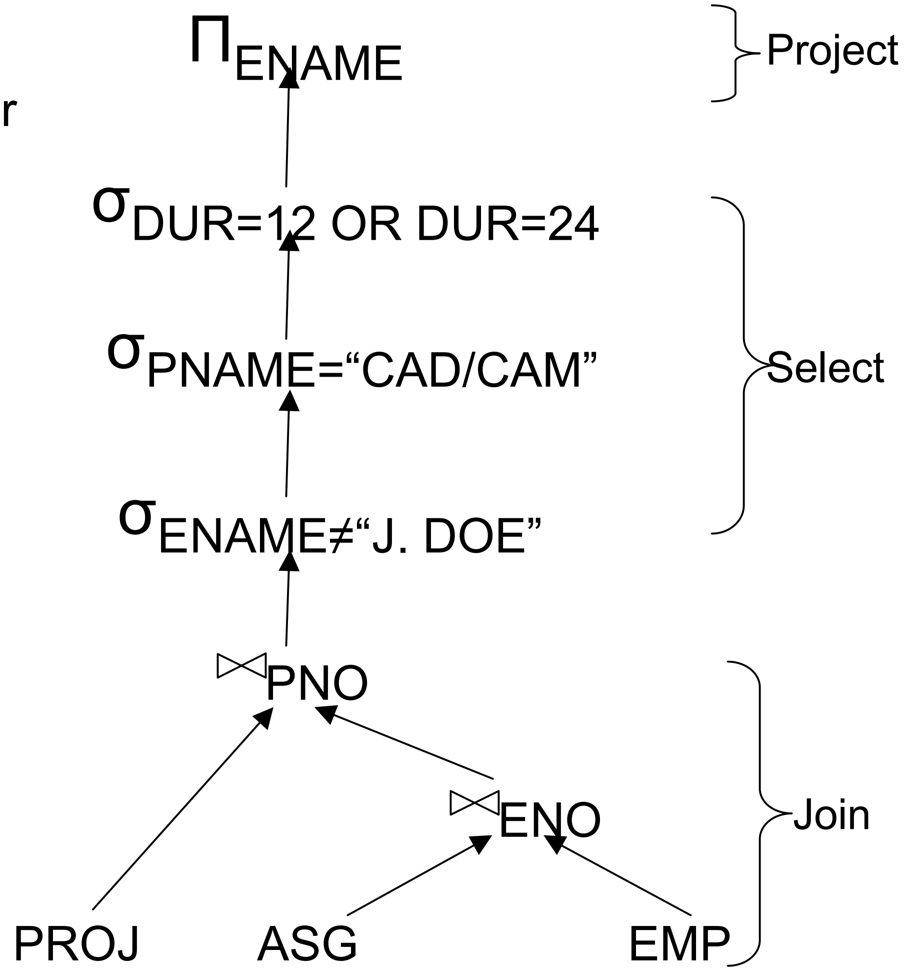
  where $R[A]$ and $A' \subseteq A$, $A'' \subseteq A$ and $A' \subseteq A''$
- Commuting selection with projection

# Example

Recall the previous example:

Find the names of employees other than J. Doe who worked on the CAD/CAM project for either one or two years.

```
SELECT ENAME
FROM   PROJ, ASG, EMP
WHERE  ASG.ENO=EMP.ENO
AND ASG.PNO=PROJ.PNO
AND ENAME≠“J. Doe”
AND PROJ.PNAME=“CAD/CAM”
AND (DUR=12 OR DUR=24)
```



$\Pi_{\text{ENAME}}$ Project

$\sigma_{\text{DUR=12 OR DUR=24}}$

$\sigma_{\text{PNAME=“CAD/CAM”}}$ Select

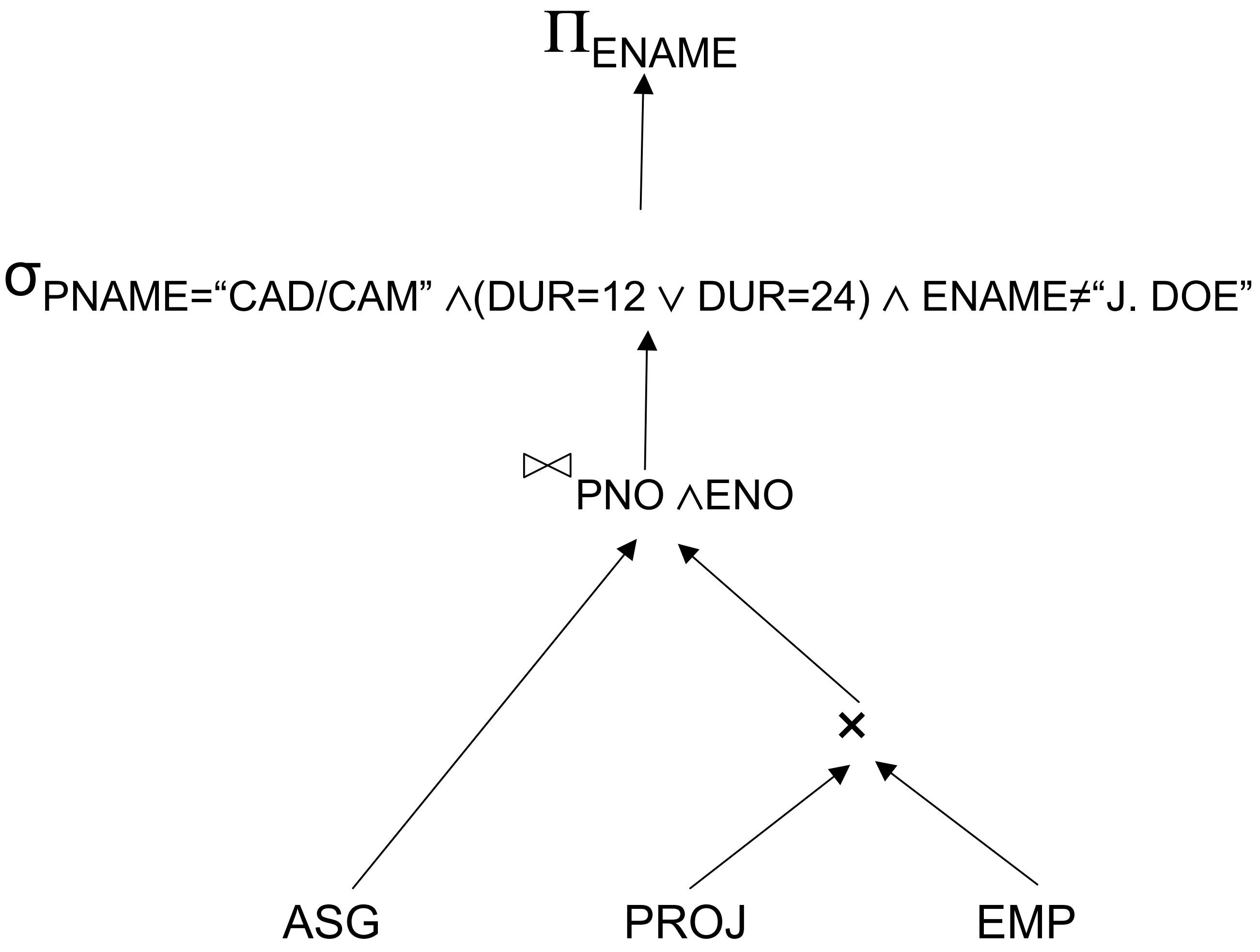
$\sigma_{\text{ENAME≠“J. DOE”}}$

$\bowtie_{\text{PNO}}$

$\bowtie_{\text{ENO}}$ Join

PROJ ASG EMP

# Equivalent Query



$\Pi_{\text{ENAME}}$

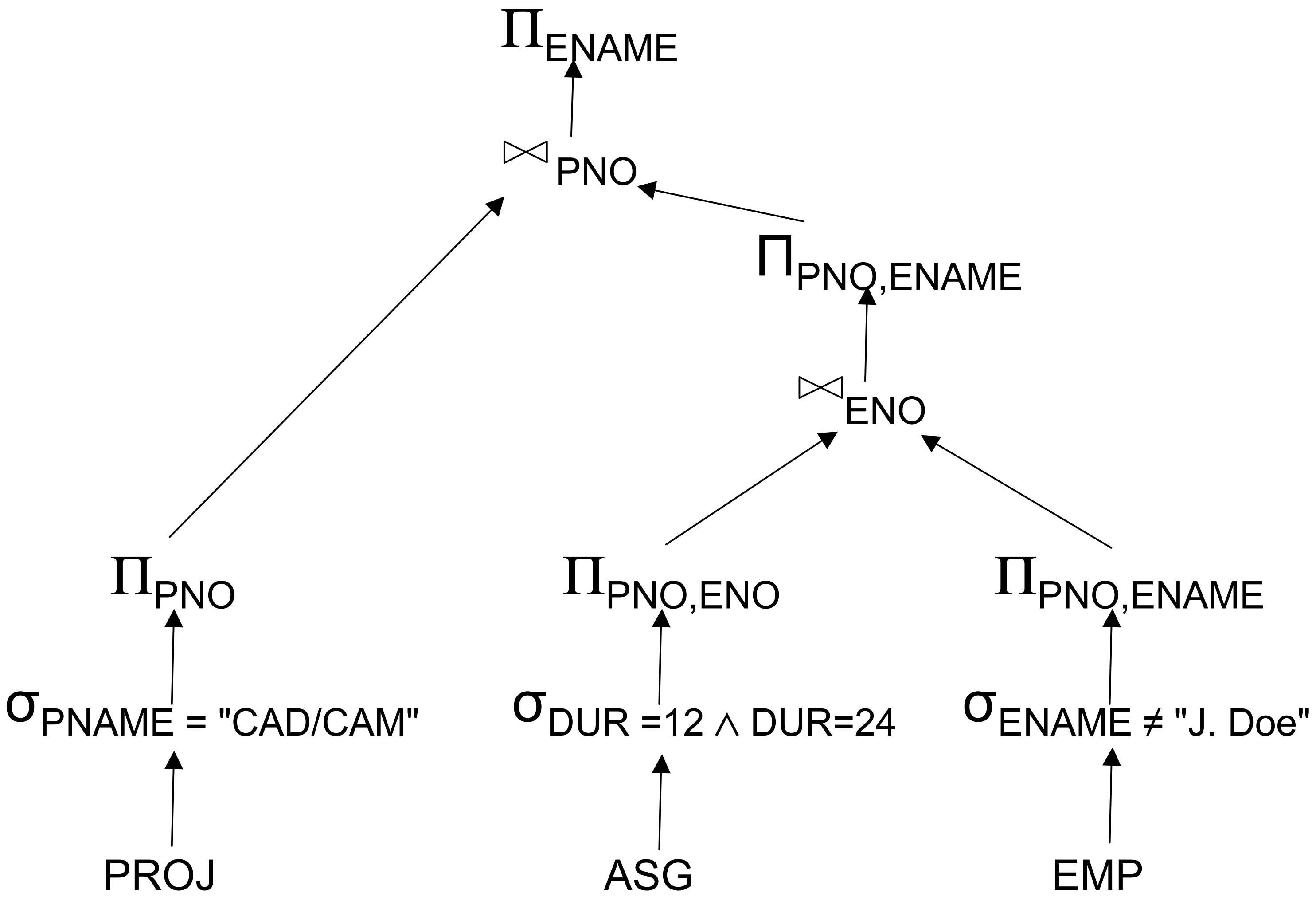
$\sigma_{\text{PNAME="CAD/CAM"} \wedge (\text{DUR=12} \vee \text{DUR=24}) \wedge \text{ENAME} \neq \text{"J. DOE"}}$

$\bowtie_{\text{PNO} \wedge \text{ENO}}$

×

ASG PROJ EMP

# Restructuring



$\Pi_{ENAME}$

$\bowtie_{PNO}$

$\Pi_{PNO,ENAME}$

$\bowtie_{ENO}$

$\Pi_{PNO}$

$\Pi_{PNO,ENO}$

$\Pi_{PNO,ENAME}$

$\sigma_{PNAME = \text{"CAD/CAM"}}$

$\sigma_{DUR = 12 \wedge DUR=24}$

$\sigma_{ENAME \neq \text{"J. Doe"}}$

PROJ

ASG

EMP

# Step 2 – Data Localization

Input: Algebraic query on distributed relations

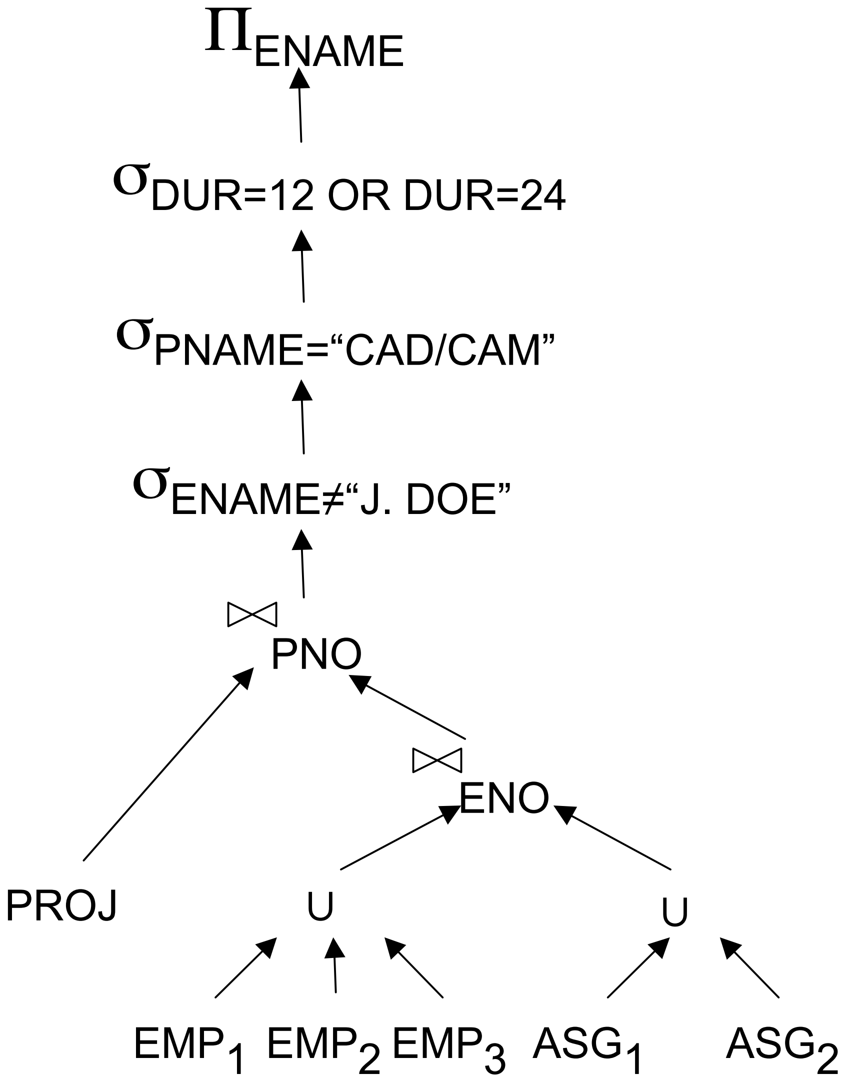
- Determine which fragments are involved
- Localization program
  - substitute for each global query its materialization program
  - optimize

# Example

Assume

- EMP is fragmented into EMP$_1$, EMP$_2$, EMP$_3$ as follows:
  - $EMP_1=\sigma_{ENO\leq\text{“E3”}}(EMP)$
  - $EMP_2= \sigma_{\text{“E3”}<ENO\leq\text{“E6”}}(EMP)$
  - $EMP_3=\sigma_{ENO\geq\text{“E6”}}(EMP)$
- ASG fragmented into ASG$_1$ and ASG$_2$ as follows:
  - $ASG_1=\sigma_{ENO\leq\text{“E3”}}(ASG)$
  - $ASG_2=\sigma_{ENO>\text{“E3”}}(ASG)$

Replace EMP by ($EMP_1 \cup EMP_2 \cup EMP_3$ ) and ASG by ($ASG_1 \cup ASG_2$) in any query



$\Pi_{ENAME}$

$\sigma_{DUR=12 \text{ OR } DUR=24}$

$\sigma_{PNAME=\text{“CAD/CAM”}}$

$\sigma_{ENAME\neq\text{“J. DOE”}}$

$\bowtie_{PNO}$

PROJ

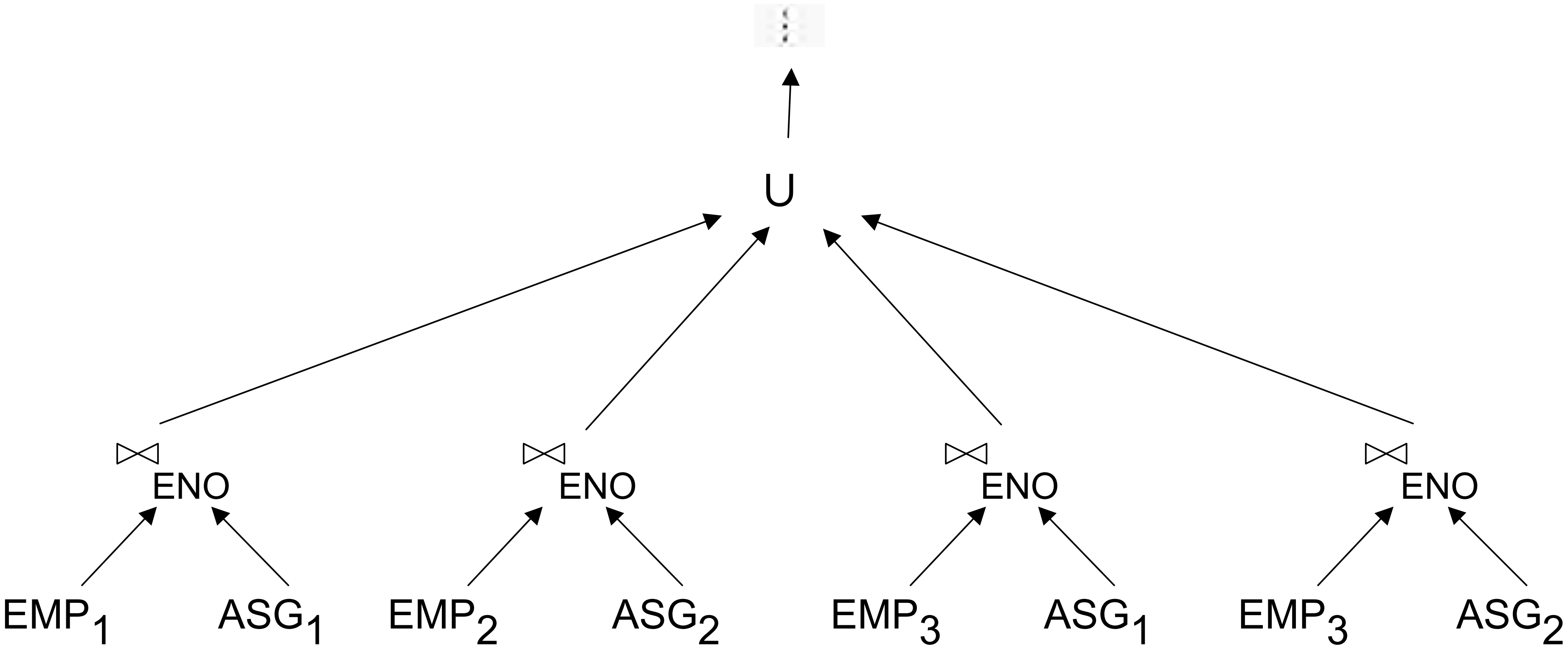
$\bowtie_{ENO}$

$\cup$

$\cup$

EMP$_1$ EMP$_2$ EMP$_3$ ASG$_1$ ASG$_2$

# Provides Parallellism

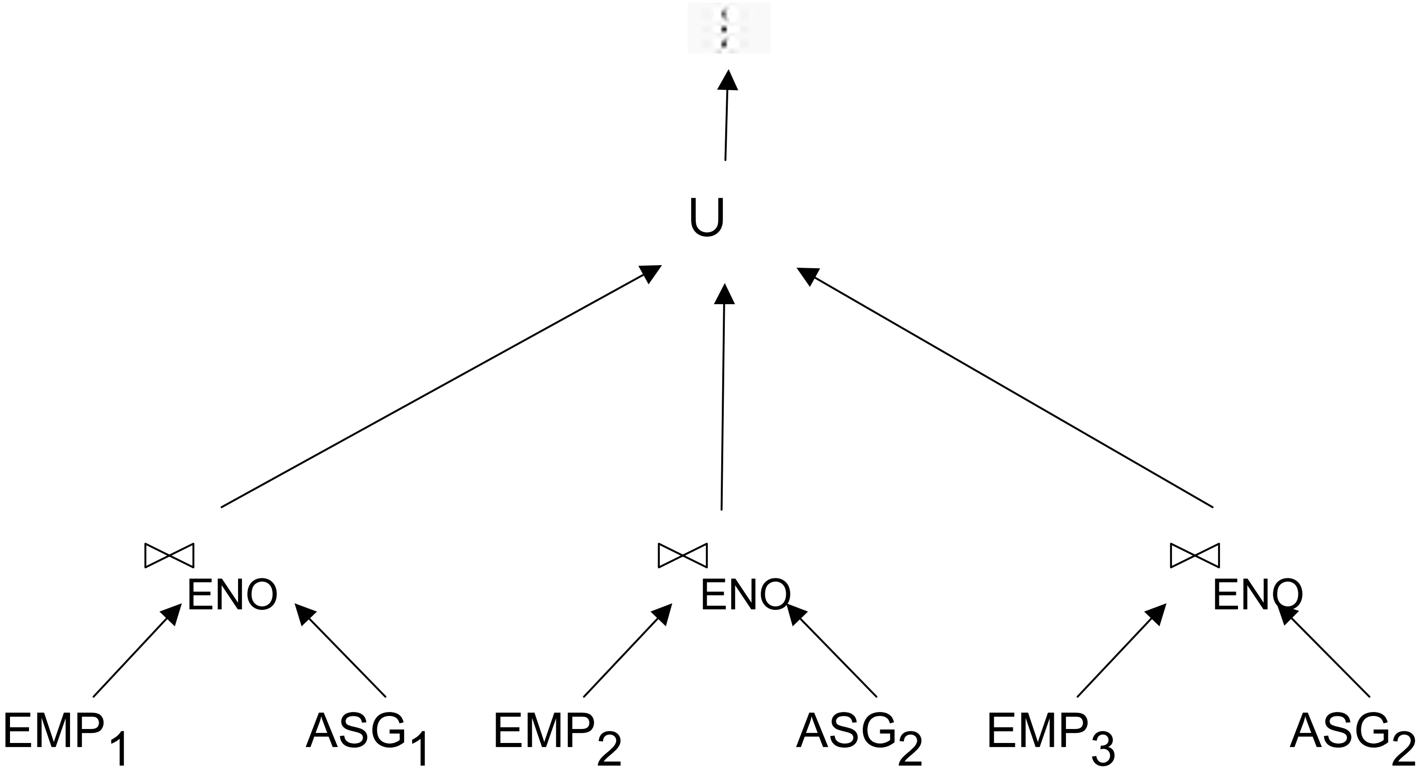


$\cup$

$\bowtie_{ENO}$ $\bowtie_{ENO}$ $\bowtie_{ENO}$ $\bowtie_{ENO}$

$EMP_1$ $ASG_1$ $EMP_2$ $ASG_2$ $EMP_3$ $ASG_1$ $EMP_3$ $ASG_2$

# Eliminates Unnecessary Work



U

$\bowtie_{ENO}$ $\bowtie_{ENO}$ $\bowtie_{ENO}$

$EMP_1$ $ASG_1$ $EMP_2$ $ASG_2$ $EMP_3$ $ASG_2$

# Step 3 – Global Query Optimization

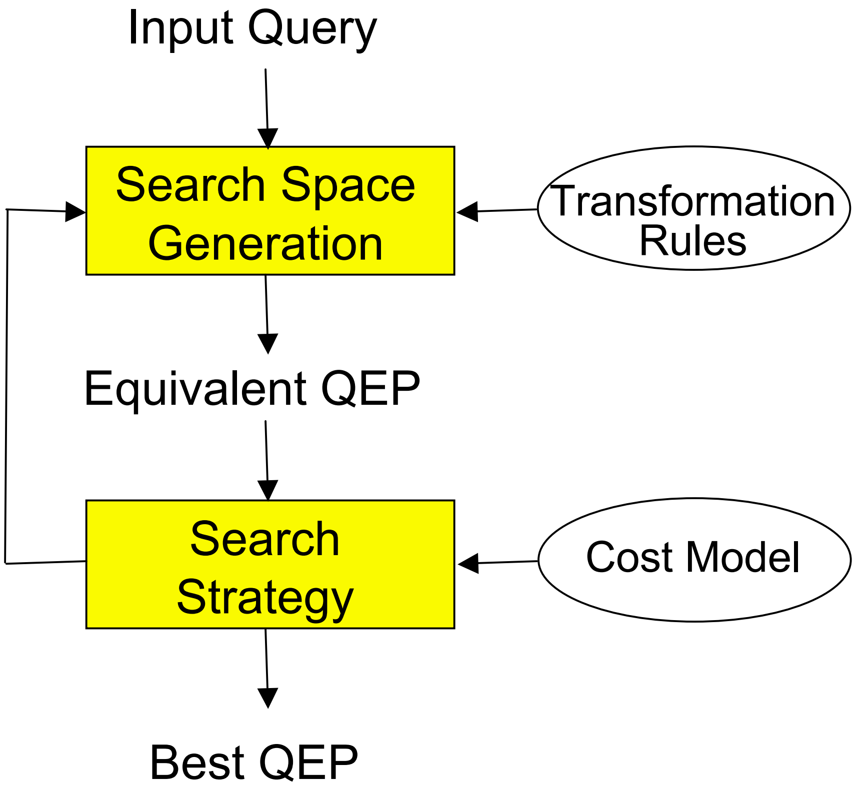
Input: Fragment query

- Find the *best* (not necessarily optimal) global schedule
  - Minimize a cost function
  - Distributed join processing
    - Bushy vs. linear trees
    - Which relation to ship where?
    - Ship-whole vs ship-as-needed
  - Decide on the use of semijoins
    - Semijoin saves on communication at the expense of more local processing.
  - Join methods
    - nested loop vs ordered joins (merge join or hash join)

# Cost-Based Optimization

- Solution space
  - The set of equivalent algebra expressions (query trees).
- Cost function (in terms of time)
  - I/O cost + CPU cost + communication cost
  - These might have different weights in different distributed environments (LAN vs WAN).
  - Can also maximize throughput
- Search algorithm
  - How do we move inside the solution space?
  - Exhaustive search, heuristic algorithms (iterative improvement, simulated annealing, genetic,…)

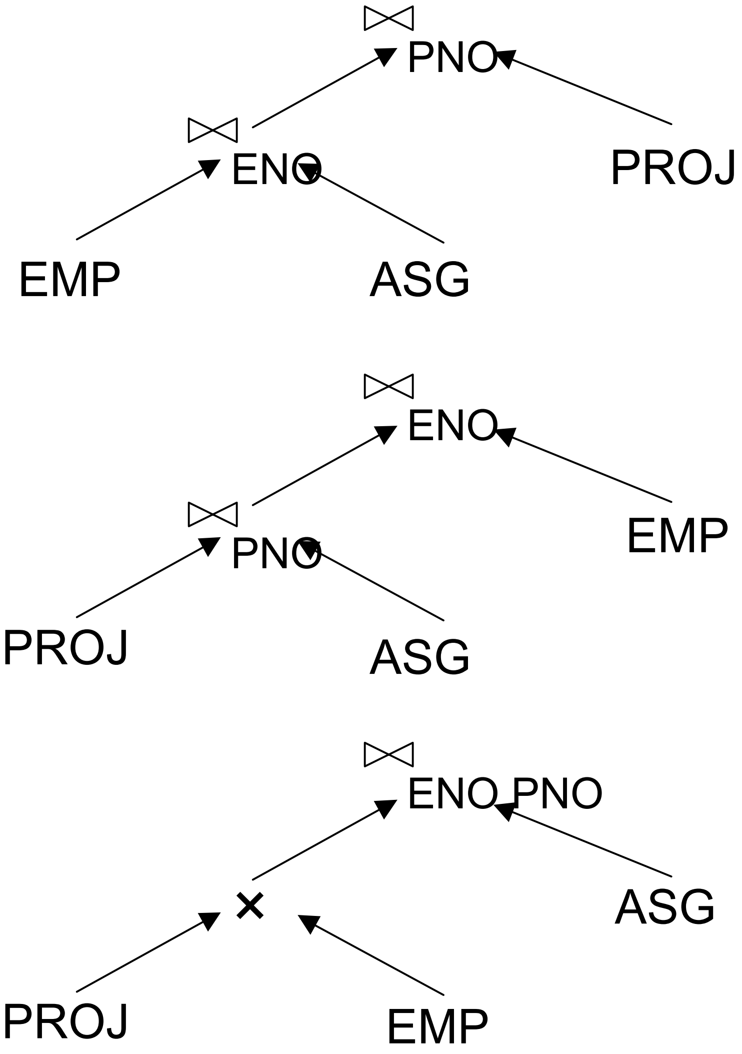
# Query Optimization Process



# Search Space

- Search space characterized by alternative execution plans
- Focus on join trees
- For *N* relations, there are O(*N*!) equivalent join trees that can be obtained by applying commutativity and associativity rules

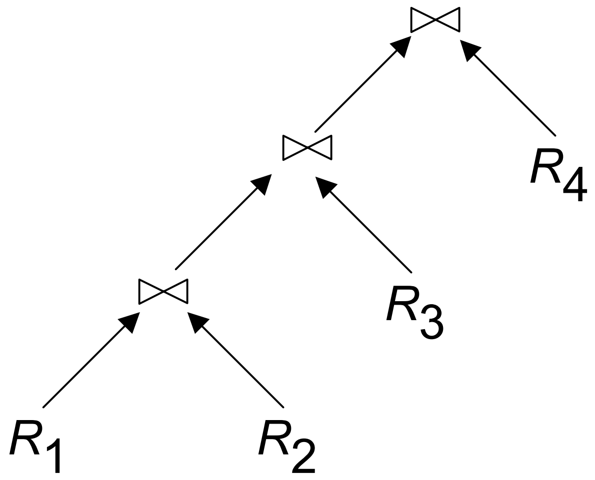
```
SELECT ENAME,RESP
FROM   EMP, ASG, PROJ
WHERE  EMP.ENO=ASG.ENO
AND    ASG.PNO=PROJ.PNO
```



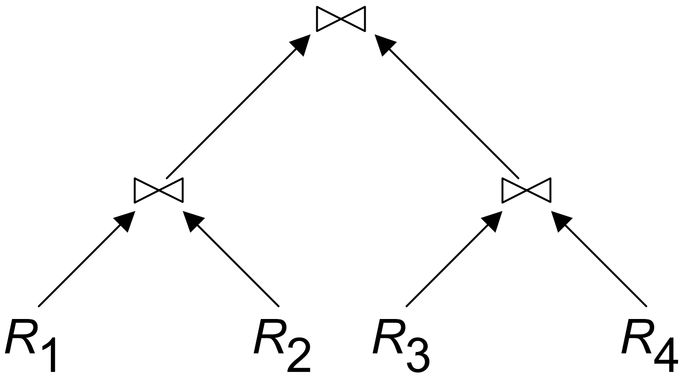
# Search Space

- Restrict by means of heuristics
  - Perform unary operations before binary operations
  - …
- Restrict the shape of the join tree
  - Consider only linear trees, ignore bushy ones

Linear Join Tree



Bushy Join Tree

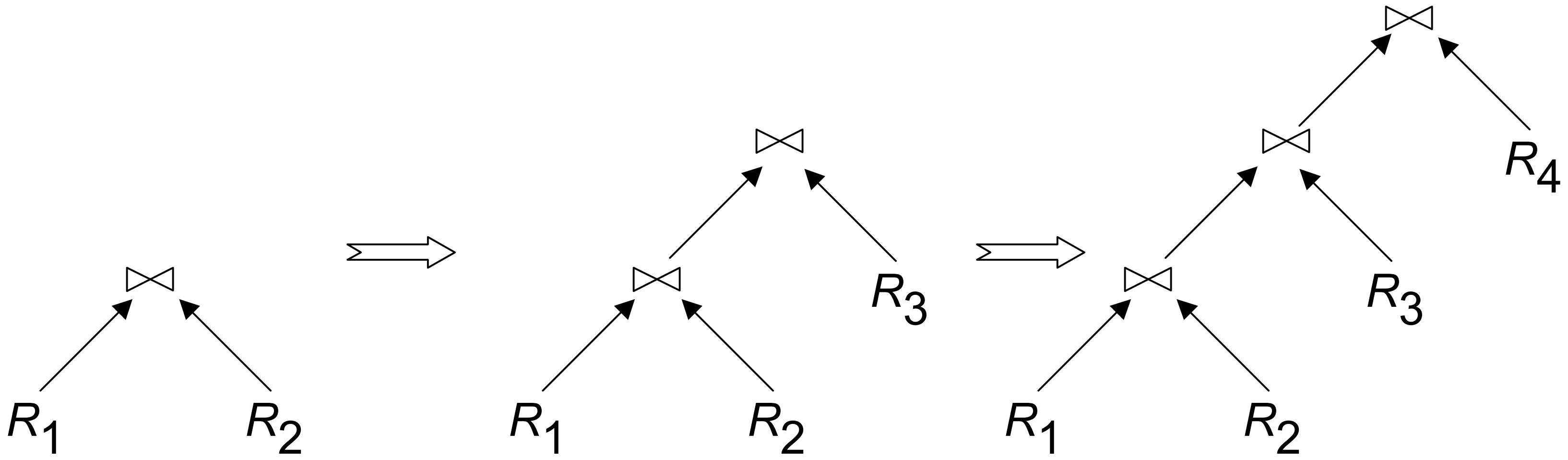


# Search Strategy

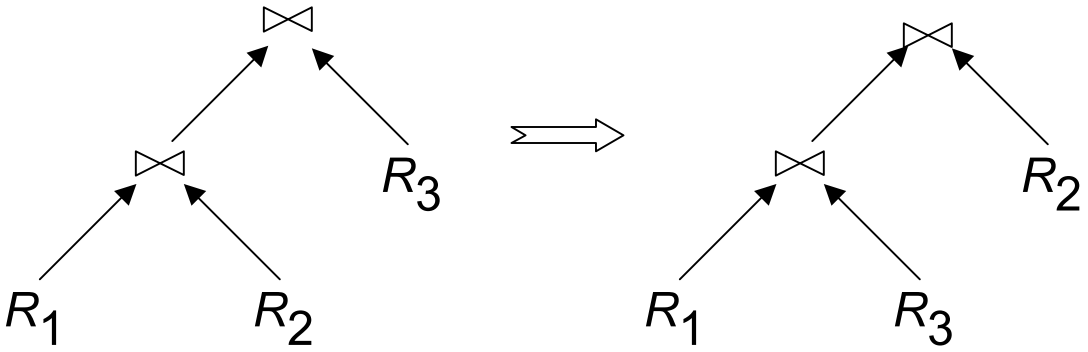
- How to “move” in the search space.
- Deterministic
  - Start from base relations and build plans by adding one relation at each step
  - Dynamic programming: breadth-first
  - Greedy: depth-first
- Randomized
  - Search for optimalities around a particular starting point
  - Trade optimization time for execution time
  - Better when > 5-6 relations
  - Simulated annealing
  - Iterative improvement

# Search Strategies

- Deterministic



- Randomized



# Cost Functions

- Total Time (or Total Cost)
  - Reduce each cost (in terms of time) component individually
  - Do as little of each cost component as possible
  - Optimizes the utilization of the resources

⇩

Increases system throughput

- Response Time
  - Do as many things as possible in parallel
  - May increase total time because of increased total activity

# Total Cost

Summation of all cost factors

Total cost = CPU cost + I/O cost + communication cost

CPU cost = unit instruction cost $*$ no.of instructions

I/O cost = unit disk I/O cost $*$ no. of disk I/Os

communication cost = message initiation + transmission

# Total Cost Factors

- Wide area network
  - message initiation and transmission costs high
  - local processing cost is low (fast mainframes or minicomputers)
  - ratio of communication to I/O costs = 20:1
- Local area networks
  - communication and local processing costs are more or less equal
  - ratio = 1:1.6

# Response Time

Elapsed time between the initiation and the completion of a query

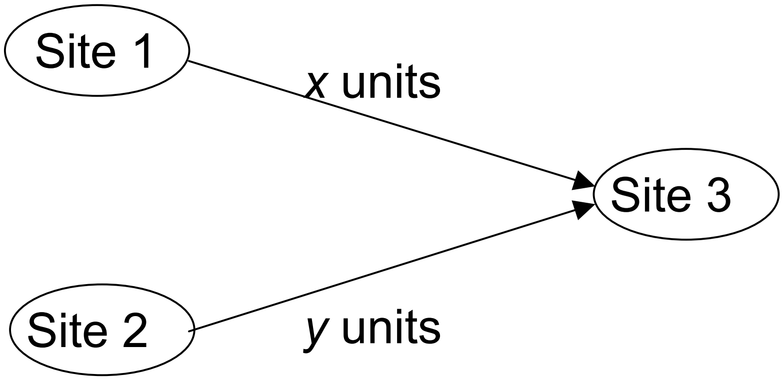
Response time = CPU time + I/O time + communication time

CPU time = unit instruction time ∗ no. of sequential instructions

I/O time = unit I/O time ∗ no. of sequential I/Os

communication time = unit msg initiation time ∗
no. of sequential msg + unit transmission time ∗
no. of sequential bytes

# Example



Assume that only the communication cost is considered

Total time = 2 $*$ message initialization time + unit transmission time $*$ $(x+y)$

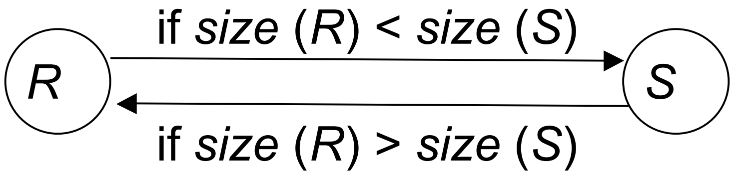
Response time = max {time to send $x$ from 1 to 3, time to send $y$ from 2 to 3}

time to send $x$ from 1 to 3 = message initialization time + unit transmission time $*$ $x$

time to send $y$ from 2 to 3 = message initialization time + unit transmission time $*$ $y$

# Join Ordering

- Alternatives
  - Ordering joins
  - Semijoin ordering
- Consider two relations only



if $\mathit{size}(R) < \mathit{size}(S)$

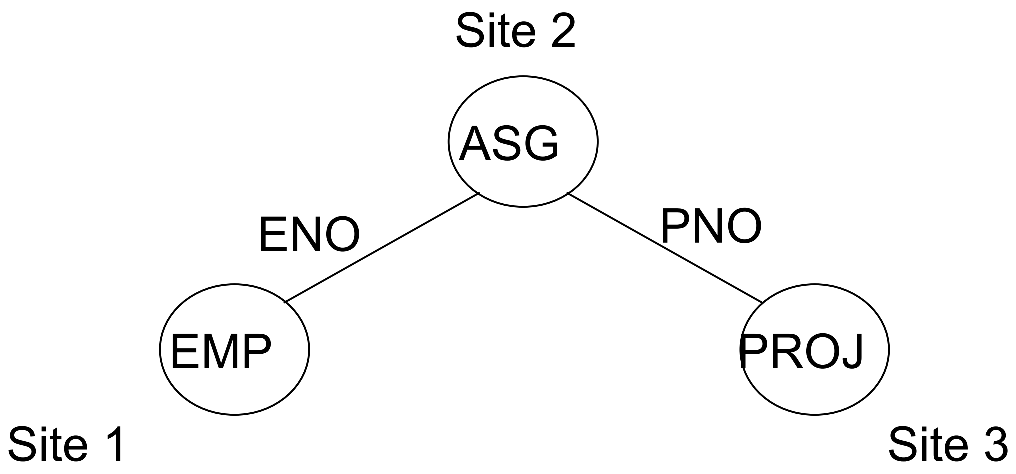
if $\mathit{size}(R) > \mathit{size}(S)$

- Multiple relations more difficult because too many alternatives.
  - Compute the cost of all alternatives and select the best one.
    - Necessary to compute the size of intermediate relations which is difficult.
  - Use heuristics

# Join Ordering – Example

Consider

$$\text{PROJ} \bowtie_{\text{PNO}} \text{ASG} \bowtie_{\text{ENO}} \text{EMP}$$



# Join Ordering – Example

Execution alternatives:

1. EMP → Site 2
   Site 2 computes EMP'=EMP $\bowtie$ ASG
   EMP' → Site 3
   Site 3 computes EMP' $\bowtie$ PROJ

2. ASG → Site 1
   Site 1 computes EMP'=EMP $\bowtie$ ASG
   EMP' → Site 3
   Site 3 computes EMP' $\bowtie$ PROJ

3. ASG → Site 3
   Site 3 computes ASG'=ASG $\bowtie$ PROJ
   ASG' → Site 1
   Site 1 computes ASG' $\bowtie$ EMP

4. PROJ → Site 2
   Site 2 computes PROJ'=PROJ $\bowtie$ ASG
   PROJ' → Site 1
   Site 1 computes PROJ' $\bowtie$ EMP

5. EMP → Site 2
   PROJ → Site 2
   Site 2 computes EMP $\bowtie$ PROJ $\bowtie$ ASG

# Semijoin Algorithms

- Consider the join of two relations:
  - $R[A]$ (located at site 1)
  - $S[A]$ (located at site 2)
- Alternatives:
  1. Do the join $R \bowtie_A S$
  2. Perform one of the semijoin equivalents

$$\begin{aligned} R \bowtie_A S &\Leftrightarrow (R \ltimes_A S) \bowtie_A S \\ &\Leftrightarrow R \bowtie_A (S \ltimes_A R) \\ &\Leftrightarrow (R \ltimes_A S) \bowtie_A (S \ltimes_A R) \end{aligned}$$

# Semijoin Algorithms

- Perform the join
  - send $R$ to Site 2
  - Site 2 computes $R \bowtie_A S$
- Consider semijoin $(R \ltimes_A S) \bowtie_A S$
  - $S' \leftarrow \prod_A(S)$
  - $S' \rightarrow$ Site 1
  - Site 1 computes $R' = R \ltimes_A S'$
  - $R' \rightarrow$ Site 2
  - Site 2 computes $R' \bowtie_A S$

Semijoin is better if

$$size(\Pi_A(S)) + size(R \ltimes_A S)) < size(R)$$

# R* Algorithm

- Cost function includes local processing as well as transmission
- Considers only joins
- Exhaustive search
- Compilation
- Published papers provide solutions to handling horizontal and vertical fragmentations but the implemented prototype does not

# R* Algorithm

Performing joins

- Ship whole
  - larger data transfer
  - smaller number of messages
  - better if relations are small
- Fetch as needed
  - number of messages = *O*(cardinality of external relation)
  - data transfer per message is minimal
  - better if relations are large and the selectivity is good

# R* Algorithm – Vertical Partitioning & Joins

1. Move outer relation tuples to the site of the inner relation

   (a) Retrieve outer tuples

   (b) Send them to the inner relation site

   (c) Join them as they arrive

Total Cost = cost(retrieving qualified outer tuples)

+ no. of outer tuples fetched $*$
cost(retrieving qualified inner tuples)

+ msg. cost $*$ (no. outer tuples fetched $*$
avg. outer tuple size) / msg. size

# R* Algorithm – Vertical Partitioning & Joins

2. Move inner relation to the site of outer relation

cannot join as they arrive; they need to be stored

Total Cost = cost(retrieving qualified outer tuples)

+ no. of outer tuples fetched ∗ cost(retrieving matching inner tuples from temporary storage)

+ cost(retrieving qualified inner tuples)

+ cost(storing all qualified inner tuples in temporary storage)

+ msg. cost ∗ (no. of inner tuples fetched ∗ avg. inner tuple size) / msg. size

# R* Algorithm – Vertical Partitioning & Joins

3. Move both inner and outer relations to another site

Total cost = cost(retrieving qualified outer tuples)

+ cost(retrieving qualified inner tuples)

+ cost(storing inner tuples in storage)

+ msg. cost ∗ (no. of outer tuples fetched ∗ avg. outer tuple size) / msg. size

+ msg. cost ∗ (no. of inner tuples fetched ∗ avg. inner tuple size) / msg. size

+ no. of outer tuples fetched ∗ cost(retrieving inner tuples from temporary storage)

# R* Algorithm –
# Vertical Partitioning & Joins

4. Fetch inner tuples as needed
    (a) Retrieve qualified tuples at outer relation site
    (b) Send request containing join column value(s) for outer tuples to inner relation site
    (c) Retrieve matching inner tuples at inner relation site
    (d) Send the matching inner tuples to outer relation site
    (e) Join as they arrive

Total Cost = cost(retrieving qualified outer tuples)

+ msg. cost $*$ (no. of outer tuples fetched)

+ no. of outer tuples fetched $*$ (no. of inner tuples fetched $*$ avg. inner tuple size $*$ msg. cost / msg. size)

+ no. of outer tuples fetched $*$ cost(retrieving matching inner tuples for one outer value)

# Step 4 – Local Optimization

Input: Best global execution schedule

- Select the best access path
- Use the centralized optimization techniques

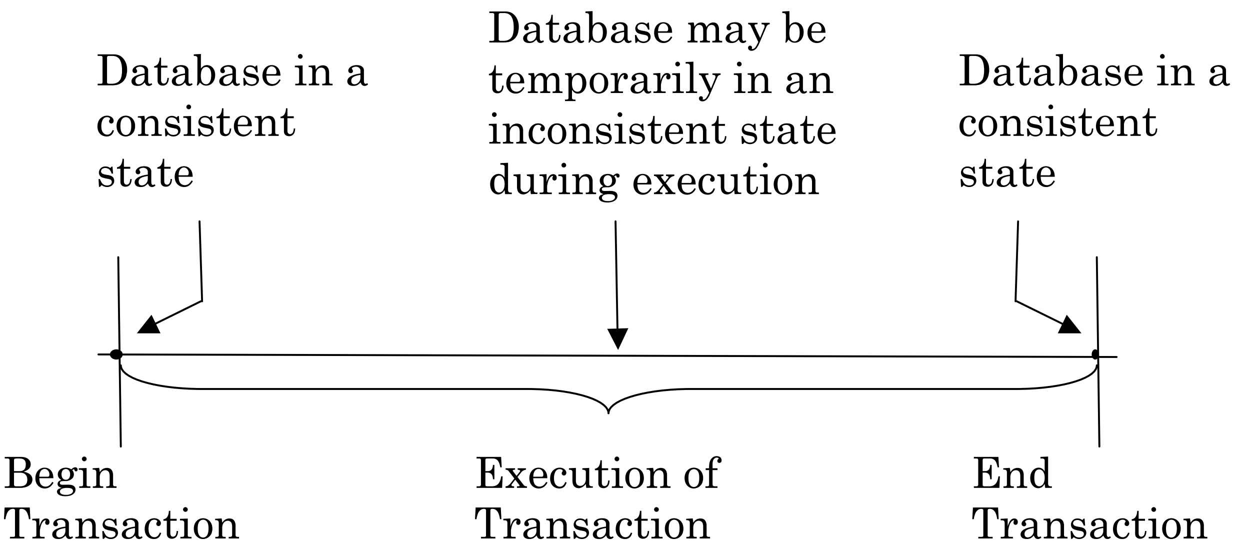
# Outline

- Introduction
- Distributed DBMS Architecture
- Distributed Database Design
- Distributed Query Processing
- Distributed Concurrency Control
  - Transaction Concepts & Models
  - Serializability
  - Distributed Concurrency Control Protocols
- Distributed Reliability Protocols

# Transaction

A transaction is a collection of actions that make consistent transformations of system states while preserving system consistency.

- concurrency transparency
- failure transparency



Database in a consistent state

Database may be temporarily in an inconsistent state during execution

Database in a consistent state

Begin Transaction

Execution of Transaction

End Transaction

# Example Database

Consider an airline reservation example with the relations:

FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
CUST(CNAME, ADDR, BAL)
FC(FNO, DATE, CNAME,SPECIAL)

# Example Transaction

```
Begin_transaction Reservation
begin
        input(flight_no, date, customer_name);
        EXEC SQL            UPDATE    FLIGHT
                  SET       STSOLD = STSOLD + 1
                  WHERE     FNO = flight_no AND DATE = date;
        EXEC SQL            INSERT
                  INTO      FC(FNO, DATE, CNAME, SPECIAL);
                  VALUES    (flight_no, date, customer_name, null);
        output("reservation completed")
end . {Reservation}
```

# Termination of Transactions

```
Begin_transaction Reservation
begin
input(flight_no, date, customer_name);
EXEC SQL        SELECT      STSOLD,CAP
                INTO        temp1,temp2
                FROM        FLIGHT
                WHERE       FNO = flight_no AND DATE =  date;
if temp1 = temp2 then
output("no free seats");
Abort
else
EXEC SQL        UPDATE      FLIGHT
                   SET      STSOLD = STSOLD + 1
                   WHERE    FNO = flight_no AND DATE = date;
EXEC SQL        INSERT
                   INTO     FC(FNO, DATE, CNAME, SPECIAL);
                   VALUES   (flight_no, date, customer_name, null);
   Commit
   output("reservation completed")
   endif
end . {Reservation}
```

# Properties of Transactions

**ATOMICITY**

- all or nothing

**CONSISTENCY**

- no violation of integrity constraints

**ISOLATION**

- concurrent changes invisible È serializable

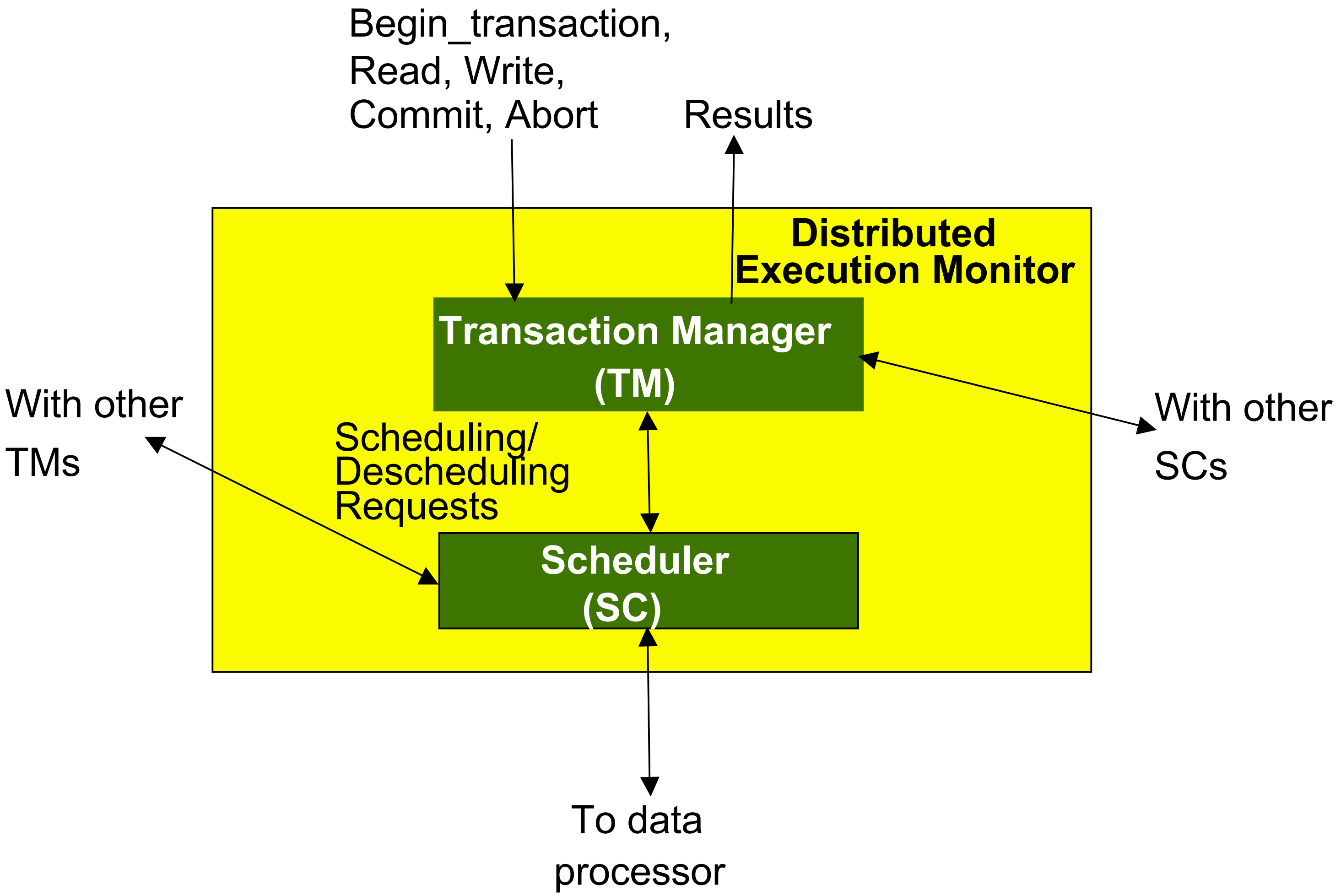
**DURABILITY**

- committed updates persist

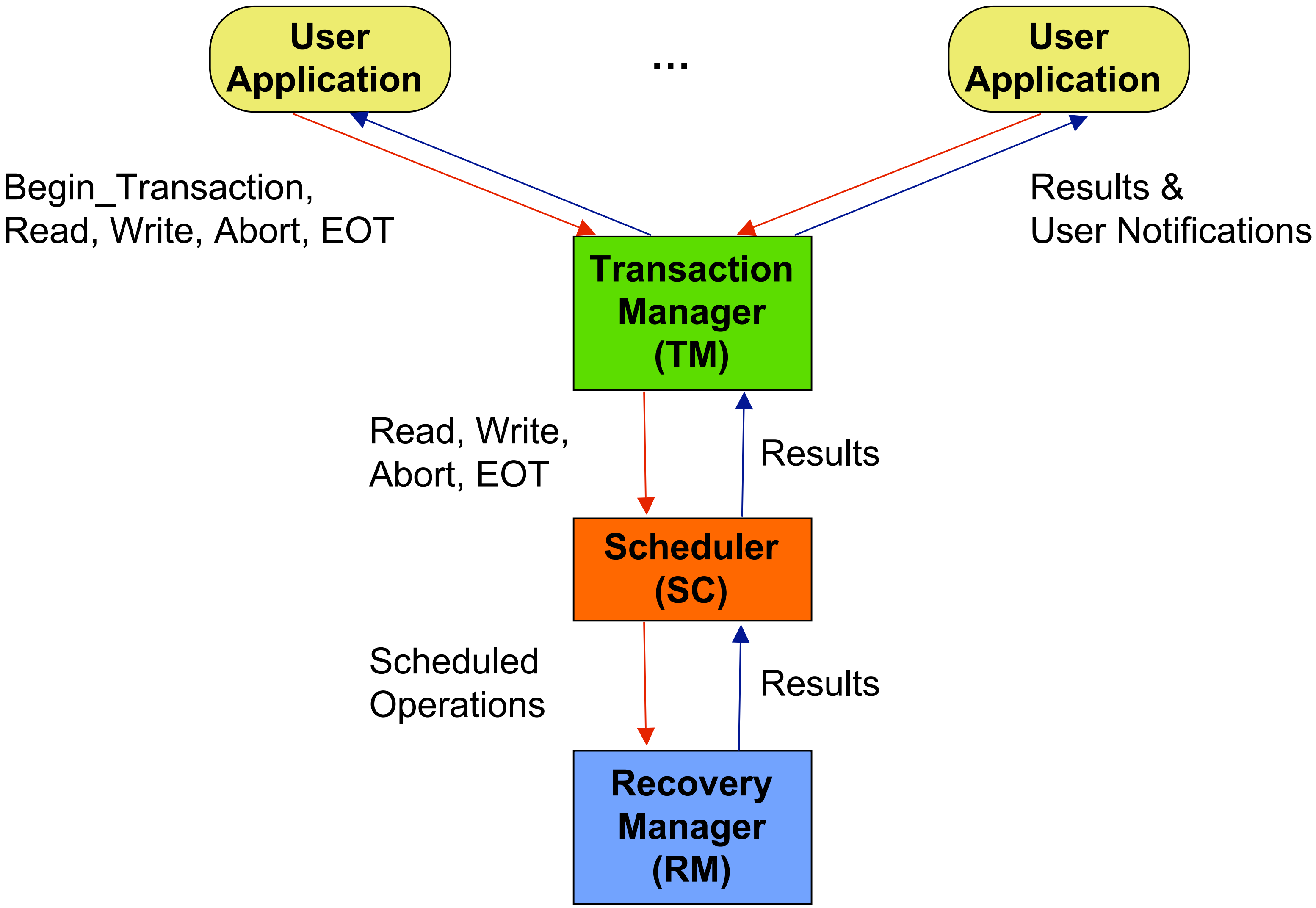
# Transactions Provide…

- *Atomic* and *reliable* execution in the presence of failures
- *Correct* execution in the presence of multiple user accesses
- Correct management of *replicas* (if they support it)

# Architecture Revisited



# Centralized Transaction Execution



User Application

…

User Application

Begin_Transaction,
Read, Write, Abort, EOT

Results &
User Notifications

Transaction Manager (TM)

Read, Write,
Abort, EOT

Results

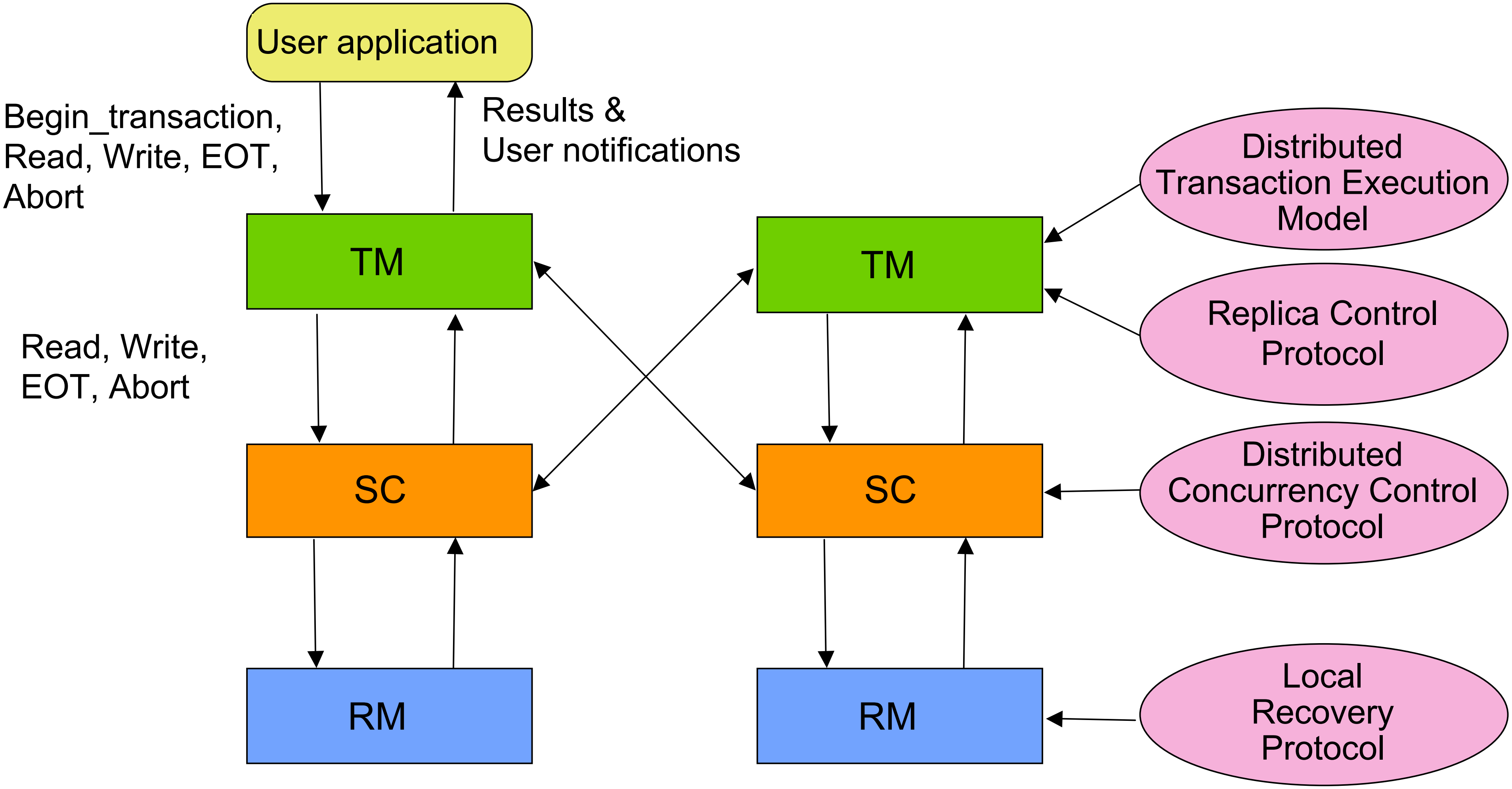
Scheduler (SC)

Scheduled
Operations

Results

Recovery Manager (RM)

# Distributed Transaction Execution



User application

Begin_transaction, Read, Write, EOT, Abort

Results & User notifications

TM

TM

Read, Write, EOT, Abort

SC

SC

RM

RM

Distributed Transaction Execution Model

Replica Control Protocol

Distributed Concurrency Control Protocol

Local Recovery Protocol

# Concurrency Control

- The problem of synchronizing concurrent transactions such that the consistency of the database is maintained while, at the same time, maximum degree of concurrency is achieved.
- Anomalies:
  - Lost updates
    - The effects of some transactions are not reflected on the database.
  - Inconsistent retrievals
    - A transaction, if it reads the same data item more than once, should always read the same value.

# Serializable History

- Transactions execute concurrently, but the net effect of the resulting history upon the database is *equivalent* to some *serial* history.
- Equivalent with respect to what?
  - ***Conflict equivalence***: the relative order of execution of the conflicting operations belonging to unaborted transactions in two histories are the same.
  - ***Conflicting operations***: two incompatible operations (e.g., Read and Write) conflict if they both access the same data item.
    - Incompatible operations of each transaction is assumed to conflict; do not change their execution orders.
    - If two operations from two different transactions conflict, the corresponding transactions are also said to conflict.

# Serializability in Distributed DBMS

- Somewhat more involved. Two histories have to be considered:
  - local histories
  - global history
- For global transactions (i.e., global history) to be serializable, two conditions are necessary:
  - Each local history should be serializable.
  - Two conflicting operations should be in the same relative order in all of the local histories where they appear together.

# Global Non-serializability

| | | | |
|---|---|---|---|
| $T_1$: | Read($x$) | $T_2$: | Read($x$) |
| | $x \leftarrow x+5$ | | $x \leftarrow x*15$ |
| | Write($x$) | | Write($x$) |
| | Commit | | Commit |

The following two local histories are individually serializable (in fact serial), but the two transactions are not globally serializable.

$$LH_1=\{R_1(x),W_1(x),C_1,R_2(x),W_2(x),C_2\}$$
$$LH_2=\{R_2(\mathrm{x}),W_2(\mathrm{x}),C_2,R_1(\mathrm{x}),W_1(\mathrm{x}),C_1\}$$

# Concurrency Control Algorithms

- Pessimistic
  - Two-Phase Locking-based (2PL)
    - Centralized (primary site) 2PL
    - Primary copy 2PL
    - Distributed 2PL
  - Timestamp Ordering (TO)
    - Basic TO
    - Multiversion TO
    - Conservative TO
  - Hybrid
- Optimistic
  - Locking-based
  - Timestamp ordering-based

# Locking-Based Algorithms

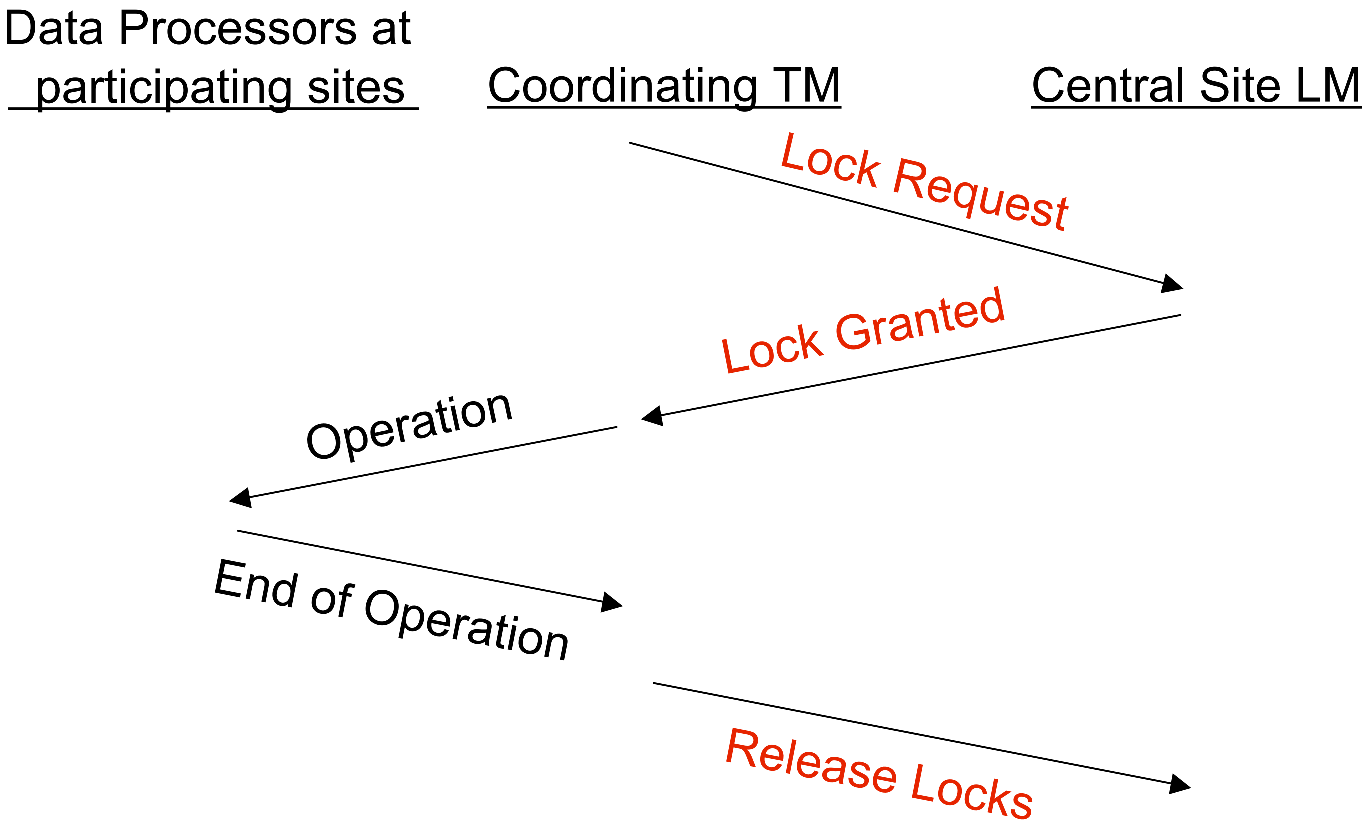
- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager).
- Locks are either read lock (*rl*) [also called shared lock] or write lock (*wl*) [also called exclusive lock]
- Read locks and write locks conflict (because Read and Write operations are incompatible

| | *rl* | *wl* |
|---|---|---|
| *rl* | yes | no |
| *wl* | no | no |

- Locking works nicely to allow concurrent processing of transactions.

# Centralized 2PL

- There is only one 2PL scheduler in the distributed system.
- Lock requests are issued to the central scheduler.



Data Processors at participating sites

Coordinating TM

Central Site LM

Lock Request

Lock Granted

Operation

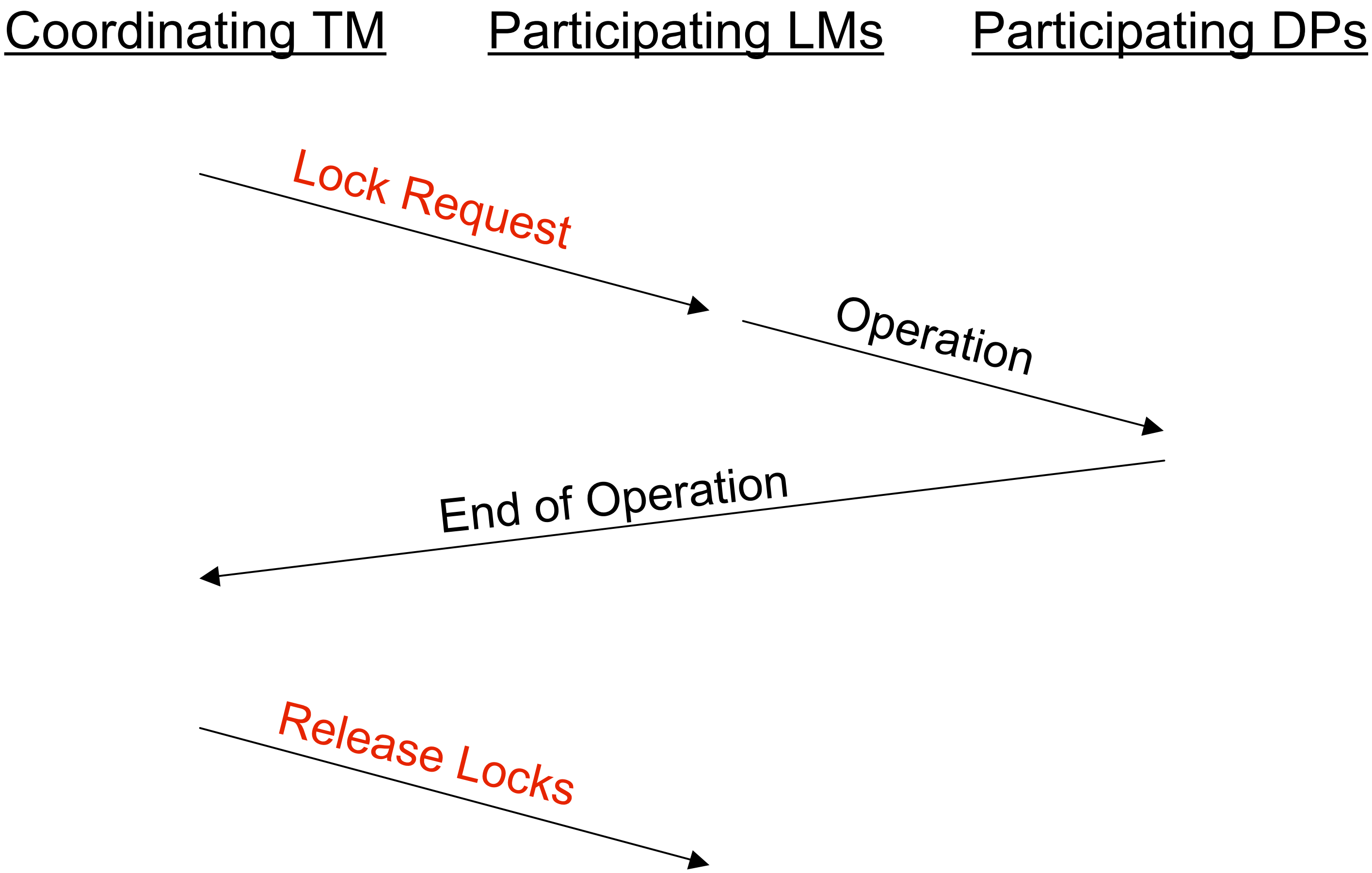
End of Operation

Release Locks

# Distributed 2PL

- 2PL schedulers are placed at each site. Each scheduler handles lock requests for data at that site.
- A transaction may read any of the replicated copies of item x, by obtaining a read lock on one of the copies of x. Writing into x requires obtaining write locks for all copies of x.

# Distributed 2PL Execution



Coordinating TM

Participating LMs

Participating DPs

Lock Request

Operation

End of Operation

Release Locks

# Timestamp Ordering

❶ Transaction ($T_i$) is assigned a globally unique timestamp $ts(T_i)$.

❷ Transaction manager attaches the timestamp to all operations issued by the transaction.

❸ Each data item is assigned a write timestamp (*wts*) and a read timestamp (*rts*):

- $rts(x)$ = largest timestamp of any read on $x$
- $wts(x)$ = largest timestamp of any read on $x$

❹ Conflicting operations are resolved by timestamp order.

Basic T/O:

```
for $R_i(x)$
if $ts(T_i) < wts(x)$
then reject $R_i(x)$
else accept $R_i(x)$
$rts(x) \leftarrow ts(T_i)$
```

```
for $W_i(x)$
if $ts(T_i) < rts(x)$ and $ts(T_i) < wts(x)$
then reject $W_i(x)$
else accept $W_i(x)$
$wts(x) \leftarrow ts(T_i)$
```

# Outline

- Introduction
- Distributed DBMS Architecture
- Distributed Database Design
- Distributed Query Processing
- Distributed Concurrency Control
- Distributed Reliability Protocols
    - Distributed Commit Protocols
    - Distributed Recovery Protocols

# Reliability

Problem:

How to maintain

atomicity

durability

properties of transactions

# Types of Failures

- Transaction failures
  - Transaction aborts (unilaterally or due to deadlock)
  - Avg. 3% of transactions abort abnormally
- System (site) failures
  - Failure of processor, main memory, power supply, …
  - Main memory contents are lost, but secondary storage contents are safe
  - Partial vs. total failure
- Media failures
  - Failure of secondary storage devices such that the stored data is lost
  - Head crash/controller failure (?)
- Communication failures
  - Lost/undeliverable messages
  - Network partitioning

# Distributed Reliability Protocols

- Commit protocols
  - How to execute commit command for distributed transactions.
  - Issue: how to ensure atomicity and durability?
- Termination protocols
  - If a failure occurs, how can the remaining operational sites deal with it.
  - *Non-blocking* : the occurrence of failures should not force the sites to wait until the failure is repaired to terminate the transaction.
- Recovery protocols
  - When a failure occurs, how do the sites where the failure occurred deal with it.
  - *Independent* : a failed site can determine the outcome of a transaction without having to obtain remote information.
- Independent recovery ⇒ non-blocking termination

# Two-Phase Commit (2PC)

*Phase 1* : The coordinator gets the participants ready to write the results into the database

*Phase 2* : Everybody writes the results into the database

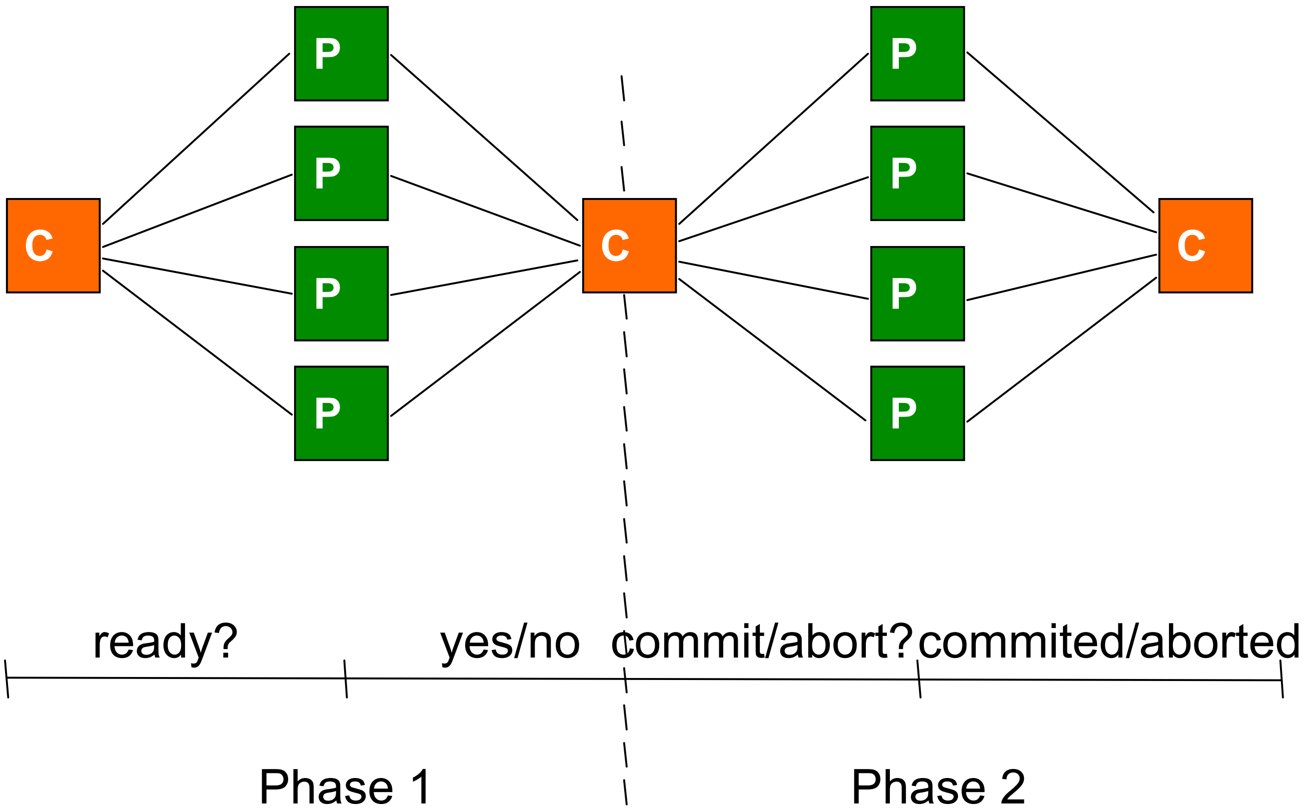
- **Coordinator** :The process at the site where the transaction originates and which controls the execution
- **Participant** :The process at the other sites that participate in executing the transaction

Global Commit Rule:

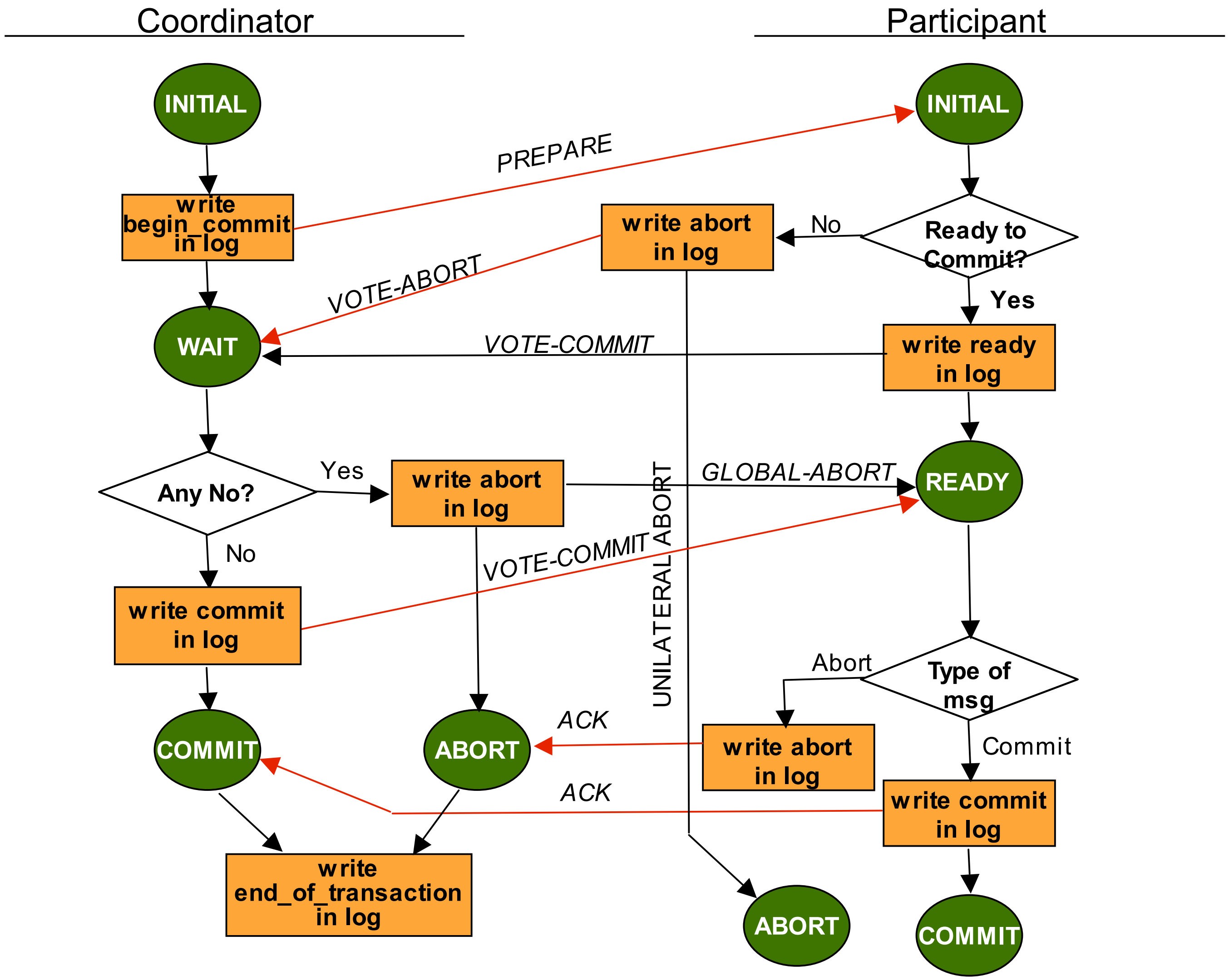
❶ The coordinator aborts a transaction if and only if at least one participant votes to abort it.

❷ The coordinator commits a transaction if and only if all of the participants vote to commit it.

# Centralized 2PC



# 2PC Protocol Actions



# Problem With 2PC

- Blocking
  - Ready implies that the participant waits for the coordinator
  - If coordinator fails, site is blocked until recovery
  - Blocking reduces availability
- Independent recovery is not possible
- However, it is known that:
  - Independent recovery protocols exist only for single site failures; no independent recovery protocol exists which is resilient to multiple-site failures.
- So we search for these protocols – 3PC