Distributed Databases

Chapter 1: Introduction

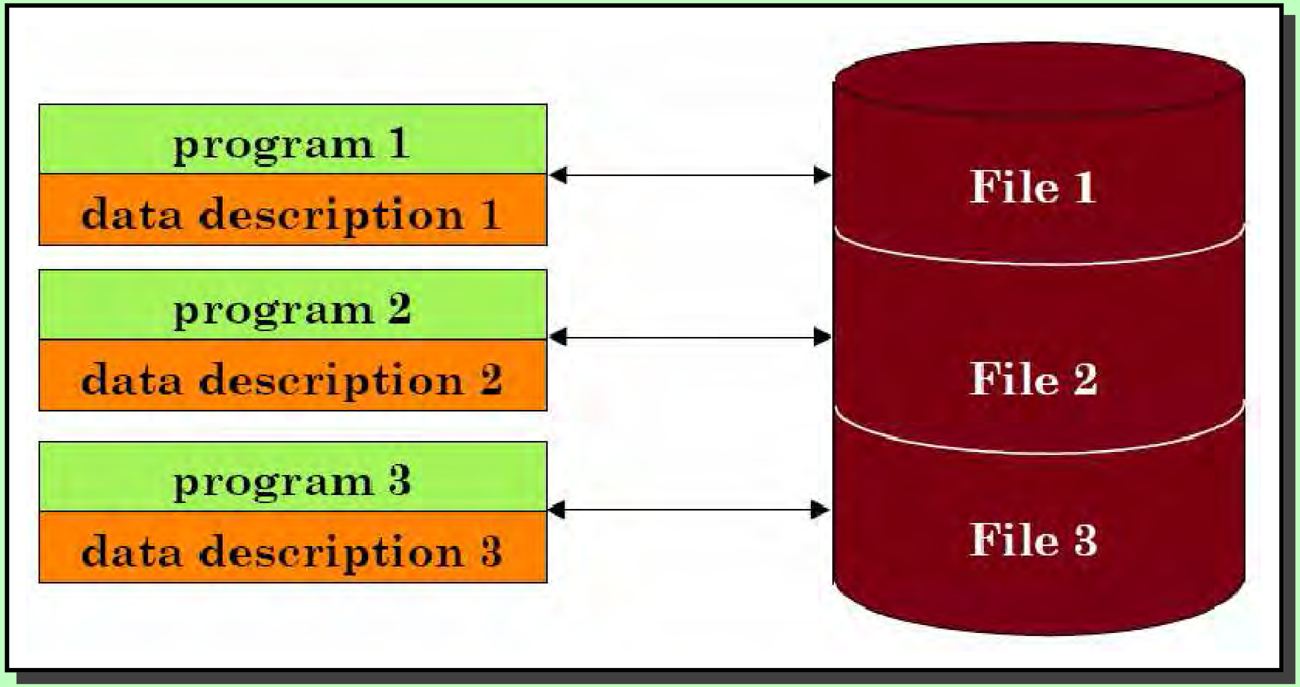
- Syllabus
- Data Independence and Distributed Data Processing
- Definition of Distributed databases
- Promises of Distributed Databases
- Technical Problems to be Studied
- Conclusion

Syllabus

- Introduction
- Distributed DBMS Architecture
- Distributed Database Design
- Query Processing
- Transaction Management
- Distributed Concurrency Control
- Distributed DBMS Reliability
- Parallel Database Systems

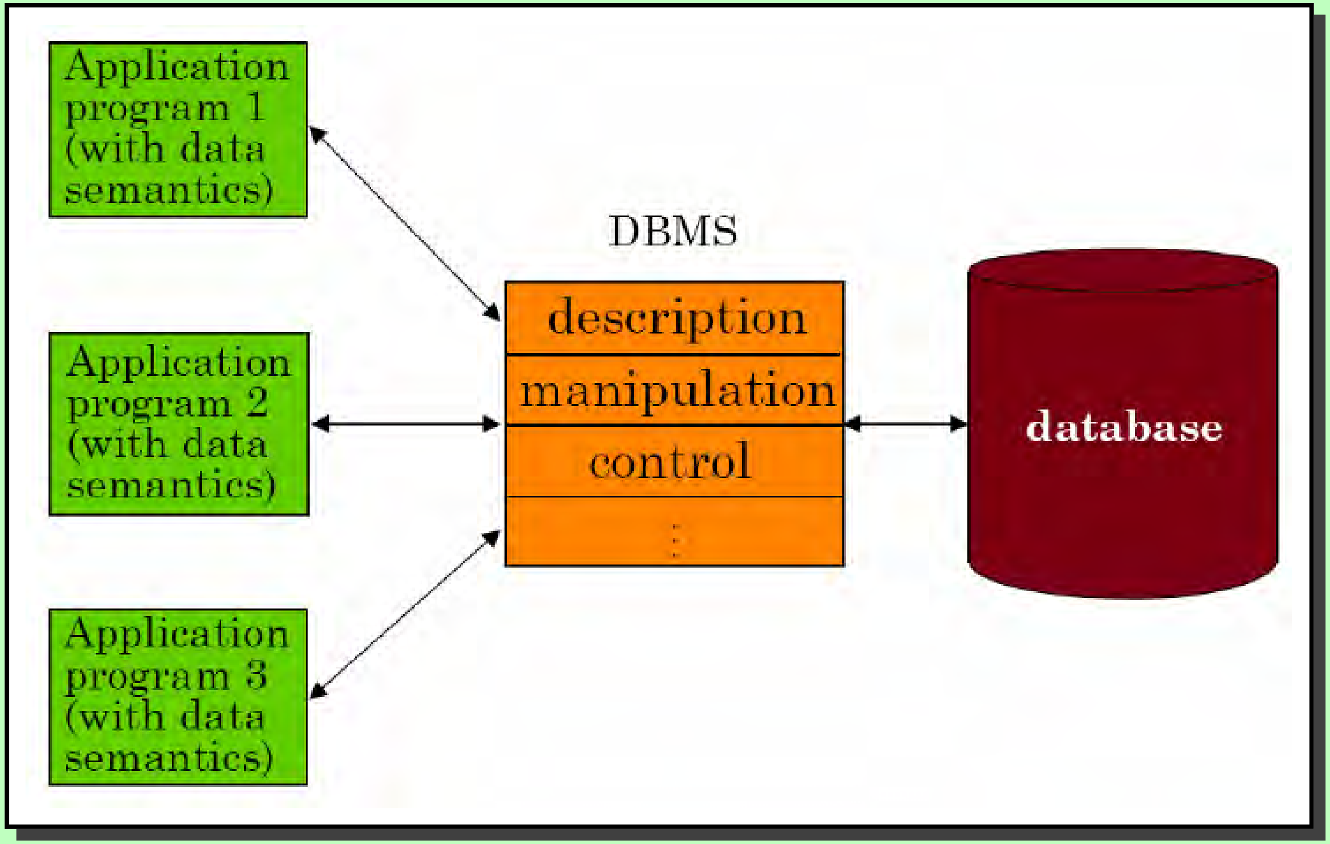
Data Independence

- In the old days, programs stored data in regular files
- Each program has to maintain its own data
  - huge overhead
  - error-prone



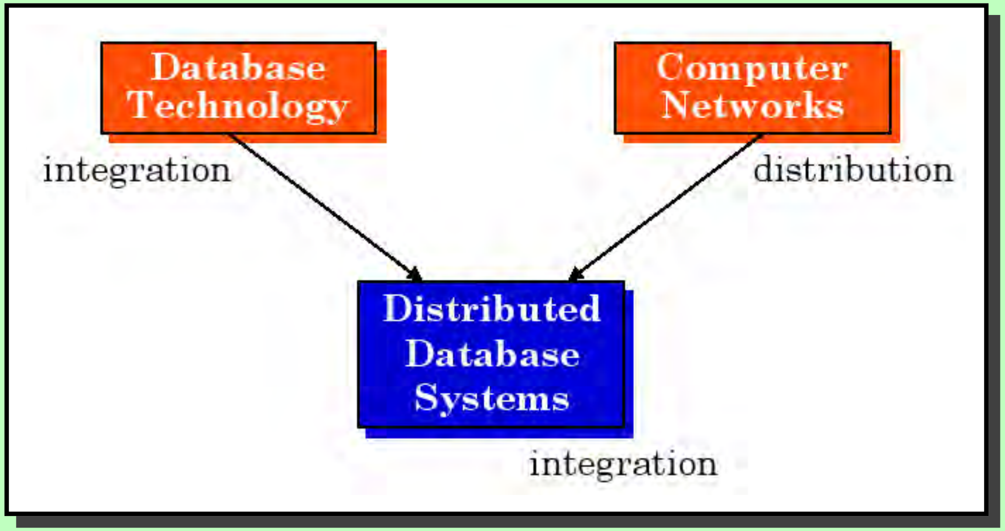
Data Independence . . .

- The development of **DBMS** helped to fully achieve data independence (transparency)
- Provide **centralized** and controlled data maintenance and access
- Application is immune to physical and logical file organization


Application program 1 (with data semantics)
Application program 2 (with data semantics)
Application program 3 (with data semantics)
DBMS
description
manipulation
control
⋮
database

Data Independence . . .

- Distributed database system is the union of what appear to be two diametrically opposed approaches to data processing: **database systems** and **computer network**
  - Computer networks promote a mode of work that goes against centralization
- Key issues to understand this combination
  - The most important objective of DB technology is **integration** not centralization
  - Integration is possible without centralization, i.e., integration of databases and networking does not mean centralization (in fact quite opposite)
- Goal of distributed database systems: achieve data integration **and** data distribution transparency



Distributed Computing/Data Processing

- A **distributed computing system** is a collection of autonomous processing elements that are interconnected by a computer network. The elements cooperate in order to perform the assigned task.
- The term “distributed” is very broadly used. The exact meaning of the word depends on the context.
- Synonymous terms:
  - distributed function
  - distributed data processing
  - multiprocessors/multicomputers
  - satellite processing
  - back-end processing
  - dedicated/special purpose computers
  - timeshared systems
  - functionally modular systems

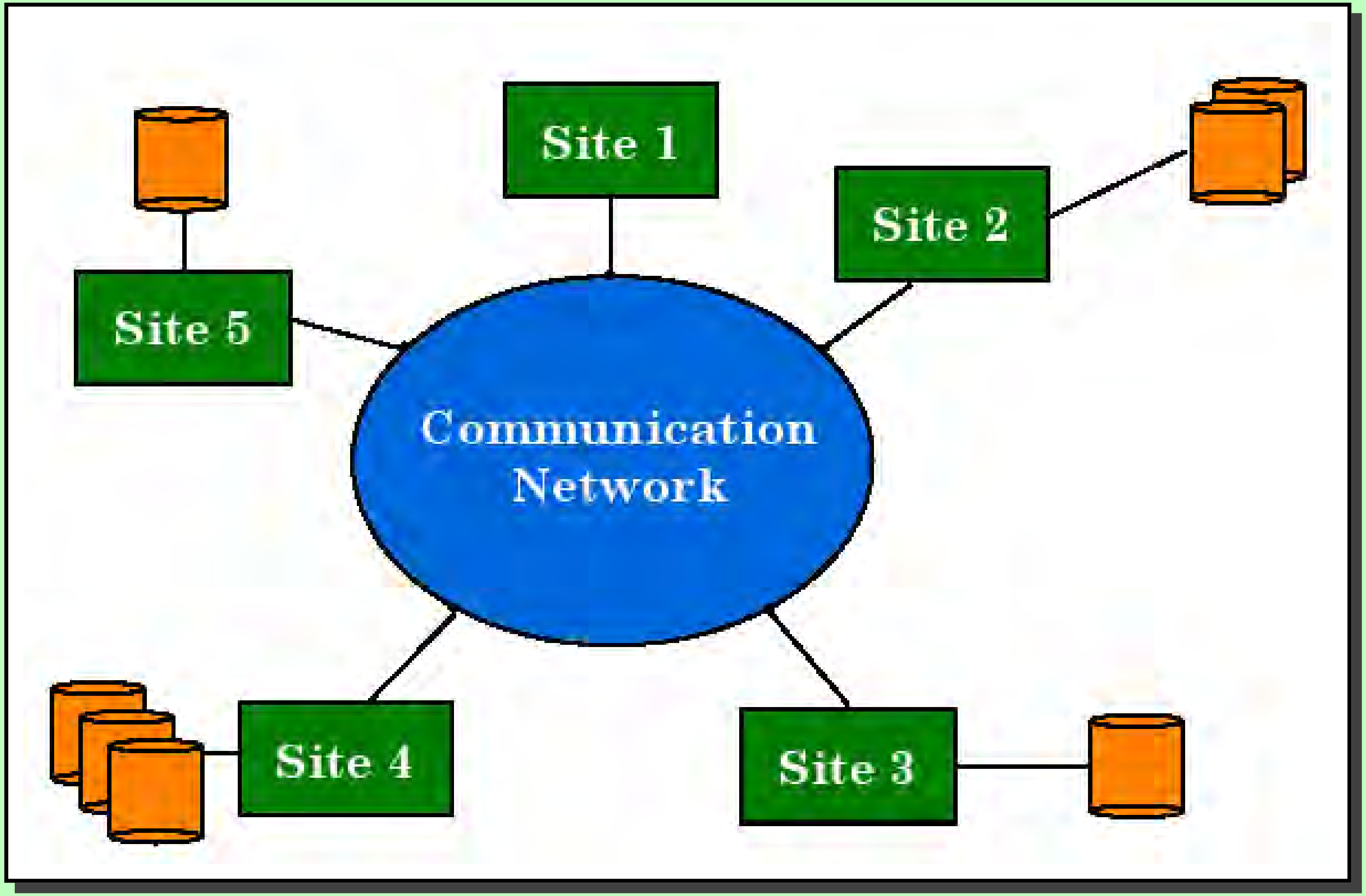
Distributed Computing/Data Processing . . .

- What can be distributed?
  - Processing logic
  - Functions
  - Data
  - Control
- Classification of distributed systems with respect to various criteria
  - Degree of coupling, i.e., how closely the processing elements are connected
    - e.g., measured as ratio of amount of data exchanged to amount of local processing
    - weak coupling, strong coupling
  - Interconnection structure
    - point-to-point connection between processing elements
    - common interconnection channel
  - Synchronization
    - synchronous
    - asynchronous

Definition of DDB and DDBMS

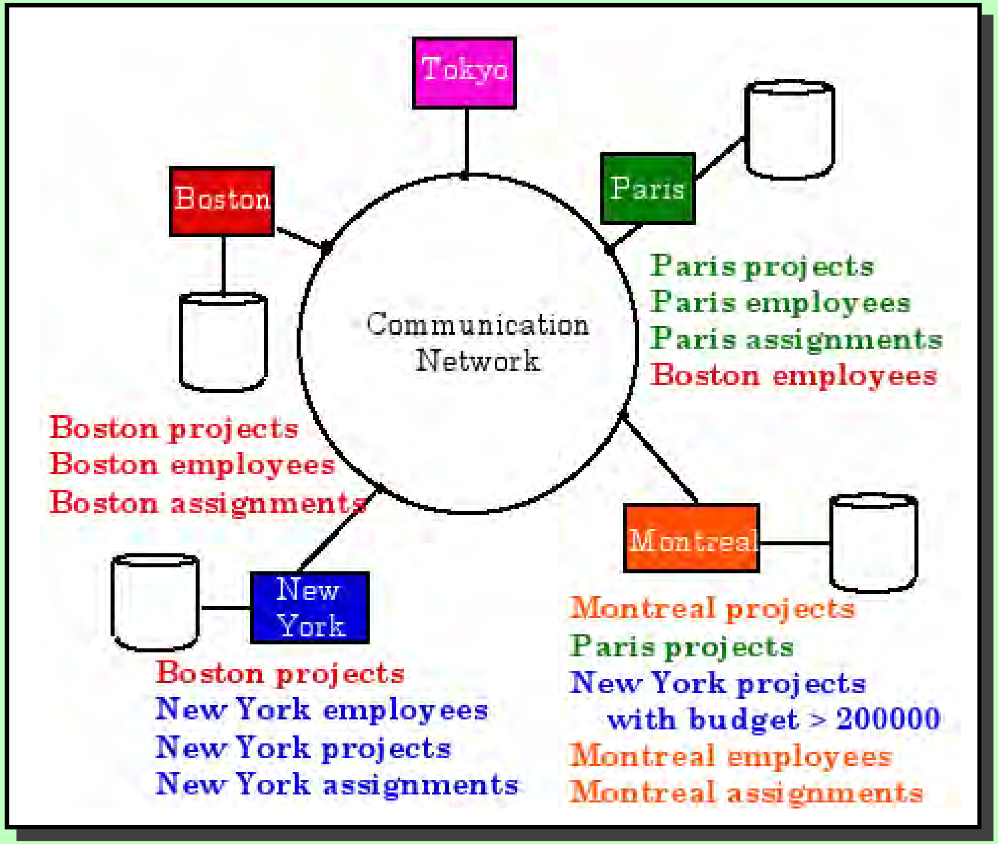
- A **distributed database** (DDB) is a collection of multiple, logically interrelated databases distributed over a computer network
- A **distributed database management system** (DDBMS) is the software that manages the DDB and provides an access mechanism that makes this distribution transparent to the users
- The terms DDBMS and DDBS are often used interchangeably
- 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 (we do not consider multiprocessors in DDBMS, cf. parallel systems)
  - DDBS is a database, not a collection of files (cf. relational data model). Placement and query of data is impacted by the access patterns of the user
  - DDBMS is a collections of DBMSs (not a remote file system)

Definition of DDB and DDBMS . . .



Definition of DDB and DDBMS . . .

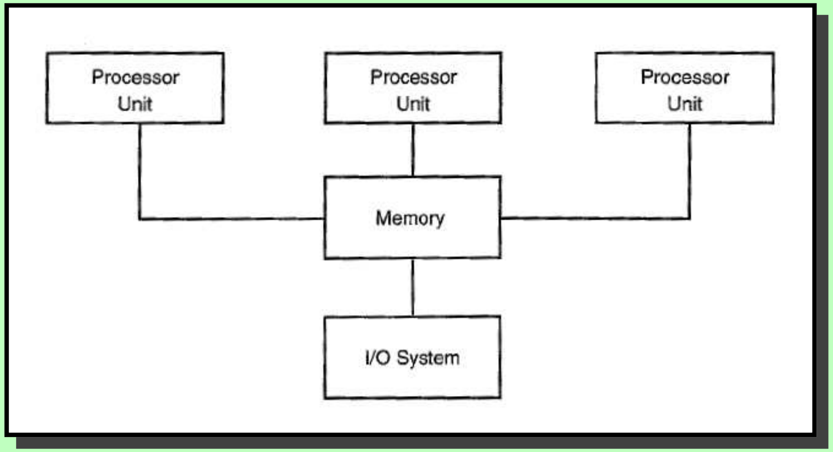
- **Example:** Database consists of 3 relations `employees`, `projects`, and `assignment` which are partitioned and stored at different sites (fragmentation).



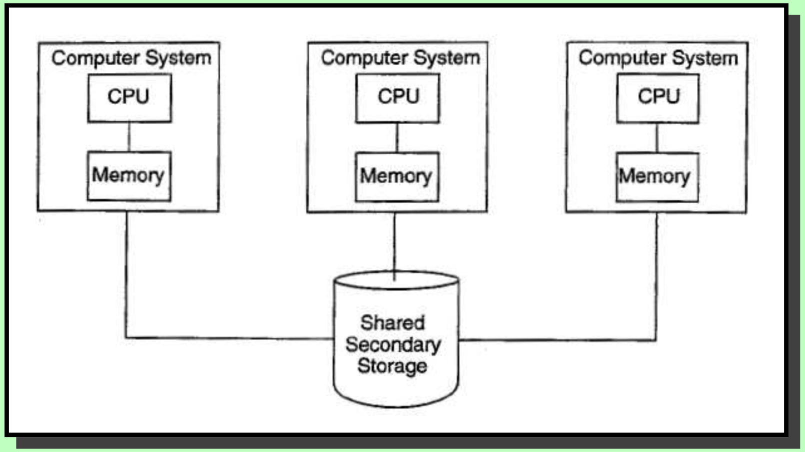
- What are the problems with queries, transactions, concurrency, and reliability?

What is not a DDBS?

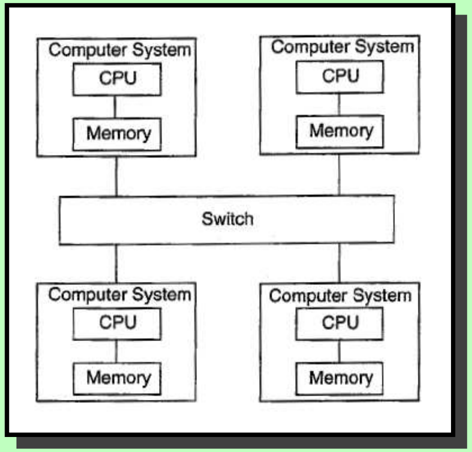
- The following systems are parallel database systems and are quite different from (though related to) distributed DB systems



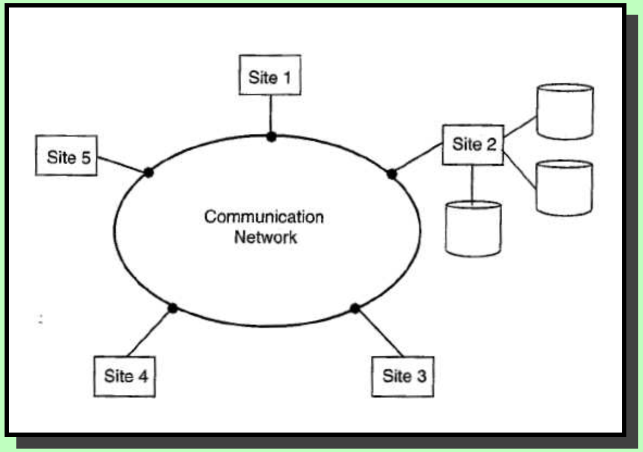
Shared Memory



Shared Disk



Shared Nothing



Central Databases

Applications

- Manufacturing, especially multi-plant manufacturing
- Military command and control
- Airlines
- Hotel chains
- Any organization which has a decentralized organization structure

Promises of DDBSs

Distributed Database Systems deliver the following advantages:

- Higher reliability
- Improved performance
- Easier system expansion
- Transparency of distributed and replicated data

Promises of DDBSs . . .

Higher reliability

- Replication of components
- No single points of failure
- e.g., a broken communication link or processing element does not bring down the entire system
- Distributed transaction processing guarantees the consistency of the database and concurrency

Promises of DDBSs . . .

Improved performance

- Proximity of data to its points of use
  - Reduces remote access delays
  - Requires some support for fragmentation and replication
- Parallelism in execution
  - Inter-query parallelism
  - Intra-query parallelism
- Update and read-only queries influence the design of DDBSs substantially
  - If mostly read-only access is required, as much as possible of the data should be replicated
  - Writing becomes more complicated with replicated data

Promises of DDBSs . . .

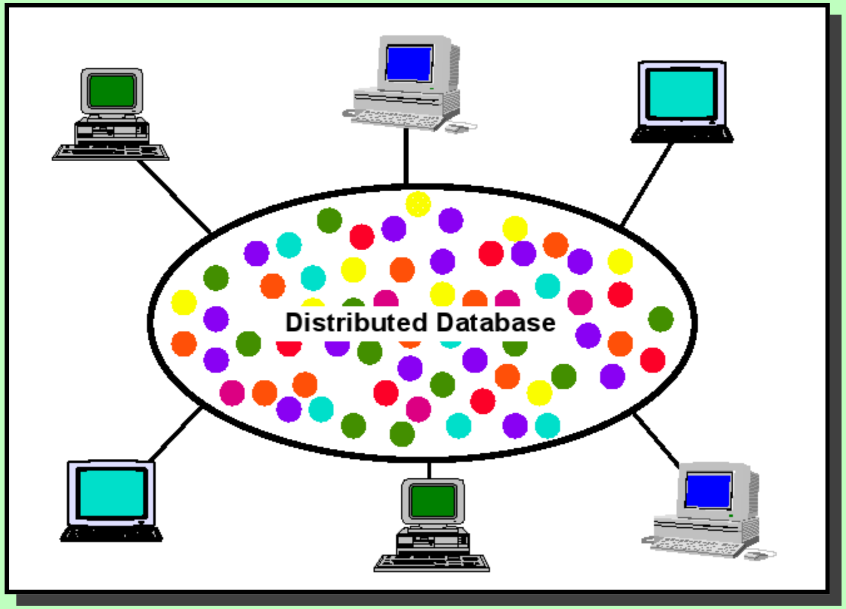
Easier system expansion

- Issue is database scaling
- Emergence of microprocessor and workstation technologies
  - Network of workstations much cheaper than a single mainframe computer
- Data communication cost versus telecommunication cost
- Increasing database size

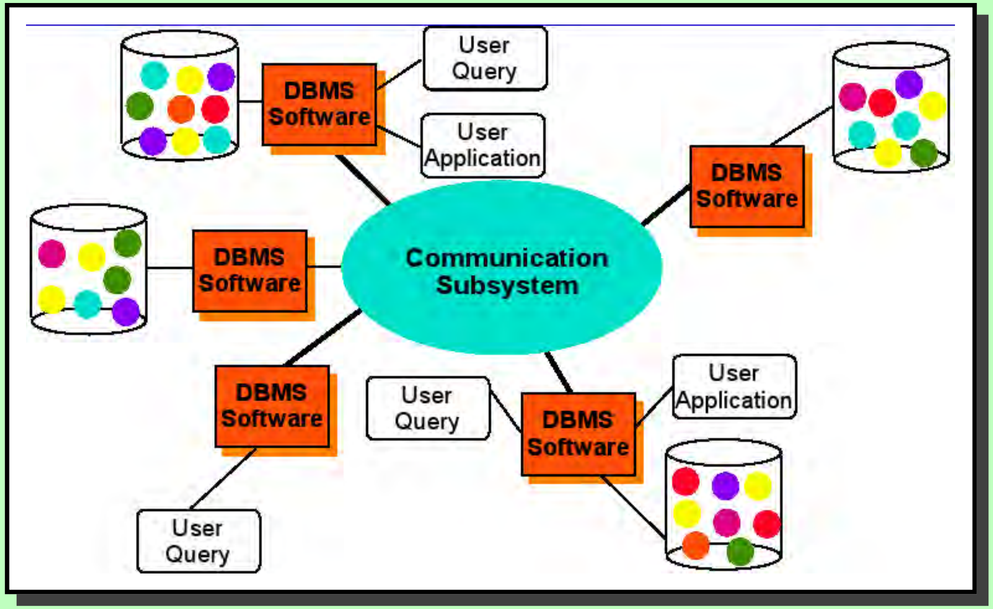
Promises of DDBSs . . .

Transparency

- Refers to the separation of the higher-level semantics of the system from the lower-level implementation issues
- A transparent system “hides” the implementation details from the users.
- A fully transparent DBMS provides high-level support for the development of complex applications.



(a) User wants to see one database



(b) Programmer sees many databases

Promises of DDBSs . . .

Various forms of **transparency** can be distingushed for DDBMSs:

- Network transparency (also called distribution transparency)
  - Location transparency
  - Naming transparency
- Replication transparency
- Fragmentation transparency
- Transaction transparency
  - Concurrency transparency
  - Failure transparency
- Performance transparency

Promises of DDBSs . . .

- **Network/Distribution transparency** allows a user to perceive a DDBS as a single, logical entity
- The user is protected from the operational details of the network (or even does not know about the existence of the network)
- The user does not need to know the location of data items and a command used to perform a task is independent from the location of the data and the site the task is performed (**location transparency**)
- A unique name is provided for each object in the database (**naming transparency**)
  - In absence of this, users are required to embed the location name as part of an identifier

Promises of DDBSs . . .

Different ways to ensure naming transparency:

- Solution 1: Create a central name server; however, this results in
  - loss of some local autonomy
  - central site may become a bottleneck
  - low availability (if the central site fails remaining sites cannot create new objects)
- Solution 2: Prefix object with identifier of site that created it
  - e.g., branch created at site S1 might be named S1.BRANCH
  - Also need to identify each fragment and its copies
  - e.g., copy 2 of fragment 3 of Branch created at site S1 might be referred to as S1.BRANCH.F3.C2
- An approach that resolves these problems uses aliases for each database object
  - Thus, S1.BRANCH.F3.C2 might be known as local branch by user at site S1
  - DDBMS has task of mapping an alias to appropriate database object

Promises of DDBSs . . .

- **Replication transparency** ensures that the user is not involved in the managment of copies of some data
- The user should even not be aware about the existence of replicas, rather should work as if there exists a single copy of the data
- Replication of data is needed for various reasons
  - e.g., increased efficiency for read-only data access

Promises of DDBSs . . .

- **Fragmentation transparency** ensures that the user is not aware of and is not involved in the fragmentation of the data
- The user is not involved in finding query processing strategies over fragments or formulating queries over fragments
  - The evaluation of a query that is specified over an entire relation but now has to be performed on top of the fragments requires an appropriate query evaluation strategy
- Fragmentation is commonly done for reasons of performance, availability, and reliability
- Two fragmentation alternatives
  - Horizontal fragmentation: divide a relation into a subsets of tuples
  - Vertical fragmentation: divide a relation by columns

Promises of DDBSs . . .

- **Transaction transparency** ensures that all distributed transactions maintain integrity and consistency of the DDB and support concurrency
- Each distributed transaction is divided into a number of sub-transactions (a sub-transaction for each site that has relevant data) that concurrently access data at different locations
- DDBMS must ensure the indivisibility of both the global transaction and each of the sub-transactions
- Can be further divided into
  - Concurrency transparency
  - Failure transparency

Promises of DDBSs . . .

- **Concurrency transparency** guarantees that transactions must execute independently and are logically consistent, i.e., executing a set of transactions in parallel gives the same result as if the transactions were executed in some arbitrary serial order.

- Same fundamental principles as for centralized DBMS, but more complicated to realize:
  - DDBMS must ensure that global and local transactions do not interfere with each other
  - DDBMS must ensure consistency of all sub-transactions of global transaction

- Replication makes concurrency even more complicated
  - If a copy of a replicated data item is updated, update must be propagated to all copies
  - Option 1: Propagate changes as part of original transaction, making it an atomic operation; however, if one site holding a copy is not reachable, then the transaction is delayed until the site is reachable.
  - Option 2: Limit update propagation to only those sites currently available; remaining sites are updated when they become available again.
  - Option 3: Allow updates to copies to happen asynchronously, sometime after the original update; delay in regaining consistency may range from a few seconds to several hours

Promises of DDBSs . . .

- **Failure transparency**: DDBMS must ensure atomicity and durability of the global transaction, i.e., the sub-transactions of the global transaction either all commit or all abort.
- Thus, DDBMS must synchronize global transaction to ensure that all sub-transactions have completed successfully before recording a final COMMIT for the global transaction
- The solution should be robust in presence of site and network failures

Promises of DDBSs . . .

- **Performance transparency**: DDBMS must perform as if it were a centralized DBMS
  - DDBMS should not suffer any performance degradation due to the distributed architecture
  - DDBMS should determine most cost-effective strategy to execute a request
- Distributed Query Processor (DQP) maps data request into an ordered sequence of operations on local databases
- DQP must consider fragmentation, replication, and allocation schemas
- DQP has to decide:
  - which fragment to access
  - which copy of a fragment to use
  - which location to use
- DQP produces execution strategy optimized with respect to some cost function
- Typically, costs associated with a distributed request include: I/O cost, CPU cost, and communication cost

Complicating Factors

- Complexity
- Cost
- Security
- Integrity control more difficult
- Lack of standards
- Lack of experience
- Database design more complex

Technical Problems to be Studied . . .

- Distributed database design
  - How to fragment the data?
  - Partitioned data vs. replicated data?
- Distributed query processing
  - Design algorithms that analyze queries and convert them into a series of data manipulation operations
  - Distribution of data, communication costs, etc. has to be considered
  - Find optimal query plans
- Distributed directory management
- Distributed concurrency control
  - Synchronization of concurrent accesses such that the integrity of the DB is maintained
  - Integrity of multiple copies of (parts of) the DB have to be considered (mutual consistency)
- Distributed deadlock management
  - Deadlock management: prevention, avoidance, detection/recovery

Technical Problems to be Studied . . .

- Reliability
  - How to make the system resilient to failures
  - Atomicity and Durability
- Heterogeneous databases
  - If there is no homogeneity among the DBs at various sites either in terms of the way data is logically structured (data model) or in terms of the access mechanisms (language), it becomes necessary to provide translation mechanisms

Conclusion

- A distributed database (DDB) is a collection of multiple, logically interrelated databases distributed over a computer network
- Data stored at a number of sites, the sites are connected by a network. DDB supports the relational model. DDB is not a remote file system
- Transparent system ‘hides’ the implementation details from the users
  - Distribution transparency
  - Network transparency
  - Transaction transparency
  - Performance transparency
- Programming a distributed database involves:
  - Distributed database design
  - Distributed query processing
  - Distributed directory management
  - Distributed concurrency control
  - Distributed deadlock management
  - Reliability

Chapter 2: DDBMS Architecture

- Definition of the DDBMS Architecture
- ANSI/SPARC Standard
- Global, Local, External, and Internal Schemas, Example
- DDBMS Architectures
- Components of the DDBMS

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Definition

- **Architecture:** The architecture of a system defines its structure:
  - the components of the system are identified;
  - the function of each component is specified;
  - the interrelationships and interactions among the components are defined.
- Applies both for computer systems as well as for software systems, e.g,
  - division into modules, description of modules, etc.
  - architecture of a computer
- There is a close relationship between the architecture of a system, standardisation efforts, and a reference model.

Motivation for Standardization of DDBMS Architecture

- DDBMS might be implemented as homogeneous or heterogeneous DDBMS
- **Homogeneous** DDBMS
  - All sites use same DBMS product
  - It is much easier to design and manage
  - The approach provides incremental growth and allows increased performance
- **Heterogeneous** DDBMS
  - Sites may run different DBMS products, with possibly different underlying data models
  - This occurs when sites have implemented their own databases first, and integration is considered later
  - Translations are required to allow for different hardware and/or different DBMS products
  - Typical solution is to use gateways

⇒ A common standard to implement DDBMS is needed!

Standardization

- The standardization efforts in databases developed reference models of DBMS.
- **Reference Model**: A conceptual framework whose purpose is to divide standardization work into manageable pieces and to show at a general level how these pieces are related to each other.
- A reference model can be thought of as an **idealized architectural model** of the system.
- Commercial systems might deviate from reference model, still they are useful for the standardization process
- A reference model can be described according to 3 different approaches:
  - component-based
  - function-based
  - data-based

Standardization . . .

- **Components-based**
  - Components of the system are defined together with the interrelationships between the components
  - Good for design and implementation of the system
  - It might be difficult to determine the functionality of the system from its components

Standardization . . .

- **Function-based**
  - Classes of users are identified together with the functionality that the system will provide for each class
  - Typically a hierarchical system with clearly defined interfaces between different layers
  - The objectives of the system are clearly identified.
  - Not clear how to achieve the objectives
  - Example: ISO/OSI architecture of computer networks

Standardization . . .

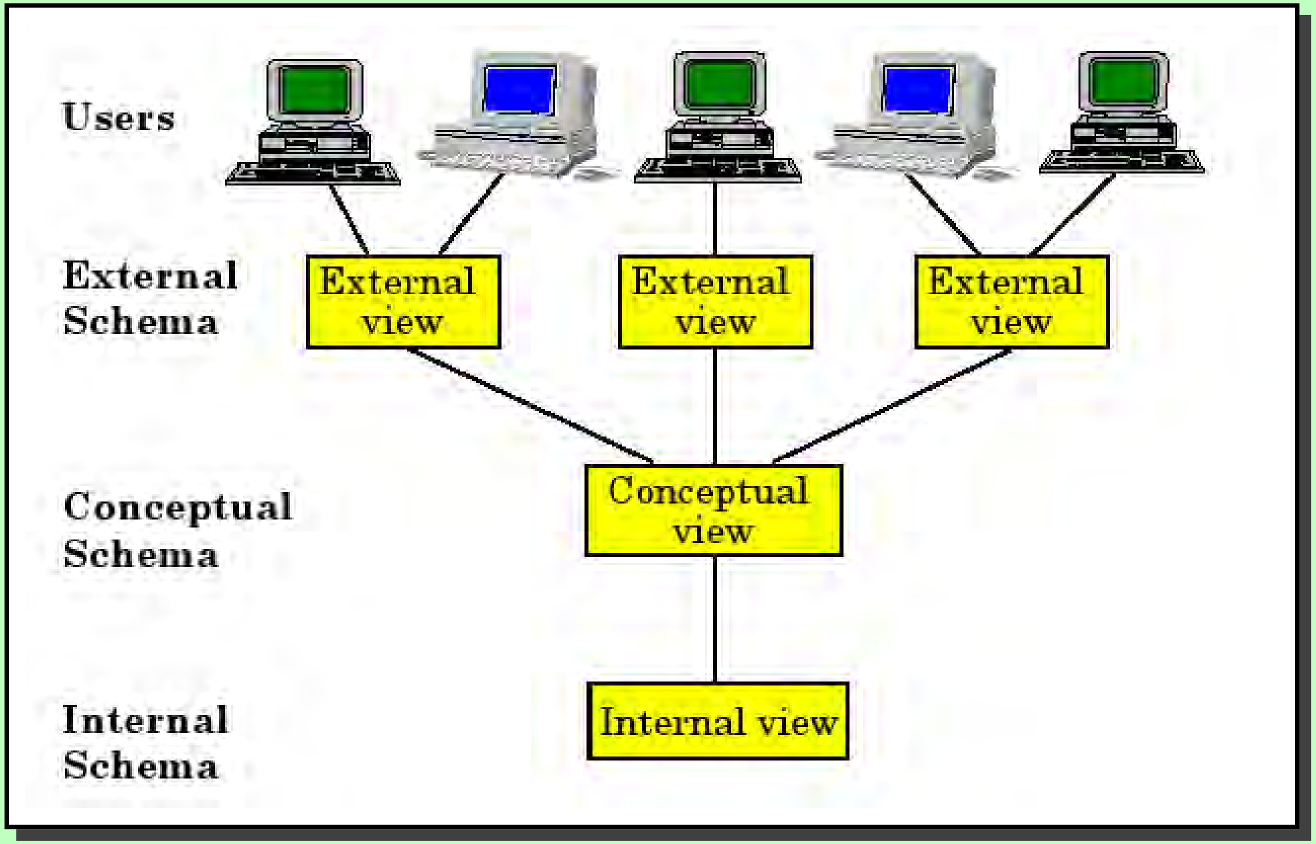
- **Data-based**
  - Identify the different types of the data and specify the functional units that will realize and/or use data according to these views
  - Gives central importance to data (which is also the central resource of any DBMS) → Claimed to be the preferable choice for standardization of DBMS
  - The full architecture of the system is not clear without the description of functional modules.
  - Example: ANSI/SPARC architecture of DBMS

Standardization . . .

- The interplay among the 3 approaches is important:
  - Need to be used together to define an architectural model
  - Each brings a different point of view and serves to focus on different aspects of the model

ANSI/SPARC Architecture of DBMS

- ANSI/SPARC architecture is based on data
- 3 views of data: external view, conceptual view, internal view
- Defines a total of 43 interfaces between these views



Example

- Conceptual schema: Provides enterprise view of entire database

```
RELATION EMP [
  KEY = {ENO}
  ATTRIBUTES = {
    ENO   :  CHARACTER(9)
    ENAME:  CHARACTER(15)
    TITLE:  CHARACTER(10)
  }
]

RELATION PAY [
  KEY = {TITLE}
  ATTRIBUTES = {
    TITLE:  CHARACTER(10)
    SAL   :  NUMERIC(6)
  }
]
```

```
RELATION PROJ [
  KEY = {PNO}
  ATTRIBUTES = {
    PNO    :  CHARACTER(7)
    PNAME :  CHARACTER(20)
    BUDGET:  NUMERIC(7)
    LOC    :  CHARACTER(15)

]

RELATION ASG [
  KEY = {ENO,PNO}
  ATTRIBUTES = {
    ENO :  CHARACTER(9)
    PNO :  CHARACTER(7)
    RESP:  CHARACTER(10)
    DUR :  NUMERIC(3)
  }
]
```

Example . . .

- Internal schema: Describes the storage details of the relations.
  - Relation EMP is stored on an indexed file
  - Index is defined on the key attribute ENO and is called EMINX
  - A HEADER field is used that might contain flags (delete, update, etc.)

```
INTERNAL_REL EMPL [
  INDEX ON E#  CALL EMINX
  FIELD =
    HEADER:  BYTE(1)
    E#    :  BYTE(9)
    ENAME :  BYTE(15)
    TIT   :  BYTE(10)

]
```

Conceptual schema:

```
RELATION EMP [
  KEY = {ENO}
  ATTRIBUTES = {
    ENO   :  CHARACTER(9)
    ENAME:  CHARACTER(15)
    TITLE:  CHARACTER(10)
  }
]
```

Example . . .

- External view: Specifies the view of different users/applications
  - Application 1: Calculates the payroll payments for engineers

```
CREATE VIEW PAYROLL (ENO, ENAME, SAL) AS
SELECT EMP.ENO,EMP.ENAME,PAY.SAL
FROM   EMP, PAY
WHERE  EMP.TITLE = PAY.TITLE
```

  - Application 2: Produces a report on the budget of each project

```
CREATE VIEW BUDGET(PNAME, BUD) AS
SELECT PNAME, BUDGET
FROM PROJ
```

Architectural Models for DDBMSs

- Architectural Models for DDBMSs (or more generally for multiple DBMSs) can be classified along three dimensions:
  - Autonomy
  - Distribution
  - Heterogeneity

Architectural Models for DDBMSs . . .

- **Autonomy**: Refers to the distribution of control (not of data) and indicates the degree to which individual DBMSs can operate independently.
  - *Tight integration*: a single-image of the entire database is available to any user who wants to share the information (which may reside in multiple DBs); realized such that one data manager is in control of the processing of each user request.
  - *Semiautonomous* systems: individual DBMSs can operate independently, but have decided to participate in a federation to make some of their local data sharable.
  - *Total isolation*: the individual systems are stand-alone DBMSs, which know neither of the existence of other DBMSs nor how to comunicate with them; there is no global control.
- Autonomy has different dimensions
  - *Design autonomy*: each individual DBMS is free to use the data models and transaction management techniques that it prefers.
  - *Communication autonomy*: each individual DBMS is free to decide what information to provide to the other DBMSs
  - *Execution autonomy*: each individual DBMS can execture the transactions that are submitted to it in any way that it wants to.

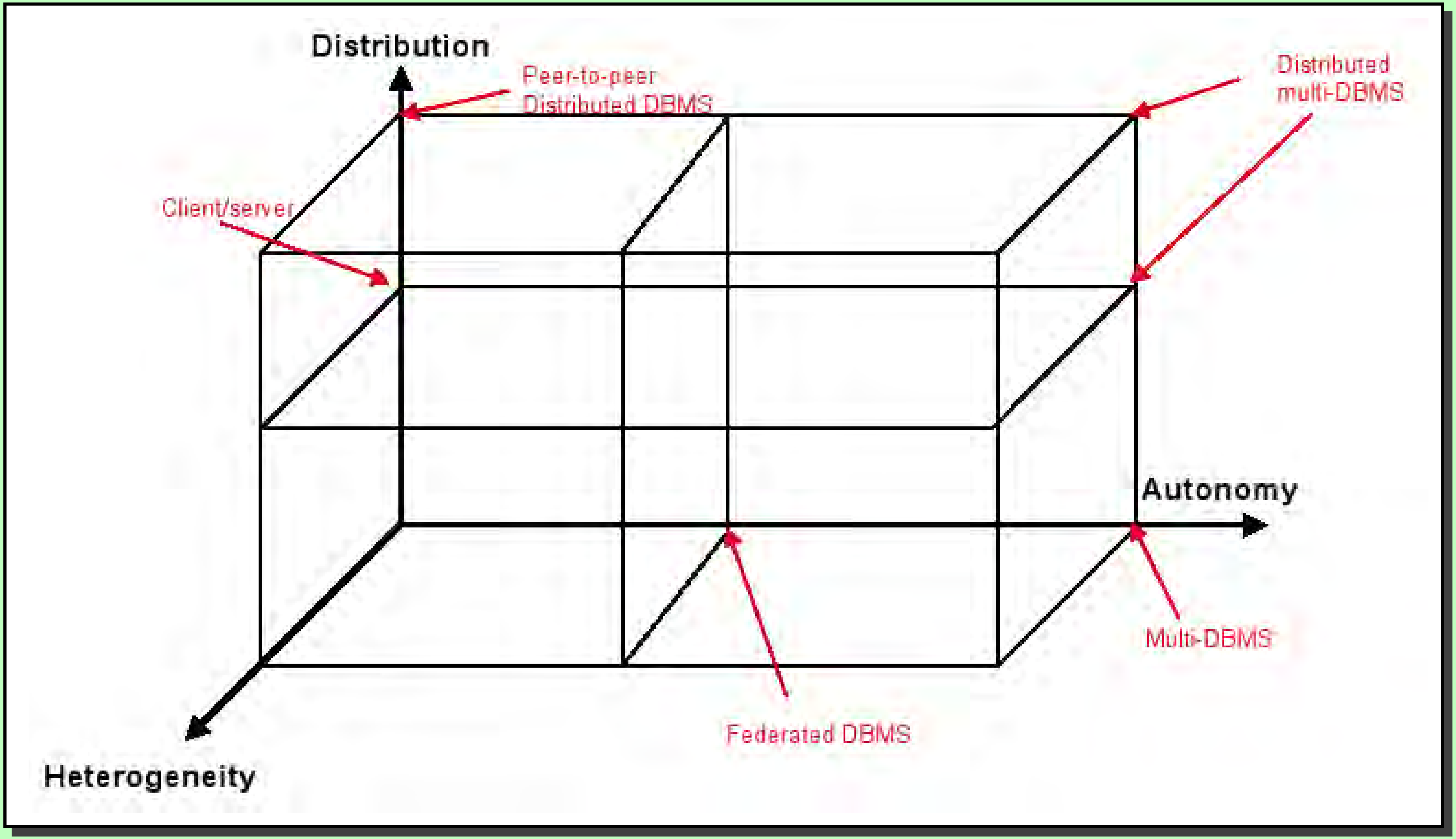
Architectural Models for DDBMSs . . .

- **Distribution**: Refers to the physical distribution of data over multiple sites.
  - *No distribution*: No distribution of data at all
  - *Client/Server distribution*:
    - Data are concentrated on the server, while clients provide application environment/user interface
    - First attempt to distribution
  - *Peer-to-peer distribution* (also called *full distribution*):
    - No distinction between client and server machine
    - Each machine has full DBMS functionality

Architectural Models for DDBMSs . . .

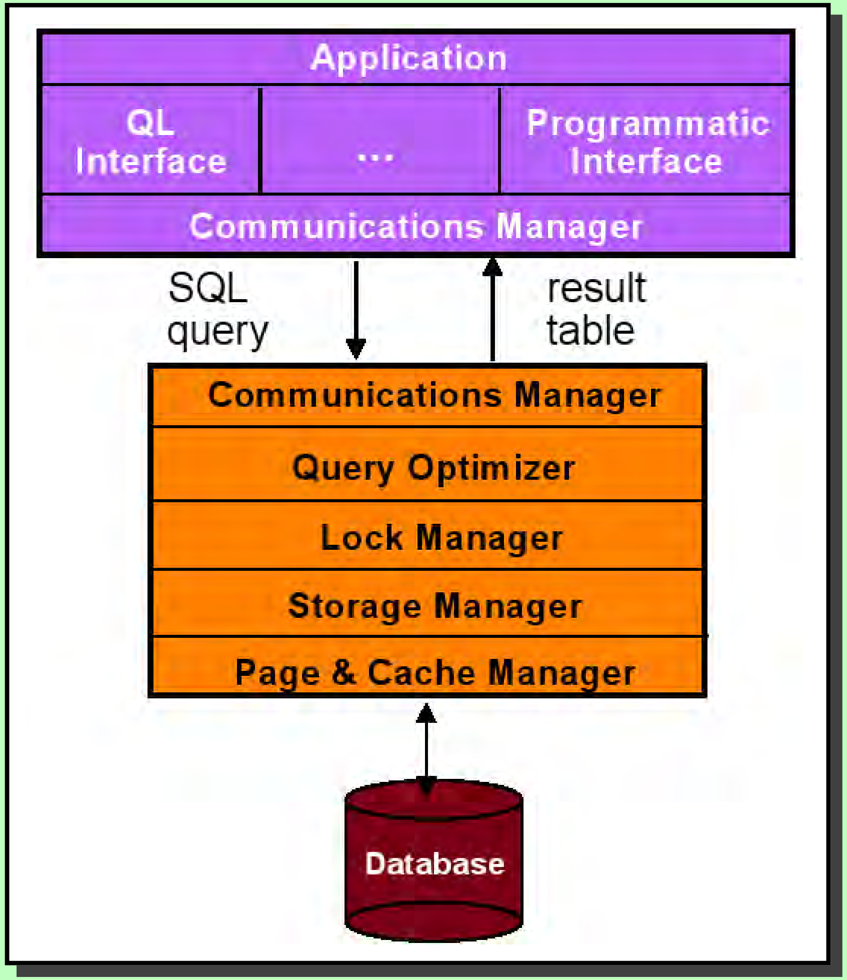
- **Heterogeneity**: Refers to heterogeneity of the components at various levels
  - hardware
  - communications
  - operating system
  - DB components (e.g., data model, query language, transaction management algorithms)

Architectural Models for DDBMSs . . .



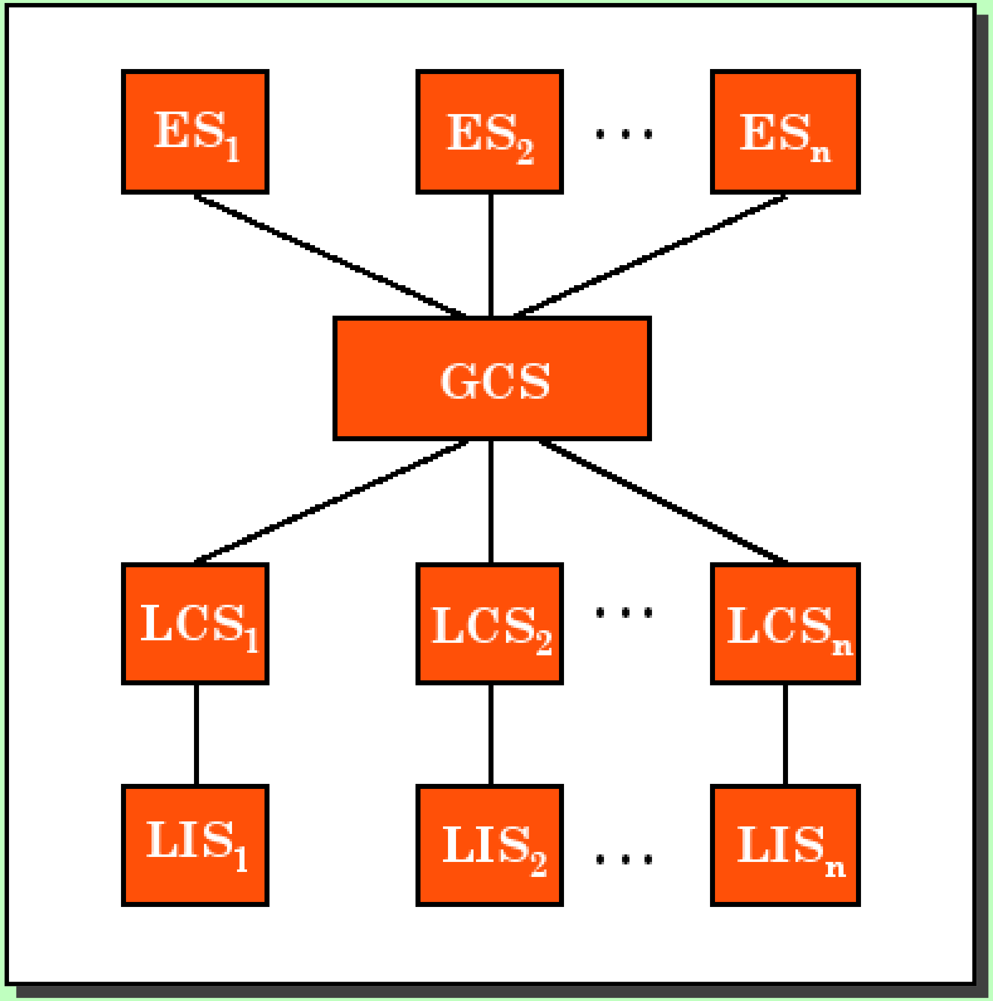
Client-Server Architecture for DDBMS (Data-based)

- General idea: Divide the functionality into two classes:
  - server functions
    - ∗ mainly data management, including query processing, optimization, transaction management, etc.
  - client functions
    - ∗ might also include some data management functions (consistency checking, transaction management, etc.) not just user interface
- Provides a two-level architecture
- More efficient division of work
- Different types of client/server architecture
  - Multiple client/single server
  - Multiple client/multiple server



Peer-to-Peer Architecture for DDBMS (Data-based)

- *Local internal schema* (LIS)
  - Describes the local physical data organization (which might be different on each machine)
- *Local conceptual schema* (LCS)
  - Describes logical data organization at each site
  - Required since the data are fragmented and replicated
- Global conceptual schema (GCS)
  - Describes the global logical view of the data
  - Union of the LCSs
- *External schema* (ES)
  - Describes the user/application view on the data

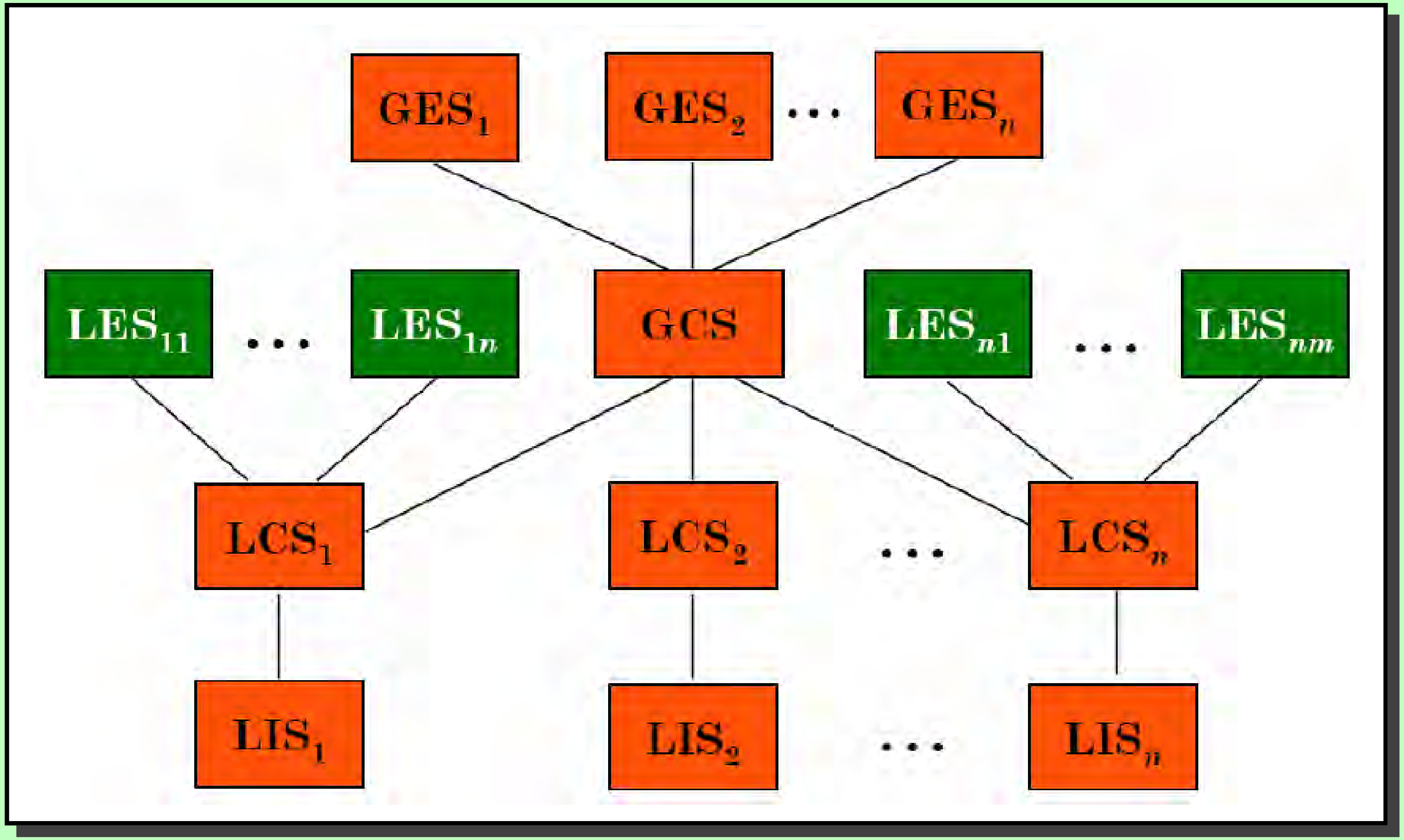


Multi-DBMS Architecture (Data-based)

- Fundamental difference to peer-to-peer DBMS is in the definition of the global conceptual schema (GCS)
  - In a MDBMS the GCS represents only the collection of *some* of the local databases that each local DBMS want to share.
- This leads to the question, whether the GCS should even exist in a MDBMS?
- Two different architecutre models:
  - Models with a GCS
  - Models without GCS

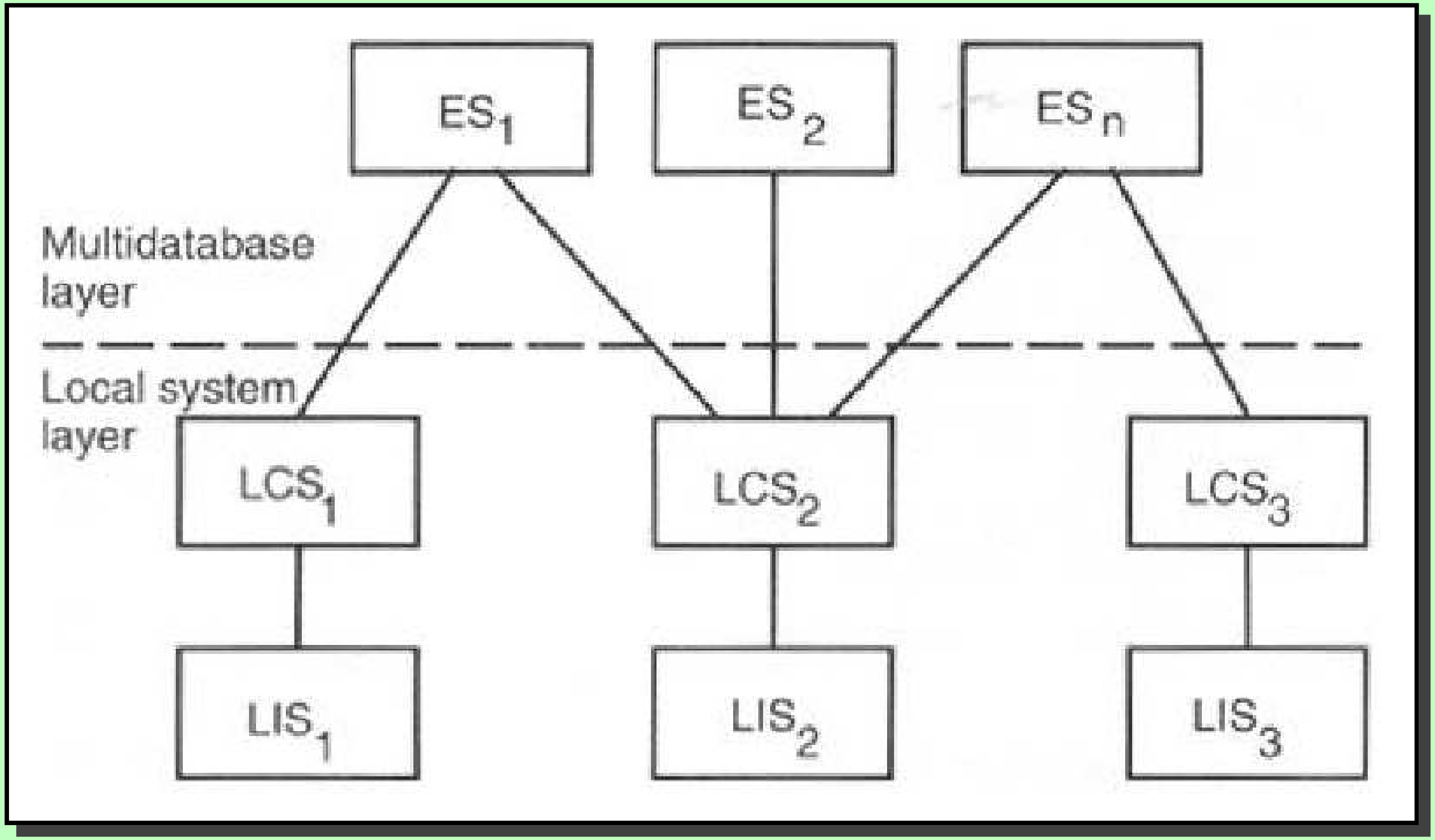
Multi-DBMS Architecture (Data-based) . . .

- Model with a GCS
  - GCS is the union of parts of the LCSs
  - Local DBMS define their own views on the local DB

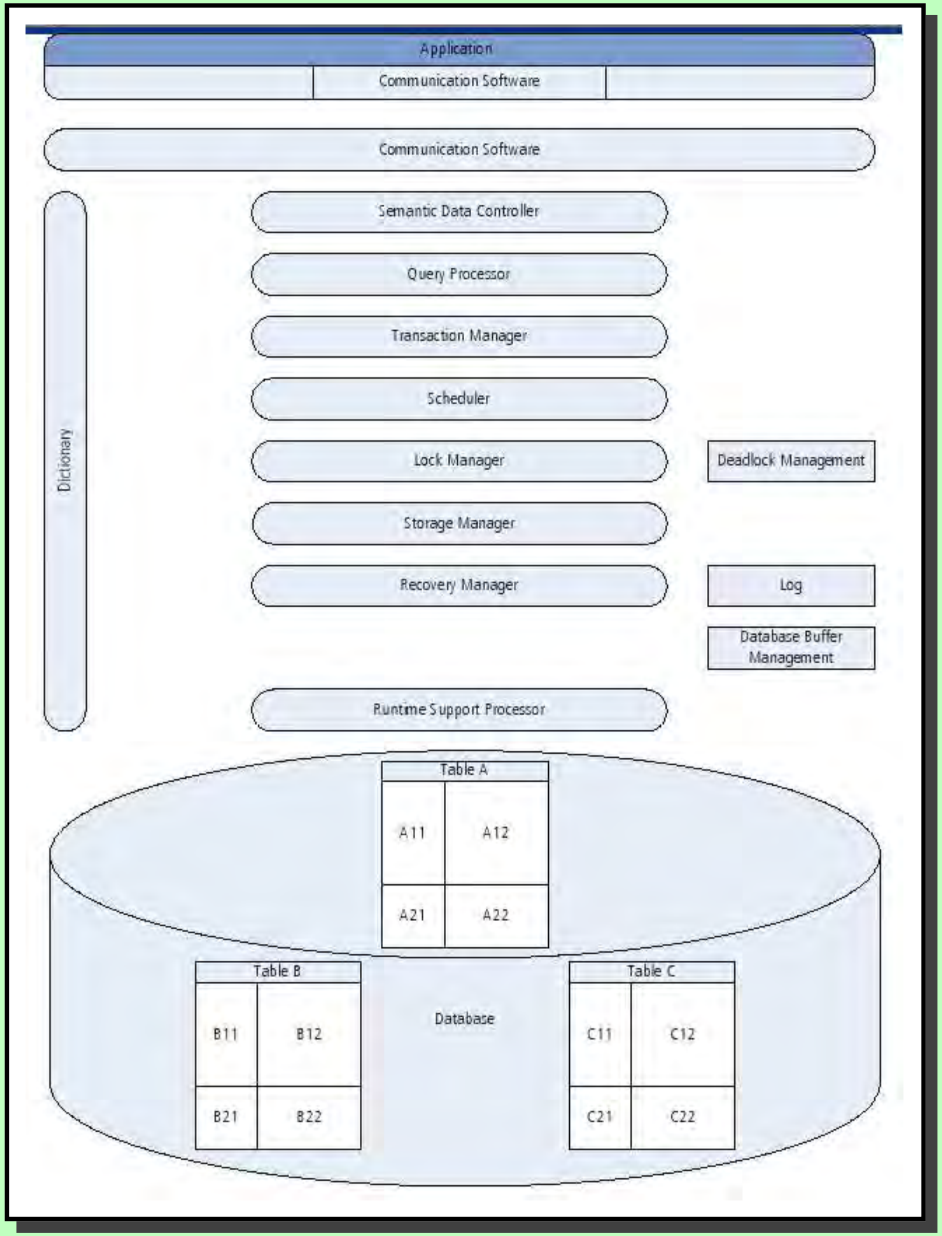


Multi-DBMS Architecture (Data-based) . . .

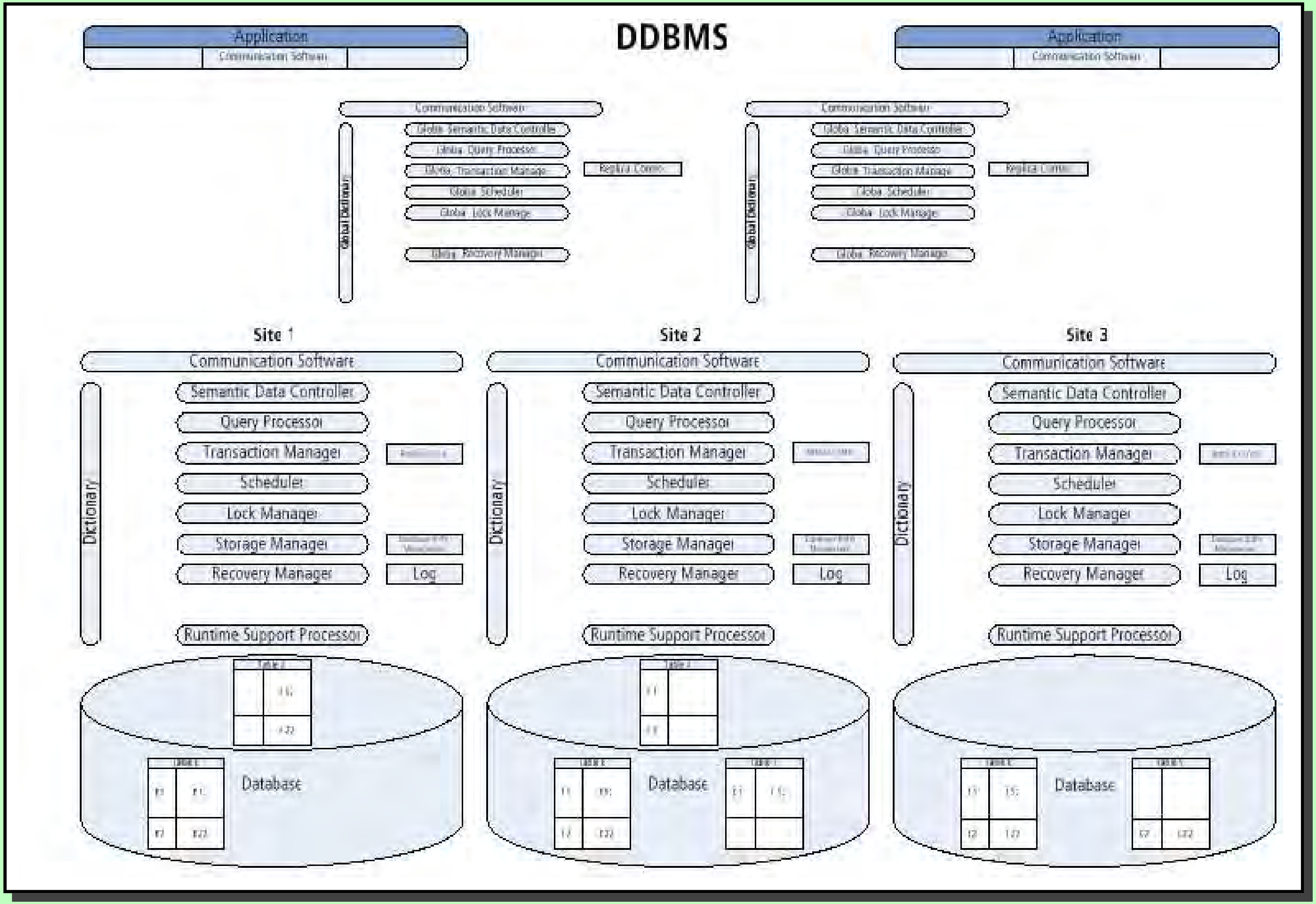
- Model without a GCS
  - The local DBMSs present to the multi-database layer the part of their local DB they are willing to share.
  - External views are defined on top of LCSs



Regular DBMS (Component-based)

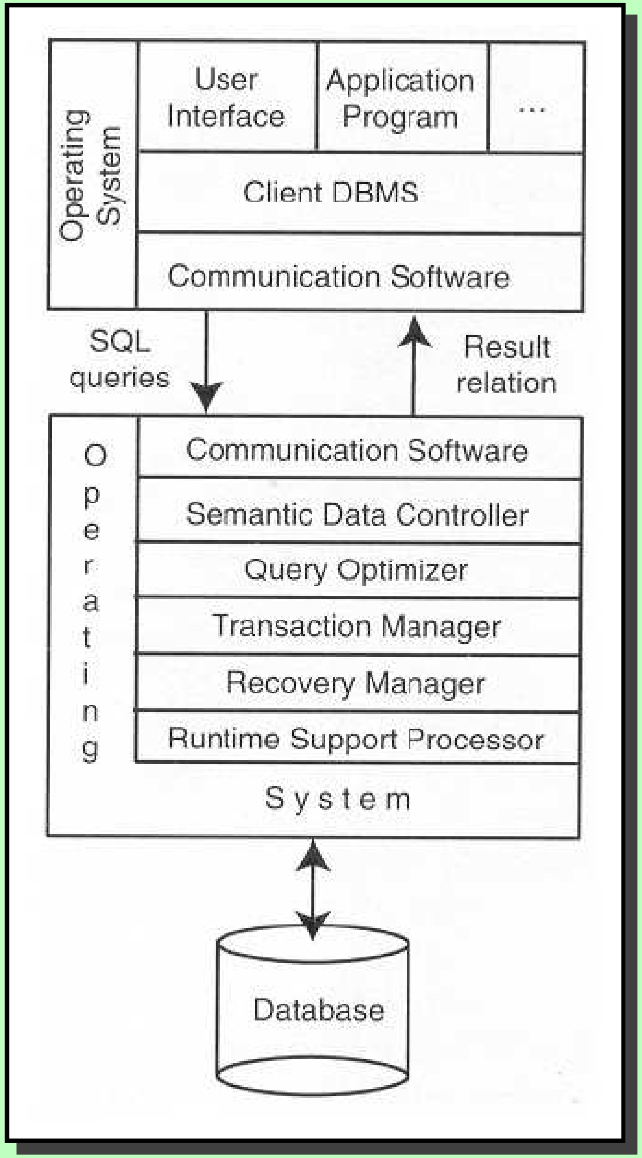


General DDBMS (Component-based)



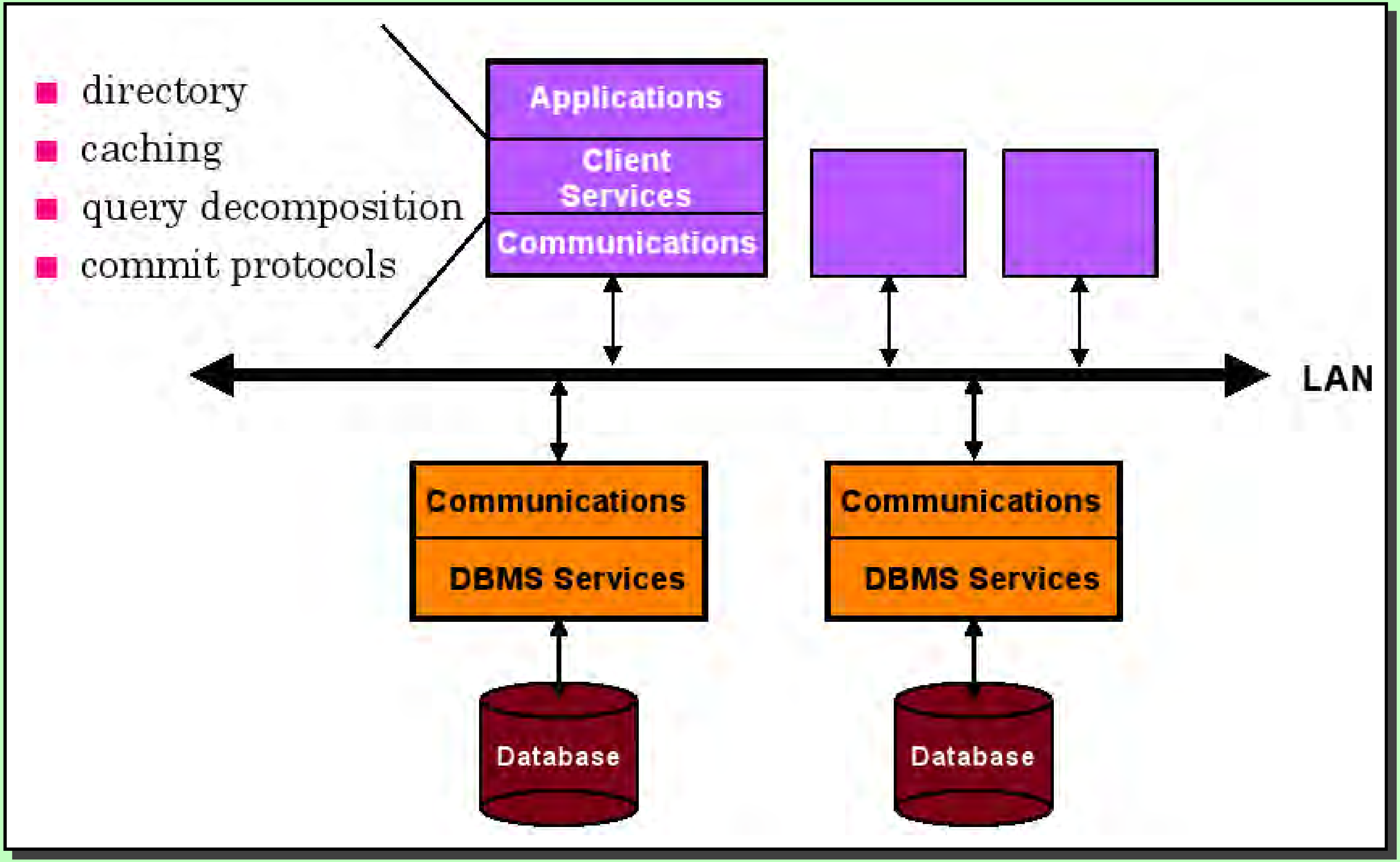
Client-Server Architecture (Component-based)

- One server, many clients



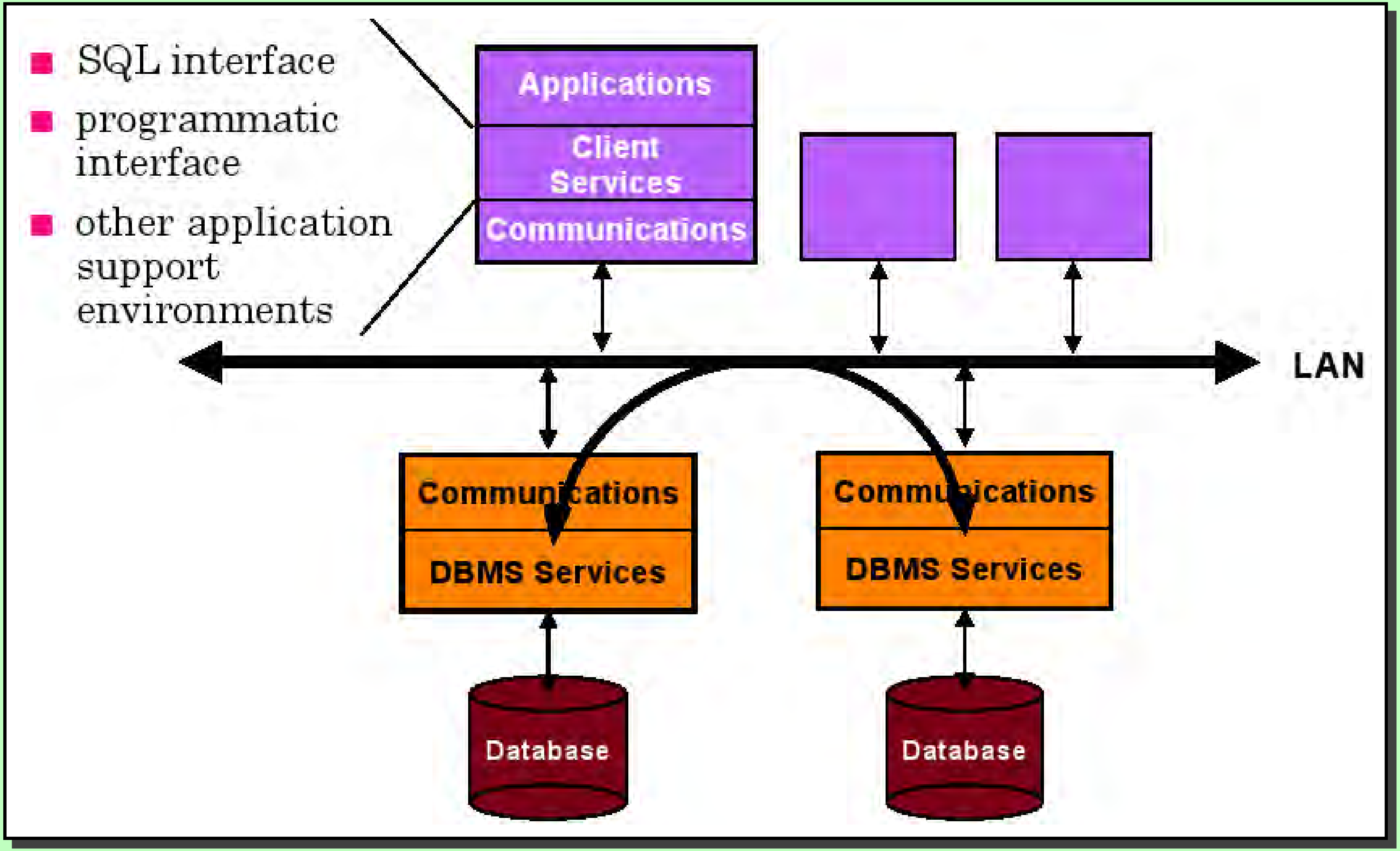
Components of Client-Server Architecture (Component-based)

- Many servers, many clients

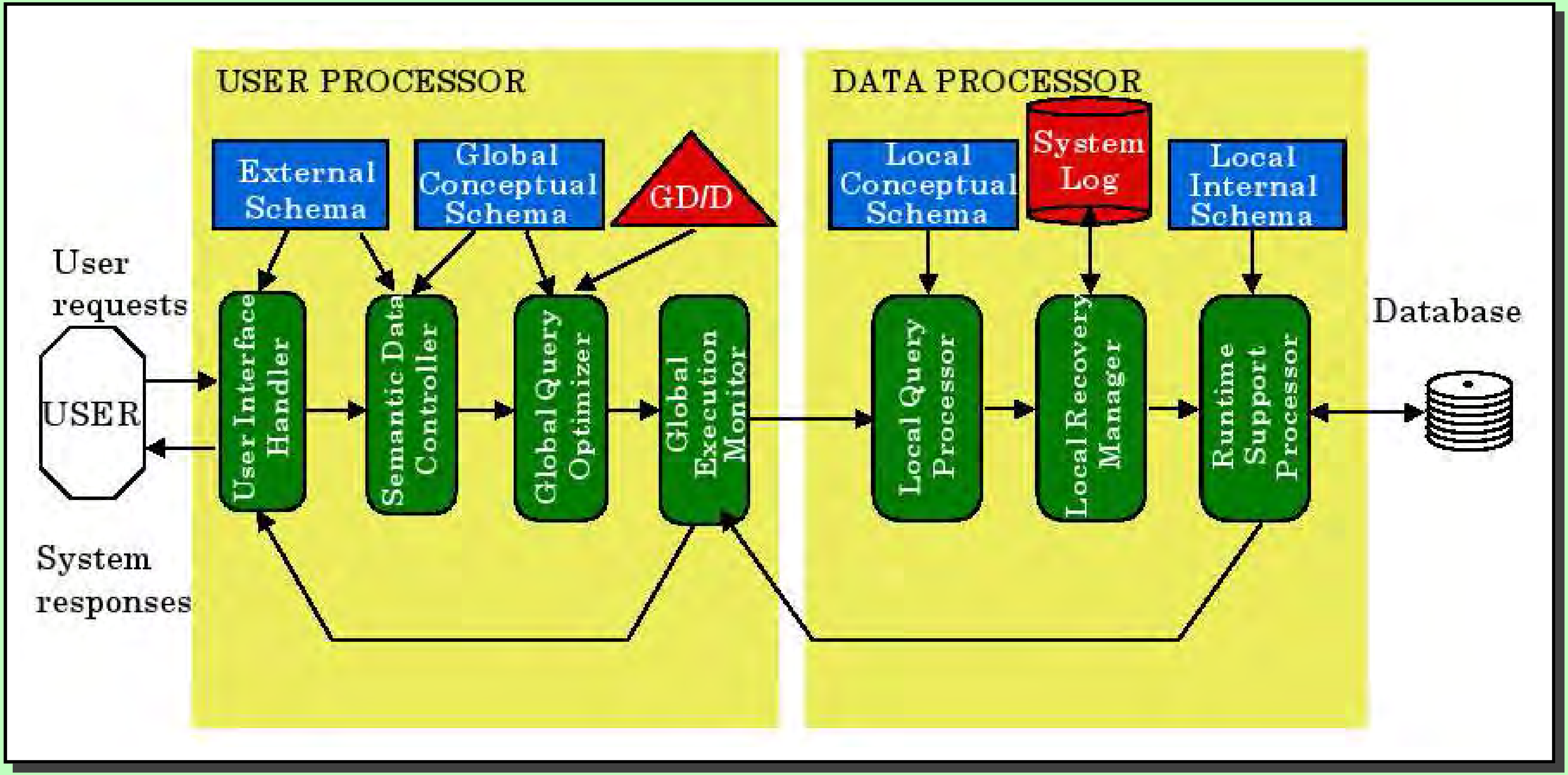


Components of Client-Server Architecture (Component-based) . . .

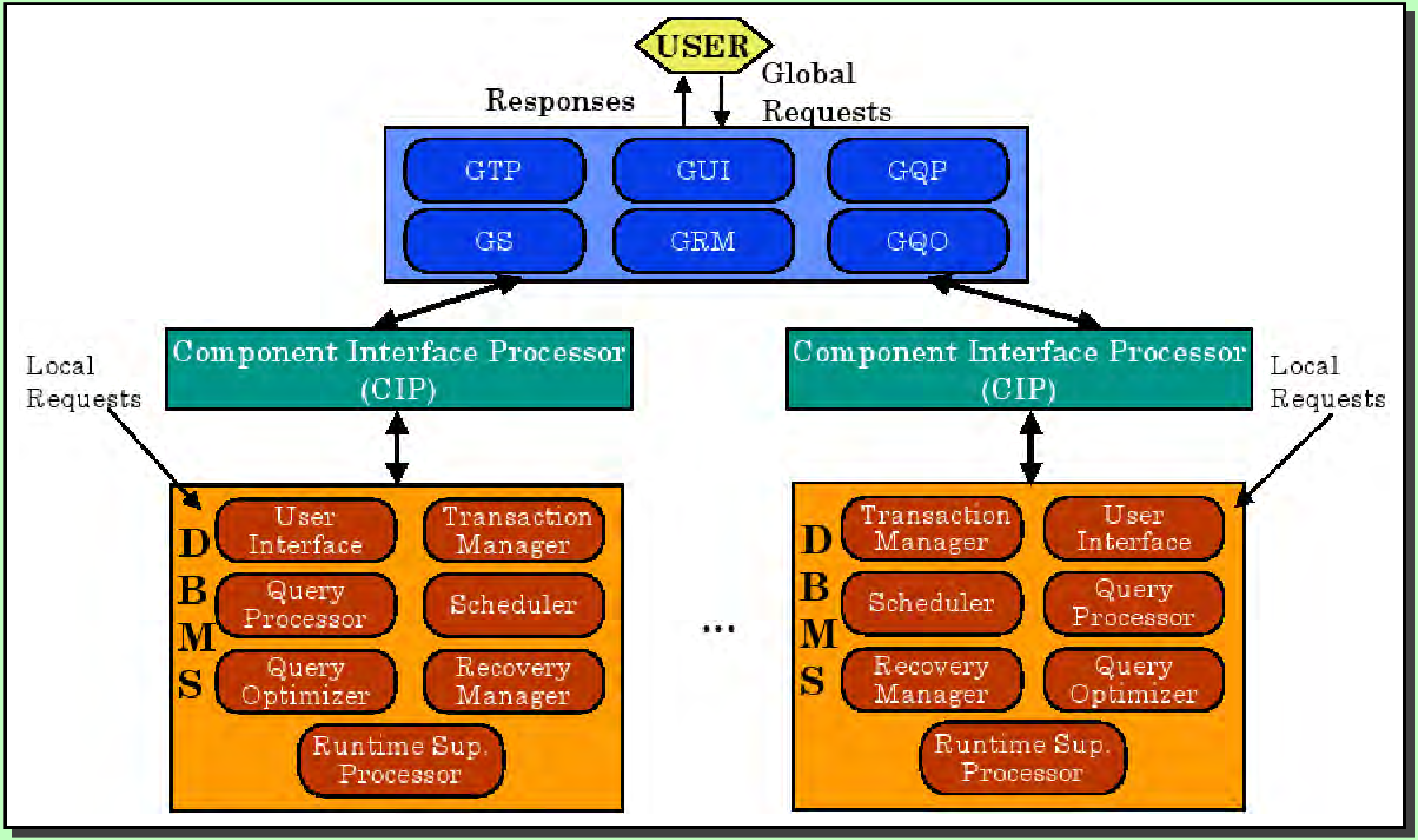
- Many servers, many clients



Components of Peer-to-Peer Architecture (Component-based)



Components of Multi-DBMS Architecture (Component-based)



Conclusion

- Architecture defines the structure of the system. There are three ways to define the architecture: based on components, functions, or data
- DDBMS might be based on identical components (homogeneous systems) or different components (heterogeneous systems)
- ANSI/SPARC architecture defines external, conceptual, and internal schemas
- There are three orthogonal implementation dimensions for DDBMS: level of distribution, autonomity, and heterogeinity
- Different architectures are discussed:
  - Client-Server Systems
  - Peer-to-Peer Systems
  - Multi-DBMS

Chapter 3: Distributed Database Design

- Design problem
- Design strategies(top-down, bottom-up)
- Fragmentation
- Allocation and replication of fragments, optimality, heuristics

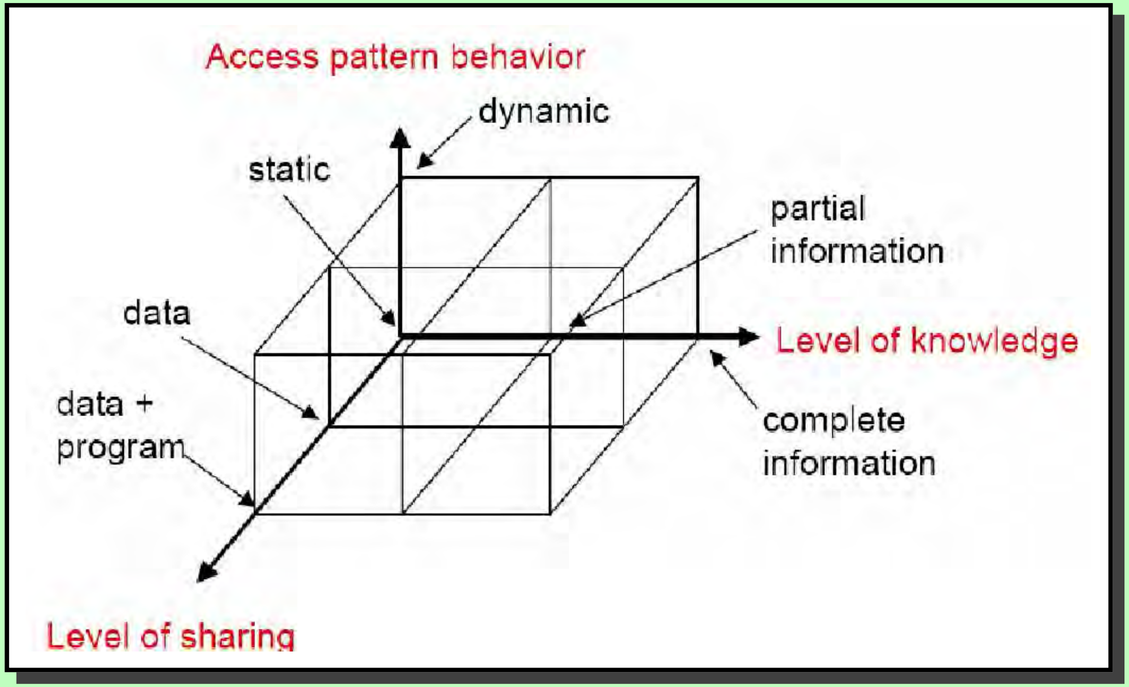
Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Design Problem

- **Design problem of distributed systems**: 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 DDBMS, the distribution of applications involves
  - Distribution of the DDBMS software
  - Distribution of applications that run on the database
- Distribution of applications will not be considered in the following; instead the distribution of data is studied.

Framework of Distribution

- Dimension for the analysis of distributed systems
  - Level of sharing: no sharing, data sharing, data + program sharing
  - Behavior of access patterns: static, dynamic
  - Level of knowledge on access pattern behavior: no information, partial information, complete information



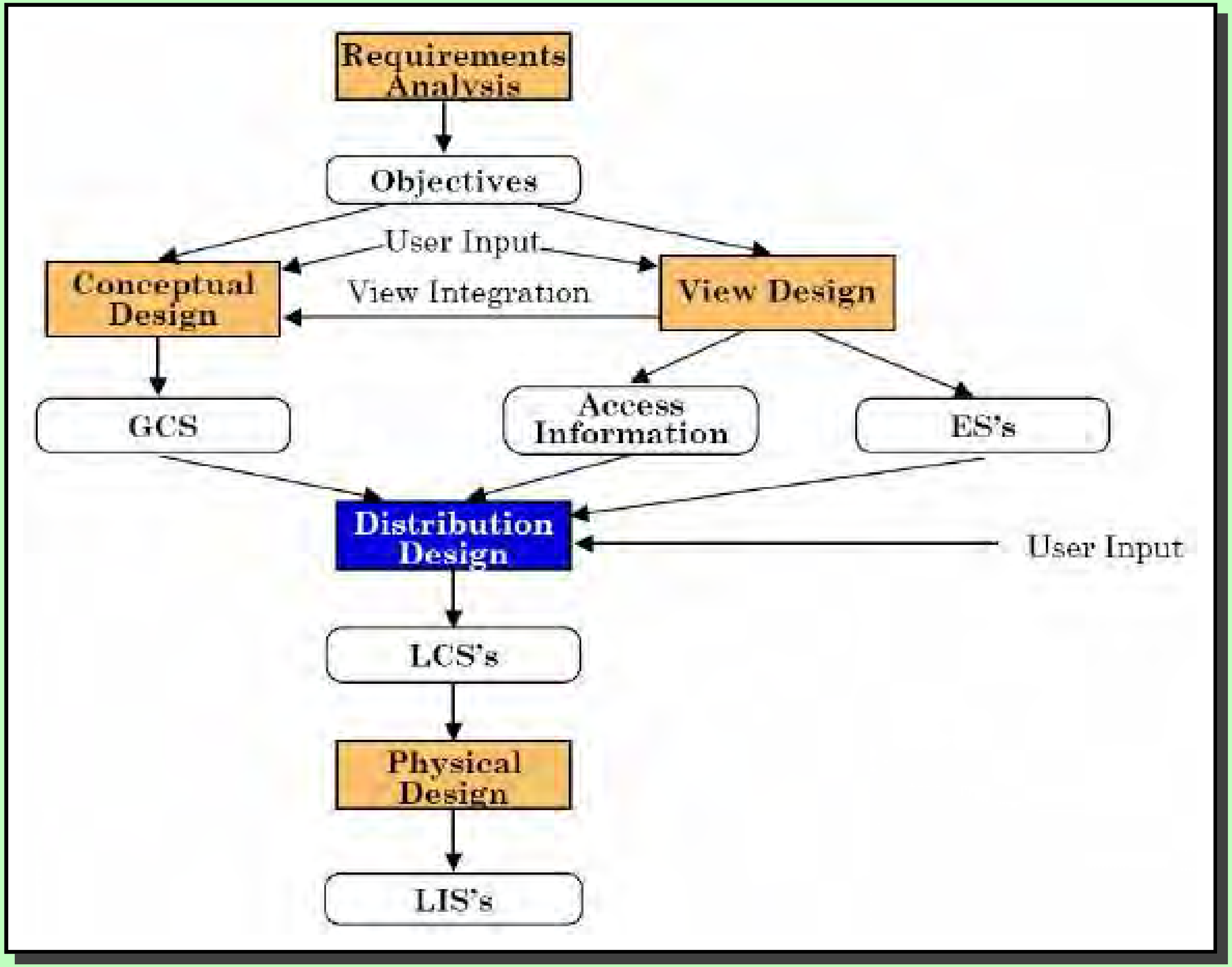
- Distributed database design should be considered within this general framework.

Design Strategies

- Top-down approach
  - Designing systems from scratch
  - Homogeneous systems
- Bottom-up approach
  - The databases already exist at a number of sites
  - The databases should be connected to solve common tasks

Design Strategies . . .

- **Top-down design strategy**

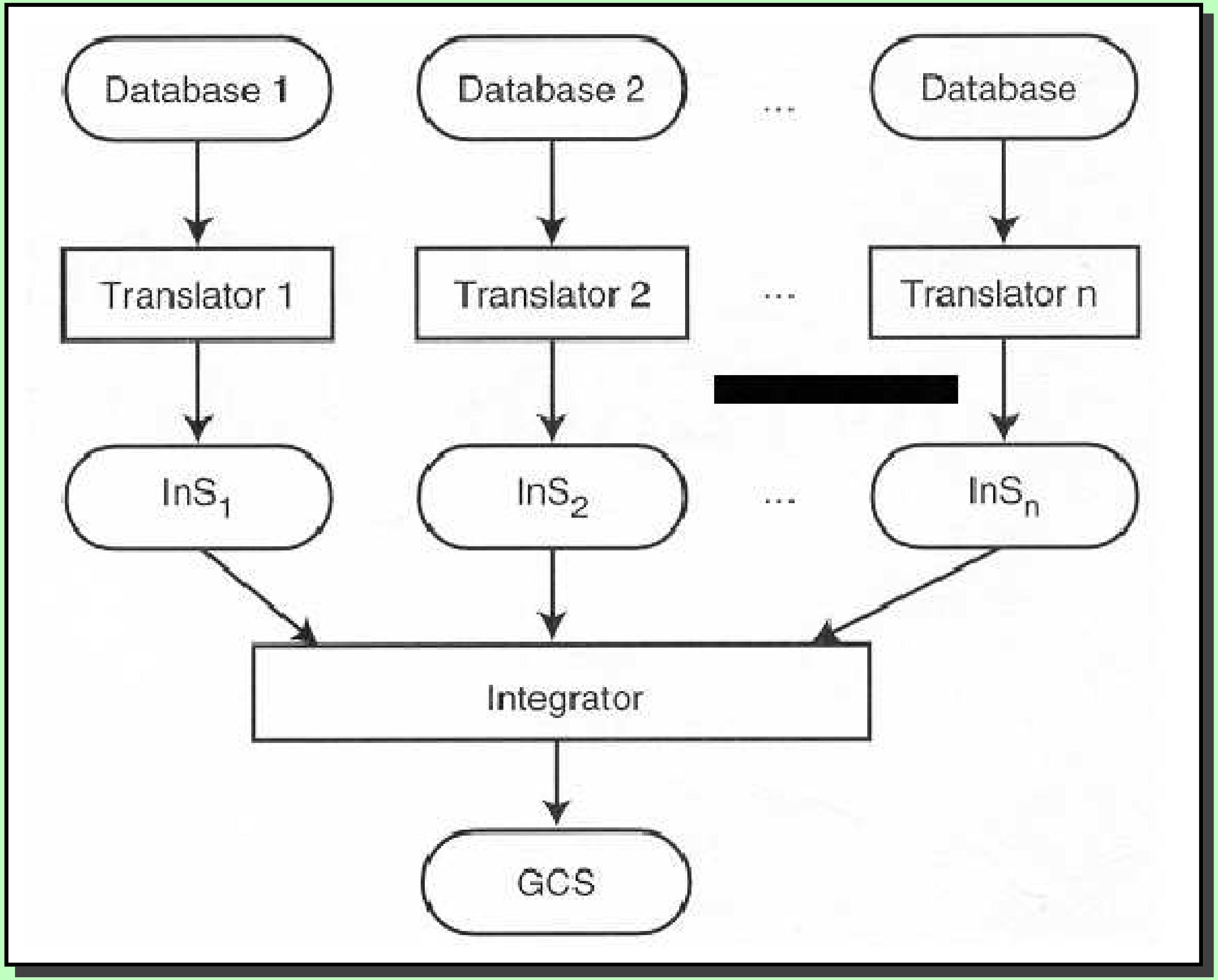


Design Strategies . . .

- **Distribution design** is the central part of the design in DDBMSs (the other tasks are similar to traditional databases)
  - **Objective**: Design the LCSs by distributing the entities (relations) over the sites
  - Two main aspects have to be designed carefully
    - **Fragmentation**
      - Relation may be divided into a number of sub-relations, which are distributed
    - **Allocation** and **replication**
      - Each fragment is stored at site with ”optimal” distribution
      - Copy of fragment may be maintained at several sites
- In this chapter we mainly concentrate on these two aspects
- Distribution design issues
  - Why fragment at all?
  - How to fragment?
  - How much to fragment?
  - How to test correctness?
  - How to allocate?

Design Strategies . . .

- **Bottom-up design strategy**



Fragmentation

- What is a reasonable unit of distribution? Relation or fragment of relation?
- **Relations** as unit of distribution:
  - If the relation is not replicated, we get a high volume of remote data accesses.
  - If the relation is replicated, we get unnecessary replications, which cause problems in executing updates and waste disk space
  - Might be an Ok solution, if queries need all the data in the relation and data stays at the only sites that uses the data
- **Fragments** of relationas as unit of distribution:
  - Application views are usually subsets of relations
  - Thus, locality of accesses of applications is defined on subsets of relations
  - Permits a number of transactions to execute concurrently, since they will access different portions of a relation
  - Parallel execution of a single query (intra-query concurrency)
  - However, semantic data control (especially integrity enforcement) is more difficult

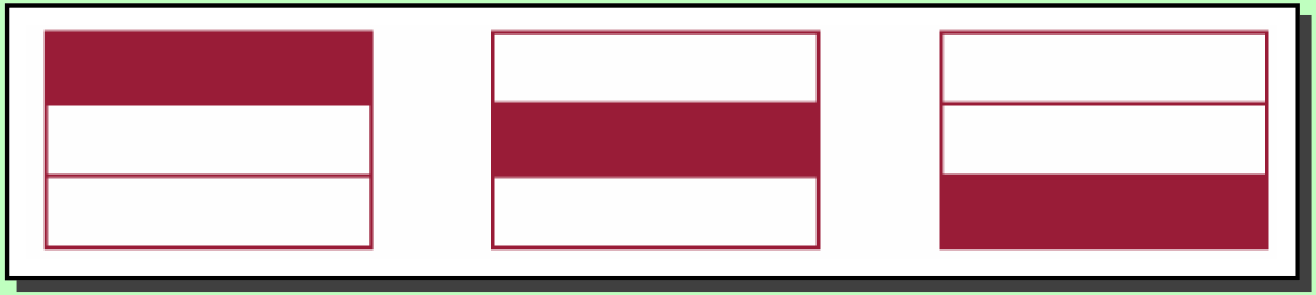
⇒ Fragments of relations are (usually) the appropriate unit of distribution.

Fragmentation . . .

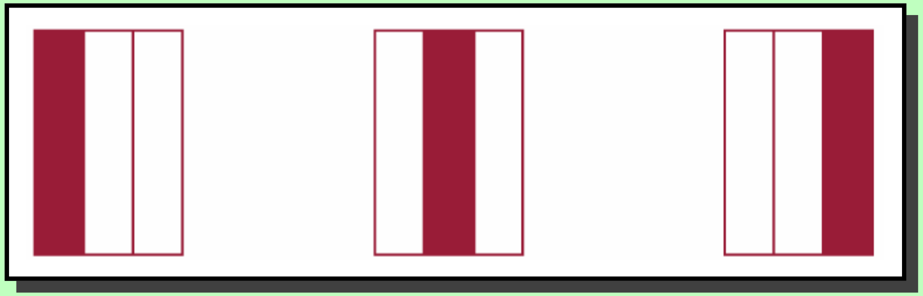
- Fragmentation aims to improve:
  - Reliability
  - Performance
  - Balanced storage capacity and costs
  - Communication costs
  - Security
- The following information is used to decide fragmentation:
  - Quantitative information: frequency of queries, site, where query is run, selectivity of the queries, etc.
  - Qualitative information: types of access of data, read/write, etc.

Fragmentation . . .

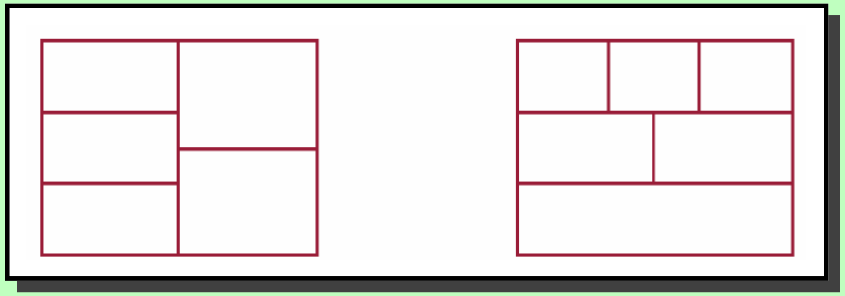
- Types of Fragmentation
  - Horizontal: partitions a relation along its tuples
  - Vertical: partitions a relation along its attributes
  - Mixed/hybrid: a combination of horizontal and vertical fragmentation



(a) Horizontal Fragmentation



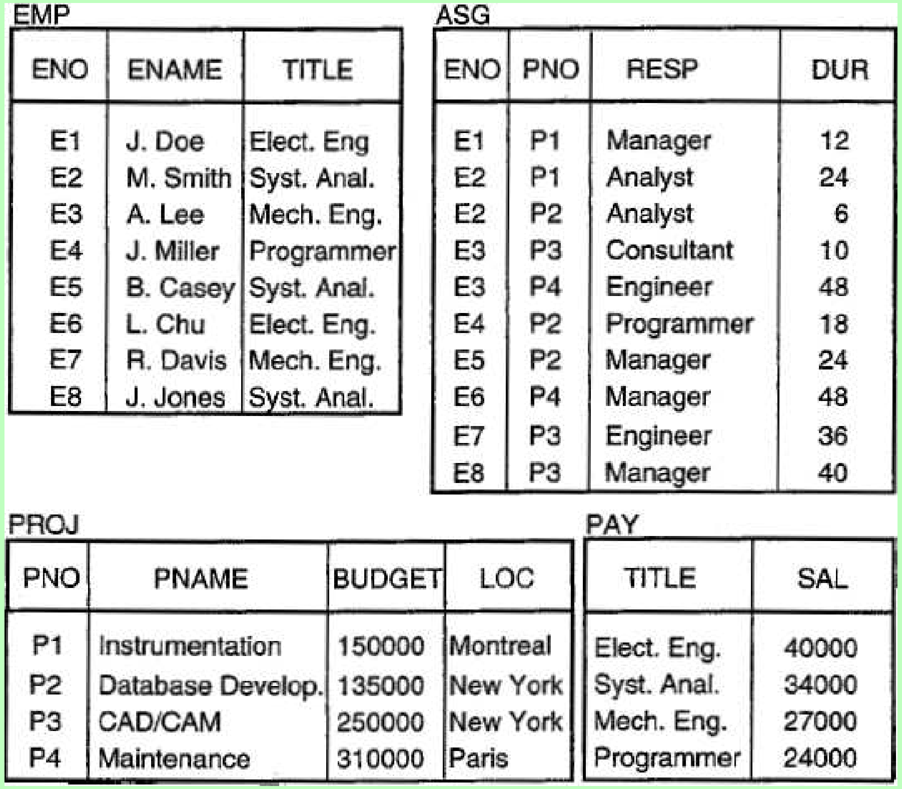
(b) Vertical Fragmentation



(c) Mixed Fragmentation

Fragmentation . . .

- **Exampe**



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
E8	P3	Manager	40

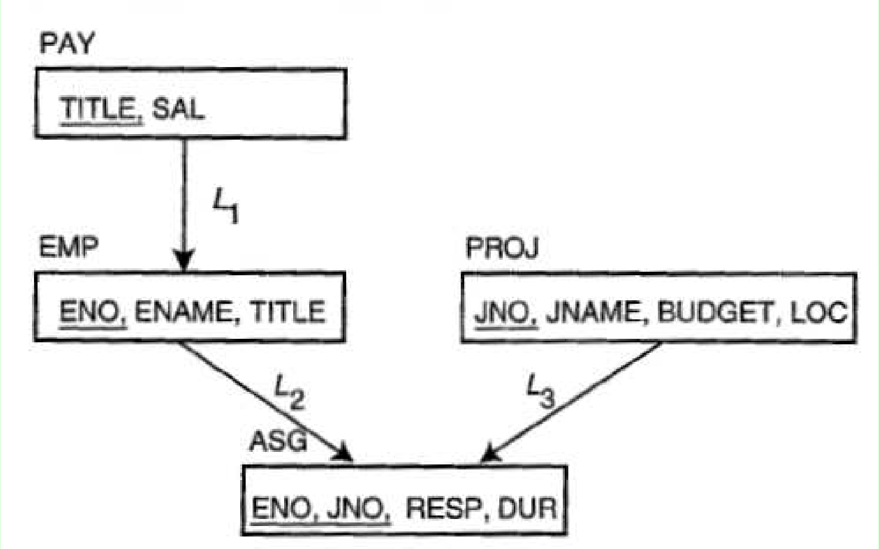
PROJ

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

PAY

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

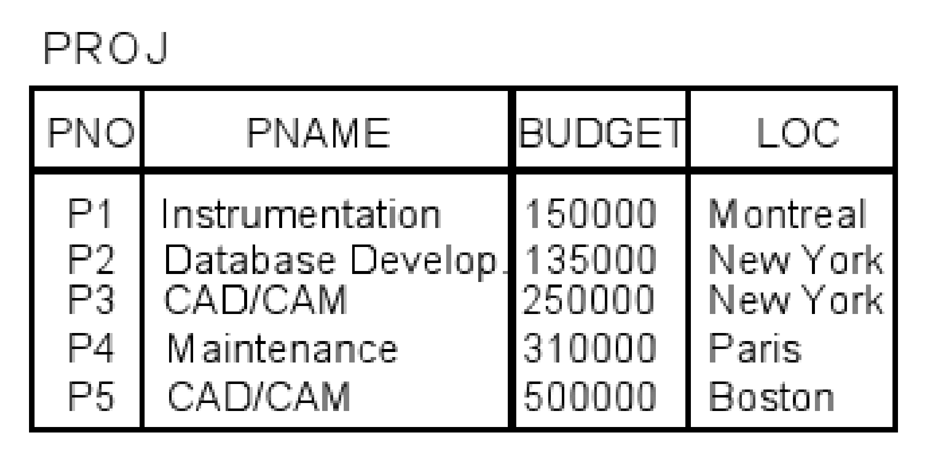
Data



E-R Diagram

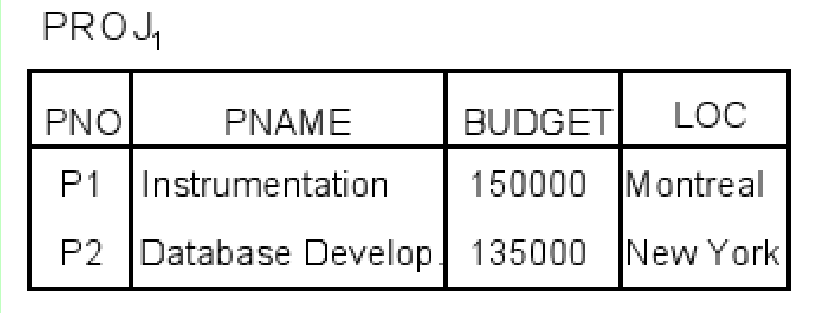
Fragmentation . . .

- **Example (contd.):** Horizontal fragmentation of PROJ relation
  - PROJ1: projects with budgets less than $200,000$
  - PROJ2: projects with budgets greater than or equal to $200,000$



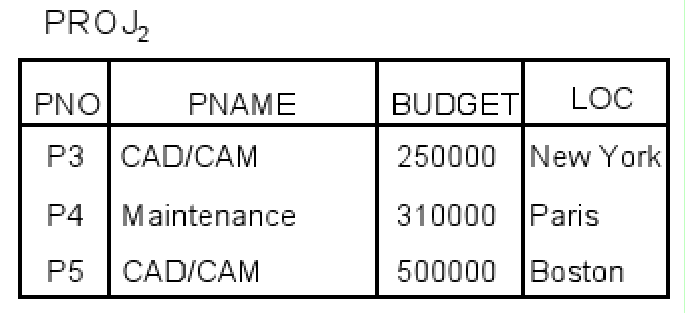
PROJ

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

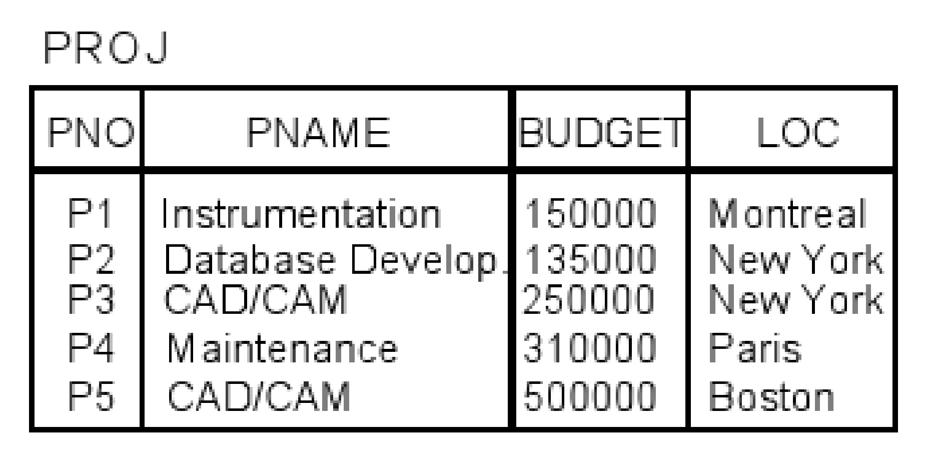


$PROJ_2$

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

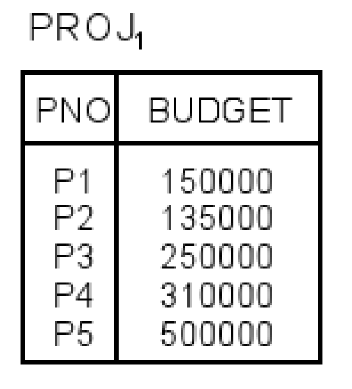
Fragmentation . . .

- **Example (contd.):** Vertical fragmentation of PROJ relation
  - PROJ1: information about project budgets
  - PROJ2: information about project names and locations



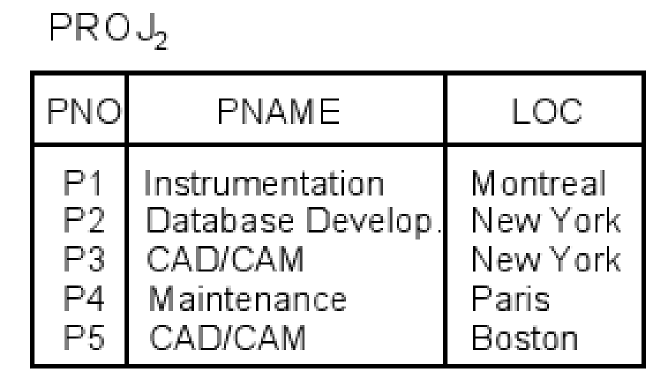
PROJ

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 Rules of Fragmentation

- **Completeness**
  - Decomposition of relation R into fragments $R_1, R_2, \ldots, R_n$ 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 ∇ that reconstructs R from its fragments, i.e., R=R1 ∇ ...∇ Rn
    - $\ast$ Union to combine horizontal fragments
    - $\ast$ Join to combine vertical fragments
- **Disjointness**
  - If relation R is decomposed into fragments $R_1, R_2, \ldots, R_n$ and data item d_i appears in fragment R_j, then d_i should not appear in any other fragment R_k, $k \neq j$ (exception: primary key attribute for vertical fragmentation)
    - $\ast$ For horizontal fragmentation, data item is a tuple
    - $\ast$ For vertical fragmentation, data item is an attribute

Horizontal Fragmentation

- **Intuition** behind horizontal fragmentation
  - Every site should hold all information that is used to query at the site
  - The information at the site should be fragmented so the queries of the site run faster
- Horizontal fragmentation is **defined as selection operation**, $\sigma \ (R)$
- **Example**:

$$\sigma_{\text{BUDGET}<200000}(PROJ)$$
$$\sigma_{\text{BUDGET}\geq 200000}(PROJ)$$

Horizontal Fragmentation . . .

- **Computing** horizontal fragmentation (idea)
  - Compute the **frequency** of the individual queries of the site $q_1, \ldots, q_Q$
  - Rewrite the queries of the site in the conjunctive normal form (disjunction of conjunctions); the conjunctions are called **minterms**.
  - Compute the **selectivity** of the minterms
  - Find the **minimal** and **complete** set of minterms (predicates)
    - $*$ The set of predicates is **complete** if and only if any two tuples in the same fragment are referenced with the same probability by any application
    - $*$ The set of predicates is **minimal** if and only if there is at least one query that accesses the fragment
  - There is an algorithm how to find these fragments algorithmically (the algorithm `CON_MIN` and `PHORIZONTAL` (pp 120-122) of the textbook of the course)

Horizontal Fragmentation . . .

- **Example:** Fragmentation of the $PROJ$ relation
  - Consider the following query: *Find the name and budget of projects given their PNO.*
  - The query is issued at all three sites
  - Fragmentation based on LOC, using the set of predicates/minterms $\{LOC =' Montreal', LOC =' NewYork', LOC =' Paris'\}$

$PROJ_1 = \sigma_{LOC=' Montreal'}(PROJ)$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal

$PROJ_2 = \sigma_{LOC=' NewYork'}(PROJ)$

PNO	PNAME	BUDGET	LOC
P2	Database Develop.	135000	New York
P3		250000	New York

$PROJ_3 = \qquad (\qquad)$

PNO	PNAME	BUDGET	LOC
P4	Maintenance	310000	Paris

- If access is only according to the location, the above set of predicates is complete
  - i.e., each tuple of each fragment $PROJ_i$ has the same probability of being accessed
- If there is a second query/application to access only those project tuples where the budget is less than $200000, the set of predicates is not complete.
  - $P2$ in $PROJ_2$ has higher probability to be accessed

Horizontal Fragmentation . . .

- **Example (contd.):**
  - Add $BUDGET \leq 200000$ and $BUDGET > 200000$ to the set of predicates to make it complete.
    $\Rightarrow \{LOC =' Montreal', LOC =' NewYork', LOC =' Paris',$
    $BUDGET \geq 200000, BUDGET < 200000\}$ is a complete set
  - Minterms to fragment the relation are given as follows:

$$(\qquad = \qquad\qquad\qquad) \quad (\qquad\qquad\qquad\qquad 200000)$$

$$(LOC = \; Montreal \;) \wedge (BUDGET > 200000)$$

$$(LOC =' NewYork') \wedge (BUDGET \leq 200000)$$

$$(LOC =' NewYork') \wedge (BUDGET > 200000)$$

$$(LOC =' Paris') \wedge (BUDGET \leq 200000)$$

$$(LOC =' Paris') \wedge (BUDGET > 200000)$$

Horizontal Fragmentation . . .

- **Example (contd.):** Now, $PROJ_2$ will be split in two fragments

$PROJ_1 = \sigma_{LOC='Montreal'}(PROJ)$

PNO	PNAME	BUDGET	LOC
P1	Instrumentation	150000	Montreal

$PROJ_2 = \sigma_{LOC='NY' \wedge BUDGET<200000}(PROJ)$

PNO	PNAME	BUDGET	LOC
P2	Database Develop.	135000	New York

$PROJ_3 = \qquad (\qquad)$

PNO	PNAME	BUDGET	LOC
P4	Maintenance	310000	Paris

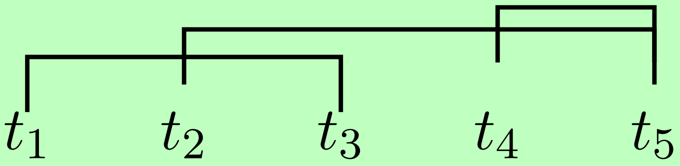
$= \qquad {}_{0000}(PROJ)$

PNO	PNAME	BUDGET	LOC
P3	CAD/CAM	250000	New York

  - $PROJ_1$ and $PROJ_2$ would have been split in a similar way if tuples with budgets smaller and greater than 200.000 would be stored

Horizontal Fragmentation . . .

- In most cases intuition can be used to build horizontal partitions. Let $\{t_1, t_2, t_3\}$, $\{t_4, t_5\}$, and $\{t_2, t_3, t_4, t_5\}$ be query results. Then tuples would be fragmented in the following way:



Vertical Fragmentation

- **Objective** of vertical fragmentation is to partition a relation into a set of smaller relations so that many of the applications will run on only one fragment.
- Vertical fragmentation of a relation R produces fragments $R_1, R_2, \ldots$, each of which contains a **subset of** R's **attributes**.
- Vertical fragmentation is defined using the **projection operation** of the relational algebra:
- **Example**:

$$PROJ_1 = \Pi_{PNO,BUDGET}(PROJ)$$
$$PROJ_2 = \Pi_{PNO,PNAME,LOC}(PROJ)$$

- Vertical fragmentation has also been studied for (centralized) DBMS
  - Smaller relations, and hence less page accesses
  - e.g., MONET system

Vertical Fragmentation . . .

- Vertical fragmentation is **inherently more complicated** than horizontal fragmentation
  - In horizontal partitioning: for n simple predicates, the number of possible minterms is 2^n; some of them can be ruled out by existing implications/constraints.
  - In vertical partitioning: for m non-primary key attributes, the number of possible fragments is equal to $B(m)$ (= the mth Bell number), i.e., the number of partitions of a set with m members.
    - For large numbers, $B(m) \approx m^m$ (e.g., $B(15) = 10^9$)
- Optimal solutions are not feasible, and heuristics need to be applied.

Vertical Fragmentation . . .

- Two types of heuristics for vertical fragmentation exist:
  - **Grouping**: assign each attribute to one fragment, and at each step, join some of the fragments until some criteria is satisfied.
    - Bottom-up approach
  - **Splitting**: starts with a relation and decides on beneficial partitionings based on the access behaviour of applications to the attributes.
    - Top-down approach
    - Results in non-overlapping fragments
    - “Optimal” solution is probably closer to the full relation than to a set of small relations with only one attribute
    - Only vertical fragmentation is considered here

Vertical Fragmentation . . .

- **Application information:** The major information required as input for vertical fragmentation is related to applications
  - Since vertical fragmentation places in one fragment those attributes usually accessed together, there is a need for some measure that would define more precidely the notion of "togertherness",i.e., how closely related the attributes are.
  - This information is obtained from queries and collected in the *Attribute Usage Matrix* and *Attribute Affinity Matrix*.

Vertical Fragmentation . . .

- Given are the user queries/applications $Q = (q_1, \ldots, q_q)$ that will run on relation $R(A_1, \ldots, A_n)$
- **Attribute Usage Matrix**: Denotes which query uses which attribute:

$$use(q_i, A_j) = \begin{cases} 1 & \text{iff } q_i \text{ uses } A_j \\ 0 & \text{otherwise} \end{cases}$$

  - The $use(q_i, \bullet)$ vectors for each application are easy to define if the designer knows the applications that willl run on the DB (consider also the 80-20 rule)

Vertical Fragmentation . . .

- **Example**: Consider the following relation:

$$PROJ(PNO, PNAME, BUDGET, LOC)$$

and the following queries:

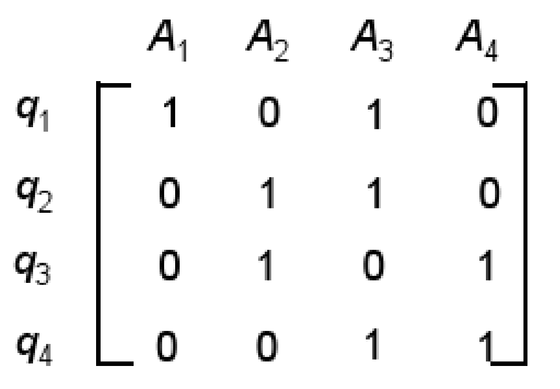
$q_1 =$ SELECT BUDGET FROM PROJ WHERE PNO=Value

$q_2 =$ SELECT PNAME,BUDGET FROM PROJ

$q_3 =$ SELECT PNAME FROM PROJ WHERE LOC=Value

$q =$ SELECT SUM(BUDGET) FROM PROJ WHERE LOC =Value

- Lets abbreviate $A_1 = PNO, A_2 = PNAME, A_3 = BUDGET, A_4 = LOC$
- Attribute Usage Matrix



$$\begin{array}{c|cccc} & A_1 & A_2 & A_3 & A_4 \\ \hline q_1 & 1 & 0 & 1 & 0 \\ q_2 & 0 & 1 & 1 & 0 \\ q_3 & 0 & 1 & 0 & 1 \\ q_4 & 0 & 0 & 1 & 1 \end{array}$$

Vertical Fragmentation . . .

- **Attribute Affinity Matrix**: Denotes the frequency of two attributes A_i and A_j with respect to a set of queries $Q = (q_1, \dots, q_n)$:

$$\mathit{aff}(A_i, A_j) = \quad \sum \quad \left(\sum \; \mathit{ref}_l(q_k) \mathit{acc}_l(q_k) \right)$$

where

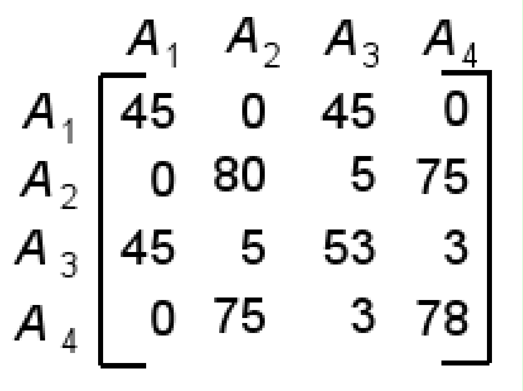
  - $\mathit{ref}_l(q_k)$ is the cost (= number of accesses to (A_i, A_j)) of query q_K at site l
  - $\mathit{acc}_l(q_k)$ is the frequency of query q_k at site l

Vertical Fragmentation . . .

- **Example (contd.):** Let the cost of each query be $ref_l(q_k) = 1$, and the frequency $acc_l(q_k)$ of the queries be as follows:

$Site1$	$Site2$	$Site3$
$acc_1(q_1) = 15$	$acc_2(q_1) = 20$	$acc_3(q_1) = 10$
$acc_1(q_2) = 5$	$acc_2(q_2) = 0$	$acc_3(q_2) = 0$
$acc_1(q_3) = 25$	$acc_2(q_3) = 25$	$acc_3(q_3) = 25$
$acc_1(q_4) = 3$	$acc_2(q_4) = 0$	$acc_3(q_4) = 0$

- Attribute affinity matrix $\mathit{aff}(A\ \ , A\ \ ) =$



$$\begin{array}{c|cccc|} & A_1 & A_2 & A_3 & A_4 \\ A_1 & 45 & 0 & 45 & 0 \\ A_2 & 0 & 80 & 5 & 75 \\ A_3 & 45 & 5 & 53 & 3 \\ A_4 & 0 & 75 & 3 & 78 \end{array}$$

  - e.g., $\mathit{aff}(A_1, A_3) = \sum_{k=1}^{1} \sum_{l=1}^{3} acc_l(q_k) = acc_1(q_1) + acc_2(q_1) + acc_3(q_1) = 45$ (q_1 is the only query to access both A_1 and A_3)

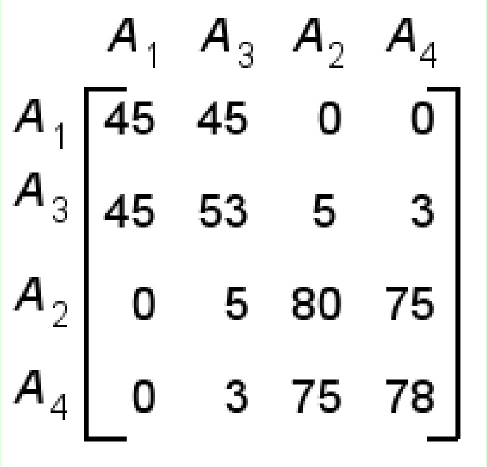
Vertical Fragmentation . . .

- Take the attribute affinity matrix (AA) and reorganize the attribute orders to form clusters where the attributes in each cluster demonstrate high affinity to one another.
- **Bond energy algorithm (BEA)** has been suggested to be useful for that purpose for several reasons:
  - It is designed specifically to determine groups of similar items as opposed to a linear ordering of the items.
  - The final groupings are insensitive to the order in which items are presented.
  - The computation time is reasonable ($O(n^2)$, where n is the number of attributes)
- **BEA**:
  - Input: AA matrix
  - Output: Clustered AA matrix (CA)
  - Permutation is done in such a way to maximize the following **global affinity mesaure** (affinity of A_i and A_j with their neighbors):

$$AM = \sum_{i=1}^{n} \sum_{j=1}^{n} \textit{aff}(A_i, A_j)[\textit{aff}(A_i, A_{j-1}) + \textit{aff}(A_i, A_{j+1}) + \\ \textit{aff}(A_{i-1}, A_j) + \textit{aff}(A_{i+1}, A_j)]$$

Vertical Fragmentation . . .

- **Example (contd.)**: Attribute Affinity Matrix CA after running the BEA



$$\begin{array}{c|cccc} & A_1 & A_3 & A_2 & A_4 \\ \hline A_1 & 45 & 45 & 0 & 0 \\ A_3 & 45 & 53 & 5 & 3 \\ A_2 & 0 & 5 & 80 & 75 \\ A_4 & 0 & 3 & 75 & 78 \end{array}$$

  - Elements with similar values are grouped together, and two clusters can be identified
  - An additional partitioning algorithm is needed to identify the clusters in CA
    - $*$ Usually more clusters and more than one candidate partitioning, thus additional steps are needed to select the best clustering.
  - The resulting fragmentation after partitioning (PNO is added in $PROJ_2$ explicilty as key):

$$PROJ_1 = \{PNO, BUDGET\}$$
$$PROJ_2 = \{PNO, PNAME, LOC\}$$

Correctness of Vertical Fragmentation

- Relation R is decomposed into fragments $R_1, R_2, \ldots, R_n$
  - e.g., $PROJ = \{PNO, BUDGET, PNAME, LOC\}$ into $PROJ_1 = \{PNO, BUDGET\}$ and $PROJ_2 = \{PNO, PNAME, LOC\}$
- **Completeness**
  - Guaranteed by the partitioning algortihm, which assigns each attribute in A to one partition
- **Reconstruction**
  - Join to reconstruct vertical fragments
  - $R = R_1 \bowtie \cdots \bowtie R_n = PROJ_1 \bowtie PROJ_2$
- **Disjointness**
  - Attributes have to be disjoint in VF. Two cases are distinguished:
    * If tuple IDs are used, the fragments are really disjoint
    * Otherwise, key attributes are replicated automatically by the system
    * e.g., PNO in the above example

Mixed Fragmentation

- In most cases simple horizontal or vertical fragmentation of a DB schema will not be sufficient to satisfy the requirements of the applications.
- **Mixed fragmentation (hybrid fragmentation)**: Consists of a horizontal fragment followed by a vertical fragmentation, or a vertical fragmentation followed by a horizontal fragmentation
- Fragmentation is defined using the selection and projection operations of relational algebra:

$$\sigma_p(\Pi_{A_1,\ldots,A_n}(R))$$
$$\Pi_{A_1,\ldots,A_n}(\sigma_p(R))$$

Replication and Allocation

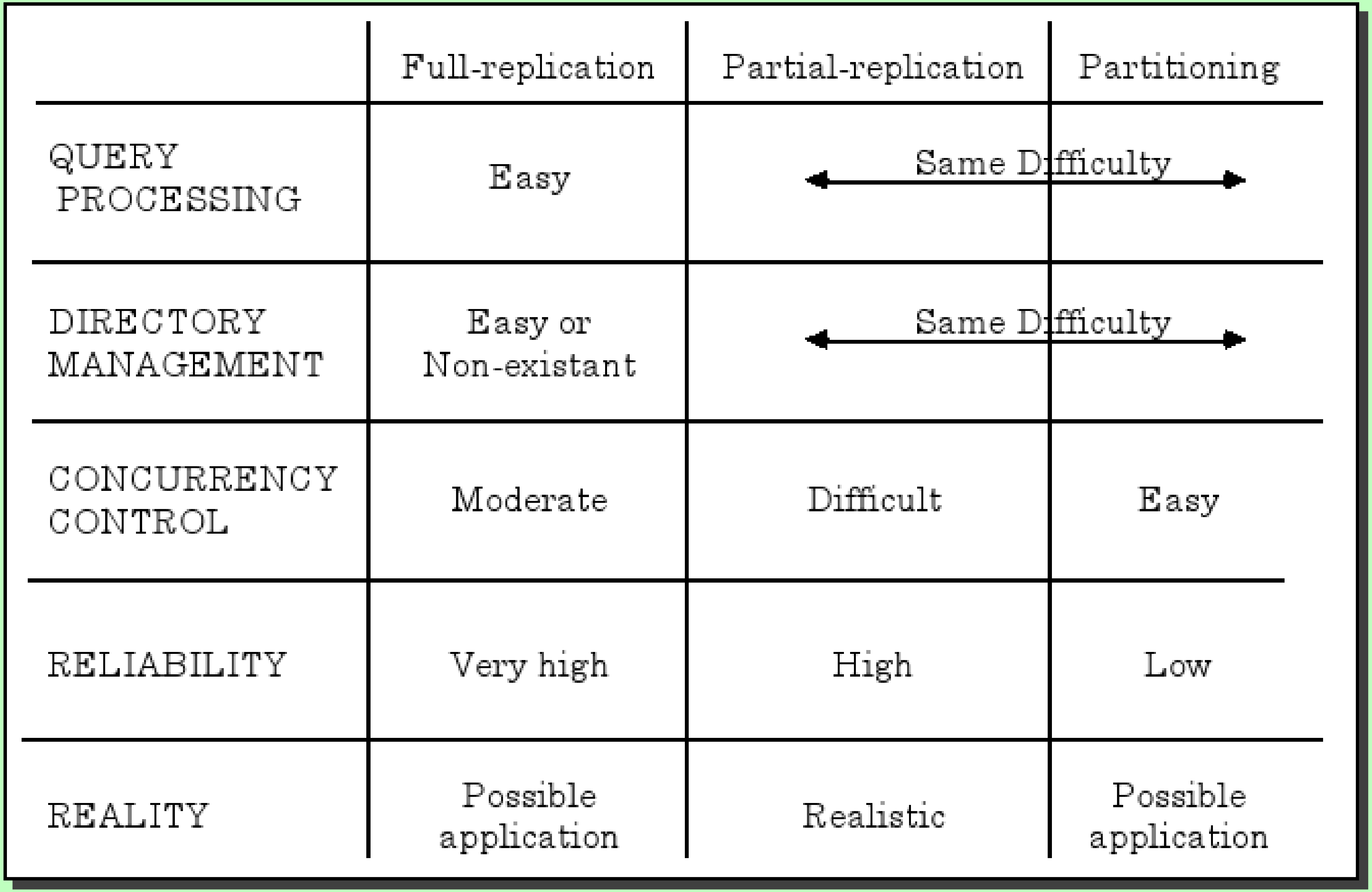
- **Replication**: Which fragements shall be stored as multiple copies?
  - Complete Replication
    - Complete copy of the database is maintained in each site
  - Selective Replication
    - Selected fragments are replicated in some sites
- **Allocation**: On which sites to store the various fragments?
  - Centralized
    - Consists of a single DB and DBMS stored at one site with users distributed across the network
  - Partitioned
    - Database is partitioned into disjoint fragments, each fragment assigned to one site

Replication . . .

- Replicated DB
  - **fully replicated**: each fragment at each site
  - **partially replicated**: each fragment at some of the sites
- Non-replicated DB (= partitioned DB)
  - **partitioned**: each fragment resides at only one site
- Rule of thumb:
  - If $\frac{\text{read only queries}}{\text{update queries}} \geq 1$, then replication is advantageous, otherwise replication may cause problems

Replication . . .

- 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

Fragment Allocation

- **Fragment allocation problem**
  - Given are:
    – fragments $F = \{F_1, F_2, ..., F_n\}$
    – network sites $S = \{S_1, S_2, ..., S_m\}$
    – and applications $Q = \{q_1, q_2, ..., q_l\}$
  - Find: the ”optimal” distribution of F to S

- **Optimality**
  - Minimal cost
    - ∗ Communication + storage + processing (read and update)
    - ∗ Cost in terms of time (usually)
  - Performance
    - ∗ Response time and/or throughput
  - Constraints
    - ∗ Per site constraints (storage and processing)

Fragment Allocation . . .

- **Required information**
  - Database Information
    - selectivity of fragments
    - size of a fragment
  - Application Information
    - RR_{ij}: number of read accesses of a query q_i to a fragment F_j
    - UR : number of update accesses of query q to a fragment F
    - u_{ij}: a matrix indicating which queries updates which fragments,
    - r_{ij}: a similar matrix for retrievals
    - originating site of each query
  - Site Information
    - USC_k: unit cost of storing data at a site S_k
    - LPC_k: cost of processing one unit of data at a site S_k
  - Network Information
    - communication cost/frame between two sites
    - frame size

Fragment Allocation ...

- We present an **allocation model** which attempts to
  - minimize the total cost of processing and storage
  - meet certain response time restrictions
- General Form:

$$\min(\text{Total Cost})$$

  - subject to
    - response time constraint
    - storage constraint
    - processing constraint

- Functions for the total cost and the constraints are presented in the next slides.
- Decision variable x_{ij}

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

Fragment Allocation . . .

- The **total cost function** has two components: storage and query processing.

$$TOC = \sum_{S_k \in S} \sum_{F_j \in F} STC_{jk} + \sum_{q_i \in Q} QPC_i$$

  - **Storage cost** of fragment F_j at site S_k:

    where USC_k is the unit storage cost at site k

  - **Query processing cost** for a query q_i is composed of two components:
    - $*$ composed of processing cost (PC) and transmission cost (TC)

$$QPC_i = PC_i + TC_i$$

Fragment Allocation . . .

- **Processing cost** is a sum of three components:
  - access cost (AC), integrity contraint cost (IE), concurency control cost (CC)

$$PC_i = AC_i + IE_i + CC_i$$

  - **Access cost**:

$$AC_i = \sum_{s_k \in S} \sum_{F_j \in F} (UR_{ij} + RR_{ij}) * x_{ij} * LPC_k$$

    where LPC_k is the unit process cost at site k
  - **Integrity and concurrency costs**:
    - Can be similarly computed, though depends on the specific constraints

- **Note:** AC_i assumes that processing a query involves decomposing it into a set of subqueries, each of which works on a fragment, ...,
  - This is a very simplistic model
  - Does not take into consideration different query costs depending on the operator or different algorithms that are applied

Fragment Allocation . . .

- The **transmission cost** is composed of two components:
  - Cost of processing updates (TCU) and cost of processing retrievals (TCR)

$$TC_i = TCU_i + TCR_i$$

  - **Cost of updates**:
    - ∗ Inform all the sites that have replicas + a short confirmation message back

$$TCU_i = \sum_{S_k \in S} \sum_{F_j \in F} u \qquad (\text{update message cost} + \text{acknowledgment cost})$$

  - **Retrieval cost**:
    - ∗ Send retrieval request to all sites that have a copy of fragments that are needed + sending back the results from these sites to the originating site.

$$TCR_i = \sum_{F_j \in F} \min_{S_k \in S} * (\text{cost of retrieval request} + \text{cost of sending back the result})$$

Fragment Allocation . . .

- Modeling the **constraints**
  - **Response time** constraint for a query q_i

    $$\text{execution time of } q_i \leq \text{ max. allowable response time for } q_i$$

  - **Storage** constraints for a site S_k

    $$\sum_{F_j \in F} \text{storage requirement of } F \text{ at } S \quad \text{storage capacity of } S_k$$

  - **Processing** constraints for a site S_k

    $$\sum_{q_i \in Q} \text{processing load of } q_i \text{ at site } S_k \leq \text{ processing capacity of} S_k$$

Fragment Allocation . . .

- **Solution Methods**
  - The complexity of this allocation model/problem is NP-complete
  - Correspondence between the allocation problem and similar problems in other areas
    - ∗ Plant location problem in operations research
    - ∗ Knapsack problem
    - ∗ Network flow problem
  - Hence, solutions from these areas can be re-used
  - Use different heuristics to reduce the search space
    - ∗ Assume that all candidate partitionings have been determined together with their associated costs and benefits in terms of query processing.
      - · The problem is then reduced to find the optimal partitioning and placement for each relation
    - ∗ Ignore replication at the first step and find an optimal non-replicated solution
      - · Replication is then handeled in a second step on top of the previous non-replicated solution.

Conclusion

- Distributed design decides on the placement of (parts of the) data and programs across the sites of a computer network
- On the abstract level there are two patterns: Top-down and Bottom-up
- On the detail level design answers two key questions: fragmentation and allocation/replication of data
  - Horizontal fragmentation is defined via the selection operation $\sigma_p(R)$
    - Rewrites the queries of each site in the conjunctive normal form and finds a minimal and complete set of conjunctions to determine fragmentation
  - Vertical fragmentation via the projection operation $\pi_A(R)$
    - Computes the attribute affinity matrix and groups “similar” attributes together
  - Mixed fragmentation is a combination of both approaches
- Allocation/Replication of data
  - Type of replication: no replication, partial replication, full replication
  - Optimal allocation/replication modelled as a cost function under a set of constraints
  - The complexity of the problem is NP-complete
  - Use of different heuristics to reduce the complexity

Chapter 4: Semantic Data Control

- View management
- Security control
- Integrity control

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of the last year course.

Semantic Data Control

- Semantic data control typically includes view management, security control, and semantic integrity control.
- Informally, these functions must ensure that **authorized** users perform **correct** operations on the database, contributing to the maintenance of database **integrity**.
- In RDBMS semantic data control can be achieved in a uniform way
  - views, security constraints, and semantic integrity constraints can be defined as rules that the system automatically enforces

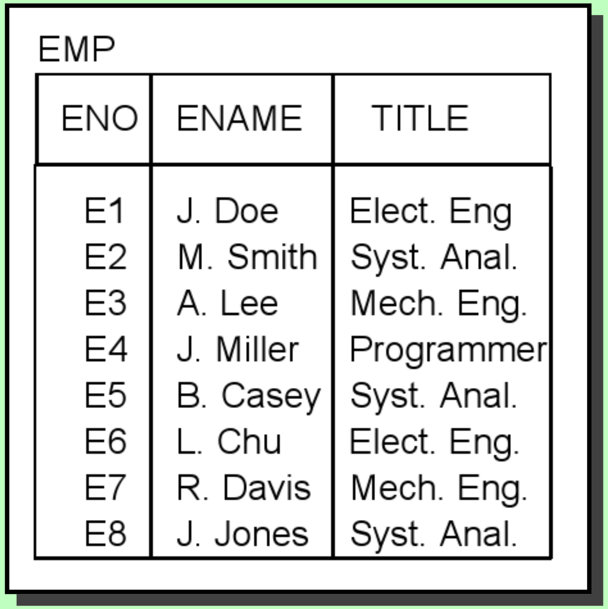
View Management

- Views enable full logical data independence
- Views are virtual relations that are defined as the result of a query on base relations
- Views are typically not materialized
  - Can be considered a dynamic window that reflects all relevant updates to the database
- Views are very useful for ensuring data security in a simple way
  - By selecting a subset of the database, views **hide** some data
  - Users cannot see the hidden data

View Management in Centralized Databases

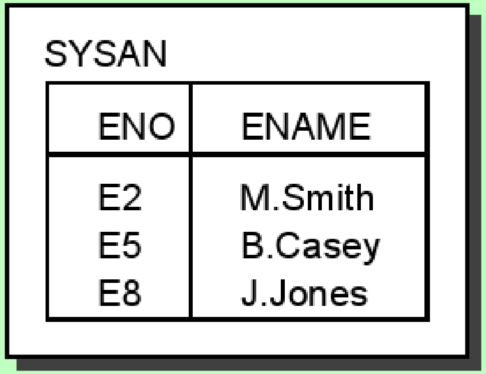
- A view is a relation that is derived from a base relation via a query.
- It can involve selection, projection, aggregate functions, etc.
- **Example:** The view of system analysts derived from relation EMP

```
CREATE VIEW SYSAN(ENO,ENAME) AS
SELECT ENO,ENAME
FROM EMP
WHERE TITLE="Syst. Anal."
```



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.



SYSAN

ENO	ENAME
E2	M.Smith
E5	B.Casey
E8	J.Jones

View Management in Centralized Databases . . .

- Queries expressed on views are translated into queries expressed on base relations
- **Example**: “Find the names of all the system analysts with their project number and responsibility?”
  - Involves the view SYSAN and the relation ASG(ENO,PNO,RESP,DUR)

```
SELECT ENAME, PNO, RESP
FROM SYSAN, ASG
WHERE SYSN.ENO = ASG.ENO
```

is translated into

```
SELECT ENAME,PNO,RESP
FROM EMP, ASG
WHERE EMP.ENO = ASG.ENO
AND TITLE = "Syst. Anal."
```

ENAME	PNO	RESP
M.Smith	P1	Analyst
M.Smith	P2	Analyst
B.Casey	P3	Manager
J.Jones	P4	Manager

- Automatic query modification is required, i.e., ANDing query qualification with view qualification

View Management in Centralized Databases . . .

- All views can be queried as base relations, but not all view can be updated as such
  - Updates through views can be handled automatically only if they can be propagated correctly to the base relations
  - We classify views as updatable or not-updatable

- **Updatable view:** The updates to the view *can* be propagated to the base relations without ambiguity.

```
CREATE VIEW SYSAN(ENO,ENAME) AS
SELECT ENO,ENAME
FROM EMP
WHERE TITLE="Syst. Anal."
```

  - e.g, insertion of tuple (201,Smith) can be mapped into the insertion of a new employee (201, Smith, “Syst. Anal.”)
  - If attributes other than TITLE were hidden by the view, they would be assigned the value *null*

View Management in Centralized Databases . . .

- **Non-updatable view:** The updates to the view *cannot* be propagated to the base relations without ambiguity.

```
CREATE VIEW EG(ENAME,RESP) AS
SELECT ENAME,RESP
FROM EMP, ASG
WHERE EMP.ENO=ASG.ENO
```

  - e.g, deletion of (Smith, ”Syst. Anal.”) is ambiguous, i.e., since deletion of “Smith” in EMP and deletion of “Syst. Anal.” in ASG are both meaningful, but the system cannot decide.

- Current systems are very restrictive about supportin gupdates through views
  - Views can be updated only if they are derived from a single relation by selection and projection
  - However, it is theoretically possible to automatically support updates of a larger class of views, e.g., joins

View Management in Distributed Databases

- Definition of views in DDBMS is similar as in centralized DBMS
  - However, a view in a DDBMS may be derived from fragmented relations stored at different sites
- Views are conceptually the same as the base relations, therefore we store them in the (possibly) distributed directory/catalogue
  - Thus, views might be centralized at one site, partially replicated, fully replicated
  - Queries on views are translated into queries on base relations, yielding distributed queries due to possible fragmentation of data
- Views derived from distributed relations may be costly to evaluate
  - Optimizations are important, e.g., snapshots
  - A snapshot is a static view
    - does not reflect the updates to the base relations
    - managed as temporary relations: the only access path is sequential scan
    - typically used when selectivity is small (no indices can be used efficiently)
    - is subject to periodic recalculation

Data Security

- **Data security** protects data against unauthorized acces and has two aspects:
  - Data protection
  - Authorization control

Data Protection

- **Data protection** prevents unauthorized users from understanding the physical content of data.
- Well established standards exist
  - Data encryption standard
  - Public-key encryption schemes

Authorization Control

- **Authorization control** must guarantee that only authorized users perform operations they are allowed to perform on the database.
- Three actors are involved in authorization
  - **users**, who trigger the execution of application programms
  - **operations**, which are embedded in applications programs
  - **database objects**, on which the operations are performed
- Authorization control can be viewed as a triple *(user, operation type, object)* which specifies that the user has the right to perform an operation of operation type on an object.
- Authentication of (groups of) users is typically done by username and password
- Authorization control in (D)DBMS is more complicated as in operating systems
  - In a file system: data objects are files
  - In a DBMS: Data objects are views, (fragments of) relations, tuples, attributes

Authorization Control . . .

- Grand and revoke statements are used to authorize triplets (user, operation, data object)
  - **GRANT** `<operations>` **ON** `<object>` **TO** `<users>`
  - **REVOKE** `<operations>` **ON** `<object>` **TO** `<users>`
- Typically, the creator of objects gets all permissions
  - Might even have the permission to GRANT permissions
  - This requires a recursive revoke process
- Privileges are stored in the directory/catalogue, conceptually as a matrix

	EMP	ENAME	ASG
Casey	UPDATE	UPDATE	UPDATE
Jones	SELECT	SELECT	SELECT WHERE RESP $\neq$ “Manager”
Smith	NONE	SELECT	NONE

- Different materializations of the matrix are possible (by row, by columns, by element), allowing for different optimizations
  - e.g., by row makes the enforcement of authorization efficient, since all rights of a user are in a single tuple

Distributed Authorization Control

- Additional problems of **authorization control in a distributed environment** stem from the fact that objects and subjects are distributed:
  - remote user authentication
  - managmenet of distributed authorization rules
  - handling of views and of user groups
- **Remote user authentication** is necessary since any site of a DDBMS may accept programs initiated and authorized at remote sites
- Two solutions are possible:
  - (username, password) is replicated at all sites and are communicated between the sites, whenever the relations at remote sites are accessed
    - ∗ beneficial if the users move from a site to a site
  - All sites of the DDBMS identify and authenticate themselves similarly as users do
    - ∗ intersite communication is protected by the use of the site password;
    - ∗ (username, password) is authorized by application at the start of the session;
    - ∗ no remote user authentication is required for accessing remote relations once the start site has been authenticated
    - ∗ beneficial if users are static

Semantic Integrity Constraints

- A database is said to be **consistent** if it satisfies a set of constraints, called **semantic integrity constraints**
- Maintain a database consistent by enforcing a set of constraints is a difficult problem
- Semantic integrity control evolved from procedural methods (in which the controls were embedded in application programs) to declarative methods
  - avoid data dependency problem, code redundancy, and poor performance of the procedural methods
- Two main types of constraints can be distinguished:
  - **Structural constraints**: basic semantic properties inherent to a data model e.g., unique key constraint in relational model
  - **Behavioral constraints**: regulate application behavior e.g., dependencies (functional, inclusion) in the relational model
- A semantic integrity control system has 2 components:
  - Integrity constraint specification
  - Integrity constraint enforcement

Semantic Integrity Constraint Specification

- **Integrity constraints specification**
  - In RDBMS, integrity constraints are defined as **assertions**, i.e., expression in tuple relational calculus
  - Variables are either universally ($\forall$) or existentially ($\exists$) quantified
  - Declarative method
  - Easy to define constraints
  - Can be seen as a query qualification which is either true or false
  - Definition of database consistency clear
  - 3 types of integrity constraints/assertions are distinguished:
    - predefined
    - precompiled
    - general constraints

- In the following examples we use the following relations:
  EMP(ENO, ENAME, TITLE)
  PROJ(PNO, PNAME, BUDGET)
  ASG(ENO, PNO, RESP, DUR)

Semantic Integrity Constraint Specification . . .

- **Predefined constraints** are based on simple keywords and specify the more common contraints of the relational model
- Not-null attribute:
  - e.g., Employee number in EMP cannot be null
    `ENO` **NOT NULL** `IN EMP`
- Unique key:
  - e.g., the pair (ENO,PNO) is the unique key in ASG
    `(ENO, PNO)` **UNIQUE IN** `ASG`
- Foreign key:
  - e.g., PNO in ASG is a foreign key matching the primary key PNO in PROJ
    `PNO` **IN** `ASG` **REFERENCES** `PNO IN PROJ`
- Functional dependency:
  - e.g., employee number functionally determines the employee name
    `ENO` **IN** `EMP` **DETERMINES** `ENAME`

Semantic Integrity Constraint Specification . . .

- **Precompiled constraints** express preconditions that must be satisfied by all tuples in a relation for a given update type
- General form:

  ```
  CHECK ON <relation> [WHEN <update type>] <qualification>
  ```

- Domain constraint, e.g., constrain the budget:

  ```
  CHECK ON PROJ(BUDGET>500000 AND BUDGET  1000000)
  ```

- Domain constraint on deletion, e.g., only tuples with budget 0 can be deleted:

  ```
  CHECK ON PROJ WHEN DELETE (BUDGET = 0)
  ```

- Transition constraint, e.g., a budget can only increase:

  ```
  CHECK ON PROJ (NEW.BUDGET > OLD.BUDGET AND
                NEW.PNO = OLD.PNO)
  ```

  - OLD and NEW are implicitly defined variables to identify the tuples that are subject to update

Semantic Integrity Constraint Specification . . .

- **General constraints** may involve more than one relation
- General form:

  CHECK ON `<variable>:<relation> (<qualification>)`

- Functional dependency:

```
CHECK ON e1:EMP, e2:EMP
(e1.ENAME = e2.ENAME IF e1.ENO = e2.ENO)
```

- Constraint with aggregate function:

  e.g., The total duration for all employees in the CAD project is less than 100

```
CHECK ON g:ASG, j:PROJ
( SUM(g.DUR WHERE g.PNO=j.PNO) < 100
  IF j.PNAME="CAD/CAM" )
```

Semantic Integrity Constraints Enforcement

- **Enforcing semantic integrity constraints** consists of rejecting update programs that violate some integrity constraints
- Thereby, the major problem is to find **efficient algorithms**
- Two methods to enforce integrity constraints:
  - **Detection:**
    1. Execute update $u : D \rightarrow D_u$
    2. If D_u is inconsistent then compensate $D_u \rightarrow D'_u$ or undo $D_u \rightarrow D$

    * Also called *posttest*
    * May be costly if undo is very large
  - **Prevention:**
    Execute $u : D \rightarrow D_u$ only if D_u will be consistent
    * Also called *pretest*
    * Generally more efficient
    * Query modification algorithm by Stonebraker (1975) is a preventive method that is particularly efficient in enforcing domain constraints.
      - Add the assertion qualification (constraint) to the update query and check it immediately for each tuple

Semantic Integrity Constraints Enforcement . . .

- **Example**: Consider a query for increasing the budget of CAD/CAM projects by 10%:

```
UPDATE PROJ
SET BUDGET = BUDGET * 1.1
WHERE PNAME = ‘‘CAD/CAM’’
```

and the domain constraint

```
CHECK ON PROJ (BUDGET >= 50K AND BUDGET <= 100K)
```

The query modification algorithm transforms the query into:

```
UPDATE PROJ
SET BUDGET = BUDGET * 1.1
WHERE PNAME = ‘‘CAD/CAM’’
      AND NEW.BUDGET >= 50K
      AND NEW.BUDGET <= 100K
```

Distributed Constraints

- Three classes of **distributed integrity constraints/assertions** are distinguished:
  - **Individual** assertions
    - Single relation, single variable
    - Refer only to tuples to be updated independenlty of the rest of the DB
    - e.g., domain constraints
  - **Set-oriented** assertions
    - Single relation, multi variable (e.g., functional dependencies)
    - Multi-relation, multi-variable (e.g., foreign key constraints)
    - Multiple tuples form possibly different relations are involved
  - Assertions involving **aggregates**
    - Special, costly processing of aggregates is required

Distributed Constraints

- Particular difficulties with distributed constraints arise from the fact that relations are fragmented and replicated:
  - Definition of assertions
  - Where to store the assertions?
  - How to enforce the assertions?

Distributed Constraints

- **Definition and storage** of assertions
  - The definition of a new integrity assertion can be started at one of the sites that store the relations involved in the assertion, but needs to be propagated to sites that might store fragments of that relation.
  - Individual assertions
    - The assertion definition is sent to all other sites that contain fragments of the relation involved in the assertion.
    - At each fragment site, check for compatibility of assertion with data
    - If compatible, store; otherwise reject
    - If any of the sites rejects, globally reject
  - Set-oriented assertions
    - Involves joins (between fragments or relations)
    - Maybe necessary to perform joins to check for compatibility
    - Store if compatible

Distributed Constraints

- **Enforcement** of assertions in DDBMS is more complex than in centralized DBMS
- The main problem is to decide where (at which site) to enforce each assertion?
  - Depends on type of assertion, type of update, and where update is issued
- Individual assertions
  - Update = insert
    - enforce at the site where the update is issued (i.e., where the user inserts the tuples)
  - Update = delete or modify
    - Send the assertions to all the sites involved (i.e., where qualified tuples are updated)
    - Each site enforce its own assertion
- Set-oriented assertions
  - Single relation
    - Similar to individual assertions with qualified updates
  - Multi-relation
    - Move data between sites to perform joins
    - Then send the result to the query master site (the site the update is issued)

Conclusion

- Views enable full logical data independence
  - Queries expressed on views are translated into queries expressed on base relationships
  - Views can be updatable and non-updatable
- Three aspects are involved in authorization: (user, operation, data object)
- Semantic integrity constraints maintain database consistency
  - Individual assertions are checked at each fragment site, check for compatibility
  - Set-oriented assertions involve joins between fragments and optimal enforcement of the constraints is similar to distributed query optimization
  - Constraint detection vs. constraint prevention

Chapter 5: Overview of Query Processing

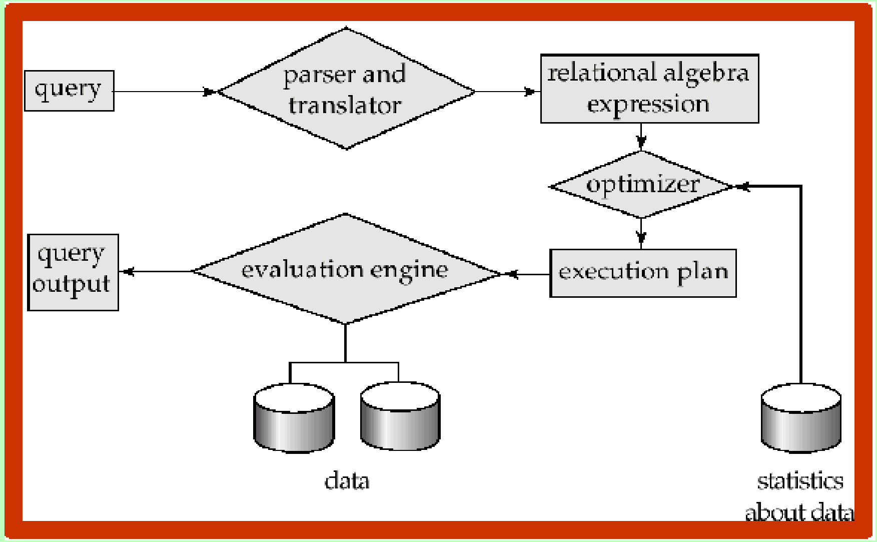
- Query Processing Overview
- Query Optimization
- Distributed Query Processing Steps

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Query Processing Overview

- **Query processing**: A 3-step process that transforms a high-level query (of relational calculus/SQL) into an **equivalent** and **more efficient** lower-level query (of relational algebra).

  1. **Parsing and translation**
     - Check syntax and verify relations.
     - Translate the query into an equivalent relational algebra expression.
  2. **Optimization**
     - Generate an optimal evaluation plan (with lowest cost) for the query plan.
  3. **Evaluation**
     - The query-execution engine takes an (optimal) evaluation plan, executes that plan, and returns the answers to the query.



Query Processing . . .

- The success of RDBMSs is due, in part, to the availability
  - of declarative query languages that allow to easily express complex queries without knowing about the details of the physical data organization and
  - of advanced query processing technology that transforms the high-level user/application queries into efficient lower-level query execution strategies.
- The query transformation should achieve both **correctness** and **efficiency**
  - The main difficulty is to achieve the efficiency
  - This is also one of the most important tasks of any DBMS
- **Distributed query processing**: Transform a high-level query (of relational calculus/SQL) on a distributed database (i.e., a set of global relations) into an **equivalent** and **efficient** lower-level query (of relational algebra) on relation fragments.
- Distributed query processing is more complex
  - Fragmentation/replication of relations
  - Additional communication costs
  - Parallel execution

Query Processing Example

- **Example:** Transformation of an SQL-query into an RA-query.
  Relations: EMP(ENO, ENAME, TITLE), ASG(ENO,PNO,RESP,DUR)
  Query: *Find the names of employees who are managing a project*?

  – High level query

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

  – Two possible transformations of the query are:

  ∗ Expression 1: $\Pi_{ENAME}(\sigma_{DUR>37 \wedge EMP.ENO=ASG.ENO}(EMP \times ASG))$

  ∗ Expression 2: $\Pi_{ENAME}(EMP \bowtie_{ENO} (\sigma_{DUR>37}(ASG)))$

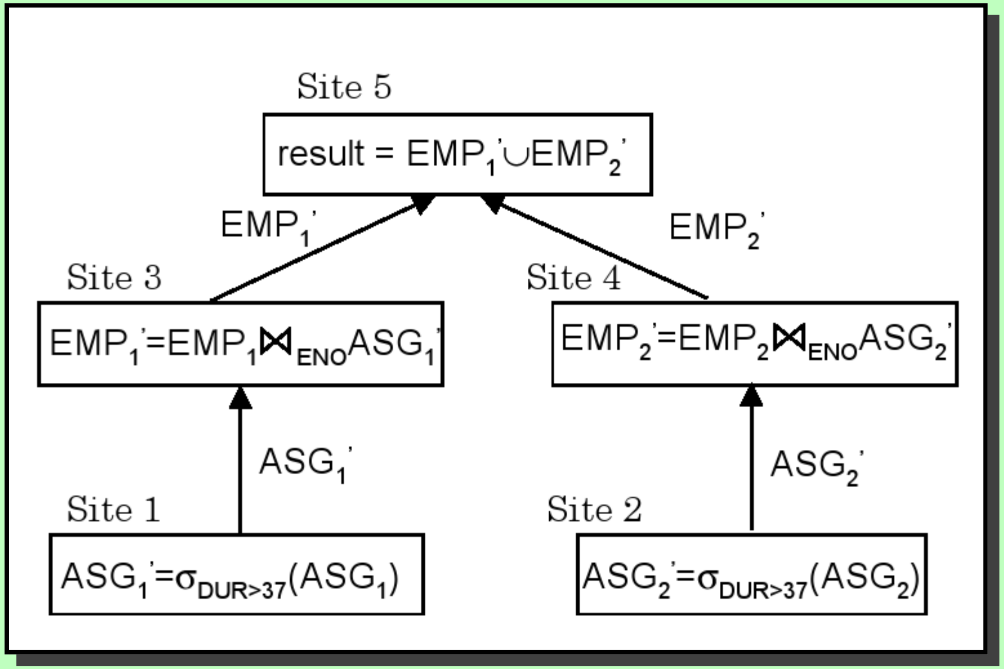
  – Expression 2 avoids the expensive and large intermediate Cartesian product, and therefore typically is better.

Query Processing Example . . .

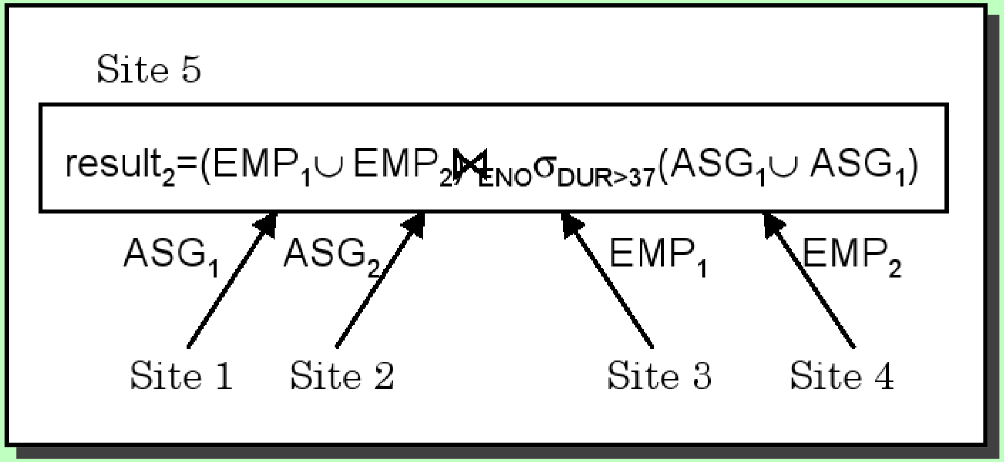
- We make the following assumptions about the data fragmentation
  - Data is (horizontally) fragmented:
    - Site1: $ASG1 = \sigma_{ENO \leq ”E3”}(ASG)$
    - Site2: $ASG2 = \sigma \qquad (ASG)$
    - Site3: $EMP1 = \sigma \qquad (EMP)$
    - Site4: $EMP2 = \sigma_{ENO > ”E3”}(EMP)$
    - Site5: Result
  - Relations ASG and EMP are fragmented in the same way
  - Relations ASG and EMP are locally clustered on attributes RESP and ENO, respectively

Query Processing Example . . .

- Now consider the expression $\Pi_{ENAME}(EMP \bowtie_{ENO} (\sigma_{DUR>37}(ASG)))$
- Strategy 1 (partially parallel execution):
  - Produce ASG$'_1$ and move to Site 3
  - Produce ASG$'_2$ and move to Site 4
  - Join ASG$'_1$ with EMP$_1$ at Site 3 and move the result to Site 5
  - Join ASG$'_2$ with EMP$_2$ at Site 4 and move the result to Site 5
  - Union the result in Site 5


Site 5
result = EMP$_1$'∪EMP$_2$'
EMP$_1$'
EMP$_2$'
Site 3
Site 4
EMP$_1$'=EMP$_1$⋈$_{ENO}$ASG$_1$'
EMP$_2$'=EMP$_2$⋈$_{ENO}$ASG$_2$'
ASG$_1$'
ASG$_2$'
Site 1
Site 2
ASG$_1$'=$\sigma_{DUR>37}$(ASG$_1$)
ASG$_2$'=$\sigma_{DUR>37}$(ASG$_2$)

- Strategy 2:
  - Move ASG$_1$ and ASG$_2$ to Site 5
  - Move EMP$_1$ and EMP$_2$ to Site 5
  - Select and join at Site 5


Site 5
result$_2$=(EMP$_1$∪EMP$_2$)⋈$_{ENO}\sigma_{DUR>37}$(ASG$_1$∪ ASG$_1$)
ASG$_1$
ASG$_2$
EMP$_1$
EMP$_2$
Site 1
Site 2
Site 3
Site 4

- For simplicity, the final projection is omitted.

Query Processing Example . . .

- Calculate the cost of the two strategies under the following assumptions:
  - Tuples are uniformly distributed to the fragments; 20 tuples satisfy DUR$>$37
  - size(EMP) = 400, size(ASG) = 1000
  - tuple access cost = 1 unit; tuple transfer cost = 10 units
  - ASG and EMP have a local index on DUR and ENO

- Strategy 1

– Produce ASG's: (10+10) * tuple access cost	20
– Transfer ASG's to the sites of EMPs: (10+10) * tuple transfer cost	200
– Produce EMP's: (10+10) * tuple access cost * 2	40
– Transfer EMP's to result site: (10+10) * tuple transfer cost	200
– Total cost	460

- Strategy 2

– Transfer EMP$_1$, EMP$_2$ to site 5: 400 * tuple transfer cost	4,000
– Transfer ASG$_1$, ASG$_2$ to site 5: 1000 * tuple transfer cost	10,000
– Select tuples from ASG$_1$ $\cup$ ASG$_2$: 1000 * tuple access cost	1,000
– Join EMP and ASG': 400 * 20 * tuple access cost	8,000
– Total cost	23,000

Query Optimization

- **Query optimization** is a crucial and difficult part of the overall query processing
- Objective of query optimization is to **minimize** the following cost function:

$$\text{I/O cost} + \text{CPU cost} + \text{communication cost}$$

- Two different scenarios are considered:
  - Wide area networks
    - ∗ Communication cost dominates
      - · 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

Query Optimization . . .

- **Ordering of the operators** of relational algebra is crucial for efficient query processing
- Rule of thumb: move expensive operators at the end of query processing
- Cost of RA operations:

Operation	Complexity
Select, Project (without duplicate elimination)	$O(n)$
Project (with duplicate elimination)	$O(n \log n)$
Group Join Semi-join Division Set Operators	$O(n \log n)$
Cartesian Product	$O(n^2)$

Query Optimization Issues

Several issues have to be considered in query optimization

- Types of query optimizers
  - wrt the search techniques (exhaustive search, heuristics)
  - wrt the time when the query is optimized (static, dynamic)
- Statistics
- Decision sites
- Network topology
- Use of semijoins

Query Optimization Issues . . .

- **Types of Query Optimizers wrt Search Techniques**
  - Exhaustive search
    - Cost-based
    - Optimal
    - Combinatorial complexity in the number of relations
  - Heuristics
    - Not optimal
    - Regroups common sub-expressions
    - Performs selection, projection first
    - Replaces a join by a series of semijoins
    - Reorders operations to reduce intermediate relation size
    - Optimizes individual operations

Query Optimization Issues . . .

- **Types of Query Optimizers wrt Optimization Timing**
  - Static
    - ∗ Query is optimized prior to the execution
    - ∗ As a consequence it is difficult to estimate the size of the intermediate results
    - ∗ Typically amortizes over many executions
  - Dynamic
    - ∗ Optimization is done at run time
    - ∗ Provides exact information on the intermediate relation sizes
    - ∗ Have to re-optimize for multiple executions
  - Hybrid
    - ∗ First, the query is compiled using a static algorithm
    - ∗ Then, if the error in estimate sizes greater than threshold, the query is re-optimized at run time

Query Optimization Issues . . .

- **Statistics**
  - Relation/fragments
    - Cardinality
    - Size of a tuple
    - Fraction of tuples participating in a join with another relation/fragment
  - Attribute
    - Cardinality of domain
    - Actual number of distinct values
    - Distribution of attribute values (e.g., histograms)
  - 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
    - Knowledge about the entire distributed database is needed
  - Distributed
    - Cooperation among sites to determine the schedule
    - Only local information is needed
    - Cooperation comes with an overhead cost
  - Hybrid
    - One site determines the global schedule
    - Each site optimizes the local sub-queries

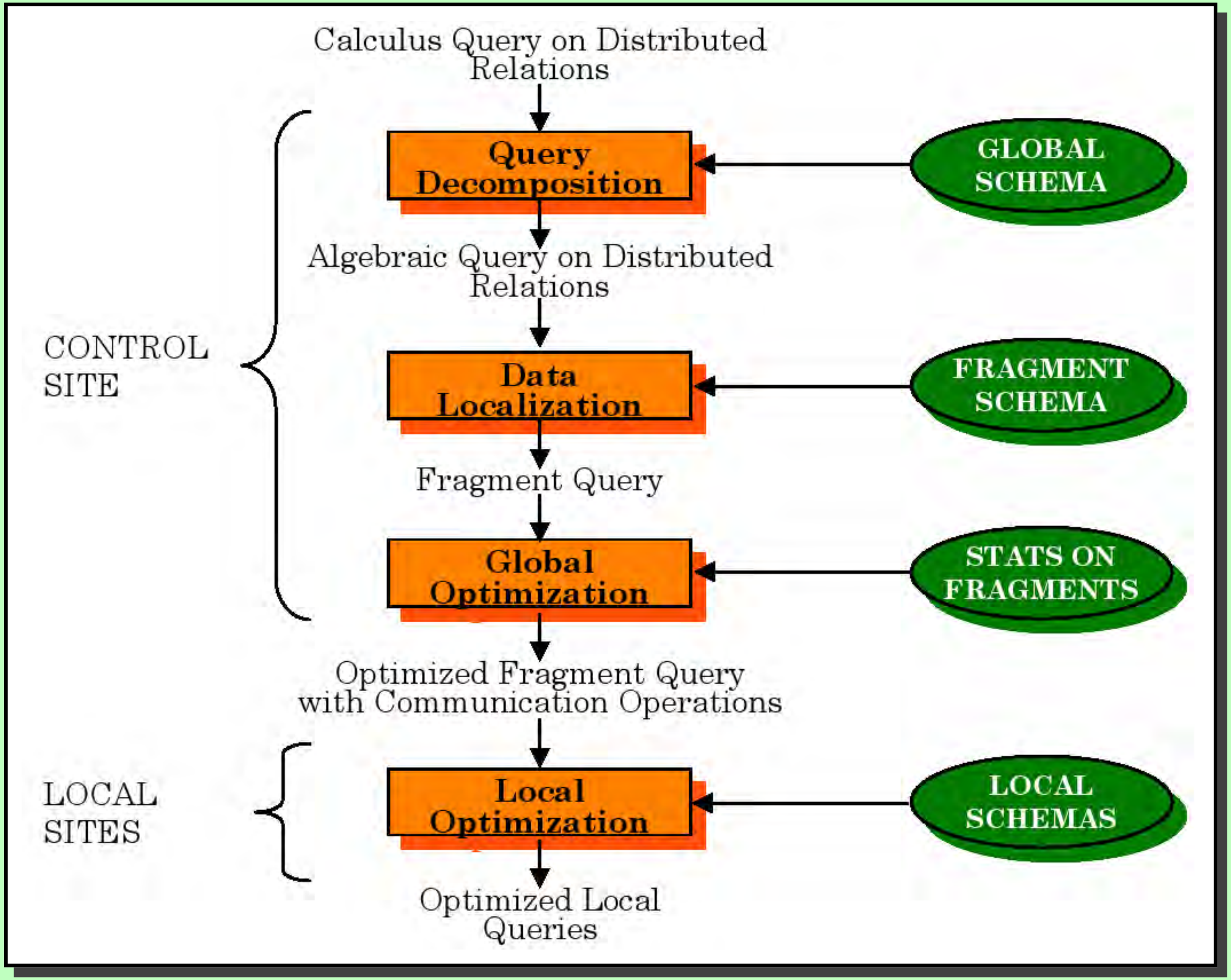
Query Optimization Issues . . .

- **Network topology**
  - Wide area networks (WAN) point-to-point
    - ∗ Characteristics
      - · Low bandwidth
      - · Low speed
      - · High protocol overhead
    - ∗ Communication cost dominate; all other cost factors are ignored
    - ∗ 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

Query Optimization Issues . . .

- **Use of Semijoins**
  - Reduce the size of the join operands by first computing semijoins
  - Particularly relevant when the main cost is the communication cost
  - Improves the processing of distributed join operations by reducing the size of data exchange between sites
  - However, the number of messages as well as local processing time is increased

Distributed Query Processing Steps



Conclusion

- Query processing transforms a high level query (relational calculus) into an equivalent lower level query (relational algebra). The main difficulty is to achieve the efficiency in the transformation
- Query optimization aims to mimize the cost function:
- Query optimizers vary by search type (exhaustive search, heuristics) and by type of the algorithm (dynamic, static, hybrid). Different statistics are collected to support the query optimization process
- Query optimizers vary by decision sites (centralized, distributed, hybrid)
- Query processing is done in the following sequence: query decomposition→data localization→global optimization→ local optimization

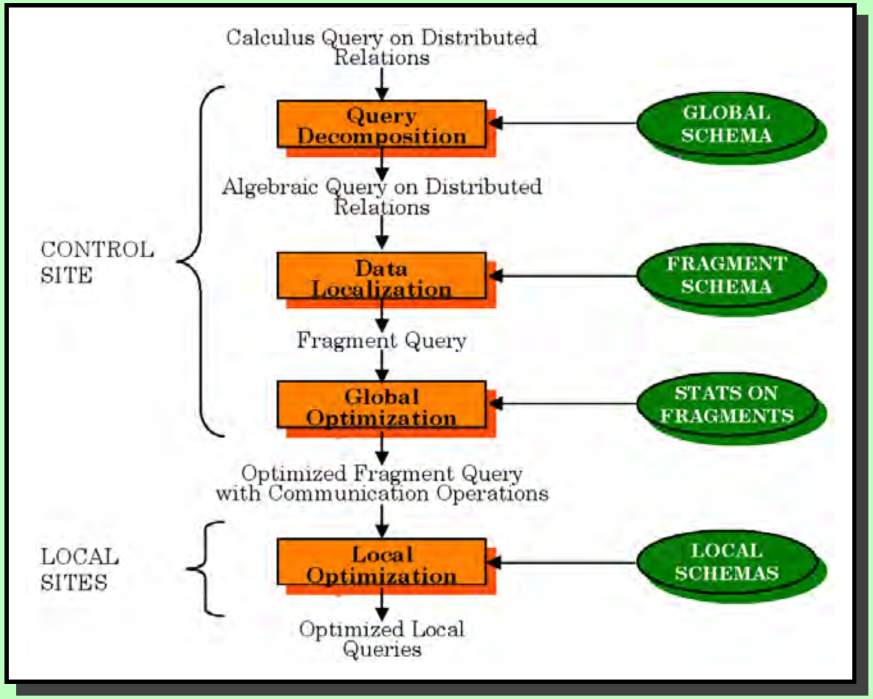
Chapter 6: Query Decomposition and Data Localization

- Query Decomposition
- Data Localization

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Query Decomposition

- **Query decomposition:** Mapping of calculus query (SQL) to algebra operations (select, project, join, rename)
- Both input and output queries refer to global relations, without knowledge of the distribution of data.
- The output query is semantically correct and good in the sense that redundant work is avoided.



- Query decomposistion consists of 4 steps:
  1. **Normalization**: Transform query to a normalized form
  2. **Analysis**: Detect and reject ”incorrect” queries; possible only for a subset of relational calculus
  3. **Elimination of redundancy**: Eliminate redundant predicates
  4. **Rewriting**: Transform query to RA and optimize query

Query Decomposition – Normalization

- **Normalization:** Transform the query to a normalized form to facilitate further processing. Consists mainly of two steps.

  1. **Lexical** and **syntactic** analysis
     - Check validity (similar to compilers)
     - Check for attributes and relations
     - Type checking on the qualification
  2. Put into **normal form**
     - With SQL, the query qualification (WHERE clause) is the most difficult part as it might be an arbitrary complex predicate preceeded by quantifiers (,)
     - Conjunctive normal form

$$(p_{11} \lor p_{12} \lor \cdots \lor p_{1n}) \land \cdots \land (p_{m1} \lor p_{m2} \lor \cdots \lor p_{mn})$$

     - Disjunctive normal form

$$(p_{11} \land p_{12} \land \cdots \land p_{1n}) \lor \cdots \lor (p_{m1} \land p_{m2} \land \cdots \land p_{mn})$$

     - In the disjunctive normal form, the query can be processed as independent conjunctive subqueries linked by unions (corresponding to the disjunction)

Query Decomposition – Normalization . . .

- **Example:** Consider the following query: *Find the names of employees who have been working on project P1 for 12 or 24 months?*

- The query in SQL:

```
SELECT ENAME
FROM   EMP, ASG
WHERE  EMP.ENO = ASG.ENO AND
       ASG.PNO = ‘‘P1’’ AND
       DUR = 12 OR DUR = 24
```

- The qualification in conjunctive normal form:

$$EMP.ENO = ASG.ENO \wedge ASG.PNO = "P1" \wedge (DUR = 12 \vee DUR = 24)$$

- The qualification in disjunctive normal form:

$$(EMP.ENO = ASG.ENO \wedge ASG.PNO = "P1" \wedge DUR = 12) \vee$$
$$(EMP.ENO = ASG.ENO \wedge ASG.PNO = "P1" \wedge DUR = 24)$$

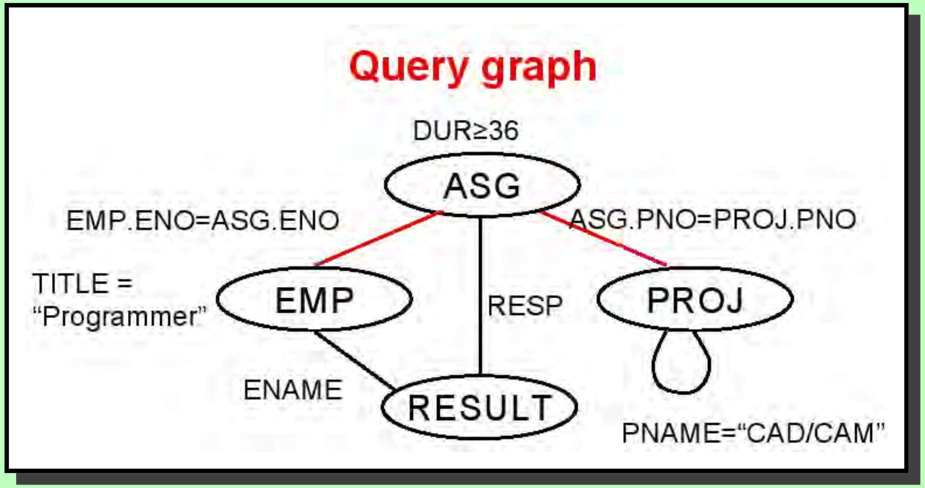
Query Decomposition – Analysis

- **Analysis:** Identify and reject type incorrect or semantically incorrect queries
- Type incorrect
  - Checks whether the attributes and relation names of a query are defined in the global schema
  - Checks whether the operations on attributes do not conflict with the types of the attributes, e.g., a comparison $>$ operation with an attribute of type string
- Semantically incorrect
  - Checks whether the components contribute in any way to the generation of the result
  - Only a subset of relational calculus queries can be tested for correctness, i.e., those that do not contain disjunction and negation
  - Typical data structures used to detect the semantically incorrect queries are:
    - $*$ Connection graph (query graph)
    - $*$ Join graph

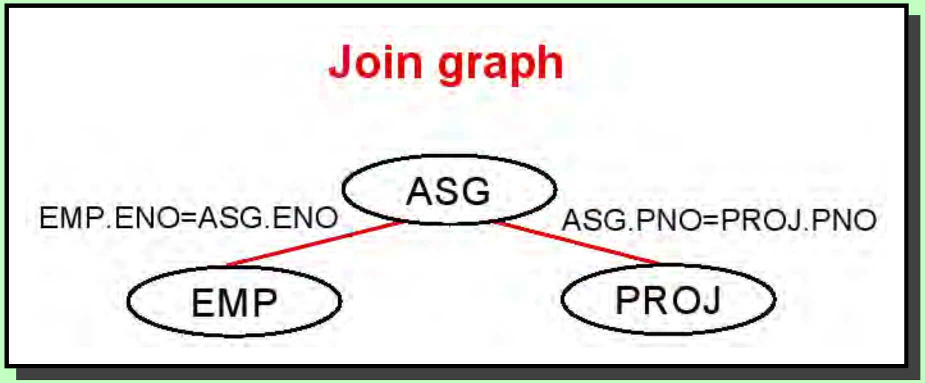
Query Decomposition – Analysis . . .

- **Example:** Consider a query:

```
SELECT   ENAME,RESP
FROM     EMP, ASG, PROJ
WHERE    EMP.ENO = ASG.ENO
AND      ASG.PNO = PROJ.PNO
AND      PNAME = "CAD/CAM"
AND      DUR ≥ 36
AND      TITLE = "Programmer"
```



- Query/connection graph
  - Nodes represent operand or result relation
  - Edge represents a join if both connected nodes represent an operand relation, otherwise it is a projection

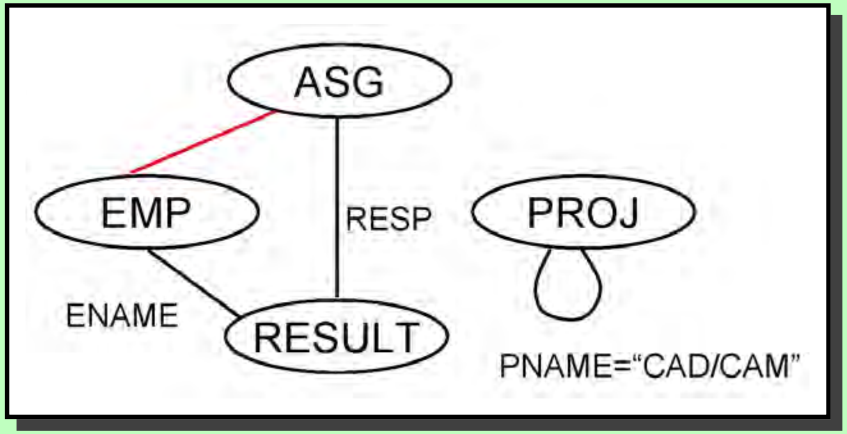


- Join graph
  - a subgraph of the query graph that considers only the joins
- Since the query graph **is connected**, the query is semantically correct

Query Decomposition – Analysis . . .

- **Example:** Consider the following query and its query graph:

```
SELECT   ENAME,RESP
FROM      EMP, ASG, PROJ
WHERE     EMP.ENO = ASG.ENO
AND       PNAME = "CAD/CAM"
AND       DUR ≥ 36
AND       TITLE = "Programmer"
```



- Since the graph **is not connected**, the query is semantically incorrect.
- 3 possible solutions:
  - Reject the query
  - Assume an implicit Cartesian Product between ASG and PROJ
  - Infer from the schema the missing join predicate ASG.PNO = PROJ.PNO

Query Decomposition – Elimination of Redundancy

- **Elimination of redundancy:** Simplify the query by eliminate redundancies, e.g., redundant predicates
  - Redundancies are often due to semantic integrity constraints expressed in the query language
  - e.g., queries on views are expanded into queries on relations that satiesfy certain integrity and security constraints
- Transformation rules are used, e.g.,
  - $p \wedge p \iff p$
  - $p \vee p \iff p$
  - $p \wedge true \iff p$
  - $p \vee false \iff p$
  - $p \wedge false \iff false$
  - $p \vee true \iff true$
  - $p \wedge \neg p \iff false$
  - $p \vee \neg p \iff true$
  - $p_1 \wedge (p_1 \vee p_2) \iff p_1$
  - $p_1 \vee (p_1 \wedge p_2) \iff p_1$

Query Decomposition – Elimination of Redundancy . . .

- **Example:** Consider the following query:

```
SELECT   TITLE
FROM      EMP
WHERE     EMP.ENAME = "J. Doe"
OR        (NOT(EMP.TITLE = "Programmer")
AND       ( EMP.TITLE = "Elect. Eng."
OR          EMP.TITLE = "Programmer" )
AND       NOT(EMP.TITLE = "Elect. Eng."))
```

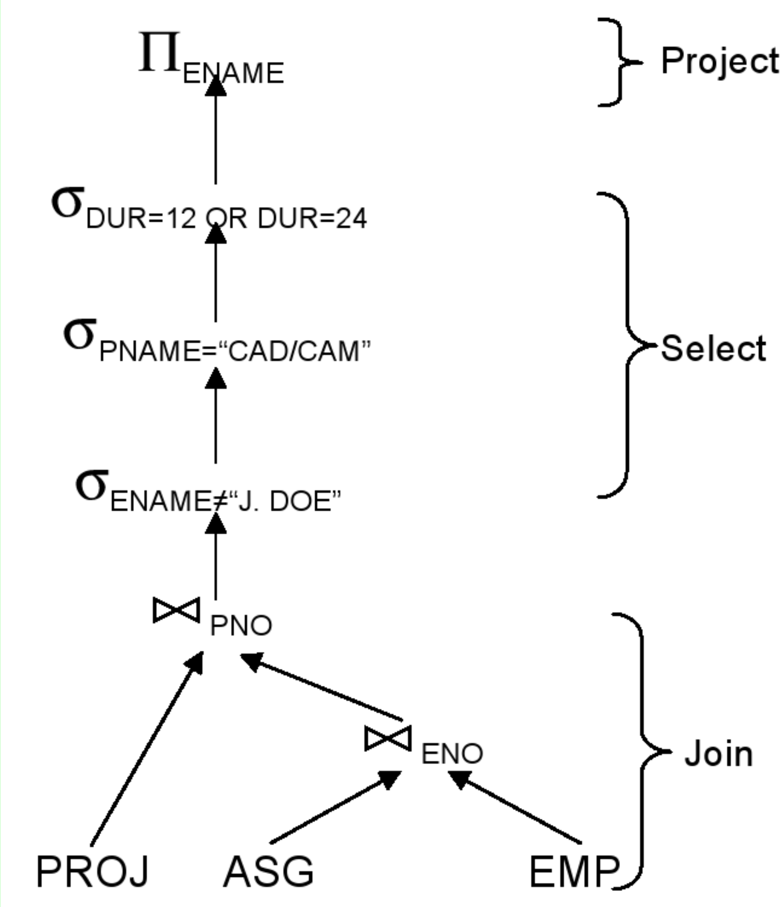
- Let p_1 be ENAME = ”J. Doe”, p be TITLE = ”Programmer” and p be TITLE = ”Elect. Eng.”
- Then the qualification can be written as $p_1 \vee (\neg p_2 \wedge (p_2 \vee p_3) \wedge \neg p_3)$ and then be transformed into p_1
- Simplified query:

```
SELECT TITLE
FROM    EMP
WHERE   EMP.ENAME = "J. Doe"
```

Query Decomposition – Rewriting

- **Rewriting:** Convert relational calculus query to relational algebra query and find an **efficient** expression.
- **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)
  ```



Π_{ENAME}
Project
$\sigma_{DUR=12 \text{ OR } DUR=24}$
$\sigma_{PNAME=\text{“CAD/CAM”}}$
$\sigma_{ENAME\neq\text{“J. DOE”}}$
Select
$\bowtie_{PNO}$
$\bowtie_{ENO}$
Join
PROJ
ASG
EMP

- A **query tree** represents the RA-expression
  - Relations are leaves (FROM clause)
  - Result attributes are root (SELECT clause)
  - Intermediate leaves should give a result from the leaves to the root

Query Decomposition – Rewriting . . .

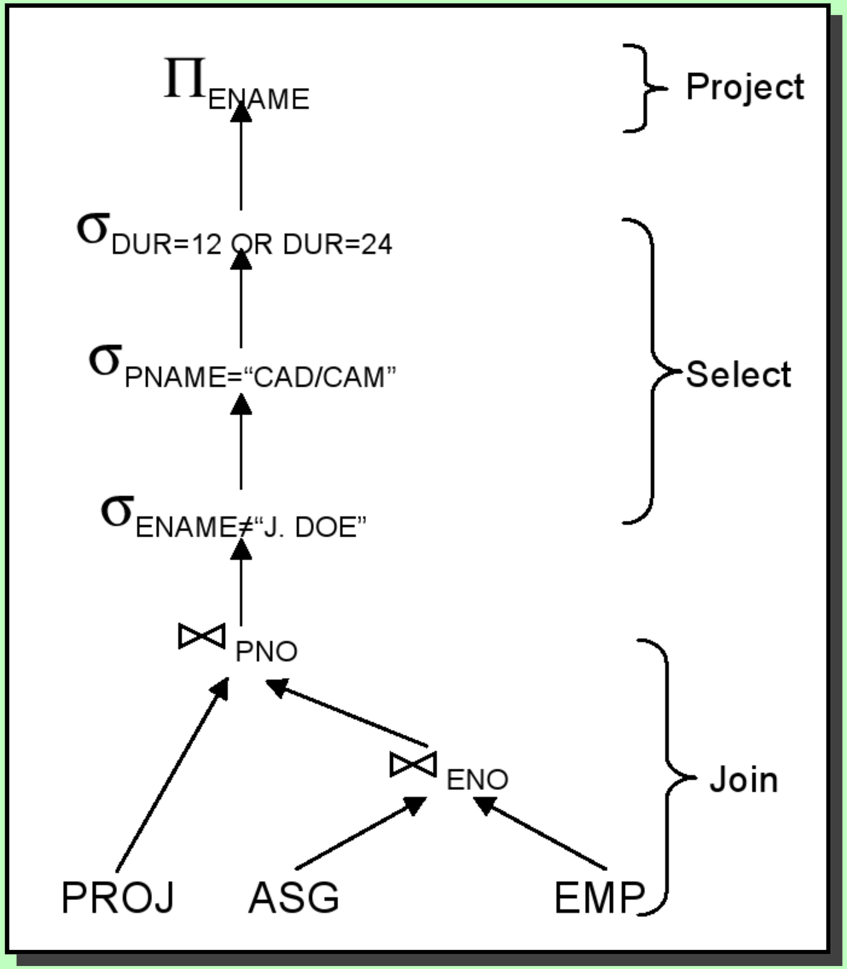
- By applying **transformation rules**, many different trees/expressions may be found that are **equivalent** to the original tree/expression, but might be more efficient.
- In the following we assume relations $R(A_1, \ldots, A_n)$, $S(B_1, \ldots, B_n)$, and T which is union-compatible to R.
- **Commutativity** of binary operations
  - $R \times S = S \times R$
  - $R \bowtie S = S \bowtie R$
  - $R \cup S = S \quad R$
- **Associativity** of binary operations
  - $(R \times S) \times T = R \times (S \times T)$
  - $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- **Idempotence** of unary operations
  - $\Pi_A(\Pi_A(R)) = \Pi_A(R)$
  - $\sigma_{p1(A1)}(\sigma_{p2(A2)}(R)) = \sigma_{p1(A1) \wedge p2(A2)}(R)$

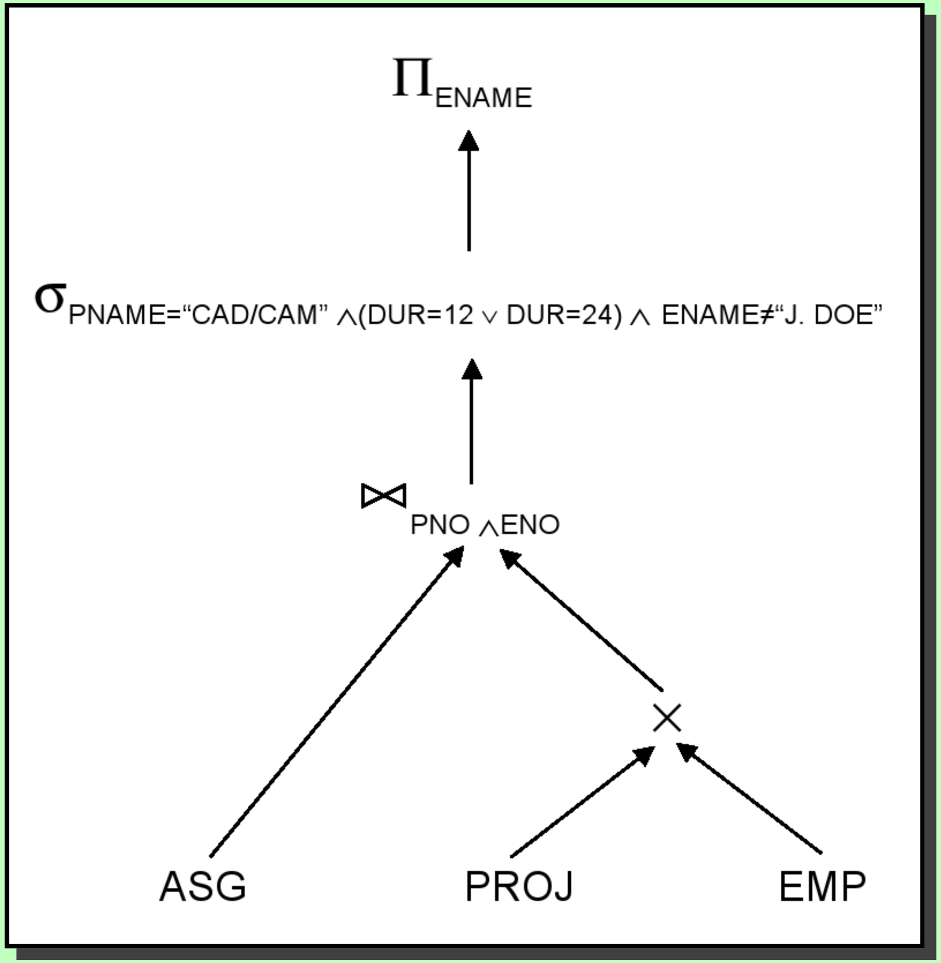
Query Decomposition – Rewriting . . .

- **Commuting selection** with binary operations
  - $\sigma_{p(A)}(R \times S) \iff \sigma_{p(A)}(R) \times S$
  - $\sigma_{p(A_1)}(R \bowtie_{p(A_2,B_2)} S) \iff \sigma_{p(A_1)}(R) \bowtie_{p(A_2,B_2)} S$
  - $\sigma_{p(A)}(R \cup T) \iff \sigma_{p(A)}(R) \cup \sigma_{p(A)}(T)$
    - (A belongs to R and T)
- **Commuting projection** with binary operations (assume $C = A \quad B$, $A' \subseteq A, B' \subseteq B$)
  - $\Pi_C(R \times S) \iff \Pi_{A'}(R) \times \Pi_{B'}(S)$
  - $\Pi_C(R \bowtie_{p(A',B')} S) \iff \Pi_{A'}(R) \bowtie_{p(A',B')} \Pi_{B'}(S)$
  - $\Pi_C(R \cup S) \iff \Pi_C(R) \cup \Pi_C(S)$

Query Decomposition – Rewriting . . .

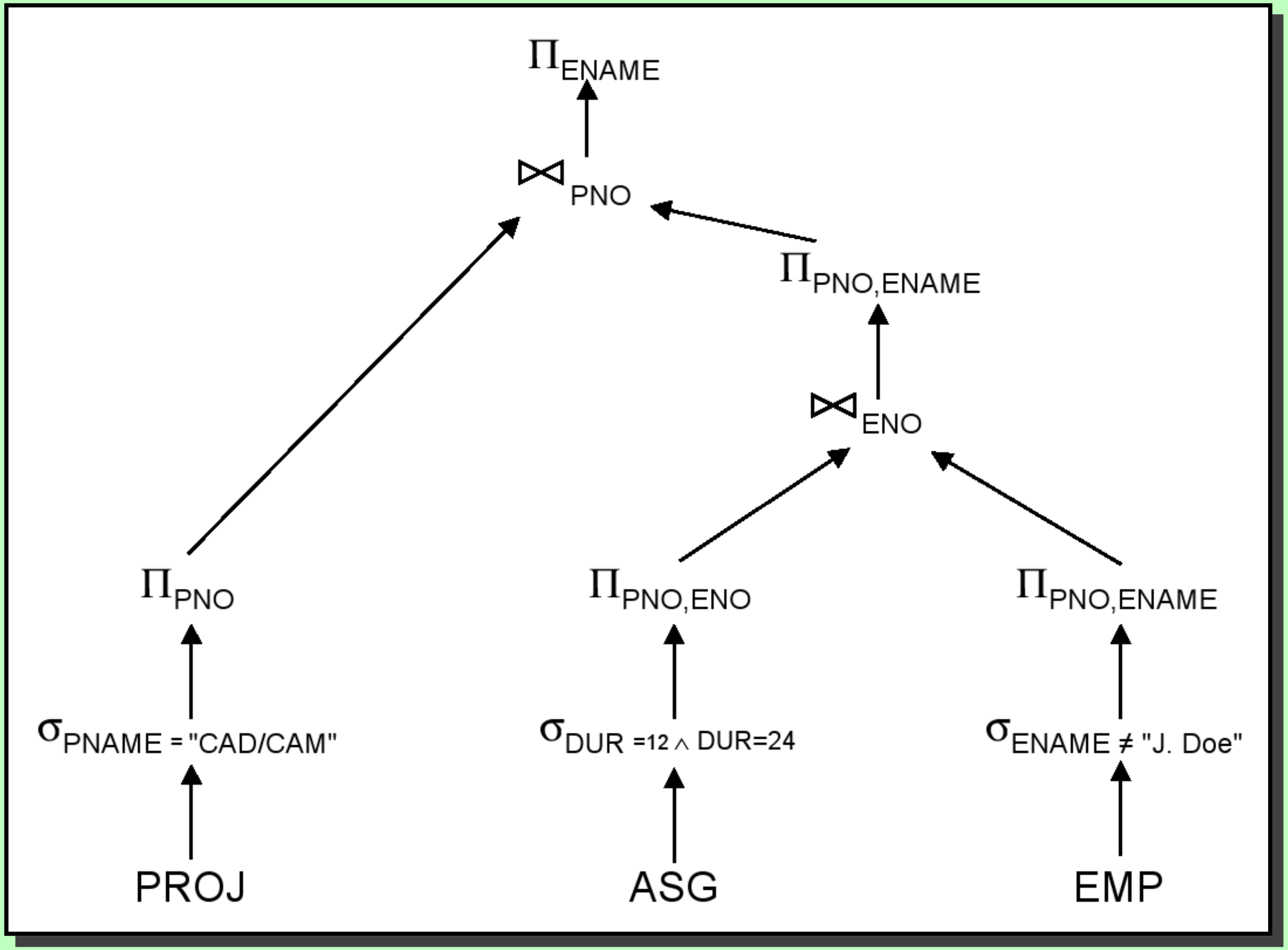
- **Example:** Two equivalent query trees for the previous example
  - Recall the schemas: EMP(ENO, ENAME, TITLE)
    PROJ(PNO, PNAME, BUDGET)
    ASG(ENO, PNO, RESP, DUR)


Π_{ENAME}
$\sigma_{DUR=12\ OR\ DUR=24}$
$\sigma_{PNAME=\text{“CAD/CAM”}}$
$\sigma_{ENAME\neq\text{“J. DOE”}}$
$\bowtie_{PNO}$
$\bowtie_{ENO}$
PROJ
ASG
EMP
Project
Select
Join


Π_{ENAME}
$\sigma_{PNAME=\text{“CAD/CAM”} \wedge (DUR=12 \vee DUR=24) \wedge ENAME\neq\text{“J. DOE”}}$
$\bowtie_{PNO \wedge ENO}$
×
ASG
PROJ
EMP

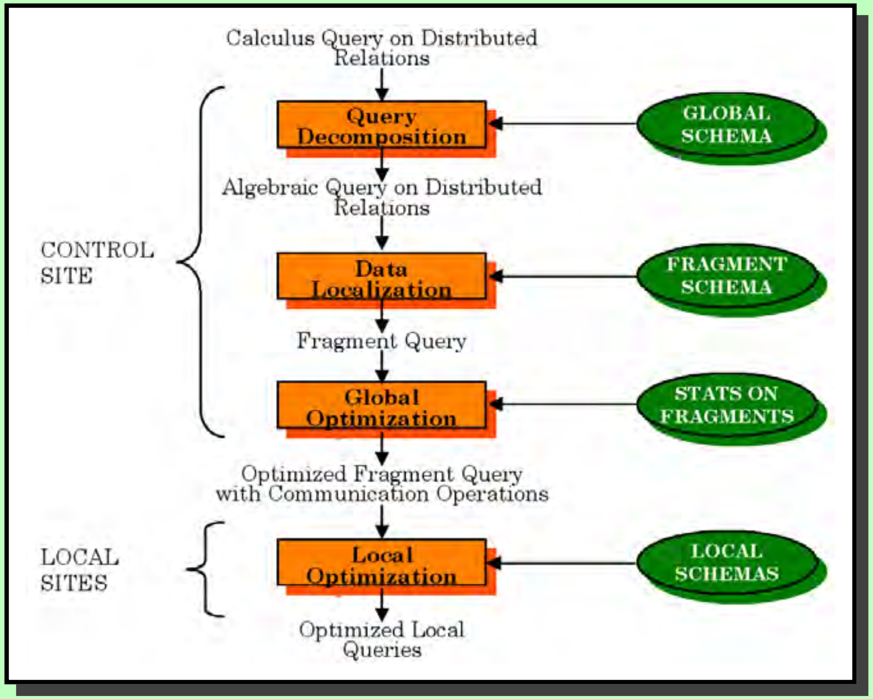
Query Decomposition – Rewriting . . .

- **Example (contd.):** Another equivalent query tree, which allows a more efficient query evaluation, since the most selective operations are applied first.



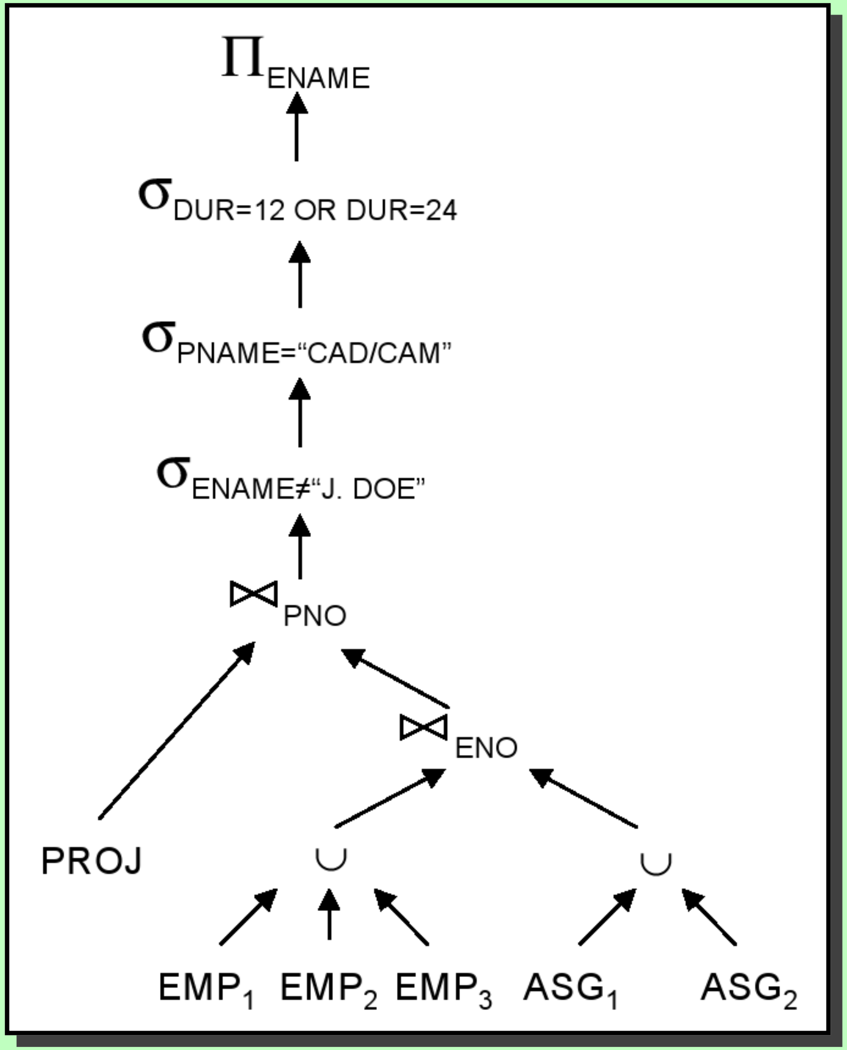
Data Localization

- **Data localization**
  - **Input:** Algebraic query on global conceptual schema
  - **Purpose:**
    - ∗ Apply data distribution information to the algebra operations and determine which fragments are involved
    - ∗ Substitute global query with queries on fragments
    - ∗ Optimize the global query



Data Localization . . .

- **Example:**
  - Assume EMP is horizontally fragmented into EMP1, EMP2, EMP3 as follows:
    - $EMP1 = \sigma_{ENO \leq "E3"}(EMP)$
    - $EMP2 = \sigma_{"E3" < ENO \leq "E6"}(EMP)$
    - $EMP3 = \sigma_{ENO > "E6"}(EMP)$
  - ASG fragmented into ASG1 and ASG2 as follows:
    - $ASG1 = \sigma \qquad\qquad (ASG)$
    - $ASG2 = \sigma_{ENO > "E3"}(ASG)$



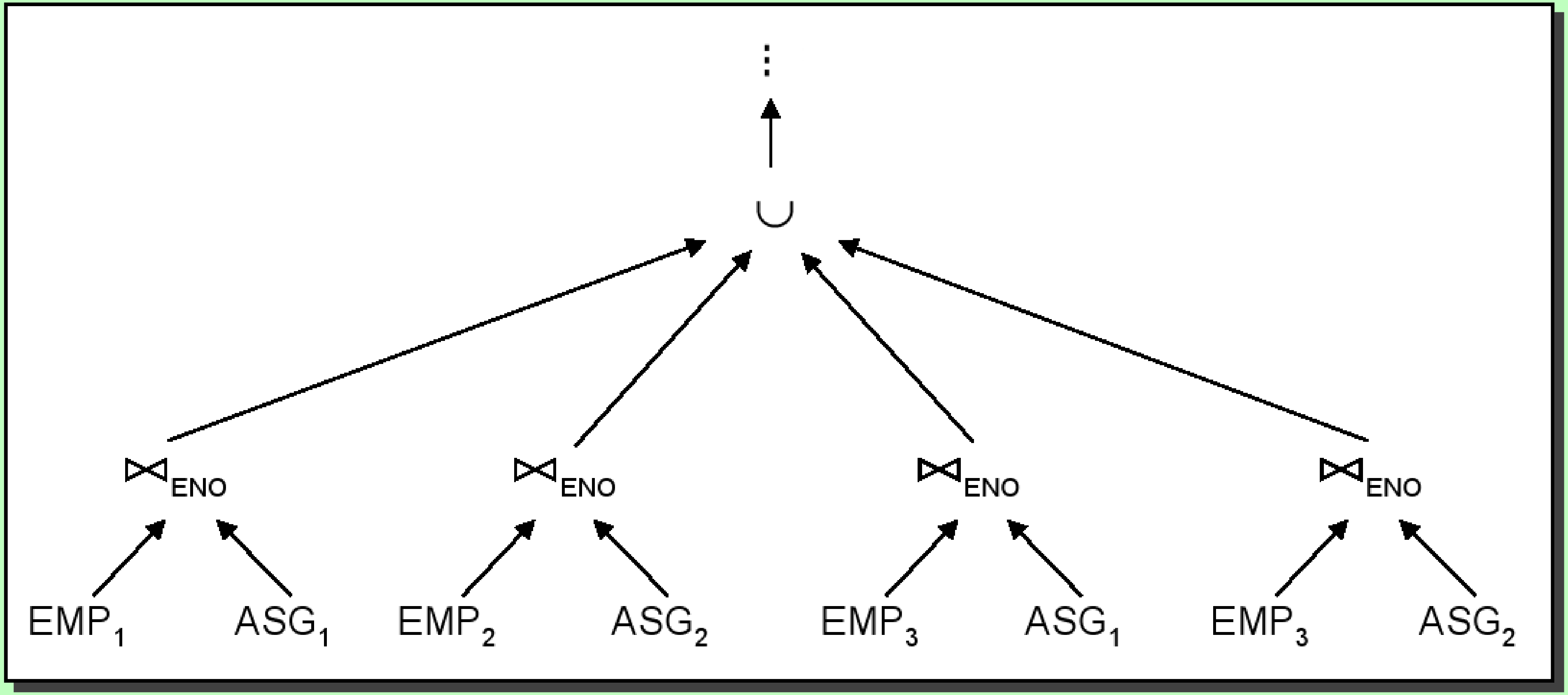
Π_{ENAME}
$\sigma_{DUR=12 \text{ OR } DUR=24}$
$\sigma_{PNAME=\text{"CAD/CAM"}}$
$\sigma_{ENAME \neq \text{"J. DOE"}}$
$\bowtie_{PNO}$
$\bowtie_{ENO}$
PROJ
$\cup$
$\cup$
EMP_1
EMP_2
EMP_3
ASG_1
ASG_2

- Simple approach: Replace in all queries
  - EMP by (EMP1∪EMP2∪ EMP3)
  - ASG by (ASG1∪ASG2)
  - Result is also called **generic query**

- In general, the **generic query is inefficient** since important restructurings and simplifications can be done.

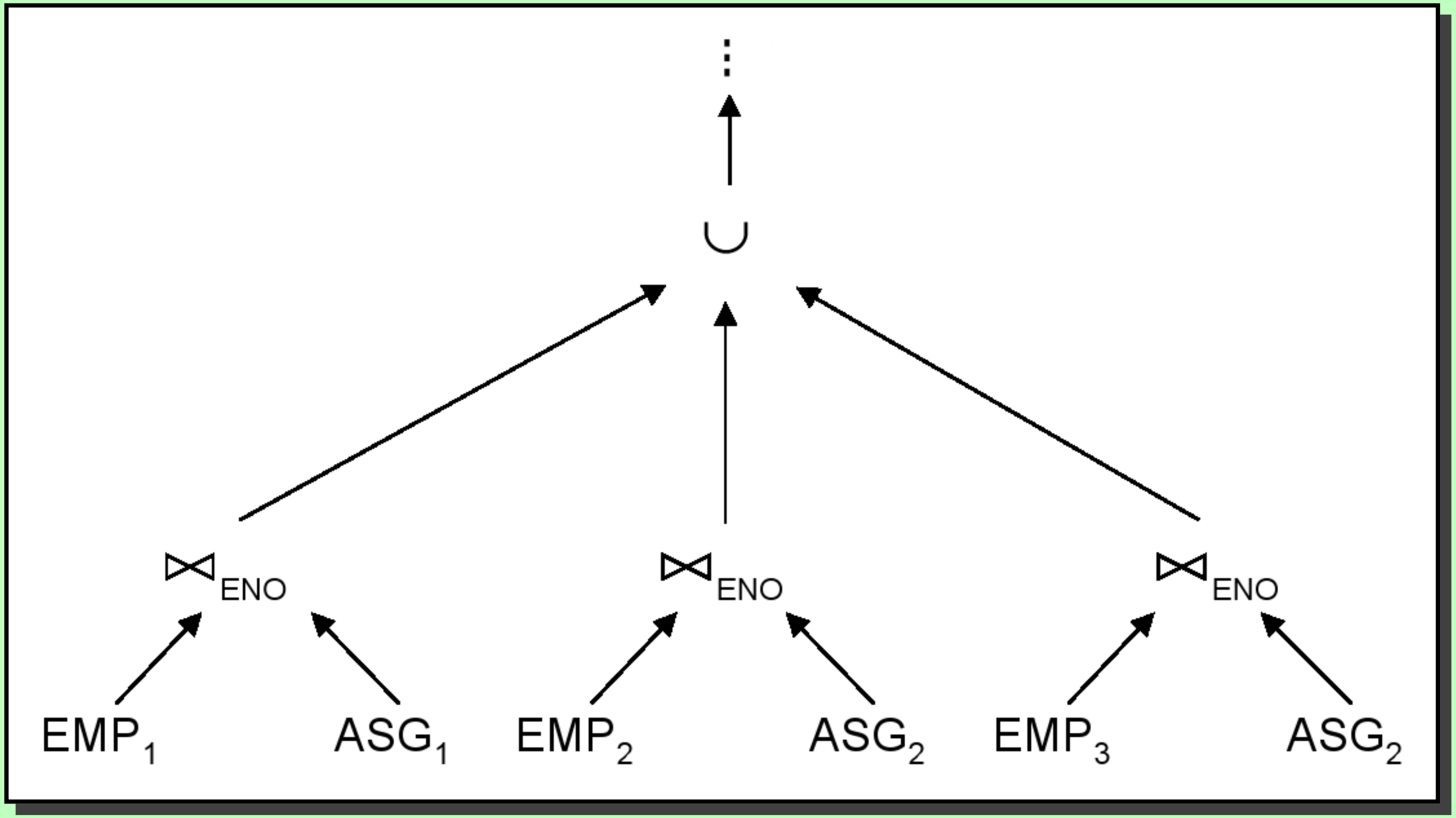
Data Localization . . .

- **Example (contd.)**: Parallelsim in the evaluation is often possible
  - Depending on the horizontal fragmentation, the fragments can be joined in parallel followed by the union of the intermediate results.



Data Localization . . .

- **Example (contd.)**: Unnecessary work can be eliminated
  - e.g., $EMP_3 \bowtie ASG_1$ gives an empty result
    - $EMP3 = \sigma_{ENO>"E6"}(EMP)$
    - $ASG1 = \sigma_{ENO\leq"E3"}(ASG)$



Data Localizations Issues

- Various more advanced **reduction techniques** are possible to generate simpler and optimized queries.
- Reduction of horizontal fragmentation (HF)
  - Reduction with selection
  - Reduction with join
- Reduction of vertical fragmentation (VF)
  - Find empty relations

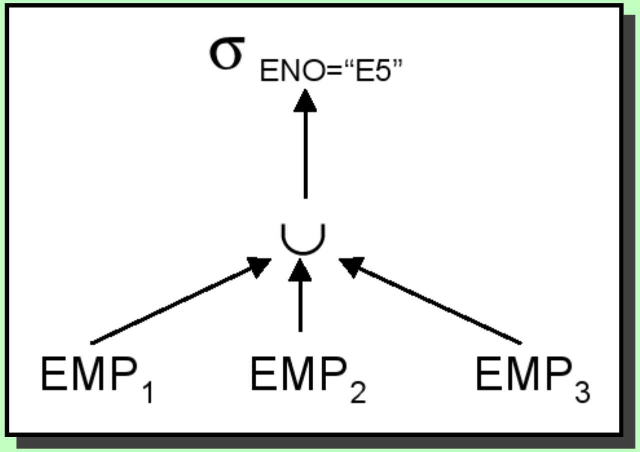
Data Localizations Issues – Reduction of HF

- **Reduction with selection for HF**
  - Consider relation R with horizontal fragmentation $F = \{R_1, R_2, \ldots, R_k\}$, where $R_i = \sigma_{p_i}(R)$
  - **Rule1:** Selections on fragments, $\sigma_{p_j}(R_i)$, that have a qualification contradicting the qualification of the fragmentation generate empty relations, i.e.,

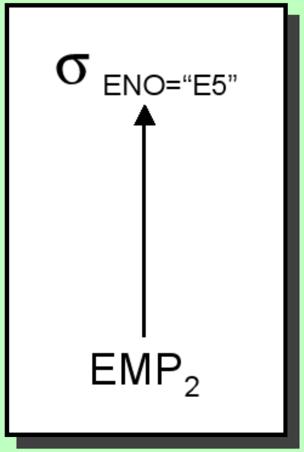
$$\sigma_{p_j}(R_i) = \emptyset \iff \forall x \in R(p_i(x) \wedge p_j(x) = false)$$

  - Can be applied if fragmentation predicate is inconsistent with the query selection predicate.

- **Example:** Consider the query: **SELECT** * **FROM** EMP **WHERE** ENO=”E5”



After commuting the selection with the union operation, it is easy to detect that the selection predicate contradicts the predicates of EMP$_1$ and EMP$_3$.

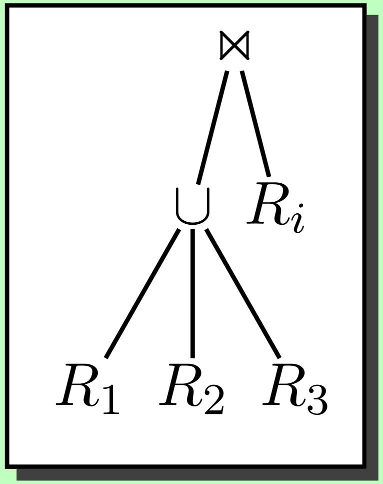


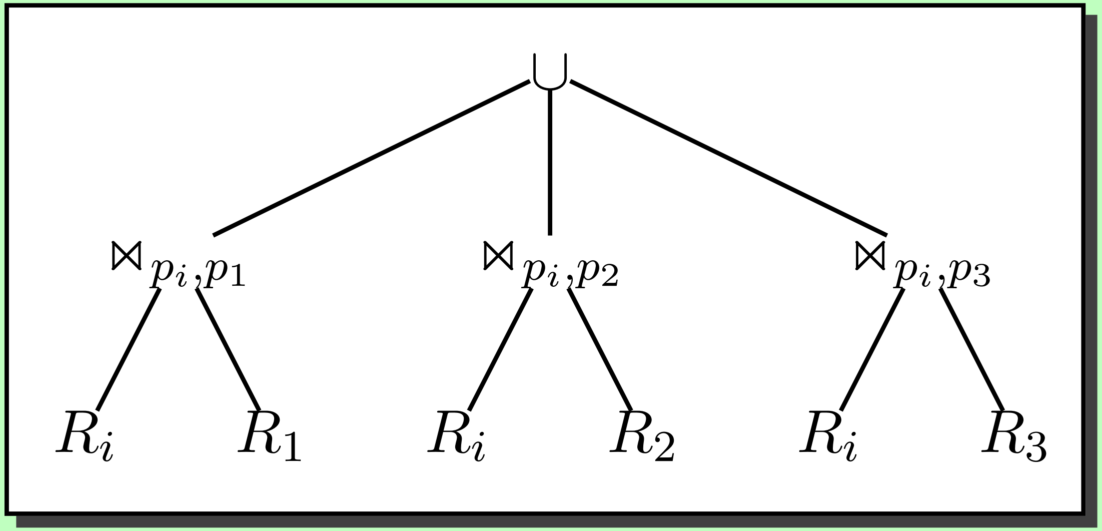
Data Localizations Issues – Reduction for HF . . .

- **Reduction with join for HF**
  - Joins on horizontally fragmented relations can be simplified when the joined relations are fragmented according to the join attributes.
  - Distribute join over union

$$(R_1 \cup R_2) \bowtie S \iff (R_1 \bowtie S) \cup (R_2 \bowtie S)$$

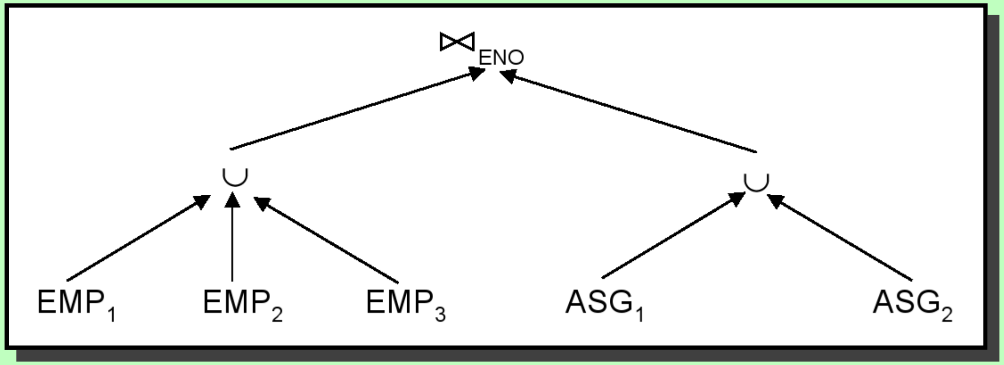
  - **Rule 2**: Useless joins of fragments, $R_i = \sigma_{p_i}(R)$ and $R_j = \sigma_{p_j}(R)$, can be determined when the qualifications of the joined fragments are contradicting.



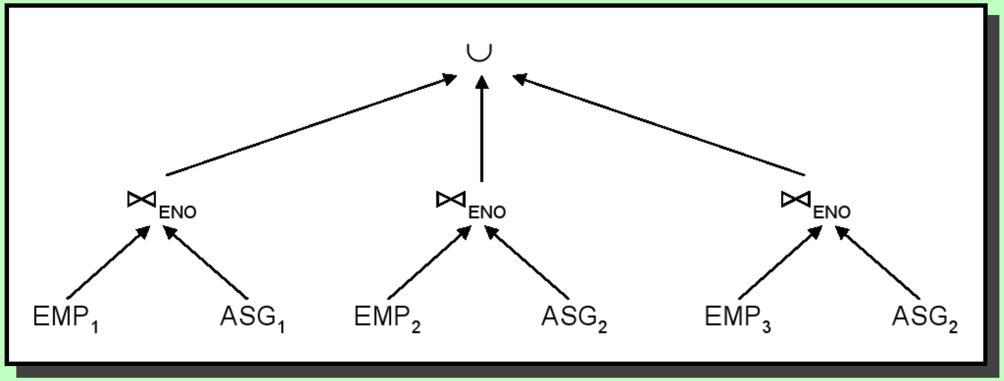


Data Localizations Issues – Reduction for HF . . .

- **Example:** Consider the following query and fragmentation:
  - Query: **SELECT** * **FROM** EMP, ASG **WHERE** EMP.ENO=ASG.ENO
  - Horizontal fragmentation:
    - ∗ $EMP1 = \sigma_{ENO \leq "E3"}(EMP)$
    - ∗ $EMP2 = \sigma_{"E3" < ENO \leq "E6"}(EMP)$
    - ∗ $EMP3 = \sigma_{ENO > "E6"}(EMP)$
    - ∗ $ASG1 = \sigma_{ENO \leq "E3"}(ASG)$
    - ∗ $ASG2 = \sigma_{ENO > "E3"}(ASG)$
  - Generic query



  - The query reduced by distributing joins over unions and applying rule 2 can be implemented as a union of three partial joins that can be done in parallel.



Data Localizations Issues – Reduction for HF . . .

- **Reduction with join for derived HF**
  - The horizontal fragmentation of one relation is **derived** from the horizontal fragmentation of another relation by using semijoins.
- If the fragmentation is not on the same predicate as the join (as in the previous example), derived horizontal fragmentation can be applied in order to make efficient join processing possible.
- **Example:** Assume the following query and fragmentation of the EMP relation:
  - Query: **SELECT** * **FROM** EMP, ASG **WHERE** EMP.ENO=ASG.ENO
  - Fragmentation (**not** on the join attribute):
    - ∗ EMP1 = $\sigma_{\text{TITLE=“Prgrammer”}}(\text{EMP})$
    - ∗ EMP2 = $\sigma_{\text{TITLE}\neq\text{“Prgrammer”}}(\text{EMP})$
  - To achieve efficient joins ASG can be fragmented as follows:
    - ∗ ASG1= ASG$\rtimes_{ENO}$EMP1
    - ∗ ASG2= ASG$\rtimes_{ENO}$EMP2
  - The fragmentation of ASG is derived from the fragmentation of EMP
  - Queries on derived fragments can be reduced, e.g., $ASG_1 \bowtie EMP_2 = \emptyset$

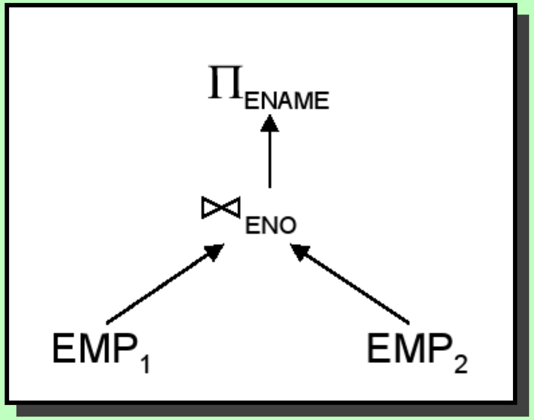
Data Localizations Issues – Reduction for VF

- **Reduction for Vertical Fragmentation**
  - Recall, VF distributes a relation based on projection, and the reconstruction operator is the join.
  - Similar to HF, it is possible to identify useless intermediate relations, i.e., fragments that do not contribute to the result.
  - Assume a relation $R(A)$ with $A = \quad A \;, \ldots, A \quad$, which is vertically fragmented as $R_i = \pi_{A'_i}(R)$, where $A'_i \subseteq A$.
  - **Rule 3**: $\pi_{D,K}(R_i)$ is useless if the set of projection attributes D is not in A'_i and K is the key attribute.
  - Note that the result is not empty, but it is useless, as it contains only the key attribute.

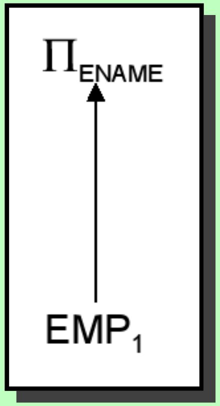
Data Localizations Issues – Reduction for VF . . .

- **Example:** Consider the following query and vertical fragmentation:
  - Query: **SELECT** ENAME **FROM** EMP
  - Fragmentation:
    - $EMP1 = \Pi_{ENO,ENAME}(EMP)$
    - $EMP2 = \Pi_{ENO,TITLE}(EMP)$

- Generic query



- Reduced query
  - By commuting the projection with the join (i.e., projecting on ENO, ENAME), we can see that the projection on EMP$_2$ is useless because ENAME is not in EMP$_2$.



Conclusion

- Query decomposition and data localization maps calculus query into algebra operations and applies data distribution information to the algebra operations.
- Query decomposition consists of normalization, analysis, elimination of redundancy, and rewriting.
- Data localization reduces horizontal fragmentation with join and selection, and vertical fragmentation with joins, and aims to find empty relations.

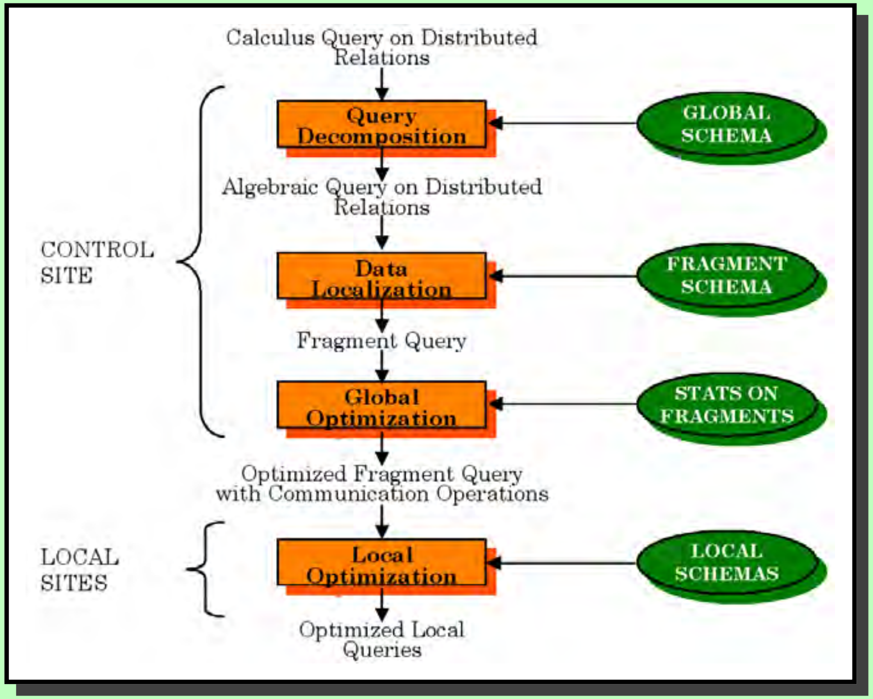
Chapter 7: Optimization of Distributed Queries

- Basic Concepts
- Distributed Cost Model
- Database Statistics
- Joins and Semijoins
- Query Optimization Algorithms

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Basic Concepts

- **Query optimization:** Process of producing an optimal (close to optimal) query execution plan which represents an execution strategy for the query
  - The main task in query optimization is to consider different orderings of the operations
- Centralized query optimization:
  - Find (the best) query execution plan in the space of equivalent query trees
  - Minimize an objective cost function
  - Gather statistics about relations

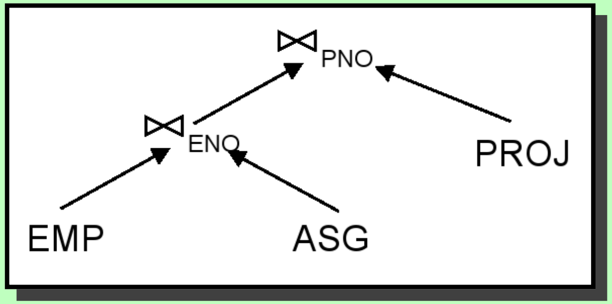


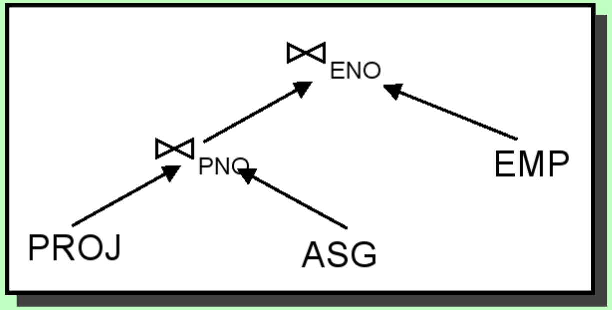
- Distributed query optimization brings additional issues
  - Linear query trees are not necessarily a good choice
  - Bushy query trees are not necessarily a bad choice
  - What and where to ship the relations
  - How to ship relations (ship as a whole, ship as needed)
  - When to use semi-joins instead of joins

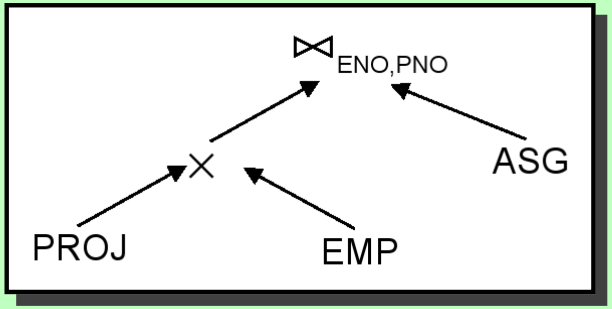
Basic Concepts . . .

- **Search space:** The set of alternative query execution plans (query trees)
  - Typically very large
  - The main issue is to optimize the joins
  - For N relations, there are $O(N!)$ equivalent join trees that can be obtained by applying commutativity and associativity rules
- **Example**: 3 equivalent query trees (join trees) of the joins in the following query

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

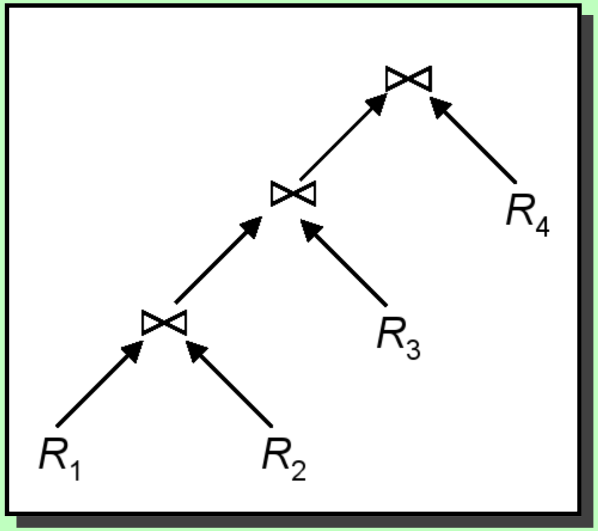




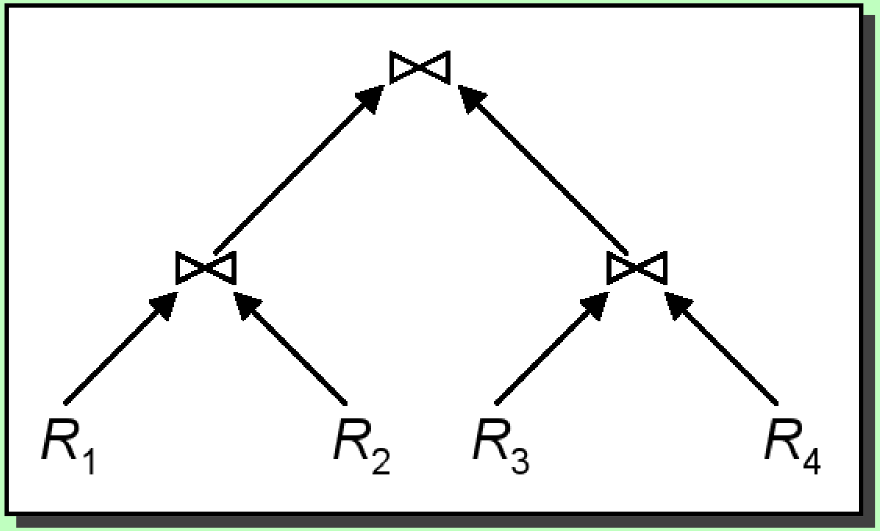


Basic Concepts . . .

- **Reduction** of the search space
  - Restrict by means of heuristics
    - ∗ Perform unary operations before binary operations, etc
  - Restrict the shape of the join tree
    - ∗ Consider the type of trees (linear trees, vs. bushy ones)



Linear Join Tree

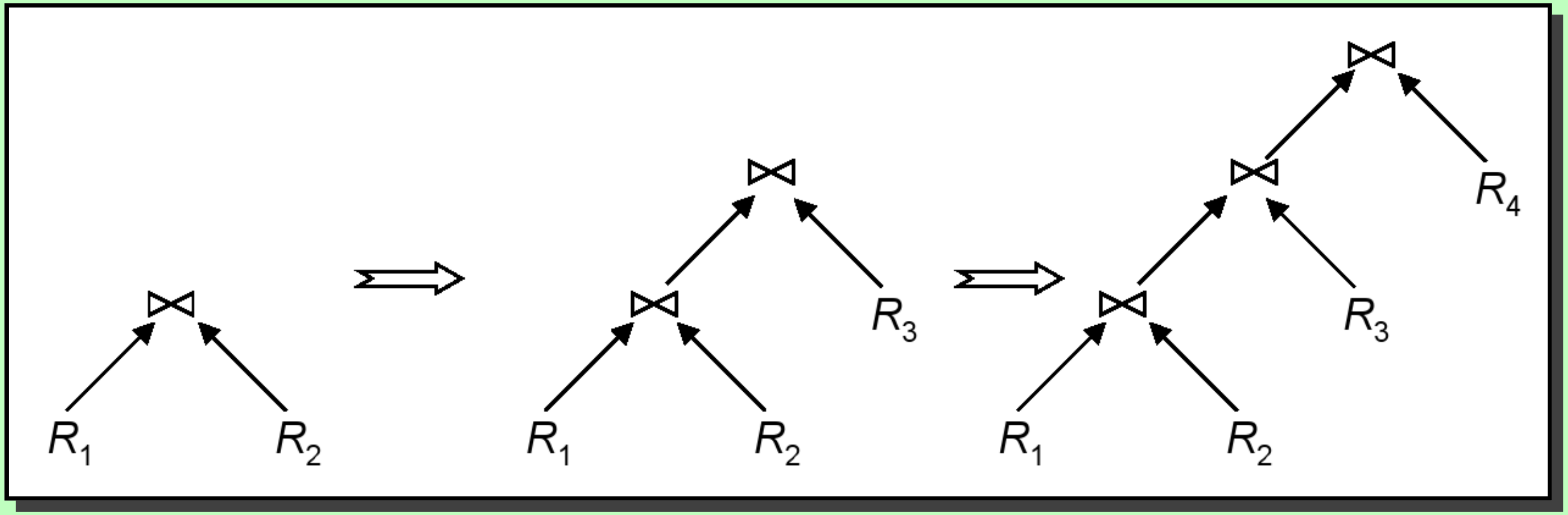


Bushy Join Tree

Basic Concepts . . .

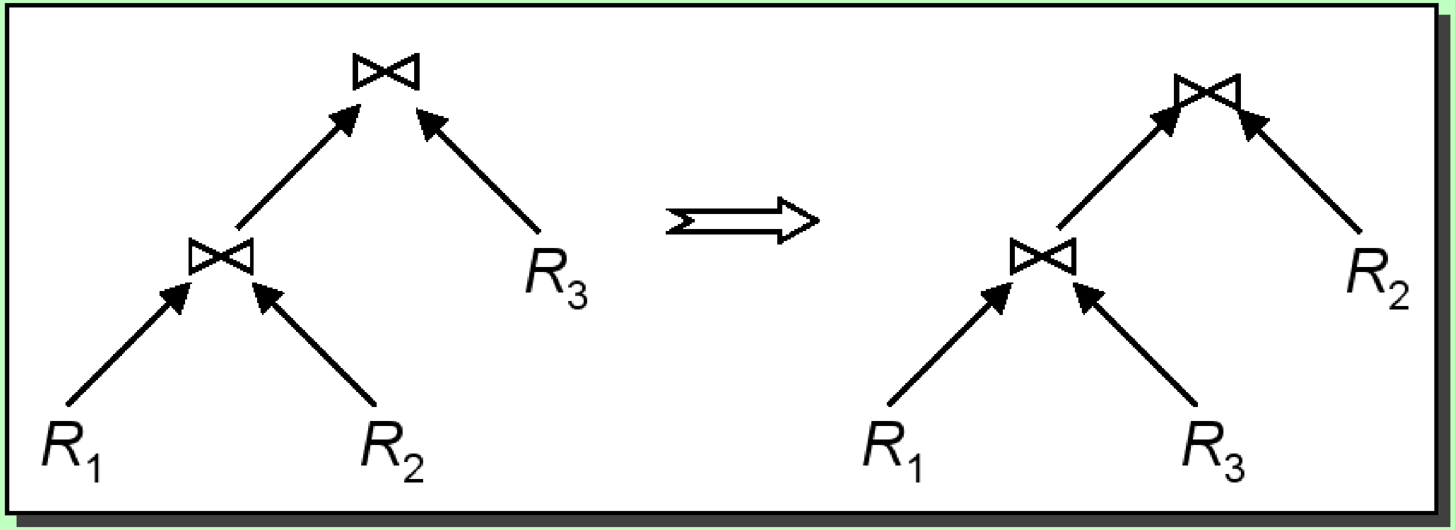
- There are two main strategies to **scan the search space**
  - Deterministic
  - Randomized

- **Deterministic scan** of the search space
  - Start from base relations and build plans by adding one relation at each step
  - Breadth-first strategy: build all possible plans before choosing the "best" plan (dynamic programming approach)
  - Depth-first strategy: build only one plan (greedy approach)



Basic Concepts . . .

- **Randomized scan** of the search space
  - Search for optimal solutions around a particular starting point
  - e.g., iterative improvement or simulated annealing techniques
  - Trades optimization time for execution time
    - ∗ Does not guarantee that the best solution is obtained, but avoid the high cost of optimization
  - The strategy is better when more than 5-6 relations are involved



Distributed Cost Model

- Two different types of **cost functions** can be used
  - Reduce **total time**
    - Reduce each cost component (in terms of time) individually, i.e., do as little for each cost component as possible
    - Optimize the utilization of the resources (i.e., increase system throughput)
  - Reduce **response time**
    - Do as many things in parallel as possible
    - May increase total time because of increased total activity

Distributed Cost Model . . .

- **Total time**: Sum of the time of all individual components
  - Local processing time: CPU time + I/O time
  - Communication time: fixed time to initiate a message + time to transmit the data

$$\begin{aligned} Total\_time = & T_{CPU} * \#instructions + T_{I/O} * \#I/Os + \\ & T_{MSG} * \#messages + T_{TR} * \#bytes \end{aligned}$$

- The individual components of the total cost have different weights:
  - Wide area network
    - $*$ Message initiation and transmission costs are high
    - $*$ Local processing cost is low (fast mainframes or minicomputers)
    - $*$ Ratio of communication to I/O costs is 20:1
  - Local area networks
    - $*$ Communication and local processing costs are more or less equal
    - $*$ Ratio of communication to I/O costs is 1:1.6 (10MB/s network)

Distributed Cost Model . . .

- **Response time**: Elapsed time between the initiation and the completion of a query

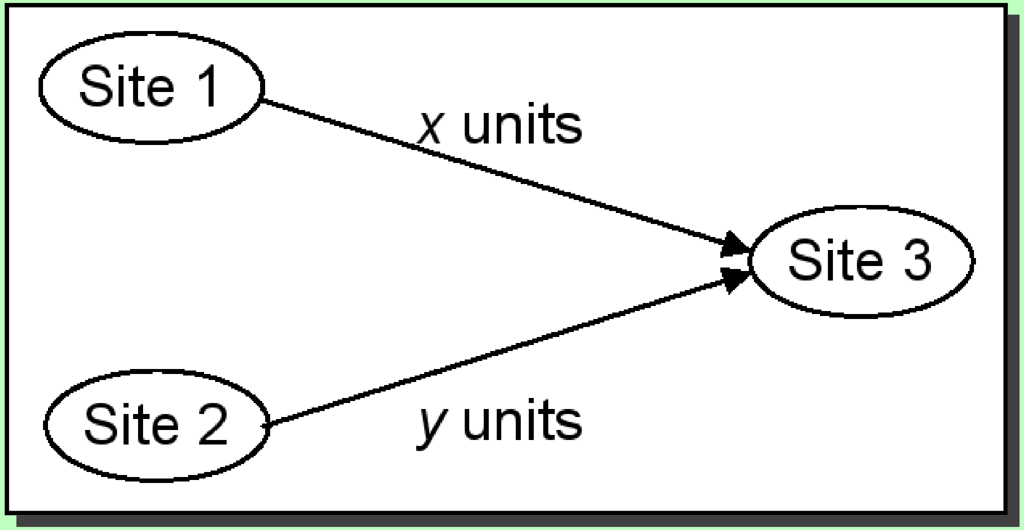
$$\begin{aligned} Response\_time = & T_{CPU} * \#seq\_instructions + T_{I/O} * \#seq\_I/Os + \\ & T\mathrm{msg} * \#seq\ messages + T\mathrm{cpu} * \#seq\ bytes \end{aligned}$$

  – where $\#seq\_x$ (x in instructions, I/O, messages, bytes) is the **maximum number** of x which must be done sequentially.

- Any processing and communication done in parallel is ignored

Distributed Cost Model . . .

- **Example:** Query at site 3 with data from sites 1 and 2.



  - Assume that only the communication cost is considered
  - $Total\_time = T_{MSG} * 2 + T_{TR} * (x + y)$
  - $Response\_time = \max\{T_{MSG} + T_{TR} * x, T_{MSG} + T_{TR} * y\}$

Database Statistics

- The **primary cost factor** is the **size of intermediate relations**
  - that are produced during the execution and
  - must be transmitted over the network, if a subsequent operation is located on a different site
- It is costly to compute the size of the intermediate relations precisely.
- Instead **global statistics of relations and fragments** are computed and used to provide approximations

Database Statistics . . .

- Let $R(A_1, A_2, \ldots, A_k)$ be a relation fragmented into $R_1, R_2, \ldots, R_r$.
- **Relation statistics**
  - min and max values of each attribute: $\min\{A_i\}$, $\max\{A_i\}$.
  - length of each attribute: $length(A_i)$
  - number of distinct values in each fragment (cardinality): $card(A\ \ )$, ($card(dom(A_i))$)
- **Fragment statistics**
  - cardinality of the fragment: $card(R_i)$
  - cardinality of each attribute of each fragment: $card(\Pi_{A_i}(R_j))$

Database Statistics . . .

- **Selectivity factor** of an operation: the proportion of tuples of an operand relation that participate in the result of that operation
- Assumption: independent attributes and uniform distribution of attribute values
- **Selectivity factor of selection**

$$SF_{\sigma}(A = value) = \frac{1}{card(\Pi_A(R))}$$

$$\frac{\max(A) \quad value}{\max(A) \quad \min(A)}$$

$$SF_{\sigma}(A < value) = \frac{value - \min(A)}{\max(A) - \min(A)}$$

- Properties of the selectivity factor of the selection

$$SF_{\sigma}(p(A_i) \wedge p(A_j)) = SF_{\sigma}(p(A_i)) * SF_{\sigma}(p(A_j))$$
$$SF_{\sigma}(p(A_i) \vee p(A_j)) = SF_{\sigma}(p(A_i)) + SF_{\sigma}(p(A_j)) - (SF_{\sigma}(p(A_i)) * SF_{\sigma}(p(A_j))$$
$$SF_{\sigma}(A \in \{values\}) = SF_{\sigma}(A = value) * card(\{values\})$$

Database Statistics . . .

- **Cardinality** of intermediate results
  - Selection

$$card(\sigma_P(R)) = SF_\sigma(P) * card(R)$$

  - Projection
    - More difficult: duplicates, correlations between projected attributes are unknown
    - Simple if the projected attribute is a key

$$card(\Pi_A(R)) = card(R)$$

  - Cartesian Product

$$card(R \times S) = card(R) * card(S)$$

  - Union
    - upper bound: $card(R \cup S) \leq card(R) + card(S)$
    - lower bound: $card(R \cup S) \geq \max\{card(R), card(S)\}$
  - Set Difference
    - upper bound: $card(R - S) = card(R)$
    - lower bound: 0

Database Statistics . . .

- **Selectivity factor** for joins

$$SF_{\bowtie} = \frac{card(R \bowtie S)}{card(R) * card(S)}$$

- **Cardinality** of joins
  - Upper bound: cardinality of Cartesian Product
    $card(R \quad\quad S) \quad\quad card(R) \quad card(S)$
  - General case (if SF is given):

$$card(R \bowtie S) = SF_{\bowtie} * card(R) * card(S)$$

  - Special case: $R.A$ is a key of R and $S.A$ is a foreign key of S;
    - $*$ each S-tuple matches with at most one tuple of R

$$card(R \bowtie_{R.A=S.A} S) = card(S)$$

Database Statistics . . .

- **Selectivity factor** for semijoins: fraction of R-tuples that join with S-tuples
  - An approximation is the selectivity of A in S

$$SF_{\rtimes}(R \rtimes_A S) = SF_{\rtimes}(S.A) = \frac{card(\Pi_A(S))}{card(dom[A])}$$

- **Cardinality** of semijoin (general case):

$$card(R \rtimes_A S) = SF_{\rtimes}(S.A) * card(R)$$

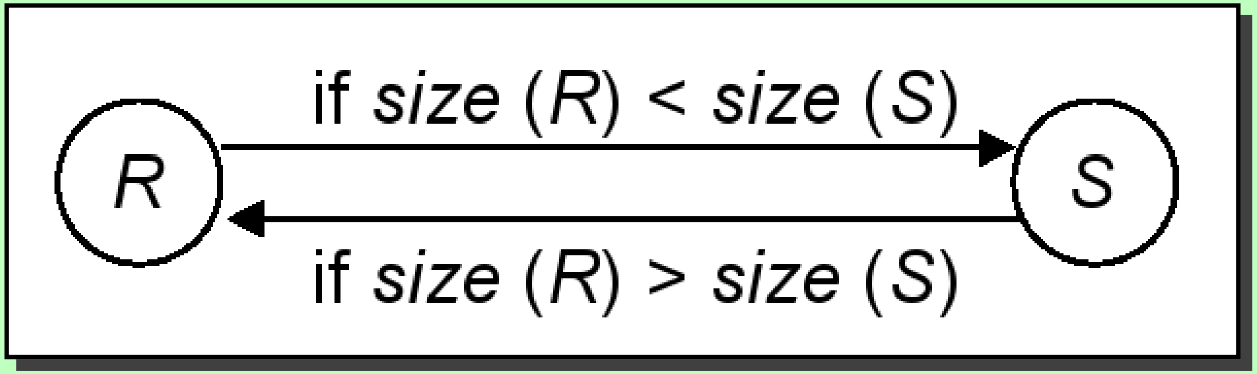
- Example: $R.A$ is a foreign key in S ($S.A$ is a primary key)
  Then $SF = 1$ and the result size corresponds to the size of R

Join Ordering in Fragment Queries

- **Join ordering** is an important aspect in centralized DBMS, and it **is even more important in a DDBMS** since joins between fragments that are stored at different sites may increase the communication time.
- Two approaches exist:
  - Optimize the ordering of joins directly
    - INGRES and distributed INGRES
    - System R and System R^*
  - Replace joins by combinations of semijoins in order to minimize the communication costs
    - Hill Climbing and SDD-1

Join Ordering in Fragment Queries . . .

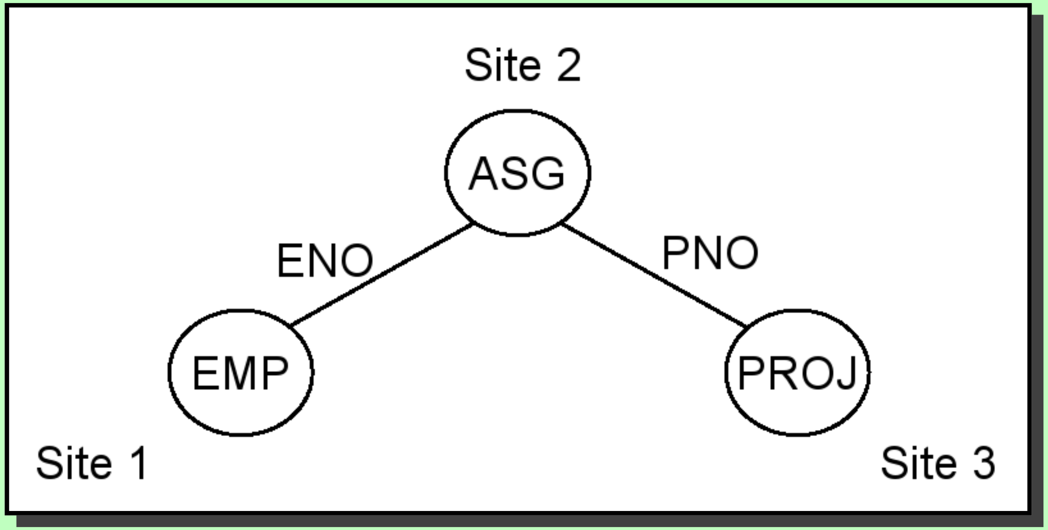
- **Direct join odering** of two relation/fragments located at different sites
  - Move the smaller relation to the other site
  - We have to estimate the size of R and S


if *size* (*R*) < *size* (*S*)
R
S
if *size* (*R*) > *size* (*S*)

Join Ordering in Fragment Queries . . .

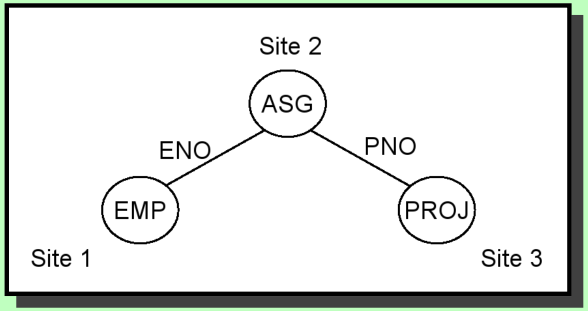
- **Direct join ordering** of queries involving more than two relations is substantially more complex
- **Example:** Consider the following query and the respective join graph, where we make also assumptions about the locations of the three relations/fragments

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



Join Ordering in Fragment Queries . . .

- **Example (contd.):** The query can be evaluated in at least 5 different ways.
  - Plan 1: EMP⟶Site 2
    Site 2: EMP'=EMP⋈ASG
    EMP'⟶Site 3
    Site 3: EMP'⋈PROJ
  - Plan 2: ASG⟶Site 1
    Site 1: EMP'=EMP⋈ASG
    EMP'   Site 3
    Site 3: EMP'   PROJ
  - Plan 3: ASG⟶Site 3
    Site 3: ASG'=ASG⋈PROJ
    ASG'⟶Site 1
    Site 1: ASG'⋈EMP



  - Plan 4: PROJ⟶Site 2
    Site 2: PROJ'=PROJ⋈ASG
    PROJ'   Site 1
    Site 1: PROJ'⋈EMP
  - Plan 5: EMP⟶Site 2
    PROJ⟶Site 2
    Site 2: EMP⋈PROJ⋈ASG

- To select a plan, a lot of information is needed, including
  - $size(EMP)$, $size(ASG)$, $size(PROJ)$, $size(EMP \bowtie ASG)$, $size(ASG \bowtie PROJ)$
  - Possibilities of parallel execution if response time is used

Semijoin Based Algorithms

- **Semijoins** can be used to efficiently implement joins
  - The semijoin acts as a size reducer (similar as to a selection) such that smaller relations need to be transferred
- Consider two relations: R located at site 1 and S located and site 2
  - Solution with semijoins: Replace one or both operand relations/fragments by a semijoin, using the following rules:

$$\begin{aligned} R \bowtie_A S &\iff (R \rtimes_A S) \bowtie_A S \\ &\iff R \bowtie_A (S \rtimes_A R) \\ &\iff (R \rtimes_A S) \bowtie_A (S \rtimes_A R) \end{aligned}$$

- The semijoin is beneficial if the cost to produce and send it to the other site is less than the cost of sending the whole operand relation and of doing the actual join.

Semijoin Based Algorithms

- **Cost analysis** $R \bowtie_A S$ vs. $(R \ltimes_A S) \bowtie S$, assuming that $size(R) < size(S)$
  - Perform the join $R \bowtie S$:
    - $*$ $R \longrightarrow$ Site 2
    - $*$ Site 2 computes $R \bowtie S$
  - Perform the semijoins $(R \ltimes S) \bowtie S$:
    - $*$ $S' = \Pi\ \ (S)$
    - $*$ S'= Site 1
    - $*$ Site 1 computes $R\ = R\ \ < S$
    - $*$ $R' \longrightarrow$Site 2
    - $*$ Site 2 computes $R' \bowtie S$
  - Semijoin is better if: $size(\Pi_A(S)) + size(R \ltimes S) < size(R)$
- The **semijoin** approach is better if the semijoin acts as a **sufficient reducer** (i.e., a few tuples of R participate in the join)
- The **join** approach is better if **almost all tuples of** R **participate** in the join

INGRES Algorithm

- **INGRES** uses a dynamic query optimization algorithm that recursively breaks a query into smaller pieces. It is based on the following ideas:
  - An n-relation query q is decomposed into n subqueries q1 ! q2 ! · · · ! qn □ Each qi is a mono-relation (mono-variable) query
    - ∗ The output of q_i is consumed by q_{i+1}
  - For the decomposition two basic techniques are used: **detachment** and **substitution**
  - There’s a processor that can **efficiently** process mono-relation queries
    - ∗ Optimizes each query independently for the access to a single relation

INGRES Algorithm . . .

- **Detachment:** Break a query q into $q' \rightarrow q''$, based on a common relation that is the result of q', i.e.
  - The query

    q: **SELECT** $R_2.A_2, \ldots, R_n.A_n$
    FROM $R_1, R_2, \ldots, R_n$
    WHERE $P_1(R_1.A'_1)$
    AND $P\ (R\ \ .A\ \ , \ldots, R\ \ .A\ \ )$

  - is decomposed by detachment of the common relation R_1 into

    q': **SELECT** $R_1.A_1$ **INTO** R'_1
    FROM R_1
    WHERE $P_1(R_1.A'_1)$

    q'': **SELECT** $R_2.A_2, \ldots, R_n.A_n$
    FROM $R'_1, R_2, \ldots, R_n$
    WHERE $P_2(R'_1.A_1, \ldots, R_n.A_n)$

- Detachment **reduces the size** of the relation on which the query q'' is defined.

INGRES Algorithm . . .

- **Example:** Consider query $q1$: “Names of employees working on the CAD/CAM project”

q_1:

```
SELECT   EMP.ENAME
FROM     EMP, ASG, PROJ
WHERE    EMP.ENO = ASG.ENO
AND      ASG.PNO = PROJ.PNO
AND      PROJ.PNAME = ”CAD/CAM”
```

- Decompose q_1 into $q_{11} \rightarrow q'$:

q_{11}:

```
SELECT   PROJ.PNO INTO JVAR
FROM     PROJ
WHERE    PROJ.PNAME = ”CAD/CAM”
```

q':

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

INGRES Algorithm . . .

- **Example (contd.):** The successive detachments may transform q' into $q_{12} \rightarrow q_{13}$:

q':

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

q_{12}:

```
SELECT    ASG.ENO INTO GVAR
FROM      ASG,JVAR
WHERE     ASG.PNO=JVAR.PNO
```

q_{13}:

```
SELECT    EMP.ENAME
FROM      EMP,GVAR
WHERE     EMP.ENO=GVAR.ENO
```

- q_1 is now decomposed by detachment into $q_{11} \rightarrow q_{12} \rightarrow q_{13}$
- q_{11} is a mono-relation query
- q_{12} and q_{13} are multi-relation queries, which cannot be further detached.
  - also called **irreducible**

INGRES Algorithm . . .

- **Tuple substitution** allows to convert an irreducible query q into mono-relation queries.
  - Choose a relation R_1 in q for tuple substitution
  - For each tuple in R_1, replace the R_1-attributes referred in q by their actual values, thereby generating a set of subqueries q' with $n-1$ relations, i.e.,

$$q(R_1, R_2, \ldots, R_n) \text{ is replaced by } \{q'(t_{1_i}, R_2, \ldots, R_n), t_{1_i} \in R_1\}$$

- **Example (contd.):** Assume $GVAR$ consists only of the tuples $E1, E2$. Then q_{13} is rewritten with tuple substitution in the following way

q_{13}:

```
SELECT   EMP.ENAME
FROM     EMP, GVAR
WHERE    EMP.ENO = GVAR.ENO
```

q_{131}:

```
SELECT   EMP.ENAME
FROM     EMP
WHERE    EMP.ENO = ”E1”
```

q_{132}:

```
SELECT   EMP.ENAME
FROM     EMP
WHERE    EMP.ENO = ”E2”
```

  - q_{131} and q_{132} are mono-relation queries

Distributed INGRES Algorithm

- The **distributed INGRES query optimization algorithm** is very similar to the centralized INGRES algorithm.
  - In addition to the centralized INGRES, the distributed one should break up each query q_i into sub-queries that operate on fragments; only horizontal fragmentation is handled.
  - Optimization with respect to a combination of communication cost and response time

System R Algorithm

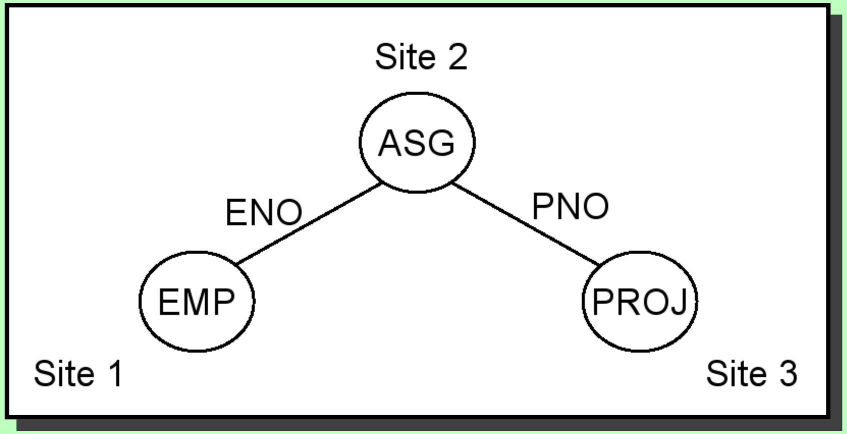
- The **System R** (centralized) query optimization algorithm
  - Performs static query optimization based on “exhaustive search” of the solution space and a cost function (IO cost + CPU cost)
    - ∗ Input: relational algebra tree
    - ∗ Output: optimal relational algebra tree
    - ∗ Dynamic programming technique is applied to reduce the number of alternative plans
  - The **optimization algorithm** consists of two steps
    1. Predict the best access method to each individual relation (mono-relation query)
       - ∗ Consider using index, file scan, etc.
    2. For each relation R, estimate the best join ordering
       - ∗ R is first accessed using its best single-relation access method
       - ∗ Efficient access to inner relation is crucial
  - Considers two different join strategies
    - ∗ (Indexed-) nested loop join
    - ∗ Sort-merge join

System R Algorithm . . .

- **Example:** Consider query $q1$: "*Names of employees working on the CAD/CAM project*"

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

  - Join graph



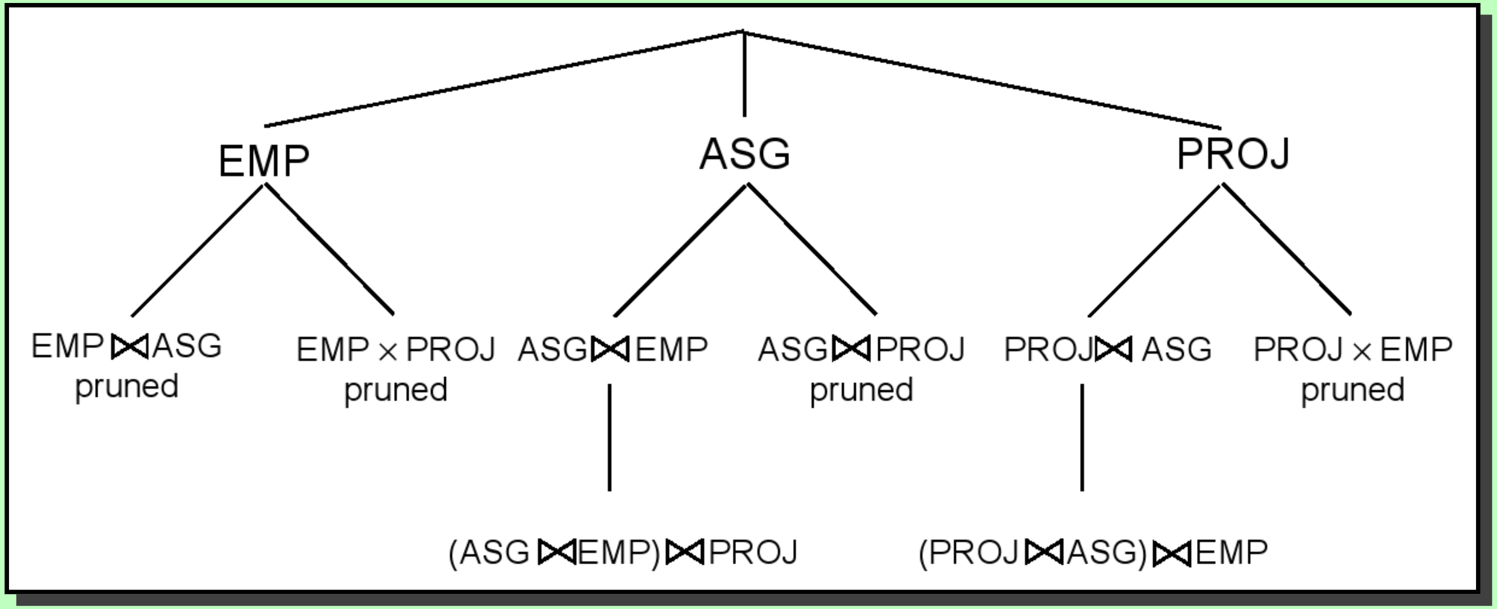
  - Indexes
    - ∗ EMP has an index on ENO
    - ∗ ASG has an index on PNO
    - ∗ PROJ has an index on PNO and an index on PNAME

System R Algorithm . . .

- **Example (contd.):** Step 1 – Select the best single-relation access paths
  - EMP: sequential scan (because there is no selection on EMP)
  - ASG: sequential scan (because there is no selection on ASG)
  - PROJ: index on PNAME (because there is a selection on PROJ based on PNAME)

System R Algorithm . . .

- **Example (contd.):** Step 2 – Select the best join ordering for each relation



  - (EMP × PROJ) and (PROJ × EMP) are pruned because they are CPs
  - (ASG × PROJ) pruned because we assume it has higher cost than (PROJ × ASG); similar for (PROJ × EMP)
  - Best total join order ((PROJ ⋈ ASG) ⋈ EMP), since it uses the indexes best
    - ∗ Select PROJ using index on PNAME
    - ∗ Join with ASG using index on PNO
    - ∗ Join with EMP using index on ENO

Distributed System R^* Algorithm

- The **System R^* query optimization** algorithm is an extension of the System R query optimization algorithm with the following main characteristics:
  - Only the whole relations can be distributed, i.e., fragmentation and replication is not considered
  - Query compilation is a distributed task, coordinated by a **master site**, where the query is initiated
  - Master site makes all inter-site decisions, e.g., selection of the execution sites, join ordering, method of data transfer, ...
  - The **local sites** do the intra-site (local) optimizations, e.g., local joins, access paths
- Join ordering and data transfer between different sites are the most critical issues to be considered by the master site

Distributed System R^* Algorithm . . .

- Two methods for **inter-site data transfer**
  - **Ship whole:** The entire relation is shipped to the join site and stored in a temporary relation
    - $*$ Larger data transfer
    - $*$ Smaller number of messages
    - $*$ Better if relations are small
  - **Fetch as needed:** The external relation is sequentially scanned, and for each tuple the join value is sent to the site of the inner relation and the matching inner tuples are sent back (i.e., semijoin)
    - $*$ Number of messages = O(cardinality of outer relation)
    - $*$ Data transfer per message is minimal
    - $*$ Better if relations are large and the selectivity is good

Distributed System R^* Algorithm . . .

- Four main **join strategies** for $R \bowtie S$:
  - R is outer relation
  - S is inner relation
- Notation:
  - LT denotes local processing time
  - CT denotes communication time
  - s denotes the average number of S-tuples that match an R-tuple
- **Strategy 1:** Ship the entire outer relation to the site of the inner relation, i.e.,
  - Retrieve outer tuples
  - Send them to the inner relation site
  - Join them as they arrive

$$\begin{aligned} Total\_cost = {} & LT(\text{retrieve } card(R) \text{ tuples from } R) + \\ & CT(size(R)) + \\ & LT(\text{retrieve } s \text{ tuples from } S) * card(R) \end{aligned}$$

Distributed System R^* Algorithm . . .

- **Strategy 2:** Ship the entire inner relation to the site of the outer relation. We cannot join as they arrive; they need to be stored.
  - The inner relation S need to be stored in a temporary relation

$$
\begin{aligned}
Total\_cost = {} & LT(\text{retrieve } card(S) \text{ tuples from } S) + \\
& CT(size(S)) + \\
& LT(\text{store } card(S) \text{ tuples in } T) + \\
& LT(\text{retrieve } card(R) \text{ tuples from } R) + \\
& LT(\text{retrieve } s \text{ tuples from } T) * card(R)
\end{aligned}
$$

Distributed System R^* Algorithm . . .

- **Strategy 3:** Fetch tuples of the inner relation as needed for each tuple of the outer relation.
  - For each R-tuple, the join attribute A is sent to the site of S
  - The s matching S-tuples are retrieved and sent to the site of R

$$\begin{aligned} Total\_cost = {} & LT(\text{retrieve } card(R) \text{ tuples from } R) + \\ & CT(length(A)) * card(R) + \\ & LT(\text{retrieve } s \text{ tuples from } S) * card(R) + \\ & CT(s * length(S)) * card(R) \end{aligned}$$

Distributed System R^* Algorithm . . .

- **Strategy 4:** Move both relations to a third site and compute the join there.
  - The inner relation S is first moved to a third site and stored in a temporary relation.
  - Then the outer relation is moved to the third site and its tuples are joined as they arrive.

$$\begin{aligned}
Total\_cost = {} & LT(\text{retrieve } card(S) \text{ tuples from } S) + \\
& CT(size(S)) + \\
& LT(\text{store } card(S) \text{ tuples in } T) + \\
& LT(\text{retrieve } card(R) \text{ tuples from } R) + \\
& CT(size(R)) + \\
& LT(\text{retrieve } s \text{ tuples from } T) * card(R)
\end{aligned}$$

Hill-Climbing Algorithm

- **Hill-Climbing query optimization** algorithm
  - Refinements of an initial feasible solution are recursively computed until no more cost improvements can be made
  - Semijoins, data replication, and fragmentation are not used
  - Devised for wide area point-to-point networks
  - The first distributed query processing algorithm

Hill-Climbing Algorithm . . .

- The hill-climbing algorithm proceeds as follows

1. Select initial feasible execution strategy ES0
   - i.e., a global execution schedule that includes all intersite communication
   - Determine the candidate result sites, where a relation referenced in the query exist
   - Compute the cost of transferring all the other referenced relations to each candidate site
   - ES0 = candidate site with minimum cost
2. Split ES0 into two strategies: ES1 followed by ES2
   - ES1: send one of the relations involved in the join to the other relation's site
   - ES2: send the join result to the final result site
3. Replace ES0 with the split schedule which gives

$$cost(ES1) + cost(\text{local join}) + cost(ES2) < cost(ES0)$$

4. Recursively apply steps 2 and 3 on ES1 and ES2 until no more benefit can be gained
5. Check for redundant transmissions in the final plan and eliminate them

Hill-Climbing Algorithm . . .

- **Example:** *What are the salaries of engineers who work on the CAD/CAM project?*

$$\Pi_{SAL}(PAY \bowtie_{TITLE} EMP \bowtie_{ENO} (ASG \bowtie_{PNO} (\sigma_{PNAME=\text{``}CAD/CAM''}(PROJ))))$$

  - Schemas: EMP(ENO, ENBAME, TITLE), ASG(ENO, PNO, RESP, DUR), PROJ(PNO, PNAME, BUDGET, LOC), PAY(TITLE, SAL)
  - Statistics

Relation	Size	Site
EMP	8	1
PAY	4	2
PROJ	1	3
ASG	10	4

  - Assumptions:
    - ∗ Size of relations is defined as their cardinality
    - ∗ Minimize total cost
    - ∗ Transmission cost between two sites is 1
    - ∗ Ignore local processing cost
    - ∗ size(EMP $\bowtie$ PAY) = 8, size(PROJ $\bowtie$ ASG) = 2, size(ASG $\bowtie$ EMP) = 10

Hill-Climbing Algorithm . . .

- **Example (contd.):** Determine initial feasible execution strategy
  - Alternative 1: Resulting site is site 1

$$\begin{aligned} Total\_cost &= cost(\text{PAY} \rightarrow \text{Site}1) + cost(\text{ASG} \rightarrow \text{Site}1) + cost(\text{PROJ} \rightarrow \text{Site}1) \\ &= 4 + 10 + 1 = 15 \end{aligned}$$

  - Alternative 2: Resulting site is site 2

Total cost =8 + 10 + 1 = 19

  - Alternative 3: Resulting site is site 3

$$\text{Total cost} = 8 + 4 + 10 = 22$$

  - Alternative 4: Resulting site is site 4

$$\text{Total cost} = 8 + 4 + 1 = 13$$

  - Therefore ES0 = EMP→Site4; PAY → Site4; PROJ → Site4

Hill-Climbing Algorithm . . .

- **Example (contd.):** Candidate split
  - Alternative 1: ES1, ES2, ES3
    - ES1: EMP→Site 2
    - ES2: (EMP⋈PAY) → Site4
    - ES3: PROJ→Site 4

$$\begin{aligned} Total\_cost = &\ cost(\text{EMP} \longrightarrow \text{Site2}) + \\ &\ cost((\text{EMP} \bowtie \text{PAY}) \longrightarrow \text{Site4}) + \\ &\ cost(\text{PROJ} \longrightarrow \text{Site4}) \\ = &\ 8 + 8 + 1 = 17 \end{aligned}$$

  - Alternative 2: ES1, ES2, ES3
    - ES1: PAY → Site1
    - ES2: (PAY ⋈ EMP) → Site4
    - ES3: PROJ → Site 4

$$\begin{aligned} Total\_cost = &\ cost(\text{PAYSite} \longrightarrow 1) + \\ &\ cost((\text{PAY} \bowtie \text{EMP}) \longrightarrow \text{Site4}) + \\ &\ cost(\text{PROJ} \longrightarrow \text{Site4}) \\ = &\ 4 + 8 + 1 = 13 \end{aligned}$$

- Both alternatives are not better than ES0, so keep it (or take alternative 2 which has the same cost)

Hill-Climbing Algorithm . . .

- **Problems**
  - Greedy algorithm determines an initial feasible solution and iteratively improves it
  - If there are local minima, it may not find the global minimum
  - An optimal schedule with a high initial cost would not be found, since it won't be chosen as the initial feasible solution
- **Example:** A better schedule is
  - PROJ$\longrightarrow$Site 4
  - ASG' = (PROJ$\bowtie$ASG)$\longrightarrow$Site 1
  - (ASG'$\bowtie$EMP)$\longrightarrow$Site 2
  - Total cost$= 1 + 2 + 2 = 5$

SDD-1

- The **SDD-1 query optimization** algorithm improves the Hill-Climbing algorithm in a number of directions:
  - Semijoins are considered
  - More elaborate statistics
  - Initial plan is selected better
  - Post-optimization step is introduced

Conclusion

- Distributed query optimization is more complex that centralized query processing, since
  - bushy query trees are not necessarily a bad choice
  - one needs to decide what, where, and how to ship the relations between the sites
- Query optimization searches the optimal query plan (tree)
- For N relations, there are $O(N!)$ equivalent join trees. To cope with the complexity heuristics and/or restricted types of trees are considered
- There are two main strategies in query optimization: randomized and deterministic
- (Few) semi-joins can be used to implement a join. The semi-joins require more operations to perform, however the data transfer rate is reduced
- INGRES, System R, Hill Climbing, and SDD-1 are distributed query optimization algorithms

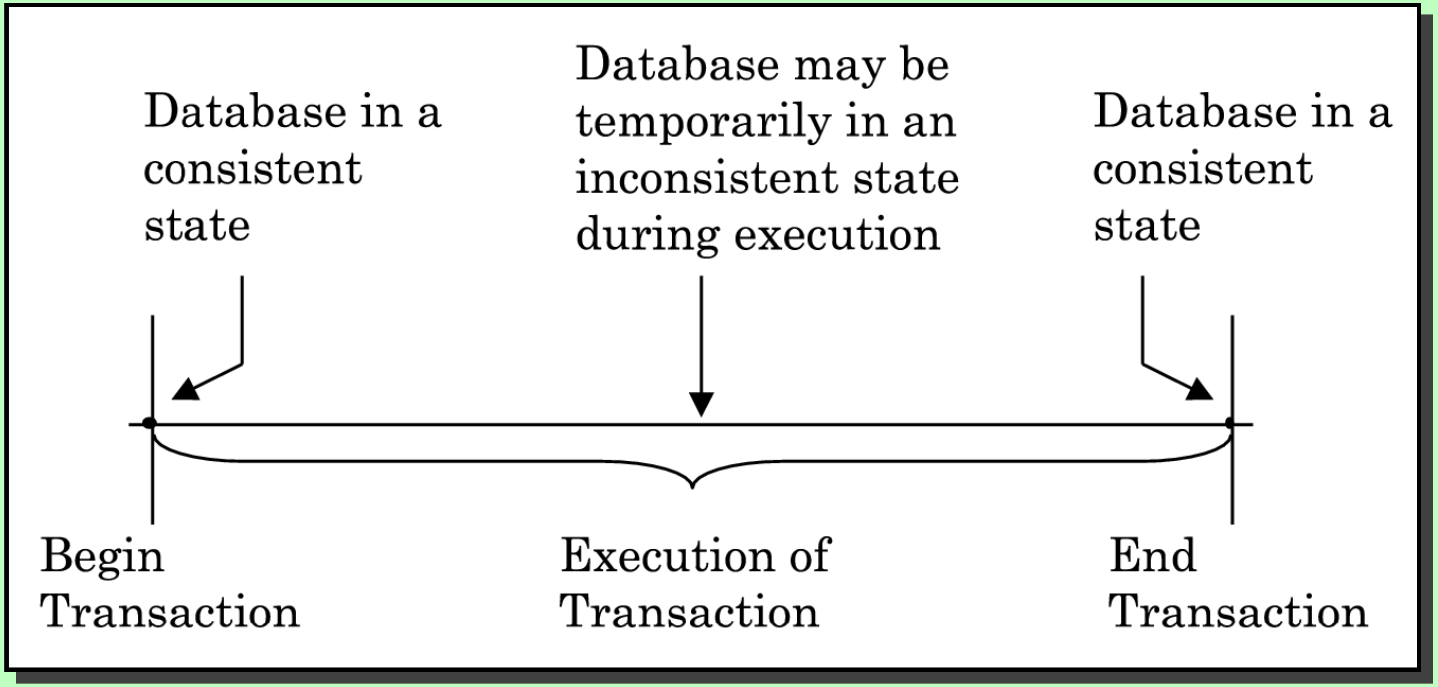
Chapter 8: Introduction to Transaction Management

- Definition and Examples
- Properties
- Classification
- Processing Issues

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Definition

- **Transaction:** A collection of actions that transforms the DB from one consistent state into another consistent state; during the exectuion the DB might be inconsistent.



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

Definition . . .

- **States** of a transaction
  - **Active**: Initial state and during the execution
  - **Paritally committed**: After the final statement has been executed
  - **Committed:** After successful completion
  - **Failed:** After the discovery that normal execution can no longer proceed
  - **Aborted:** After the transaction has been rolled back and the DB restored to its state prior to the start of the transaction. Restart it again or kill it.

Example

- **Example:** Consider an SQL query for increasing by 10% the budget of the CAD/CAM project. This query can be specified as a transaction by providing a name for the transaction and inserting a begin and end tag.

```
Transaction  BUDGET_UPDATE
begin
    EXEC SQL
    UPDATE  PROJ
    SET      BUDGET = BUDGET * 1.1
    WHERE    PNAME = "CAD/CAM"
end.
```

Example . . .

- **Example:** Consider an airline DB with the following relations:

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

- Consider the reservation of a ticket, where a travel agent enters the flight number, the date, and a customer name, and then asks for a reservation.

```
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 .
```

Example . . .

- **Example (contd.):** A transaction always terminates – commit or abort. Check the availability of free seats and terminate the transaction appropriately.

```
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.
```

Example . . .

- Transactions are mainly characterized by its Read and Write operations
  - Read set (RS): The data items that a transaction reads
  - Write set (WS): The data items that a transaction writes
  - Base set (BS): the union of the read set and write set

- **Example (contd.):** Read and Write set of the “Reservation” transaction

$$
\begin{aligned}
\text{RS[Reservation]} &= \{\text{FLIGHT.STSOLD, FLIGHT.CAP}\} \\
\text{WS[Reservation]} &= \{\text{FLIGHT.STSOLD, FC.FNO, FC.DATE,} \\
&\quad\ \text{FC.CNAME, FC.SPECIAL}\} \\
\text{BS[Reservation]} &= \{\text{FLIGHT.STSOLD, FLIGHT.CAP,} \\
&\quad\ \text{FC.FNO, FC.DATE, FC.CNAME, FC.SPECIAL}\}
\end{aligned}
$$

Formalization of a Transaction

- We use the following notation:
  - T_i be a transaction and x be a relation or a data item of a relation
  - $O_{ij} \in \{R(x), W(x)\}$ be an atomic $read$/$write$ operation of T_i on data item x
  - $OS_i = \bigcup_j O_{ij}$ be the set of all operations of T_i
  - $N_i \in \{A, C\}$ be the termination operation, i.e., $abort/commit$
- Two operations $O_{ij}(x)$ and $O_{ik}(x)$ on the same data item are in **conflict** if at least one of them is a $write$ operation
- A **transaction** T is a **partial order** over its operations, i.e., $T \ = \ \{\Sigma$,<i$\}$, where
  - $\Sigma_i = OS_i \cup N_i$
  - For any $O_{ij} = \{R(x) \vee W(x)\}$ and $O_{ik} = W(x)$, either $O_{ij} \prec_i O_{ik}$ or $O_{ik} \prec_i O_{ij}$
  - $\forall O_{ij} \in OS_i (O_{ij} \prec_i N_i)$
- Remarks
  - The partial order $\prec$ is given and is actually application dependent
  - It has to specify the **execution order** between the conflicting operations and between all operations and the termination operation

Formalization of a Transaction . . .

- **Example:** Consider the following transaction T

```
Read(x)
Read(y)
x ← x + y
Write(x)
Commit
```

- The transaction is formally represented as

$$\Sigma = \{R(x), R(y), W(x), C\}$$

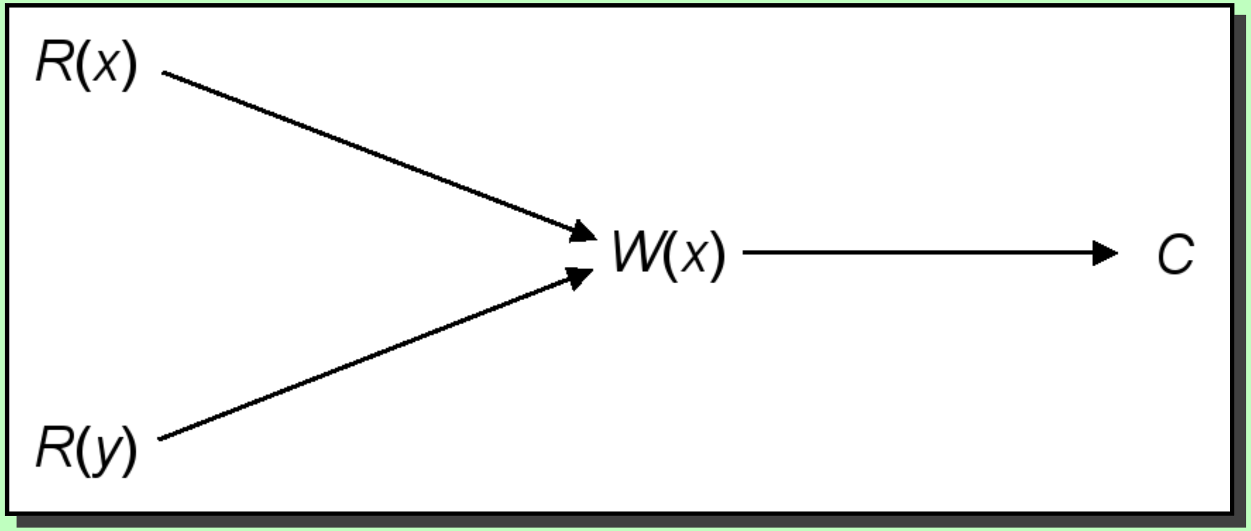
$$\prec = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$$

Formalization of a Transaction . . .

- **Example (contd.):** A transaction can also be specified/represented as a directed acyclic graph (DAG), where the vertices are the operations and the edges indicate the ordering.
  - Assume

$$\prec = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$$

  - The DAG is



Formalization of a Transaction . . .

- **Example:** The reservation transaction is more complex, as it has two possible termination conditions, but a transaction allows only one
  - BUT, a transaction is the **execution** of a program which has obviously only one termination
  - Thus, it can be represented as two transactions, one that aborts and one that commits

Transaction T1:

$$\Sigma = \{R(ST, \mathrm{SOLD}) \quad (\mathrm{R\ CAP}),\mathrm{A}\}$$

$$\prec = \{(R(ST, \mathrm{SOLD}\ ),\mathrm{A}\ )\ (\ (\mathrm{R\ CAP}\ ),\mathrm{A}\ )\}$$

Transaction T2:

$$\Sigma = \{R(STSOLD), R(CAP), W(STSOLD), W(FNO), W(DATE), W(CNAME), W(SPECIAL), C\}$$

$$\prec = \{(R(STSOLD), W(STSOLD)), \ldots\}$$

```
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.
```

Properties of Transactions

- The **ACID properties**
  - **Atomicity**
    - ∗ A transaction is treated as a single/atomic unit of operation and is either executed completely or not at all
  - **Consistency**
    - ∗ A transaction preserves DB consistency, i.e., does not violate any integrity constraints
  - **Isolation**
    - ∗ A transaction is executed as if it would be the only one.
  - **Durability**
    - ∗ The updates of a committed transaction are permanent in the DB

Properties of Transactions . . .

- **Atomicity**
  - Either **all or none** of the transaction's operations are performed
  - Partial results of an interrupted transactions must be undone
  - **Transaction recovery** is the activity of the restoration of atomicity due to input errors, system overloads, and deadlocks
  - **Crash recovery** is the activity of ensuring atomicity in the presence of system crashes

Properties of Transactions . . .

- **Consistency**
  - The consistency of a transaction is simply its correctness and ensures that a transaction transforms a consistent DB into a consistent DB
  - Transactions are **correct** programs and do not violate database integrity constraints
  - **Dirty data** is data that is updated by a transaction that has not yet committed
  - Different **levels of DB consistency** (by Gray et al., 1976)
    - ∗ Degree 0
      - · Transaction T does not overwrite dirty data of other transactions
    - ∗ Degree 1
      - · Degree 0 + T does not commit any writes before EOT
    - ∗ Degree 2
      - · Degree 1 + T does not read dirty data from other transactions
    - ∗ Degree 3
      - · Degree 2 + Other transactions do not dirty any data read by T before T completes

Properties of Transactions . . .

- **Isolation**
  - Isolation is the property of transactions which requires each transaction to see a consistent DB at all times.
  - If two concurrent transactions access a data item that is being updated by one of them (i.e., performs a **write** operation), it is not possible to guarantee that the second will read the correct value
  - Interconsistency of transactions is obviously achieved if transactions are executed serially
  - Therefore, if several transactions are executed concurrently, the result must be the same as if they were executed serially in some order (⟶ serializability)

Properties of Transactions . . .

- **Example:** Consider the following two transactions, where initially $x = 50$:

```
T1: Read(x)                    T2: Read(x)
    x ← x+1                        x ← x+1
    Write(x)                       Write(x)
    Commit                         Commit
```

- Possible execution sequences:

```
T1: Read(x)                    T1: Read(x)
T1: x<--x+1                    T1: x<--x+1
T1: Write(x)                   T2: Read(x)
T1: Commit                     T1: Write(x)
T2: Read(x)                    T2: x ← x+1
T2: x ← x+1                    T2: Write(x)
T2: Write(x)                   T1: Commit
T2: Commit                     T2: Commit
```

  - Serial execution: we get the correct result $x = 52$ (the same for $\{T_2, T_1\}$)

  - Concurrent execution: T_2 reads the value of x while it is being changed; the result is $x = 51$ and is incorrect!

Properties of Transactions . . .

- SQL-92 specifies 3 phenomena/situations that occur if proper isolation is not maintained
  - **Dirty read**
    - T_1 modifies x which is then read by T_2 before T_1 terminates; if T_1 aborts, T_2 has read value which never exists in the DB:
  - **Non-repeatable (fuzzy) read**
    - T_1 reads x; T then modifies or deletes x and commits; T tries to read x again but reads a different value or can't find it
  - **Phantom**
    - T_1 searches the database according to a predicate P while T_2 inserts new tuples that satisfy P

Properties of Transactions . . .

- Based on the 3 phenomena, SQL-92 specifies different isolation levels:
  - **Read uncommitted**
    - For transactions operating at this level, all three phenomena are possible
  - **Read committed**
    - Fuzzy reads and phantoms are possible, but dirty reads are not
  - **Repeatable read**
    - Only phantoms possible
  - **Anomaly serializable**
    - None of the phenomena are possible

Properties of Transactions . . .

- **Durability**
  - Once a transaction commits, the system must guarantee that the results of its operations will never be lost, in spite of subsequent failures
  - Database recovery is used to achieve the task

Classification of Transactions

- **Classification** of transactions according to various criteria
  - **Duration** of transaction
    - On-line (short-life)
    - Batch (long-life)
  - **Organization** of **read** and **write** instructions in transaction
    - General model

$$T \ : \ \{R(x),R(y),W(y),R(z),W(x),W(z),W(w),C\}$$

    - Two-step (all reads before writes)

$$T_2 : \{R(x), R(y), R(z), W(x), W(z), W(y), W(w), C\}$$

    - Restricted (a data item has to be read before an update)

$$T_3 : \{R(x), R(y), W(y), R(z), W(x), W(z), \mathbf{R}(\mathbf{w}), W(w), C\}$$

    - Action model: each (read,write) pair is executed atomically

$$T_2 : \{[R(x), W(x)], [R(y), W(y)], [R(z), W(z)], [R(w), W(w)], C\}$$

Classification of Transactions . . .

- **Classification** of transactions according to various criteria . . .
  - **Structure** of transaction
    - ∗ **Flat** transaction
      - · Consists of a sequence of primitive operations between a begin and end marker

```
Begin_transaction Reservation
  ...
end.
```

    - ∗ **Nested** transaction
      - · The operations of a transaction may themselves be transactions.

```
Begin_transaction Reservation
  ...
  Begin_transaction Airline
    ...
  end.
  Begin_transaction Hotel
    ...
  end.
end.
```

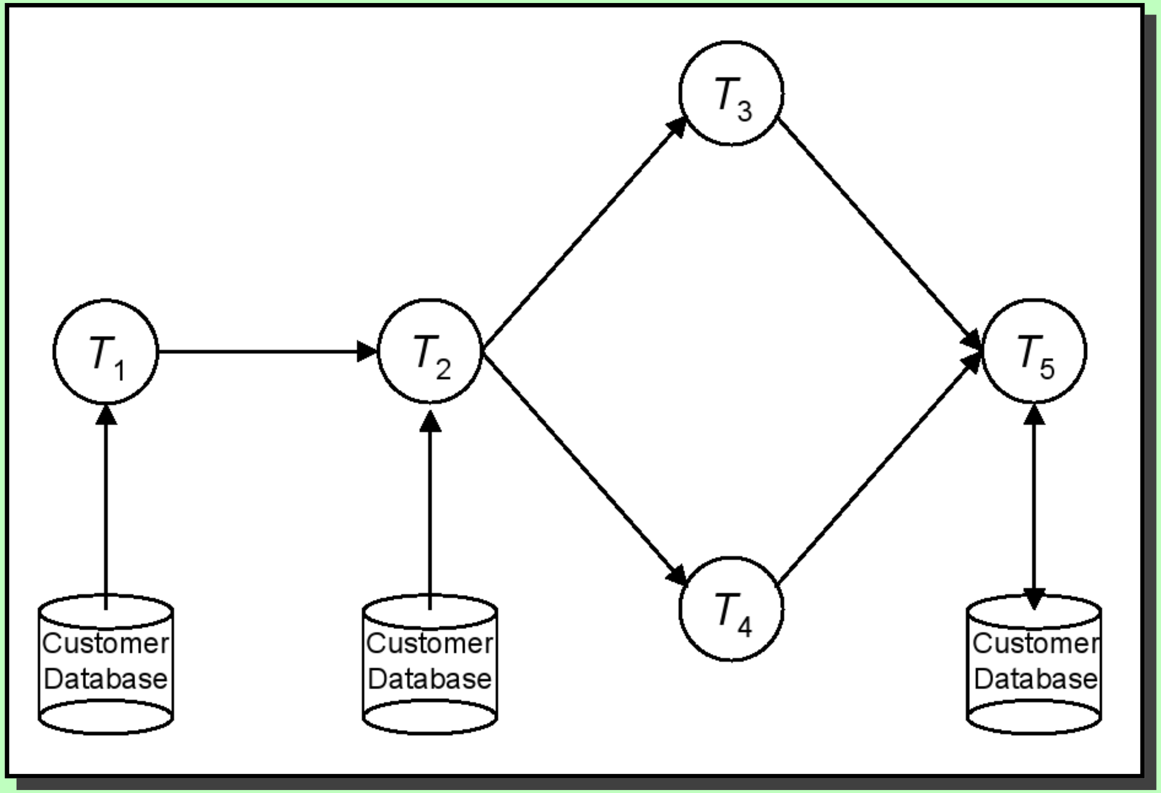
    - ∗ **Workflows** (next slide)

Classification of Transactions . . .

- **Workflows:** A collection of tasks organized to accomplish a given business process
  - Workflows generalize transactions and are more expressive to model complex business processes
  - Types of workflows:
    - Human-oriented workflows
      - Involve humans in performing the tasks.
      - System support for collaboration and coordination; but no system-wide consistency definition
    - System-oriented workflows
      - Computation-intensive and specialized tasks that can be executed by a computer
      - System support for concurrency control and recovery, automatic task execution, notification, etc.
    - Transactional workflows
      - In between the previous two; may involve humans, require access to heterogeneous, autonomous and/or distributed systems, and support selective use of ACID properties

Classification of Transactions . . .

- **Example:** We extend the reservation example and show a typical workflow



- $T1$: Customer request
- $T2$: Airline reservation
- $T3$: Hotel reservation
- $T4$: Auto reservation
- $T5$: Bill

Transaction Processing Issues

- Transaction structure (usually called transaction model)
  - Flat (simple), nested
- Internal database consistency
  - Semantic data control (integrity enforcement) algorithms
- Reliability protocols
  - Atomicity and Durability
  - Local recovery protocols
  - Global commit protocols
- Concurrency control algorithms
  - How to synchronize concurrent transaction executions (correctness criterion)
  - Intra-transaction consistency, isolation
- Replica control protocols
  - How to control the mutual consistency of replicated data

Conclusion

- A transaction is a collection of actions that transforms the system from one consistent state into another consistent state
- Transaction T can be viewed as a partial order: $T = \{\Sigma, \prec\}$, where Σ is the set of all operations, and $\prec$ denotes the order of operations. T can be also represented as a directed acyclic graph (DAG)
- Transaction manager aims to achieve four properties of transactions: atomicity, consistency, isolation, and durability
- Transactions can be classified according to (i) time, (ii) organization of reads and writes, and (iii) structure
- Transaction processing involves reliability, concurrency, and replication protocols to ensure the four properties of the transactions

Chapter 9: Concurrency Control

- Concurrency, Conflicts, and Schedules
- Locking Based Algorithms
- Timestamp Ordering Algorithms
- Deadlock Management

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Chapter 10: Distributed DBMS Reliability

- Definitions and Basic Concepts
- Local Recovery Management
- In-place update, out-of-place update
- Distributed Reliability Protocols
- Two phase commit protocol
- Three phase commit protocol

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Reliability

- A **reliable DDBMS** is one that can continue to process user requests even when the underlying system is unreliable, i.e., failures occur
- **Failures**
  - Transaction failures
  - System (site) failures, e.g., system crash, power supply failure
  - Media failures, e.g., hard disk failures
  - Communication failures, e.g., lost/undeliverable messages
- Reliability is closely related to the problem of how to maintain the **atomicity** and **durability** properties of transactions

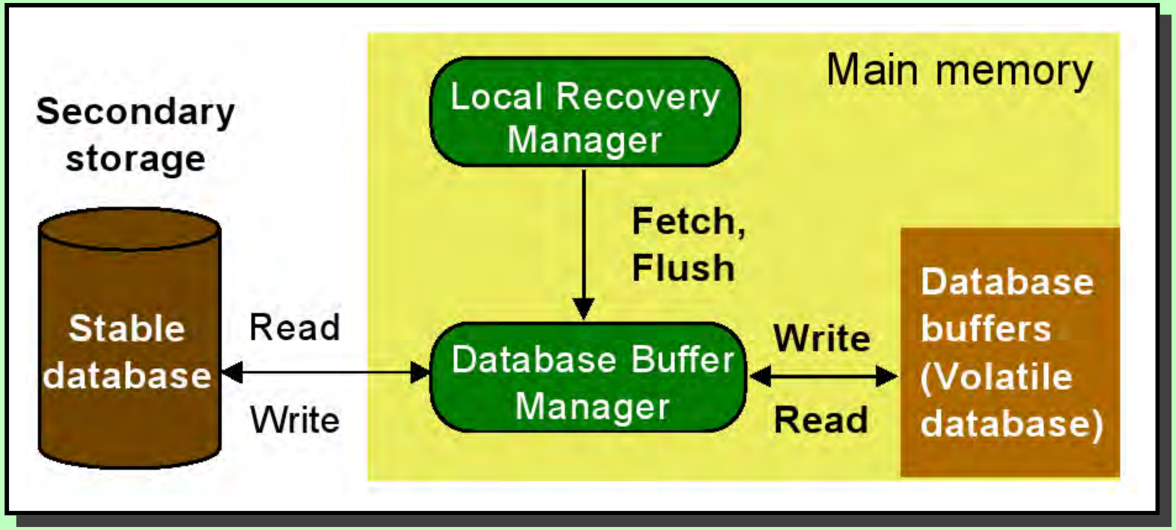
Reliability . . .

- **Recovery system:** Ensures atomicity and durability of transactions in the presence of failures (and concurrent transactions)
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the DB contents to a state that ensures atomicity, consistency and durability

Local Recovery Management

- The **local recovery manager (LRM)** maintains the atomicity and durability properties of local transactions at each site.

- **Architecture**
  - **Volatile** storage: The main memory of the computer system (RAM)
  - **Stable** storage
    - ∗ A storage that “never” looses its contents
    - ∗ In reality this can only be approximated by a combination of hardware (non-volatile storage) and software (stable-write, stable-read, clean-up) components

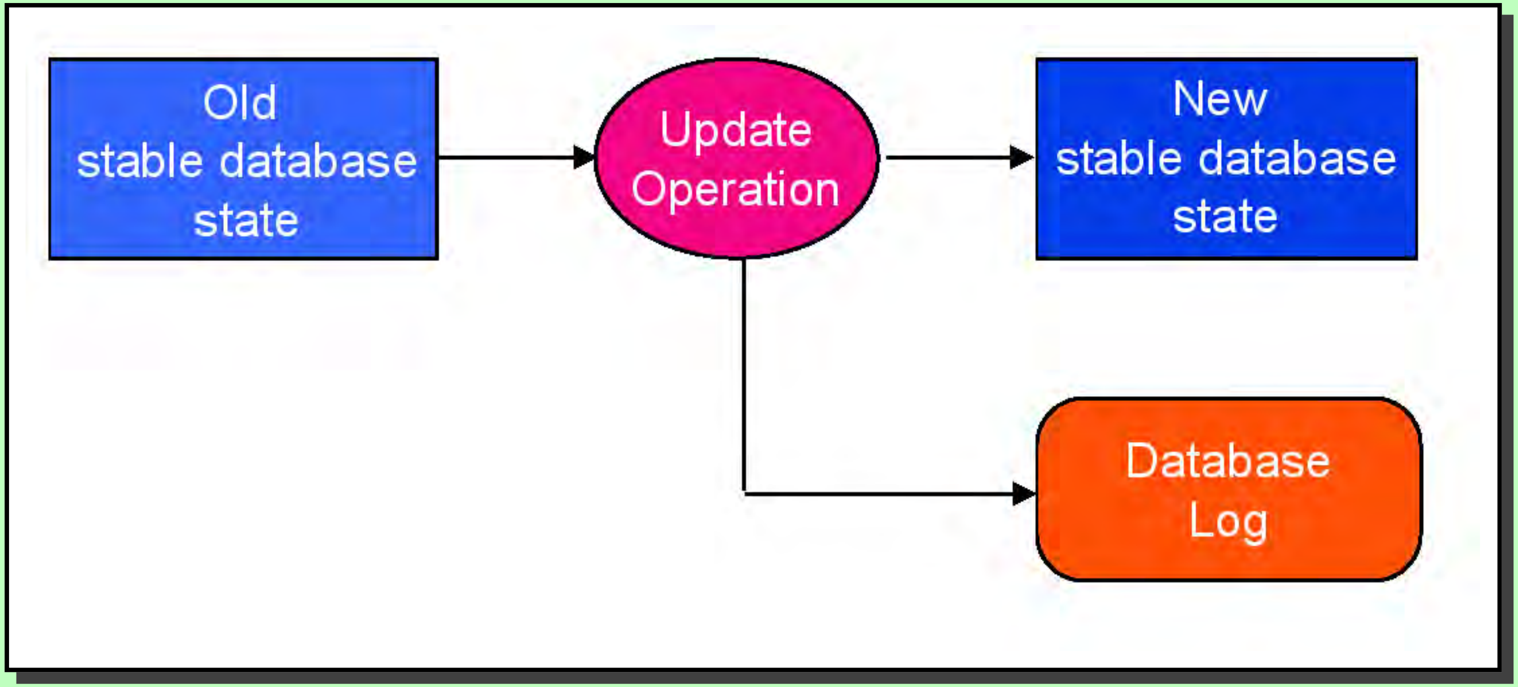


Local Recovery Management . . .

- Two ways for the LRM to deal with update/write operations
  - **In-place update**
    - Physically changes the value of the data item in the stable database
    - As a result, previous values are lost
    - Mostly used in databases
  - **Out-of-place update**
    - The new value(s) of updated data item(s) are stored separately from the old value(s)
    - Periodically, the updated values have to be integrated into the stable DB

In-Place Update

- Since in-place updates cause previous values of the affected data items to be lost, it is necessary to keep enough information about the DB updates in order to allow recovery in the case of failures
- Thus, every action of a transaction must not only perform the action, but must also write a log record to an append-only **log file**

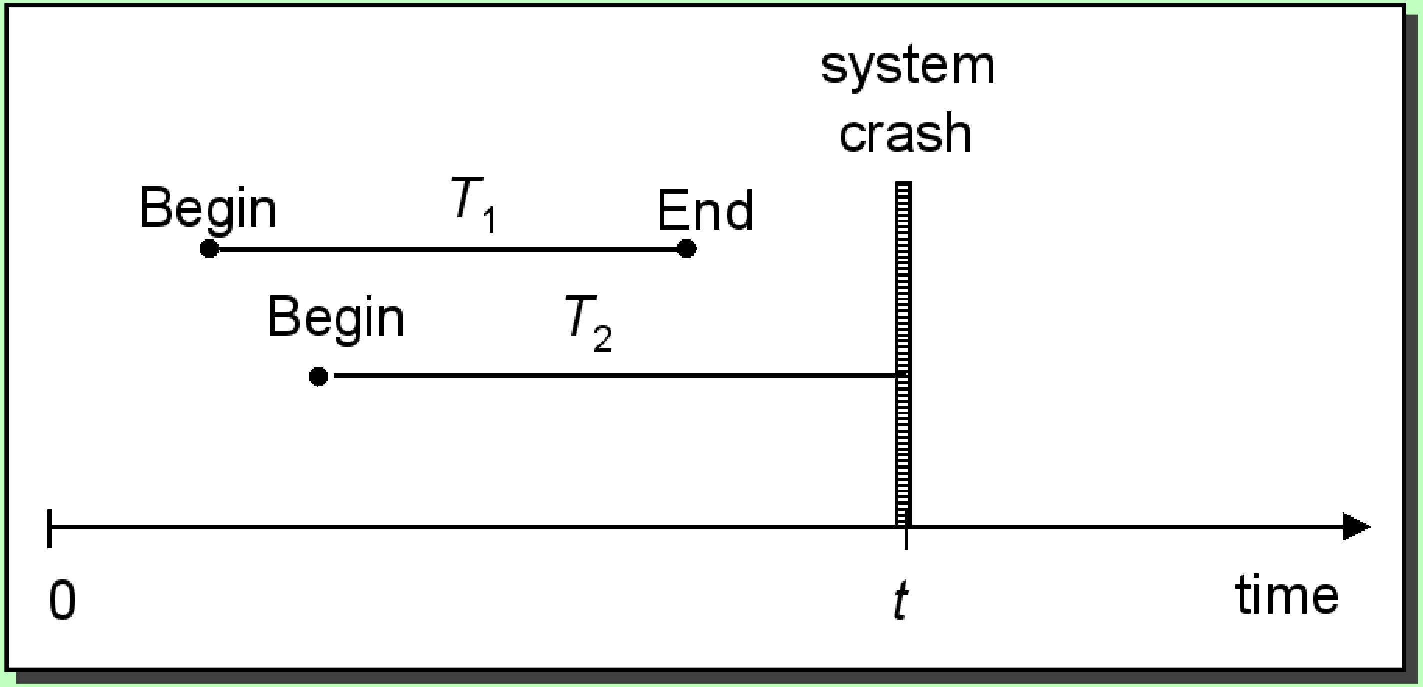


In-Place Update . . .

- A **log** is the most popular structure for recording DB modifications on stable storage
- Consists of a sequence of **log records** that record all the update activities in the DB
- Each log record describes a significant event during transaction processing
- Types of log records
  - $< T_i, \mathbf{start} >$: if transaction T_i has started
  - $< T_i, X\ , V\ , V\ \ >$: before T executes a $(X\ )$, where V is the old value before the write and V_2 is the new value after the write
  - $< T_i, \mathbf{commit} >$: if T_i has committed
  - $< T_i, \mathbf{abort} >$: if T_i has aborted
  - $< \mathbf{checkpoint} >$
- With the information in the log file the recovery manager can restore the consistency of the DB in case of a failure.

In-Place Update . . .

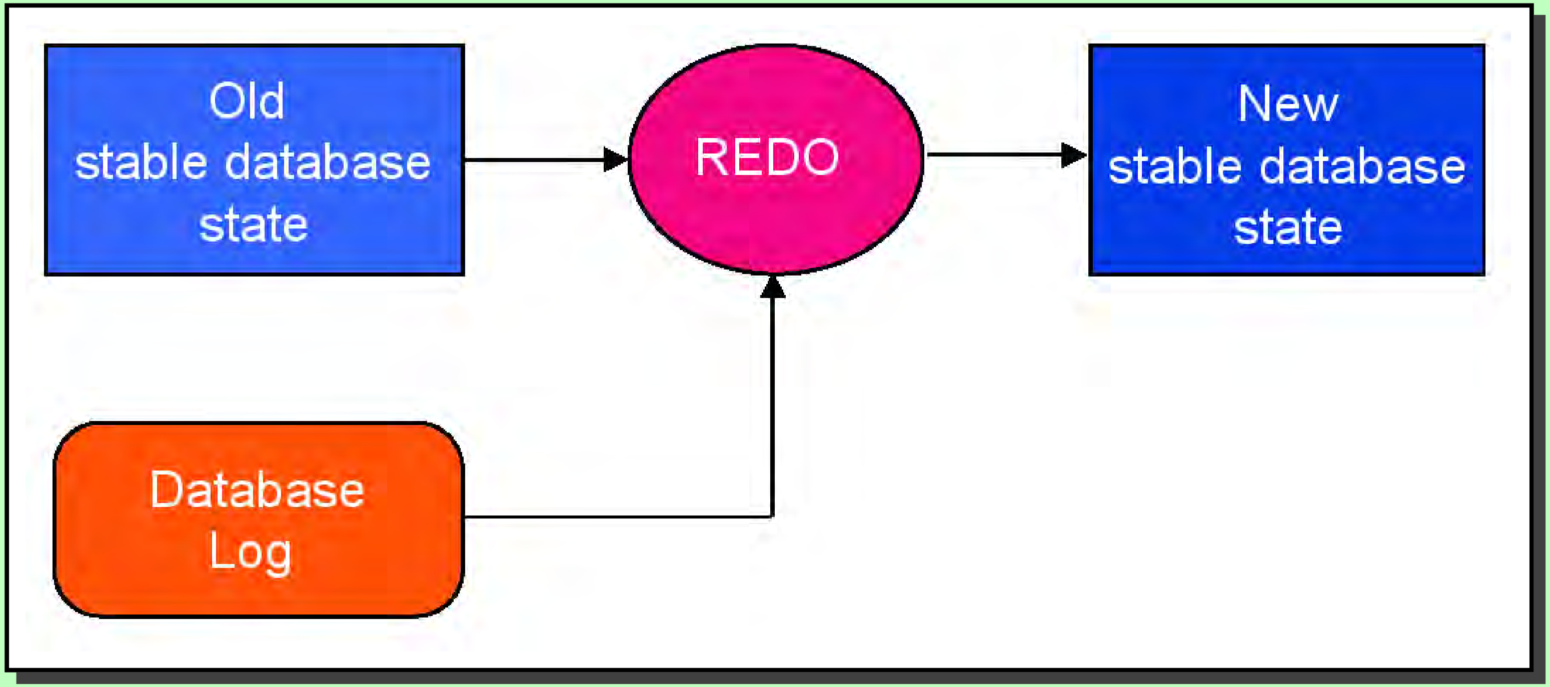
- Assume the following situation when a system crash occurs



- Upon recovery:
  - All effects of transaction T_1 should be reflected in the database (⇒ REDO)
  - None of the effects of transaction T_2 should be reflected in the database (⇒ UNDO)

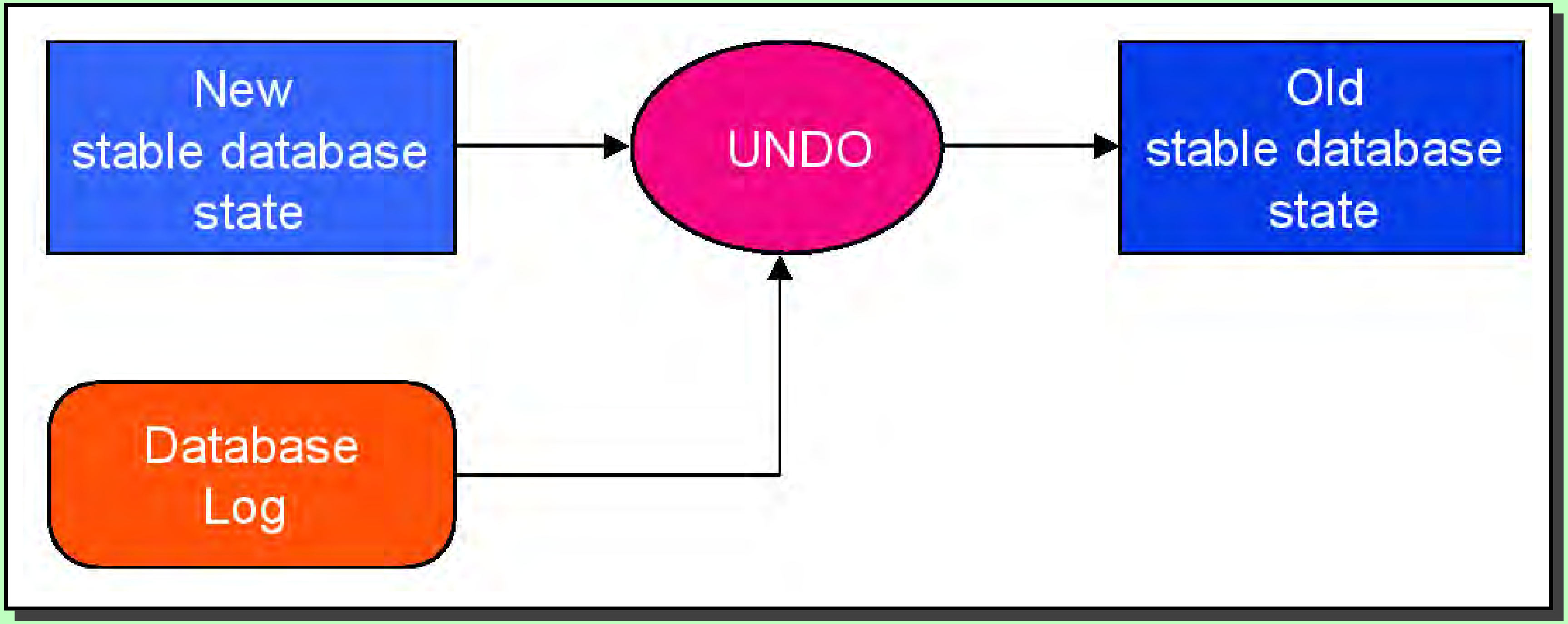
In-Place Update . . .

- **REDO Protocol**
  - REDO’ing an action means performing it again
  - The REDO operation uses the log information and performs the action that might have been done before, or not done due to failures
  - The REDO operation generates the new image



In-Place Update . . .

- **UNDO Protocol**
  - UNDO’ing an action means to restore the object to its image before the transaction has started
  - The UNDO operation uses the log information and restores the old value of the object



In-Place Update . . .

- **Example:** Consider the transactions T_0 and T_1 (T_0 executes before T_1) and the following initial values: $A = 1000$, $B = 2000$, and $C = 700$

T_0: $read(A)$
$A = A - 50$
$write(A)$
$read(B)$
$B = B + 50$
$write(B)$

T_1: $read(C)$
$C = C - 100$
$write(C)$

  – Possible order of actual outputs to the log file and the DB:

Log	
$< T_0, start >$	
$< T_0, A, 1000, 950 >$	
$< T_0, B, 2000, 2050 >$	
$< T_0, commit >$	
	$A = 950$
	$B = 2050$
$< T_1, start >$	
$< T_1, C, 700, 600 >$	
$< T_1, commit >$	
	$C = 600$

In-Place Update . . .

- **Example (contd.):** Consider the log after some system crashes and the corresponding recovery actions

$$
\begin{aligned}
(a)\ & < T_0, start > \\
& < T_0, A, 1000, 950 > \\
& < T_0, B, 2000, 2050 >
\end{aligned}
$$

$$
\begin{aligned}
(b)\ & < T_0, start > \\
& < T_0, A, 1000, 950 > \\
& < T_0, B, 2000, 2050 > \\
& < T\ \ , commit > \\
& < T\ \ , start > \\
& < T_1, C, 700, 600 >
\end{aligned}
$$

$$
\begin{aligned}
(c)\ & < T_0, start > \\
& < T_0, A, 1000, 950 > \\
& < T_0, B, 2000, 2050 > \\
& < T\ \ , commit > \\
& < T\ \ , start > \\
& < T_1, C, 700, 600 > \\
& < T_1, commit >
\end{aligned}
$$

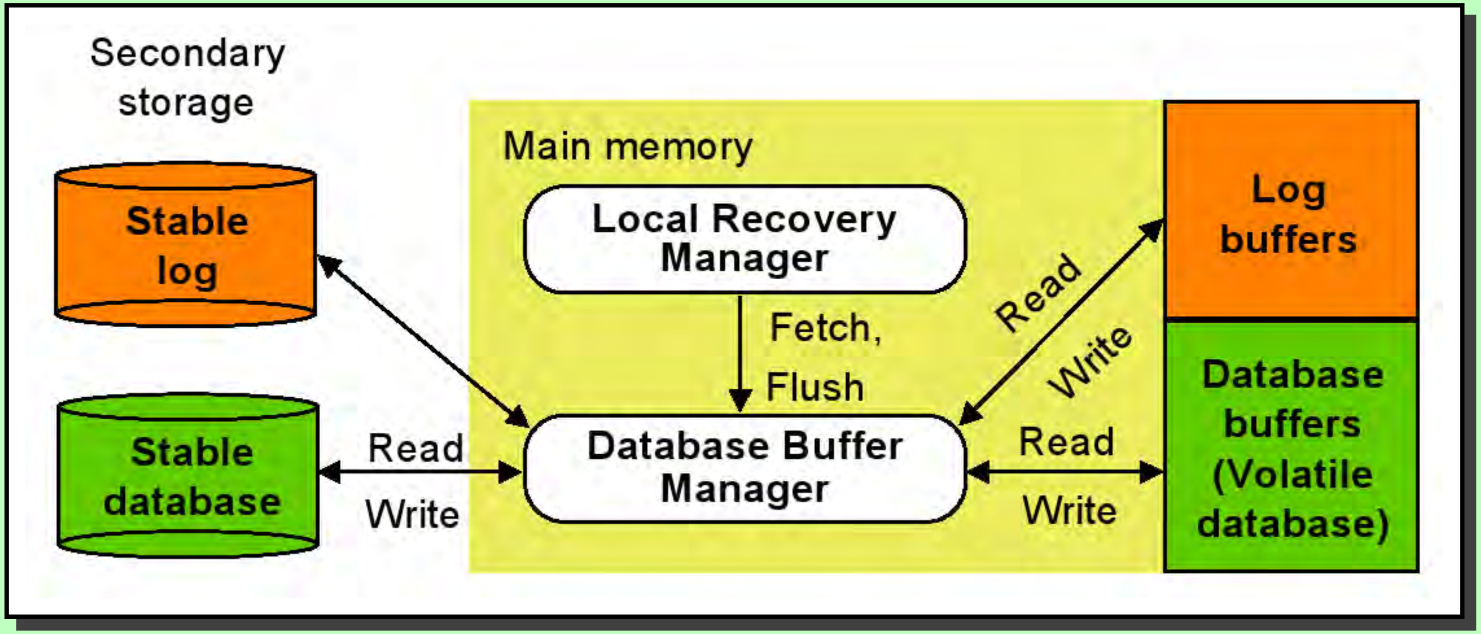
(a) undo(T0): B is restored to 2000 and A to 1000

(b) undo(T1) and redo(T0): C is restored to 700, and then A and B are set to 950 and 2050, respectively

(c) redo(T0) and redo(T1): A and B are set to 950 and 2050, respectively; then C is set to 600

In-Place Update . . .

- **Logging Interface**



- Log pages/buffers can be written to stable storage in two ways:
  - **synchronously**
    - ∗ The addition of each log record requires that the log is written to stable storage
    - ∗ When the log is written synchronoously, the executtion of the transaction is supended until the write is complete ⟶ delay in response time
  - **asynchronously**
    - ∗ Log is moved to stable storage either at periodic intervals or when the buffer fills up.

In-Place Update . . .

- **When to write** log records into stable storage?
- Assume a transaction T updates a page P
- Fortunate case
  - System writes P in stable database
  - System updates stable log for this update
  - SYSTEM FAILURE OCCURS!... (before T commits)
  - We can recover (undo) by restoring P to its old state by using the log
- Unfortunate case
  - System writes P in stable database
  - SYSTEM FAILURE OCCURS!... (before stable log is updated)
  - We cannot recover from this failure because there is no log record to restore the old value
- Solution: Write-Ahead Log (WAL) protocol

In-Place Update . . .

- Notice:
  - If a system crashes before a transaction is committed, then all the operations must be undone. We need only the before images (undo portion of the log)
  - Once a transaction is committed, some of its actions might have to be redone. We need the after images (redo portion of the log)
- **Write-Ahead-Log (WAL) Protocol**
  - Before a stable database is updated, the undo portion of the log should be written to the stable log
  - When a transaction commits, the redo portion of the log must be written to stable log prior to the updating of the stable database

Out-of-Place Update

- Two out-of-place strategies are shadowing and differential files
- **Shadowing**
  - When an update occurs, don't change the old page, but create a shadow page with the new values and write it into the stable database
  - Update the access paths so that subsequent accesses are to the new shadow page
  - The old page is retained for recovery
- **Differential files**
  - For each DB file F maintain
    - a read-only part FR
    - a differential file consisting of insertions part (DF^+) and deletions part (DF^-)
  - Thus, $F = (FR \cup DF^+) - DF^-$

Distributed Reliability Protocols

- As with local reliability protocols, the distributed versions aim to maintain the atomicity and durability of distributed transactions
- Most problematic issues in a distributed transaction are commit, termination, and recovery
  - **Commit protocols**
    - ∗ How to execute a commit command for distributed transactions
    - ∗ How to ensure atomicity (and durability)?
  - **Termination protocols**
    - ∗ If a failure occurs at a site, how can the other 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

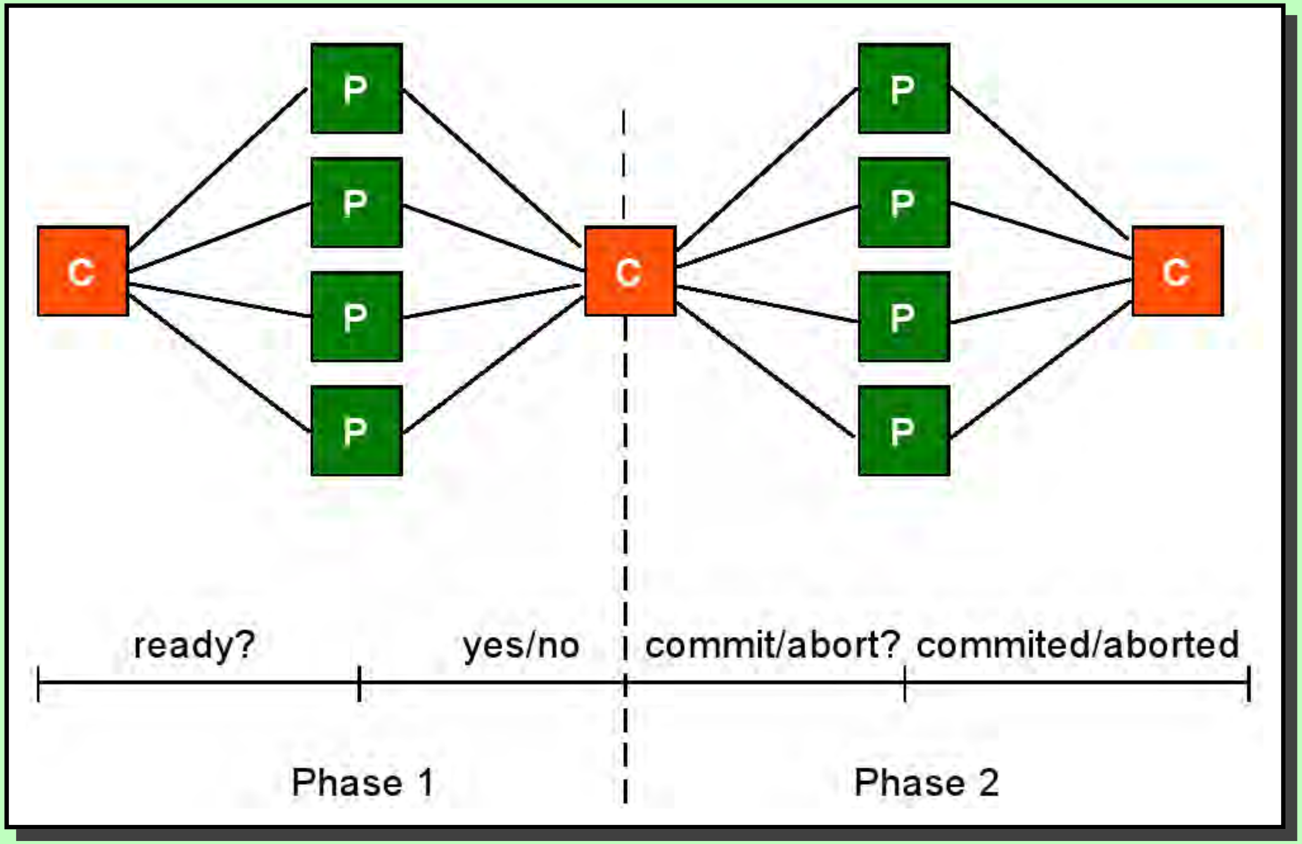
Commit Protocols

- Primary requirement of commit protocols is that they maintain the atomicity of distributed transactions (**atomic commitment**)
  - i.e., even though the exectution of the distributed transaction involves multiple sites, some of which might fail while executing, the effects of the transaction on the distributed DB is all-or-nothing.
- In the following we distinguish two roles
  - 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

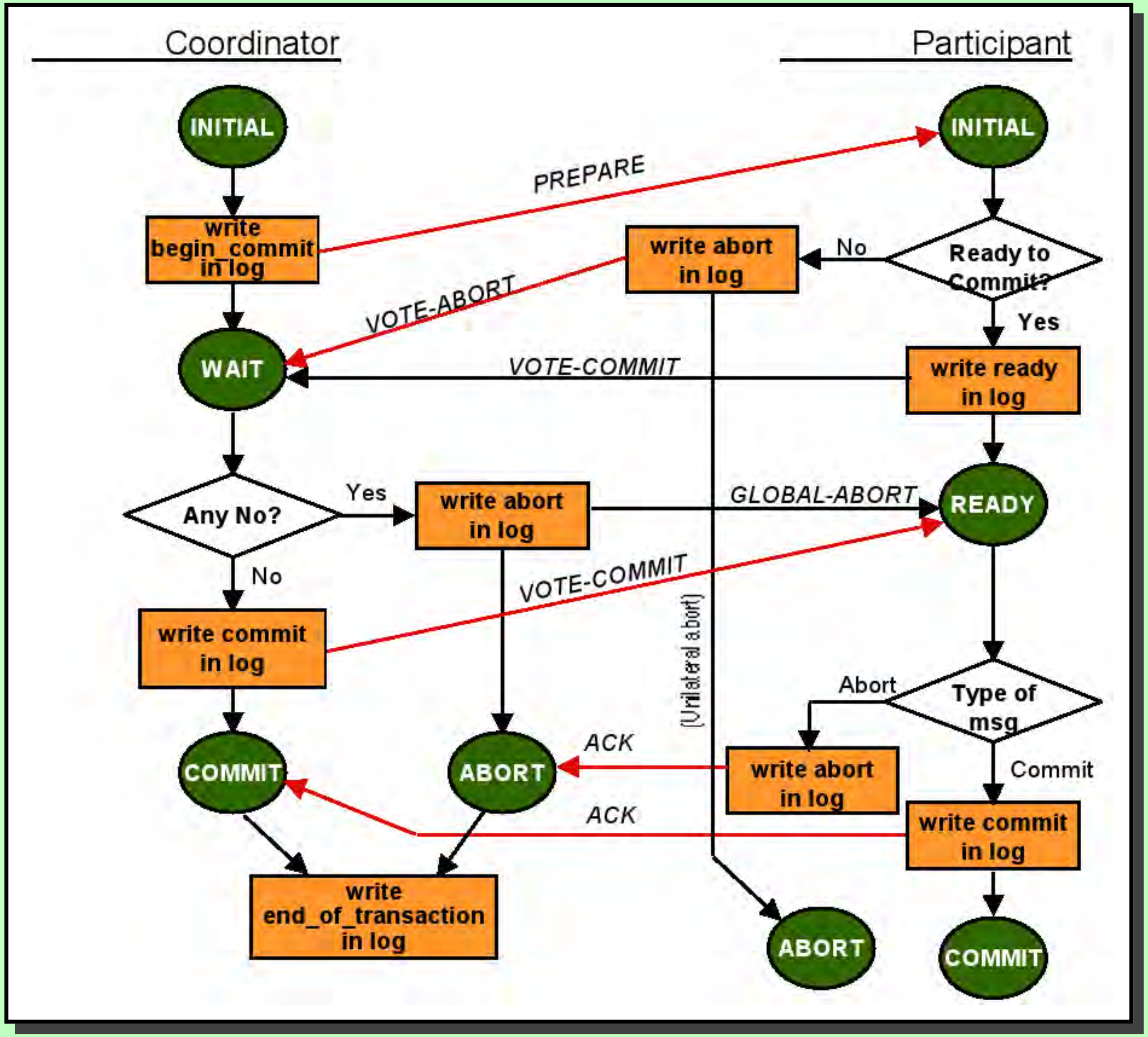
Centralized Two Phase Commit Protocol (2PC)

- Very simple protocol that ensures the atomic commitment of distributed transactions.
- **Phase 1:** The coordinator gets the participants ready to write the results into the database
- **Phase 2:** Everybody writes the results into the database
- **Global Commit Rule**
  - The coordinator aborts a transaction if and only if at least one participant votes to abort it
  - The coordinator commits a transaction if and only if all of the participants vote to commit it
- **Centralized** since communication is only between coordinator and the participants

Centralized Two Phase Commit Protocol (2PC) . . .

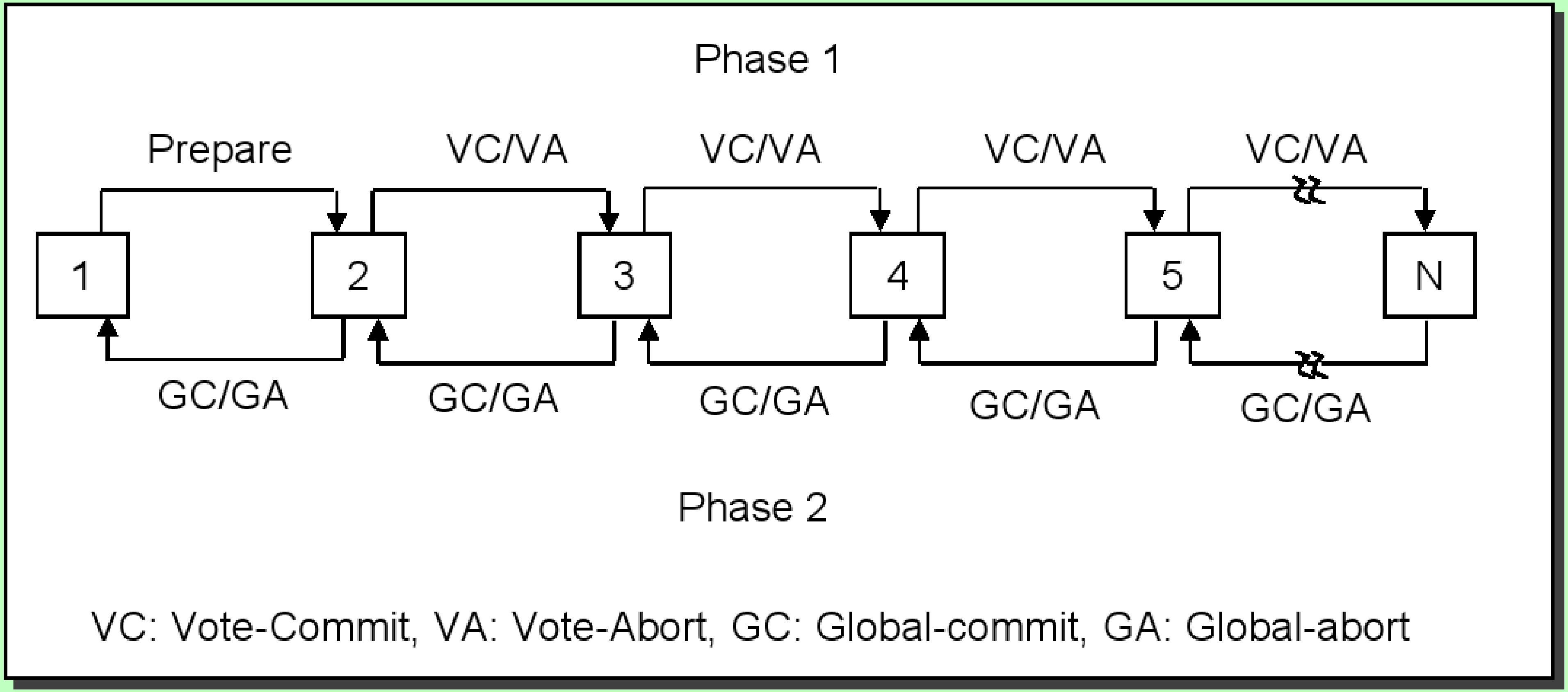


Centralized Two Phase Commit Protocol (2PC) . . .



Linear 2PC Protocol

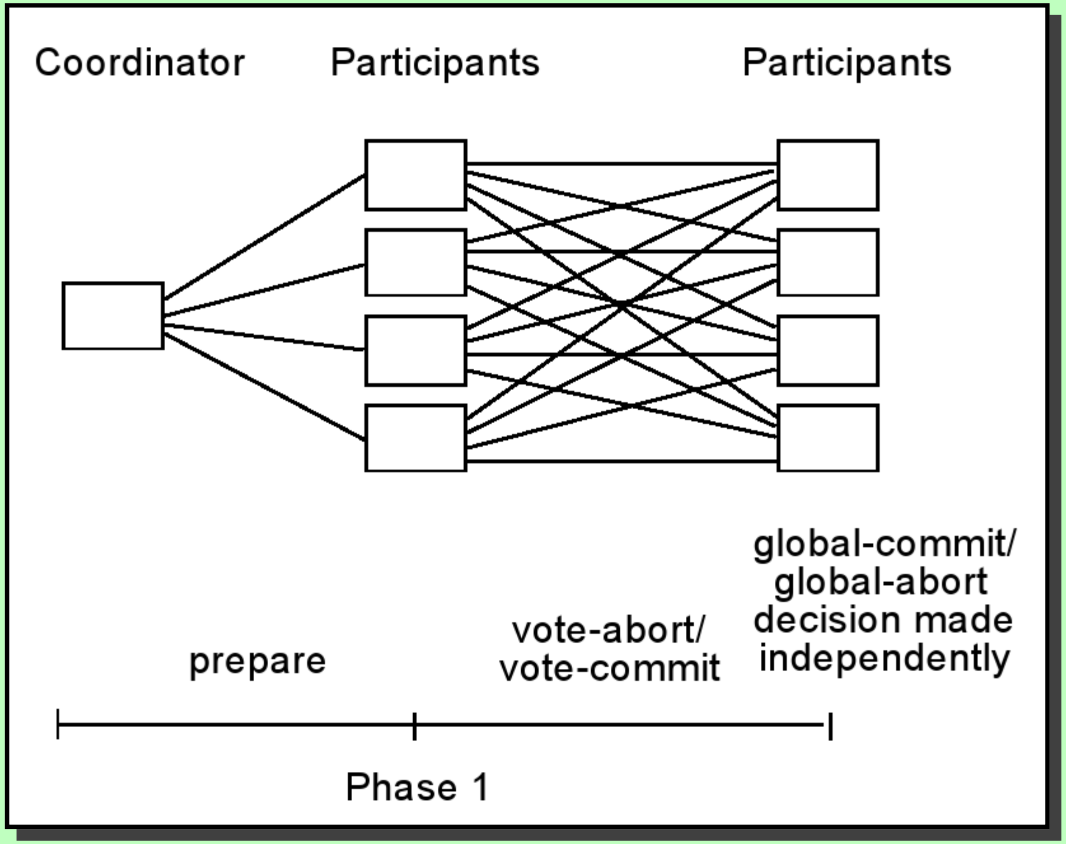
- There is linear ordering between the sites for the purpose of communication
- Minimizes the communication, but low response time as it does not allow any parallelism



VC: Vote-Commit, VA: Vote-Abort, GC: Global-commit, GA: Global-abort

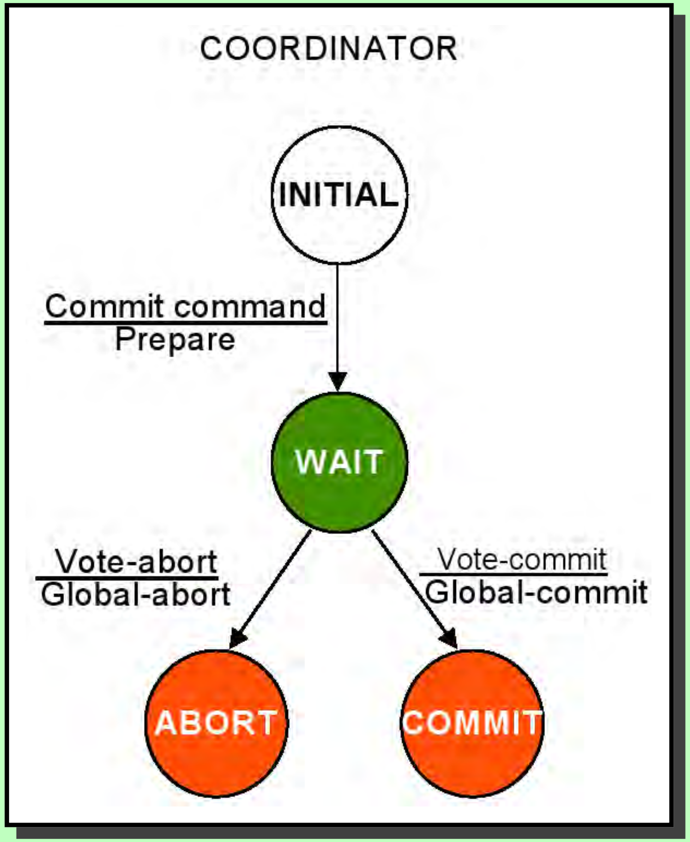
Distributed 2PC Protocol

- Distributed 2PC protocol increases the communication between the nodes
- Phase 2 is not needed, since each participant sends its vote to all other participants (+ the coordinator), thus each participants can derive the global decision



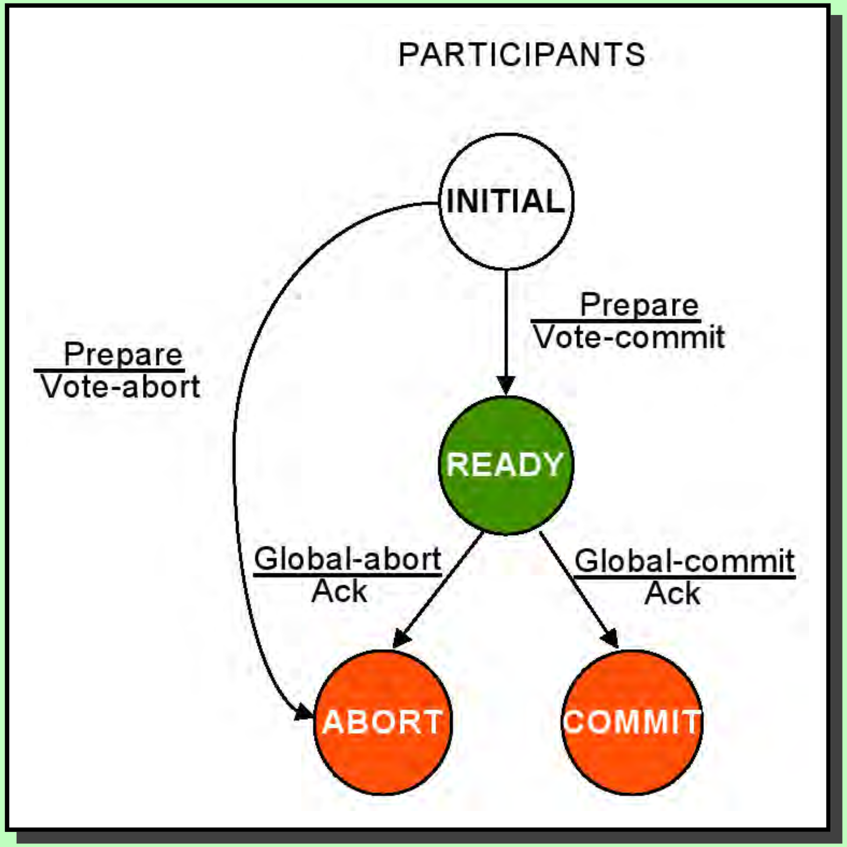
2PC Protocol and Site Failures

- **Site failures** in the 2PC protocol might lead to **timeouts**
- Timeouts are served by **termination protocols**
- We use the state transition diagrams of the 2PC for the analysis
- **Coordinator timeouts:** One of the participants is down. Depending on the state, the coordinator can take the following actions:
  - Timeout in INITIAL
    - Do nothing
  - Timeout in WAIT
    - Coordinator is waiting for local decisions
    - Cannot unilaterally commit
    - Can unilaterally abort and send an appropriate message to all participants
  - Timeout in ABORT or COMMIT
    - Stay blocked and wait for the acks (indefinitely, if the site is down indefinitely)



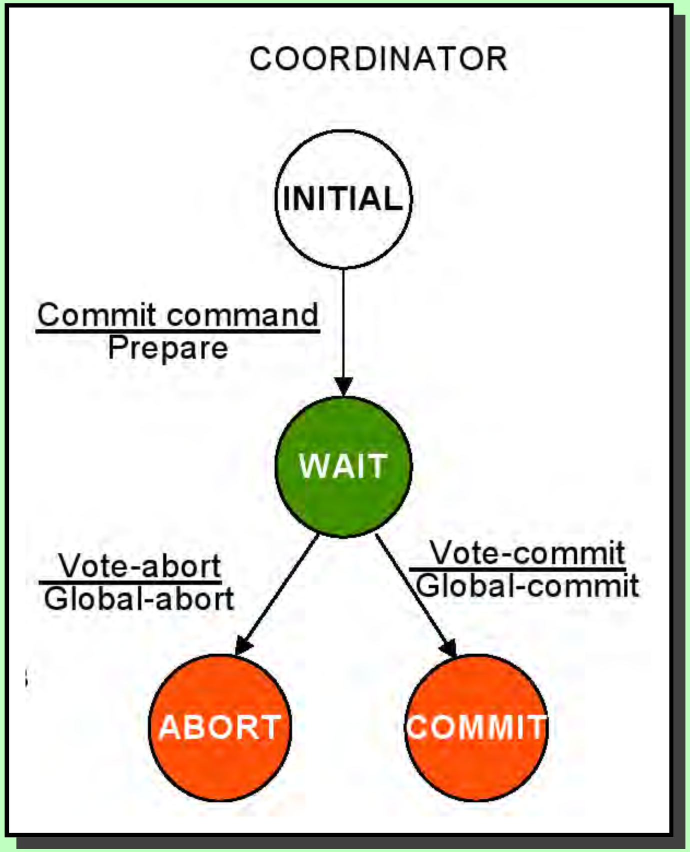
2PC Protocol and Site Failures . . .

- **Participant timeouts:** The coordinator site is down. A participant site is in
  - Timeout in INITIAL
    - ∗ Participant waits for “prepare”, thus coordinator must have failed in INITIAL state
    - ∗ Participant can unilaterally abort
  - Timeout in READY
    - ∗ Participant has voted to commit, but does not know the global decision
    - ∗ Participant stays blocked (indefinitely, if the coordinator is permanently down), since participant cannot change its vote or unilaterally decide to commit



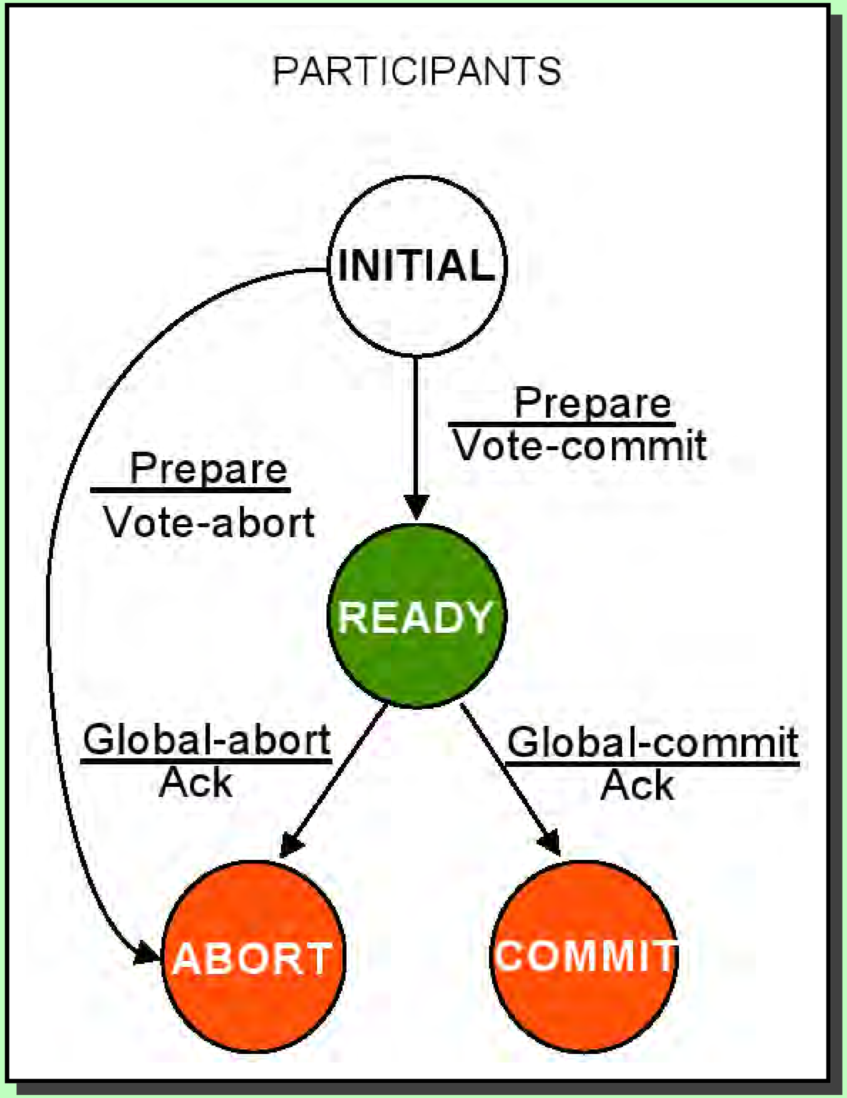
2PC Protocol and Site Failures . . .

- The actions to be taken after a recovery from a failure are specified in the **recovery protocol**
- **Coordinator site failure:** Upon recovery, it takes the following actions:
  - Failure in INITIAL
    - ∗ Start the commit process upon recovery (since coordinator did not send anything to the sites)
  - Failure in WAIT
    - ∗ Restart the commit process upon recovery (by sending “prepare” again to the participants)
  - Failure in ABORT or COMMIT
    - ∗ Nothing special if all the acks have been received from participants
    - ∗ Otherwise the termination protocol is involved (re-ask the acks)



2PC Protocol and Site Failures . . .

- **Participant site failure:** The coordinator sites recovers
  - Failure in INITIAL
    - Unilaterally abort upon recovery as the coordinator will eventually timeout since it will not receive the participant’s decision due to the failure
  - Failure in READY
    - The coordinator has been informed about the local decision
    - Treat as timeout in READY state and invoke the termination protocol (re-ask the status)
  - Failure in ABORT or COMMIT
    - Nothing special needs to be done



2PC Protocol and Site Failures . . .

- Additional cases
  - Coordinator site fails after writing ”begin_commit” log and before sending ”prepare” command
    - treat it as a failure in WAIT state; send ”prepare” command
  - Participant site fails after writing ”ready” record in log but before ”vote-commit” is sent
    - treat it as failure in READY state
    - alternatively, can send ”vote-commit” upon recovery
  - Participant site fails after writing ”abort” record in log but before ”vote-abort” is sent
    - no need to do anything upon recovery
  - Coordinator site fails after logging its final decision record but before sending its decision to the participants
    - coordinator treats it as a failure in COMMIT or ABORT state
    - participants treat it as timeout in the READY state
  - Participant site fails after writing ”abort” or ”commit” record in log but before acknowledgement is sent
    - participant treats it as failure in COMMIT or ABORT state
    - coordinator will handle it by timeout in COMMIT or ABORT state

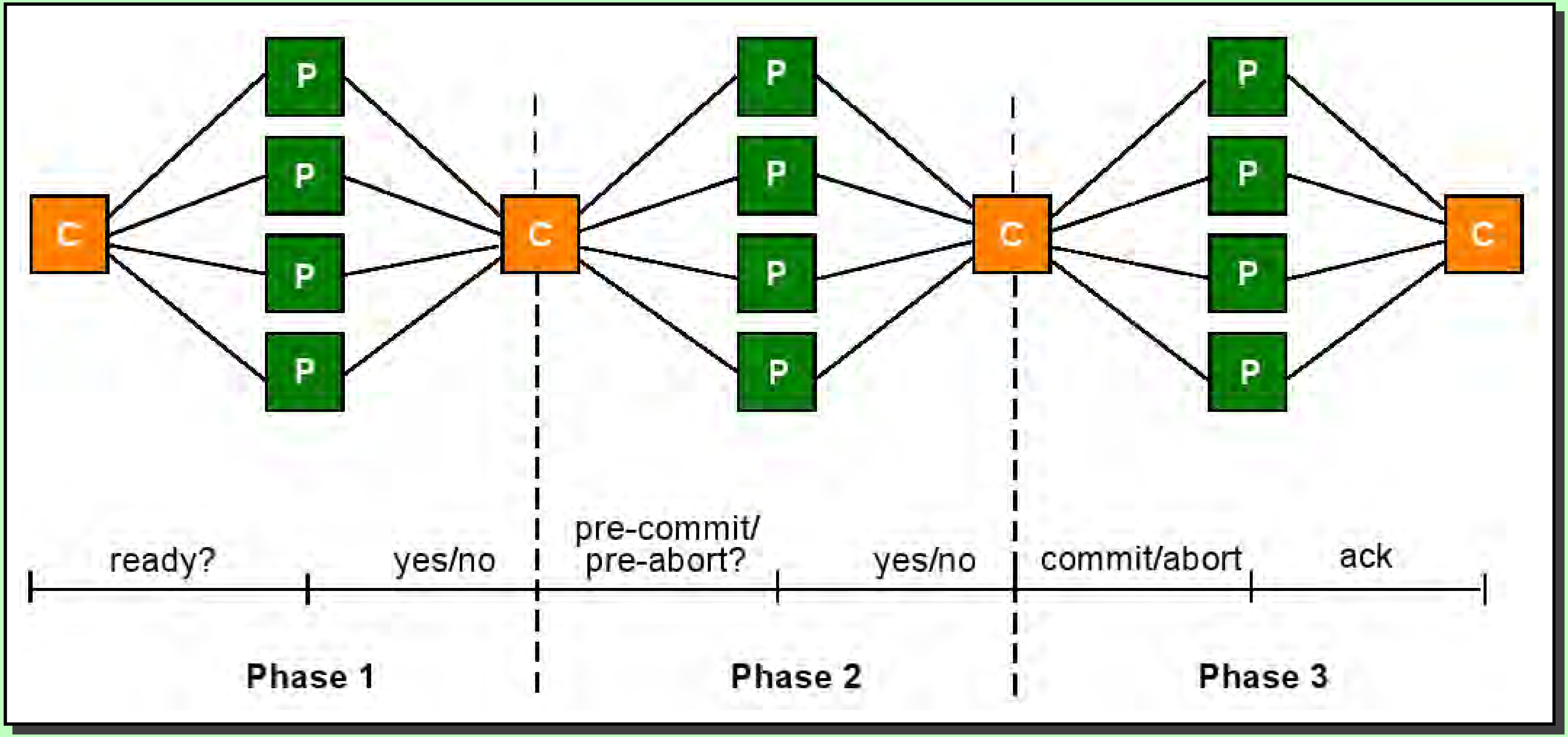
Problems with 2PC Protocol

- A protocol is **non-blocking** if it permits a transaction to terminate at the operational sites without waiting for recovery of the failed site.
  - Significantly improves the response-time of transactions
- 2PC protocol is **blocking**
  - Ready implies that the participant waits for the coordinator
  - If coordinator fails, site is blocked until recovery; independent recovery is not possible
  - The problem is that sites might be in both: commit and abort phases.

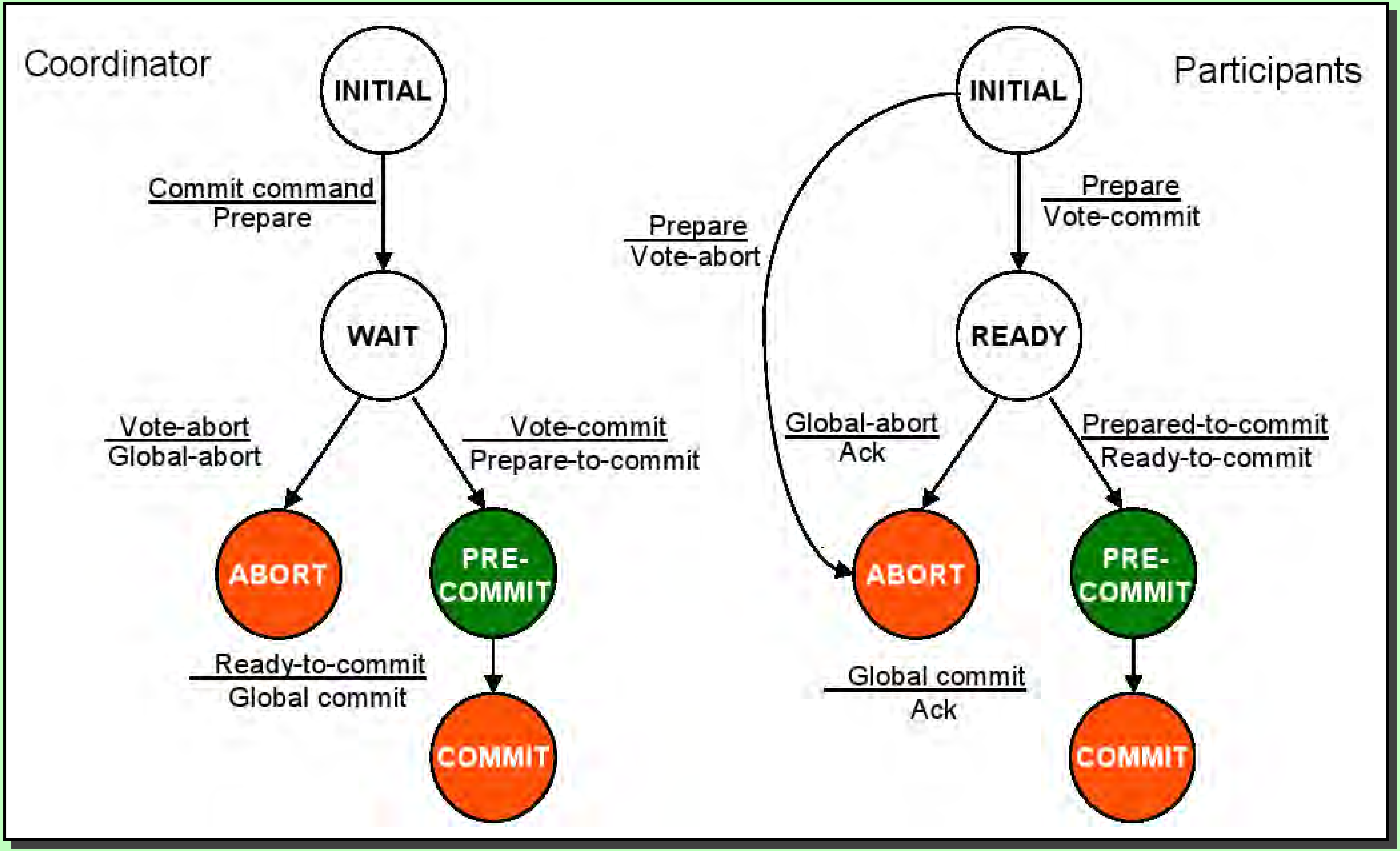
Three Phase Commit Protocol (3PC)

- 3PC is a **non-blocking protocol** when failures are restricted to **single site** failures
- The state transition diagram contains
  - no state which is ”adjacent” to both a commit and an abort state
  - no non-committable state which is ”adjacent” to a commit state
- Adjacent: possible to go from one status to another with a single state transition
- Committable: all sites have voted to commit a transaction (e.g.: COMMIT state)
- Solution: Insert another state between the WAIT (READY) and COMMIT states

Three Phase Commit Protocol (3PC) . . .



Three Phase Commit Protocol (3PC) . . .



Conclusion

- Recovery management enables resilience from certain types of failures and ensures atomicity and durability of transactions
- Local recovery manager (LRM) enables resilience from certain types of failures locally. LRM might employ out-of-place and in-place strategies to deal with updates. In case of the in-place strategy an additional log is used for recovery
- Distributed reliablity protocols are more complicated, in particular the commit, termination, and recovery protocols.
- 2PC protocol first gets participants ready for the transaction (phase 1), and then asks the participants to write the transaction (phase 2). 2PC is a blocking protocol.
- 3PC first gets participants ready for the transaction (phase 1), pre-commits/aborts the transaction (phase 2), and then asks the participants to commit/abort the transaction (phase 3). 3PC is non-blocking.

Concurrency

- **Concurrency control** is the problem of synchronizing concurrent transactions (i.e., order the operations of concurrent transactions) such that the following two properties are achieved:
  - the consistency of the DB is maintained
  - the maximum degree of concurrency of operations is achieved
- Obviously, the serial execution of a set of transaction achieves consistency, if each single transaction is consistent

Conflicts

- **Conflicting operations:** Two operations $O_{ij}(x)$ and $O_{kl}(x)$ of transactions T_i and T_k are in **conflict** iff at least one of the operations is a write, i.e.,
  - $O_{ij} = read(x)$ and $O_{kl} = write(x)$
  - $O_{ij} = write(x)$ and $O_{kl} = read(x)$
  - $O_{ij} = write(x)$ and $O_{kl} = write(x)$
- Intuitively, a conflict between two operations indicates that their order of execution is important.
- Read operations do not conflict with each other, hence the ordering of read operations does not matter.
- **Example:** Consider the following two transactions

$$
\begin{array}{llcll}
T_1: & Read(x) & \qquad & T_2: & Read(x) \\
 & x \leftarrow x+1 & & & x \leftarrow x+1 \\
 & Write(x) & & & Write(x) \\
 & Commit & & & Commit
\end{array}
$$

  - To preserve DB consistency, it is important that the $read(x)$ of one transaction is not between $read(x)$ and $write(x)$ of the other transaction.

Schedules

- A **schedule** (history) specifies a possibly interleaved order of execution of the operations O of a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$, where T_i is specified by a partial order $(\Sigma_i, \prec_i)$. A schedule can be specified as a partial order over O, where
  - $\Sigma_T = \bigcup_{i=1}^{n} \Sigma_i$
  - $\prec_T \supseteq \quad {}_{i=1}^{n} \prec_i$
  - For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} \prec_T O_{kl}$ or $O_{kl} \prec_T O_{ij}$

Schedules . . .

- **Example:** Consider the following two transactions

T_1: $Read(x)$
$x \leftarrow x + 1$
$Write(x)$
$Commit$

T_2: $Read(x)$
$x \leftarrow x + 1$
$Write(x)$
$Commit$

  - A possible schedule over $T =$ $\{T1,T2\}$ can be written as the partial order $S =$ Σ ,{T1 <T}, where

$$\Sigma_T = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$$

$$\begin{aligned} \prec_T = \{&(R_1, W_1), (R_1, C_1), (W_1, C_1), \\ &(R_2, W_2), (R_2, C_2), (W_2, C_2), \\ &(R_2, W_1), (W_1, W_2), \ldots\} \end{aligned}$$

Schedules . . .

- A schedule is **serial** if all transactions in T are executed serially.
- **Example:** Consider the following two transactions

$$\begin{array}{llll} T_1: & Read(x) & T_2: & Read(x) \\ & x \leftarrow x+1 & & x \leftarrow x+1 \\ & Write(x) & & Write(x) \\ & Commit & & Commit \end{array}$$

  - The two serial schedules are $S_1 = \{\Sigma_1, \prec_1\}$ and $S_2 = \{\Sigma_2, \prec_2\}$, where

$$\begin{aligned} \Sigma_1 = \Sigma\ 2 \ \ = \ \ & \{R1\ \ (x), W1\ \ (x), C1\ \ , R2\ \ (x), W2(x), C2\} \\ \prec_1 = & \{(R_1, W_1), (R_1, C_1), (W_1, C_1), (R_2, W_2), (R_2, C_2), (W_2, C_2), \\ & (C_1, R_2), \ldots\} \\ \prec_2 = & \{(R_1, W_1), (R_1, C_1), (W_1, C_1), (R_2, W_2), (R_2, C_2), (W_2, C_2), \\ & (C_2, R_1), \ldots\} \end{aligned}$$

- We will also use the following notation:
  - $\{T_1, T_2\} = \{R_1(x), W_1(x), C_1, R_2(x), W_2(x), C_2\}$
  - $\{T_2, T_1\} = \{R_2(x), W_2(x), C_2, R_1(x), W_1(x), C_1\}$

Serializability

- Two schedules are said to be **equivalent** if they have the same effect on the DB.
- **Conflict equivalence:** Two schedules S_1 and S_2 defined over the same set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ are said to be **conflict equivalent** if for each pair of conflicting operations O_{ij} and O_{kl}, whenever $O_{ij} <_1 O_{kl}$ then $O_{ij} <_2 O_{kl}$.
  - i.e., conflicting operations must be executed in the same order in both transactions.
- A concurrent schedule is said to be **(conflict-)serializable** iff it is conflict equivalent to a serial schedule
- A conflict-serializable schedule can be transformed into a serial schedule by swapping non-conflicting operations
- **Example:** Consider the following two schedules

$$\begin{array}{ll} T_1\text{:} & Read(x) \\ & x \leftarrow x+1 \\ & Write(x) \\ & Write(z) \\ & Commit \end{array} \qquad \begin{array}{ll} T_2\text{:} & Read(x) \\ & x \leftarrow x+1 \\ & Write(x) \\ & Commit \end{array}$$

  - The schedule $\{R_1(x), W_1(x), R_2(x), W_2(x), W_1(z), C_2, C_1\}$ is conflict-equivalent to $\{T_1, T_2\}$ but not to $\{T_2, T_1\}$

Serializability . . .

- The **primary function** of a concurrency controller is to generate a serializable schedule for the execution of pending transactions.
- In a DDBMS two schedules must be considered
  - Local schedule
  - Global schedule (i.e., the union of the local schedules)
- **Serializability** in DDBMS
  - Extends in a straightforward manner to a DDBMS if data is *not replicated*
  - Requires more care if data is *replicated*: It is possible that the local schedules are serializable, but the mutual consistency of the DB is not guaranteed.
    - ∗ Mutual consistency: All the values of all replicated data items are identical
- Therefore, a **serializable global schedule** must meet the following conditions:
  - Local schedules are serializable
  - Two conflicting operations should be in the same relative order in all of the local schedules they appear
    - ∗ Transaction needs to be run on each site with the replicated data item

Serializability . . .

- **Example:** Consider two sites and a data item x which is replicated at both sites.

$$\begin{array}{llll} T_1: & Read(x) & T_2: & Read(x) \\ & x \leftarrow x + 5 & & x \leftarrow x * 10 \\ & Write(x) & & Write(x) \end{array}$$

  - Both transactions need to run on both sites
  - The following two schedules might have been produced at both sites (the order is implicitly given):
    - Site1: $S_1 = \{R_1(x), W_1(x), R_2(x), W_2(x)\}$
    - Site2: $S_2 = \{R_2(x), W_2(x), R_1(x), W_1(x)\}$
  - Both schedules are (trivially) serializable, thus are correct in the local context
  - But they produce different results, thus violate the mutual consistency

Concurrency Control Algorithms

- **Taxonomy** of concurrency control algorithms
  - **Pessimistic** methods assume that many transactions will conflict, thus the concurrent execution of transactions is synchronized early in their execution life cycle
    - Two-Phase Locking (2PL)
      - Centralized (primary site) 2PL
      - Primary copy 2PL
      - Distributed 2PL
    - Timestamp Ordering (TO)
      - Basic TO
      - Multiversion TO
      - Conservative TO
    - Hybrid algorithms
  - **Optimistic** methods assume that not too many transactions will conflict, thus delay the synchronization of transactions until their termination
    - Locking-based
    - Timestamp ordering-based

Locking Based Algorithms

- **Locking-based concurrency algorithms** ensure that data items shared by conflicting operations are accessed in a mutually exclusive way. This is accomplished by associating a “lock” with each such data item.
- Two types of **locks** (lock modes)
  - **read lock** (rl) – also called **shared** lock
  - **write lock** (wl) – also called **exclusive** lock
- **Compatibility matrix** of locks

	$rl\mathrm{i}(x)$	$wl\mathrm{i}(x)$
$rl_j(x)$	compatible	not compatible
$wl_j(x)$	not compatible	not compatible

- General locking algorithm
  1. Before using a data item x, transaction requests lock for x from the lock manager
  2. If x is already locked and the existing lock is incompatible with the requested lock, the transaction is delayed
  3. Otherwise, the lock is granted

Locking Based Algorithms

- **Example:** Consider the following two transactions

T_1:	$Read(x)$	T_2:	$Read(x)$
	$x \leftarrow x+1$		$x \leftarrow x*2$
	$Write(x)$		$Write(x)$
	$Read(y)$		$Read(y)$
	$y \leftarrow y-1$		$y \leftarrow y*2$
	$Write(y)$		$Write(y)$

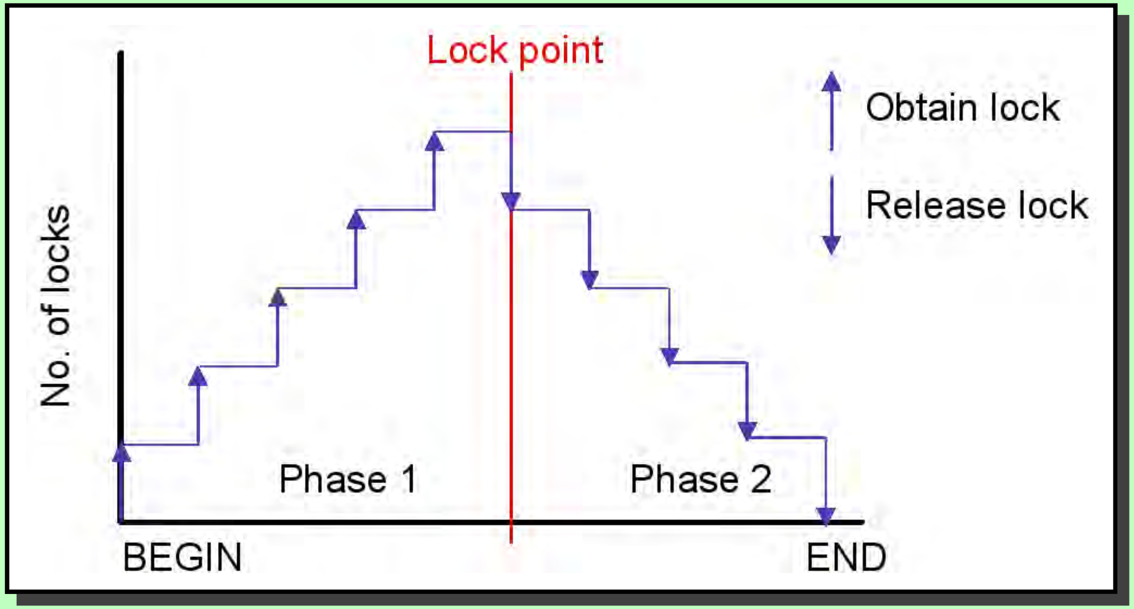
  - The following schedule is a valid locking-based schedule ($lr\ (x)$ indicates the release of a lock on x):

$$\begin{aligned} S = \{&wl_1(x), R_1(x), W_1(x), lr_1(x)\\ &wl_2(x), R_2(x), W_2(x), lr_2(x)\\ &wl_2(y), R_2(y), W_2(y), lr_2(y)\\ &wl_1(y), R_1(y), W_1(y), lr_1(y)\}\end{aligned}$$

  - However, S is not serializable
    - $*$ S cannot be transformed into a serial schedule by using only non-conflicting swaps
    - $*$ The result is different from the result of any serial execution

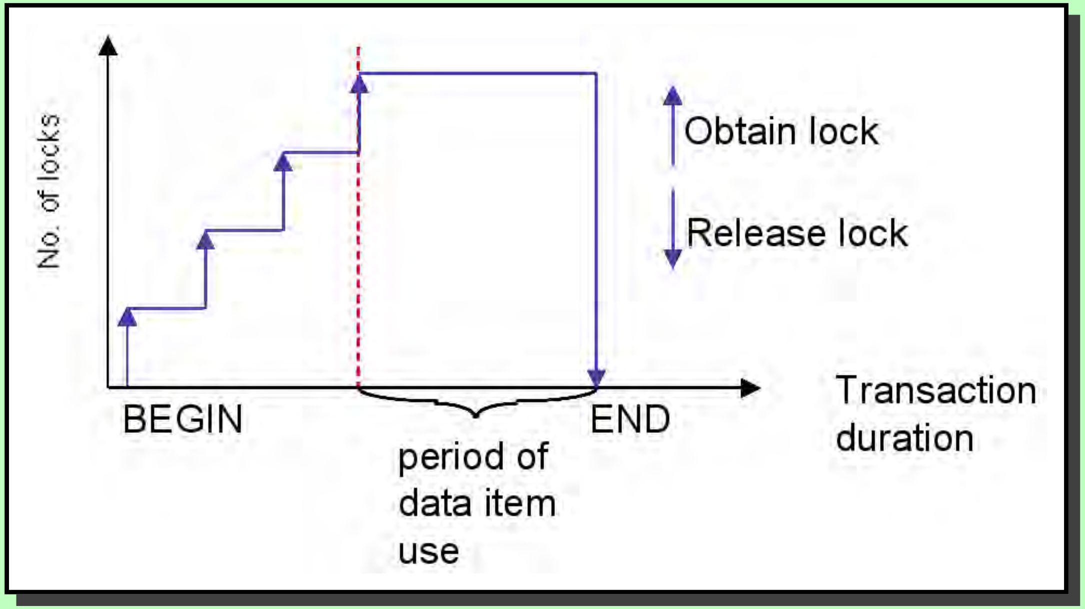
Two-Phase Locking (2PL)

- **Two-phase locking** protocol
  - Each transaction is executed in two phases
    - ∗ **Growing phase:** the transaction obtains locks
    - ∗ **Shrinking phase:** the transaction releases locks
  - The **lock point** is the moment when transitioning from the growing phase to the shrinking phase



Two-Phase Locking (2PL) . . .

- **Properties** of the 2PL protocol
  - Generates **conflict-serializable** schedules
  - But schedules may cause **cascading aborts**
    - ∗ If a transaction aborts after it releases a lock, it may cause other transactions that have accessed the unlocked data item to abort as well
- **Strict 2PL locking** protocol
  - Holds the locks till the end of the transaction
  - Cascading aborts are avoided



Two-Phase Locking (2PL) . . .

- **Example:** The schedule S of the previous example is not valid in the 2PL protocol:

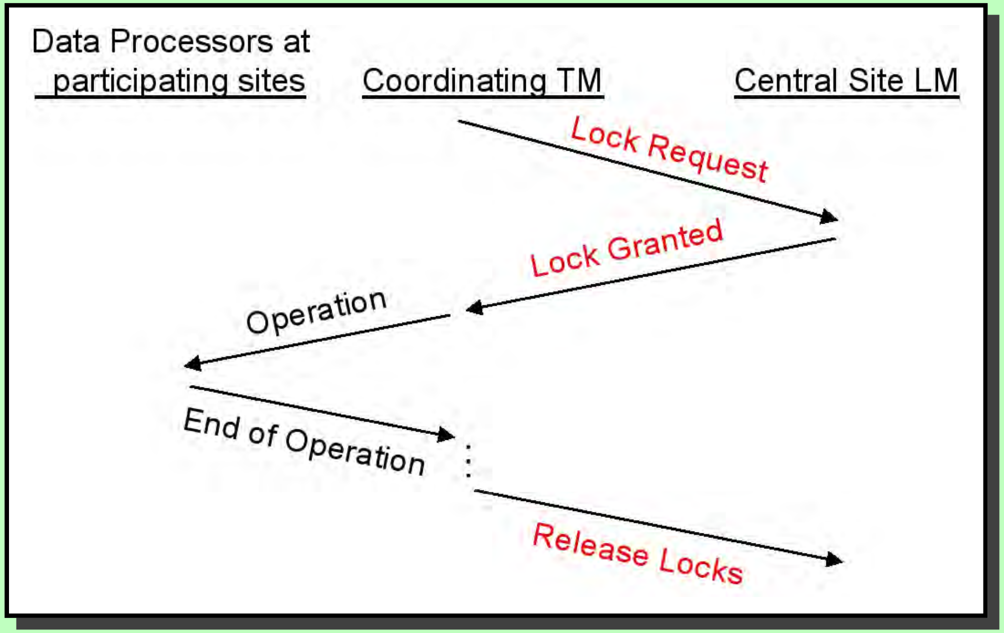
$$\begin{aligned} S = \{ & wl_1(x), R_1(x), W_1(x), lr_1(x) \\ & wl_2(x), R_2(x), W_2(x), lr_2(x) \\ & wl_2(y), R_2(y), W_2(y), lr_2(y) \\ & wl_1(y), R_1(y), W_1(y), lr_1(y)\} \end{aligned}$$

  - e.g., after $lr\ \ (x)$ (in line 1) transaction T cannot request the lock $wl\ \ (y)$ (in line 4).
  - Valid schedule in the 2PL protocol

$$\begin{aligned} S = \{ & wl_1(x), R_1(x), W_1(x), \\ & wl_1(y), R_1(y), W_1(y), lr_1(x), lr_1(y) \\ & wl_2(x), R_2(x), W_2(x), \\ & wl_2(y), R_2(y), W_2(y), lr_2(x), lr_2(y)\} \end{aligned}$$

2PL for DDBMS

- Various extensions of the 2PL to DDBMS
- **Centralized 2PL**
  - A single site is responsible for the lock management, i.e., one lock manager for the whole DDBMS
  - Lock requests are issued to the lock manager
  - Coordinating transaction manager (TM at site where the transaction is initiated) can make all locking requests on behalf of local transaction managers



- Advantage: Easy to implement
- Disadvantages: Bottlenecks and lower reliability
- Replica control protocol is additionally needed if data are replicated (see also primary copy 2PL)

2PL for DDBMS . . .

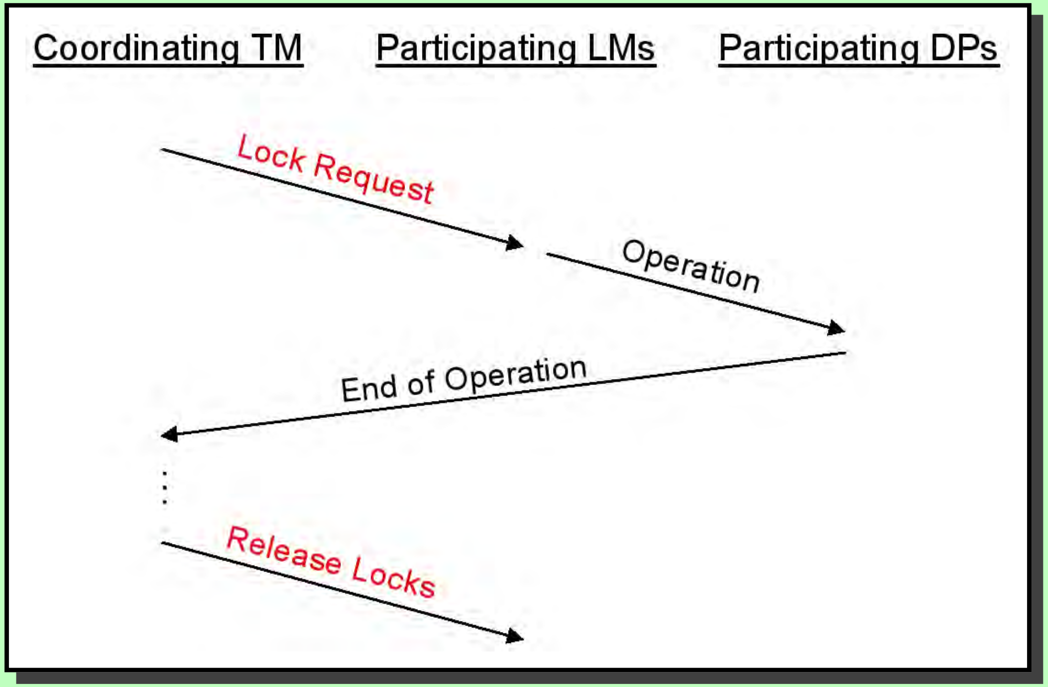
- **Primary copy 2PL**
  - Several lock managers are distributed to a number of sites
  - Each lock manager is responsible for managing the locks for a set of data items
  - For replicated data items, one copy is chosen as primary copy, others are slave copies
  - Only the primary copy of a data item that is updated needs to be write-locked
  - Once primary copy has been updated, the change is propagated to the slaves
- Advantages
  - Lower communication costs and better performance than the centralized 2PL
- Disadvantages
  - Deadlock handling is more complex

2PL for DDBMS . . .

- **Distributed 2PL**
  - Lock managers are distributed to all sites
  - Each lock manager responsible for locks for data at that site
  - If data is not replicated, it is equivalent to primary copy 2PL
  - If data is replicated, the Read-One-Write-All (ROWA) replica control protocol is implemented
    Read(x): Any copy of a replicated item x can be read by obtaining a read lock on the copy
    * $Write(x)$: All copies of x must be write-locked before x can be updated
- Disadvantages
  - Deadlock handling more complex
  - Communication costs higher than primary copy 2PL

2PL for DDBMS . . .

- Communication structure of the distributed 2PL
  - The coordinating TM sends the lock request to the lock managers of all participating sites
  - The LMs pass the operations to the data processors
  - The end of the operation is signaled to the coordinating TM



Timestamp Ordering

- **Timestamp-ordering** based algorithms do not maintain serializability by mutual exclusion, but select (a priori) a serialization order and execute transactions accordingly.
  - Transaction T_i is assigned a globally unique timestamp $ts(T_i)$
  - Conflicting operations O_{ij} and O_{kl} are resolved by timestamp order, i.e., O_{ij} is executed before O_{kl} iff $ts(T_i) < ts(T_k)$.
- To allow for the scheduler to check whether operations arrive in correct order, 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 write on x
- Then the scheduler has to perform the following checks:
  - Read operation, $R_i(x)$:
    - If $ts(T_i) < wts(x)$: T_i attempts to read overwritten data; abort T_i
    - If $ts(T_i) \geq wts(x)$: the operation is allowed and $rts(x)$ is updated
  - Write operations, $W_i(x)$:
    - If $ts(T_i) < rts(x)$: x was needed before by other transaction; abort T_i
    - If $ts(T_i) < wts(x)$: T_i writes an obsolete value; abort T_i
    - Otherwise, execute $W_i(x)$

Timestamp Ordering . . .

- Generation of **timestamps** (TS) in a distributed environment
  - TS needs to be locally and globally **unique** and **monotonically increasing**
  - System clock, incremental event counter at each site, or global counter are unsuitable (difficult to maintain)
  - Concatenate local timestamp/counter with a unique site identifier: $<$*local timestamp, site identifier*$>$
    - site identifier is in the least significant position in order to distinguish only if the local timestamps are identical
- Schedules generated by the basic TO protocol have the following **properties**:
  - Serializable
  - Since transactions never wait (but are rejected), the schedules are deadlock-free
  - The price to pay for deadlock-free schedules is the potential restart of a transaction several times

Timestamp Ordering . . .

- Basic timestamp ordering is “**aggressive**”: It tries to execute an operation as soon as it receives it
- **Conservative** timestamp ordering delays each operation until there is an assurance that it will not be restarted, i.e., that no other transaction with a smaller timestamp can arrive
  - For this, the operations of each transaction are buffered until an ordering can be established so that rejections are not possible
- If this condition can be guaranteed, the scheduler will never reject an operation
- However, this delay introduces the possibility for deadlocks

Timestamp Ordering . . .

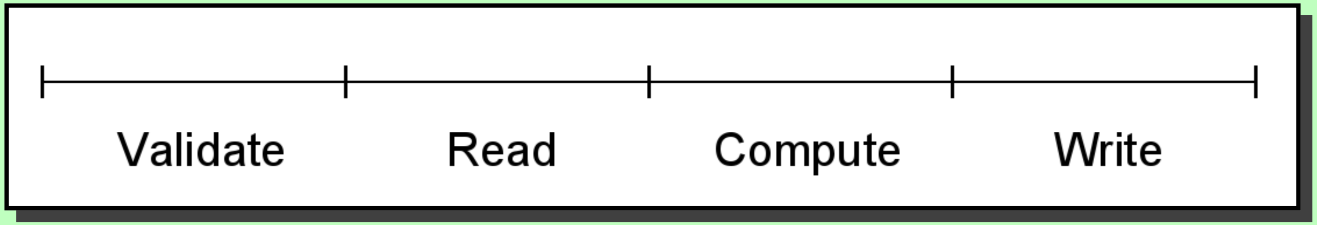
- **Multiversion timestamp ordering**
  - Write operations do not modify the DB; instead, a new version of the data item is created: $x_1, x_2, \ldots, x_n$
  - $R_i(x)$ is always successful and is performed on the appropriate version of x, i.e., the version of x (say x) such that $wts(x\mathrm{w}\ \ )$ is the largest timestamp less than $ts(T_i)$
  - $W_i(x)$ produces a new version $x$wwith $ts(x\mathrm{w}\ \ ) = ts(T\mathrm{i}\ )$ if the scheduler has not yet processed any $R_j(x_r)$ on a version x_r such that

    $$ts(T_i) < rts(x_r)$$

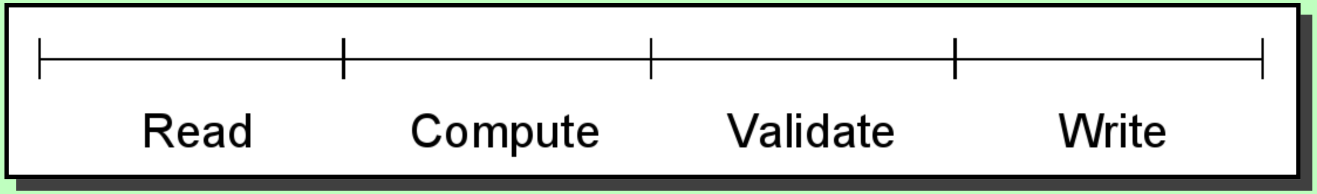
    i.e., the write is too late.
  - Otherwise, the write is rejected.

Timestamp Ordering . . .

- The previous concurrency control algorithms are pessimistic



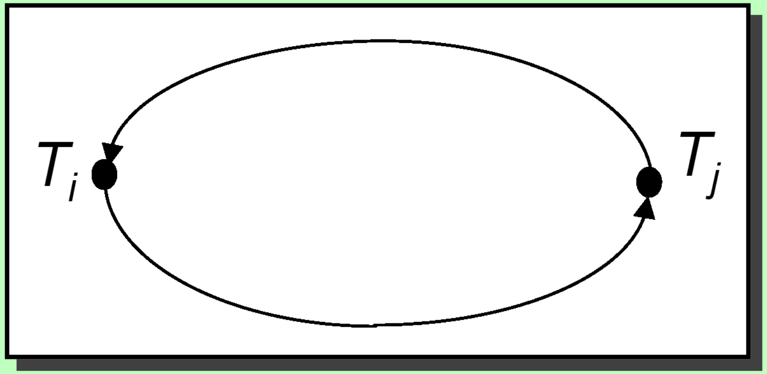
- **Optimistic concurrency control algorithms**
  - Delay the validation phase until just before the write phase
  - T_i run independently at each site on local copies of the DB (without updating the DB)
  - Validation test then checks whether the updates would maintain the DB consistent:
    - ∗ If yes, all updates are performed
    - ∗ If one fails, all T_i's are rejected



- Potentially allow for a higher level of concurrency

Deadlock Management

- **Deadlock:** A set of transactions is in a deadlock situation if several transactions wait for each other. A deadlock requires an outside intervention to take place.
- Any locking-based concurrency control algorithm may result in a deadlock, since there is mutual exclusive access to data items and transactions may wait for a lock
- Some TO-based algorihtms that require the waiting of transactions may also cause deadlocks
- A **Wait-for Graph** (WFG) is a useful tool to identify deadlocks
  - The nodes represent transactions
  - An edge from T_i to T_j indicates that T_i is waiting for T_j
  - If the WFG has a cycle, we have a deadlock situation

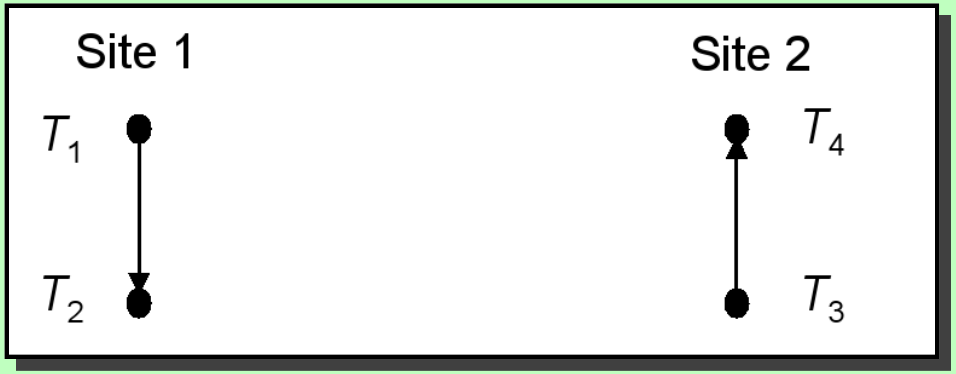


Deadlock Management . . .

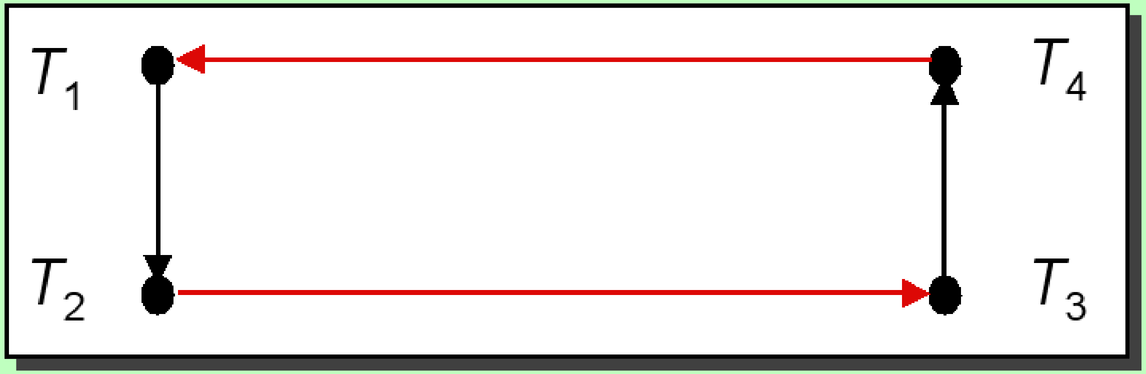
- Deadlock management in a DDBMS is more complicate, since lock management is not centralized
- We might have **global deadlock**, which involves transactions running at different sites
- A Local Wait-for-Graph (LWFG) may not show the existence of global deadlocks
- A Global Wait-for Graph (GWFG), which is the union of all LWFGs, is needed

Deadlock Management . . .

- **Example:** Assume T_1 and T_2 run at site 1, T_3 and T_4 run at site 2, and the following wait-for relationships between them: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$. This deadlock cannot be detected by the LWFGs, but by the GWFG which shows intersite waiting.
  - Local WFG:



  - Global WFG:



Deadlock Prevention

- **Deadlock prevention**: Guarantee that deadlocks never occur
  - Check transaction when it is initiated, and start it only if all required resources are available.
  - All resources which may be needed by a transaction must be predeclared
- Advantages
  - No transaction rollback or restart is involved
  - Requires no run-time support
- Disadvantages
  - Reduced concurrency due to pre-allocation
  - Evaluating whether an allocation is safe leads to added overhead
  - Difficult to determine in advance the required resources

Deadlock Avoidance

- **Deadlock avoidance:** Detect potential deadlocks in advance and take actions to ensure that a deadlock will not occur. Transactions are allowed to proceed unless a requested resource is unavailable
- Two different approaches:
  - **Ordering of data items**: Order data items and sites; locks can only be requested in that order (e.g., graph-based protocols)
  - **Prioritize transactions:** Resolve deadlocks by aborting transactions with higher or lower priority. The following schemes assume that Ti requests a lock hold by Tj:
    Wait-Die Scheme: if ts(Ti) < ts(Tj) then Ti waits else Ti dies
    Wound-Wait Scheme: if ts(Ti) < ts(Tj) then Tj wounds (aborts) else Ti waits
- Advantages
  - More attractive than prevention in a database environment
  - Transactions are not required to request resources a priori
- Disadvantages
  - Requires run time support

Deadlock Detection

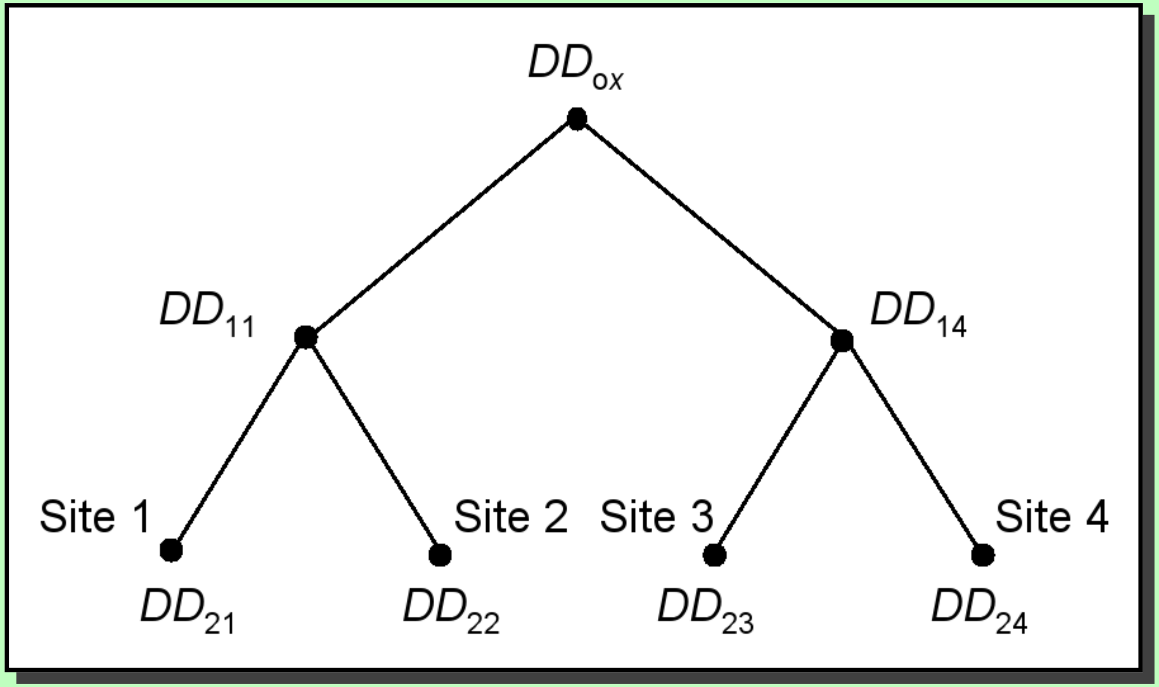
- **Deadlock detection and resolution:** Transactions are allowed to wait freely, and hence to form deadlocks. Check global wait-for graph for cycles. If a deadlock is found, it is resolved by aborting one of the involved transactions (also called the victim).
- Advantages
  - Allows maximal concurrency
  - The most popular and best-studied method
- Disadvantages
  - Considerable amount of work might be undone
- Topologies for deadlock detection algorithms
  - Centralized
  - Distributed
  - Hierarchical

Deadlock Detection . . .

- **Centralized deadlock detection**
  - One site is designated as the deadlock detector (DDC) for the system
  - Each scheduler periodically sends its LWFG to the central site
  - The site merges the LWFG to a GWFG and determines cycles
  - If one or more cycles exist, DDC breaks each cycle by selecting transactions to be rolled back and restarted
- This is a reasonable choice if the concurrency control algorithm is also centralized

Deadlock Detection . . .

- **Hierarchical deadlock detection**
  - Sites are organized into a hierarchy
  - Each site sends its LWFG to the site above it in the hierarchy for the detection of deadlocks
  - Reduces dependence on centralized detection site



Deadlock Detection . . .

- **Distributed deadlock detection**
  - Sites cooperate in deadlock detection
  - The local WFGs are formed at each site and passed on to the other sites.
  - Each local WFG is modified as follows:
    - Since each site receives the potential deadlock cycles from other sites, these edges are added to the local WFGs
    - i.e., the waiting edges of the local WFG are joined with waiting edges of the external WFGs
  - Each local deadlock detector looks for two things:
    - If there is a cycle that does not involve the external edge, there is a local deadlock which can be handled locally
    - If there is a cycle involving external edges, it indicates a (potential) global deadlock.

Conclusion

- Concurrency orders the operations of transactions such that two properties are achieved: (i) the database is always in a consistent state and (ii) the maximum concurrency of operations is achieved
- A schedule is some order of the operations of the given transactions. If a set of transactions is executed one after the other, we have a serial schedule.
- There are two main groups of serializable concurrency control algorithms: locking based and timestamp based
- A transaction is deadlocked if two or more transactions are waiting for each other. A Wait-for graph (WFG) is used to identify deadlocks
- Centralized, distributed, and hierarchical schemas can be used to identify deadlocks