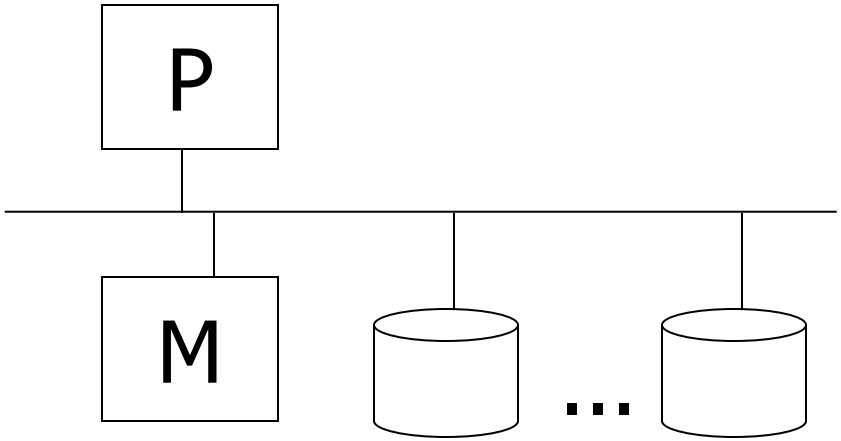
Distributed Data Management

Alex Libis , Ph.D

Based on Department of Computer Science UC Irvine

Centralized DB systems



Software:

Application
SQL Front End
Query Processor
Transaction Proc.
File Access

- Simplifications:
  - single front end
  - one place to keep data, locks
  - if processor fails, system fails, ...

Distributed Database Systems

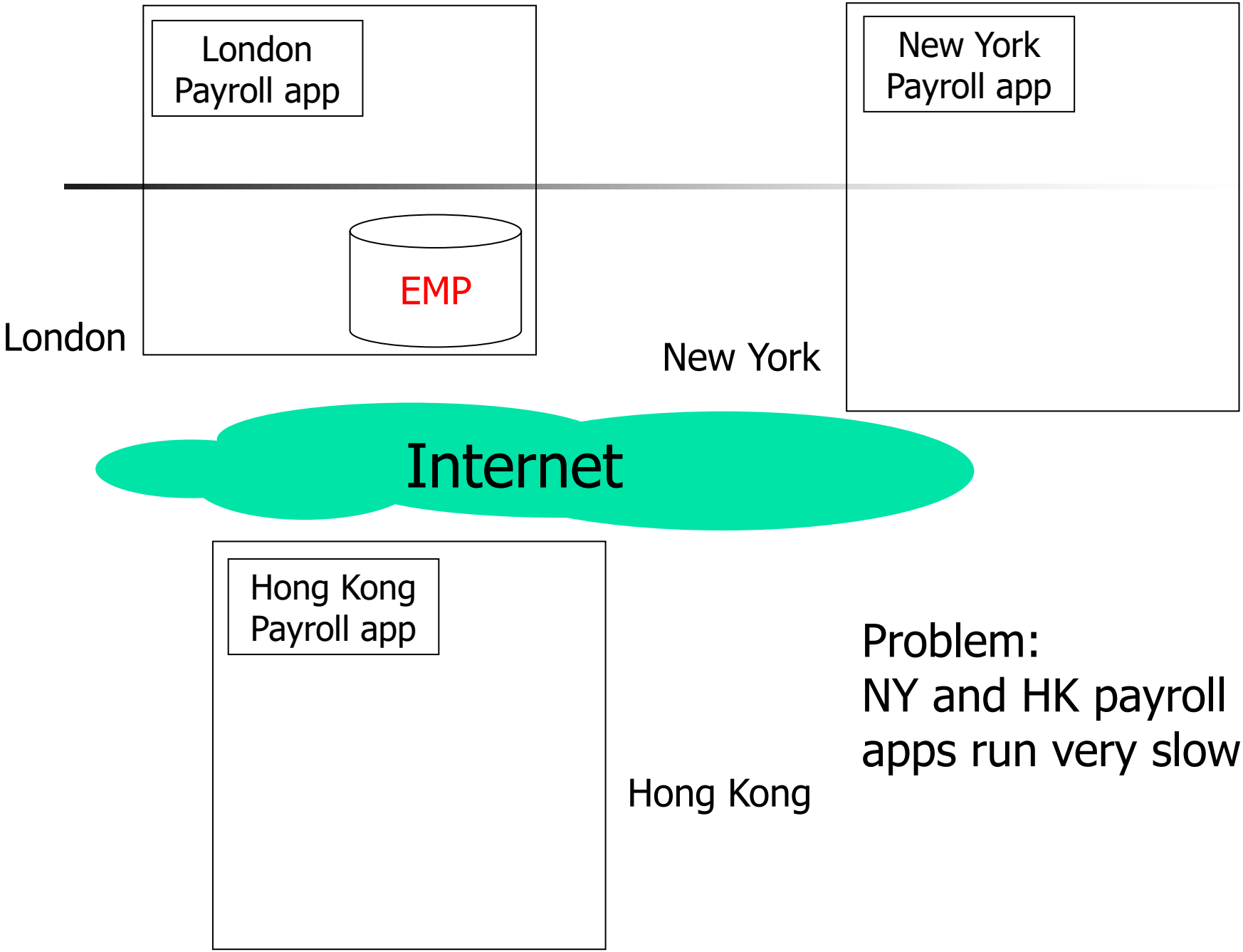
- Multiple processors (+ memories)
- Heterogeneity and autonomy of "components"

Why do we need Distributed Databases?

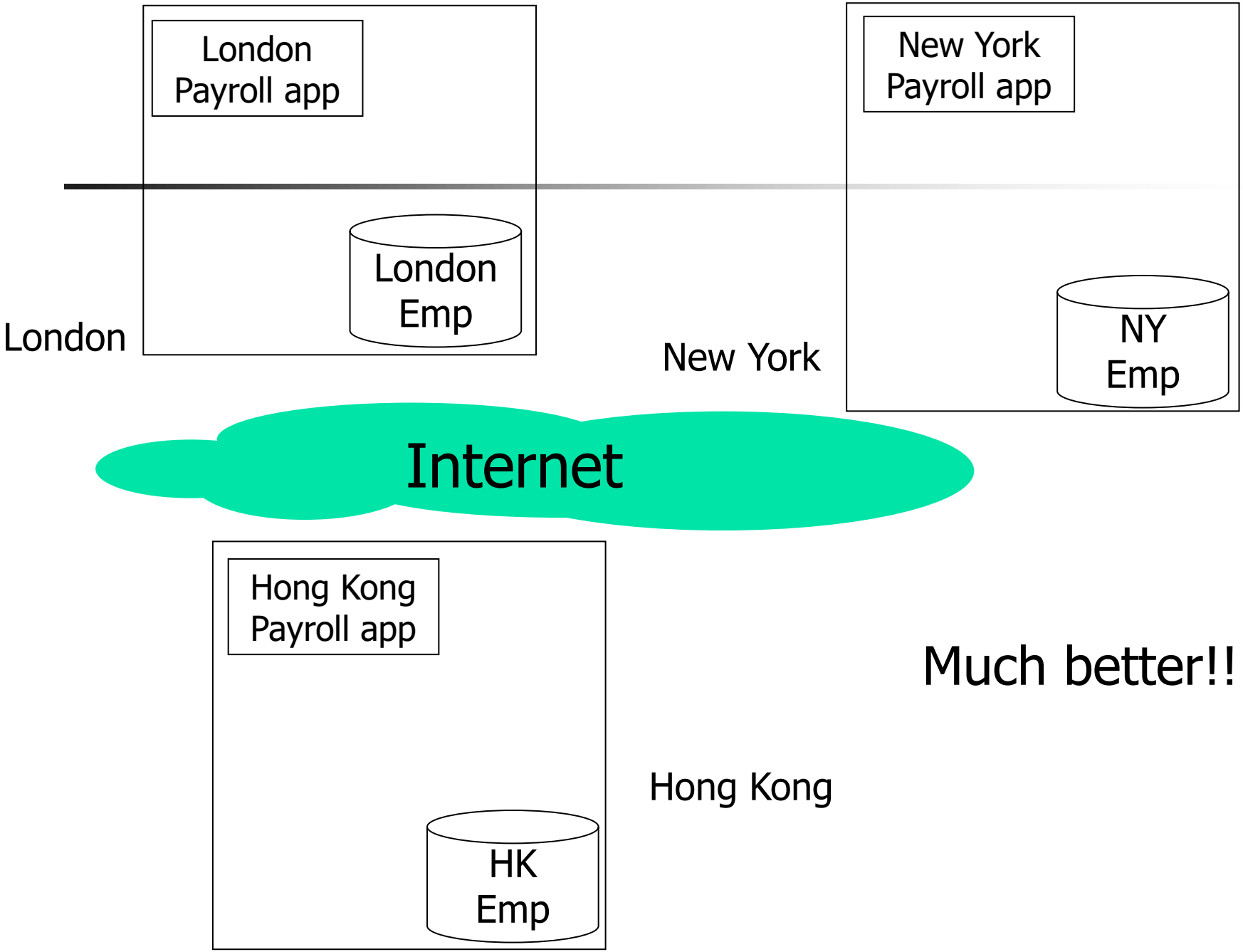
- Example: IBM has offices in London, New York, and Hong Kong.
- Employee data:
  - EMP(ENO, NAME, TITLE, SALARY, …)
- Where should the employee data table reside?

IBM Data Access Pattern

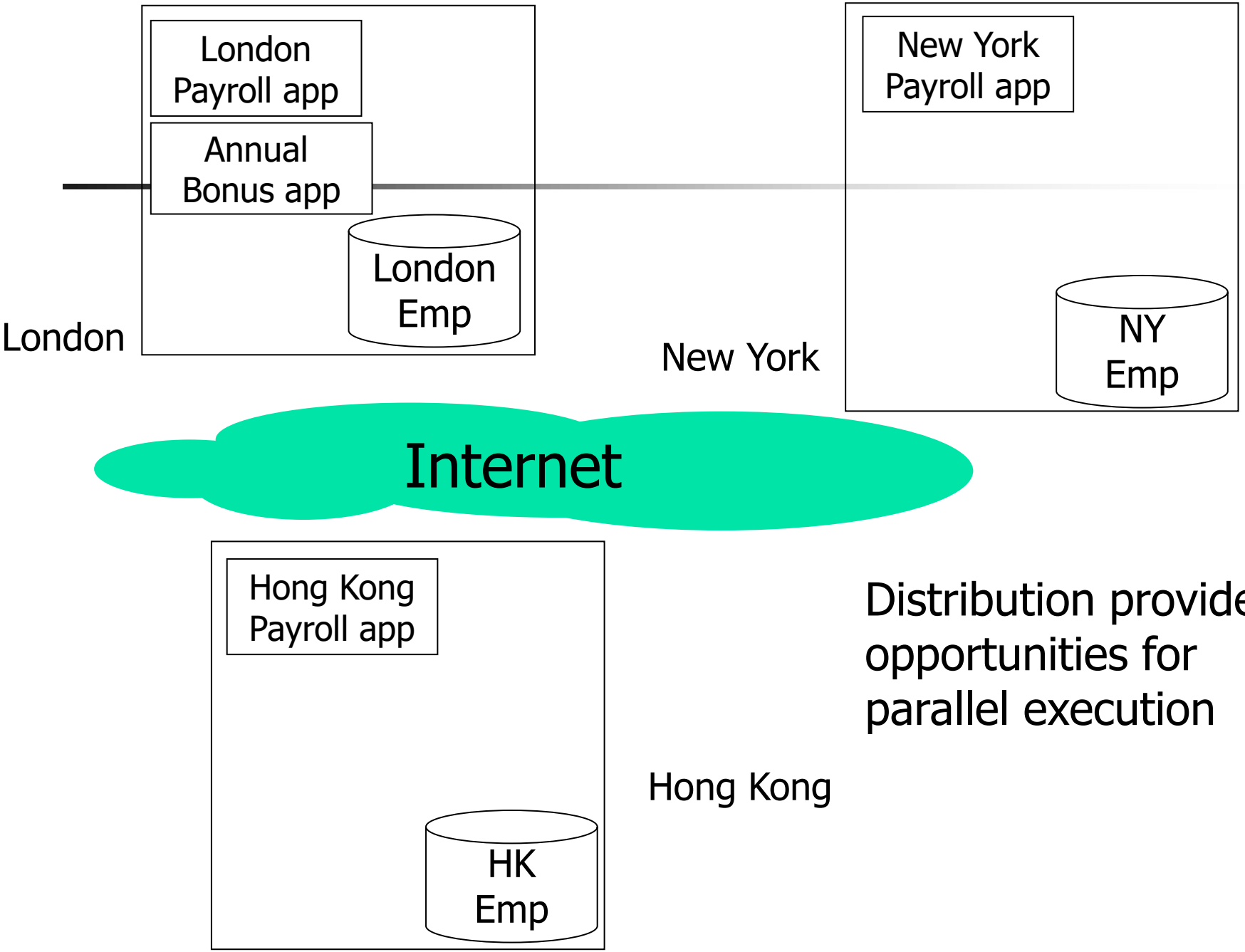
- Mostly, employee data is managed at the office where the employee works
  - E.g., payroll, benefits, hire and fire
- Periodically, IBM needs consolidated access to employee data
  - E.g., IBM changes benefit plans and that affects all employees.
  - E.g., Annual bonus depends on global net profit.



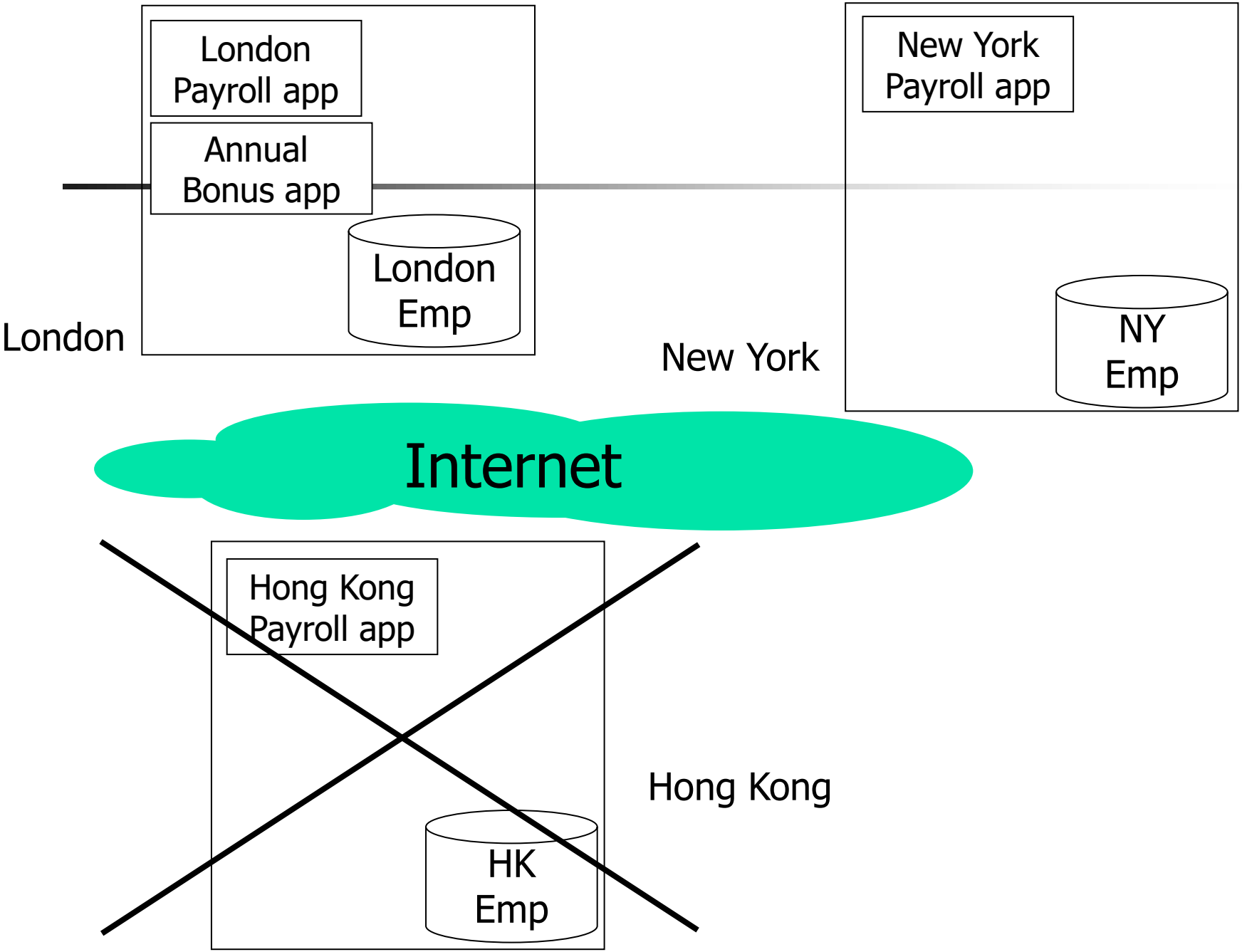
Problem:
NY and HK payroll apps run very slowly!

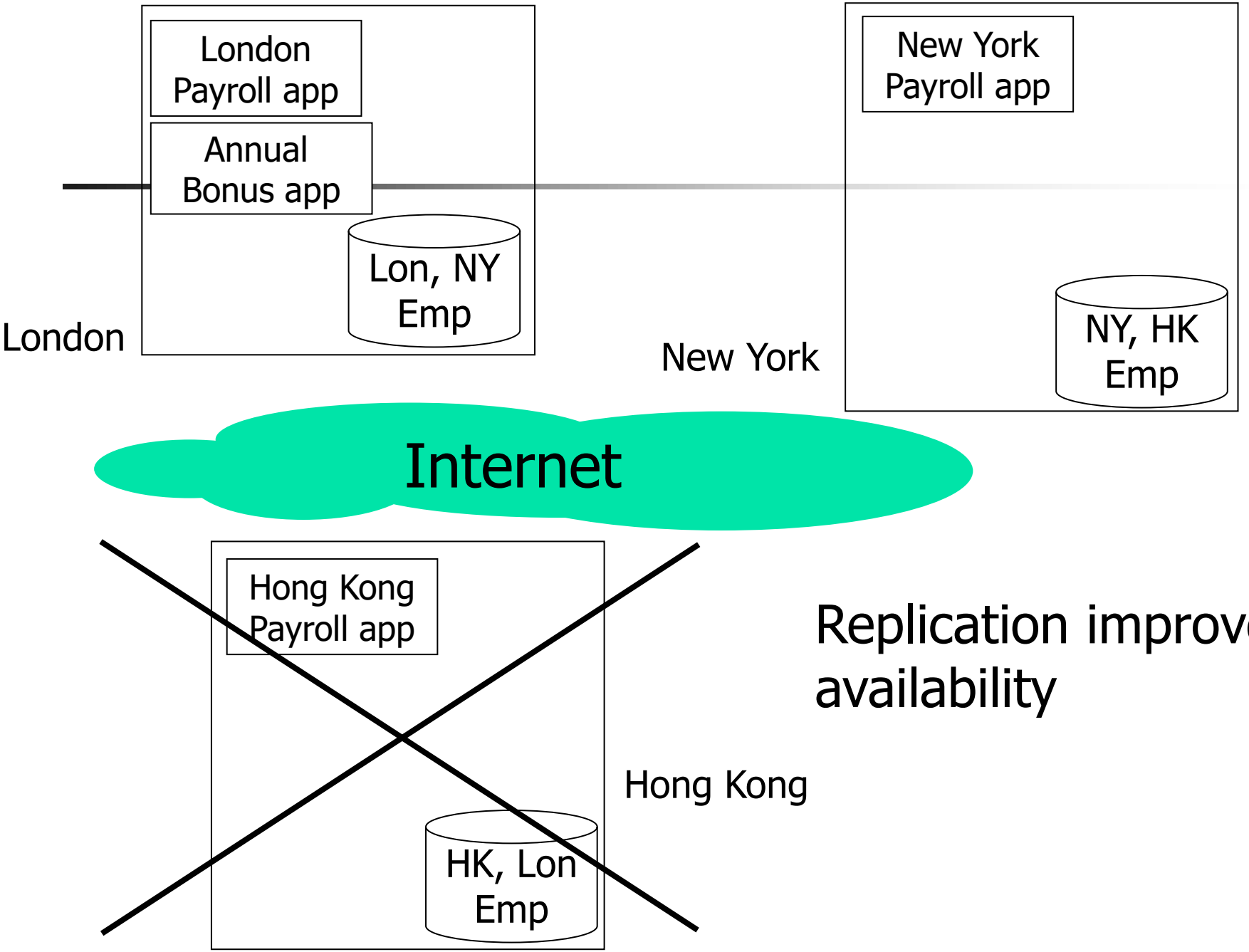

London
Payroll app
London
Emp
London
New York
Payroll app
NY
Emp
New York
Internet
Hong Kong
Payroll app
HK
Emp
Hong Kong

Much better!!



Distribution provides opportunities for parallel execution


London
Payroll app
Annual
Bonus app
London
Emp
London
New York
Payroll app
NY
Emp
New York
Internet
Hong Kong
Payroll app
HK
Emp
Hong Kong



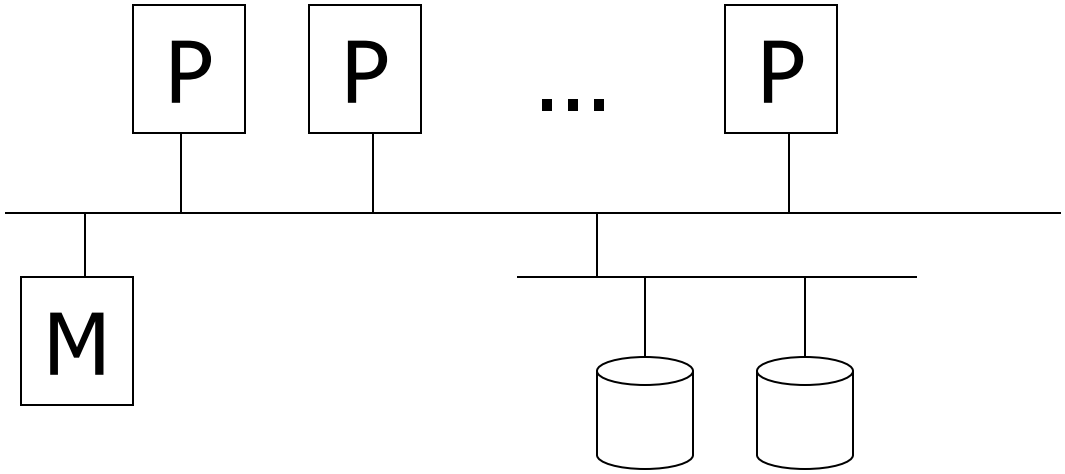
Replication improves availability

Homogeneous Distributed Databases

- **In a homogeneous distributed database**
  - All sites have identical software
  - Are aware of each other and agree to cooperate in processing user requests.
  - Each site surrenders part of its autonomy in terms of right to change schemas or software
  - Appears to user as a single system
- **In a heterogeneous distributed database**
  - Different sites may use different schemas and software
    - Difference in schema is a major problem for query processing
    - Difference in software is a major problem for transaction processing
  - Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing

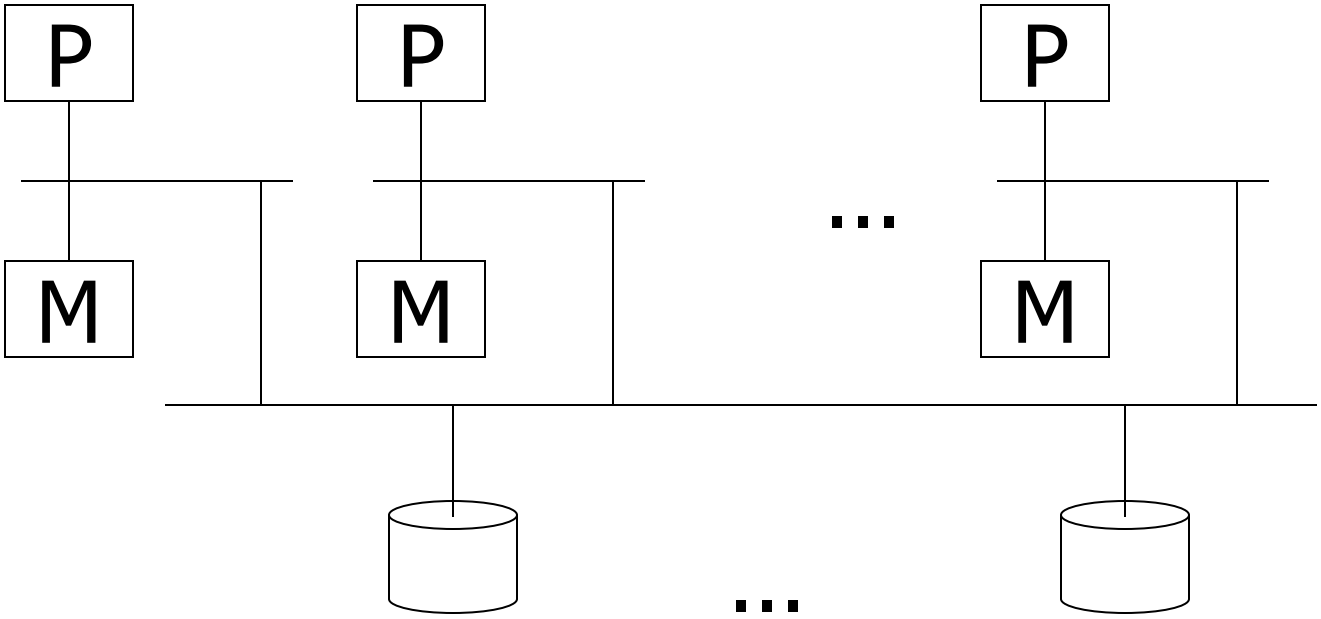
DB architectures

(1) Shared memory



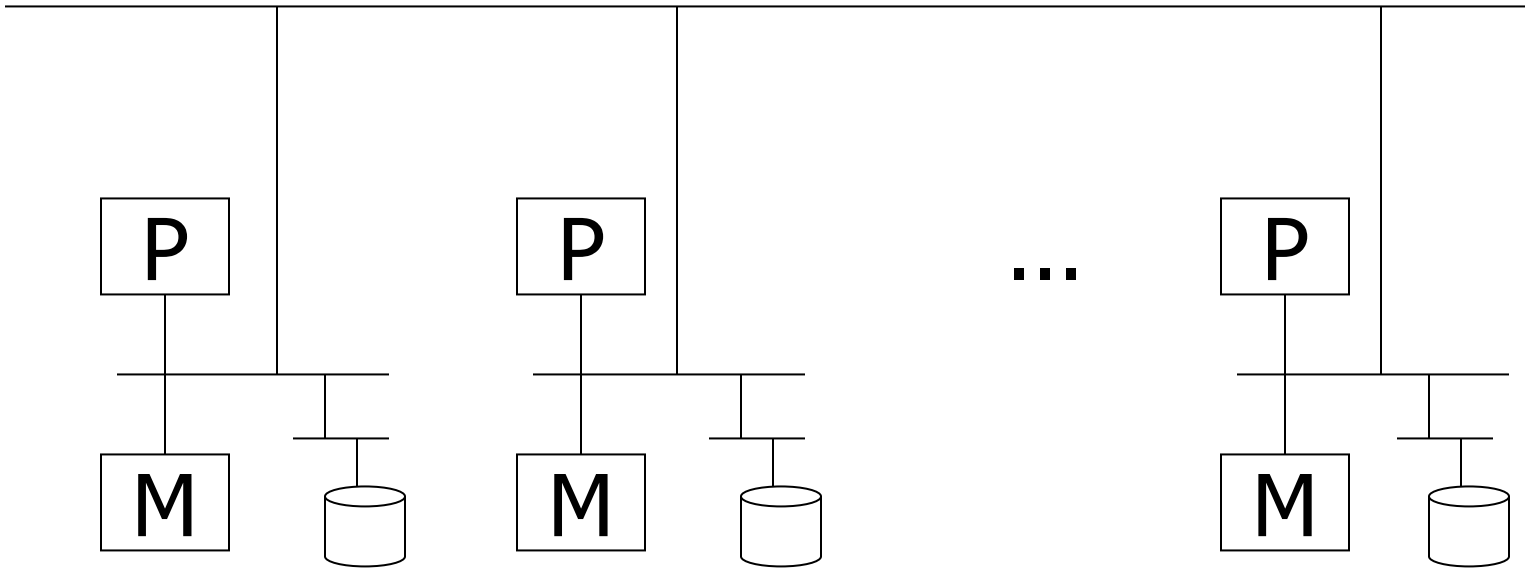
DB architectures

(2) Shared disk



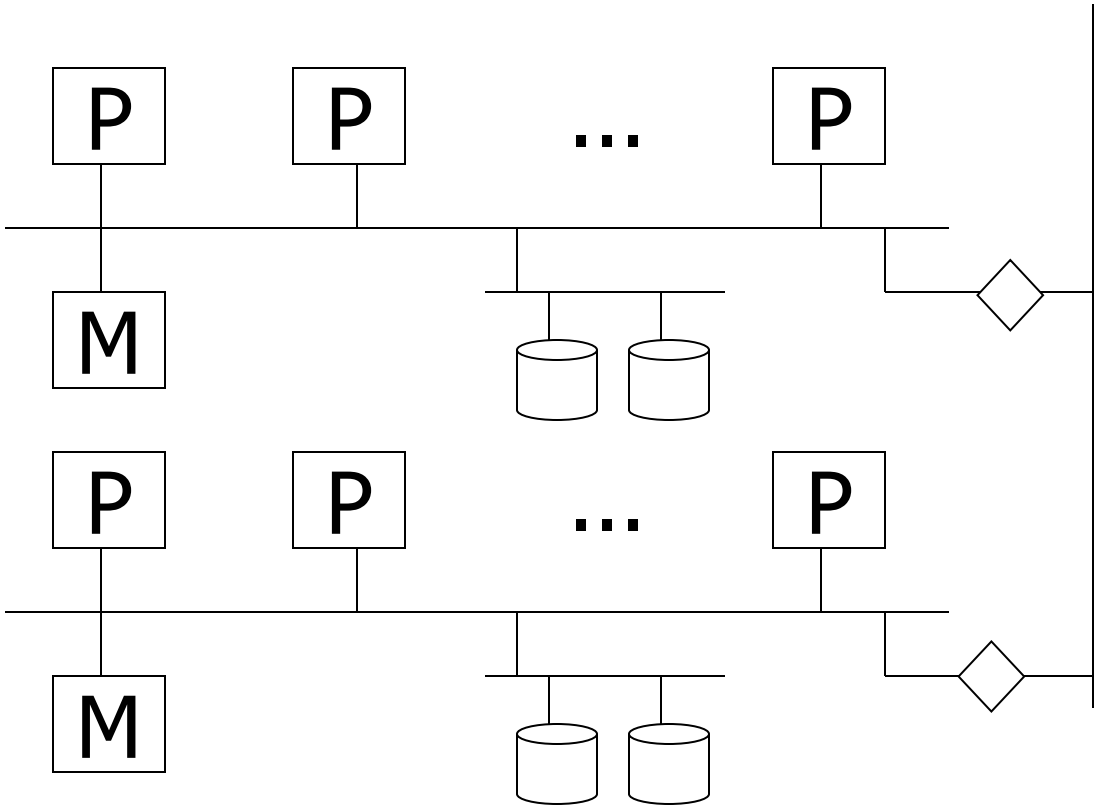
DB architectures

(3) Shared nothing



DB architectures

(4) Hybrid example – Hierarchical or Clustered



Issues for selecting architecture

- Reliability
- Scalability
- Geographic distribution of data
- Performance
- Cost

Parallel or distributed DB system?

- More similarities than differences!

- Typically, parallel DBs:
  - Fast interconnect
  - Homogeneous software
  - High performance is goal
  - Transparency is goal

- Typically, distributed DBs:
  - Geographically distributed
  - Data sharing is goal (may run into heterogeneity, autonomy)
  - Disconnected operation possible

Distributed Database Challenges

- Distributed Database Design
  - Deciding what data goes where
  - Depends on data access patterns of major applications
  - Two subproblems:
    - Fragmentation: partition tables into fragments
    - Allocation: allocate fragments to nodes

Distributed Data Storage

- Assume relational data model
- **Replication**
  - System maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance.
- **Fragmentation**
  - Relation is partitioned into several fragments stored in distinct sites
- Replication and fragmentation can be combined
  - Relation is partitioned into several fragments: system maintains several identical replicas of each such fragment.

Data Replication

- A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites.
- Full replication of a relation is the case where the relation is stored at all sites.
- Fully redundant databases are those in which every site contains a copy of the entire database.

Data Replication (Cont.)

- Advantages of Replication
  - **Availability**: failure of site containing relation *r* does not result in unavailability of *r* is replicas exist.
  - **Parallelism**: queries on *r* may be processed by several nodes in parallel.
  - **Reduced data transfer**: relation *r* is available locally at each site containing a replica of *r*.
- Disadvantages of Replication
  - Increased cost of updates: each replica of relation *r* must be updated.
  - Increased complexity of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.
    - One solution: choose one copy as **primary copy** and apply concurrency control operations on primary copy

Data Fragmentation

- Division of relation r into fragments r_1, r_2, …, r_n which contain sufficient information to reconstruct relation r.
- **Horizontal fragmentation**: each tuple of *r* is assigned to one or more fragments
- **Vertical fragmentation**: the schema for relation *r* is split into several smaller schemas
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property.
  - A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key.
- Example : relation account with following schema
- *Account* = (*branch_name*, *account_number, balance*)

Horizontal Fragmentation of *account* Relation

branch_name	*account_number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$$account_1 = \sigma_{branch\_name=\text{"Hillside"}}(account)$$

branch_name	*account_number*	*balance*
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

$$account_2 = \sigma_{branch\_name=\text{"Valleyview"}}(account)$$

Vertical Fragmentation of *employee_info* Relation

branch_name	*customer_name*	*tuple_id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$deposit_1 = \Pi_{branch\_name,\ customer\_name,\ tuple\_id}(employee\_info)$

account_number	*balance*	*tuple_id*
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

$deposit_2 = \Pi_{account\_number,\ balance,\ tuple\_id}(employee\_info)$

Advantages of Fragmentation

- Horizontal:
  - allows parallel processing on fragments of a relation
  - allows a relation to be split so that tuples are located where they are most frequently accessed
- Vertical:
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation
- Vertical and horizontal fragmentation can be mixed.
  - Fragments may be successively fragmented to an arbitrary depth.

Data Transparency

- **Data transparency**: Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system
- Consider transparency issues in relation to:
  - Fragmentation transparency
  - Replication transparency
  - Location transparency

Naming of Data Items - Criteria

1. Every data item must have a system-wide unique name.
2. It should be possible to find the location of data items efficiently.
3. It should be possible to change the location of data items transparently.
4. Each site should be able to create new data items autonomously.

Centralized Scheme - Name Server

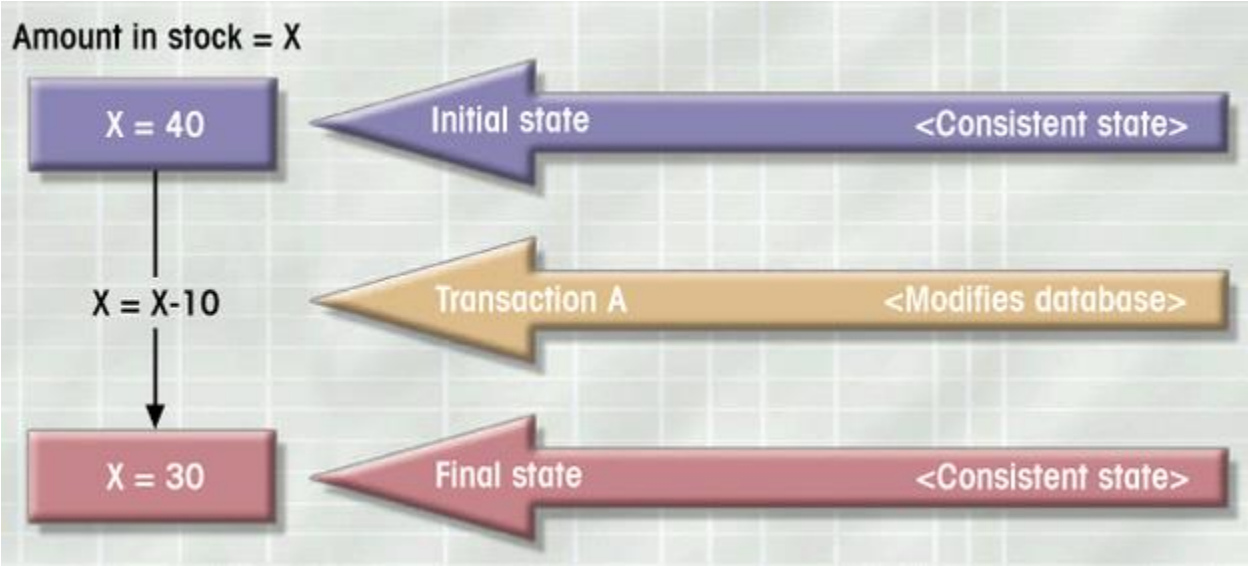
- Structure:
  - name server assigns all names
  - each site maintains a record of local data items
  - sites ask name server to locate non-local data items
- Advantages:
  - satisfies naming criteria 1-3
- Disadvantages:
  - does not satisfy naming criterion 4
  - name server is a potential performance bottleneck
  - name server is a single point of failure

Use of Aliases

- Alternative to centralized scheme: each site prefixes its own site identifier to any name that it generates i.e., *site* 17.a*ccount.*
  - Fulfills having a unique identifier, and avoids problems associated with central control.
  - However, fails to achieve network transparency.

What is a Transaction?

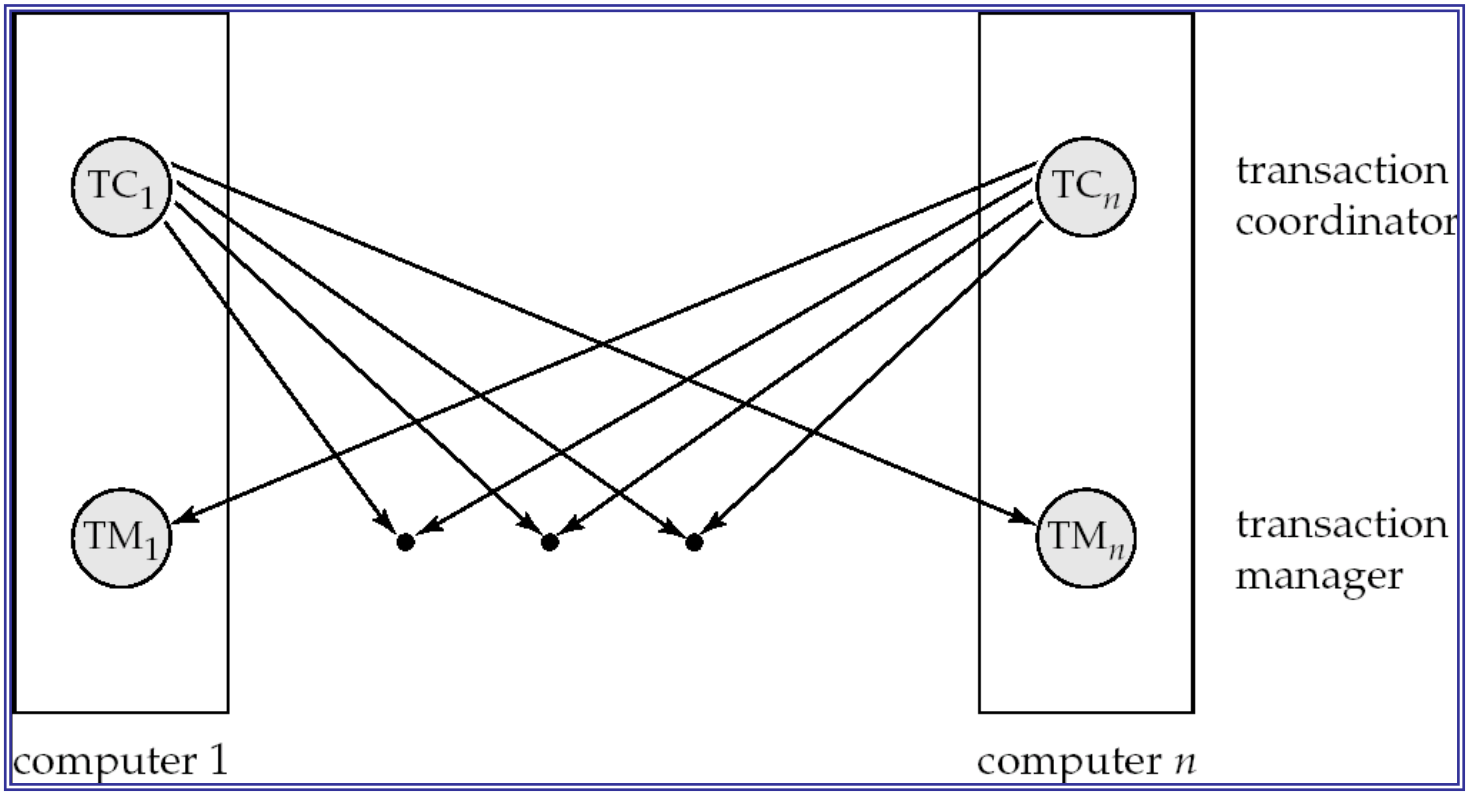
- A set of steps completed by a DBMS to accomplish a single user task.
- Must be either entirely completed or aborted
- No intermediate states are acceptable



Distributed Transactions

- Transaction may access data at several sites.
- Each site has a local transaction manager responsible for:
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a transaction coordinator, which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.

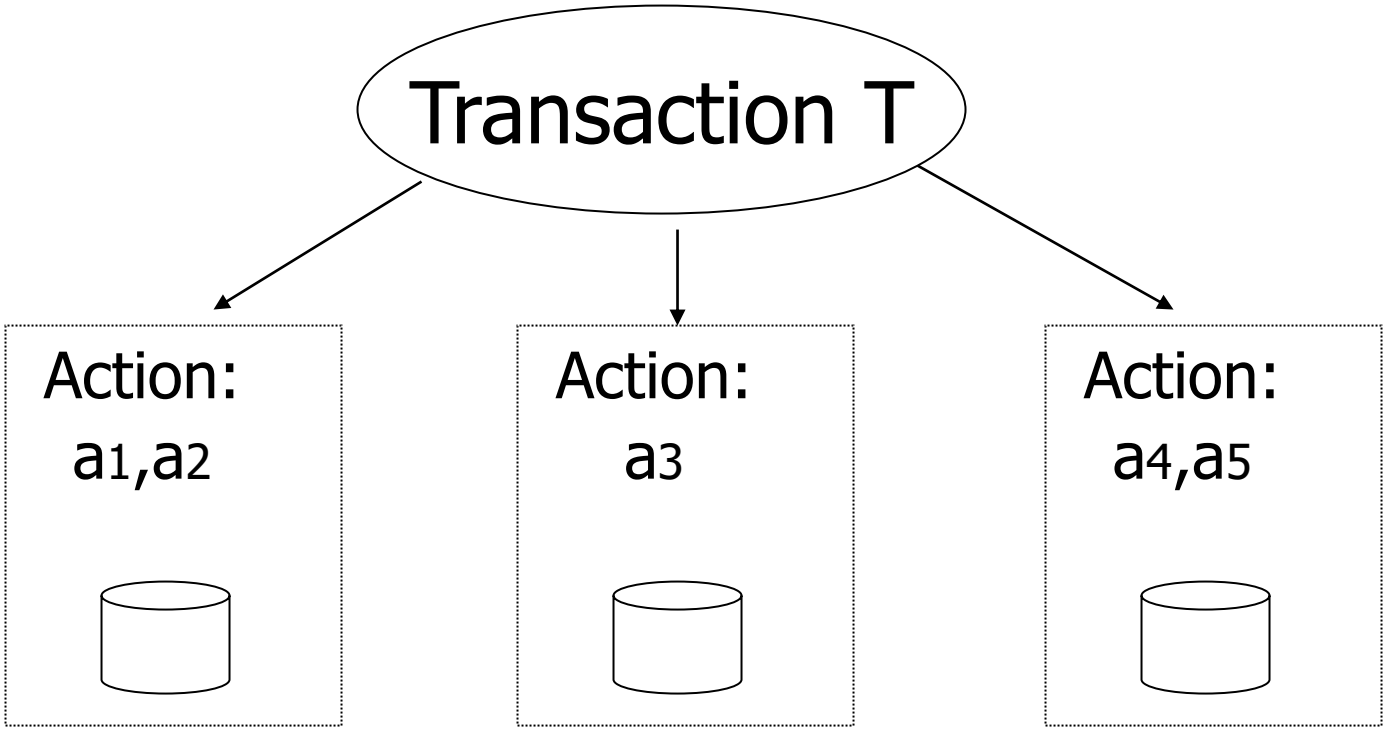
Transaction System Architecture



System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of massages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them
      - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.

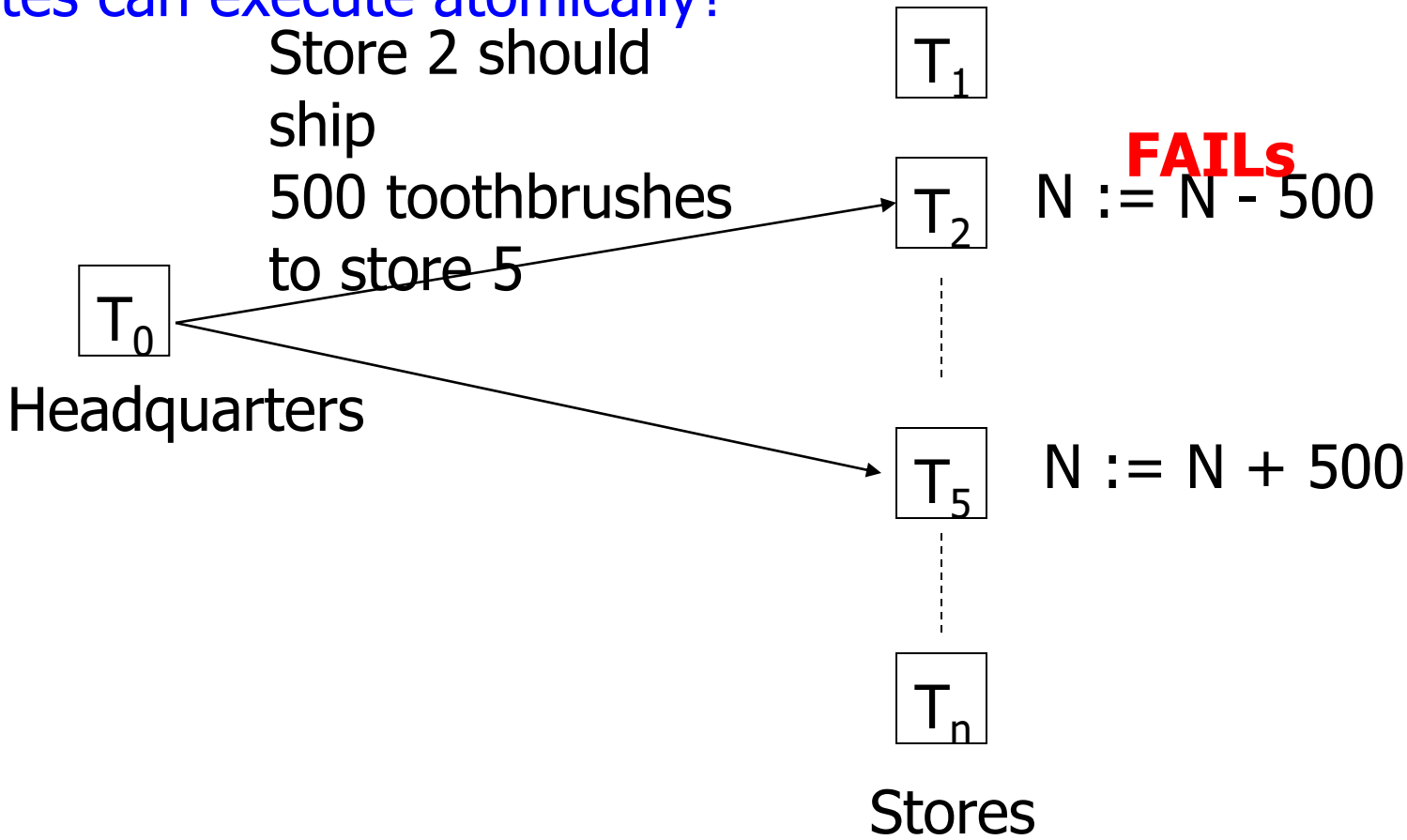
Distributed commit problem



Commit must be atomic

Example of a Distributed Transaction

How a distributed transaction that has components at several sites can execute atomically?

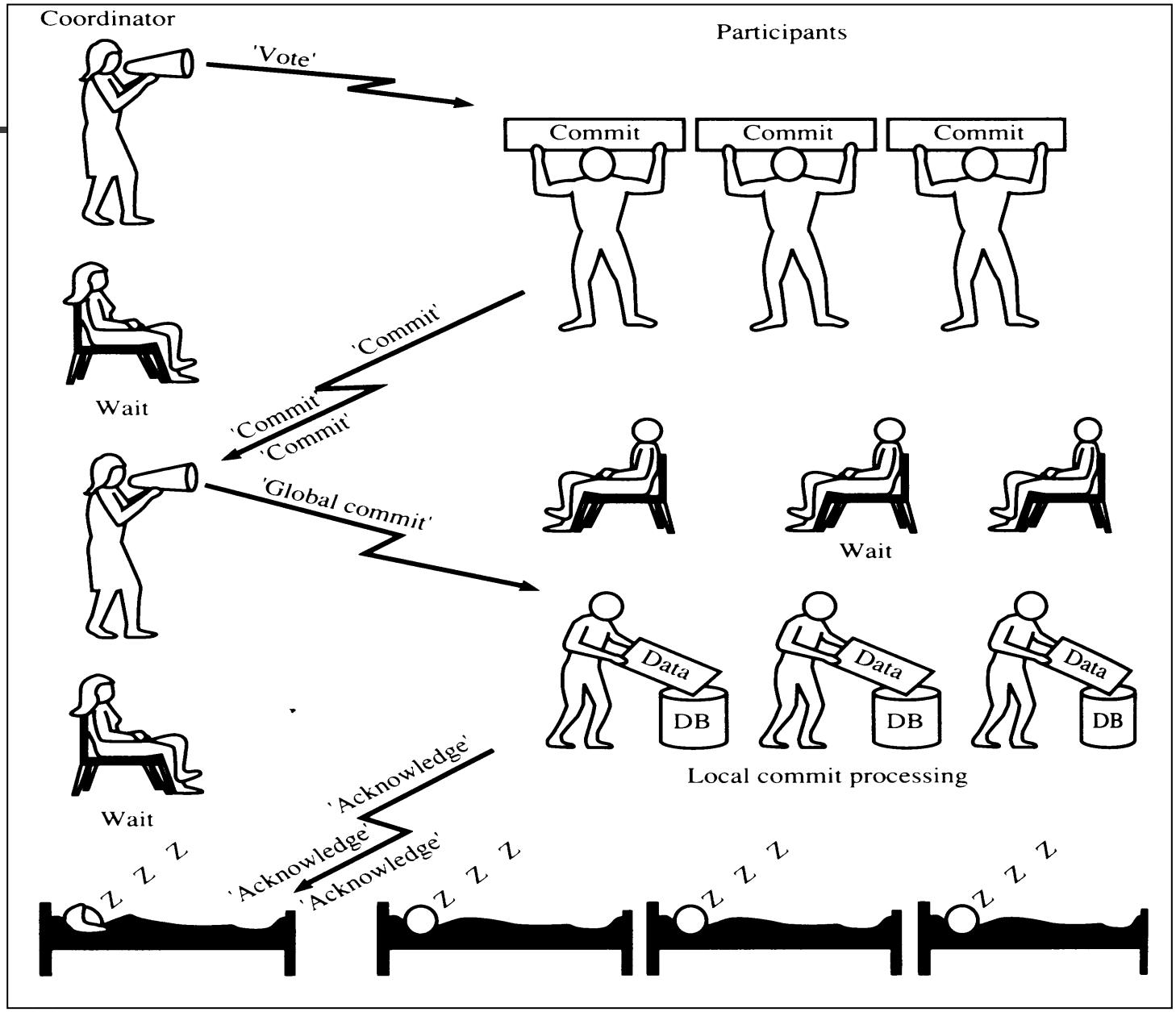


Each T_i is executed atomically, but T_0 itself is not atomic.

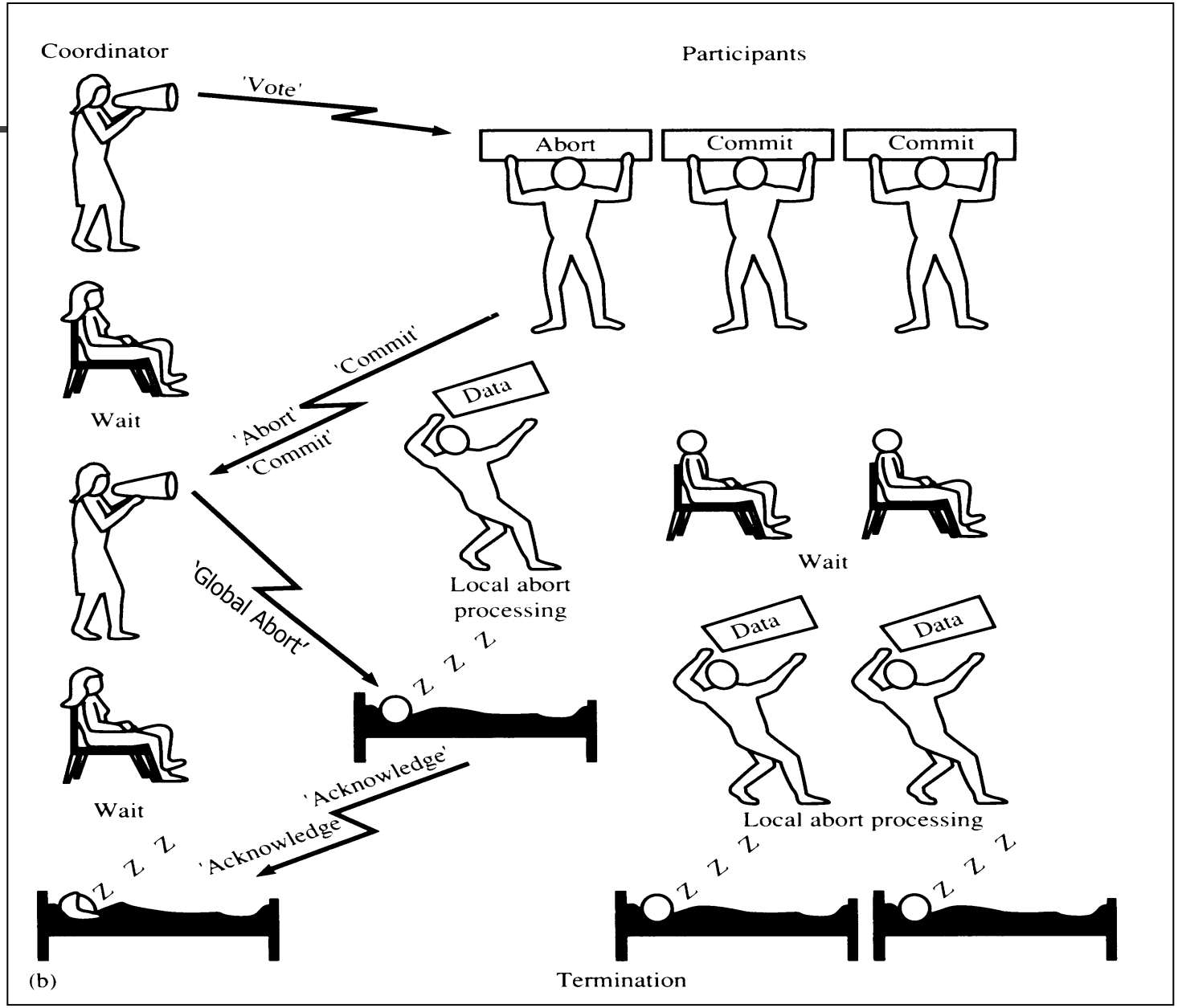
Distributed commit problem

- Commit must be atomic
- Solution: Two-phase commit (2PC)
  - Centralized 2PC
  - Distributed 2PC
  - Linear 2PC
  - Many other variants…

TWO-PHASE COMMIT (2PC) - OK

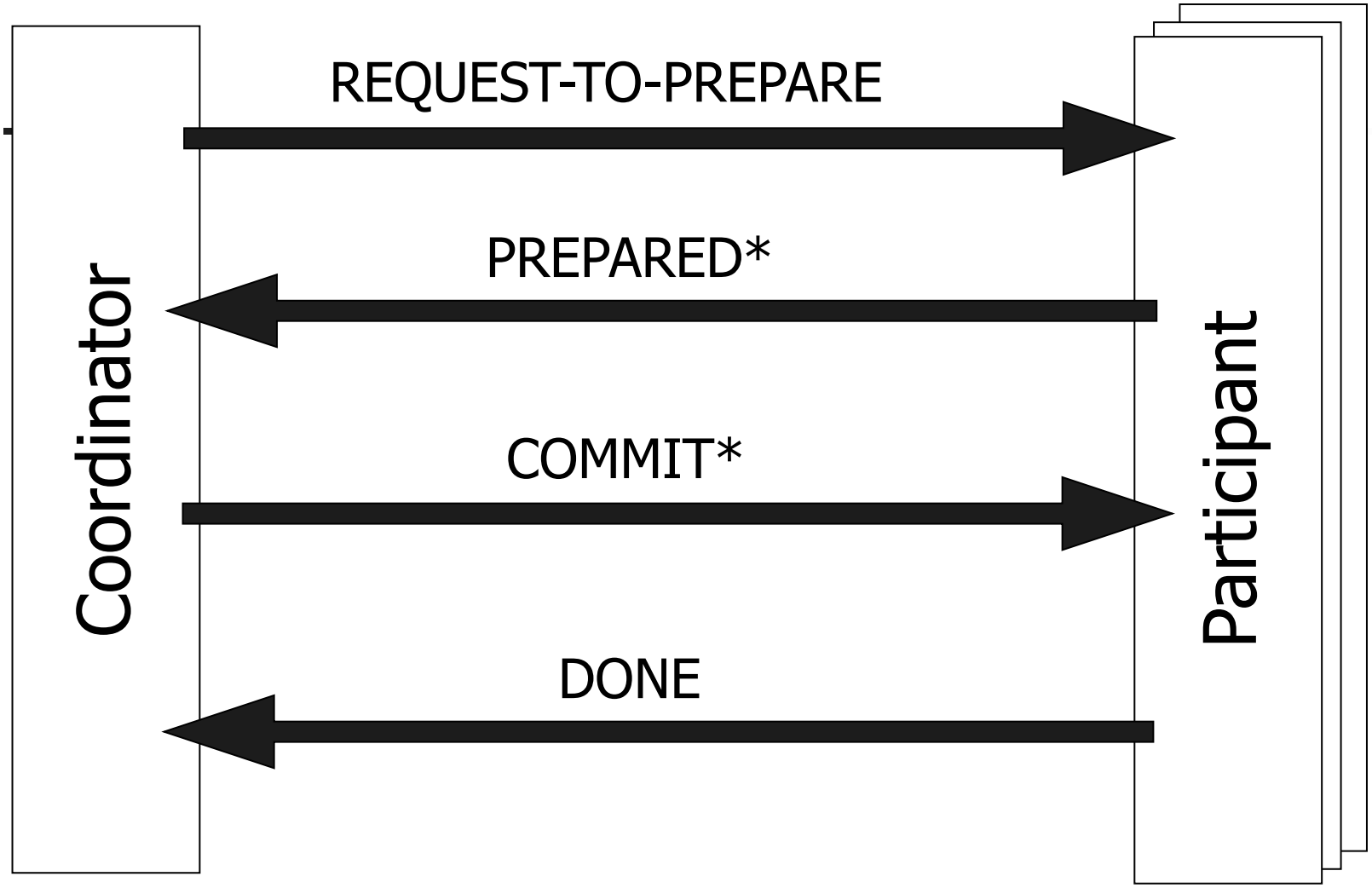


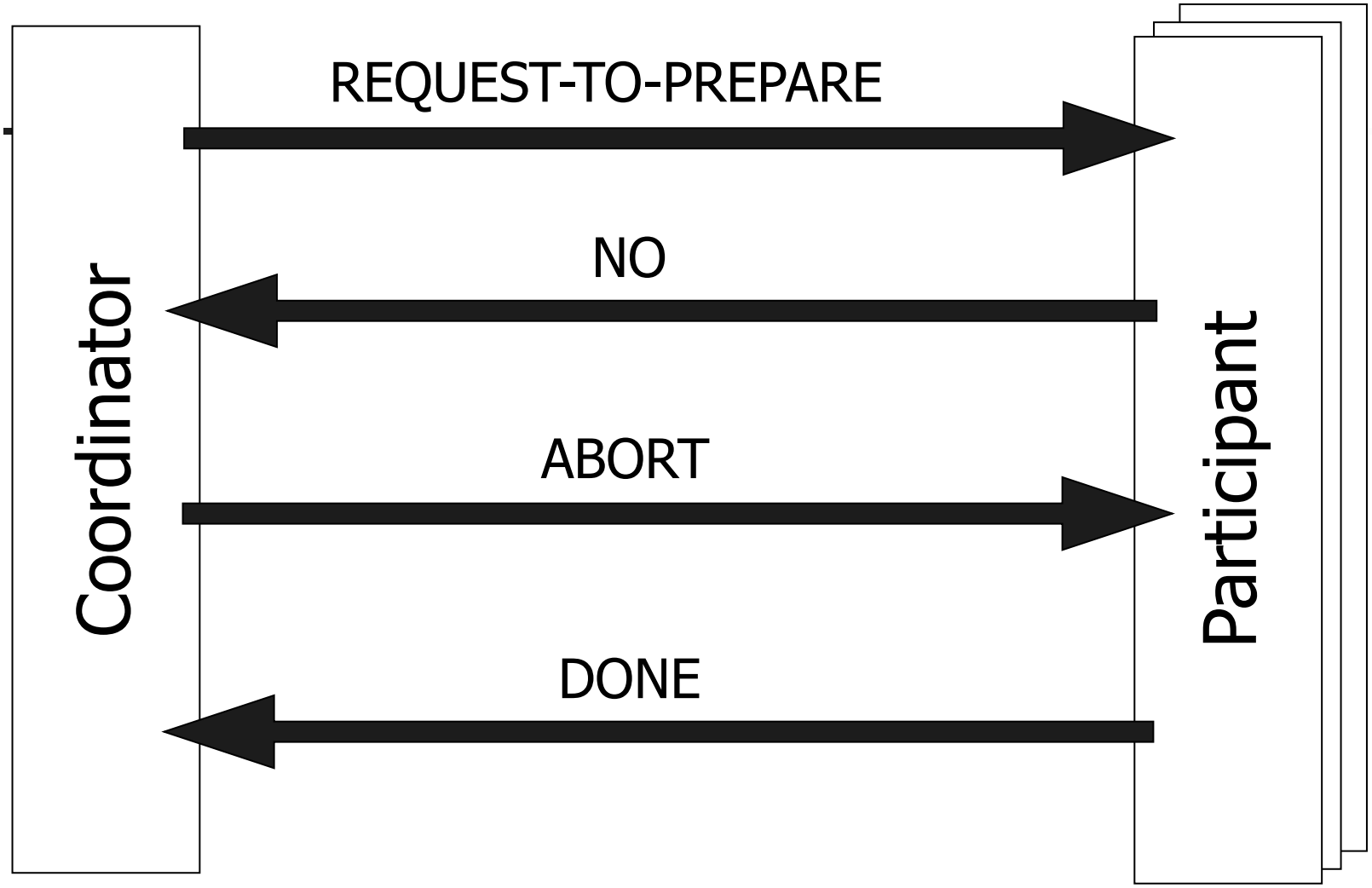
TWO-PHASE COMMIT (2PC) - ABORT



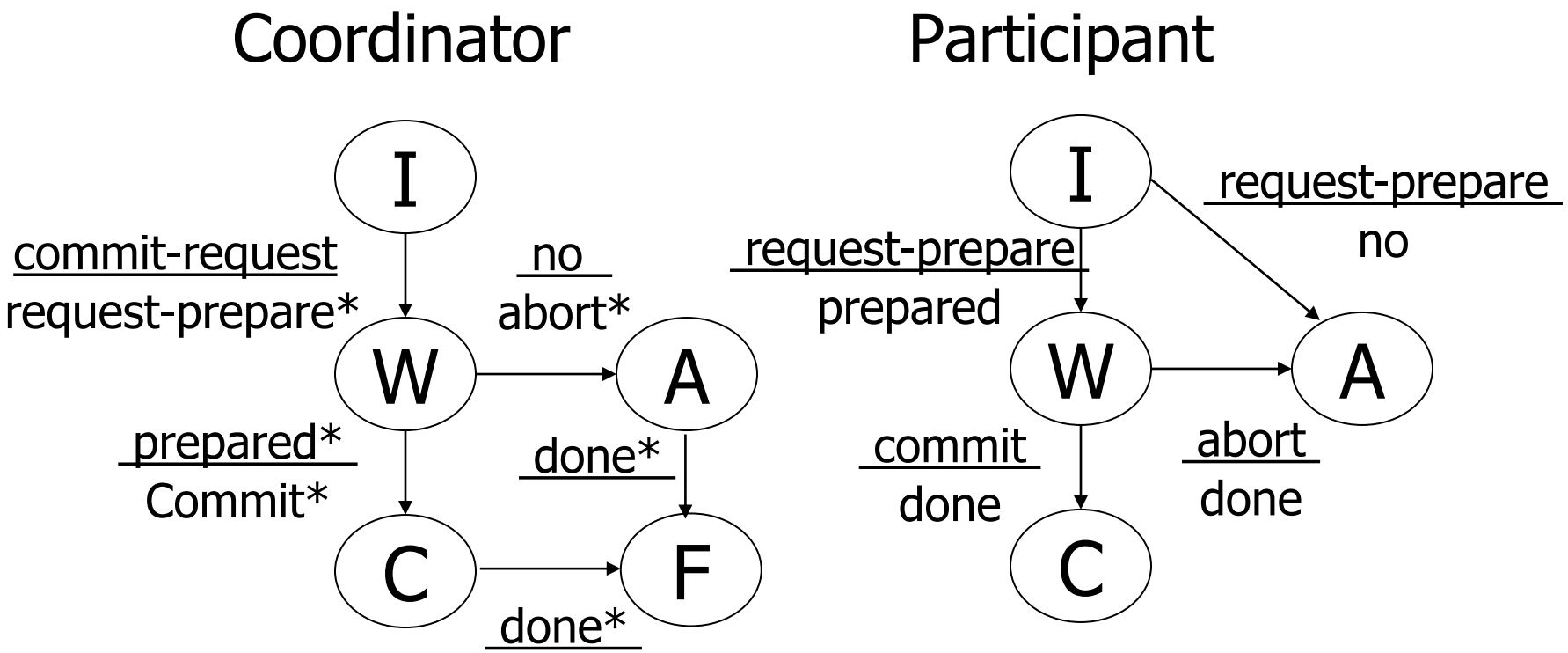
Terminology

- Resource Managers (RMs)
  - Usually databases
- Participants
  - RMs that did work on behalf of transaction
- Coordinator
  - Component that runs two-phase commit on behalf of transaction


Coordinator
REQUEST-TO-PREPARE
PREPARED*
COMMIT*
DONE
Participant


Coordinator
REQUEST-TO-PREPARE
NO
ABORT
DONE
Participant

Centralized two-phase commit



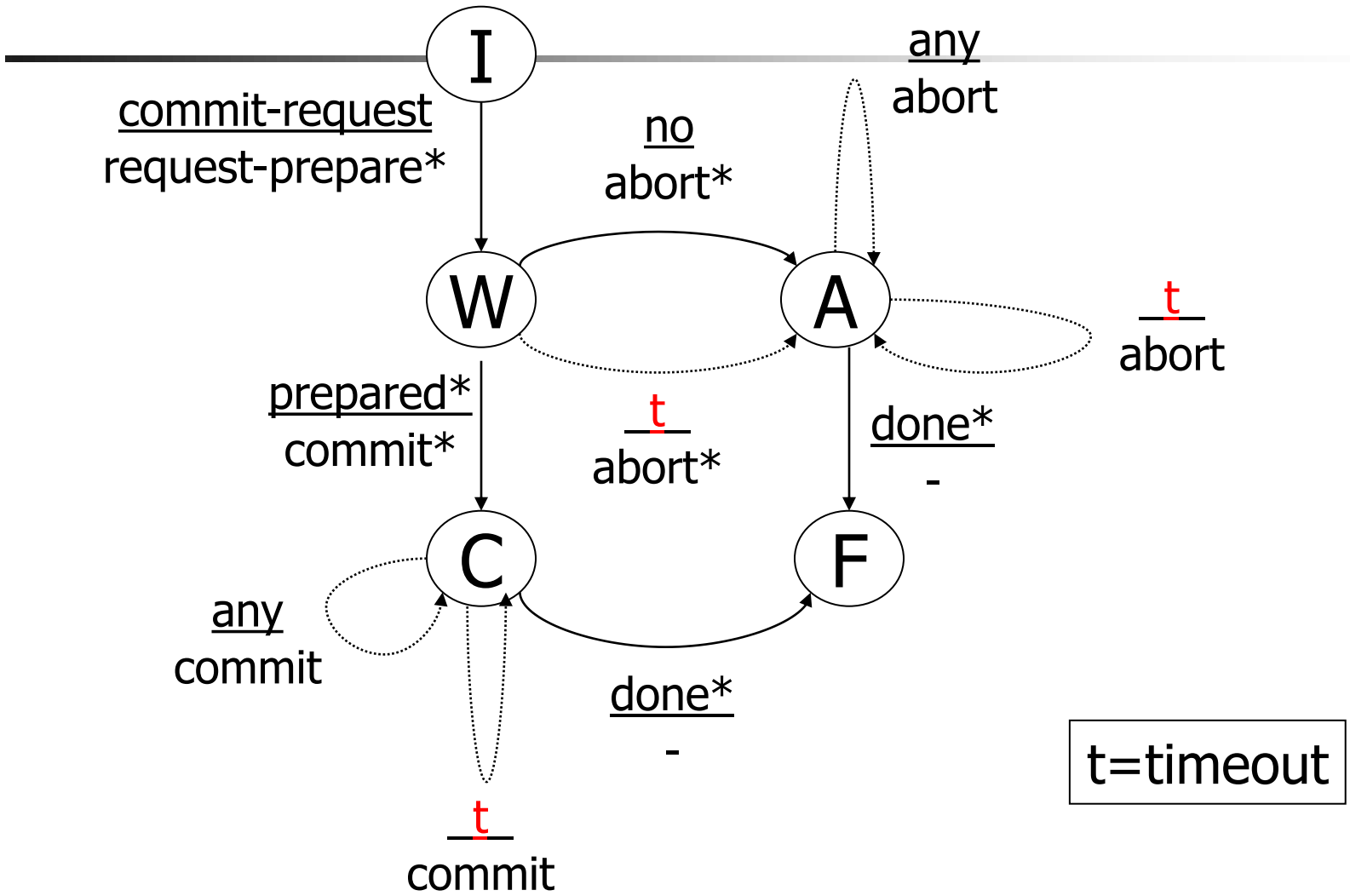
- Notation: <u>Incoming message</u>
  Outgoing message
  (* = everyone)
- When participant enters "W" state:
  - it must have acquired all resources
  - it can only abort or commit if so instructed by a coordinator
- Coordinator only enters "C" state if all participants are in "W", i.e., it is certain that all will <u>eventually</u> commit

- After coordinator receives DONE message, it can forget about the transaction
  - E.g., cleanup control structures

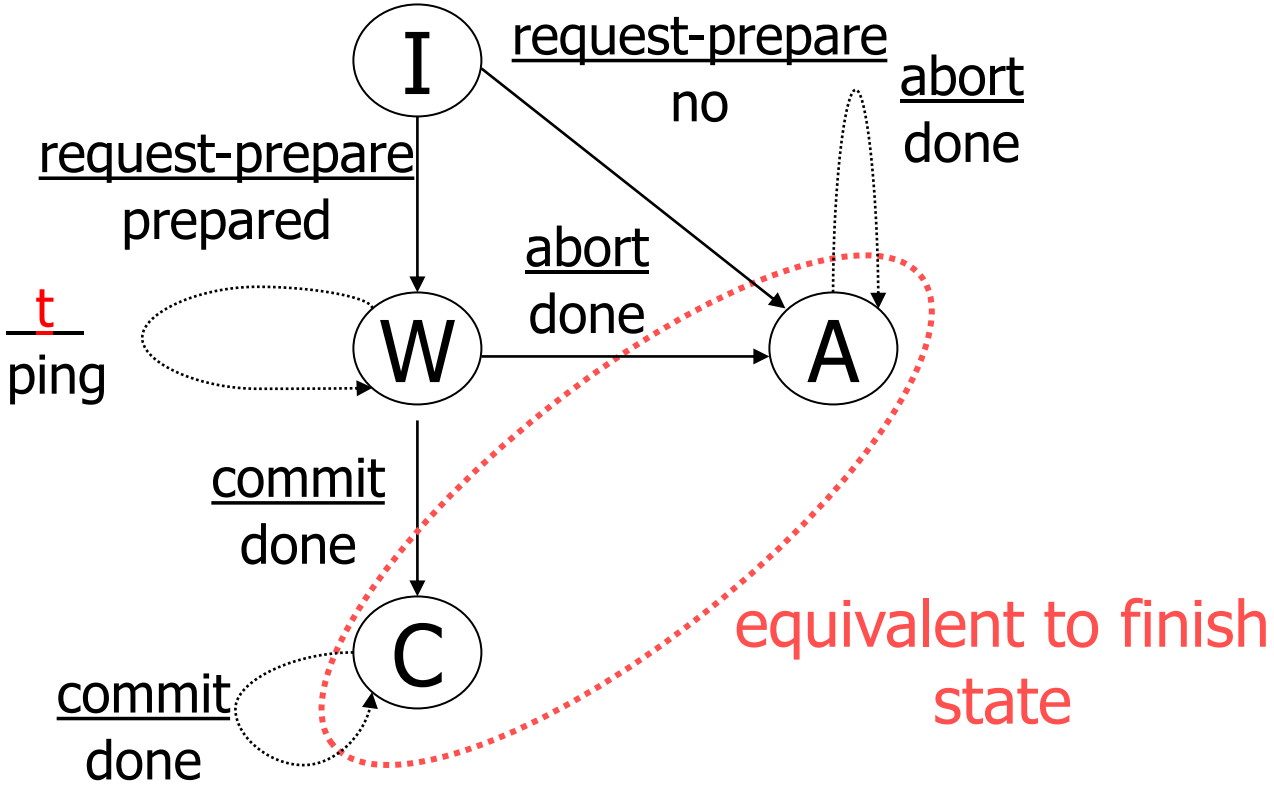
Next

- Add timeouts to cope with messages lost during crashes

Coordinator



Participant



Distributed Query Processing

- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.
- In a distributed system, other issues must be taken into account:
  - The cost of a data transmission over the network.
  - The potential gain in performance from having several sites process parts of the query in parallel.

Problem Statement

- Input: Query

  How many times has the moon circled around the earth in the last twenty years?

- Output: Answer

  240!

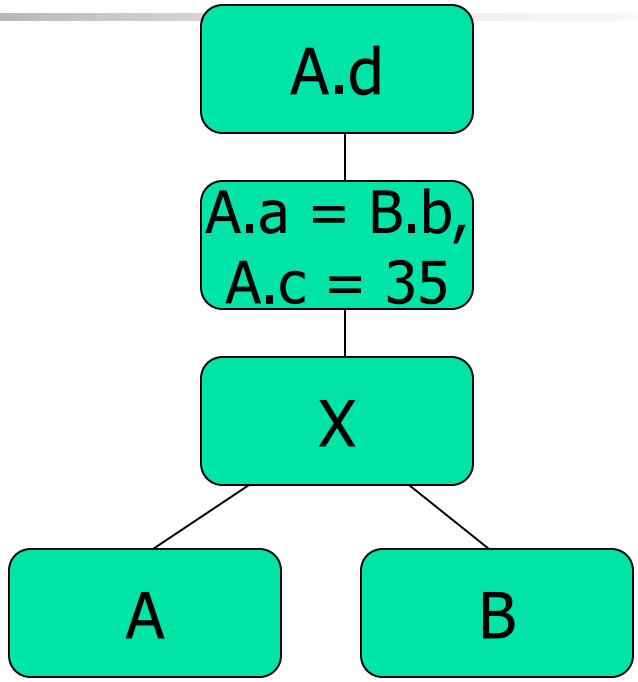
- Objectives:
  - response time, throughput, first answers, little IO, ...
- Centralized vs. Distributed Query Processing
  - same problem
  - but, different parameters and objectives

Query Processing

- Input: Declarative Query
  - SQL, OQL, XQuery, ...
- Step 1: Translate Query into Algebra
  - Tree of operators
- Step 2: Optimize Query (physical and logical)
  - Tree of operators
  - (Compilation)
- Step 3: Interpretation
  - Query result

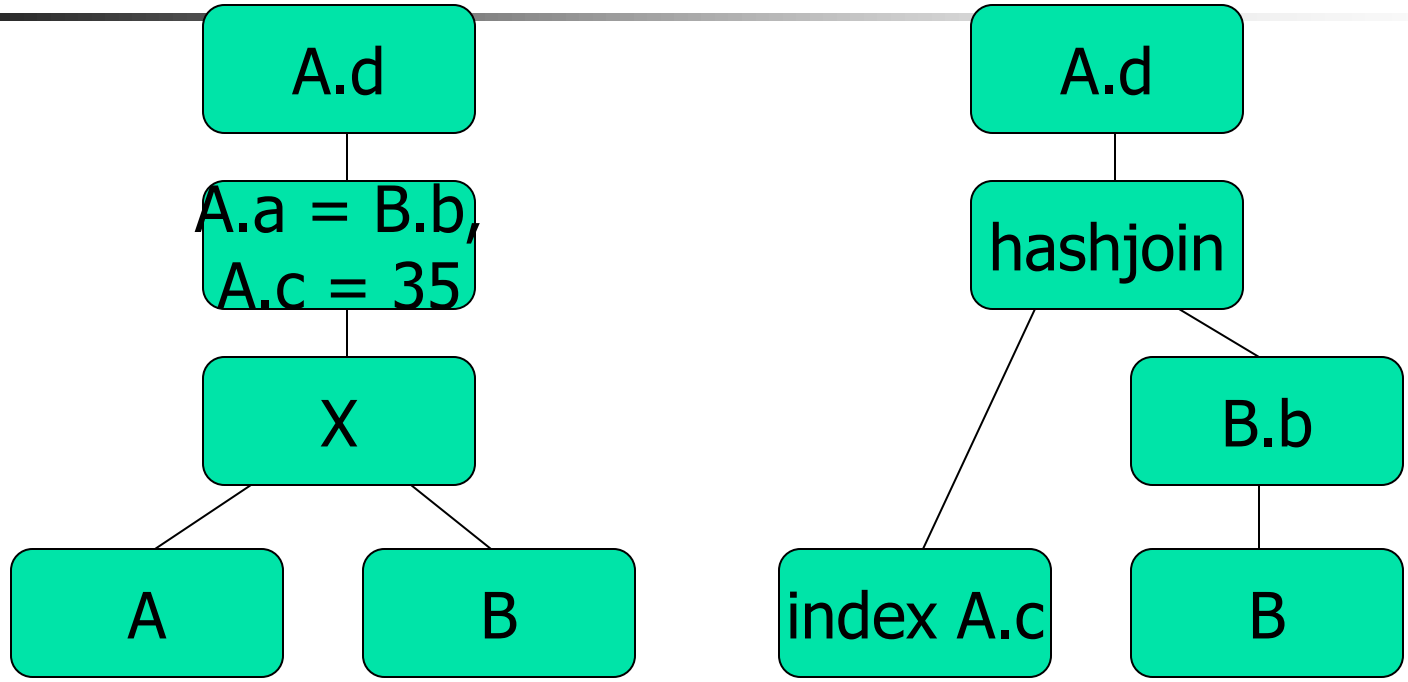
Algebra

```
SELECT A.d
FROM    A, B
WHERE  A.a = B.b
       AND A.c = 35
```



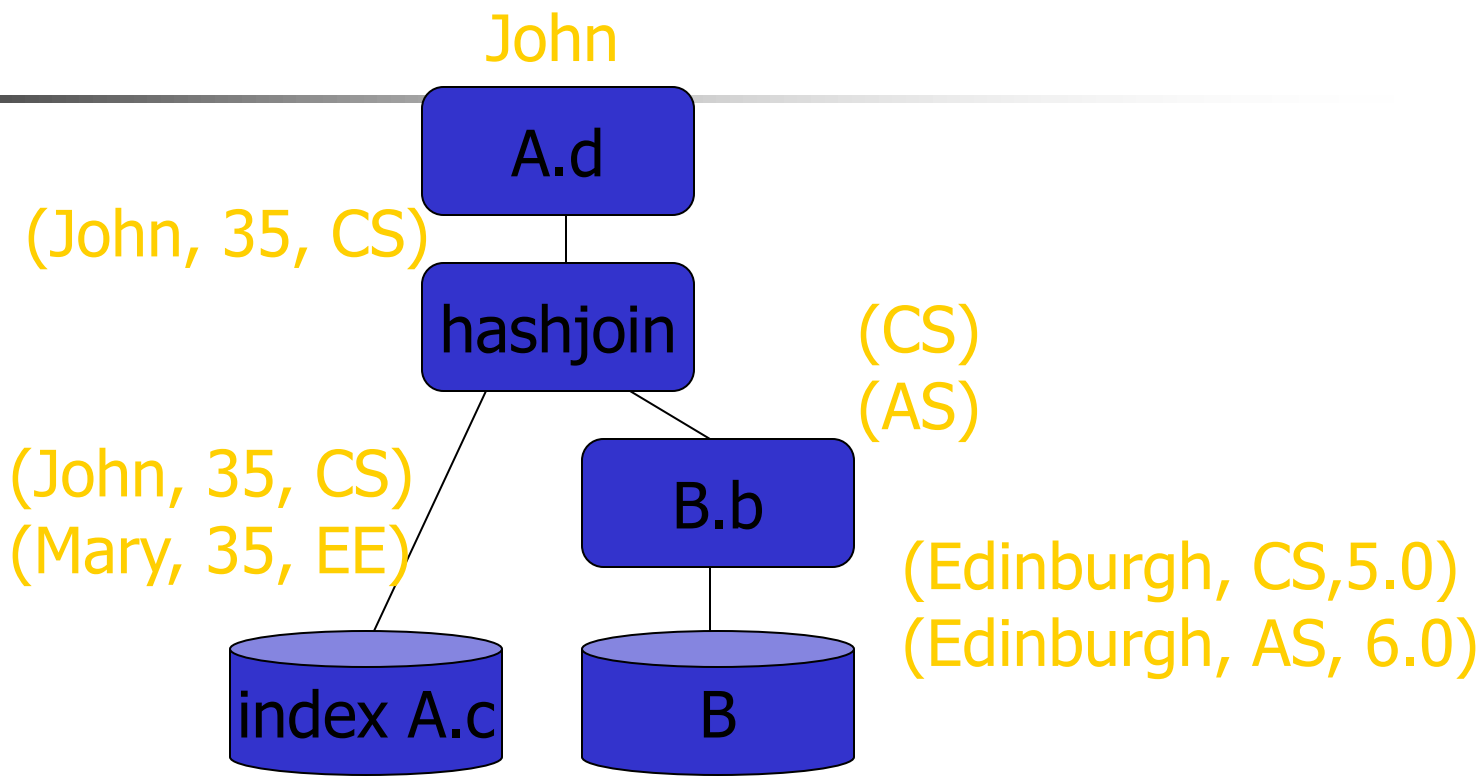
- relational algebra for SQL very well understood
- algebra for XQuery (work in progress)

Query Optimization



- no brainers (e.g., push down cheap predicates)
- enumerate alternative plans, apply cost model
- use search heuristics to find cheapest plan

Query Execution



- library of operators (hash join, merge join, ...)
- pipelining (iterator model)
- lazy evaluation
- exploit indexes and clustering in database

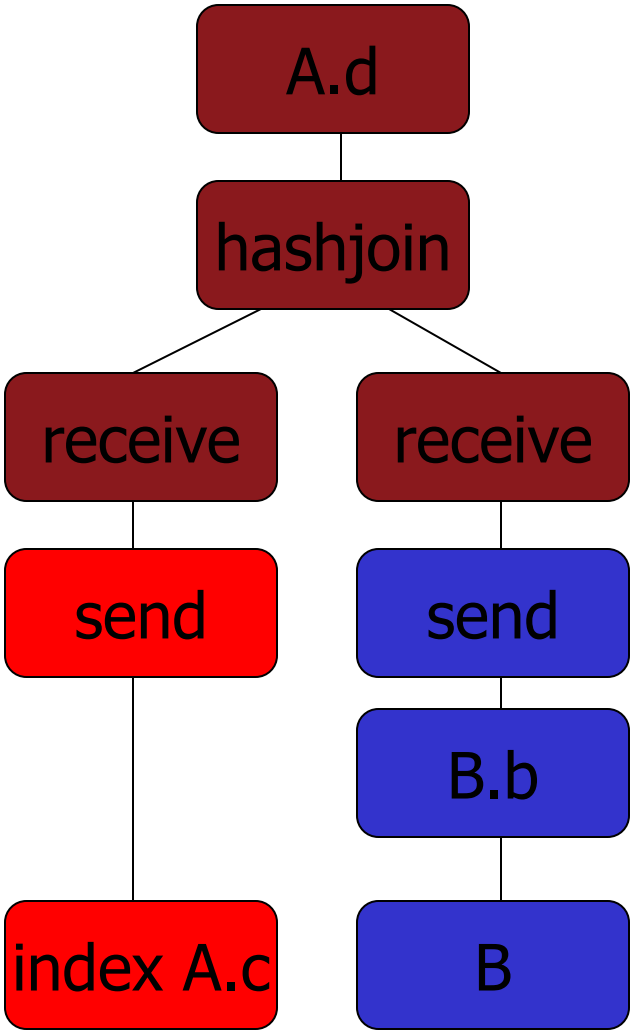
Distributed Query Processing

- Idea:

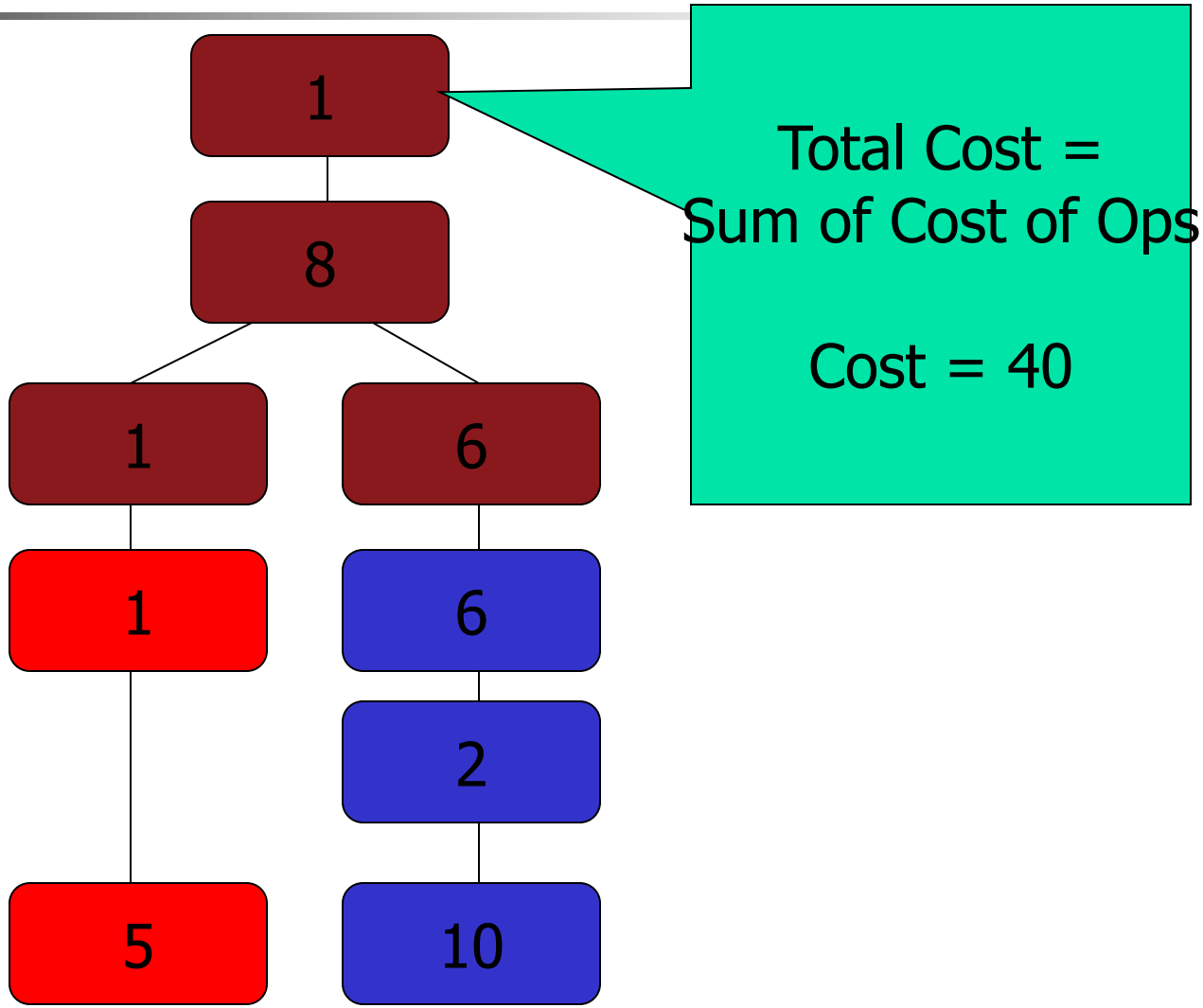
  This is just an extension of centralized query processing. (System R* et al. in the early 80s)

- What is different?
  - extend physical algebra: send&receive operators
  - resource vectors, network interconnect matrix
  - caching and replication
  - optimize for response time
  - less predictability in cost model (adaptive algos)
  - heterogeneity in data formats and data models

Distributed Query Plan



Cost



Query Transformation

- Translating algebraic queries on fragments.
  - It must be possible to construct relation *r* from its fragments
  - Replace relation *r* by the expression to construct relation *r* from its fragments
- Consider the horizontal fragmentation of the *account* relation into

  $account_1 = \sigma_{branch\_name = \text{"Hillside"}} (account)$

  $account_2 = \sigma_{branch\_name = \text{"Valleyview"}} (account)$
- The query $\sigma_{branch\_name = \text{"Hillside"}} (account)$ becomes

  $\sigma_{\text{branch\_name} = \text{"Hillside"}} (account_1 \cup account_2)$

  which is optimized into

  $\sigma_{branch\_name = \text{"Hillside"}} (account_1) \cup \sigma_{branch\_name = \text{"Hillside"}} (account_2)$

Simple Join Processing

- Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented

  $account \bowtie depositor \bowtie branch$
- *account* is stored at site S_1
- *depositor* at S_2
- *branch* at S_3
- For a query issued at site S_I, the system needs to produce the result at site S_I

Possible Query Processing Strategies

- Ship copies of all three relations to site S_I and choose a strategy for processing the entire locally at site S_I.
- Ship a copy of the account relation to site S_2 and compute $temp_1 = account \bowtie depositor$ *at* S_2. Ship $temp_1$ from S_2 to S_3, and compute $temp_2 = temp_1$ *branch* at S_3. Ship the result $temp_2$ to S_I.
- Devise similar strategies, exchanging the roles S_1, S_2, S_3
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site

Semijoin Strategy

- Let r_1 be a relation with schema R_1 stores at site S_1

  Let r_2 be a relation with schema R_2 stores at site S_2
- Evaluate the expression $r_1 \bowtie r_2$ and obtain the result at S_1.

1. Compute $temp_1 \leftarrow \Pi_{R1 \cap R2}$ (r1) at S1.

- 2. Ship $temp_1$ from S_1 to S_2.
- 3. Compute $temp_2 \leftarrow r_2 \bowtie$ temp1 at S_2
- 4. Ship $temp_2$ from S_2 to S_1.
- 5. Compute $r_1 \bowtie temp_2$ at S_1. This is the same as $r_1 \bowtie r_2$.

Formal Definition

- The **semijoin** of r_1 with r_2, is denoted by:

$$r_1 \ltimes r_2$$

- it is defined by:
- $\Pi_{R1} (r_1 \bowtie r_2)$
- Thus, $r_1 \ltimes r_2$ selects those tuples of r_1 that contributed to $r_1 \bowtie r_2$.
- In step 3 above, $temp_2 = r_2 \ltimes r_1$.
- For joins of several relations, the above strategy can be extended to a series of semijoin steps.

Join Strategies that Exploit Parallelism

- Consider $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$ where relation ri is stored at site S_i. The result must be presented at site S_1.
- r_1 is shipped to S_2 and $r_1 \bowtie r_2$ is computed at S_2: simultaneously r_3 is shipped to S_4 and $r_3 \bowtie r_4$ is computed at S_4
- S_2 ships tuples of $(r_1 \bowtie r_2)$ to S_1 as they produced; S_4 ships tuples of $(r_3 \bowtie r_4)$ to S_1
- Once tuples of $(r_1 \bowtie r_2)$ and $(r_3 \bowtie r_4)$ arrive at S_1 $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$ is computed in parallel with the computation of $(r_1 \bowtie r_2)$ at S_2 and the computation of $(r_3 \bowtie r_4)$ at S_4.

Conclusion- Advantages of DDBMSs

- Reflects organizational structure
- Improved shareability and local autonomy
- Improved availability
- Improved reliability
- Improved performance
- Economics
- Modular growth

Conclusion- Disadvantages of DDBMSs

- ☹ Architectural complexity
- ☹ Cost
- ☹ Security
- ☹ Integrity control more difficult
- ☹ Lack of standards
- ☹ Lack of experience
- ☹ Database design more complex

References

- Chapter 22 of database systems concepts (Silberschatz book)
- ICS courses on DBMS: CS222, CS223