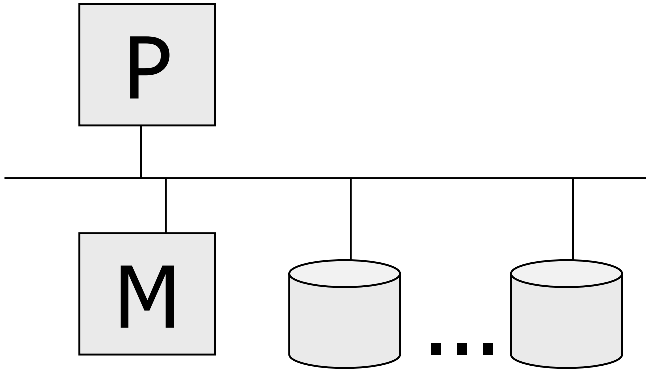
Distributed Databases

Introduction

- Centralized DB Systems:
  - single front end
  - one place to keep locks
  - if processor fails, system fails, ...



- Distributed systems:
  - Multiple processors (+ memories)
  - Heterogeneity and autonomy of “components”

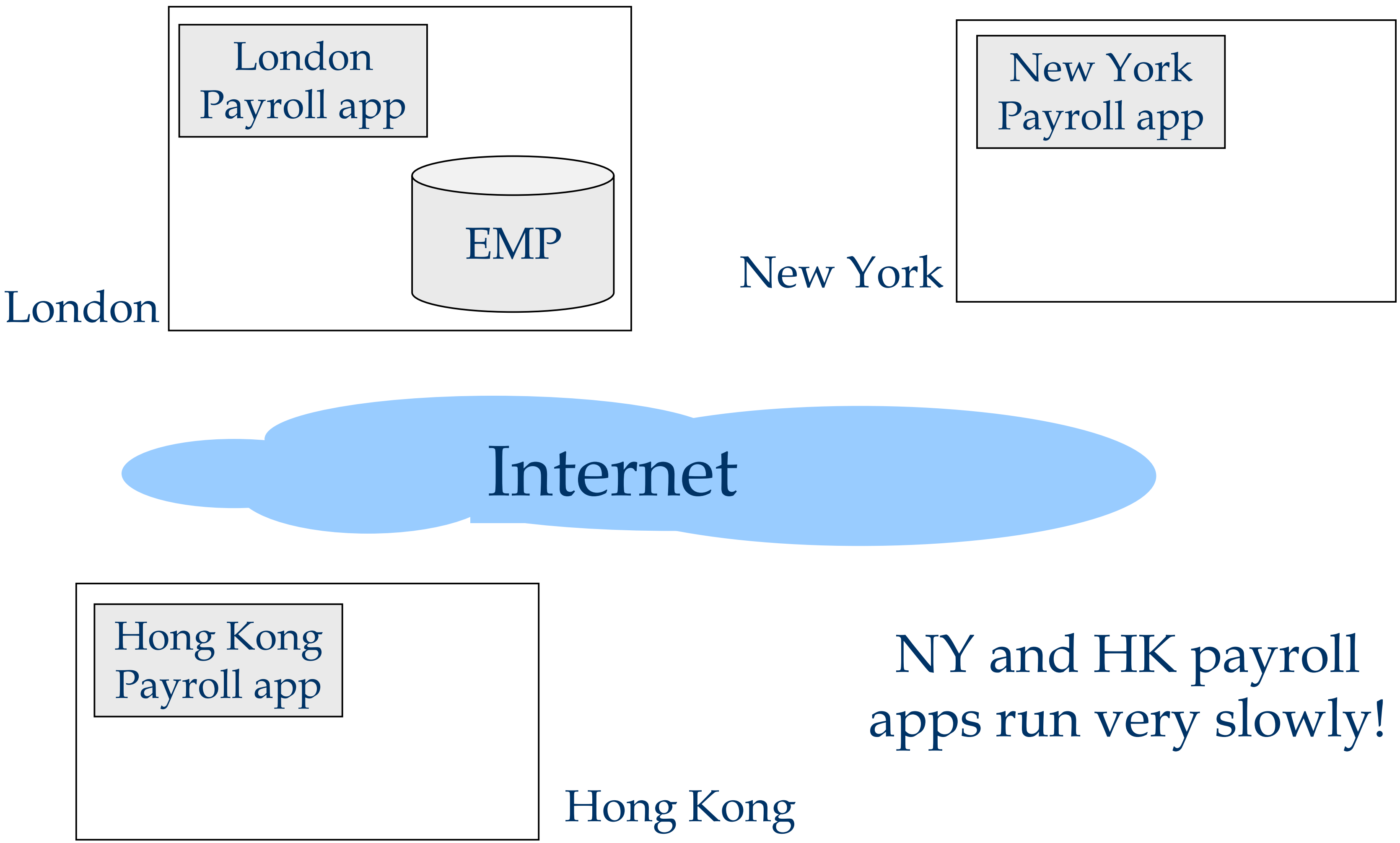
Distributed Databases

- Distributed Data Independence
- Distributed Transaction Atomicity

Why do we need Distributed Databases?

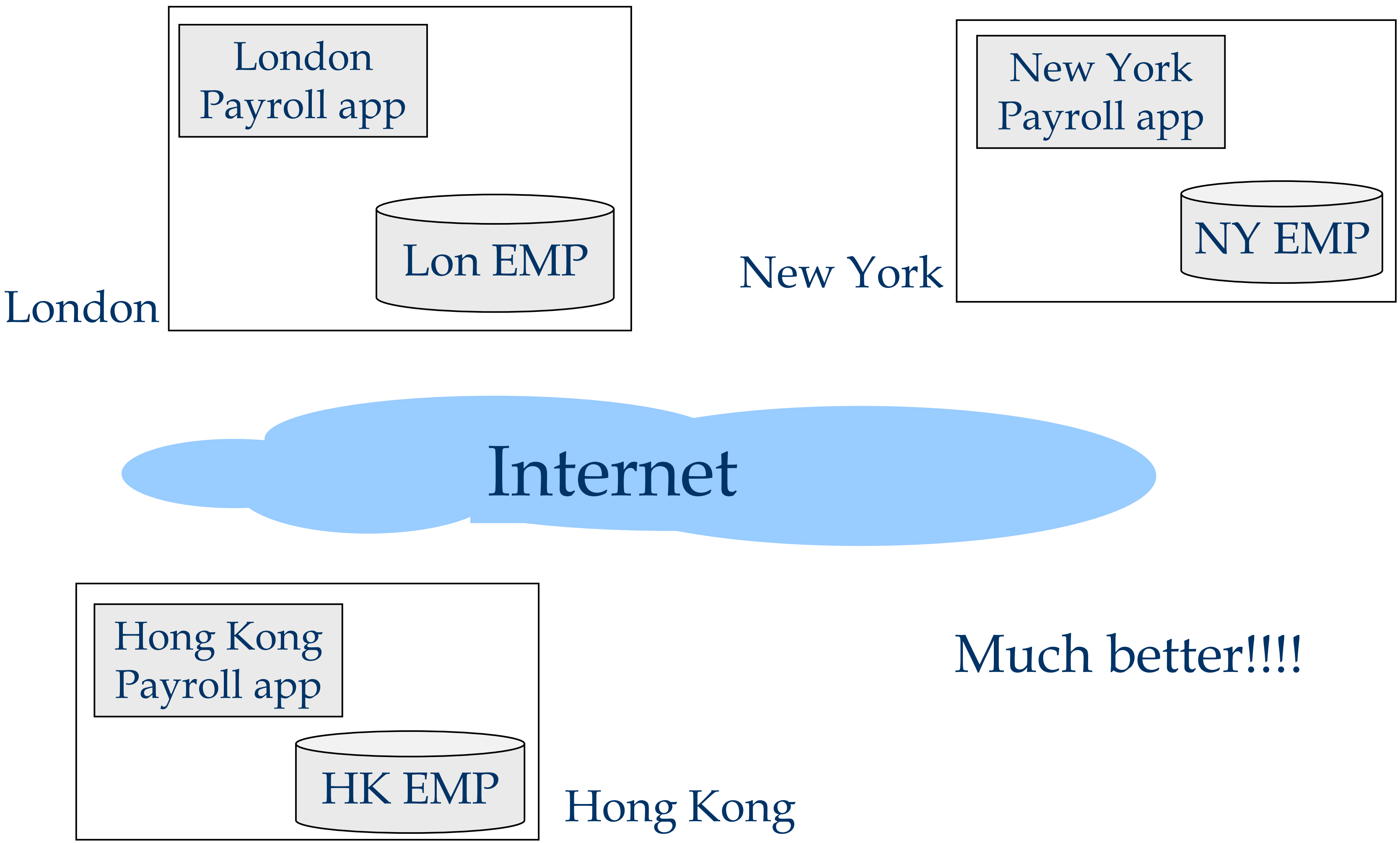
- Example: Big Corp has offices in London, New York and Hong Kong.
- Mostly, employee data is managed at the office where the employee works
    - E.g., payroll, benefits, hiring
- Periodically, Big Corp needs consolidated access to employee data
    - E.g Compute total payroll expenses for the balance sheet
    - E.g. Annual bonus depends on global net profit.
- Where should the employee data table reside?

Why do we need Distributed Databases?



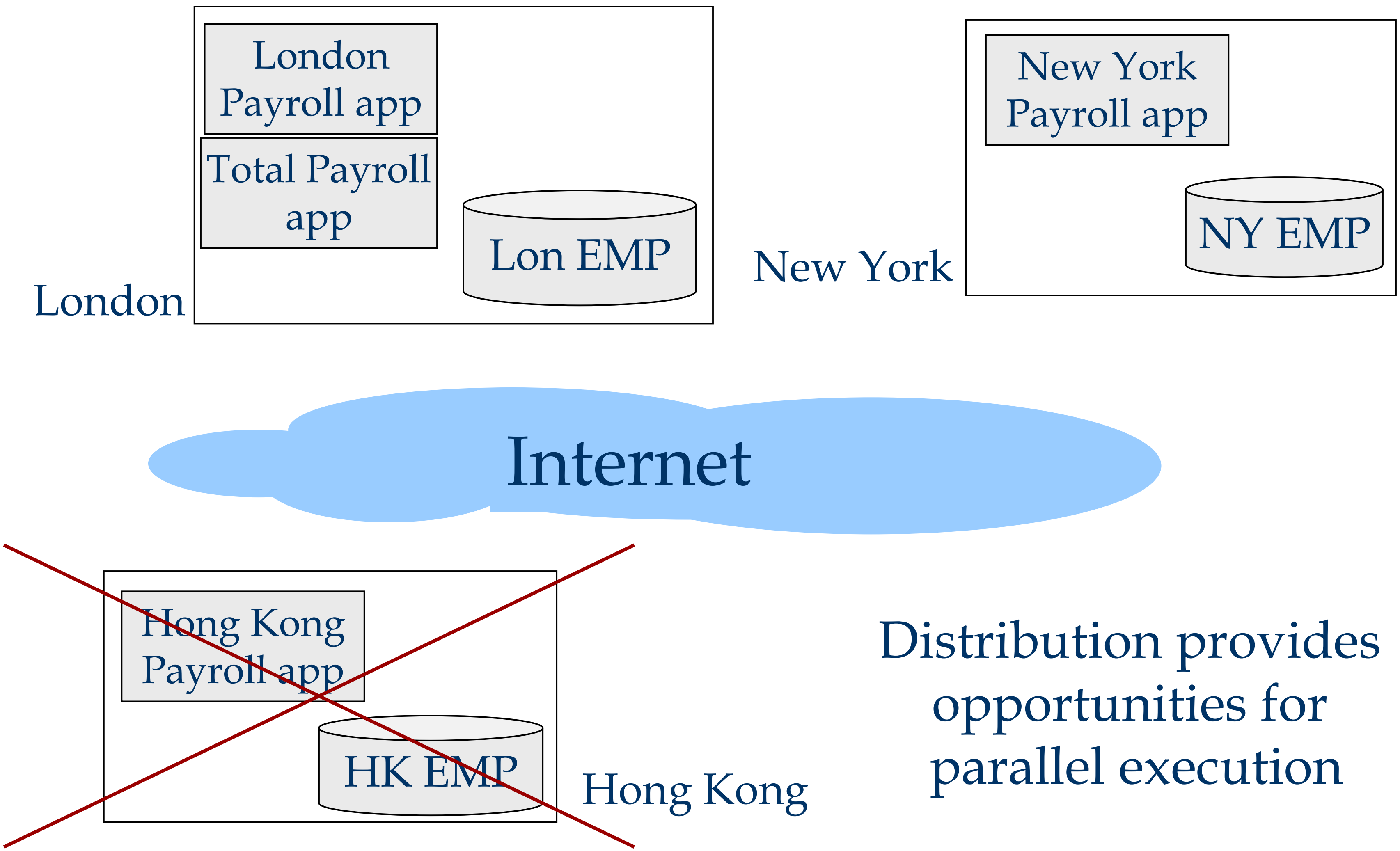
NY and HK payroll apps run very slowly!

Why do we need Distributed Databases?



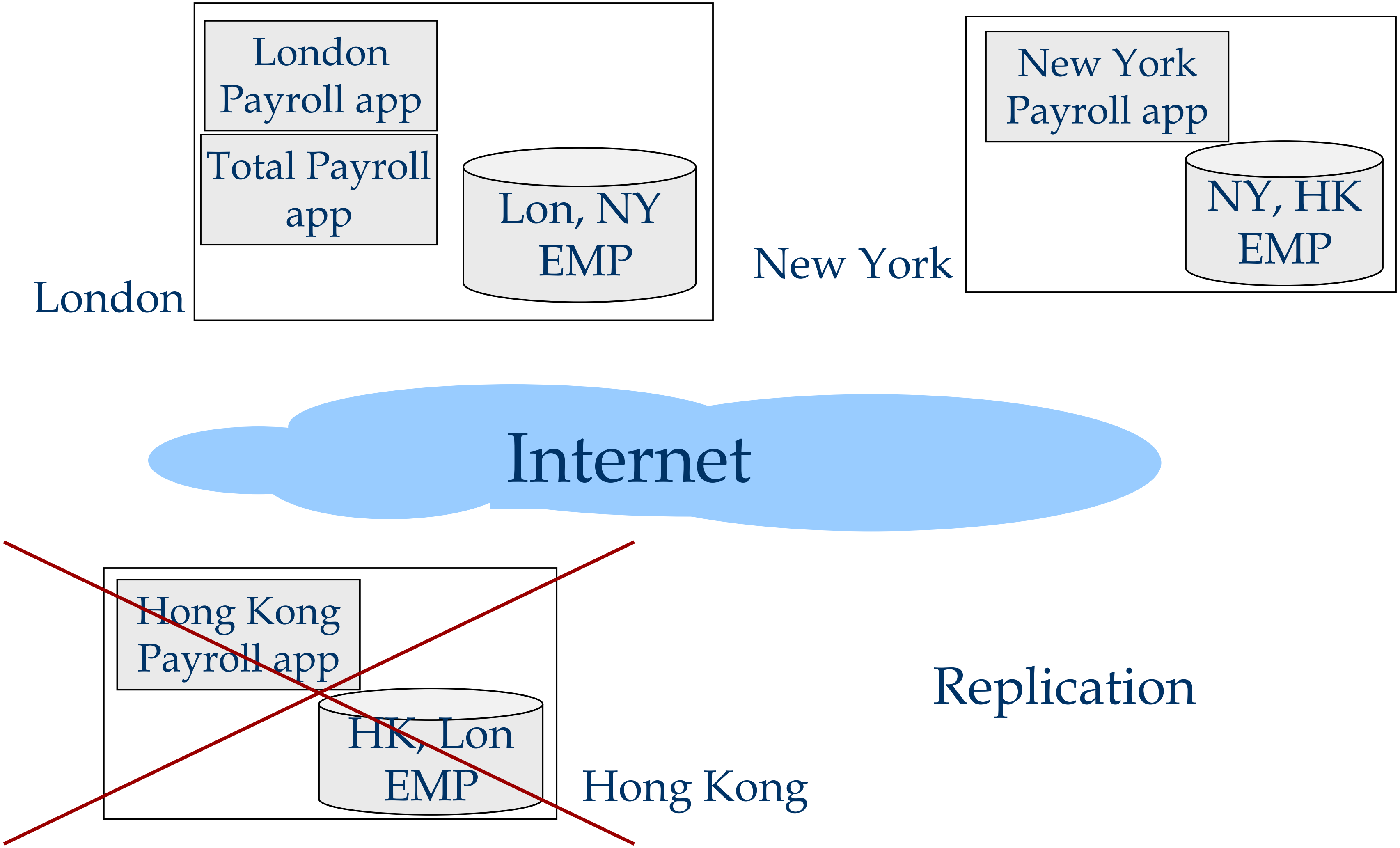
Much better!!!!

Why do we need Distributed Databases?



Distribution provides opportunities for parallel execution

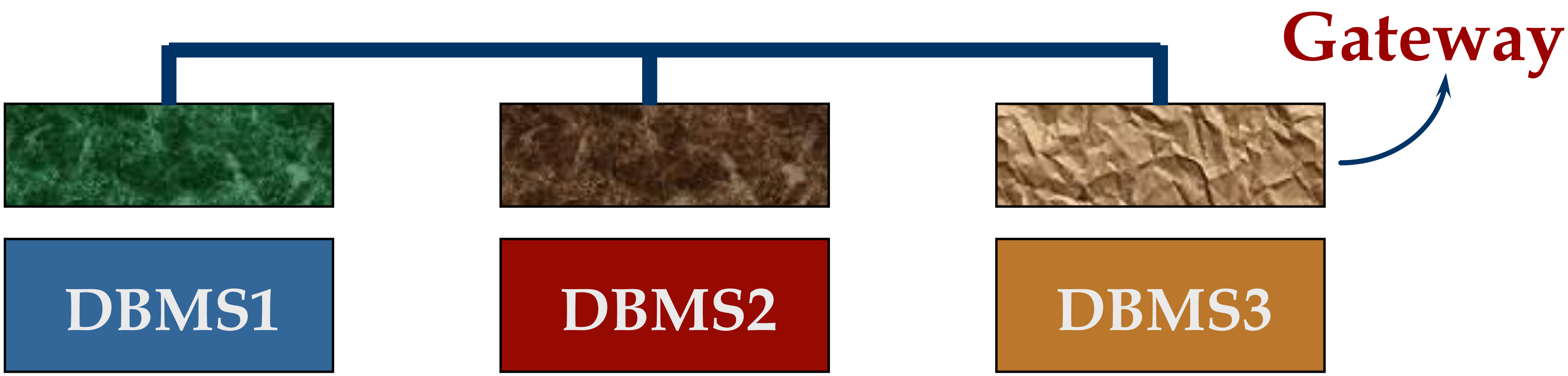
Why do we need Distributed Databases?



Replication

Types of Distributed Databases

- Homogeneous
- Heterogeneous



Distributed Database Challenges

Distributed Database Design

- Fragmentation &Allocation

Distributed Query Processing

- Communication costs
- Opportunity for parallelism

Distributed Concurrency Control

- Serializability
- Deadlock Management
- Data must be kept in sync

Reliability of Distributed Databases

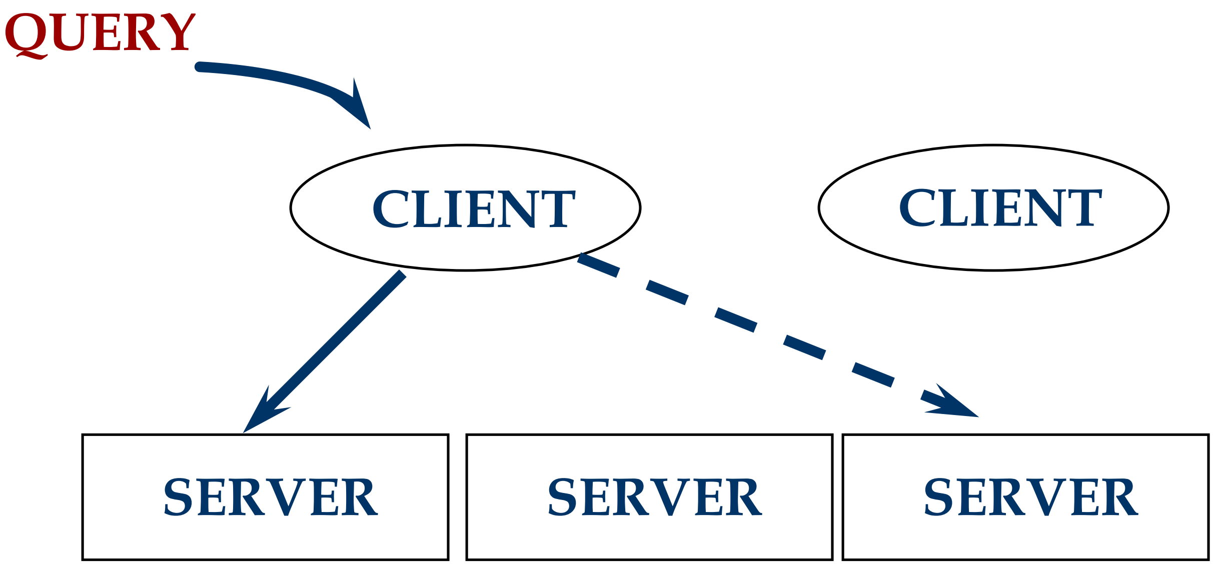
- Distributed database failure model
- Data must be kept in sync :)

Distributed DBMS Architectures

- Client-Server

Client ships query to single site. All query processing at server.

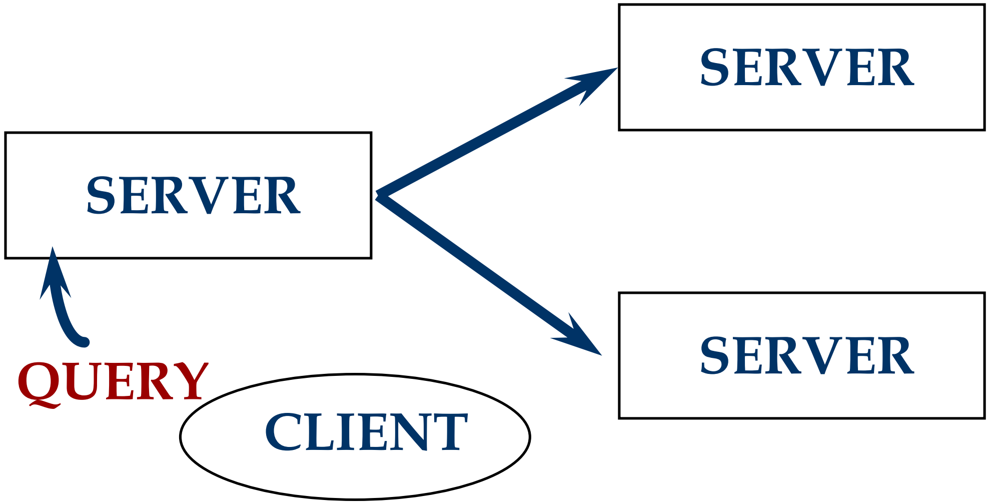
- *Thin* vs. *fat* clients.
- Set-oriented communication, client side caching.



Distributed DBMS Architectures

- Collaborating Server

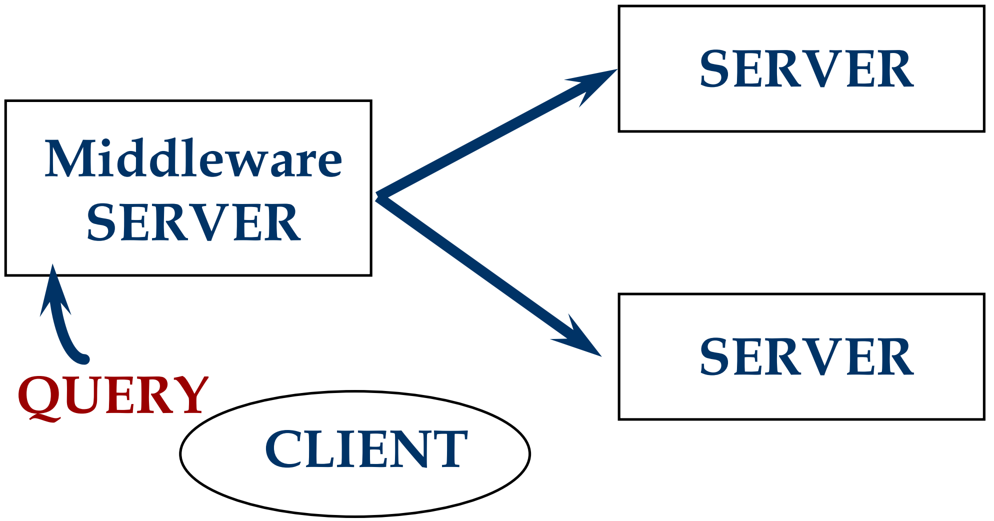
Query can span multiple sites.



Distributed DBMS Architectures

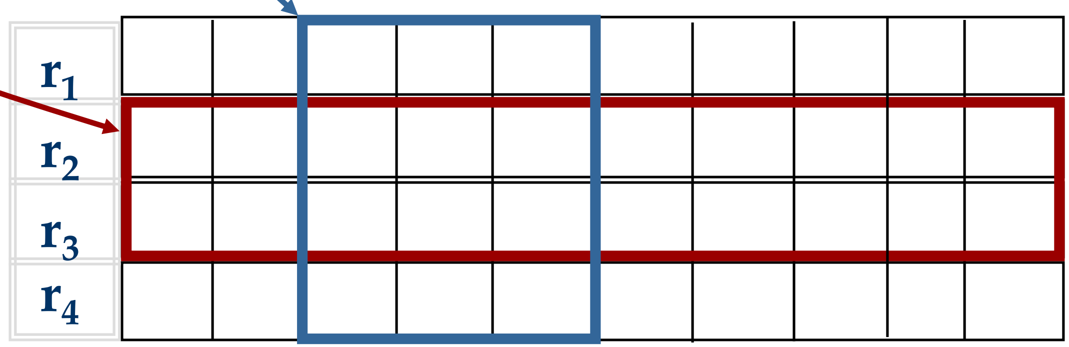
- Middleware System

One server manages
queries and transactions
spans multiple servers



Fragmentation

- *Horizontal*
  - **Primary**
  - **Derived**
- *Vertical*



Storing Data

- Fragmentation properties

$$R \Rightarrow \mathbf{F} = \{F_1, F_2, \ldots, F_n\}$$

Completeness

$\forall\, x \in R,\ \exists\, F_i \in \mathbf{F}$ such that $x \in F_i$

Disjointness

$\forall x \in F_i,\ \neg\exists\, F_j$ such that $x \in F_j,\ i \neq j$

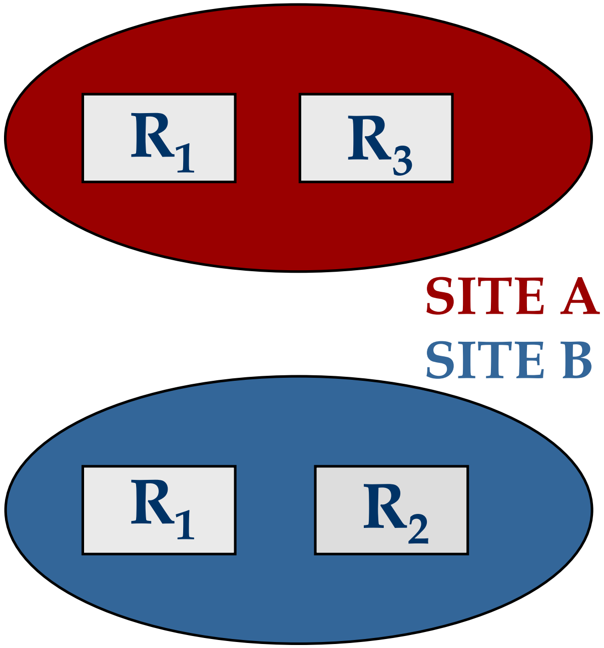
Reconstruction

There is function g such that

$$R = g(F_1, F_2, \ldots, F_n)$$

Storing Data (cont)

- Replication: store copies of a relation or relation fragment. An entire relation can be replicated at one or more sites.
  - Gives increased availability. (if a server goes down we can use other active server)
  - Faster query evaluation. (can use a local copy)
  - Synchronous vs. Asynchronous.
    - Vary in how current copies are.



R is fragmented in R_1, R_2, R_3
R_1 is replicated on both sites

Distributed Catalog Management

- Must keep track of how data is distributed across sites.
- Must be able to name each replica of each fragment. To preserve local autonomy:
  - <local-name, birth-site>
  - <local-name, birth-site, replica_id> - global replica name
- Site Catalog: Describes all objects (fragments, replicas) at a site + Keeps track of replicas of relations created at this site.
  - To find a relation, look up its birth-site catalog.
  - Birth-site never changes, even if relation is moved.

Updating Distributed Data

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

Synchronous Replication

- There are 2 basic techniques for ensuring that transactions see the same value regardless of which copy of an object they access.
- Voting: transaction must write a majority of copies to modify an object; must read enough copies to be sure of seeing at least one most recent copy.
    - E.g., 10 copies; 7 written for update; 4 copies read.
    - Each copy has version number.
    - Not attractive usually because reads are common.
- Read-any Write-all: Writes are slower and reads are faster, relative to Voting.
    - Most common approach to synchronous replication.
- Choice of technique determines *which* locks to set

Cost of Synchronous Replication

- Before an update transaction can commit, it must obtain locks on all modified copies.
  - Sends lock requests to remote sites, and while waiting for the response, holds on to other locks!
  - If sites or links fail, transaction cannot commit until they are back up.
  - Even if there is no failure, committing must follow an expensive ***commit protocol*** with many messages.
- So the alternative of *asynchronous replication* is becoming widely used.

Asynchronous Replication

- Allows modifying transaction to commit before all copies have been changed (and readers nonetheless look at just one copy).
  - Users must be aware of which copy they are reading, and that copies may be out-of-sync for short periods of time.
- Two approaches: Primary Site and Peer-to-Peer replication.
  - Difference lies in how many copies are ``updatable'' or ``master copies''.

Peer-to-Peer Replication

- More than one of the copies of an object can be a master in this approach.
- Changes to a master copy must be propagated to other copies somehow.
- If two master copies are changed in a conflicting manner, this must be resolved. (e.g., Site 1: Joe's age changed to 35; Site 2: to 36)
- Best used when conflicts do not arise:
  - E.g., Each master site owns a disjoint fragment.
  - E.g., Updating rights owned by one master at a time.

Primary Site Replication

- Exactly one copy of a relation is designated the primary or master copy. Replicas at other sites cannot be directly updated.
  - The primary copy is published.
  - Other sites subscribe to (fragments of) this relation; these are secondary copies.
- Main issue: How are changes to the primary copy propagated to the secondary copies?
  - Done in two steps. First, capture changes made by committed transactions; then apply these changes.

Implementing the *Capture* Step

- Log-Based Capture: The log (kept for recovery) is used to generate a Change Data Table (CDT).
  - If a transaction aborts when the log tail is written to disk, must somehow remove changes due to subsequently aborted transactions.
- Procedural Capture: A procedure that is automatically invoked (e.g. trigger!) does the capture; typically, just takes a snapshot.
- Log-Based Capture is better (cheaper, faster) but relies on proprietary log details.

Implementing the Apply Step

- The *Apply* process at the secondary site periodically obtains (a snapshot or) changes to the CDT table from the primary site and updates the copy.
  - Period can be timer-based or user/application defined.
- Replica can be a view over the modified relation!
  - If so, the replication consists of incrementally updating the materialized view as the relation changes.
- Log-Based Capture plus continuous Apply minimizes delay in propagating changes.
- Procedural Capture plus application-driven Apply is the most flexible way to process changes.

Data Warehousing and Replication

- A hot trend: Building giant "warehouses" of data from many sites.
  - Enables complex decision support queries over data from across an organization.
- Warehouses can be seen as an instance of asynchronous replication.
  - Source data typically controlled by different DBMSs; emphasis on "cleaning" data and removing mismatches ($ vs. lei) while creating replicas.
- *Procedural capture* and *application* apply best for this environment.