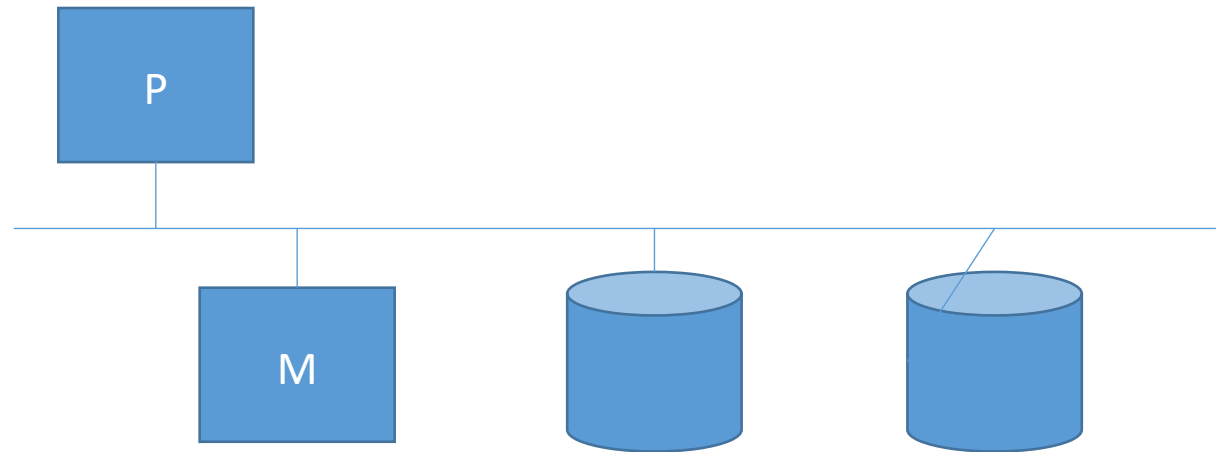


# Baze de Date Distribuite

# Introducere

- Sisteme de BD centralizate:
  - centralizarea blocărilor
  - dacă procesorul eșuează,  
întreg sistemul eșuează...

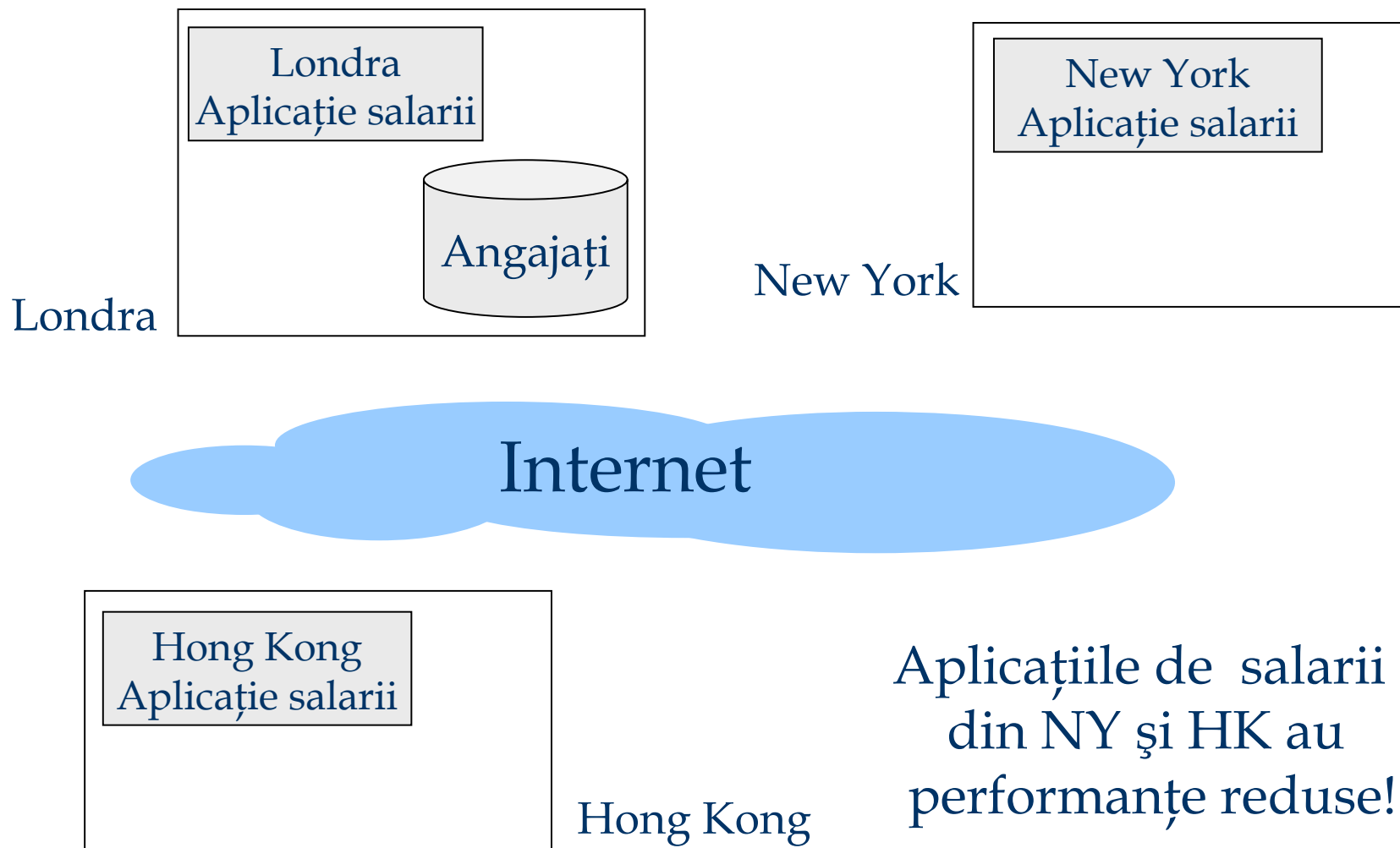


- Sisteme distribuite:
  - Procesoare (+ memorii) multiple
  - “Componente” autonome și eterogene

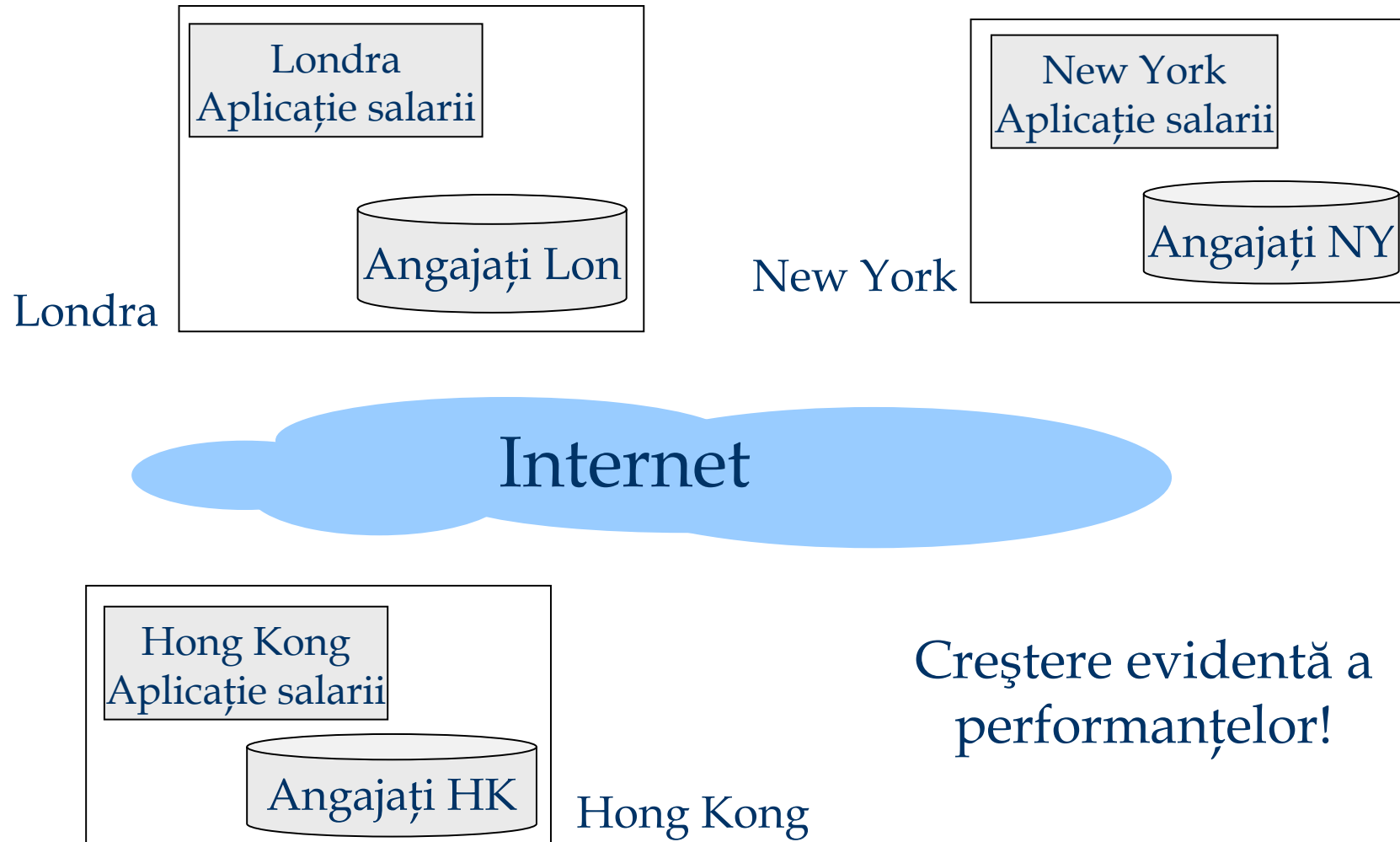
# De ce este nevoie de baze de date distribuite?

- Exemplu: Big Corp are birouri în *Londra, New York* și *Hong Kong*.
- În general, datele unui angajat sunt gestionate de la biroul unde acest angajat lucrează
  - De ex. date legate de salarii, beneficii etc
- Periodic, Big Corp are nevoie de rapoarte ce conțin informații despre toți angajații săi
  - Ex. Calculul bonusului anual ce depinde de profitul global net.
- Unde ar trebui să fie salvată baza de date de angajați?

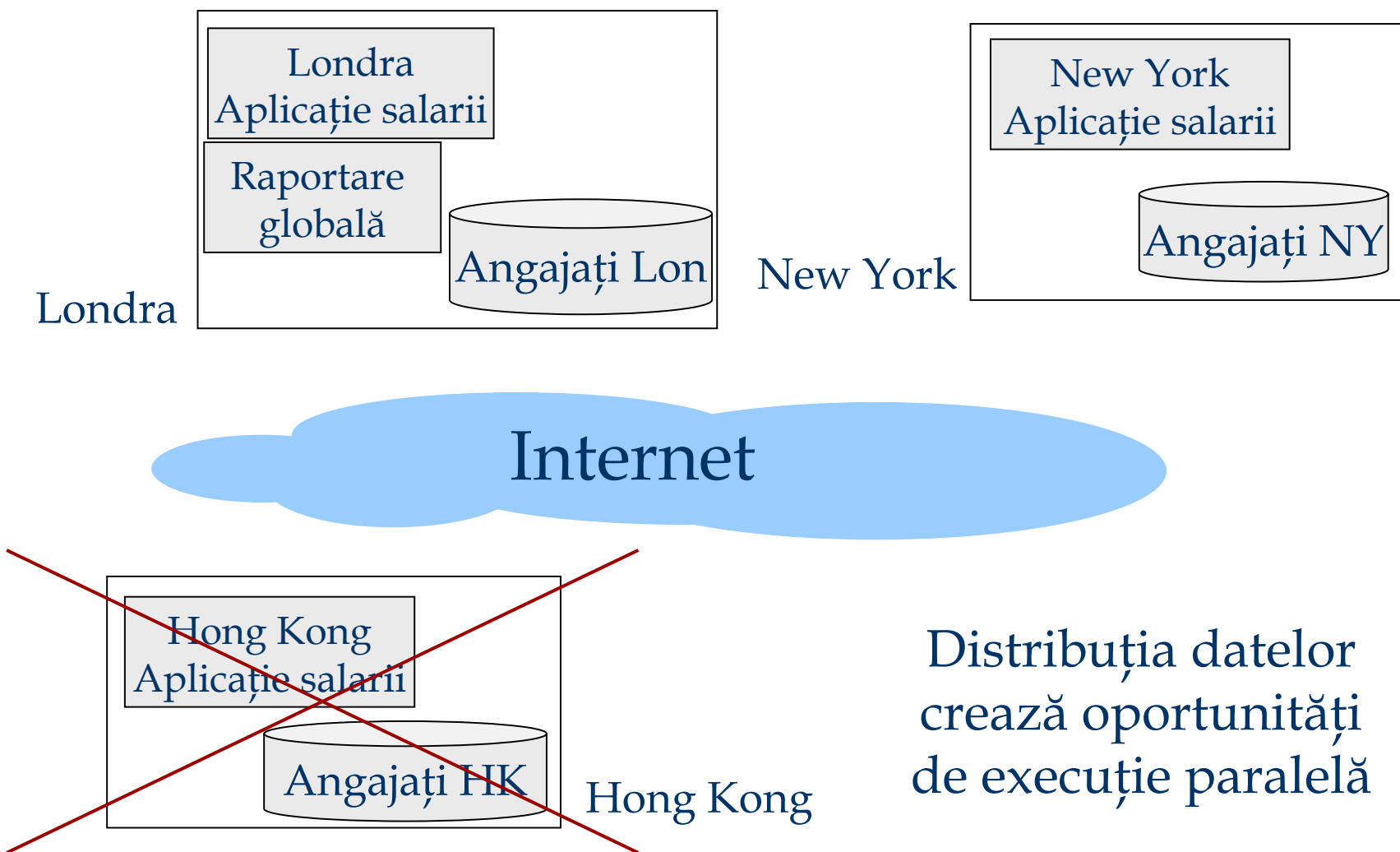
# De ce este nevoie de baze de date distribuite?



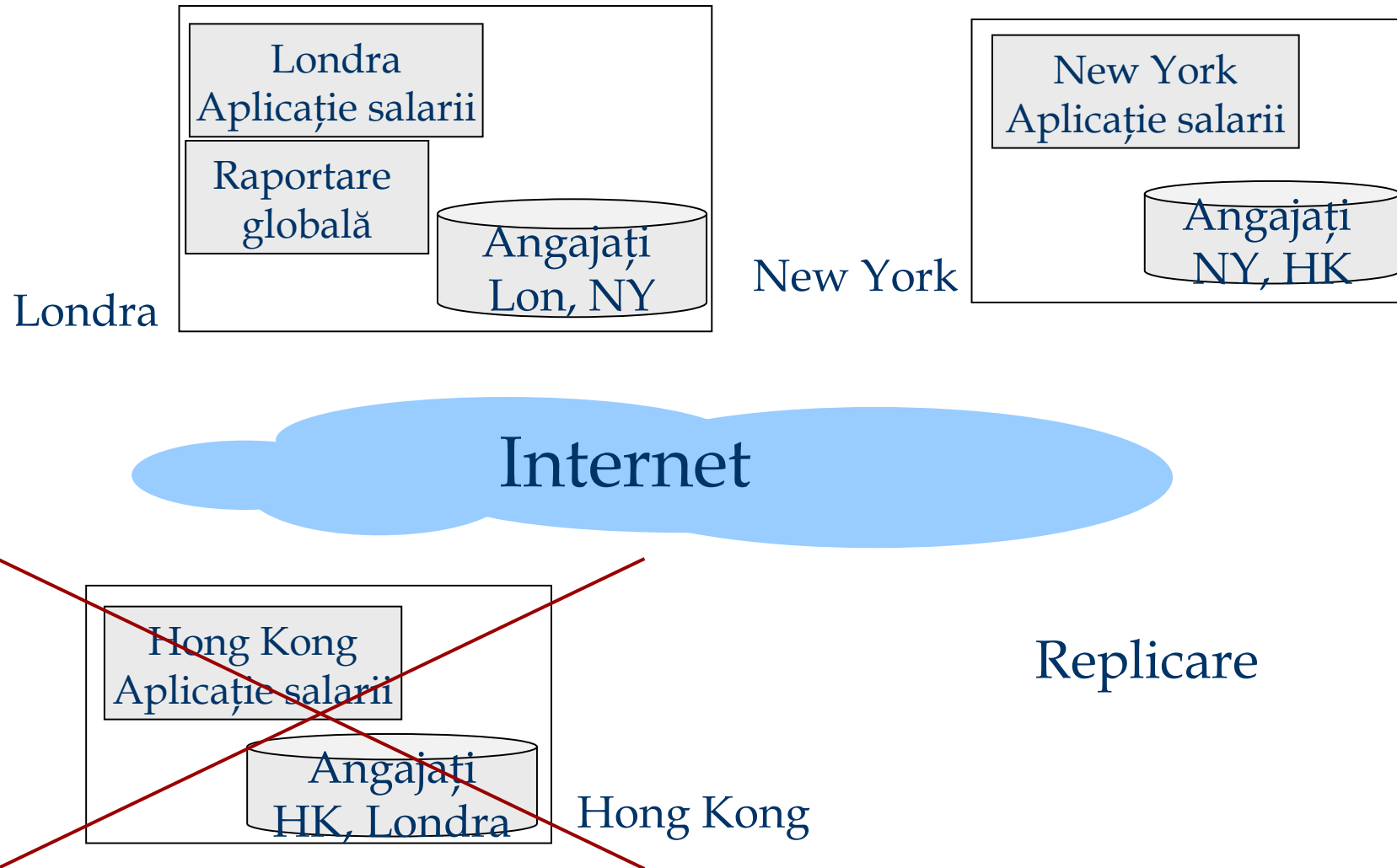
# De ce este nevoie de baze de date distribuite?



# De ce este nevoie de baze de date distribuite?



# De ce este nevoie de baze de date distribuite?



# Baze de date distribuite

- Independența Datelor Distribuite
- Atomicitatea Tranzacțiilor Distribuite



# BDD – Avantaje

- autonomia locală
- performanță în accesarea datelor
- disponibilitate
- modularitate

# BDD

## ■ *Avantaje:*

- **autonomia locala:** un grup de utilizatori / administratori care fac parte dintr-un departament pot controla datele care se regasesc în baza de date situată în acel sector;
- **performanta in accesarea datelor:** datele ar trebui localizate la diferite site-uri a.i. sa se minimizeze costul de comunicare si timpul de raspuns (ideea e sa fie localizate unde sunt accesate cel mai frecvent) sau a.i. sa se echilibreze volumul de operatii efectuate la fiecare locatie (to equalize the workload) => se reduc comunicatiile dintre diferitele locatii ale BDD;
- **disponibilitate:** in cazul replicarii – daca pica un server pe care este memorata o copie, se poate accesa o alta copie de la o alta locatie; in general, daca sunt folosite mai multe server-e de capacitate mai mica decat mai putine server-e de capacitate mare – este mai eficient din punct de vedere al costurilor (economie) si al riscului (pierderea unui server nu este atat de grava pentru intreg sistemul);
- **modularitatea:** se permite adugarea de noi module sau stergerea lor fără a afecta semnificativ întregul sistem.

# Baze de date distribuite – (dez)avantaje

- Proiectarea bazelor de date
  - Fragmentarea si alocarea datelor,
  - Tipuri de fragmentare:
    - **orizontală**, unde rândurile sunt împărțite în subseturi și stocate separat;
    - **verticală**, unde coloanele sunt divizate și stocate în locații diferite; și
    - **hibridă**, care combină ambele metode.
  - Asigura scalabilitate si disponibilitate, complica gestiunea datelor, necesita strategii pt a evita inconsistentele, pierderi de performanta

# Baze de date distribuite – (dez)avantaje

- Procesarea interogărilor distribuite
  - Costuri de comunicare între site-uri/ servere diferite,
  - Oportunitatea procesării în paralel
- Controlul concurenței
  - Serializabilitate
  - Gestiunea deadlock-urilor
  - Propagarea modificărilor ( la replici)

# Baze de date distribuite – (dez)avantaje

- Pastrarea consistentei
  - Multiple modalitati de esec
  - Sincronizarea datelor
- Controlul concurenței
  - Serializabilitate
  - Gestiunea deadlock-urilor
  - Propagarea modificarilor ( la replici)

# Tipuri de baze de date distribuite

- SGBD singular
- SGBD multiplu
  - Omogene
  - Eterogene

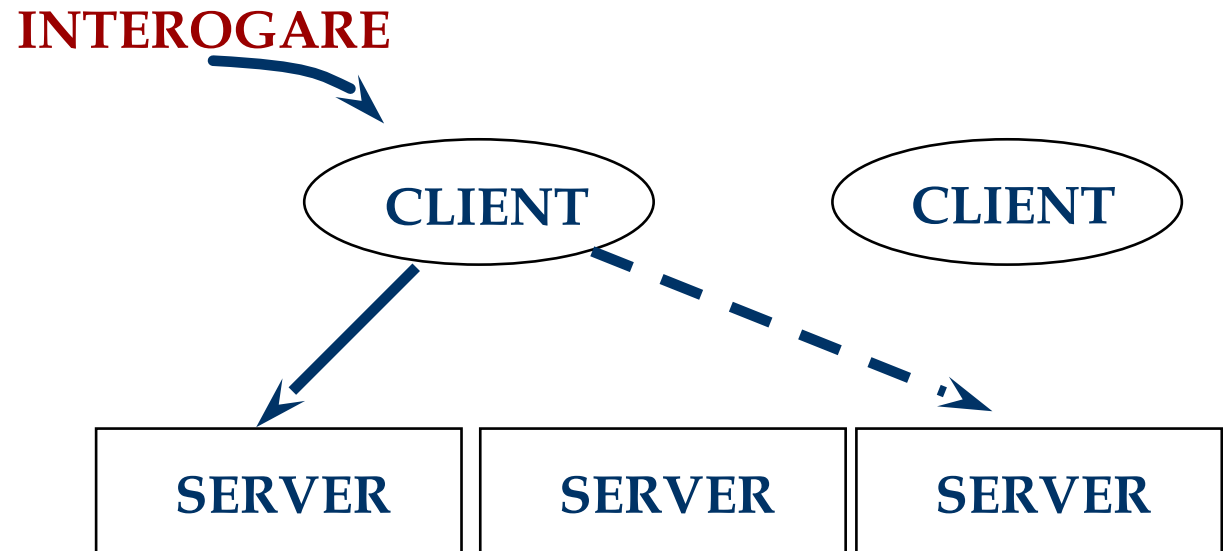


# Arhitecturi de SGBD distribuite

## ■ *Client-Server*

Clientul transmite interogările către un singur *site*. Toate interogările sunt procesate pe *server*.

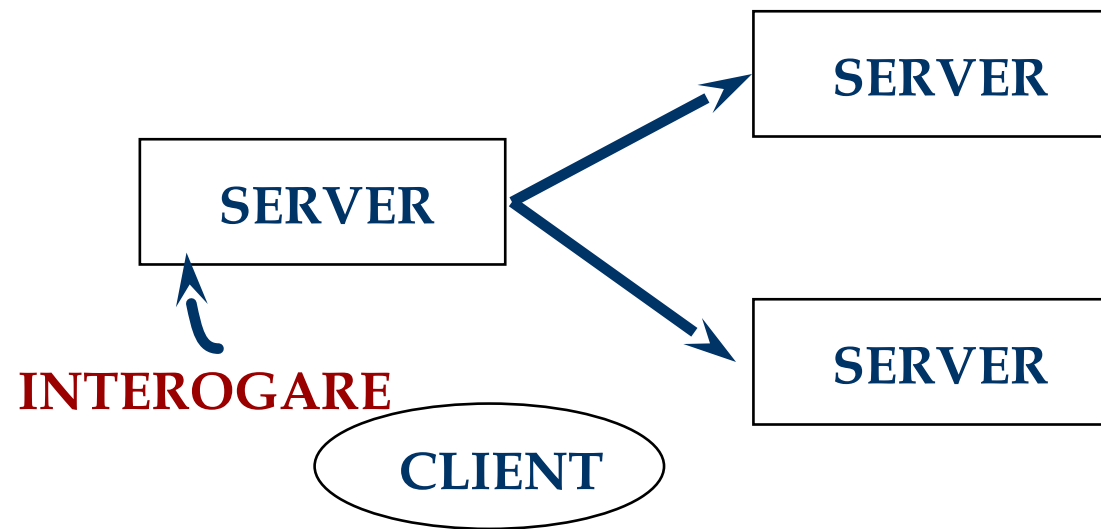
- Clienți *thin* și *fat*.
- Comunicarea orientată pe mulțimi de date



# Arhitecturi de SGBD distribuite

## ■ Server colaborativ

Interogările “acoperă”  
mai multe *site-uri*

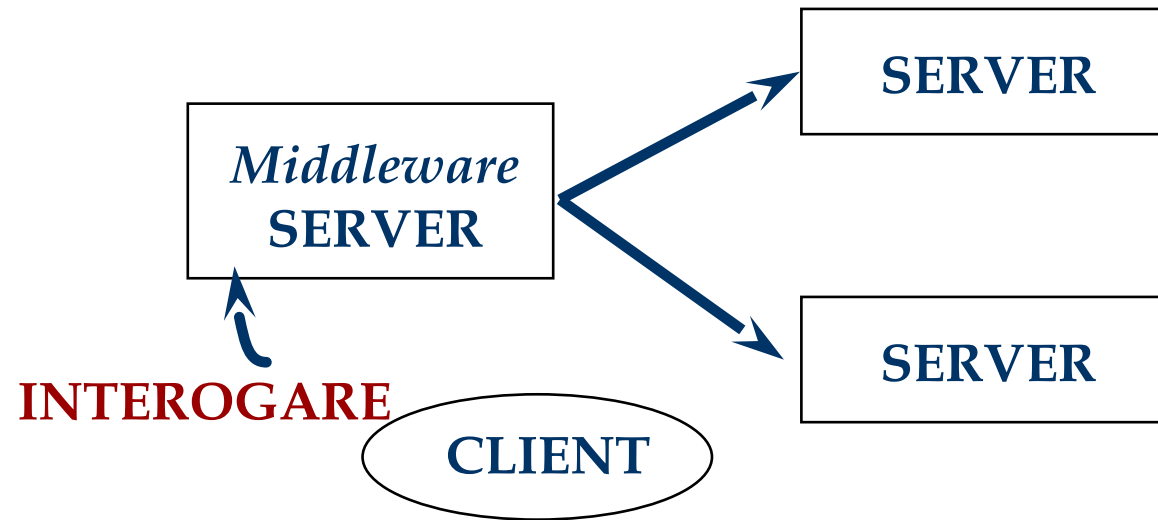




# Arhitecturi de SGBD distribuite

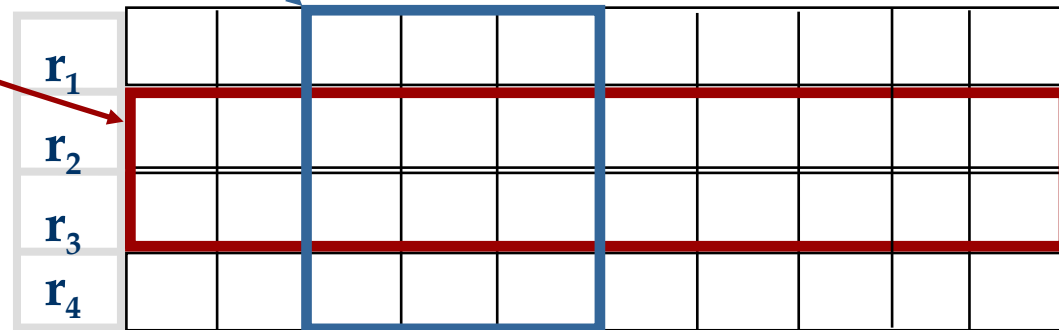
## ■ *Middleware System*

Un server gestionează interogările și tranzacțiile executate pe servere multiple



# Stocarea datelor - Fragmentare

- *Orizontală*
  - Primară
  - Derivată
- *Verticală*



# Stocarea datelor - Fragmentare

## ■ Proprietățile ale fragmentării

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

### *Completitudine*

$$\forall x \in R, \exists F_i \in \mathbf{F} \text{ astfel încât } x \in F_i$$

### *Disjunctivitate*

$$\forall x \in F_i, \neg \exists F_j \text{ astfel încât } x \in F_j, i \neq j$$

### *Reconstrucție*

Există o funcție  $g$  astfel încât

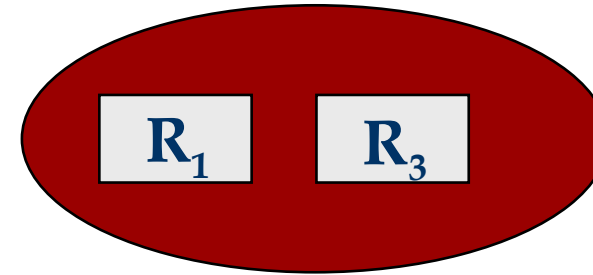
$$R = g(F_1, F_2, \dots, F_n)$$

# Stocarea datelor - Replicare

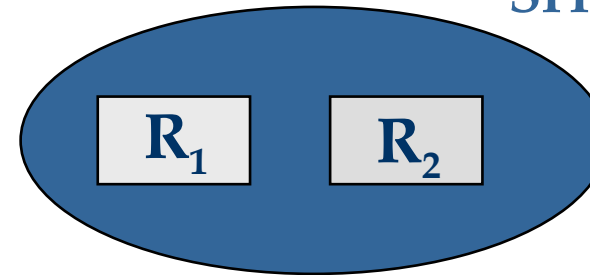
- Avantaje:
  - Crește disponibilitatea datelor.
  - Evaluarea rapidă a interogărilor

R e fragmentat în  $R_1, R_2, R_3$   
 $R_1$  e replicat pe ambele site-uri

**SITE A**



**SITE B**



- Probleme:
  - Propagarea modificărilor
    - **Sincron** vs. **Asincron**

# Catalog distribuit

- **Catalog:** Descrie toate obiectele (fragmente, replici) aflate pe un *site* + ține evidența tuturor replicilor tabelelor create pe acel *site*
  - Pentru găsirea unei table se va consulta catalogul *site*-ului unde această tabelă a fost creată
- Replica fiecărui fragment are un nume global unic:
  - **<nume-local, site-origine, id\_replica>**

# Catalog distribuit

## Catalog global centralizat

- conține informația corespunzătoare tuturor relațiilor, fragmentelor, cópiilor și este memorat pe un singur site
- un asemenea catalog ar aglomera serverul respectiv și este vulnerabil la o cădere a serverului pe care se găsește

# Catalog distribuit

## Catalog global replicat la fiecare site

- fiecare copie a catalogului conține informația corespunzătoare întregii baze de date distribuite
- nu este vulnerabil la căderea serverului (deoarece informația necesară se poate prelua de la o altă locație)
- orice actualizare a catalogului la o locație trebuie propagată pe toate celelalte locații (se compromite autonomia locală)

# Catalog distribuit

## Catalog local distribuit

- fiecare site menține un catalog local care descrie toate copiile datelor stocate pe acel *site*
- autonomie locală + nu este vulnerabil la eșecul unui *site*
- catalogul de la *site*-ul de origine al unei relații ține evidența fragmentelor / replicilor relației
- la crearea unei noi replici sau la mutarea unei replici la o altă locație, trebuie actualizată informația de la *site*-ul unde a fost creată relația (de origine)



# Actualizarea datelor distribuite

## Replicare sincronă:

- Toate copiile unei tabele modificate de o tranzacție trebuie să fie actualizate înainte ca tranzacția să se comită.
- Distribuirea datelor e transparentă utilizatorilor.

# Actualizarea datelor distribuite

## Replicare asincronă:

- Copiile tabelelor sunt actualizate doar periodic
- Utilizatorii sunt conștienți de faptul că datele sunt distribuite
- Multe dintre produsele curente urmează această abordare

# Tehnici de replicare sincronă

## A. Citește-orice / Modifică-tot (ROWA)

- Modificările sunt mai lente și citirile sunt mai rapide în comparație cu tehnica votării.
  - Cea mai utilizată metodă de sincronizare a replicărilor.
  - Aduc informația pentru citire de la cel mai apropiat server-> citirile rapide
  - Scriere – proces mult mai lent
- Alegerea tehnicii determină *ce* blocări sunt utilizate

# Tehnici de replicare sincronă

## B. Votare (consensus al cvorumului)

- Tranzacția trebuie să modifice o majoritate de copii ale unui obiect; de asemenea trebuie citite suficiente copii pentru a se asigura accesul la una dintre copiile recente.
  - Ex. 10 copii; 7 actualizate la modificări; 4 copii la citiri.
  - Fiecare copie are un număr de versiune.
  - Citirile fiind activități comune  $\Rightarrow$  nu e o abordare des utilizată.

# Costul replicării sincrone

- Înainte ca o tranzacție ce face o modificare să fie comisă, aceasta va trebui să blocheze toate copiile tablei/fragmentului modificat.
- Se transmit cereri de blocare către diverse site-uri, iar până la primirea răspunsului se mențin alte blocări!
- Dacă rețeaua/site-urile eșuează, tranzacția nu se poate comite până ce acestea nu-și revin.
- Chiar și în absența eșuărilor, *protocolul de comitere* poate fi costisitor, cu multe mesaje
- Alternativa *replicării asincrone* este, de aceea, mai utilizată.

# Replicare asincronă

- Permite ca tranzacțiile să fie comise înainte ca toate copiile să fie actualizate (și citirile se fac folosind o singură copie).
- Utilizatorii trebuie să fie conștienți de ce copie citesc și de faptul că, pentru o scurtă perioadă de timp, copiile pot să fie desincronizate.
- Două abordări: **Site Principal** și **Peer-to-Peer**
  - Diferența constă în numărul de copii ``actualizabile'' sau ``master''.

# Replicare *Peer-to-Peer*

- Mai multe copii ale unui obiect pot fi *master* în această abordare.
- Modificările unei copii *master* trebuie să fie propagate către celelalte copii.
- Trebuie rezolvat conflicte ce apar atunci când două copii *master* sunt modificate (conflict: Site 1: vârsta lui Joe se modifică la 35; Site 2: la 36)
- E cea mai bună abordare în cazurile când nu pot apărea conflicte:
  - Ex: fiecare site *master* deține un fragment disjunct.
  - Ex: Drepturile de actualizare sunt deținute de un singur *master* la un moment dat

# Replicare cu *site* principal

- Doar o copie a unei tabele este considerată copie **primară** sau *master*. Replicile facute pe alte site-uri nu pot să fie modificate direct.
- Copia primară este **publicată**.
- Celelalte *site*-uri **subscriu** la această copie; ele se numesc copii **secundare**.
- Cum se propagă modificările dinspre copia primară către copiile secundare?
  - În două etape: mai întâi se **capturează** modificările făcute de tranzacțiile comise apoi se **aplică** aceste modificări



# Implementarea etapei Capture

## Pe bază de log

- logul (menținut pentru recuperare) se utilizează la generarea structurii *Change Data Table* (CDT)
- modificările tranzacțiilor care se anulează trebuie înlăturate din CDT
- în final, CDT conține doar înregistrările log de tip update ale tranzacțiilor comise

## Procedural

- captarea este realizată de o procedură invocată automat (e.g., un *trigger*); aceasta realizează un *snapshot* al copiei primare

# Implementarea etapei **Capture**

Captarea *pe bază de log* este mai bună  
(mai puțin costisitoare, mai rapidă),  
dar se bazează pe unele particularități ale logului specifice  
sistemului

# Implementarea etapei **Apply**

Etapa *Apply* aplică schimbările captate în faza anterioară (în CDT sau *snapshot*) cópiilor secundare

- *site-ul* primar poate trimite continuu CDT

sau

- *site-ul* secundar poate solicita periodic (ultima porțiune din) CDT sau un *snapshot* de la *site-ul* primar; intervalul dintre solicitări poate fi controlat de un *timer* sau din aplicație
- pe fiecare *site* secundar rulează o copie a procesului *Apply*

# Blocare distribuită

## ■ Centralizat

- vulnerabilitate ridicată – depinde de un *site*.

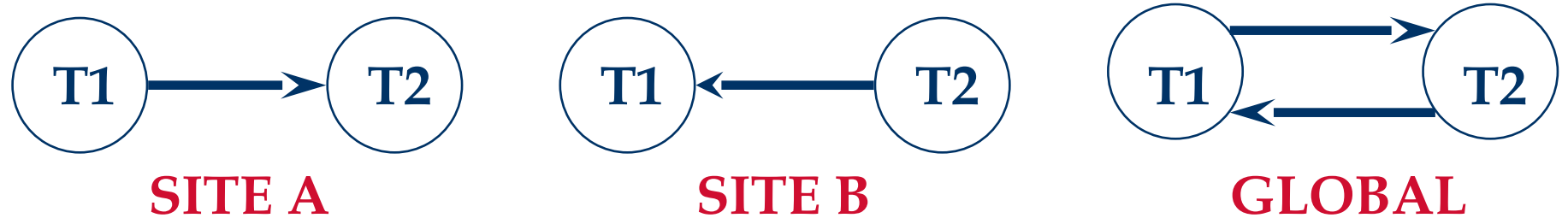
## ■ Copie principală

- Citirea unui obiect necesită acces atât la *site*-ul principal cât și la *site*-ul unde e salvat obiectul

## ■ Complet distribuit

- Modificarea unui obiect presupune blocarea tuturor *site*-urilor unde se găsește obiectul respectiv

# Detectarea *deadlock*-urilor distribuite



**Centralizat** (grafurile locale sunt trimise către un singur *site*);

**Ierarhic** (organizare ierarhică a *site*-urilor, grafurile locale sunt transmise părintelui);

**Timeout** (tranzacțiile sunt întrerupte dacă durează prea mult).