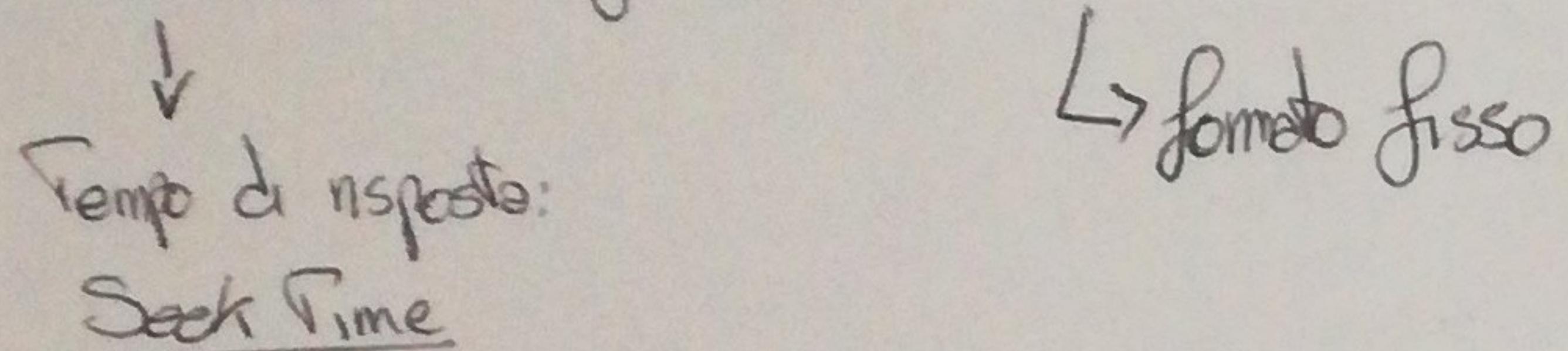


## Organizzazione Fisica

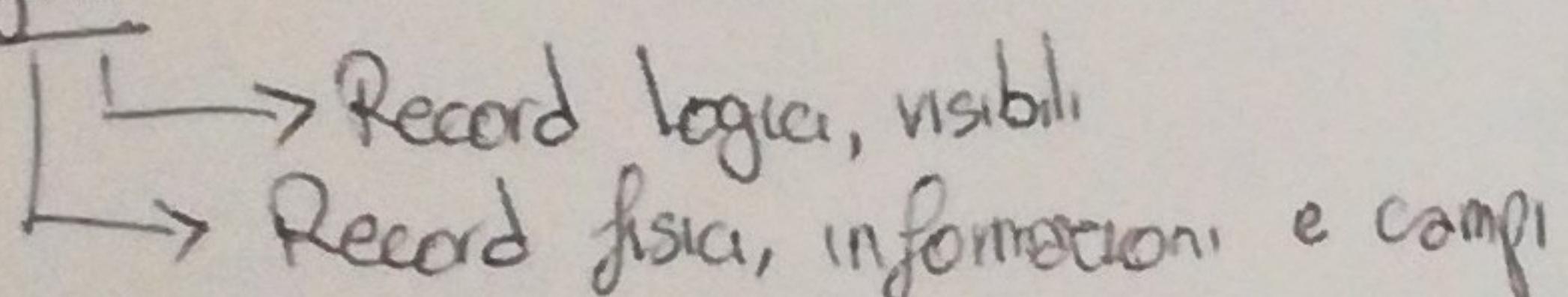
- 3 Livelli: → Utente: Tabelle  
 → Admin: Schema ER  
 → Fisica: memoria, buffer, indice, file
- Organizzazione efficiente

Memorie secondarie: organizzate in blocchi, con lettura/scrittura



Buffer: tramite mem. principale e secondaria, organizzato in pagine

Dati organizzati in file:



3 File:

- Heap: sequenze: record inseriti in modo disordinato, ricerche singole costive  
 cluster predefiniti a priori (array)  
 associazioni indici coordinate)

- Hash: record suddivisi in bucket, collegati con puntatori

$$F = \text{resto}(I/B) \quad f: \text{funzione hash}$$

$$\Sigma: \text{chiave (int)}$$

$$B: \text{num. bucket}$$

$$B = \lceil (f \times F) \rceil \quad \Gamma: \text{numero tuple}$$

$$f: \text{frazione fisica}$$

$$F: \text{fattore blocco}$$

$$B: \text{num. bucket}$$

- Indice: struttura ad albero

- ↳ Primo: chiave primaria + dati  
 ↳ Secondario: chiave valore chiave + indirizzo fisico

## Index Tuning

Accesso random  $\rightarrow$  Index

(piú lento: seek time)

Accesso sequenziale  $\rightarrow$  Full Table Scan

• Dati return  $\leq 20\%$  del tot

↓  
Index Access

- Colonna indicizzata con dati (distribuiti) non uniformi  $\Rightarrow$  Index dipende dal valore

↓

Hint

Straight-Join

Use Index / Ignore Index / Force Index

Operazioni      Sorting: effettuata su mem. secondaria (lento)

- Distinct
- Group by
- Union
- Order by

- Index su più colonne  $\rightarrow$  vanta sort
- Index su colonna: min/max veloce

# ~~Architettura~~ Ottimizzazione Query

## Verificare:

- Dati necessari
- Tempo esecuzione
- Righe esaminate - Righe Tornate

## 3) Parser/Pre-processor

- Divisione query in parse tree
- Interpreta query

## 5) Execution Plan

- Poco istruzioni
- show warnings;

## Architettura MySQL

- 1) Client/Server:
  - Half duplex
  - Invia dati, sleep, query
- 2) Query Cache:
  - Cerca, con hash, la presenza della query
  - Memorizza risultato

## 4) Query optimizer

- Prende costo dal Parse Tree
- Riordina join, crea BTree, alg. relazionale

## 6) Execution Engine

- Piano esecuzione come struttura dati
- Invoca metodi API

## Join

- 1) Nested loop: full table scan su tabella esterna + accesso diretto tabella interna
- 2) Merge scan: due scansioni parallele, stop su condizione di join  
(tabelle ordinate)
- 3) Hash join: file hash per memorizzare la copia di 2 tabelle, tabelle suddivise in bucket per valore di join



Come scegliere

- op. da eseguire (scan o accesso)
- ordine operazioni (~~in~~ in che ordine il join)
- quale tipo di join (Nested, Merge, Hash)

## Piano ottimizzazione

- Albero decisione - Piani Esecuzione  $\Rightarrow$  soluzione buona, non ottima
- Calcolo costo di ogni piano
- Sceglio piano con costo minore

## Gestione Transazioni

- Unità elementare di lavoro, inizio e fine  $\Rightarrow$  esecuzione
  - ↓
  - commit
  - Rollback

Sistema Transazionale (OLTP)

Definisce l'esecuzione di transazioni  
per determinate applicazioni

Proprietà:

- 1) Atomicità: Transazione atomica (unica), deve essere completata (esito positivo/esito negativo)
- 2) Consistenza: rispetta vincoli integrità
- 3) Isolamento: la transazione NON deve intacciare altro
- 4) Durabilità: l'effetto della transazione rimane nel tempo

Gestori (modelli DBMS)

Affidabilità: gestisce esecuzione comandi, assicura atomicità, durabilità

Concordanza: le transazioni non possono essere sovrapposte  $\rightarrow$  conflitti

LOG

- File sequenziale, tutte le operazioni vengono riportate
- Utilizzare dump (backup) e, checkpoint per ripristinare il DB (crash)
  - ↓
  - UNDO
  - REDO
  - Copia completa del DB
- Regole: Write-ahead (scrive prima di eseguire)  
Commit - Precedenza (prima esegue, poi scrive)

Crash

- ↳ Soft: crash sistema  $\rightarrow$  warm restart (checkpoint)
- ↳ Hard: rottura disco  $\rightarrow$  cold restart (dump)

# Gestione Transazione

## Controllo Concorrenza

- Altre transazioni al secondo, NON possono essere senali.
- Esecuzione concorrente  $\rightarrow$  anomalie
  - Perdita aggiornamento: esecuzione senale
  - Lettura sporca: stato intermedio (transazione precedente non terminata)
  - Lettura inconsistente: doppia lettura, valori diversi
  - Aggiornamento fantasma: aggiornamento non coerente (vincolo <sup>non</sup> rispettato)
  - Inserimento fantasma: modifica nuovo dato

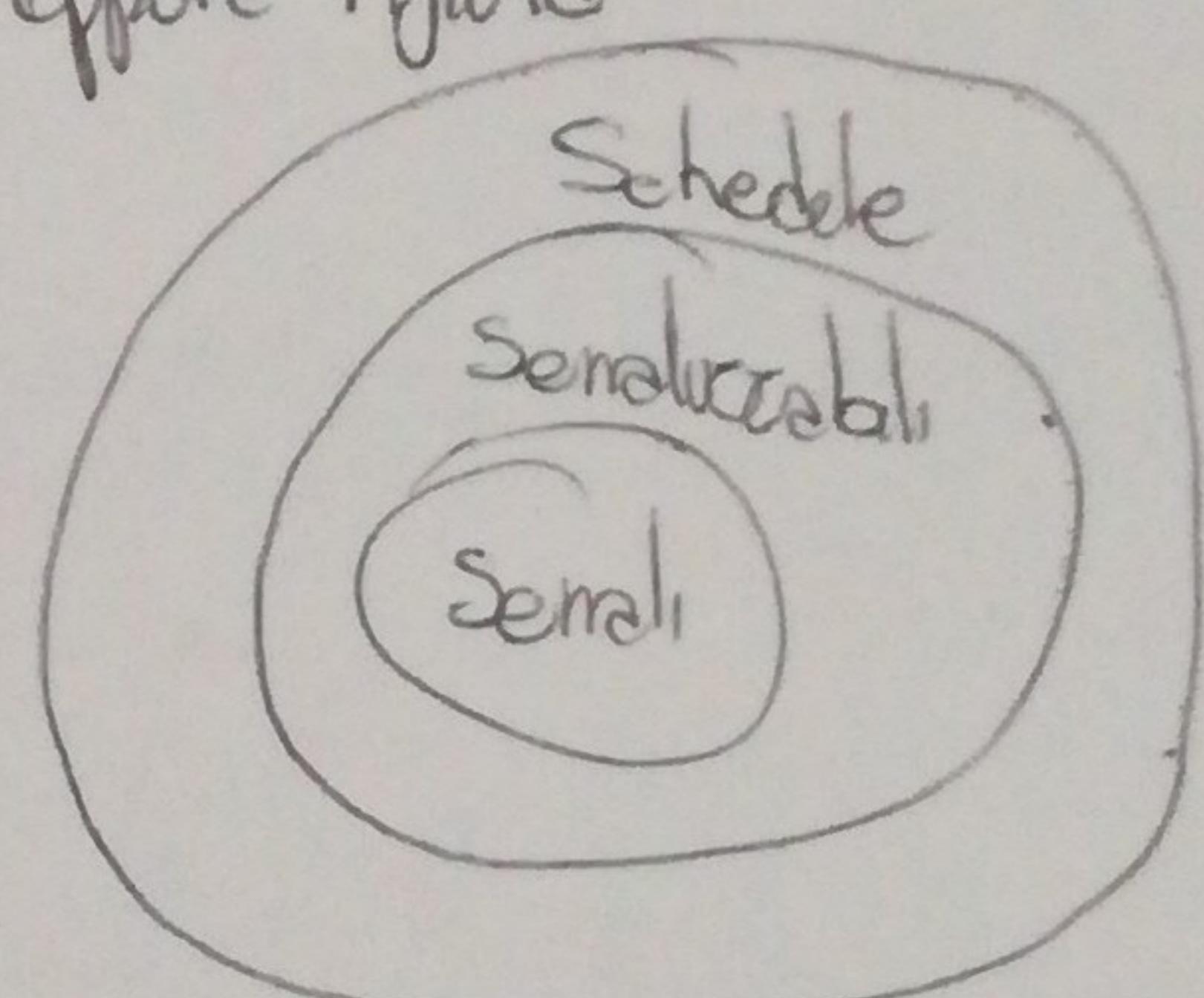
W-W  
R-W (W-W)  
+ abort  
R-W  
R-W  
R-W

## Schedule

Sequenza di operazioni I/O per transazioni, accetta oppure rifiuta

Schedule senale: transazioni separate, una alla volta

Schedule serializzabile: stesso effetto di quello senale



### View serializzabilità: (VSR)

- $R_i(x)$  legge da  $W_j(x)$  in schedule S se:
  - $W_j(x)$  precede  $R_i(x)$
  - Non ci sono altre  $W_k(x)$  nel mentre
- Stessa relazione "legge da"  $\Rightarrow$  schedule  $\leftrightarrow$  view equivalenti
- View serializzabile se equivalente a schedule senale

### Conflict-serializzabilità: (CSR)

- Azione ~~che~~ ai conflitti  $\xrightarrow{\text{stesso oggetto}}$  con  $A_3$  (RW-ww)
- Conflict-serializzabile se equivalente a schedule senale
- Stesse operazioni e stesso ordine  $\Rightarrow$  schedule conflict equivalenti

- Ogni CSR è VSR, ma non viceversa

## Gestione Transazione

### Lock

- Lettura: r-lock + unlock (condiviso)
- Scrittura: w-lock + unlock (esclusivo)
- Lettura-Scrittura:
  - lock esclusivo
  - Lock condiviso + esclusivo

Lock manager: gestisce Lock : lock, tabella conflitti

Lock 2 fasi: garantisce la CSH, basato su due regole

- proteggere W-R con lock
- una transazione, dopo aver ribassato un lock, non può richiederne altri

→ Dopo abort/commit

Sistema transactionale:

- Transazioni formate rispetto al locking
- lock manager rispetta tabella conflitti
- Transazioni seguono 2PL

## Gestione Transazioni

2PL  $\rightarrow$  Tanti lock  $\rightarrow$  molte transazioni terminate

$\downarrow$   
 $\Rightarrow$  passa al Timestamp

$\hookrightarrow$  ogni transazione ha un timestamp, rappresenta l'istante d'inizio

~~Schedule accettato se riflette l'ordine del timestamp~~

~~Schedule accettato se riflette l'ordine serale delle transazioni indotto dal timestamp~~

Contatori per ogni oggetto

• RTH(x)  
(ultimo read)

• WTM(x)  
(ultimo write)

• Uccise molte transazioni

- read( $x, ts$ ): ~~ts < RTH(x)~~  $\rightarrow$  richiesta respinta, transazione uccisa

- altrimenti, accetta  $\Leftarrow RTH(x) = \max(CURRENT_RTH(x), ts)$

- write( $x, ts$ ):  $ts < WTM(x)$  o  $ts < RTH(x)$   $\rightarrow$  richiesta respinta, transazione uccisa

- altrimenti, richiesta accetta e  $WTM = ts$

2PL vs Timestamp

• 2PL: transazioni in attesa, timestamp: transazioni uccise

• 2PL ~~causa~~  $\Rightarrow$  deadlock (stallo)

• Attesa commit  $\rightarrow$  rimuovere commit-protezione

Evoluzione  
Timestamp  
MVCC

MVCC

- read( $x, ts$ ): lettura sempre accettata

• RTH(x) -  $WTM_n(x)$   
(ogni transazione)

•  $RTH(x) = \max(CURRENT_RTH(x), ts)$

- write( $x, ts$ ): ~~ts < RTH(x)~~  $\rightarrow$  richiesta respinta, transazione uccisa

• altrimenti, nuova versione  $x$  con  $WTM_n(x) = ts$

Come risolvere

• Lock pessimistico: blocca tutto quando scrivo

• Controllo imperativo: sincronizzazione e mutex

• Sistema build out: ognuno il proprio DB

• A caso: qualsiasi cosa succede

## Lock Manager

- Processo invocato manualmente o automaticamente, tutti i processi DB
- Parametri:
  - r-lock( $T, x, encode, timeout$ )
  - w-lock( $T, x, encode, timeout$ )
  - unlock( $T, x$ )
- Timeout  $\Rightarrow$  rollback

## Lock Geografico

Granularità lock:

DB  $\rightarrow$  Toper  $\rightarrow$  Tabelle  $\rightarrow$  Tuple  $\rightarrow$  Campi  $\rightarrow$  Campo

- granularità fine: modificate localmente  $\rightarrow$  Troppo lock manager
- granularità grande: grande mole dati  $\rightarrow$  limite parallelismo

## Deadlock

- Due transazioni si incrociano, entrambe detengono una risorsa necessaria all'altra

↓  
Soluione  $\rightarrow$  statto = nodo in grafo attese (nodo = transazione,  
arco = attesa)

1. Timeout (scelta intervallo, trade-off)
2. Rilevamento statto (cerca cicli di attesa)
3. Prevenzione statto (uccisione transazioni "sospette")

# Architettura Distribuita

## Architettura Client-Server

- Basata su tecnologia server per gestione dati
- Comunicazione gestita con macchina client

## Basi di Dati Parallele

- Puntano su Prestazioni
- Macchine multiprocessor multiprocessor
- Portabilità: possibilità di trasportare dati da un sistema ad un altro
- Interoperabilità: capacità di interazione di sistemi ~~stocaghi~~ diversi

- DB Distribuiti:
- natura distribuita delle organizzazioni
  - evoluzione elaboratori (aumento capacità elaborativa)
  - evoluzione DBMS
  - standard di interoperabilità

- Problemi:
- Autonomia e cooperazione (accesso a più dati)
  - Trasparenza → frammentazione dati (completatezza e ricostruttibilità)
  - Efficienza
  - Affidabilità

## Basi di Dati Distribuite

- 2 o più server
- Grandi molti di dati
- Gestione transazioni + interazione
- Proprietà ACIDE

## Basi di Dati Replicati

- Collezione di dati
- Massima affidabilità e disponibilità

Orientale: (O, selezione)

[insieme di tuple], unione

Verticale: (T, proiezione)

[insieme di attributi], con PK/FK

## Livelli

Frammentazione (normale)

Allocazione (spazio pos. fisico)

Lingaggio (struttura frammenti + alterazione)

- Optimizzazione query

- Modello d'esecuzione:

- seconde
- parallela

# Architecture Distribute

## Classificazione Transazioni

Schema basato su statement SQL, classificazione:

- Remote request: lettura DBMS remoto
- Distributed transaction: transazioni multiple su molti DBMS remoti
- Remote transaction: transazioni multiple su DBMS remoto
- Distributed request: transazioni arbitrare (+DBMS)

Proprietà ACID:

- Atomicità: problema principale
- Consistenza: ciascuna sottotransazione preserva integrità locale, dati consistenti
- Isolamento: ciascuna transazione a 2 fasi, transazione serializzabile
- Durabilità: ciascuna transazione gestisce log, dati persistenti

Commit a 2 fasi

Garantisce atomicità di transazioni distribuite

- Transaction Manager (TM) (coordinatore)
- Resource Manager (RM) (partecipanti)

↓  
Log

- Prepare: identità partecipanti
- Global commit/abort: decisione
- Complete: termine protocollo

↓  
Log:

- Ready: disponibilità commit
- Local commit/abort: decisione

Fase 1:

TM: prepare

MSG=Ready

→ RM: ready

TM:  
~~if~~ MSG=Ready  
Commit  
if MSG=NO  
Abort

Fase 2:

TM: send decision

→ RM: receive decision  
write decision → log  
Send Ack

TM: Receive Ack  
write log  
complete

# Architettura Distribuita

## Commit 2 Fasi:

Complessità protocollo: possibili guasti →

- caduta TM (Coordinatore)
- caduta RM (partecipanti)
- perdite messaggi

## Recovery partecipanti

- Durante warm restart, ogni transazione dipende dall'ultimo record nel log
  - Azione genere oppure abort → azione disfatta
  - Commit → azione infatta
  - Ready → guasto durante commit → stato incerto
- Durante warm restart, si collegano le transazioni dubbie dubbie, si richiede l'esito (recovery remote)

## Recovery coordinatore

- Ultimo record log - "Prepare" → guasto TM causato da blocco RM. 2 opzioni:
  - "Global abort" su log + 2<sup>a</sup> fase da ripetere
  - ripetere 2<sup>a</sup> fase → raggiungere commit globale
- Ultimo record = "decisione globale" → RMs già bloccati (alcuni). TM ripete 2<sup>a</sup> fase

## Perdita messaggi

- Perdita "ready" o "prepare" sono indistinguibili → scatta Time-out, global abort
- Perdita "decision" o "ack" sono indistinguibili → scatta Time-out, TM 2<sup>a</sup> fase
- Partitionamento Rete: Nessun problema → global commit se RM e TM appartengono stessa partizione

## Architettura Distribuita

### Ottimizzazione 2PC

Abort presunto: TM riceve "remote recovery" da transazione utente  $\Rightarrow$  "global abort"

- Se vengono persi "prepare" e "global abort"  $\rightarrow$  comportamento corretto
- Complete messo  $\Rightarrow$  scrive su log: ready, global commit e commit locale

### Read-Only

RM svolge q.d. lettura  $\rightarrow$  risponde "read only" al "prepare" ed esce  
 $\rightarrow$  il TM ignora i RM "read-only"

### Protocollo di commit a fasi

- Processo TM duplicato su nodo differente (il 2<sup>o</sup> TM è a conoscenza delle decisioni precedenti)
- TM guasto, backup interviene:
  - attiva un ~~ultimo~~ ulteriore backup
  - continua esecuzione protocollo commit

### Standard Protocollo

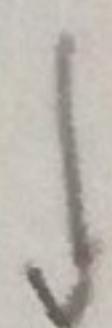
X-Open Distributed Transaction Processing (ODTP):

- TM: definisce i servizi, esegue commit parte o parti etrogeni
- XA: definisce i servizi partecipanti, rispondono al coordinatore

# Architettura Distribuita

## Basi Dati Parallelhe

- Utile per efficienza
- 2 Tipologie  
Parallelismo



- Carico: insieme delle query
- Scalabilità: conservare prestazioni derivate al crescere del carico
- Dimensioni: numero query crescenti      complessità query

Inter-query: oascuna query affidata ad 1 CPU (carico transazionale)

→ Intra-query: oascuna query affidata a + CPU (analisi dati)

Benchmark: confronto prestaionale

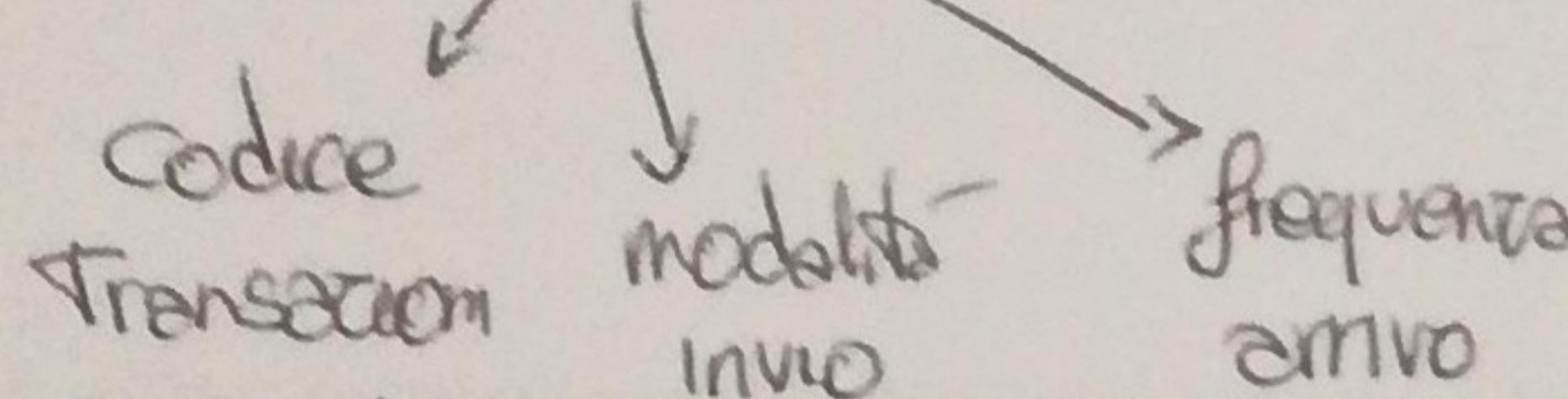
↳ 1. database

↳ 2. il carico

Tipologie Carico:

- transazionale
- misto
- analisi dati

↳ 3. modalità misurazione

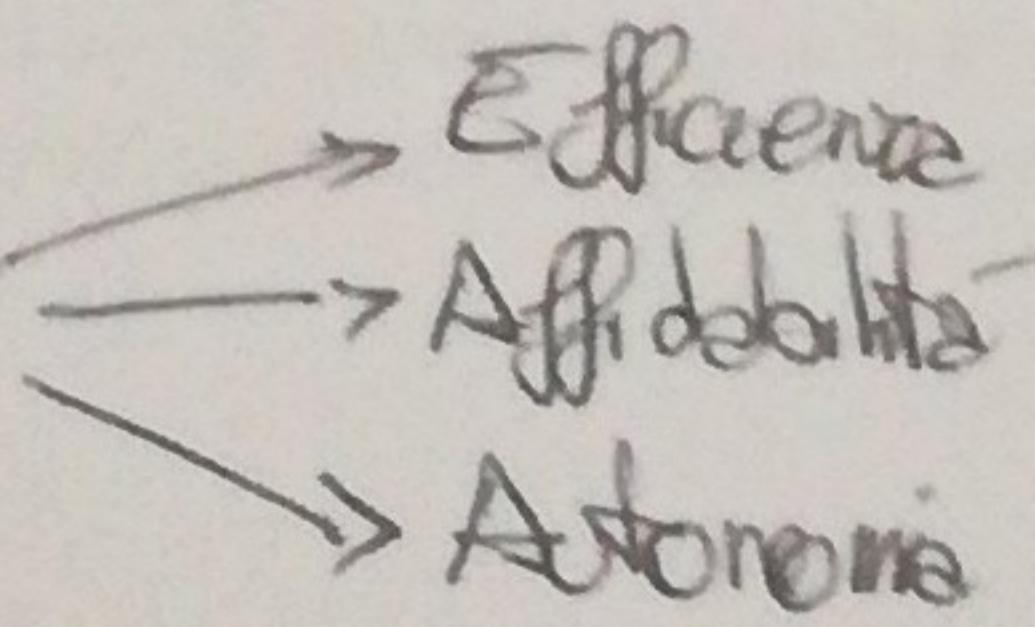


Curve speed-up: misura crescita efficienza per num. CPU

Curve scale-up: misure crescita costo complessivo per ~~transazione~~ per num. CPU

## Basi Dati Replicate

Replicazione di dati: fondamentale nei sistemi informativi



Modalità: Assimmetrica → modifica in copia primaria, propagazione copia secondaria

Simmetrica → modifiche su entrambe le copie, doppia propagazione

## Trasmissione versioni

Asincrona ⇒ transazione master (copia 1), transazione allineamento (copia 2)

Sincrona ⇒ transazione master (copia 1 + copia 2)

## Architetture Distribuite

### Alimentato:

- Refresh: contenuto copia 1 → copia 2 (periodico, a comando, accounto rinnovo)
- Incrementale: delta-plus (informazioni aggiunte) → copia 2  
delta-minus (informazioni rimosse)
  - Trigger, after insert ⇒ inserisce in delta-plus
  - after delete ⇒ inserisce in delta-minus