

## RIPRESA A FREDDO:

Accedo al DUMP e Ricepio Selettivamente La Parte deteriorata in  
Modo da Riportarmi ad Un guasto di dispositivo (Vista che ho  
ripristinato la MEM. CENTRALE)

Accedo al Log dal DUMP

Ripercorro in Avanti il LOG Ricseguendo TUTTE le Operazioni;

Applico RIPRESA A CALDO

## RIPRESA A CALDO:

Partendo dal Fondo Risalgo fino al Primo CheckPoint

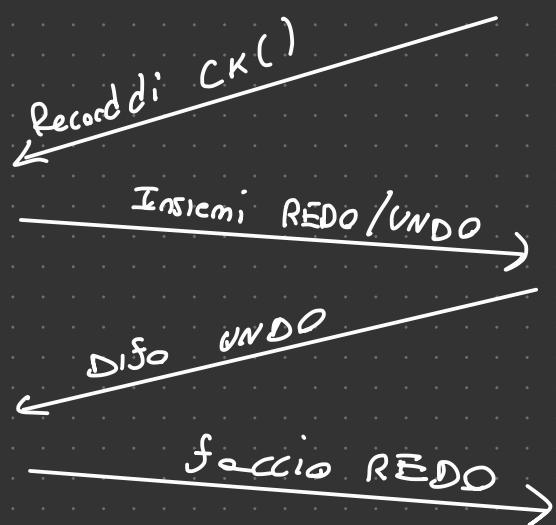
Inizializzo 2 Insiemi:

UNDO  $\rightarrow$  Transazioni Correnti

REDO  $\rightarrow$  A Vuoto

Scorro in Avanti il File di Log e Se Incontro  $B(T)$  Aggiungo T all' Insieme UNDO mentre con  $C(T)$  all' Insieme REDO eliminandolo da UNDO

Ripercorro All' Indietro il LOG disfacendo TUTTE le OPERAZIONI!  
Eseguite da  $T_i \in UNDO$  e poi Rifaccio quelle di  $T_i \in REDO$



- ① Parto dai Nodi foglia e li Collego Tra di Loro
  - ② Quanti Puntatori Mi Servono?
  - ③ Creo il Nodo Radice
- 

### VINCOLI DI RIEMPIMENTO:

**Nodo Foglia**  $\Rightarrow \left\lceil \frac{n-1}{2} \right\rceil \leq \# \text{Chiavi} \leq n-1$

**Nodo Intermedio**  $\Rightarrow \left\lceil \frac{n-1}{2} \right\rceil \leq \# \text{Puntatori} \leq n$

PER INSERIMENTO Controllare VINCOLI RIEMPIMENTO MAX. ed eventualmente fare lo **SPLIT** e Propagarlo Se Necessario

Aggiungere 1 Puntatore al PADRE e Modificare Gli Altri

PER RIMOZIONE Controllare VINCOLI RIEMPIMENTO MINIMI ed eventualmente fare il **MERGE**

Togliere un PUNTATORE dal Padre e Adattare Gli Altri

## SERIALIZZABILITÀ:

① ANALIZZO SE RISPETTA IL 2PL  $\Rightarrow$  Dopo la Prima W la Ti non deve fare Nuove R/W che significano nuovi lock

Se 2PL  $\Rightarrow$  CSR  $\Rightarrow$  VSR

## ② CSR

① Insieme dei Conflitti

② Creo Grafo:

■ Un Nodo A Transazione

■ Arco (i,j) per Ogni Conflitto (ai,aj)

③ È Aciclico?

④ Seguo ORDINAMENTO TOPOLOGICO per Restituire uno SCHEDULE

Parto da un NODO SENZA ARCHI ENTRANTI, che è un Candidato Per Essere Primo e Scelgo Quelli che non hanno Vincoli Tra di Loro. Seguendo gli Archi USCENTI

## ③ VSR

① Insieme Legge-Da e Scritture finali

② Condizioni per S' dei 2 Insiemi

③ Se le CONDIZIONI non Generano l'Assurdo Allora ho Trovato uno Schedule Equivalente

④ Verificare se i 2 Insiemi Sono uguali

## OTTIMIZZAZIONI & STIMA COSTO:

**SELECT**  $\Rightarrow \frac{NP(Leggo)}{LETTURA} + \frac{\text{Pagine Salvataggio}}{\text{SCRITTURA}} \cdot \text{Scrittura}$

**JOIN**  $\Rightarrow \frac{NP(R)}{NR(R)} + \frac{NP(S)}{NR(R) \cdot VAL(A,R)}$

eventuale Scrittura fatte Prima

Se fosse Nel BUFFER Sarebbe  $\emptyset$

$\left[ \begin{array}{l} \text{Solo le Righe Selezionate} \\ \text{Nella Select (se } \neq \text{ devo} \\ \text{fare TOT - risultato} \end{array} \right]$

Se Avessi più  $\bowtie$ , Allora devo fare 2 Stime per Capire #Righe

$$\frac{NR(R)}{VAL(A,R)} \cdot \frac{NR(R')}{VAL(A,R')}$$

**JOIN B+-TREE**  $\Rightarrow NP(R) + NR(R) \left( Prof.Indice + \frac{NR(S)}{VAL(A,S)} \right)$

Stare ATTENTI dove Viene messo l'INDICE, a Volte può essere inutile come se fosse sulla Tab. R (va Scansionata Comunque tutta)

Se Cambia la SELEZIONE allora al Posto di  $NP(R)$  diventa:

$Prof. Indice + \frac{NR(R)}{VAL(A,R)}$

**DBMS** è un Sistema Software utilizzato per Gestire Collezioni di dati che Siano GRANDI / PERSISTENTI / CONDIVISE assicurando la loro AFFIDABILITÀ e PRIVATEZZA.

DBMS estende le funzionalità dei FILESYSTEM essendo che integrato con la MEMORIA SECONDARIA

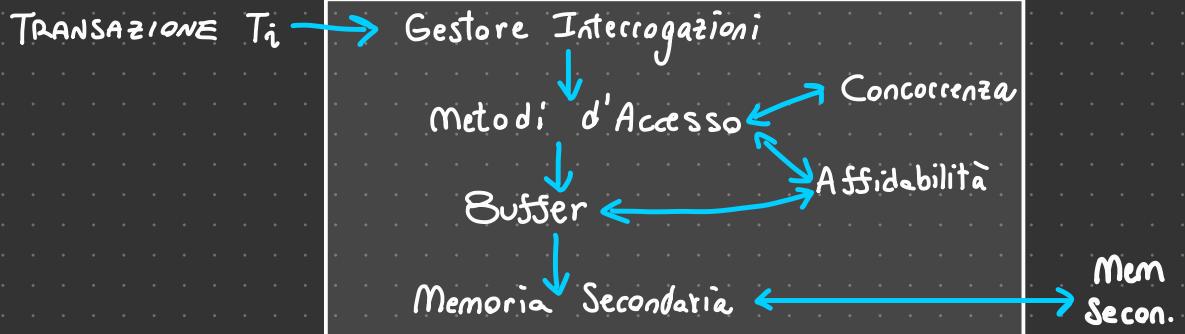
La Transazione è un' UNITÀ DI LAVORO svolta da un programma Applicativo, è un'operazione ATOMICA, o tutto o NIENTE

Eseguito il Commit e ci deve essere Effetto Sulla Base di Dati

Fatto ABORT e si deve fare un ROLLBACK  
Per TORNARE alle situazioni precedente

- PROPRIETÀ**
- A → Atomicità ⇒ Esecuzione INDIVISIBILE
  - C → CONSISTENZA ⇒ Rispetta VINCOLI DI INTEGRITÀ
  - I → ISOLAMENTO ⇒ Indipendente da EXEC Altre TRANSAZIONI
  - D → PERSISTENZA ⇒ Effetto dopo il commit NON Perso

### ARCHITETTURA DBMS



METODI D'ACCESSO → Consistenza

ESECUZIONE CONCORRENTE → Atomicità e Isolamento

AFFIDABILITÀ → Atomicità e Persistenza

---

IL BUFFER è cruciale per le PRESTAZIONI e si colloca tra la Memoria Centrale e Quella Secondaria.

È Organizzato in PAGINE e si occupa del CARICAMENTO/SALVATAGGIO delle Pagine in Memoria Secondaria date delle RICHIESTE R/W

■ LETTURA → Se presente nel BUFFER Si Restituisce il puntatore Altrimenti Si Cerca una Pagina libera e si Carica il Blocco Per Poi Restituisce il PUNTATORE.

■ SCRITURE → Può DIFFERIRE LA SCRITTURA in un Secondo Momento Per mantenere le PRESTAZIONI al Massimo.

Ogni Pagina del BUFFER ha:

■ Bit di Controllo ⇒ Ad 1 se è Stata Modificata

■ Contatore ⇒ #Transazioni che La UTILIZZANO

---

### GESTORE AFFIDABILITÀ:

Garantisce Atomicità e Persistenza attraverso l'utilizzo del LOG, archivio persistente di TUTTE le OPERAZIONI Eseguite dal DBMS, salvato Nella MEMORIA STABILE (Mem. Resistente Ai Guasti)

Nel file di LOG Si Può Trovare come Record di Transazione:

■ B( $\tau$ ) ⇒ Begin

■ C( $\tau$ ) ⇒ Commit

- $A(T) \Rightarrow$  Abort
- $I(T, O, AS) \Rightarrow$  Insert
- $D(T, O, BS) \Rightarrow$  Delete
- $U(T, O, BS, AS) \Rightarrow$  Update

Operazioni hanno **PROPRIETÀ DI IDEMPOTENZA**, se vengono rieseguite più volte l'output è invariato

Come **RECORD DI SISTEMA** sul log troviamo:

- **Dump**  $\Rightarrow$  Copia Completa BASE DI DATI
- **CheckPoint**  $\Rightarrow$  Indica le Transazioni Attive in Quell' Istante

Regole di Scrittura sul log:

- **WAL**  $\Rightarrow$  Write Ahead Log, log scritti prima di fare le Operazioni, così posso fare **UNDO**
- **Commit Precedenza**  $\Rightarrow$  log scritti prima del Commit così posso fare **REDO**

Posso avere questo di:

- **SISTEMA**  $\Rightarrow$  Perdo mem. centrale  $\Rightarrow$  Ripresa a **caldo**
- **DISPOSITIVO**  $\Rightarrow$  Perdo mem. secondaria  $\Rightarrow$  Ripresa a **freddo**

## GESTORE METODI D'ACCESSO:

Esegue il PIANO D'ESECUZIONE che è Stato prodotto dall'ottimizzatore e Produce RICHIESTE ai BLOCCI IN MEM. SECONDARIA per il GESTORE del BUFFER che caricherà i Bloccchi Necessari in PAGINE della MEM. CENTRALE

Conosce l'ORGANIZZAZIONE delle TUPLE e sia fisica INTERNA delle PAGINE  
ORGANIZZAZIONE DELLA PAGINA:

- Block Header/Trailer  $\Rightarrow$  Per filesystem
- Page Header/Trailer  $\Rightarrow$  Per DBMS
- Dizionario di Pagina  $\Rightarrow$  Puntatori a Dati Utili } Crescono come
- Parte Utile  $\Rightarrow$  Contiene i Dati } STACK CONTRAPPASTI
- Bit di Parità  $\Rightarrow$  Info Pagina Valida o Meno

Tupla può avere DIMENSIONE fissa o Variabile

## STRUTTURA PRIMARIA

Stabilisce come sono disposte le TUPLE nel file in MEMORIA di Massa:

- SEQUENZIALI  $\Rightarrow$  Si Basa su un Criterio (come ORDINE INSERIMENTO)
- ACCESSO CALCOLATO  $\Rightarrow$  Tuple in Posizioni Specifiche in Base all' HASH
- AD ALBERO  $\Rightarrow$  Soprattutto per STRUTTURE Secondarie (Inoltre)

# STRUTTURA SEQUENZIALE: Blocchi logicamente Consecutivi

**DISORDINATA/SERIALE**  $\Rightarrow$  In Base All' ORDINE DI INSERIMENTO

Più Semplice e Più diffusa

**INSERIMENTO** Molto EFFiciente, 1 Solo Accesso Mem. Sec

**RICERCA** non EFFiciente, Lineare Nel #BLOCCHI

**ELIMINAZIONE/MODIFICHE** Portano All' Uso POCO EFFICIENTE della MEM.

**AD ARRAY**  $\Rightarrow$  Posizione dipende dall' **INDICE**, Tuple devono Avere la dimensione FISSA

**ELIMINAZIONE** Si Creano dei BUCHI

**ORDINATA**  $\Rightarrow$  In Base ad una chiave di ORDINAMENTO

**INSERIMENTO** Nel Blocco Che CONTIENE La TUPLA che PRECEDI, se c'è Spazio Si Aggiunge Altrimenti si Aggiunge Nuovo Blocco detto **OVERFLOW PAGE**

**MODIFICHE** Richiedono il MANTENIMENTO dell' ORDINE

---

**INDICI:** (File Sequentiali)

Strutture AUSILIARIE per Migliorare Prestazioni D'Accesso alle TUPLE

**PRIMARIO**  $\Rightarrow$  Chiave **ORDINAMENTO** del file **COINCIDE** con la CHIAVE di Ricerca del File

**SECONDARIO**  $\Rightarrow$  CHIAVE ORDINAMENTO e RICERCA Sono **DIVERSE**

## INDICE PRIMARIO:

Chiave Ricerca = Chiave Ordinamento del file Seq. dove ci Sono TUPPE

Ogni RECORD è una Coppia:  $\langle v_i, p_i \rangle$  → Puntatore al Primo Record  
Nel file  
Valore della Chiave di Ricerca ←

■ DENSO  $\Rightarrow$  Ogni Occorrenza della Chiave è File  $\exists$  Nell' Indice

■ SPARSO  $\Rightarrow$  Solo Alcune  $\in$  Indice (Solitamente 1 per Blocco)

## RICERCA:

1 Accesso

1 Accesso

■ DENSO  $\Rightarrow$   $\in$  Indice, faccio Scansione Seq. + Accesso tramite un puntatore

■ SPARSO  $\Rightarrow$  Potrebbe non Essere Nell' INDICE Quindi:

- ① Scansione fino al Valore  $k'$  più Grande che sia  $\leq$  di  $K$
- ② Accedo al Blocco e lo Scansione

Ho  $\geq 1$  Accesso All' Indice + 1 Accesso Al Blocco Dati

## INSERIMENTO:

■ DENSO  $\Rightarrow$  Solo Se  $K \notin$  Indice ( $\in$  un VALORE NUOVO)

■ SPARSO  $\Rightarrow$  Se Per Effetto dell' Inserimento Si Aggiunge un Blocco

## CANCELLAZIONE:

■ DENSO  $\Rightarrow$  Solo Se è l'ultima Tupla Con Chiave  $K$

■ SPARSO  $\Rightarrow$  Se il Blocco dove c'è  $K$  Viene Eliminato Oppure

dovo SOSTituIRE Con il Primo Valore K' Che ci Sarà  
Nel Blocco in Assenza di K.

## INDICE SECONDARIO: sempre DENSI

Chiave di RICERCA NON Coincide con Chiave Ordinamento dei file Seq.

Ogni RECORD è una COPPIA:  $\langle V_i, p_i \rangle$  → Puntatore al BUCKET DI PUNTATORI  
→ Valore CHIAVE RICERCA

RICERCA: Scansione, Accesso BUCNET, Accesso al file  
1 Accesso Indice + 1 Bucket + n Accessi Pagine DATI

## B+TREE

Struttura Ad ALBERO BILANCIATO, Ogni NODO è Memorizzato in una Pagina della MEM. SECONDARIA.

Legami Tra Nodi: Sono PUNTATORI A PAGINA, Tipicamente molti Nodi ma pochi Livelli

STRUTTURA: Dato un  $Fan-Out = n$

Nodo FOGLIA ⇒ fino ad  $n-1$  Valori di Chiave e n Puntatori ha un VINCOLO DI RIEMPIMENTO ⇒  $\lceil \frac{n-1}{2} \rceil \leq \# \text{chiavi} \leq (n-1)$

Nodo INTERMEDIO ⇒ Seq.  $m \leq n$  Valori Ordinati di Chiave Con VINCOLO RIEMPIMENTO ⇒  $\lceil \frac{n}{2} \rceil \leq \# \text{puntatori} \leq n$

## RICERCA:

② Cerco il Più Piccolo Valore Maggiore di K;

■ Se Esiste Seguo Quel PUNTATORE

■ Altrimenti Seguo l'ultimo Puntatore e forse Scorrendo il Nodo al Quale PUNTA lo Trovo

② Se il Nodo è una foglia allora cerco K altrimenti Ripetere il PASSO 1.

PROFONDITÀ DELL' ALBERO:  $\leq 1 + \log \lceil \frac{n}{2} \rceil \left( \frac{\# \text{Valori Chiave}}{\lceil \frac{n-1}{2} \rceil} \right)$

INSERIMENTO: ① Ricerca il nodo nel quale va Inserita K

② Se K è Presente Aggiorna i BUCKET dell' Indice SECONDARIO  
Altrimenti inserisco il PUNTATORE alla TUPLA con Valore K Nell'  
Indice PRIMARIO e Nel SECONDARIO Crea un BUCKET NUOVO

Se Viola i Vincoli di RIEMPIMENTO Allora devo fare uno SPLIT

SPLIT NODO FOGLIA:

Crea 2 nodi foglia, con  $\lceil \frac{n-1}{2} \rceil$  Nel Primo ed : Rimanenti nel Secondo.

Nel PADRE un nuovo puntatore Per le Nuova Foglia Generata e  
Raggiustare i Valori Chiave Nel nodo Padre ed Eventualmente  
Propagare Lo SPLIT

**CANCELLAZIONE:** ① Ricerca Nodo da Cancellare

② Rimuovo K dal Nodo

Su INDICE PRIMARIO Nulla e Sul SECONDARIO devo Liberare BUCKET  
e Se non Soddisfo i vincoli di RIEMPIIMENTO Allora MERGE

**MERGE NODO:** Se nel Nodo da UNIRE ci Sono  $\left\lceil \frac{n-1}{2} \right\rceil - 1$  Nodi:

① Individuo fratello Adiacente da UNIRE

② Nodo1 + Nodo2 al Max ( $n-1$ ) Valori Chiave Allora:

■ Genero UNICO NODO

■ Tolgo Un Puntatore al PADRE

■ Aggiusto i Valori di Chiave del PADRE

Altrimenti distribuisco i Valori Su 2 Nodi e Aggiusto il PADRE ma  
Se c'è Bisogno Propago il MERGE

### HASH:

Garantisce un **ACCESSO ASSOCIATIVO** ai DATI (Locazione fisica dipende dal Valore della Chiave)

fz. di HASH Mappa da **Dominio CHIAVI** a **Dominio LOCAZIONI**

Sono SEMPRE POSSIBILI **Collisioni** cioè Valori Con Chiave Ricerca ≠ che Vengono Mappati Sulla Stesso Valore di INDICE.

Buona fz. di HASH Se minimizza le **COLLISIONI**, distribuisce in Modo uniforme e Casuale i Valori della CHIAVE Nei BUCKET.

Non Si può CAMBIARE fz. di hash Ma Si Ricostuisce l'indice da CAPO

funziona Se c'è un BASSO COEFF. DI RIEMPIMENTO

COSTO ZERO 1 Accesso a Pagina  $\leq n$  Accessi a Pagina

RICERCA: Calcolo fz Hash, Accedo Al Bucket B, Accedo n TUPLE Attraverso Puntatori del BUCKET

INSERIMENTO e CANCELLAZIONE: Complessità Simile alle Ricerche

Per Gestire le COLLISIONI Si Allocano BUCKET DI OVERFLOW Collegati al BUCKET DI BASE (Peggiorando le Prestazioni)

### B+TREE vs HASH:

SELEZIONI SU UGUALIANZA: A = Cost

- HASH (senza Overflow) è COSTANTE
- B+TREE è logaritmico #Chiavi

SELEZIONI SU INTERVALLI (range): A  $\geq$  COST  $\wedge$  A  $\leq$  COST

- HASH richiede più Selezioni Per Scandire il RANGE
- B+TREE Logaritmico Accede al PRIMO Valore dell' Intervallo e Poi Scansiona

INSERIMENTI & CANCELLAZIONI

- HASH Costante + Gestione OVERFLOW
- B+TREE Logaritmico #Chiavi + costo SPLIT/MERGE

## CONCORRENZA:

Ciclo Applicativo viene Misurato in TPS, Transaction Per Second Solitamente Tra 100 e 1000 e Quindi la CONCORRENZA Mi può Generare delle ANOMALIE/PROBLEMI DI CORRETTEZZA.

GESTORE CONCORRENZA: Riceve Richieste Accesso ai DATI e decide Se Autorizzarle o Meno, Eventualmente Riordinandole (SCHEDULER)

## ANOMALIE TIPICHE:

- PERDITA AGGIORNAMENTO, Effetti Ti Andati Persi
- LETTURA INCONSENTE, Accessi Successivi Stesso dato RISULTATI ≠
- INSERIMENTO FANTASMA, Valutazioni ≠ Valori Aggregati a Causa di Inserimenti Intermedi
- AGGIORNAMENTO FANTASMA, Osservo STATO INTERMEDIO che non Soddisfa i VINCOLI D'INTEGRITÀ
- LETTURA SPORCA, Letto un DATO che Rappresenta uno STATO INTERMEDIO

## PERDITA D'AGGIORNAMENTO:

Gli Effetti di una Ti Sono Andati PERSI, Si Genera ANOMALIA se 2 Transazioni Vogliono Incrementare Stessa Risorsa Ma leggono Entrambe prima Che una faccia il Commit e Quindi Salvi la Sua Elaborazione

## LETIURA INCONSISTENTE:

faccio una Lettura, Altra Ti Modifica Risorsa, fa il Commit ed io faccio Ancora una lettura ed OTTENGO un Valore diverso

---

## LETIURA SPORCA:

Viene letto uno STATO INTERMEDIO NELL' Evoluzione di una Transazione.

T<sub>1</sub> Modifica una Risorsa e prima che T<sub>1</sub> faccia il Commit T<sub>2</sub> legge la RISORSA

---

## AGGIORNAMENTO FANTASMA:

Osservo uno STATO INTERMEDIO Che NON Soddisfa VINCOLI D'INTEGRITÀ

Leggo una RISORSA da T<sub>1</sub>, T<sub>2</sub> Viene Eseguita Completamente e fa il Commit (Cambiando i Valori della Risorsa) e le Modifiche che faccio con T<sub>1</sub> (che con il Valore PRECEDENTE Erano Corrette ed Andavano Bene) Violano i Vincoli D'INTEGRITÀ con il Valore Nuovo

---

## INSERIMENTO FANTASMA:

Valutazioni ≠ di Valori Aggregati a Causa di Inserimenti Intermedi

Esempio T<sub>i</sub> che Valuta un Valore Aggregato Relativo agli Elementi che Soddisfano un PREDICATO di Selezione (come MEDIA) se Viene Inserita una NUOVA ENTRY Tra una Valutazione e L'Altra Allora i Valori Potrebbero Essere ≠.

Questa ANOMALIA fa RIFERIMENTO ~~non~~ hai dati PRESENTI ma a Nuove TUPLE che compaiono IMPROVVISAMENTE.

---

### SCHEDULE :

Accetto Solo SCHEDULE Equivalenti a Quello SERIALE, dove ogni OPERAZIONE NON Viene Inframezzata da OPERAZIONI di Altre TRANSAZIONI.

$S_1$  e  $S_2$  Sono EQUIVALENTI Se Producono Gli Stessi Effetti Sulla Base di Dati

Supponendo che le TRANSAZIONI Abbiano Esito Noto, ci Sono 2 Modi Per Gestire la Concorrenza:

- VIEW-EQUIVALENZA
  - CONFLICT-EQUIVALENZA
- 

### VIEW EQUIVALENZA:

**LEGGE-DA**  $\Rightarrow$  Si dice che una  $r_i(x)$  legge da una  $w_j(x)$  se  $w_j(x)$  Precede  $r_i(x)$  e Non c'è Nessun'altra  $w_k(x)$  Tra le 2.

**SCRITTURE FINALI**  $\Rightarrow$  Prendo l'ultima Scrittura per Quella Risorsa, Avrò Cardinalità dell'Insieme = #Risorse  $\neq$

View-Equivalenza ( $S_1 \approx_v S_2$ ) se Possidono Stessi Insiemi Legge-Da e Scritture finali

View-Serializzabilità (VSR) Se  $\exists S . S \approx_v S . S$  lo Genero Valutando tutte le possibili PERMUTAZIONI delle Transazioni,

senza Però Alterare l'ORDINE DELLE Operazioni, Altrimenti andrei a Cambiare la Transazione.

Algoritmo per View-EQUIVALENZA è lineare. Mentre Quello per VIEW-SERIALIZZABILITÀ è di Complessità Esponenziale (PERMUTO)

Quindi:

- VSR Complessità Troppo Elevata
  - Richiede COMMIT PROIEZIONE
- } Non APPLICABILE NEI SISTEMI REALI

### CONFLICT EQUIVALENZA:

Si dice che una Coppia di Operazioni  $(a_i, a_j)$  è in CONFLITTO

se: ■  $i \neq j$  (sono T diverse)

■ Operano Sulla Stessa Risorsa

■ Almeno una delle 2 è una Scrittura

■  $a_i$  Compare Prima di  $a_j$

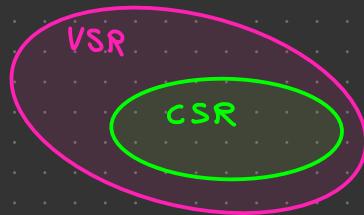
CONFLICT-EQUIVALENZA  $(S_1 \approx_c S_2)$  Se possiedono le stesse Operazioni e gli stessi Conflitti

CONFLICT-SERIALIZZABILITÀ (CSR) se  $\exists$ . uno Schedule  $S$ .  $S \approx_c S'$

Ha Complessità Lineare ed Opera Così:

■ Si Costruisce il Grafo dei Conflitti;

■ Se GRAFO è ACICCLICO allora è CSR



CSR è Sufficiente ma non Necessaria Per ∈ VSR

# Ha Anche lui Però L'IPOTESI DI COMMIT PROIEZIONE

## TECNICHE REALI: (senza IPOTESI esito Transazioni)

- TS-Buffer (Time Stamp con Scritture BUFFERIZZATE)
- 2 PL-Stretto (Locking a 2 fasi Stretto)

## LOCKING A 2 FASI:

3 Caratteristiche:

- Meccanismo di Base per Gestire i Lock
- Politica di Concessione dei Lock
- Regola che Garantisce la Serializzabilità

## PRIMITIVE DI LOCK:

- $r\text{-lock}_x(x)$ : Richiesta Lock Condiviso di  $T_x$  sulla Risorsa  $x$  per una Lettura.
- $w\text{-lock}_x(x)$ : Richiesta Lock Esclusivo di  $T_x$  su  $x$  per Scrittura.
- $unlock_x(x)$ : Richiesta di  $T_x$  di Liberare Risorsa  $x$  dal Lock

## REGOLE PER L'USO:

- R1  $\Rightarrow$  Ogni lettura deve Essere preceduta da un R-LOCK e Seguita da un UNLOCK.

Lock è Condiviso, Posso avere + R-LOCK su Stessa Risorsa da Parte di Transazioni diverse

■ R2  $\Rightarrow$  Ogni Scrittura deve Essere Preceduta da un W-LOCK e Seguita da un UNLOCK.

Questo LOCK è Esclusivo, non posso Avere più W-LOCK oppure W-LOCK e R-LOCK Sulla Stessa Risorsa

Se una TRANSAZIONE Rispetta  $R_1 \wedge R_2 \Rightarrow$  Ben formata Rispetto al locking

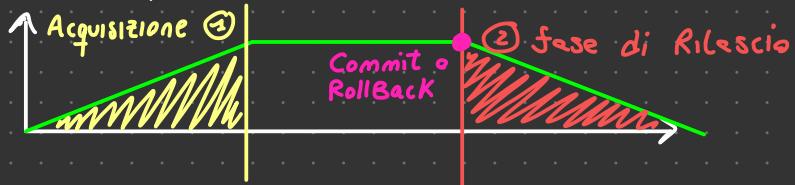
### POLITICA CONCESSIONE LOCK:

Gestore Lock/Concorrenza Salva le Seguenti Informazioni per ogni RISORSA:

■ STATO  $\Rightarrow$   $s(x) \in \{R-LOCK, W-LOCK, LIBERO\}$

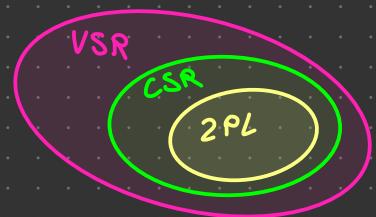
■ TRANSAZIONI In R-LOCK  $\Rightarrow$   $c(x) = \{T_1, \dots, T_N\}$

REGOLA CHE GARANTISCE SERIAZZABILITÀ: dice che una T una Volta che ha Acquisito il Blocco non può più acquisire altri.



Una T può Rilasciare il LOCK Solamente una Volta che è ha Eseguito CORRETTAMENTE Un COMMIT o un ROLLBACK

Bisogna Prestare Attenzione alla Possibilità dello STALLO CRITICO



## Blocco Critico : Si Verifica Quando:

- 2 Ti hanno Bloccato delle Stesse Risorse: T<sub>1</sub> R<sub>1</sub> e T<sub>2</sub> R<sub>2</sub>
- T<sub>1</sub> è in WAIT su R<sub>2</sub>
- T<sub>2</sub> è in WAIT su R<sub>1</sub>

Quanto Spesso Si Verifica?  $P(\text{deadlock len } 2) = \frac{1}{n^2}$

## TECNICHE PER RISOLUZIONE:

• **TIMEOUT**  $\Rightarrow$  Trascorso il TIMEOUT Viene ABORTITA Transazione

• **PREVENZIONE**  $\Rightarrow$  Una T Blocca Tutte le Risorse a cui deve accedere in Lettura o Scrittura (o blocca tutto o nulla).

Ogni Transazione Ti Acquisisce Un TimeStamp TS<sub>i</sub> All'Inizio della Sua Esecuzione. Ti può Attendere una Risorsa Bloccata da T<sub>j</sub> solo se TS<sub>i</sub> < TS<sub>j</sub> altrimenti Viene ABORTITA e fatto Ripartire con Stesso TS.

• **RILEVAMENTO**  $\Rightarrow$  Analisi periodica della Tabella di LOCK per Rilevare la PRESENZA di un Blocco CRITICO

## STARVATION TRANSAZIONE:

Anche se Nei DBMS Risulta Poco Probabile, si Risolve Con Tecniche Simili a Quelle del Blocco Critico.

Posso Analizzare la TABELLA DELLE RELAZIONI DI ATESA e Verificare da Quanto Le TRANSAZIONI Stanno Attendendo e di Conseguenza

Sospendere la CONCESSIONE DI R-LOCK Condivisi per Poder  
Permettere la Scrittura delle Transazioni in Attesa.

SQL Prevede la POSSIBILITÀ di RINUNCIARE tutto o in PARTE al Controllo della CONCORRENZA per AUMENTARE le PRESTAZIONI del Sistema.

Ciò Significa che RIESCO a Tollerare alcune Anomalie di Esecuzione Concorrente

Livello di isolamento	Perdita di update	Lettura sporca	Lettura inconsistente	Update fantasma	Inserimento fantasma
serializable	✓	✓	✓	✓	✓
repeatable read	✓	✓	✓	✓	
read committed	✓	✓			
read uncommitted	✓				

Tutti i lvl Richiedono il 2PL Stretto per le LETTURE

**SERIALIZABLE** Richiede Anche per le letture e Applica il Lock di predicato per EVITARE l'INSERIMENTO FANTASMA

**REPEATABLE READ** Applica il 2PL Stretto per Tutti i Lock in lettura Applicati a lvl di TUPLA e non Tabella. C'è Però la Possibilità di INSERIMENTO FANTASMA

**READ Committed** Richiede il lock Condivisi Per TUTTE le letture ma non 2PL Stretto.

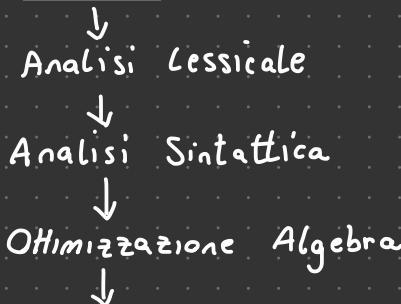
**READ UnCommitted** Non Applica Lock in lettura

## OTTIMIZZAZIONE INTERROGAZIONI:

Ogni Interrogazione Per il DBMS Viene Espressa in un Lingueggio DICHIA  
RATIVO, e Bisogna Trovare L'ESPRESSIONE PROCEDURALE Equivalente per Generare il Piano d'Esecuzione.

L'Espressione ALGEBRICA però Va OTTIMIZZATA Rispetto alle Caratteristiche del DBMS a livello fisico (METODI DI ACCESSO DISPONIBILI) e della BASE DI dati CORRENTE

Query



Optimizzazione sui Costi di Esecuzione

## OTTIMIZZAZIONE ALGEBRICA:

Si Basa fondamentalmente Sulle Regole Già Note dell' Algebra:

■ ANTICIPAZIONE DELLE PROIEZIONI

■ ANTICIPAZIONE DELLE SELEZIONI

# OTTIMIZZAZIONE DIPENDENTE METODI D'ACCESSO:

## Operazioni Tipiche di Accesso al DBMS:

- Scansione:**
- + PROIEZIONE senza Rimozione DUPLICATI
  - + SELEZIONE in Base ad un Predicato SEMPLICE
  - + INSERIMENTO/CANCELLAZIONE/Modifica

Costo Su Relazione  $R = NP(R) [\# \text{Pagine di } R]$

**ORDINAMENTO INSIEME DI TUPLE** Viene UTILIZZATO per:

- Eliminare Duplicati → DISTINCT
- Ordinare Risultato → ORDER BY
- Raggruppare Tuple → GROUP BY

## Z-WAY SORT-MERGE:

**① SORT INTERNO** Nelle pagine delle TUPLE, Salvate in un file TMP  
Su Mem. Secondaria (ANCHE DETIA RUN)

**② MERGE**, uno o più Passi di fusione le RUN Vengono UNITE fino a  
Produrre un UNICA RUN.

Devo TENERE in Considerazione NB (#Buffer) ricordando che uno  
è destinato all'OUTPUT

COSTO Passi di MERGE  $\longrightarrow [\log_2 NP]$  ed il COSTO COMPLESSIVO  
è:  $2 \times NP([\log_2 NP] + 1)$

Lettura e  
Scrittura

## ACCESSO DIRETTO (via INDICE)

Operazioni che Usano l'Indice Come:

- Selezione con **CONDIZIONE Atomica** di Uguaglianza  $A = v$  che Richiede **INDEX hash o B+Tree**
- Selezione Con **CONDIZIONE di Range**  $A \geq v$  AND  $A < c$  che richiede **B+Tree**

Ci Sono 2 Casi Possibili:

- **CONGIUNZIONE Condizioni UGUAGLIANZA**  $\Rightarrow$  Utilizzo l'**INDICE** Sulla **CONDIZIONE più Selettiva**
- **DISGIUNZIONE Condizioni UGUAGLIANZA**  $\Rightarrow$  Uso **più Indici** In Parallello facendo poi il **MERGE dei RISULTATI**

## ≠ IMPLEMENTAZIONI DEL JOIN (Operazione più Gravosa)

- NESTED LOOP JOIN
- MERGE SCAN JOIN
- HASH BASED JOIN

JOIN è Commutativo per il RISULTATO ma non per le PRESTAZIONI

---

## NESTED LOOP JOIN:

$\forall$  Tupla  $T_R \in R \{$

$\forall$  Tupla  $T_S \in S \{$

SE  $(T_R, T_S)$  soddisfa PJ Allora Aggiungi al RISULTATO

}

}

Costo è Dipendente dallo Spazio Nei Buffer ma Se Avessi  
1 Buffer Per RELAZIONE:

$$NP(R) + NR(R) \cdot NP(S)$$

Se Potessi Allocare  $NP(S)$  Buffer Per La RELAZIONE INTERNA  
il Costo Si Riduce a  $NP(R) + NP(S)$

Posso Sfruttare Gli Indici Per Gli Attributi di JOIN:

$$- B+TREE \rightarrow NP(R) + NR(R) \cdot \left( \text{Prof. Indice} + \frac{\text{Selettività Attr.}}{NR(S)} \right)$$

Se PRIMARY-KEY Allora

$$\frac{NR(R)}{VAL(A,R)} = 1$$

Valori distinti dell'Attributo A in S

BLOCK NESTED LOOP JOIN: (Variante Nested loop)

Invece che leggere Relazione Interna per Ogni Riga Lo si fa per  
Ogni Pagina di R (Esterna)

Costo dipende Sempre dallo Spazio e disposizione Nei BUFFER

e si ha (Con 1 Buffer Per Ciascuno):  $NP(R) + NP(R) \cdot NP(S)$

---

## MERGE SCAN JOIN:

Applicabile Solo Se:

- Join è un EQUI-JOIN
- Insiemi di Tuple in INPUT Sono ORDINATI Su Attributi di JOIN

Ciò Accade Se per Entrambe le Relazioni è Válida Almeno una delle Seguenti Aff:

- Relazione fisicamente ORDINATA Su Attributi JOIN (file seg. Ordinato come Struttura fisica)
- Esiste un INDICE Sugli Attributi di JOIN che mi permette la SCANSIONE ORDINATA delle TUPLE

Ordine dei #lettura è  $NP(R) + NP(S)$  se ho Indici B+Tree e indici Già nel BUFFER arrivano max. a  $NR(R) + NR(S)$  se invece ho un Solo INDICE Ma con Relazioni Ordinate al max. avrò  $NP(R) + NR(S)$  o viceversa

---

## HASH BASED JOIN:

Applicabile Solo Nel Caso di EQUI-JOIN, Molto Vantaggioso per RELAZIONI MOLTO GRANDI.

Suppongo di Avere fz. hash h() che Applicherò agli Attributi di JOIN per Capire Se fanno Parte del Risultato o meno. (MATCHING TURE)

Costo Cостruzione Hash Map  $NP(R) + NP(S)$ , l'Accesso ad una Matching Tuple (nel caso Pessimo) può Costare  $NP(R) \cdot NP(S)$  ma dipende fortemente dal #Buffer e dalla DISTRIBUZIONE degli Attributi di JOIN (uniforme Sarebbe Top)

---

## SCELTA FINALE DEL PIANO D'ESECUZIONE:

Dato un'ESPRESSIONE OTIMIZZATA in Algebra:

② Genero Tutti i Possibili Piani d'Esecuzione (Alberi) considerando le Seguenti dimensioni:

- ☰ Ordine delle Operazioni da Seguire (ASSOCIAZIVITÀ)
- ☰ Operatori Alternativi Applicabili Nei Nodi (Nested-Loop/Hash Join)
- ☰ Operatori Alternativi Per Accesso ai Dati (Seq. Scan / Indice)

Con delle formule Approssimative faccio la Stima di Costo e Selgo l'Albero con Costo Minimo.

---

## MONGODB:

**REPLICAZIONE:** Componente fondamentale dell'Architettura di MongoDB Garantisce Accessibilità ai Dati e Resilienza in Caso di Guasti.

Consiste nel Disseminare Copie Identiche degli Stessi SERVER diversi Salvaguardando la disponibilità dell'INFO in Caso di Guasti o fallimento di un Singolo Server  $\Rightarrow$  RIDONDANZA & AFFIDABILITÀ

**SHARDING:** Consiste Nell Suddividere un Dataset di Grandi Dimensioni in Parti più Piccole.

Ciascun SHARD Memorizza una piccola Porzione del DATASET Complessivo in una  $\neq$  Istanza del DB-Server.

Ogni SHARD può Essere Replicato per Garantire INTEGRITÀ ED DISPONIBILITÀ

---

## REPLICAZIONE:

REPLICA-SET è una Collezione di PROCESSI MongoDB Daemon che mantengono una Copia dello Stesso DATASET.

A cosa Servono?

- Migliora Prestazioni delle Operazioni di SCRITTURA
- Ridondanza e Alta Disponibilità
- Tolleranza ai Guasti Rispetto ai FAILURE di un Server
- Possibilità di Aggiornare senza Perdita del SERVIZIO

**NODO PRIMARIO:** Gestisce tutte le SCRITTURE e Mantiene un OPLG

Ciascun REPLICASET può Avere un SOLO NODO PRIMARIO

**NODI SECONDARI:** Replicano le Operazioni e OPLG del Nodo Primario Nelle loro Copie del DATASET

Se NODO PRIMARIO dovesse essere non disponibile Allora uno Tra i NODI SECONDARI Verrebbe Eletto Come Primario

## ALGORITMO DI ELEZIONE:

Elezioni dentro ad un REPlica SET avviene Su Protocollo Basato Sull' Algoritmo RAFT.

→ Viene Attivato se:

- Inizializzo un REPlica-SET
- Aggiungo/Rimuovo Nodo da REPlica-SET
- Scadere TIMEOUT Nella comunicazione

IL NODO con TIMESTAMP di Scrittura più Recente è Quello con più PROBABILITÀ DI ESSERE ELETTO (voglio Quello più Aggiornato)

Dopo Aver Concluso L'ELEZIONE ; Nodi Entrano in Uno STATO di CONGELAMENTO durante il Quale NON possono più Votare.

→ Evita Elezioni Ripetute Rapidamente

---

## OPLOG:

Le op. di SCRITTURA prima Vengono fatte e Solo poi Vengono Scritte sull' OPLOG  $\Rightarrow$  Nodi Secondari Replicano le Stesse Op. in Maniera Asincrona.

Ogni MEMBRO della Replica ha il Suo OPLOG per Poter Verificare la RELAZIONE Tra Stato del Nodo e Stato Complessivo del DB.

---

## ARBITRI:

Utile Quando ha Solo 1 PRIMARIO e 1 SECONDARIO, non può diventare Primario e non ha una COPIA del DATASET.

## NODI NASCOSTI:

Non possono diventare Nodi PRIMARI e **non** Sono Accessibili da Nodi Client perché Sono usati per TASK SPECIFICI come BACK-UP e REPORTING.

Possono Essere **RITARDATI** = fanno Operazioni e OPLAG dopo un Ritorio **PRE-STABILITO**

---

## WRITE CONCERN:

Per Garantire che Le Scritture Siano Segnalate Come Tali Solo dopo che un CERTO #Nodi Abbìa Effettuato La Scrittura

Posso Impostarlo a:

- 0
  - 1
  - Majority
  - Numero Preciso
- } Valore Basso di w Riduce la LATENZA ma diminuisce il livello di SICUREZZA mentre Se Alta garantisce PERSISTENZA del DATO

## LETTURE: / READ CONCERN:

Possono Essere distribuite su più Nodi Per Bilanciare il CARICO

- LOCAL → Dato più Recente
- AVAILABLE → Minimizza Latenza, Quello disponibile
- MAJORITY → Ritorna dati CONFERMATI dalla Maggioranza
- LINEARIZZABLE → Confermato da TUTTI
- SNAPSHOT → Ritorna dati CONFERMATI dalla Maggioranza in un Certo Istante Temporale

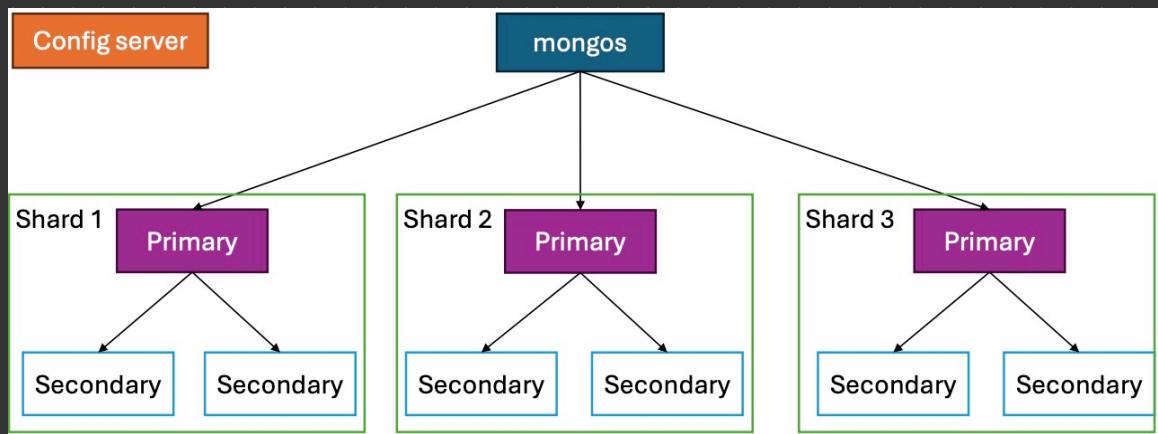
## SHARDING:

Dividere un DB in Componenti più PICCOLE in Varie Istanze del SRV.

Shard: Un REPOCA-SET Che Memorizza una Porzione di dati di un CLUSTER

Mongos: Sunge da DISTRIBUTORE delle Query, Reindirizzandole agli SHARDS

Config-Srv: Mantiene METADATI Relativi allo Sharding



MICRO-SHARDING Se ho più SHARD Sullo Stesso HOST per:

Esecuzione Parallelia

Aumentare Capacità di STORAGE COMPLESSIVA

Sfruttare LOCALITÀ DEL DATO

Distribuzione del DATO Avviene Attraverso La SHARD KEY, I DOCUMENTI Vengono Suddivisi in CHUNK con 2 Strategie diverse

## RANGED SHARDING:

Intervalli Continui di Valori delle chiave di SHARD:

- ottimo Per RANGE QUERY
- Metodo di Default
- Sarebbe ottimale distribuzione UNIFORME dei DATI

## HASHED SHARDING:

Opera Calcolando la fz. HASH della Chiave di SHARD, NON È EFFICIENTE Per le RANGE-QUERY ed I Valori Molto Simili Avranno però un Valore di HASH Molto diverso  $\Rightarrow \neq$  CHUNK

---

## BALANCER:

Processo in Background Per BILANCIARE #Dati sugli SHARD.

---

## PROPRIETÀ ACID SU MONGODB:

ATOMICITÀ Op. su un Singolo Documento Sono Sempre ATOMICHE

## CONSISTENZA

■ Eventual  $\Rightarrow$  Giusto Cul di Prestazioni, dopo la Propagazione e La Stabilizzazione Verrà Restituito Ultimo DATO.

■ Strong  $\Rightarrow$  Cul Marx.  $\rightarrow$  Problemi Prestazioni ma Garantisce che Ogni lettura leggerà Sempre l'ultimo Valore Confermato

**PERSISTENZA** Tramite il Meccanismo WRITE AHEAD Log ogni 100ms e le op. vengono prima scritte nel file e poi eseguite in modo da poter permettere il RIPRISTINO il caso di ERRORI.

**ISOLAMENTO** si stabilisce impostando i parametri di WRITE CONCERN e READ CONCERN.