Documentazione Associata

- **Dizionario dei Dati**:

  ENTITÀ: Entità | Descrizione | Attributi | Identificatore

  RELATIONSHIP: Associazione | Descrizione | Componenti | Attributi

  - Vincolo Integrità: <concetto> deve / non deve <...>
  - Regola di Derivazione: <concetto> si ottiene <...>

- **Glossario dei Termini**:

  Termine | Descrizione | Sinonimi | Collegamenti

Progettazione Logica

- **Tavola dei Volumi**: Concetto | Tipo (E\R) | Volume
- **Tavola degli Accessi**: Concetto | Costrutto (E\R) | Accessi | Tipo (L\S)

  è per operazione!

- **Eliminazione Generalizzazione**:

  1) Accorpamento entità figlie nel genitore, con attributi che diventano opzionali + un attributo "TIPO";

     QUANDO: accessi a genitore e figlio sono contestuali.

  2) Accorpamento del genitore nei figli, sostituendo opportunamente relationship cui partecipava il padre e distribuendo gli attributi;

     QUANDO: SOLO SE GENERALIZZAZIONE TOTALE! Conviene se accessi ai figli decontestualizzati dal padre.

  3) Sostituzione generalizzazione con relationships, verso cui i figli sono entità DEBOLI.

     QUANDO: Se gli accessi ai figli sono separati degli accessi al padre.

- **Eliminazione Attributo Multivalore**

Se ha cardinalità (x,m), lo si RELIFICA e lo si collega con una relationship, valutando se è debole o meno.

- **FOREIGN KEY:**

FK di $R_1(X_1)$ che fa riferimento a $R_2(X_2)$ è un sottoinsieme di attributi $FK \subseteq X_1$ tale che: $|FK| = |K|$, dove $K :=$ chiave primaria di $R_2(X_2)$

e il valore di $t_1 \in r_1(R_1)$ deve essere presente in una tupla $t_2 \in r_2(R_2)$ e $t_1[FK] = t_2[K]$ oppure $t_1[FK] = NULL$.

TRADUZIONE NEL MODELLO RELAZIONALE

- **ATTRIBUTO COMPOSTO:** si mettono nella relazione cui appartiene i singoli attributi che lo compongono.
- **ATTRIBUTO MULTIVALORE:** si reifica e si aggiunge anche la K primaria dell'entità di partenza, con vincolo di integrità ref.
  Nuova K = chiave attributo + chiave entità.
- **ASSOCIAZIONE m-m:** 3 relazioni: 2 per entità e 1 per associazione
  K associazione = K entità 1 + K entità 2
  + 2 vincoli integrità ref.
- **ASSOCIAZIONE OTTIMA (1,1)-(y,m):** entità che partecipa con (1,1) ha anche K dell'altra entità e attributi associazione + vincoli integrità ref.
  K = K entità (1,1)
- **ASSOCIAZIONE (0,1)-(y,m):**

  BISOGNA SCEGLIERE (potrebbero esserci troppi valori NULL):
  - 2 entità + 1 per associazione con 2 vincoli di integrità, una K = K di entità (0,1)!
  - solo 2 entità, come la OTTIMA (1,1)-(y,m).
- **ASSOCIAZIONE (1,1)-(1,1):** Scegliere in quale delle 2 accorpare l'associazione, in base a volume dei dati e operazioni.
- **ENTITÀ DEBOLE:** tradurre entità debole + K entità forte + attributi associazione; aggiungere vincoli di integrità ref.

  K = K entità forte + K entità debole.

ALGEBRA RELAZIONALE

È un DML procedurale (COME) "task centered"

SELEZIONE: $\sigma_F(r) = \{ t \mid t \in r \text{ e } F \text{ è vera su } t \}$

PROIEZIONE: $r(X)$, $Y \subseteq X \Rightarrow \pi_Y(r) = \{ t[Y] \mid t \in r \}$ → NON SI MANTENGONO I DUPLICATI !

$$\left[* \;\; |\pi_Y(r)| \leq r(X) \right] ; \left[\text{se } Y \text{ è SK di } r \Rightarrow |\pi_Y(r)| = r(X) \right]$$

RIDENOMINAZIONE: $\rho_{B_1,\dots,B_k \leftarrow A_1,\dots,A_k}(r)$, dove $B_1,\dots,B_k$ = nuovi nomi; $A_1,\dots,A_k$ = vecchi nomi

JOIN NATURALE: $r_1 \bowtie r_2 = \{ t \text{ su } X_1X_2 \mid \exists t_1 \in r_1(X_1), \exists t_2 \in r_2(X_2) : t_1[X_1] = t_1 \text{ e } t[X_2] = t_2 \}$

- **CARDINALITÀ DEL JOIN:**
  - In generale: $\left[0 \leq |r_1 \bowtie r_2| \leq |r_1| \cdot |r_2| \right]$
  - Se coinvolge K di r_2: $\left[0 \leq |r_1 \bowtie r_2| \leq |r_1| \right]$

    Poiché ogni tupla di r_1 può essere uguale al massimo ad 1 tupla di r_2, in quanto è coinvolta una chiave (UNIVOCA) di r_2.
  - Se coinvolge K_1 di r_1 e K_2 di r_2: $\left[0 \leq |r_1 \bowtie r_2| \leq \min\{|r_1|, |r_2|\} \right]$
  - Se coinvolge K di r_2 con vincolo di integrità referenziale da r_1 a r_2: $\left[0 \leq |r_1 \bowtie r_2| = |r_1| \right]$
- **π e $\bowtie$:** $r_1(X_1)$ e $r_2(X_2)$
  - Join e proiezione: $\pi_{X_1}(r_1 \bowtie r_2) \subseteq r_1 \longrightarrow \leq |r_1|$
  - Proiezioni e join: $\pi_{X_1}(r) \bowtie \pi_{X_2}(r) \supseteq r \longrightarrow \geq |r|$

- **Query con QUANTIFICATORE UNIVERSALE:**

DOPPIA NEGAZIONE: 1) Si nega la condizione da rispettare

2) Si trovano tutte le tuple che NON la rispettano, poi si sottraggono quest'ultime da tutte le altre tuple!

Esempio: IMPIEGATI (Matricola, Nome, Età, Stipendio)
SUPERVISIONE (Impiegato, Capo)

- Trovare le matricole dei capi i cui impiegati guadagnano TUTTI più di 40.

1) Troviamo gli impiegati che guadagno MENO di 40 (1° negazione);
2) Troviamo i capi di tali impiegati (2° negazione) e la soluzione è data da tutti i capi MENO quelli trovati.

$$\Pi_{Capo}(SUPERVISIONE) - \Pi_{Capo}(SUPERVISIONE \bowtie_{Impiegato = Matricola} (\sigma_{Stipendio \leq 40} IMPIEGATI))$$

- **ESPRESSIONI EQUIVALENTI:**

ASSOLUTA: $\Pi_{AB}(\sigma_{A>0}(R)) \equiv \sigma_{A>0}(\Pi_{AB}(R)), \forall R$

DIPENDENTI DALLO SCHEMA:

- $\Pi_{AB}(R_1) \bowtie \Pi_{BC}(R_2) \equiv_R \Pi_{ABC}(R_1 \bowtie R_2)$

Se e solo se $R_1(X_1), R_2(X_2)$ e $X_1 \cap X_2 = A$!

- $\sigma_F(E_1 \bowtie E_2) \equiv_R E_1 \bowtie (\sigma_F(E_2))$

Se e solo se F coinvolge SOLO attributi di E_2

→ PUSH SELECTION DOWN (ottimizzatore)

- **VISTE:** Sono RELAZIONI DERIVATE (da altre relazioni, tramite query). Sono VIRTUALI e non vengono salvate. Comportano i seguenti vantaggi:
  - Ogni utente vede solo ciò che gli interesse nel modo adatto;
  - Vede solo ciò che è autorizzato a vedere → PRIVILEGI (PoLP)
  - Utili per semplificare la scrittura di query.

NORMALIZZAZIONE

Una FORMA NORMALE è una proprietà di una base di dati relazionale che ne garantisce l'assenza di anomalie; di solito le anomalie sono dovute a ridondanze e a schemi semanticamente non omogenei; possono essere:

ANOMALIE: DI AGGIORNAMENTO, DI INSERIMENTO, DI CANCELLAZIONE.

- **Dipendenza funzionale (FD):**

Sia r istanza di relazione su $R(X)$, $Y \subset X$, $Z \subset X$.

$\exists$ FD $Y \longrightarrow Z$ se $\forall (t_1, t_2)$ di r:

$t_1[Y] = t_2[Y] \Rightarrow t_1[Z] = t_2[Z]$, cioè i valori su Y determinano univocamente i valori su Z.

$Y \longrightarrow Z$ è NON BANALE se $Z \not\subseteq Y$, $Y \cap Z = \emptyset$, Y non è SK.

- **BCNF:**

Una relazione r è in BCNF se, per ogni dipendenze funzionale non banale $X \longrightarrow Y$ definite su di esse, X contiene una (super) chiave di r.

- **Normalizzare:**

Per ogni FD $X \longrightarrow Y$ che viola la BCNF, definire una nuova relazione su XY ed eliminare Y dalla relazione originarie.
(Non sempre basta; vale per i casi semplici).

- **Decomposizione senza perdita:**

Sia R decomposta su X_1 e X_2 in $R_1(X_1)$ ed $R_2(X_2)$. E' decomposta SENZA PERDITA se: $\pi_{X_1}(r) \bowtie \pi_{X_2}(r) = r$.

Ciò si ha $\Longleftrightarrow$ $X_0 = X_1 \cap X_2$ contiene almeno una chiave k per $R_1(X_1)$ o $R_2(X_2)$

- **Conservazione delle dipendenze:**

$\Longleftrightarrow$ $\forall$ FD $X \longrightarrow Y$ in r, nelle relazioni decomposte $R_1, \dots, R_m$, gli attributi XY compaiono tutti in una relazione tra $R_1, \dots, R_m$.

SQL

Structured Query Language. È sia DDL che DML; inoltre è di tipo DICHIARATIVO (CHE COSA e non COME!)

- **Semantica query SQL:**

```
SELECT ListeAttributi  → π
FROM ListeTabelle      → ⋈
WHERE Condizione;      → σ
```

1) Prodotto cartesiano delle tabelle in ListeTabelle;
2) Selezione delle tuple che soddisfano la condizione nel where;
3) Proiezione sugli attributi in ListeAttributi.

* Il join non elimina le tuple duplicate come in algebra relazionale! (Utilizzare SELECT DISTINCT, se necessario)

- **UNION:** ATTENZIONE, elimina i duplicati! Usare UNION ALL nel caso; inoltre, attenzione alla NOTAZIONE POSIZIONALE!

* Operatori insiemistici non usabili in subquery nidificate.

in ≅ = any	;	not in ≅ <> all

- **PASSAGGIO DI BINDING:**

- Trovare i padri i cui figli guadagnano TUTTI più di 20.

```
SELECT DISTINCT Padre
FROM Paternità Z
WHERE NOT EXISTS ( SELECT *
                   FROM Paternità W, Persone
                   WHERE W.Padre = Z.Padre
                     AND W.Figlio = Nome
                     AND Reddito <= 20 );
```

* Sempre DOPPIA NEGAZIONE (Reddito ≤ 20 e NOT EXISTS)!

1) Scegli tutti i diversi padri (Z);
2) Per ognuno, seleziona i figli che guadagnano 20 o meno;
3) Togli quei padri per cui esiste un tale figlio.

- **TRANSAZIONI:**

Atomicità
Consistenza
Isolamento
Durabilità (Persistenza)

ATOMICITÀ → sequenze di operazioni indivisibili "all or nothing"

CONSISTENZA → vincoli di integrità soddisfatti al termine delle TX.

ISOLAMENTO → correttezza nella gestione della concorrenza; equivalenza con esecuzione sequenziale

DURABILITÀ → Commit = impegno, nel mantenere traccia dei risultati in modo permanente, anche in presenza di guasti.

ORGANIZZAZIONE FISICA

- **GESTORE DEL BUFFER:**
  - È di fondamentale importanza per limitare gli accessi diretti alla memoria secondaria, ottimizzando le letture e scritture sul buffer, sfruttando ad esempio la contiguità.
  - È un area di memoria in MEMORIA PRINCIPALE preallocata e gestita dal DBMS, è CONDIVISA tra tutte le transazioni.
  - Gestisce: 1) Il buffer;
    2) Per ogni pagina un descrittore che contiene:
       - il file fisico e il numero del blocco
       - due variabili di stato: il dirtybit e il contatore delle TX che usano la pagina attualmente.
  - Riceve richieste di letture/scritture, fornendo le seguenti primitive da invocare e che lui esegue:
    - FIX: richiede una lettura; se presente nel buffer di, altrimenti se c'è posto la si carica; se non c'è posto: 1) "STEAL" → VITTIMA + sostituzione
      2) "NO STEAL" → attese
    - UNFIX: diminuisce contatore pagine di 1
    - FORCE: scrittura sincrona
    - FLUSH: scritture asincrona
    - SETDIRTY: setta il dirtyBit a 1.

- **FATTORE DI BLOCCO**: è il numero di record contenibili in 1 Blocco.

Se: L_R = lunghezza record
L_B = lunghezza blocco
$\Rightarrow$ $F_B = \lfloor L_B / L_R \rfloor$ (anche detto bfr)

Lo spazio residuo può essere utilizzato (record "spanned") o meno ("unspanned").

- **TAVOLA HASH**:

Supponiamo di dover inserire 40 record e di avere una tavola hash con 50 posizioni; la funzione hash è: $h(K) = K \bmod 50$

Abbiamo il seguente numero di collisioni:

- 1 collisione a 4 (4 record con stesso valore di $h(K)$)
- 2 collisioni a 3
- 5 collisioni a 2.

La collisione a 2 costa: 1 per accesso tabella hash + 1 tavola overflow = 2

→ avendone 5 $\Rightarrow$ $5 \cdot 2 = 10$ accessi

Quelle a 3 costano: $2 + 3 = 5$ → avendone 2 $\Rightarrow$ $2 \cdot 5 = 10$ accessi

Quelle a 4 costa: $2+3+4 = 9$ → avendone solo 1 $\Rightarrow$ $1 \cdot 9 = 9$ accessi

I record che richiedono accesso più costoso sono: $5 \cdot 1 + 2 \cdot 2 + 1 \cdot 3 = 12$

$\Rightarrow$ I restanti $40 - 12 = 28$ record hanno costo di accesso pari a 1.

$\Rightarrow$ # accessi totali = $28 + 10 + 10 + 9 = (57)$

$\Rightarrow$ NUMERO MEDIO DI ACCESSI = $\frac{57}{40} = \boxed{1,425}$

- **FILE HASH**:

Sia T = # record $(= 40)$; F = fattore di blocco $= 10$;

f = fattore di riempimento = $\frac{\text{\# record}}{\text{\# dim. spazio utilizzato}}$ $\Rightarrow$ $f = \frac{40}{50} = 0,8$

$\Rightarrow$ # BLOCCHI NECESSARI = $\boxed{B = \left\lceil \frac{T}{f \cdot F} \right\rceil} = \left\lceil \frac{40}{0,8 \cdot 10} \right\rceil = 5$

* Ora l'indirizzo identifica il blocco, non il record!

⇒ Funzione Hash: $h(M) = M \bmod B = \boxed{M \bmod 5}$

⇒ Abbiamo solo 2 collisioni e 2 ⇒ { 38 record → 1 accesso ; 2 record → 2 accessi }

⇒ # accessi = $38 \cdot 1 + 2 \cdot 2 = 42$ ⇒ NUMERO MEDIO DI ACCESSI = $\frac{42}{40} = \boxed{1,05}$

VANTAGGI

- organizzazione molto efficiente per l'accesso diretto puntuale (uguaglianze di valori)

SVANTAGGI

- gestione overflow per le collisioni
- non efficiente con ricerche su intervalli!
- funzionano bene solo con file la cui dimensione non varia molto nel tempo.

GLI INDICI

È un file organizzato su alcuni campi/attributi di una tabella.
Ogni record di un indice è formato da una coppia (campo indice, indirizzo del record);
il campo indice contiene il valore dell'attributo su cui è organizzato l'indice e l'indirizzo del record è l'indirizzo fisico del record che ha quel campo indice.

- **INDICE PRIMARIO:**
  Quando contiene al suo interno i dati, oppure è realizzato su un file ordinato sul campo su cui è definito l'indice; cioè è definito su un CAMPO CHIAVE per la tabella.

- **INDICE SECONDARIO:**
  Definito su un campo non chiave della tabella
  - INDICE DENSO: contiene TUTTI i valori della chiave → 1 per record se indice primario
  - INDICE SPARSO: contiene 1 RECORD per ogni BLOCCO.
    possibile solo se indici primari, perché gli altri record sono adiacenti nello stesso blocco, in quanto l'ordinamento è sul campo chiave.

* • Un indice primario di solito è SPARSO; se è denso permette di eseguire operazioni sugli indirizzi;
  • Un indice SECONDARIO PUÒ ESSERE SOLO DENSO!
  Poiché i record non sono adiacenti in base al campo dell'indice.

- **DIMENSIONE DELL'INDICE:**

r = # di record nel file

B = dim. blocco

R = dim. record

K = lunghezza del campo chiave

P = lunghezza degli indirizzi ai blocchi

- $\left\lfloor \frac{B}{R} \right\rfloor$ = fattore di blocco (del file)
- $\frac{B}{K+P}$ = fattore di blocco dell'indice
- $N_F = \frac{r}{B/R}$ → numero di blocchi per il file (circa)
- $N_D = \frac{r}{B/(K+P)}$ → numero di blocchi per un INDICE DENSO (indirizza per record)
- $N_S = \frac{N_F}{B/(K+P)}$ → numero di blocchi per un INDICE SPARSO (indirizza per blocco)

* Se il file ha N_F blocchi da indicizzare e in un blocco riesco a mettere $B/(K+P)$ indici, ho bisogno di $\frac{N_F}{B/(K+P)}$ blocchi!

* Si possono avere anche indici MULTILIVELLO, ma se l'indice è SECONDARIO, l'ultimo livello deve essere per forza DENSO!

- **ALBERO DI RICERCA DI ORDINE F+1:**

Ogni NODO ha fino a F+1 FIGLI e fino a F ETICHETTE, chiavi ordinate.

P_0	K_1	P_1	---	K_i	P_i	---	K_F	P_F

VINCOLI:

1) Per ogni nodo: $K_1 < K_2 < \dots < K_F$ → chiavi ordinate

2) Per tutti i valori K presenti nel sottoalbero puntato da P_i:

$K_i \leq K < K_{i+1}$ per $0 < i < F$

$K < K_i$ per $i = 0$

$K_F < K$ per $i = F$

- **SPLIT e MERGE:**

Se albero di ordine $F+1 = 2t$ $\Rightarrow$ $2t$ puntatori e $2t-1$ nodi figli

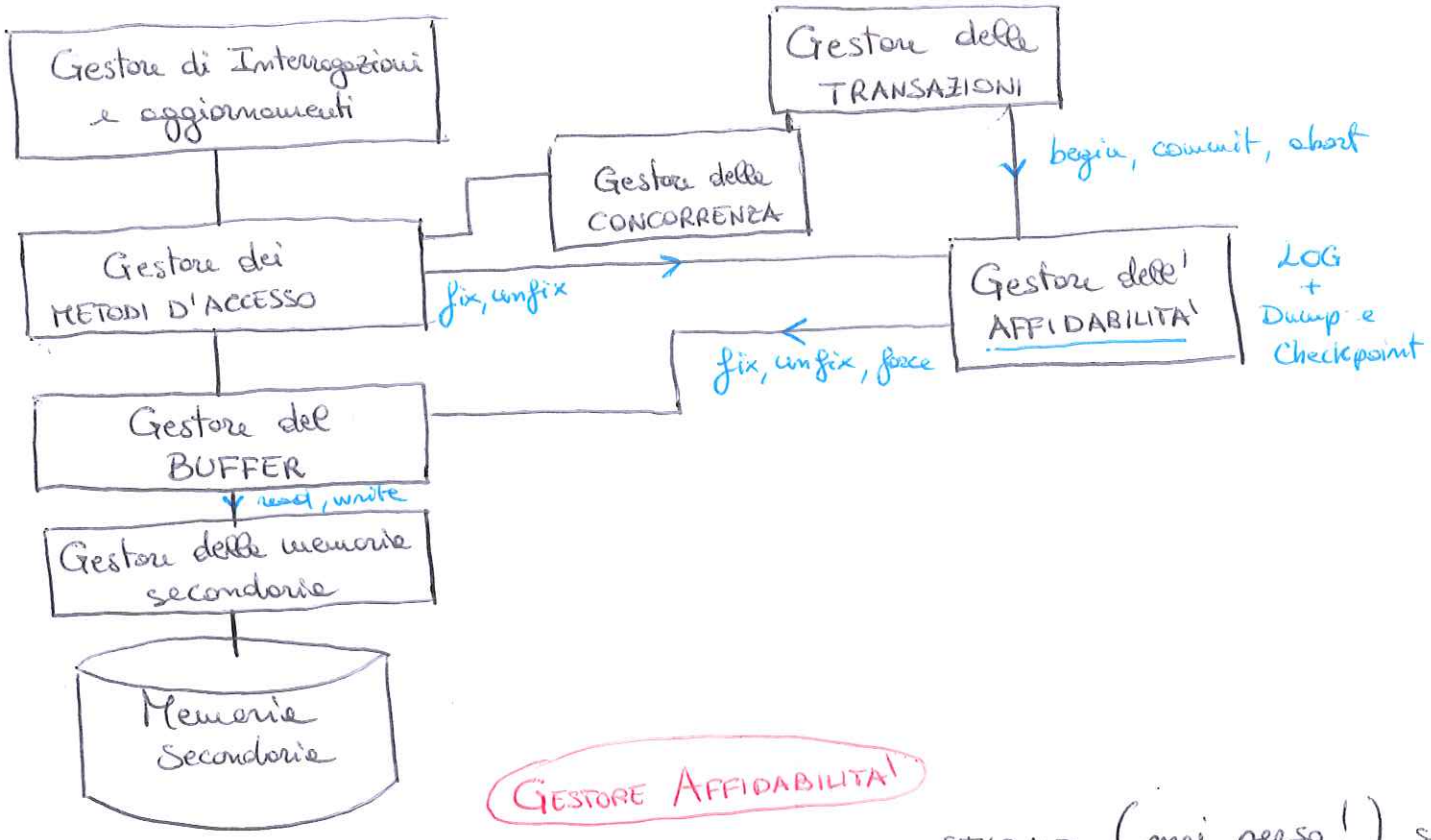
OCCUPAZIONE MINIMA = $\boxed{t-1}$

- SPLIT se il nodo è saturo → viene suddiviso e si SALE AL GENITORE con un puntatore in più!
- MERGE se il livello di occupazione di un nodo scende sotto l'occupazione minima, si fa scendere una chiave da un nodo GENITORE e si fa il merge con un nodo fratello se necessario.

- **B+ TREE:**
  - Le chiavi compaiono TUTTE nelle foglie (ripetizioni)
  - Le foglie sono collegate tra loro in una lista!
  - Molto usati dai DBMS ed efficienti per ricerche ad intervallo
  - Se primario, le foglie possono contenere tuple; se secondario, le foglie contengono solo puntatori a tuple.

- **B TREE:**
  - Le chiavi dei Ev. intermedi non sono ripetute nelle foglie
  - Se primario, i nodi intermedi contengono tuple; se secondario, puntatori alle tuple.
  - Nodi intermedi possono puntare direttamente ai dati.

COMPONENTI DBMS



GESTORE AFFIDABILITA'

LOG: FILE SEQUENZIALE scritto in MEMORIA STABILE (mai perso!), su cui vengono scritti:

- RECORD DI AZIONI:
  - begin B(T)
  - Commit C(T)
  - abort A(T)
  - insert I(T, O, AS)
  - delete D(T, O, BS)
  - update U(T, O, BS, AS)
- RECORD DI SISTEMA:
  - record di DUMP (stato BD)
  - record di CHECKPOINT (tx. attive)

REGOLE LOG:

(1) WRITE-AHEAD-LOG: scrivere sul log perlomeno la parte BS del record, PRIMA di effettuare la modifica sulla BD
→ permette UNDO

(2) COMMIT-PRECEDENZA: scrivere sul log perlomeno la parte AS del record, PRIMA di fare commit
→ permette REDO

Scrittura nel Log

- **Modalità Immediata:**

Le pagine della BD vengono scritte PRIMA DEL COMMIT, → NO REDO
ma ovviamente dopo le scritture dei record nel Log.

* UNDO solo di TX ancora in corso al momento del crash!

- **Modalità Differita:**

Le pagine vengono scritte nella BD DOPO DEL COMMIT, → NO UNDO
e sempre dopo le scritture di record sul Log.

* REDO solo di TX andate in COMMIT PRIMA del CRASH e DOPO l'ULTIMO CHECKPOINT.

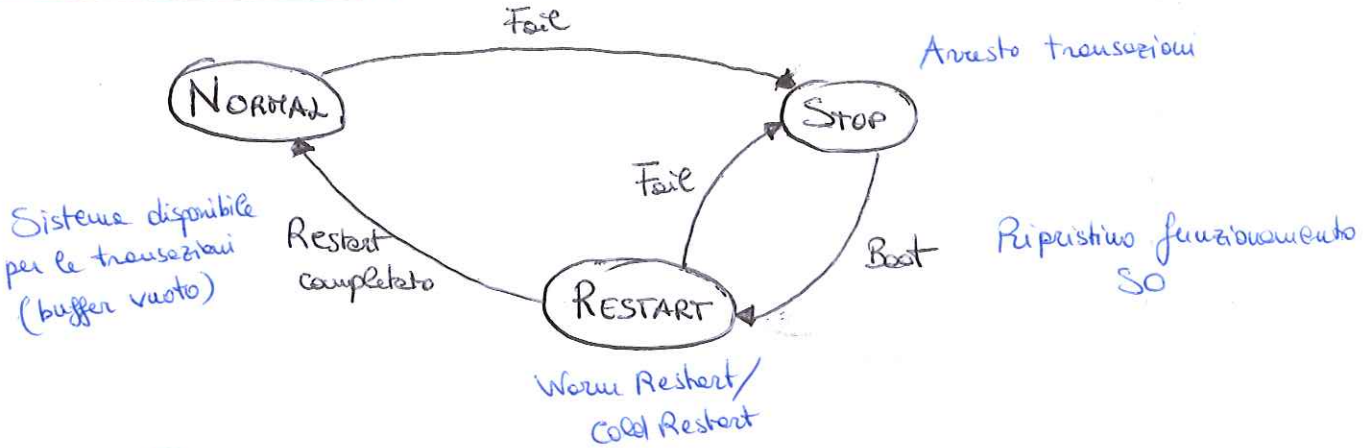
- **Modalità Mista:**

Consente al gestore del buffer di ottimizzare le FLUSH indipendentemente dal controllare dell'affidabilità → UNDO e REDO

* UNDO di quelle in corso al momento del crash
REDO di quelle andate in COMMIT tra ultimo CK e crash!

* Record di COMMIT sempre scritto in modo SINCRONO (Force)!

- **Modello "Fail-Stop":**



- **Ripresa a Caldo:**

1) Trovare ultimo CHECKPOINT → Log a ritroso;
2) Costruire insieme UNDO e REDO → Log in avanti;
3) Log all'indietro, fino alla più vecchia azione delle ~~azioni~~ TX in UNDO e REDO, disfacendo le UNDO;
4) Rifare tutte le REDO → Log in avanti.

- **Ripresa a Freddo:**

1) Accesso al DUMP e ripristino BD e del Log (solo commit, no abort)
2) Ripresa a caldo.

CONTROLLO DELLA CONCORRENZA

IL GESTORE DELLA CONCORRENZA comunica ovviamente con il Gestore delle Transazioni e anche con il gestore dei metodi di accesso.

ANOMALIE

- PERDITA DI AGGIORNAMENTO: di tipo (w-w) sullo stesso dato, e si perde un aggiornamento → ~~REPEATABLE~~ READ UNCOMMITTED
- LETTURA SPORCA: si leggono dati aggiornati da TX che andranno in ABORT; è di tipo (r-w) → READ COMMITTED
- LETTURE INCONSISTENTI: Più letture di uno stesso dato nelle stesse TX, ma tra l'una e l'altra il dato viene modificato da un'altra TX. Molteplici (r-w) → REPEATABLE READ
- AGGIORNAMENTO FANTASMA: Due oggetti con vincolo di tupla e letture in istanti differenti; nel frattempo dati modificati e vincolo non rispettato al momento della lettura nella singola TX. (r-w). → REPEATABLE READ (Lock sui dati letti)
- INSERIMENTO FANTASMA: Di tipo (r-w) su select ripetute o correlate di tipo count, avg o con intervalli nel where. Se tra l'una e l'altra avviene un inserimento di un dato che soddisfa la condizione, le letture sono inconsistenti. → SERIALIZZABILE

- SCHEDULER: Deve garantire l'ISOLAMENTO, accoglie le transazioni e assegna loro un ID univoco, chiede al gestore del buffer di eseguire operazioni di lettura/scrittura in un DATO ORDINE → (SCHEDULE)

- VIEW-SERIALIZZABILITÀ:
  - $r_i(x)$ LEGGE-DA $w_j(x)$ in uno schedule S se e solo se
    1) $w_j(x)$ precede $r_i(x)$;
    2) $\nexists\, w_k(x)$ tra $w_j(x)$ e $r_i(x)$.
  - $w_i(x)$ è una SCRITTURA FINALE se è l'ultima scrittura dell'oggetto x.

Due schedule sono View-Equivalenti ($\approx_v$) se hanno gli stessi insiemi LD e SF!

⇒ S è VIEW-SERIALIZZABILE ⟺ $S \approx_v R$, con R schedule seriale.

- **CONFLICT - SERIALIZZABILITÀ:**

Un'azione a_h è IN CONFLITTO con a_e se:
- $h \neq e$
- operano sullo stesso oggetto
- almeno 1 delle due è una WRITE.

Abbiamo 2 casi:
- conflitto read-write: $r(A)\ w(A)$ oppure $w(A)\ r(A)$
- conflitto write-write: $w_1(A)\ w_2(A)$

Schedules CONFLICT-EQUIVALENTI se ogni coppia di operazioni in conflitto compare nello stesso ordine.

TEOREMA Costruendo il GRAFO DEI CONFLITTI nel seguente modo:
- un nodo per ogni transazione t_i;
- un arco orientato da t_i a t_j se $\exists$ un conflitto tra a_i e a_j e nello schedule a_i precede a_j.

Schedule CONFLICT-SERIALIZZABILE $\iff$ Grafo conflitti è ACICLICO!

CONTROLLO BASATO SU LOCK

Lo scheduler riceve richieste delle transazioni di tipo r_lock, w_lock e unlock e le gestisce tramite strutture dati, come le Tabelle dei Lock.

VINCOLI:
1) Tutte le letture sono precedute da r_lock (lock condiviso) e seguite da unlock;
2) Tutte le scritture sono precedute da w_lock (lock esclusivo) e seguite da unlock.

- Se una transazione fa richieste di un lock su un oggetto al quale al momento non può accedere perché già lockato da un'altra TX, allora viene messa IN ATTESA.

- **TABELLA DEI LOCK:**

DATA ITEM	LISTA LOCKS	LISTA D'ATTESA
A	$[t_1, r]$, $[t_2, r]$	$[t_3, w]$
B	$[t_4, w]$	$[t_5, w]$, $[t_6, r]$
...	---	---

REGOLE TRANSAZIONI

Lo scheduler organizza lo schedule delle transazioni, inserendo anche le primitive di lock, in modo da garantire la serializzabilità.

1) TRANSAZIONE BEN FORMATA: ogni azione $p(A)$ deve essere contenuta in una SEZIONE CRITICA del tipo: $l_i(A) \dots w_i(A) \dots u_i(A)$

2) SCHEDULE LEGALE: ogni coppia lock-unlock di uno schedule (ogni sez. critica) deve essere esclusiva: $S = l_i(A) \dots l_j(A) \dots u_i(A)$ ✗ non permesso

3) 2PL: In ogni transazione (TUTTE) le richieste di lock precedono (TUTTI) gli unlock.
→ Previene PHANTOM UPDATE, ma non risolve DIRTY READ e PHANTOM INSERT!
Inoltre è sotto ipotesi di commit-proiezione:
- rollback a cascata → Leggere solo da TX COMMITTED
- Letture sporche → NO COMMIT fino al COMMIT di tutte le TX da cui ha letto

4) 2PL STRETTO: I locks possono essere RILASCIATI soltanto DOPO il COMMIT o l'ABORT.
Per evitare PHANTOM INSERT → Lock di Predicato per vietare inserimenti che verificano il predicato.

* Uno schedule S in 2PL è sempre CONFLICT-SERIALIZZABILE. NON il viceversa

CONTROLLO BASATO SU TIMESTAMP

- TIMESTAMP = identificatore che definisce un ordinamento totale sugli eventi di un sistema.
  - Le TX sono ordinate secondo l'ordine di arrivo;
  - Le TX eseguono liberamente;
  - Ad ogni operazione r/w, lo scheduler controlla che i timestamps delle TX coinvolte non violino l'ordinamento seriale; se lo violano, le UCCIDE.

Lo Scheduler ha 2 contatori RTM(x) e WTM(x) per ogni oggetto x:

- RTM(x) = più alto timestamp tra i ts delle transazioni che hanno letto x (ULTIMA LETTURA)
- WTM(x) = più alto timestamp tra i ts delle transazioni che hanno scritto x (ULTIMA SCRITTURA).

- POLITICA DELLO SCHEDULER:

Lo scheduler riceve richieste di tipo r/w con il (ts) delle TX:

(1) • READ (x, ts):
- se ts < WTM(x) ⇒ richieste respinte e TX con timestamp = ts viene uccisa
- altrimenti, richieste accolte e RTM(x) = max{RTM(x), ts}

(2) • WRITE (x, ts):
- se ts < WTM(x) o ts < RTM(x) ⇒ richieste respinte e TX uccise
- altrimenti, richieste accolte e WTM(x) = ts.

* Funziona sotto l'ipotesi di commit-protezione

(3) "Bufferizzare" le scritture e scriverle in memoria di massa solo DOPO il COMMIT.

(1) → Evita letture inconsistenti e aggiornamento fantasma

(2) → Evita la perdita di aggiornamento (w-w)

(3) → Evita dirty reads (READ COMMITTED sostanzialmente).

ESERCIZIO ALBERO

Albero di ORDINE 6 $\Rightarrow 2t = 6 \Rightarrow$ Ogni nodo ha al più:

- $2t$ nodi figli (puntatori)
- $2t-1$ chiavi

OCCUPAZIONE MINIMA: $t-1$

- INSERIMENTO:
  1. Tramite Binary Search, giungere al nodo foglia dove inserire la chiave;
  2. Inserire la chiave;
  3. Se il nodo foglia ha $2t$ chiavi (1 in più), fare uno SPLIT, facendo risalire nel padre la chiave in posizione a metà nodo.

- CANCELLAZIONE:
  1. Se chiave da rimuovere è in un nodo V NON FOGLIA:
     2. Individua W, nodo contenente il PREDECESSORE di K;
     3. Sposta la MASSIMA CHIAVE di W al posto di K da rimuovere;
     4. Rimuovere la massima chiave da W.
  5. Altrimenti, se K è in un nodo V FOGLIA:
     6. Se V contiene più di $t-1$ chiavi, rimuovere K e termina;
     7. Se V contiene $t-1$ chiavi, rimuovere K e agire così:
        8. Se almeno uno dei FRATELLI ADIACENTI ha $> t-1$ chiavi, si ridistribuiscono le chiavi;
        9. Se nessuno dei fratelli adiacenti ha $> t-1$ chiavi, si effettua un MERGE! (Scende la K padre)

ESERCIZIO CONCORRENZA

Richieste che arrivano al gestore di controllo della concorrenza:

$r_3(x)\ r_2(x)\ r_4(y)\ w_2(x)\ c_2\ r_6(y)\ r_1(x)\ c_1\ w_3(x)\ c_3\ w_4(y)\ c_4\ w_7(x)\ c_7\ w_6(y)\ c_6\ r_5(x)\ c_5$

Quali operazioni e in che ordine vengono eseguite da controllori di tipo:

- 2PL stretto;
- Timestamp ?

- **ASSUNZIONI**:
  1) Se TX bloccata, poi quando viene concesso il lock le sue richieste arrivano una dopo l'altra;
  2) Deadlock subito rilevato e uccise TX che ha fatto ultime richieste;
  3) Ogni TX uccisa, viene riavviata subito dopo aver sbloccato il deadlock concedendo i lock.

X QUEUE

tx in attesa	tx su cui è bloccata	tx uccise
2	3	
3	2	3

Y QUEUE

tx in attesa	tx su cui è bloccata	tx uccise
4	6	
6	4	6

$S = r_3(x)\ r_2(x)\ r_4(y)\ r_6(y)\ r_1(x)\ c_1\ a_3\ w_2(x)\ c_2\ r_3(x)\ w_3(x) c_3\ w_7(x)\ c_7\ a_6\ w_4(y)\ c_4$
$r_6(y)\ w_6(y)\ c_6\ r_5(x)\ c_5$

- **ASSUNZIONI**:
  1) L'identificatore della transazione corrisponde al timestamp (t_i è più giovane di $t_j \iff i > j$)
  2) Ogni TX uccisa riparte subito con un timestamp opportuno (il max tra tutte + 1).

X QUEUE

RTM	WTM	TX uccise
3		2
4.2	4.2	1
6.1		3
6.3	6.3	
	7	5
7.6.5		

Y QUEUE

RTM	WTM	TX uccise
4		
6		4
6.4	6.4	6
7.6	7.6	

$S = r_3(x)\ r_2(x)\ r_4(y)\ a_2\ r_{4.2}(x)\ w_{4.2}(x)\ c_{4.2}\ r_6(y)\ a_1\ r_{6.1}(x)\ c_{6.1}\ a_3\ r_{6.3}(x) w_{6.3}(x) c_{6.3}$
$a_4\ r_{6.4}(y)\ w_{6.4}(y)\ c_{6.4}\ w_7(x)\ c_7\ a_6\ r_{7.6}(y)\ w_{7.6}(y)\ c_{7.6}\ a_5\ r_{7.6.5}(x)\ c_{7.6.5}$

Forme Normali

- **1NF:** Se
  - ha una PK
  - non ci sono attributi ripetuti
  - non ci sono attributi compositi
- **2NF:** NO DIPENDENZE PARZIALI → I valori non dipendono da parti di chiavi, ma da intere chiavi.
- **3NF:** NO DIPENDENZE TRANSITIVE → No dipendenze funzionali "a cascata"
- **BCNF:** Se per ogni FD: $X \rightarrow Y$, X è una superchiave di R ed R è in 3NF.

Livelli di Isolamento

- **READ UNCOMMITTED:** Le SELECT sono eseguite in modalità non bloccante.
  NON vengono acquisiti Lock.
  ANOMALIE → dirty read, unrepeatable read, phantom update, phantom insert
- **READ COMMITTED:** Lock in scrittura rilasciati alla fine della TX;
  Lock in lettura rilasciati alla fine della lettura.
  Risolve dirty read.
  ANOMALIE → unrepeatable read, phantom update, phantom insert
- **REPEATABLE READ:** Sia i Lock in scrittura che in lettura vengono rilasciati alla fine della transazione, dopo il COMMIT o l'ABORT.
  Ciò viene fatto applicando il 2PL STRETTO.
  Risolve unrepeatable read e phantom update.
  ANOMALIE → phantom insert
- **SERIALIZABLE:** Tutti i Lock vengono rilasciati alla fine della transazione, utilizzando il 2PL stretto; inoltre viene mantenuto un Lock di predicato che rispecchia la condizione nel WHERE della SELECT e vieta l'inserimento di tuple che rispettano il predicato.
  Risolve phantom insert.
  ANOMALIE → Nessuna. Tuttavia alto OVERHEAD per i Lock e aumenta il rischio di DEADLOCK!

C → search_struttura_primario (AMMINISTRATORE.c , riga 758)

SQL → search_strutture_primario (privilegies → EXECUTE (amministratore))

- setup_prepared_stmt:
  1. mysql_stmt_init (conn)
  2. mysql_stmt_prepare (*stmt, statement, strlen(statement))
     → Prepara lo statement con i PLACEHOLDERS!
  3. mysql_stmt_attr_set (*stmt, STMT_ATTR_UPDATE_MAX_LENGHT, &update_length);
     → Aggiorna i metadati sulla lunghezza delle colonne, quando fa il fetching dei dati.

- mysql_stmt_bind_param (p_stmt, param):
  Fa il binding dei parametri in input al posto dei PLACEHOLDERS.

- mysql_stmt_execute (p_stmt):
  Sostituisce al posto dei placeholders i parametri di input ed esegue la query specificata nello stmt.

- do { ...
  } while (status == 0) → mysql_stmt_next_result (p_stmt)
  - >0 → errore
  - =0 → ancora result set da estrarre
  - -1 → EOT, finito

- dump_result_set:
  1. mysql_stmt_store_result → bufferizza il result set completo sul client
  2. dump_result_set_header → stampa l'intestazione delle tabelle
  3. mysql_stmt_bind_result → associa le colonne in output ai buffer dei dati e ai buffer delle lunghezze
  4. while(true) { status = mysql_stmt_fetch() → ritorna la prossima riga del RS nei buffer di cui si è fatto il binding
     ...
  5. print_dashes (rs_metadata)
  6. mysql_free_result (rs_metadata) → free dei metadati
  7. free dei buffer usati per il binding (malloc in precedenza)