BASI DI DATI 3° parte

TRANSAZIONI e ARCHITETTURA di un DBMS

CONTESTO

Argomenti del Modulo di Tecnologie per basi di dati

→ Tecniche per l'implementazione dei sistemi che gestiscono basi di dati (DBMS)

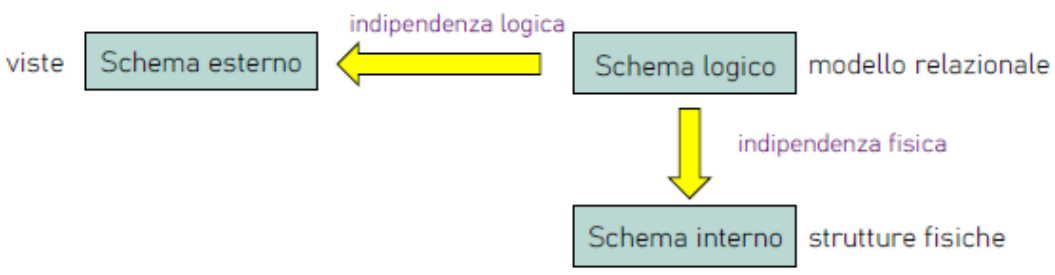
→ DBMS: sistema per la gestione di basi di dati (DataBaseManagement System)

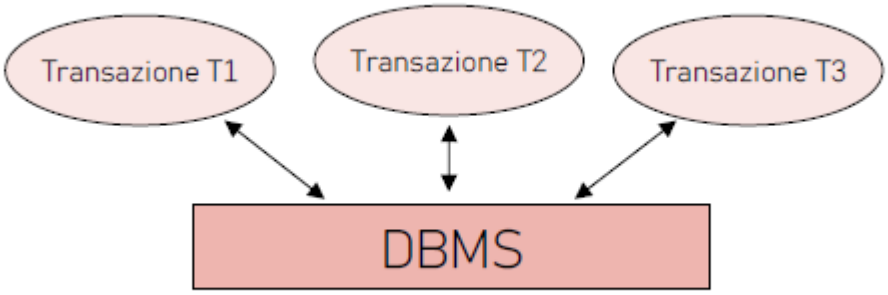
Un DBMS (Database Management System) o sistema di gestione di basi di dati è un sistema software in grado di gestire collezioni di dati che siano grandi, condivisee persistenti, assicurando la loro affidabilitàe privatezza.

Un DBMS in quanto sistema software deve essere efficienteed efficace.Dipendenza con la bontà della progettazione della base di dati.

Un DMBS estende le funzionalità del file systemGestione in memoria secondaria collezioni di dati.

Una base di dati è una collezione di dati gestita da un DBMS.





RIASSUNTO

TRANSAZIONI:

Definizione di transazione:

È un'unità di lavorosvolta da un programma applicativo (che interagisce con una base di dati) per la quale si vogliono garantire proprietà di correttezza, robustezza e isolamento.

Principale caratteristica di una transazione:

Una transazione o va a buon fine e ha effetto sulla base di dati o abortisce e non ha nessun effetto sulla base di dati, non sono ammesse esecuzioni parziali.

Sintassi per definire una transazione in SQL

<transazione> → begin transaction
<programma>
end transaction

<programma> → ⟨<istruzione> | commit work | rollback work⟩
{⟨<istruzione> | commit work | rollback work⟩}

La transazione va a buon fine all'esecuzione di un commit work.

La transazione non ha alcun effetto se viene eseguito un rollback work.

Una transazione è ben formata se:

Inizia con un begin transaction.

Termina con un end transaction.

La sua esecuzione comporta il raggiungimento di un commit o di un rollback worke dopo il commit/rollbacknon si eseguono altri accessi alla base di dati.

Esempio di transazione ben formata:

```
begin transaction;
        update CONTO set saldo = saldo – 1200
                where filiale = '005' and numero = 15;
        update CONTO set saldo = saldo + 1200
                where filiale = '005' and numero = 205;
        commit work;
end transaction;
```

PROPRIETA' ACIDE:

Un DBMS che gestisce transazioni dovrebbe garantire per ogni transazione che esegue le proprietà di:

ATOMICITÀ	Atomicity
CONSISTENZA	Consistency
ISOLAMENTO	Isolation
PERSISTENZA	Durability

ATOMICITA':

DESCRIZIONE:

- Una transazione è una unità di esecuzione indivisibile. O viene eseguita completamente o non viene eseguita affatto.

IMPLICAZIONI:

- Se una transazione viene interrotta primadel commit, il lavoro fin qui eseguito dalla transazione deve essere disfatto ripristinando la situazione in cui si trovava la base di dati prima dell'inizio della transazione.
- Se una transazione viene interrotta dopol'esecuzione del commit (commit eseguito con successo), il sistema deve assicurare che la transazione abbia effetto sulla base di dati.

CONSISTENZA:

DESCRIZIONE:

- L'esecuzione di una transazione non deve violare i vincoli di integrità.

IMPLICAZIONI:

- Verifica immediata:
    - Fatta nel corso della transazione
    - Viene abortita solo l'ultima operazione e il sistema restituisce all'applicazione una segnalazione d'errore
    - L'applicazione può quindi reagire alla violazione.

- Verifica differita:
    - Al commit se un vincolo di integrità viene violato la transazione viene abortita senza possibilità da parte dell'applicazione di reagire alla violazione.

ISOLAMENTO:

DESCRIZIONE:

- L'esecuzione di una transazione deve essere indipendente dalla contemporanea esecuzione di altre transazioni.

IMPLICAZIONI:

- Il rollback di una transazione non deve creare rollback a catena di altre transazioni che si trovano in esecuzione contemporaneamente.
- Il sistema deve regolare l'esecuzione concorrente con meccanismi di controllo dell'accesso alle risorse.

PERSISTENZA:

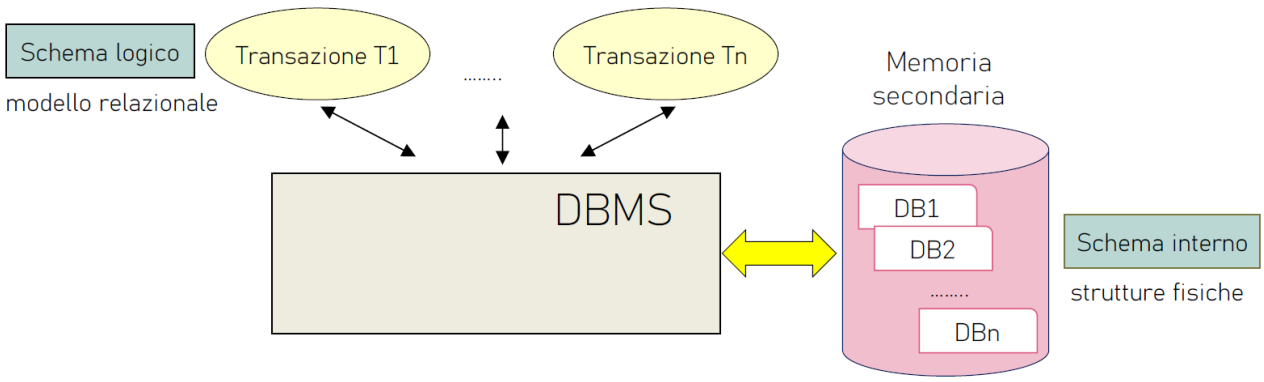
DESCRIZIONE:

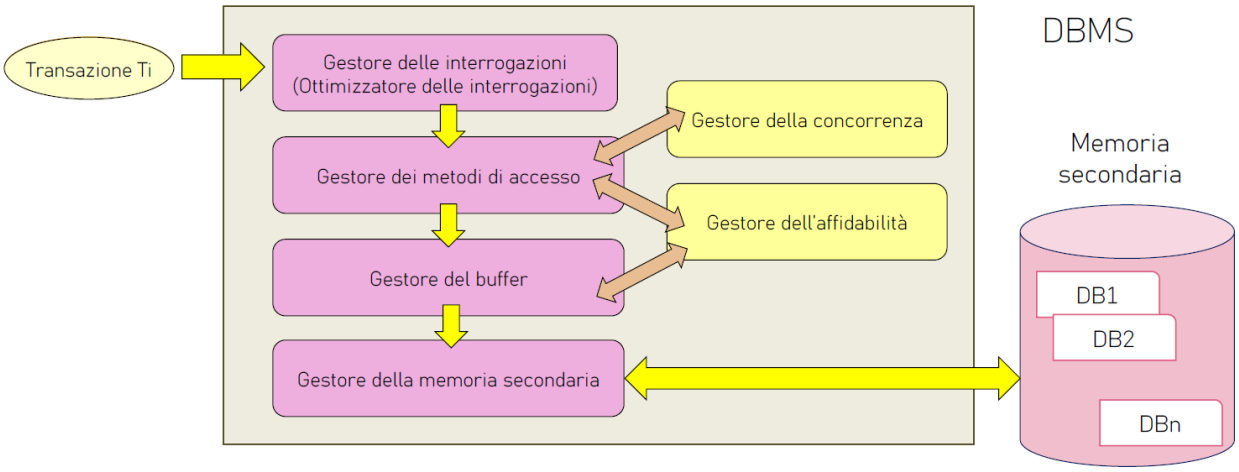
- L' effetto di una transazione che ha eseguito il commit non deve andare perso.

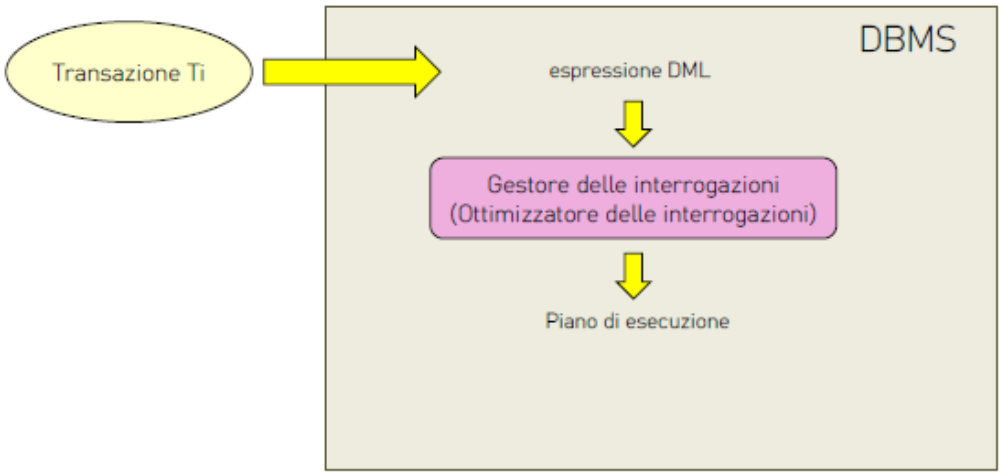
IMPLICAZIONI:

- Il sistema deve essere in grado, in caso di guasto, di garantire gli effetti delle transazioni che al momento del guasto avevano già eseguito un commit

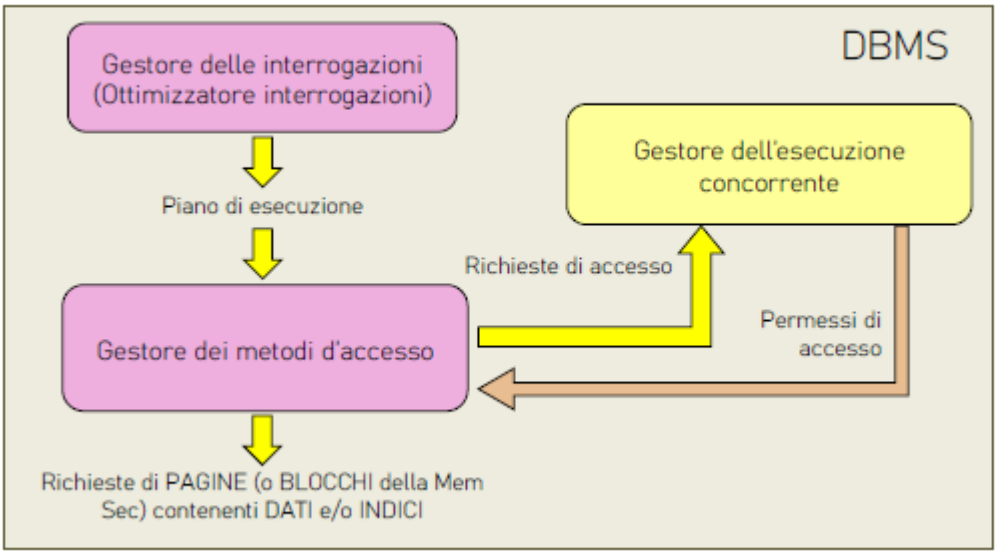
MODULI DBMS:



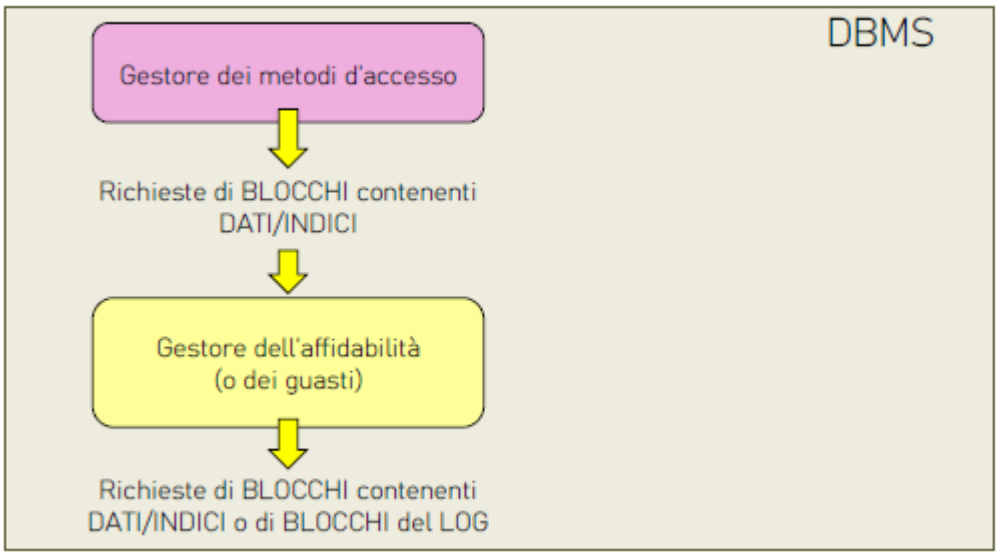

DBMS
Transazione Ti
Gestore delle interrogazioni
(Ottimizzatore delle interrogazioni)
Gestore della concorrenza
Gestore dei metodi di accesso
Gestore dell'affidabilità
Gestore del buffer
Gestore della memoria secondaria
Memoria
secondaria
DB1
DB2
DBn

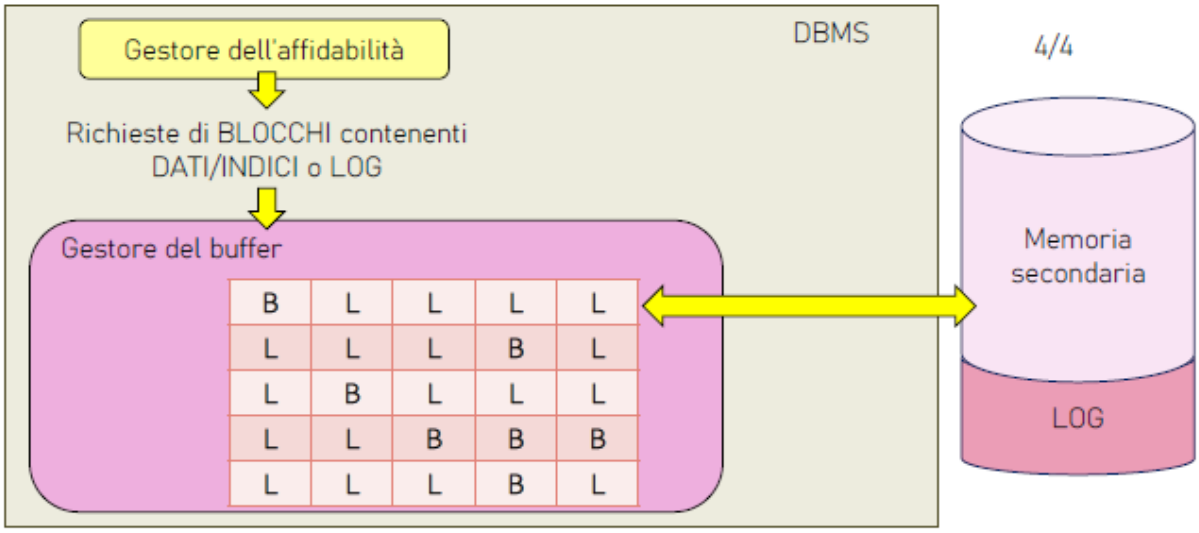

DBMS
Transazione Ti
espressione DML
Gestore delle interrogazioni
(Ottimizzatore delle interrogazioni)
Piano di esecuzione

1/4


DBMS
Gestore delle interrogazioni
(Ottimizzatore interrogazioni)
Piano di esecuzione
Gestore dell'esecuzione
concorrente
Richieste di accesso
Permessi di
accesso
Gestore dei metodi d'accesso
Richieste di PAGINE (o BLOCCHI della Mem
Sec) contenenti DATI e/o INDICI

2/4





BUFFER:

Le basi di dati gestite da un DBMS devono risiedere necessariamente nella memoria secondaria in quanto sono molto grandi e persistenti(hanno un tempo di vita che non è limitato all'esecuzione dei programmi che le utilzzano).

Tuttavia accedere in memoria secondaria è costoso di conseguenza si cerca il più possibile di diminuire gli accessi in memoria secondaria, questo è possibile con una gestione ottimale del buffer(*).

Di questa gestione ottimale del buffer si occupa il modulo del DBMS: GESTORE DEL BUFFER.

(*)Il buffer è una zona di memoria dedicata all'interazione tra memoria secondaria e memoria centrale, è organizzato in pagine le quali hanno le dimensioni di un blocco della memoria secondaria.

GESTORE DEL BUFFER:

- Il gestore del buffer si occupa del caricamento/salvataggio delle pagine in memoria secondaria a fronte di richieste di lettura/scrittura.

  - Lettura di un blocco:
    - se il blocco è presente in una pagina del buffer allora non si esegue una lettura su memoria secondaria e si restituisce un puntatore alla pagina del buffer,
    - altrimenti si cerca una pagina libera e si carica il blocco nella pagina, restituendo il puntatore alla pagina stessa.
  - Scrittura di un blocco:
    - In caso di richiesta di scrittura di un blocco precedentemente caricato in una pagina del buffer, il gestore del buffer può decidere di differire la scrittura su memoria secondaria in un secondo momento.

  In entrambi i casi (lettura e scrittura) l'obiettivo e quello di aumentare la velocità di accesso ai dati.

- Tale comportamento del gestore dei buffer si basa sul principio di LOCALITÀ:
  "I dati referenziati di recente hanno maggiore probabilità di essere referenziati nuovamente in futuro".
- Inoltre, una nota legge empirica dice che:
  "il 20% dei dati e acceduto dall'80% dell'applicazioni".
- Tutto ciò rende conveniente dilazionare la scrittura su memoria secondaria delle pagine del buffer →
  anziché scrivere immediatamente i dati modificati dalla memoria del buffer alla memoria secondaria ogni volta che vengono modificati, si ritarda questa operazione fino a quando diventa strettamente necessaria.

Per ogni pagina del buffer si memorizza il blocco B contenuto indicando

- il file
- numero di blocco (o offset)

Per ogni pagina del buffer si memorizza anche un insieme di variabili di stato tra cui si trovano sicuramente:

- un contatore I: indica il numero di transazioni che utilizzano le pagine
- un bit di stato J: indica se la pagina è stata modificata(=1), oppure no(=0).

Per la gestione del buffer vengono utilizzate alcune primtive, tra cui:

- **fix**: Richiesta di accesso a un blocco, restituisce un puntatore.
    - Cerca il blocco nel buffer.
    - Se non presente, sceglie una pagina libera o "rubata" basandosi su criteri come LRU o FIFO.
    - Carica il blocco e aggiorna i riferimenti.
    - Incrementa il contatore per la pagina acceduta.

- **flush**: Salvataggio asincrono del blocco in memoria secondaria.
- **force**: Salvataggio sincrono del blocco in memoria secondaria.
- **setDirty**: Indica modifica della pagina, imposta bit di stato.
- **unfix**: Indica fine uso del blocco, decrementa contatore I.

GESTORE DELL' AFFIDABILITA'

E' responsabile per ciò che riguarda:

- L'esecuzione delle istruzioni per la gestione delle transazioni
- La realizzazione le operazioni necessarie al ripristino della base di dati dopo eventuali malfunzionamenti

Per il suo funzionamento il gestore dell'affidabilità deve disporre di un dispositivo di MEMORIA STABILE ovvero una memoria resistente ai guasti.

- In **memoria stabile** viene memorizzato il **file di LOG** che registra in modo sequenziale le operazioni eseguite dalle transazioni sulla base di dati.
- RECORD memorizzati nel file di LOG:
    - Record di TRANSAZIONE
    - Record di SISTEMA

RECORD DI TRANSAZIONE:

- BEGIN → B(T)
- COMMIT → C(T)
- ABORT → A(T)
- INSERT → I(T, O, AS)
- DELETE → D(I, O, BS)
- UPDATE → U(I, O, BS, AS)

BS = BEFORE STATE

AS = AFTER STATE

- I record di transazione salvati nel LOG consentono di eseguire in caso di ripristino le seguenti operazioni:
  - UNDO: per disfare un'azione su un oggetto O è sufficiente ricopiare in O il valore BS; l'insert/delete viene disfatto cancellando/inserendo O;
  - REDO: per rifare un'azione su un oggetto O è sufficiente ricopiare in O il valore AS; l'insert/delete viene rifatto inserendo/cancellando O;

RECORD DI SISTEMA:

DUMP → salva lo stato del database

CHECKPOINT → CK(T1, … , Tn) → indica che all'esecuzione del CheckPoint le transazioni attive erano T1, …, Tn.

Operazione svolta periodicamente dal gestore dell'affidabilità; prevede i seguenti passi:

- Sospensione delle operazioni di scrittura, commit e abort delle transazioni
- Esecuzione della primitiva force sulle pagine "dirty" di transazioni che hanno eseguito il commit
- Scrittura sincrona (primitiva force) sul file di LOG del record di CheckPoint con gli identificatori delle transazioni attive
- Ripresa delle operazioni di scrittura, commit e abort delle transazioni

Guasto prima della scrittura del RECORD di COMMIT → UNDO

Guasto dopo la scrittura del RECORD di COMMIT → REDO

Guasto:

- Guasto di sistema → perdita di contenuto nella memoria centrale
  - RIPRESA A CALDO
- Guasto di dispositivo → perdita di contenuto della base di dati in memoria secondaria
  - RIPRESA A FREDDO

COSE IMPORTANTI:

RIPRESA A CALDO:

1) Inizialmente si accede all'ultimo blocco del LOG e si ripercorre all'indietro il log fino al più recente record CK.
2) Successivamente si procede a identificare le transazioni da rifare o disfare.
   Questo viene fatto inizializzando due insiemi: UNDO e REDO.
   L'insieme UNDO viene popolato con le transazioni attive al momento del checkpoint, mentre l'insieme REDO viene inizializzato come vuoto

3) Il processo procede con la scansione del LOG in avanti (dal checkpoint al guasto).
   Per ogni record B(T) incontrato, T viene aggiunto all'UNDO mentre invece per ogni record C(T) incontrato T viene spostato da UNDO a REDO.

4) Adesso si ripercorre all'indietro il file di LOG e si disfano le operazioni eseguite dalle transazioni dell'insieme UNDO partendo dal guasto risalendo fino alla prima azione della transazione più vecchia.
5) In ultimo vengono rifatte le operazioni delle transazioni dell'insieme REDO.

RIPRESA A FREDDO:

1) Si accede al DUMP e si ricopia selettivamente la parte deteriorata della base di dati
2) Dopo si accede al LOG risalendo al record di DUMP
3) A questo punto si procede in avanti nel file di LOG rieseguendo tutte le operazioni relative alla parte deteriorata comprese le azioni di commit e abort
4) Infine si applica una ripresa a caldo

Quali moduli contribuiscono a garantire le proprietà delle transazioni?

Gestore dei metodi d'accesso garantisce la proprietà di CONSISTENZA.

Gestore dell'esecuzione concorrente garantisce le proprietà di ATOMICITA' e ISOLAMENTO

Gestore dell'affidabilità garantisce le proprietà di ATOMICITA' e PERSISTENZA

GESTIONE GUASTO: ALGORITMO A RIPRESA A CALDO E ALGORITMO A RIPRESA A FREDDO

RISPOSTE ALLE DOMANDE:

Domande di teoria tratte dalla terza prova intermedia dell'esame 06/2015.
(3 punti) Illustrare l'architettura di un DBMS descrivendo in particolare il modulo di gestione dei buffer; si indichi inoltre, per ogni modulo dell'architettura, quali sono le proprietà delle transazioni che contribuisce a garantire.

Domande di teoria tratte dalla terza prova intermedia dell'esame 07/06/2016.
(3 punti) Illustrare l'architettura di un DBMS descrivendo in particolare il modulo di gestione dei guasti (o gestore dell'affidabilità); si indichi inoltre, per ogni modulo dell'architettura, quali sono le proprietà delle transazioni che contribuisce a garantire.

Domande di teoria tratte dalla terza prova intermedia dell'esame 21/04/2022.
(3 punti) Illustrare l'architettura di un DBMS descrivendo in particolare il modulo di gestione dei buffer; si indichi inoltre quali moduli garantiscono le proprietà di persistenza e consistenza delle transazioni.

(2 punti) Lo studente illustri l'algoritmo di ripresa a caldo.

Domande di teoria tratte dalla terza prova intermedia dell'esame 22/04/2022.
(3 punti) Illustrare le proprietà delle transazioni; si indichi inoltre quali moduli di un DBMS garantiscono ciascuna di tali proprietà.

Domande di teoria tratte dalla terza prova intermedia dell'esame 10/06/2022.
(2 punti) Illustrare le proprietà delle transazioni ed indicare da quali moduli del DBMS vengono garantite.

ESERCIZIO: APPLICARE ALGORITMO DI RIPRESA A CALDO

Passaggi:

STRUTTURE FISICHE e STRUTTURE di ACCESSO ai DATI

CONTESTO

Abbiamo visto prima il GESTORE DELL' AFFIDABILITA' e il GESTORE DEL BUFFER, adesso vediamo il GESTORE DEI METODI DI ACCESSO:

RIASSUNTO

GESTORE DEI METODI DI ACCESSO

Metodi d'accesso sono i moduli software che implementano gli algoritmi di *accesso* e *manipolazione* dei dati organizzati in specifiche strutture fisiche come:

- Scansione sequenziale
- Accesso via indice
- Ordinamento
- Varie implemetazioni del join

Ogni metodo d'accesso ai dati conosce:

- L'organizzazione delle tuple/record di indice nei blocchi DATI/INDICE salvati in memoria secondaria (come una tabella/indice viene organizzata in blocchi DATI della memoria secondaria)

- L'organizzazione fisica interna dei blocchi sia quando contengono DATI (vale a dire, tuple di una tabella) sia quando contengono strutture fisiche di accesso o INDICI (vale a dire, record di un indice).

In un blocco DATI sono presenti informazioni utili e informazioni di controllo:

- Informazioni utili: tuple della tabella
- Informazioni di controllo: dizionario, bit di parità, altre informazioni del file system o della specifica struttura fisica.

STRUTTURA SEQUENZIALE ORDINATA:

- Una struttura sequenziale ordinata è un file sequenziale dove le tuple sono ordinate secondo una chiave di ordinamento

Operazioni:

- Inserimento di una tupla:
    - Individuare il blocco B che contiene la tupla che precede , nell'ordine della chiave, la tupla da inserire.
    - Inserire la tupla nuova in B; se l'operazione non va a buon fine si aggiunge un nuovo blocco (detto, overflow page) alla struttura: il blocco contiene la nuova tupla, altrimenti si prosegue.
    - Aggiustare la catena di puntatori.
- Scansione sequenziale ordinata secondo la chiave (seguendo i puntatori)
- Cancellazione di una tupla:
    - Individuare il blocco B che contiene la tupla da cancellare.
    - Cancellare la tupla da B.
    - Aggiustare la catena di puntatori.
- Riorganizzazione: si assegnano le tuple ai blocchi in base ad opportuni coefficienti di riempimento, riaggiustando i puntatori.

Per aumentare le prestazioni degli accessi alle tuple memorizzate nelle strutture fisiche (file sequenziale), si introducono strutture ausiliarie (dette strutture di accesso ai dati o INDICI Tali strutture velocizzano l'accesso casuale via chiave di ricerca. La chiave di ricerca è un insieme di attributi utilizzati dall'indice nella ricerca.

INDICE PRIMARIO: la chiave di ordinamento del file sequenziale coincide con la chiave di ricerca dell'indice.

Ogni record dell'indice primario contiene una coppia <vi, pi>:

- vi : valore della chiave di ricerca;
- pi : puntatore al primo record nel file sequenziale con chiave *vi.*

Esistono due varianti dell'indice primario:

- Indice **denso**: per ogni occorrenza della chiave presente nel file esiste un corrispondente record nell'indice.

- Indice **sparso**: solo per alcune occorrenze della chiave presenti nel file esiste un corrispondente record nell'indice, tipicamente una per blocco.

OPERAZIONI INDICE PRIMARIO **DENSO**:

RICERCA di una tupla con chiave di ricerca K:

Scansione sequenziale dell'indice per trovare il record (K, pk):

- Accesso al file attraverso il puntatore pk
- Costo: 1 accesso indice + 1 accesso blocco dati

INSERIMENTO di un record nell'indice:

Come inserimento nel FILE SEQUENZIALE (nel blocco della memoria secondaria invece di tuple ci sono record dell'indice)

L'inserimento nell'indice avviene solo se la tupla inserita nel file ha un valore di chiave K che non è già presente.

CANCELLAZIONE di un record nell'indice

Come cancellazione nel FILE SEQUENZIALE:

- La cancellazione nell'indice avviene solo se la tupla cancellata nel file è l'ultima tupla con valore di chiave K

OPERAZIONI INDICE PRIMARIO **SPARSO**:

RICERCA di una tupla con chiave di ricerca K:

Scansione sequenziale dell'indice per trovare il record (K', pk') dove K' è il valore più grande che sia minore o uguale a K:

- Accesso al file attraverso il puntatore pk' e scansione del file (blocco corrente) per trovare le tuple con chiave K.
- Costo: 1 accesso indice + 1 accesso blocco dati

INSERIMENTO di un record nell'indice:

Come inserimento nel FILE SEQUENZIALE (nel blocco della memoria secondaria invece di tuple ci sono record dell'indice)

L'inserimento avviene solo quando, per effetto dell'inserimento di una nuova tupla, si aggiunge un blocco dati alla struttura; in tutti gli altri casi l'indice rimane invariato.

CANCELLAZIONE di un record nell'indice

Come cancellazione nel FILE SEQUENZIALE

- La cancellazione nell'indice avviene solo quando *K* è presente nell'indice e il corrispondente blocco viene eliminato; altrimenti, se il blocco sopravvive, va sostituito *K* nel record dell'indice con il primo valore *K*' presente nel blocco.

INDICE SECONDARIO: in questo caso invece la chiave di ordinamento e la chiave di ricerca sono diverse (gli indici secondari sono sempre DENSI)

Ogni record dell'indice secondario contiene una coppia <vi, pi>:

- vi : valore della chiave di ricerca;

- pi : puntatore al bucket di puntatori che individuano nel file sequenziale tutte le tuple con valore di chiave vi.

OPERAZIONI:

RICERCA:

- Scansione sequenziale dell'indice per trovare il record (K, pk)
- Accesso al bucketB di puntatori attraverso il puntatore pk
- Accesso al file attraverso i puntatori del bucket B.

Costo: 1 accesso indice + 1 accesso al bucket + n accessi pagine dati

INSERIMENTO E CANCELLAZIONE:

- Come indice primario denso con in più l'aggiornamento dei bucket.

B+-TREE

Caratteristiche generali dell'indice B+-tree:

- È una struttura ad albero;
- Ogni nodo corrisponde ad una pagina della memoria secondaria;
- I legami tra nodi diventano puntatori a pagina;
- Ogni nodo ha un numero elevato di figli, quindi l'albero ha tipicamente pochi livelli e molti nodi foglia;
- L'albero è bilanciato: la lunghezza dei percorsi che collegano la radice ai nodi foglia è costante;
- Inserimenti e cancellazioni non alterano le prestazioni dell'accesso ai dati: l'albero si mantiene bilanciato.

Struttura di un B+-tree (fan-out = n):

→ NODO FOGLIA

- può contenere fino a (n-1) valori ordinatidi chiave di ricerca e fino a n puntatori.
- I nodi foglia sono ordinati. Inoltre, dati due nodi foglia Li e L j con i<j risulta: ∀K t ∈ L i : ∀K s ∈ Lj: K t <Ks.
- Vincolo cardinalità #chiavi → ⌈ (n −1) / 2 ⌉ <= #chiavi <= (n −1)

→ NODO INTERMEDIO

- Sequenza di m' ≤n valori ordinati di chiave può contenere fino a n puntatori a nodo
- Ogni chiave Ki è seguita da un puntatore pi
- Vincolo cardinalità #puntatori→ ⌈ n / 2 ⌉ <= #puntatori <= n

Operazioni: RICERCA CON CHIAVE K

Passo 1) Cercare nel nodo radice il più piccolo valore di chiave maggiore di K.

Se tale valore esiste (supponiamo sia Ki) allora seguire il puntatore pi altrimenti se non esiste si segue il puntatore successivo

Passo 2) Se il nodo raggiunto è un nodo foglia cercare il valore K nel nodo e seguire il corrispondente puntatore verso le tuple(nel caso ho indice primario) o verso al buckets puntatori(nel caso ho indice secondario), altrimenti riprendere il passo 1.

Il costo di una ricerca nell'indice, in termini di numero d'accessi alla memoria secondaria è uguale al numero di nodi acceduti alla ricerca che nella struttura ad albero è uguale alla sua profondità che è funzione del fan-out n e del #valoriChiave e ha questa formula:

$$prof_{B+tree} \leq 1 + \log_{\lceil n/2 \rceil}\left(\frac{\#valoriChiave}{\lceil (n-1)/2 \rceil}\right)$$

<u>Operazioni: INSERIMENTO con chiave K</u>

Passo 1 → ricerca del nodo foglia NF dove il valore K va inserito

Passo 2 →

se K è presente in NF, allora:

- Indice primario: nessuna azione
- Indice secondario: aggiornare il bucket di puntatori

altrimenti, inserire K in NF rispettando l'ordine e:

- Indice primario: inserire puntatore alla tupla con valore K della chiave
- Indice secondario: inserire un nuovo bucket di puntatori contenente il puntatore alla tupla con valore K della chiave.

se non è possibile inserire K in NF(inserendo K in NF supererei il vincolo massimo di #chiavi che è n-1), allora eseguire uno SPLIT di NF.

SPLIT di un nodo foglia

Nel nodo da dividere esistono n valori chiave, si procede come segue:

- Creare due nodi foglia;
- Inserire i primi $\lceil (n-1) / 2 \rceil$ valori nel primo;
- Inserire i rimanenti nel secondo;
- Inserire nel nodo padre un nuovo puntatore per il secondo nodo foglia generato e riaggiustare i valori chiave presenti nel nodo padre.
- Se anche il nodo padre è pieno (n puntatori già presenti) lo SPLIT si propaga al padre e così via, se necessario, fino alla radice.

<u>Operazioni: CANCELLAZIONE con chiave K</u>

Passo 1– ricerca del nodo foglia NF dove il valore K va cancellato

Passo 2 –cancellare K da NF insieme al suo puntatore:

- Indice primario: nessuna ulteriore azione
- Indice secondario: liberare il bucket di puntatori

Se dopo la cancellazione di K da NF viene violato il vincolo di riempimento minimo di NF, allora eseguire un MERGE di NF.

MERGE di un nodo foglia

Se nel nodo da unire esistono $\lceil (n-1) / 2 \rceil - 1$ valori chiave, si procede come segue:

- Individuare il nodo fratello adiacente da unire al nodo corrente;
- Se i due nodi hanno complessivamente al massimo n-1 valori chiave, allorasi genera un unico nodo contenente tutti i valori
- si toglie un puntatore dal nodo padre
- si aggiustano i valori chiave del nodo padre
- Altrimenti si distribuiscono i valori chiave tra i due nodi e si aggiustano i valori chiave del nodo padre

- Se anche il nodo padre viola il vincolo minimo di riempimento (meno di ⌈ n / 2 ⌉ puntatori presenti), il MERGE si propaga al padre e così via, se necessario, fino alla radice.

HASH

Caratteristiche generali delle strutture ad accesso calcolato:

Gli indici hash sono una struttura fisica che permette l'indicizzazione efficiente e veloce delle tuple di una base di dati.

Si basano su una funzione di hash che mappa i valori della chiave di ricerca sugli indirizzi di memorizzazione(buckets di puntatori) delle tuple nelle pagine dati della memoria secondaria.

Per ottenere la funzione una funzione di hash ottimale :

- Si stima il numero di valori chiave che saranno contenuti nella tabella;
- Si alloca un numero di bucket di puntatori (B) uguale al numero stimato;
- Si definisce una funzione di FOLDING che trasforma i valori chiave in numeri interi positivi:f: K →Z+
- Si definisce una funzione di HASHING:h: Z+→B

Una buona funzione di hashing distribuisce in modo UNIFORME e CASUALE i valori della chiave nei bucket, se infatti non fosse così ci sarebbe un numero eccessivo di collisioni(*) che porterebbe alla saturazione dei bucket corrispondenti rendendo necessari i bucket di overflow vanificando i vantaggi in termini di accesso veloce garantiti dalla struttura hash.

(*)COLLISIONE: si verifica quando, dati due valori di chiave K1 e K2 con K1≠K2, risulta: h(f(K1)) = h(f(K2))

OPERAZIONI : RICERCA di un valore chiave K

Dato un valore di chiave K trovare la corrispondente tupla:

- Calcolare b = h(f(K)) (costo zero)
- Accedere al bucket b (costo: 1 accesso a pagina)
- Accedere alle *n* tuple attraverso i puntatori del bucket (costo: *m* accessi a pagina con *m*<=*n*)

CONFRONTO B+-TREE E HASHING

- Selezioni basate su condizioni di uguaglianza → A = cost
  - Hashing (senza overflow buckets): tempo costante
  - B+-tree: tempo logaritmico nel numero di chiavi

- Selezioni basate su intervalli (range) → A>cost1 AND A<cost2
  - Hashing: numero elevato di selezioni su condizioni di uguaglianza per scandire tutti i valori del range
  - B+-tree: tempo logaritmico per accedere al primo valore dell'intervallo, scansione dei nodi foglia (grazie all'ultimo puntatore) fino all'ultimo valore compreso nel range.

Inserimenti e cancellazioni:

- Hashing: tempo costante + gestione overflow
- B+-tree: tempo logaritmico nel numero di chiavi + split/merge

COSE IMPORTANTI: OPERAZIONI B+-TREE

RISPOSTE ALLE DOMANDE:

Domande di teoria tratte dalla terza prova intermedia dell'esame 06/2015.
(2 punti) Lo studente illustri la struttura di accesso ai dati denominata indice primario denso: caratteristiche della struttura, ricerca, inserimento e cancellazione di entry dall'indice.

Domande di teoria tratte dalla terza prova intermedia dell'esame 07/06/2016.
(2 punti) Lo studente illustri la struttura di accesso ai dati denominata indice primario sparso, si descrivano in particolare i seguenti punti: (i) le caratteristiche della struttura di accesso, (ii) l'algoritmo di ricerca di una tupla con chiave K usando l'indice.

Domande di teoria tratte dalla terza prova intermedia dell'esame 21/04/2022.
(2 punti) Lo studente illustri la struttura di accesso ai dati denominata struttura ad accesso calcolato (hashing), si descrivano in particolare i seguenti punti: (i) le caratteristiche della struttura di accesso, (ii) l'algoritmo di ricerca di una tupla con chiave K usando l'indice.

Domande di teoria tratte dalla terza prova intermedia dell'esame 22/04/2022.
(2 punti) Lo studente illustri la struttura di accesso ai dati denominata B+-tree, si descrivano in particolare i seguenti punti: (i) le caratteristiche della struttura di accesso, (ii) l'algoritmo di ricerca di una tupla con chiave K usando l'indice.

Domande di teoria tratte dalla terza prova intermedia dell'esame 10/06/2022.
(2 punti) Lo studente illustri la struttura di accesso ai dati denominata indice secondario, si descrivano in particolare i seguenti punti: (i) le caratteristiche della struttura di accesso, (ii) l'algoritmo di ricerca di una tupla con chiave K usando l'indice

---

ESERCIZIO: APPLICARE OPERAZIONI SU UN B+-TREE

Passaggi, operazioni, formule:

ESECUZIONE CONCORRENTE di TRANSAZIONI

CONTESTO

Per gestire con prestazione accettabili il carico di lavoro tipico delle applicazioni gestionali (100 o 1000 tps) un DBMS deve eseguire le transazioni in modo concorrente.

- L'esecuzione concorrente di transazioni senza controllo può generare anomalie o problemi di correttezza (come vedremo in alcuni esempi).
- È quindi necessario introdurre dei meccanismi di controllo nell'esecuzione delle transazioni per evitare tali anomalie.

RIASSUNTO

ANOMALIE DI ESECUZIONE CONCORRENTE

Anomalie tipiche:

Perdita di aggiornamento (update)

Gli effetti di una transazione concorrente sono persi.

Due transazioni leggono lo stesso dato e ciascuna esegue un' operazione di scrittura basata sul valore letto inizialmente.

L'aggiornamento di una transazione viene sovrascritto dall'altra, causando la perdita delle modifiche effettuate da uno degli aggiornamenti.

Lettura inconsistente

Accessi successivi ad uno stesso dato all'interno di una transazione ritornano valori diversi.

Una transazione legge un valore, un'altra transazione lo modifica e fa commit, e la prima transazione legge di nuovo trovando un valore diverso, portando a incoerenze all'interno della stessa transazione.

Lettura sporca

Viene letto un dato che rappresenta uno stato intermedio nell'evoluzione di una transazione.

Si verifica quando una transazione legge dati che sono stati modificati da un'altra transazione ma non ancora committati; se la seconda transazione viene annullata (rollback), la prima transazione ha basato le sue operazioni su dati non validi.

Aggiornamento/inserimento fantasma

Valutazioni diverse di valori aggregati a causa di inserimenti intermedi.

CONTESTO:

Per efficienza è necessaria l'esecuzione concorrente ma non si vuole incorrere in anomalie, si vuole quindi accettare solo sequenze di operazioni in lettura e scrittura di diverse operazioni di diverse transazioni (*SCHEDULE*) solo se questi sono equivalenti a uno schedule seriale.

CONCETTI:

- ***SCHEDULE***: Sequenza di operazioni di lettura e scrittura eseguite su risorse della base di dati da diverse transazioni concorrenti →Esso rappresenta una possibile esecuzione concorrente di diverse transazioni.

- ***SCHEDULE SERIALE***: E' uno schedule dove le operazioni di ogni transazione compaiono in sequenza senza essere inframezzate da operazioni di altre transazioni.

- ***SCHEDULE SERIALIZZABILE***: È uno schedule "equivalente" ad uno schedule seriale

- ***EQUIVALENZA TRA SCHEDULE***: Si vogliono considerare equivalenti due schedule che producono gli stessi effetti sulla base di dati

Ipotesi di commit proiezione

- Si suppone che le transazioni abbiano esito noto. Quindi si tolgono dagli schedule tutte le operazioni delle transazioni che non vanno a buon fine.

VIEW-EQUIVALENZA

LEGGE_DA: insieme di operazioni di lettura da un risorsa che sono immediatamente precedute da operazioni di scrittura sulla stessa risorsa

SCRITTURE_FINALI: insieme di ultime operazioni di scrittura di ciascuna risorsa

Due schedule S1 e S2 sono view-equivalenti (**S1** $\approx_V$ **S2**) se possiedono le *stesse relazioni* ***LEGGE_DA*** e le *stesse* ***SCRITTURE FINALI***.

VIEW-SERIALIZZABILITA'

Uno schedule S è view-serializzabile (VSR) se esiste uno schedule seriale S' tale che S' $\approx_V$ S.

ALGORITMO

Trovo tutti gli schedule seriali che posso generare considerando tutte le possibili permutazioni delle transazioni che compaiono in S, SENZA cambiare l'ordine delle operazioni di una transazione.

Per ognuno di questi calcolo gli insiemi LEGGE_DA e SCRITTURE_FINALI, se almeno uno schedule fra quelli seriali generati permutando le transazioni che compaiono in S ha gli insiemi LEGGE_DA e SCRITTURE_FINALI uguali a quelli di S allora vuol dire che ∃ uno schedule seriale S' tale che S' $\approx_v$ S e quindi S è uno schedule view-serializzabile e dunque è uno schedule accettabile di esecuzione concorrente di transazioni perché certamente non produrrà anomalie.

CONFLICT-EQUIVALENZA

CONFLITTO

Coppia di operazioni (a_i , a_j) che compaiono nello stesso *SCHEDULE* che:

- Appartengono a transazioni diverse
- Operano sulla stessa risorsa
- Almeno una delle due è un operazione di scrittura
- *a_i compare in S prima di a_j (confronti un'operazione sempre con le successive)*

Conflict equivalenza

- Due schedule S1 e S2 sono conflict equivalenti (S1 $\approx_c$ S2) se possiedono *le stesse **operazioni*** e gli *stessi **conflitti*** (= le operazioni in conflitto sono *nello stesso **ordine*** nei due schedule).

CONFLICT-SERIALIZZABILITA'

Conflict-serializzabilità

- Uno schedule S è conflict-serializzabile (CSR) se esiste uno schedule seriale S' tale che S' $\approx_c$ S.

ALGORITMO

Algoritmo (dato uno schedule S):

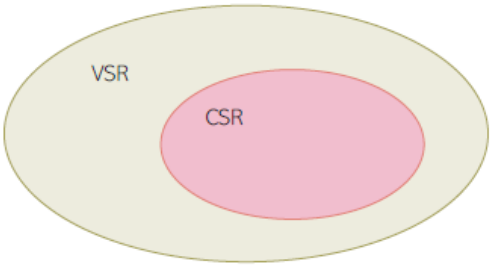
- Si costruisce il grafo dei conflitti G(N,A) dove :
    - N={t1, …, tn} con t1, …, tn transazioni di S;
    - (ti, tj) ϵ A se esiste almeno un conflitto (a_i, a_j) in S.
- Se il grafo così costruito è ACICLICO allora S è conflict-serializzabile.

RELAZIONE CSR E VRS e DIMOSTRAZIONE FORMALE DI TALE RELAZIONE

CSR vs VSR

Osservazioni

- La conflict-serializzabilità è condizione sufficiente ma non necessaria per la view-serializzabilità.



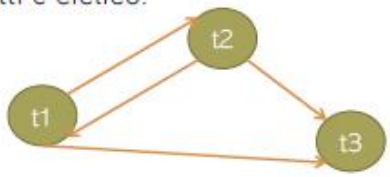
CSR ⊂ VSR

Controesempio per la non necessità:

- r1(x) w2(x) w1(x) w3(x) LEGGE_DA=∅ SCR_FIN={w3(x)}

- view-serializzabile: in quanto view-equivalente allo schedule seriale
  - r1(x)w1(x)w2(x)w3(x)LEGGE_DA= ∅ SCR_FIN={w3(x)}

- non conflict-serializzabile: in quanto il grafo dei conflitti è ciclico.



CSR ⇒ VSR

Supponiamo S1 ≈C S2 e dimostriamo che S1 ≈V S2:

- Osservazione: poiché S1 ≈C S2 i due schedule hanno:
  - stesse scritture finali: se così non fosse, ci sarebbero almeno due scritture sulla stessa risorsa in ordine diverso e poiché due scritture sono in conflitto i due schedule non sarebbero ≈C.
  - stessa relazione "legge_da": se così non fosse, ci sarebbero scritture in ordine diverso o coppie lettura-scrittura in ordine diverso e quindi, come sopra sarebbe violata la ≈C.

→ Si presenti la definizione di View-serializzabilità e la relazione tra l'insieme degli schedule VSR e l'insieme degli schedule CSR. Presentare la dimostrazione formale di tale relazione.

Definizione di View-serializzabilità (VSR): Uno schedule è view-serializzabile se è view-equivalente a uno schedule seriale. Due schedule sono view-equivalenti se:

1. Hanno le stesse scritture finali per ogni dato.
2. Se una transazione legge un valore iniziale in uno schedule, deve leggere lo stesso valore iniziale nell'altro schedule.
3. Se una transazione legge un valore scritto da un'altra transazione in uno schedule, deve leggere lo stesso valore scritto dalla stessa transazione nell'altro schedule.

Relazione tra VSR e CSR: L'insieme degli schedule conflict-serializzabili (CSR) è un sottoinsieme degli schedule view-serializzabili (VSR). Questo significa che ogni schedule CSR è anche VSR, ma non tutti gli schedule VSR sono CSR.

Dimostrazione della relazione CSR ⊆ VSR:

1. **Osservazione preliminare:** La conflict-serializzabilità implica la view-serializzabilità.
2. **Controesempio per la non necessità:** Consideriamo lo schedule:
   - r1(x) w2(x) w1(x) w3(x)
   - LEGGE_DA = ∅
   - SCR_FIN = {w3(x)}

   Questo schedule è view-serializzabile perché view-equivalente allo schedule seriale:

   - r1(x) w1(x) w2(x) w3(x)
   - LEGGE_DA = ∅
   - SCR_FIN = {w3(x)}

   Tuttavia, non è conflict-serializzabile perché il grafo dei conflitti è ciclico.

3. **Dimostrazione che CSR ⇒ VSR:** Supponiamo che S1 ≈C S2 e dimostriamo che S1 ≈V S2. Poiché S1 ≈C S2, i due schedule hanno:

- **Stesse scritture finali:** Se non fosse così, ci sarebbero almeno due scritture sulla stessa risorsa in ordine diverso, e quindi non sarebbero ≈C.
- **Stessa relazione "legge_da":** Se non fosse così, ci sarebbero scritture o coppie lettura-scrittura in ordine diverso, violando la ≈C.

Quindi, ogni schedule conflict-serializzabile è anche view-serializzabile, dimostrando che CSR ⊆ VSR.

→ Si presenti in dettaglio la definizione di view-equivalenza tra due schedule.

Definizione di view-equivalenza tra due schedule: Due schedule S1 e S2 sono view-equivalenti se soddisfano le seguenti condizioni:

1. **Scritture finali identiche:** Per ogni dato, la transazione che esegue l'ultima scrittura deve essere la stessa in entrambi gli schedule.
2. **Leggi iniziali identiche:** Se una transazione legge un valore iniziale in uno schedule, deve leggere lo stesso valore iniziale nell'altro schedule.
3. **Letture da scritture identiche:** Se una transazione legge un valore scritto da un'altra transazione in uno schedule, deve leggere lo stesso valore scritto dalla stessa transazione nell'altro schedule.

→ Si presenti in dettaglio la definizione di conflict-equivalenza tra due schedule.

Definizione di conflict-equivalenza tra due schedule: Due schedule S1 e S2 sono conflict-equivalenti se:

1. Eseguono le stesse operazioni di lettura e scrittura.
2. Per ogni coppia di operazioni in conflitto (stesse risorse coinvolte, almeno una scrittura), l'ordine relativo delle operazioni in S1 è lo stesso in S2.

→ Si presenti in dettaglio la definizione di Conflict-Serializzabilità (CSR).

Definizione di Conflict-Serializzabilità (CSR): Uno schedule è conflict-serializzabile se è conflict-equivalente a uno schedule seriale. Questo significa che è possibile riordinare le operazioni nello schedule senza alterare l'ordine relativo delle operazioni in conflitto, rendendolo equivalente a uno schedule in cui le transazioni sono eseguite una dopo l'altra, senza interleaving.

TECNICHE PER VERIFICARE LA SERIALIZZABILITA' DI UNO SCHEDULE SENZA L'IPOTESI DI COMMIT-PROIEZIONE

CONTESTO:

Gli SCHEDULE SERIALIZZABILI sono EQUIVALENTI a quelli SERIALI, ovvero non danno ANOMALIE nell'esecuzione CONCORRENTE di operazioni di diverse transazioni che interagiscono contemporaneamente con la stessa base di dati.

Le tecniche finora viste per verificare la serializzabilità di uno schedule sono:

- La view-serializzabilità
- La conflict-serializzabilità

Queste tecniche richiedono l'ipotesi di commit-proiezione.

Le tecniche che non richiedono di conoscere l'esito delle transazioni:

- Timestamp con scritture bufferizzate
- Locking a due fasi stretto

L'ultimo è il metodo applicato nei sistemi reali per la gestione dell'esecuzione concorrente delle transazioni.

LOCKING A DUE FASI:

Caratteristiche:

Meccanismo di base per la gestione dei lock:

Si basa sull'introduzione di alcune primitive di lock che consentono alle transazioni di bloccare le risorse sulle quali vogliono agire con operazioni di lettura e di scrittura.

La politica di concessione dei lock sulle risorse:

Il gestore dei lock (della concorrenza) mantiene per ogni risorsa le seguenti informazioni:

- Stato: $s(x) \in \{libero, r\_lock, w\_lock\}$
- Transazioni in r_lock : $c:(x) = \{t1, \dots , tn\}$

Per la lettura il lock è condiviso: più transazioni possono contemporaneamente accedere alla stessa risorsa in lettura.

Per la scrittura il lock è condiviso: non è ammesso che più transazione contemporaneamente accedono alla stessa risorsa in scrittura.

Regola che garantisce la serializzabilità

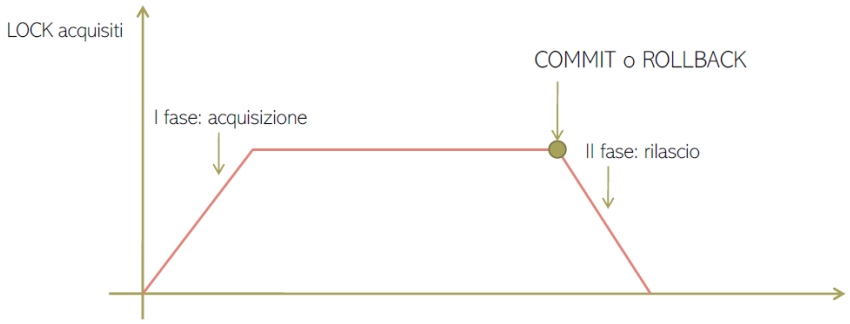
La regola che garantisce la serializzabilità è che una transazione dopo aver rilasciato un Lock non può acquisirne altri.

Ogni transazione ha una fase di acquisizione e una fase di rilascio delle risorse e una volta che una transazione ha iniziato a rilasciare lock non può più acquisirne.

Quando il 2PL è STRETTO significa che la transazione deve mantenere tutti i lock fino a quando non esegue correttamente un commit o un rollback.

Questo aggiunge ulteriore sicurezza perché evita che altre transazioni vedano i dati parzialmente aggiornati da una transazione non ancora confermata.

(transazione in 2pl)



(transazione in 2pl stretto)

2PL A CONFRONTO CON TECNICHE PRECEDENTI

DIMOSTRAZIONE: $2PL \subset CSR$

1) $\forall s \in 2PL : s \in CSR$
2) $\exists s \in CSR \mid s \notin 2PL$

---

1) $\forall s \in 2PL : s \in CSR$

DIMOSTRAZIONE PER ASSURDO:

Ipotizziamo che $\exists s \in 2PL \mid s \notin CSR$

Allora se $s \notin CSR$ esisterà una catena di conflitti ciclici.

Ma uno schedule che è 2PL avrà invece necessariamente un grafo di conflitti che è aciclico in quanto per costruzione non ci possono essere cicli di dipendenze tra le transazioni.

Si arriva a una contraddizione pertanto si può concludere che $\nexists s \in 2PL \mid s \notin CSR$, quindi vale che $\forall s \in 2PL : s \in CSR$.

2) $\exists s \in CSR \mid s \notin 2PL$

DIMOSTRAZIONE CON UN CONTROESEMPIO:

Consideriamo lo schedule s definito dalle seguenti operazioni:

- r1(x)

- w1(x)

- r2(x)

- w2(x)

- r3(y)

- w1(y)

Questo schedule è CSR poiché il grafo dei conflitti risultante non contiene cicli.

Tuttavia, non è 2PL poiché la transazione t1 rilascia il lock su x prima di acquisire il lock su y, violando le regole del protocollo 2PL.

Pertanto, abbiamo dimostrato che può esistere uno schedule che è CSR ma non 2PL.

Di conseguenza si può affermare che $\exists s \in CSR \mid s \notin 2PL$.

Avendo dimostrato formalmente i punti (1) e (2) ho dimostrato che $2PL \subset CSR$.

BLOCCO CRITICO

Un blocco critico è una situazione di stallo che si verifica nell'esecuzione di transazioni concorrenti quando un gruppo di due o più transazioni hanno bloccato delle risorse in modo tale che ciascuna transazione del gruppo attenda una risorsa che è bloccata da un'altra transazione dello stesso gruppo.

TECNICHE PER RISOLVERE IL BLOCCO CRITICO

- Timeout
  - una transazione in attesa su una risorsa trascorso il timeout viene abortita.
- Prevenzione
  - Una transazione blocca tutte le risorse a cui deve accedere in lettura o scrittura in una sola volta (o blocca tutto o non blocca nulla)
  - Tecnica Basata sui Timestamp: nei blocchi critici, la transazione piu' vecchia, quindi con timestamp piu' piccolo, e' quella che ha la priorita' e che ottiene le risorse che le servono per concludere
- Rilevamento
  - analisi periodica della tabella di LOCK per rilevare la presenza di un blocco critico.

STARVATION:

Se una risorsa x viene costantemente bloccata da una sequenza di transazioni che acquisiscono r_lock su x, un’eventuale transazione in attesa di scrivere su x viene bloccata per un lungo periodo fino alla fine della sequenza di letture.

TECNICA PER RISOLVERE LA STARVATION

- analizzare la tabella delle relazioni di attesa
- verificare da quanto tempo le transazioni stanno attendendo la risorsa
- sospendere temporaneamente la concessione di lock in lettura condivisi per consentire la scrittura da parte della transazione in attesa.

GESTIONE CONCORRENZA SQL:

Per aumentare le prestazioni è possibile rinunciare in tutto o in parte al controllo della concorrenza per aumentare la prestazione del sistema, significa che si decide di tollerare alcune anomalie a livello di transazione concorrente.

SQL mette a disposizione diversi LIVELLI DI ISOLAMENTO per gestire la concorrenza tra transazioni:

1. **Serializable:**
   - **Prevenzione:** Perdita di update, lettura sporca, lettura inconsistente, update fantasma, inserimento fantasma.
   - **Note:** Richiede 2PL (Two-Phase Locking) stretto per le scritture e le letture. Applica il lock di predicato per evitare l'anomalia di inserimento fantasma.
2. **Repeatable read:**
   - **Prevenzione:** Perdita di update, lettura sporca, lettura inconsistente.
   - **Note:** Applica 2PL stretto per tutti i lock in lettura a livello di tupla e non di tabella. Permette inserimenti e quindi non evita l'anomalia di inserimento fantasma.
3. **Read committed:**
   - **Prevenzione:** Lettura sporca, lettura inconsistente.
   - **Note:** Richiede lock condivisi per le letture ma non il 2PL stretto.
4. **Read uncommitted:**
   - **Prevenzione:** Nessuna delle anomalie.
   - **Note:** Non applica lock in lettura.

Questi livelli di isolamento permettono di bilanciare il compromesso tra l'integrità dei dati e le prestazioni del sistema. Ciascun livello offre un diverso grado di protezione contro le anomalie che possono verificarsi durante l'esecuzione concorrente delle transazioni.

OTTIMIZZAZIONE

Contesto:

Ogni interrogazione eseguita su un Database Management System (DBMS) è formulata in un linguaggio dichiarativo come SQL.

È quindi essenziale tradurre questa interrogazione in un linguaggio procedurale, come l'algebra relazionale, per generare un piano di esecuzione efficiente *(ovvero **ottimizzato**)*.

Questo processo coinvolge diverse fasi:

- l'analisi lessicale
- l'analisi sintattica
- l'ottimizzazione algebrica indipendente dal modello di costo
- l'ottimizzazione basata sui costi di esecuzione

Analisi Lessicale e Sintattica

Prima di eseguire un'interrogazione, il DBMS analizza la sua struttura linguistica e sintattica per comprendere le operazioni richieste.

Ottimizzazione Algebrica

L'ottimizzazione algebrica si basa sulle regole di ottimizzazione ben note dell'algebra relazionale:

- Anticipo delle selezioni
- Anticipo delle proiezioni

Ottimizzazione Dipendente dai Metodi di Accesso

Il DBMS supporta diverse operazioni di accesso ai dati, che devono essere considerate durante l'ottimizzazione del piano di esecuzione.

Operazioni di Accesso Supportate:

- Scansione delle tuple di una relazione
- Ordinamento di un insieme di tuple
- Accesso diretto alle tuple attraverso un indice
- Diverse implementazioni del join

Scansione

L'operazione di scansione legge le tuple di una relazione, potenzialmente eseguendo altre operazioni contemporaneamente, come la proiezione o la selezione.

Varianti possibili:

- Scan + proiezione senza eliminazione di duplicati
- Scan + selezione in base ad un predicato semplice
- Scan + inserimento/cancellazione/modifica

Costo di una scansione sulla relazione R: NP(R)

NP(R) = Numero Pagine dati della relazione R

Ordinamento

L'ordinamento è utile per organizzare i risultati di un'interrogazione, eliminare duplicati o raggruppare tuple. Viene eseguito attraverso tecniche come lo Z-way Sort-Merge, che coinvolge la suddivisione e la fusione di run di tuple.

L’ordinamento viene utilizzato per:

- Ordinare il risultato di un’interrogazione (clausola *order by*)
- Eliminare duplicati (*select distinct*)
- Raggruppare tuple (*group by*)

Z-way Sort-Merge

Lo Z-way Sort-Merge è un algoritmo di ordinamento utilizzato nei database per gestire grandi quantità di dati. Esso coinvolge una tecnica di suddivisione e fusione delle tuple per ordinare i risultati di un'interrogazione.

Descrizione dello Z-way Sort-Merge:

1. **Fase di Sort Interno**:
    - Le tuple vengono divise in gruppi più piccoli chiamati "run".
    - Ogni run è ordinato individualmente utilizzando un algoritmo di sort interno, come il QuickSort.
    - Ogni run ordinato viene scritto su memoria secondaria in un file temporaneo.
2. **Fase di Merge**:
    - I run ordinati vengono uniti attraverso uno o più passaggi di fusione.
    - Durante ogni passaggio di fusione, due run vengono letti contemporaneamente e mescolati in un unico run ordinato.
    - Questo processo continua fino a quando non si ottiene un'unica run contenente tutte le tuple ordinate.

Costo dello Z-way Sort-Merge:

Considerando come costo solo numero di accessi alla memoria secondaria, nel caso base Z = 2 e con NB =3

- Durante il sort interno, si leggono e si riscrivono NP pagine
- Durante ogni passaggio di merge, si leggono e si riscrivono NP pagine
- Il numero complessivo di passaggi di fusione è pari a $\lceil \log_2 (NP) \rceil$ in quanto ad ogni passo il numero di run si dimezza (Z = 2)

- Il costo complessivo è pertanto pari a: $2 * NP * \lceil \log_2 (NP) \rceil$

Accesso Diretto via Indice

Le interrogazioni possono fare uso degli indici per migliorare le prestazionI:

- Selezioni con condizione atomica di uguaglianza → richiede indice hash o B+-tree.

- Selezioni con condizione di range → richiede indice B+-tree.

- Selezioni con condizione costituita da una congiunzione di condizioni di uguaglianza → in questo caso si sceglie per quale delle condizioni di uguaglianza utilizzare l'indice; la scelta ricade sulla condizione più selettiva. L'altra si verifica direttamente sulle pagine dati.

- Selezioni con condizione costituita da una disgiunzione di condizioni di uguaglianza → in questo caso è possibile utilizzare più indici in parallelo, facendo un merge dei risultati eliminando i duplicati oppure, se manca anche solo uno degli indici, è necessario eseguire una scansione sequenziale.

Algoritmi per il Join nei DBMS

Il join è una delle operazioni più gravose per un sistema di gestione di basi di dati (DBMS).

Le implementazioni più diffuse si basano su diversi operatori fisici, ciascuno con caratteristiche e prestazioni differenti.

Le implementazioni più diffuse si riconducono ai seguenti operatori fisici:

- Nested Loop Join
- Merge Scan Join
- Hash-based Join

Nested Loop Join

Descrizione

L'algoritmo Nested Loop Join opera come segue:

```
Per ogni tupla tR della relazione esterna R{

      Per ogni tupla tS della relazione interna S{

            Se la coppia (tR,tS)(tR,tS) soddisfa il predicato di join PJ:

                  Aggiungi (tR,tS)(tR,tS) al risultato

      }

}
```

Costo

- Dipende dallo spazio nei buffer.
- Caso base con 1 buffer per R e 1 buffer per S:
  - Bisogna leggere 1 volta R.
  - NR(R) volte S, dove NR(R) è il numero di tuple di R.
  - Costo totale: NP(R) + NR(R) * NP(S) accessi a memoria secondaria.
- Con NP(S) buffer per la relazione interna:
  - Costo ridotto a NP(R)+NP(S) .

Variante con Indice B+-tree

- Sostituisce la scansione completa della relazione interna S con una scansione basata su un indice sugli attributi di join.
- Costo: NP(R) + NR(R) * (ProfIndice + NR(S) / VAL(A,S))
- dove VAL(A,S) è il numero di valori distinti dell'attributo A in S.

Block Nested Loop Join

- Bufferizza le righe lette della tabella esterna per ridurre il numero di letture nel ciclo interno.
- Caso base con 1 buffer per R e 1 buffer per S:
  - Costo totale: NP(R) + NP(R) * NP(S).
- Con NP(S) buffer per la relazione interna:
  - Costo ridotto a NP(R)+NP(S)

Merge-Scan Join

Descrizione

- Applicabile quando entrambe le relazioni in input sono ordinate sugli attributi di join ed il join è un equi-join.
- Condizioni per l'applicabilità:
  - Relazioni fisicamente ordinate sugli attributi di join.
  - Esistenza di un indice sugli attributi di join che consente una scansione ordinata delle tuple.

Costo

- Numero di letture totali: NP(R)+NP(S) se si accede sequenzialmente alle relazioni ordinate.
- Con indici B+-tree su entrambe le relazioni: costo massimo NR(R)+NR(S).
- Con una relazione ordinata e un solo indice: costo massimo NP(R)+NR(S) (o viceversa).

Hash-Based Join

Descrizione

- Applicabile solo in caso di equi-join.
- Non richiede né la presenza di indici né input ordinati, risultando vantaggioso per relazioni molto grandi.
- Utilizza una funzione hash h sugli attributi di join delle due relazioni R e S.

Funzionamento

- Se tR e tS sono tuple di R e S:
  - tR.J = tS.J solo se h(tR.J)=h(tS.J)
  - Se h(tR.J)≠h(tS.J) allora tR.J≠tS.J
- Le relazioni R e S vengono partizionate sulla base dei valori della funzione di hash h, con la ricerca di matching tuples solo tra partizioni con lo stesso valore di h.

Costo

- Costruzione hash map: NP(R)+NP(S).
- Accesso alle matching tuples (caso pessimo): NP(R)*NP(S).
- Dipende dal numero di buffer disponibili e dalla distribuzione dei valori degli attributi di join.

Scelta Finale del Piano di Esecuzione

Generazione dei Piani

- Si generano tutti i possibili piani di esecuzione alternativi (o un sottoinsieme) considerando:
  - Operatori alternativi per l'accesso ai dati (scan sequenziale o via indice).
  - Operatori alternativi nei nodi (nested-loop join o hash-based join).
  - Ordine delle operazioni (associatività).

Valutazione dei Costi

- Si valuta con formule approssimate il costo di ogni alternativa in termini di accessi a memoria secondaria richiesti (stima).
- Si sceglie l'albero con costo approssimato minimo.

Profilo delle Relazioni

- Stime memorizzate nel Data Dictionary per ogni relazione T:
  - Cardinalità (#tuple): CARD(T).
  - Dimensione di una tupla: SIZE(T).
  - Dimensione di ciascun attributo A: SIZE(A,T).
  - Numero di valori distinti per ogni attributo A: VAL(A,T).
  - Valore massimo e minimo per ogni attributo A: MAX(A,T) e MIN(A,T).