A.A. 2020/21

BASI DI DATI - TECNOLOGIE

SARA MIGLIORINI

FABS :)

NOTA

Questi appunti/sbobinatura/versione “discorsiva” delle slides sono per mia utilità personale, quindi pur avendole revisionate potrebbero essere ancora presenti typos, commenti/aggiunte personali (che anzi, lascio di proposito) e nel caso peggiore qualche inesattezza!

Comunque spero siano utili! 

Questa sbobina fa parte della mia collezione di sbobinature,
che è disponibile (e modificabile!) insieme ad altre in questa repo:
https://github.com/fabfabretti/sbobinamento-seriale-uniVR

Indice

FONDAMENTI DEL DBMS **6**

ARCHITETTURA DI UN DBMS **8**

DBMS E MEMORIA SECONDARIA **10**

RAPPRESENTAZIONE DI UNA TABELLA A LIVELLO FISICO **17**

ESECUZIONE CONCORRENTE DI TRANSIZIONI **30**

GESTIONE DELLE ANOMALIE **33**

OTTIMIZZAZIONI **44**

INTERAZIONE TRA BASI DI DATI E APPLICAZIONI **51**

Fondamenti del DBMS

DBMS

Un DBMS gestisce una collezione di dati in memoria secondaria. Queste collezioni di dati sono

- Grandi
- Condivise
- Persistenti

e bisogna assicurare:

- **Affidabilità**: sicurezza che i dati rimangano salvati e siano sempre 5accessibili
- **Privatezza**: permettere di accedere solo ad alcuni dati per ogni utente
- **Accesso efficiente**: permettere di recuperare velocemente i dati.

Un DBMS basato sul modello relazionale è solitamente definito un **sistema transazionale**, ovvero ha dei meccanismi che permettono di eseguire delle transazioni. L'interazione che abbiamo verso una base di dati di questo tipo avviene attraverso il costrutto fondamentale definito transazioni.

Transazione

> Una transazione è un'unità di lavoro acronica che viene svolta dal programma applicativo che interagisce con al base di dati.

Garantisce il buon funzionamento del DBMS, in quanto se l'applicazione viene bloccata a metà transazione (abort prematuro), tutto ciò che è già stato eseguito deve essere annullato

→ **Codifica "tutto o niente"**: no esecuzioni parziali.

In SQL, una transazione è un'unità di lavoro che può contenere al suo interno più istruzioni. Per delimitarla uso le istruzioni **begin transaction** e **end transaction**. NB: Se l'SQL è già atomico posso non mettere begin/end transaction.

- **commit work**→ confermo e termino la transazione con successo
- **rollback work**→ torno indietro e disfo tutto quello che ho fatto.

La transazione è ben formata se:

- Inizia con il comando **begin transaction**
- Termina con l'istruzione **end transaction**
- Deve esserci un **commit** o un **rollback**, e devono essere **l'ultima istruzione** prima dell'end transaction.

Il caso tipico per spiegare la necessità di transazioni è il caso in cui si vuole fare un trasferimento di fondi fra due conti corrente: chiaramente non voglio che, annullando a metà, dei soldi finiscano nel limbo!

Un DBMS deve garantire che ogni transazione abbia queste quattro proprietà:

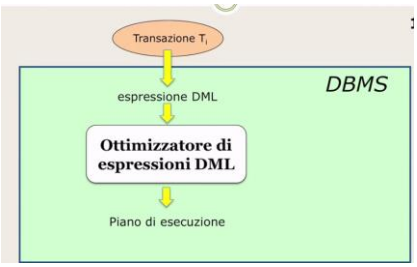
Proprietà ACID(e)

- **Atomicità** : deve essere **indivisibile** (eseguita tutta o niente)
  → se una transazione viene interrotta prima del commit, il lavoro svolto fino a quel momento viene disfatto e si ripristina la situazione iniziale.
  → Se una transazione arriva al commit devo garantire che tutte le operazioni siano svolte sulla base di dati.
- **Consistenza** : L'esecuzione di una transizione **non deve violare i vincoli di integrità**.
  La verifica può essere:
  - **Immediata**: abortita l'ultima operazione e il sistema restituisce un errore; l'applicazione può reagire alla violazione
  - **Differita al commit**: se un vincolo di integrità viene violato la transazione viene abortita senza possibilità di reagire alla violazione.
- **Isolamento** : l'esecizione di una transazione deve essere indipendente dalla contemporanea esecuzione di altre operazioni.
  Implicazioni:
  - Il sistema deve regolare l'esecuzione concorrente
  - Il rollback non deve creare rollback a catena di altre transazioni concorrenti.
- **Durability** / Persistenza→ **L'effetto** di una transazione che ha eseguito il commit **non deve andare perso.**

→ Non devo solo registrare, ma anche controllare la propedeuticità.

Architettura di un DBMS

È fatto da diversi moduli; ciascun modulo supporta le varie funzionalità. Per ciascun modulo presentiamo le funzionalità e e tipologie.



Ottimizzatore di espressioni DML

Passiamo un'espressione in **Data Manipulation Language**, e il sistema la **trasforma in instruzioni vere e proprie.**

Queste però non sono eseguite proprio come le abbiamo specificate: esiste una parte del DBMS che **ottimizza la query** e traduce l'espressione SQL (dichiarativa) in un **piano di esecuzione**.

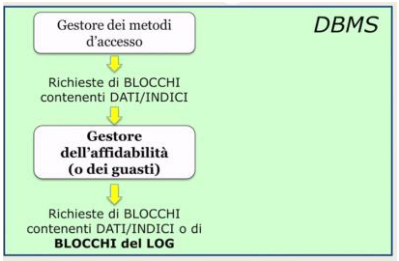
In postgres, possiamo vedere il piano di esecuzione (join, where … in ordine) con il comando **explain**.

Gestore dei metodi d'accesso

Il piano d'esecuzione è dato a un gestore dei metodi di accesso, che **recupera i dati della memoria secondaria**.

Siccome essi sono in **pagine**, mi servirà un oggetto che mi descriva le pagine o gli indici dove trovo i dati.

Il gestore dei metodi di accesso lavora in maniera stretta con il **gestore dell'esecuzione concorrente**: il gestore dei metodi di accesso ha bisogno di richiedere al gestore di esecuzione concorrente il permesso.

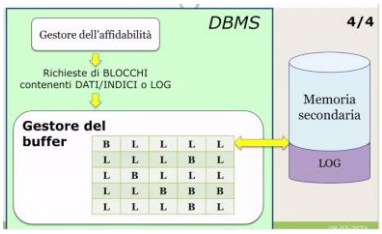


Gestore dell'affidabilità / dei guasti

Il gestore dei guasti garantisce due delle 4 proprietà fondamentali: **atomicità** e **persistenza**.

Per fare questo, il gestore dell'affidabilità si avvale del **LOG.**

> **LOG:** archivio persistente che registra tutte le operazioni svolte sulla base di dati. Questo permette di:
>
> Tornare indietro nel caso di un **rollback**
>
> **Garantire l'affidabilità**: permette anche di ricostruire il contenuto della DB prima di un guasto (dato che tutte le operazioni che sarebbero andate perse possono essere rifatte).



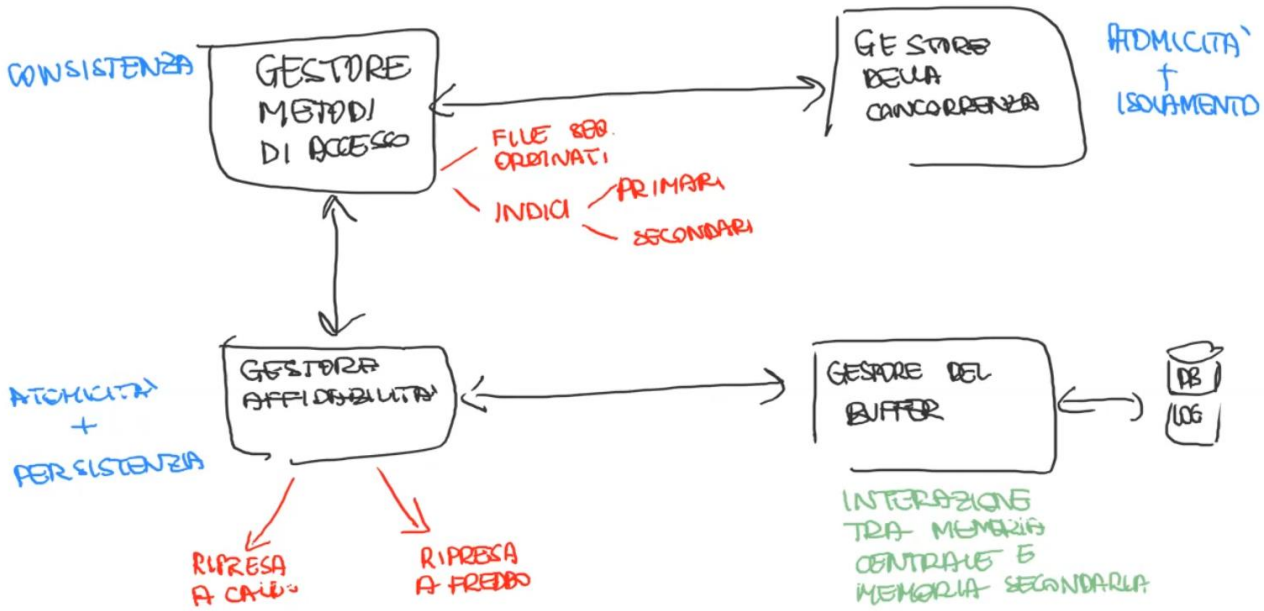
Gestore del buffer

Dal punto di vista fisico, i dati sono memorizzati nel blocchi e con il gestore del buffer si garantisce di poter recuperare i dati salvati nella memoria secondaria.

Approfondito più avanti.

Quali moduli contribuiscono a garantire le proprietà?

	Atomicità	*Consistenza*	*Isolamento*	*Durability*
Gestore dei **metodi di accesso**		x		
Gestore dell'**esecuzione concorrente**	x		x	
Gestore dell'**affidabilità**	x			x



DBMS e memoria secondaria

I dati in memoria secondaria sono memorizzati in modo leggermente controintuitivo: non posso recuperare ***il dato***, ma recupero l'intero blocco da cui seleziono il dato.

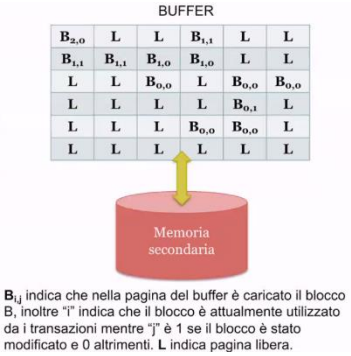
Inoltre, il costo di recupero del blocco e del dato è ordini di grandezza superiore a quello della memoria centrale.

→ Ho bisogno di un componente centrale chiamato **BUFFER**, che **gestisca l'interazione tra la memoria secondaria e la memoria centrale.**

Buffer

> Il buffer è una parte della memoria centrale in cui vengono caricati i dati presenti nella memoria secondaria.

È **condiviso** fra tante applicazioni: questo permette di condividere i dati fra più applicazioni, dunque permette di **minimizzare gli accessi alla memoria secondaria** (=accedo una volta per tutti).



$B_{i,j}$ indica che nella pagina del buffer è caricato il blocco B, inoltre "i" indica che il blocco è attualmente utilizzato da i transazioni mentre "j" è 1 se il blocco è stato modificato e 0 altrimenti. **L** indica pagina libera.

I dati rimangono nel buffer finché possibile, per evitare di doverli recuperare ogni volta. Quando esso si riempie, si decide cosa mantenere secondo le gestioni di **politica del buffer**, che sono strategiche per avere buone prestazioni. L'ideale è **mantenere sempre in memoria i dati a cui accedo spesso**. Una strategia è condividere fra più iterazioni lo stesso dato.

Il buffer è organizzato in **pagine**: le pagine del buffer avranno la **stessa dimensione** delle pagine in memoria secondaria. Questo permette di avere una corrispondenza facile fra le due ^-^

Gestore del buffer

Il buffer deve anche gestire le politiche di accesso in read/write ai dati del buffer. Per esempio:

Lettura	Scrittura
• Se il blocco è presente allora non si esegue la lettura su memoria secondaria e si restituisce un puntatore alla pagina buffer. Questo rende le letture successive molto più veloci! ○ Quali pagine rimangono su? Similarmente alla cache, cercherò una politica che mi permetta di fare quasi sempre buffer-hit. • Altrimenti cerco una pagina libera e carico il blocco	• Posso scrivere e trasferire la modifica immediatamente • Posso differire la memoria secondaria, a patto di garaantire persistenza e consistenza.

Principi di gestione del buffer

- Il gestore del buffer utilizza **politiche sulla località**: i dati referenziati di recente hanno maggiori probabilità di essere referenziati in futuro. Dunque, se il buffer ha bisogno di liberare spazio, cancella i dati a cui non accedo da più tempo.
- Inoltre, si usa anche il **principio empirico secondo cui il 20% dei dati è acceduto dall'80%** delle applicazioni. Quindi quel 20% ha senso che stia sempre nel buffer.

Questo non solo ci dà un incipit su come gestire il buffer, ma ci dà anche un'altra proprietà che è quella di poter **dilazionare la scrittura** su memoria secondaria delle pagine del buffer.

Memorizzazione di un blocco in memoria centrale

Quando carico una pagina devo memorizzare un identificatore con segnato:

- **Nome del file**
- **Numero del blocco** (= offset) **nella memoria fisica**.
- **Variabili di stato**, fra cui
    - Contatore **I** che conta il **numero di transazioni** che utilizzano le pagine
    - Bit di stato **J** che indica se la pagina è stata **modificata (1) oppure no (0)**.

Primitive

fix: Viene utilizzata dalle transazioni per **richiedere l'accesso** ad un blocco. Viene restituito un puntatore alla pagina contenente il blocco richiesto.

Data una richiesta, vengono eseguite una serie di operazioni:

- Cerco nel buffer se la pagina è già presente o meno.
  → Se sì, restituisco il puntatore nel buffer.
- Devo scegliere una pagina libera (= contatore I = 0) nel buffer ove salvare la nuova pagina.
    - Se **ho una pagina vuota**, la carico e imposto il contatore i a 0.
    - Se **non ho pagine vuote**, scelgo una pagina libera da sostituire secondo. Scelta la pagina **verifico il bit di stato**;
        - Se è 1, devo **salvarla** in memoria secondaria con la primitiva **flush**.
- Se non ho nemmeno pagine libere:
    - Politica **STEAL**: ruba una pagina ad un'altra transazione applicando la flush
    - Politica **NO STEAL**: sospende la transazione inserendola in una coda di attesa e fa la flush solo dopo che una pagina è diventata lbera (=contatore I = 0)

setDirty: Serve a indicare che una transazione ha modificato il blocco.

unfix: Serve a indicare che una transazione ha finito di usare un blocco. Il risultato è che il contatore I viene decrementato di 1.

force: Si usa per forzare il salvataggio in memoria secondaria del blocco in maniera **sincrona**. Potrebbe essere chiamata dal gestore dell'affidabilità, che in certe situazioni ha bisogno che le modifiche siano salvate subito. → forza il salvataggio sincrono: quando scrivo la modifica la salvo subito in memoria secondaria

flush: è utilizzata dal gestore del buffer per salvare i blocchi in maniera **asincrona**. Questo consente di liberare funzioni dirty. → salvataggio asincrono; mi serve prima di liberare la pagina dal buffer (quindi iene fatta ***dopo*** eventuali modifiche.

Gestore dell'affidabilità

È il modulo responsabile di

- Esecuzione delle istruzioni → **atomicità**
- Ripristino della base di dati in caso di malfunzionamenti. → **persistenza**

Ha bisogno di una **memoria stabile**, ovvero una memoria **resistente ai guasti**. La memoria stabile è utilizzata per memorizzare i file di **LOG**, ovvero il registro di tutte le operazioni eseguite.

Azione di COMMIT

Una transazione T sceglie in modo atomico l'esito del COMMIT nel momento in cui scrive nel file di LOG in modo sincrono (=force) il suo recordo di COMMIT -C(T)

Ovvero: una transazione ha completato soltatno quando nel file di log ci ritroviamo scritto COMMIT.

Regole di scrittura sul log

- **Regola WAL (write ahead log)**: I record di LOG sono scritti sul log **prima di eseguirle**.
  → garantisce di poter **sempre fare UNDO**
- **ReGola commit-precedenza**: i log sono scritti **prima dell'esecuzione del commit**
  → Garantisce di poter **sempre fare REDO**

Tipologie di record

Record di TRANSAZIONE

- **Begin** della transazione: **record B(T)**
- **Commit** della transazione: **record C(T)**
- **Abort** della transazione: **record A(T)**
- **Insert**: **I(T,O,AS)** con AS = after state
- **Delete**: **D(T,O,BS)** con BS = before state
- **Update** : **U(T,O,BS,AS)**

Questi record permettono di eseguire un ripristino delle operazioni in caso di guasto.

UNDO: per disfare un'azione fatta su un oggetto O, devo recuperare il suo stato precedente (BS) → es. per la delete inserisco O, per la insert cancello O **IDEMPOTENZA**: UNDO(UNDO(A)) = UNDO(A)	**REDO**: rifaccio l'azione, e mi basta copiare il suo AS. **IDEMPOTENZA**: REDO(REDO(A)) = REDO(A)

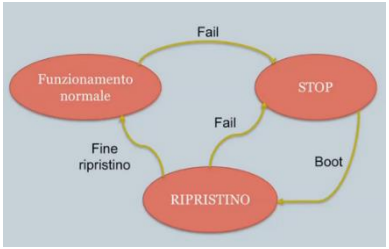
Record di SISTEMA

Queste operazioni sono **svolte in maniera periodica** dal gestore di affidabilità, e permettono di monitorare il sistema.

- **DUMP**: record di **DUMP**
- **CheckPoint**: **record CK(T_1...T_n)** indica che all'esecuzione del CheckPoint le transazioni attive erano T_1...T_n.
  - Sospensione delle operazioni di scrittura, commit e abort
  - Esecuzione della primitiva force sulle pagine dirty di transazioni che hanno eseguito il commit

- Scrittura sincrona (force) sul file di log del record di checkpoint, con gli identificatori delle transazioni attive
- Ripresa delle operazioni.

Guasto



Per decidere che cosa fare si guarda al tipo di sistema.

1. **Guasto di sistema**: *perdita del contenuto della memoria centrale* → **Ripresa a caldo**; non serve fermare tutto il sistema, basta ricaricarlo
2. **Guasto di dispositivo**: *perdita di dati dalla memoria secondaria* → **Ripresa a freddo**: fermo tutte le operazioni e erecumpero il contenuto attraverso il file di log

Ripresa a caldo

1. Accedo al log e **ripercorro fino al più recente checkpoint**. I check sono momenti in cui garantisco che tutto quello che ho fatto è salvato!
2. Si decidono le trasnsizioni da rifare e disfare **inizializzando l'insieme UNDO con le transizioni attive al checkpoint**, e l'insieme **REDO con l'insieme vuoto**.
3. Percorro in avanti il LOG:
   a. **B(T)** → aggiungo T in **UNDO(T)**
   b. **C(T)** → T **da UNDO a REDO**

   Ovvero, se un'operazione è arrivata al commit prima del guasto, la rifaccio; se un'operazione NON era arrivata al commit prima del guasto, disfo i pezzi che avevo già fatto.
4. Per eseguire gli UNDO, **ripercorro il LOG all'indietro** fino alla transazione più vecchia.
5. **Eseguo le transazion in REDO**.

Ripresa a freddo

È stata compromessa anche la memoria secondaria, quindi non posso ripartire da essa! Dovrò partira da un DUMP della base di dati, e posso ricopiare in maniera selettiva la parte compromessa.

1. Accedo **all'ultimo DUMP**
2. **Riprecorro in avanti il LOG**, rieseguendo tutte le operazioni relative alla parte deteriorata
3. **Applico la ripresa a caldo**.

Esempi di ripresa a caldo

Esempio 1

B(T1), B(T2), U(T2,O1,B1,A1), I(T1,O2,A2), B(T3), C(T1), B(T4), U(T3,O2,B3,A3), U(T4,O3,B4,A4), CK(T2,T3,T4), C(T4), B(T5), U(T3,O3,B5,A5), U(T5,O4,B6,A6), D(T3,O5,B7), A(T3), C(T5), I(T2,O6,A8) guasto

- B(T1)→ Inizio transazione T1
- B(T2) → Inizio transazione T2
- U(T2,O1,B1,A1) → Update transazione T2 con l'oggetto O1; salviamo anche stato before e after
- I(T1,O2,A2) → Inserimento in transizione T1 dell'oggetto O2; salviamo anche lo stato after
- …
- CK(T2,T3,T4) → specifichiamo solo le transazioni attive; T1 è terminata perché ha fatto il commi
- …
- D(T3,O5,B7) → eliminazione in transizione T3 dell'oggetto O5; salvo anche lo stato before
- A(T3) → aborto la transizione T3. Essa era fatta da due update; essi non vanno a buon fine

Passo 1: risalgo fino all'ultimo checkpoint

Passo 2: inizializzo UNDO e REDO

UNDO = {T2,T3,T4}

REDO = {}

Passo 3: ripercorro in avanti il file di log;
Per ciascun B aggiungo la corrispondente transazione in UNDO. Per ciascun C metto la transazione da UNDO a REDO

- Sposto T4 in REDO
- Metto T5 in UNDO
- Sposto T5 in REDO

→ Quindi arrivo con **UNDO = {T2,T3}** e **REDO = {T4, T5}**

Passo 4: ripercorro in indietro il file di LOG per disfare le operazioni di UNDO

Undo = {T2,T3} → eseguo l'operazione opposta per ogni operazione di T2 o T3

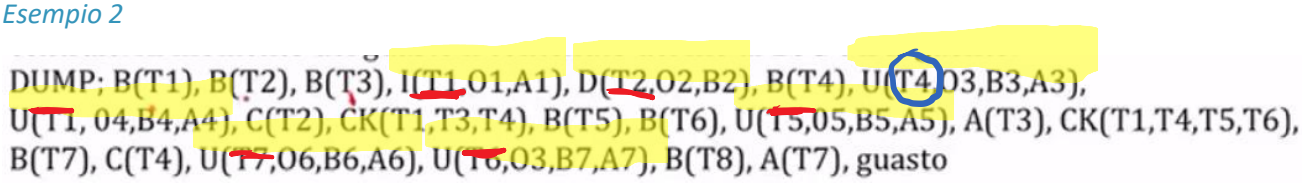
- I(T2,O6,A8) → **Delete(O6)**
- D(T3,O5,B7) → **Insert(O5)**
- U(T3,O3,B5,A5) → **O3 := B5 (**torno nello stato before)
- U(T3,O2,B3,A3) → **O2 := B3**
- U(T2,O1,B1,A1) → **O1 := B1**

Passo 5: ripercorro il LOG in avanti per rifare le operazioni di REDO.

REDO = {T4, T5}

- U(T4,O3,B4,A4) → **O3 := A4**
- U(T5,O4,B6,A6) → **O4 := A6**

Esempio 2



Passo 1: risalgo fino all'ultimo checkpoint

Passo 2: inizializzo UNDO e REDO

- UNDO = {T1,T4,T5,T6}
- REDO = {}

Passo 3: ripercorro in avanti il file di log;
Per ciascun B aggiungo la corrispondente transazione in UNDO. Per ciascun C metto la transazione da UNDO a REDO

- Metto T7 in UNDO
- Sposto T4 in REDO
- Metto T8 in UNDO

UNDO = {T1,T5,T6,T7,T8}

REDO = {T4}

Passo 4: ripercorro in indietro il file di LOG per disfare le operazioni di UNDO

UNDO = {T1,T5,T6,T7,T8}

- O3 := B7
- O6 := B6
- O5 := B5
- O4 := B4
- Delete(O1)

Passo 5: ripercorro il LOG in avanti per rifare le operazioni di REDO.

REDO = {T4}

- O3 := A3

Gestore dei metodi di accesso

È il modulo della DBMS che **esegue il piano di esecuzione prodotto dall'utilizzatore**; ovvero, sottomettendo una query, il DBMS ne costruisce un piano di esecuzione ottimizzato (ovvero la trasforma in un insieme).

L'ottimizzatore è un componente che **rende ottimale il piano di esecuzione**.

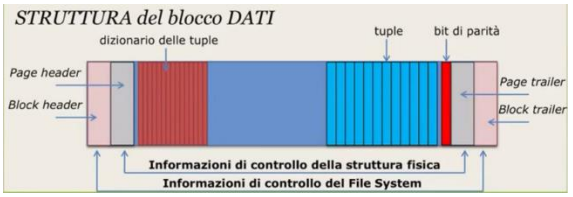
Prodotto il piano di esecuzione, devo decidere anche la sequenza di accessi ai blocchi in memoria secondaria.

Metodi di accesso

Sono i moduli software che implementano gli algoritmi di accesso e manipolazione dei dati.

Alcune politiche sono:	Un metodo di accesso conosce:
• Scansione sequenziale • Accesso via indice • Ordinamento	• **Come sono organizzate le tuple** (o i record degli indici) nei blocchi dati/indice, ovvero come una tabella/indice è organizzata in memoria secondaria. • **L'organizzazione fisica interna dei blocchi**, sia nelle tuple di tabella (dati) che nelle strutture di accesso (record di indici).

Organizzazione blocco di dati



Il blocco di dati contiene:

- **Informazioni utili**: tuple di tabella (i dati veri)
- **Informazioni di controllo**: dizionario, bit di parità…

Il **dizionario** è una struttura che consente di accedere alle tuple.

- Tuple a **lunghezza fissa:** il dizionario non è necessario in quanto basta sapere la dimensione delle tuple e l'offset.
- Tuple a **lunghezza variabile**: il dizionario memorizza l'offset di ciascuna tupla e di ciascun attributo.

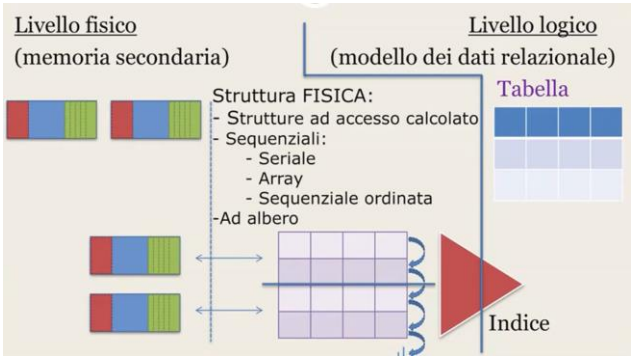
La dimensione massima dipende dalla dimensione massima del blocco. In alcuni casi, tuttavia, postgres è possibile memorizzare una tupla lungo più blocchi.

In postgres si usa TOAST.

Operazioni

- Inserimento della tupla:
  - Se esiste spazio contiguo sufficiente posso inserire direttamente
  - Sì spazio sufficiente ma non contiguo: posso riorganizzare lo spazio ed eseguire un inserimento semplice
  - No spazio sufficiente: operazione rifiutata.
- Cancellazione: sempre possibile
- Accesso alla tupla
- Accesso ad attributo della tupla
- Accesso sequenziale, di solito in ordine di chiave primaria
- Riorganizzazione

Rappresentazione di una tabella a livello fisico



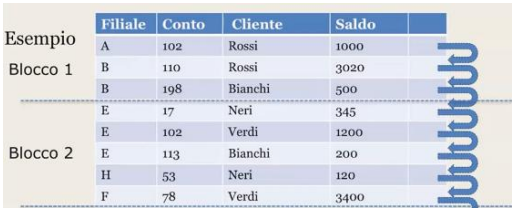
Struttura fisica sequenziale seriale

Dati memorizzati in ordine di inserimento. Fa cagher perché recuperare le cose è un bordello.

Operazioni

Inserimento	› Mi basta inserire la tupla nel blocco più recente.

Struttura sequenziale ordinata



File sequenziale dove le tuple sono ordinate secondo una chiave di ordinamento (di solito la PK).

Operazioni

Inserimento	› Devo individuare il punto dove dovrò eseguire la tupla! › Se l'operazione non va a buon fine ho bisogno di aggiungere un nuovo blocco, detto overflow page, dove aggiungo la mia nuova tupla. Dovrò poi aggiustare la catena di puntatori nel dizionario per poter accedere a questa nuova tupla.
Scansione sequenziale ordinata	› Mi basta usare i puntatori del dizionario per accedere alle righe
Cancellazione della tupla	› Indiviuo il blocco che contiene la tupla › La cancello › Aggiusto la catena di puntatori. [!! Non modifico la struttura dati riorganizzando: lascio i buchi qua e là]
Riorganizzazione	› Quando non ho spazio contiguo, posso riaccorpare le tuple. Riassegno le tuple ai blocchi in base a dei coefficienti di riempimento (ovvero lascio sempre degli spazi liberi, in mododa poter fare operazioni di inserimento senza dover riorganizzare). › Riaggiusto i puntatori.

Strutture ad array

Poco interessante perché vale solo se le tuple hanno dimensione fissa.

Strutture ad accesso calcolato: definizion di indici

La definizione degli indici è fondamentale per ottimizzare il mio tempo di accesso alle tuple.
Gli indici sono stutture dati ausiliare che garantiscono efficienza nel caso di accesso casuale rispetto a una chiave di ricerca.

La **chiave di ricerca** è un insieme di attributi utilizzati dall'indice nella ricerca;non è detto che coincida con la chiave primaria!

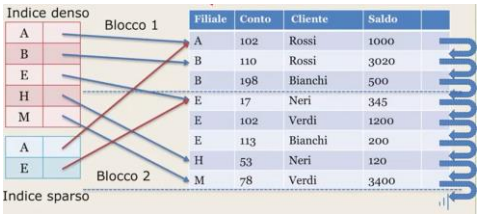
La **chiave di ordinamento** è la chiave secondo cui è stato fatto l'ordinamento.

----- ----- ----- *Su file sequenziale*

Indice primario

la chiave di ordinamento del file coincide con la chiave di ricerca dell'indice. Ogni record contiene una coppia $< v_i, p_i >$ dove

- v_i è il valore della chiave di ricerca
- p_i è il puntatore al primo record nel file sequenziale che corrisponde alla chiave v_i



La chiave di ordinamento v_i è la filiale. Le frecce sarebbero il puntatore al primo record p_i
Nell'indice denso le ho tutte, mentre nell'indice sparso ne ho solo la prima per blocco.

Ne esistono due varianti:

Indice denso : Per ciascuna 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 ce ne sta uno per blocco.

Operazioni:

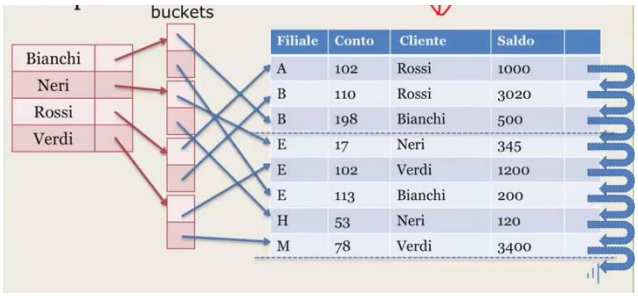
	Denso: K è sicuramente nell'indice.	**Sparso**: K potrebbe non essere nell'indice.
Ricerca di una tupla con chiave di ricerca K	› Scansione sequenziale dell'indice ala ricerca del record (K, p_k) › Accesso al file attraverso il puntatore p_k → **Costo**: 1 accesso indice + 1 accesso blocco dati	› Scansione sequenziale dell'indice alla ricerca di $(K', p_{k'})$ dove K' è il valore più grande che sia minore o uguale a K. › Accesso al file attraverso il puntatore p_k e scansione del file per trovare le tuple di K. → **Costo**: 1 accesso indice + 1 accesso blocco dati. Se la chiave non esiste potrei dover scansionare tutto il blocco!
Inserimento	Devo anche verificare se il file di indice contiene la chiave, e in caso negativo aggiungerla. (Se esiste e il mio nuovo record è il primo, agisco sul blocco di modo che il puntatore rimanga uguale).	Se l'indice è sparso, l'inserimento vviene solamente se l'aggiunta della nuova tupla mi porta a dover aggiungere un nuovo blocco.
Cancellazione	Cancello nell'indice solo se la tupla cancellata è l'ultima tupla con valore di chiave K (nel senso che non ne ho nessun'altra)	La cancellazione nell'indice avviene solo quando K è presente nell'indice ma il corrispondente blocco viene eliminato. Se il blocco sopravvive, va sostituito K nel record dell'indice con il valore successivo presente nel blocco.

Indice secondario

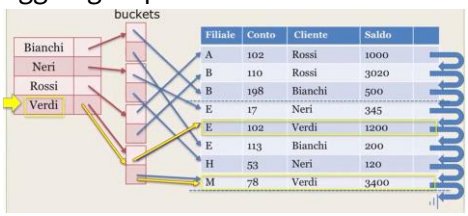
In questo caso la chiave di ordinamento e la chiave di ricerca sono diverse.

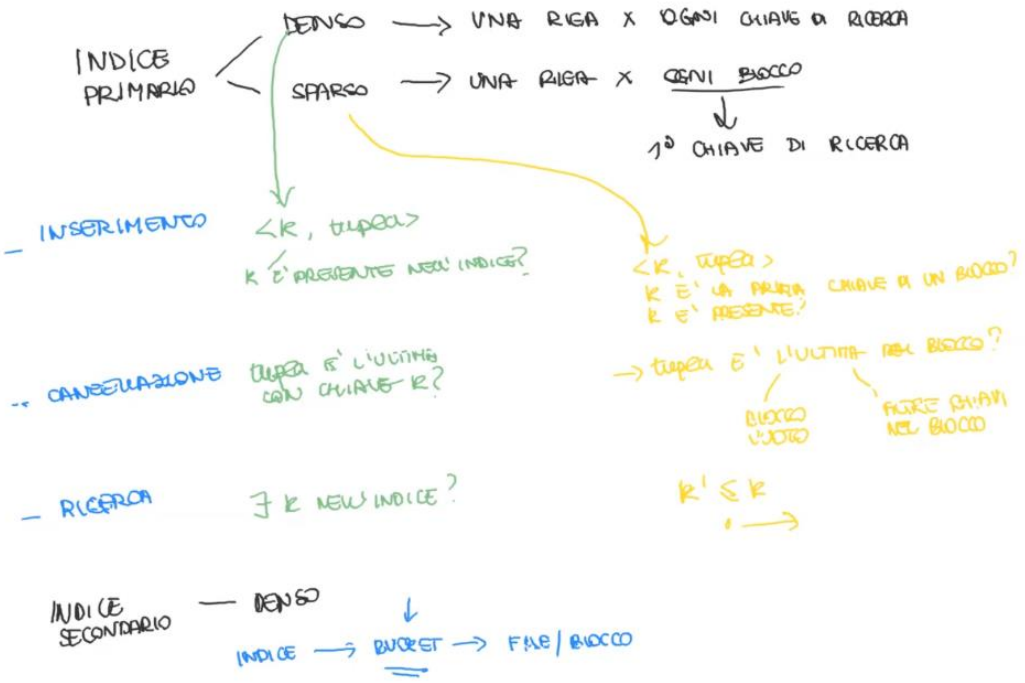
Anche qui i record saranno fatti da $< v_i, p_i >$, ma in questo caso p è un puntatore a un **bucket di puntatori** che individuano tutte le tuple con valore di chiave v_i.

Gli indici secondari sono sempre e solo densi.



Operazioni:

Ricerca di una tupla con chiave di ricerca K	› Scansiono l'indice e trovo il record $< K, p_k >$ › Accesso al bucket puntato da p_k › Accedo alla tabella nella posizione data dal bucket
Inserimento	Ho sempre due casi: › **Chiave di ricerca già presente** → devo aggiungere il puntatore nel bucket › **Chiave di ricerca K non presente** → aggiorno l'indice, creo un nuovo bucket e aggiungo il puntatore. 
Cancellazione	› **Ultima tupla ad avere chiave K** → cancello il record e il bucket › **Esistono altre tuple con chiave K** → cancello solo il puntatore dal bucket



Struttura ad albero: B+ tree

Quando anche l'indice aumenta di dimensioni, non può più stare in memoria centrale; anch'esso dovrà stare in memoria secondaria. Posso usare un file sequenziale ordinato per rappresentare l'indice in memoria secondaria, ma questo degrada molto facilmente e richiederebbe frequenti organizzazioni.

Per questo si preferiscono strutture ad albero e strutture ad ccesso calcolato.

- Struttura ad **albero**
- Ogni **nodo** corrisponde a una **pagina di memoria** in memoria secondaria
- I legami sono puntatori a pagine di memoria
- Ciascun nodo ha un elevato numero di figli: **pochi livelli e tanti nodi foglia**
- L'albero è **bilanciato**, ovvero tutti i rami sono lunghi uguali
  → il tempo di accesso al nodo foglia è sempre lo stesso
  → Inserimento e cancellazione non alterano le prestazioni, poiché manteniamo l'albero bilanciato.

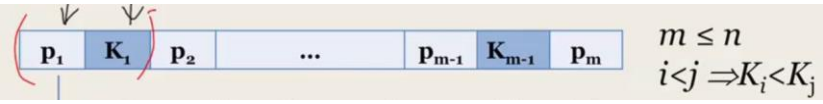
Struttura nodi foglia

Un nodo foglia contiene sempre una **coppia puntatore-chiave**. L'ultimo puntatore punta al nodo foglia successivo.

- **Se ho indice primario**:punta alla prima tupla con chiave K1
- **Se ho indice secondario**: punta al bucket di puntatori verso le tuple con chiave K1

Fanout

> Un "fanout n" corrisponde a dire che l'albero può contenere fino a n-1 valori della chiave di ricerca, e fino a n puntatori. Di fatto dice **quanti puntatori posso avere al massimo nel nodo foglia**.



Caratteristica fondamentale: **vincolo di ordinamento**

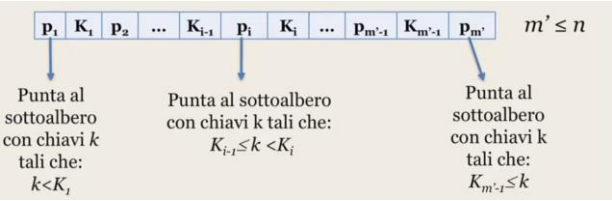
- Nel nodo foglia le chiavi sono ordinate
- Contatenando i nodi foglia, devo contnuare a rimanere ordinato
  → Il nodo foglia successivo è quello che mantiene i nodi ordinati

$$m \leq n$$
$$i<j \Rightarrow K_i<K_j$$

Esiste anche una variante in cui, al posto dei valori chiave, il nodo foglia contiene direttamente le tuple; si parla di **struttura fisica integrata dati/indice.**

Struttura nodo intermedio

Servono a **navigare** all'interno della struttura ad albero e servono ad arrivare ai dati in modo efficiente. In questo caso, **ciascun puntatore punta a un sottoalbero** che ha **chiavi k<K1**. L'ultimo puntatore punterà al sottoalbero con chiavi maggiori di k.



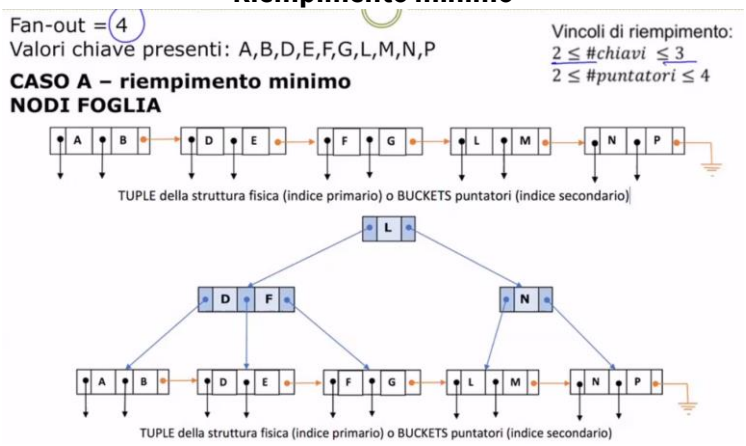
Caratteristica fondamentale: **vincolo di riempimento**

Nodi foglia	**Nodi intermedi**
Ciascun nodo foglia contiene un numero di chiavi vincolato come segue: $\left\lceil \frac{n-1}{2} \right\rceil \leq \#\,chiavi \leq (n-1)$	Ciascun nodo intermedio contiene un numero di chiavi vincolato come segue: $\left\lceil \frac{n}{2} \right\rceil \leq \#\,puntatori \leq n$ ad eccezione della radice, per la quale non vale il minimo.

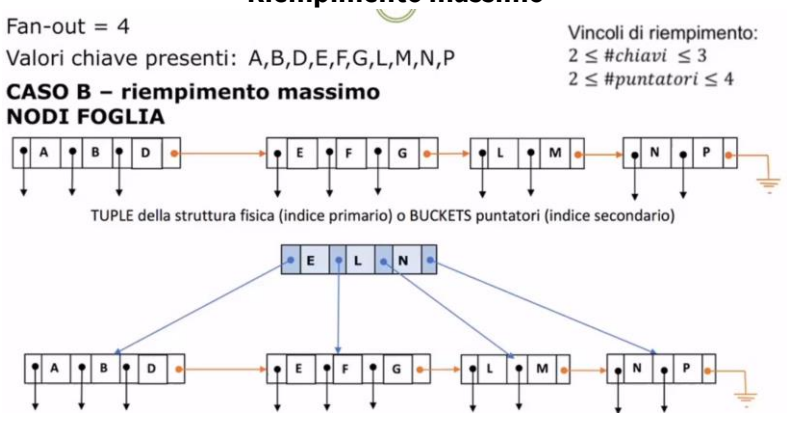
Questo ci garantisce di non avere troppi livelli, semza riempire troppo il nodo foglia in modo da lasciare spazio agli inserimenti.

Esempi

Riempimento minimo



Riempimento massimo

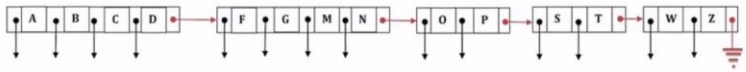


Esercizio: costruzione di un B+ tree

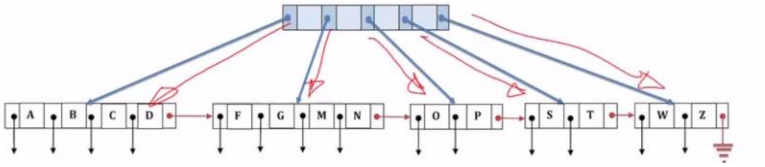
Costruire un B+ tree con fanout 5, ovvero:

- Nodi foglia: #chiavi in 2,4
- Nodi intermedi: #puntatori in 3,5

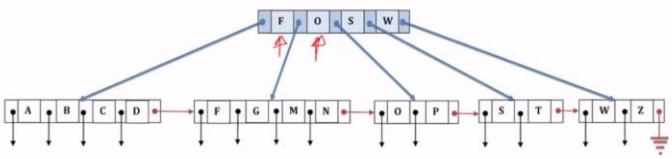
1. **Costruiamo il livello dei nodi foglia**



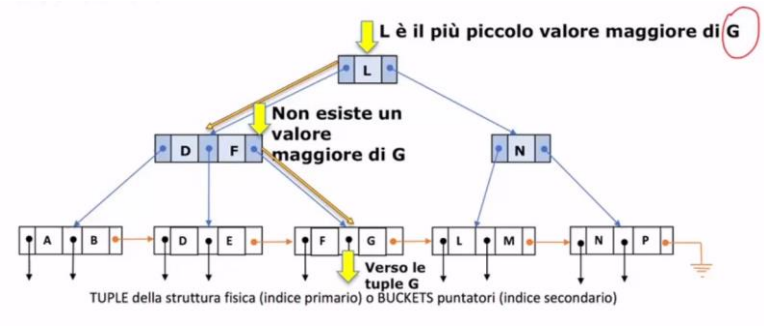
2. **Costruiamo il livello dei nodi intermedi**. Abbiamo un fanout di 5, quindi siamo nel caso fortunato: posso costruire un solo nodo intermedio e ciascun puntatore punterà a uno dei nodi foglia



I valori di chiave da inserire vanno costruiti in modo tale che il puntatore precedente e successivo rispettino il vincolo di puntare a valori compresi tra.
Il primo valore di chiave corrisponde alla prima chiave del nodo successivo.



Ricerca di una chiave K



1. Cerco nel nodo **radice il più piccolo valore di chiave maggiore di K**.
   → Se non esiste (aka K è maggiore di tutti gli altri valori), allora il puntatore che mi serve è l'ultimo.
2. A. **Se non ho raggiunto un nodo foglia riapplico il passo 1**.
   B. Altrimenti, **ricerco nel nodo foglia il valore di chiave K** che sto cercando, e **seguo il puntatore** corrispondente. Se non esiste, significa che non c'è alcuna tupla che contiene quel valore.

Costo: poiché I b+ TREE sono bilanciati, dipende dal fanout e dal numero di valori di chiave.

$$depth_{\{B+tree\}} = costo = 1 + \log_{\lceil n/2 \rceil}\left(\frac{\#\ valori\ chiave}{\lceil (n-1)/2 \rceil}\right)$$

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

- Dato un certo numero di valori chiave da inserire nell'albero (*#valoriChiave*) il numero massimo di nodi foglia è pari a:

$$NF\max = \frac{\#valoriChiave}{\lceil (n-1)/2 \rceil}$$

riempimento minimo

- Quindi partendo dal numero massimo di nodi foglia NF_{max} il numero massimo di livelli dell'albero, in presenza di nodi intermedi a riempimento minimo, risulta pari a:

$$NI_{max} = \log_{\lceil n/2 \rceil}(NF_{max})$$

- Contando il livello dei nodi foglia si ottiene la profondità massima pari a:

$$1 + NI_{max}$$

Inserimento di una chiave K

1. **Cerco il nodo foglia dove voglio inserire il valore con chiave K**
2.
   a. Se **K è presente**, allora
      i. Indice primario: **nessuna azione**
      ii. Indice secondario: **aggiorno il bucket** di puntatori
   b. **Se K non è presente**, allora **inserisco K** rispettando l'ordine
      i. Indice primario: inserisco il **puntatore alla tupla** e K della chiave.
      ii. Indice secondario: inserisco un **nuovo bucket di puntatori** contenente il puntatore alla tupla con valore K della chiave.
3. Se il nodo foglia **vincola il vincolo di riempimento massimo**, faccio uno **SPLIT**.

Cancellazione di un valore della chiave K

1. **Ricerco il nodo foglia** che contiene K
2. **Cancello** il valore K e il suo puntatore.
   a. Indice primario: **nessuna operazione**

b. Indice secondario: **elimino anche il bucket** di puntatori

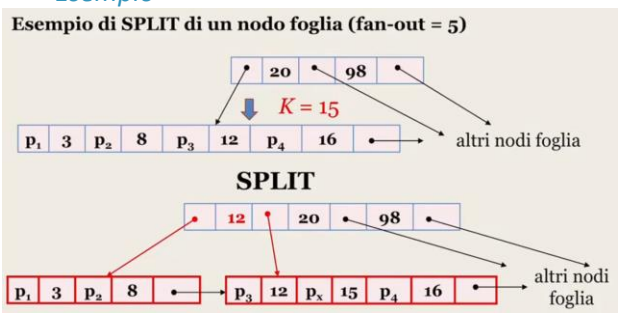
3. Se il nodo foglia **vincola il vincolo di riempimento minimo**, faccio un **MERGE**.

SPLIT di un nodo foglia

Quando devo fare uno split, significa che nel mio nodo ci sono n valori di chiave. Devo dunque creare due nodi foglia:

1. **Creo due nodi foglia**
2. Inserisco i primi $\lceil (n-1)/2 \rceil$ valori nel primo nodo e i rimanenti nel secondo
3. Inserisco nel **nodo padre** un **nuovo puntatore per il secondo nodo foglia generato**, e riaggiusto i valori chiave presenti nel nodo padre.
4. **Se anche il nodo padre è pieno eseguo lo SPLIT**, propagandolo se necessario fino alla radice.

Esempio

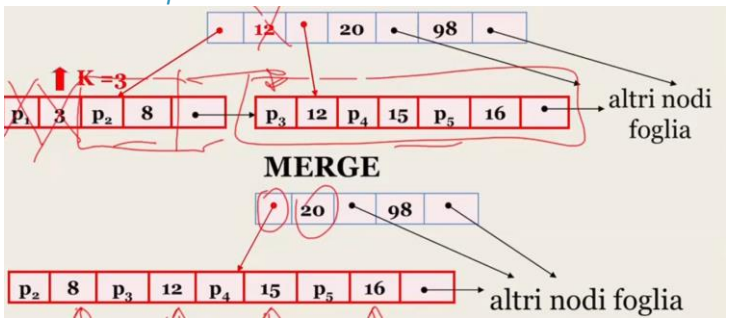


- Il nodo figlio è troppo grande: il massimo da vincolo sarebbe fanout-1 = 4 nodi! Quindi:
  1. Metto i primi $\lceil 5/2 \rceil = 2$ valori (3,8) nel primo nodo foglia, e i restanti (12,15,16) nel secondo.
  2. Modifico il nodo padre: avendo aggiunto un nodo con valori più piccoli di 12 devo aggiungere il 12.

MERGE di due nodi foglia

1. Devo **individuare il nodo fratello adiacente** da unire al nodo corrente
2. Se i due nodi hanno complessivamente al **massimo $n-1$ valori chiave**, allora
   a. **Genero un unico nodo contenente tutti i valori**
   b. **Tolgo un puntatore dal nodo padre**
   c. **Aggiusto i valori** chiave del nodo padre
3. Altrimenti distribuisco i valori chiave fra i due nodi, e aggiusto i valori chiave del nodo padre
4. Se anche il nodo padre viola il vincolo di riempimento minimo, ovvero ha meno di $\lceil n/2 \rceil$ puntatori, il **MERGE** si propaga verso l'alto.

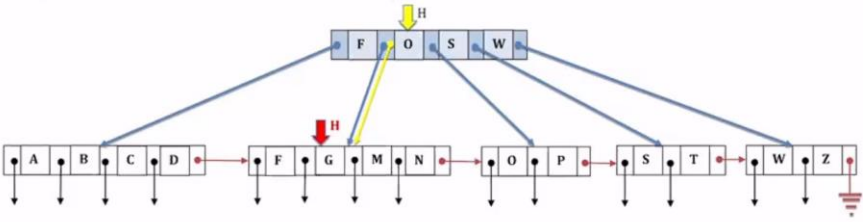
Esempio



- Il nodo figlio è troppo piccolo rimuovendo 3 rimango con 1 nodo, ma il minimo è 2. Quindi:
  1. Costruisco un nodo unico prendendo il nodo e il fratello. Sono nel caso fortunato: il numero di chiavi rispetta il vincolo!
  2. Aggiorno il padre, togliendo 12 e il puntatore + aggiorno il puntatore.

Esercizio: costruire un B+ tree (con split!)

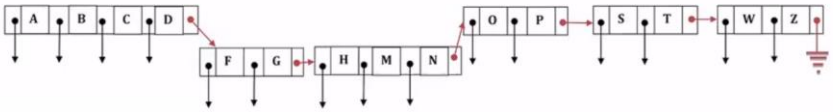
2) Inserire il valore chiave H nel **B+-tree** ottenuto al punto 1.



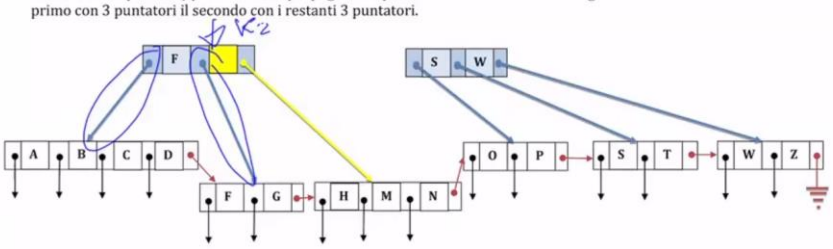
Poiché il nodo foglia raggiunto contiene già il numero massimo di valori chiave è necessario applicare uno SPLIT.
ATTENZIONE: non si esegue alcuna analisi dei nodi adiacenti per adottare soluzioni diverse!

SPLIT: metto i primi 2 valori nel primo nodo e i rimanenti nel secondo

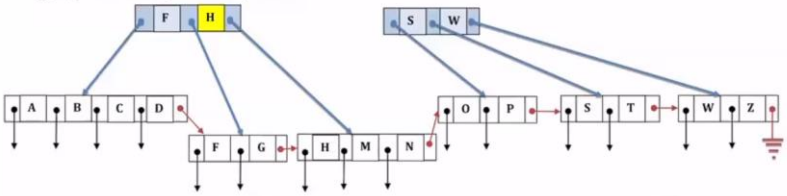
Situazione dei nodi foglia dopo lo SPLIT:



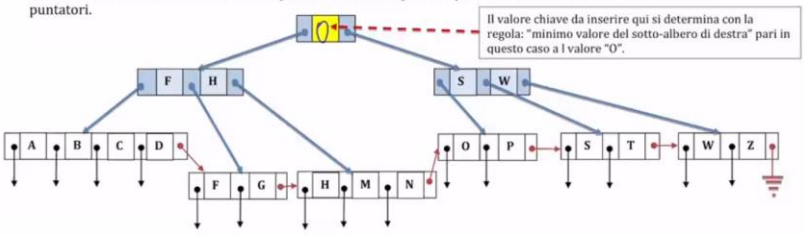
Ci sono a questo punto 6 nodi foglia e il nodo intermedio preesistente conteneva 5 puntatori (il massimo consentito dai vincoli di riempimento) pertanto devo propagare lo split anche sul nodo intermedio generando due nodi intermedi: il primo con 3 puntatori il secondo con i restanti 3 puntatori.



Occorre quindi controllare i valori chiave presenti nei nuovi nodi intermedi generati per verificarne la correttezza a aggiungere un valore per il nuovo puntatore.

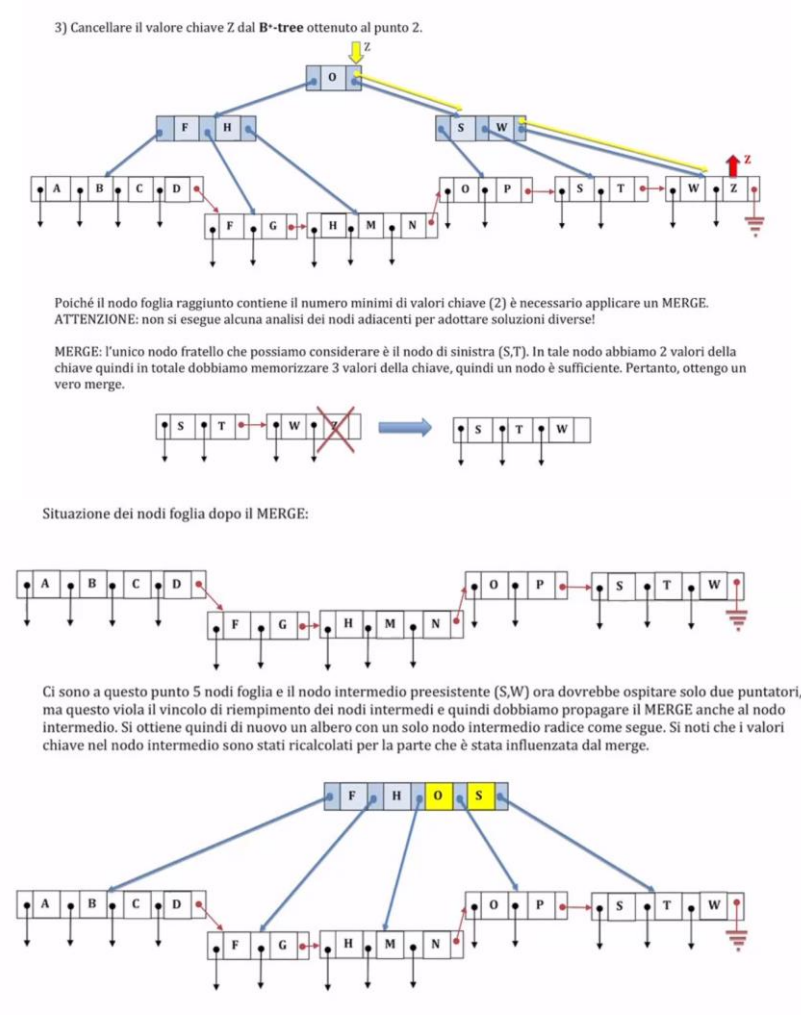


A questo punto occorre completare l'albero aggiungendo un nuovo nodo radice e quindi un nuovo livello. Si noti che per il nodo radice non vale il vincolo di riempimento minimo e quindi si possono inserire in tale nodo anche solo 2 puntatori.



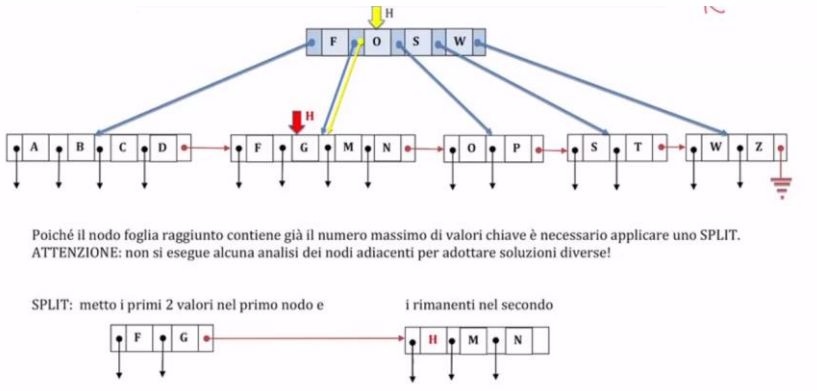
1. **Tramite la ricerca, trovo che la più piccola chiave più grande è O**. Per il fanout di 5 non posso contenere più d 4 valori chiavi: devo fare uno split
2. **SPLIT**
    a. Costruisco due nodi: nel primo metto n-1/2 valori, e nel secondo tutti gli altri (incluso il nuovo valore).
    b. Dopo questo primo split, ho due nodi foglia nuovi: aggiorno i puntatori dei 6 nodi foglia e aggiorno il padre.
3. **ALTRO SPLIT: anche il padre è pieno, quindi propago lo split**
    a. Separo il padre in due nodi. Il nuovo valore per il primo dei due nodi sarà H.
4. **A questo punto non ho più un nodo radice, quindi lo aggiungo**.
    a. Costruisco un nuovo nodo radice
    b. Aggiorno i suoi 2 puntatori
    c. Come valore centrale mi serve un valore tale per cui a sinistra punto a valori più piccoli (<) , e a destra a valori uguali o più grandi (>=) . Tutti i sottoalberi sono interessati da questo vicolo, quindi (per esempio) non posso guardare solo qualcosa più piccolo di S; deve essere compreso fra N e O. **Quindi metterò O**.

Esercizio: costruire un B+ tree (con split!)



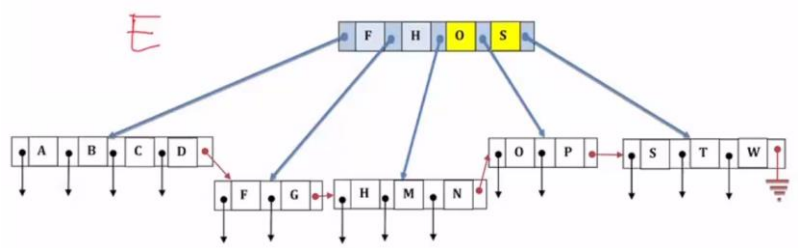
1. Anche qui, tramite ricerca **trovo il nodo Z da eliminare.**
2. Eliminando il nodo, vado sotto il minimo dato dal fanout; dovrò fare un **MERGE**.
    a. Prendo il fratello e costruisco un unico nodo.
    b. Siamo nel caso fortunato: posso unirli direttamente.
3. Anche sopra, dovrei eliminare uno dei valori finendo fuori dal range permesso: faccio un altro **MERGE**.
    a. Prendo il fratello e costruisco un unico nodo.
4. Avendo un unico nodo intermedio **posso eliminare la radice**.

Esercizio: inserire il valore chiave R nel seguente albero



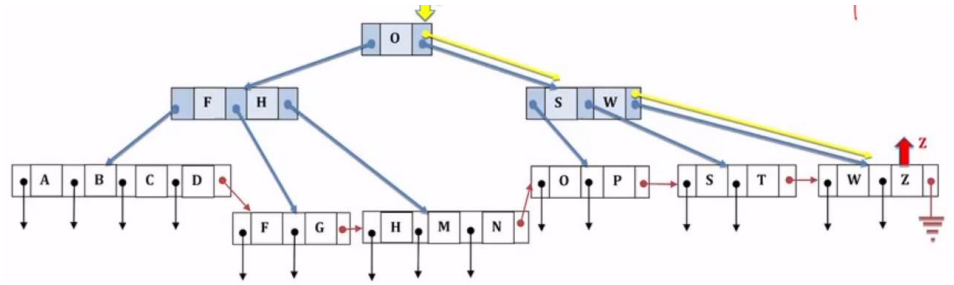
1. Cerco dove inserire R
    a. Parto dal nodo radice e trovo il più grande valore minore di R → S
    b. Seguendo il puntatore, trovo il più grande valore minore di R → P
2. Inserisco R
    a. Aggiungo R dopo P, e imposto il puntatore dopo P affimché punti a S.

Esercizio: inserire il valore chiave E nel seguente albero



1. Cerco dove inserire E
    a. Parto dal nodo radice e cerco il più grande valore minore di E → Non c'è; vado sul puntatore prima di F
    b. Cerco il più grande valore minore di E → D
2. Per inserire E dopo D avrei 5 elementi in un solo nodo; devo quindi fare uno SPLIT
    a. Creo un nuovo nodo
    b. Lascio A,B nel vecchio nodo e aggiorno il puntatore dopo B affinché punti al nuovo nodo
    c. Metto C,D,E nel nuovo nodo e aggiorno il puntatore dopo E affinché punti a F
    d. Il nodo padre dovrà avere C,F,H,O,S con il puntatore dopo C che punta al nuovo nodo
3. Per aggiornare il padre affinché punti al nuovo nodo avrei 5 elementi in un solo nodo; devo fare uno SPLIT
    a. Creo un nuovo nodo
    b. Metto C,F nel vecchio nodo e imposto il puntatore dopo F verso il nuovo nodo NOPE! VA FATTO SOLO NEI NODI FOGLIA → NON PUNTA A NULLA
    c. Metto H,O,S nel nuovo nodo e imposto il puntatore dopo S verso il nodo di F
4. Non ho più nodo radice:
    a. Creo un nodo radice i cui puntatori puntano ai due nodi del punto 3
    b. La key centrale sarà H

Esercizio: cancellare valore T



1. Identifico il nodo foglia che contiene T
2. Cancello T; sto violando il vincolo minimo quindi faccio una MERGE.
    a. ! Posso scegliere con quale dei due mergiare! Se ci fosse una violazione di vincolo a fronte del MERGE, scelgo quello più facile :)
    b. Mergiamo con il terzultimo nodo. Il risultato non viola il vincolo :)
    c. Il nodo padre cambia: avrò un solo valore, e sarà W.
3. Il nuovo nodo padre viola il vincolo minimo, quindi devo fare un'altra MERGE.
    a. Mergio i due. Rimangono F,H,W.
4. Posso togliere il nodo radice.

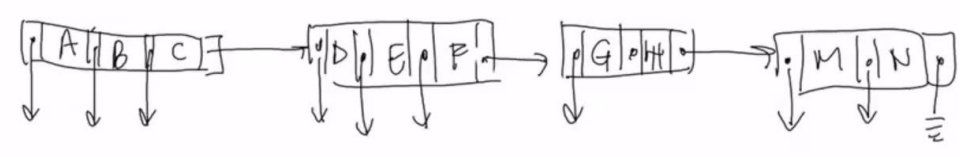
Esercizio: Costruire B+ tree

1) Costruire un **B$^+$-tree** di fan-out=5 con i seguenti nodi foglia: (A,B,C,D), (F,G,M,N), (O,P), (S,T), (W,Z)

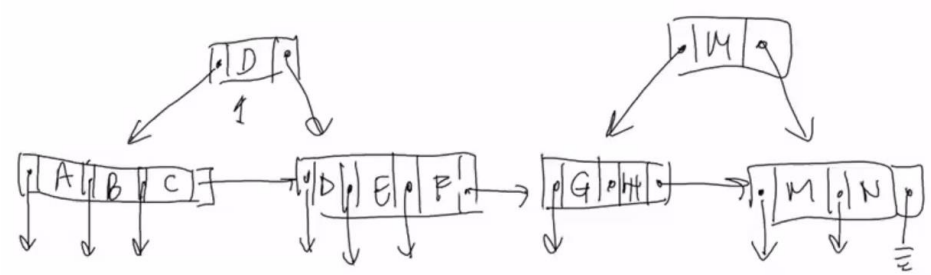
Con fanout 4:

- 1 ≤ # chiavi ≤ 3
- 3 ≤ # puntatori ≤4

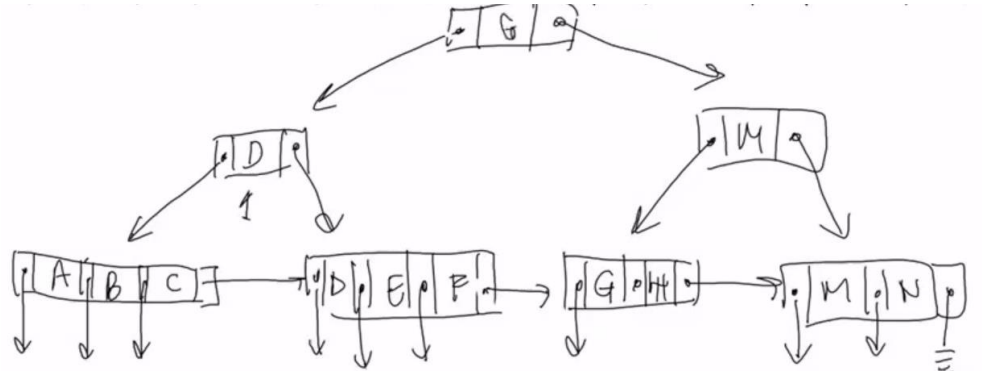
1. Costruisco il livello dei nodi foglia rispettando il fanout



2. Aggiungo nodi intermedi.
    a. Il fanout è 4, quindi facendo il riempimento minimo saranno 1 chiave/ 2 puntatori per nodo.
    b. Le chiavi saranno D,M



3. Costruisco la radice; la chiave centrale sarà G



Esercizio: cancella M dall'albero precedente

1. Scorro fino a trovare il nodo foglia M
2. Il nuovo nodo foglia viola il vincolo minimo (1 sola chiave!): faccio la MERGE
    a. Unisco G,H,M,N

Strutture ad accesso calcolato: hashing

Si basano su una funzione di hash, che mappa i valori della chiave di ricercr su indirizzi di memorizzazione delle tuple. In pratica, è una funzione $h : K \rightarrow B$ *con* K: *dominio delle chiavi*, B : *dominio degli indirizzi*.

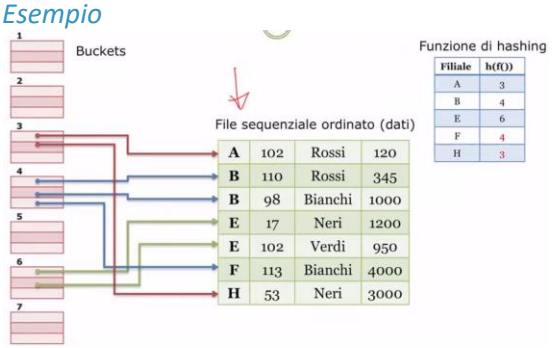
Per poter utilizzare questa funzione:

- Stimo il numero di valori chiave che saranno contenuti nella tabella
- Alloco il numero di buceket di puntatori (B) uguale al numero stimato
- Definisco una **funzione di FOLDING** che trasforma i valori chiave in numeri interi positivi
$$f : K \rightarrow Z^+$$
- Definisco una **funzione di HASHING**
$$h : Z^+ \rightarrow B$$

> Una funzione di HASH è ottimale quando garantisce una distribuzione **uniforme** e **casuale** dei valori della chiave nei bucket.

Cambiare la funzione di hashing a posteriori è molto pesante! Quindi è bene che la stima iniziale sia precisa :')

Esempio



Nel mio bucket di puntatori ho un numero per ciascun numero che puà uscire dalla funzione di HASH. Può succedere che la funzione di HASH produca lo stesso output a partire da numeri in input diversi.

Operazioni

Ricerca	Molto efficiente: dato un valore di K • Calcolo il bucket come $b = h(f(K))$ • Accedo al bucket b • Accedo alle n tuple attraverso i puntatori del bucket **Costo**: 0 + 1 accesso + m accessi, con m <=n
Inserimento e cancellazione	Come la ricerca! • Trovo il bucket attraverso hash e folding • Scorro le tuple del bucket • Cancello/inserisco

Gestione delle collisoini

La struttura di hash funziona bene se i bucket hanno basso coefficiente di inserimento.
Nelle struttura di hash possono esserci delle **collisioni**: questo si verifica quando, dati due valori con chiave diversa, la funzione di hash dà **risultato uguale**:

$$K_1 \neq K_2 \wedge h\big(f(K_1)\big) = h(f(K_2))$$

Un numero eccessivo di collisioni porta alla **saturazione del bucket** corrispondente: i bucket iniziano ad aumentare di dimensione, e la ricerca diventa sempre meno efficiente/selettiva.

La probabilità di avere una collisione è “una formula complicatissima” (cit.)

Probabilità che un bucket riceva t chiavi su n inserimenti: $p(t) = \binom{n}{t}\left(\frac{1}{B}\right)^t \left(1 - \frac{1}{B}\right)^{n-t}$

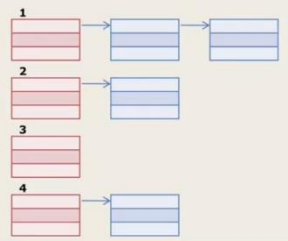
$con\ B = numero\ totale\ di\ buckets$

Probabilità di avere più di F collisioni:

$$p_K = 1 - \sum_{i=0}^{F} p(i)$$

Quindi, le collisioni dipendono dal numero di chiavi e dal numero di bucket; se faccio una buona stima all'inizio del numero di chiavi (e quindi definisco un numero di bucket appropriato) minimizzo le collisioni.

Bucket di overflow



Quando c'è un numero eccessivo di collisioni, arrivo ad avere una saturazione del bucket. Non abbiamo una soluzione semplice come il MERGE/SPLIT dell'albero, ma usiamo un **bucket di overflow**: aggiungo dei bucket "successivi.

L'inserimento rimane invariato, ma la ricerca peggiora moltissimo: individuato il bucket principale potrebbe esserci bisogno di accedere a un gran numero di overflow.

Il caso pessimo è che tutti i dati siano memorizzati nello stesso bucket.

Confronto fra B+ tree e hashing

	B+ tree	Hashing
Ricerca su costante	Logaritmico nel numero di chiavi	Costante se non ho overflow
Ricerca su range	Tempo logaritmico per accedere al primo valore dell'intervallo, poi scansione dei nodi foglia fino all'ultimo valore	Numero elevato di selezioni su condizioni di uguaglianza per scandire tutti i valori
Inserimento e cancellazione	Tempo logaritmico nel numero di chiavi + split/merge	Tempo costante + gestione overflow

Qual è il migliore? Non c'è una risposta univoca, dipende dall'applicazione.

- B+ tree ha tempo peggiore *in generale*, ma riesce a gestire in tempo logaritmico gli intervalli
- Hash più veloce per la maggior parte delle operazioni, ma molto male negli intervalli

Normalmente si costruisce una struttura ad albero.

Esecuzione concorrente di transizioni

Per gestire con prestazioni accettabili il carico di lavoro tipico(100-1000 tps), un DBMS deve riuscire a eseguire le transazioni in modo concorrente. L'esecuzione concorrente, tuttavia, se non controllata accuratamente può generare anomalie o problemi di correttezza: è quindi necessario introdurre dei meccanismi di controllo.

Anomalie tipiche:

- **Perdita di aggiornamento**: gli effetti di una transazione sono persi perché altre transazioni concorrenti ne eliminano gli effetti
- **Lettura inconsistente**: accessi successivi ad uno stesso dato all'interno di una transazione ritornano valori diversi → all'interno della stessa transazione ci sono due letture e ritornano due risultati diversi
- **Lettura sporca**: viene letto un dato che rappresenta uno stato intermedio di un'altra transazione concorrente → leggo qualcosa che poi viene rollbacked
- **Aggiornamento/inserimento fantasma**: otteniamo valori aggregati diversi all'interno della stessa transazione perché nel frattempo c'è stato un inserimento intermedio.

Notazione:

- t_i : transazione
- $r_i\,(x)$: lettura della transazione t sulla risorsa x
- $w_i(x)$: scrittora della transazione t sulla risorsa x

Perdita di aggiornamento

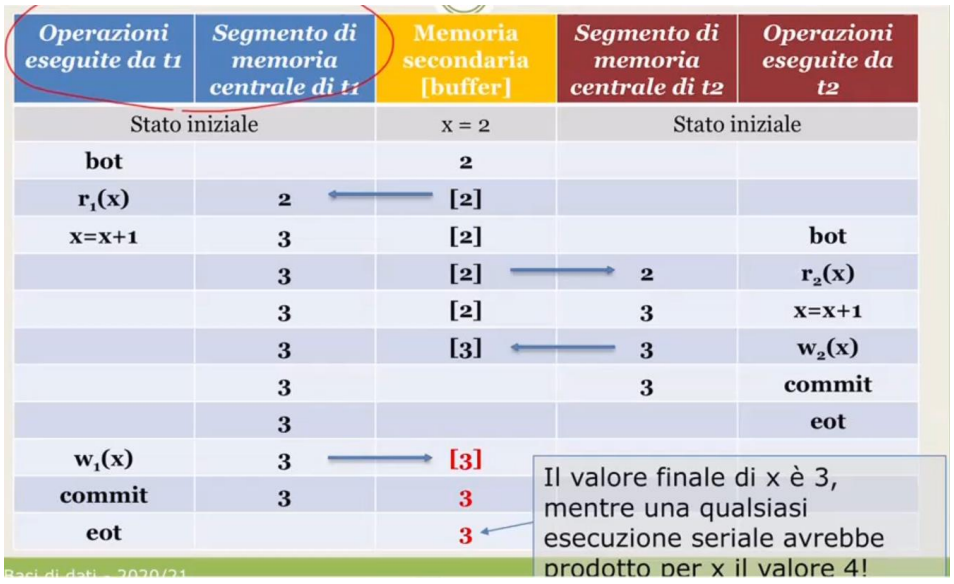
> Gli effetti di una transazione vengono persi a fronte dell'esecuzione di un'altra transazione.

Esempio

t_1:
$bot\ r_1(x); x = x + 1; w_1(x)\ ; commit; eot$

t_2:
$bot\ r_2(x); x = x + 1; w_2(x); commit; eot$

Esempio di esecuzione che causa una perdita di aggiornamento: [1:33 Lezione 27]



Parentesi quadrata: non è ancora stato fatto il commit

Lettura inconsistente

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

Esempio

t_1: $bot\ r_1(x); r_1'(x)\ ; commit; eot$ | t_2: $bot\ r_2(x); x = x + 1; w_2(x); commit; eot$

Operazioni eseguite da t1	Segmento di memoria centrale di t1	Memoria secondaria [buffer]	Segmento di memoria centrale di t2	Operazioni eseguite da t2
Stato iniziale		x = 2	Stato iniziale	
bot		2		
$r_1(x)$	2	[2]		
	2	[2]		bot
	2	[2]	2	$r_2(x)$
	2	[2]	3	x=x+1
	2	[3]	3	$w_2(x)$
	2	3	3	commit
	2	3		eot
$r_1'(x)$	3	[3]		
commit	3	3		
eot		3		

Il valore di x è diventato 3, ma la transazione t_1 non ha modificato x!

Sebbene t_1 non abbia eseguito alcuna modifica sul valore, due letture successive hanno generato un diverso valore della stessa variabile!

Lettura sporca

Abbiamo un dato che rappresenta uno stato intermedio di un'altra transazione. (= quel valore intermedio non rimane nemmeno, ad esempio perché l'altra transazione ha eseguito un abort).

Esempio

t_1: $bot\ r_1(x); commit; eot$ | t_2: $bot\ r_2(x); x = x + 1; w_2(x); \dots abort$

Operazioni eseguite da T1	Segmento di memoria centrale di T1	Memoria secondaria [buffer]	Segmento di memoria centrale di T2	Operazioni eseguite da T2
Stato iniziale		x = 2	Stato iniziale	
bot		2		
		2		bot
		[2]	2	$r_2(x)$
		[2]	3	x=x+1
		[3]	3	$w_2(x)$
$r_1(x)$	3	[3]	3	abort
commit	3	2		
eot		2		

Il valore di x letto da t_1 è 3, ma la transazione t_2 non ha ancora eseguito il commit!

T1 ha letto il valore prima che venisse eseguito l'abort, ma poi eseguendo l'abort quel valore non esiste nemmeno più :(

Aggiornamento fantasma

Si verifica quando vado a calcolare dei valori aggregati.

t1: bot r1(y); r1(x); r1(z); s = x+y+z; eot
t2: bot r2(y); y = y + 10; r2(z); z = z – 10; w2(y); w2(z); commit; eot

Simulazione dell'esecuzione concorrente che genera l'anomalia.

Operazioni eseguite da t1	*Segmento di memoria centrale di t1*	*Memoria secondaria [buffer]*	*Segmento di memoria centrale di t2*	*Operazioni eseguite da t2*
	Stato iniziale	(x,y,z)=20,20,60	**Stato iniziale**	
bot		20,20,60		
r1(y)	-,20,-	20,[20],60		
	-,20,-	20,[20],60		**bot**
	-,20,-	20,[20],60	-,20,-	**r2(y)**
	-,20,-	20,[20],60	-,30,-	**y = y + 10**
	-,20,-	20,[20],[60]	-,30,60	**r2(z)**
	-,20,-	20,[20],[60]	-,30,50	**z = z - 10**
	-,20,-	20,[30],[60]	-,30,50	**w2(y)**
	-,20,-	20,[30],[50]	-,30,50	**w2(z)**
	-,20,-	20,30,50	-,30,50	**commit**
	-,20,-	20,30,50		**eot**
r1(x)	20,20,-	[20],30,50		
r1(z)	20,20,50	[20],30,[50]		
s=x+y+z	20,20,50	[20],30,[50]		
eot		20,30,50		

S=90
VIOLAZIONE VINCOLO

Vincolo: sommatoria di x,y,z = 100

Gestione delle anomalie

Schedule

> Uno **schedule** è un possibile ordine di esecuzione di una serie di transazioni.

Il problema è determinare se un certo ordine di esecuzione può essere accettato o meno: si stabilisce di accettare esclusivamente gli schedule equivalnti a uno schedule seriale.

Schedule seriale

È uno schedule dove le operazioni di ciascuna transazione compaiono in sequenza, senza essere inframmezzate da operazioni di altre transazioni.

- **Non seriale:** $r_1(x), w_2(x), w_1(x)$
- **Seriale:** $r_1(x), w_1(x), r_2(x), w_2(x)$

Schedule serializzabile

> È uno schedule equivalente allo schedule seriale, ovvero che ne produce gli stessi effetti.

A questo punto dobiamo definire bene la relazione di equivalenza fra schedule, ovvero quando uno schedule diventa serializzabile. L'idea di base è che gli effetti ottenuti alla fine sono gli stessi.

Ipotesi di commit-proiezione

Per poter determinare se uno schedule è serializzabile servono delle procedure. Per farlo si parte dall'**ipotesi di commit-proiezione**, ovvero supponiamo che tutte le transizioni abbiano un esoto noto – ovvero sappiamo se andrà a buon fine oppure no. A questo punto eliminiamo dallo schedule tutte le operazioni che derivano da transazioni che non andranno a buon fine.

Questa ipotesi è necessaria nelle seguenti tecniche di gestione della concorrenza:

- Gestione basata sulla **view-equivalenza**
- Gestione basata sulla **conflict-equivalenza**

Definizioni preliminari

Relazione "LEGGE_DA"	**Relazione "SCRITTURE_FINALI"**
Dato uno schedule S, si dice che un'operazione di lettura $r_i(x)$, che compare in S, $r_i(x)$ **LEGGE_DA** un'operazione di scrittura $w_j(x)$ che compare in S se e solo se: • $w_j(x)$ **precede** $r_i(x)$ in S • Non vi è alcuna operazione $w_k(x)$ tra le due. (=è **l'ultima scrittura prima della lettura**). → Noob-like: sarebbero letture che seguono scritture! Esempio: • $S1$: $r_1(x), r_2(x), w_2(x), w1(x)$ → **LEGGE_DA(S1) = ∅** • $S2$: $r_1(x), w_1(x), r_2(x), w_2(x)$ → **LEGGE_DA(S2) =** $\{(r_2(x), w1(x))\}$	Dato uno schedule S, un'operazione di scrittura $w_j(x)$ si dice **SCRITTURA_FINALE** se è l'ultima operazione di scrittura della risorsa x in S. Esempio: • $S1$: $r_1(x), r_2(x), w_2(x), w_3(y), w_1(x)$ → **SCRITTURE_FINALI(S1) =** $\{w_1(x), w_3(y)\}$ • $S2$: $r_1(x), w_1(x), r_2(x), w_2(x), w_3(y)$→ **SCRITTURE_FINALI(S2) =** $\{w_3(y), w_2(x)\}$

View-equivalenza

View-equivalenza	View-serializzabilità
Due schedule S1 e S2 sono view-equivalenti (**S1≈ᵥ S2**) se possiedono le stesse relazioni **LEGGE_DA** e **SCRITTURE_FINALI**.	Uno schedule S è view-serializzabile (**VSR**) se esiste uno schedule seriale S' tale per cui **S'≈ᵥS.**

! ATTENZIONE: gli schedule seriali S' si generano considerando tutte le possibili permutazioni possibili delle transizioni che compaiono in S. Inoltre**, NON SI DEVE CAMBIARE L'ORDINE** delle operazioni all'interno della transizione: questo significherebbe modificare la transizione!

Tutti gli schedule che corrispondono ad anomalie di perdita di aggiornamento sono non view-serializzabili. → **se è view-serializzabile, allora non avrò problemi di aggiornamento. :)**

Esempi di calcolo della VRS: perdita di aggiornamento

T_1: $r_1(x)$ $w_1(x)$ T_2: $r_2(x)$ $w_2(x)$

Lo schedule che rappresenta l'anomalia è il seguente

S_{PA} = $r_1(x)$ $r_2(x)$ $w_2(x)$ $w_1(x)$

1. Calcolo le relazioni viste prima
   a. LEGGE_DA: identifico quali sono le opera immediatamente precedente. Nel nostro caso, nessuna lettura e preceduta da una lettura: l'insieme è quindi vuoto. **LEGGE_DA(S1) = Ø**
   b. SCRITTURE_FINALI: per ciascuna risorsa utilizzata scrivo l'ultima write. Qui l'unica risorsa è la x! **SCRITTURE_FINALI: { $w1(x)$ }**
2. Genero tutti i possibili schedule seriali. Sono tutte le possibili permutazioni delle transazioni. Qui sono due: T2T1 o T1T2. (! Considero la permutazione delle transazioni, non la permutazione delle operazioni!)
   i. $T_1T_2 = S_1 = r_1(x)w_1(x)r_2(x)w_2(x)$
   ii. $T_2T1_2 = S_2 = r_2(x)w_2(x)r_1(x)w_1(x)$
3. Guardo se ci sono schedule seriali view equivalenti.
   i. LEGGE_DA(T1T2) = $\{(r_2(x),w_1(x)\}$ SCRITTURE_FINALI = $\{w_2(x)\}$ → **NON è ≈ᵥ**
   ii. LEGGE_DA(T2T1) = $\{(r_1(x),\ w_2(x)\}$ SCRITTURE_FINALI = $\{w_1(x)\}$ → **NON è ≈ᵥ**

→ **NON VSR**: Non esiste nessuna S view-equivalente! Allora S non è view-serializzabile, e produrrà delle anomalie.

Esempi di calcolo della VRS: lettura inconsistente

T_1: $r_1(x)$ $r_1'(x)$ T_2: $r_2(x)$ $w_2(x)$

Lo schedule che rappresenta l'anomalia è il seguente

S_{LI} = $r_1(x)$ $r_2(x)$ $w_2(x)$ $r_1'(x)$

1. Calcolo le relazioni viste prima
   a. **LEGGE_DA(S) =** $\{(r_1'(x), w_2(x)\}$
   b. **SCRITTURE_FINALI: { $w_2(x)$ }**
2. Genero tutti i possibili schedule seriali
   i. $T_1T_2 = S_1 = r_1(x)r_1'(x)r_2(x)w_2(x)$

ii. $T_2T1_2 = S_2 = r_2(x)w_2(x)r_1(x)r_1'(x)$

3. Guardo se ci sono schedule seriali view equivalenti.

   i. LEGGE_DA(T1T2) = ∅ SCRITTURE_FINALI = $\{w_2(x)\}$ → **NON è $\approx_v$**

   ii. LEGGE_DA(T2T1) = $\{(r_1(x),\ w_2(x), (r_1'(x),\ w_2(x)\}$ SCRITTURE_FINALI = $\{w_2(x)\}$ → **NON è $\approx_v$**

→ **NON VSR**: Non esiste nessuna S view-equivalente! Allora S non è view-serializzabile, e produrrà delle anomalie.

Potevo risolvere semplicemente: se avessi spostato w_2 alla fine, il problema delle letture inconsistenti non sarebbe sorto.

Costo:

- Determinare se due schedule sono view equivalenti è di complessità lineare.
- L'algoritmo di view-serializzabilità, però, è che in un sistema reale ho centinaia-migliaia di transazioni concorrenti; **generare tutte le possibili permutazioni** di schedule concorrenti significa avere un calcolo di complessità esponenziali.

Conclusione:

→ La VSR richiede algoritmi di **complessità troppo elevata**
→ Ci vuole l'ipotesi della **commit-proiezione**
→ Non è applicabile nei sistemi reali.

Conflict-equivalenza

Dato uno schedule S, si dice che una coppia di operazioni (a_i, a_j) rappresentano un conflitto se:

$i \neq$ j (ovvero sono due transizioni diverse)

Stessa risorsa

Almeno una delle due è un'operazione di scrittura

a_i compare prima di a_j

→ ovvero qualsiasi coppia di operazioni di transazioni diverse che operano sulla stessa risorsa, e di cui almeno una delle due è di scrittura.

Conflict-equivalenza	**Conflict-serializzabilità**
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)	Uno schedule S è view-serializzabile (**CSR**) se esiste uno schedule seriale S' tale per cui **S'$\approx_c$S.**

Costo: complessità lineare

Algoritmo

- Costruisco il grafo dei conflitti G(N,A) dove
  - N = $\{t_1, \dots, t_n\}$ con t transazioni di S
  - A contiene coppie t_i, t_j se essite almeno un conflitto a_i, a_j in S

- Se il grafo è **aciclico** allora S è conflict-serializzabile.

Per definizione, uno schema conflict-serializzabile è conflict-equivalente ad uno schedule che abbiamo S_ser.

Algoritmo sul grafo

Diciamo che se S è CSR, allora il suo grafo è aciclico.

- Per definizione, se uno schedule è CSR allora è $\approx_c$ a uno schedule seriale S_{ser}
- Supponiamo che Sser abbia transazioni ordinate per il loro ID. Poiché Sser è $\approx_c$ a S, Sser ha tutti i conflitti di S esattamente nello stesso ordine
- Poiché Sser è uno schedule seriale, possiamo avere solo archi (i,j) con i<j, e quindi il grafo non può avere cicli. Se il grafico è aciclico, allora possiamo dare un rodinamento topologico ai nodi, ovvero possiamo dare un numero a ciascun nodo (=transazioni) tali per cui il grafo contiene solo archi i,j con i<j.
- Poiché S è $\approx_c$ a Sser, anche il grafo di S sarà aciclico.

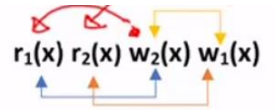
Esempi di calcolo della CRS: perdita di aggiornamento

T_1: $r_1(x)$ $w_1(x)$ T_2: $r_2(x)$ $w_2(x)$

Lo schedule che rappresenta l'anomalia è il seguente

$S_{PA} = r_1(x)\ r_2(x)\ w_2(x)\ w_1(x)$

1. Calcoliamo l'insieme dei conflitti: un conflitto è una copia di operazioni che appartengono a transazioni diverse, agiscono sulla stessa risorsa, una è unascrittura e sono in ordine inverso rispetto allo schedule. Partiamo dalle operazioni di scrittura:



   a. w_2 è in conflitto con r r_1
   b. w_1 è in conflitto con r_2
   c. w_1 è in conflitto con w_2

   Conflitti: $\{(r_1, w_2) = t1 \rightarrow t2, (r_2, w_1) = t2 \rightarrow t1, (w_2, w_1) = t2 \rightarrow t1\}$
2. Costruisco il grafo, con un nodo per transazione

3. È CICLICO: **non è CSR.**

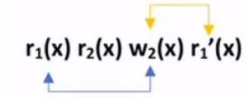
Esempi di calcolo della VRS: lettura inconsistente

T_1: $r_1(x)$ $r_1'(x)$ T_2: $r_2(x)$ $w_2(x)$

Lo schedule che rappresenta l'anomalia è il seguente

$S_{LI} = r_1(x)\ r_2(x)\ w_2(x)\ r_1'(x)$

1. Calcoliamo l'insieme dei conflitti



   Conflitti: $\{(r_1, w_2) = t1 \rightarrow t2, (w_2, r_1') = t2 \rightarrow t1\}$
2. Grafo:

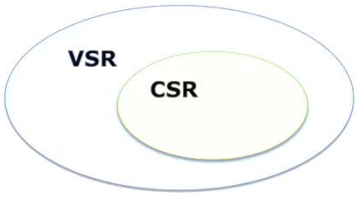
3. Ho un ciclo! **Non è CSR.**

Esempi di calcolo della CRS

S = $r_1(x), r_2(x), r_1'(x), w_2(x)$

1. Conflitti: $\{(r_1, w_2) = t1 \rightarrow t2, (r_1', w_2) = t1 \rightarrow t2\}$
2. Grafo ACICLICO: **è CSR.**

Relazione fra VSR e CSR?



Non sono concetti equivalenti ma sono legati fra loro. La conflict-serializzabilità è **condizione sufficiente ma non necessaria** per la view-serializzabilità.

Dimostrazione:

VSR ! ⟹ CSR	*CSR ⟹ VSR*
Controesempio: schema CSR ma non VSR S = $r_1(x), w_2(x), w_1(x), w_3(x)$ • View-serializzabilità: sì › LEGGE_DA = ∅, SCR_FN = $\{w_3(x)\}$ › T1T2T3== $r_1(x), w_1(x), w_2(x), w_3(x)$ è VSR • Conflict-serializzabilità: no › Conflitti: = $\{(r_1, w_2) = t1 \rightarrow t2, (w_2, w_1) = t2 \rightarrow t1, (w_2, w_3) = t2 \rightarrow t3, (w_1, w_3) = t1 \rightarrow t3, (r_1, w_3) = t1 \rightarrow t3\}$	Per assurdo: ipotizziamo $\approx_c$ e dimostriamo $\approx_v$ • Stesse scritture finali: se così non fosse, ci sarebbero almeno due scritture sulla stessa risorsa in ordine inverso, e poiché due scritture sono in conflito i due schedule non sarebbero $\approx_c \rightarrow \perp$ • Stessa LEGGE_DA: se così non fosse ci sarebbero scritture in ordine diverso o coppie scrittura-lettura in ordine diverso e i due schedule non sarebbero $\approx_c \rightarrow \perp$

Esercizi: classificare se CSR e VSR

1. $r_1(x), w_1(x), r_2(z), r_1(y), w_1(y), r_2(x), w_2(x), w_2(z)$

VSR:

1. Costruisco gli insiemi
   a. LEGGE_DA: $\{(r_2(x), w_1(x))\}$
   b. SCR_FIN: $\{w_2(x), w_1(y), w_2(z)\}$
2. Due transazioni:
   a. T1T2 = $r_1(x), w_1(x), r_1(y),\ w_1(y), r_2(z), r_2(x), w_2(x), w_2(z)$
      i. LEGGE_DA = $\{(r_2(x), w_1(x))\}$
      ii. SCR_FIN = $\{w_2(x), w_1(y), w_2(z)\}$

      → È VSR
   b. T2T1 = $r_2(z), r_2, w_2(x), w_2(z), r_1(x), w_1(x), r_1(y), w_1(y)$

CSR:

1. Costruisco insieme dei conflitti:
   Conflicts = $\{(r_1(x), w_2(x)) = t1 \rightarrow t2, (w_1(x), r_2(x)) = t1 \rightarrow t2, (w_1(x), w_2(x)) = t1 \rightarrow t2\}$
2. È aciclico? ✓

   → È CSR

$r_1(x), w_1(x), w_3(x), r_2(y), r_3(y), w_3(y), w_1(y), r_2(x)$

VSR:

1. Insiemi
    a. LEGGE_DA: $\{(r_2(x), w_3(x)\}$
    b. SCR_FIN: $\{w_3(x), w_1(y)\}$
2. Permutazioni → 6 possibili schedule seriali :(
   Per non doverli generare tutti, possiamo scartare a priori alcuni schemi. Ad esempio, sappiamo che vogliamo ottenere un insieme LEGGE_DA fatto da una lettura $r_2(x)$ che legge su una scrittura $w_3(x)$. Ci serve quindi uno schedule seriale dove la T3 PRECEDE la transazione T2.
   Inoltre, per evitare che si formino altre coppie in LEGGE_DA, scartiamo tutti quelli schedule in cui T1 precede T3
    a. T1T2T3, T1T3T2 = scarto per il secondo motivo
    b. T3T1T2, T3T1T2 = scarto per il primo motivo
    c. T2T1T3 = scarto per il secondo motivo
    d. T2T3T1 = scarto per il primo motivo

→ Non è VSR, quindi non è nemmeno CSR.

3. $r_1(x), r_2(x), w_2(x), r_3(x), r_4(z), w_1(x), w_3(y), w_3(x), w_1(y), w_5(x), w_1(z), w_5(y), r_5(z)$

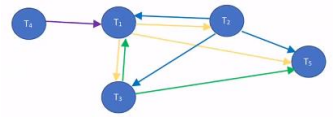
CSR

$r_1(x), r_2(x), w_2(x), r_3(x), r_4(z), w_1(x), w_3(y), w_3(x), w_1(y), w_5(x), w_1(z), w_5(y), r_5(z)$

1. Conflitti

$\{(r1(x), w_2(x)), (r_1(x), w_3(x)), (r_1(x), w_5(x)),$ $(r_2(x), w_1(x)), (r_2(x), w_3(x)), (r_2(x), w_5(x)),$ $(w_2(x), r_3(x)), (w_2(x), w_1(x)), (w_2(x), w_3(x)), (w_2(x), w_5(x)),$ $(r_3(x), w_1(x)), (r_3(x), w_5(x)),$ $(r_4(z), w_1(z)),$ $(w_1(x), w_3(x)), (w_1(x), w_5(x),$ $(w_3(y), w_1(y)), (w_3(y), w_5(y)),$ $(w_3(x), w_5(x),$ $(w_1(y), w_5(y)),$ $(w_1(z), r_5(z))\}$ (ma porca puttana)	$t1 \rightarrow t2, \sout{t1 \rightarrow t3}, t1 \rightarrow t5$ $t2 \rightarrow t1, t2 \rightarrow t3, t2 \rightarrow t5$ $t2 \rightarrow t3, t2 \rightarrow t1, t2 \rightarrow t3, t2 \rightarrow t5$ $\sout{t3 \rightarrow t1}, t3 \rightarrow t5$ $t4 \rightarrow t1$ $t1 \rightarrow t3, t1 \rightarrow t5$ $t3 \rightarrow t1, t3 \rightarrow t5$ $t3 \rightarrow t5$ $t1 \rightarrow t5$ $t1 \rightarrow t5$

2. Grafo!
   È ciclico. **NON CSR**



VSR

$r_1(x), r_2(x), w_2(x), r_3(x), r_4(z), w_1(x), w_3(y), w_3(x), w_1(y), w_5(x), w_1(z), w_5(y), r_5(z)$

1. LEGGO_DA: $\{(r_3(x), w_2(x)), (r_5(z), w_1(z))\}$
   SCR_FIN: $\{w_5(x), w_5(y), w_1(z)\}$

2. PERMUTAZIONI

    a. Scrematura per **scr_fin**:
    Ultima scrittura su y : 5 → tutte quelle che scrivono su y verranno prima→ **T1,T3<T5**
    Ultima scrittura su x : 5 →**T3,T1,T2 < T5**
    Ultima scrittura su z : 1 → nessun interessato :(

    b. Scremura per **leggo_da**:
    # Non spiega perché, ma T2<T3, T1<T5

A questo punto valutiamo le relazioni legge_da che potrebbero generarsi. Consideriamo le transazioni t_1 e t_2.

t_1: $r_1(x)$, $w_1(x)$, $w_1(y)$, $w_1(z)$
t_2: $r_2(x)$, $w_2(x)$

Poiché e richiesto che $t_2 < t_3$, ipotizziamo di considerare il caso in cui anche $t_1 < t_3$.
Se $t_1 < t_2$ allora si genera la legge_da: $(r_2(x), w_1(x))$, relazione non presente nell'insieme LEGGE_DA(S_3).
Se invece $t_2 < t_1$ si genera la legge_da: $(r_1(x), w_2(x))$, relazione non presente nell'insieme LEGGE_DA(S_3). In quest'ultimo caso, fra l'altro si perde la relazione $(r_3(x), w_2(x))$.

Poiché in qualsiasi permutazione o $t_1 < t_2$ o $t_2 < t_1$ nessun schedule seriale può presentare lo stesso insieme LEGGE_DA di S_3. Quindi **S_3 non è VSR**. Pertanto **è nonSR**.

Nei sistemi reali, in realtà, vengono utilizzate tecniche diverse, che non richiedono di conoscere l'esito delle transazioni:

TS Buffer

Timestamp con scritture bufferizzate. Le risorse mantengono le risorse per un certo lasso di tempo, scaduto il quale le devono rilasciare

Locking a due fasi

È il metodo applicato nei sistemi reali per la gestione dell'esecuzione concorrente di transazioni. Il suo vantaggio principale è che non richiede di conoscere in anticipo l'esito delle transazioni. Si basa su tre aspetti:

- Meccanismo di gestione dei lock per le varie transazioni
- Politica di concessione dei lock sulle risorse
- Regola che garantisce la serializzabilità.

Meccanismo di base

Si fonda sull'introduzione di tre **primitive di lock**, che consentono alle transazioni di bloccare le risorse sulle quali vogliono agire in scrittura o lettura. Esse sono:

- **r_lock$_K$(x):** richiesta di un lock condiviso da parte della transazione t_k sulla risorsa x per **lettura**.
- **w_lock$_K$(x):** richiesta di un lock condiviso da parte della transazione t_k sulla risorsa x per **scrittura**.
- **unlock$_K$(x):** richiesta da parte della transazione t_k di liberare la risorsa x da un precedente lock.

Regole delle primitive:

1. Ogni lettura deve essere preceduta da un r_lock e seguita da un unlock.
   Sono ammesis più r_lock contemporanei sulla stessa risorsa → **lock condiviso**
2. Ogni scrittura deve essere preceduta da un w_lock e seguita da un unlock.
   Non sono ammessi più w_lock né w_lock – r_lock in contemporanea → **lock esclusivo**.

Una transizione che segue queste due regole è detta **ben formata rispetto al locking**.

Politica di gestione dei LOCK

Per poter gestire i lock, è necessario che il gestore dei lock abbia accesso a due informazioni:

- **Stato della risorsa:** $s(x) \in \{libero, r\_lock\_w\_lock\}$
- **Transizioni in r_lock sulla risorsa:** $c(x) = \{t_1, \dots, t_n\}$ dove $\{t_1, \dots, t_n\}$ hanno un lock su x

Questo ci serve perché, a seconda dello stato e delle transizioni in r_lock, a fronte di una richiesta posso avere esiti diversi (concesso, attesa) e vengono eseguite operazioni opportune.

Stato x / Richiesta	LIBERO		R_LOCK		W_LOCK	
$r\_lock_K(x)$	Esito OK	Operazioni $s(x) = r\_lock$ $c(x) = \{k\}$	Esito OK	Operazioni $c(x) = c(x) \cup \{k\}$	Esito Attesa	Operazioni -
$w\_lock_K(x)$	Esito OK	Operazioni $s(x) = w\_lock$	if $\lvert c(x)\rvert=1$ and $k \in c(x)$ then: Esito OK	Operazioni $s(x) = w\_lock$	Esito Attesa	Operazioni -
			else Attesa	-		
$unlock_K(x)$	Esito Errore	Operazioni -	Esito OK	Operazioni $c(x) = c(x) - \{k\}$ if $c(x)=\emptyset$ then $s(x)$=Libero verifica coda	Esito OK	$c(x)=\emptyset$ $s(x)$=Libero verifica coda

[L29 – 24:45]

→ va bene solo se la cosa che chiede il write è la stessa che chiede il write

Regole che garantiscono serializzabilità e commit-proiezione

PL-Serializzablilità:

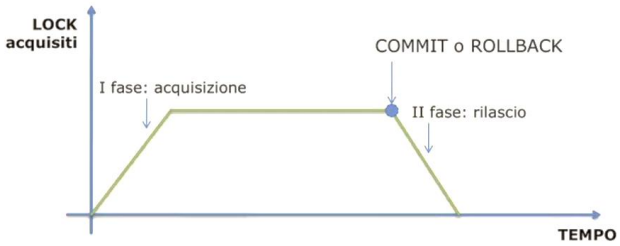
Una transazione dopo aver rilasciato un LOCK non può acquisirne altri. → si chiama locking a due fasi perché ciascuna transazione deve eseguire due fasi:

- Acquisizione di tutti i lock di cui ha bisogno
- Rilascio di tutti i lock di cui ha bisogno: non può riprenderne altri.

Quindi, anche se all'interno avrebbe finite con una risorsa, il rilascio avviene solo alla fine.

Commit-proiezione:

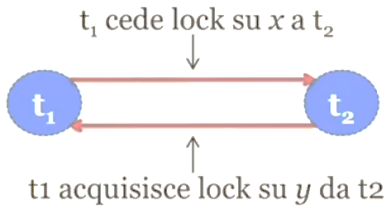
Una transazione può rilasciare I LOCK solo quando ha eseguito correttamente un COMMIT o un ROLLBACK. Questa regola aggiuntiva viene detta 2PL STRETTO



Relazione fra CSR s 2PL

Dimostrazione: se uno schema è 2PL allora è anche conflict-serializzabile.

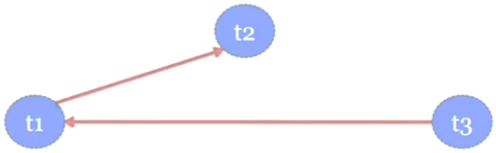
$s \in 2PL \implies s \in CSR$	$CSR \; ! \Longrightarrow 2PL$
Per assurdo: ipotizziamo s ∈ 2PL ⟹ s ∉ CSR Se s ∉ CSR, allora esiste una coppia di conflitti ciclici	Mi basta dimostrare che esistono degli schedule CSR ma non 2PR. Prendiamo lo schedule $r_1(x) w_1(x) r_2(x) w_2(x) r_3(y) w_1(y)$ rilascia lock x acquisisce lock y • Non è PL, perché acquisisce t1 un lock dopo averne rilasciato un altro



Esempio:$w_1(x)r_2(x)w_2(x)w_2(x)w_2(y)r_1(y)$

Ma allora significa che ad un certo punto t1 avrà ceduto un lock sulla risorsa x a t2! E t1 deve aver acquisito un lock su y in t2. Quindi non è PL. ⊥

- È CSR! Conflitti:
  $\{(r_1(x), w_2(x)), (w_1(x), r_2(x)), (w_1(x), w_2(x)), (r_3(y), w_1(y))\}$



Esempio di perdita di aggiornamento

[L29, 45:44]

t1	M.C.	Database			M.C.	t2
		c(x)	s(x)	Val		
bot		∅	L	2		
$r\_lock_1(x)$		{1}	RL	2		
$r_1(x)$	2	{1}	RL	2		
x = x +1	3	{1}	RL	2		
	3	{1}	RL	2		bot
	3	{1,2}	RL	2		$r\_lock_2(x)$
	3	{1,2}	RL	2	2	$r_2(x)$
	3	{1,2}	RL	2	3	x=x+1
	3	{1,2}	RL	2	3	$w\_lock_2(x)$ → ATTESA
$w\_lock_1(x)$ → ATTESA	3	{1,2}	RL	2	3	

BLOCCO CRITICO

Blocco critico

Il blocco critico è una situazione che si realizza quando più transazioni sono in attesa sulle stesse risorse, ovvero:

- Due transazione hanno bloccato delle risorse: t1 blocca r1 e t2 blocca r2
- t1 è in attesa sulla risorsa r2
- t2 è in attesa sulla risorsa r1

Se il numero medio di tuple per una tabella è n e assumiamo che la granularità del lock è la tuple, la probabilità che si verifichi un lock fra due transazioni è

$$P(deadlock\ di\ lunghezza\ 2) = 1/n^2$$

Risoluzione del blocco critico

- **Timeout**
  È la più semplice; quando una transazione entra in blocco critico, viene fatto partire un contatore. Se a fine countdown la risorsa non è stata acquisita l'operazione viene abortita, cioè si esegue il rollback e l'altra transazione può procedere.
- **Prevenzione**
  Una transazione blocca tutte le sue risorse in una sola volta. La transazione acquisisce un timestamp, e attende una certa risorsa solo se il suo timestamp è minore di quello dell'altra transazione; altrimenti viene abortita e fatta ripartire.
- **Rilevamento**
  Si esegue un'analisi periodica della tabella di lock, per rilevare eventuali blocchi critici. Questo è quello che succede nei sistemi reali :)

Starvation

È meno probabile del blocco critico. Deriva dal fatto di gestire i lock in lettura come lock condivisi, ma permettere la scrittura solo quando tutti sono stati rilasciati.

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

Viene gestito dal DBMS nello stesso modo del blocco critico: si analiza la tabella delle relazioni di attesa,e si verifica d quanto tempo el transazioni stanno attendendi la risorsa e di conseguenza si sospende temporaneamente la concessione di lock in lettura condivisi per consentire la scrittura da parte della transazione in attesa.

Livelli di isolamento

Poiché risulta molto oneroso per un DBMS gestire l'esecuzione concorrente, SQL prevede la possibilità di rinuncare in tutto o in parte al controllo di concorrenza per aumentare le prestazioni del sistema. Ciò significa che è possibile, a livello di singola transazione, decidere di tollerare alcune anomalie.

Quello che abbiamo visto è la serializzabilità, ovvero il livello dove tutte le anomalie sono evitate.

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

Quello che viene garantito sempre è la non perdita di update: tutti i livelli applicano 2PL per le scritture, mentre progressivamente rilasciano il controllo sulle operazioni di lettura.

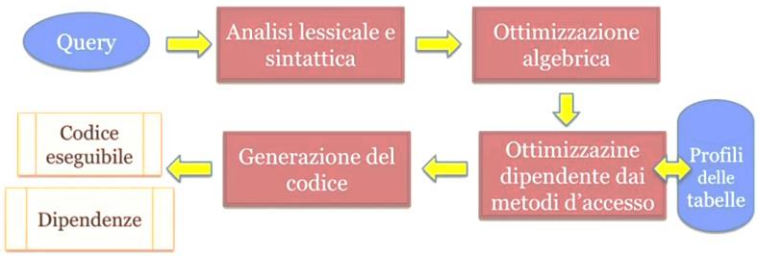
- **serializzabile**: 2PL richiesto anche in lettura
- **repeatable read**: 2PL stretto nelle letture di tupla e non di tabella(= posso leggere la singola tupla, ma se faccio un'operazione su più righe (es. AVG) allora si applica il lock); ovvero consente inserimenti e non evita l'anomalia di inserimento fanasma
- **read committed**: in lettura richiede lock condivisi, ma non il 2PL stretto
- **read uncommitted**: non applica alcun lock in lettura.

Ottimizzazioni

Le interrogazioni sottomessa un DBMS sono sottomesse con un linguaggio dichiarativo, ovvero dove dichiaro la forma del risultato ma non do indicazioni su come esse siano prodotte. Devo dunque tradurre da SQL a un piano di esecuzione in un **linguaggio procedurale**.

Il risultato finale deve essere ottimizzato rispetto alle caratteristiche del DBMS a livello fisico (= metodi di accesso disponibili) e alla base di dati corrente (=statistiche del dizionario dei dati).

La compilazione di una query si compone di:



- **Analisi lessicale e sintattica**
- **Ottimizzazione algebrica** (=in base alle caratteristiche del DBMS)
- **Ottimizzazione basata sui metodi di accesso, sui costi di esecuzione** (=anche in base al riempimento delle tabelle)

Ottimizzazione algebrica

Già vista col Belussi! ^-^ Si basa fondamentalmente sulle regole dell'algebra relazionale:

- Anticipo delle selezioni → **selection push**
- Anticipo delle proiezioni → **projection push**

Ottimizzazione basata sui metodi di accesso

Le operazioni tipicamente supportate dai DBMS e che richiedono ottimizzazione sono:

- Scansione delle tuple di una relazione
- Ordinamento di un insieme di tuple
- Accesso diretto alle tuple via indice
- Implementazioni diverse del join

Scansione

Un'operazione di scansione opera contestualmente ad altre operazioni. Alcune varianti possibili sono:

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

Il costo di una scansione sulla relazione R è $NP(R)$, ovvero il numero Pagine dati della relazione R.

Il costo rimane invariato anche eseguendo l'elenchino sopra.

Ordinamento

L'ordinamento è utile per:

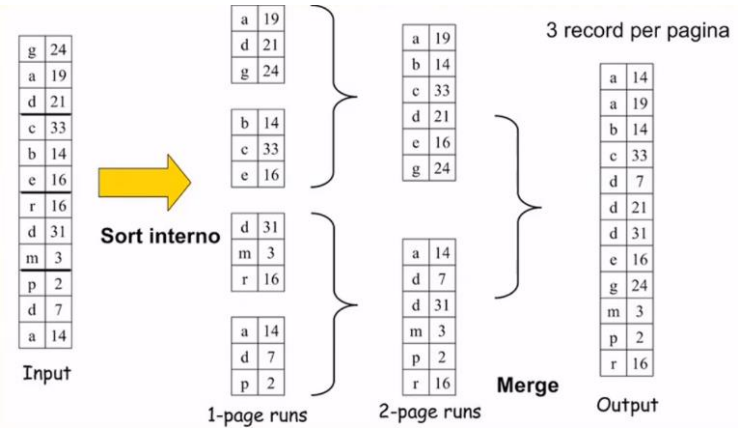
- Ordinare il risultato di un'interrogazione, con **ORDER BY**
- Eliminare duplicati, con **SELECT DISTINCT**
- Raggruppare tuple, con **GROUP BY**

Ordinamento su memoria secondaria: Z-way-Sort-Merge

Si compone di due fasi:

1. **Sort interno**: si leggono una alla volta le pagine della tabella; le tuple di ogni pagina vengono ordinate facendo uso di un algoritmo di sort interno tipo Quick Sort. Ogni pagina così ordinata, detta anche **run**,viene scritta su memoria secondaria in un file temporaneo
2. **Merge**: applicando uno o più passi di fusione, le run vengono unite fino ad arrivare ad avere una run unica.

Esempio



Supponiamo di dover ordinare un input che consiste di una tabella di NP pagine, e di avere a disposizione solo NB buffer in memoria centrale, con NB < NP. Per semplicità supponiamo il caso base a due vie (Z = 2) e supponiamo di avere a disposizione solo 3 buffer in memoria entrale (NB = 3)

→ Nel passaggio fra primo e secondo livello di run succede il casuno: una run è più grande di una pagina e io ho solo trepagine di butter a disposizione! Pescherà **la prima pagina della prima run e la prima pagina della seconda run.**

La Z significa quante run si fondono n ogni volta.

Il buffer significa che ciascun buffer viene associato ad una run, e il terzo buffer viene usato per produrre l'output. Quando il buffer di output è pieno lo scrivo sul disco (=con due pagine dovrò fare 2 giri! Perché ho un solo buffer di output).

Costo

Consideriamo come costo il numero di accessi alla memoria secondaria. Nel caso base di Z = 2 e NB = 3:

- Nella fase di sort interno **si leggono e riscrivono NP pagine**
- Ad ogni passo di merge si leggono e si scrivono NP pagine+il numero di passi di merge è pari a $\lceil \log_2(NP) \rceil$, in quanto ad ogni passo il numero di run si dimezza.

Il costo complessivo, dunque, è $2 \times NP \times (\lceil \log_2 NP \rceil + 1)$ → Due volte in quanto lettura + scrittura.

Accesso diretto alle tuple via indice

Il modo in cui viene fatto dipende dal tipo di selezione, e dal tipo di indice: B+ tree e hash.

A = v	BETWEEN v1 AND v2	A = v1 AND a = v2	A = v1 OR B = v2
richiede hash o B+ tree	richiede B+ tree	si sceglie per quale delle condizioni di uguaglianza utilizare l'indice; la scelta ricade sulla condizione più selettiva. L'altra si verifica direttamente sulle pagine dati	possiamo usare più indici in parallelo, facendo un merge dei risultati eliminandoi duplicati, oppure (se manca anche solo uno degli indici) è necessario eseguire una scansione sequenziale.

Es. se ho query con OR e prestazioni molto basse, probabilmente mi sono dimenticato di definire un indic su una delle due. Postgres definisce in automatico un indice sulla chiave primaria proprio perché essa è utilizzata molto spesso nel JOIN!

JOIN

È l'operazione più costosa, poiché la dimensione delle tabelle da maneggiare può esplodere.
Ne esistono varie implementazioni:

- Nested loop join
- Merge scan join
- Hash-based join

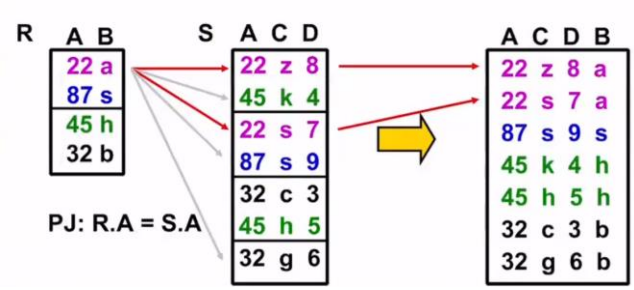
Benché dal punto di vista logico-teorico il join è un'operazione commutativa, dal punto di vista fisico del sistema reale l'ordine di dichiarazione influenza le prestazioni :(
Ecco perché, quando vediamo gli algoritmi di join, distinguiamo **l'operando sinistro** → **outer** dall'operando destro→ **inner**.

Nested loop join

Supponiamo di avere due relazioni R → esterna e S → interna fra i quali definiamo un predicato di join PJ.
Nel nested loop join definiamo il join come due cicli annidati; prima iteriamo su tutte le tuple esterne R e per ciascuna tupla di questa relazione esterna iteriamo su tutte le tuple della relazione esterna. Se qualcuna di queste soddisfa il predicato di join, allora aggiungiamo al risultato.

$$
\begin{aligned}
&Per\ ogni\ tupla\ t_r in\ R\{\\
&\quad Per\ ogni\ tupla\ t_s in\ S\{\\
&\quad\quad Se\ la\ coppia\ (t_r, t_s) soddisfa\ PJ\\
&\quad\quad\quad allora\ aggiungi\ (t_r, t_s) al\ risultato.\\
&\quad\}\\
&\}
\end{aligned}
$$

Cioè: confronto la prima riga di R con tutte le righe di S, poi la seconda di R con tutte le righe di S e così via.



Costo

Dipende da quanto spazio a disposizione c'è nel buffer; più le tabelle sono grosse (=non ci stanno nel buffer), peggio è.

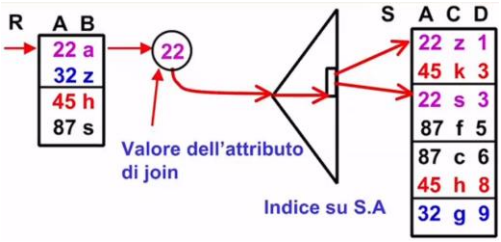
- Caso base: 1 buffer per R e 1 buffer per S
  - Leggo 1 volta R
  - Leggo $NR(R)$ volte S, ovvero tante volte quante sono le tuple di R
  - → Totale: $NP(R) + NR(R) \times NP(S)$ accessi in memoria (NP sarebbe pagine, NR sarebbe righe)

Se è possibile allocare $NP(S)$ buffer per la relazione interna, ovvero posso caricare tutta S, allora mi basta $NP(R) + NP(S)$

Posso migliorare questa prestazione facendo ricorso a degli indici (god save us).

Block nested loop join → nested loop join con indice B+ tree

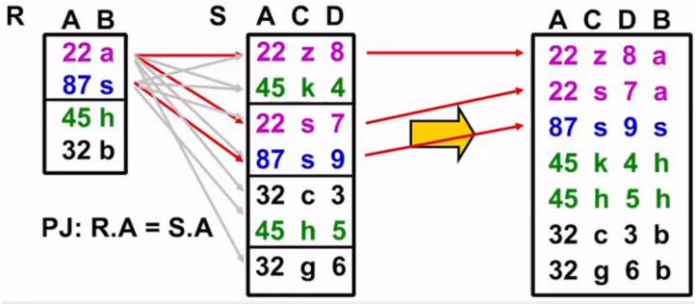
La relazione R va sempre letta tutta, ma possiamo migliorare gli accessi a S a patto che gl indici di S siano definiti sugli attributi che usiamo per fare la join. Data una tupla della relazione esterna R, la scansione completa della relazione interna S può essere sostituita da una scansione basata su un indice costruito sugli attrbuti di join di S.



Costo

Caso base: 1 buffer per R e 1 buffer per S

- Leggo 1 volta R
- Accedo $NR(R)$ volte a S, ovvero tante quante sono le tuple della relazione esterna
- → Totale: $NP(R) + NR(R) \times (Profondità\ Indice + NR(S)/\ VAL\ (A,S))$
  La selettività (NR/VAL) dell'attributo A mi dice in media quanti valori distinti dell'attributo A compaiono nella relazione S; quindi spiega (in media) per ciascun attributo A quante righe devo caricare in media. Più è selettivo, meno sono le righe che devo caricare per ciascun valore distinto di A. L'idea è che comunque la profondità dell'indice e la selettività dovrebbero essere molto minori di NR(S)*NP(R).



Es. dato il valore 22 carico solamente le righe che corrispondono a 22 (mentre nel caso senza indice, qui in grigio, dovrei confrontare con tutte le righe!)

L'unico modo che abbiamo di ridurre le letture della tabella esterna è di fare una lettura bufferizzata, ovvero invece di caricare una riga carico un buffer e lo mantengo in memoria (leggendo quindi in buffer e non in memoria secondaria).

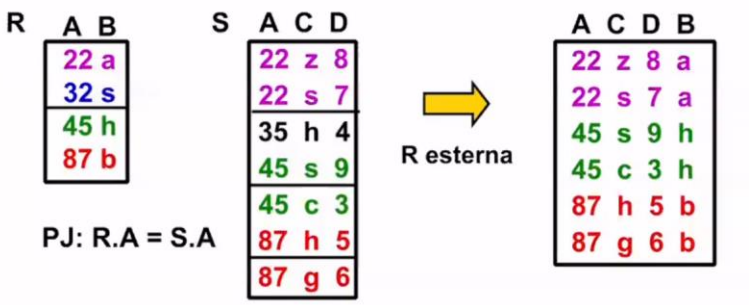
In conclusione, il costo di esecuzione dipende fortemente dallo spazio a disposizione nei buffer: se è sufficiente a caricare tuttoil contenuto è molto efficiente, ma andando avanti è semrpe peggio.

Merge-scan JOIN

Si applica soltanto quando:

- Il join è un **equijoin**.
- entrambe le relazioni in input sono **ordinate sugli attributi su cui è definito il join**, ovvero:
  - La relazione è fisicamente ordinata sugli attributi di join (file sequenziale ordinato come struttura fisica)
  - Esiste un indice sugli attributi di join della tabella che consente una scansione ordinata delle tuple.

Esempio:



La logica dell'algoritmo è di scansionare contemporaneamente le due tabelle:

- Parto da R
- Scansiono S finché trovo tuple con lo stesso valore (o superiore) delle prime tuple di R
- Finite le righe di S interessate, passo alla successiva riga di R

La logica dell'algoritmo sfrutta il fatto che entrambi sono ordinati, dunque non fa scansioni inutili

Costo: $NP(R) + NP(S)$

Se ho un indice B+ tree su entrambe le relazioni, e gli indici sono già nel buffer, il minimo corso è $NR(R) + NR(S)$

Se invece ho un solo indice, $NP(R) + NR(S)$ $o\ viceversa$.

Hash-based JOIN

- **Si può applicare solo nelle equi-join.**
- Non richiede né la presenza di indici né che sia ordinato; risulta utile per relazioni molto grandi.

Come intuibile si basa sul concetto di hashing, e in particolare si ipotizza che prendendo due tuple, esse parteciperanno alla relazione se e solo se i loro valori di hash sono uguali; se l'hash è diverso sicuramente non parteciperanno. Partendo da questa idea ci sono diverse implementazioni: l'idea fondamentale è che partiziono le mie relazioni in base alle hash. Faccio una ricerca di "matching tuples": la ricerca avviene solo per quelle partizioni che hanno lo stesso valore di hash. → **AL più posso avere falsi positivi, ma se i valori di hash sono diversi le tuple hanno un valore certamente diverso**.

Costo:

A. **Costruzione della hashmap**: dovrò scansionare entrambe le relazioni per intero. $\boldsymbol{NP(R) + NP(S)}$
B. **Accesso alle matching tuples**, che nel caso pessimo (=tutte le tuple vanno a finire nellos tesso valore di hash) costa quanto il block neasted loop join: $\boldsymbol{NP * NP(S)}$

Ma dipende fortemente dal numero di buffer e dalla bontà della funzione di hash, o dalla distribuzione dei valori degli attributi.

Scelta del piano di esecuzione

- **Genera tutti i possibili piani di esecuzione** (alberi), o un sottoinsieme se è possibile scartarne alcuni. Si considerano:
  - Operatori alternativi applicavili per l'accesso ai dati, tipo scan sequenziale vs. indice
  - Operatori alternativi applicabili nei nodi, tipo nested-loop join vs. hash-based join
  - Ordine delle operazioni da compiere (associatività)
- **Valuta con *formule approssimate* il costo di ogni alternativa in termini di accessi a memoria secondaria richiesti, scegliendo l'albero con costo minore**

Data dictionary

Per generare la stima di costo deve valutare il costo associato ad una certa relazione, considerando anche il profilo (?) delle relazioni. Per fare questo usa un **dizionario dei dati**, Data Dictionary, **che contiene per ciascuna relazione T:**

CARD(T) stima della cardinalità (numero di tuple)	→ Se le tabelle hanno poche righe, potrebbe essere vantaggiorso accedere direttamente anziché via indici
SIZE(T) stima della dimensione di una tupla	→ Se ho tantissime colonne, magari non mi conviene caricare molto in memoria
VAL(A,T) stima del numero di valori distinti per ogni attributo A	→ Posso capire se usare un hash oppure no; se non è molto selettivo è meglio fare divresamente
MAX(A,T), MIN(A,T) Stima del valore massimo e minimo per ogni attributo A	→ Stima di quante saranno le tuple che appaiono nel risultato

Esercizio su ottimizzazione e stima del costo

L'unità di misura del costo sono gli accessi a memoria secondaria, poiché altre operazioni (es. confronti) non risultano davvero influenti.

Calcolare il costo in caso di:

- La selezione su ricetta richiede una scansione sequenziale della tabella $RICETTA$
- L'ordine di esecuzione del join è $(RICETTA \bowtie COMPOSIZIONE) \bowtie INGREDIENTE$
- Le operazioni di join vengono eseguite come Nested Loop Join, con una pagina di join per ciascuna tabella
- Il risultato intermedio del primo join viene memorizzato nel buffer

TABELLA	NP	NR	VAL(Regione)	VAL(Ricetta)
RICETTA	12	260	20	
INGREDIENTE	40	1200		
COMPOSIZIONE	200	13000		260

Trovare gli ingredienti usati in ricette della Regione Veneto, riportando, il codice della ricetta e il nome e le calorie dell'ingrediente.

```
SELECT   R.CodiceRicetta, I.Nome, I.Calorie

FROM     RICETTA R JOIN COMPOSIZIONE C ON R.CodiceRicetta = C.Ricetta
                   JOIN INGREDIENTE I ON C.Ingrediente = I.Codice

WHERE R.Regione = 'Veneto'
```

1. **Join fra ricetta(con selezione) e composizione**
   $= NP(RICETTA) + NR(RICETTA\ con\ selezione\ regione = veneto) * NP(COMPOSIZIONE)$
   $= 12 + NR(RICETTA)/VAL(Regione, RICETTA) * 200$
   $= 12 + 260\ /\ 20\ * 200\ = 2612$
   [DEVE LEGGERE TUTTE LE RIGHE DI RICETTA, E POI SOLO NELLA SECONDA PARTE APPLICA LA SELEZIONE – 53:31 – PERCHé???]
2. Risultato precedente in JOIN con ingrediente
   Abbiamo ipotizzato che il risultato del primo join sta in un buffer; questo significa che il suo costo sarà 0!
   $= 0 + (stima\ delle\ tuple\ di\ composizione\ che\ rappresentano\ ingredienti\ di\ ricette\ del\ veneto) * 40$
   $= (numero\ medio\ di\ tuple\ di\ COMPOSIZIONE\ epr\ ricetta) * (numero\ di\ ricette\ del\ Veneto) * 40$
   $= (NR(COMPOSIZIONE)\ /\ VAL(ricetta, COMPOSIZIONE)\ * 13 * 40 =$
   $= (13000 * 13) * 40 = 26000$

Alternativa or smth

COSTO TOTALE = 28612

(2) Come cambia il costo se è disponibile un indice B+-tree sull'attributo Codice della tabella INGREDIENTE che ha profondità 2.

COSTO A = come prima

COSTO B = (**nessun costo di lettura della tabella esterna: è già nel buffer**) +
NR(**COMPOSIZIONE per le ricette con selezione Regione='Veneto'**) *
(**Profondità Indice + selettività di Codice in INGREDIENTE**)

COSTO A = **2612**

COSTO B = **(13000/260 * 13) * (2 + 1) = 650 * 3 = 1950**

COSTO TOTALE = 4562

→ la selettività diventa 1 in quanto è la pchiave primaria /superchiave.

Interazione tra basi di dati e applicazioni

Vi sono diversi tipi di interazione:

- **Interazione via cursore (classica):** abbiamo una API messa adisposizione dal DBMS che permette di mandare comandi SQL da un programma e ricevere le tabelle sotto forma di cursore, ovvero un iteratore sulle tuple del risultato.
- **Interazione via cursore con API standardizzata:** indipendente dal DBMS, si usano gli stessi metodi (alcuni esempi sono JDBC di Java o ODBC di Microsoft).
- **Object Relational Mapping (ORM):** è una tecnica che consente di estire nell'applicazione oggetti "persistenti", e astrarre la base di dati relazionale. Ad esempio Java Persistece API o Hibernate.

Esistono poi altre evoluzioni in diverse direzioni per tentare di superare i problemi legati al cosiddetto object-relational impedance mismatch.
Lato DBMS:

- **Object-relational model (SQL3):** esistono basi di dati SQL che anziché memorizzare le tuple permettono di utilizzare un SQL esteso dove ho una struttura gerarchica e sono oggetti.
- **Basi di dati semistrutturate:** hanno basi di dati più complessa e a schema variabile, tipo XML o JSON.
- **Document-based models** (NoSQL, visti col B elussi??) : collezioni di documenti con struttura complessa, dati ridondanti e voluminosi.
- **Sistemi basati su cluster per big data:** collezione di dati a struttura complessa, variabile e ridondante, gestiti da sistemi diversi, dati voluminosi, interogazioni distribuite ed eseguite in parallelo.

  → Le ultime due sono pensate ed efficaci per dati enormi e che cambiano poco nel tempo (es. files di log, internet of things), aka tanti dati no update se non incrementali.

Interazione via cursore in JDBC

Usando JDBC possiamo usare delle implementazioni standard per mandare query in SQL a un database, indipendente dal DBMS scelto. Il driver JDBC è un modulo software che traduce i metodi della classe JDBC in comandi SQL (in accordo con il DBMS scelto).

Come usarlo?

1. Caricare il driver JDBC per il DBMS che si utizza, casicando la classe corrispondente.
   Class.forName("nomedriver") , o nel caso di postgres **org.postgresql.Driver**
2. Definisco una stringa di connessione che specifica tutto quello che serve: `include tipo di DBMS, indirizzo del DBMS server e il nome del database.`
   String URL = "jdbc:postgresql://dbserver/did2014"
3. Stabilisco una connessione istanziando l'oggetto Connection, che necessita di URL, username e password
   String user = "username"
   String passwd = "password"
   Connection con = DriverManager.getConnection(URL,user,passwd);
4. Attraverso l'oggetto Connection posso sottomettere dei comandi, oggetti di tipo Statement
   Statement stat = con.createStatement();
5. Eseguo un'interrogazione SQL attraverso lo statement. pongo la query dentro una stringa e la submitto.
   a. **Interrogazione**: ritorna un insieme di tuple risultato
      String query = "SELECT * FROM PERSONA";
      ResultSet res ? stat.executeQuery(query);

b. **Aggiornamento**: ritorna un true/false
```
String update "UPDATE PERSONA"+"SET NOME = Rosa" WHERE id = 1";
òstat.executeUpdate(update)
```

In realtà esiste una variante a questi comandi, che è l'utilizzo di un Prepared Statement, utile quando una query deve essere eseguita più volte. Tale classe permette di fare un'interrogazione parametrizzata, valorizzandoli con specifici metodi.

```
Connection con = DriverManager.getConnection(URL,user,password)
String q = "SELECT nome, cognome" + "FROM persona" + WHERE id = ?";
PreparedStatement pstat = con.prepareStatement(q);
pstat.setInt(1,15);
ResultSet res = pstat.executeQuery();
```

6. Il result set è in realtà un puntatore alla tupla corrente e che itera sulle tuple. Per poter scandire il risultato ottenuto devo invocare il metodo next(), che mi sposta alla tupla successiva.
```
while(res.next())
```
Ci sono vari metodi che dato l'oggetto ritornato da next permette di recuperare i singoil valori delle tuple:

   a. `getX(par)`, dove X è un tipo base di Java e par è un indice di posizione oppure il nome di un attrivuto della relazione risultato dell'interrogazione. Restituisce il valore in posizione par o dell'attributo di nome par.

   b. `wasNull`: si riferisce all'ultima invocazione di getX e restituisce true se il valore letto era uguale al valore nullo.

   c. … a lot more sulla documentazione.

7. Chiudere la connessione usando l'oggetto apposito.
```
con.close();
```

L'unità atomica di interazione è la transazione. Se volessi eseguire più query nella stessa transazione devo disabilitare l'autocommit ed eseguirlo manualmente (e mettere il rollback se ci sono errori!) al termine della mia sequenza di transazioni.

```
con.setAutoCommit(false);
…
con.commit()
con.setAutoCommit(true);
```

L'impostazione di default è che il risultato della query è trasferito in blocco nella mia memoria; ma può succedere che la mia memroia non sia sufficiente, e sia necessario dividerlo in blocchi di tuple. Per alterare tale situazione si usa il metodo setFetchRow(int rows) di Statement: avverrà in lotti di al massimo rows tuple, prevedendo quindi eventualmente più operazioni per scaricare tutto il risultato.

```
conn.setAutoCommit(false)
Statement st = conn.createStatement();
// attiviamo il cursore
st.setFetchSize(50);
ResultSet rs = st.executeQuery("SELECT * FROM mytable");
while (rs.next()){
        System.out.println("a row is here");
}
rs.close();
//disattiviamo il cursore
st.setFetchSize(0); // → default
```

È tutto molto manuale insomma, e devo gestire a mano anche casi dove in SQL avrei una tabella per ciascun esame; per trasformarlo in esami come proprietà dell'oggetto studente è un po' lunga.

Object Relational Mapping

È uno strato softare che si interpone fra l'applicazione e la base di date; recupera gli *oggetti persistenti*, che sono oggetti di java tipici mappati nella base di dati relazionale. Il come viene specificato in un file di mapping XML o direttamente nelle classi Java via annotazioni

Vantaggi	Svantaggi
+ Gestione automatica della persistenza: moltequery che andrebbero fatte a mano sono gestite in maniera trasparente. → aka, una volta caricato l'oggetto se i relativi campi cambiano nella DB vengono aggiornati anche nell'oggetto. + Navigazione più semplice delle proprietà + Alcune query sono più semplici: spesso basta un get, senza SQL	- Le interrogazioni più complesse – che non possono sfruttare i puntatori fra oggetti, tipo le join credo? – vanno comunque gestite in SQL. - Se le query hanno cardinalità elevata, potrei arrivare a dover caricare l'intera libreria in memoria! Ma si risolve circa: a. Le librerie hanno degli accorgimenti "lazy", per cui anziché caricare tutta la catena (es. studente + tutte le tabelle relative) si caricano solo quelle immediatamente collegate, mentre le altre solo se e quando sono richieste. b. C'è la cache.

Mapping XML

- Associazione di classe "Event" e tabella "EVENTS"
  <class name = "Events" table "EVENTS">
  </class>
- Dichiarazione di quale colonna della tabella contiene l'id degli oggetti
  <id name = "id" column = "EVENT_ID">
  </id>
- Dichiarazione di alcune proprietà
  <property name = "date" type = "timestamp" column = "EVENT_DATE"/>
  <property name = "title"/>

I sistemi di ORM definiranno automaticamente le conversioni.

Mapping attraverso annotazioni

```
Import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;
@Entity
@Table (name = "employee")
public class Employee implements Serializable{
        @Column ( name = "employee_name")
        private String employeeName;
}
```