Un Database è una collezione organizzata di dati. DBMS è l'acronimo di Database Managment System, gestiscono i dati e si appoggiano su una macchina. È un sistema di gestione di basi di dati, cioè un sistema software che garantisce collezione di dati grandi, condivise e persistenti.

L'**informazione** (elemento che consente di avere conoscenza) è un'entità astratta che sta alla base della comunicazione ed è un concetto primitivo.

Il **dato** (ciò che è immediatamente presente alla conoscenza) è un modo con cui rappresentiamo l'informazione.

Schema a cascata (Ciclo di vita di un sistema)

Il progetto di utilizzo di un DBMS richiede diversi passi:

1. Studio di fattibilità;
2. Analisi dei requisiti - lista di specifiche (funzionalità), ovvero quali sono le cose che il nostro committente si aspetta che il sistema faccia;
3. Progettazione concettuale - (informazioni) quando si usa la progettazione concettuale si lavora al livello di astrazioni con informazioni;
4. Progettazione logico/fisica - (dati) definire la struttura della tabella;
5. Implementazione;
6. Messa in opera.

Per l'appunto questo ciclo prende il nome di waterfall (a cascata).

Modello Entità-Relazione (ER)

È un modello per la rappresentazione concettuale dei dati.

- **Entità**: rappresentano classi di oggetti che hanno proprietà comuni ed esistenza autonoma. In uno schema, ogni entità ha un nome che la identifica univocamente, e viene rappresentano graficamente tramite un rettangolo con il nome dell'entità al suo interno;
- **Attributi**: le entità e le relazioni possono essere descritte usando una serie di attributi. Tutti gli oggetti della stessa classe entità hanno gli stessi attributi;
- **Relazione**: Dati due insieme F ed M $R(F, M) \subseteq F \times M$è l'insieme di tutte le possibili coppie di elementi contenuti a destra e a sinistra.

Cardinalità delle relazioni

Vengono specificate per ciascuna entità che partecipa a una relazione e descrivono il numero minimo e massimo di occorrenze di relazione a cui le occorrende delle entità coinvolte possano partecipare. Dicono quindi quante volte in una relazione tra entità, un'occorrenza di una di queste entità può essere legata a occorrenze delle altre entità coinvolte nella relazione.

Vincolo di integrità

È una proprietà che deve essere soddisfatta dalle istanze che rappresentano info correte per l'applicazione. Ogni vincolo può essere visto come un predicato che associa ad ogni istanza il valore vero o falso. Se il predicato assume il valore vero diciamo che l'istanza soddisfa il vincolo.

Schema relazionale

È Dato da un nome di relazione seguito da una lista di attributi ciascuno dei quali è un proprio dominio ovvero una lista di attributi da cui attingere. Quindi una relazione in una base di dati sarà indicata da:

$$R(A_1{:}D_1, A_2{:}D_2, \ldots, A_n{:}D_n) \rightarrow \text{schema relazionale}$$

Questa è la struttura che dovranno avere le tabelle e NON contiene le tuple.

Istanza di relazione

Uno schema relazionale può avere più istanze, ovvero una tabella fisicamente creata una o più volte.

r istanza di R: è una tabella popolata, ovvero un qualunque insieme di righe che è conforme allo schema Relazionale (R). N.B lo schema non è un insieme.

$$\text{r è un'istanza di } R(A_1,\ldots,A_m) \text{ se } r \subseteq D_1 \times D_2 \times \ldots \times D_n$$

r è un sottoinsieme del prodotto cartesiano dei domini delle tuple. Sottolineare gli attributi significa dire che non tutti i sottoinsiemi degli elementi possono essere presenti in r.

Schema di database

L'insieme degli schemi relazionali di database è detta schema di database:

$$R_1(A_1{:}\,D_1', \ldots, A_n{:}\,D_n')$$
$$R_2(B_1{:}\,D_1'', \ldots, A_n{:}\,D_n'')$$
$$R_x(\ldots)$$

È un insieme di schemi di relazioni con nomi diversi:

$$R = \{R_1(x_1), \ldots, R_n(x_n)\}$$

Istanza di database

Un'istanza di Data Base è un insieme di istanze di $r_1, r_2, \ldots, r_x$. È un insieme di tabelle ciascuna avente struttura indicata dal corrispondente modello.

$$d \text{ è } un'instanza di\ D\ se\ D = \{r_i, .., r_n\}\ dove\ \forall\ i \in [1, \ldots, n]\ r_i \text{ è } un'istanza di\ R_i$$

Chiave (Superchiave)

Un'istanza è consistente rispetto al vincolo di chiave se non esiste t_1,t_2 che appartiene ad r tale che $t_1[A_1] = t_2[A_1] \wedge t_1[A_2] = t_2[A_2]\ con\ t_1 \neq t_2$

Chiave candidata

Dato un uno schema relazione R(A_1, …,A_n) definiamo chiave candidata un sottoinsieme K non vuoto minimale di attributi $A_1,\ldots,A_n$ => $K = \{K_1, \ldots, K_n\} \subseteq \{A_1, \ldots, A_n\}$ tale che in un'istanza r non esistono più tuple che coincidono nel valore degli attributi della chiave (cioè non esistono più tuple che hanno gli stessi valori) non esiste $t_1, t_2 \in r\ con\ t_1 \neq t_2\ t.c\ t_1[K_1] = t_2[K_1], \ldots, t_1[K_n] = t_2[K_n]$

Chiave primaria

È un insieme di attributi che permette di individuare univocamente una tupla in una tabella.

Chiave esterna (vincolo di integrità referenziale)

Un vincolo di chiave esterna è definito su una coppia di schemi di relazione, tale vincolo impone che per ogni tupla dell'istanza di R1 esista una tupla dell'istanza di R2 che coincida in R1 nei valori degli attributi indicati.

In parole povere un vincolo di chiave esterna è quando destra c'è una relazione con delle chiavi e a sinistra c'è la stessa struttura con lo stesso numero di attributi.

$$R_1(K_1, .., K_n, A_1, \ldots, A_n)$$
$$R_2(B_1, .., B_x)$$
$$R_2[B'_1, \ldots, B'_m] \subseteq_{FK} R_1[K_1, \ldots, K_m]$$

Dove $B'_1, \ldots, B'_m \subseteq B_1, \ldots, B_m$ e r_1, r_2 istanza di R_1, R_2 sono consistenti se:

$$\forall\, t_1 \in r_1\ \exists\, t_2 \in r_2\ t.c\ t_1[K_1] = t_2[B'_1] \wedge \ldots \wedge t_1[K_m] = t_2[B'_m]$$

Dipendenze Funzionali

Le dipendenze funzionali servono a garantire la consistenza dei dati in maniera più generale ai vincoli di integrità referenziale. Serve ad esprimere legami funzionali tra gli attributi di una relazione. Esempio:

Quindi per ogni coppia di tuple t_1, t_2 appartenenti a r, se t_1, t_2 coincidono nel primo attributo di X, nel secondo e nell'α-esimo, allora devono coincidere in tutti gli attributi di Y.

Posto lo schema $R(A_1, \ldots, A_n); X \to Y\ dove\ X \subseteq \{A_1, \ldots, A_n\}\ \ X = \{X_1, \ldots, X_\alpha\}\ e\ Y \subseteq \{A_1, \ldots, A_n\}\ \ Y = \{Y_1, \ldots, Y_\alpha\}. Un'stanza\ r\ di\ R$ è $consistente\ rispetto\ a\ X \to Y\ se \forall\ t_1, t_2 \in r\ t_1[X_1] = t_2[X_1] \wedge \ldots \wedge t_1[X_\alpha] = t_2[X_\alpha] => t_1[Y_1] = t_2[Y_1] \wedge \ldots \wedge t_1[Y_\beta] = t_2[Y_\beta]$

Dunque la dipendenza funzionale è una generalizzazione del vincolo di chiave. /*concetto matematico da ricordare*/ Una funzione è una relazione particolare che ad ogni x della funzione associa una sola y del codominio.

BCNF

Uno schema $R(A_1,\ldots,A_n)$ con insieme di dipendenze funzionali F si dice in BCNF (Boyce-Codd Normal Form) se $\forall\ X \to Y\ \in F\ non\ triviale, la\ parte\ sinistra\ (X)$ è $una\ superchiave$. Ovvero da essa deve dipendere tutto l'insieme di attributi. Una relazione è priva di ridondanze se e solo se è in BCNF.

Scomposizione in BCNF

È detta scomposizione in BCNF l'insieme di dipendenze funzionali scritti in BCNF. Questo tipo di scomposizione è ottima sia perché non crea ridondanze, sia perché rappresenta le stesse cose anche dopo la decomposizione, mantenendo anche gli stessi vincoli. È detta preservazione dell'informazioni quando non si creano e non si cancellano tuple durante il cambio di schema (decomposizione in BCNF).

Pur esistendo sempre una decomposizione in BCNF senza perdita di informazioni (LOSELESS JOIN), alcuni schemi relazioni non sono decomponibili in BCNF senza perdita di dipendenze funzionali.

1NF

1. Tutte le tuple della relazione hanno lo stesso numero di attributi;
2. Non presenta gruppi di attributi che si ripetono;
3. I valori di un attributo sono dello stesso tipo (appartengono allo stesso dominio);
4. Esiste una chiave primaria;
5. L'ordine delle righe è irrilevante.

2NF

Una base di dati è in 2NF quando è in 1NF e per ogni relazione tutti gli attributi non chiave dipendono funzionalmente dall'intera chiave composta.

3NF

Un insieme si dice in 3NF (Third Normal Form) se soddisfa questa condizione:

$$Dato\ uno\ schema\ relazione\ R(A_1, \dots, A_n)\ \forall\ X \to Y\ o\ X \text{ è } superchiave\ o\ Y \in ad\ una\ chiave\ candidata$$

Inoltre se una decomposizione è in BCNF allora è anche in 3NF.

Copertura minimale

È quando nessun sottoinsieme ha le stesse proprietà. Si arriverà ad un insieme di dipendenze funzionali F' che è logicamente equivalente ad D ed è minimale (non ci sono ridondanze).

Implicazione logica tra copertura minimali: F⊨ $X \to Y$ dove è verificato F è verificato il vincolo $X \to A$

Assiomi di Armstrong

Se una dipendenza è logicamente implicata da un'altra, sicuramente si verifica l'altra. Per capire quale dipendenza funzionali sono ridondanti dobbiamo calcolare la copertura minimale. Ovvero quando nessun sottoinsieme ha le stesse proprietà.

Per calcolare la copertura minimale dobbiamo applicare le regole di equivalenza datti Assiomi di Armstrong.

1. **Riflessività**:
   Se $X \subseteq Y \subseteq U$ allora $Y \to X$;
2. **Augmentation**:
   $se\ X, Y, Z \subseteq U$ se ho X → $Y\ e\ Z\ \ allora\ XZ \to$ YZ;
3. **Transitività**:
   $X, Y, Z \subseteq U$ Se $X \to Y$ e $Y \to Z$ allora $X \to Z$

F è un insieme di dipendenze funzionali e ⊨ implica logicamente, mentre ⊩ derivabile.

$$F \vDash (X \to Y) <=> F \Vdash_a (X \to Y)$$

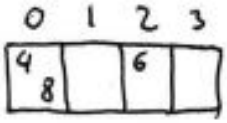
Tutto ciò che è derivabile è logicamente implicato e viceversa.

Chiusura

La chiusura di F è indicata con F^+ e sono tutte le dipendenze funzionali che si possono ottenere da F applicando gli assiomi di Armstrong.

Hashing

Per creare una tabella hash in modalità secondaria invece delle tuple ho delle pagine. Se nella modalità primaria ho un vettore di oggetti nella secondaria ho un vettore di pagine ciascuna contenente delle tuple. Indichiamo con M il numero delle pagine e supponiamo di avere una funzione H. Con M=4, H(K)= K Mod M.



Il costo di un inserimento di una tupla è costante e come granularità dobbiamo considerare il numero di pagine che compongono le strutture. Considerando:

T=100.000 tuple
M=4 pagine
Ricerca: $\theta(M) \rightarrow \theta(1)$
Inserimento: $\theta(1)\ o\ \theta(M)$

Se la pagina al suo interno contiene dei buchi allora si dice che è frammentata, in questo caso la ricerca sarà $\theta(T)$.

Se si verificano delle collisioni, una posizione conterrà una lista delle collisioni anziché un solo oggetto. In assenza di collisioni si può affermare che l'inserimento sarà $\theta(1)$, invece se ci sono trabocchi sarà $\theta(n)$. Allo stesso modo la ricerca senza trabocchi sarà $\theta(1)$.

Se prendo più pagine che tuple la possibilità che ci siano collisione è bassa, però non conviene utilizzare molte pagine in quanto non sfrutteremmo il fatto che gli accessi vengono fatti in gruppo.

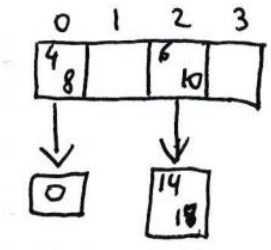
Il fattore di caricamento è la percentuale di spazio occupato, fatto rispetto alle tuple:

T= tuple effettivamente presenti
C= capacità (numero massimo di tuple che posso inserire all'interno di una pagina)
M> $\frac{T}{C}\ \frac{10\ Tuple}{2\ Tuple/Pagina}$

Le pagine che ho allocato inizialmente sono l'**aera primaria**. Ciascuna cella a cui si viene rimandati dalla tecnica di hashing non è l'indirizzo a una tupla ma è l'indirizzo a una pagina, in cui si trovano più tuple.

Lista per pagina

Supponiamo che ogni pagina abbia capienza due. Quando arriva la tupla 0 e dobbiamo inserirla nel vettore precedente, ma essendo H(0) = 0 mod 2 (dalla formula H(K) = K mod M) 0, non ci entra per cui dobbiamo creare delle pagine di trabocco separate, una per ogni pagina.

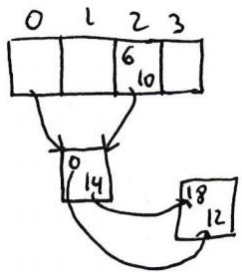


Questa tecnica però crea dei problemi di frammentazione poiché non tutte le pagine verranno utilizzate.

Notiamo che in questa situazione le collisioni sono una proprietà positiva perché permettono di ottenere una suddivisione ottimale degli oggetti per pagina la ricerca per tanto sarà $\theta(1)$. Potrebbe accadere che si crearsi una pagina per ogni tupla ma comunque la ricerca sarà sempre $\theta(1)$.

Nel caso in cui la pagina sia satura bisogna garantire che ogni tupla possa riscendere alla pagina dovuta: si crea un puntatore a parte dalla pagina piena verso un'altra con la stessa capacità. Questa lista di trabocchi sono possibili per ogni pagina. Si potrebbe pensare di creare una lista di trabocco per ogni pagina ma non sarebbe ottimo dal punto di vista della ricerca per pagina.

Lista dei Record

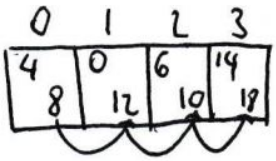


Un modo per ridurre la frammentazione è quello di creare una sola lista di trabocco è quella di creare una sola lista di trabocco. Man mano che arrivano le tuple non cambio M ma creo una lista di trabocco (area secondaria) a seconda di dove servano. Qui i puntatori sono tra i record di attivazione. Questa tecnica crea dei puntatori a livello dei record (RECORD LINCATI), secondo cui l'ultima tupla della pagina contiene un puntatore alla tupla successiva con lo stesso hash che sarà però contenuta in una pagina di trabocco generica.

Anche in questo caso però rimane comunque il problema della frammentazione, cioè che rimangono pagine non utilizzate nell'aria primaria.

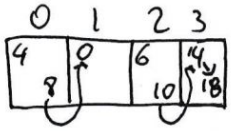
Per risolvere questo problema si è pensato di sfruttare le pagine vuote nell'aria primaria riempendole con i trabocchi che si verificano.

Metodo delle liste coalescenti



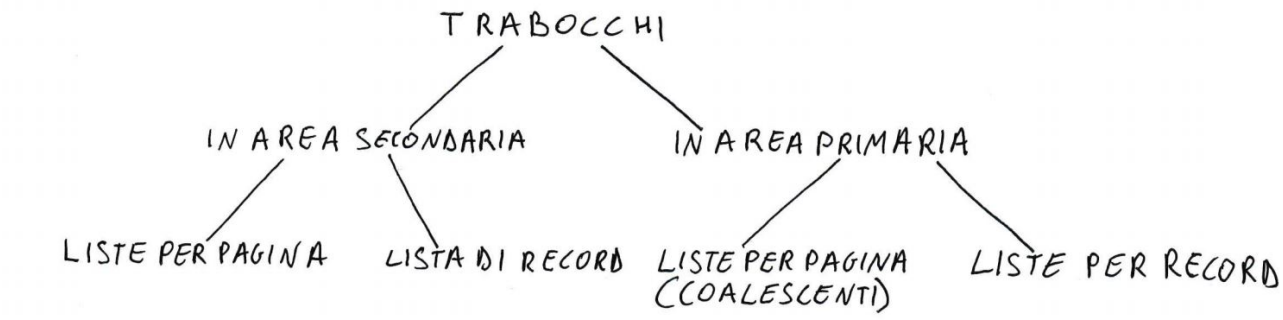
Considerando sempre il vettore iniziale, quando arriva lo 0 andrebbe inserito nella prima pagina, ma essendo già riempita al massimo non ho spazio e lo metto nella seconda in cui ho spazio disponibile inserendo un puntatore che va dalla pagina che ha traboccato in quella in cui ho inserito effettivamente il trabocco. Si chiamano coalescenti perché queste liste finiscono per unirsi.

Metodo delle liste per record



Un'altra alternativa è quello di creare dei nuovi puntatori al livello di record e non di pagina inserendo dei puntatori internamente.

Ricapitolando avremo questa situazione:

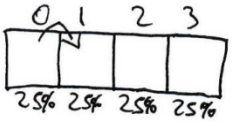


Gestire i trabocchi in area secondaria mi permette di creare liste di trabocco più corte, mentre gestirli in area primaria mi consente di sfruttare le pagine vuote e può accadere che una stessa pagina può essere usata per trabocchi di più pagine.

Hashing statico ad indirizzamento aperto

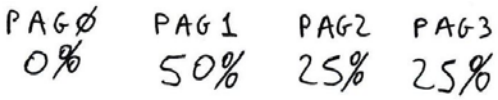
Tende ad utilizzare sempre le pagine dell'aria primaria. È detto ***aperto perché la funzione hash cambia nel tempo***, mentre ***statico perché il valore M non viene modificato*** (l'area primaria è grande sempre M).

$H_i(K) = (H(K) + S*i) \bmod M$



Questo oggetto non fa altro che creare delle liste senza bisogno di puntatore, se c'è un trabocco si sposta di S celle.

Probabilità che una tupla vada a finire nelle pagine se la prima pagina è tutta piena:



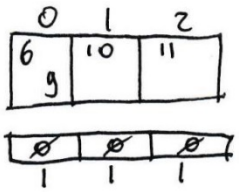
La pagina 1 avrà maggiore possibilità di creare trabocco perché ci sposiamo a destra quando quella di sinistra è piena. Un modo per evitare questa cosa è che S venga scelto in modo casuale.

Le pagine dell'aria primaria le alloca all'inizio in modo che sia facile spostarsi da una all'altra, ad esempio in un disco le alloco in modo che siano contigue rispetto alla testina.

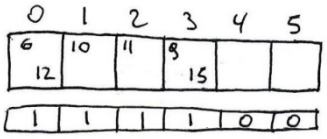
Hashing dinamico

Con le tecniche di hashing dinamico la funziona hash varia dinamicamente, infatti il numero di pagine M può aumentare o diminuire in base ai dati. Se cambia il numero di pagine cambierà anche la funzione di hashing. Quando parliamo di chiave qui intendiamo l'attributo su cui viene fatta l'organizzazione dei dati e la ricerca delle tuple (si parla infatti di ***chiave di ricerca***)

Hashing virtuale



Parto con un numero di pagine M. Sotto ciascuna pagina creo un vettore di bit costituito da tutti 0, quando faccio un inserimento all'interno di una pagina il suo bit corrispondente viene impostato a 1, ciò mi consente di sapere se una pagina è vuota senza andare a vedere a visitare la pagina stessa che mi costerebbe 4kb. La funziona hash in questo caso sarà $H_0(K) = K \bmod M$



Se devo aggiungere un'altra tupla ed ho un trabocco, raddoppio la pagina e di conseguenza il vettore di bit. Le pagine traboccate le ricopio. La funzione sarà così scritta $H_1(K) = K \bmod M$. L'aver raddoppiato serve a distribuire le tuple non più in una pagina ma in due.

Se avrà più di due trabocchi la mia funziona hash diventerà $H_i(K) = K \bmod 2^i M$, ovvero crescerà in modo esponenziale.

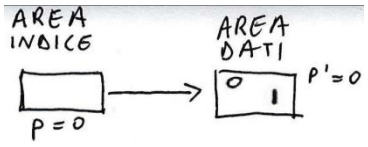
Quando faccio una ricerca farò riferimento al vettore di bit, se è impostato a zero vuol dire che quella pagina attualmente è vuota, per questo motivo non fermo la ricerca ma uso la funziona hash precedente perché è possibile che sia presente in un'altra pagina. Se invece il valore del bit è impostato ad 1 e non è presente in quella pagina significa che quel dato non è mai stato inserito.

In definitiva l'hashing virtuale funziona così: ogni volta che c'è un trabocco si raddoppia l'area dati. Man mano che raddoppiamo vengono creare nuove funzioni hash. Per la ricerca, se il bit è 0, parto dall'ultima funzione hash e poi si procede a ritroso, se è 1 e non è presente in quella pagina vuol dire che non è mai stato inserito. La complessità di ricerca è O(1). Questo tipo di hashing spreca però molto spazio.

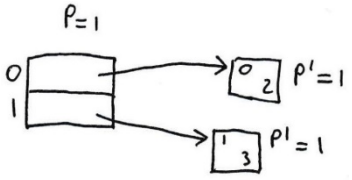
Hashing estendibile

Per la distribuzione non uso il modulo ma dei puntatori espliciti e per rappresentare le tuple uso il formato binario.

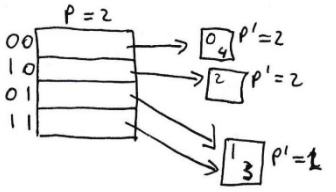
All'inizio abbiamo una cella ed una pagina dove P indica il numero di volte che ho raddoppiato l'aera indice, mentre P' indica il numero di trabocchi che ha portato alla creazione della pagina in questione.



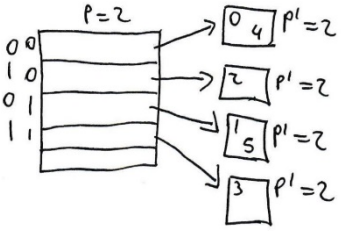
A questo punto arrivano i valori 2, che in binario è scritto come 10 e 3 che è rappresentato come 11. Essendo l'area dati piena si verifica un trabocco e allora raddoppio l'aera indice e a fianco ad ogni cella inserisco il valore 0,1. Con il primo punto alle tuple che hanno valore k che finisce con 0 con il secondo quelle che finiscono con 1. Nel nostro esempio la tupla che aveva generato trabocco è quella con valore 2 (in binario 10) quindi va messa nella prima pagina.



Se ora prova ad inserire il valore 4 (in binario 100) non c'è spazio quindi devo raddoppiare la pagina che è traboccata. Quindi aggiungo anche i penultimi bit di fianco all'area indice e ogni volta che ora faccio un inserimento ne devo tenere conto. Se una pagina non genera trabocco rimane intatta e le aree dati che non hanno traboccato che contengono tuple che coincidono nell'ultimo sono puntate dalle aree indici avente gli l'ultimo bit uguale e dato che i puntatori puntano alla stessa pagina posso dividerli aggiungendo una pagina senza raddoppiare l'aera indice (P>P') Vedi esempio a destra.

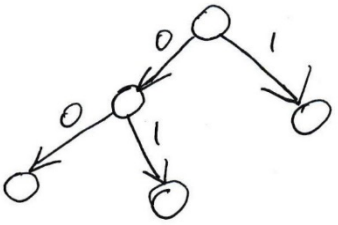


Se ora volessi aggiungere il valore 5 (in binario 101) devo sdoppiare la cella che crea trabocco, ma in questo caso non devo raddoppiare l'area indice perché ***P va raddoppiato SOLO se è = a P'***. In questo caso non c'è bisogno di raddoppiare l'indice, poiché il numero di trabocchi è minore del numero dell'indice, ciò significa che quella pagina è puntata da due puntatori. Quindi alla fine avremo la situazione come nell'esempio.



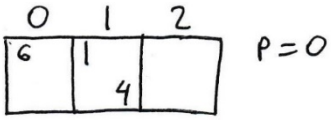
In conclusione possiamo dire che P e P' servono a sapere se devo raddoppiare o no l'area indice. Possiamo vederlo come se fosse un albero:

- Costo ricerca O(1);
- Costo inserimento:
  - Senza trabocco: 1 lettura e 1 scrittura;
  - Con trabocco: 1 lettura e 2 scritture.

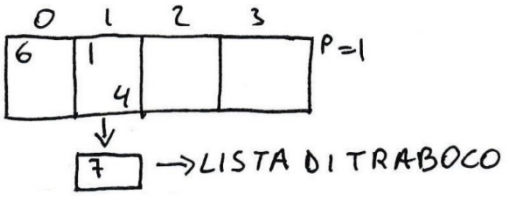


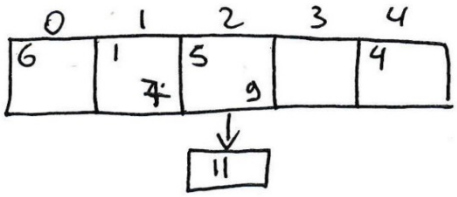
Hashing lineare

L'hashing lineare è un misto tra l'hashing estendibile e l'hashing virtuale. Supponendo di avere M=3 e quindi avremo le pagine le pagine 0,1,2, e la funzione hash H(K) = K mod M. Ogni qual volta avremo un trabocco ridistribuisco la pagina (P) più a sinistra non ancora ridistribuita. Quindi P=0; $H_0(K) = K_0$ mod M, quindi se c'è il trabocco diventa $H_1(K)$= K mod 2M.



Se ho vari trabocchi P raggiungerà la pagina che lo ha generato e quindi la ridistribuirà. Se so che X = K mod M quindi se faccio K mod 2M può valere X (sarebbe la pagina P) oppure X+M (questo valore è proprio la pagina che abbiamo appena aggiunto, ne abbiamo parlato nell'hashing virtuale).





Scrivere K mod M+1 non va bene perché non ridistribuisce solo la pagina che ha traboccato ma tutte le pagine. Quando ho raggiunto l'ultima pagina ovvero P = M-1 dovrò raddoppiare M e quindi avrò M' per cui $H_0(K) = K_0 \bmod M'$ e $H_1(K) = K \bmod 2M'$. Quando cerco una tupla devo usare sempre $H_1(K)= K \bmod 2M'$ per accorgermi se la pagina restituita esiste devo verificare che sia minore di M+P quindi $H_1(K) < M+P$ se è maggiore, quella pagina non esiste, quindi vado ad usare $H_0(K) = K_0 \bmod M$, se non esiste nemmeno qui allora la tupla cercata non esiste.

Tutto ciò funziona bene perché sfrutta al meglio le pagine. Il tipo di hashing supporta la ricerca solo delle chiavi primarie e sono di tipo procedurale.

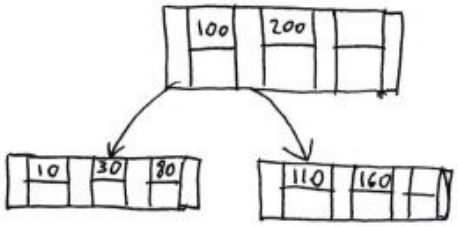
A differenza dell'hashing virtuale, che usa la funzione $H(K)_i= K \bmod 2^iM$, l'hashing lineare sua esclusivamente le funzioni $H_0(K) = K_0 \bmod M$ e $H_1(K)= K \bmod 2M'$

B-Tree

Ora vedremo le tecniche tabellari ovvero le pagine in cui cercare la tupla si trova in una tabella, quest'ultima non è sempre una tabella ma è una coppia chiave, indirizzo. La tecnica tabellare principale è quella dei B-Tree, ovvero degli alberi fatti per mappare in modo efficiente gli inserimenti, ricerche e update.

L'idea di base al B-Tree è: cerco di far corrispondere un nodo ad una pagina ed avrà grado pari al numero di tuple che mette nella pagina. Ciascuna pagina è una lista di tuple. M è il grado (massimo m-1 chiavi di ricerca) e dipende da quante tuple ci sono all'interno.

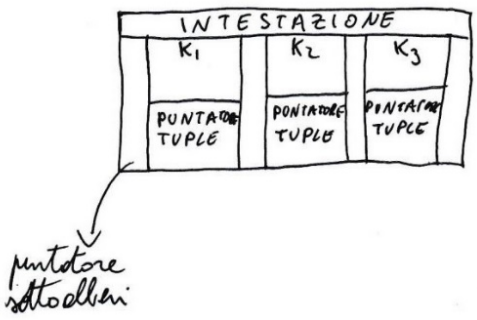
All'interno della lista viene rappresentato solo un riferimento alla tupla e non l'intera tupla. A sinistra e a destra di ogni nodo viene memorizzato l'indirizzo ai rispettivi figli.



Proprietà di un B-Tree

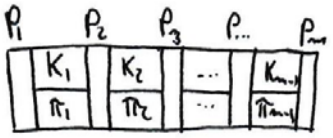
1) Tutte le foglie sono alla stessa profondità;
2) Tutti i nodi hanno al più m figli;
3) In Tutti i nodi non foglia ci sono X chiavi se e solo se ci sono X+1 figli;
4) Tutte le pagine sono "piene almeno a metà" vuol dire che il numero di chiavi contenute deve essere $\frac{m-1}{2}$; mentre per la radice devo avere almeno 2 figli, i nodi ne devono avere $\frac{m-1}{2}$+1.

Struttura pagina di un B-Tree:



B^+-Tree

I B-Tree vengono implementati con i B^+-Tree. La differenza sta nella struttura dei noi intermedi e nei nodi foglia. Nei B-Tree:



- K sono le chiavi di ricerca;
- π sono i puntatori alle tuple intere;
- P sono i puntatori ai nodi successivi.

Logicamente anche un albero può essere visto come una tabella. Ogni nodo oltre alla chiave contiene a posizione della tupla; in questo nodo dovremmo avere la massima efficienza. Questa cosa però non è vera perché per gradi di B-Tree elevati (Come 100), comporta che, pur non essendoci tanti livelli, la gran parte delle chiavi sarà indicizzata all'ultimo livello. Quindi per trovare la chiave, probabilmente, dovremo arrivare comunque alle foglie per trovare la tupla che ho cercato.

Ciò ha spinto i programmatori a spostare il puntatore direttamente all'ultimo livello. Nei nodi intermedi si usa solo il puntatore ai figli e si fa in modo che le chiavi indicizzate al livello intermedio di trovino anche a livello foglia. A questo livello invece ci sarà il puntatore direttamente alle tuple vere e proprie.

A parità di spazio nel B^+-Tree entrano più chiavi, quindi posso creare un albero largo e di conseguenza meno profondo. In questo modo le operazioni saranno più efficienti.

Ogni volta che si crea una chiave primaria il DBMS crea un indice di tipo B^+-Tree. Su una stessa tabella si possono creare più indici. Ad esempio si potrebbe creare un'indice sulla chiave primaria ed uno su una chiave candidata.

Il B^+-Tree è un indice di memorizzazione secondario, non primario come per l'hash. L'hashing anche se cerca sull'attributo chiave mi crea problemi sulla ricerca per range perché non è detto che due chiavi con lo stesso concetto siano in posizioni vicine.

Nel B^+-Tree tutte le chiavi si trovano al libello foglia e fa sì che tutte le pagine siano concatenate, in questo modo posso effettuare più efficacemente le ricerche per range.

Quindi questo indice supporta la collezione veloce per le ricerche per range. Poiché mi basta trovare la foglia che sodisfa il primo requisito.

In generale un B-Tree non impone dei vincoli su come ordinare i dati nella pagina. Però se so che farò ricerche in un determinato modo posso imporre l'associazione ad un'organizzazione primaria (hash, B-Tree) e secondaria (B-Tree). NB: l'organizzazione primaria "detta legge" su dove inserire i dati nella memoria primaria.

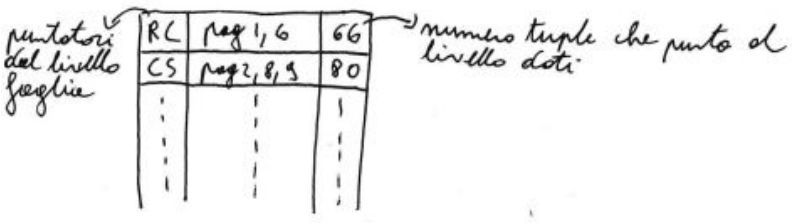
L'indice può essere di due tipi: ***denso*** e ***sparso***:

- **Denso**: tutte le chiavi sono indicizzate;
- **Sparso**: non tutte le chiavi sono indicizzate.

Quando un indice è sparso mi consente di avere un B-Tree piccolissimo perché non devo indicizzare tutte le chiavi. Questo indice si può usare solo con l'organizzazione primaria, quindi le tuple verranno tenute ordinate solo in base alla chiave primaria.

Questi indici sono utilizzati per puntare a tuple su cui non c'è ambiguità sulla chiave di ricerca. Ovvero se sono chiavi primarie o candidate. Un'indice di questo tipo si dice che è di tipo primario, ovvero quando è unicamente indicato dal valore di ricerca.

La tabella dei riferimenti (ad indice invertito) si usa per usare gli indici secondari. Ad esempio per ogni città avremo un puntatore alle pagine che contengono i riferimenti.



Per creare gli indici bisogna usare questo comando:

CREATE INDEX ON … (attributo)

Transazioni

Quelle che sono le operazioni degli utenti in gergo tecnico sono dette Transazioni, ovvero un'unità di programma in esecuzione che modifica lo stato della base di dati. La Transazione, quindi, è un'unità svolta da un programma applicativo. Un sistema che mette a disposizione un meccanismo per la definizione e l'esecuzione di transazioni viene detto sistema transazionale. Ogni transazione è definita all'interno di due comandi: begin transaction ed end transaction. All'interno del codice delle transazioni può comparire l'istruzione commit. La transazione va a buon fine solo a seguito di una commit (Esegue le modifiche che sto facendo sul database) . Se nella commit qualcosa va storto tutte le modifiche vengono annullate.

Le transazioni rispondono a delle proprietà dette ACID:

- Atomicità (insieme di operazioni viste come un unico corpo): rappresenta il fatto che una transazione è un'unità indivisibile di esecuzione; o vengono resi visibili tutti gli effetti di una transazione, oppure la transazione non deve avere alcun effetto sulla base di dati. In pratica non è possibile lasciare la base di dati in uno stato intermedio: se durante l'esecuzione dell'operazione si verifica un errore, allora il sistema deve ripartire dall'inizio della transazione.
- Consistenza: è una proprietà delle single transaction. Lo stato a cui mi porta una transazione deve essere consistente. La consistenza richiede che l'esecuzione della transazione non violi i vincoli di integrità definiti sulla base di dati. Si annulla la transazione o si coregge la violazione del vincolo (verifica immediata o differita)
- Isolamento: il DBMS gestisce più transazioni concorrenti, garantendo che ciascuna è (indipendente) separata dalle altre.
- Durabilità: Una volta che una transazione ha avuto successo, il suo effetto è persistente. In pratica, una base di dati deve garantire che nessun dato venga perso per nessun motivo.

Esempio di utilizzo delle proprietà ACID

Supponiamo di estrarre tutte le tuple dalla tabella A e supponiamo che è vuota. Un mio collega fa la stesso con lo stesso risultato Se ora uno dei due fa un insert di una tupla con valore "1" e facciamo una SELCET* ci sarà restituita la tupla. Se invece al posto di farla noi la facciamo dare al nostro collega avrà come risultato un insieme vuoto perché la transazione precedente non era finita. Per fagli capire che abbiamo finito dobbiamo scrivere COMMIT. Quindi per gestire l'atomicità il DBMS si aspetta che noi gli segnaliamo quando abbiamo concluso di scrivere il corpo di operazioni che deve essere considerata come un'unica operazione logica. Per fare ciò possiamo anche usare i seguenti comandi:

STAR TRANSACTION (qual cosa)
END TRANSACTION

Queste sono le linee di codice che indicano una sola transazione. NB COMMIT esegue le modifiche su un DBMS condiviso.

Se usiamo le variabili SQL c'è il comando AUTOCOMMIT, che se vale 1 ogni cose che facciamo fa commit. Se vogliamo eseguire blocchi di operazione ci dobbiamo assicurare che sia 0. Tutte le operazioni sono soggette a controlli sui vincoli per garantire la consistenza.

SCHEDULER

Garantire l'isolamento vuol dire che le transazioni che accedono al DBMS siano isolate, ovvero non devono rendersi conto che ci sia qualcuno che sta modificando quegli stessi dati. Per fare ciò abbiamo bisogno di fare le operazioni in serie. Il componente che si occupa di fare ciò si chiama SCHEDULER.

Differenza tra SCHEDULE e SCHEDULER

Gli **SCHEDULE** sono una sequenza di azioni in lettura o scrittura, mentre gli **SCHEDULER** si occupano del controllo della concorrenza, e dovranno rifiutare gli schedule che porterebbero a inconsistenze o anomalie ed accettare tutti gli altri.

Se arrivano T_a,T_b,T_c uno scheduler seriale assegna:

1 - T_a
2 - T_b
3 - T_c

T_1:
read (A)
tmp = A/10
A=A+tmp
write(A)
read(B)
B=B-tmp
write(B)

Nel frattempo arriva T_2

T_2:
read(A)
A=A+20
write(A)
read(B)
B=B-20
write(B)

Uno scheduler seriale decide quale delle due transazioni deve gestire. Questa cosa non va bene. Nei DBMS lo scheduler fa delle operazioni interlacciate tra i due blocchi e mantiene comunque l'ordine di esecuzione. I numeri che assegna si chiamano SCHEDULE e devono essere serializzabili, ovvero quando è equivalente ad uno schedule seriale.

SCHEDULER SERIALIZZABILE

Uno scheduler si dice serializzabile quando è equivalente ad uno scheduler seriale. Per capire se uno scheduler è serializzabile dovrei andare ad analizzare le righe del codice. Purtroppo non esiste alcun

algoritmo che dato in input un altro algoritmo possa capire cosa faccia. In pratica ci sono delle tecniche e regole che ci aiutano a capire se lo è o se non lo è, ma non sapremo mai con certezza se è così.

Condizione sufficiente di serializzabilità

Dato uno schedule S, se da esso è possibile ottenere tramite una sequenza di scambi di operazioni NON in conflitto, uno schedule seriale S', allora S è serializzabile

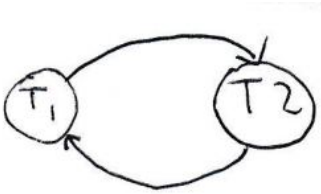
Conflict-equivalence

$$s \equiv_c s'$$

Se è possibile ottenere l'uno dall'altro mediante una sequenza di scambi di operazioni non in conflitto.

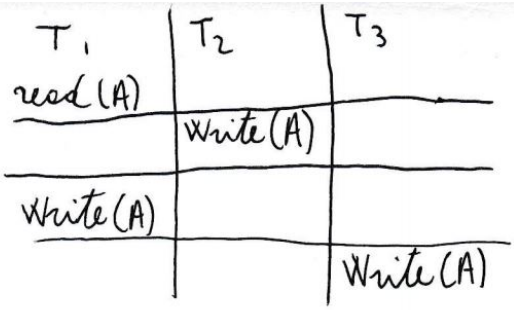
s è conflict-serializable se esiste s' seriale tale che $s \equiv_c s'$

Il grafo delle precedenze contiene un nodo per ogni transazione e mette un arco da T_1 a T_2 se esiste una coppia di operazioni in conflitto, se una è eseguita in T_1 l'altra in T_2.



Lo schedule è conflict-serializable se e solo se il grafo ottenuto è aclico.

Supponiamo di avere il seguente scheduele S:



Questo schedule è serializzabile anche se non è conflict-serializable perché contiene delle scritture cieche (blind writes), ovvero che scrivono senza leggere il valore della variabile.

View-equivalence

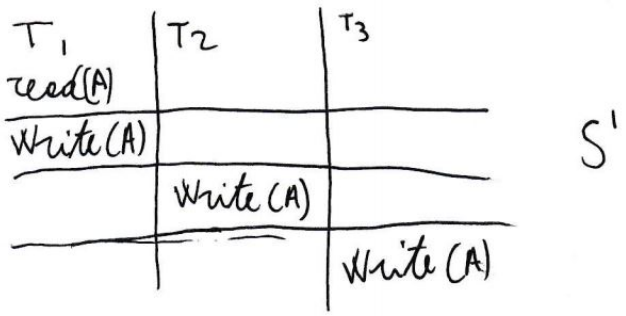
$$s' \equiv_v s''$$

Due schedule s', s'' sono view-equivalent se:

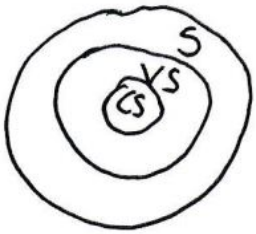
1. Per ogni risorsa R, ogni transazione che legge il valore iniziale di R in s' legge il valore iniziale di R anche in s'';
2. Per ogni risorsa R, se la transazione T_i legge il valore di R scritto da T_j in s', T_i deve leggere il valore scritto da T_j su R anche in s'';
3. Per ogni risorsa R, la transazione (se esiste) che scrive il valore finale di R in s', deve scrivere il valore finale di R in s''.

Se uno schedule è conflict-equivalence allora è view-serializabele, ma non vale il contrario.

Uno schedule è view-serializable quando è view-equivalent ad uno schedule seriale. Esempio di schedule View-Serializzable:



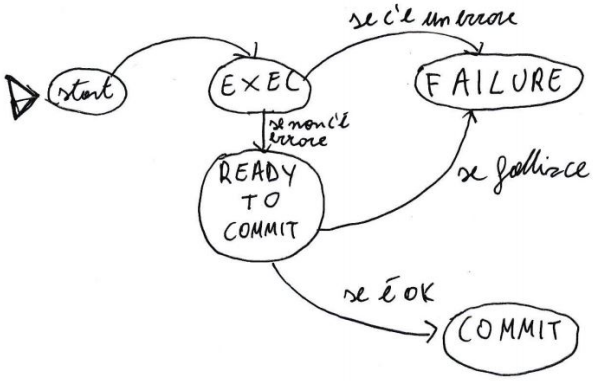
Se uno schedule è conflict-serializable allora sarà anche view-serializzable, perché il primo è un sotto insieme di quest'ultimo. Ma non vale il contrario.



Garantire l'isolamento delle transazioni in pratica vuol dire garantire la serializzabilità. Gli schedule generati nei DBMS sono conflict-serializable. Un'altra proprietà importante è la robustezza, ovvero quando è capace di gestire in modo corretto gli errori.

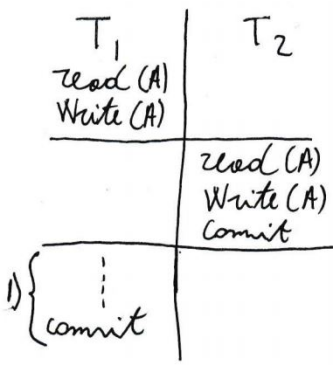
Uno modo per fare ciò sono i Lock, ovvero dei registri delle transazioni effettuate che permettono, in caso di errori, di ripristinare lo stato precedente. Questa operazione di ripristino è detta roll-back. Alla fine di ogni transazione viene effettuata una commit che rende definitiva la transazione. Anche se tutto è andato bene può accadere che la commit fallisce e anche in questo caso viene effettuata una roll-back.

In generale quando viene fatta partire una transazione viene eseguita secondo questo diagramma di stato:



Problema delle Kill a cascata

Se sono più transazioni quando si fa il roll-back potrebbero esserci dei problemi:

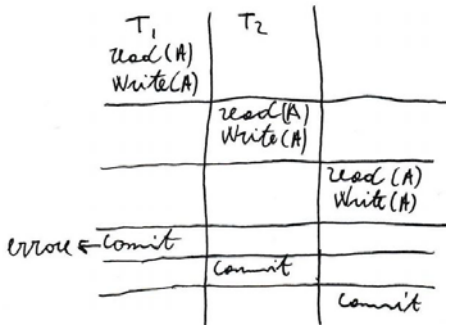


1) Viene generato un errore e dovremo cancellare tutte le modifiche di T_1; però non possiamo farlo perché T_2 dipende dal valore di A scritto da T_1. Per provare a risolvere questo problema basta imporre una politica specifica su commit. Basta postare la commit di T_2 dopo la commit di T_1. In questo modo vengono cancellati anche i valori di T_2, così sono sicuro di avere uno stato consistente.

Se una transazione legge il valore scritto da un'altra altra transazione allora T_2 deve fare commit dopo T_1. In questo modo la robustezza è garantita.

Se uno schedule è conflict-serializable il grafo delle precedenze è aciclico, perché non posso invertire l'ordine di lettura/scrittura di due variabili.

Se ci sono molte transazioni può succedere:



Quindi devo cancellare tutte le operazioni precedenti facendomi perdere molto tempo. Questo effetto è detto effetto cascata che fa una kill a cascata di altre transazioni. In questo caso il problema non è di robustezza ma di efficienza.

N.B una transazione può fare commit solo alla fine e mai a metà e poi continuare. In questo caso posso fargli fare commit subito dopo la write. In questo modo metto dei limiti sul parallelismo, però riguardano solo le variabili condivise.

Condizione sufficiente ad evitare kill a cascata

Se T_2 legge da T_1, T_1 deve aver fatto commit prima di T_2.

Tipi di schedule

- **Schedule Recoverable** (sono robusti rispetto agli errori);
- **Schedule Cascadless** (sono senza l'effetto cascata).

Se uno schedule è cascadeless allora è anche recoverable

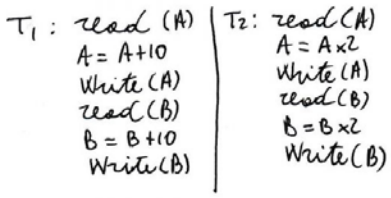
Lock

È una struttura dati che garantisce la muta esclusione, chi scrive la transazione deve richiedere il lucchetto e decidere quando rilasciarlo anche se ciò avviene a volte implicitamente nella pratica. Esistono due tipi di lucchetti:

- **LOCK-SHARED** (LOCK-S): su un dato possono essere attivi più lock di questo tipo, infatti si limitano a bloccare la modifica, ma consentono comunque di accedere al dato in lettura;
- **LOCK-EXCLUSIVE** (LOCK-X): non può coesistere con altri lock sulla stessa risorsa, in quanto blocca qualsiasi tipo di accesso al dato.

Ogni qual volta una risorsa viene rilasciata avviene un unlock.

Supponiamo di avere:

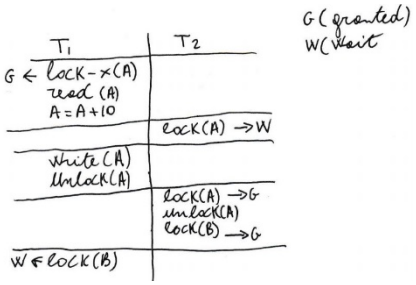


Poiché le due transazioni fanno richiesta allo schedule faremo richiesta dei lucchetti.

T_1: richiede lock-x(A) e dopo la write rilascia il lucchetto, stessa cosa fa per B.

T_2: farà la stessa cosa di T_1.

Ed avrò ottenuto quindi:



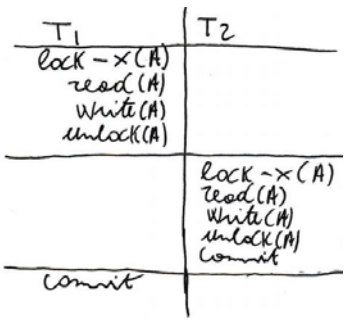
Questo ci fa capire che lo schedule non è serializzable perché fa T_1,T_2,T_2,T_1 e non T_1,T_2,T_1,T_2.

Per evitare la non serializzabilità ciascuna transazione dopo che fa unlock non può più chiedere lock.

Protocollo a due fasi (2 Phase Locking) 2PL

Questo tipo di protocollo viene così chiamato perché ci sono due principali fasi in cui agisce:

1. **Growing**: fase di accrescimento, ovvero quando la transazione richiede i lucchetti e di conseguenza il numero di lucchetti a sua disposizione aumenta.
2. **Shrinking**: fase di rilascio, ovvero quando viene fatta un unlock sui lucchetti.



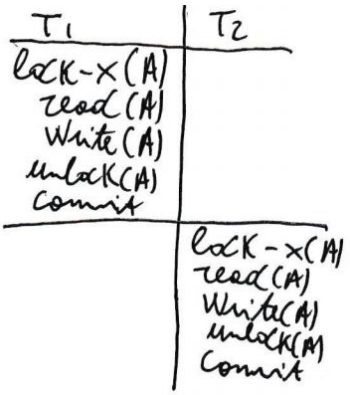
Una transazione dopo aver rilasciato un lock non può acquisirne altri. Se le transazioni seguono il 2PL vuol dire che il loro schedule è conflict-serializable. Inoltre dobbiamo dimostrare che le transazioni che seguono il 2PL sono serializzabili, dobbiamo dimostrare anche che il terming point è il momento in cui si passa dal growing allo shirinking.

Questo dimostra che lo schedule che usano la politica di 2PL sono serializzabili, perché tutto ciò che avviene in una transazione fa parte di un corpo unico, le operazioni non possono intrecciarsi.

S2PL (Strict 2 Phase Locking)

Con il 2PL in caso di errore non ho recuperabilità e poiché devo garantire la cascadeless, la commit di T_1 deve essere effettuata prima di quella di T_2.

Per consentire entrambe, recoverability e cascadless, devo alterare la 2PL e ottenere così la S2PL.



- **S2PL**: una transazione deve essere conforme a 2PL e rilasciare i suoi lock esclusivi (blocchi di scrittura) al momento della commit. Si garantisce così recoverability e cascadeless;
- **SS2PL** (Strong S2PL): Non si fa distinzione per gli unclock tra lock esclusivi e lock condivisi, questo tipo di protocollo evita alle

transazioni di fare gli unlock, ciò vuol dire che tutti i lucchetti vengono rilasciati nel momento della commit. È impossibile in questo modo fare richiesta di altri lock dopo aver fatto l'unlock.

Livelli di isolamento

1. **Serializzabile**: è il livello di isolamento massimo, viene garantito nei DBMS che si poggiano su lock e che utilizzano il protocollo 2PL. Alcuni DB implementano questo livello di isolamento in modo che in certi casi potrebbero consentire esecuzioni non serializzabili. Non si verificano anomalie;
2. **Repetable Read**: se una transazione legge un valore, una tupla, allora nessun'altra transazione avrà la possibilità di toccare quella tupla, finché la prima transazione non finisce. Un DBMS che segue questa politica non è sicuro che sia serializzabile. Presenta 1 anomalia: Lettura Fantasma;
3. **Read Commited**: in questo livello di isolamento può verificarsi il fenomeno delle letture non ripetibili. Garantisce che qualsasi dato letto sia impegnato nel moment in cui viene letto. Non garantisce in alcun modo che se la transazione riemette la lettura troverà gli stessi dati perché il dato può essere già stato committato. Presenta 2 anomali: Lettura Fantasma e Letture non ripetibili;
4. **Read uncommitted**: è il più basso livello di isolamento. Qualunque transazione può leggere dei valori che non sono stati commited, cioè una transazione potrebbe visualizzare modifiche non ancora commesse da altre transazioni (Letture sporche). Presenta 3 anomalie: Letture sporche, letture fantasma e letture non ripetibili.

- **Letture fantasma**: se rieseguo la stessa query durante la transazione, potrei trovare righe in più di quelle che ho letto in precedenza, ma mai in meno o modificate;
- **Letture non ripetibili**: in pratica, rieseguendo due volte la stessa SELECT nel corso di una transazione, potrei ottenere dati diversi se altre transazioni sono terminate nel tempo intercorso tra le due letture;
- **Letture sporche**: nella transazione corrente si possono leggere dati che qualche altra transazione sta scrivendo in quel momento senza aver ancora fatto COMMIT, quindi può capitare di leggere chiavi violate, dati inconsistenti, eccetera.

Esiste anche un altro livello di isolamento chiamato snapshot isolation, che non presenta anomalie, somiglia a ciò che può essere fatto tramite file system; quando uno prova a scrivere sul DBMS, gli viene data una snapshot cioè una copia locale, ciascuno di questi lavorerò sulla propria copia (gli do solo una copia della porzione dei dati che richiedono, non di tutto il DBMS).

Algebra Relazionale

È un linguaggio procedurale, basato su concetti di tipo algebrico. È Costituito da un insieme di operatori, definiti su relazione (insiemi) e che producono ancora relazioni come risultati.

- **Unione**: l'unione di due relazioni r_1 e r_2 definire sullo stesso insieme di attributi X è indicata con $r_1 \cup r_2$ ed è una relazione ancora su X contenente le tuple che appartengono a r_1 oppure r_2, oppure ad entrambe.
- **Differenza**: la differenza di $r_1(X)$ ed $r_2(X)$ è indicata con $r_1 - r_2$ ed è una relazione su X contente le tuple che appartengono ad r_1 e non a r_2.
- **Intersezione**: l'intersezione di $r_1(X)$ ed $r_2(X)$ è indicata con $r_1 \cap r_2$ ed è una relazione su X contenente le tuple che appartengono sia ad r_1 che ad r_2.

Selezione

È un operatore monadico ed è definita su un operando e produce come risultato una porzione dell'operando. Più precisamente produce un sottoinsieme delle tuple, su tutti gli attributi (decomposizione orizzontali). È denotato dal simbolo σ che ha come pedice la condizione di selezione.

Proiezione

Produce un risultato a cui contribuiscono tutte le tuple, ma su un sottoinsieme degli attributi. Produce decomposizioni verticali. Dati una relazione r(X) e un sottoinsieme Y di X, la proiezione di r su Y indicate con ($\Pi_y(r)$) è l'insieme di tuple Y ottenute dalle tuple di r considerando i valori solo su Y: $\Pi_y(r) = \{t[Y] | t \in r\}$.

Join

È un quantificatore esistenziale. Questo operatore correla i dati in relazioni diverse, sulla base di valori uguali in attributi con lo stesso nome. Il risultato è costituito da una relazione sull'unione degli insieme di attributi degli operandi e le sue tuple sono ottenute combinando le tuple degli operandi con valori uguali sugli attributi comuni.

Theta-Join

Operatore derivato definito come prodotto cartesiano seguito da una selezione.

Interrogazione in algebra relazionale

Può essere vista come una funzione che applicata a istanze di database produce relazioni. Dato uno schema R di base di dati, un'interrogazione è una funzione che, per ogni istanza r di R, produce una relazione su un dato insieme di attributi X.

SQL (Structured Query Language)

È un linguaggio di interrogazione per basi di dati relazionali. SQL non è solo un linguaggio di interrogazione, ma contiene al suo interno funzionalità di:

- DML (linguaggio per la manipolazione dei dati)
- DDL (linguaggio per la definizione dei dati)

Nel DML (Data Manipulation Language) si intente la possibilità di interagire con i dati per manipolarli o estrapolarli. La manipolazione non cambia la struttura ma solo i contenuti delle istanze, mentre il DDL (Data Definition Language) crea nuove strutture.

Definizione dei dati

- Le parentesi quadre indicano il termine è opzionale
- Le parentesi graffe indicano che il termine può non comparire o essere ripetuto un numero arbitrario di volte
- Le barre verticali indicano che deve essere scelto uno tra i termini separati dalle barre, un elenco di termini in alternativa può essere racchiuso tra parentesi angolate.

Interrogazioni in SQL

SQL esprime le interrogazioni in modo dichiarativo, ovvero non si specifica l'obiettivo, ciò si contrappone a linguaggi di interpretazione procedurali, come l'algebra relazione, in cui si specifica il modo in cui un'interrogazione dev'essere eseguita.

Le operazioni di interrogazione in SQL vengono specificate per mezzo dell'istruzione SELECT. Il risultato di un'interrogazione SEL è una tabella con una riga per ogni riga selezionata, con un insieme di colonne dato dalle espressioni che compaiono nella target list (specifica gli elementi dello schema della tabella risultato. * = selezione di tutti li attributi delle tabelle elencate nella clausola from.), ciascuna eventualmente ridenominate con l'alias.

Clausole SQL

- FROM: specifica la tabella da cui prelevare i dati;
- WHERE: definisce una condizione sui valori degli attributi delle tabelle selezionate (espressioni booleane, predita)
- DISTINCT: elimina i duplicati (stesse righe)

Operatori aggregati

Sono funzioni che si applicano ad un insieme di tuple di una tabella.

- COUNT: Restituisce il numero di volte in cui l'attributo è non nul;
- MAX;
- MIN;
- SUM;
- AVG: media dei valori dell'attributo;

Interrogazioni con raggruppamento.

- **GROUP BY**: partiziona le righe di una tabella in sottoinsieme. Ha come argomento un insieme di attributi e raggruppa le righe che posseggono lo stesso valore per gli attributi dell'argomento;
- **HAVING**: seleziona solo i gruppi che soddisfano un certo predicato. Serve per filtrare i gruppi da GROUP BY.
- Operatori insiemistici: union, intersect, except.

Interrogazioni modificate

SQL ammette anche l'uso di predicati con una struttura più complessa, in qui si confronta un valore (ottenuto come risultati di un'espressione valutata sulla singola riga), con il risultato dell'esecuzione di un'interrogazione SQL.

- EXISTS: operatore logico. Restituisce vero solo se l'interrogazione fornisce un risultato non vuoto.
- IN: operatore insiemistico. Ha come argomenti una tupla e una SELCET. Restituisce true se la tupla è in quella interrogazione.

I sistemi relazionali cercano il più possibile le interrogazioni in modo set-oriented (ovvero orientato agli insieme), con l'obiettivo di effettuare poche operazioni sui dati.