APOGEO education

Sistemi di basi di dati e applicazioni

Angelo Chianese
Vincenzo Moscato
Antonio Picariello
Lucio Sansone

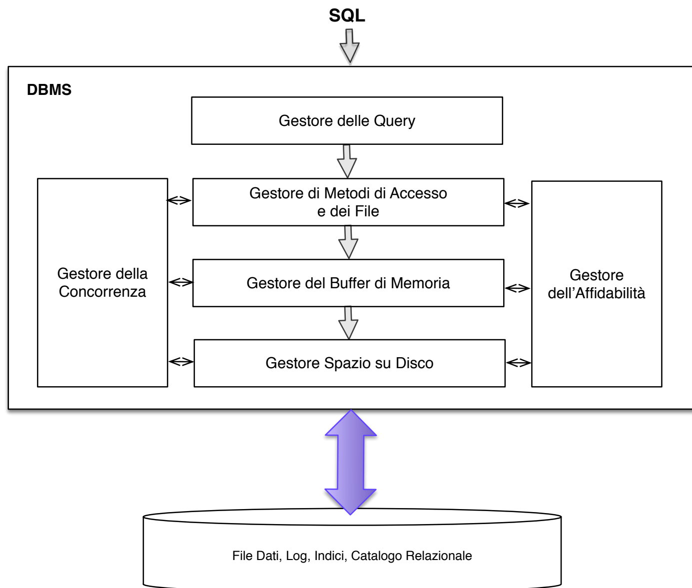
Tecnologia dei DBMS

Angelo Chianese - Vincenzo Moscato - Antonio Picariello

29 novembre 2017

1. Introduzione: la struttura di un DBMS
2. Memorizzazione dei dati ed organizzazione dei File
3. Indici
4. La gestione del buffer di memoria
5. Ottimizzazione delle query
6. La gestione delle transazioni
7. Controllo di concorrenza
8. Controllo di affidabilità
9. Basi di Dati Distribuite
10. Gestione delle transazioni nei DDBMS
11. Replicazione di basi di dati

Schema Architettura



Architettura - 1

- Ogni query viene quindi analizzata, attraverso un apposito modulo denominato *Gestore delle Query*,
  - .... ed eventualmente “ottimizzata”, selezionando il *piano di esecuzione* più efficiente in termini di operazioni di basso livello (es. scansioni, accessi diretti, ordinamenti, join, ecc.) richieste per l’esecuzione della query stessa.
- La risoluzione di una query in termini di operatori di basso livello necessita di metodi per l’accesso ai dati e per la gestione delle informazioni sui file della base di dati
  - ... ogni file può essere visto, da un punto di vista logico, come una sequenza di *record*, o più in generale, come una collezione di *pagine* di record.

Architettura - 2

- Il *Gestore dei Metodi di Accesso e dei File* a sua volta necessita di informazioni legate a come il *buffer* dei dati è gestito in memoria centrale
  - .. quella parte della memoria centrale demandata a contenere pagine di record che costituiscono un sottoinsieme, in genere, delle informazioni presenti su memoria di massa.
- Il *Gestore del Buffer di Memoria* ha il compito di determinare come e quando trasferire queste informazioni da memoria centrale a memoria di massa
- Il (*Gestore dello Spazio su Disco*) permette di implementare le funzioni di lettura, scrittura, allocazione, rilasciamento delle pagine su disco

Architettura - 3

- il *Gestore della Concorrenza* assicura l'esecuzione corretta delle transazioni garantendone le proprietà ACID e filtrando, quindi, opportunamente le richieste degli utenti sulla base di dati.
- Il l *Gestore dell'Affidabilità* interviene in caso di guasti del sistema. Attraverso una registrazione continua di quanto accade nel sistema che modifica lo stato di una base di dati (o *log*), è possibile poi mettere in piedi meccanismi che permettono di ripristinare il sistema in caso di eventuali guasti.

Un ulteriore modulo detto *Gestore dell'Integrità* assicura che i vincoli di integrità siano sempre soddisfatti a valle delle operazioni di modifica dello stato della base di dati, mentre il *Gestore degli Accessi* garantisce che solo utenti e applicazioni autorizzati abbiano accesso alle informazioni della basi di dati e che le loro operazioni siano compatibili con i loro privilegi.

Memorizzazione dei dati

- Al fine di gestire in mondo persistente le grosse quantità di informazioni tipiche di un sistema di basi di dati, il DBMS deve memorizzare i relativi dati su dispositivi di memoria di massa, quali ad esempio dischi o anche nastri.
- Le informazioni richieste dall'elaborazione dovranno essere poi spostate da tale dispositivi di memoria in memoria centrale: in particolare, i dati saranno letti nella memoria per essere elaborati e scritti su disco per poter essere persistentemente memorizzati.

File di Record

- La struttura dati tipica per la memorizzazione delle informazioni di una base di dati è quella dei *file di record*
- L'unità di informazione che viene letta o scritta da un dispositivo di memorizzazione di massa è detta anche *pagina*, tipicamente di dimensione pari a 4 o 8 KB.
- L'operazione di lettura o di scrittura di una pagina ha associato un costo (*costo di I/O*), che è di gran lunga il più pesante rispetto ai costi delle altre operazioni su una base di dati.
- Dal punto di vista logico, un file è una sequenza di record, ove ogni record, formato da uno o più *campi* è identificato attraverso un unico *identificatore* che permette di identificare sul dispositivo quale è l'indirizzo fisico della pagina contenente il record su disco.

Fattore di Blocco

ID	***Nome***	***Cognome***	***Stipendio***
003	Antonio	De Aurelis	5000
001	Marco	Marchi	2000
005	Antonio	Di Padova	3000
004	Francesco	Di Sales	1000

- Assumiamo che ogni tupla in questa relazione sia mappata in un record di un file gestito dal sistema operativo.
- Quando un utente richiede una tupla dal DBMS, quest'ultimo mappa un *record logico* in un *record fisico*, e lo associa ad una pagina in memoria primaria.

Il record fisico consiste di uno o più record logici ed il numero di record logici contenuti in un record fisico è detto *fattore di blocco*.

Organizzazione dei file

- *file non ordinati* (o *file heap*),
- *file ordinati* (o *file sequenziali*)
- *file ad accesso calcolato* (o *hash file*).

Definiamo *metodo di accesso* le tecniche necessarie per memorizzare e per recuperare i record da un file

- ... strettamente collegato al modo in cui un file è organizzato.

File HEAP

I record sono memorizzati su file in modo del tutto casuale.

- Un nuovo record che viene inserito nel file, viene posizionato nell'ultima pagine del file
  - se lo spazio è insufficiente, allora viene creata una nuova pagina e viene aggiunta in coda al file stesso
- le operazioni di *inserimento* nel file sono molto efficienti.
- l'unica possibilità di recupero dei record è quella basata su una ricerca lineare
- L'operazione di cancellazione del record richiede, al solito, la ricerca nel file del record da cancellare
  - il record viene marcato come logicamente cancellato e non viene più usato
  - deterioramento progressivo dell file quando ci sono frequenti cancellazioni; necessità di riorganizzazione del file per recuperare gli spazi lasciati inutilizzati dalle cancellazioni.

File ORDINATI

I record di un file sono ordinati su uno o più campi.

- L'operazione di ricerca di un record può fare uso di tecniche di *ricerca binaria* più efficiente della ricerca lineare.
- Le operazioni di inserimento e di cancellazione richiedono maggiori passi computazionali per mantenere l'ordine trai record.
  - L'inserimento richiede prima l'individuazione della posizione in cui inserire il record, quindi occorre fare spazio e solo a questo punto è possibile fare inserimenti. Se non c'è spazio nella pagina, occorrerà creare una nuova pagina e spostare uno o più record in essa.
  - E' utile utilizzare un file temporaneo in cui effettuare inserimenti non ordinati ed opportuni algoritmi di *merge-sort* possono essere usati ad intervalli regolari di tempo per riottenere un unico file ordinato.
  - Per la cancellazione, occorrerà riorganizzare i file per rimuovere lo spazio lasciato libero.

File HASH

La posizione del record nel file viene calcolata attraverso un'apposita funzione detta *funzione di hash*.

- Tale funzione riceve in ingresso uno o più campi del record e restituisce una posizione. Se i campi di hash sono anche chiavi del file, si parla di *chiavi di hash*.
- La funzione di hash deve essere scelta in modo tale da distribuire i record nel file in modo uniforme, anche se apparentemente casuale.
  - Un tecnica usuale che implementa la funzione di hash consiste nel convertire i valori delle chiavi di hash in numeri interi (esempio posizione alfabetica oppure codice ASCII) e nel sommare il numero ottenuto ad un opportuno offset (*tecnica di folding*) oppure effettuando una divisione in modulo per individuare la posizione (*tecnica della divisione in modulo*).

Collisioni nei File di HASH

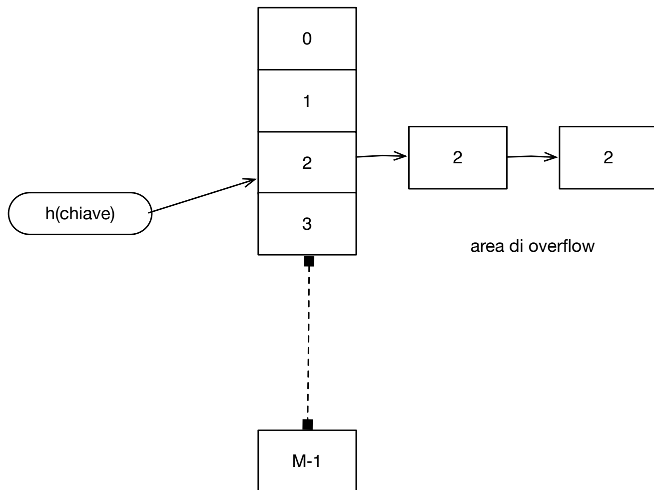
non è possibile garantire un indirizzo univoco ad ogni record, in quanto il numero dei possibili valori calcolati dei campi di hash è molto maggiore del numero di posizioni disponibile nel file.

- ogni indirizzo generato dalla funzione di hash è associato ad un *bucket* su memoria di massa contenente più record.
- all'interno del bucket i record sono memorizzati secondo l'ordine di arrivo.

In caso di record sinonimi si parla di *collisioni*: in presenza di collisioni, se il bucket è pieno occorre inserire il record in una nuova posizione.

- Uso di un'*area di overflow* in cui posizionare record sinonimi, ponendo i record in quest'area in modo libero (secondo l'ordine di arrivo) oppure collegandoli ad una lista concatenata collegata ai vari bucket.

File hash con gestione di collisioni a lista concatenata



Indici di Accesso

Un indice è una *struttura dati* che permette di organizzare in modo opportuno i record al fine di rendere efficiente il recupero dell'informazione, attraverso una *chiave* di ricerca sull'indice.

Secondo l'enciclopedia Treccani on line, un indice è "in senso generico ed etimologico (da cui si sviluppano tutti i significati particolari), qualsiasi cosa che serve a indicare. In origine usato anche come aggettivo, con il significato di *che indica, che serve a indicare*".

Perchè gli indici

Un indice è una *struttura dati* ordinata che permette di localizzare un particolare record in un file dati in modo veloce, diminuendo tempi di risposta di una query di utente.

- ... pur non essendo necessari al funzionamento di un DBMS, gli indici ne migliorano le prestazioni.
- Ad ogni file (anche non ordinato) viene associato un *file di indice* contenente la struttura dati dell'indice stesso
  - .. di solito formata da coppie chiave di ricerca e identificatore di un record.

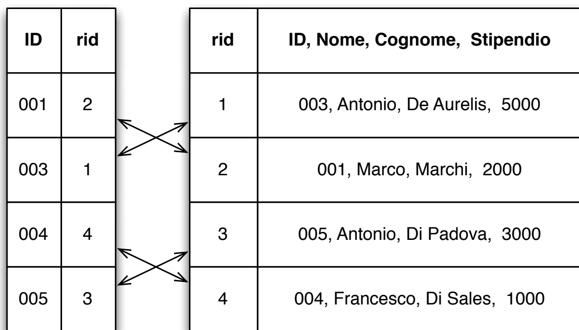
  Un indice velocizza le selezioni sui campi che compongono la chiave di ricerca per l'indice,
  - la chiave può essere formata da qualunque sottoinsieme dei campi di una relazione

Esempio

Possiamo memorizzare le informazioni in un file non ordinato mentre le posizioni si possono registrare in un file ordinato sul campo *ID*.

- Abbiamo bisogno di due file: uno (disordinato) contente i record dati identificati ognuno dal proprio identificatore (*record identifier*, *rid*); l'altro, ordinato, contenente i record dell'indice ordinati sul campo *ID*
- Nel caso in cui si vogliano fare query sul campo stipendio, può essere utile aggiungere un altro file di indice, ordinato su stipendio (*file indice ausiliario* o *secondario*).

ID	Nome	Cognome	Stipendio
003	Antonio	De Aurelis	5000
001	Marco	Marchi	2000
005	Antonio	Di Padova	3000
004	Francesco	Di Sales	1000



Dat Entry

Il *data entry* è il record memorizzato in un file indice: un indice è dunque una collezione di data entry.

- il data entry è costituito da un intero record di dati, con la sua chiave di ricerca: in questo caso, l'indice è un caso particolare di organizzazione di file;
- il data entry è una coppia formata da *(chiave, rid)*, come nell'esempio precedente;
  - in questo caso l'indice è completamente indipendente dall'organizzazione del file con i dati (heap o ordinato);
- il data entry è una coppia *(chiave, lista_di_rid)*, dove *lista_di_rid* è una lista di identificatori di record dati aventi un particolare valore della chiave di ricerca;
  - anche in questo caso l'indice è indipendente dall'organizzazione del file, permettendo una migliore gestione dello spazio.

Tipi di Indice

In letteratura sono stati presentati diversi tipi di indici:

- *indice primario*: si tratta di un indice costruito su un file sequenziale ordinato su una chiave – in altre parole, un indice primario è un indice su di un insieme di campi che include la chiave primaria di una relazione
- *indice secondario*: si tratta di un indice costruito su una chiave non primaria di una relazione
- *indice clustering*: si tratta di un indice costruito su un campo non chiave, di modo che ad ogni valore di tale campo corrispondono più record (cluster di records)

Un file può avere un solo indice primario, un solo indice di clustering e diversi indici secondari. Si distinguono inoltre gli indici *sparsi* – ovvero gli indici che hanno un record di indice solamente per alcuni valori della chiave nel file – dagli indici *densi* – ovvero un indice che ha un record di indice per ogni valore di chiave di ricerca nel file.

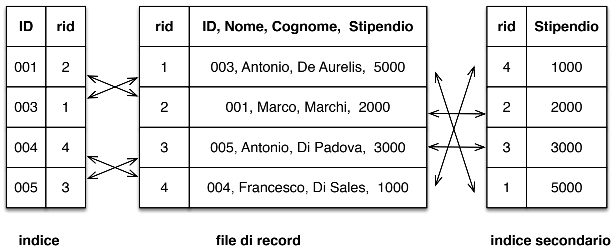


Figura: File di record di dati e di indice primario e secondario

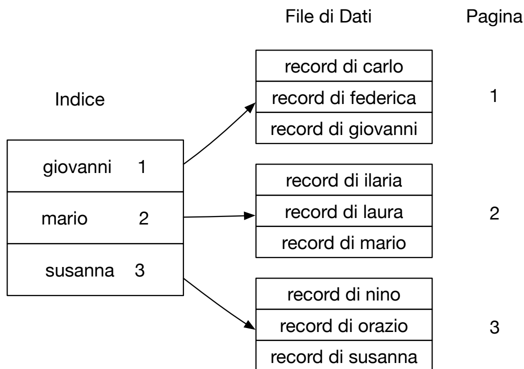
Indexed Sequential Files

Un *file sequenziale indicizzato* è un file ordinato con un indice primario. Il più famoso indice è quello definito da IBM con la struttura detta ISAM (*Indexed Sequential Access Method*), indice fortemente dipendente dall'hardware dello storage, la cui evoluzione hardware-independent ha dato luogo al *Virtual Sequential Access Method* (VSAM).

Organizzazione

Un file sequenziale indicizzato è di solito organizzato in un'area di memorizzazione primaria, in uno o più indici separati, e in una area di overflow.

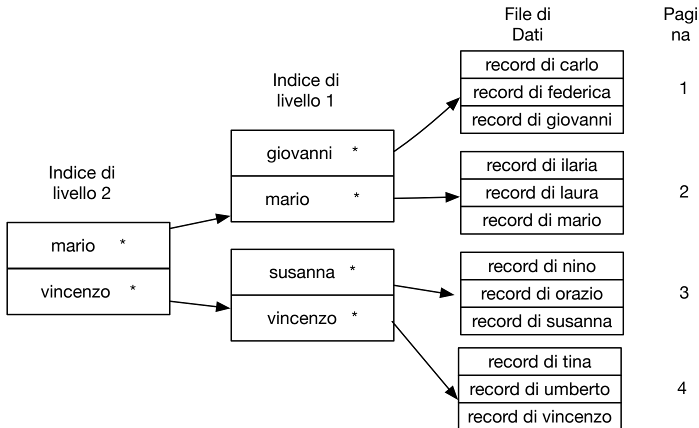
Esempio di Indice Sparso



Buona parte di un indice primario è memorizzato in memoria principale. Essendo ordinato, si possono sfruttare per le ricerche algoritmi di ricerca binaria. Nel contempo, se il file è sottoposto a continue operazioni di inserimento e di cancellazione, risulta complesso il mantenimento dell'ordine nel file di dati e nel file di indice.

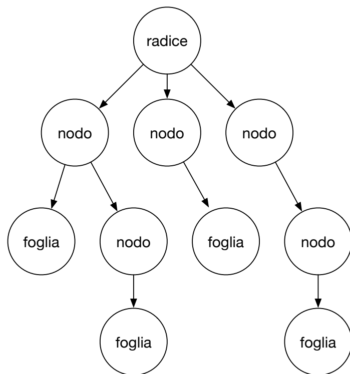
Indice Multilivello

Intuitivamente, il file comprendente k pagine viene diviso in in numero di indici di dimensione molto minore di k: per poter accedere a tali indici, si può pensare ad un *indice degli indici*.



Indici B^+ Tree

L'albero è una struttura dati gerarchica formata da *nodi* e *archi*. Ogni nodo, eccetto il nodo radice, ha un unico nodo detto *padre* e zero o più nodi *figli*: come intuibile, un nodo radice non ha padri. Un nodo che non ha nessun nodo figlio è detto anche nodo *foglia* (vedi Figura 26).



Balanced Tree

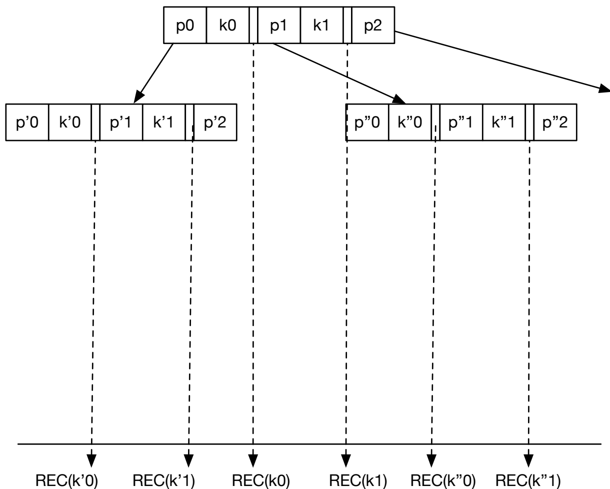
Un albero è detto *bilanciato* se la sua profondità

- ovvero il massimo numero di livelli tra il nodo radice e una foglia

è la stessa per ogni nodo foglia (in tal caso si parla anche di *Balanced Tree* o *B-Tree*)

- In generale, mantenere un albero bilanciato è molto importante ai fini dell'efficienza di una ricerca: ciò garantisce che nessun nodo si troverà a profondità troppo alta rispetto agli altri, richiedendo così molti accessi ai blocchi durante la ricerca stessa.
- A tal fine, è necessario disporre di algoritmi che, in fase di inserimento o di cancellazione di un nuovo record, provvedano ad aggiornare l'albero cercando di mantenere tutti i nodi foglia allo stesso livello.

Il B-tree



Definizione di un B- tree

- ogni nodo dell'albero contiene $\langle P_1, K_1, P_{K_1}, P_2, K_2, P_{K_2} \ldots P_q \rangle$, con $q \leq p$, ove $K_1 < K_2 \ldots < K_{q-1}$ ed essendo ogni K_i una chiave, P_{K_i} punta ai dati relativi alla chiave K_i, P_i è il puntatore ad un sottoalbero che contiene tutti i valori di chiave $K \leq K_i$, P_q punterà ad un sottoalbero con valori di chiavi $K > K_{q-1}$;
- ogni nodo può contenere al massimo p puntatori dell'albero;
- ogni nodo, tranne la radice e i nodi foglia, deve contenere almeno $p/2$ puntatori dell'albero e il nodo radice contiene almeno 2 nodi figli;
- tutti i nodi foglia sono allo stesso livello.

Costruzione del B - tree (1)

- Un albero B inizia con un solo nodo radice a livello 0.
- Quando il nodo radice diventa completo, il nodo si divide (operazione di *split*) in due nodi di livello 1: il nodo radice conterrà solo il valore centrale del nodo, mentre il resto dei valori sarà equamente diviso tra i due nodi figli.
- Quando uno dei due nodi si riempie, tale nodo viene diviso in due nodi dello stesso livello e viene posto un puntatore al nodo padre ponendo in esso i valore di chiave centrale.
- Quando il nodo padre si riempie, viene diviso pure esso: se la divisione si propaga fino al nodo radice, e se anche la radice deve essere divisa, si crea un nuovo livello dell'albero.

Costruzione del B - tree (2)

- Per quanto attiene invece la cancellazione, se la cancellazione di un valore fa sì che il nodo sia completo per meno della metà, allora il nodo viene fuso (operazione di *merge*) con uno dei suoi vicini: si noti che anche la cancellazione può propagarsi fino alla radice.

B^+Tree

La maggior parte delle implementazioni commerciali utilizza una variante dell'albero B detta *B^+ Tree*.

- Nell'albero B^+Tree, i puntatori ai dati sono memorizzati esclusivamente nei nodi foglia dell'albero.
- Tali nodi foglia, inoltre, sono collegati tra di loro (lista collegata).
- Un B^+Tree richiede approssimativamente lo stesso tempo per accedere ad un qualunque record da esso indicizzato, e quindi assicura che vengono attraversati circa lo stesso numero di nodi prima di arrivare alla radice (il tempo di ricerca è dunque funzione della profondità dell'albero).

Esempio di B^{+}Tree

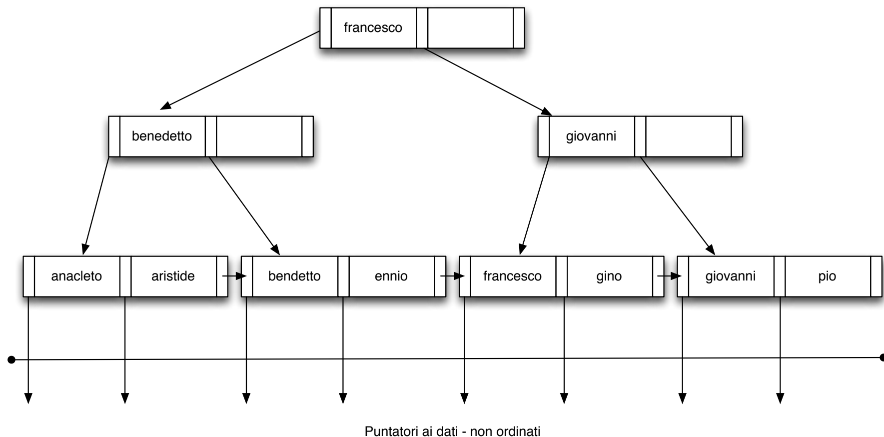


Figura: Esempio di B^{+}Tree

Vantaggi di un B^+Tree

L'indice in questione è un *indice denso*: ogni record è indicizzato e questo fa sì che il file di dati non debba essere ordinato.

- Ciò che è costoso è, come per il B-Tree, l'inserimento e la cancellazione.
- Si noti inoltre che rispetto al B-Tree, il B$^+$Tree permette di implementare in modo più efficienti le *range query*, ovvero le ricerche su intervalli di valori.

In letteratura (ed in commercio) sono stati presentati diverse varianti del B$^+$Tree, in particolare il B^* *Tree*, che è un B$^+$Tree con il vincolo che ogni nodo sia completo almeno per due terzi.

Indici per DWH

Indici di Bitmap

Sono usati generalmente su quegli attributi i cui domini presentano un numero ridotto di valori possibili. Memorizzano un vettore di bit per ogni valore di un attributo, in particolare un bit per ogni tupla: il bit ha valore 1 se la tupla contiene il particolare valore dell'attributo, 0 altrimenti.

Indici di Join

Permettedi pre-calcolare una operazione di join, evitando di ricalcolare il tutto ogni qualvolta viene eseguita una query. L'indice di join rende quindi efficiente l'operazione di join effettuandolo in anticipo e creando un indice per rispondere a query di analisi sui dati integrati.

Esempio di Indice di Bitmap

Tabella IMPIEGATI

ID	Nome	Cognome	Ruolo	Sesso
1	Carlo	Carli	manager	m
2	Giancarlo	Sperlì	analista	m
3	Lucio	Sansone	manager	m
4	Flora	Amato	programmatore	f
5	Antonio	Picariello	programmatore	m
6	Vicenzo	Moscato	analista	m

Indice di Bitmap sull'attributo "Ruolo"

rid	manager	analista	programmatore
001	1	0	0
002	0	1	0
003	1	0	0
004	0	0	1
005	0	0	1
006	0	1	0

Gestione della Memoria

Il Buffer Manager

E’ responsabile della gestione efficiente dei buffer che sono usati per trasferire pagine da e verso la memoria secondaria, cercando di ridurre il numero di accessi alla memoria secondaria

Buffer

E’ una area di memoria centrale, gestita dal DBMS e condivisa fra le transazioni: esso è di solito organizzato in pagine di dimensioni pari o multiple di quelle dei blocchi di memoria secondaria (1KB-100KB)

Buffer Manager: come funziona

Il gestore del buffer deve in generale leggere pagine dal disco e portarle nel buffer fino a che quest'ultimo non si riempie.

- Quando si è riempito, occorre definire una strategia per decidere come liberare spazio nel buffer per una nuova pagina che deve ad esempio essere letta dal disco.

Esempi di politiche

Esempi classici di tali strategie sono le politiche di tipo

- FIFO (*First In First Out*)
- LRU (*Last Recently Used*).

Funzionamento del BM - 1

Variabili di stato, *count* e *dirty*, inizialmente posti a zero.

- *count* indica quanti programmi stanno utilizzando una certa pagina,
- *dirty* indica se la pagina è stata modificata (sporcata) o meno.

Il gestore del buffer riceve dai layers di livello superiore richieste di lettura o di scrittura, e le esegue, utilizzando il buffer in memoria centrale o, quando non possibile, accedendo alla memoria di massa.

Funzionamento del BM - 2

Quando si richiede una pagina dal disco, il gestore del buffer verifica se la pagina è già presente in memoria centrale. In particolare:

- se la pagina cercata è presente nel buffer, incrementa il *count* di 1; se viene modificata, pone *dirty* a 1;
- se la pagina non è presente, viene scelta nel buffer una pagina libera usando una delle politiche FIFO o LRU: tale pagina, se *dirty*, viene salvata su memoria secondaria;
- se non esiste alcuna pagina libera, il gestore del buffer si può comportare con politiche di tipo *steal* o *no steal*;
- a questo punto la pagina richiesta viene trasferita in memoria centrale, con $count = 1$ e $dirty = 0$.

Politiche

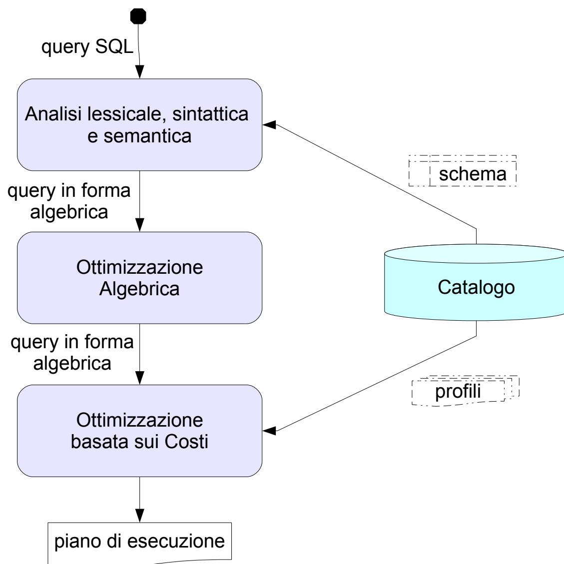
- politica *steal* – permette al gestore del buffer di scrivere su disco prima del commit di una transazione: il gestore “ruba” una pagina dalla transazione.
- ... se ciò non avviene, si parla di politica *no steal* e l’operazione viene posta in attesa.
- politica *force* – assicura che tutte le pagine modificate da una transazione siano immediatamente scritte su memoria di massa non appena la transazione ha fatto il commit.
- ... alternativamente si ha una politica *no force*.

Query Processor

Il *Gestore delle Query* ha il compito di effettuare l'*ottimizzazione* delle query.

- l'ottimizzatore sceglie la strategia esecutiva "migliore" per una data query (di solito fra più alternative) a partire dall'istruzione SQL al fine di minimizzare i tempi di risposta del sistema di basi di dati.

Processo di ottimizzazione delle query - 1



Processo di ottimizzazione delle query - 2

- *analisi lessicale, sintattica e semantica* – basata sulle informazioni sullo schema della base di dati presenti nel catalogo, per verificare la correttezza di una query, di cui viene poi determinata la relativa *forma algebrica*;
- *ottimizzazione algebrica* – per trasformare la query in ingresso una forma *equivalente* a quella di partenza ma che sia più efficientemente eseguibile;
- *ottimizzazione basata sui costi* – sfruttando le informazioni sulle relazioni della base di dati prelevate anch'esse dal catalogo, è in grado di determinare il piano di esecuzione finale (sequenza di operazioni di basso livello che consentono di eseguire la query).

L'ottimizzazione algebrica si basa sul concetto di *equivalenza algebrica* dell'algebra relazionale.

Definizione (Espressioni dell'algebra equivalenti)

Due espressioni dell'algebra relative ad interrogazioni sulla base di dati sono equivalenti se producono lo stesso risultato qualunque sia l'istanza attuale della base di dati.

Euristica fondamentale: *push selections down* e *push projections down*).

Esempio

IMPIEGATI(*Matricola*, *Nome*, *Cognome*,*Dip*)
DIPARTIMENTI(*Cod*, *Nome*, *Indirizzo*,*Città*):

SELECT *
FROM DIPARTIMENTI D **JOIN** IMPIEGATI I **ON** D.Cod=I.Dip
WHERE I.Cognome='Moscato';

Esempio di Ottimizzazione Algebrica

$$\sigma_{I.Cognome=`Moscato'}(D \bowtie_{D.Cod=I.Dip} I)$$

pushing selection down

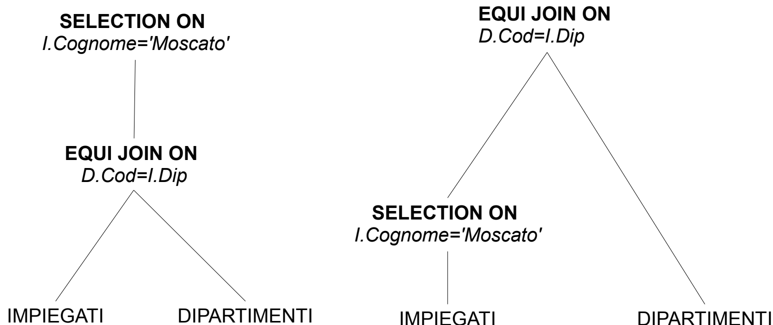
Se A e B due generiche relazione e $\hat{\chi}$ una condizione di selezione valida sugli attributi della relazione B allora:

$$\sigma_{\hat{\chi}}(A \bowtie_{\chi} B) \equiv (A \bowtie_{\chi} \sigma_{\hat{\chi}}(B))$$

$$(D \bowtie_{D.Cod=I.Dip} \sigma_{I.Cognome=`Moscato'}(I))$$

che può essere eseguita più efficientemente in quanto anticipando la selezione rispetto al join permette di ridurre il numero di tuple nell'operazione di join.

Rappresentazione con alberi di query



Procedura di ottimizzazione

- decomporre le selezioni congiuntive (*AND*) in successive selezioni atomiche;
- anticipare il più possibile le selezioni;
- in una sequenza di selezioni, anticipare le più selettive;
- combinare prodotti cartesiani e selezioni per formare join;
- anticipare il più possibile le proiezioni (anche introducendone di nuove).

Ottimizzazione basata sui costi

- Una volta trovato per una data query l'albero della sua rappresentazione algebrica di più "conveniente", il DBMS effettua un'ottimizzazione basata sui costi sfruttando alcune informazioni sulle relazioni della base di dati
- In particolare, le informazioni sulle relazioni presenti in una base di dati sono dette *profili* e contengono una serie di dati quantitativi come:
  - cardinalità di ciascuna relazione
  - dimensioni delle tuple
  - dimensioni degli attributi
  - numero di valori distinti degli attributi
  - valore minimo e massimo di ciascun attributo, ecc.

Piani di Query

I profili sono utilizzati nell'ottimizzazione delle query per stimare le dimensioni dei risultati intermedi e scegliere, di volta in volta, quale sia la migliore sequenza possibile di operazioni di più basso livello:

- *scansione*, *accesso diretto*, *ordinamento*, *join*

Compile & Store

Il piano di esecuzione può essere memorizzato all'interno del catalogo ed essere richiamato più volte qualora la stessa query necessiti di essere eseguita nuovamente

Compile & Go

È possibile determinare di volta in volta il piano di esecuzione delle query

Gestione delle Transazioni: Perchè

La presenza simultanea di programmi utente che richiedono l'accesso al contenuto di una base di dati può generare un numero elevato i di operazioni contemporanee che un DBMS deve gestire garantendo, nel contempo,

- la *consistenza*
- *integrità* dei dati,
- *affidabilità* del sistema di basi di dati

Gestore della Concorrenza e Gestore dell'Affidabilità

Il DBMS possiede un modulo per la *Gestione della Concorrenza* ed uno per la *Gestione dell'Affidabilità* che si occupano di garantire l'accesso concorrente alla base di dati, preservandone il contenuto anche in corrispondenza di crash del sistema.

Definizione (Transazione)

Una transazione su una base di dati è un'operazione, o una sequenza di operazioni, generata da un utente o programma applicativo che legge o aggiorna il contenuto di una base di dati.

Transazioni

- Una transazione rappresenta quindi un'*unità logica di elaborazione* che corrisponde a una serie di operazioni fisiche elementari (letture/scritture) sulla base di dati.
  - Essa può coincidere con un intero programma, una parte di un programma o con un singolo comando (es. uno statement SQL di INSERT o UPDATE) che vengono eseguiti in un ambiente DBMS,
  - ... nel contempo, costituisce una unità *atomica* (non divisibile) di modifiche fatte allo stato di una base di dati.

Commit e Abort

una transazione o termina in uno stato finale previsto dal programma in esecuzione – *commit* – o porta il sistema ad uno stato precedente all'esecuzione della transazione – *abort*.

Esempio di Transazione

UPDATE PRODOTTI **SET** Quantità=Quantità-y **WHERE** Codice=x;
INSERT INTO CARRELLO(Cliente, Prodotto, Qta, Data)
VALUES (z,x,y, '13-Gen-2015 20:30:54');

- transazione *committed*– la base di dati è portata in in un nuovo stato consistente.
- transazione *aborted* – la basi di dati è riportata nello stato precedente alla sua esecuzione.
  - Nel caso di abort, eventuali azioni ed effetti parziali della transazione devono essere essere "disfatti" (*rollback* o *undo*), in quanto lascerebbero la basi di dati in uno stato di inconsistenza.

Proprietà ACID - 1

Definizione (Proprietà ACID: atomicità)

È la cosiddetta proprietà del “tutto o niente”: una transazione atomica deve essere eseguita nella sua interezza oppure non eseguita per niente.

Definizione (Proprietà ACID: consistenza)

Una transazione è consistente quando causa una trasformazione di uno stato consistente della base di dati in un altro stato consistente.

- *Un DBMS, in particolare, deve poi assicurare che tutti i vincoli definiti sulla base di dati siano soddisfatti durante l’esecuzione di transazioni.*

Proprietà ACID - 2

Definizione (Proprietà ACID: isolamento)

Le transazioni concorrenti sono isolate quando sono eseguite in modo indipendente l'una dalle altre.

- *Questo sta a significare che gli effetti parziali di transazioni incomplete non devono essere visibili alle altre transazioni.*

Definizione (Proprietà ACID: durability o persistenza)

Una transazione che è terminata con un "commit" deve essere persistente, ovvero i suoi effetti devono essere registrati in modo permanente nella base di dati e non devono essere mai persi – per alcun motivo.

Blocco di una Transazione

Una transazione coincide con un blocco di istruzioni caratterizzato da un inizio (begin transaction) e da una fine (end transaction) e al cui interno deve essere eseguito una e una sola volta almeno uno dei seguenti comandi:

- commit, per terminare correttamente e rendere persistenti le azioni delle transazione,
- rollback, per abortire la transazione.

begin transaction
    UPDATE PRODOTTI **SET** Quantità=Quantità-y **WHERE** Codice=x;
    INSERT INTO CARRELLO(Cliente, Prodotto, Qta, Data);
    VALUES (z,x,y, '13-Gen-2015 20:30:54');
commit

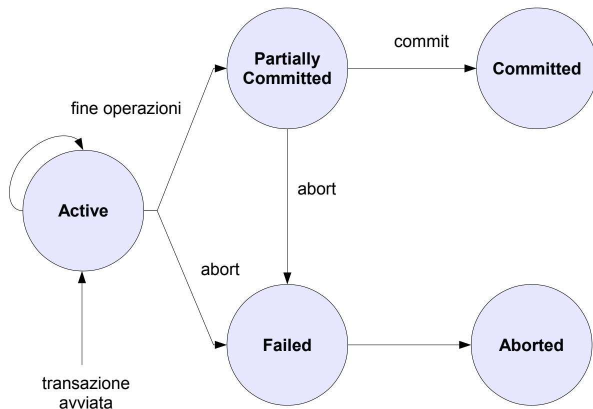
Transazioni e Vincoli di Integrità

I vincoli di integrità dei dati possono essere "controllati" in modo

- *differito* – ovvero a valle della richiesta di commit della transazione
- *diretto* – ovvero durante l'esecuzione della transazione stessa evitandone l'abort.

```
SELECT Quantità-x INTO Q FROM PRODOTTI WHERE Codice=x;
IF (Q> 0) THEN
        UPDATE PRODOTTI SET Quantità=Quantità-y WHERE Codice=x;
        INSERT INTO CARRELLO(Cliente, Prodotto, Qta, Data)
        VALUES (z,x,y, '13-Gen-2015 20:30:54');
        commit
ELSE
        rollback
END IF
```

Diagramma degli Stati di una Transazione



Stati di una Transazione

- *Partially Committed* (Parzialmente Committed): si verifica quando viene eseguita l'ultima istruzione di una transazione.
  - A questo punto, la transazione potrebbe essere abortita se, ad esempio ..
    - viene violato qualche vincolo di integrità
    - oppure se il sistema ha subito un crash e non si è avuta la possibilità di salvare in memoria secondaria i dati aggiornati dalla transazione stessa.
  - In entrambi i suddetti casi, la transazione si porta prima nello stato *Failed* in attesa di essere poi definitivamente abortita.
  - Se invece, all'atto del commit, la transazione viene completata con successo e tutti i suoi effetti sono registrati su memoria secondaria, essa può passare allo stato *Committed*.
- *Failed* (Fallita): si verifica quando
  - la transazione non può essere per qualche motivo committata in maniera corretta oppure
  - se la transazione viene abortita mentre si trova in esecuzione.

Gestione delle Transazioni: moduli

- l'atomicità e persistenza sono garantite dal *Gestore dell'Affidabilità*;
- l'isolamento sono garantite dal *Gestore della Concorrenza*;
- la consistenza sono garantite dal *Gestore dell'integrità a tempo di esecuzione* – con il supporto del compilatore del DDL.

Supporto all'Accesso Concorrente

Abbiamo visto che uno degli obiettivi principali di un sistema di basi di dati è quello di supportare l'accesso *concorrente* da parte di utenti ed applicazioni a dati condivisi.

- accessi concorrenti, di cui almeno uno prevede un aggiornamento, potrebbero provocare problemi di integrità e consistenza dei dati, noti anche col nome di *anomalie*.

Soluzione Banale

Si potrebbe prescrivere un accesso "serializzato" alla base di dati, in cui le operazioni delle varie transazioni si succedono temporalmente in sequenza senza causare anomalie.

Tuttavia, è impensabile nei sistemi OLTP, con decine o centinaia di transazioni generate al secondo, prevedere un accesso di tipo seriale che ne rallenterebbe il tempo di risposta.

Massimizzare tps

Obiettivo del Controllo della Concorrenze

Un DBMS deve fornire un apposito modulo per il controllo della concorrenza il cui obiettivo sia massimizzare il numero di transazioni servite per secondo (*throughput*), minimizzando i tempi medi di risposta.

- .... eseguire più transazioni in concorrenza, alternando l'esecuzione di operazioni di una transazione con quella di operazioni di altre transazioni (*interleaved execution*).
  - Come avviene nei sistemi operativi, si sfrutta il fatto che, mentre una transazione è in attesa del completamento di una operazione di I/O, un'altra può utilizzare la CPU.

L'esecuzione alternata di più transazioni concorrenti però non previene il verificarsi di possibili anomalie. Il gestore della concorrenza dovrà quindi implementare un'apposita politica in grado di prevenire la maggior parte delle anomalie.

Modello di Transazioni ed Anomalie

Modello di Transazione

Prevede un solo livello di controllo ed in cui: il comando begin transaction (bot) dichiara l'inizio di una transazione, il comando commit indica il raggiungimento di un nuovo stato consistente, mentre il comando rollback un abort.

- le singole operazioni di una transazione sono modellate attraverso operazioni di *read(x,y)* e *write(x,y)*, dove *x* rappresenta un oggetto astratto della base di dati (es. campi di tabelle) ed *y* il valore letto o da scrivere; inoltre le singole operazione sono "etichettate" attraverso l'istante temporale in cui avvengono
  - Nella pratica, viene effettuata un'esemplificazione delle operazioni in memoria centrale su tali oggetti, ignorando che le operazioni richiedono l'intera pagina dove risiedono i dati.

Esempio di Modello di Transazione

t_1: **bot**
t_2: $read(qtaP_x, A)$
t_3: $A = A - y$
t_4 : $write(qtaP_x, A)$
t_5: $B = z$
t_6: $write(clienteC_{xz}, B)$
t_7 : $C = x$
t_8 : $write(prodC_{xz}, C)$
t_7 : $D = y$
t_{10}: $write(qtaC_{xz}, D)$
t_{11}: $E =$‘13-Gen-2015 20:30:54’
t_{12}: $write(dataC_{xz}, E)$
t_{13}: **commit**

Esemplificazione

Supponendo che nel momento della procedura di acquisito di un prodotto sia automaticamente creato per ogni utente un record nel carrello con la data di sistema (con NULL in corrispondenza della quantità acquistata), la relativa transazione potrebbe semplificarsi come segue.

t_1: **bot**
t_2: $read(qtaP_x, A)$
t_3: $A = A - y$
t_4: $write(qtaP_x, A)$
t_5: $write(qtaC_{xz}, y)$
t_6: **commit**

Perdita di aggiornamento

La *perdita di aggiornamento* (*Lost Update*) si verifica quando un'operazione di aggiornamento di una transazione si sovrappone a quella di un'altra che stava aggiornando lo stesso oggetto (prima che quest'ultima abbia effettuato il commit), annullandone nella pratica gli effetti.

Lost Update

Ad esempio, si considerano due transazioni di acquisito dello stesso prodotto generate da due utenti diversi utenti (z_1 e z_2) e sovrapposte in parte temporalmente. In particolare, si suppone che la seconda transazione inizia la propria esecuzione e legge il valore della quantità del prodotto x disponibile in magazzino prima che la prima transazione abbia effettuato alcun aggiornamento ed il conseguente commit. .

Lost Update

In tal caso, se supponiamo che il valore iniziale della quantità disponibile sia 100 e che la prima transazione abbia acquistato 50 unità di prodotto, mentre la seconda 30, alla fine delle esecuzione delle due transazioni il valore finale della quantità disponibile in magazzino sarà 70 anziché 20, causando la perdita di aggiornamento della prima transazione ed un'inconsistenza pericolosa (nel carrello risulta che la prima transazione ha acquistato correttamente 50 unità di prodotto)

Lost Update Esempio

Time	T_1	T_2	$qtaP_x$
t_1	**bot**		100
t_2	$read(qtaP_x, A)$	**bot**	100
t_3	$A = A - 50$	$read(qtaP_x, A)$	100
t_4	$write(qtaP_x, A)$	$A = A - 30$	50
t_5	$write(qtaC_{xz_1}, 50)$	$write(qtaP_x, A)$	70
t_6	**commit**	$write(qtaC_{xz_2}, 30)$	70
t_7		**commit**	70

Tabella: Esempio di perdita di aggiornamento

Lettura Sporca

La *lettura sporca* (*Dirty Read*) si verifica quando una transazione, durante la sua esecuzione, legge un valore “sporco” di un oggetto, ovvero modificato da un’altra transazione che poi viene abortita.

Esempio di Lettura Sporca

É possibile immaginare una situazione in cui due transazioni di acquisto dello stesso prodotto generate da utenti diversi si sovrappongono temporalmente.
Questa volta si suppone che la prima, dopo avere aggiornato la quantità di prodotto disponibile ed il relativo record nel carrello, effettua in un istante di tempo successivo agli aggiornamenti un abort spontaneo (ad esempio voluto dall’utente che desidera per qualche motivo annullare la transazione). La seconda transazione, che si ipotizza essere iniziata dopo le operazioni di aggiornamento della prima, opererà su un valore non corretto della quantità di prodotto disponibile che non è 50 ma in realtà 100, limitando di conseguenza le unità di prodotto che la stessa può potenzialmente acquistare.

Esempio di Lettura Sporca

Time	T_1	T_2	$qtaP_x$
t_1	**bot**		100
t_2	$read(qtaP_x, A)$		100
t_3	$A = A - 50$		100
t_4	$write(qtaP_x, A)$		50
t_5	$write(qtaC_{xz_1}, 50)$	**bot**	50
t_6	...	$read(qtaP_x, A)$	50
t_7	...	$A = A - 30$	50
t_8	...	$write(qtaP_x, A)$	20
t_9	**rollback**	$write(qtaC_{xz_2}, 30)$	70
t_{10}		**commit**	70

Tabella: Esempio di lettura sporca

Lettura Inconsistente

La *lettura inconsistente* (i*Inconsistent Read*) si verifica quando una transazione legge in due diversi istanti della sua esecuzione due valori differenti valori dello stesso oggetto che viene, intanto, modificato da un'altra transazione.

Esempio Lettura Inconsistente - 1

Si ipotizza la presenza di una transazione di acquisto a cui si sovrappone temporalmente l'esecuzione di una transazione che effettua due letture in istanti successivi della quantità di un prodotto disponibile in magazzino: la prima prima dell'inizio della transazione d'acquisto la seconda a valle del commit.

Esempio Lettura Inconsistente -2

Ad esempio, la transazione che effettua le letture può essere quella generata da una una procedura applicativa di riordino di merci che dopo la prima lettura seleziona i prodotti sotto scorta per cui vanno richieste nuove unità ai fornitori e dopo la seconda controlla le quantità prima di confermare il riordino.
In questo caso, se l'obiettivo è quello di mantenere almeno 50 unità in magazzino per ogni prodotto, la quantità iniziale del prodotto è 20 e la transazione d'acquisto riguarda 20 unità di prodotto, prima della selezione dei prodotti sotto scorta verrà letto il valore 20 e verrà generato un ordine di 30 unità per il prodotto in questione, prima della conferma degli ordini ai fornitori invece verrà letto il valore 0 che potrebbe comportare l'annullamento o aggiornamento delle unità da ordinare in quanto non più corrette rispetto all'ordine precedentemente generato.

Esempio di Inconsistent Read

Time	T_1	T_2	$qtaP_x$
t_1	**bot**		20
t_2	$read(qtaP_x, A)$		20
t_3	...		20
t_5	...	**bot**	20
t_6	...	$read(qtaP_x, A)$	20
t_7	...	$A = A - 20$	20
t_8	...	$write(qtaP_x, A)$	0
t_9	...	$write(qtaC_{xz}, 30)$	0
t_{10}	...	**commit**	0
t_{11}	$read(qtaP_x, A)$		0
t_{12}	...		0
t_{13}	**commit**		0

Tabella: Esempio di lettura inconsistente

Aggiornamento Fantasma

L'*aggiornamento fantasma* (*Ghost Update*) si verifica quando una o più operazioni di aggiornamento di una transazione che riguardano due o più oggetti della base di dati il cui valore è correlato (ad esempio, causa presenza di un vincolo di integrità) non vengono "viste" correttamente da altre transazioni temporalmente sovrapposte.

- Per queste ultime, è come se alcuni degli aggiornamenti non siamo mai stati effettuati sulla base di dati e per questo si definiscono "fantasmi".

Esempio Aggiornamento Fantasma – 1

In questo caso si può ipotizzare la presenza di due transazioni sovrapposte temporalmente. Una relativa all'acquisito di un prodotto costituito da due componenti (x_1 e x_2 anch'esse prodotti) che vanno ordinati in coppia (ad esempio un kit con cuffie e microfono) e per cui esiste il vincolo che le quantità in magazzino delle due componenti sia uguale ($qtaP_{x_1} = qtaP_{x_2}$).

Esempio Aggiornamento Fantasma – 2

L'altra transazione legge invece periodicamente le quantità disponibili per le due componenti per controllare ad esempio che coincidano i valori dei relativi oggetti della base di dati. Se quest'ultima legge il valore della quantità disponibile di una delle due componenti prima dell'esecuzione della transazione d'acquisito e il valore dell'altra componente quando invece la transazione d'acquisto ha modificato il valore della prima componente ma non della seconda, allora il vincolo di integrità per la transazione di controllo risulterà violato (quando in realtà non lo è) ed è come se il secondo aggiornamento non fosse mai avvenuto.

Time	T_1	T_2	$qtaP^{x_1}$	$qtaP^{x_2}$
t_1	**bot**		100	100
t_2	$read(qtaP_{x_1}, A)$		100	100
t_2	...	**bot**	100	100
t_3	...	$read(qtaP_{x_1}, A)$	100	100
t_4	...	$A = A - 30$	100	100
t_5	...	$write(qtaP_{x_1}, A)$	70	100
t_6	...	$write(qtaC_{x_1 z}, 30)$	70	100
t_7	...	$read(qtaP_{x_2}, B)$	70	100
t_8	...	$B = B - 30$	70	100
t_9	...	$write(qtaP_{x_2}, B)$	70	70
t_{10}	...	$write(qtaC_{x_2 z}, 30)$	70	70
t_{11}	...	**commit**	70	70
t_{12}	$read(qtaP_{x_2}, B)$		70	70
t_{13}	$qtaP_{x_1} \neq qtaP_{x_2}$		70	70
t_{14}	**commit**		70	70

Controllo della Concorrenza -CdC: Teoria

Definizione (Transazioni CdC)

Ogni transazione coincide con una sequenza temporale di azioni di lettura/scrittura su oggetti della base di dati.

Sono esempi di transazioni le seguenti sequenze:

$$T_1 : r_1(X), w_1(X), r_1(Y), r_1(Z), w_1(Z)$$
$$T_2 : r_2(X), r_2(Y), r_2(Z), w_2(Y)$$

- Nella pratica viene omesso ogni riferimento alle operazioni di manipolazione in memoria da parte della transazione, così come sono omessi il comando di begin transaction e quelli di commit e rollback.

Transazioni e CdC

- Le transazioni avvengono concorrentemente e pertanto le operazioni di lettura e scrittura vengono richieste in istanti successivi da varie transazioni.
- L'obiettivo di un protocollo di controllo di concorrenza è quello di eseguire le singole operazioni delle varie transazioni in un ordine tale da evitare il verificarsi di anomalie.

Definizione (Schedule)

Uno schedule è una sequenza di operazioni di lettura/scrittura generate da un insieme di transazioni concorrenti che preserva l'ordine temporale di esecuzione delle operazioni di ciascuna transazione.

Definizione (Scheduler)

Uno scheduler è un sistema che accetta o rifiuta o riordina le operazioni richieste dalle transazioni.

Esempi di schedulazioni possibili

Esempi di possibili schedule per le transazioni T_1 e T_2 sono mostrati di seguito.

$$S_1 : r_1(X), r_2(X), w_1(X), r_1(Y), r_1(Z), w_1(Z), r_2(Z), r_2(Y), w_2(Y)$$
$$S_1 : r_2(X), r_1(X), w_1(X), r_1(Y), r_2(Z), r_2(Y), w_2(Y), r_1(Z), w_1(Z)$$

Commit Proiezione e Schedulatori Seriali

Commit Proiezione

Vengono ignorate le transazioni che vanno in abort, rimuovendo tutte le loro azioni dallo schedule.

Definizione (Schedule seriale)

Uno schedule seriale è un particolare schedule in cui le operazioni di ciascuna transazione vengono eseguite in maniera consecutiva senza la sovrapposizione di operazioni generate da altre transazioni.

Definizione (Schedule serializzabile)

Uno schedule serializzabile è un particolare schedule non seriale che però produce gli stessi risultati (in termini di aggiornamento dello stato di una base di dati) di uno schedule seriale.

Esempi di Schedulatori Seriale e Serializzabili

Un esempio di schedule seriale è il seguente.

$$S_3 : r_1(X), w_1(X), r_1(Y), r_1(Z), w_1(Z), r_2(X), r_2(Y), r_2(Z), w_2(Y)$$

Lo schedule che segue è non seriale.

$$S_4 : r_1(X), w_1(X), r_2(X), r_2(Y), r_2(Z), w_2(Y), r_1(Y), r_1(Z), w_1(Z)$$

- Si nota però che tale schedule è serializzabile, in quanto l'oggetto X è utilizzato esclusivamente dalla prima transazione, e quindi quest'ultima dopo aver usato tale oggetto può lasciare spazio alle operazioni della seconda transazione prima di riprendere la sua esecuzione.

Obiettivo del Gestore della Concorrenze

- Un gestore della concorrenza dovrà implementare quindi apposite politiche di controllo di concorrenza (scheduler) capaci di produrre schedule serializzabili. per le transazioni in esecuzione concorrente.
- L'obiettivo delle teoria del controllo di concorrenza è quindi individuare classi di schedule serializzabili che siano sottoclassi degli schedule possibili e la cui proprietà di serializzabilità sia però verificabile a basso "costo".

Nella pratica i DBMS implementano apposite tecniche di controllo di concorrenza che garantiscono direttamente la serializzabilità degli schedule generati in ambienti concorrenti. Tali tecniche si suddividono in due classi principali:

- metodi basati su *lock*,
- metodi basati su *timestamp*.

Tali metodi sono anche detti *pessimistici* o *conservativi* in quanto tendono a "ritardare" l'esecuzione di transazioni che potrebbero generare conflitti, e quindi anomalie, rispetto allo schedule corrente.

- In contrapposizione a tali tecniche, esistono metodi, detti *ottimistici* ed applicabili in caso di ambienti concorrenti in cui conflitti tra le operazioni delle transazioni sono rari, che di contro permettono l'esecuzione sovrapposta e non sincronizzata di transazioni ed effettuano un controllo sui possibili conflitti generati solo a valle del commit.

Metodi basati sui lock - 1

- ogni oggetto della base di dati è protetto da un apposito lock (blocco);
- ogni transazione che vuole effettuare una lettura di un oggetto deve “richiederne” l’autorizzazione attraverso un’operazione di $read_lock(x)$ e, solo una volta “acquisito” il lock, può eseguire la lettura;
  - il lock in lettura su un dato oggetto può essere *condiviso* da più transazioni;
- ogni transazione che vuole effettuare una scrittura su un oggetto deve “richiederne” l’autorizzazione attraverso un’operazione di $write_lock(x)$ e, solo una volta “acquisito” il lock, può eseguire la scrittura;
  - il lock in scrittura su un dato oggetto è *esclusivo*, ovvero può essere acquisito da una sola transazione per volta;

Metodi basati sui lock - 2

- quando una stessa transazione prima legge e poi scrive un oggetto, può richiedere subito un lock esclusivo in scrittura oppure chiedere prima un lock condiviso in lettura e successivamente uno esclusivo (*lock escalation*);
- una volta che una transazione ha ottenuto un lock in lettura e scrittura su un oggetto, una volta terminata l'operazione, deve rilasciarlo attraverso un'operazione di *unlock*.
- ogni oggetto della base di dati si può trovare in uno dei seguenti stati: "libero" (*free*), "bloccato in lettura" da qualche transazione (*r_locked*) o "bloccato in scrittura" (*r_locked*).

Il gestore della concorrenza possederà una componente particolare, detta *lock manager*, che riceve le varie richieste di lock dalle transazioni e le accoglie o rifiuta, sulla base della *tavola dei conflitti*, dove per ogni richiesta di lock su un oggetto e stato dello stesso (*free*, *r_locked*, *w_locked*) vengono riportati l'esito della richiesta (Si/No) ed il nuovo stato.

	Stato dell'Oggetto		
Richieste	*free*	*r_locked*	*w_locked*
read_lock	(Si,*r_locked*)	(Si,*r_locked*)	(No,*w_locked*)
write_lock	(Si,*w_locked*)	(No,*r_locked*)	(No,*w_locked*)
unlock	errore	(Si,*free*/*r_locked*)	(Si,*free*)

Tabella: Tavola dei conflitti del lock manager

2 Phase Locking

- La *granularità* del lock può essere variegata: è possibile imporre dei lock su interi file della base di dati, su particolari pagine, su record o su campi di un record.

Locking a Due Fasi

Si richiede che ogni transazione acquisisca mediante richieste di *read_lock* e *write_lock*- in una prima fase detta *crescente* - tutti i lock sugli oggetti su cui dovrà effettuare operazioni di lettura/scrittura. Solo dopo avere terminato tutte le operazioni, la transazione rilascerà - in una seconda fase detta *decrescente* - tutti i lock precedentemente acquisiti mediante le primitive di *unlock*.

- In altri termini i vincoli imposti dal 2PL sono: i) ogni transazione deve proteggere tutte le letture e scritture con lock; ii) una transazione, dopo aver rilasciato un lock non può acquisirne altri.

Esempio

Time	T_1	T_2	$qtaP_x$
t_1	**bot**		100
t_2	$read(qtaP_x, A)$	**bot**	100
t_3	$A = A - 50$	$read(qtaP_x, A)$	100
t_4	$write(qtaP_x, A)$	$A = A - 30$	50
t_5	$write(qtaC_{xz_1}, 50)$	$write(qtaP_x, A)$	70
t_6	**commit**	$write(qtaC_{xz_2}, 30)$	70
t_7		**commit**	70

Esempio (2PL)

La transazione T_2 prima di acquisire il lock in scrittura sull'oggetto $qtaP_x$ dovrà attendere che la transazione T_1 (che temporalmente è la prima ad acquisirne il lock) abbia terminato l'aggiornamento del valore dell'oggetto stesso. T_2 quindi prima di effettuare la scrittura vedrà il valore corretto (50).

2PL ristretto

Una variante del 2PL in grado di eliminare anche l'anomalia delle letture sporche è il *2PL stretto* (o *Strict 2PL*) che aggiunge rispetto al 2PL classico la condizione aggiuntiva: *i lock possono essere rilasciati solo dopo il commit o l'abort*.

Problemi del 2PL:

- Nei metodi basati su lock potrebbero verificarsi situazioni di *deadlock* (stallo), in cui due (o più) transazioni sono "bloccate" nella loro esecuzione in quanto in attesa del rilascio di un lock, l'uno posseduto dall'altra.
- Per tale motivo i DBMS utilizzano delle tecniche di *prevenzione del deadlock* per riconoscere, in anticipo, situazione in cui potrebbe verificarsi un deadlock e negando la concessione di lock sugli oggetti.

Metodi basati su timestamp

I metodi basati su timestamp sono in grado di generare schedule serializzabili sfruttando da un lato le informazioni sugli istanti temporali in cui le transazioni nascono, e dall'altro, tenendo traccia dei tempi in cui le transazioni accedono agli oggetti per effettuare le operazioni di lettura/scrittura.

- Ad ogni transazione è associato un *timestamp* ovvero un identificatore che rappresenta l'istante di inizio della transazione stessa .
- Sotto l'ipotesi della commit-proiezione, uno schedule è accettato solo se riflette l'ordinamento seriale delle transazioni indotto dai timestamp ed in particolare:
  - una transazione non può leggere un oggetto scritto da una transazione più "giovane" (con un timestamp superiore);
  - una transazione non può scrivere un oggetto letto o scritto da una transazione più "giovane" (con un timestamp superiore).

TS Monoversione

- Ad ogni oggetto X della base sono associati due variabili:
  - $RTM(X)$ - che rappresenta il timestamp dell'ultima transazione che ha effettuato una lettura sull'oggetto X;
  - $WTM(X)$ - che rappresenta il timestamp dell'ultima transazione che ha effettuato una lettura sull'oggetto X;
- Lo scheduler riceve richieste di lettura $read(\tau, X)$ e scrittura $write(\tau, X)$, con indicato il timestamp τ della transazione:
- Nel caso di richiesta di lettura, questa viene respinta se $\tau < WTM(X)$ e la transazione viene uccisa; altrimenti, la richiesta viene accolta e $RTM(X) = max(\tau, RTM(X))$.
- Nel caso di richiesta di scrittura, questa viene respinta se $\tau < WTM(X)$ oppure se $\tau < RTM(X)$ e la transazione viene uccisa; altrimenti, la richiesta viene accolta e $WTM(X) = \tau$.

Approcci Ottimistici

- In alcuni sistemi di basi di dati i conflitti tra transazioni concorrenti che tentano di accedere agli stessi oggetti sono rari; pertanto, l'utilizzo di tecniche basate su lock o timestamp risulta non necessario.
- In questi ambienti, si preferiscono utilizzare le cosiddette *tecniche ottimistiche* in cui ogni transazione effettua "liberamente" le proprie operazioni sugli oggetti della base di dati secondo l'ordine temporale con cui le operazioni stesse sono generate.

Solo all'atto del commit, viene effettuato un controllo per stabilire se sono stati riscontrati eventuali conflitti, e nel caso, viene effettuato il rollback delle azioni delle transazioni e la relativa riesecuzione (*restarting*).

- La riesecuzione di transazioni comporta chiaramente un allungamento significativo dei relativi tempi di risposta e quindi risulta tollerabile solo se essa accade raramente.

Fasi di Approcci Ottimistici

In generale, un protocollo di controllo di concorrenza ottimistico è basato su 3 fasi:

- *fase di lettura*: ogni transazioni legge i valori degli oggetti della base di dati su cui deve operare e li memorizza in variabili (copie) locali dove sono effettuati eventuali aggiornamenti;
- *fase di validazione*: vengono effettuali dei controlli sulla serializzabilità degli schedule nel caso che gli aggiornamenti locali delle transazioni dovessero essere propagati sulla base di dati;
- *fase di scrittura*: gli aggiornamenti delle transazioni che hanno superato la fase di validazione sono propagati definitivamente sugli oggetti della base di dati.

Si vuole fare notare come la fase di rollback delle transazioni - che non superano la validazione e devono essere rieseguite - è meno onerosa in quanto riguarda variabili locali e non oggetti della base di dati.

Gestione della Concorrenza in SQL

Il DBMS permette al DBA di stabilire la politica per il controllo di concorrenza che si desidera adottare per il sistema di basi di dati. In SQL è possibile differenziare transazioni che effettuano solo letture da quelle che effettuano letture/scritture. E' possibile stabilire il cosiddetto *livello di isolamento* richiesto, tra i seguenti:

- READ UNCOMMITTED: La transazione accetta di leggere dati modificati da una transazione che non ha ancora fatto il commit (ignora i lock esclusivi e non acquisisce lock in lettura).
- READ COMMITTED: La transazione accetta di leggere dati modificati da una transazione solo se questa ha fatto il commit. Se però essa legge due volte lo stesso dato, può trovare dati diversi.
- REPEATABLE READ: La transazione accetta di leggere dati modificati da una transazione solo se questa ha fatto il commit. Inoltre se un dato è letto due volte, si avrà sempre lo stesso risultato.
- SERIALIZABLE: produce schedule serializzabili senza anomalie.

Controllo di Affidabilità

Controllo di Affidabilità

Il controllo di affidabilità ha come obiettivo quello di “ripristinare” il corretto stato (*recovery*) di un sistema di basi di dati a valle di un possibile malfunzionamento delle sue componenti hardware o software dovuto a guasti accidentali o intenzionali.

Esso deve garantire le proprietà di *atomicità* e *persistenza* delle transazioni che rappresentano l’unità base delle attività di recovery.

Memorie

- La memorizzazione dei dati in un DBMS coinvolge, come visto, tre differenti tipologie di memorie (in gergo *storage)*: *memoria centrale*, *memoria di massa* (es. dischi magnetici) e *memoria stabile* (es. nastri, repliche RAID di dischi, ecc.).
  - La memoria centrale rappresenta lo *storage primario* di un sistema di basi di dati con caratteristiche di elevata velocità per le operazioni di lettura/scrittura, capacità ridotta e volatilità delle informazioni,
  - La memoria di massa e la memoria stabile rappresentano lo *storage secondario* aventi caratteristiche di persistenza ed elevata capacità, e di contro, velocità più ridotte. Inoltre, la memoria stabile identifica in maniera astratta una particolare memoria che non può danneggiarsi.

Gestore dell'Affidabilità

Il *Gestore dell'Affidabilità* deve garantire che gli effetti di tutte le transazioni che hanno effettuato il commit siano memorizzate in maniera permanente, in modo tale che, a valle di guasti sia sempre possibile ripristinare il contenuto corretto di una base di dati.

- Il problema principale è dovuto al fatto che le operazioni di scrittura delle transazioni non sono azioni atomiche: gli effetti di una transazione che ha effettuato il commit in memoria primaria potrebbero essere non riportati subito su memoria secondaria e quindi essere resi persistenti.

Concetti base del recovery

Il gestore dell'afffidabilità deve gestire l'esecuzione dei comandi transazionali di begin transaction, commit, rollback (abort) e tutte le operazioni di ripristino dopo guasti.

Definizione (File di log)

Un log è un file presente su memoria stabile che registra le tutte operazioni svolte dalle transazioni nel loro ordine di esecuzione.

- Il log è quindi una sorta di "diario di bordo" che, in un qualsiasi istante, permette di ricostituire il contenuto corretto della base dei dati a seguito di malfunzionamenti.

Check Point e Dump

- CHECK POINT – è un'operazione che serve ad effettuare il "punto della situazione" (semplificando le successive operazioni di ripristino), registrando le transazioni attive in un certo istante e verificando che le altre o non siano iniziate o siano finite (viene effettuata una sorta di "sincronizzazione" tra il contenuto della base di dati ed il file di log).
- DUMP — corrisponde alla classica operazione di *backup* o copia (solitamente pianificata nel tempo e prodotta mentre il sistema non è operativo) del contenuto della base di dati su memoria stabile necessaria alla ricostruzione del suo contenuto soprattutto nel caso di danneggiamento della memoria secondaria non stabile.

In realtà, un’operazione di checkpoint è abbastanza complessa; nella sua forma più semplice, essa prevede le seguenti azioni:

1. si sospende l’accettazione di richieste di ogni tipo da parte delle transazioni;
2. si trasferiscono in memoria di massa (tramite le primitive force del buffer manager) tutte le pagine sporche relative a transazioni andate in commit;
3. si registra sul log in modo sincrono (con un force) un record di checkpoint contenente gli identificatori delle transazioni attive;
4. si riprende l’accettazione delle operazioni.

A valle di un checkpoint si è certi che, per tutte le transazioni che hanno effettuato il commit, i dati sono in memoria di massa e che siano state individuate correttamente le sole transazioni “in corso”.

Log di Sistema

Nel dettaglio, il log di un sistema di basi di dati contiene le seguenti informazioni:

- *Record di transazione*, contenenti a loro volta:
  - l'identificativo della transazione (T);
  - il timestamp delle varie azioni (t_s);
  - la particolare azione (O_p) svolta da una transazione: begin (B), update (U), delete (D), insert (I), commit (C), abort (A);
  - l'identificativo dell'oggetto della base di dati coinvolto (O);
  - il valore dell'oggetto prima della modifica (B_I) da parte di una transazione (in gergo *before-image*);
  - il valore dell'oggetto dopo della modifica (A_I) da parte di una transazione (in gergo *after-image*);
- *Record di sistema*
  - *Dump* (DP) contenente l'istante di tempo (t_s) in cui è stato effettuato l'ultimo backup della base di dati ed alcuni dettagli pratici (es. file e dispositivi coinvolti);
  - *Checkpoint* (CK) contenente l'insieme delle transazioni attive ($\mathcal{T}$) in un dato istante (t_s) sulla base di dati.

Esempio

Per semplicità, è possibile schematizzare i record delle transazioni nella forma $O_p(T, t_s, O, B_I, A_I)$ mentre i record di sistema nella forma $DP(t_s)$ e $CK(t_s, \mathcal{T})$.

- Se non si è interessati all'istante preciso in cui avvengono le varie operazioni è possibile omettere l'informazioni t_s (record consecutivi nel log indicano che le relative operazioni sono avvenute in istanti successivi).
- Utilizzando tali notazioni e semplificazioni, di seguito è riportato un esempio di file di log (con riferimento allo scenario dell'e-commerce).

DP, B(T_1,-,-,-,-), U(T_1,-,qtaP,100,90), U(T_1,-,qtaC,NULL,10), C(T_1,-,-,-,-), B(T_2,-,-,-,-), CK(T_2), U(T_2,-,qtaP,90,70), U(T_1,-,qtaC,NULL,20), C(T_2,-,-,-,-), B(T_3,-,-,-,-), U(T_3,-,qtaP,100,90), U(T_3,-,qtaC,NULL,10), C(T_3,-,-,-,-), ...

Ripristino

- L'esito di una transazione è determinato quando viene scritto il record di commit (in modo sincrono) nel log.
- Un guasto prima di tale istante potrebbe comportare un "disfacimento" di tutte le azioni di una transazione registrate all'interno del log (in gergo *undo*), qualora i dati siano già stati memorizzati sulla base di dati.
- Di contro, un guasto successivo al commit potrebbe comportare il "rifacimento" (in gergo *redo*) di tutte le azioni svolte da una transazione e registrate all'interno del log, qualora queste non siano state ancora registrate nella memoria secondaria della base di dati.

Regole di Scrittura sul log

Per garantire un corretto ripristino di un sistema di basi di dati, esistono delle precise regole per la scrittura sul log:

- *write ahead*: ogni transazione scrive sul log i relativi record prima di effettuare la medesima azione sulla base di dati (consente di disfare le azioni poiché per ogni aggiornamento viene reso disponibile nel log il valore prima della scrittura sul database);
- *commit precedenza*: ogni transazione scrive sul log tutti i relativi record prima di effettuare il commit (consente di rifare le azioni poiché se le pagine modificate non sono state ancora trascritte dal buffer manager viene reso disponibile nel log il valore in esse registrato).

Scrittura sulla Base di Dati

Esistono invece tre differenti modalità:

- *modalità immediata*: ogni scrittura sul log è "immediatamente" seguita da una scrittura sulla base di dati (tale modalità evita operazioni di redo e comporta solo operazioni di undo);
- *modalità differita*: ogni scrittura di transazioni sulla base di dati è "differita" alla successiva memorizzazione del record di commit nel file di log (tale modalità evita operazioni di undo e comporta solo operazioni di redo);
- *modalità mista*: la scrittura può avvenire in modalità sia immediata sia differita (richiede sia undo sia redo).

Le azioni di recovery quindi mireranno a ricostruire sulla base dell'analisi del file di log e partendo dall'ultimo dump utile della base di dati il relativo contenuto applicando azioni di undo o redo.

Tipi di Guasto

- Nel caso di *guasti soft* (es. errori di programma, crash di sistema, caduta di tensione, ecc.) si perde il contenuto della sola memoria centrale (mentre rimangono intatte la memoria secondaria e quella stabile). In tale situazioni, viene effettuata la cosiddetta *ripresa a caldo* (*warm restart*).
- Nel caso di *guasti hard* sui dispositivi di memoria di massa si perde la memoria centrale e quella secondaria ma non quella stabile.). In tale situazioni, viene effettuata la cosiddetta *ripresa a freddo* (*cold restart*).

In entrambi i casi, la procedura di ripristino avviene nelle seguenti fasi (modello *fail-stop*):

- si forza l'arresto completo delle transazioni attive sul sistema di basi di dati;
- viene ripristinato il corretto corretto funzionamento del sistema operativo;
- viene effettuata la procedura di ripristino.

Ripresa a Caldo - 1

1. viene ripercorso il log a ritroso finché non viene trovato l'ultimo record di checkpoint
2. sulla base delle transazioni attive vengono costruiti l'insieme *undo* delle transazioni da disfare
   - ovvero quelle che hanno effettuato l'abort prima del guasto o che non hanno effettuato in tempo il commit ma hanno scritto sulla base di dati

   e quello *redo* delle transazioni da rifare
   - ovvero quelle che hanno effettuato il commit prima del guasto

Ripresa a Caldo - 2

3. viene ripercorso il log “all’indietro”, fino alla “più vecchia” azione delle transazioni negli insiemi *undo* e *redo* disfacendo tutte le azioni delle transazioni presenti nell’insieme *undo*
   - nel caso di update si ripristina il valore B_I, nel caso di insert si effettua la cancellazione dell’oggetto inserito
   - nel caso di delete si rieffettua l’inserimento
4. viene ripercorso il log “in avanti”, rifacendo tutte le azioni delle transazioni appartenenti all’insieme *redo*
   - nel caso di update si ripristina il valore A_I, nel caso di insert si rieffettua l’inserimento dell’oggetto
   - nel caso di delete la cancellazione)

Supponendo di partire dal seguente log semplificato (non viene considerata l'informazione relativa al timestamp t_s delle operazioni, vengono astratti sia gli oggetti della base di dati sia i relativi valori B_I e A_I e per ogni tipo di operazione vengono considerati solo i corrispondenti parametri di I/O):

DP, B(T_1), B(T_2), B(T_3), I(T_1,O_1,A_1), D(T_2,O_2,B_2), B(T_4), U(T_4,O_3,B_3,A_3), U(T_1,O_4,B_4,A_4), C(T_2), CK(T_1,T_3,T_4), B(T_5), U(T_5, O_5, B_5,A_5), A(T_3), CK(T_1,T_4,T_5), B(T_6), A(T_4), U(T_6,O_6,B_6,A_6), U(T_6,O_3,B_7,A_7), C(T_6), crash

La ripresa a caldo avviene attraverso le seguenti azioni:
si ripercorre all'indietro il log fino all'ultimo record di checkpoint (evidenziato in grassetto):

DP, $B(T_1)$, $B(T_2)$, $B(T_3)$, $I(T_1,O_1,A_1)$, $D(T_2,O_2,B_2)$, $B(T_4)$, $U(T_4,O_3,B_3,A_3)$, $U(T_1,O_4,B_4,A_4)$, $C(T_2)$, $CK(T_1,T_3,T_4)$, $B(T_5)$, $U(T_5,O_5,B_5,A_5)$, $A(T_3)$, **$CK(T_1,T_4,T_5)$**, $B(T_6)$, $A(T_4)$, $U(T_6,O_6,B_6,A_6)$, $U(T_6,O_3,B_7,A_7)$, $C(T_6)$, crash

dopodiché sulla base della transazioni attive (T_1, T_4, T_5, T_6) vengono determinati gli insiemi *undo* e *redo*:

undo=$\{T_1, T_4, T_5\}$
redo= $\{T_6\}$

successivamente si ripercorre di nuovo il log all'indietro fino a determinare l'azione più vecchia (quella sottolineata) delle transazioni attive:

DP, $\underline{B(T_1)}$, $B(T_2)$, $B(T_3)$, $I(T_1,O_1,A_1)$, $D(T_2,O_2,B_2)$, $B(T_4)$, $U(T_4,O_3,B_3,A_3)$, $U(T_1,O_4,B_4,A_4)$, $C(T_2)$, $CK(T_1,T_3,T_4)$, $B(T_5)$, $U(T_5,O_5,B_5,A_5)$, $A(T_3)$, **$CK(T_1,T_4,T_5)$**, $B(T_6)$, $A(T_4)$, $U(T_6,O_6,B_6,A_6)$, $U(T_6,O_3,B_7,A_7)$, $C(T_6)$, crash

a questo punto si ripercorre il log dalla fine al punto sottolineato (la parte in grigio) disfacendo tutte le azioni delle transazioni dell'insieme *undo*:

DP, $B(T_1)$, $B(T_2)$, $B(T_3)$, $I(T_1,O_1,A_1)$, $D(T_2,O_2,B_2)$, $B(T_4)$, $U(T_4,O_3,B_3,A_3)$, $U(T_1,O_4,B_4,A_4)$, $C(T_2)$, $CK(T_1,T_3,T_4)$, $B(T_5)$, $U(T_5,O_5,B_5,A_5)$, $A(T_3)$, $CK(T_1,T_4,T_5)$, $B(T_6)$, $A(T_4)$, $U(T_6,O_6,B_6,A_6)$, $U(T_6,O_3,B_7,A_7)$, $C(T_6)$, crash

azioni di undo:

$O_5=B_5$
$O_4=B_4$
$O_3=B_3$
$D(O_1)$

a questo punto si ripercorre il log in avanti dal punto sottolineato alla fine rifacendo le azioni delle transazioni dell'insieme di *redo*:

azioni di redo:
$O_6 = A_6$
$O_3 = A_7$

Ripresa a freddo

La ripresa a freddo avviene a seguito di guasti che provocano danni alla base di dati (siano essi hard, soft); la tecnica più diffusa prevede l'esecuzione dei seguenti passi:

1. si ripristinano i dati a partire dal backup accedendo al più recente record di dump del log;
2. si eseguono tutte le operazioni registrate sul log relativamente alla parte deteriorata riportandosi all'istante precedente il guasto;
3. si esegue una ripresa a caldo.

DBMS Distribuiti

- passaggio da sistemi centralizzati ....
  - sistemi con un database locale unico controllato da un unico DBMS,
- .... ai *DBMS distribuiti*, o *Distributed Database Management System* (DDBMS)
  - che permettono l'accesso a dati memorizzati in più siti remoti.

Basi di Dati Distribuita e DBMS Distribuito

Un DB distribuito è un Collezione di dati condivisi e logicamente correlati che sono fisicamente distribuiti su una rete di calcolatori. Un DDBMS è il sistema software che gestisce una base di dati distribuita, rendendo la distribuzione completamente trasparente all'utente finale.

Frammenti

- Un base di dati distribuita consiste in un unico database (dal punto di vista logico) che è costituito da un insieme di *frammenti*
  - ogni frammento è a sua volta memorizzato in uno o più server in rete, sotto il controllo di un proprio DBMS.

  In un sistema siffatto, ogni server è in grado di elaborare in modo indipendente le richieste degli utenti che richiedono accesso a dati locali, ed è in grado di elaborare i dati che sono memorizzati in altri siti della rete.

Pro e Contro dei DBMS distribuiti

PRO

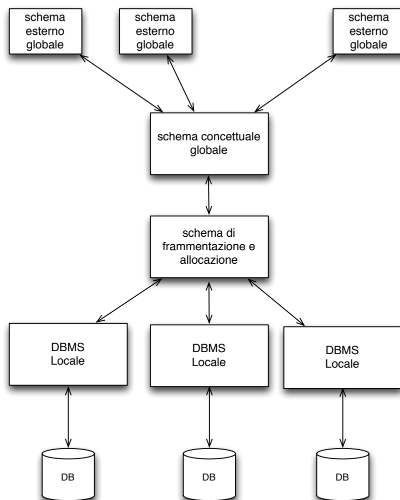
La distribuzione dei dati su più server con hardware meno costoso e performante, può favorire l'economicità di una soluzione, la scalabilità del sistema, e permette una modalità semplice di integrazione di sistemi differenti.
a soluzione distribuita, inoltre, aumenta l'affidabilità, la disponibilità dei dati e le prestazioni complessive.

CONTRO

Una soluzione distribuita aumenta la complessità del sistema hardware e software, anche a fronte di mancanza di standard di mercato.

Se su tutti i server è usato lo stesso DBMS si parla di database *omogenei*; ovviamente se i DBMS sono diversi, si parla di database *eterogenei*. Se ogni server mantiene completa autonomia, si parla anche di sistemi *multidatabase*.

Architettura di Riferimento



Architettura di Riferimento: Blocchi

- un insieme di schemi esterni globali;
- uno schema concettuale globale – ovvero la descrizione dell'intera base di dati, come se essa fosse non distribuita;
- uno schema di frammentazione e di allocazione – ovvero la descrizione di come i dati sono logicamente partizionati e di dove i dati sono allocati;
- un insieme di DBMS locali basati sull'architettura ANSI SPARC.

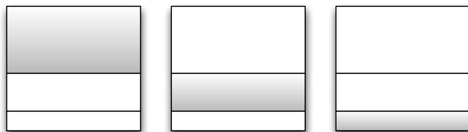
Frammentazione

- frammentazione orizzontale: consiste in un insieme di tuple (frammentazione della tabella)
- frammentazione verticale: consiste in un insieme di attributi (frammentazione dello schema)

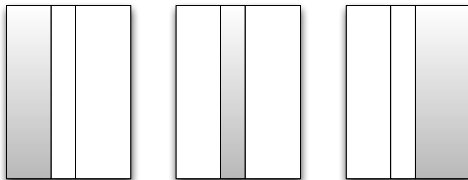
Una operazione di frammentazione di una relazione una relazione r in frammenti $r_1, r_2, \ldots r_m$ è *corretta* se e solo se valgono le seguenti proprietà

- *completezza* – ogni tupla che compare i r deve poter essere trovata in almeno in un frammento r_i;
- *ricostruibilità* – è possibile definire una operazione (combinando gli operatori dell'algebra relazionale) che dai frammenti r_i permetta di ricostruire la relazione r, in modo da mantenere le dipendenze funzionali di r stessa;
- *disgiunzione* – se un dato appare in un frammento, non deve essere presente in nessun altro frammento della frammentazione, per evitare ridondanza

Frammentazione - 2



frammentazione orizzontale



frammentazione verticale

Data una relazione r,

- un frammento orizzontale si può ottenere attraverso un operazione di selezione avente una condizione di selezione χ che coinvolge uno o più attributi dello schema di relazione R di r: $\sigma_{\chi}(r)$.
  - L'operazione di ricostruzione in questo caso si può effettuare attraverso una operazione di unione insiemistica: $r_1 \cup r_2 \cup \ldots r_m = r$.
- Una operazione di frammentazione verticale si può invece ottenere con una operazione di proiezione su un sottoinsieme di attributi A dello schema R di r: $\pi_A(r)$.
  - L'operazione di ricostruzione in questo caso si può effettuare attraverso una operazione di join: $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_m = r$.

Allocazione di Frammenti su Server

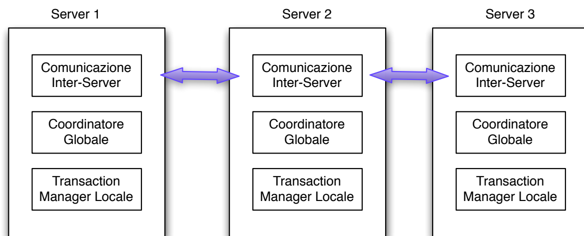
- Ogni frammento è realizzato tramite una o più tabella in un database allocato su un certo server.
- L'allocazione è
  - *non ridondante* se ogni frammento viene allocato su un solo server
  - è *ridondante* se alcuni frammenti sono allocati su più server.

Trasparenza

La trasparenza nasconde i dettagli implementativi. È possibile avere sia una trasparenza di frammentazione (quando l'utente non sa come i dati siano frammentati) che trasparenza di allocazione (quando l'utente non sa dove risiedono i dati).

Gestione delle transazioni nei DDBMS

- L'obiettivo: Devono essere verificate le proprietà *ACID* della transazione globale e di ogni singola transazione componente.
  - Necessità di un *Coordindatore Globale* su ogni server che ha il compito di coordinare l'esecuzione sia delle transazioni globali sia di quelle locali sul sito (gestite attraverso un *Transaction Manager Locale*).
  - Inoltre, le comunicazioni tra i diversi server devono avvenire attraverso una componenti di comunicazione (*Comunicazione Inter-Server*).



Verifica delle Proprietà ACID - 1

Consistenza e Atomicità

I vincoli sono locali: per cui i meccanismi utilizzati per il caso centralizzato restano validi anche nel caso globale. Lo stesso si può dire in grosse linee per l'atomicità (senza considerare la presenza di eventuali fallimenti del sistema).

Isolamento

Se ogni server locale usa il 2PL ristretto, la scheduling globale è serializzabile; se ogni sistema usa il metodo dei timestamp, ed essi sono assegnati in maniera globale alle sottotransazioni, lo scheduling globale è serializzabile. Si può quindi concludere che se lo schedule dell'esecuzione delle transazioni ad ogni server è serializzabile, allora lo è anche quello globale, inteso come l'unione di tutti gli schedule locali.

Verifica delle Proprietà ACID - 2

Persistenza

Particolare attenzione richiede invece la persistenza, in particolare per quanto attiene la gestione di fallimenti in un ambiente distribuito. Infatti, in questo ambiente di server connessi in rete, occorre tener presente il verificarsi di:

- perdita di messaggi,
- caduta di connessione,
- caduta di un server,
- eventuale partizionamento della rete di comunicazione.

Atomicità anche i presenza di fallimenti del sistema

- Per assicurare l'atomicità di una transazione globale, occorre che sia assicurato che ogni singola sottotransazione vada in abort o in commit, attraverso l'uso di opportuni protocolli di commit che mettano al riparo da eventuali fallimenti.
  - *commit a 2 Fasi*, o *Two Phase Commit* (2PC).

Ipotesi del 2PC

Nel seguito si assumerà che ogni transazione globale sia gestita da un server che agisce da *coordinatore* della transazione, che è generalmente il server che ha iniziato la transazione, e dagli altri server locali coinvolti, detti *partecipanti*. Il coordinatore conosce l'identità di tutti i partecipanti e ogni partecipante conosce il coordinatore.

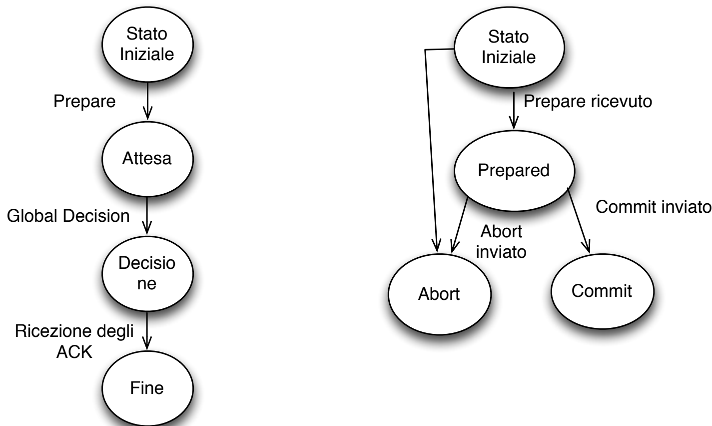
Il ”Rito” del Matrimonio

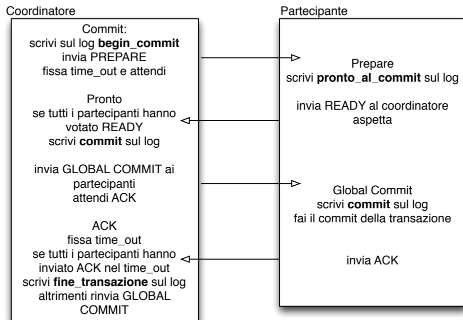
Il Rito

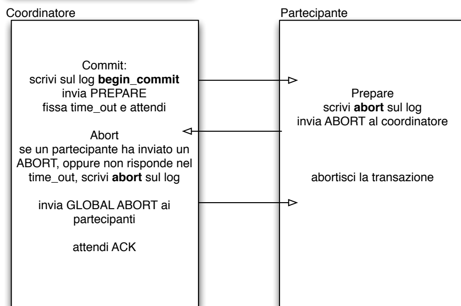
Il coordinatore chiede ai partecipanti se vogliono sposarsi (cioè se vogliono concludere positivamente la transazione): se tutti i partecipanti sono d'accordo, il matrimonio si fa. Se almeno uno dei partecipanti non è d'accordo il matrimonio si annulla.

- Il coordinatore chieda a tutti i partecipanti se sono preparati a fare il commit delle transazioni (*fase di voting*).
- Se esiste un partecipante che invece genera abort, oppure se fallisce nel rispondere entro un certo prestabilito intervallo di tempo, allora il coordinatore informa tutti i partecipanti che la transazione sarà abortita (fase di decisione), e tale decisione dovrà essere adottata da tutti i partecipanti.
- Se un partecipante ha votato per l'abort, può abortire la transazione in modo unilaterale; se invece ha votato per il commit, deve attendere la decisione globale (*global commit* o *global abort*) e ad essa attenersi.

Ovviamente il global commit avverrà da parte del coordinatore solo se tutti i partecipanti hanno votato in tempo per il commit. Il coordinatore ed ogni partecipante dovranno avere i propri log locali, per poter effettuare eventuale rollback o commit. NB – Problemi: *deadlock* e caduta del coordinatore.

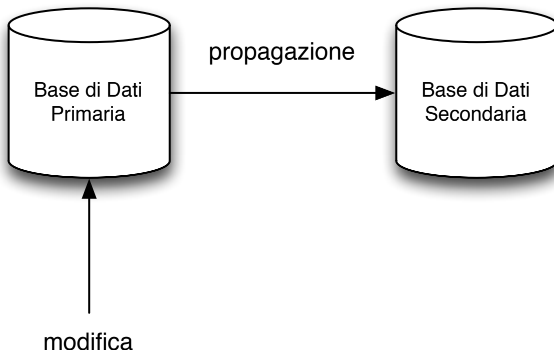



Coordinatore
Partecipante
Commit:
scrivi sul log begin_commit
invia PREPARE
fissa time_out e attendi
Prepare
scrivi pronto_al_commit sul log
invia READY al coordinatore
aspetta
Pronto
se tutti i partecipanti hanno
votato READY
scrivi commit sul log
invia GLOBAL COMMIT ai
partecipanti
attendi ACK
Global Commit
scrivi commit sul log
fai il commit della transazione
ACK
fissa time_out
se tutti i partecipanti hanno
inviato ACK nel time_out
scrivi fine_transazione sul log
altrimenti rinvia GLOBAL
COMMIT
invia ACK


Coordinatore
Partecipante
Commit:
scrivi sul log begin_commit
invia PREPARE
fissa time_out e attendi
Prepare
scrivi abort sul log
invia ABORT al coordinatore
Abort
se un partecipante ha inviato un
ABORT, oppure non risponde nel
time_out, scrivi abort sul log
abortisci la transazione
invia GLOBAL ABORT ai
partecipanti
attendi ACK

Replicazione di basi di dati

Un modello di replicazione di basi di dati (modello *asimmetrico*), consiste nel copiare e mantenere le informazioni gestite da una base di dati in più server distribuiti.La replicazione richiede che una qualsiasi modifica fatta su un oggetto della base di dati su di un server, sia applicati anche a tutte le altre copie fisicamente memorizzate su altri siti.



Master e Slave

- un server *master* controlla le replicazioni in modo che tutte le modifiche su di esso fatte si propaghino a tutti gli altri siti detti *slaves*.
- Gli slaves mantengono le copie del master che gli competono, e vengono periodicamente modificati per mantenersi in linea con le informazioni presenti sul master.

La sincronizzazione tra master e slave può avvenire sostanzialmente seguendo due modalità:

- *sincrona* – prevede una modifica immediata degli oggetti modificati nel master anche in tutti gli slave correlati, facendo uso di opportuni protocolli come il 2PC;
- *asincrona* – la modifica degli slave a seguito di modifica del master avviene solo in certi intervalli di tempo (entro qualche minuto, varie ore o anche giorni).

Vantaggi

- Viene garantita l'*affidabilità* del sistema: avere a disposizione copie multiple dello stesso dato permette in caso di guasti di uno (o anche più siti) di effettuare il recovery a caldo in modo efficace ed efficiente.
- Invece che incidere su un unico server centralizzato, inoltre, si può prevedere l'uso di più server remoti: per questo un altro effetto benefico è il miglioramento complessivo delle *prestazioni* e un *bilanciamento del carico del sistema*, permettendo nel contempo un numero sempre maggiore di utenti.