Appunti Basi di Dati

Luca Seggiani

23 Maggio 2024

1 Tecnologia della base di dati

Finora abbiamo dato una descrizione della base di dati (o meglio, del DBMS) in quanto astrazione, cioè nei limiti delle funzionalità che offriva all'utente. Vediamo ora alcuni dettagli del funzionamento interno e dell'implementazione effettiva dei DBMS, per poterne fare un'uso più adeguato ed avere accesso ad alcuni servizi particolari. Avevamo specificato che un DBMS è un software in grado di gestire collezioni di dati che siano

- **Grandi**: molto maggiori della memoria centrale dei sistemi di calcolo utilizzati;
- **Persistenti**: con un periodo di vità maggiore delle esecuzioni dei programmi che le utilizzano;
- **Condivise**: usate in contemporanea da più applicazioni diverse.

Tutto questo mantenendosi:

- **Affidabili**: in grado di resistere a malfunzionamenti;
- **Sicuri**: in grado di assicurare la privacy degli utenti e la sicurezza dei dati attraverso una politica sugli accessi;
- **Efficienti**: in grado di non sprecare risorse di sistema e tempo;
- **Efficaci**: utili nel rendere produttive le attività degli utenti.

Vediamo più nel dettaglio alcune di queste caratteristiche.

Grandezza e persistenza

Il fatto che le moli di dati gestite dai DBMS sono grandi e persistenti richiede l'utilizzo di una memoria secondaria, in quanto sarebbe impossibile

mantenere l'intero database in memoria centrale. Questo implica lo sviluppo di una certa rappresentazione fisica più efficiente per i dati immagazzinati in memoria secondaria, che differisce dal modello logico che è disponibile agli utenti. Inoltre, la memoria secondaria sarà più lenta della memoria centrale, ergo bisogna sviluppare interazioni fra memoria principale e secondaria che limiti il più possibile gli accessi alla secondaria, in modo da ottimizzare i tempi e l'uso delle risorse.

Condivisione

L'accesso condiviso ai dati delle basi di dati (che sono cosiddette **risorse integrate**) consiste in:

- **Attività diverse** su dati condivisi in parte (ergo meccanismi di autorizzazione);
- **Attività multi-utente** su dati condivisi (ergo controllo della concorrenza).

Questo tipo di situazioni viene risolto dal meccanismo delle **transazioni**, che verrà approfondito più avanti. Intuitivamente, le transazioni sono corrette quando sono seriali, ovvero eseguite in sequenza. Di contro, imporre questa sequenzialità renderebbe la base di dati inefficiente. Per questo si usano meccanismi di **controllo della concorrenza**, che raggiungono un buon compromesso.

Affidabilità

L'affidabilità di una base di dati consiste nella sua capacità di resistere e conservare informazioni anche in presenza di malfunzionamenti. Questo diventa difficile nel caso le transazioni (o semplicemente gli aggiornamenti della base di dati) siano frequenti, e richiedano sistemi di gestione anche piuttosto complessi della memoria. Nelle basi di dati, l'affidabilità viene assicurata attraverso la cosiddetta **acidità** (*acid compliance*), dall'acronimo Atomiche Consistenti Isolate Definitive. Le transazioni acide devono essere:

- **Atomiche**: una transazione viene eseguita o meno, senza vie di mezzo;
- **Consistenti**: una transazione non deve violare i vincoli di integrità già stabiliti sulla base di dati;
- **Isolate**: una transazione deve essere poter eseguita assieme ad altre come da sola, e dare lo stesso risultato;
- **Definitive**: una volta eseguita, una transazione non deve essere dimenticata.

2 Gestione delle transazioni

Vediamo quindi la parte del DBMS che gestisce le transazioni, ergo la concorrenza e l'affidabilità delle operazioni di modifica operate sul database. Un criterio per classificare le basi di dati è il numero di utenti che ne possono usufruire simultaneamente. Si ha che una DBMS può essere:

- **Monoutente**: se può essere usata da un solo utente per volta, senza rischi di concorrenza;
- **Multiutente**: se può essere usata da più utenti per volta, quindi con il rischio di accessi concorrenti. La stragrande maggioranza dei DBMS è di questo tipo.

Multitasking

L'accesso di più utenti alla base di dati è permesso dal concetto del multitasking. Il multitasking consente al calcolatore (anzi uno dei calcolatori che ospita la base di dati) di eseguire più programmi (o meglio processi) contemporaneamente. Se esiste una sola CPU, in verità, si può eseguire effettivamente un solo processo per volta, ma i sistemi operativi sono forniti di meccanismi che sospendono e riprendono i processi in corso per assicurare l'esecuzione contemporanea. Si parla in questo caso di esecuzione **alternata** (*interleaved*). Su processori multithreaded (ovvero a più CPU) si può invece vere l'**esecuzione parallela** dei processi.

Transazioni

Una transazione identifica un'unità elementare di lavoro svolta da un'applicazione, a cui si vogliono associare caratteristiche acide. Sostanzialmente, una transazione è una successione di operazioni da eseguire successivamente che forma un'unità logica da eseguire sui dati. Una transazione comprende una o più operazioni di accesso alla base di dati, che possono essere:

- Inserimenti;
- Cancellazioni;
- Modifiche;
- Interrogazioni.

Che si traducano effettivamente in una sequenza di letture e scritture. Un sistema che mette a disposizione un meccanismo per la definizione ed esecuzione, sopratutto concorrente, di transazioni è detto **sistema transazionale**, o OLTP (*online transaction processing*).

Una transazione, più formalmente è una parte di programma caratterizzata da un inizio e una fine al cui interno deve essere eseguito uno e uno solo fra i comandi:

- **Commit**: applica le modifiche e prosegue;
- **Rollback/Abort**: annulla la transazione e riporta la base di dati allo stato precedente.

Stato di una transazione

Lo stato di una transazione è importante alla sua corretta esecuzione. I possibili stati in cui può trovarsi una transazione sono:

- **Attivo**: la transazione è stata appena avviata, e resterà in questo stato fino alla fine della sua esecuzione;
- **Commit parziale**: la transazione è terminata e le modifiche alla base di dati sono pronte ad essere applicate;
- **Commit**: la transazione è stata eseguita con successo e tutti i suoi aggiornamenti alla base di dati sono permanenti;
- **Fallito**: la transazione non può fare commit, per un errore interno autorilevato (suicidio), o è stata interrotta per fallimento mentre si trovava nello stato attivo (omicidio).
- **Abortito**: la transazione è stata interrotta e tutti gli aggiornamenti alla base di dati sono stati annullati.

Di norma, una transazione parte nello stato attivo, e si va a trovare poi, a seconda dell'esecuzione, in uno stato di commit parziale o in uno stato di fallimento. Non è detto che un commit parziale non possa all'ultimo essere rilevato come fallimento. In ogni caso, se il commit parziale va a buon fine, ci si ritrova nello stato di commit, mentre se si verifica un fallimento, ci si ritrova nello stato abortito.

Proprietà delle transazioni

Vediamo nel dettaglio le proprietà acide delle transazioni:

- **Atomicità**
  Una transazione è un'unità atomica di elaborazione, che non può lasciare la base in stati intermedi, ma solamente in uno stato pre-transazione o post-transazione. Un guasto prima del commit non deve avere conseguenze della base di dati, e deve causare un annullamento (*undo*) delle operazioni. Allo stesso tempo, nemmeno un guasto dopo il commit deve

avere conseguenze, ed eventualmmente si deve poter ripetere la transazione (*redo*). Nella maggior parte dei casi, l'esito di una transazione deve essere il commit con successo. Nella minoranza dei casi di errore, si deve poter effettuare un rollback (o abort), che può essere richiesto sia dall'applicazione, per il rilevamento di un'errore interno (suicidio), o dal sistema, per la violazione delle regole della base di dati (omicidio).

- **Consistenza**
  Una transazione deve rispettare i vincoli di integrità. Applicata ad una base di dati corretta, una transazione deve lasciare la base di dati in uno stato corretto. In altre parole, all'inizio e alla fine di una transazione il sistema è lasciato in uno stato consistente. Può accadere che durante la transazione la base di dati si ritrovi *temporaneamente* in uno stato incosistente.

- **Isolamento**
  La transazione non deve risentire degli effetti di altre transazioni concorrenti, e l'esecuzione concorrente delle transazioni non deve dare risultati diversi dalla loro esecuzione sequenziale. Questo significa che una transazione non espone i suoi stati intermedi, e si evita il cosiddetto "effetto domino", dove gli effeti di una transazione si ripercuotono su altre transazioni concorrenti.

- **Durabilità** (o persistenza)
  Gli effetti di una transazione dopo il commit non vanno perduti (perdurano per sempre), e la transazione stessa deve poter essere ripetuta. Questo significa che eventuali guasti dopo il commit devono poter essere ripristinati e non devono avere effetti permanenti sulla base di dati.

Controllo della concorrenza

La concorrenza è la carattersitica fondamentale delle transazioni. Visto che la mole di transazioni da eseguire in media su una base di dati è molto grande, non si può optare per un sistema seriale. Si rende quindi necessario un sistema di gestione della concorrenza. Il *modello di riferimento* sarà formato da operazioni di input-output su oggetti astratti $x, y, z...$, dove il problema è quello di eliminare le anomalie causate dall'esecuzione concorrente. Possiamo classificare queste anomalie:

- **Perdita di aggiornamento**
  Poniamo di avere due transazioni identiche:
  - $T_1 : r(x), x \to x + 1, w(x)$

- $T_2 : r(x), x \to x + 1, w(x)$

Ergo, leggi x, incrementalo di 1, e scrivi. Partendo da $x = 2$. un'esecuzione seriale di queste due transazioni dovrebbe giustamente portare $x = 4$. Vediamo cosa potrebbe accadere nel caso di esecuzione concorrente:

```
T_1     |  T_2
--------|--------
bot     |
r(x)    |
x++     |
        | bot
        | r(x)
        | x++
w(x)    |
commit  |
        |  w(x)
        | commit
```

In questo caso, la lettura di T_2 avverrebbe dopo l'incremento di T_1, ma prima del suo commit: tutto ciò che è stato fatto da T_1 verrebbe quindi perso, e il risultato (scorretto) sarebbe $x = 3$.

- **Lettura sporca**
  Abbiamo le transazioni:

  - $T_1 : r(x), x \to x + 1, w(x)$
  - $T_2 : r(x)$

  Consideriamo l'esecuzione:

```
T_1     |  T_2
--------|--------
bot     |
r(x)    |
x++     |
w(x)    |
        | bot
        | r(x)
abort   |
        | commit
```

In questo caso, T_2 ha letto lo stato di x mentre era in uso da T_1, ergo violando l'isolamento e costringendo T_1 ad abortire.

- **Letture inconsistenti**
  Abbiamo le transazioni:

- $T_1 : r(x), r(x)$
- $T_2 : r(x), x \to x + 1, w(x)$

Consideriamo l'esecuzione:

```
T_1     |  T_2
--------|--------
bot     |
r(x)    |
        | bot
        | r(x)
        | x++
        | w(x)
        | commit
r(x)    |
commit  |
```

In questo caso, durante la stessa esecuzione, T_1 avrebbe letto due valori diversi di x, cosa impossibile!

- **Aggiornamenti fantasma**
  Abbiamo le transazioni:

  - $T_1 : r(y), r(z), s \to y + z$
  - $T_2 : y \to y - 100, r(z), z \to z + 100, w(y), w(z)$

  Assumiamo valga il vincolo $y + z = 1000$:

```
T_1          |  T_2
-------------|--------------
bot          |
r(y)         |
             | bot
             | y <- y - 100
             | r(z)
             | z <- z + 100
             | w(y)
             | w(z)
             | commit
r(z)         |
s <- y + z   |
commit       |
```

In questo caso alla fine avremo $s = 1100$, il vincolo non è soddisfatto per T_1, in quanto vede un aggiornamento non coerente (lo sarebbe diventato un'istante dopo).

- **Inserimento fantasma**
  Poniamo di avere due transazioni su database, che comporta un'operazione di media:

```
T_1                       |  T_2
--------------------------|----------------------
bot                       |
r(stipendi_impiegati)     |
avg(stipendi_impiegati)   |
                          | bot
                          | inserisci(impiegato)
                          | commit
                          |
                          |
r(stipendi_impiegati)     |
avg(stipendi_impiegati)   |
commit                    |
```

  In questo caso, nel corso della stessa esecuzione, T_1 ha calcolato due medie diverse a causa di un inserimento imprevisto da parte di T_2.

Riassumendo, abbiamo quindi le anomalie:

- **Perdita di aggiornamento** (W - W);
- **Lettura sporca** (R - W o W - W con abort);
- **Letture inconsistenti** (R - W);
- **Aggiornamento fantasma** (R - W);
- **Inserimento fantasma** (R - W nuovo dato).

Gestore della concorrenza

La risoluzione di queste concorrenze è operata da un gestore della concorrenza. Il gestore della concorrenza entra in gioco dopo altri due componenti, il **gestore dei metodi d'accesso**, che si occupa dell'autorizzazione a letture e scritture (read e write), e il **gestore delle transazioni**, che gestisce begin, commit e abort. Il gestore della concorrenza si occupa sempre di operazioni read / write, ma nel contesto della loro esecuzione concorrente Per fare ciò ha l'ausilio della **tabella dei lock**.

Schedule

Una schedule (*"tabella di marcia"*) è una sequenza di operazioni lettura / scrittura di transazioni concorrenti, che compaiono nel loro ordine cronologico di esecuzione. Ad esempio, possiamo avere:

$$S : r_1(x) \quad r_2(z) \quad w_1(x) \quad w_2(z)$$

dove $r_1(x)$ e $w_1(x)$ sono letture e scrittura della transazione T_1 sull'oggetto x, e $r_2(z)$ e $w(z)$ sono letture e scritture della transazione T_2 sull'oggetto z.

Controllo di concorrenza

Il controllo di concorrenza, ovvero l'eliminazione di anomalie, è effettuato dallo **scheduler**, che tiene traccia di tutte le operazioni eseguite sulla base di dati e decide se accettare o rifiutare le transazioni che vengono via via richieste. Si assume, temporaneamente, che l'esito delle transazioni sia noto a priori (la cosiddettta ipotesi **commit-proiezione**), ergo lo stato finale di commit o abort sia noto prima dell'esecuzione della transazione. In questo modo possiamo automaticamente eliminare dallo schedule tutte le transazioni abortite. Si noti che questa assunzione non permette di trattare alcuni tipi di anomalie, tra cui quelle di lettura sporca.

Schedule seriali

Uno schedule di un'insieme di transazioni $T = T_1, ..., T_n$ è detto seriale se, per ogni coppia di transazioni $T_i, T_j \in T$, tutte le operazioni di T_i sono eseguite prima di qualsiasi operazione di T_j o viceversa. Una schedule si dice **serializzabile** se la sua esecuzione produce lo stesso risultato di uno schedule seriale sulle stesse transazioni. Si richiede quindi una nozione di equivalenza fra schedule. Abbiamo, intuitivamente, che nell'insieme di tutte le schedule S, l'insieme delle schedule serializzabili $S_s \in S$ è sottoinsieme di S, e l'insieme delle schedule seriali $S'_s \in S_s$ è a sua volta sottoinsieme di S_s.

View-serializzabilità

Diciamo che esiste una relazione **legge-da** tra le operazioni $r_i(x)$ e $w_j(x)$ in uno schedule S se $w_j(x)$ precede $r_i(x)$ in S e non c'è nessun $w_k(x)$ con $k \neq j$ tra $r_i(x)$ e $w_j(x)$ in S. A questo punto la scrittura $w_j(x)$ in S è detta **scrittura finale** su x se è l'ultima scrittura su x presente in S. Due schedule S_i e S_j sono dette **view-equivalenti** (in simboli $S_i i \approx_v S_j$) se hanno la stessa relazione legge-da e le stesse scritture su ogni oggetto. Uno schedule S è poi **view-serializzabile** se è view-equivalente a un qualche schedule seriale. L'insieme degli schedule view-serializzabili si indica con VSR.
Possiamo riscrivere alcune delle anomalie incontrate attraverso il linguaggio delle schedule:

- **Perdita di aggiornamento** : $S = r_1(x)\, r_2(x)\, w_1(x)\, w_2(x)$
- **Lettura inconsistente** : $S = r_1(x)\, r_2(x)\, w_2(x)\, r_1(x)$
- **Aggiornamento fantasma** : $S = r_1(x)\, r_1(y)\, r_2(z)\, r_2(y)\, w_2(y)\, w_2(z)\, r_1(z)$

Ovviamente, le ultime tre schedule riportate non sono view-equivalenti a nessuno schedule seriale. Sono anomalie! Facciamo le considerazioni del caso sulla

complessità view-serializzabilità. La verifica della view-equivalenza di due dati schedule ha complessità polinomiale. Ergo decidere la view-serializzabilità di uno schedule è un problema NP-completo: occorre confrontare lo schedule con tutti i possibili schedule seriali. Questo non è concepibile nella pratica. La soluzione è quella di definire una condizione di equivalenza più ristretta, che non copra tutti i casi ma che sia utilizzabile nella pratica, cioò abbia complessità inferiore.

Conflict-serializzabilità

Un'operazione a_i è in **conflitto** con un'altra operazione a_j, con $i \neq j$, se operano sullo stesso oggetto e almeno una di esse è una scrittura. N.B.: in un conflitto conta l'ordine delle operazioni. Possiamo dividere i conflitti in due casi:

- **conflitto read-write** (R - W o W - R);
- **conflitto write-write** (W - W).

Due schedule sono dette **conflict-equivalenti**, $S_i \approx_c S_j$ se hanno le stesse operazioni e ogni coppia di operazioni in conflitto compare nello stesso ordine in entrambi. Uno schedule S è **conflict-serializzabile** se è conflict-equivalente a un qualche schedule seriale. L'insieme degli schedule conflict-serializzabili è indicato con CSR.

VSR e CSR

Si può dimostrare che ogni schedule conflict-serializzabile è view-serializzabile, ma non necessariamente viceversa. Ad esempio, $r_1(x)w_2(x)w_1(x)w_3(x)$ è view-serializzabile, in quanto view-equivalente a $r_1(x)w_1(x)w_2(x)w_3(x)$, ma non è conflict-serializzabile. Dimostrare questo teorema significa dimostrare che la conflict-equivalenza ($\approx_v$) implica la view-equivalenza ($\approx_c$). Supponiamo quindi $S_1 \approx_c S_2$ I due schedule hanno:

- **Stesse scritture finali**: se così non fosse, ci sarebbero almeno due scritture in ordine diverso, e poichè scritture in ordine diverso sono in conflitto le schedule non sarebbero $\approx_c$.
- **Stesse relazioni "legge-da"**: se così non fosse, ci sarebbero scritture in ordine diverso p coppie lettura-scritture in ordine diverso e quindi, come sopra, sarebbe violata la $\approx_c$.

Abbiamo quindi introdotto due nuovi insiemi: nell'insieme di tutte le schedule S, $VSR \in S$ è l'insieme delle schedule view-serializzabili ed è sottoinsieme di S, $CSR \in VSR$ è l'insieme delle schedule conflict-serializzabili ed è sottoinsieme di VSR. Le schedule seriali $S'_s \in VSR$ sono sottoinsieme di VSR

e CSR. Sia VSR che CSR sono a loro volta sottoinsieme delle schedule serializzabili S_s.

Verifica della conflict-serializzabilità
Vediamo quindi il perché dell'introduzione della conflict-serializzabilità: possiamo verificarla attraverso la costruzione di un **grafo dei conflitti**, contenente un nodo per ogni transazione T_i e un'arco orientato da T_i a T_j se c'é almeno un conflitto fra un'azione a_i e un'azione a_j tale che a_i precede a_j Uno schedule è CSR se e solo se il grafo è aciclico. In verità, anche la conflict-serializzabilità, sebbene sia verificabile in modo molto più rapido (con opportune strutture dati in tempo lineare), è inutilizzabile in pratica. Questo perché la tecnica richiede di conoscere il grafo dall'inizio, ma uno scheduler deve lavorare incrementalmente, ovvero rispondere in modo dinamico ad ogni richiesta in arrivo. Inoltre, la tecnica si basa sull'assunzione della commit-proiezione. Nella pratica, si usano tecniche che garantiscono la conflict-serializzabilità senza dover costruire grafi a priori, e senza assumere l'ipotesi della commit-proiezione.

Lock
Una tecnica è quella di precedere ogni lettura e scrittura da un lock, e di effettuare l'unlock solo dopo che vengono ultimate. Il lock manager effettua quetste operazioni e decide se accogliere o rifiutare ulteriori richieste sulla base dei lock già applicati. Per aumentare la concorrenza si possono adoperare lock di tipo diverso:

- **Lock condiviso**: un lock condiviso permette a più transazioni di accedere alla risorsa, senza che nessun'altra transazione possa scrivere sulla risora. Tutte le letture sono precedute da un r_lock (lock condiviso) e seguite dall'unlock;

- **Lock esclusivo**: un lock esclusivo assicura l'accesso alla risorsa a una sola transazione, che potrà quindi scrivere senza impattare altre transazioni. Tutte le scritture sono precedute da un w_lock (lock esclusivo) e seguite dall'unlock.

Una transazione che legge e poi scrive un'oggetto può richiedere subito un lock esclusivo, oppure può chiedere un lock condiviso e poi un lock esclusivo (*lock escalation* o *lock promotion*). Il lock manager riceve queste richieste dalle transazioni e le accoglie o le rifiuta, sulla base della tavola dei conflitti. Quando si concede un lock su una risorsa ad una transazione, si dice che la risorsa è **acquisita** dalla transazione. Quando si effettua l'unlock, si dice che la risorsa viene **rilasciata**. Un lock condiviso impedisce la creazione di lock esclusivi (almeno che il lock condiviso sia unico). Di contro, un lock

esclusivo impedisce la creazione di altri lock esclusivi o inclusivi. Un lock condiviso viene rilasciato quando il contatore dei lock scende a 0. Nel caso una transazione richieda una risorsa che non sia concessa, essa è messa in attesa (eventualmente in coda) finché la risorsa non diventa disponibile.

Lock a due fasi ($2PL$)
Un'algoritmo di scheduling usato dalla maggior parte dei sistemi commerciali è basato su due regole:

- Se una transazione vuole leggere (scrivere) un dato, prima deve acquisire un lock condiviso (esclusivo) su quel dato. Se la transazione entra in conflitto su un lock, la si pone in attesa.
- Una transazione, dopo aver rilasciato un lock, non può acquisirne un'altro.

Il meccanismo della 2PL significa che tutti i lock vanno ottenuti sequenzialmente, e rilasciati sequenzialmente. Non è assolutamente permesso rilasciare un lock per ottenerne un'altro.