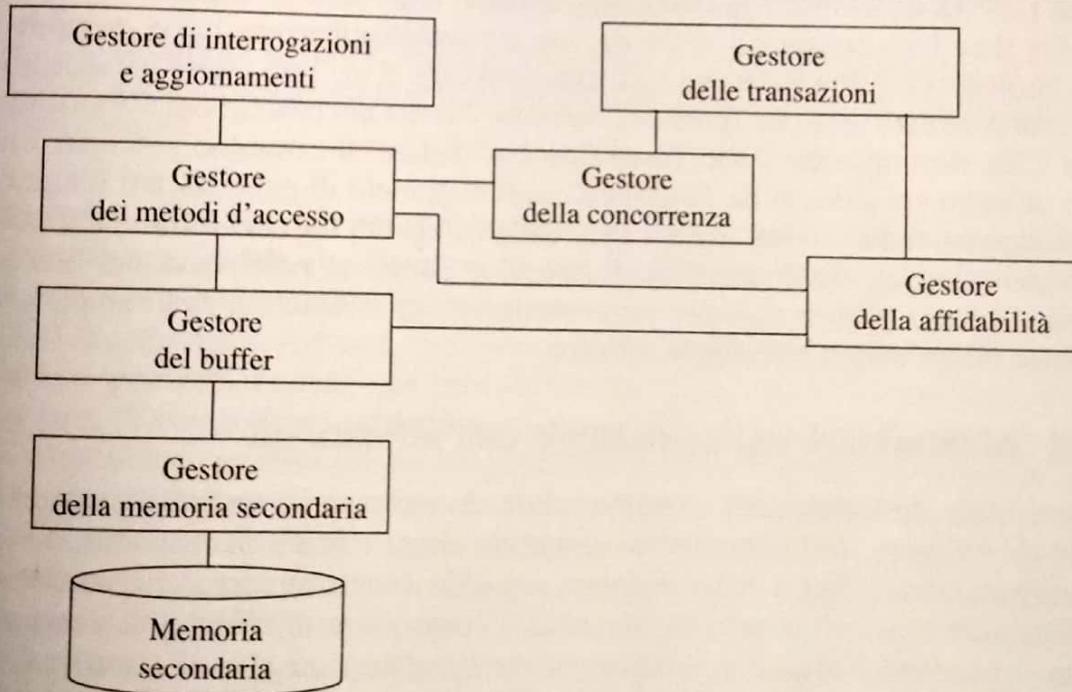


# 12

## Gestione delle transazioni

In questo capitolo continuiamo la discussione degli aspetti tecnologici che caratterizzano il funzionamento dei DBMS, illustrando le modalità secondo cui essi garantiscono la possibilità di conservare a lungo termine e in modo corretto il contenuto delle basi di dati, anche in presenza di guasti e favorendo anche gli indispensabili accessi concorrenti. Queste funzionalità sono particolarmente delicate in presenza di aggiornamenti delle basi di dati, cioè quando le operazioni non sono solo *interrogazioni*, ma anche *transazioni*.

La nozione di transazione è stata introdotta nel Paragrafo 5.6, con la definizione delle proprietà acide (atomicità, consistenza, isolamento e durabilità o persistenza), e utilizzata poi nel Capitolo 10 e in particolare nei Paragrafi 10.3 e 10.5.2. La Figura 12.1 mostra, in modo schematico, come l'architettura di un DBMS già discussa nel capitolo precedente (Figura 11.1) con riferimento alle sole interrogazioni, possa essere estesa per tenere conto della necessità di garantire le proprietà acide delle transazioni. Il modulo indicato nella figura come *gestore delle transazioni* coordina tutte le attività connesse appunto con le transazioni, attraverso l'esecuzione delle istruzioni *start transaction*, *commit* e *rollback*. Il modulo di gestione della



**Figura 12.1** Componenti di un DBMS coinvolti nella gestione di interrogazioni e transazioni.

affidabilità ha l'obiettivo di garantire atomicità e persistenza e, allo scopo, interagisce sia con il gestore dei metodi d'accesso (per tenere traccia delle operazioni richieste) sia con il gestore del buffer (per richiedere, quando necessario, scritture fisiche sui dischi, al fine di evitare che informazioni delicate rimangano solo nei buffer, che sono volatili). Il gestore della concorrenza si occupa invece di garantire l'isolamento, e lo fa "filtrando," ed eventualmente ripianificando, le operazioni di accesso richieste dal gestore degli accessi. Notiamo che invece la consistenza è gestita dai compilatori del DDL, che introducono opportuni controlli sui dati e opportune procedure per la verifica, eseguite poi dalle transazioni.

I due paragrafi di questo capitolo sono quindi rispettivamente dedicati alla presentazione delle tecniche per il controllo dell'*affidabilità* e di quelle per il controllo della *concorrenza*.

## 12.1 Controllo di affidabilità

Il controllo dell'affidabilità garantisce due proprietà fondamentali delle transazioni: l'atomicità e la persistenza. In pratica, esso garantisce che le transazioni non vengano lasciate incomplete, con alcune operazioni eseguite e le altre no, e che gli effetti di ciascuna transazione conclusa con un commit siano mantenuti in modo permanente. Il controllore dell'affidabilità svolge il proprio compito attraverso il *log*, un archivio persistente su cui registra le varie azioni svolte dal DBMS. Come vedremo, ogni azione di scrittura sulla base di dati viene protetta tramite una azione sul log, in modo che sia possibile "disfare" (*undo*) le azioni a seguito di malfunzionamenti o guasti precedenti il commit, oppure "rifare" (*redo*) queste azioni qualora la loro buona riuscita sia incerta e le transazioni abbiano effettuato un commit.

Per dare l'intuizione sul ruolo del log è possibile ricorrere a due metafore, una mitologica e l'altra basata su una fiaba popolare. Il log può essere paragonato al "filo di Arianna", usato da Teseo per ritrovare l'uscita del palazzo del Minotauro; in tal caso, riavvolgendo il filo Teseo riesce a "disfare" il cammino percorso. Un ruolo analogo era affidato da Hansel e Gretel ai granelli di pane lasciati lungo il loro cammino nella foresta, ma nel caso della favola dei fratelli Grimm i granelli venivano mangiati dagli uccellini, e Hansel e Gretel si perdevano nel bosco. Questa analogia mostra che, per poter svolgere con efficacia il suo compito, il log deve essere sufficientemente robusto.

### 12.1.1 Architettura del controllore dell'affidabilità

Il controllore dell'affidabilità è responsabile di realizzare i comandi transazionali e di realizzare le operazioni di ripristino dopo i malfunzionamenti, dette rispettivamente *ripresa a caldo* e *ripresa a freddo*. L'architettura del controllore di affidabilità è descritta nella Figura 12.2. Il controllore di affidabilità riceve richieste di accessi a pagine in lettura e scrittura, che passa al buffer manager, e genera altre richieste di letture e scritture di pagine necessarie a garantire la robustezza e la resistenza ai guasti. Infine, il controllore dell'affidabilità predisponde i dati necessari per eseguire i meccanismi di ripristino dai guasti, in particolare realizzando azioni di *checkpoint* e di *dump*.

Per semplicità, in questo paragrafo (e, con qualche differenza, nel prossimo), facciamo riferimento alle azioni sulla base di dati come se fossero sem-

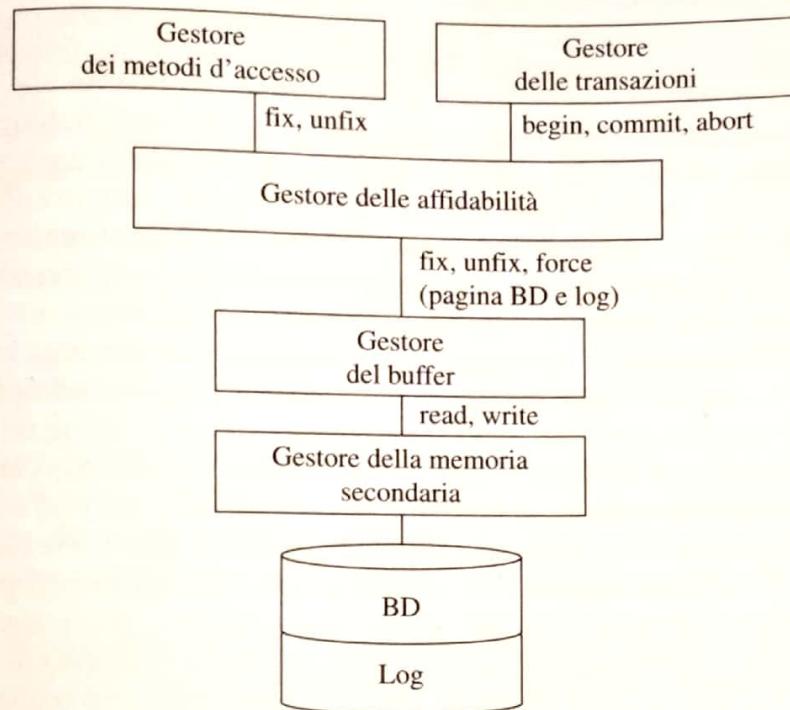


Figura 12.2 Architettura del controllore di affidabilità.

pre operazioni di lettura o scrittura di un'intera pagina. Ovviamente, la realtà è molto più complessa, ma la schematizzazione è sufficiente per capire i principi fondamentali.

**Memoria stabile** Per poter operare, il controllore dell'affidabilità deve disporre di *memoria stabile*, cioè di memoria che risulti resistente ai guasti. La memoria stabile è una astrazione, in quanto nessuna memoria può avere una probabilità nulla di fallimento; tuttavia, meccanismi di replicazione e protocolli di scrittura robusti possono rendere tale probabilità realmente prossima allo zero. I meccanismi di controllo di affidabilità vengono definiti come se la memoria stabile fosse effettivamente esente da guasti; un guasto della memoria stabile viene considerato *catastrofico* e non previsto, perlomeno in questo paragrafo.

A seconda dei casi, la memoria stabile viene realizzata in modi diversi. In alcune applicazioni, si assume che una unità a nastro sia stabile. In altri casi, si assume come stabile una coppia di dispositivi, per esempio una unità a nastro e un disco su cui vengono scritte le stesse informazioni. Una tipica realizzazione di memoria stabile utilizza, al posto di una sola unità a disco, due unità a disco che si dicono "speculari" (*mirrored*), destinate a contenere esattamente la stessa informazione e a essere scritte con un'operazione di "scrittura attenta" che si ritiene riuscita solo se l'informazione viene registrata su entrambi i dischi. In questo modo, l'informazione stabile è anche "in linea", cioè disponibile su un dispositivo ad accesso diretto.

### 12.1.2 Organizzazione del log

Il *log* è un file sequenziale di cui è responsabile il controllore dell'affidabilità, scritto in memoria stabile e contenente informazione ridondante che permette di ricostruire il contenuto della base di dati a seguito di guasti. Sul log vengono registrate le azioni svolte dalle varie transazioni, nell'ordine temporale di esecuzione delle azioni stesse. Pertanto, il log ha un blocco corrente (detto *top*), l'ultimo a essere stato allocato al log stesso; i record nel log vengono scritti sequenzialmente nel blocco corrente, e quando esso termina vengono scritti in un successivo blocco, allocato al log, che diventa il blocco corrente.

I record del log sono di due tipi: di transazione e di sistema. I record di transazione registrano le attività svolte da ciascuna transazione, nell'ordine in cui esse vengono effettuate. Pertanto, ogni transazione inserisce nel log un record di *begin* (corrispondente al comando *start transaction*), vari record relativi alle azioni effettuate (corrispondenti ai comandi *insert*, *delete*, *update*) e un record di *commit* (corrispondente al comando *commit*) oppure di *abort* (corrispondente al comando *rollback*). La Figura 12.3 mostra la sequenza di record presenti in un log. Vengono evidenziati i record scritti relativamente alla transazione  $t_1$ , in un log che viene scritto anche da altre transazioni. La transazione  $t_1$  esegue due modifiche (U, per update) prima di andare a buon fine terminando con un *commit*. I record di sistema indicano l'effettuazione di operazioni specifiche del controllore dell'affidabilità chiamate *dump* (abbastanza rare) e di *checkpoint* (più frequenti), che illustreremo in dettaglio più avanti. La Figura 12.3 evidenzia la presenza nel log di un record di *dump* e di vari record di *checkpoint*.

**Struttura dei record nel log** I record di log che vengono scritti per descrivere le attività di una transazione  $t_i$  sono elencati di seguito.

- I record di *begin*, *commit* e *abort*, che contengono, oltre all'indicazione del tipo di record, anche l'identificativo  $T$  della transazione.
- I record di *update*, che contengono l'identificativo  $T$  della transazione, l'identificativo  $O$  dell'oggetto su cui avviene l'*update*, e poi due valori  $BS$  e  $AS$  che descrivono rispettivamente il valore dell'oggetto  $O$  precedentemente alla

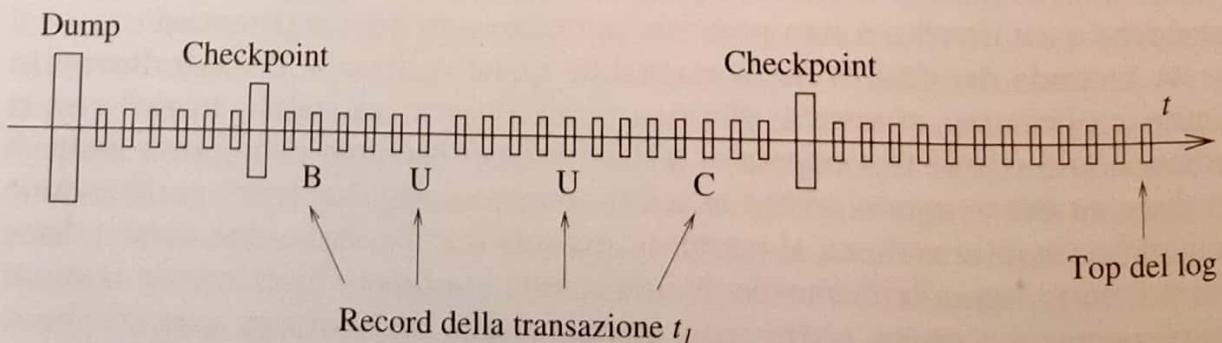


Figura 12.3 Descrizione di un log.

modifica, detto *before state*, e successivamente alla modifica, detto *after state*. Possiamo assumere per semplicità che *AS* e *BS* contengano copie complete delle pagine modificate, ma nella realtà queste informazioni sono molto più sintetiche.

- I record di *insert* e di *delete* sono analoghi a quelli di *update*, da cui si differenziano per l'assenza nei primi del *before state* e nei secondi dell'*after state*.

Nel seguito, useremo i simboli  $B(T)$ ,  $A(T)$ ,  $C(T)$  per denotare i record di *begin*, *abort* e *commit* e  $U(T, O, BS, AS)$ ,  $I(T, O, AS)$  e  $D(T, O, BS)$  per denotare i record di *update*, *insert* e *delete*. Questi record consentono di disfare e rifare le rispettive azioni sulla base di dati, attraverso operazioni specifiche, di competenza del gestore dell'affidabilità, chiamate *Undo* e *Redo* e realizzate nel modo seguente.

- *Undo*: per disfare una azione su un oggetto  $O$  è sufficiente ricopiare nell'oggetto  $O$  il valore  $BS$ ; l'*insert* viene disfatto cancellando l'oggetto  $O$ .
- *Redo*: per rifare una azione su un oggetto  $O$  è sufficiente ricopiare nell'oggetto  $O$  il valore  $AS$ ; il *delete* viene rifatto cancellando l'oggetto  $O$ .

Dato che le primitive di *Undo* e *Redo* sono definite tramite una azione di copia-tura, vale per esse una proprietà essenziale, detta *idempotenza*, per la quale l'ef-fettuazione di un numero arbitrario di undo o redo della stessa azione equivale allo svolgimento di tale azione una sola volta:

$$Undo(Undo(A)) = Undo(A) \quad Redo(Redo(A)) = Redo(A)$$

Questa proprietà è utile perché, come vedremo, si potrebbero avere errori du-rante le operazioni di ripristino, che portano alla ripetizione delle operazioni di *Undo* e *Redo*.

**Checkpoint e dump** I record di log che abbiamo appena visto potrebbero essere sufficienti per svolgere un'operazione di ripristino a seguito di un guasto. Questa sarebbe però molto lunga, perché dovrebbe utilizzare l'intero log; per semplificarla sono stati inventati opportuni accorgimenti, che ora illustriamo. Un *checkpoint* è una operazione che viene svolta periodicamente dal gestore dell'affidabilità (in stretto coordinamento con il buffer manager), con l'obiettivo di registrare quali transazioni sono attive. Esistono in effetti diverse versioni dell'operazione, ma qui, per semplicità didattica, ci limitiamo a vedere la più semplice e intuitiva, costituita dai passi sotto elencati.

1. Si sospende l'accettazione di operazioni di scrittura, *commit* o *abort*, da parte di ogni transazione.
2. Si trasferiscono in memoria di massa (tramite opportune operazioni di *force*) tutte le pagine del buffer su cui sono state eseguite modifiche da parte di transazioni che hanno già effettuato il *commit*.
3. Si scrive in modo sincrono (*force*) nel log un record di *checkpoint* che contiene gli identificatori delle transazioni attive.
4. Si riprende l'accettazione delle operazioni sopra sospese.

Si noti che questa tecnica garantisce che gli effetti delle transazioni che hanno eseguito il commit siano registrati nella base di dati in modo permanente, mentre nel checkpoint sono elencate le transazioni non hanno ancora espresso la loro scelta relativamente all'esito finale (commit o abort).

Un *dump* è una copia completa e consistente della base di dati, che viene normalmente effettuata in mutua esclusione con tutte le altre transazioni quando il sistema non è operativo (per esempio, di notte oppure durante il fine settimana). La copia viene memorizzata su memoria stabile, tipicamente su nastro, ed è anche chiamata *backup*. Dopo la conclusione dell'operazione di dump viene scritto nel log un *record di dump*, che segnala appunto la presenza di una copia fatta in un determinato istante; dopodiché il sistema può tornare al suo funzionamento normale. I sistemi commerciali oggigiorno offrono spesso la possibilità di eseguire le operazioni di creazione del backup mentre il sistema transazionale rimane attivo. La funzionalità è chiamata *hot backup*. Non tratteremo la gestione interna di questa modalità ed assumeremo che il *dump* venga eseguito durante la sospensione del sistema transazionale.

Nel seguito, useremo i simboli *DUMP* per denotare il record di dump e *CK( $T_1, T_2, \dots, T_n$ )* per denotare un record di checkpoint, ove  $T_1, T_2, \dots, T_n$  denotano gli identificatori delle transazioni attive all'istante del checkpoint.

### 12.1.3 Esecuzione delle transazioni e scrittura del log

Durante il funzionamento normale delle transazioni, il controllore dell'affidabilità deve garantire che siano seguite due regole, che definiscono i requisiti minimi che consentono di ripristinare la correttezza della base di dati a fronte di guasti.

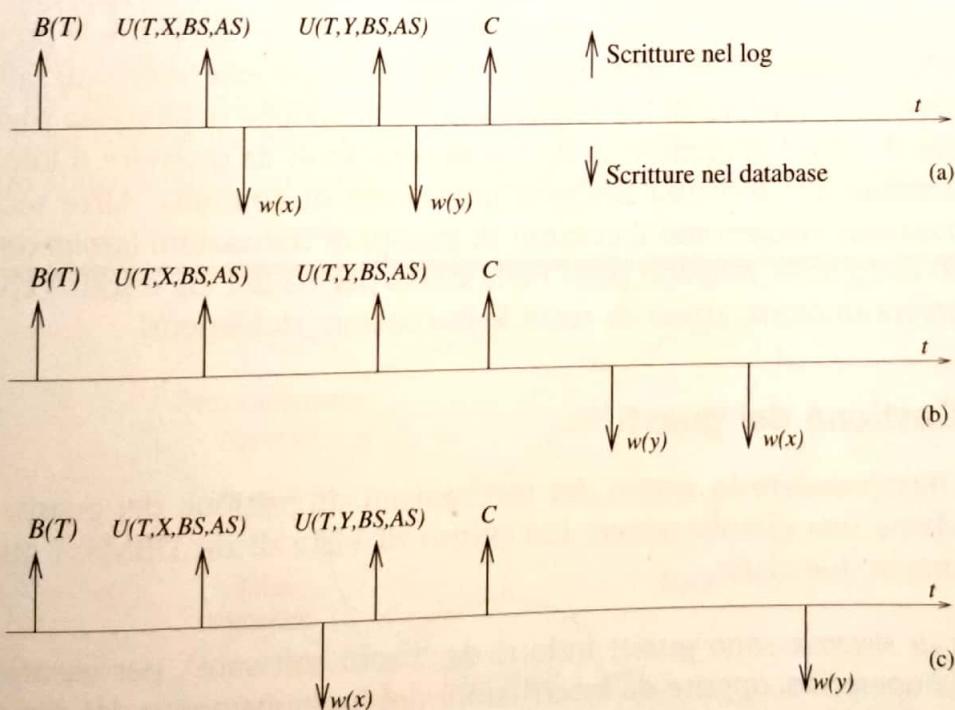
- La *regola WAL* (dall'inglese *Write Ahead Log*, cioè letteralmente: scrivi il log per primo) impone che la parte before state dei record di un log venga scritta nel log (con una operazione di scrittura su memoria stabile) *prima* di effettuare la corrispondente operazione sulla base di dati. Questa regola consente di disfare le scritture già effettuate in memoria di massa da parte di una transazione che non ha ancora effettuato un commit, poiché per ogni aggiornamento viene reso disponibile in modo affidabile il valore precedente la scrittura.
- La *regola di Commit-Precedenza* impone che la parte after state dei record di un log venga scritta nel log (con una operazione di scrittura su memoria stabile) *prima* di effettuare il commit. Questa regola consente di rifare le scritture già decise da una transazione che ha effettuato il commit ma le cui pagine modificate non sono ancora state trascritte dal buffer manager in memoria di massa.

In realtà, anche se le regole fanno riferimento a before state e after state dei record di log, nella pratica entrambe le componenti del record di log vengono scritte assieme; pertanto, una versione semplificata di WAL impone che *i record di log siano scritti prima dei corrispondenti record della base di dati*, mentre una versione semplificata della regola del commit impone che *i record di log siano scritti prima della effettuazione dell'operazione di commit*.

La transazione sceglie, in modo atomico e indivisibile, l'esito di commit nel momento in cui scrive sul log, in modo sincrono (tramite la primitiva *force*), il *record di commit*. Prima di questa azione, un eventuale guasto comporta l'*Undo* delle azioni effettuate, ricostruendo così lo stato iniziale della base di dati. Dopo l'azione di commit, un eventuale guasto comporta il *Redo* delle azioni effettuate, in modo da ricostruire con certezza lo stato finale della transazione. Il record di abort definisce in modo atomico la scelta di abortire, prodotta dal "suicidio" della transazione o imposta dal sistema; dato però che essa, come vedremo, non modifica le decisioni del controllore dell'affidabilità, il record di abort può essere semplicemente scritto in modo asincrono nel buffer che contiene il blocco corrente del log, che può essere successivamente riscritto sul log con una operazione di flush generata dal gestore del buffer.

**Scrittura congiunta di log e base di dati** Le due regole WAL e di Commit-Precedenza impongono alcune restrizioni ai protocolli per la scrittura del log e della base di dati, ma lasciano aperte varie possibilità, illustrate in Figura 12.4. Supponiamo per semplicità che le azioni svolte dalle transazioni siano update, ma non cambierebbe nulla nel caso di *insert* o *delete*. Distinguiamo tre schemi.

- Nel primo schema, illustrato in Figura 12.4a, la transazione scrive inizialmente il record  $B(T)$ , quindi svolge le sue azioni di update scrivendo prima il record di log  $U(T, O, BS, AS)$  e successivamente la pagina della base di dati, che così



**Figura 12.4** Descrizione del protocollo per la scrittura congiunta di log e base di dati.

passa dal valore *BS* al valore *AS*. Tutte queste pagine sono effettivamente scritte (autonomamente dal gestore del buffer oppure con esplicite richieste di *force*) dal buffer manager prima del commit, il quale comporta una scrittura sincrona (*force*). In questo modo, al commit tutte le pagine della base di dati modificate dalla transazione sono certamente scritte in memoria di massa; questo schema non richiede operazioni di *Redo*.

- Nel secondo schema, illustrato in Figura 12.4b, la scrittura dei record di log precede quella delle azioni sulla base di dati, che però avvengono dopo la decisione di commit e la conseguente scrittura sincrona del record di commit sul log; questo schema non richiede operazioni di *Undo*.
- Il terzo schema, più generale e comunemente usato, viene illustrato in Figura 12.4c. Secondo questo schema, le scritture nella base di dati, una volta protette dalle opportune scritture sul log, possono avvenire in un qualunque momento rispetto alla scrittura del record di commit sul log. Questo schema consente al buffer manager di ottimizzare le operazioni di flush relative ai suoi buffer, indipendentemente dal controllore dell'affidabilità; esso però richiede sia *Undo* sia *Redo*.

Si noti che tutti e tre i protocolli rispettano entrambe le regole (WAL e Commit-Precedenza) e scrivono il record di commit in modo sincrono; essi si differenziano solo per il momento in cui scrivono le pagine della base di dati.

Abbiamo così visto quali azioni vengono svolte dal controllore dell'affidabilità per predisporre nel log informazione utile ad azioni di ripristino da malfunzionamenti. Queste azioni hanno un costo, paragonabile al costo di aggiornare la base di dati; in effetti, l'uso di protocolli transazionali robusti rappresenta un sensibile sovraccarico per il sistema, ma è irrinunciabile per garantire le proprietà "acide" delle transazioni. Le operazioni sul log possono essere ottimizzate, per esempio scrivendo i record di log relativi a una transazione nella stessa pagina in cui si scrive il record di commit della transazione (così da ottenere il loro flush contestualmente alla scrittura sincrona del record di commit). Altre tecniche di ottimizzazione consentono il commit di gruppi di transazioni (*group-commit*): vari record di commit vengono posti nella stessa pagina del log e scritti con una unica scrittura sincrona, attesa da tutte le transazioni richiedenti.

#### 12.1.4 Gestione dei guasti

Prima di intraprendere lo studio dei meccanismi di gestione dei guasti, è opportuno darne una classificazione. Dal punto di vista di un DBMS, i guasti si suddividono in due classi:

- *Guasti di sistema*: sono guasti indotti da "bachi software", per esempio del sistema operativo, oppure da interruzioni del funzionamento dei dispositivi, per esempio dovuti a cali di tensione, i quali portano il sistema in uno stato inconsistente. Si traducono in una perdita del contenuto della memoria centrale (e quindi di tutti i buffer), mantenendo invece valido il contenuto della memoria di massa (e quindi della base di dati e del log).

- **Guasti di dispositivo:** sono guasti relativi ai dispositivi di gestione della memoria di massa (per esempio, lo strisciamento delle testine di un disco, che causa la perdita del suo contenuto). Data la nostra assunzione che il log venga scritto sulla memoria stabile, i guasti di dispositivo si traducono in una perdita del contenuto della base di dati, ma non del log. La perdita del contenuto del log è perciò classificata come evento catastrofico, per il quale non viene suggerito alcun rimedio.

Il modello ideale in cui ci poniamo è detto di *fail-stop*: quando il sistema individua un guasto, sia di sistema sia di dispositivo, esso forza immediatamente un arresto completo delle transazioni e il successivo ripristino del corretto funzionamento del sistema operativo (*boot*). Quindi, viene attivata una procedura di ripresa, denominata *ripresa a caldo (warm restart)* nel caso di guasto di sistema e *ripresa a freddo (cold restart)* nel caso di guasto di dispositivo. Al termine delle procedure di ripresa, il sistema diventa nuovamente utilizzabile dalle transazioni; il buffer è completamente vuoto e può riprendere a caricare pagine della base di dati o del log. Questo modello di comportamento è illustrato in Figura 12.5.

Con questo modello, il guasto è un evento istantaneo che accade a un certo istante dell'evoluzione della base di dati. Vediamo quali obiettivi si pone la procedura di ripresa: esisteranno transazioni potenzialmente attive all'atto del guasto, cioè delle quali non si conosce se abbiano ultimato le loro azioni sulla base di dati (perché il buffer manager ha perso ogni informazione utile), e queste si classificano in due categorie in base all'informazione presente nel log. Alcune di esse hanno effettuato il commit, e per loro è necessario rifare le azioni al fine di garantire la persistenza (poiché la transazione che ha effettuato il commit si è impegnata a eseguire tutte le proprie azioni). Altre non hanno effettuato il commit, e per loro è necessario disfare le azioni, in quanto non è noto quali azioni siano state realmente effettuate, ma vi è un impegno a lasciare la base di dati nel suo stato iniziale precedente alla esecuzione della transazione. Si noti che

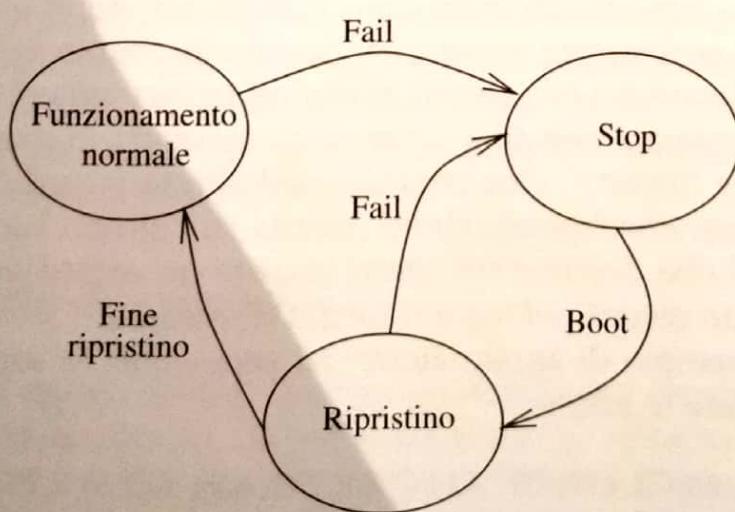


Figura 12.5 Modello fail-stop di funzionamento di un DBMS.

alcuni sistemi, per semplificare i protocolli di ripristino, aggiungono al log un altro record, detto *record di end*, quando le operazioni di trascrizione (*flush*) delle pagine a opera del buffer manager sono terminate. Ciò consente di individuare una terza classe di transazioni, per le quali non è necessario né disfare né rifare le azioni. Però in genere il record di end non è previsto, in modo da non complicare la gestione delle transazioni. Nel seguito assumeremo un modello fail-stop dei guasti e l'assenza di record di end.

**Ripresa a caldo** La ripresa a caldo è articolata in quattro fasi successive.

1. Si accede all'ultimo blocco del log, che era corrente all'istante del guasto, e si ripercorre all'indietro il log fino all'ultimo (cioè più recente) record di checkpoint.
2. Si decidono le transazioni da rifare o disfare. Si costruiscono due insiemi, detti di *UNDO* e di *REDO*, contenenti identificativi di transazioni. Si inizializza l'insieme di *UNDO* con le transazioni attive al checkpoint e l'insieme di *REDO* con l'insieme vuoto. Si percorre poi il log in avanti, aggiungendo all'insieme di *UNDO* tutte le transazioni di cui è presente un record di begin, e spostando dall'insieme di *UNDO* all'insieme di *REDO* tutti gli identificativi delle transazioni di cui è presente il record di commit. Al termine di questa fase, gli insiemi di *UNDO* e *REDO* contengono rispettivamente tutti gli identificativi delle transazioni da disfare e di quelle da rifare.
3. Si ripercorre all'indietro il log disfacendo le transazioni nel set di *UNDO*, risalendo fino alla prima azione della transazione più "vecchia" nei due insiemi di *UNDO* e *REDO*; si noti che questa azione potrebbe precedere il record di checkpoint nel log.
4. Infine, nella quarta fase si applicano le azioni di *Redo* nell'ordine in cui sono registrate nel log. In questo modo, viene replicato esattamente il comportamento delle transazioni originali.

Questo meccanismo garantisce l'atomicità e la persistenza delle transazioni. Per quanto concerne l'atomicità, viene garantito che le transazioni in corso all'istante del guasto lascino la base di dati nello stato iniziale oppure in quello finale; per quanto concerne la persistenza, viene garantito che le pagine nel buffer relative a transazioni completate ma non ancora trascritte in memoria di massa vengano effettivamente completate con una scrittura in memoria di massa. Si noti che ciascuna transazione "incerta", cioè presente nell'ultimo record di checkpoint o iniziata dopo l'ultimo checkpoint, viene disfatta se l'ultimo suo record scritto nel log è un record che descrive un'azione oppure un record di *abort*, oppure rifatta se l'ultimo suo record nel log è il record di *commit*.

Vediamo un esempio di applicazione del protocollo. Si supponga che nel log vengano registrate le azioni:

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

Successivamente, si verifica un guasto. Il protocollo opera come segue.

1. Si accede al record di checkpoint;  $UNDO = \{T2, T3, T4\}$ ,  $REDO = \{\}$ .
2. Successivamente si percorre in avanti il record di log, e si aggiornano gli insiemi di  $UNDO$  e  $REDO$ :
  - (a)  $C(T4)$ :  $UNDO = \{T2, T3\}$ ,  $REDO = \{T4\}$
  - (b)  $B(T5)$ :  $UNDO = \{T2, T3, T5\}$ ,  $REDO = \{T4\}$
  - (c)  $C(T5)$ :  $UNDO = \{T2, T3\}$ ,  $REDO = \{T4, T5\}$
3. Successivamente si ripercorre indietro il log fino all'azione  $U(T2, O1, B1, A1)$ , eseguendo le seguenti azioni di *Undo*:
  - (a) Delete( $O_6$ )
  - (b) Re-Insert( $O_5 = B_7$ )
  - (c)  $O_3 = B_5$
  - (d)  $O_2 = B_3$
  - (e)  $O_1 = B_1$
4. Infine, vengono svolte le azioni di *Redo*:
  - (a)  $O_3 = A_4$  (nota:  $A_4 = B_5!$ )
  - (b)  $O_4 = A_6$

**Ripresa a freddo** La ripresa a freddo risponde a un guasto che provoca il deterioramento di una parte della base di dati; è articolata in tre fasi successive.

1. Durante la prima fase, si accede al *dump* e si ricopia selettivamente la parte deteriorata della base di dati. Si accede anche al più recente record di *dump* nel log.
2. Si ripercorre in avanti il log, applicando relativamente alla parte deteriorata della base di dati sia le azioni sulla base di dati sia le azioni di commit o abort e riportandosi così nella situazione precedente al guasto.
3. Infine, si svolge una ripresa a caldo.

Questo schema ricostruisce tutto il lavoro svolto sulla parte della base di dati soggetta al guasto e quindi, con una ripresa a caldo, garantisce la persistenza e atomicità relativamente all'istante del guasto. La seconda fase dell'algoritmo può essere ottimizzata, per esempio effettuando solo le azioni di transazioni che abbiano eseguito precedentemente il commit.

## 12.2 Controllo di concorrenza

Un DBMS deve spesso servire diverse applicazioni, e rispondere alle richieste provenienti da diversi utenti. Un'unità di misura che viene solitamente utilizzata per caratterizzare il carico applicativo di un DBMS è il *numero di transazioni al secondo*, abbreviato in *tps* (dall'inglese: *transactions per second*) che devono essere gestite dal DBMS per soddisfare le applicazioni. Sistemi tipici, quali per esempio sistemi informativi bancari o finanziari, devono rispondere a carichi dell'ordine

di decine o centinaia di tps; sistemi di prenotazione delle grandi compagnie aeree o di gestione delle carte di credito possono arrivare a migliaia di tps. Per questo motivo, è indispensabile che le transazioni di un DBMS vengano eseguite concorrentemente; è impensabile infatti una loro esecuzione seriale, in cui cioè le transazioni vengono eseguite una alla volta. Solo la concorrenza delle transazioni consente un uso efficiente del DBMS, massimizzando il numero di transazioni servite per secondo e minimizzando i tempi di risposta.

### 12.2.1 Architettura

Le funzionalità del controllo di concorrenza interagiscono, come abbiamo già visto nella Figura 12.1, con quelle relative alla gestione dei metodi di accesso. In modo un po' semplificato, ma efficace, possiamo pensare che il controllore della concorrenza riceva le richieste di accesso ai dati (che, come abbiamo visto nel capitolo precedente, sono in effetti richieste al buffer) e decide se autorizzarle o meno, eventualmente riordinandole (come mostrato in modo schematico nella Figura 12.6, in cui sono ignorate le problematiche relative alla gestione del buffer e dell'affidabilità). Poiché in pratica esso stabilisce l'ordine degli accessi, il controllore della concorrenza viene anche chiamato *scheduler*.

In questo paragrafo daremo una descrizione astratta della base di dati nei termini di oggetti  $x, y, z$  ed esemplificheremo operazioni in memoria centrale su di esse come se i loro valori fossero numerici. In realtà la loro lettura e scrittura richiede la lettura e scrittura dell'intera pagina in cui risiedono tali dati. Infatti, le azioni  $r(x)$  e  $w(x)$  denotano rispettivamente la lettura e la scrittura della pagina in cui il dato  $x$  è memorizzato nel DBMS.

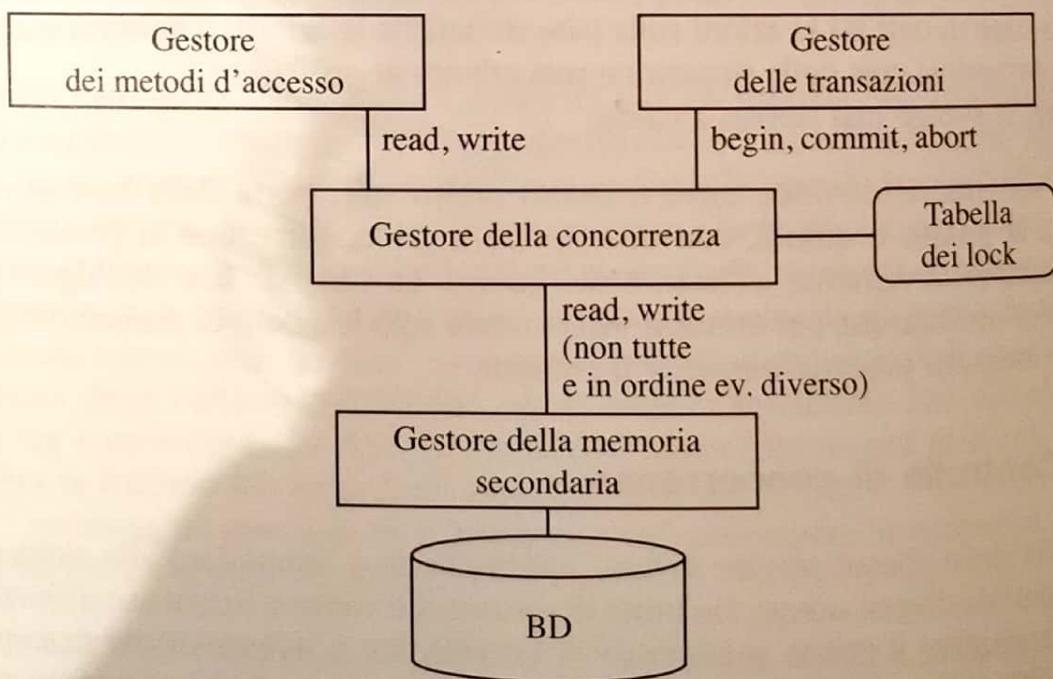


Figura 12.6 Architettura per il controllo di concorrenza.

### 12.2.2 Anomalie delle transazioni concorrenti

L'esecuzione concorrente di varie transazioni può causare alcuni problemi di correttezza, o *anomalie*; la presenza di queste anomalie motiva la necessità di controllare la concorrenza. Vediamo cinque casi tipici.

**Perdita di aggiornamento** Supponiamo di avere due transazioni identiche che operano sullo stesso oggetto della base di dati:

$$\begin{aligned} t_1: & r_1(x), x = x + 1, w_1(x) \\ t_2: & r_2(x), x = x + 1, w_2(x) \end{aligned}$$

dove  $r_i(x)$  denota la lettura del generico oggetto  $x$  da parte della transazione  $t_i$  e  $w_i(x)$  la scrittura dello stesso oggetto. L'incremento dell'oggetto  $x$  è effettuato per opera di un programma applicativo, per esempio un update in SQL. Supponiamo che il valore iniziale di  $x$  sia 2. Se eseguiamo in sequenza le due transazioni  $t_1$  e  $t_2$ , alla fine  $x$  assumerà il valore 4. Analizziamo ora una possibile esecuzione concorrente delle due transazioni, che evidenzia la progressione temporale delle azioni:

Transazione $t_1$	Transazione $t_2$
$r_1(x)$	
$x = x + 1$	
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$w_1(x)$	
commit	

In questo caso il valore finale di  $x$  è 3, perché entrambe le transazioni leggono 2 come valore iniziale di  $x$ . Questa anomalia è nota come *perdita di aggiornamento* (o *lost update*), in quanto gli effetti della transazione  $t_2$  (la prima che scrive il nuovo valore per  $x$ ) sono persi.

**Lettura sporca** Supponiamo di avere una transazione come le precedenti, che incrementa  $x$  di 1, ma va in abort dopo avere scritto, e un'altra transazione che legge tale dato, eseguite in modo concorrente come segue:

Transazione $t_1$	Transazione $t_2$
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	$r_2(x)$
	commit
abort	

Il valore finale di  $x$  al termine dell'esecuzione è 2, ma la seconda transazione ha letto (e potenzialmente comunicato all'esterno) il valore 3, che, essendo la

transazione andata in abort, è come se non fosse mai esistito. L'aspetto critico di questa esecuzione è la lettura della transazione  $t_2$ , che vede uno stato intermedio generato dalla transazione  $t_1$ ; ma  $t_2$  non avrebbe dovuto vedere tale stato, perché prodotto dalla transazione  $t_1$ , che successivamente esegue un abort. Questa anomalia prende il nome di *lettura sporca* (o *dirty read*), in quanto viene letto un dato che rappresenta uno stato intermedio nell'evoluzione di una transazione. Si noti che l'unico modo per ripristinare la correttezza a seguito dell'abort di  $t_1$  sarebbe imporre l'abort di  $t_2$  e, ricorsivamente, di tutte le transazioni che avessero letto dati modificati da  $t_2$ ; questa situazione, denominata "effetto domino", è però assai onerosa da gestire e talvolta non praticabile, perché (come nel caso in esame) il risultato può essere stato comunicato all'esterno.

**Letture inconsistenti** Supponiamo invece che la transazione  $t_1$  svolga solamente operazioni di lettura, ma che ripeta la lettura del dato  $x$  in istanti successivi, come descritto nella seguente esecuzione:

Transazione $t_1$	Transazione $t_2$
$r_1(x)$	
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$r_1(x)$	
commit	

In questo caso,  $x$  assume il valore 2 dopo la prima operazione di lettura e il valore 3 dopo la seconda operazione di lettura. Invece, è opportuno che una transazione che accede due volte alla base di dati trovi esattamente lo stesso valore per ciascun dato letto, e non risenta dell'effetto di altre transazioni.

**Aggiornamento fantasma** Si consideri una base di dati con 3 oggetti  $x$ ,  $y$  e  $z$  che soddisfano un vincolo di integrità, tali cioè che  $x + y + z = 1000$ , ed eseguiamo le seguenti transazioni:

Transazione $t_1$	Transazione $t_2$
$r_1(x)$	
$r_1(y)$	$r_2(y)$
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
$r_1(z)$	commit
$s = x + y + z$	
commit	

La transazione  $t_2$  non altera la somma dei valori e quindi non viola il vincolo di integrità; però la variabile  $s$  della transazione  $t_1$ , che dovrebbe contenere la somma di  $x$ ,  $y$  e  $z$ , contiene in effetti al termine dell'esecuzione il valore 1100. In altri termini, la transazione  $t_1$  osserva solo in parte gli effetti della transazione  $t_2$ , e quindi osserva uno stato che non soddisfa i vincoli di integrità. Questa anomalia prende il nome di *aggiornamento fantasma* (o *ghost update*).

**Inserimento fantasma** Accenniamo infine a una situazione la cui importanza risulterà chiara quando spiegheremo come viene gestito il controllo della concorrenza. Consideriamo una transazione che valuta un valore aggregato relativo all'insieme di tutti gli elementi che soddisfano un predicato di selezione; per esempio, il voto medio degli studenti del primo anno. Consideriamo il caso in cui tale valore aggregato viene valutato due volte, e tra la prima e la seconda valutazione viene inserito un nuovo studente del primo anno. In tal caso, i due valori medi letti dalla transazione potranno essere differenti; notiamo che questa anomalia non può essere evitata facendo riferimento solo ai dati già presenti nella base di dati: è necessario tenere presente che vi è una nuova tupla che compare improvvisamente, come uno spettro (da cui il termine usato in inglese per questa anomalia: *phantom*).

### 12.2.3 Gestione della concorrenza in SQL e in JDBC

Come abbiamo già osservato nel Paragrafo 10.3, la garanzia dell'isolamento può essere molto onerosa e per questo i DBMS (e gli standard) prevedono la possibilità di specificare un grado di protezione ridotto, definito specificando che alcune delle anomalie che abbiamo visto nel paragrafo precedente vengono evitate, ma altre possono manifestarsi.<sup>1</sup> In questo modo, si permette al programmatore (e all'utente evoluto) di decidere di rinunciare a un controllo di concorrenza stringente, proprio al fine di migliorare le prestazioni.

Specificamente, è possibile, in SQL e, in modo analogo anche in JDBC, indicare per ciascuna transazione<sup>2</sup> il livello di isolamento, scegliendo tra quattro possibilità che si differenziano rispetto alle varie anomalie di lettura:

- *read uncommitted*: permette le anomalie di lettura sporca, lettura inconsistente, aggiornamento fantasma, inserimento fantasma;
- *read committed*: evita la lettura sporca, ma non la lettura inconsistente, l'aggiornamento fantasma, e l'inserimento fantasma;
- *repeatable read*: evita tutte le anomalie, escluso l'inserimento fantasma (*phantom*);
- *serializable*: evita tutte le anomalie.

<sup>1</sup>È interessante notare che non vi è una definizione rigorosa di questi concetti che, anche nella specifica degli standard, sono indicati per mezzo di esempi.

<sup>2</sup>Si noti quindi che in generale possono esistere diverse transazioni attive in un certo istante con livelli di isolamento diversi.

I livelli di isolamento diversi da serializable (l'unico che garantisce i massimi requisiti di isolamento) vanno usati esclusivamente con transazioni di sola lettura; quando una transazione effettua letture e scritture è opportuno usare comunque il livello di isolamento massimo, anche perché alcuni sistemi si comportano erroneamente se in una stessa transazione letture ad un livello di isolamento inferiore precedono le scritture, e ciò può causare la perdita di aggiornamento, cioè la anomalia più grave.

I sistemi più evoluti mettono a disposizione del programmatore tutti e quattro i livelli di isolamento. Sta al programmatore delle applicazioni scegliere quale livello utilizzare in funzione del livello di isolamento e delle prestazioni richieste. Per le applicazioni nelle quali la correttezza dei dati letti è vitale (per esempio, applicazioni finanziarie) verrà scelto il livello più elevato, mentre per altre applicazioni in cui la correttezza non è importante (per esempio, valutazioni statistiche in cui valori approssimati sono accettabili) verranno scelti livelli inferiori.

#### 12.2.4 Teoria del controllo di concorrenza

Diamo ora una trattazione precisa dei problemi posti dall'esecuzione concorrente di transazioni. Per questo scopo, è necessario definire un modello formale di transazione. Definiamo una transazione come una sequenza di azioni di lettura o scrittura. Quindi, rispetto agli esempi di anomalie illustrati precedentemente, viene omesso da questo modello ogni riferimento alle operazioni di manipolazione dei dati da parte delle transazioni; per quanto concerne la teoria del controllo di concorrenza, ogni transazione è un oggetto sintattico, di cui si conoscono soltanto le azioni di ingresso/uscita. Per esempio, una transazione  $t_1$  è rappresentata dalla sequenza:

$$t_1 : \quad r_1(x) \quad r_1(y) \quad w_1(x) \quad w_1(y)$$

Dato che le transazioni avvengono in modo concorrente, le operazioni di ingresso/uscita vengono richieste da varie transazioni in istanti successivi. Uno *schedule* rappresenta la sequenza di operazioni di ingresso/uscita presentate da transazioni concorrenti. Uno *schedule*  $S_1$  è quindi una sequenza del tipo:

$$S_1 : \quad r_1(x) \quad r_2(z) \quad w_1(x) \quad w_2(z) \quad \dots$$

dove  $r_1(x)$  rappresenta la lettura dell'oggetto  $x$  effettuata dalla transazione  $t_1$ , e  $w_2(z)$  la scrittura dell'oggetto  $z$  effettuata dalla transazione  $t_2$ . Le operazioni compaiono nello *schedule* seguendo l'ordine temporale con cui sono eseguite sulla base di dati.

Il controllo di concorrenza ha la funzione di accettare alcuni *schedules* e rifiutarne altri, in modo per esempio di evitare che si verifichino le anomalie illustrate nel paragrafo precedente. Per questa ragione, il modulo che gestisce il controllo di concorrenza è chiamato anche *scheduler*: esso ha il compito di intercettare le operazioni compiute sulla base di dati dalle transazioni, decidendo per ciascuna se rifiutarla o accettarla, eventualmente dopo una fase di attesa, con la possibilità quindi di sequenze di esecuzione diverse rispetto a quelle di richiesta.

Inizialmente ci occuperemo di caratterizzare la correttezza degli schedule assumendo che le transazioni che compaiono in esse abbiano un esito (commit o abort) noto; in questo modo, è possibile ignorare le transazioni che producono un abort, togliendo dallo schedule tutte le loro azioni, e concentrarsi solo sulle transazioni che producono un commit. Lo schedule si dice in tal caso una *commit-proiezione* della esecuzione delle operazioni di ingresso/uscita, contenente le sole azioni di transazioni che producono un commit. Questa assunzione è funzionale a uno studio teorico ma è inaccettabile in pratica, perché lo scheduler deve decidere se accettare o rifiutare le azioni di una transazione senza conoscere il suo esito finale, a priori incerto. Per esempio, questa assunzione non consente di trattare l'anomalia di "lettura sporca" vista in precedenza, che si genera quando una transazione decide un abort. Perciò, dovremo rinunciare a questa assunzione quando affronteremo i metodi per il controllo di concorrenza che vengono effettivamente utilizzati nei DBMS.

Definiamo *seriale* uno schedule in cui le azioni di ciascuna transazione compaiono in sequenza, senza essere inframmezzate da istruzioni di altre transazioni. Lo schedule  $S_2$  è per esempio uno schedule seriale in cui vengono eseguite in sequenza le transazioni  $t_0$ ,  $t_1$  e  $t_2$ :

$$S_2 : \quad r_0(x) \quad r_0(y) \quad w_0(x) \quad r_1(y) \quad r_1(x) \quad w_1(y) \quad r_2(x) \quad r_2(y) \quad r_2(z) \quad w_2(z)$$

L'esecuzione di uno schedule  $S_i$  è corretta quando produce lo stesso risultato prodotto da un qualunque schedule seriale  $S_j$  delle stesse transazioni. In tal caso, diremo che  $S_i$  è *serializzabile*. Si noti che questo corrisponde ad assumere che ciascuna transazione sia corretta e che l'esecuzione di una sequenza di transazioni seriali, cioè isolate l'una dall'altra, sia corretta. Per formalizzare che cosa si intende con la frase "due schedule producono lo stesso risultato" è necessario disporre di una definizione di equivalenza fra schedule. La prossima sezione è dedicata all'introduzione della nozione di *view-equivalenza*, concettualmente molto interessante ma purtroppo inutilizzabile in pratica per ragioni computazionali. Pertanto, introdurremo poi una nozione più restrittiva di equivalenza (detta *conflict-equivalenza*) e successivamente due metodi utilizzabili (e utilizzati) in pratica, il *locking a due fasi* e il controllo di concorrenza basato su *timestamp*, che garantiscono la serializzabilità, nel senso che accettano solo schedule serializzabili (anche se, oltre agli schedule non serializzabili, rifiutano anche alcuni schedule che sono serializzabili).

**View-equivalenza** Definiamo dapprima una relazione che lega coppie di operazioni di lettura e scrittura: una operazione di lettura  $r_i(x)$  *legge da* una scrittura  $w_j(x)$  quando  $w_j(x)$  precede  $r_i(x)$  e non vi è alcun  $w_k(x)$  compreso tra le due operazioni. Una operazione di scrittura  $w_i(x)$  viene detta una *scrittura finale* se è l'ultima scrittura dell'oggetto  $x$  che appare nello schedule.

Due schedule vengono detti *view-equivalenti* ( $S_i \approx_V S_j$ ) se possiedono la stessa relazione "legge-da" e le stesse scritture finali. Uno schedule viene detto *view-serializzabile* se esiste uno schedule seriale view-equivalente ad esso. Indichiamo con  $VSR$  l'insieme degli schedule view-serializzabili.

Si considerino gli schedule seguenti:

$$S_3 : w_0(x) r_2(x) r_1(x) w_2(x) w_2(z)$$

$$S_4 : w_0(x) r_1(x) r_2(x) w_2(x) w_2(z)$$

$$S_5 : w_0(x) r_1(x) w_1(x) r_2(x) w_1(z)$$

$$S_6 : w_0(x) r_1(x) w_1(x) w_1(z) r_2(x)$$

$S_3$  è view-equivalente allo schedule seriale  $S_4$  (quindi è view-serializzabile);  $S_5$  non è invece view-equivalente a  $S_4$ , ma è view-equivalente allo schedule seriale  $S_6$ , e quindi risulta anch'esso view-serializzabile.

Notiamo che i seguenti schedule, corrispondenti alle anomalie di perdita di aggiornamento, di letture inconsistenti e di aggiornamento fantasma, non sono view-serializzabili:

$$S_7 : r_1(x) r_2(x) w_2(x) w_1(x)$$

$$S_8 : r_1(x) r_2(x) w_2(x) r_1(x)$$

$$S_9 : r_1(x) r_1(y) r_2(z) r_2(y) w_2(y) w_2(z) r_1(z)$$

Determinare la view-equivalenza di due schedule è un problema con complessità lineare. Determinare se uno schedule è view-equivalente a un qualsiasi schedule seriale è però un problema NP-difficile, perché può esistere un numero esponenziale di schedule seriali (tutte le permutazioni delle transazioni) con cui confrontare lo schedule dato, senza alcun indizio per scegliere quelli con cui confrontarlo. Questo risultato sulla complessità illustra che il concetto di view-equivalenza non può essere usato al fine di caratterizzare la serializzabilità. Si preferisce quindi definire una condizione di equivalenza più ristretta, la quale non copra tutti i casi di equivalenza tra schedule, ma sia utilizzabile nella pratica, presentando una complessità inferiore.

**Conflict-equivalenza** Una nozione di equivalenza più facilmente utilizzabile si basa sulla definizione di conflitto. Date due azioni  $a_i$  e  $a_j$ , con  $i \neq j$ , si dice che  $a_i$  è in *conflitto* con  $a_j$  se esse operano sullo stesso oggetto e almeno una di esse è una scrittura. Possono quindi esistere conflitti lettura-scrittura (*rw* o *wr*) e conflitti scrittura-scrittura (*ww*).

Si dice che lo schedule  $S_i$  è *conflict-equivalente* allo schedule  $S_j$  ( $S_i \approx_C S_j$ ) se i due schedule presentano le stesse operazioni e ogni coppia di operazioni in conflitto è nello stesso ordine nei due schedule. Uno schedule risulta quindi *conflict-serializzabile* se esiste uno schedule seriale a esso conflict-equivalente. Chiamiamo CSR l'insieme degli schedule conflict-serializzabili.

È possibile dimostrare che la classe degli schedule CSR è strettamente inclusa in quella degli schedule VSR: esistono cioè schedule che appartengono a VSR ma non a CSR, mentre tutti gli schedule CSR appartengono a VSR. Quindi la conflict-serializzabilità è condizione sufficiente, ma non necessaria, per la view-serializzabilità.

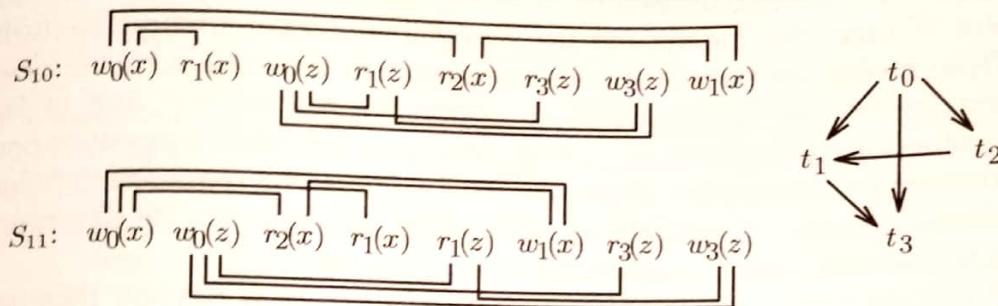


Figura 12.7 Uno schedule  $S_{10}$  conflict-equivalente a uno schedule seriale  $S_{11}$ .

La Figura 12.7 illustra uno schedule  $S_{10}$  conflict-serializzabile, di cui sono posti in evidenza i conflitti, e il corrispondente schedule equivalente seriale  $S_{11}$ .

È possibile determinare se uno schedule è conflict-serializzabile tramite il *grafo dei conflitti*. Il grafo è costruito facendo corrispondere un nodo a ogni transazione. Si traccia quindi un arco orientato da  $t_i$  a  $t_j$  se esiste almeno un conflitto tra un'azione  $a_i$  e un'azione  $a_j$  e si ha che  $a_i$  precede  $a_j$ ; si veda l'esempio in Figura 12.7. Si può dimostrare che uno schedule è in CSR se e solo se il suo grafo dei conflitti è aciclico:

- Se uno schedule  $S$  è in CSR allora, per definizione, esso è conflict-equivalente ad uno schedule seriale  $S_0$ . Sia  $t_1, t_2, \dots, t_n$ , la sequenza delle transazioni in  $S_0$ . Poiché lo schedule seriale  $S_0$  ha tutti i conflitti nello stesso ordine dello schedule  $S$ , nel grafo di  $S$  ci possono essere solo archi  $(i, j)$  con  $i < j$  e quindi il grafo non può avere cicli, perché un ciclo richiede almeno un arco  $(i, j)$  con  $i > j$ .
- Se il grafo di  $S$  è aciclico, allora esiste fra i nodi un "ordinamento topologico" (cioè una numerazione dei nodi tale che il grafo contiene solo archi  $(i, j)$  con  $i < j$ ). Lo schedule seriale le cui transazioni sono ordinate secondo l'ordinamento topologico è equivalente a  $S$ , perché per ogni conflitto  $(i, j)$  si ha sempre  $i < j$ .

L'analisi di ciclicità di un grafo ha una complessità lineare rispetto alle dimensioni del grafo stesso, però la conflict-serializzabilità risulta ancora eccessivamente onerosa in pratica. Per esempio, in un sistema con 100 tps e transazioni che accedono a 10 pagine e durano mediamente 5 secondi, sarà necessario in ogni istante gestire un grafo con 500 nodi e ricordare i 5000 accessi delle 500 transazioni attive; questo grafo continua peraltro a modificarsi dinamicamente, rendendo molto ardue le decisioni dello scheduler. La tecnica risulta assolutamente inaccettabile in un contesto di base di dati distribuita perché, come vedremo nel secondo volume, il grafo deve essere ricostruito a partire da archi che vengono riconosciuti sui diversi server del sistema distribuito.

**Locking a due fasi** I meccanismi di controllo di concorrenza utilizzati in pratica superano le limitazioni discusse in precedenza. Tra di essi, il più noto, e

il primo usato dai DBMS commerciali, si basa sul *locking*, una tecnica che, come quasi tutte le idee che incontrano un notevole successo applicativo, utilizza un principio molto semplice: tutte le operazioni di lettura e scrittura devono essere protette tramite la esecuzione di opportune primitive, *r\_lock*, *w\_lock* e *unlock*; lo scheduler (che viene in questo caso detto anche *lock manager*, perché la sua funzione fondamentale è di gestire i lock) riceve una sequenza di richieste di esecuzione di queste primitive da parte delle transazioni, e ne determina la correttezza con una semplice *ispezione* di una struttura dati.

Nell'esecuzione di operazioni di lettura e scrittura si devono rispettare i seguenti vincoli.

1. Ogni operazione di lettura deve essere preceduta da un *r\_lock* e seguita da un *unlock*; il lock si dice in questo caso *condiviso*, perché su un dato possono essere contemporaneamente attivi più lock di questo tipo.
2. Ogni operazione di scrittura deve essere preceduta da un *w\_lock* e seguita da un *unlock*; il lock si dice in tal caso *esclusivo*, perché non può coesistere con altri lock (esclusivi o condivisi) sullo stesso dato.

Quando una transazione segue queste regole si dice *ben formata rispetto al locking*; si noti che le regole di precedenza precedentemente illustrate non sono strette, e quindi l'operazione di lock di una risorsa può avvenire molto prima di una azione di lettura o scrittura di quella risorsa. Se una transazione deve contemporaneamente leggere e scrivere, la transazione può richiedere solo un lock esclusivo, oppure passare al momento opportuno da un lock condiviso a un lock esclusivo, "incrementando" il livello di lock (*lock upgrade*). In alcuni sistemi, è disponibile una sola primitiva di lock, che non distingue tra lettura e scrittura, e di fatto si comporta come un lock esclusivo.

In genere, le transazioni sono automaticamente ben formate rispetto al locking, poiché emettono le opportune richieste di lock e unlock in modo trasparente al programma applicativo. Il gestore della concorrenza riceve le richieste di lock provenienti dalle transazioni, e concede i lock in base ai lock precedentemente concessi alle altre transazioni. Quando una richiesta di lock è concessa, si dice che la corrispondente risorsa viene *acquisita* dalla transazione richiedente; all'atto dell'*unlock*, la risorsa viene *rilasciata*. Quando una richiesta di lock non viene concessa, la transazione richiedente viene messa in *stato di attesa*; l'attesa termina quando la risorsa viene sbloccata e diviene disponibile. I lock già concessi vengono memorizzati in *tabelle di lock*, gestite dal lock manager.

Ogni richiesta di lock che perviene al lock manager è caratterizzata solo dall'identificativo della transazione che fa la richiesta, e dalla risorsa per la quale la richiesta viene effettuata. La politica che viene seguita dal lock manager per concedere i lock è rappresentata nella *tabella dei conflitti* illustrata in Figura 12.8, in cui le righe identificano le richieste, le colonne lo stato della risorsa richiesta, il primo valore nella cella l'esito della richiesta e il secondo valore nella cella lo stato che verrà assunto dalla risorsa dopo l'esecuzione della primitiva.

I tre *No* presenti nella tabella rappresentano i conflitti che si possono presentare, quando si richiede una lettura o una scrittura su un oggetto già bloccato in scrittura, o una scrittura su un oggetto già bloccato in lettura. In pratica, solo

Richiesta	Stato risorsa		
	libero	r_locked	w_locked
r_lock	OK / r_locked	OK / r_locked	No / w_locked
w_lock	OK / w_locked	No / r_locked	No / w_locked
unlock	error	OK / dipende	OK / libero

Figura 12.8 Tabella dei conflitti per il metodo di locking.

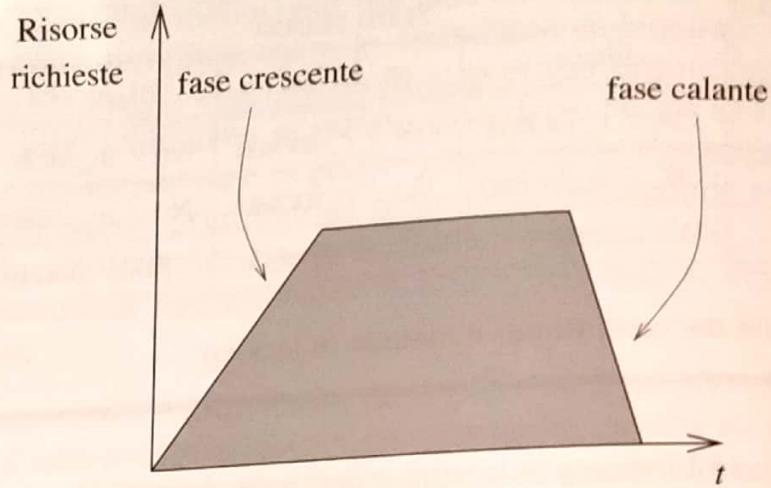
quando un oggetto è bloccato in lettura è possibile dare risposta positiva a un'altra richiesta di lock in lettura; da questa caratteristica discende il nome di *lock condiviso* attribuito al lock in lettura. Nel caso di unlock di una risorsa bloccata in modo condiviso, la risorsa ritorna *libera* quando non ci sono altre transazioni in lettura che operano su di essa; altrimenti, essa rimane bloccata in lettura. Per questo motivo, la corrispondente casella della matrice dei conflitti assume il valore *dipende*. Per tener conto del numero di lettori è necessario introdurre un contatore, che viene incrementato a ogni richiesta di *r\_lock* concessa, e decrementato a ogni *unlock*.

I meccanismi di locking possono essere usati per garantire che le transazioni che lavorano sulla base di dati accedano ai dati in mutua esclusione; è questo il tradizionale contesto d'uso dei meccanismi di controllo su lettura e scrittura presenti nell'ambito dei sistemi operativi. Per avere però la garanzia che le transazioni seguano uno schedule serializzabile è necessario porre una restrizione sull'ordinamento delle richieste di lock, che prende il nome di principio del *locking a due fasi* (in inglese: *Two Phase Locking*, abbreviato in 2PL).

*Locking a due fasi* (2PL): Una transazione, dopo aver rilasciato un lock, non può acquisirne altri.

Come conseguenza di questo principio si possono distinguere nell'esecuzione della transazione due diverse fasi: una prima fase in cui si acquisiscono i lock per le risorse cui si deve accedere (*fase crescente*), e una seconda fase in cui i lock acquisiti vengono rilasciati (*fase calante*). Si tenga conto che il passaggio da un *r\_lock* a un *w\_lock* costituisce un incremento del livello di lock sulla risorsa, che può quindi comparire nella fase crescente della transazione. La Figura 12.9 mostra una rappresentazione grafica del comportamento richiesto dal protocollo di lock a due fasi. L'ascissa rappresenta il tempo, l'ordinata rappresenta il numero di risorse ottenute da una transazione durante la sua esecuzione.

Un sistema in cui le transazioni sono ben formate rispetto al locking (ovvero richiedono sempre un opportuno lock prima di accedere alle risorse e lo rilasciano prima del termine della transazione), con un lock manager che rispetta la politica descritta nella tabella, e in cui le transazioni seguono il principio del lock a due fasi, è un sistema transazionale caratterizzato dalla serializzabilità.



**Figura 12.9** Rappresentazione delle risorse allocate a una transazione con un protocollo di lock a due fasi.

tà delle proprie transazioni. La classe 2PL contiene gli schedule che soddisfano queste condizioni.

Dimostriamo, sia pure informalmente, che ogni schedule che rispetti i requisiti del protocollo di lock a due fasi risulta anche uno schedule serializzabile rispetto alla conflict-equivalenza, ovvero che *la classe 2PL è contenuta nella classe CSR*. Ipotizziamo per assurdo che esista uno schedule  $S$  tale che  $S \in 2PL$  e  $S \notin CSR$ . Se lo schedule non appartiene a CSR vuol dire che costruendo il grafo delle dipendenze tra le transazioni si ottiene un ciclo  $t_1, t_2, \dots, t_n, t_1$ . Se esiste un conflitto tra  $t_1$  e  $t_2$ , vuol dire che esiste una risorsa  $x$  su cui operano entrambe le transazioni in modo conflittuale. Affinché la transazione  $t_2$  possa procedere, è necessario che la transazione  $t_1$  rilasci il lock su  $x$ . D'altra parte, se osserviamo il conflitto tra  $t_n$  e  $t_1$ , vuol dire che esiste una risorsa  $y$  su cui operano entrambe le transazioni in modo conflittuale. Affinché la transazione  $t_1$  possa procedere, è necessario che la transazione  $t_1$  acquisisca il lock sulla risorsa  $y$ , rilasciato da  $t_n$ . Quindi, la transazione  $t_1$  non può essere a due fasi: essa rilascia la risorsa  $x$  prima di acquisire la  $y$ .

Si può poi dimostrare che le classi 2PL e CSR non sono equivalenti, e quindi che 2PL è inclusa strettamente in CSR. Per questo basta mostrare un esempio di schedule non in 2PL ma in CSR, come il seguente:

$$S_{12} : r_1(x) w_1(x) r_2(x) w_2(x) r_3(y) w_1(y)$$

In questo caso, la transazione  $t_1$  deve cedere un lock esclusivo sulla risorsa  $x$  e successivamente richiedere un lock esclusivo sulla risorsa  $y$ , risultando pertanto non a due fasi. Viceversa, lo schedule è conflict-serializzabile rispetto alla sequenza:  $t_3, t_1, t_2$ .

In conclusione, quindi possiamo dire che *la classe 2PL è strettamente contenuta nella classe CSR*.

$t_1$	$t_2$	$x$	$y$	$z$
$r\_lock_1(x)$		free	free	free
$r_1(x)$		1:read		
	$w\_lock_2(y)$		2:write	
	$r_2(y)$		1:wait	
$r\_lock_1(y)$				
	$y = y - 100$			
	$w\_lock_2(z)$			2:write
	$r_2(z)$			
	$z = z + 100$			
	$w_2(y)$			
	$w_2(z)$			
	commit			
	$unlock_2(y)$		1:read	
$r_1(y)$				
$r\_lock_1(z)$				1:wait
	$unlock_2(z)$			1:read
$r_1(z)$				
$s = x + y + z$				
commit				
$unlock_1(x)$		free		
$unlock_1(y)$			free	
$unlock_1(z)$				free

Figura 12.10 Prevenzione di un aggiornamento fantasma tramite l'uso del locking a due fasi.

Riprendendo gli esempi discussi nel Paragrafo 12.2.2, possiamo vedere come il 2PL permette di evitare l'insorgere delle anomalie (anche se potrebbe introdurre situazioni di blocco critico o stallo, che discuteremo nel Paragrafo 12.2.6).

Esaminiamo in particolare una delle anomalie, l'aggiornamento fantasma. Consideriamo l'esempio introdotto nel Paragrafo 12.2.2; rappresentiamo la stessa sequenza di accessi e mostriamo che il 2PL risolve il problema. La Figura 12.10 descrive per ogni risorsa il suo stato come libero (*free*), bloccato in lettura dalla  $i$ -esima transazione ( $i : read$ ) oppure bloccato in scrittura dalla  $i$ -esima transazione ( $i : write$ ); illustriamo anche l'esito negativo di una richiesta di lock della  $i$ -esima transazione, posta in stato di attesa ( $i : wait$ ). Si noti che, per effetto del 2PL, le richieste di lock di  $t_1$  relative alle risorse  $z$  e  $x$  vengono messe in attesa, e la transazione  $t_1$  può procedere solo quando tali risorse vengono sbloccate da  $t_2$ . Al termine della esecuzione, la variabile  $s$  contiene il valore corretto della somma  $x + y + z$ .

È possibile vedere che le anomalie di lettura inconsistente e di perdita di aggiornamento vengono ugualmente risolte dal 2PL. Qualche osservazione in

più è necessaria invece per quanto riguarda le anomalie di lettura sporca e di inserimento fantasma.

Per la lettura sporca, è evidente che finora abbiamo ignorato il problema, in quanto abbiamo ragionato nell'ipotesi di commit-proiezione, trascurando le transazioni che si concludono con un abort. Per rimuovere tale ipotesi ed evitare l'anomalia, si può procedere attraverso una restrizione del protocollo 2PL, che porta al cosiddetto *2PL stretto (strict 2PL)*:

*Locking a due fasi stretto (strict 2PL):* I lock di una transazione possono essere rilasciati solo dopo aver correttamente effettuato le operazioni di commit/abort.

In pratica, con questo vincolo i lock vengono rilasciati solo al termine della transazione, dopo che ciascun dato è stato portato nel suo stato finale. Questa versione del 2PL viene usata da molti sistemi commerciali. L'esempio in Figura 12.10 utilizza il 2PL stretto, poiché le azioni di rilascio dei lock seguono l'azione di commit, esplicitamente indicata nello schedule. Tramite il 2PL stretto viene reso impossibile il verificarsi di letture sporche, perché viene impedito l'accesso (da parte di altre transazioni) a dati scritti da transazioni che ancora non hanno effettuato il commit.

L'anomalia di inserimento fantasma (*phantom*) richiede invece una più precisa definizione del concetto di lock: finora abbiamo assunto che i lock siano definiti con riferimento agli oggetti *presenti* nella base di dati. L'anomalia è legata al fatto che la lettura ripetuta fa riferimento alle tuple che soddisfano una certa condizione, indipendentemente dal fatto che siano presenti nella base di dati oppure no. Il problema sarebbe evitato se potessimo impedire l'inserimento di un nuovo studente del primo anno (visto che stiamo lavorando sugli studenti del primo anno). Allo scopo, è necessario che i lock possano essere definiti anche con riferimento a condizioni (o *predicati*) di selezione, impedendo non solo l'accesso ai dati coinvolti ma anche la scrittura di nuovi dati che soddisfano il predicato. I *lock di predicato* sono realizzati nei sistemi relazionali con l'ausilio degli indici o, nel caso in cui essi non esistano, bloccando intere relazioni. È quindi evidente come essi possano penalizzare molto l'efficienza dei sistemi.

Concludiamo questo paragrafo osservando come nel contesto del 2PL possono essere realizzati i diversi livelli di isolamento (per le transazioni di sola lettura) illustrati nel Paragrafo 12.2.3:

- **read uncommitted:** la transazione non chiede lock e non osserva nemmeno i lock esclusivi posti da altre transazioni;
- **read committed:** richiede lock per le letture, rilasciandoli subito dopo, quindi senza 2PL; in questo modo si evitano le letture sporche, ma non le altre anomalie tipiche delle letture;
- **repeatable read:** applica in 2PL stretto, ma applicando i lock a singole tuple; sono evitate tutte le anomalie ma non l'inserimento fantasma (*phantom*), perché non è possibile impedire l'inserimento di nuove tuple;
- **serializable:** applica il 2PL stretto e i lock di predicato e quindi evita tutte le anomalie.

**Controllo di concorrenza basato sui timestamp** Illustriamo infine un altro metodo per il controllo di concorrenza assai semplice da realizzare nella sua versione base e diffusosi attraverso una interessante variante. Questo metodo utilizza i *timestamp*, cioè identificatori associati ad ogni evento temporale che definiscono un ordinamento totale sugli eventi. Nei sistemi centralizzati, il timestamp viene generato leggendo il valore dell'orologio di sistema al momento in cui è avvenuto l'evento. Il controllo di concorrenza mediante timestamp (*metodo TS*) avviene nel seguente modo:

- a ogni transazione si associa un timestamp che rappresenta il momento di inizio della transazione;
- si accetta uno schedule solo se esso riflette l'ordinamento seriale delle transazioni in base al valore del timestamp di ciascuna transazione.

Questo metodo di controllo di concorrenza, forse il più semplice di tutti dal punto di vista della realizzazione, impone che le transazioni risultino serializzate in base all'ordine in cui esse acquisiscono il loro timestamp. A ogni oggetto  $x$  vengono associati due indicatori,  $\text{WTM}(x)$  e  $\text{RTM}(x)$ , che sono rispettivamente i timestamp della transazione che ha eseguito l'ultima scrittura e della transazione con  $t$  più grande che ha letto  $x$ . Allo scheduler arrivano le richieste di accesso agli oggetti del tipo  $r_t(x)$  o  $w_t(x)$ , dove  $t$  rappresenta il timestamp della transazione che esegue la lettura o la scrittura. Lo scheduler non fa altro che permettere o no l'operazione, secondo la seguente politica:

- $r_t(x)$ : se  $t < \text{WTM}(x)$  la transazione viene uccisa, altrimenti la richiesta viene accettata; in tal caso  $\text{RTM}(x)$  viene aggiornato e posto pari al massimo tra  $\text{RTM}(x)$  e  $t$ ;
- $w_t(x)$ : se  $t < \text{WTM}(x)$  o  $t < \text{RTM}(x)$  la transazione viene uccisa, altrimenti la richiesta viene accettata; in tal caso  $\text{WTM}(x)$  viene aggiornato e posto pari a  $t$ .

In pratica, ogni transazione non può leggere o scrivere un dato scritto da una transazione con timestamp superiore, e non può scrivere su un dato che è già stato letto da una transazione con timestamp superiore.

Vediamo un esempio. Si supponga che sia  $\text{RTM}(x) = 7$  e  $\text{WTM}(x) = 5$  (ovvero l'oggetto  $x$  è stato letto da transazioni con timestamp 7 o minore e scritto l'ultima volta dalla transazione con timestamp 5). Nel seguito, descriviamo la risposta dello scheduler alle richieste di lettura e scrittura ricevute:

Richieste	Risposte	Nuovi valori
$r_6(x)$	ok	
$r_7(x)$	ok	
$r_9(x)$	ok	$\text{RTM}(x) = 9$
$w_8(x)$	no	$t_8$ uccisa
$w_{11}(x)$	ok	$\text{WTM}(x) = 11$
$r_{10}(x)$	no	$t_{10}$ uccisa

Il metodo TS comporta l'uccisione di un gran numero di transazioni; inoltre, questa versione del metodo è corretta sotto l'ipotesi di uso di commit-proiezioni. Per rimuovere questa ipotesi è necessario "bufferizzare" le scritture, cioè effettuarle in memoria e trascriverle in memoria di massa solo dopo il commit; ciò comporta che le letture da parte di altre transazioni dei dati memorizzati nel buffer e in attesa di commit vengano a loro volta messe in attesa del commit della transazione scrivente, in pratica introducendo meccanismi di attesa analoghi a quelli di locking.

Una variante del metodo TS prevede l'uso della cosiddetta *Regola di Thomas*, la quale specifica un comportamento leggermente diverso per le operazioni di scrittura. La regola è che per un'operazione  $w_t(x)$ , se  $t < \text{RTM}(x)$  la transazione viene uccisa, altrimenti se  $t < \text{WTM}(x)$  la scrittura viene *scartata* e la transazione continua, altrimenti la richiesta viene accettata. L'idea è che si possa evitare di eseguire una scrittura su un oggetto che è già stato modificato da una transazione più giovane e che non è stato ancora letto da una transazione più giovane. L'applicazione di questa regola riduce il numero di situazioni in cui una transazione deve essere uccisa, ma presenta un beneficio per transazioni reali molto limitato, in quanto sono rare le situazioni in cui le transazioni eseguono operazioni di scrittura di un nuovo valore di un oggetto senza aver prima acquisito il valore corrente.

Un'altra variante del metodo, molto interessante sia sul piano teorico sia su quello pratico, è l'uso delle *multiversioni*, che consiste nel mantenere diverse copie degli oggetti della base di dati, per ogni transazione che modifica la base di dati. Ogni volta che una transazione scrive un oggetto, la vecchia copia non viene persa, ma una nuova  $N$ -esima copia viene creata, con un corrispondente  $\text{WTM}_N(x)$ . Si ha invece un solo  $\text{RTM}(x)$  globale. Quindi, in un generico istante sono attive  $N \geq 1$  copie di ciascun oggetto  $x$ ; con questo metodo, le richieste di lettura non vengono mai rifiutate, ma vengono dirette alla versione dei dati corretta rispetto al timestamp della transazione richiedente. Le copie vengono rilasciate quando sono divenute inutili, in quanto non esistono più transazioni in lettura interessate al loro valore. Le regole di comportamento diventano:

- $r_t(x)$ : una lettura è sempre accettata. Si legge un  $x_k$  siffatto: se  $t > \text{WTM}_N(x)$ , allora  $k = N$ , altrimenti si prende  $i$  in modo che sia  $\text{WTM}_i(x) < t < \text{WTM}_{i+1}(x)$ ;
- $w_t(x)$ : se  $t < \text{RTM}(x)$  si rifiuta la richiesta, altrimenti si aggiunge una nuova versione del dato ( $N$  cresce di uno) con  $\text{WTM}_N(x) = t$ .

L'idea di adottare più versioni, introdotta teoricamente nel contesto dei metodi basati su timestamp, è poi stata estesa anche agli altri metodi, e in particolare viene usata nel contesto del locking a due fasi. Un uso interessante delle versioni si ottiene limitando le copie a due, tenendo cioè una copia precedente e una successiva a ogni aggiornamento durante le operazioni di scrittura; le transazioni in lettura che sono sincronizzate prima della transazione in scrittura possono in tal caso accedere alla copia più vecchia.

Come avviene per il 2PL, anche un sistema che utilizza il controllo di correnza multiversione può essere utilizzato in una modalità che offre un minore livello di isolamento, con il beneficio di offrire prestazioni migliori. La soluzione

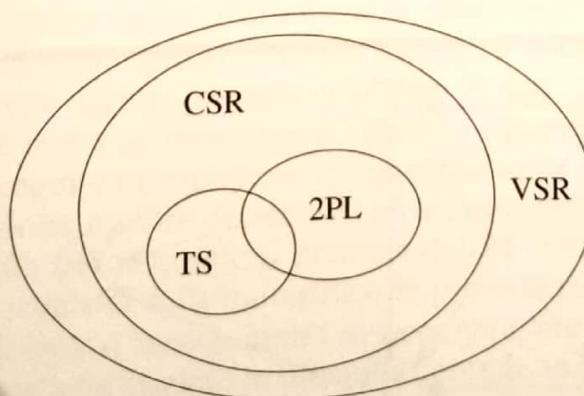
adottata da molti sistemi moderni corrisponde al livello di isolamento chiamato *snapshot isolation*. In questo livello si assume che ciascuna transazione sia in grado di accedere allo stato della base di dati così come si presentava al momento dell'inizio della transazione. Ciò viene ottenuto mediante il supporto per la gestione delle versioni. Nel momento in cui una transazione esegue un'operazione di scrittura, si verifica se i dati che stanno per essere modificati sono stati nel frattempo modificati da altre transazioni. In tal caso, per evitare l'anomalia di perdita di update, la transazione viene uccisa.

Il livello di protezione offerto è inferiore al livello *serializable*. Supponiamo infatti di avere una coppia di transazioni che operano su una tabella che descrive il colore di un insieme di palline che possono essere bianche o nere:

```
 $T_1: \text{update Palline set Colore='Bianco' where Colore='Nero'}$ 
 $T_2: \text{update Palline set Colore='Nero' where Colore='Bianco'}$ 
```

L'esecuzione seriale delle due transazioni produrrà uno stato in cui le palline saranno tutte bianche o tutte nere. Con la *snapshot isolation*, l'esecuzione concorrente delle due transazioni potrà produrre invece uno stato finale in cui i colori delle palline saranno stati scambiati, dimostrando la non serializzabilità di questa esecuzione.

**Confronto fra VSR, CSR, 2PL e TS** La Figura 12.11 illustra la tassonomia dei metodi VSR, CSR, 2PL e TS fin qui introdotti. Si osserva che la classe VSR è la classe più generale: essa include strettamente al suo interno CSR, la quale a sua volta include sia la classe 2PL sia la classe TS. 2PL e TS a loro volta presentano una intersezione non nulla, ma nessuna presenta una relazione di inclusione con l'altra. Quest'ultima caratteristica può essere verificata facilmente, costruendo schedule che sono in TS ma non in 2PL, oppure in 2PL ma non in TS, oppure infine in 2PL e in TS.



**Figura 12.11** Tassonomia delle classi di schedule accettate dai metodi VSR, CSR, 2PL e TS.

Dapprima mostriamo che esistono degli schedule che sono in TS ma non in 2PL. Si consideri lo schedule  $S_{13}$ :

$$S_{13} : r_1(x) w_2(x) r_3(x) r_1(y) w_2(y) r_1(v) w_3(v) r_4(v) w_4(y) w_5(y)$$

Il corrispondente grafo dei conflitti, illustrato in Figura 12.12, mostra l'assenza di ciclicità e quindi l'appartenenza a CSR dello schedule. L'ordinamento seriale delle transazioni conflict-equivalenti allo schedule di partenza è  $t_1 t_2 t_3 t_4 t_5$ . Lo schedule non risulta essere 2PL in quanto  $t_2$  prima rilascia  $x$  affinché venga letto da  $t_3$  e poi acquisisce  $y$ , rilasciato da  $t_1$ , ma risulta essere in TS, poiché presso ogni oggetto le transazioni operano nell'ordine indotto dai timestamp.

Uno schedule che è sia in TS sia in 2PL è per esempio il semplice schedule (seriale)  $r_1(x) w_1(x) r_2(x) w_2(x)$ . Invece lo schedule  $r_2(x) w_2(x) r_1(x) w_1(x)$ , in cui cioè la transazione  $t_2$  acquisisce il timestamp dopo la transazione  $t_1$  ma si presenta per prima all'oggetto  $x$ , non appartiene a TS ma appartiene a 2PL.

Confrontiamo tra di loro le tecniche 2PL e TS. Emergono alcune differenze significative.

- Nel 2PL le transazioni sono poste in attesa. Nel TS esse sono uccise e poi riavviate.
- L'ordine di serializzazione nel 2PL è imposto dai conflitti, mentre nel TS è imposto dai timestamp stessi.
- La necessità di attendere l'esito della transazione comporta l'allungarsi del tempo di blocco in 2PL (il passaggio da 2PL a 2PL stretto) e la creazione di condizioni di attesa in TS.
- Il metodo 2PL può presentare il problema del blocco critico, che vedremo nel prossimo paragrafo.
- Il restart usato dal TS costa in genere più dell'attesa imposta da 2PL.

I DBMS commerciali utilizzano in effetti varianti di queste tecniche che cercano di limitare gli inconvenienti, soprattutto in termini di prestazioni.

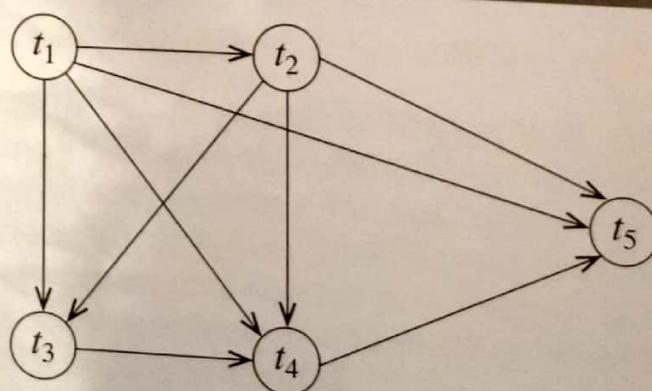


Figura 12.12 Grafo dei conflitti per lo schedule  $S_{13}$ .

## 12.2.5 Meccanismi per la gestione dei lock

Un lock manager è un processo in grado di essere invocato da tutti i processi che intendono accedere alla base di dati. I processi per accedere alle risorse dovranno eseguire delle procedure di *r\_lock*, *w\_lock* e *unlock*, in genere caratterizzate dai seguenti parametri:

$$\begin{aligned} r\_lock(T, x, \text{errcode}, \text{timeout}) \\ w\_lock(T, x, \text{errcode}, \text{timeout}) \\ unlock(T, x) \end{aligned}$$

*T* rappresenta l'identificativo della transazione, *x* l'elemento per il quale si richiede o si rilascia il lock. Rispetto alla definizione della procedura che era stata data nel precedente paragrafo, compare qualche ulteriore parametro: *errcode* rappresenta un valore restituito dal lock manager, e vale zero qualora la richiesta sia stata soddisfatta, mentre assume un valore diverso da zero qualora la richiesta non sia stata soddisfatta; *timeout* rappresenta l'intervallo massimo di tempo che la procedura chiamante è disposta ad aspettare per ottenere il lock sulla risorsa.

Quando un processo richiede una risorsa e la richiesta può essere soddisfatta, il lock manager tiene traccia del cambiamento dello stato della risorsa nelle sue tabelle interne e restituisce immediatamente il controllo al processo; in questo caso, il ritardo introdotto dal lock manager sul tempo di esecuzione della transazione è molto modesto, in quanto la richiesta non comporta operazioni di ingresso-uscita.

Quando invece la richiesta non può essere immediatamente soddisfatta, il sistema inserisce il processo richiedente in una coda associata alla risorsa; ciò comporta un'attesa arbitrariamente lunga, e quindi il processo associato alla transazione viene sospeso. Appena una risorsa viene rilasciata, il lock manager controlla se esistono dei processi in attesa della risorsa e nel caso prende il primo processo della coda e concede a esso la risorsa. L'efficienza del lock manager dipende perciò dalla probabilità che le richieste di una transazione vadano in conflitto; tale probabilità è pari circa a  $k \times m/n$ , dove *k* è il numero di transazioni operanti sul sistema, *m* il numero medio di risorse cui accede una transazione, e *n* il numero di diversi oggetti presenti nella base di dati.

Quando infine scatta un timeout e la richiesta è insoddisfatta, la transazione richiedente può eseguire un *rollback*, cui generalmente seguirà una ripartenza della stessa transazione, oppure decidere di proseguire, richiedendo nuovamente il lock, in quanto un fallimento nella richiesta di lock non comporta un rilascio delle altre risorse acquisite dalla transazione in precedenza.

Alle tabelle di lock si accede molto di frequente; per questo, il lock manager mantiene queste informazioni in memoria centrale, in modo da minimizzare i tempi di accesso. Le tabelle hanno la seguente struttura: a ciascun oggetto si associano due bit di stato (per rappresentare i tre possibili stati servono almeno due bit) e un contatore, che rappresenta il numero di processi in attesa di quell'oggetto.

**Lock gerarchico** Per ora si è parlato dei problemi di lock citando generiche risorse e oggetti della base di dati, perché i principi teorici su cui si basa il 2PL sono indipendenti dalla tipologia degli oggetti cui il metodo viene applicato. In molti sistemi reali, è però possibile specificare i lock a livelli diversi: si parla allora di *granularità dei lock*. Per esempio, è possibile bloccare intere tabelle, o insiemi di tuple, o campi di singole tuple.

Per introdurre livelli diversi di granularità del locking, si opera una estensione del protocollo di lock tradizionale, detta *lock gerarchico (hierarchical locking)*. Si consideri la Figura 12.13, che illustra la gerarchia delle risorse che fanno parte di una base di dati. La tecnica del lock gerarchico permette alle transazioni di definire in modo molto efficiente i lock di cui hanno bisogno, operando al livello prescelto della gerarchia. Così è possibile per una transazione ottenere un lock per l'intera base di dati (come può essere richiesto quando si vuole effettuare un salvataggio dello stato della base di dati), o per una specifica tupla.

La tecnica fornisce un insieme più ricco di primitive di richiesta di lock, ciascuna con gli opportuni parametri, come visto nel paragrafo precedente:

- XL: lock esclusivo (*exclusive lock*). Corrisponde al write-lock del protocollo normale;
- SL: lock condiviso (*shared lock*). Corrisponde al read-lock del protocollo normale.

I tre lock successivi sono specifici di questa tecnica:

- ISL: intenzione di lock condiviso (*intentional shared lock*). Esprime l'intenzione di bloccare in modo condiviso uno dei nodi che discendono dal nodo corrente;
- IXL: intenzione di lock esclusivo (*intentional exclusive lock*). Esprime l'intenzione di bloccare in modo esclusivo uno dei nodi che discendono dal nodo corrente;

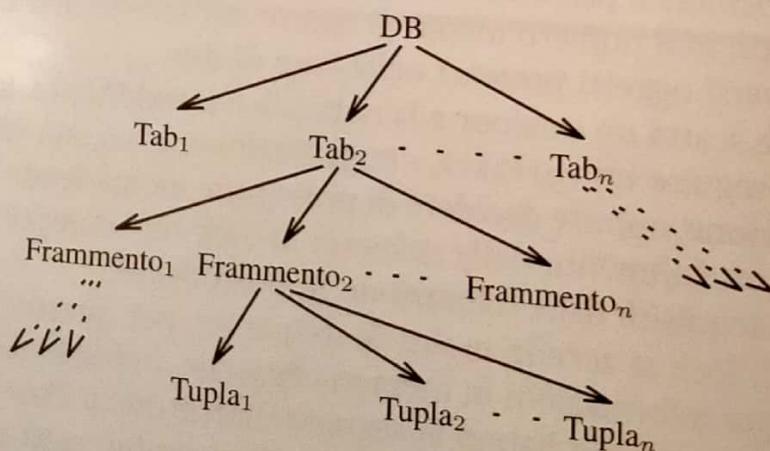


Figura 12.13 La gerarchia delle risorse.

- SIXL: lock condiviso, intenzione di lock esclusivo (*shared intentional-exclusive lock*). Blocca il nodo corrente in modo condiviso ed esprime l'intenzione di bloccare in modo esclusivo uno dei nodi che discendono dal nodo corrente.

Se per esempio si vuole bloccare in scrittura una tupla della tabella, e la gerarchia è quella rappresentata in Figura 12.13, allora bisognerà prima richiedere un IXL al livello della base di dati. Quando la richiesta verrà soddisfatta si potranno chiedere in sequenza un IXL per la relazione e uno per la partizione in cui risiede la tupla desiderata. Quando il lock IXL al livello della partizione sarà stato concesso, si potrà richiedere un lock esclusivo per la particolare tupla. Quando poi la transazione avrà terminato, dovrà rilasciare i lock nell'ordine inverso in cui questi sono stati ottenuti, risalendo un passo alla volta la gerarchia.

Descriviamo in modo più formale le regole che devono essere rispettate dal protocollo.

1. Si richiedono i lock partendo dalla radice e scendendo lungo l'albero.
2. Si rilasciano i lock partendo dal nodo bloccato di granularità più piccola e risalendo lungo l'albero.
3. Per poter richiedere un lock SL o ISL su un nodo, si deve già possedere un lock ISL o IXL sul suo nodo padre.
4. Per poter richiedere un lock IXL, XL o SIXL su un nodo, si deve già possedere un lock SIXL o IXL sul suo nodo padre.
5. Le regole di compatibilità utilizzate dal lock manager per decidere se accettare la richiesta di lock in base allo stato del nodo e al tipo di richiesta sono riportate nella tabella in Figura 12.14.

La scelta del livello di lock è lasciata al progettista delle applicazioni o dell'amministratore della base di dati, sulla base delle caratteristiche delle transazioni: transazioni che effettuano modifiche "localizzate", accedendo a un insieme limitato di oggetti, utilizzeranno una granularità fine; transazioni che effettuano accessi a grandi moli di dati utilizzeranno granularità più grossolana. La scelta deve essere svolta con accortezza, in quanto l'uso di un livello troppo grossolano può porre

Richiesta	Stato risorsa				
	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	No
IXL	OK	OK	No	No	No
SL	OK	No	OK	No	No
SIXL	OK	No	No	No	No
XL	No	No	No	No	No

Figura 12.14 Compatibilità tra le modalità di lock in presenza di gerarchie.

limitazioni al parallelismo (perché rende elevata la probabilità di conflitti), mentre l'uso di un livello troppo fine costringe a richiedere un gran numero di lock uno per uno, costringendo il lock manager a una quantità considerevole di lavoro ed esponendolo al rischio di un fallimento dopo aver acquisito molte risorse.

### 12.2.6 Blocco critico

Il blocco critico (detto anche stallo, oppure abbraccio mortale, dal termine inglese *deadlock*) costituisce un problema rilevante, tipico dei sistemi concorrenti in cui si introducono condizioni di attesa. Supponiamo di avere una transazione  $t_1$  che deve eseguire le operazioni  $r(x)$ ,  $w(y)$ , e una seconda transazione  $t_2$  che deve eseguire  $r(y)$ ,  $w(x)$ . Se viene usato il protocollo di lock a due fasi, si può presentare il seguente schedule:

$$r\_lock_1(x), r\_lock_2(y), read_1(x), read_2(y), w\_lock_1(y), w\_lock_2(x)$$

A questo punto nessuna delle due transazioni riesce a procedere e il sistema è bloccato. Il problema è dato dal fatto che  $t_1$  è in attesa che si liberi l'oggetto  $y$ , che è bloccato da  $t_2$ , e a sua volta  $t_2$  è in attesa dell'oggetto  $x$ , bloccato da  $t_1$ . Questa situazione è caratteristica di tutti i sistemi in cui si utilizzano dei meccanismi di blocco delle risorse.

Valutiamo la probabilità che si verifichi un evento del genere: consideriamo una tabella che consiste di  $n$  diverse tuple, con identica probabilità di accesso. La probabilità che due transazioni che operano un solo accesso vadano in conflitto è  $1/n$ ; la probabilità che si verifichi un blocco critico di lunghezza 2 è pari alla probabilità di un secondo conflitto, e quindi vale  $1/n^2$ . Non consideriamo il caso dei blocchi critici generati da catene più lunghe, poiché in questo caso la probabilità decresce in modo esponenziale con la crescita della catena, e il contributo al numero totale di blocchi critici che si possono verificare in un sistema reale è trascurabile. Limitatamente al caso di blocchi critici costituiti da coppie di transazioni, la probabilità di conflitto cresce linearmente col numero globale  $k$  di transazioni presenti nel sistema e quadraticamente col numero medio  $m$  di risorse cui ciascuna transazione fa accesso. L'effettiva probabilità di occorrenza del blocco critico è leggermente superiore a quello che la semplice analisi statistica precedente faccia pensare, a causa delle dipendenze che esistono tra i dati oppure tra le diverse transazioni (per cui, quando una transazione accede a un dato, è più probabile che acceda a un altro dato legato a questo da una qualche relazione). In conclusione, possiamo assumere che la probabilità che si verifichi un blocco critico nei sistemi transazionali sia bassa, ma non nulla; questa considerazione è confermata da valutazioni sperimentali su sistemi standard nel mondo delle applicazioni finanziarie.

Tre sono le tecniche che vengono comunemente usate per risolvere il problema del blocco critico:

1. timeout;
2. prevenzione (*deadlock prevention*);
3. rilevamento (*deadlock detection*).

**Uso del timeout** La tecnica del timeout è molto semplice. Le transazioni rimangono in attesa di una risorsa per un tempo prefissato. Se passa questo tempo e la risorsa non è stata ancora concessa, allora alla richiesta di lock viene data risposta negativa; in questo modo, una transazione che fosse in deadlock verrebbe comunque tolta dalla condizione di attesa, e presumibilmente abortita. Per la sua semplicità, questa tecnica si lascia preferire nella stragrande maggioranza dei DBMS commerciali.

Per quanto riguarda la scelta del valore di timeout, bisogna saper valutare pro e contro tra due diversi aspetti: da una parte un valore elevato del timeout tende a risolvere tardi i blocchi critici, dopo che le transazioni coinvolte nel blocco hanno passato del tempo in attesa, d'altra parte un timeout troppo basso corre il rischio di rilevare come blocchi critici anche situazioni in cui una transazione sta aspettando una risorsa senza che vi sia un vero deadlock, uccidendo inutilmente una transazione e sprecando il lavoro già svolto dalla transazione.

**Prevenzione dei blocchi critici** Vi sono diverse tecniche che possono essere utilizzate per prevenire l'insorgenza di un blocco critico. Una tecnica prevede di richiedere il lock di tutte le risorse necessarie alla transazione in una sola volta. Essa presenta però il problema che le transazioni spesso non conoscono a priori le risorse cui vogliono accedere.

Un altro metodo si basa sul fatto che le transazioni acquisiscano un timestamp, e consiste nel consentire l'attesa di una transazione  $t_i$  su una risorsa acquisita da  $t_j$  solamente se vale una determinata relazione di precedenza fra i timestamp di  $t_i$  e  $t_j$  (per esempio,  $i < j$ ). In questo modo, circa il 50% delle richieste che generano un conflitto possono attendere in coda, mentre nel restante 50% dei casi una transazione deve essere uccisa. Per quanto riguarda la politica di scelta della transazione da uccidere vi sono diverse alternative. Distinguiamo prima di tutto le politiche in politiche interrompenti (*preemptive*) e non interrompenti. Una politica è interrompente se può risolvere il conflitto uccidendo la transazione che possiede la risorsa (in modo tale che questa rilasci la risorsa, che può così essere concessa all'altra transazione). In caso contrario, la politica è non interrompente, e una transazione può essere uccisa solo all'atto di fare una nuova richiesta. Una politica può essere quella di uccidere le transazioni che hanno fatto meno lavoro. Un problema di questa politica è che può capitare che una transazione faccia accesso, all'inizio della propria elaborazione, a un oggetto cui accedono molte altre transazioni. Può così capitare che la transazione trovi sempre un conflitto, ed essendo la transazione che ha fatto meno lavoro, venga uccisa ripetutamente. La situazione che si presenta è quella di un sistema senza blocchi critici, ma in cui vi sono delle transazioni in *blocco individuale* (*starvation*). Per risolvere questo problema è necessario garantire che ogni transazione non possa essere uccisa un numero illimitato di volte. Una soluzione che si adotta è quella di mantenere lo stesso timestamp quando una transazione viene fatta abortire e ripartire, dando nel contempo priorità crescente alle transazioni più "anziane". Questa tecnica non viene mai usata nei DBMS commerciali, in quanto mediamente si uccide una transazione ogni due conflitti, mentre la probabilità di insorgenza del blocco critico è di gran lunga inferiore alla probabilità di un conflitto.

Un'altra tecnica finalizzata alla prevenzione, recentemente proposta, si basa sull'osservazione che molti deadlock si verificano all'atto di "incrementare" un lock (lock upgrade), passando cioè da un lock in lettura ad un lock di scrittura. Questa situazione si verifica spesso durante l'esecuzione delle transazioni, quando sono previste inizialmente istruzioni di lettura, seguite dalla valutazione di condizioni (talvolta basate su input esterni), e quindi da istruzioni di modifica. Un deadlock si genera quando due transazioni acquisiscono lock condivisi sullo stesso dato e poi cercano entrambe di incrementare il lock, bloccandosi a vicenda. Per risolvere questo problema, è stato recentemente introdotto un nuovo tipo di lock, detto di *update*, che viene chiesto da una transazione all'atto di leggere un dato su cui intende successivamente scrivere. Questo lock è incompatibile con altri lock di update (quindi, in un certo istante, una sola transazione può avere un lock di update su un dato), ma è compatibile con un lock di lettura (quindi, vari lettori possono essere concorrenti ad un unico lettore che detiene il lock di update). All'atto di passare alla scrittura, la transazione deve comunque richiedere un lock di scrittura, e quindi resta in attesa che eventuali lettori terminino la loro azione. Però con questa modalità di locking si escludono due letture di un medesimo dato su cui due transazioni potrebbero voler poi scrivere, e quindi si esclude la condizione che portava a generare il deadlock descritto in precedenza.

**Rilevamento dei blocchi critici** Questa tecnica prevede di non porre vincoli al comportamento del sistema, ma di controllare il contenuto delle tabelle di lock, tutte le volte che si ritiene necessario, per rilevare eventuali situazioni di blocco. Il controllo può essere effettuato a intervalli prefissati, o quando scade un timeout di attesa di una transazione. Il rilevamento di un blocco critico richiede di analizzare le relazioni di attesa tra le varie transazioni e di determinare se esiste un ciclo. La ricerca di cicli in un grafo, specie se effettuata periodicamente, risulta abbastanza efficiente; per questo motivo, alcuni DBMS commerciali utilizzano questa tecnica, che verrà descritta in modo diffuso nel secondo volume, nel contesto dei sistemi distribuiti.

## Note bibliografiche

Molti riferimenti di interesse per questo capitolo coincidono con quelli già citati nel capitolo precedente. In particolare, il riferimento principale per l'organizzazione del capitolo è il monumentale libro *Transaction Processing Systems*, di Gray e Reuter [48]. Un testo più recente su tematiche simili e ugualmente di ampio respiro è quello di Weikum e Vossen [83]. Per una visione pragmatica, è interessante anche il testo di Bernstein e Newcomer [12]. Segnaliamo inoltre per il controllo di concorrenza la trattazione di Vossen [82] e, per una visione più formale di controllo di concorrenza e di affidabilità, quello di Bernstein, Hadzilacos e Goodman [11]. L'impostazione del controllo di affidabilità è tratta dal libro di Ceri e Pelagatti [23], con opportuni aggiornamenti. Il concetto di transazione introdotto in questo capitolo è stato recentemente esteso, introducendo modelli transazionali più complessi, tra cui le transazioni nidificate o di "lunga vita"; un buon riferimento è il libro edito da Elmagarmid [40].