Stato dell'Arte nel Controllo della Concorrenza [1]

T. Belli
C. Bernardeschi
A. Bondavalli
D. Latella

Rapporto Interno CNUCE C87-36

Consiglio Nazionale delle Ricerche
Istituto CNUCE - Pisa

---

1) Questo lavoro e' stato svolto nell'ambito di uno Study Contract con l'IBM

Indice

Sommario

In questo lavoro e` illustrato lo stato attuale dell'arte per quanto riguarda le tematiche connesse alla gestione degli accessi concorrenti a risorse condivise in sistemi transazionali. E` presentata una rassegna dettagliata delle principali tecniche di controllo della concorrenza proposte nella letteratura in cui si fa uso del costrutto di transazione.

Nel capitolo 1 e` condotta una breve analisi dei problemi legati all'introduzione della concorrenza in un sistema di elaborazione; e` inoltre presentato il concetto di transazione ed e` mostrata la sua utilita` in ambiente concorrente;

nel capitolo 2 e` introdotto il concetto di serializzabilita`, che e` stato considerato a lungo il criterio di correttezza di un'esecuzione concorrente di piu` transazioni;

nel capitolo 3 sono mostrati due approcci che assumono criteri di correttezza piu` deboli della serializzabilita` e fanno rispettivamente uso della semantica dei dati ([SC-SP '84]) e della semantica delle transazioni ([GAR '83]);

nel capitolo 4 sono analizzati in dettaglio i meccanismi per il controllo della concorrenza piu` usati: meccanismi di locking e meccanismi che fanno uso di timestamps;

nel capitolo 5, infine, e` mostrato un approccio al controllo della concorrenza alternativo ai precedenti, noto con il nome di approccio ottimistico.

1. Introduzione

La concorrenza e' stata introdotta nei sistemi di elaborazione per garantire una maggiore utilizzazione delle potenzialita' del sistema. Si dice **concorrente** un ambiente di computazione in cui piu' attivita' competono per l'uso di risorse condivise. Per **attivita'** si intendono processi, esecuzioni di programmi o, come vedremo in seguito, transazioni; per **risorse** si intendono sia risorse hardware: memorie, CPUs e periferiche; sia risorse software: dati e librerie di programmi. Se tutte le attivita'si svolgono in modo sequenziale, l'unica entita' attiva in un certo istante ha a disposizione tutte le risorse del sistema e puo' evolvere fino al termine con la massima velocita'. Il sistema e', pero', sottoutilizzato perche' in ogni istante l'entita' attiva usa solo una minima parte delle risorse che questo mette a disposizione. Inoltre in un ambiente sequenziale ogni attivita', prima di essere eseguita, deve attendere la terminazione di tutte le precedenti.

Ogni architettura di sistema puo' supportare la concorrenza se il sistema operativo e' progettato opportunamente; tuttavia, mentre in un'architettura uniprocessor il parallelismo e' simulato - si deve sospendere temporaneamente l'esecuzione di un programma per permettere ad un altro di proseguire nell'esecuzione - in architetture multiprocessor e di tipo rete, o nei multicomputer, il parallelismo puo' essere reale perche' su ogni processor puo' girare un programma diverso. Nelle architetture di tipo rete in cui sono messe a disposizione un numero elevato di risorse condivise e in cui piu' utenti si affacciano al sistema operando in maniera interattiva con esso, il parallelismo e quindi la concorrenza sono caratteristiche naturali dell'ambiente di calcolo.

La presenza della concorrenza non deve pero' provocare malfunzionamenti nell'esecuzione di programmi che in ambiente dedicato sarebbero corretti. Generalmente ogni programma, oltre ad operare su strutture dati private del suo ambiente locale, ha accesso ad un insieme di risorse condivise, che costituiscono l'ambiente globale. Se per risorse condivise si intendono solo le risorse hardware: il dispositivo di memoria principale, le CPUs, la consolle, le stampanti e i dispositivi di memoria secondaria, per garantirne un'utilizzazione corretta in regime concorrente e' sufficiente un meccanismo che implementi la **mutua esclusione** fra le attivita' per l'accesso ad ognuna di tali risorse ([BERN '81]). La **mutua esclusione**, come il termine stesso suggerisce, assicura che in ogni istante al piu' una attivita' abbia il diritto di accesso ad una certa risorsa. Se invece le risorse condivise sono anche i dati, memorizzati ad esempio in un database, allora la mutua esclusione non e' piu' sufficiente. Vediamo un esempio.

Consideriamo un sistema di automazione bancaria. Sia A1

un'attivita` che trasferisce una quota Q di denaro dal conto X al conto Y ed A2 un'attivita` che esegue il bilancio dei conti X e Y e restituisce il risultato all'utente. Con la mutua esclusione e` permessa una interazione tra le due attivita` in cui A1 toglie la quota Q dal conto X e, prima che A1 abbia aggiunto Q al conto Y, viene eseguita A2 che calcola il bilancio dei conti:

```
A1:  X := X-Q
A2:  BILANCIO := X+Y
A2:  output BILANCIO
A1:  Y := Y+Q
```

Possiamo osservare che il risultato restituito da A2 non si sarebbe mai avuto se le due attivita` fossero state eseguite una alla volta in assenza di concorrenza. Sui valori dei dati memorizzati in un database sono imposti dei vincoli legati alle particolari applicazioni. Ad esempio in una banca il denaro non deve mai andare perduto. C'e` un momento nell'esecuzione di A1 e A2 mostrata in cui tale vincolo non e` soddisfatto, infatti ad un certo istante la quota sottratta al conto X non e` ancora stata trasferita sul conto Y e poiche` A2 osserva il sistema proprio in questo momento, vede una situazione in cui la quota e` come se fosse andata perduta. Si puo` dunque affermare che la mutua esclusione non e` una condizione sufficiente a garantire interazioni corrette fra le attivita` di un sistema.

Vediamo, in dettaglio, i malfunzionamenti che possono sorgere in regime di concorrenza non controllata, come illustrato in [GRAY '78].

Perdita di aggiornamenti

Supponiamo di avere attivita` che leggono il valore dello stesso dato e aggiornano il dato in base al valore letto. Se due di tali attivita` sono eseguite concorrentemente, puo` accadere che entrambe leggano lo stesso valore ed eseguano poi le rispettive modifiche . Al termine delle due attivita` il risultato e` quello che si avrebbe avuto se fosse stata eseguita solo l'attivita` la cui modifica e` avvenuta per seconda.

Esempio:

```
A1: read(x)
    x := x+1
```

```
A2: read(x)
    x := x+2
```

Con l'esecuzione concorrente:

```
A1: read(x)
A2: read(x)
A1: x := x+1
A2: x := x+2
```

il risultato finale e` che il valore di x e` incrementato di 2 e non di 3 come sarebbe avvenuto in qualsiasi esecuzione seriale di A1 e A2; nell'esempio l'aggiornamento di A1 e` andato perduto.

C'e` un'altro caso in cui si puo` avere la perdita di un aggiornamento. Supponiamo che una attivita` A non possa evolvere fino al termine in seguito a qualche fallimento che si verifica durante l'esecuzione. E` ragionevole suppore che, per la maggior parte delle applicazioni, tali esecuzioni parziali non siano accettabili perche` insignificanti o addirittura possibile causa di errori futuri. Si puo` quindi pensare che in questi casi si faccia in modo da ripristinare lo stato precedente all'esecuzione di A e si tenti di rieseguire l'attivita` dopo aver eliminato la causa di fallimento. Se cosi` e` si puo` avere la perdita di aggiornamenti quando un'attivita` A2 legge ed aggiorna un dato precedentemente aggiornato da un'attivita` A1 che successivamente fallisce. Il fallimento di A1 e` gestito riportando il dato al valore che esso aveva prima dell'esecuzione di A1 con la conseguente perdita dell'aggiornamento di A2.

Esempio:

```
A1: read(x)                    A2: read(x)
    x := x+50                      x := x+1
```

Se il valore iniziale di x e` 100, con l'esecuzione concorrente:

```
A1: read(x)
A1: x := x+1
A2: read(x)
A2: x := x+50
A2: <termina>
A1: <fallisce>
```

il valore di x e` riportato al valore letto da A1, cioe` 100 e l'aggiornamento di A2 e` andato perduto.

Letture sporche

Supponiamo che l'attivita` A1 legga il dato x che, successivamente, un'altra attivita` A2 legga lo stesso dato, lo modifichi e termini. Se A1 esegue di nuovo una lettura di x vi trova un valore diverso da quello letto la prima volta: fra le due letture di A1 si e` interposta la scrittura di A2 e cio` puo` compromettere il risultato di A1. Infatti, affinche` il comportamento sia analogo a quello che si avrebbe in ambiente sequenziale, se un'attivita` legge piu` volte uno stesso dato i valori letti devono riflettere solo le modifiche che essa stessa ha effettuato.

Le applicazioni d'utente, in presenza di parallelismo, reale o simulato, devono tenere conto della possibilita` di interferenze da parte di attivita` concorrenti e quindi devono prevedere gli strumenti per evitare che cio` porti ai malfunzionamenti visti. Per sollevare l'utente da questo compito e` stato introdotto il costrutto di transazione. In un sistema che fornisce le transazioni ogni utente puo` sviluppare le proprie applicazioni e lanciarle come transazioni senza doversi

La semantica informale dei comandi e` la seguente:

begin-transaction: segna l'inizio della transazione. Quando una transazione inizia le viene comunemente assegnato un identificatore unico che la distingue da tutte le altre transazioni nel sistema.

abort-transaction: con questo comando l'utente puo` interrompere l'esecuzione della transazione che si dice abortita. L'aborto di una transazione puo` essere provocato esplicitamente dall`utente o implicitamente dal sistema, quando viene rilevata una situazione critica in cui la transazione non puo` essere eseguita fino al termine mantenendo la consistenza. Per la proprieta` di atomicita`, in entrambi i casi, e` necessario annullare gli effetti dell'esecuzione parziale. Dopo che una transazione e` stata abortita si puo` richiedere di nuovo la sua esecuzione con un comando di reinizializzazione (restart)

end-transaction: questo comando corrisponde a richiedere l'esecuzione dell'operazione di commit. Quando una transazione giunge al termine dopo avere seguito con successo tutte le sue operazioni si dice che "committa". Committare equivale a rendere effettive nel sistema le eventuali modifiche apportate allo stato. Per garantire che la transazione sia atomica e` necesşario che tutte le modifiche siano rese visibili e permanenti simultaneamente, altrimenti il sistema puo` transitare in uno stato inconsistente. Da questo si deduce che l'operazione di commit deve essere eseguita come un'azione atomica e se non e` possibile portarla a termine con successo, per la presenza di guasti, la transazione non puo` committare e il sistema ne deve forzare l`aborto.

Ogni transazione risulta dunque strutturata in due fasi:

- una prima fase in cui vengono eseguite le operazioni che costituiscono l'intera transazione (compresa l'eventuale gestione di comandi di abort);

- una seconda fase, nota con il nome di commitment, in cui la transazione richiede al sistema di rendere visibili e permanenti la modifiche operate.

In un sistema transazionale distribuito si devono prevedere sia transazioni eseguibili su un singolo nodo sia transazioni distribuite su piu` nodi. Queste ultime eseguono in modo autonomo azioni atomiche sui vari nodi, utilizzando le risorse locali del nodo. Tali transazioni sfruttano al massimo le potenzialita` di un sistema distribuito ma complicano l'esecuzione di alcune operazioni tra cui quella di commit. Le esecuzioni sui vari nodi, infatti, sebbene di solito possano procedere indipendentemente, almeno nella fase di commitment si devono sincronizzare per garantire l'atomicita` della computazione complessiva. Si deve determinare un accordo tra i nodi partecipanti all'esecuzione della transazione in modo da evitare che solo una parte delle modifiche venga riportata sullo stato del sistema. Se su almeno

un nodo e` richiesto l'aborto della transazione tutti i partecipanti devono mettersi d'accordo per eseguire l'aborto della transazione nell`intero sistema distribuito. Per ottenere questo comportamento in [GRAY '78] e` riportato un protocollo, il Two-Phase-Commit, che coordina la fase di terminazione di una transazione sui vari nodi interessati attraverso lo scambio di particolari messaggi.

Diamo adesso una schematizzazione dell'architettura del sistema, proposta in [BERN '81], a cui d'ora in avanti faremo riferimento. Non considereremo un sistema general-purpose, ma ci restringeremo al caso piu` tradizionale di un sistema per la gestione di basi di dati (D.B.M.S.). Il nostro D.B.M.S. puo` risiedere su un'unica macchina (sistema centralizzato) ma, nel caso piu` generale, e` un sistema distribuito su un insieme di calcolatori collegati da una rete di comunicazione e, in entrambi i casi, e` un sistema multiutente.

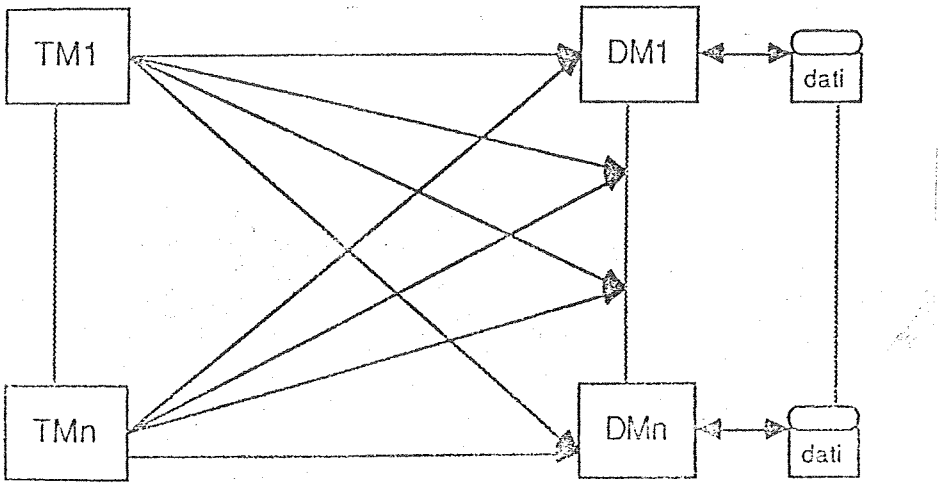
La rete e` supposta completamente affidabile, cioe` un messaggio inviato dal nodo A al nodo B giunge in B entro un tempo finito e non affetto da errori. Generalmente su ogni nodo della rete risiedono due moduli software: un Transaction Manager (TM), gestore delle transazioni, e un Data Manager (DM), gestore dei dati, ma in alcuni casi uno dei due moduli puo` mancare.

Il TM supervisiona l'esecuzione delle transazioni e controlla le interazioni con gli utenti. Tra le sue funzionalita` c'e` quella di forzare l'aborto delle transazioni in seguito a guasti hardware o software, quella di gestire l'operazione d'aborto, con l'eventuale reinizializzazione della transazione, e quella di gestire il commitment. Il TM si preoccupa anche di assegnare gli identificatori alle transazioni quando vengono create.

Il DM e` il vero e proprio gestore dei dati, cioe` colui che esegue sui dati tutte le operazioni previste dal sistema e regola l'accesso delle transazioni ai dati mantenuti sul nodo locale.

Riassumendo il sistema appare costituito da quattro componenti: le transazioni, i TMs, i DMs e i dati memorizzati. Gli utenti interagiscono con il database solo eseguendo transazioni. La richiesta di eseguire una transazione e` inviata ad un TM ed ogni transazione e` supervisionata da un unico TM. I TMs comunicano con i vari DMs richiedendo che vengano eseguite delle operazioni sui dati e ricevendo poi i risultati. I DMs sono l'unica componente ad avere accesso ai dati.

Il modello di interazione e` dunque a Cliente-Servente: i TMs fungono da cliente per conto delle transazioni d'utente e i DMs svolgono la funzione di data-server. In figura e` illustrato lo schema del sistema:



L'implementezione del sistema transazionale deve garantire il mantenimento delle proprieta` delle transazioni. Come vedremo cio` comporta l'adozione di una politica per la gestione degli accessi concorrenti ai dati condivisi che vincoli l'esecuzione delle operazioni da parte delle transazioni.

Il problema del controllo della concorrenza e` stato ampiamente studiato soprattutto nell'ambito dei sistemi per la gestione di basi di dati. Nei capitoli successivi e` presentata una rassegna dei risultati piu` interessanti ottenuti in questi studi.

2. Esecuzioni concorrenti e correttezza

Dopo aver caratterizzato il costrutto di transazione passiamo ad analizzare i problemi che sorgono quando piu` transazioni sono eseguite concorrentemente. L'obiettivo del controllo di concorrenza e` di permettere esecuzioni con il massimo grado di concorrenza, continuando a dare ad ogni transazione la visione di uno stato consistente del sistema.

Sia T = {t1...tm} un insieme di transazioni. Come abbiamo visto ([GRAY '80]) ogni transazione ti e` una sequenza di azioni:

$$ti = \langle \langle ni, aj, ej \rangle \ / \ j = 1..n \rangle$$

dove ni e` il nome della transazione, aj sono le operazioni ed ej sono gli oggetti su cui l'azione aj e` eseguita.

Un'esecuzione di T e` definita formalmente come una sequenza di tutte le azioni di tutti i membri di T tale che sia preservato l'ordinamento delle azioni all'interno di ogni ti di T. Nella letteratura il termine esecuzione e' spesso sostituito con quello

di schedule.

Uno schedule S e` seriale se ogni transazione e` eseguita fino al termine prima che ne inizi un'altra. In uno schedule seriale le transazioni sono eseguite una alla volta, dunque non c'e` concorrenza. Uno schedule non e` seriale se le azioni delle transazioni sono "interleaved", ovvero, se le azioni di una transazione ti sono intercalate con alcune azioni di altre transazioni, tj ... tk di T, pur preservando l'ordinamento tra le azioni di ogni singola transazione. In questo caso si parla di esecuzioni interleaved di transazioni. Cio` che distingue uno schedule da un'altro e` l'ordine di esecuzione relativo tra operazioni di transazioni distinte.

Non tutti gli schedules non seriali sono da ritenersi accettabili; taluni, infatti, possono produrre transizioni di stato del sistema non desiderabili. Gli schedules accettabili sono solo quelli corretti o consistenti. La definizione di correttezza per gli schedules non e` unica e nella letteratura se ne possono trovare varie versioni che rappresentano, l'una rispetto all'altra, un' evoluzione di questo concetto.

La prima definizione di consistenza la troviamo in [ESW '76]. E` detto consistente uno schedule che da` ad ogni transazione la visione di uno stato consistente. In generale due schedules consistenti possono restituire all'utente risultati differenti e far transitare il sistema in stati diversi. Quindi vincolare le esecuzioni concorrenti di un insieme T di transazioni agli schedules consistenti non impone alle transazioni un comportamento deterministico.

Per gli schedules seriali la proprieta` di consistenza deriva direttamente dalla proprieta` di consistenza delle transazioni. Infatti, se uno schedule seriale inizia in uno stato consistente e ogni transazione, eseguita singolarmente, trasforma uno stato corretto in uno ancora corretto, tutte le transazioni hanno una visione consistente dello stato.

Ogni schedule S introduce una relazione di dipendenza tra transazioni, indicata con DEP(S), definita su TXEXT, dove E e` l'insieme delle entita` del database e T e` l'insieme di tutti i nomi delle transazioni in S.

Definizione:

(t1, e, t2) $\in$ DEP(S) se e solo se, esistono i, j con i<j, tali che:

S = (.. , (t1, ai, e), ... , (t2, aj, e)..)

e non esiste k tale che i<k<j con ek = e.

Ovvero (t1, e, t2) $\in$ DEP(S) se in S t1 opera su un'entita` su cui successivamente opera t2 e nessuna transazione, nel frattempo, ha operato sulla stessa entita`. In [ESW '76] la

relazione di dipendenza e` usata per esprimere formalmente la nozione di equivalenza fra schedules.

Definizione:

Due schedules S1 e S2 sono equivalenti se:

DEP(S1) = DEP(S2).

La definizione di consistenza a cui si giunge e`la seguente:

Definizione:

Uno schedule S e` consistente se ha uno schedule seriale equivalente.

In [ESW '76], dunque, uno schedule S e` considerato consistente se e` seriale o se induce una relazione di dipendenza fra transazioni uguale a quella indotta da almeno uno schedule seriale.

Passiamo ora ad esaminare l'approccio descritto in [BERN '81]. Innanzitutto, supponendo che le singole transazioni garantiscano la consistenza se eseguite singolarmente, e` definito corretto uno schedule seriale. E` poi introdotto il concetto di equivalenza computazionale.

Definizione:

Due shedule S1 e S2 si dicono computazionalmente equivalenti se producono gli stessi risultati ed hanno gli stessi effetti sul database.

Le operazioni sono suddivise in due tipi: quelle di lettura (Read) e quelle di scrittura (Write). Con riferimento alla suddivisione delle operazioni in Read e Write, e` fornita una definizione alternativa dell'equivalenza:

Definizione:

S1 e S2 sono schedule computazionalmente equivalenti se:

1) in entrambi gli schedule ogni Read legge il valore scritto dalla stessa Write;
2) la Write finale su ogni dato e` la stessa sia in S1 che in S2.

Definizione:

Uno schedule S e` serializzabile se e` computazionalmente equivalente ad uno seriale.

Poiche` ogni schedule seriale e` corretto ed ogni schedule serializzabile e` computazionalmente equivalente ad uno seriale, anche ogni schedule serializzabile e` corretto. Se il sistema

permette solo schedules serializzabili qualunque esecuzione concorrente di piu' transazioni risulta corretta. Diamo ancora una definizione:

Definizione ([BERN '81]):

Due operazioni sono in conflitto se:
- sono eseguite da transazioni diverse,
- operano sullo stesso dato e
- almeno una di esse e' un'operazione di scrittura.

Secondo la definizione si possono distinguere tre tipi di conflitto: R-W (lettura-scrittura), W-R (scrittura-lettura) e W-W (scrittura-scrittura); mentre non sono tra loro in conflitto due operazioni di lettura. Si puo' facilmente osservare che l'ordine con cui sono eseguite le operazioni e' significativo se e solo se esse sono in conflitto. Se t1 esegue una lettura di x e t2 una scrittura di x , il valore letto da t1 e' diverso a seconda dell'ordine di esecuzione delle due operazioni: se viene eseguita prima la lettura si ha un conflitto di tipo R-W, altrimenti il conflitto e' di tipo W-R. Se entrambe t1 e t2 eseguono una scrittura di x il valore finale del dato dipende da quale delle due scritture e' stata eseguita per ultima: si ha un conflitto di tipo W-W. Se invece t1 e t2 eseguono entrambe una lettura di x, ne' t1, ne' t2, ne' nessun'altra transazione puo' essere in grado di distinguere quale delle due letture e' avventa per prima e quale per seconda; e' per questo che due letture non sono considerate operazioni in conflitto. Vediamo con due esempi come solo l'ordine di esecuzione di operazioni in conflitto sia rilevante al fine di ottenere schedule serializzabili.

Esempio

Siano t1 e t2 due transazioni cosi' definite:

```
t1: read(x)          t2: read(x)
    write(y)             write(x)
```

Consideriamo la seguente esecuzione concorrente:

schedule S1:

```
t1: read(x)
t2: read(x)
t1: write(y)
t2: write(x)
```

Tale schedule e' equivalente allo schedule S2 in cui viene eseguita prima t1 e poi t2:

schedule S2:

```
t1: read(x)
t1: write(y)
t2: read(x)
t2: write(x)
```

Lo schedule S1 e` dunque serializzabile, e, per le definizioni introdotte, e` corretto. Si puo` passare dallo schedule S1 allo schedule S2 perche` l'operazione write(y) di t1 non e` in conflitto con nessuna operazione di t2 e si puo` quindi cambiare l'ordine di esecuzione delle operazioni senza che si modifichino gli effetti sul database o i risultati restituiti.

Esempio

Consideriamo ora il caso in cui t1 e t2 leggono ed aggiornano entrambe il dato x:

```
t1 = t2: read(x)
         write(x)
```

e lo schedule S1:

```
t1: read(x)
t2: read(x)
t1: write(x)
t2: write(x)
```

Lo schedule S1 non e` serializzabile perche` non e` possibile trovare uno schedule seriale equivalente ad S1. Infatti in questo caso l'operazione write(x) di t1 e` in conflitto con le operazioni di t2 ed essendo significativo l'ordine di esecuzione non e` possibile scambiare nello schedule l'operazione read(x) di t2 e l'operazione write(x) di t1 ed ottenere uno schedule computazionalmente equivalente.

In [BERN '81] le definizioni sono estese per coprire il caso di sistemi distribuiti in cui l'esecuzione concorrente delle transazioni e` suddivisa nelle computazioni locali di ogni nodo che evolvono tra loro in parallelo.

Pur usando una diversa terminologia, Bernstein ed Eswaran-Gray giungono alla formulazione della stessa condizione, la serializzabilita`, per la correttezza\consistenza degli schedules. L'equivalenza computazionale di Bernstein infatti e` l'analogo dell'equivalenza di Eswaran-Gray se, in quest'ultimo, si considerano tutte le dipendenze escluse quelle in cui entrambe le operazioni sono di lettura. In entrambi i modelli si rileva la rinuncia a ricercare una condizione per la correttezza che non sia solo sufficiente ma anche necessaria. In seguito con altri approcci si e` dimostrato che si puo` ottenere un grado maggiore di concorrenza imponendo agli schedules condizioni meno stringenti della serializzabilita`. Il primo passo in questa direzione e` consistito nel formulare una diversa definizione di

correttezza. Si e` rinunciato ad esprimere la correttezza degli schedules basandosi sulla proprieta` di consistenza delle singole transazioni e si e` invece concentrata l'attenzione sulle proprieta` dell'intera computazione concorrente. Una proposta di definizione di correttezza e` la seguente:

Definizione ([GAR '83]):

Uno schedule e` corretto se:

1) i valori restituiti all'utente dalle transazioni rappresentano stati consistenti del sistema;
2) il risultato finale e` quello di lasciare il sistema in uno stato consistente.

Secondo la definizione, quando i vincoli dell'applicazione lo permettono, le transazioni possono vedere stati inconsistenti purche` cio` non influenzi negativamente il loro risultato finale.

Se il sistema transazionale garantisce la proprieta` di serializzabilita` degli schedules, siamo certi che l'esecuzione concorrente delle transazioni e` corretta perche` le due condizioni della definizione sono verificate. Alcune volte pero` il mantenimento della serializzabilita` porta a limitare inutilmente la concorrenza facendo si che esecuzioni corrette non siano permesse.

Vi sono infatti applicazioni in cui lo stato del sistema continua a rispettare i vincoli di consistenza imposti anche permettendo schedules non serializzabili, vediamone un caso ([SC-SP '84]). Consideriamo un buffer circolare per lo scambio di messaggi tra un insieme di produttori e un insieme di consumatori. L'ordine esatto di inserzione dei messaggi nel buffer non ha importanza, cio` che e` necessario garantire e` che i messaggi non restino nel buffer per sempre, ovvero che messaggi inseriti all'incirca nello stesso momento abbiano tra loro un comportamento "fair".

Consideriamo due transazioni t1 e t2 tali che t1 inserisce nel buffer B i messaggi m1 e m2 e t2 inserisce nel buffer il messaggio m':

```
t1: insert ( B, m1 )
    insert ( B, m2 )

t2: insert ( B, m' )
```

Il seguente schedule e` corretto, secondo la definizione di correttezza precedente, perche` rispetta i vincoli dell'applicazione in esame:

```
S:  t1: insert ( B, m1 )
    t2: insert ( B, m' )
    t1: insert ( B, m2 )
```

Lo schedule S non e' serializzabile perche' non esiste alcuno schedule seriale il cui effetto sia quello di far apparire nel buffer i messaggi di t1 e t2 mescolati fra loro: o tutti quelli di t1 precedono tutti quelli di t2 o viceversa. Lo schedule S quindi non sarebbe permesso in un sistema in cui si garantisca la serializzabilita' anche se e' perfettamente accettabile per le esigenze dell'applicazione. L'esempio mostra chiaramente che la serializzabilita' non deve essere assunta come condizione necessaria per la correttezza.

Nel capitolo successivo vedremo come sia possibile formulare condizioni di correttezza meno stringenti della serializzabilita'.

3. Uso della semantica per incrementare la concorrenza

Al termine del capitolo precedente abbiamo visto con un esempio che la serializzabilita' non e' una condizione necessaria per la correttezza. Abbiamo sfruttato la conoscenza semantica della struttura dati buffer circolare, e le caratteristiche dell'applicazione di scambio messaggi fra processi produttori e consumatori, per mostrare che possono considerarsi corretti anche schedules non serializzabili. E' importante notare che si e' raggiunto questo risultato studiando non un contesto astratto ma l'esecuzione concorrente di transazioni nell'ambito di una particolare applicazione di cui possedevamo una conoscenza semantica specifica.

Si intuisce che la conoscenza semantica puo' essere sfruttata per cercare di imporre agli schedules concorrenti i vincoli strettamente necessari al mantenimento della correttezza nell'ambito dell'applicazione in esame. Questi ultimi saranno gli stessi o un sottoinsieme di quelli necessari per garantire la serializzabilita' ed e' facile comprendere che, se e' minore il numero dei vincoli imposti agli schedules e' maggiore il grado di concorrenza ottenibile. Nella direzione dell'uso della semantica si collocano i lavori di Garcia-Molina e di Schwarz e Spector.

I due approcci si differenziano per la metodologia usata nello studio delle applicazioni. Per ricavare l'insieme di vincoli necessari per la correttezza Garcia-Molina muove dallo studio della semantica delle transazioni previste nel sistema, mentre Spector e Schwarz si concentrano sullo studio della semantica dei tipi di dato usati. Nei paragrafi 3.1 e 3.2 mostreremo i due approcci in maggior dettaglio.

3.1. Semantica delle transazioni

H.Garcia-Molina ([GAR '83]) muove dallo studio della semantica delle transazioni previste nel sistema per risolvere in modo efficiente il controllo della concorrenza in un database

distribuito. Cio' che del suo lavoro ci interessa non e' tanto il meccanismo di controllo della concorrenza proposto, un protocollo di locking particolarmente indirizzato ai sistemi distribuiti, quanto l'approccio nuovo al controllo della concorrenza. Il criterio tradizionale di correttezza, la serializzabilita', e' sostituito con quello di consistenza semantica.

Definizione:

Una transazione t e' detta sensitiva se:

1) t restituisce dei dati che sono visti dall'utente;
2) i dati restituiti da t devono rappresentare uno stato consistente del database.

Non tutte le transazioni che restituiscono dati all'utente sono sensitive poiche' gli utenti possono accettare di vedere delle inconsistenze quando queste sono ininfluenti per l'uso che essi fanno dei dati. Ad esempio una transazione che restituisce la media dei salari di tutti gli impiegati di una ditta puo' fornire un risultato accettabile anche se sono presenti delle inconsistenze nei valori di alcuni salari.

Definizione:

Sia S uno schedule di un insieme di transazioni T, alcune delle quali possono essere sensitive; S e' uno schedule semanticamente consistente se e solo se:

1) l'esecuzione di S trasforma lo stato iniziale del database, supposto consistente, in uno stato anch'esso consistente;
2) tutte le transazioni sensitive vedono uno stato consistente del database.

La consistenza semantica e' praticamente analoga alla definizione di correttezza che abbiamo dato alla fine del paragrafo 2.1 con l'unica differenza che, in quel caso, le transazioni erano considerate tutte sensitive. Non e' banale decidere se uno schedule non serializzabile e' semanticamente consistente. Per fare questo e' necessaria un'approfondita conoscenza semantica dell'applicazione e delle possibili transazioni. Infatti solo se si e' in grado di caratterizzare il comportamento concorrente delle transazioni e' possibile decidere quali interazioni mantengono la consistenza semantica e quali la violano.

Una volta stabilita l'utilita' di far uso della semantica delle transazioni, resta da vedere il modo in cui sfruttare questa conoscenza per ottenere un meccanismo di controllo della concorrenza efficiente.

L'idea base del metodo e' quella di suddividere le possibili transazioni che si hanno nell'applicazione in esame in tipi semantici. Il tipo della transazione definisce il suo

comportamento nella computazione concorrente. Un tipo di transazione e' caratterizzato da certi fattori tipici come l'insieme degli oggetti acceduti, il tipo delle operazioni eseguite sugli oggetti e il risultato che si attende dalle operazioni, se esatto o approssimato. E' l'utente stesso che deve definire i tipi delle transazioni. Naturalmente il metodo e' applicabile solo se tutte le transazioni del sistema si possono far ricadere in uno dei tipi previsti.

Ad ogni tipo di transazione e' associato un insieme di compatibilita' che descrive come le transazioni di quel tipo possono essere eseguite in modo concorrente ("interleaved") con altre transazioni senza violare la consistenza semantica. Ad esempio possono essere eseguite in modo interleaved due tipi che accedono insiemi distinti di oggetti oppure che accedono eventuali oggetti a comune solo per eseguirvi operazioni non in conflitto. Ogni insieme di compatibilita' e' formato da descrittori di interleaving, ognuno dei quali descrive un tipo di "interleaving" ammissibile fra transazioni. Supponiamo di avere n tipi di transazioni y1,...,yn e che l'insieme di compatibilita' di y1 contenga i descrittori { y1, y2 } e { y1, y3, y5 }. Questo significa che le transazioni di tipo y1 possono essere eseguite "interleaved" con transazioni dei tipi y2, y3 e y5, ma le transazioni di tipo y2 non possono essere "interleaved" con quelle di tipo y3 e y5, altrimenti avremmo avuto un unico descrittore di interleaving ({y1, y2, y3, y5}). Se le transazioni di tipo y1 dovessero essere eseguite in assenza di concorrenza l'insieme di compatibilita' per il tipo y1 sarebbe vuoto. Anche gli insiemi di compatibilita', come i tipi, sono definiti dall'utente.

Il sistema transazionale proposto rende disponibile un meccanismo di controllo della concorrenza che, prendendo in ingresso la definizione dei tipi e degli insiemi di compatibilita', esegue la sincronizzazione delle transazioni tramite una forma particolare di locking (si veda il capitolo 4 per la descrizione dei meccanismi).

Nel suo lavoro Garcia-Molina non fornisce pero' nessuna metodologia per studiare l'ambiente concorrente basandosi sulla semantica delle transazioni, che pure e' il perno del metodo. La definizione dei tipi delle transazioni e la specifica del loro insieme di compatibilita' e' lasciata completamente al buon senso dell'utente. Il meccanismo di controllo della concorrenza che si ottiene sara' dunque fortemente influenzato dalla bonta' delle specifiche e da questa dipendera' anche la correttezza del suo comportamennto. In ogni caso la correttezza risultera' difficile da provare essendo basata solo sull'idea intuitiva che l'utente ha dell'applicazione.

3.2. Transazioni e tipi di dato astratti

Nel modello di sistema transazionale che abbiamo presentato, ogni transazione appare come una sequenza di operazioni di lettura/scrittura e gode delle proprieta' di consistenza, atomicita', invisibilita' e permanenza. Inoltre un'esecuzione concorrente di un insieme qualsiasi di transazioni e' considerata corretta se e' serializzabile. Il modello e' stato introdotto per trattare in modo uniforme i fallimenti e il controllo della concorrenza in un sistema per la gestione di basi di dati. Schwarz e Spector hanno studiato come estenderlo per affrontare ancora la gestione dei fallimenti e della concorrenza ma nell'ambito piu' ampio dei sistemi general-purpose ([SP-SC '83], [SC-SP '84]).

L'idea innovativa e' quella di adottare l'astrazione dei dati e di immetterla nel modello di transazione.

Se adottiamo l'astrazione dei dati, l'informazione risiede in oggetti che sono istanze di tipi astratti e puo' essere manipolata solo con le operazioni caratteristiche del tipo dell'oggetto. La specifica del tipo astratto consiste nella definizione delle operazioni che si possono richiedere su oggetti di quel tipo e nella descrizione dei loro effetti. I dettagli che riguardano il modo in cui gli oggetti di un particolare tipo sono rappresentati e come vengono eseguite le operazioni sono noti soltanto all'implementatore. Ogni tipo astratto gia' definito puo' venire usato come componente primitivo nella definizione di tipi piu' complessi e poiche' l'astrazione nasconde la complessita' della loro implementazione, si possono costruire astrazioni successive di livello sempre piu' alto che sono ancora semplici da comprendere. Ricordiamo che l'astrazione dei dati e' uno dei metodi piu' usati per affrontare il problema della complessita' del software, perche' permette che sistemi complicati e di grosse dimensioni possano venire compresi in termini di componenti piu' semplici.

Nel modello di sistema transazionale proposto i dati sono istanze di tipi di dato astratti e le transazioni sono viste come "sequenze di operazioni su istanze di tipi di dato astratti" . Le operazioni, a loro volta, possono essere implementate come transazioni su tipi di dato astratti piu' semplici. Un modello di transazione che incorpora l'astrazione dei dati deve infatti permettere che gli oggetti gia' definiti vengano aggregati in strutture arbitrarie per definire nuovi tipi che appaiano ai loro utenti come tipi primitivi con le operazioni appropriate.

Sono ancora proprieta' delle transazioni l'atomicita' e la permanenza; non e' invece considerata la proprieta' di invisibilita' e quindi e' permesso che una transazione osservi i risultati di altre transazioni non ancora terminate. L'incoveniente e che se t accede ad un dato modificato da t' e successivamente t' fallisce, anche t deve essere abortita. Questo fenomeno e' detto aborto in cascata. Nel modello proposto e' assunta come proprieta' delle transazioni anche quella di non

provocare mai aborto in cascata: nessuna transazione deve essere abortita a causa dell'aborto di un'altra.

Gli oggetti su cui le transazioni operano concorrentemente sono definiti come tipi di dato astratti condivisi. La specifica di un tipo astratto condiviso comprende la specifica del tipo astratto piu` la descrizione di come le operazioni del tipo di dato possono essere eseguite concorrentemente da transazioni distinte mantenendo la correttezza degli schedules. In altri termini, la condivisione del dato e la sincronizzazione necessaria per la gestione degli accessi concorrenti e` parte integrante della definizione del tipo di dato stesso.

Il criterio di correttezza per le esecuzioni di transazioni che richiedono operazioni su un tipo di dato e` dettato dall'insieme dei vincoli di consistenza definiti sul tipo di dato stesso:

- uno schedule e` corretto se, applicato ad uno stato dei dati che soddisfa i vincoli di consistenza, fa transitare il sistema in uno stato che soddisfa ancora tutti i vincoli.

L'insieme dei vincoli di consistenza per un tipo di dato astratto condiviso include anche vincoli sul comportamento concorrente atteso per transazioni che operano su quel tipo. Ad esempio si puo` richiedere che ogni transazione veda un oggetto di quel tipo come se fosse l'unica ad operarvi.

Considerare l'informazione memorizzata in tipi di dato astratti, anziche` vederla come un insieme di records, permette di sfruttare la conoscenza che si ha sulla semantica delle operazioni del tipo di dato. Il criterio di correttezza degli schedules puo` coincidere con la serializzabilita`, ma in generale e` meno restritivo della serilizzabilita`.

Ad esempio il criterio di correttezza puo` coincidere con la serializzabilita` astratta. Vediamo qual'e` la differenza tra le due proprieta` ([SCH '84]). Supponiamo di avere il tipo di dato Coda FIFO e due transazioni tali che una inserisce k elementi e l'altra toglie h elementi dalla stessa coda. Supponiamo inoltre che la coda non sia vuota. Se si permette che le operazioni di inserzione e di rimozione da parte delle due transazioni siano eseguite in modo alternato, prima una inserzione poi una rimozione poi ancora una inserzione e cosi` via fino al termine delle operazioni, non si ha vera e propria serializzabilita` (serializzabilita` concreta). Infatti lo stato della coda in cui viene inserito ad esempio il secondo elemento e` diverso da quello che c'era al termine della prima inserzione. Tuttavia il comportamento e` serializzabile da un punto di vista astratto infatti le transazioni non sono in grado di cogliere questa differenza di stato e il loro risultato, cosi` come il risultato delle transazioni successive, non ne e` in alcun modo influenzato. Poiche` la serializzabilita` astratta impone un minor numero di restrizioni agli schedules di quante se ne devono imporre se si vuole garantire la serializzabilita` concreta,

sara' possibile ottenere un maggior grado di concorrenza. Grazie alla conoscenza semantica che abbiamo del tipo di dato Coda FIFO e delle sue operazioni si puo' in questo caso scegliere come criterio di correttezza la serializzabilita' astratta.

Inoltre esistono applicazioni in cui la correttezza non coincide ne' con la serializzabilita' concreta ne' con la serializzabilita' astratta.

Ad esempio, il tipo di dato MAILBOX, cassetta della posta, permette interazioni non serializzabili tra transazioni. La semantica del dato non impone che due lettere inviate insieme alla stessa destinazione appaiano contigue nella cassetta del ricevente. Tuttavia la serializzabilita' e' violata se le lettere non arrivano contiguamente infatti lo stato non e' quello che si avrebbe se il mittente avesse inviato le lettere senza interferenza da parte di altri mittenti.

La proprieta' di correttezza degli schedules, ovvero il comportamento concorrente che si vuole ottenere tra le transazioni, deve essere tradotto dall'implementatore nella specifica della sincronizzazione del tipo astratto condiviso. Nel capitolo 1 abbiamo visto che gli schedules descrivono l'ordine in cui le operazioni di un insieme T di transazioni sono eseguite sugli oggetti e che esaminando uno schedule e' possibile determinare quali dipendenze esistono tra le transazioni nello schedule. Le dipendenze forniscono un ottimo mezzo per ragionare sul comportamento concorrente delle transazioni.

Definizione ([SC-SP '84]):

Se una transazione ti esegue l'operazione x sull'oggetto o e successivamente tj esegue un'operazione y sullo stesso oggetto, si dice che fra ti e tj si forma una dipendenza D ed e' indicata con:

D: ti:x -o-> tj:y.

Le operazioni x e y sono due di tipo qualsiasi fra quelle che si possono richiedere su oggetti dello stesso tipo di o. L'insieme delle coppie ordinate {(ti, tj)} per le quali esistono x, y, o tali che ti e tj sono legate dalla dipendenza D forma una relazione indicata con <D . Se ti <D tj allora diciamo che ti precede tj, oppure che tj dipende da ti, secondo la relazione di dipendenza <D. Notiamo che le dipendenze si formano su uno specifico oggetto ma la relazione di dipendenza (<D) comprende tutti gli oggetti su cui le dipendenze occorrono. Indichiamo con <D* la chiusura transitiva della relazione <D.

Definizione:

Uno schedule S e' ordinabile rispetto alla relazione di dipendenza <D se e solo se <D* e' un ordinamento parziale in S, in altri termini se e solo se non esistono cicli della forma:
ti <D tj <D ..... <D ti

Definizione:

Uno schedule S e' ordinabile rispetto ad un insieme I di relazioni di dipendenza se e solo se ogni relazione in I ha una chiusura transitiva che e' un ordinamento parziale in S.

Poiche' le dipendenze sono caratterizzate dal tipo delle operazioni eseguite dalle transazioni, se non abbiamo informazioni semantiche sulle possibili operazioni, avremo un'unica dipendenza. Indichiamo tale dipendenza con D':

D': ti:op -o-> tj:op

Se tutte le transazioni sono consistenti l'ordinabilita' rispetto a <D' garantisce la consistenza degli schedules.

Consideriamo l'esempio classico del database con le operazioni di lettura/scrittura. In questo caso ci sono quattro possibili dipendenze:

D1: ti:R -o-> tj:R (R=lettura)
D2: ti:R -o-> tj:W (W=scrittura)
D3: ti:W -o-> tj:R
D4: ti:W -o-> tj:W

La dipendenza D', nel caso non si abbiano informazioni sulle operazioni, implicitamente riunisce le varie dipendenze ora individuate, cioe' D'= D1UD2UD3UD4. Prendiamo in esame la dipendenza D1. Se ti precede tj secondo la dipendenza D1 vuol dire che tj legge un dato letto precedentemente da ti. Le transazioni ti e tj sono pero' insensibili al fatto di eseguire la lettura di un dato una prima dell'altra infatti ne' ti, ne' tj, ne' nessun'altra transazione puo' determinare se ti <D1 tj o tj <D1 ti. Abbiamo dunque individuato una dipendenza particolare per la quale si possono permettere cicli nella chiusura transitiva della relazione di dipendenza corrispondente senza influire sulle proprieta' degli schedules che si possono ottenere. La condizione di serializzabilita' puo' essere ristretta all'ordinabilita' rispetto alla relazione di dipendenza <D2UD3UD4.

D'altra parte avevamo gia' classificato due operazioni di lettura come operazioni non in conflitto e notato che il loro ordine di esecuzione non e' importante ai fini della serializzabilita' degli schedules. La proprieta' di ordinabilita' rispetto alla relazione di dipendenza che comprende tutte le dipendenze per un certo tipo di dato escluse quelle di tipo D2 e' abbastanza forte da garantire la serializzabilita'.

E' evidente che nella specifica di come possono interagire le operazioni sugli oggetti condivisi, il grado di concorrenza dipende in parte da quanto e' dettagliata la conoscenza semantica sulle operazioni. Il controllo della concorrenza quando e' considerata la suddivisione delle operazioni in letture e scritture permette maggiore concorrenza di quando si ha un solo

tipo di operazione. Esistono situazioni in cui e` possibile aumentare ancora il grado di concorrenza senza compromettere la serializzabilita`, avvalendosi di una maggiore conoscenza semantica sulle operazioni che si possono eseguire. Prendiamo ad esempio il tipo di dato array, se non consideriamo la semantica delle operazioni eseguibili su un array due operazioni di modifica di entrate distinte dell'array, essendo scritture dello stesso oggetto, devono essere sincronizzate. La semantica del tipo di dato ci assicura pero` che quelle non sono operazioni in conflitto e quindi non necessitano di sincronizzazione.

Definiti i vincoli di consistenza per il tipo di dato una dipendenza D e` detta importante se e` necessario garantire l'ordinabilita` degli schedules rispetto alla relazione di dipendenza <D per avere esecuzioni concorrenti che rispettino i vincoli. Gli schedules ordinabili rispetto alla relazione di dipendenza data dall'unione di tutte le dipendenze importanti sono detti corretti o consistenti.

Alcune dipendenze sono inoltre interessanti per lo studio del fenomeno degli aborti in cascata. Se, ad esempio, una transazione t legge un dato modificato precedentemente da un'altra transazione t' non ancora terminata, nel caso in cui t' abortisce, anche t deve essere abortita perche` ha letto un dato scorretto. Tale situazione e` identificabile con il formarsi di una dipendenza di tipo D3 tra le due transazioni t' e t, quando t' non e` ancora terminata. Le altre dipendenze, D1, D2 e D4, non causano mai aborto in cascata perche` la seconda transazione non dipende da nessuna modifica apportata dalla prima; al contrario D3 indica proprio un trasferimento di informazione fra la prima e la seconda transazione. Per evitare gli aborti in cascata sara` dunque necessario identificare le dipendenze interessanti a tale scopo e controllare la loro formazione.

La specifica di un tipo astratto condiviso comprende:

1) la specifica delle operazioni del tipo; ad esempio in temini di precondizioni, postcondizioni e invarianti;

2) la specifica dell'operazione di undo per ogni operazione definita al punto precedente; ancora, se si vuole, in termini di precondizioni, postcondizioni e invarianti. Le operazioni di undo sono necessarie perche` il modello di transazione prevede che ogni operazione possa successivamente essere "disfatta" se la transazione che l'ha invocata abortisce. Percio` la specifica di interleaving (punto 3) di un certo tipo definisce un insieme di schedules consistenti che comprende anche quelli in cui sono inserite le operazioni di undo in un qualsiasi punto dopo l'invocazione dell'operazione e prima della fine della transazione invocante;

3) la specifica di interleaving cioe` una descrizione di come le operazioni di piu` transazioni possono essere intercalate fra loro negli schedules. Vedremo come questo corrisponda a elencare tutte le possibili dipendenze, a individuare quelle

importanti e a stabilire la relazione di dipendenza rispetto a cui garantire l'ordinabilita';

4) la lista delle dipendenze che devono essere controllate per prevenire gli aborti in cascata.

Vediamo la definizione del tipo di dato astratto condiviso Coda-FIFO. La Coda-FIFO e' una coda con le operazioni caratteristiche di inserzione e rimozione di un elemento. L'inserzione (QEnter) ha come parametri una coda e un elemento e il suo effetto e' quello di inserire l'elemento come l'ultimo della coda. La rimozione (QRemove) ha come parametro una coda e restituisce, togliendolo dalla coda, l'elemento in testa alla coda, che e' anche quello che vi e' stato inserito per primo.

Operazioni:

- QEnter (coda, elem) : aggiunge l'elemento, elem, alla coda. L'undo rimuove l'elemento cosi' inserito.

- QRemove (coda) : rimuove l'entrata in testa alla coda. Se la coda e' vuota l'operazione non puo' essere eseguita e la transazione richiedente si mette in attesa che venga inserito un elemento nella coda. L'undo reinserisce l'entrata in testa alla coda.

Supponiamo di considerare i seguenti vincoli di consistenza:

- se una transazione aggiunge piu' elementi alla coda questi devono apparire contigui in testa alla coda e nello stesso ordine in cui sono stati inseriti;

- ogni elemento non puo' essere osservato da nessun'altra transazione se quella che lo ha inserito non e' terminata;

- se due transazioni inseriscono elementi in due code distinte l'ordine delle entrate delle due transazioni deve essere lo stesso in entrambe le code.

Per ragionare sul comportamento concorrente delle transazioni che operano sulla coda vediamo l'insieme di tutte le dipendenze che possono formarsi:

D1: ti: E(e) -----> tj: E(e')
ti inserisce l'elemento e subito dopo tj inserisce l'elemento distinto e';

D2: ti: E(e) -----> tj: R(e')
ti inserisce l'elemento e e successivamente tj rimuove l'elemento e';

D3: ti: E(e) -----> tj: R(e)
ti inserisce l'elemento e e successivamente tj estrae lo stesso elemento;

D4: ti: R(e) -----> tj: E(e')
ti rimuove l'elemento e dopo di che tj inserisce l'elemento e';

D5: ti: R(e) -----> tj: R(e')
ti rimuove l'elemento e e successivamente tj rimuove l'elemento e'.

Le dipendenze D2 e D4 sono non importanti perche` non e` osservabile l'ordine di esecuzione delle due operazioni: l'elemento inserito da una transazione e` distinto da quello estratto dall'altra transazione. Notiamo che con la sola suddivisione delle operazioni in letture e scritture le dipendenze D2 e D4 non sarebbero state considerate non importanti. Se modellassimo una QEnter come scrittura e una QRemove come una lettura (del dato in testa alla coda) seguita da una scrittura (l'eliminazione del dato), dovremmo impedire cicli nelle dipendenze D2 e D4. Invece la conoscenza semantica che abbiamo delle operazioni sulla coda ci permette di escludere che tali dipendenze siano significative perche` sappiamo per certo che tra due transazioni fra cui si formano le dipendenze precedenti non puo` essrci interferenza. Per garantire la serializzabilita` degli schedules dobbiamo garantire che siano ordinabili rispetto alla relazione di dipendenza <D1UD3UD5, come esposto precedentemente.

Si puo` dimostrare che per mantenere tutti i vincoli di consistenza sul tipo di dato, le dipendenze D1, D3 e D5 devono essere considerate importanti. La dipendenza D1 e` importante perche` se non e` garantita l'ordinabilita` degli schedules rispetto alla relazione di dipendenza <D1, elementi inseriti da una stessa transazione possono apparire in posizioni non contigue nella coda. La dipendenza D3 e` importante perche` se si permette ad una transazione tj di rimuovere un elemento inserito da una transazione non ancora terminata ti, si viola il secondo vincolo di consistenza. La dipendenza D5 e` importante perche` se una transazione ti ha chiesto la rimozione dell'elemento in testa alla coda e si permette ad un'altra transazione tj di rimuovere l'elemento successivo, nel caso in cui tj termina con successo e ti abortisce non e` rispettato il comportaamento FIFO della coda. La relazione di dipendenza rispetto a cui garantire l'ordinabilita` e` ancora: <D1UD3UD5. In questo caso il rispetto dei vincoli di consistenza garantisce la serializzabilita` degli schedules.

Le dipendenze che possono provocare aborto in cascata sono D3 e D5 infatti se tj dipende da ti secondo la dipendenza D3 allora tj estrae dalla coda l'elemento inserito da ti e in caso di aborto di ti anche tj deve essere abortita. Se fra tj e ti si forma la dipendenza D5 vuol dire che ti ha rimosso un elemento dalla coda e successivamente tj ne ha tolto un altro. Se ti abortisce l'elemento che aveva tolto viene reinserito nella coda e tj deve essere abortita perche` ha prelevato un elemento che non e` piu` il primo della coda.

Vediamo ora una variante del tipo coda. Uno degli usi piu`

comuni di una coda e` quello di fungere da buffer di scambio fra attivita` che producono e attivita` che consumano informazione. Come la Coda FIFO, anche questo tipo di coda prevede due operazioni: l'inserzione di un elemento in "coda" alla coda e la rimozione dell'elemento in testa; tuttavia per questa coda sono diversi i vincoli di consistenza che e` necessario mantenere. Alcuni vincoli che sono definiti per il tipo coda FIFO possono venire rilasciati. Le uniche condizioni che si devono rispettare sono che un elemento inserito nella coda non vi resti per sempre, ma che prima o poi compaia alla testa, e che si abbia un comportamento "fair" tra elementi inseriti all'incirca nello stesso momento. Al contrario della coda FIFO si puo`, ad esempio, permettere che elementi inseriti da una transazione non appaiano in posizioni contigue nella coda ma che siano mescolati con elementi inseriti da altre transazioni.

Questo tipo di coda e` detto Coda-FIFO-Debole (Weak FIFO Queue) e le operazioni, analoghe a quelle della coda FIFO, sono indicate: WQenter (coda, elem) e WQremove (coda).

Anche l'insieme delle dipendenze e` lo stesso che si ottiene per la coda FIFO, cambia solo il nome delle operazioni:

D1: ti: WE(e) --> tj: WE(e');
D2: ti: WE(e) --> tj: WR(e');
D3: ti: WE(e) --> tj: WR(e');
D4: ti: WR(e) --> tj: WE(e');
D5: ti: WR(e) --> tj: WR(e').

Si puo` osservare che per questo tipo di dato, oltre che nelle D2 e D4, si possono permettere cicli anche nelle relazioni di dipendenza che includono <D1 e <D5 senza violare i vincoli di consistenza. Infatti la D1 e la D5 sono significative se si vuole, rispettivamente, che gli elementi inseriti da una transazione appaiano contigui nella coda e che le rimozioni effettuate da una transazione estraggano un insieme contiguo di elementi in testa alla coda. Poiche` nella coda debole questi vincoli sono rilasciati per mantenere la consistenza e` sufficiente garantire l'ordinabilita` rispetto alla relazione di dipendenza <D3. Gli schedules che si ottengono non sono serializzabili e tuttavia rispettano il comportamento atteso per il tipo di dato coda-debole.

Accenniamo ora brevemente alla situazione in cui le transazioni che operano nel sistema coinvolgono operazioni su piu` tipi di dato astratti condivisi. Siano y1,..., yn i tipi di dato coinvolti e supponiamo che per yi sia garantita l'ordinabilita` rispetto alla relazione di dipendenza <Dyi. Gli schedules consistenti per y1,..., yn sono definiti come quelli ordinabili rispetto alla relazione di dipendenza <Dy1UDy2U..UDyn. Se e` verificata tale proprieta` allora y1,..., yn sono detti tipi cooperativi.

Se l'ordinabilita` rispetto alla relazione di dipendenza <Dyi garantisce la serializzabilita` degli schedules di

transazioni che eseguono operazioni sul tipo di dato yi, allora l' ordinabilita` rispetto alla relazione di dipendenza: <Dy1U ..... UDyn, garantisce la serializzabilita` dell'esecuzione concorrente di transazioni che eseguono operazioni sui tipi di dato y1, .., yn.

Il modello proposto si puo` considerare come una evoluzione rispetto alle teorie di Eswaran-Gray e Bernstein. Il concetto di ordinabilita` rispetto a tutte le dipendenze, infatti, equivale alla prima condizione di consistenza definita in [ESW '76], in cui non e` considerata la suddivisione delle operazioni in letture e scritture.

La serializzabilita` come e` vista in [BERN '81], che e` una condizione gia` meno stringente della precedente, equivale all'ordinabilita` rispetto a tutte le dipendenze in cui almeno una delle operazioni coinvolte e` una scrittura. Come abbiamo avuto occasione di mostrare, si possono trovare condizioni di ordinabilita` che non implicano la serializzabilita` e che comunque garantiscono il mantenimento della correttezza ed e` proprio in cio` che possiamo riconoscere l'evoluzione della teoria apportata da Schwarz e Spector.

4. Meccanismi per il controllo della concorrenza

Riassumendo quanto visto finora possiamo dire che un sistema transazionale deve limitare le possibili esecuzioni concorrenti a quelle che soddisfano la condizione di correttezza. Resta da affrontare il problema di come fare ad imporre che gli schedules siano corretti, cioe' restano da definire i meccanismi per il controllo della concorrenza. Le tecniche che si possono adottare per vincolare gli schedules sono fondamentalmente due ([KUN '81]): far attendere le transazioni, o farle abortire. Siano ti e tj transazioni; puo' accadere che in uno schedule l'operazione opj di tj debba seguire l'operazione opi di ti affinche' il risultato sia corretto. Per l'evoluzione delle transazioni l'operazione di tj puo' essere pronta per l'esecuzione prima di quella di ti, che invece la dovrebbe precedere. Per forzare l'ordinamento desiderato nello schedule si puo' far attendere tj, sospendendo la sua esecuzione, fino a che ti non ha eseguito opi. Oppure si puo' permettere che le operazioni vengano eseguite in un ordine qualsiasi riservandosi la possibilita' di abortire una delle due transazioni qualora lo schedule risultante non sia corretto. Ricordiamo che con l'aborto si eliminano dallo schedule tutte le azioni eseguite fino a quel momento dalla transazione abortita. La transazione abortita puo' essere reinizializzata o implicitamente dal sistema stesso o esplicitamente dall'utente al quale, in quest'ultimo caso, e' necessario notificare l'avvenuto aborto.

Nei casi in cui si impone alle transazioni di attendere si deve di solito affrontare il problema del deadlock, infatti si corre il rischio che si crei una situazione di attesa circolare. Vediamo cosa e' una attesa circolare. Se la transazione ti attende che tj esegua un'operazione e tj e' a sua volta in attesa dell'esecuzione di un'operazione da parte di ti stessa, si ha una situazione di blocco che, se non e' risolta, puo' durare all'infinito. In questo caso il ciclo di attesa coinvolge due sole transazioni. Piu' in generale un insieme di n transazioni (n>=2) e' in uno stato di deadlock quando ogni transazione dell'insieme sta attendendo per un evento che puo' essere causato solo da un'altra transazione dello stesso insieme ([PET]). Il deadlock e' un problema classico dei sistemi operativi che sorge nell'allocazione delle risorse del sistema ai processi e sono state studiate molte tecniche per la sua risoluzione. Noi tratteremo in dettaglio alcuni metodi per la risoluzione del deadlock in relazione all'applicazione di un protocollo di locking (paragrafo 4.1.2).

Il problema del controllo della concorrenza e' di solito affrontato con la costruzione di protocolli che, se applicati da tutte le transazioni, permettono di ottenere solo schedules corretti. L'implementazione dei protocolli richiede l'adozione di un meccanismo per il controllo della concorrenza. I meccanismi piu' usati e studiati sono due:

1) il locking

2) l'uso di timestamps.

Con queste tecniche si possono sviluppare molte politiche di gestione della concorrenza. Nei paragrafi 4.1 e 4.2 tratteremo in dettaglio rispettivamente i meccanismi basati sul locking e i meccanismi che fanno uso di timestamps. Esistono poi altri approcci al problema del controllo della concorrenza alternativi ai precedenti, noi vedremo nel capitolo 5 l'approccio ottimistico.

4.1. Meccanismi di locking

La prima classe di meccanismi per il controllo della concorrenza che illustreremo e' quella dei meccanismi basati sul locking. La tecnica del locking e' stata utilizzata per la costruzione di innumerevoli protocolli di sincronizzazione anche molto diversi tra loro. Dei protocolli di locking piu' importanti daremo una descrizione abbastanza approfondita, per altri meno usati rimandiamo alla letteratura ([BERN '81], [ROS '78], [ESW '76], [GRAY '74], [GRAY1 '75], [GRAY2 '75], [GRAY '78], [TRA '82], [SC-SP '84], [KLU '83], [KOR '83], [SPEC '85], [BAY '80]).

4.1.1. Descrizione del metodo e protocollo 2PL

Quando una transazione esegue un'operazione di lock su un dato rende il dato inaccessibile alle altre entita' attive. Quando ha terminato di operare sul dato lo rilascia eseguendo l'operazione di unlock.

L'implementazione delle operazioni di lock e unlock e' di solito delegata ad una entita', il gestore degli accessi, che si incarica di eseguire le operazioni per conto delle transazioni. Nel nostro modello di sistema transazionale possiamo vedere il Data Manager come il modulo che svolge le funzioni di questa entita'.

Se una transazione deve lavorare su un dato momentaneamete inaccessibile, questa non puo' eseguire l'operazione di lock e deve attendere fino a che non viene rilasciata la chiave di accesso al dato dalla transazione che la possiede. Si possono avere piu' transazioni in attesa per il rilascio di una lock, in questo caso si puo' pensare di gestire le transazioni che stanno attendendo con politica FIFO per evitare l'attesa infinita, o con una politica diversa, dettata dalle esigenze del sistema. Il protocollo che ogni transazione deve seguire per accedere ad un dato x e' dunque il seguente:

1) richiedere l'operazione di lock sul dato x, che indicheremo con lock(x); se l'operazione non puo' essere eseguita immediatamente la transazione deve attendere;

2) eseguire sul dato tutte le operazioni che desidera;

3) rilasciare il dato eseguendo un'operazione di unlock su x, che indicheremo con unlock(x).

Se tutte le transazioni seguono il protocollo le operazioni su uno stesso dato da parte di transazioni distinte vengono sempre eseguite in maniera mutuamente esclusiva. Il protocollo illustrato introduce, pero`, delle dipendenze tra le transazioni dovute all'attesa. Come in tutti i casi in cui si adotta l'attesa come tecnica di sincronizzazione, puo` sorgere il problema del deadlock. Vediamo un semplice esempio.

Esempio

Siano t1 e t2 transazioni che accedono entrambe ai dati x e y. Supponiamo che t1 acceda prima ad x e poi ad y e che t2 acceda prima ad y e poi ad x. Se lo schedule dell'esecuzione concorrente e` il seguente, si verifica una situazione di deadlock.

Schedule:

t1: lock(x)
t1: <accede ad x>
t2: lock(y)
t2: <accede ad y>
t1: lock(y) e si mette in attesa
t2: lock(x) e si mette in attesa

La transazione t1 attende che t2 rilasci il dato y, ma t2 stessa e` impegnata in una attesa per il dato x posseduto da t1 e entrambe t1 e t2 resteranno per sempre in attesa.

Il protocollo di acquisizione e rilascio delle lock visto e` quello piu` semplice. Esso prevede un solo tipo di lock con cui si realizza la mutua esclusione tra operazioni di transazioni distinte.

Poiche` l'obiettivo del controllo della concorrenza e` l'ottenimento di schedules corretti, resta da verificare se tale protocollo e` sufficiente a garantire la proprieta` di correttezza. Tradizionalmente la correttezza e` stata identificata con la serializzabilita` ([ESW '76], [BERN '81]). Vediamo con un esempio che la mutua esclusione non e` sufficiente a garantire la serializzabilita`.

Esempio

Siano t1 e t2 due transazioni che accedono ai dati x e y per modificarli. Supponiamo che t1 acceda prima ad x e poi ad y, e che t2 acceda prima ad y e poi ad x. Lo schedule dell'esecuzione concorrente sia S.

```
S:
    t1: lock(x)
    t1: <modifica x>
    t1: unlock(x)

    t2: lock(y)
    t2: <modifica y>
    t2: unlock(y)

    t1: lock(y)
    t1: <modifica y>
    t1: unlock(y)

    t2: lock(x)
    t2: <modifica x>
    t2: unlock(x)
```

Come si puo` facilmente vedere tale schedule non e` equivalente a nessuno schedule seriale; infatti t1 e t2 operano sui dati comuni x e y ma mentre il valore finale di x e` quello assegnato da t2, il valore finale di y e` quello assegnato da t1.

In [ESW '76] e` dimostrato che occorre che le transazioni siano ben formate e a due fasi affinche` gli schedules siano serializzabili.

Definizione ([GRAY '78]):

Una transazione si dice ben formata (well-formed) se rispetta le seguenti regole:

1) esegue l'operazione di lock prima di accedere ad un dato;

2) non richiede mai la lock su un dato lockato da se stessa;

3) prima di terminare rilascia tutte le locks acquisite.

Definizione ([GRAY '78]):

Una transazione e` a due fasi (two-phase) se non richiede altre locks dopo aver rilasciato almeno una lock.

In altri termini una transazione e` two-phase se evolve attraverso una prima fase di acquisizione delle locks (growing phase) ed una seconda fase di rilascio (schrinking phase).

Le transazioni t1 e t2 dell'esempio precedente sono ben formate ma non sono a due fasi.

Definizione

Uno schedule S e` legale se, quando viene eseguita la k-esima azione (t, a, e), e e` lockato da t e non e` lockato da nessun'altra transazione diversa da t.

Se tutte le transazioni sono ben formate e a due fasi allora ogni schedule legale e` corretto ([ESW '76]).

Il protocollo Two-Phase-Locking (2PL) e` quello che impone alle transazioni di essere ben formate e a due fasi. Definendo punto di lock ([BERN '81]) per una transazione il punto al termine della sua fase di acquisizione si osserva che, se tutte le transazioni seguono il protocollo 2PL, si ottiene uno schedule equivalente all'esecuzione seriale delle transazioni secondo l'ordinamento dato dai punti di lock.

Il protocollo di 2PL illustrato prevede una sola modalita` di lock che l'utente deve richiedere prima di accedere ad ogni dato. Un primo affinamento del metodo e` ottenuto semplicemente osservando che non tutte le operazioni su un dato devono essere eseguite in maniera esclusiva. Piu` operazioni di lettura possono, ad esempio, andare in parallelo senza pericolo di violare la consistenza dei dati. Per questo motivo sono stati introdotti due tipi di lock ([ESW '76] e [BERN '81]):

1) read-lock detto anche lock condiviso, che indicheremo con r-lock;

2) write-lock detto anche lock esclusivo, che indicheremo con w-lock.

Il protocollo di locking deve essere modificato per quanto riguarda il passo di acquisizione della lock. Una transazione eseguira` un'operazione di r-lock(x) se intende solo leggere il dato x oppure un'operazione di w-lock (x) se lo vuole modificare.

Si introduce il concetto di compatibilita` tra tipi di lock:

diciamo che una lock di tipo k e` compatibile con una lock di tipo k' se, su uno stesso oggetto, possono essere eseguite entrambe contemporaneamente.

Se si considerano solo i due tipi r-lock e w-lock si osserva che il tipo r-lock e` compatibile con se` stesso, perche` piu` letture possono essere eseguite contemporaneamente, ma non lo e` con il tipo w-lock perche` non si puo` leggere un dato mentre e` modificato da qualcun altro. Il tipo w-lock non e` compatibile ne` con se` stesso ne` con il tipo r-lock perche` le modifiche devono avvenire in modo esclusivo. Due tipi di lock sono compatibili se le corrispondenti operazioni non sono in conflitto, secondo la definizione di operazioni in conflitto data nel capitolo 2.

La compatibilita` puo` essere espressa mediante una matrice, detta Tabella di Compatibilita`, dove l'indice di colonna indica la lock concessa e l'indice di riga la lock richiesta. La Tabella di Compatibilita` per i tipi di lock read e write e` la seguente:

	r-lock	w-lock
r-lock	si	no
w-lock	no	no

Questo protocollo di acquisizione e rilascio delle lock realizza la mutua esclusione per l'esecuzione da parte di transazioni distinte di operazioni in conflitto.

Anche l'applicazione del protocollo 2PL porta, in alcuni casi, a comportamenti non desiderati, se non addirittura a degli errori. Vediamolo con un esempio:

Esempio:

Sia t1 una transazione che accede ai dati x e y per modificarli e t2 una transazione che legge il valore di x. Sia S lo schedule dell'esecuzione concorrente.

```
S:
   t1: w-lock(x)
   t1: w-lock(y)
   t1: <modifica x>
   t1: unlock(x)
   t2: r-lock(x)
   t2: <legge x>
   t2: unlock(x)
   t2: end
   t1: <modifica y>
   t1: abort
```

L'aborto di t1, per definizione, riporta i valori di x e y a quelli che avevano prima dell'inizio di t1; da cio` deriva che il valore letto da t2 per il dato x non e` corretto perche` x non ha mai assunto quel valore in uno stato consistente del database (il valore letto da t2 e` un fantasma).

Se non si prendono altre precauzioni, il protocollo 2PL non mantiene la proprieta` di atomicita` delle transazioni perche` in alcuni casi puo` rendere visibili all'esterno i risultati parziali delle transazioni. Nell'esempio t2 ha visto i risultati parziali dell'esecuzione di t1. Un modo di ovviare a questo inconveniente e` quello di non far terminare mai una transazione che ha usato dei dati modificati da altre transazioni attive finche` queste non sono a loro volta terminate. Infatti se t2 non e` ancora terminata quando t1 abortisce si puo` ancora recuperare la situazione forzando anche t2 ad abortire. A sua volta l'aborto di t2 puo` portare a dover abortire altre transazioni che hanno osservato i valori dei dati modificati da t2 ed avere cosi` aborto in cascata. L'aborto in cascata non e` un caso di errore pero`, e` un effetto indesiderato perche` diminuisce l'efficienza del sistema. Si costringono a ripartire dall'inizio transazioni

gia` in parte eseguite con un notevole costo dovuto sia all'annullamento degli effetti dell'ultima esecuzione, sia alla nuova esecuzione.

Il comportamento messo in luce nell'esempio si puo` eliminare imponendo alle transazioni di rispettare un'altra regola che stabilisce di rilasciare le locks tutte in una sola volta alla terminazione o all'aborto ([ROS '78]). Il rilascio delle locks avverra` nella fase di commitment della transazione e servira` a rendere visibili solo i risultati dell'esecuzione completa della transazione. Questa regola e` applicata in molti sistemi anche se la ritenzione delle locks ha come contropartita una limitazione della concorrenza.

Una forma particolare di 2PL e` quella in cui una transazione deve acquisire le locks su tutti i dati su cui opera prima di iniziare l'esecuzione vera e propria, tale forma e` nota con il nome di Predeclaration. La condizione perche` il protocollo di 2PL con predeclaration sia applicabile e` che sia rilevabile da un'analisi statica della transazione il sottoinsieme dei dati a cui accede. Nei casi in cui il requisito e` soddisfatto si ha che i conflitti sono tutti risolti prima dell'effettiva esecuzione della transazione che, una volta iniziata, puo` quindi procedere senza ritardi dovuti al meccanismo di controllo della concorrenza. Inoltre, in caso di deadlock (par. 4.1.2), la reinizializzazione della transazione comporta un costo minimo in quanto non c'e` necessita` di annullare gli effetti delle operazioni perche` non sono ancora state eseguite.

Passiamo ora a considerare l'implementazione dei protocolli di locking. L'implementazione di 2PL richiede la costruzione di un modulo software, lo scheduler, che riceve le richieste di lock/unlock dalle transazioni e le gestisce in accordo alle regole del protocollo. In un sistema centralizzato questo modulo e` unico. Riferendoci al modello di sistema transazionale che abbiamo presentato (capitolo 1) le funzionalita` dello scheduler possono essere associate al Data Manager. In un sistema distribuito si puo` evere un unico scheduler centralizzato (implementazione centralizzata) con i vantaggi e svantaggi che comporta l'introduzione di un punto di centralizzaaione. Lo scheduler centralizzato ha infatti una conoscenza globale della situazione del sistema il che facilita, come vedremo, il trattamento del deadlock, ma, d'altro canto, viene imposto un notevole carico di lavoro al nodo su cui risiede, se il nodo cade si blocca l'intero sistema. Inoltre, l'implementazione centralizzata richiede un alto numero di messaggi da inviare sulla rete. I messaggi scambiati saranno richieste di lock/unlock dai Transaction Managers (TMs) allo scheduler e messaggi di richiesta di esecuzione di operazioni tra lo scheduler e i Data Managers (DMs).

Lo scheduler puo`, alternativamente, essere distribuito sulla rete (implementazione distribuita) associando uno scheduler locale ad ogni DM. Ogni scheduler gestisce gli accessi

concorrenti ai dati mantenuti dal DM a cui e` associato; in questo caso la richiesta di lock puo` essere inviata implicitamente con la richiesta di operazione.

D'ora in avanti considereremo il 2PL come il protocollo standard per garantire la serializzablita`, usando l'opzione che prevede il rilascio delle lock solo alla terminazione della transazione. Un protocollo che prevede il rilascio delle locks prima del commit della transazione e` stato studiato in [NETT], in cui e` mostrato come, con un meccanismo di recovery che tenga conto del propagarsi dei dati ad altre transazioni, ad esempio mediante l'uso di un grafo, e con il coordinamento dei commit delle transazioni, si puo` ugualmente mantenere la proprieta` di serializzabilita` delle esecuzioni concorrenti. Qui non ci interesseremo oltre a questo protocollo rimandando alla letteratura.

4.1.2. Tecniche per il trattamento del deadlock

Il grosso svantaggio dei meccanismi di locking consiste nel dover sempre tener conto della possibilita` di avere deadlock. Ci sono due categorie di metodi per trattare il problema del deadlock note nella letteratura con il nome di **deadlock prevention** e **deadlock detection** ([PET], [BERN '81], [GRAY '74]). Nel **deadlock prevention** si usa un protocollo per assicurare che il sistema non si trovi mai nello stato di deadlock, nel **deadlock detection**, invece, e` permesso che si creino cicli di attesa nel sistema ed e` implementato un meccanismo di rilevamento del deadlock e di recupero da tali situazioni. All'interno di queste due categorie tratteremo le tecniche interessanti per la risoluzione del problema del deadlock in relazione all'applicazione di un protocollo di locking per il controllo della concorrenza.

Una tecnica diffusa per il trattamento del deadlock e` quella basata sull'**uso dei time-out**. L'idea e` di supporre che, se una transazione non termina entro un tempo stabilito a prori non potra` piu` terminare perche` si trova in uno stato di deadlock. In questo caso si forza l'aborto della transazione per spezzare il ciclo d'attesa. La tecnica non rientra tra quelle di **deadlock prevention**, perche` e` permesso che il deadlock si verifichi, e neppure fra quelle di **deadlock detection**, perche` lo scadere del time-out non indica necessariamente la presenza di deadlock. E` comunque la piu` usata per la sua semplicita` d`implementazione.

Deadlock Prevention

Ricordiamo che un'operazione di lock non e` accettata quando e` in conflitto con almeno un'altra lock gia` richiesta. Due lock sono in conflitto se sono eseguite da transazioni distinte, operano sullo stesso dato e, se si usa il modello read/write, una delle due e` una w-lock. Diciamo che una transazione t entra in

conflitto con una transazione t` se t richiede una lock su un dato per cui t' ha gia` richiesto la lock e non ha ancora eseguito la relativa unlock e le due lock richieste da t e da t' sono in conflitto. Notiamo che t' puo` essere essa stessa in attesa perche` gia` entrata in conflitto con altre transazioni per lo stesso dato.

Nei metodi di deadlock prevention, quando una transazione ti entra in conflitto con un'altra transazione tj, ti e` posta in attesa se e solo se cio` non puo` portare mai a deadlock, altrimenti una delle due transazioni coinvolte nel conflitto e` abortita e reinizializzata [BERN '81]. L'essenza del metodo sta nel riuscire a stabilire quando l'attesa di una transazione non portera` mai a deadlock. Eludendo la soluzione a tale problema esiste un approccio molto semplice che e` quello di non permettere mai che una transazione attenda per accedere a dati gia` posseduti da un'altra. Ogni volta che una transazione entra in conflitto, tale transazione viene abortita e reinizializzata: il problema del deadlock e` eliminato, ma abbiamo un numero molto elevato di restarts che influisce negativamente sulla performance del sistema.

Quando e` possibile conoscere a priori gli oggetti acceduti da una transazione, una tecnica per la prevenzione del deadlock e` il preordine delle risorse. Si stabilisce un ordinamento totale sui dati assegnando a tutti i dati un numero intero unico e si impone che le transazioni richiedano i dati solamente nell'ordine crescente di numerazione stabilito. Una transazione puo` all'inizio richiedere un dato con numero qualsiasi, ad esempio n, dopo di che la transazione puo` richiedere solo dati con numero maggiore di n. Se tale protocollo e` usato da tutte le transazioni si puo` dimostrare facilmente che non ci puo` essere attesa circolare ([PET]). Il vantaggio della tecnica appena descritta e` quella di evitare i restarts, ma non sempre e` applicabile e comporta una ridotta utilizzazione delle risorse. Infatti una transazione che vuole accedere ad un dato d deve prima acquisire tutti i locks sui dati che usera` e che precedono d nell'ordine di numerazione, anche se tali dati saranno utilizzati solo dopo un certo tempo, durante il quale non sono accessibili ad altre transazioni.

Passiamo ora ad analizzare un meccanismo di prevenzione del deadlock che, per la sua generalita`, e` quello finora piu` usato nei sistemi esistenti ([ROS '78]). Il controllo di concorrenza usa un ordinamento totale tra le transazioni: quando una transazione t inizia le viene assegnata una marca numerica unica nel sistema, detta timestamp che indicheremo con TS(t). Tale marca contraddistingue la transazione dall'inizio dell'esecuzione fino alla terminazione ed e` mantenuta costante anche in caso di aborto e reinizializzazione. Poiche` e` ragionevole supporre che due transazioni non inizino nello stesso ciclo di clock, la funzione di assegnazione dei timestamp puo` essere la seguente:

TS(ti) = orologio di macchina quando ti inizia, se il sistema e` centralizzato;

TS(ti) = orologio di macchina quando ti inizia, dove ha inizio, * identificatore unico del nodo dove ha inizio (con * si e' indicata la concatenazione), se il sistema e' distribuito.

Quando una transazione ti entra in conflitto con un'altra tj il controllo di concorrenza risolve il conflitto basandosi sul confronto dei timestamps associati alle transazioni coinvolte e scegliendo una delle seguenti alternative:

- far attendere ti, indicata con il termine WAIT;
- far abortire e reinizializzare ti, indicata DIE;
- far abortire e reinizializzare tj, indicata con WOUND.

Sempre in [ROS '78] sono mostrati due metodi di controllo della concorrenza, il WAIT-DIE e il WOUND-WAIT, che in base al valore dei timestamps applicano una delle scelte indicate.

Metodo WAIT-DIE

Assumiamo che ti entri in conflitto con tj, il conflitto e' risolto applicando la seguente regola:

se TS(ti) < TS(tj) allora WAIT altrimenti DIE

Se la transazione ti e' piu' vecchia della transazione con cui e' entrata in conflitto (il suo timestamp e' minore di quello di tj), ti e' messa in attesa, altrimenti ti e' abortita e reinizializzata. Se una richiesta di lock pone ti in conflitto con piu' di una transazione la regola precedente deve essere applicata a tutte le transazioni con cui e' entrata in conflitto. Si puo' osservare che ti puo' attendere solo se e' piu' vecchia di tutte le transazioni coinvolte.

Metodo WOUND-WAIT

Assumiamo che ti entri in conflitto con tj, il conflitto e' risolto applicando la seguente regola:

se TS(ti) < TS(tj) allora WOUND altrimenti WAIT

Se la transazione ti e' piu' vecchia di quella con cui e' entrata in conflitto tj, allora tj e' abortita e reinizializzata altrimenti ti e' posta in attesa. Anche in questo caso se una richiesta di lock pone ti in conflitto con piu' di una transazione la regola deve essere applicata a tutte quelle con cui e' entrata in conflitto.

Vantaggi e svantaggi dei due metodi

Entrambi i metodi risolvono il problema del deadlock permettendo ad una transazione ti di attendere il rilascio di un dato da un'altra transazione tj solo se i loro timestamps sono in un certo ordine, nel WAIT-DIE se TS(ti) < TS(tj) mentre nel WOUND-WAIT se TS(ti) > TS(tj). Stabilire un ordinamento monotono crescente\decrescente fra i timestamps delle transazioni che si possono attendere evita la formazione di cicli di attesa fra le transazioni. Ci sono comunque differenze significative nel modo

in cui i due metodi operano in alcune situazioni. Il WAIT-DIE permette ad una transazione di attendere per transazioni piu' giovani, di conseguenza quanto piu' una transazione invecchia tanto piu' tende ad aspettare; vediamo con un esempio una situazione in cui cio' si verifica.

Esempio

Sia TS(ti) = i e supponiamo che i dati a e b non siano ancora stati acceduti da nessuna transazione. Consideriamo il seguente schedule:

S: <nasce t1>;
<nasce t2>;
t2 esegue lock(a) ed acquisisce il diritto di accesso ad a; t1 esegue lock(a) e poiche' TS(t1) <TS(t2), t1 si mette in attesa;
<nasce t3>;
t3 esegue lock(b) ed acquisisce il diritto di accesso a b;
t2 esegue unlock(a);
t1 di conseguenza acquisisce il diritto di accesso ad a;
t1 esegue lock(b) e poiche' TS(t1) < TS(t3), t1 si mette in attesa.

Mentre t1 attende per accedere il dato a, nasce t3, transazione per la quale t1 in seguito deve attendere per accedere al dato b.

Nel WOUND-WAIT, invece, una transazione non attende mai per una piu' giovane; quanto piu' una transazione resta nel sistema, tanto piu' acquista privilegio e tende ad essere eseguita velocemente. Quando ti entra in conflitto con una transazione tj con timestamp maggiore tj e' fatta abortire. Una critica a tale metodologia e' che permettere ad una transazione di provocare l'aborto di un'altra puo' degradare le prestazioni del sistema perche' si puo' far tornare all'inizio anche transazioni che potrebbero tranquillamente evolvere fino al termine. Ricordiamo che se tj e' nella fase di commit l'aborto di tj non e' piu' possibile, questo pero' non crea problemi per il deadlock perche' , se tj e' in tale fase dell'esecuzione, non richiedera' mai altre lock e quindi non possono formarsi cicli di attesa.

Un vantaggio del metodo WAIT-DIE e' che una volta acquisite tutte le locks una transazione t non sara' piu' restartata a causa dei conflitti, tale proprieta' puo' risultare necessaria in alcuni casi. Consideriamo una transazione t strutturata in due parti: una prima parte in cui acquisisce tutti i locks per accedere ad un insieme di dati ed una seconda parte in cui interagisce con dei dispositivi periferici. Quando t entra nella seconda parte dell'esecuzione non puo' piu' essere abortita perche' non si possono annullare gli effetti dei comandi ai dispositivi. L'uso di un metodo WOUND-WAIT in presenza di tali transazioni puo' portare a situazioni di errore.

Rosenkrantz ha proposto una diversa interpretazione di WOUND che rende il metodo applicabile. Il WOUND assume il significato di aborto e reinizializzazione di una transazione se tale transazione e` in attesa in quel momento, altrimenti la transazione puo` procedere finche` o inizia la terminazione o si pone in attesa. Se si pone in attesa si ha ancora l'aborto e la reinizializzazione della transazione. Riprendiamo l'esempio precedente: se t e` nella seconda parte, t non richiede piu` lock quindi non entrera` mai in attesa e il WOUND non provochera` mai l'aborto di tale transazione.

Un'altra osservazione che in generale si puo` fare sui metodi WAIT-DIE e WOUND-WAIT riguarda le catene di restarts. Nel WAIT-DIE se una transazione ti e` restartata per un conflitto con tj (quindi TS(ti) > TS(tj)) e nella nuova esecuzione ti entra ancora in conflitto con tj, ti e` restartata di nuovo. Se si hanno lunghe sequenze di restart la transazione ti consuma inutilmente le risorse del sistema. Nel WOUND-WAIT tali catene di restart non si verificano, se infatti ti entra in conflitto con tj e TS(ti) < TS(tj) allora tj e` restartata. Nell'esecuzione successiva di tj in caso di un conflitto con ti, poiche` TS(tj) > TS(ti), tj viene messa in attesa.

Tutte le tecniche di deadlock prevention risolvono il problema del deadlock introducendo numerosi aborti di transazioni non necessari, infatti spesso il deadlock non si sarebbe verificato neppure permettendo alla transazione abortita di attendere. In un sistema centralizzato, dove non e` difficile mantenere l'informazione su quali sono le transazioni in attesa e di chi, e` facile conoscere quando l'attesa di una transazione porta realmente al deadlock. In questo caso e` preferibile usare la tecniche di deadlock detection.

Deadlock Detection

I metodi di deadlock detection permettono ad una transazione di attendere per il rilascio di dati da parte di qualsiasi altra transazione, mantengono l'informazione su quali transazioni sono in attesa e di chi e forniscono un algoritmo che utilizza tali informazioni per rilevare gli eventuali cicli di attesa. Tale algoritmo e` applicato periodicamente allo stato del sistema e quando determina una situazione di deadlock si adotta la soluzione di far abortire una o piu` transazioni coinvolte nel ciclo per riportare il sistema in uno stato privo di deadlock ([BERN '81]).

Il deadlock puo` essere descritto in maniera chiara facendo uso di un grafo diretto noto come GRAFO D'ATTESA (G):

$$G = (V, E)$$

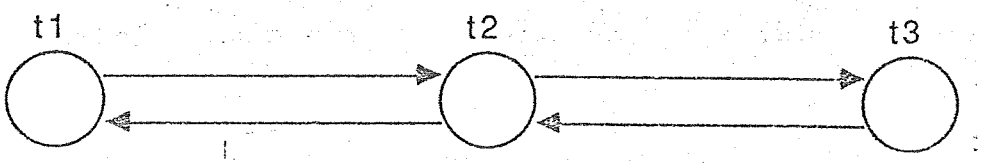
dove:

V e` l'insieme dei vertici ed ogni vertice corrisponde ad una transazione;

E e` l'insieme degli archi ed ogni arco rappresenta una dipendenza d'attesa.

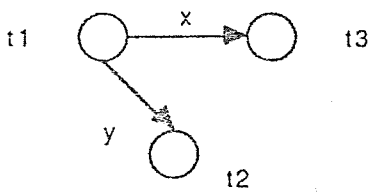
Un arco da vi a vj indica che la transazione ti (associata a vi) e` in attesa di un evento che puo` essere causato da tj. L'arco puo` essere etichettato con l'evento atteso. La presenza di cicli nel grafo rivela una situazione di attesa circolare.

La scelta della transazione da abortire, indicata come la vittima, puo` essere determinata da molti fattori. Puo` infatti interessare fino a che punto la transazione e` stata eseguita e quanto ancora dovra` essere eseguita per arrivare alla terminazione, oppure quante risorse la transazione gia` possiede e di quante ancora ha necessita`, o quante transazioni devono essere coinvolte per eliminare tutti i cicli d'attesa individuati. Se il grafo d'attesa e` quello indicato in figura , un esame completo di esso mostra che l'aborto di t2 elimina entrambi i cicli presenti:

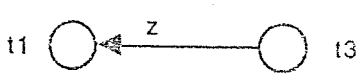


La vittima verra` selezionata in base ai parametri che piu` interessano nell'applicazione. In un sistema distribuito con controllo della concorrenza distribuito si deve anche scegliere se si vuole implementare il rilevamento del deadlock con la costruzione di un Rilevatore Centralizzato o di piu` Rilevatori Gerarchici.

Nel caso del rilevatore centralizzato un nodo viene designato come rilevatore del deadlock. Lo scheduler locale di ogni nodo invia periodicamente il suo grafo d'attesa a tale nodo che ricostruisce, a partire da queste informazioni, un grafo globale che illustra la situazione complessiva delle tarnsazioni in attesa in tutto il sistema. Se i grafi d'attesa del nodo n1 e del nodo n2 sono quelli indicati in figura:

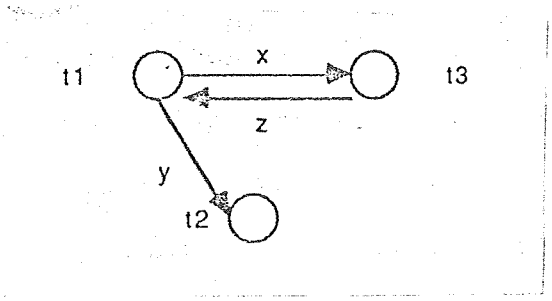


NODO 1



NODO 2

il nodo rilevatore, combinando tali grafi, otterra` il seguente grafo globale, che indica una situazione di attesa circolare fra le transazioni t1 e t3:



Sul nodo n1 t1 attende che t3 rilasci il dato x, sul nodo n2 t3 attende che t1 rilasci il dato z.

Se usiamo rilevatori gerarchici stabiliamo una gerarchia ad albero tra i nodi del sistema ed ogni nodo ha la funzione di rilevatore di deadlock per il sottoalbero di cui e' radice. Il grafo di ogni nodo viene inviato periodicamente al nodo padre il quale ricostruisce il grafo per il sottoalbero di cui e' a sua volta radice e risolve i cicli presenti in tale grafo. Ripetendo il procedimento fino al livello piu' alto della gerarchia fra i nodi si realizza un rilevamento del deadlock "gerarchicamente distribuito".

Nei sistemi distribuiti i metodi di deadlock detection impongono, periodicamente, il trasferimento fra i nodi dei grafi d'attesa. L'overhead che ne deriva puo' essere notevole a seconda della quantita' d'informazione da trasferire. Inoltre il deadlock puo' rimanere presente nel sistema per un tempo proporzionale al tempo che intercorre fra due trasmissioni successive dei grafi, influendo negativamente sull'utilizzazione del sistema.

Tutti i protocolli di locking devono adottare una tecnica per la risoluzione del problema del deadlock; l'orientamento verso un metodo di deadlock prevention o deadlock detection sara' dettato dal tipo di sistema (centralizzato o distribuito), dalle caratteristiche della rete di comunicazione, dal tipo di applicazione e dal grado di parallelismo del sistema.

4.1.3. Sviluppi del meccanismo di locking

Passiamo ora ad analizzare alcuni sviluppi della tecnica di locking tesi all'ottenimento di protocolli che migliorino il grado di concorrenza rispetto al 2PL con modalita' di lock Read e Write. Possiamo determinare due direzioni in cui la ricerca si e' mossa: una prima direzione, anche in senso temporale, che si e' sviluppata dall'introduzione del del concetto di granularita' della lock ([GRAY1 '75], [GRAY2 '75]); una seconda, la piu' recente, che ha preso adito dall'uso della conoscenza semantica sulle operazioni ([KOR '83]).

Introduzione del concetto di granularita'

Fin qui abbiamo sempre parlato di dato lockabile come di un dato elementare che puo' essere identificato con un singolo record. In [GRAY1 '75] e' introdotto il concetto di granularita' della lock, indicando con questo termine la quantita' di informazione su cui e' eseguibile un'operazione di lock.

Due granularita' diverse in un sistema per la gestione di basi di dati possono essere il record e il file. E' facile intuire che la scelta di una granularita' "fine" permette una maggiore concorrenza di una granularita' "grossolana". Se la granularita' della lock e' il record, due transazioni che vogliono modificare records distinti di uno stesso file possono essere eseguite concorrentemente, se la granularita' e' il file la loro esecuzione e' sicuramente seriale. Poiche' ad ogni unita' lockabile devono essere associate le strutture dati per l'implementazione dell'operazione di lock ed ogni transazione deve eseguire tante lock quante sono le unita' lockabili che accede, gli svantaggi della scelta di una granularita' fine sono una maggiore occupazione di memoria ed un maggiore overhead computazionale. Si deve percio' trovare un compromesso tra la concorrenza che si vuole ottenere e il prezzo che si e' disposti a pagare.

La scelta ottimale dipende soprattutto dal tipo di transazione, se una transazione accede solo pochi records di un file puo' convenire scegliere come granularita' il record permettendo cosi' ad altre transazioni dello stesso tipo di operare concorrentemente sul file. Se invece accede a molti records del file la scelta migliore e' quella di lockare l'intero file con un'unica operazione di lock, in questo modo si guadagna in efficienza senza perdita di concorrenza, che sarebbe comunque impedita dal tipo di transazione.

Da quanto detto si deduce che e' utile avere dei metodi che permettano una molteplicita' di granularita' di lock. Il sistema o l'utente dovrebbe essere in grado di scegliere la granularita' migliore da adottare in base ad un'analisi della transazione. Questo e' il punto critico dell'approccio in quanto non si tratta di una scelta facile soprattutto se si considera che non e' sufficiente conoscere le caratteristiche della singola

transazione in causa, ma e` necessario conoscere la situazione globale di tutte le transazioni del sistema. Nel seguito accenniamo a due tecniche che adottano questo concetto e lo applicano a sistemi diversi: nel caso dei Predicate/Expression locks ([ESW '76], [KLU '83]) il sistema e` un database relazionale, nel caso del Locking Gerarchico ([GRAY1 '75]), [GRAY2 '75]) si tratta di un database con struttura gerarchica. Entrambi i metodi impongono alle transazioni di seguire il protocollo standard di 2PL per l'acquisizione e il rilascio delle lock.

Predicate Locks e Expression Locks

Nei predicate locks il sistema considerato e` un database relazionale. In questi sistemi le operazioni che si possono eseguire interessano un sottoinsieme di dati formato dalle tuple che soddisfano un certo predicato. L'idea del metodo e` di scegliere come granularita` della lock proprio l'insieme dei dati cosi` identificato, da cui il nome di predicate locks.

Usando i predicate locks le tuple che si lockano sono proprio tutte e solo quelle che soddisfano il predicato in ogni relazione. Notiamo che di questo insieme fanno parte non solo tuple esistenti ma anche tuple che si potranno creare successivamente e che soddisfano il predicato.

Un'implementazione di questa tecnica deve basarsi su un modulo software che riceve le richieste di predicate locks dalle transazioni, mantiene in una tabella il predicato delle predicate locks concesse, ed accetta o rifiuta una lock a seconda che questa sia o no in conflitto con altre locks concesse. Per vedere quando due locks sono in conflitto si deve calcolare l'intersezione degli insiemi di tuple identificati dai rispettivi predicati. Si dimostra che tale problema e` indecidibile.

In [ESW '76] per rendere la tecnica realizzabile, e` stata scelta una classe di predicati per i quali e` assicurata la decidibilita` del problema dei conflitti. I predicati considerati sono detti "semplici" ed hanno le seguenti caratteristiche:

1) ogni predicato coinvolge una sola relazione;
2) ogni predicato e` una combinazione booleana di predicati elementari, detti "atomici", della forma:

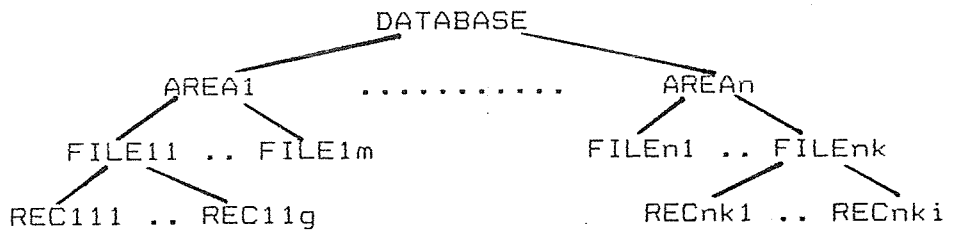
predicato elementare =
<nome di un campo della rel.> { <,=,/=,> } <costante>.

In [KLU '83] sono studiati gli algoritmi per la risoluzione dei conflitti nel caso in cui si ammettano predicati che per essere valutati richiedono l'esecuzione di un join fra due relazioni. Tali predicati sono detti "espressioni" e la tecnica risultante e` nota con il nome expression locks. In questa sede non ci soffermeremo oltre sull'argomento perche` i risultati di questi studi rivestono un particolare interesse nell'ambito dei database relazionali e non sono di utilita` generale.

Locking Gerarchico

L'altra tecnica che adotta il concetto di granularita` di lock e` quella del **Locking Gerarchico**, in cui i dati sono organizzati in una gerarchia e gli utenti possono richiedere lock con granularita` diverse stabilite in base alla gerarchia stessa. L'esempio di gerarchia riportato in [GRAY1 '75] e [GRAY2 '75], in cui si e` interessati ad un sistema per la gestione di basi di dati, e` il seguente:

La struttura e` ad albero: i dati sono organizzati in records, i records in files, i files in aree e tutte le aree sono organizzate in un unico database come illustrato in figura:



Si puo` lockare ogni nodo della gerarchia in modo esclusivo richiedendo una **exclusive-lock** (la cui semantica e` equivalente a quella della modalita` write-lock) o condiviso richiedendo una **shared-lock** (la cui semantica e` equivalente a quella della modalita` read-lock). Per l'organizzazione gerarchica che abbiamo stabilito sui dati, se si locka un nodo in maniera esclusiva (o condivisa) si ha accesso esclusivo (o condiviso) al nodo e implicitamente a tutti i suoi discendenti. Ad esempio exclusive-lock(AREA1) concede l'accessso esclusivo a tutti i records di tutti i files dell'AREA1.

Occorre dunque fare attenzione alle lock poste implicitamente e prima di lockare un nodo N occorre impedire che N venga implicitamente lockato da una lock su un suo predecessore. A questo scopo viene introdotta una nuova modalita` di lock, la **intention-lock**, il cui compito e` quello di segnalare nei livelli piu` alti della gerarchia che c'e` una lock a livello piu` fine. Il protocollo di locking e` il 2PL con il vincolo che prima di lockare un nodo in modo esclusivo o condiviso si devono lockare tutti i suoi antenati in modo **intention**. Inoltre nel caso in cui non e` previsto il rilascio delle lock tutte insieme alla terminazione della transazione, le lock devono essere rilasciate

dal livello piu` basso a quello piu` alto della gerarchia.

Per garantire il comportamento desiderato e` necessario che la intention-lock non sia compatibile ne` con la modalita` exclusive-lock ne` con shared-lock, secondo la definizione di compatibilita` data nel paragrafo 4.1.1. Se indichiamo con I il tipo intention-lock, con S il tipo shared-lock e con X il tipo exclusive-lock, la Tabella di Compatibilita` per le modalita` di lock indicate e`:

	I	S	X
I	si	no	no
S	no	si	no
X	no	no	no

Ricordiamo che l'indice di riga indica la modalita` della lock richiesta mentre l'indice di colonna la modalita` di lock gia` concessa.

Per aumentare la concorrenza possiamo affinare la modalita` intention-lock distinguendo ad esempio in:

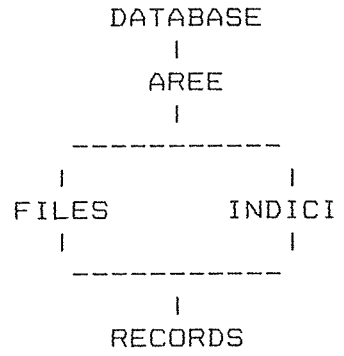
intention-shared-lock con il significato di notifica di una lock di tipo shared ad un livello piu` basso della gerarchia;

intention-exclusive-lock con il significato di notifica di una lock di tipo exclusive ad un livello piu` basso della gerarchia;

shared and intention-exclusive che locka in modo S il nodo stesso e, implicitamente, tutti i suoi discendenti e riserva all'utente la possibilita` di poter lockare in modo X qualche discendente.

Avremo cosi` una nuova tabella di compatibilita` delle lock e un nuovo protocollo di locking che tenga conto delle modalita` aggiunte.

Sempre in [GRAY1 '75] e [GRAY2 '75], e` mostrato come la gerarchia ad albero fra i dati puo` essere estesa ad un grafo.



Se i dati sono organizzati a grafo il protocollo di locking va leggermente modificato per tener conto che possono esistere piu` cammini che portano dalla radice ad un nodo.

Come abbiamo visto il locking gerarchico permette all'utente di richidere lock con granularita` diverse stabilite in base alla gerarchia sull'organizzazione dei dati allo scopo di ottimizzare l'uso delle risorse del sistema. Il metodo del locking gerarchico, inoltre, permette un grado di concorrenza tanto maggiore quanto piu` e` spinta la gerarchia nell'organizzazione dei dati.

Modalita` di Lock

Passiamo ora ad illustrare la seconda direzione di sviluppo delle tecniche di locking, ovvero i meccanismi che usano la conoscenza semantica sulle operazioni.

Un primo meccanismo di questo genere e` stato proposto da Korth ([KOR '83]). Esso si basa sui concetti di modalita` di lock e compatibilita` fra le varie modalita`. Egli suggerisce di considerare un insieme molto piu` vasto di modalita` rispetto alle tradizionali Read-lock e Write-lock. Ad esempio e` proposto di associare una modalita` ad ogni operazione elementare. Sono inoltre definiti dei modi, come il modo Intention, che, combinati con altre modalita`, generano ancora modalita`. E` poi sviluppata una teoria per la definizione della funzione di compatibilita` di tutte le modalita` cosi` ottenute. I risultati possono vedersi come una formalizzazione di teorie gia` note. Tuttavia l'insieme dei tipi di lock che si ottiene applicando la teoria e` tanto vasto e non e` realistico pensare che l'utente sia in grado di scegliere la modalita` di lock ottimale in una certa situazione. Korth delega questo compito ad un modulo di "compilazione" di transazioni, ma non specifica come tale compilatore deve funzionare. Alcune delle sue idee sono riprese nei meccanismi di locking type-specific (paragrafo 4.1.4).

4.1.4 Meccanismo di locking type-specific

I meccanismi di locking che finora abbiamo considerato prevedono l'uso di un insieme di modalita` di lock e considerano

il 2PL il protocollo standard che le transazioni devono seguire per l'acquisizione e il rilascio delle lock. Lo schema piu' semplice di protocollo usa un solo tipo di lock, in [ESW '76] e [BERN '81] troviamo la suddivisione in read-lock e write-lock, in [GRAY '75] e' introdotta la modalita' intention e in [KOR '83] l'insieme delle modalita' di lock e' ampliato ulteriormente. Se e' seguito il protocollo 2PL e' garantita la proprieta' di serializzabilita' degli schedules sia nei sistemi centralizzati che in quelli distribuiti ([TRAI '82]). La serializzabilita', del resto, e' stata considerata per molto tempo coincidere con la proprieta' di correttezza delle esecuzioni concorrenti.

Schwarz e Spector ([SC-SP '84]), hanno individuato una condizione meno stringente per garantire la correttezza degli accessi concorrenti ad un tipo di dato nella proprieta' di ordinabilita' degli schedules rispetto ad un insieme di dipendenze specifiche per quel tipo di dato. Essi hanno introdotto la definizione di tipo di dato astratto condiviso in cui, oltre alla definizione del tipo di dato astratto, sono elencate le dipendenze rispetto a cui garantire l'ordinabilita' (paragrafo 3.2).

La tecnica proposta da usare nell'implementazione per sincronizzare le operazioni sui tipi di dato astratti condivisi, e' ancora basata su un meccanismo di locking. Con le lock infatti si puo' controllare la formazione di dipendenze fra transazioni, limitando gli schedules ottenibili a quelli desiderati. Quando una transazione ti attende per acquisire una lock incompatibile con una gia' posseduta da un'altra transazione tj, la formazione della dipendenza fra le due transazioni e' ritardata fino a quando tj non rilascia la lock.

In questo metodo per ogni tipo di dato e' definito un insieme di modalita' di lock specifiche del tipo e una transazione, per eseguire un'operazione, deve rispettare un protocollo di acquisizione e rilascio delle lock anch'esso specifico per il tipo di dato e per l'operazione.

Schwarz e Spector criticano i tradizionali meccanismi di locking perche', limitandosi alla suddivisione nelle due modalita' di lock, read-lock e write-lock, sfruttano poco la conoscenza semantica che si ha del tipo di dato. La loro idea e' invece quella di avere un meccanismo di locking specifico per il tipo di dato che utilizzi tutta la conoscenza ricavabile dalle dipendenze. Dalle dipendenze, infatti, si puo' derivare il modo in cui le operazioni di piu' transazioni possono essere "interleaved" in un'esecuzione concorrente, preservando la correttezza degli schedules.

Nello schema di locking proposto i tipi delle operazioni e le modalita' di lock sono poste in corrispondenza uno a uno e vengono definite tante classi di lock quanti sono i tipi delle operazioni permesse sul tipo di dato. Poiche' un'operazione si distingue non solo per il suo tipo ma anche per i suoi parametri, e' introdotto il concetto di istanza di lock che tiene

in considerazione il dato su cui la lock e' richiesta. Un'istanza di lock consiste di due parti:

1) la classe di lock;

2) l' oggetto (il dato) a cui e' applicata l'operazione di lock.

La notazione usata per indicare un'istanza di lock e':

{Classe di lock (oggetto)}.

Le modalita' di lock sono definite in modo da sfruttare tutte le informazioni implicite nelle dipendenze e con una sintassi che riflette il modo in cui le dipendenze stesse sono date.

Una volta definite le istanze di lock occorre fornire la loro Tabella di Compatibilita' e il protocollo per l'acquisizione e il rilascio delle lock specifico per l'esecuzione di ogni operazione del tipo di dato. Per quanto riguarda la Tabella di Compatibilita' si puo' dire che due istanze di lock sono compatibili se la dipendenza che contiene i due tipi di operazione corrispondenti alle modalita' di lock richieste, e' non importante. Per quanto riguarda il protocollo puo' essere usato il tradizionale 2PL; si scegliera' un protocollo type specific se cio' puo' incrementare la concorrenza.

Nel paragrafo 3.2 abbiamo visto la definizione del tipo di dato astratto condiviso Coda FIFO ora mostriamo il meccanismo di locking type specific per questo tipo di dato.

Coda FIFO

Le operazioni definite sulla Coda FIFO sono:
- QEnter (coda, elem);
- QRemove (coda).

Ci saranno allora due classi di lock, QEnter e Qremove corrispondenti ai due tipi di operazione, e poiche' le lock sulla coda devono identificare l'elemento particolare a cui si riferisce l'operazione per cui e' richiesta la lock, la notazione per le possibili istanze di lock e':

- {QEnter (e)}-lock;
- {QRemove (e)}-lock.

Una {QEnter (e)}-lock e' richiesta quando un elemento con identificatore e deve essere inserito nella coda. Una {QRemove (e)}-lock e' richiesta quando un elemento con identificatore e deve essere rimosso dalla coda.

Il protocollo di acquisizione e rilascio delle lock e' il seguente:

1) prima di eseguire un'operazione di QEnter si deve

ottenere una {QEnter (e)}-lock, dove e e' l'identificatore assegnato all'oggetto da inserire. Tale lock e' mantenuta fino al commit della transazione.

2) prima di eseguire un'operazione di QRemove si deve ottenere una {QRemove (e)}-lock, dove e e' l'identificatore dell'elemento in testa alla coda. Tale lock e' mantenuta fino al commit della transazione.

Se indichiamo con QE la modalita' di lock QEnter e con QR la modalita'di lock QRemove, la Tabella di Compatibilita' per le lock e' la seguente:

	QE(e)	QR(e)
QE(e)	/	/
QE(e1)	no	si
QR(e)	no	/
QR(e1)	si	no

La sbarra nella tabella sta ad indicare situazioni che non possono mai presentarsi perche' gli identificatori degli elementi sono unici.

La Tabella di Compatibilita' riflette la limitazione della concorrenza imposta dal tipo di dato stesso. Ad esempio una {QEnter (e1)}-lock richiesta da una transazione ti e' posta incompatibile con una {QEnter (e)}-lock gia' concessa ad una transazione tj, perche' fra i vincoli di consistenza del tipo di dato vi e' quello che elementi inseriti dalla stessa transazione, appaiano in posizioni contigue della coda. Se tj richiede l'operazione QEnter di un'altro elemento e2 nella coda dopo che ti ha eseguito QEnter (e1), fra e ed e1 inseriti da tj si trova e1 inserito da ti.

Un esempio interessante di sistema che fornisce il supporto alle transazioni al livello del sistema operativo e che adotta un meccanismo di type specific locking per il controllo della concorrenza e' il T.A.B.S. (Transaction Based System) ([SPEC '85]). L'utente di T.A.B.S. definisce gli oggetti astratti, ovvero i dati e i dispositivi di I/O, come tipi di dato astratti condivisi e il sistema stesso garantisce la correttezza degli accessi concorrenti agli oggetti.

4.2. Meccanismi basati sui timestamps

Presentiamo ora una categoria di meccanismi per il controllo della concorrenza che, per le loro caratteristiche, vengono usati quasi esclusivamente in sistemi distribuiti. L'architettura del sistema distribuito che prendiamo a modello e' quella presentata nel capitolo 1. Ricordiamo che tale architettura prevede su ogni nodo della rete due moduli software: il Transaction Manager (TM) e il Data Manager (DM) con le funzionalita' illustrate e che in alcuni casi uno dei due puo' mancare. La letteratura a cui faremo riferimento e': [BERN1 '80], [BERN2 '80], [BERN '81] e [REED '83].

Descrizione del metodo

Il **timestamp** e' una marca numerica che viene assegnata ad ogni transazione che ha inizio dal suo TM. La marca ha la proprieta' di essere unica in tutto il sistema.

L'idea che sta alla base di tutti i meccanismi che fanno uso di timestamps e' quella di stabilire a priori un ordinamento fra le transazioni e di imporre che gli schedules concorrenti siano equivalenti allo schedule seriale in cui le transazioni sono eseguite nell'ordine prestabilito ([BERN '81]). Gli schedules che si possono ottenere sono tutti serializzabili e con lo stesso ordine di serializzazione dato dall'ordinamento numerico crescente dei timestamps.

Il **timestamp** di solito e' ottenuto concatenando il valore dell'orologio di macchina del nodo su cui la transazione ha inizio con l'identificatore del nodo stesso. Imponendo che nello stesso nodo non inizi piu' di una transazione nello stesso ciclo di clock si assicura l'unicita' dei timestamps. L'ordine di serializzazione previene l'attesa infinita perche' ogni transazione puo' essere preceduta nell'ordinamento solo da un numero finito di altre transazioni. Se si mantiene fra i clock dei vari nodi un certo grado di sincronizzazione l'ordinamento rispecchia anche l'ordine temporale di inizio delle transazioni.

Per forzare l'esecuzione concorrente a seguire l'ordinamento i TMs provvedono ad associare ad ogni operazione il timestamp della transazione che la richiede e i DMs eseguono le operazioni in conflitto nell'ordine indicato dai timestamps. Ricordiamo che due operazioni sono in conflitto se operano sullo stesso dato ed una di queste e' una scrittura. Se l'esecuzione di un'operazione viola l'ordinamento, la transazione richiedente e' fatta abortire. In questo modo e' come se non fosse mai stata eseguita nessuna sua operazione ed essa scompare dall'ordinamento. Indichiamo con TS(t) il timestamp associato alla transazione t.

Esempio

Siano t1 e t2 transazioni con TS(t1) < TS(t2); lo schedule indicato comporta l'aborto di t1 perche' t1 vuole modificare un

dato gia` letto da una transazione "piu` giovane", che deve seguire t1 nell'ordine di serializzazione.

t2: read (x)
t2: <commit>
t1: write (x)

Quando una transazione viene abortita il suo TM le assegna un nuovo timestamp e la fa ripartire. Nella versione base del metodo valgono le regole seguenti per l'esecuzione delle operazioni:

- una richiesta di lettura di un dato da parte di una transazione e` accettata solo se nessun'altra transazione con timestamp piu` grande ha eseguito una scrittura del dato;

- una richiesta di scrittura e` accettata solo se nessun'altra transazione con timestamp piu` grande ha letto o scritto il dato.

L'implementazione si basa sulla costruzione di un modulo software, lo scheduler, che riceve dai TMs le richieste di lettura/scrittura e, solo se e` permessa la loro esecuzione, le inoltra ai DMs interessati, altrimenti fa abortire la transazione richiedente. Lo scheduler deve mantenere per ogni dato x l'informazione su:

- il timestamp maggiore di tutte le operazioni di lettura eseguite sul dato, indichiamolo con R-TS(x);

- il timestamp maggiore di tutte le operazioni di scrittura eseguite sul dato, indichiamolo con W-TS(x).

L'algoritmo applicato dallo scheduler e` il seguente:

1) se arriva una richiesta di lettura di x con timestamp TS e TS < W-TS(x) fa abortire la transazione che ha fatto la richiesta, perche` il dato che essa vuole leggere e` gia` stato modificato da una transazione che la deve seguire nell'ordinamento; altrimenti accetta la richiesta, modifica R-TS(x) ponendo R-TS(x) = max (R-TS(x), TS) e invia la richiesta al DM opportuno.

2) se arriva una richiesta di scrittura di x con timestamp TS e TS < R-TS(x) la transazione che ha fatto la richiesta e` abortita. Il dato che dovrebbe essere modificato e` gia` stato letto da una transazione che la segue nell'ordinamento imposto e lo schedule risultante non soddisferebbe l'ordine di serializzazione. Se TS < W-TS(x) la transazione e` abortita perche` in questo caso e` gia` stata eseguita una scrittura successiva sul dato. Se non si verifica nessuna delle due condizioni precedenti lo scheduler accetta la richiesta, pone W-TS(x) = TS e invia l'operazione al DM opportuno.

Lo scheduler, per la conoscenza di cui deve disporre e l'algoritmo che deve eseguire, puo` essere completamente

distribuito sui vari nodi del sistema. Una scelta possibile e' quella di associare lo scheduler ai DMs; in questo modo ogni scheduler gestisce le richieste di operazione per i dati mantenuti dal relativo DM.

La memorizzazione di due timestamps per ogni dato puo' risultare pesante in termini di occupazione di memoria. Sono stati studiati degli accorgimenti che permettono di non mantenere tutti i timestamps. In [BERN '81] ad esempio, e' mostrato un metodo per ridurre il numero dei timestamp da mantenere ad una quantita' prestabilita e indipendente dal numero dei dati.

Un inconveniente del metodo e' quello di rendere visibili i risultati parziali delle transazioni: si possono percio' avere aborti in cascata di transazioni oppure, se la situazione e' incontrollata, dei veri e propri errori per dati fantasma. Supponiamo di avere due transazioni t1 e t2 con TS(t1) < TS(t2) e che R-TS(x) <= W-TS(x) = 0. Il DM che gestisce il dato x puo' ricevere la seguente sequenza di operazioni:

t1: write (x)
t2: read (x).

Se dopo l'esecuzione delle operazioni, t2 termina con successo e t1 fallisce si verifica un errore dovuto al fatto che t2 ha letto un valore scorretto del dato. Una soluzione al problema e' quella di vincolare una transazione con timestamp K a terminare solo dopo che sono terminate con successo o con fallimento tutte le transazioni con timestamp minore di K. Dal punto di vista implementativo la soluzione ha un costo notevole. Ogni TM deve infatti conoscere i timestamps di tutte le transazioni attive nel sistema e ogni volta che una transazione di cui e' il gestore termina o abortisce deve inviare il messaggio relativo in broadcast a tutti gli altri TMs.

Un altro inconveniente e' il numero elevato di aborti di transazioni, molti dei quali sono dovuti solo al meccanismo di sincronizzazione e non sarebbero necessari per un corretto controllo della concorrenza. Gli aborti ingiustificati derivano dall'imporre a priori un ordine di serializzazione fra tanti possibili. Se t1 e t2 sono due transazioni in conflitto per il solo dato x e TS(t1) < TS(t2), il meccanismo di sincronizzazione impone di eseguire su x prima le operazioni di t1, poi quelle di t2. Nel caso in cui per i ritardi della rete arrivino prima le richieste di operazione relative a t2, t1 viene successivamente fatta abortire anche se sarebbe ugualmente corretto un ordine di serializzazione in cui t2 precede t1. Infine e' da notare che, per come e' definito il metodo, anche le letture inducono una forma di aggiornamento dei dati per l'eventuale modifica di R-TS(x) del dato letto.

I meccanismi di controllo della concorrenza che fanno uso dei timestamps sono stati pensati per i sistemi distribuiti; la loro applicazione ai sistemi centralizzati, anche se possibile, non e' mai stata considerata perche', in questo caso, i

meccanismi di locking sono sicuramente piu' efficienti. Nei sistemi distribuiti, invece, stabilendo a priori l'ordine di serializzazione delle transazioni si permette ai nodi di decidere localmente la schedulazione delle operazioni. Il vantaggio e' che il controllo della concorrenza puo' essere completamente distribuito sui nodi senza introdurre attesa per la sincronizzazione anche se, come accennato, puo' essere necessario sincronizzare i nodi nella fase di terminazione.

Nell'ottica di minimizzare il numero dei restarts delle transazioni sono state studiate delle ottimizzazioni da apportare alla versione base del meccanismo. Un'osservazione di Thomas, nota come "Thomas Write Rule", porta ad una prima modifica del metodo. Sia write(x) una richiesta di scrittura del dato x e TS(write(x)) < W-TS(x); Thomas ha osservato che anziche' abortire la transazione richiedente la scrittura si puo' semplicemente ignorare la richiesta. Cio' e' corretto perche' lo stato finale del dato rispecchia la condizione imposta dall'ordine di serializzazione; l'effetto di questa scrittura sarebbe stato annullato in ogni caso dalla scrittura effettuata dalla transazione t, successiva nell'ordinamento, con timestamp W-TS(x).

Nel seguito verranno esaminate le tecniche Multiversion ([REED '83]) e Conservative T/O ([BERN '81]) che tendono anch'esse a diminuire il numero degli aborti e quindi dei restart di transazioni.

Multiversion

In [REED '83] troviamo la proposta di un meccanismo di controllo della concorrenza, il Multiversion, che fa uso dei timestamps. La sua caratteristica e' quella di poter accettare sempre una richiesta di lettura e di essere piu' permissivo della versione base anche per le richieste di scrittura, ammettendo che le modifiche di un dato possano venire eseguite in un ordine diverso da quello indicato dai timestamps.

Per ogni dato e' mantenuta una sequenza di versioni, cioe' di copie. Ogni versione e' caratterizzata dalla coppia di informazioni (valore, TSW), dove TSW e' il timestamp della transazione che ha prodotto quel valore del dato. Ad ogni versione e' associato anche un altro timestamp, TSR, che e' il massimo timestamp fra quelli delle transazioni che hanno letto quella versione del dato. Un dato e' il puntatore alla testa di una lista di versioni, ordinata per timestamps (TSWs) decrescenti, che rappresenta la storia dell'oggetto.

Per non mostrare i risultati parziali di una transazione ogni operazione di modifica crea una versione dell'oggetto cosiddetta possibile. Le versioni possibili sono visibili solo all'interno della transazione che le ha create. Quando una transazione termina con successo tutte le versioni possibili che ha creato divengono permanenti e quindi visibili a tutte le transazioni del sistema.

Vediamo come sono gestite le operazioni di lettura e scrittura di un dato. Se una transazione t con timestamp TS vuole leggere il valore di un dato x, deve cercare nella sequenza delle versioni di x quella creata dall'ultima delle transazioni che precedono t nell'ordinamento dei timestamps (e che hanno effettuato modifiche di x). In altri termini si vede x nel momento della sua storia che a noi interessa. Se la versione selezionata e' stata creata dalla stessa transazione t, si ritorna il valore del dato, altrimenti si guarda lo stato della transazione che ha creato la versione: se e' terminata si restituisce il valore del dato; se invece e' ancora attiva si attende che termini o che abortisca, dopo di che si agisce di conseguenza; se e' abortita si puo' rimuovere la versione dalla lista e riapplicare il procedimento alla versione precedente.

Dopo che e' avvenuta la lettura si deve porre TSR = max {TS, TSR} nella versione selezionata.

Se una transazione t con timestamp TS vuole modificare il dato x, deve selezionare dalla sequenza delle versioni quella con TSW maggiore che non supera TS e controllare il campo TSR. Se TSR > TS vuol dire che la richiesta di scrittura e' arrivata in ritardo rispetto ad un'operazione di lettura di una transazione che segue t nell'ordinamento dei timestamps. Se in questo caso permettessimo di creare una nuova versione di x con TSW = TS avremmo che la/le lettura/e gia' eseguita/e avrebbe/ro fornito alla transazione richiedente un valore di x diverso da quello desiderato. La transazione t deve essere abortita; notiamo che questo e' l'unico caso in cui la tecnica Multiversion forza l'aborto di transazioni. Se TSR <= TS si crea una nuova versione e si inserisce nella sequenza immediatamente prima della versione precedentemente selezionata.

Il metodo proposto presenta un notevole overhead di memoria, perche' per ogni scrittura e' creata una nuova versione, ed un certo overhead di computazione per la scansione della lista delle versioni che puo' provocare accessi multipli al disco ([MOH '86]). Un miglioramento si puo' ottenere gestendo oppurtunamente le versioni; e' infatti inutile mantenere delle versioni a cui nessuno fara' piu' riferimento. Le versioni piu' vecchie si possono eliminare scegliendo appropriatamente un tempo di sopravvivenza, trascorso il quale il sistema stesso provvede a buttare via la versione.

Conservative T/O

Per evitare completamente il problema dei restarts, Bernstein ha proposto una tecnica detta conservativa, in cui una transazione non deve mai essere abortita per non aver rispettato l'ordine di serializzazione imposto dai timestamps.

L'idea e' la seguente: se l'esecuzione di un'operazione puo' provocare dei restarts in futuro, l'operazione e' ritardata finche' non si e' certi che non causera' alcun restart.

La tecnica conservativa richiede che i TMs inviino le richieste di lettura/scrittura nell'ordine dei timestamps; cio' garantisce che, una volta arrivata allo scheduler una read(x)/write(x) con timestamp K dal TMi-esimo, quest'ultimo non inviera' piu' richieste di lettura/scrittura con timestamp minore di K. Occorre anche supporre che i collegamenti di rete siano canali FIFO, cioe' che due messaggi provenienti dallo stesso mittente per la stessa destinazione arrivino sempre nell'ordine in cui sono stati inviati. Lo scheduler mantiene per ogni TMi il minimo timestamp delle operazioni di lettura richieste da quel TM in una variabile min-RTS(TMi) e il minimo timestamp delle operazioni di scrittura nella variabile min-WTS(TMi). Queste variabili valgono -infinito se non sono mai state inviate richieste di operazioni di quel tipo dal TM in esame. Le richieste di operazione ritardate vengono mantenute in un buffer dello scheduler.

La regola di accettazione delle richieste di lettura e' la seguente:

- quando giunge da un TM una richiesta di lettura, read(x), questa e' accettata solo se lo scheduler ha ricevuto richieste di scrittura da tutti i TMi e TS(read(x)) < min-WTS(TMi) per ogni i, altrimenti e' ritardata ponendola nel buffer fino a che non si verifica la condizione di accettazione. Nella versione base del Timestamp Ordering un'operazione di lettura eseguita da t puo' provocare il restart di una transazione t' che precede t nell'ordinamento dei timestamps, se t' richiede successivamente una write per lo stesso dato. Con questo metodo se da ogni TMi sono giunte richieste di scrittura con timestamp maggiore di TS(read(x)), poiche' i TMs richiedono la scrittura nell'ordine dei timestamp, siamo sicuri che nessun TMi richiedera' mai una scrittura con timestamp minore di TS(read(x)) e quindi l'accettazione di read(x) non provochera' mai restarts in futuro.

La regola di accettazione delle richieste di scrittura e' la seguente:

- quando giunge da un TM una richiesta di scrittura, write(x), questa e' accettata solo se lo scheduler ha ricevuto richieste di lettura da tutti i TMi e TS(write(x)) < min-RTS(TMi) per ogni i, altrimenti e' ritardata ponendola nel buffer fino a che non si verifica la condizione di accettazione.

Sia l'esecuzione che la bufferizzazione di una richiesta inviata dal TMi, puo' modificare min-RTS(TMi) o min-WTS(TMi) rendendo eseguibili operazioni precedentemente ritardate. Se il TMj non interagisce per un certo periodo con uno scheduler min-RTS(TMj) e min-WTS(TMj) possono valere -infinito il che impone la bufferizzazione di tutte le read e le write bloccando il funzionamento di quello scheduler. In assenza di operazioni significative i TMs devono percio' inviare operazioni nulle di lettura e di scrittura con il significato che il TM non ha niente da eseguire. Se il timestamp associato alle operazioni nulle e' grande il TM che le ha inviate non intralcia l'esecuzione delle

operazioni degli altri TM.

La tecnica conservativa e' migliore della versione base anche per quanto riguarda l'occupazione di memoria. I timestamps sono associati alle operazioni e non ai dati, non abbiamo dunque da mantenere, per ogni dato x, la coppia (RTS(x), WTS(x)).

Una delle principali critiche a questo meccanismo di controllo della concorrenza e' di essere troppo conservativo perche' si attendono i messaggi da tutti i TM e non solo da quelli da cui possono giungere richieste di operazioni che creano conflitti. In [BERN1 '80], [BERN2 '80] e [BERN '81] sono presentati ulteriori raffinamenti del meccanismo T/O Conservativo, basati essenzialmente su una meggiore conoscenza della semantica delle transazioni. Tali raffinamenti consentono di limitare il numero e la durata delle attese, ma reintroducono il problema del deadlock, non presente nella versione base.

Esperienze implementative hanno mostrato che, in molti casi, le tecniche di locking sono preferibili a quelle basate sui timestamps non solo nei sistemi centralizzati, dove le ultime non hanno ragione di essere, ma anche nei sistemi distribuiti. Ad esempio, la Computer Corporation of America che aveva adottato per le sue macchine DEC il sistema SDD-1 ([BERN2 '80]), basato sul Conservative T/O, lo ha poi abbandonato in favore di un sistema, il DDM, che fa uso di un meccanismo di locking.

5. Approccio ottimistico

In questo paragrafo vedremo un approccio al controllo della concorrenza alternativo a quelli in cui per mantenere la correttezza si sincronizza l'esecuzione delle operazioni delle transazioni.

Nell'approccio ottimistico si permette alle transazioni di evolvere fino al termine senza applicare nessun protocollo di sincronizzazione. Solo quando la transazione e' terminata si va a controllare se ha visto uno stato consistente del sistema e se le eventuali modifiche che queste intende apportare allo stato possono essere rese effettive e permanenti senza perdita di consistenza. Nel caso ottimistico che cio' si verifichi non si deve far altro che rendere permanenti le modifiche nel database, altrimenti la transazione e' abortita e fatta ripartire.

Nei metodi ottimistici non e' imposto nessun vincolo all'esecuzione delle operazioni di lettura perche' non modificano mai lo stato del sistema; l'unica cosa che si controlla e' che i valori che restituiscono rappresentino uno stato consistente dei dati. Per quanto riguarda le scritture, queste sottostanno alla restrizione di essere effettuate in uno spazio di lavoro privato della transazione e di venire riportate nel sistema solo se cio' non causa la perdita di integrita' ([KUN '81]).

Nei metodi ottimistici l'evoluzione di una transazione e` suddivisa in fasi: una fase detta di lettura, una fase di validazione e, solo per le transazioni che effettuano modifiche, una fase di scrittura. Quando una transazione va in esecuzione le viene associato uno spazio di lavoro che e` un'area di memoria privata della transazione.

La fase di lettura corrisponde all'esecuzione completa della transazione con la particolarita` che le eventuali scritture sono effettuate su copie dei dati nello spazio di lavoro. Durante questa fase sono inoltre creati due insiemi: READ-SET e WRITE-SET, in cui sono raccolte informazioni sugli oggetti rispettivamente letti e scritti dalla transazione.

Nella seconda fase la transazione e` sottoposta ad un meccanismo di validazione che controlla la correttezza dell'esecuzione concorrente della transazione avvalendosi degli insiemi READ-SET e WRITE-SET. Questi, infatti, indicano su quali dati le operazioni della transazione possono essere entrate in conflitto con operazioni di altre transazioni. Gli algoritmi proposti per la fase di validazione in genere sono progettati per garantire che l'esecuzione concorrente sia equivalente ad una esecuzione seriale, ma si possono trovare condizioni meno stringenti della serializzabilita` facendo uso di conoscenza semantica sui dati e/o sull'applicazione. Il meccanismo di validazione, per garantire la serializzabilita`, impone che ad ogni transazione ti venga assegnata una marca numerica unica t(i). Lo schedule dell'esecuzione concorrente e` forzato ad essere equivalente all'esecuzione seriale delle transazioni nell'ordine dettato dalle marche:

ti precede tj se e solo se t(i) < t(j).

Sia tj una transazione; affinche` lo schedule goda della proprieta` voluta, per ogni ti tale che t(i) < t(j) si deve verificare una almeno delle seguenti condizioni:

1) ti ha completato la sua fase di scrittura prima che tj inizi la fase di lettura;

2) ti ha completato la sua fase di scrittura prima che tj inizi la fase di scrittura e
WRITE-SET(ti) $\cap$ READ-SET(tj) = 0;

3) ti ha completato la sua fase di lettura prima che tj abbia completato la sua fase di lettura ed e` verificato che:
WRITE-SET(ti) $\cap$ READ-SET(tj) = 0 e
WRITE-SET(ti) $\cap$ WRITE-SET(tj) = 0.

Considerando solo la prima condizione si forza un'esecuzione seriale in quanto una transazione non puo` iniziare prima che tutte quelle che la precedono nell'ordinamento non sono completamente terminate. Aggiungendo la seconda condizione si impone che solo le fasi di scrittura avvengano serialmente nell'ordine indicato dalle marche almeno per quelle transazioni

tra cui non esistono conflitti di tipo scrittura-lettura. Considerando tutte e tre le condizioni si permette che anche le fasi di scrittura possano avvenire concorrentemente per le transazioni tra cui non ci sono conflitti di tipo scrittura-lettura o scrittura-scrittura.

Gli algoritmi di validazione operano garantendo il soddisfacimento di una o piu' di queste condizioni. Un primo meccanismo di validazione mostrato in [KUN '81] guarda se per la transazione t richiedente la validazione, tutte le transazioni con marca inferiore rispettano una delle prime due condizioni elencate sopra ed e' presentato anche un secondo algoritmo che sfrutta il soddisfacimento di tutte e tre le condizioni.

Una volta superata la fase di validazione con successo le transazioni di sola lettura sono completamente terminate mentre le transazioni che contengono operazioni di scrittura entrano nella terza fase. Se la validazione ha esito negativo la transazione deve essere abortita e reinizializzata.

Una scelta da fare nell'implementazione del metodo riguarda l'assegnazione delle marche. In [KUN '81] viene mantenuto un contatore t.n.c., (transaction number counter), globale per il sistema da cui le transazioni ricavano la marca al termine della loro fase di lettura. Con questa scelta l'ordine di serializzazione che si ottiene e' quello in cui le transazioni giungono al termine della fase di lettura. Un'altra possibilita' e' quella di assegnare le marche all'inizio della fase di lettura. Cio' comporta lo svantaggio che, se t1 e t2, iniziate all'incirca insieme, con marche n e (n+1) rispettivamente, sono tali che la fase di lettura di t1 e' molto piu' lunga di quella di t2 , la transazione t2 prima di essere validata deve attendere la fine della fase di lettura di t1 perche' e' necessario conoscere il WRITE-SET di t1. L'assegnazione delle marche e' quindi un punto critico per stabilire l'ordine di serializzazione. Se l'assegnazione avviene all'inizio della fase di lettura cio' corrisponde a stabilire l'ordinamento senza considerare l'evoluzione della transazione, se invece avviene al termine tiene conto dell'ordine in cui le transazioni arrivano al al commit. Nel secondo caso, intuitivamente, il risultato e' piu' significativo che nel primo.

Il metodo puo' comportare **attesa infinita** perche' una transazione t puo' essere reinizializzata un numero qualsiasi di volte. Per evitare questo comportamento, dopo che t e' stata reinizializzata un numero n di volte scelto a priori, si deve prevedere un meccanismo che blocchi l'accesso ai dati alle altre transazioni sospendendo la concorrenza fintanto che t non e' stata eseguita con successo fino alla terminazione.

La prerogativa dell'approccio ottimistico e' quella di avere una buona efficienza nel caso "ottimistico" in cui le transazioni non interferiscono causando comportamenti scorretti. Se infatti non si verificano conflitti la transazione non risente della presenza del controllo di concorrenza; al contrario, in caso di

conflitti che provocano l'aborto della transazione, questa subisce un ritardo generalmente notevole perche' l'aborto avviene dopo che la transazione e' stata eseguita fino al termine.

Dalla considerazione precedente si ricava che questo approccio puo' rilevarsi vantaggioso in quei sistemi in cui la probabilita' di avere conflitti e' bassa. In generale si trattera' di sistemi nei quali la quantita' dei dati acceduti da ogni transazione e' piccola rispetto all'ampiezza del database.

Uno svantaggio notevole dei metodi ottimistici e' l'alto costo di applicazione nei sistemi distribuiti. In primo luogo, infatti, occorre un algoritmo distribuito per l'assegnazione delle marche (per esempio del tipo di quello di Le Lann ([LEL '78])). Cio' che pero' e' piu' pesante da gestire e' la fase di validazione perche', per la certificazione di ogni transazione, l'algoritmo richiede che vengano interrogati tutti i nodi del sistema ([MOH '86]).

Bibliografia

[AGR '85]
R. AGRAWAL, D.J.DEWITT
"Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation"
ACM Trans. on Database Systems 10, 4 Dec. '85.

[BAY '80]
R. BAYER, H. HELLER, A. REISER
"Parallelism and Recovery in Database Systems"
ACM Trans. on Database Systems 5, 2 June '80.

[BERN1 '80]
P.A. BERNSTEIN, D.W. SHIPMAN, J.B. ROTHNIE
"Concurrency Control in a System for Distributed Databases (SDD-1)" ACM Trans. on Database Systems 5, 1 March '80.

[BERN2 '80]
P.A. BERNSTEIN, D.W. SHIPMAN
"The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1)"
ACM Trans. on Database Systems 5, 1 March '80.

[BERN '81]
P.A. BERNSTEIN, N. GOODMAN
"Concurrency Control in Distributed Database Systems"
ACM Computing Surveys 13, 2 June '81.

[ESW '76]
K. ESWARAN, J. GRAY, R. LORIE, I. TRAIGER
"The Notions of Consistency and Predicate Locks in a Database System" Communication ACM 19, 11 November '76.

[GAR '83]
H. GARCIA-MOLINA
"Using Semantic Knowledge for Transaction Processing in a Distributed Database" ACM Trans. on Database Systems 8, 2 June '83.

[GRAY '74]
J. GRAY
"Locking in a Decentralized Computer System"
IBM Research Laboratory, San Jose, California
Research Report RJ1346, February 8, 1974.

[GRAY1 '75]
J. GRAY, R. LORIE, G. PUTZOLU
"Granularity of Locks in a Large Shared Data Base"
IBM Research Laboratory, San Jose, California
Research Report RJ1606, June 30, 1975.

[GRAY2 '75]
J. GRAY, R. LORIE, G. PUTZOLU, I. TRAIGER
"Granularity of Locks and Degrees of Consistency in a Shared Data Base" IBM Research Laboratory, San Jose, California Research Report RJ1654, September 19, 1975.

[GRAY '78]
J. GRAY
"Notes on Database Operating Systems" in "Operating Systems: an Advanced Course" Springer-Verlag 1978.

[GRAY '80]
J. GRAY
"A Transaction Model" IBM Research Laboratory, San Jose, California Research Report RJ2895 (36591) 8/7/80.

[GRAY '81]
J. GRAY
"The Transaction Concept: Virtues and Limitations"
Proc. of Very Large Database Conference Sept. '81.

[KLU '83]
A.KLUG
"Locking Expressions for Increased Database Concurrency"
Journal of the ACM 30, 1 January 1983.

[KOR '83]
H.F. KORTH
"Locking Primitives in a Database System"
Journal of the ACM 30, 1 January 1983.

[KUN '81]
H.T. KUNG, J.T. ROBINSON
"On Optimistic Methods for Concurrency Control"
ACM Trans. on Database Systems 6, 2 June '81.

[LAM '81]
B. LAMPSON
"Atomic Transactions" in "Distributed System Architecture and Implementation: an Advanced Course" Springer-Verlag '81.

[LEL '78]
G. LE LANN
"Algorithms for Distributed Data-sharing System which use tickets" Proc. 3rd Berkley Works. on Distributed Data Management and Computer Networks Berkley, 1978.

[MOH '86]
C. MOHAN
"Tutorial: Recent Advances in Distributed Database and Transaction Systems" Distribuited Operating System: Theory and Practice, NATO Advanced Study Inst., Izmir, Turkey Aug. '86.

[NETT]
E. NETT, K.E. GROSSPIETSCH, J. KAISER, R. KROGER
"Implementing Fault-Tolerance in a Distributed Systems Architecture"
Gesellschaft fur Mathematik und Datenverarbeitung
Schloss Birlinghoven 5205 St Augustin 1.

[PET]
J.L. PETERSON, A. SILBERSCHATZ
"Operating System Concepts"

[ROS '78]
D.J. ROSENKRANTZ, R.F. STEARNS, P.M. LEWIS II
"System Level Concurrency Control for Distributed Database Systems" ACM Trans. on Database Systems 3, 2 June 1978.

[REED '83]
D.P. REED
"Implementing Atomic Actions on Decentralized Data"
ACM Trans. on Computer Systems 1, 1 Febr. 1983.

[SCH '84]
P. SCHWARZ
"Transactions on Typed Objects"
Doct. Diss. Department of Computer Science, Carnegie-Mellon Univ. Pittsburgh Pensylvania '84.

[SC-SP '84]
P. SCHWARZ, A. SPECTOR
"Syncronizing Shared Abstract types"
ACM Trans. on Computer Systems 2, 3 Aug. 1984.

[SPEC '85]
A. SPECTOR et Al.
"Distribuited Transactions for Reliable Systems"
Symp. on Operating System Principles, Sept. '85.

[SP-SC '83]
A. SPECTOR, P. SCHWARZ
"Transactions: A Construct for Reliable Distributed Computing" ACM Operating Systems Review 17, 2 April 1983.

[TRA '82]
I. TRAIGER, J. GRAY, G. GALTIERI, B. LINDSAY
"Transactions and Consistency in Distributed Database Systems" ACM Trans. on Database Systems 7, 3 Sept. 1982.