Tecnologie delle Base di Dati M

Antonio Davide Calì

28 marzo 2014

WWW.ANTONIOCALI.COM
Anno Accademico 2013/2014
Docenti: Marco Patella, Paolo Ciaccia

Indice

Parte I
Gestione delle transazioni

1 Transazioni

1.1 Cosa è una transazione

Una transazione è *un'unità logica di elaborazione* che corrisponde a *una serie di operazioni fisiche elementari* (es. letture/scritture) sul *database*.

Facciamo subito qualche esempio per tentare di capire meglio cosa una transazione sia. Mettiamo di voler trasferire una somma di denaro da un conto corrente ad un altro, le operazioni necessarie da fare sono:

1. Prelevo i soldi dal primo conto corrente

   ```
   UPDATE CC
   SET Saldo=Saldo-50
   WHERE Conto=123
   ```

2. Deposito i soldi nel secondo conto corrente

   ```
   UPDATE CC
   SET Saldo=Saldo+50
   WHERE Conto=235
   ```

La transazione è data dall'insieme delle due operazioni elementari.

Un altro esempio potrebbe essere l'aggiornamento degli stupendi degli impiegati di una sede.

```
UPDATE Imp
SET Stupendio=1.1*Stipendio
WHERE Sede='S01'
```

In questo secondo caso, anche se l'operazione da eseguire è una sola, essa deve essere eseguita su tutti gli impiegati della sede e non può ritenrsi conclusa fintanto che l'aggiornamento non comprende anche l'ultimo impiegato.

1.2 ACID

L'acronimo **ACID** indica le 4 proprietà che il *dbms* deve garantire per ogni transazione:

1. **Atomicity**: una transazione è una unità di elaborazione unica ed indibisibile (atomica). Il *dbms* deve garantire che la transazione debba essere eseguita come un tutt'uno, cioè applicando la proprietà del tutto o niente, cioè la transazione deve essre completata in tutte le sue singole operazioni elemantari oppure deve annullarne l'effetto come se non fossero mai accadute.

2. **Consistency**: una transazione lascia il database in uno stato consistente, cioè il *dbms* deve garantire che nessuno dei vincoli di integrità del database venga violato. Se la transazione termina correttamente deve portare il sistema in un nuovo stato consistente, altrimenti deve riuscire a portare il sistema allo stato iniziale precedente (che si suppone essere consistente).

3. **Isolation**: una transazione deve essere eseguita in maniera indipendentemente dalle altre. Se più transazioni esegono in concorrenza, il *dbms* deve garantire che l'effetto netto sia equivalente a quello di una qualche esecuzione sequenziale delle stesse (cioè come se fossero eseguite in maniera isolata).

4. **Durability**: gli effetti di una transazione che ha terminato correttamente la sua esecuzione devono essere *persistenti* nel tempo. Il *dbms* deve proteggere il database a fronte di guasti.

Si noti che, a volte, potrebbe essere preferibile non far rispettare alcune delle proprietà prima indicate a favore della performance. Una delle prime proprietà che può "cadere" è la **consistenza**: si parla, difatti, molte volte dell' *eventually-consistency* cioè di far rispettare sì la consistenza, ma solo 'eventually' (cioè alla fine), cioè solo in uno stato finale, facendo passare però il sistema nel tempo, attraverso stati non consistenti poichè sicuri che alla fine esso convergerà in uno stato consistente.

1.2.1 Moduli DBMS

Il *dbms* utilizza moduli diversi per riuscire a garantire le proprietà **ACID**.

- **Transaction Manager:** si occupa di gestire le transazioni nella loro interessa. Riceve i comandi SQL e fa in modo di coordinare le transazioni.

- **Logging & Recovery Manager:** si fa carico di gestire la proprietà *atomicity* e *durability*, gestendo dunque la persistenza e la proprietà del tutto o niente.

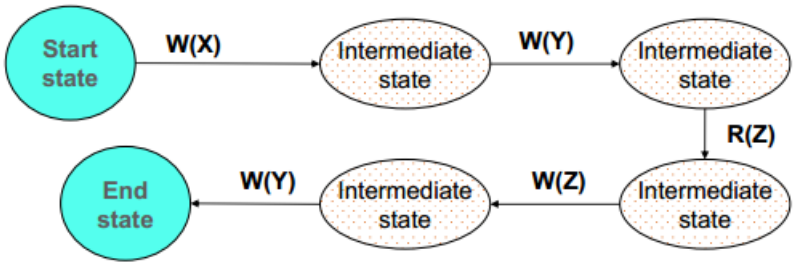
- **Concurrency Manager:** serve a garantire la proprietà di *Isolation*. È un componente simile alla gestione del sistema operativo per gestire la concorrenza.

- **DDL Compiler:** si prende carico della gestione della *Consistency*. In pratica è un compilatore per il linguaggio DDL (Definition Data Language) del database. Esso genera un codice per ogni vincolo di integrità e ad ogni transazione esegue un determinato blocco di codice riguardante le tuple e/o le tabelle che hanno riguardato la transazione controllando che i vincoli di integrità siano ancora rispettati.

1.3 Modello delle transazioni

Nel modello che consideriamo una *transazione* viene vista come una sequenza di operazioni elementari di *lettura* (**R**) e *scrittura* (**W**) di *oggetti* (tuple e/o attributi di tupla) del database che, a partire da uno stato iniziale consistente del database stesso, porta il database in un nuovo stato finale consistente.

Figura 1:



In generale non è richiesto, e non è neanche di interesse, che gli stati intermedi in cui si trova il database siano consistenti (ed è inevitabile che sia così, si veda l'esempio della transazione tra conti correnti).

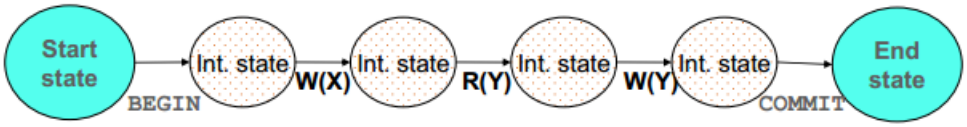
1.3.1 Esiti di una transazione

Nel modello considerato una transazione (il cui inizio viene indicato dalla parola chiave `BEGIN`, anche se in SQL è una word-key implicita) può avere solo due esiti:

1. **Terminare correttamente:**

   Avviene solo quando l'applicazione, dopo aver eseguito tutte le operazioni elementari che gli appartengono, esegue una particolare istruzione SQL detta *COMMIT* (o COMMIT WORK) che comunica "ufficialmente" al *transaction manager* il termine delle operazioni (e quindi di essere giunti ad un nuovo stato 'consistente')

Figura 2:

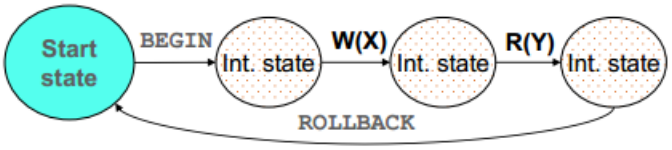


2. **Terminare non correttamente** (anticipatamente):

   La terminazione non corretta può distinguersi ulteriormente in due casi.

(a) È la transazione stessa che, per qualche motivo (si accorge che qualcosa non va), decide di *abortire* poichè non ha più senso proseguire con le azioni, eseguendo una particolare istruzione SQL chiamata *ROLLBACK* (o ROLLBACK WORK). Ad esempio nella transazione di denaro, il primo conto corrente non ha i soldi da trasferire.

Figura 3:

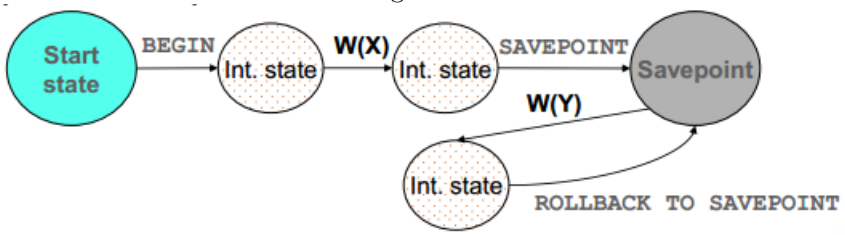


(b) È il sistema che non è in grado di garantire la corretta prosecuzione della transazione che viene dunque abortita. Ad esempio per un guasto al *database* o per la violazione di qualche vincolo di integrità.

In generale se per qualche motivo la transazione non può terminare correttamente la sua esecuzione, il *dbms* deve "disfarsi" (*UNDO*) delle eventuali modifiche apportate dalla transazione al database.

Il modello di transazione usato dai *dbms* è in realtà più articolato, in particolare è possibile definire dei cosiddetti ***savepoint*** che vengono utilizzati, in caso di *rollback*, per non dover tornare allo stato iniziale consistente e "gettare" via tutto il lavoro fatto dalle operazioni, ma per permettere di tornare ad un ultimo stato accettato come "valido": il savepoint stesso.

Figura 4:



Per definire un *savepoint* in **DB2** si usa la seguente sintassi
`SAVEPOINT <name> ON ROLLBACK RETAIN CURSORS`
e per eseguire un *rollback parziale* (dunque ad un savepoint) si utilizza
`ROLLBACK WORK TO SAVEPOINT <name>`

1.3.2 Schedule

Se un *dbms* deve supportare l'esecuzione contemporanea di diverse transazioni che accedono a dati in comune, potrebbe decidere di far eseguire le transazioni in modo diverso: in maniera *sequenziale* o in maniera *interleaved.*

La tecnica **sequenziale** (*serial execution*) prevede effettivamente di far eseguire due transazioni contemporanee una successiva all'altra (sequenzialmente appunto). Chiameremo ***schedule*** la *successione temporale di più azioni elementari.* Ad esempio due transazioni T1 e T2 potrebbero essere eseguite in maniera sequenziale come mostrato in figura (Fig. 5)

Figura 5:

T1	R(X)	W(X)	Commit			
T2				R(Y)	W(Y)	Commit

Alternativamente il *dbms* potrebbe decidere di eseguire più transazioni in concorrenza, cioè utilizzando la tecnica **interleaved** (*interleaved execution*), alternando l'esecuzioni di operazione elementari di una transazione alle operazioni elementari di altre transazioni. Prendiamo le stesse transazioni T1 e T2 mostrate nell'esempio precedente e applichiamo la tecnica *interleaved* (Fig.)

Figura 6:

T1	R(X)				W(X)	Commit
T2		R(Y)	W(Y)	Commit		

Eseguire più transazione concorrentemente è necessario per garantire buone prestazioni, infatti, mettiamo di avere due transazioni, una "breve" e una "lunga": l'esecuzione sequenziale, nel caso sfortunato (prima lunga e poi breve) porterebbe ad un tempo medio di risposta del sistema molto alto, mentre con l'esecuzione concorrente il tempo medio viene ridotto notevolmente. Si sfrutta la possibilità di poter assegnare la CPU ad una transazione mentre una transazione concorrente è in attesa, magari, di una operazione di I/O: il tutto porta ad aumentare il *throughput* (numero di transazioni elaborate nell'unità di tempo) del sistema. Si veda l'esempio in figura (Fig. 7) per una maggiore comprensione.

Figura 7:

- *Supponiamo che T2 venga richiesta a time=2*

time	1	2	...	999	1000	1001	1002	1003	1004
T1	R(X1)	W(X1)		R(X500)	W(X500)	Commit			
T2							R(Y)	W(Y)	Commit

- *Tempo medio di risposta = (1001 + (1004-1))/2 = 1002*

time	1	2	3	4	5	...	1002	1003	1004
T1	R(X1)	W(X1)					R(X500)	W(X500)	Commit
T2			R(Y)	W(Y)	Commit				

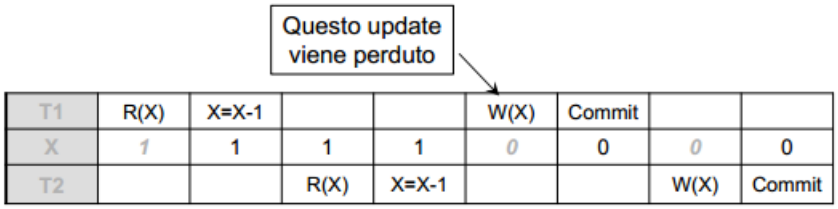
- *Tempo medio di risposta = (1004 + 3)/2 = 503.5*

Finora abbiamo supposto che la proprietà di **isolation** sia garantita in maniera del tutto generale, ma è il *transaction manager* che deve effettivamente gestire la concorrenza in modo tale da non far interferire tra loro le transazioni: infatti, se questo non avvenisse, potrebbero nascere 4 tipi di problemi:

1. **Lost Update:** dovuto ad update concorrenti sul database
2. **Dirty Read:** dovuto a letture di dati "sporchi" (non ancora "*committed*")
3. **Unrepeatable Read:** dovuto all'interleave di letture e scritture
4. **Phantom Row:** dovuto all'inserimento di nuovi dati che potrebbero non comparire nel risultato di una query.

Lost Update Un tipico caso di *lost update* è mostrato dallo *schedule* in figura (Fig. 8) in cui per comodità si evidenziano, oltre alle operazioni elementari delle transazioni, anche i valori che assume il valore X nel database.

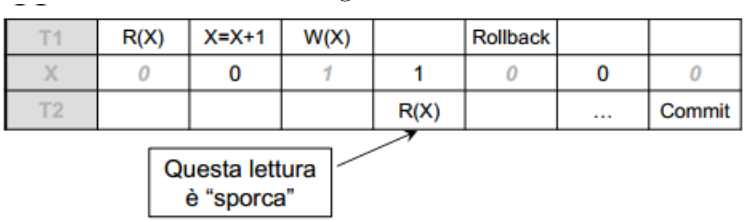
Figura 8:



Il problema nasce perchè T2 legge il valore di X prima che T1 (che ha già letto il valore di X precedentemente e salvato localmente per applicare la modifica X-1) lo modifichi e lo renda permanente (Commit).

Dirty Read Un tipico caso di *dirty read* è mostrato dallo *schedule* in figura (Fig. 9).

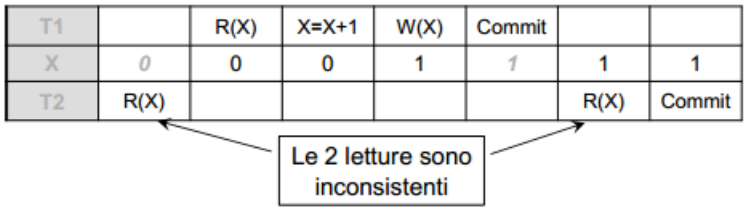
Figura 9:



In questo caso il problema è dovuto al fatto che la transazione T2 legge un dato "che non c'è", cioè T2 legge un valore "sporco" di una transazione (T1) non ancora terminata, dovuto al fatto che T1 richiede un *rollback* facendo perdere tutte le modifiche che aveva apportato: le operazioni svolte da T2 si basano su un valore di X intermedio e quindi non stabile. Le conseguenze sono impredicibili (dipende dalle operazioni di T2) e si presenterebbero anche se T1 non abortisse.

Unrepeatable Read Un tipico caso di *unrepeatable read* è mostrato dallo *schedule* in figura (Fig. 10).

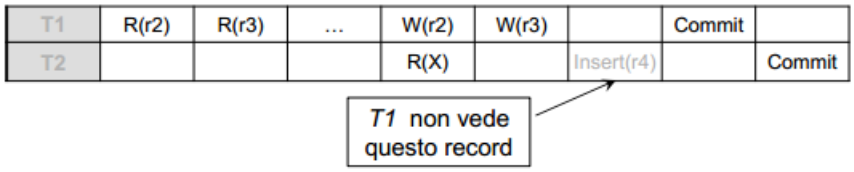
Figura 10:



Il problema è dovuto al fatto che una transazione (T2) legge due volte un dato e trova valori diversi (dati inconsistenti), poichè nel frattempo un'altra transazione (T1) ha modificato il dato. Anche in questo caso si possono avere gravi conseguenze: lo stesso problema si potrebbe presentare per transazioni di "analisi", ad esempio se T1 eseguisse la somma dell'importo fra due conti correnti e T2 eseguisse il trasferimento di fondi da un conto corrente ad un altro, a seconda di quando T1 esegue la sua operazione potrebbe riportare un totale errato.

Phantom Row Un tipico caso di *phantom row* è mostrato dallo *schedule* in figura (Fig. 11).

Figura 11:



Il problema del *phantom row* può verificarsi quando vengono inserite o cancellate tuple che un'altra transazione dovrebbe logicamente considerare. Nell'esempio il record *r4* è "*phantom*" poichè non è visto da T1. Ad esempio se T1 fosse `UPDATE Prog SET Sede='Firenze' WHERE Sede='Bologna'` e T2 fosse `INSERT INTO Prog VALUES('P03', 'Bologna')` l'ultima tupla aggiunta da T2 potrebbe non essere vista dall'aggiornamento richiesto da T1.

1.3.3 Proprietà di uno schedule

Uno schedule può avere (essere) diverse proprietà:

Seriale uno schedule si dice *seriale* se le transazioni sono eseguite in sequenza

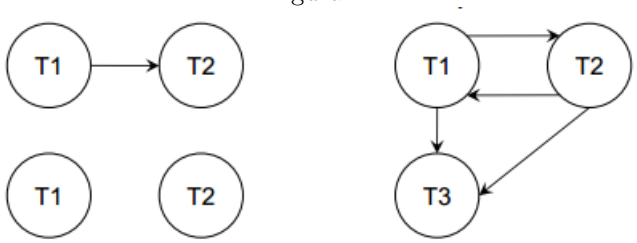
Serializzabile uno schedule si dice *serializzabile* se il suo effetto su una qualsiasi istanza consistente di un database è equivalente all'effetto di uno schedule *seriale* che coinvolga solamente le transazioni terminate correttamente (committed). Date *n* transazioni andate a buon fine (commited) ho *n!* possibili *schedule seriali,* allora uno schedule si dice serializzabile se l'effetto dello schedule è equivalente ad uno di questi *n!* schedule seriali. Questo implica che ci sono tanti *schedule serializzabili* ed ognuno di questo sarà equivalente ad uno dei tanti schedule seriali. **NB** I *schedule serializzabili* sono in numero maggiore ai *schedule seriali* poichè i serializzabili comprendo anche i schedule *interleaved* che hanno però un effetto equivalente a uno schedule seriale.

Ripristinabile (recoverable) uno schedule si dice *ripristinabile* se una transazione T1 che legge un dato modificato da una transazione T2 esegue il commit successivamente al commit di T2. **NB** I schedule ripristinabili sono un sottoinsieme dei schedule serializzabili.

Senza cascata di abort (cascadeless) uno schedule si dice *privo di cascata di abort* se ogni transazione puà leggere solo dati di transazioni terminete correttamente **NB** Uno schedule senza cascata di abort è anche uno schedule ripristinabile.

Stretto (strict) uno schedule si dice *stretto* se ogni transazione non legge o scrive valori usati da altre transazioni attive. **NB** Uno schedule stretto è anche uno schedule senza cascata di abort (e dunque ripristinabile). Implica quindi che due schedule possano lavorare in parallelo solo se agiscono su dati distinti.

Figura 12:



1.3.4 Garantire Isolation

Una comune tecnica usata dai *dbms* per evitare i problemi visti nelle sezioni precedente, consiste nell'uso dei ***lock***. I *lock* ("blocchi") sono un meccanismo comunemente usato dai sistemi operativi per disciplinare l'accesso a risorse condivise. Imporre un lock su una risorsa implica prendersi il diritto (esclusivo) all'uso della risorsa stessa. Per eseguire un'operazione è prima necessario "*acquisire*" un lock sulla risorsa interessata (ad esempio su un record). La richiesta di *lock* è implicita, cioè gestita automaticamente dal *dbms*, dunque non è visibile a livello SQL (anche se vedremo che anche SQL permette qualcosa).

Grafo di serializzabilità Il grafo di serializzabilità è un buon metodo grafico (non implementanto del *dbms* ma utilizzato per la dimostrazione di serializzabilità) che permette di catturare potenziali conflitti tra transazioni di uno *schedule*.

Nel grafo di serializzabilità (Esempio in Fig. 12):

- Ogni nodo rappresenta una transazione commited (terminata), cioè una transazione terminata correttamente (transazioni serializzabili)
- Ogni arco tra T_i e T_j rappresenta un'azione di T_i che precede un'azione di T_j e con essa in conflitto (ovvero se entrambe agiscono su uno stesso dato ed almeno una delle due azioni è di scrittura). Vuol dire che T_i ha un'azione che precede un'azione di T_j e che con questa è in conflitto (e dunque una qualsiasi delle due azioni è una scrittura). Se la freccia e doppia (caso a destra fra T1 e T2) implica che ci sono dei conflitti su due risorse differenti.

Lock Esistono diversi tipi di lock (DB2 ne ha ben 11), ma quelli base e più importanti sono:

- **S** (**S**hared): un lock condiviso è necessario per leggere
- **X** (e**X**clusive): un lock esclusivo è necessario per scrivere/modificare

Il ***Lock manager*** è un modulo del *dbms* che si occupa di tener traccia di quali sono le risorse correntemente utilizzate e di quali transizioni le stiano usando

(e in quale modo). Quando una transazione T vuole operare su un dato Y, viene inviata la richiesta di acquisizione del lock corrispondente (su Y) al lock manager, il quale accorda il lock a T in fuzione della seguente (Fig. 13) tabella di compatibilità.

Figura 13:

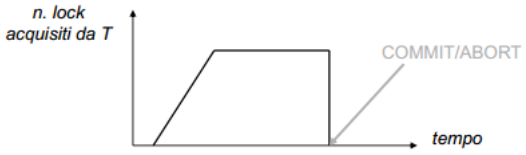
T richiede un lock di tipo	Su Y un'altra transazione ha un lock di tipo: S	Su Y un'altra transazione ha un lock di tipo: X
S	OK	NO
X	NO	NO

Come si vede il lock esclusivo (lock X) è incompatibile con tutti i restanti lock, mentre il lock condiviso (lock S) è compatibile solo con sé stesso. Se su una risorsa Y è presente un lock S e viene richiesto da un'altra transazione un nuovo lock S sulla stessa risorsa essa è concessa, invece se venisse richiesto un lock X esso verrebbe negato; allo stesso modo se su una risorsa Y è presente un lock esclusivo (lock X) qualsiasi altra richiesta di lock (di qualsiasi tipo) sulla risorsa viene negato. Quando T ha terminato di usare Y può (e deve) comunicare il rilascio del look al lock manager attraverso l'*unlock*(Y).

Strict 2-phase lock (Strict 2PL) Il modo con cui le transazioni rilasciano i lock acquisiti è la chiave per risolvere i problemi di concorrenza. Si può dimostrare che l'*Isolation* è garantita se una transazione

1. acquisisce in un primo momento tutti i lock necessari
2. rilascia i lock solo al termine dell'esecuzione (che sia *COMMIT* o *ABORT*)

Figura 14:



Se vediamo il grafo come il tempo di esecuzione della transazione, essa man mano acquisisce i lock a lei necessari, mantenendoli per tutta la sua esecuzione rilasciando solo alla sua terminazione (e dunque non rilascia un lock di una risorsa quando ha terminato di usare la risorsa, ma lo rilascia solo quando la transazione termina). Questo procedimento è chiamato **Protocollo Strict 2-phase lock,** protrocollo stretto di lock a due fasi, questo perchè si può pensare di suddividerlo in due fasi: una fase crescente (acquisizione dei lock) e una fase di rilascio dei lock (al termine dell'esecuzione). Con questo protocollo si

possono evitare alcuni dei problemi prima citati (nel caso speifico il lost update, il dirty read e unrepeatable read) poichè non sarà più possibile che due transazioni *modifichino* dati contemporaneamente e/o avvengano letture sporche. Lo *Strict 2PL* può però avere come effetto collaterale il *deadlock*, ossia situazioni di stallo che vengono però risolte facendo abortire una transazione.

Applicando lo *Strict 2PL* si vede come facilmente vengono risolti i problemi di *lost update*, *dirty read* e *unrepeatable read,* ma vediamolo applicato ad esempi concreti.

Assenza Lost Update Riprendiamo l'esempio precedente (Fig. 8) e applichiamo lo Strict 2PL, lo *schedule* si modifica come segue (Fig. 15)

Figura 15:

T1	S-lock(X)	R(X)	X=X-1				X-lock(X)	wait	wait
X	*1*	1	1	1	1	1	1	1	1
T2				S-lock(X)	R(X)	X=X-1		X-lock(X)	wait

Ora nè T1 nè T2 riescono ad acquisire il lock per poter modificare X restando in attesa (condizione di *wait*). Vediamo infatti che essendo presente sulla risorsa X un lock shared (lock S), in realtà come si vede sono presenti due lock S sulla risorsa, la richiesta del lock esclusivo (lock X) viene negata. Si è verificato però un *deadlock*, ma non avviene più il *lost update*. Se il sistema decidesse di abortire T2, ad esempio, allora T1 sarebbe in grado di proseguire.

Assenza Dirty Read Riprendiamo l'esempio precedente (Fig. 9) e applichiamo lo Strict 2PL, lo schedule si modifica come segue (Fig. 16)

Figura 16:

T1	S-lock(X)	R(X)	X=X+1	X-lock(X)	W(X)		rollback	unlock(X)	
X	*0*	0	0	0	*1*	1	*0*	0	0
T2						S-lock(X)	wait	wait	R(X)

In questo caso l'esecuzione corretta richiede che T2 aspetti la terminazione di T1 prima di poter leggere il valore di X. Come si nota dall'immagine T2 rimane in attesa dopo aver richiesto il lock **S** poichè T1 ha acquisito il lock **X** (non permettendo alcun altro lock sulla risorsa) e ne deve dunque aspettare il rilascio. Effettivamente una volta effettuato l'abort (e dunque il rollback) di T1, esso lascia tutti i lock presenti sulla risorsa, permettendo a questo punto la prosecuzione a T2 che non leggerà dati inconsistenti.

Assenza Unrepeatable Read Riprendiamo l'esempio precedente (Fig. 10) e applichiamo lo Strict 2PL, lo schedule si modifica come segue (Fig. 17)

Figura 17:

T1			S-lock(X)	R(X)	X=X+1	X-lock(X)	wait	wait	wait	W(X)
X	*0*	0	0	0	0	0	0	0	0	*1*
T2	S-lock(X)	R(X)					R(X)	commit	unlock(X)	

In questo caso T1 viene messo in attesa poichè richiede il lock **X** su una risorsa detenuta anche da T2 con il lock **S**. Quando T2 si completa e termina rilasciando il lock allora T1 può riprendere la sua esecuzione e terminare anch'esso.

Proprietà di Strict 2PL Due *schedule* si dicono ***conflict equivalent*** (equivalenti sui conflitti) se:

- Hanno le stesse transazioni
- Le operazioni in conflitto delle transazioni che terminano sono ordinate alla stesso modo

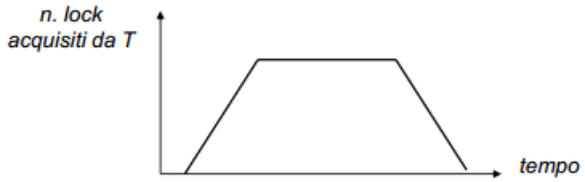
cioè se ordinano allo stesso modo i conflitti.

Si dice che uno *schedule* è **conflict serializable** se è *conflict equivalent* ad uno *schedule serializzabile.* Ogni *schedule conflict seriazable* è serializzabile se non consideriamo il fatto che si possano aggiungere o cancellare dati (per evitare la presenza del phantom row). Uno schedule è conflict serializable *se e solo se* il corrispondente grafo di serializzabilità è aciclico (cioè non presenta cicli). Il protocollo Strict 2PL genera solo grafi aciclici e di conseguenza ogni *schedule* realizzato attraverso il protocollo Strict 2PL risulta essere *conflict serializable* e dunque *serializzabile.* Dunque non solo ogni schedule prodotto da Strict 2PL risulta essere serializzabile, ma evita anche le cascate di abort poichè sempre stretto. Ogni schedule serializzabile è anche conflict serializable con sé stesso.

2PL Introduciamo ora il *protocollo* **2-phase lock (2PL)**. Esso si differenzia rispetto allo *strict 2PL* poichè mantiene il primo vincolo (acquisizione dei lock in un primo momento) ma ne modifica il secondo infatti ora *ogni transazione non può più richiedere un lock dopo averne rilasciato almeno uno.* Questo vuol dire che appena rilascia il primo lock, la transazione non può più acquisirne altri. Nuovamente ci troviamo a dividere ogni transazione in due fasi:

- **Growing**: (fase crescente) in cui vengono acquisiti i lock
- **Shrinking**: (fase calante) in cui vengono rilasciati i lock

Figura 18:



Come si nota dall'immagine (Fig. 18), paragonandola con quella dello *strict 2PL* (Fig. 14), il protocollo 2PL permette alla transazione di rilasciare i lock mentre è ancora in esecuzione e non esclusivamente al suo termine.

Anche il protocollo 2PL genere solo grafi *aciclici* generando quindi sempre *schedule serializzabili*. Se il protocollo *Strict 2PL* genera solo *schedule stretti*, gli schedule del procollo 2PL invece non lo sono, di conseguenza Strict 2PL evita anche le cascate di abort. Tipicamente i *dbms* utilizzano il protocollo *Strict 2PL* poichè gestire la cascata di abort (cosa che nel protocollo 2PL può avvenire) risulta essere molto oneroso e di difficile implementazione.

Ma come è possibile che se sia *Strict 2PL* che *2PL* producano entrambi schedule serializzabili uno sia "migliore" dell'altro, cioè uno non produca la cascata di abort e l'altro sì? Dobbiamo ricordare la definizione di *schedule serializzabile* infatti esso coinvolge solamente le transazioni terminate correttamente non garantendo in alcun modo la presenza o meno di problemi quali l'abort.

Assenza Phantom Row Tra tutti i problemi visti, l'assenza di *Phantom Row* è il più difficile da risolvere. Esistono diverse soluzioni al problema che differiscono per complessità e livello di concorrenza. Vediamone in breve qualcuna:

1. Viene permesso ad una transazione di acquisire un *S-lock* su tutta la tabella e richiedere quindi gli *X-lock* per le tuple che si vogliono modificare (realizzazione semplice ma implica bassa concorrenza)

2. Introdurre un nuovo tipo di lock, denominato *predicate lock*, riguardante tutte le tuple che soddisfano un certo predicato (ad esempio Sede="Bologna") (di più difficile realizzazione poichè è difficile capire quali tuple siano "consistenti" con un predicato, molto simile a ciò che avveninva nel GiST, però di contro permette un'alta concorrenza)

3. Nel caso esista un indice sulla relazione *Sede*, si pone un lock sulla foglia che contiene la voce (chiave) "Bologna" bloccandone in maniera esclusiva (o condivisa) l'accesso alla foglia stessa (poichè in caso di modifica e in caso di esistenza dell'indice, bisogna come prima cosa modificare l'indice, ma se risulta bloccato allora non viene il lock non viene concesso).

Nei *dbms* in realtà la situazione è assai più complessa, sia per i tipi di *lock* presenti sia per la "granularità" con cui i lock possono essere richiesti e acquisiti (a livello di attributo, tupla, pagina, relazione, ...).

1.3.5 Gestione dei lock

Il *Lock Manager* per gestire i lock deve avere a disposizione almeno

- Una *tabella delle transazioni attive* (in realtà è una componente del *Transaction Manager* e il lock manager non fa altro che accedervi) comprendente una lista dei lock posseduti dalle transazioni attive
- Una *tabella dei lock* che indica per ogni "oggetto" il tipo di lock esistente: gli "oggetti" possono essere pagine, record, nodi di alberi, ecc...

La tabella dei lock è composta di entry contenenti diverse informazioni quali:

- l'identificatore *dell'oggetto*
- il *tipo di lock* esistente (S-lock, X-lock)
- il *numero di transazioni* che possiedono un lock. Non è di molto interesse sapere quali transazioni possiedano il lock, ma solo il numero.
- una *coda di richieste* di lock poichè se una transazione si mette in attesa sull'oggetto per un determinato lock è inopportuno che la sua richiesta venga semplicemente rifiutata senza tener conto di un ordine di arrivo delle richieste (gestite proprio dalla coda)

1.3.6 Implementazione protocollo lock

Una transazione può richiedere un lock su un "oggetto" specificandone il tipo. Il Lock Manager, a seconda del tipo di richiesta, può agire diversamente:

- Se la transazione richiede un *S-lock* e la coda delle richieste è vuota e sull'oggetto richiesto non vi è alcun *X-lock*, allora la richiesta è accettata e la *tabella dei lock* viene aggiornata. Se la coda non fosse vuota sarebbe inopportuno garantire alla nuova transazione l'accesso immediato, renderebbe poco "corretto" la presenza in attesa delle altre transazioni.
- Se la transazione richiede un *X-lock* e sull'oggetto richiesto non vi è alcun tipo di lock (dunque la coda di richieste è vuota), allora la richiesta è accettata e la *tabella dei lock* viene aggiornata
- In caso contrario, la richiesta viene respinta ed inserita in coda, e la transazione che ha richiesto il lock viene sospesa.

Al termine di una transazione (implica che ogni transazione deve tenere memoria di quali lock ha richiesto a quali risorse), che finisca con COMMIT o con ABORT, vengono eseguite le seguenti azioni

1. Vengono rilasciati tutti i *lock* che la transazione aveva ottenuto. Questo vuol dire che nella tabella dei lock viene decrementato di uno il numero di lock presenti sugli oggetti che sono stati appena rilasciati. **NB** In caso si tratti di un X-lock è ovvio che il numero massimo è 1, in caso invece

si tratti di un S-lock il numero può essere superiore a 1 (posso avere più S-lock sulla stessa risorsa) e in questo caso se decrementando di uno il numero dei S-lock il numero risulta esser uguale a zero (quindi non vi è più nessun'altra transazione che abbia un S-lock sull'oggetto) allora posso eliminare anche l'*entry* relativa dalla tabella dei lock.

2. Viene esaminata la prima richiesta in coda presente nella coda delle richieste ed eventualmente viene svegliata la corrispondente transazione (che era in attesa)

3. Non vengono considerate le richieste successive presenti nella coda per evitare *starvation* delle transazioni (cioè si esamina solo la prima richiesta nella coda).

Ovviamente l'implementazione dei comandi di *lock* e *unlock* deve garantirne l'*atomicità*: occorre quindi implementare meccanismi di sincronizzazione per permettere l'accesso concorrente (ad esempio i *semafori*).

1.3.7 Gestione dei deadlock

I protocolli di gestione dei *deadlock* sono simili a quelli già visti in altri corsi, utilizzano tecniche di prevezione e tecniche di rilevamento. In entrambi i casi si rende necessario eliminare (abortire) una transazione, sia perchè se sono in situazione di *deadlock* per poterne uscire occorre abortire una trandazione (deadlock detection) sia perchè mi accorgo che la richiesta di una transazione mi fa cadere in deadlock (deadlock prevention). In genere sono preferite le tecniche di rilevamento in quanto le situazioni di *deadlock* sono relativamente rare.

Deadlock prevention Per prevenire il *deadlock* si usa una semplice tecnica di *deadlock prevention:*

1. Si assegna ad ogni transazione una priorità, tipicamente data dal *timestamp*

2. Supponiamo che la transazione T1 richieda un lock sull'oggetto O e che una transazione T2 possieda già un lock su O in conflitto con la richiesta di T1, si può scegliere di operare in due modi:

   (a) Se T1 ha priorità $>$ di T2 allora T1 aspetta, in caso contrario T1$<$T2 allora T1 viene abortita (*wait-die*)

   (b) Se T1 ha priorità $>$ di T2 allora T2 viene abortita, in caso contrario T1$<$T2 allora T1 aspetta alternativa aspetta (*wound-wait*)

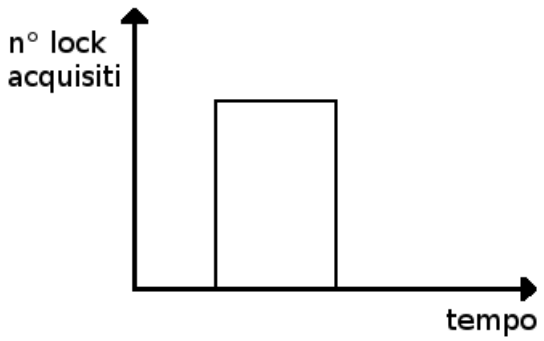
   In entrambi i casi non si possono verificare deadlock.

Quando una transazione è abortita e viene successivamente ricreata deve riuscire ad ottenere lo stesso *timestamp* (e dunque la stessa priorità) originale per riuscire ad evitare il problema della *starvation.*

Notiamo che *wait-die* è *non-preemptive*: ne consegue che T può essere abortita solo perchè richiede un lock e non a causa della richiesta di lock da altre transazione; la *wait-die* viene preferita proprio per questo motivo, cioè non può essere che una transazione che sta eseguendo venga abortita a causa di un'altra, può solo accadere che essa venga abortita poichè richiede un lock a una transazione "più importante". Con il *wait-die* una transazione che possiede tutti i lock di cui necessità **non può mai essere abortita**, di contro una transazione "vecchia" tende a rimanere in attesa (e ad "invecchiare" ulteriormente), mentre una transazione "giovane" può essere ripetutamente abortita fino ad arrivare ad avere una priorità abbastanza elevata da essere messa in attesa.

Conservative 2PL Un'altra politica che permette il *deadlock prevention* si basa sul protocollo di gestione delle transazioni *Conservative 2PL*. Il Conservative 2PL garantisce l'assenza di deadlock senza abortire alcuna transazione. L'idea di base è che una transazione richieda subito (all'inizio della sua esecuzione) tutti i lock di cui necessita: evidentemente quando riesce ad ottenere tutti i lock necessari la transazione prosegue la sua esecuzione senza paura di essere abortita fino a che non termina. Questo previene sicuramente i *deadlock* e le transazioni non possono essere messe in attesa; di contro però proprio perchè richiedere tutti i lock necessari, essi vengono mantenuti più a lungo del necessario (poichè richiedendo all'inizio tutti i lock, uno di essi potrebbe servire solo su una risorsa che viene utilizzata verso la fine), sopratutto se ci sono poche richieste ed inoltre potrebbe richiedere più lock di quelli che in realtà servono (ad esempio se avesse una condizione di scelta, tipo "se succede X fai Y altrimenti fai Z" richiede di acquisire tutti i lock di Y e di Z anche se ovviamente solo uno di questi verrà utilizzato) e dunque riduce il parallelismo, ma in compenso, se di richieste c'è ne sono molte, le transazioni non sono mai in attesa e quindi i lock vengono mantenuti (mediamente) per un tempo minore (rispetto ai protocolli di Strict 2PL, poichè se usassi lo strict 2PL necessito di un metodo di deadlock detection, in più i deadlock durano più a lungo conseguentemente le transazioni rimangono in attesa una volta cominciate e mediamente mantengono il lock per un tempo maggiore rispetto al conservative).

Figura 19:



Deadlock detection I deadlock non si verificano frequentemente e coinvolgono tipicamente poche transazioni. Il *deadlock detection* viene effettuato periodicamente sfruttando il *grafo delle attese* nel quale ogni nodo rappresenta una transazione attiva, e ogni arco uscente da un nodo X_i ed entrante in un altro nodo X_j descrive una transazione in attesa (X_i) a causa del lock posseduto da X_j. Un deadlock corrisponde ad un ciclo all'interno del *grafo delle attese*.

Esiste un modo molto semplice per fare *deadlock detection*: impostare un timer ad ogni transazione, se allo scadere del timer la transazione è ancora in attesa a causa di un lock (molto) probabilmente è perchè ci troviamo in presenza di un deadlock, dunque la transazione viene abortita e al ciclo successivo (cioè quando viene ricreata) le si assegna un timer di durata maggiore (poichè in realtà potrebbe essere stata in attesa proprio perchè la transazione richiede molto tempo computazionale).

Transazione da abortire Per sbloccare una situazione di deadlock occorre scegliere una transazione all'interno di un ciclo ed interromperla. La scelta su quale transazione abortire si può basare su diversi criteri, ad esempio:

- Minor lavoro svolto
- Maggior lavoro ancora da svolgere
- Minor numero di volte in cui la transazione è stata abortita
- Minor numero di lock posseduti. Si potrebbe pensare che sarebbe più utile abortire una transazione con molti lock cosicche molti lock risulterebbero libero, però se una transazione ha molti lock probabilmente è perchè gli servono dunque potrebbe essere poco opportuno abortirla.

e molti altri criteri, ognuno valido e dunque la scelta dipende solo dal progettista.

1.3.8 Controllo concorrenza in SQL

Benchè il problema di gestire transazioni concorrenti sia a carico del *dbms*, con conseguente semplificazione del codice delle applicazioni, SQL mette a disposizione due meccanismi di base per influenzare il modo con cui una transazione viene eseguita. Come già detto la richiesta del *lock* è implicita; tuttavia, poichè acquisire i lock e gestirli richiede tempo e spazio, se una transazione sa di dover elaborare molte tuple di una relazione puà richiedere esplicitamente di porre un lock (ad esempio *SHARE-lock* o *EXCLUSIVE MODE-lock*) sull'intera relazione utilizzando la seguente semantica

```
LOCK TABLE Studenti in SHARE MODE;
SELECT *
FROM Studenti
WHERE DataNascita < '11/07/1982';
```

1.4 Livelli di isolamento

Per alcune applicazioni non è necessario garantire l'assenza di tutti i problemi prima descritti, infatti potrebbe essere preferibile una maggior performance piuttosto che avere dati esattamente precisi. Si prenda per esempio un'applicazione che chiede la somma di tutte le vendite nazionali di un determinato supermercato, non è importante che venga garantito il risultato esatto (quindi che si eviti una dirty read o un lost update) quanto piuttosto è preferibile che l'esecuzione di questa applicazione non infici la performance sul database. Il livello di isolamento di un'applicazione determina il grado di protezione dei dati utilizzati da tale applicazione rispetto alle altre applicazioni eseguite concorrentemente. Scegliere di operare a un livello di isolamento in cui si sa essere possibile la presenza di problemi ha il vantaggio di aumentare il grado di concorrenza raggiungibile e quindi di migliorare le prestazioni.

Lo standard SQL definisce 4 livelli di isolamento, mostrati nella tabella sottostante (vengono riportati anche i nomi usati da DB2 che differiscono dallo standard).

Tabella 1:

Isolation Level	DB2 term	Phantom Row	Unrepeatable Read	Dirty Read	Lost Update
Serializable	Repeatable Read (**RR**)	NO	NO	NO	NO
Repeatable Read	Read Stability (**RS**)	YES	NO	NO	NO
Read Commited	Cursor Stability (**CS**)	YES	YES	NO	NO
Uncommited Read	Uncommited Read (**UR**)	YES	YES	YES	NO

In DB2 il livello di deafult è il Cursors Stability (CS), per cambiarlo (prima di connettersi al database) si usa l'istruzione SQL
`CHANGE ISOLATION TO [RR|RS|CS|UR]`

Isolamento in DB2 Controlliamo ora i vari livelli di isolamento permessi da DB2.

Repeatable Read In questo livello viene posto un lock su tutti i dati usati dall'applicazione. Se si esegue una SELECT su una tabella il lock è posto su tutta la tabella non solo sul risultato

Read Stability In questo livello viene posto un lock su tutti i dati recuperati dall'applicazione. Se si esegue una SELECT su una tabella il lock è posto solo sul risultato

Cursor Stability In questo livello viene posto un lock solo sul dato attualmente utilizzato dall'applicazione. Se si esegue una SELECT su una tabella il lock è posto solo sul record attualmente utilizzato (si ricordi l'inconsistenza tra i dati di SQL, cioè insiemi, e linguaggi ad alto livello che gli insiemi non sanno gestire e quindi si appoggiano al cursor).

Uncommited Read In questo livello l'applicazione è in grado di accedere ai dati *uncommited* (non importa se hanno il lock esclusivo, io ci accedo

ugualmente in lettura con il rischio di avere dati "errati") di altre applicazioni. Risulta utile se si usano solo tabelle *read-only* o istruzioni SELECT

Argomenti avanzati Tutto ciò che finora abbiamo discusso è solo uno dei modi per gestire la concorrenza e risolvere il problema, tipicamente è il modo utilizzato dai dbms relazioni: non è detto però che sia l'unico modo possibile di operare, si veda infatti i sistemi *nosql* che utilizzano l'eventually consistency (prima o poi arriverò in uno stato consistente) con implicazioni sulla consistenza di transazione e il permesso di usare protocolli per gestire la concorrenza più semplici e/o con performance migliori. Bisogna inoltre aggiungere che tutto il "castello" costruito finora ha un costo e in determinati sistemi a bassa concorrenza in cui ci sono poche transazioni che eseguono contemporaneamente, il gioco potrebbe non valere la candela.

Un *dbms* deve essere in grado di gestire la concorrenza a diversi livelli di granularità. Il controllo della concorrenza deve essere effettuato anche sulle strutture di indicizzazione. Nei sistemi a bassa concorrenza *l'overhead* per la gestione dei lock può essere troppo costosa, di conseguenza è possibile affidarsi ad un *controllo ottimistico*, nel quale il controllo viene effettuato alla fine all'atto del commit eventualmente abortendo la transazione (in caso di errori), o ad un *controllo con timestamp*, in cui si crea un ordine tra le transazioni.

1.5 Persistenza

Fino ad ora ci siamo occupati dell'isolamento di una transazione. Come già detto in precedenza spetta invece al *Logging & Recovery Manager* l'incarico di garantire l'*atomicità* e la *persistenza*: sostanzialmente occorre garantire che le azioni effettuate da transazioni terminate correttamente (*committed*) sopravvivano a malfuzionamenti del sistema (come crash) o dei dischi.

Per risolvere questo problema ci viene in aiuto **ARIES** (*Algorithms for Recovery and Isolation Expliting Semantics*). ARIES è una *famiglia di algoritmi* di *locking*, *logging* e *recovery* per la gestione di dati persistenti. Presentato originariamente per *System R*, il *dbms* studiato da IBM, attualmente viene utilizzato da diversi *dbms* tra cui DB2, SQL Server, NT File System, etc...

Esistono essenzialmente 3 tipi di malfunzionamenti che bisogna analizzare:

- **Transaction failure:** è il caso in cui una transazione abortisce e dunque il Recovery Manager deve riuscire ad annullare sul database gli effetti della transazione che è stata abortita, cioè se una transazione che viene abortita ha effettuato delle modifiche al database che sono state scritte su disco è necessario riuscire a ripristinare i valori del database ad uno stato precedente l'esecuzione della transazione stessa.

- **System failure:** accade quando il sistema (o anche il dbms) ha un guasto hardware o software (un crash) che provoca l'interruzione di tutte le transazioni in esecuzione senza però danneggiare la memoria permanente

(i dischi). Quando il sistema riparte i dati salvati sul disco sono ancora tutti presenti, ma bisogna gestire le transazioni che risultavano attive al momento del crash, infatti se una transazione attiva ma non ancora terminata aveva modificato i dati su disco bisogna riuscire a riportare i valori a prima del suo avvio, allo stesso modo se una transazione risultava terminata correttamente (commited) ma i suoi dati non erano ancora stati salvati su disco deve essere possibile riuscire a recuperare le modifiche apportate e renderle persistenti su disco

- **Media/Device failure:** questo tipo di malfunzionamento porta a danneggiare il contenuto (persistente) del databse presente su disco

Come può capitare che una transazione non ancora terminata possa scrivere dati sul disco e che una transazione terminata correttamente invece non abbia ancora reso i dati persistenti? Questo è dovuto ad alcune possibili politiche che il Buffer Manager può applicare. Vediamo dunque come gestire i malfunzionamenti assumendo che la scrittura di una pagina su disco sia un'operazione atomica.

Gestione dei buffer Quando una transazione T modifica una pagina P, il *Buffer Manager* ha due possibilità:

1. *Politica no-steal*: Il buffer manager mantiene la pagina P nel buffer e attende che T abbia eseguito COMMIT prima di scriverla su disco (questo mi garantisce che P sia sempre in memoria).
2. *Politca steal*: Il buffer manager scrive la pagina P quando "più conviene" (ad esempio per liberare il buffer o per ottimizzare le prestazioni di I/O poichè la testina si trova vicino alla pagina in questione), eventualmente anche prima della terminazione della transazione T. Per motivi di efficienza, DB2 adotta questa politica

Esecuzione del Commit Quando una transazione T esegue COMMIT si hanno due possibilità:

1. *Politica force*: quando la transazione T esegue il comando COMMIT, prima di scrivere il record di COMMIT sul Log (che ricordiamo tiene traccia di tutte le modifiche avvenute nel database), che "ufficializza" la conclusione della transazione T, si forza la scrittura su disco di tutte le pagine P modificate da T. In pratica prima di dichiarere terminata la transazione forzo la scrittura delle pagine modificate sul disco (alcune potrebbero anche essere già state scritte poichè utilizzo politica steal).
2. *Politica no-force*: quando la transazione T esegue il comando COMMIT, si scrive subito il record di COMMIT sul LOG; quando T termina alcune delle sue modifiche potrebbero non essere ancora state rese persistenti su disco, dunque la transazione T viene considerata terminata ma le pagine che essa ha modificato non sono ancora state modificate su disco e rese quindi persistenti. Per motivi di efficienza DB2 adotta questa politica.

Atomicità e Persistenza Per far fronte ai malfunzionamenti un *dbms* fa uso di diversi strumenti, in particolare del **Database Dump** che crea una copia di archivio (backup) del *database* (o parte di esso) per far fronte al Media Failure e il **Log File** ("giornale di bordo"), un file sequenziale in cui vengono registrate le operazioni di modifica eseguite dalle transazioni che sarà il punto chiave per garantire persistenza a fronte di Transaction Failure e System Failure.

1.5.1 Log File

Sul **Log File**, gestito attraverso una tabella e quindi contente record, viene scritto un record in seguito a una delle seguenti azioni:

- **Begin:** quando una transazione inizia
- **Update:** aggiornamento di una pagina. Avviene quando una transazione rende "sporca" una pagina (ricordiamo il metodo *setDirty* del buffer manager)
- **Commit:** completamente corretto di una transazione
- **Abort:** completamente errato (Abort) di una transazione
- **End:** terminazione di una transazione (successiva al commit/abort) cioè quando i dati effettivamente vengono resi permamenti su disco
- **Compensation:** registra l'annullamento degli aggiornamenti di una transazione, ad esempio quando una transazione abortisce (per auto-aborto attraverso rollback o abortita dal sistema) bisogna disfarsi delle modifiche effettuate dalla transazione stessa, quindi si effettuano delle modifiche "all'indietro" per riuscire a recuperare i valori iniziali.

Il **Log File** è composto da Record, i quali hanno strutture particolari a seconda dell'azione eseguita: vediamo un esempio di **Record di Update** e di **Record di compensazione**.
Il formato di un *record di update* per una transazione T che modifichi una pagina P del database è il seguente:
`(LSN, prevLSN, T, type, PID, before(P), after(P))`

LSN	*L*og *S*equence *N*umber, è un numero progressivo del record (identifica quindi il record)
prevLSN	identifica LSN del precedente record del LOG relativo alla transazione T in modo da avere i record di una stessa transazione collegati a lista
T	identificatore della transazione
type	è il tipo del record, *update* in questo caso
PID	identificatore della pagina modificata

before(P) "*before image*" di P, ovvero il contenuto della pagina P prima della modifica (utile per annllare le modifiche di una transazione abolita)

after(P) "*after image*" di P, ovvero il contenuto della pagina P dopo la modifica (utile per ripristinare le modifiche di una transazione terminata con successo ma i cui dati non sono stati resi persistenti)

Il *record di compensazione* è usato quando il risultato di un'azione di modifica viene annullato, ad esempio se la transazione abortisce. Il formato di un record di compensazione per una transazione T è il seguente:
`(LSN, prevLSN, T, type, undoNextLSN, PID, before(P))`

LSN Log Sequence Number

prevLSN identifica LSN del precedente record del LOG relativo alla transazione T

T identificatore della transazione

type è il tipo del record, *compensation* in questo caso

undoNextLSN rappresenta il prossimo record da annullare: se stiamo annullando il record U corrisponde al prevLSN di U (poichè stiamo operando all'indietro)

PID identificatore della pagina modificata

before(P) "*before image*" di P, ovvero il contenuto della pagina P prima della modifica

Mostriamo infine un esempio di LOG (Fig. 20) che potrà meglio chiarire le idee.

Figura 20:

LSN	prevLSN	T	type	PID	before(P)	after(P)
...						
235	-	T1	BEGIN			
236	-	T2	BEGIN			
237	235	T1	UPDATE	P15	(abc, 10)	(abc, 20)
238	236	T2	UPDATE	P18	(def, 13)	(ghf, 13)
239	237	T1	COMMIT			
240	239	T1	END			
241	238	T2	UPDATE	P19	(def, 15)	(ghf, 15)
242	-	T3	BEGIN			
243	241	T2	UPDATE	P19	(ghf, 15)	(ghf, 17)
244	242	T3	UPDATE	P15	(abc, 20)	(abc, 30)
245	243	T2	ABORT			
246	244	T3	COMMIT			
247	243	T2	END			
...						

Come si vede ad esempio i record 235, 237, 239 e 240 forniscono una lista (attraverso i prevLSN) di tutte le modifiche effettuate dalla transazione T1.

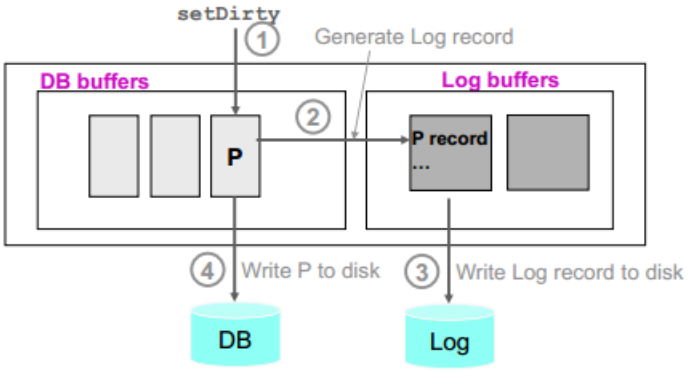
Protocollo WAL Affinchè il LOG possa essere utilizzato per ripristinare lo stato del database a fronte di malfuzionamenti è importante che venga applicato il cosiddetto protocollo **WAL** (Write-Ahead Logging): *prima di scrivere su disco una pagina P modificata il corrispondete LOG record deve essere già stato scritto nel LOG*. Intuitivamente se il protocollo WAL non venisse rispettato potrebbe accadere quanto di seguito:

1. Una transazione T modifica il database aggiornando una pagina P
2. Prima di scrivere il LOG record sul LOG relativo alla modifica di P avviene un system failure

In questa situazione è evidente che non sarebbe in alcun modo possibile riportare il database allo stato iniziale poichè non si hanno informazioni sufficienti (poichè sul LOG non sono presenti) per riuscire a effettuare un "rollback" allo stato iniziale.

La responsabilità di garantire il rispetto del protocollo WAL è del Buffer Manager che gestisce, oltre ai buffer del database, anche i buffer del LOG (che risultano diversi dai buffer del database). In figura (Fig. 21) viene riportato l'ordine in cui si succedono le varie operazioni relative alla modifica di una pagina P.

Figura 21:



L'esecuzione dunque si suddivide in:

1. Una transazione avvisa il Buffer Manager che la pagina P è sporca (attraverso il *setDirty*) poichè è stata modificata
2. Il Buffer Manager accede ai buffer del Log e scrive il record di LOG relativo alla modifica sul LOG stesso
3. Il LOG precedentemente caricato in memoria può essere scritto su memoria stabile (anche in un momento successivo), che come si nota è un disco diverso da quello su cui è presente il database

4. Infine, a seconda delle politiche attuate dal Buffer Manager, la modifica della pagina P viene resa persistente e scritta su disco.

Come detto il LOG deve essere scritto su *memoria stabile* poichè è evidente che il log deve sopravvivere sia al system che al media failure. La memoria stabile può essere realizzata mantendendo copie delle informazioni (eventualmente in luoghi diversi) su dispostivi permanenti (eventualmenti diversi), adottando tecniche di RAID, mirroring, bit di parità, ...

Notiamo che il disco del LOG è inoltre diverso dal disco del database: oltre ad essere una memoria stabile, essendo il LOG un file sequenziale (quindi i record vengono sempre aggiunti in fondo), il tempo di latenza è molto basso poichè la testina non ha bisogno di muoversi, rimane fissa nella posizione in cui è pronta a scrivere nel blocco successivo all'ultimo che ha già scritto, dunque è preferibile avere un disco "sicuro" piuttosto che "performante".

Il LOG permette al Recovery Manager di annullare (azione di *undo*) le azioni di transazioni abortite e incomplete, e di ri-effettuare (azione di *redo*) le azioni di transazioni terminate correttamente (commited) ma le cui modifiche non sono ancora state rese consistenti. **NB** Una transazione può essere definita *commited* solo quando i suoi log record sono stati scritti su memoria stabile (poichè bastano i LOG record per recuperare al più le informazioni necessarie).

Vediamo dunque come il LOG può risolvere il transaction failure e il system failure.

Transaction Failure Adottando la politica *steal* (cioè la pagina P viene scritta quando "conviene"), se una transazione T abortisce è possibile che alcune delle pagine da essa modificate siano state già scritte su disco. Bisogna dunque disfarsi delle modifiche della transazione abortita: per annullare (*undo*) queste modifiche si scandisce il log a ritroso (usando i *prevLSN*) e si ripristinano nel database le *before image* delle pagine modificate da T.

Nell'esempio (Fig. 22) viene mostrato *undo* della transazione T2 che abortisce.

Figura 22:

LSN	prevLSN	T	type	PID	before(P)	after(P)
...						
236	-	T2	BEGIN			
237	235	T1	UPDATE	P15	(abc, 10)	(abc, 20)
238	236	T2	UPDATE	P18	(def, 13)	(ghf, 13)
239	237	T1	COMMIT			
240	238	T2	UPDATE	P19	(def, 15)	(ghf, 15)
241	-	T3	BEGIN			
242	240	T2	UPDATE	P19	(ghf, 15)	(ghf, 17)
243	241	T3	UPDATE	P15	(abc, 20)	(abc, 30)
244	242	T2	ABORT			

System Failure In caso di un System Failure vengono disfatte (applicando lo stesso algoritmo del transaction failure) tutte le transazioni per il quale non si trova il record COMMIT sul LOG, questo perchè erano transazioni attive di cui non si sanno gli esiti.

Adottando la politica *no-force* (all'atto del commit si scrive solo sul LOG, non è garantita la scrittura immediata su disco delle modifiche), una transazione T terminata correttamente (ha cioè eseguito COMMIT) non ha certezza di aver reso permanenti le modifiche sul disco, pertanto bisogna "rifare" (*redo*) T, cioè bisogna eseguire gli stessi passi che T aveva fatto durante la sua esecuzione, riscrivendo le *after image* che si trovano sul LOG.

Nell'esempio (Fig. 23) viene mostrato il *redo* della transazione T1.

Figura 23:

LSN	prevLSN	T	type	PID	before(P)	after(P)
...						
235	-	T1	BEGIN			
236	-	T2	BEGIN			
237	235	T1	UPDATE	P15	(abc, 10)	(abc, 20)
238	236	T2	UPDATE	P18	(def, 13)	(ghf, 13)
239	237	T1	COMMIT			
...						

In entrambi i casi precedenti, non era certo che le pagine (tutte o solo alcune) erano state modificate o meno, cioè in caso di una transazione abortita non è detto che effetivamente delle pagine siano state scritte su disco, come non è detto che in caso di transazione commited alcune pagine debbano ancora essere rese permanti, di conseguenza avvengono riscritture inutili (che causano inefficienza). E' possibile dunque evitarle?

Per evitare di riscrivere tutte le *after image* delle pagine modificate da tutte le transazioni che hanno eseguito COMMIT, il Buffer Manager adotta il seguente accorgimento: quando una pagina P è modificata da una transazione T viene generato il log record al quale viene assegnato il propro LSN (identificativo); viene quindi scritto l'LSN nel page header della pagina P (insieme al PID), si veda la figura (Fig. 24).

Figura 24:

Pagina P15 su disco

Page Header	PID	LSN
	P15	293

LSN	prevLSN	T	type	PID	before(P)	after(P)
...						
237	...	T1	UPDATE	P15	(abc, 10)	(abc, 20)
238	...	T2	UPDATE	P15	(abc, 20)	(ghf, 13)
...						
327	...	T3		P15	(ghf, 13)	(ghf, 18)
...						

Indichiamo con LSN(P) l'identificativo salvato sul page header della pagina P e con k l'LSN di un log record relativo alla pagina P: quando la transazione

T viene rifatta, se $LSN(P) \geq k$ allora non è necessario riscrivere la pagina P (cioè non è necessario aggiornare con *after image* del log record) poichè siamo certi che le modifiche di questo record erano già state rese permanenti su disco. Quindi si leggono sì tutte le pagine modificate dalla transazione T, ma vengono riscritte solo quelle necessarie. Un discorso analogo, ma applicato all'inverso, può essere fatto in caso di transazione abortita.

Checkpoint Come vedremo, la procedura di restart si occupa di riportare il database in uno stato consistente a fronte di un system failure. Per ridurre i tempi di restart è possibile eseguire periodicamente un "*checkpoint*", ovvero una scrittura forzata su disco di tutte le pagine modificate.

Figura 25:

LSN	prevLSN	T	type	PID	before(P)	after(P)
237	...	T3	UPDATE	P15	...	...
238	...	T2	UPDATE	P18	...	...
239	...	T1	UPDATE	P17	...	...
240	...	T1	COMMIT			
241	...	T2	COMMIT			
242			CKP			
243	...	T3	UPDATE	P19	...	...

L'esecuzione del *checkpoint* viene registrata sul LOG grazie al record *CKP* (si veda figura 25) che comprende la tabella delle transazioni attive e la tabella delle pagine "sporche": cioè il record di checkpoint contiene informazioni diverse dai record prima visti, d'altronde è inevitabile, poichè per riuscire a capire quali informazioni sono state salvate e quali no, il record checkpoint necessita di informazioni aggiuntive, come appunto, le transazioni attive e le pagine sporche (così in caso di system failure sa esattamente quali pagine e quali transazioni deve controllare). In questo modo se la transazione T ha eseguito COMMIT prima del checkpoint si è sicuri che T (dopo il checkpoint stesso) non debba essere rifatta, poichè siamo certi che le pagine della transazione T sono state scritte su disco.

1.5.2 ARIES

ARIES permette di lavorare con politiche *steal* e *no-force* (cioè, in caso di crash, dovremo disfare azioni compiute da transazioni non terminate correttamente, che risultavano dunque attive durante il crash, e ri-effettuare le azioni delle transazioni concluse correttamente ma di cui non troviamo il log di END associato, che quindi potrebbero aver sporcato pagine, ma cui modifiche potrebbero non essere state rese persistenti sul disco). Il restart del *dbms* in seguito ad un crash viene effetuato in 3 passi:

1. **Analisi:** attraverso l'analisi del LOG si identificano le pagine dirty e le transazioni attive all'istante del crash

2. **Redo:** ripete tutte le azioni a partire da un certo punto del LOG

3. **Undo:** annulla tutte le azioni di tutte le transazioni abortite

ARIES segue il protocollo WAL (che come detto ci permette di avere effettivamente un LOG completo che permette di ricostruire la storia delle azioni), inoltre come principi base utilizza il *redo* per ripetere la "storia" attraverso il LOG e inoltre tiene traccia delle modifiche nel LOG durante l'*undo* così da permettere che non venga ripetetuto (l'undo) in caso di ulteriori failure/restart.

Vediamo ora come ARIES gestisce il **system failure**.
Per prima cosa effettua la fase di *analisi* identificando le pagine dirty e le transazioni attive, per far ciò si serve del LOG analizzandolo dalla fine e procedendo all'indietro fino a trovare il primo *Checkpoint* (o fino all'inizio se non esiste alcun checkpoint). Successivamente durante la fase di *redo* si ri-effettuano tutte le modifiche alle pagine dirty che ho trovato nella fase di *analisi*: la fase di *redo* è una fase in avanti, cioè si legge il log procedendo in avanti. Durante la fase di *undo* invece si annullano tute le modifiche delle transazioni attive all'istante del crash trovate nella fase di *analisi*: la fase di *undo* è una fase all'indietro, cioè legge il LOG procedendo a ritroso. Detta in breve: utilizziamo le pagine sporche nella fase di *redo* e utilizziamo le transazioni attive nella fase di *undo*. Vediamo dunque nel dettaglio le varie fasi.

La **fase di analisi** consiste nella ricerca del *checkpoint più recente*: una volta trovato ripristina la *tabella delle transazioni* e la *tabella delle pagine dirty* corrispondenti (ricorda che nel record CHK del LOG corrispondente al checkpoint sono salvate queste informazioni). A questo punto si analizza il LOG in avanti per aggiungere alle tabelle ulteriori transazioni attive e/o ulteriori pagine sporche:

1. Se una transazione termina (quindi ha registrato il record ABORT/COMMIT) allora viene rimossa dalla tabella delle transazioni

2. Se trova una nuova transazione (quindi ha registrato il record BEGIN) allora viene aggiunta alla tabella delle transazioni

3. Infine si aggiungono alla tabella delle pagine dirty le pagine "sporcate" dai record di update e di compensazione (cioè per ogni UPDATE e COMP viene aggiunta, se non già presente, la pagina associata all'insieme delle pagine sporche)

La tabella delle transazioni conterrà dunque solo le transazioni effettivamente attive al crash (quelle che dopo il checkpoint sono terminate correttamente vengono eliminate dall'insieme delle transazioni attive).

La **fase di redo** comincia utilizzando il record con il LSN minore fra tutti quelli presenti nella tabella delle pagine dirty costruita nella fase di analisi (è possibile che tale record preceda l'ultimo CKP, poichè essendo presente nel checkpoint già una lista di pagine sporche, l'LSN minore potrebbe precede il checkpoint stesso). Si percorre il LOG in avanti ri-effettuando tutti i record di *update* e di *compensazione* a meno che la pagina corrispondente non sia tra

quelle "sporche" (cioè se la pagina associata al record non risulta sporca la possiamo saltare, e questo può accadere solo per le pagine che sono presenti solo nell'intervallo precedente al checkpoint e il checkpoint stesso) o il valore di LSN della pagina risulti $\geq$ del LSN del record con cui abbiamo iniziato (poichè siamo sicuri che sia stata apportata una modifica alla pagina, resa persistente, successiva al record che stiamo leggendo) . Infine il valore di LSN della pagina viene aggiornato con il LSN del record attuale.

La **fase di undo** comincia identificando le transazioni attive all'istante del crash utilizzando la tabella delle transazioni costruita in fase di analisi. Tutte le azioni di tale transazioni vanno annullate all'indietro. Si inizia dalla transazione avente il LSN più recente e:

- Se il *record è di compensazione* si passa al record precedente (campo undoNextLSN) a meno che questo sia 0 nel qual caso l'undo della transazione è completo (se il undoNextLSN è uguale a 0 vuol dire che il precedente LSN è un'azione di BEGIN della transazione).
- Se il *record è di update* si effettua l'annullamento (quindi si utilizza il *before image*), si scrive un record di compensazione (in fondo al LOG) e si passa al record precedente (seguendo la catena utilizzando il campo prevLSN).

Finora abbiamo supposto che durante la procedura di *restart* non fosse possibile un system failure: ovviamente un problema di questo tipo potrebbe benissimo verificarsi, ecco dunque che ARIES deve riuscire a gestire i crash anche durante il restart. La scrittura dei record di compensazione permette di gestire il verificarsi di crash durante la procedura di restart nella fase di *undo*: infatti i record di compensazione indicano che le meodifiche sono già state annullate durante la fase di *redo*. Se il crash si verifica durante l'analisi, questa va ripresa dall'inizio, allo stesso modo se il crash si verifica durante la fase di *redo* esso va ripreso dall'inizio ed eventualmente alcune pagine non verrano riscritte al nuovo redo (poichè magari già scritte nel redo precedente).

Trattiamo ora il caso in cui ARIES debba gestire il *media failure*.
Nel caso di media failure si ha bisogno di utilizzare la copia di backup del database (il *database dump*) per effetuare un ripristino. Facendo uso del LOG, si rieffettuano tutte le transazioni che hanno eseguito il COMMIT e si disfano tutte quelle per cui il record di COMMIT non si trova nel LOG.

Notiamo che se il Buffer Manager usa la politica *no-steal* allora non è necessario l'*undo* (poichè non troveremmo mai sul disco aggiornamenti di una transazione non ancora completata), allo stesso modo se il Buffer Manager utilizza la politica *force* allora non è necessario il *redo* (poichè prima di scrivere nel LOG forziamo la scrittura della pagina). Allora perchè ARIES è così usato (ricordiamo che ARIES utilizza la politica *steal* e *no-force*)? ARIES ha trovato una vasta applicazione perchè favorisce il funzionamento "normale" del database, supponendo dunque che i guasti siano rari (cosa d'altronde vera).

Esistono ovviamente algoritmi diversi da ARIES, che vedremo in breve, che utilizzano le politiche *no-steal* e *force*, ma essi complicano notevolmente la gestione delle transazioni.

Algoritmo UNDO/no-REDO prevede che le modifiche di una transazione T siano scritte in memoria stabile prima della fine della transazione stessa. La complicazione è dovuta all'uso della memoria stabile.

Algoritmo no-UNDO/REDO prevede che le modifiche di una transazione T siano scritte in memoria stabile solo dopo la fine della transazione stessa. La complicazione è dovuta all'uso della memoria stabile.

Algoritmo no-UNDO/no-REDO prevede che le modifiche di una transazione T siano scritte in memoria stabile al momento della fine della transazione stessa come un'azione atomica. La complicazione è dovuta all'uso della memoria stabile e ad eseguire la scrittura sulla memoria come azione atomica

1.5.3 Esempi

Mostriamo di seguito alcuni esempi che potranno chiarire le idee.

Figura 26:

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA'
4	2	T2	UPDATE	PB	ValB	ValB'
5	4	T2	UPDATE	PA	ValA'	ValA"
6	3	T1	UPDATE	PC	ValC	ValC'
7	5	T2	COMMIT			

Esempio 1 Supponiamo di avere il LOG mostrato in figura (Fig. 26) e che ora la transazione T1 modifichi la pagina PD, ma si verifichi un crash del sistema prima della scrittura del corrispondente log record (poichè usiamo il protocollo WAL, è come se questa ultima modifica apportata a PD da T1 non esistesse). In questa situazione abbiamo la certezza che la pagina PB sia stata scritta (T2 ha eseguito COMMIT) ma che PA, PC e PD no.

Essendo insorto un crash ARIES agisce e inizia con la fase di *analisi*. Facendo riferimento sempre alla figura 26, vediamo di seguito il risultato dell'analisi (notiamo che non c'è alcun *checkpoint* quindi si parte dall'inizio):

- T1 viene aggiunta alla tabella delle transazioni attive, poichè si legge LSN 1 e si trova un BEGIN
- T2 viene aggiunta alla tabella delle transazioni attive, poichè si legge LSN 2 e si trova un BEGIN
- La pagina PA viene aggiunta alla tabella delle pagine sporche, poichè si legge LSN 3 e si trova un UPDATE (e si indica che la pagina è sporca a causa del LSN=3)

- La pagina PB viene aggiunta alla tabella delle pagine sporche, poichè si legge LSN 4 e si trova un UPDATE (e si indica che la pagina è sporca a causa del LSN=4)
- La pagina PC viene aggiunta alla tabella delle pagine sporche, poichè si legge LSN 6 e si trova un UPDATE (e si indica che la pagina è sporca a causa LSN=6)
- La transazione T2 viene rimossa dalla tabella delle transazione attive, poichè si legge LSN 7 e si trova il COMMIT di T2

La pagina PD non viene aggiunta alle pagine dirty poichè non è presente il record sul LOG (ricordiamoci il protocollo WAL).

Finita la fase di analisi, si procede alla fase di *redo*. La fase di redo inizia cercando il record con LSN minore fra tutte le pagine dirty presenti nella tabella delle pagine sporche. L'insieme delle pagine sporche è {PA, PB, PC}, fra tutti i record di update associati a questa pagina quello con LSN minore è LSN 3, dunque la fase di *redo* parte da questo record del LOG e procede in avanti esaminando il LOG stesso producendo il seguente risultato:

- PA viene letta da disco. Nel Page Header di PA è presente come LSN (che da ora in poi chiameremo pageLSN) uno zero, poichè nessuno ha effettivamente aggiornato la pagina. $pageLSN = 0 < LSN = 3$ allora si ri-effettua l'azione e il pageLSN viene modificato a $pageLSN = 3$
- PB viene letta da disco. Il pageLSN di PB è uguale a 4 (poichè effettivamente l'aggiornamento su PB è stato completato grazie al COMMIT di T2). $pageLSN = 4 \geq LSN = 4$ allora non si ri-effettua l'azione
- PA viene letta da disco. $pageLSN = 3 < LSN = 5$ allora si ri-effettua l'azione e il pageLSN viene modificato a $pageLSN = 5$
- PC viene letta da disco. Il pageLSN di PC è uguale a 0 (nessuno ha effettivamente aggiornato la pagina). $pageLSN = 0 < LSN = 6$ allora si ri-effettua l'azione e il pageLSN viene modificato a $pageLSN = 6$
- Viene aggiunto un nuovo record di END per T2

Vediamo dunque l'ultima fase, la fase di *undo*. La fase di *undo* inizia cercando fra tutte la transazione più recente fra tutte le transazioni attive durante il crash. L'insieme delle transazoni attive è {T1} dunque l'algoritmo partendo da questa transazione inizia ad esaminare il LOG all'indietro annullando tutte le modifiche della transazione, producendo il seguente risultato:

- La pagina PC viene ripristinata a *valC* (si ricordi che deve essere utilizzato il *before image*). Si aggiunge in fondo un *record di compensazione* con $undoNextLSN = 3$ (che è proprio il prevLSN del record 6).

- La pagina PA viene ripristina a *valA*. Si aggiunge in fondo un *record di compensazione* con $undoNextLSN = 0$. (Si noti bene che il record LSN=5 non viene annullato poichè appartiene alla transazione T2 che non risulta essere fra le transazioni attive)

- Viene aggiunto un nuovo record di END per T1

Notiamo che abbiamo riportato il valore di PA a *ValA* (valore modificato da T1) e non a *ValA'* modificato da T2 e dunque la modifica effettuata da T2 viene 'persa' (con Strict 2PL non sarebbe successo perchè T2 non avrebbe potuto accedere alla risorsa PA poichè T1 avrebbe avuto un lock esclusivo sulla pagina).

Esempio 2 Il seguente esempio mostra come gestire un crash durante il restart.

Supponiamo che il LOG contenga i seguenti record (Fig. 27)

Figura 27:

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA'
4	2	T2	UPDATE	PB	ValB	ValB'
5	3	T1	ABORT			
6	5	T1	COMP	PA	ValA	undoNextLSN: -
7	-	T3	BEGIN			
8	7	T3	UPDATE	PC	ValC	ValC'
9	4	T2	UPDATE	PA	ValA	ValA"

Si verifica ora un crash del sistema che viene gestito come descritto in Esempio 1. Notiamo inoltre come l'ABORT di T1 (LSN 5) sia gestito come un normale *undo*.
Inizia la fase di *analisi* che dà come risultato:

- Pagine PA (LSN=3), PB (LSN=4) e PC (LSN=8) aggiunte alla tabella delle pagine sporche

- Transazioni T2 e T3 aggiunte alla tabella delle transazione attive. T1 viene cancellata poichè ha abortito (dunque terminata).

Fase di *redo* procede allo stesso modo dell'Esempio 1 utilizzando come primo record LSN=3.

Fase di *undo*: i record da disfare sono LSN=9 e LSN=4 per la transazione T2, e LSN=8 per la transazione T3. Si parte con la transazione più recente (T2) e si procede a ritroso sul LOG, producendo il seguente risultato:

- La pagina PA viene ripristina a *ValA*. Si aggiunge in fondo un *record di compensazione* con $undoNextLSN = 4$

- La pagina PC viene ripristina a *ValC*. Si aggiunge in fondo un *record di compensazione* con $undoNextLSN = 0$

Avviene ora un nuovo crash. Si noti che la procedura di *undo* non è terminata (manca da esaminare e da disfare ancora LSN=4 per la transazione 2), ritrovandoci ad una tabella di LOG così di seguito aggiornata (Fig. 28)

Figura 28:

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA'
4	2	T2	UPDATE	PB	ValB	ValB'
5	3	T1	ABORT			
6	5	T1	COMP	PA	ValA	undoNextLSN: -
7	-	T3	BEGIN			
8	7	T3	UPDATE	PC	ValC	ValC'
9	4	T2	UPDATE	PA	ValA	ValA"
CRASH, RESTART						
10	9	T2	COMP	PA	ValA	undoNextLSN: 4
11	8	T3	COMP	PC	ValC	undoNextLSN: -
12	11	T3	END			

Bisogna dunque ora ripartire con la procedura di *restart.*
La fase di analisi produce il seguente risultato

- Pagina PA (LSN=3), PB (LSN=4) e PC (LSN=8) aggiunte alla tabella delle pagine dirty

- Transazione T2 aggiunta alla tabella delle transazioni attive. T1 e T3 vengono cancellate poichè T1 ha abortito, e T3 con la procedura di restart precedente era stata già gestita e fatta terminare.

La fase di *redo*, che ancora una volta parte da LSN=3, dovrà ri-effettuare tutte le azioni fino a LSN=12, potendo evitare di scrivere le pagine su disco il cui LSN risulta essere già aggiornato.

La fase di *undo*, che parte dall'unica transazione T2 presente nell'insieme delle transazioni attive al momento del crash, dovrà disfarsi dell'unico record LSN=10 per T2. Il risultato produce:

- LSN=4 viene inserito nei record da disfare. Infatti è stato letto LSN=10, record di compensazione, il quale indicava come prossimo record da disfare $undoNextLSN = 4$

- Viene effettivamente disfatto il record LSN 4, dunque la pagina PB viene ripristinata a *ValB* e si aggiunge in fondo un *record di compensazione* con $undoNextLSN = 0$

La situazione finale del LOG è mostrata in figura (Fig. 29)

Figura 29:

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA'
4	2	T2	UPDATE	PB	ValB	ValB'
5	3	T1	ABORT			
6	5	T1	COMP	PA	ValA	undoNextLSN: -
7	-	T3	BEGIN			
8	7	T3	UPDATE	PC	ValC	ValC'
9	4	T2	UPDATE	PA	ValA	ValA"
CRASH, RESTART						
10	9	T2	COMP	PA	ValA	undoNextLSN: 4
11	8	T3	COMP	PC	ValC	undoNextLSN: -
12	11	T3	END			
CRASH, RESTART						
13	10	T2	COMP	PB	ValB	undoNextLSN: -
14	13	T2	END			