

ATTENZIONE: Anche se ho impiegato cura ed impegno nella realizzazione di questo documento, consiglio vivamente chiunque voglia applicare i suggerimenti da me proposti, di eseguire tutte le opportune prove in un ambiente di test, prima di effettuare eventuali modifiche in produzione. Questa convinzione nasce dal fatto che nonostante mi sia documentato attraverso libri di testo, manuali e documenti in rete, esiste sempre la probabilità di eventuali errori nella stesura ed imprecisioni nella esposizione.

ATTENZIONE: Quanto scritto in questo documento può essere utilizzato in modo del tutto libero. Pregherei comunque tutti coloro che desiderano delle modificarlo, di comunicarlo al [mio](mailto:mio) indirizzo di posta in modo tale da permettermi di aggiornare e/o correggere eventuali errori.

<http://www.oral.com> Febbraio 2006

Di [Andrea Salzano](#)

## Sommario

1	<i>Architettura di un RDBMS</i> .....	2
2	<i>Transazioni</i> .....	2
2.1	<i>Atomicità</i> .....	6
2.2	<i>Consistenza</i> .....	6
2.3	<i>Isolamento</i> .....	6
2.4	<i>Persistenza</i> .....	6
3	<i>Il Gestore della Concorrenza</i> .....	7
3.1	<i>Anomalie della concorrenza</i> .....	7
3.1.1	<i>Perdita di aggiornamento</i> .....	7
3.1.2	<i>Lecture sporche</i> .....	8
3.1.3	<i>Lecture inconsistenti</i> .....	9
3.1.4	<i>Aggiornamento fantasma</i> .....	9
3.2	<i>Controllo della concorrenza</i> .....	10
3.2.1	<i>Locking Pessimistico</i> .....	10
3.2.2	<i>Locking Ottimistico</i> .....	12
3.2.3	<i>Locking ottimistico, utilizzando il versioning della colonna</i> .....	13
3.2.4	<i>Locking ottimistico, utilizzando una funzione di hash</i> .....	13
3.2.5	<i>Locking ottimistico, utilizzando ORA_ROWSCN</i> .....	13
4	<i>Il Gestore dell’Affidabilità</i> .....	13
5	<i>L’ottimizzatore</i> .....	13
6	<i>Indice</i> .....	13

# 1 Architettura di un RDBMS

Un modello semplificato dell'architettura di un database relazionale centralizzato può essere rappresentato come in figura1. Questa è caratterizzata dalle seguenti parti:

- L' "SQL processor", il gestore dei comandi SQL
- Lo "scheduler", il gestore della concorrenza
- L' "access method manager", il gestore dei metodi di accesso
- Lo "storage structures manager", il gestore delle strutture di memorizzazione
- Il "buffer manager", il gestore del buffer
- Lo "storage manager", il gestore della memoria permanente
- Il "transaction and recovery manager", il gestore dell'affidabilità

Ovviamente questa è solo un'astrazione che consente di comprendere meglio lo scopo di ognuno di essi. Nella realtà questi moduli non sono nettamente separati come la figura potrebbe far pensare.

## 2 Transazioni

Una delle funzionalità più importanti di un RDBMS è la protezione dei dati da malfunzionamenti e da interferenze dovute all'accesso concorrente ai dati da parte di più utenti. Tale funzionalità è realizzata attraverso le *transazioni*. Queste richiedono opportune strutture e algoritmi per la loro gestione che sono fra i più importanti e complessi da realizzare.

Partendo dalla divisione logica fatta in figura 1, analizziamo le transazioni sia dal punto di vista del *transaction manager*, come meccanismo per la protezione dei dati da malfunzionamenti, sia dal punto di vista dello *scheduler*, come meccanismo per la protezione dei dati da interferenze dovute ad accessi concorrenti.

Una transazione è l'unità elementare di lavoro di un'applicazione costituita da un insieme di istruzioni di lettura e scrittura (read/write) sulla base dati. Ad essa sono associate le cosiddette proprietà acide di cui parleremo tra breve. Un sistema che realizza un meccanismo atto a definire le transazioni è detto transazionale.

Una transazione può essere definita sintatticamente: ogni transazione, quale che sia il linguaggio con cui viene implementata, è incapsulata all'interno di due comandi: BEGIN TRANSACTION (BOT) ed END TRANSACTION (EOT). All'interno del codice delle transazioni possono comparire due istruzioni particolari: COMMIT e ROLLBACK (abort). L'effetto dei due comandi è quello di

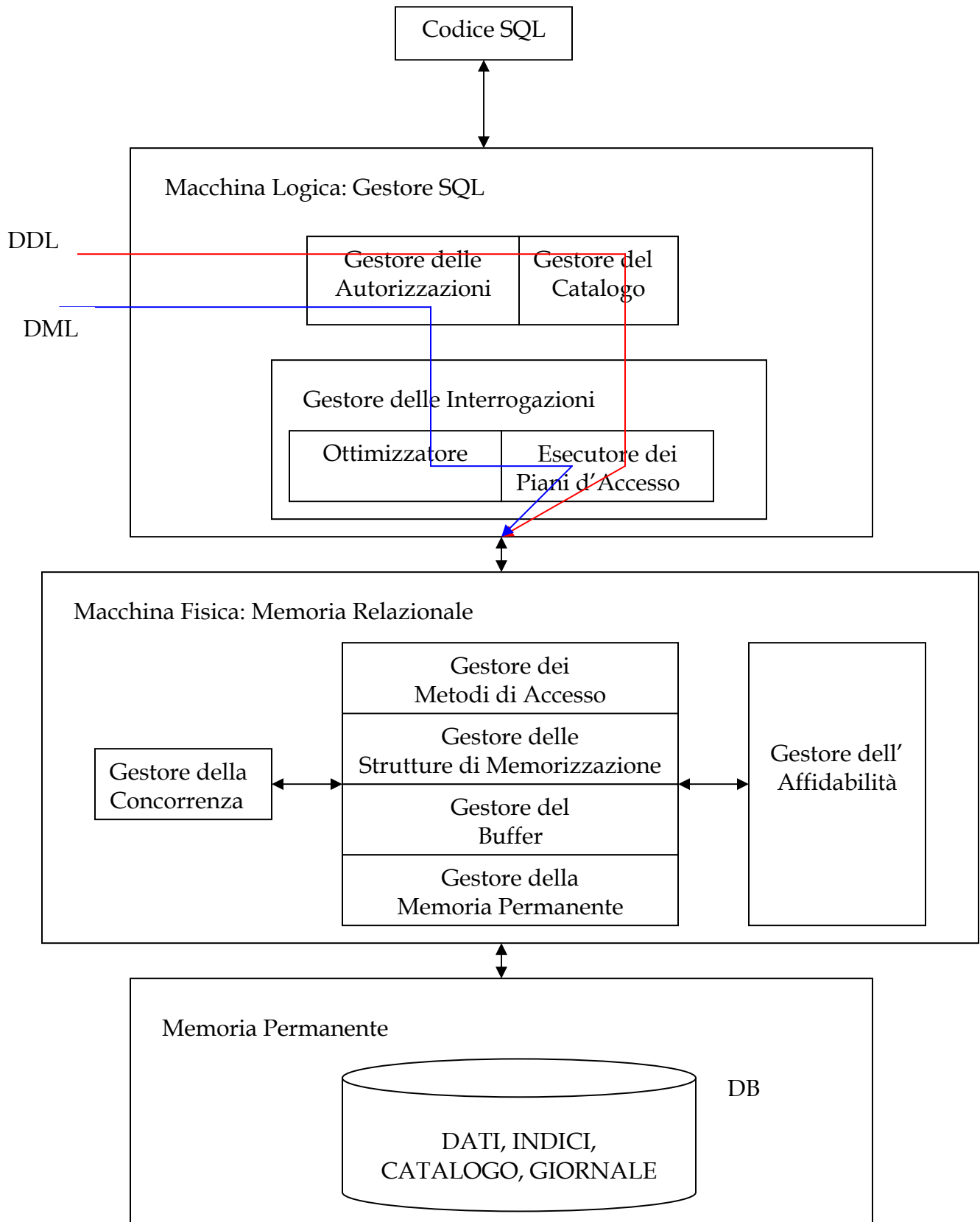
far andare a buon fine la transazione o di non generare alcun effetto sulla base dati, rispettivamente.

Ancora. Viene definita "ben formata" una transazione iniziata con BEGIN TRANSACTION e conclusa con END TRANSACTION, nel cui corso viene eseguito uno solo dei due comandi COMMIT o ROLLBACK ed in cui non avvengono operazioni di accesso e/o modifica alle base di dati successive all'esecuzione del comandi di COMMIT o ROLLBACK.

In generale la coppia di comandi END TRANSACTION, BEGIN TRANSACTION viene eseguita immediatamente ed implicitamente dopo ogni COMMIT o ROLLBACK, in modo da rendere tutte le transazioni ben formate.

Quando si afferma che una transazione è stata eseguita con successo, non vuol dire che deve considerarsi terminata normalmente: in teoria infatti è possibile che i suoi effetti non siano stati ancora resi permanenti ed un eventuale malfunzionamento potrebbe impedire che ciò avvenga. Dire allora che gli effetti di una transazione sulla base dati sono permanenti significa che non sono alterabili da eventuali malfunzionamenti. Vedremo questo quando parleremo del gestore dell'affidabilità.

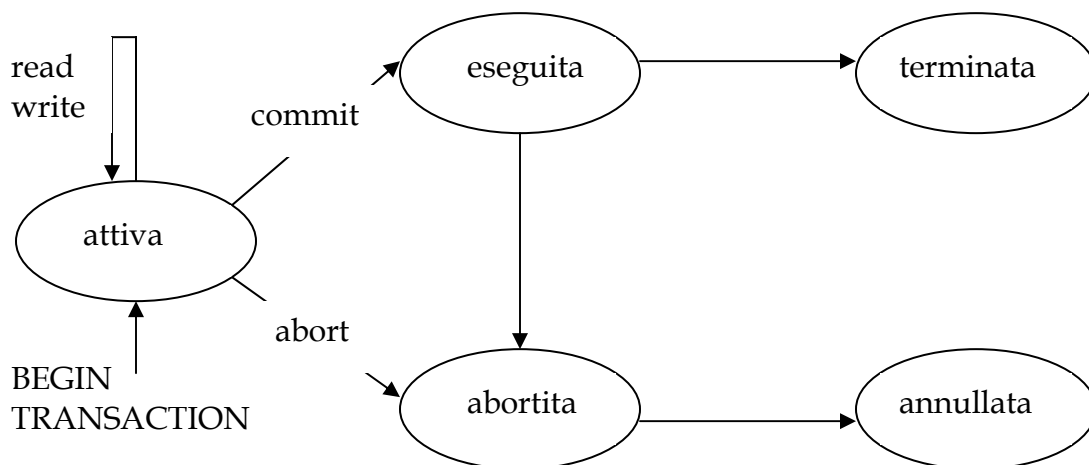
Figura1



Prima di procedere, vediamo gli stati che una transazione può assumere:

- Una transazione è detta *attiva*, se è ancora in fase di esecuzione
- Una transazione è detta *eseguita*, se l'esecuzione si è conclusa con successo
- Una transazione è detta *terminata* (normalmente), se l'esecuzione si è conclusa con successo e tutti i suoi effetti sulla base dati sono stati resi permanenti.
- Una transazione è detta *interrotta*, se si è verificato un malfunzionamento che ha causato un fallimento di transazione, un fallimento di sistema oppure un disastro,
- Una transazione è detta *abortita* o *fallita*, se è stata interrotta a causa di un fallimento di transazione,
- Una transazione è detta *annullata*, se dopo essere stata interrotta, tutte le sue modifiche alla base dati sono state disfatte o annullate.

Di seguito è rappresentato il grafo degli stati sopra descritti:



Come anticipato il codice eseguito tra un BEGIN TRANSACTION ed END TRANSACTION gode di particolari proprietà, le cosiddette proprietà acide: atomicità, consistenza, isolamento e persistenza (il termine è un acronimo che deriva dall'inglese ACID: Atomicity, Consistency, Isolation, Durability).

A questo punto, vorrei dare una definizione formale formale di transazione.

Una transazione T è definita come una sequenza di azioni di lettura (read) o scrittura (write).

## 2.1 Atomicità

Rappresenta il fatto che la transazione è un'unità indivisibile: gli effetti di una transazione o vengono resi visibili tutti o nessuno. L'approccio è del tipo "tutto o niente". Questa proprietà ha una conseguenza molto importante. Se durante le operazioni di lavoro sulla base dati si verifica un errore ed una di tali operazioni di lettura o scrittura non può essere completata, il sistema deve essere in grado di ricostruire un'immagine della base dati esistente prima dell'inizio della transazione, *disfacendo* il lavoro svolto dalle operazioni fino a quel momento (operazione di *undo* o rollback). Viceversa, dopo il COMMIT, il sistema deve assicurare che la transazione lasci la base dati nel suo stato finale. Come vedremo, questo può comportare il dover *rifare* il lavoro svolto (operazione di *redo*). In questo modo, il COMMIT fissa il momento atomico ed indivisibile in cui la transazione va a buon fine.

## 2.2 Consistenza

In questo caso viene richiesto che la transazioni non violi i vincoli di integrità definiti sulla base dati. Quando il sistema rivela che la transazione sta violando uno dei vincoli, ad esempio inserendo una riga duplicata, deve intervenire per annullare la transazione o per correggere la violazione.

Tale controllo può essere *immediato* o *differito*. Nel primo caso il controllo avviene nel corso della transazione, mentre nel secondo avviene alla fine della transazione, ovvero a seguito del COMMIT.

## 2.3 Isolamento

Tale proprietà garantisce che l'esecuzione della transazione è indipendente dalla contemporanea esecuzione di altre transazioni. In altre parole possiamo dire che il risultato dell'esecuzione concorrente di un insieme di transazioni è equivalente al risultato ottenuto qualora ciascuna di esse fosse eseguita da sola (è come cioè se non esistesse concorrenza). L'obiettivo di tale proprietà è anche quello di rendere ciascuna transazione indipendente da tutte le altre. Si vuole impedire cioè che il ROLLBACK di una non causi il ROLLBACK di un'altra, generando eventualmente un effetto a catena.

## 2.4 Persistenza

Si desidera in questo caso che l'effetto del COMMIT di una transazione non venga più perso. In pratica, una base dati deve garantire che nessun dato venga perso per alcun motivo.

## 3 Il Gestore della Concorrenza

Una delle sfide più importanti nello sviluppo di database multi utente, è quella di massimizzare non solo l'accesso concorrente, ma nello stesso tempo assicurare che ogni utente sia capace di leggere e modificare i dati in modo consistente. I controlli di **locking** e **concorrenza**, che consentono di realizzare tutto ciò, sono caratteristiche chiave di un qualunque database.

La concorrenza fa riferimento in linea di massima, alle "operazioni di ingresso e uscita" (il livello più basso nell'architettura di un RDBMS) che realizzano il trasferimento dei blocchi dati dalla memoria di massa alla memoria centrale. E questo viene implementato ovviamente con le chiamate di sistema di lettura (read) e scrittura (write). Tali chiamate di sistema vengono gestite dal modulo dell'RDBMS detto scheduler, il quale determina se le richieste possono essere soddisfatte o meno.

### 3.1 Anomalie della concorrenza

Daremo uno sguardo su come Oracle esegue i locks sui dati e le implicazioni di questo modello nello scrivere applicazioni multi utente. L'esecuzione concorrente tuttavia può causare alcuni difetti di correttezza. La presenza di tali anomalie motiva la necessità di controllare la concorrenza. Esistono 4 quattro casi tipici a cui possiamo andare incontro. Prima di procedere, diamo alcune definizioni: indichiamo con  $r(x)$  e  $w(x)$ , la lettura e la scrittura, rispettivamente, di un generico oggetto  $x$  della base dati.

#### 3.1.1 Perdita di aggiornamento

Supponiamo che due transazioni  $t1$  e  $t2$  stiano operando sullo stesso oggetto  $x$  della base dati, nel seguente modo:

$t1: r(x), x:=x+1, w(x)$   
 $t2: r(x), x:=x+1, w(x)$

Come si vede, stiamo considerando due transazioni identiche che agiscono nello stesso modo sullo stesso oggetto. Supponendo che il valore iniziale di  $x$  sia 5, eseguendo le due transazioni in modo sequenziale (o *seriale*), il valore finale di  $x$  sarà 7. Infatti dapprima la transazione  $t1$  modificherà il valore di  $x$ , portandolo da 5 a 6, poi successivamente  $t2$  leggerà il valore di  $x$  (che adesso vale 6) e lo incrementerà di una unità, portandolo a 7. Supponiamo adesso invece che l'accesso all'oggetto  $x$  da parte di  $t1$  e  $t2$ , avvenga in modo concorrente e disponiamo l'ordine di esecuzione in modo tale da evidenziarne l'ordine temporale. Allo scopo, costruiamo una tabella come segue:

Transazione t1	Valore di x per t1	Transazione t2	Valore di x per t2
BOT			
r1(x)	5		
x:=x+1	6		
		BOT	
		r2(x)	5
		x:=x+1	6
		w2(x)	
		COMMIT	6
w1(x)			
COMMIT	6		

Come si vede, in questo caso il valore finale di x coincide con quello di t1 ovvero 6 (e non più 7). Questo è dovuto al fatto che entrambe le transazioni leggono 5 come valore iniziale di x. Questa anomalia prende il nome di “perdita di aggiornamento” (o anche *lost update*), in quanto gli effetti della transazione t2 sono persi.

### 3.1.2 Letture sporche

Consideriamo le stesse due transazioni t1 e t2, ed analizziamo la seguente esecuzione:

Transazione t1	Valore di x per t1	Transazione t2	Valore di x per t2
BOT			
r1(x)	5		
x:=x+1	6		
w1(x)			
		BOT	
		r2(x)	6
		x=x+1	7
		w2(x)	
		COMMIT	7
ROLLBACK	7		

Il valore finale di x è 7 invece di 6, come dovrebbe essere. Il problema in questo caso è la lettura della transazione t2 che genera uno stato intermedio generato dalla transazione t1: il fatto è che t2 non dovrebbe vedere tale stato. Questa anomalia prende il nome di “lettura sporca” (o *dirty buffer*), in quanto viene letto un dato che rappresenta uno stato intermedio nell’evoluzione di una transazione.

### 3.1.3 Letture inconsistenti

Supponiamo adesso che la transazione t1 esegua solo letture, ma che questo avvenga in istanti diversi come descritto di seguito:

Transazione t1	Valore di x per t1	Transazione t2	Valore di x per t2
BOT			
r1(x)	5		
		BOT	
		r2(x)	5
		x:=x+1	6
		w2(x)	
		COMMIT	6
r1(x)	6		
COMMIT	6		

In questo caso x assume il valore 5 dopo la prima lettura ed il valore 6 dopo la seconda. Sarebbe invece opportuno che una transazione trovi sempre lo stesso valore per ciascuna lettura, e non risenta gli effetti delle altre transazioni.

### 3.1.4 Aggiornamento fantasma

Supponiamo adesso di avere tre oggetti, x, y, z, tali da soddisfare un vincolo di integrità del tipo:  $x+y+z=100$ . Le due transazioni t1 e t2, eseguono le seguenti operazioni:

Transazione t1	Valore per t1	Transazione t2	Valore per t2
BOT			
r1(x)	X		
		BOT	
		r2(y)	Y
r1(y)	Y		
		y:=y-10	y-10
		r2(z)	Z
		z:=z+10	z+10
		w2(y)	
		w2(z)	
		COMMIT	$s=x+y-10+z+10 (=100)$
r1(z)	z+10		
s:=x+y+z	$x+y+z+10 (=110)$		
COMMIT	110		

La transazione t2 non altera la somma dei valori e quindi non viola il vincolo di integrità, tuttavia la variabile s che contiene la somma di x, y e z vale, per la transazione t1, 110. Quello che succede è che la transazione t1 osserva solo una parte degli effetti di t2, e quindi osserva uno stato che non soddisfa i vincoli di integrità. Questa anomalia prende il nome di “aggiornamento fantasma” (o *ghost update*).

## 3.2 Controllo della concorrenza

Viste le anomalie dovute all’accesso concorrente, vediamo quali sono i meccanismi di Oracle, atti a proteggere la base dati, in modo tale che non perda in consistenza.

Un lock è un meccanismo utilizzato per regolare l’accesso concorrente a risorse condivise. Nota come abbia utilizzato il termine risorsa condivisa e non “riga del database”. E’ vero che Oracle esegue dei lock su tabelle a livello di riga, ma è vero anche che utilizza lock a molti altri livelli per fornire accesso concorrente a varie risorse.

Vedremo che:

- \* Si deve rimandare il COMMIT, fino al momento opportuno. Non va fatto per evitare di stressare il sistema, così come non si stressa il sistema nell’averne lunghe transazioni. La regola è quella di eseguire il COMMIT quando si deve e non prima. La tua transazione deve essere grande o piccola in base alla logica di business;
- \* Devi mantenere i locks finché lo ritieni necessario: sono tools che devi usare, non cose da evitare. I lock non sono risorse sacre.;
- \* non c’è un overhead con il locking a livello di riga;
- \* Non devi mai scalare un lock (ad esempio usare il lock di una tabella invece che quello di riga), perché potrebbe essere meglio per il sistema. In Oracle, un tale atteggiamento non risparmia le risorse

Ci sono due modi di affrontare applicativamente gli accessi concorrenti. Questi sono descritti di seguito.

### 3.2.1 Locking Pessimistico

Questo metodo di lock dovrebbe essere messo in azione un istante prima di modificare un valore sullo schermo. Supponiamo ad esempio che un utente seleziona una specifica riga ed indica la sua intenzione di eseguire l’update:

```

SELECT MATRICOLA, COGNOME, SALARIO
  FROM IMPIEGATI
 WHERE DIPID = 10
/

```

MATRICOLA	COGNOME	SALARIO
7782	ROSSI	2450
7839	LORE	5000
7934	SALZANO	1300

Diciamo che in questo caso, si decide di aggiornare la riga relativa all'impiegato SALZANO. A questo punto, la nostra applicazione esegue il seguente comando<sup>1</sup>:

```

SELECT MATRICOLA, COGNOME, SALARIO
  FROM IMPIEGATI
 WHERE MATRICOLA = :MATID
 AND COGNOME = :COGNOMEIMP
 AND SALARIO = :STIPENDIO
FOR UPDATE NOWAIT
/

```

MATRICOLA	COGNOME	SALARIO
7934	SALZANO	1300

Questo approccio è detto locking pessimistico, perché noi eseguiamo un lock della riga interessata prima di tentarne l'aggiornamento in quanto siamo dubbiosi (pessimisti), che la riga resti inalterata durante la nostra attività. Ora, poiché tutte le tabelle dovrebbero avere una chiave primaria e poiché questa dovrebbe essere immutabile (nel senso che non dovremmo mai aggiornarla), da questo esempio possiamo ricavare tre risultati:

- Se i dati sottostanti non sono cambiati<sup>2</sup>, noi avremo sotto controllo la riga SALZANO e questa sarà bloccata per tutte le alter eventuali modifiche. In pratica, tutti gli altri scrittori di SALZANO saranno bloccati eccetto i lettori.

---

<sup>1</sup> Come esempio, riporto come, con SQL\*PLUS, è possibile realizzare l'UPDATE:  
 SQL> variable matid number  
 SQL> variable cognomeimp varchar2(20)  
 SQL> variable stipendio number  
 SQL> exec :matid:= 7934; :cognomeimp:= SALZANO; :stipendio:= 1300

<sup>2</sup> Se i dati non sono cambiati, noi abbiamo sotto controllo la riga SALZANO

- Se un'altro utente tenta di modificare la nostra riga, otterrà l'errore ORA-00054 risorsa occupata<sup>3</sup>. In pratica, quell'utente è bloccato e deve aspettare che noi finiamo il nostro lavoro.
- Se , tra l'operazione si selezionare i dati ed indicare la nostra intenzione di aggiornamento, qualcuno ha già modificato la riga<sup>2</sup> allora otterremo zero righe.

Una volta che sulla riga è stato messo un lock con successo, eseguiremo alcuni aggiornamenti e li renderemo permanenti:

```
UPDATE IMPIEGATI
  SET COGNOME = :COGNOMEIMP, SALARIO = :STIPENDIO
  WHERE MATRICOLA = :MATID
/

COMMIT
/
```

Abbiamo così modificato I dati, in tutta sicurezza: non è possibile cioè sovrascrivere altri cambiamenti. Come abbiamo verificato, i dati non cambiano tra l'istante in cui li leggiamo e quello in cui eseguiamo l'aggiornamento.

### 3.2.2 *Locking Ottimistico*

Il secondo metodo , chiamato **locking ottimistico**, rimanda tutti i lock giusto un attimo prima che l'UPDATE sia eseguito. In altre parole modificheremo le informazioni senza che un lock venga acquisito. Siamo cioè ottimisti che nessuno modificherà i nostri dati e quindi aspetteremo l'ultimo momento. Questo metodo funziona su tutti i database, ma incrementa la probabilità che un aggiornamento vada "perduto".

Un modo per implementare questo tipo di lock è mantenere i dati vecchi e nuovi nell'applicazione (da qualche parte), fino a che i dati vengono aggiornati, come nel seguente modo, per esempio:

```
UPDATE table
  SET colonna1 = :new_colonna1, colonna2 = :new_colonna2
  WHERE column1 = :old_column1
  AND column2 = :old_column2
/
```

---

<sup>3</sup> Dal manuale di Oracle 10gR2

**ORA-00054: resource busy and acquire with NOWAIT specified**

**Cause:** Resource interested is busy.

**Action:** Retry if necessary.

In questo caso, siamo ottimisticamente speranzosi che i dati non vengono modificati. Se nessuno li cambia, allora aggiorniamo **una** riga (siamo fortunati: i dati non sono cambiati tra l'istante in cui li leggiamo e quello in cui sottomettiamo l'UPDATE), altrimenti aggiorniamo **zero** righe.

Ci sono diversi modi di implementare il locking ottimistico. Ne abbiamo visto uno piuttosto semplice che consiste nel mantenere le "before image" delle righe da modificare in qualche variabile ed utilizzare tali valori successivamente. Di seguito ne descriveremo altri tre:

- i) utilizzare una colonna speciale, aggiornata da un trigger o dall'applicazione stessa, per dirci qual è la "versione" del record;
- ii) utilizzare un hash calcolato a partire dai dati stessi;
- iii) utilizzare la nuova caratteristica di Oracle 10g, ORA\_ROWSCN;

### *3.2.3 Locking ottimistico, utilizzando il versioning della colonna*

### *3.2.4 Locking ottimistico, utilizzando una funzione di hash*

### *3.2.5 Locking ottimistico, utilizzando ORA\_ROWSCN*

## *4 Il Gestore dell'Affidabilità*

## *5 L'ottimizzatore*

L'ottimizzatore è quel componente dell'RDBMS che ha il compito di decidere le strategie di accesso ai dati: riceve in ingresso un'interrogazione, di cui svolge un'analisi lessicale, sintattica e semantica. A questo punto determina e sceglie il miglior percorso per accedere ai dati.

## *6 Indice*

Aggiornamento fantasma; 9

Controllo della concorrenza; 10

Lecture inconsistenti; 9  
Lecture sporche; 8  
Locking Ottimistico; 12  
Locking Pessimistico; 10  
Perdita di aggiornamento; 7  
transazione; 2; 5  
Transazione; 2  
transazioni; 2