Capitolo 12

Esercizio 12.1

Descrivere la ripresa a caldo, indicando la costituzione progressiva degli insiemi di UNDO e REDO e le azioni di recovery, a fronte del seguente log:

DUMP, B(T_1), B(T_2), B(T_3), I(T_1, O_1, A_1), D(T_2, O_2, B_2), B(T_4), U(T_4, O_3, B_3, A_3), U(T_1, O_4, B_4, A_4), C(T_2), CK(T_1, T_3, T_4), B(T_5), B(T_6), U(T_5, O_5, B_5, A_5), A(T_3), CK(T_1, T_4, T_5, T_6), B(T_7), A(T_4), U(T_7, O_6, B_6, A_6), U(T_6, O_3, B_7, A_7), B(T_8), A(T_7), guasto

Soluzione:

1) Per prima cosa bisogna percorrere il log a ritroso fino al più recente record di check-point: CK(T1,T4,T5,T6)

   Si costruiscono gli insiemi di UNDO e di REDO:

   UNDO= { T_1, T_4, T_5, T_6 } REDO={}

2) Il log viene percorso in avanti, aggiornando i due insiemi:

B(T_7)	UNDO= { T_1, T_4, T_5, T_6, T_7 }	REDO={}
A(T_4)	UNDO= { T_1, T_4, T_5, T_6, T_7 }	REDO={}
B(T_8)	UNDO= { T_1, T_4, T_5, T_6, T_7, T_8 }	REDO={}
A(T_7)	UNDO= { T_1, T_4, T_5, T_6, T_7 }	REDO={}

3) Il log viene ripercorso ancora a ritroso, fino all'operazione I(T1,O1,A1), eseguendo le seguenti operazioni:

   O_3=B_7
   O_6=B_6
   O_5=B_5
   O_4=B_4
   O_3=B_3
   Delete O_1

4) Il log viene ripercorso in avanti per rieseguire le operazioni di REDO, ma essendo vuoto questo insieme, nessuna operazione verrà eseguita.

Esercizio 12.2

Si supponga che nella situazione precedente si verifichi un guasto di dispositivo che coinvolge gli oggetti O1, O2, O3; descrivere la ripresa a freddo.

Soluzione:

La ripresa a freddo è articolata in tre fasi successive.

1. Il log viene percorso a ritroso fino al primo record DUMP e si ricopia selettivamente la parte deteriorata della base dati.

2. Si ripercorre in avanti il log, applicando relativamente alla parte deteriorata della base di dati sia le azioni sulla base di dati sia le azioni di commit o abort e riportandosi così nella situazione precedente al guasto.

   Insert $O_1=A_1$
   Delete O_2
   $O_3=A_3$
   Commit (T_2)
   Abort (T_4)
   $O_3=A_7$

3. Si svolge una ripresa a caldo.

Esercizio 12.3

Il check-point, nei vari DBMS, viene realizzato in due modi diversi:

1. in alcuni sistemi si prende nota delle transazioni attive e si rifiutano (momentaneamente) nuovi commit
2. in altri si inibisce l'avvio di nuove transazioni e si attende invece la conclusione (commit o abort) delle transazioni attive

Spiegare, intuitivamente, le differenze che ne conseguono sulla gestione delle riprese a caldo.

Soluzione:

Nel primo caso la ripresa a caldo è articolata in quattro fasi ben precise:
1) Si accede al check point e si costruiscono gli insiemi di UNDO e di REDO.
2) Si percorre il log in avanti aggiornando i due insiemi
3) Il log viene ripercorso all'indietro disfacendo le operazioni contenute nell'insieme di UNDO.
4) Si ripercorre il log in avanti effettuando le operazioni di REDO.

Nel secondo caso il check point non conterrà nessuna transazione, in quanto tutte le transazioni si sono concluse o con un commit o con un abort, quindi non ci sono transazioni attive.

Le operazioni di ripresa a caldo saranno sicuramente molto più semplici e veloci se si utilizza ,la seconda tecnica, in quanto è sufficiente rieseguire tutte le operazioni che seguono il record di check point. Di contro bisogna dire che nel secondo caso la base di dati viene fermata ogni volta che si deve eseguire un check point, con un conseguente degrado delle prestazioni.

Dovendo decidere quale delle due soluzioni adottare, la prima sarà normalmente preferibile, in quanto è opportuno avere delle buone prestazioni per la maggior parte del tempo piuttosto che al verificarsi di eventi che richiedono una ripresa a caldo, considerati rari.

Esercizio 12.4

Indicare se i seguenti schedule possono produrre anomalie; i simboli *ci* e *ai* indicano l'esito (commit o abort) della transazione.

1. r1(x), w1(x), r2(x), w2(y), a1, c2
2. r1(x), w1(x), r2(y), w2(y), a1, c2
3. r1(x), r2(x), r2(y), w2(y), r1(z), a1, c2
4. r1(x), r2(x), w2(x), w1(x), c1, c2
5. r1(x), r2(x), w2(x), r1(y) , c1, c2
6. r1(x), w1(x), r2(x), w2(x) , c1, c2

Soluzione:

1) r1(x), w1(x), r2(x), w2(y), a1, c2

L'operazione r2(x) legge il valore scritto da w1(x), ma la transazione 1 termina con un abort. Questo è un caso di lettura sporca (Dirty Read)

2) r1(x), w1(x), r2(y), w2(y), a1, c2

Questo schedule non produce anomalie, perché le due transazioni fanno riferimento a oggetti differenti.

3) r1(x), r2(x), r2(y), w2(y), r1(z), a1, c2

Questo schedule non produce anomalie, perché la transazione 1 che termina in abort non effettua operazioni di scrittura.

4) r1(x), r2(x), w2(x), w1(x), c1, c2

Questo schedule ha una perdita di aggiornamento(lost update), in quanto gli effetti della transazione 2 vengono persi.

5) r1(x), r2(x), w2(x), r1(y) , c1, c2

Questo schedule non produce anomalie.

6) r1(x), w1(x), r2(x), w2(x) , c1, c2

Questo schedule non produce anomalie

Esercizio 12.5

Indicare se i seguenti schedule sono VSR.

1. r1(x), r2(y), w1(y), r2(x) w2,(x)
2. r1(x), r2(y), w1(x), w1(y), r2(x) w2,(x)
3. r1(x), r1(y), r2(y), w2(z), w1(z), w3(z), w3(x)
4. r1(y), r1(y), w2(z), w1(z), w3(z), w3(x), w1(x)

Soluzione:

Gli archi presentati indicano solo la relazione LEGGE - DA

1) r1(x), r2(y), w1(y), r2(x) w2(x)

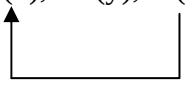
   Questo schedule non è VSR, perché i due schedule seriali:

   S1: r1(x), w1(y), r2(y), r2(x), w2(x) e

   S2: r2(y), r2(x), w2(x), r1(x), w1(y)

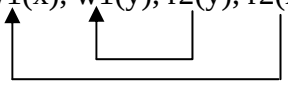
   non sono view-equivalenti con lo schedule dato. Hanno entrambi una differente relazione LEGGE - DA

2) r1(x), r2(y), w1(x), w1(y), r2(x) w2,(x)

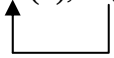


   Questo schedule non è VSR perché gli schedule

   S1: r1(x), w1(x), w1(y), r2(y), r2(x), w2(x)



   S2: r2(y), r2(x), w2(x), r1(x), w1(x),w1(y)



   hanno entrambi una differente relazione LEGGE – DA

3) r1(x), r1(y), r2(y), w2(z), w1(z), w3(z), w3(x)

   Questo schedule è VSR e view-equivalente allo schedule seriale, in quanto sono caratterizzati dalle stesse scritture finali e dalle stesse relazioni LEGGI –DA.

   S: r2(y), w2(z), r1(x), r1(y), w1(z), w3(z), w3(x)

4) r1(y), r1(y), w2(z), w1(z), w3(z), w3(x), w1(x)

Si noti che la transazione 1 ha due scritture, una su Z ed un'altra su X, ma anche la transazione 3 ha due scritture, una su Z ed un'altra su X. Nello schedule di partenza, le scritture finali su X e Z sono originate rispettivamente dalle transazioni 1 e 3. Nessuno schedule seriale potrà esibire le stesse scritture finali. Questo schedule non è quindi VSR.

Esercizio 12.6

Classificare i seguenti schedule (come: NonSR, VSR, CSR); nel caso uno schedule sia VSR oppure CSR, indicare tutti gli schedule seriali e esso equivalenti.

1. r1(x), w1(x), r2(z), r1(y), w1(y), r2(x), w2(x), w2(z)
2. r1(x), w1(x), w3(x), r2(y), r3(y), w3(y), w1(y), r2(x)
3. r1(x), r2(x), w2(x), r2(x), r4(z), w1(x), w3(y), w3(x), w1(y), w5(x), w1(z), w5(y), r5(z)
4. r1(x), r3(y), w1(y), w4(x), w1(t), w5(x), r2(z), r3(z), w2(z), w5(z), r4(t), r5(t)
5. r1(x), r2(x), w2(x), r3(x), r4(z), w1(x), r3(y), r3(x), w1(y), w5(x), w1(z), r5(y), r5(z)
6. r1(x), r1(t), r3(z), r4(z), w2(z), r4(x), r3(x), w4(x), w4(y), w3(y), w1(y), w2(t)
7. r2(x), r4(x), w4(x), r1(y), r4(z), w4(z), w3(y), w3(z), w1(t), w2(z), w2(t)

Soluzione:

1) r1(x), w1(x), r2(z), r1(y), w1(y), r2(x), w2(x), w2(z)

   In questo schedule non ci sono conflitti, quindi è sia VSR che CSR, ed è conflict-equivalente a:

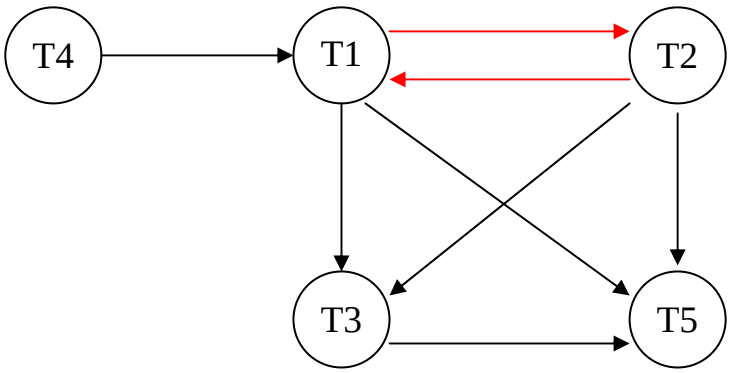
   S: r1(x), w1(x), r1(y), w1(y), r2(z), r2(x), w2(x), w2(z)

2) r1(x), w1(x), w3(x), r2(y), r3(y), w3(y), w1(y), r2(x)

   Questo schedule è NonSR. In uno schedule seriale view-equivalente a questo schedule la transazione 1 dovrebbe seguire la transazione 3 a causa delle SCRITTURE FINALI su Y, ma dovrebbe anche precedere la transazione 3 a causa della relazione LEGGE – DA su X.

3) r1(x), r2(x), w2(x), r3(x), r4(z), w1(x), w3(y), w3(x), w1(y), w5(x), w1(z), w5(y), r5(z)

   Per classificare questo schedule si deve realizzare un grafo dei conflitti.
   Consideriamo le operazioni relative a ogni risorsa separatamente:

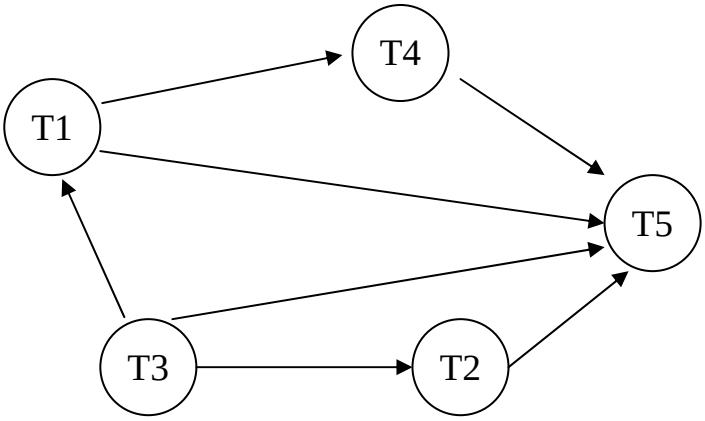


X: r1, r2, w2, r3, w1, w3, w5
Y: w3, w1, w5
Z: r4, w1, r5

   Questo schedule non è CSR perchè il suo grafo dei confliti è ciclico.
   Questo schedule è anche NonSR, perché, considerando la risorsa X, la transazione 1 dovrebbe precedere la transazione 2, e la transazione 2 dovrebbe precedere la transazione 1 (entrambe le transazioni leggono X prima di qualsiasi altra operazione).

4) r1(x), r3(y), w1(y), w4(x), w1(t), w5(x), r2(z), r3(z), w2(z), w5(z), r4(t), r5(t)

Per classificare questo schedule si deve realizzare un grafo dei conflitti.
Consideriamo le operazioni relative a ogni risorsa separatamente:



T: w1, r4, r5
X: r1, w4, w5
Y: r3, w1
Z: r2, r3, w2, w5

Questo schedule è sia CSR che VSR.

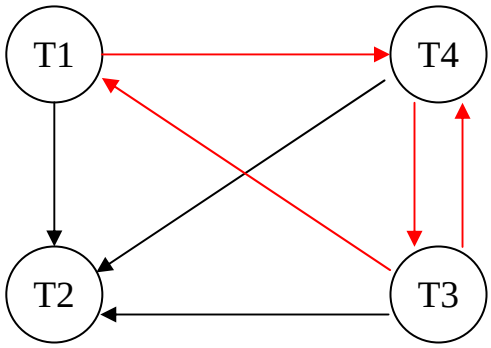
Gli schedule seriali equivalenti sono:
S1: r3(y), r3(z), r1(x), w1(y), w1(t), r2(z), w2(z), w4(x), r4(t), w5(x), w5(z), r5(t)
S2: r3(y), r3(z), r2(z), w2(z), r1(x), w1(y), w1(t), w4(x), r4(t), w5(x), w5(z), r5(t)

5) r1(x), r2(x), w2(x), r3(x), r4(z), w1(x), r3(y), r3(x), w1(y), w5(x), w1(z), r5(y), r5(z)

Lo schedule è NonSR ; le transazioni 1 e 2 leggono e scrivono X. Leggono X prima di ogni altra operazione, e così nessuna sequenza di schedule con queste transazioni potrà verificare la relazione LEGGE - DA

6) r1(x), r1(t), r3(z), r4(z), w2(z), r4(x), r3(x), w4(x), w4(y), w3(y), w1(y), w2(t)

Per classificare questo schedule si deve realizzare un grafico dei conflitti.
Consideriamo le operazioni relative a ogni risorsa separatamente:

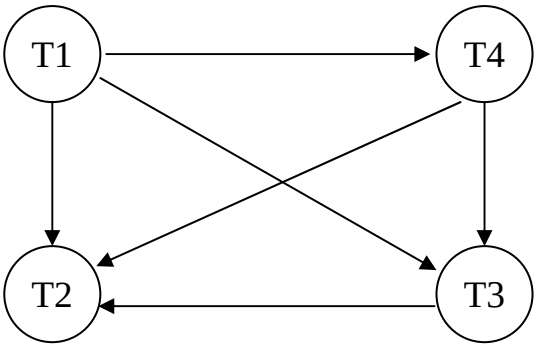


T: r1, w2
X: r1, r4, r3, w4
Y: w4, w3, w1
Z: r3, r4, w2

Questo schedule non è CSR perchè il suo grafo dei confliti è ciclico.
Questo schedule è anche NonSR; la transazione 1 ha la scrittura finale su Y, perciò dovrebbe seguire la transazione 4. La transazione 4 scrive su X, e dovrebbe seguire la transazione 1.

7) r1(x), r4(x), w4(x), r1(y), r4(z), w4(z), w3(y), w3(z), w1(t), w2(z), w2(t)

Per classificare questo schedule si deve realizzare un grafo dei conflitti.
Consideriamo le operazioni relative a ogni risorsa separatamente:



T: w1, w2
X: r1, r4, w4
Y: r1, w3
Z: r4, w4, w3, w2

Visto che il grafo è aciclico è CSR, quindi anche VSR.

Lo schedule seriale equivalente è:

S: r1(x), r1(y), w1(t), r4(x), w4(x), r4(z), w4(z), w3(y), w3(z), w2(z), w2(t)

Esercizio 12.7

Se gli schedule dell'esercizio precedente si presentassero a uno scheduler che usa il locking a due fasi, quali transazioni verrebbero messe in attesa? Si noti che, una volta posta in attesa una transazione, le sue successive azioni non vanno più considerate.

Soluzione:

1) r1(x), w1(x), r2(z), r1(y), w1(y), r2(x), w2(x), w2(z)

   Nessuna transazione in attesa

2) r1(x), w1(x), w3(x), r2(y), r3(y), w3(y), w1(y), r2(x)

   Le transazioni 3 e 1 e poi 2 vengono messe in attesa, producendo una situazione di deadlock; infatti, l'azione r2(x) deve aspettare l'oggetto x, messo in lock dalla transazione 1, e la transazione 1 è in attesa su w1(y) dell'oggetto y, messo in lock dalla transazione 2.

3) r1(x), r2(x), w2(x), r3(x), r4(z), w1(x), w3(y), w3(x), w1(y), w5(x), w1(z), w5(y), r5(z)

   Le transazioni 2, 1, 3 e 5 sono messe in attesa a causa del lock in lettura su x. Si crea una situazione di deadlock dovuta alla richiesta di lock in scrittura su x da parte delle transazioni 1 e 2, entrambe con un lock in lettura sulla stessa risorsa x.

4) r1(x), r3(y), w1(y), w4(x), w1(t), w5(x), r2(z), r3(z), w2(z), w5(z), r4(t), r5(t)

   Le transazioni 1, 4, 5 e 2 sono in attesa. La transazione 1 deve aspettare per y (allocato da t3), le transazioni 4 e 5 devono aspettare per x (allocato da t1) e la transazione 2 deve aspettare per z (allocato da t3).

5) r1(x), r2(x), w2(x), r3(x), r4(z), w1(x), r3(y), r3(x), w1(y), w5(x), w1(z), r5(y), r5(z)

   Le transazioni 2, 1 e 5 sono in attesa. Devono aspettare per x (allocata da t1, t2 e t3)

6) r1(x), r1(t), r3(z), r4(z), w2(z), r4(x), r3(x), w4(x), w4(y), w3(y), w1(y), w2(t)

   Le transazioni 2, 3 e 4 vengono messe in attesa. t2 e t4 devono aspettare per z (allocata da t3) e t3 deve aspettare per y (allocata da t1)

7) r1(x), r4(x), w4(x), r1(y), r4(z), w4(z), w3(y), w3(z), w1(t), w2(z), w2(t)

   Assumendo che t1 rilasci il lock esclusivo su t al termine delle sue operazioni, le transazioni 3 e 4 sono in attesa. Devono aspettare per x e y, allocate da t1.

Esercizio 12.8

Definire le strutture dati necessarie per la gestione del locking supponendo un modello non gerarchico e con read ripetibili; implementare in un linguaggio di programmazione a scelta le funzioni lock_r, lock_w e unlock. Si supponga disponibile un tipo di dato astratto "coda" con le opportune funzioni per inserire un elemento in coda ed estrarre il primo elemento da una coda.

Soluzione:

Le transazioni sono identificate con un numero, gli oggetti con una stringa.

```
typedef struct resource {
     char *identifier;
     int free; // 1 is free, 0 is busy
     int r_locked; // 1 or more locked, 0 free
     int w_locked; // 1 locked
     queue waiting_transaction;
}

typedef struct queue_element {
     int transaction;
     int type // 0 read, 1 write
}

typedef *resource locktable;

locktable lt=new locktable [N]; // N is the number of resources

int lock_r (int transaction, char *resource) {
     int i=0;
     int end=0;
     while ((i<N) && (!end))
       if (strcmp(resource, lt[i])!=0) i++; else end=1;
     if (!end) return -1; // resource not found
     if (lt[i].free)
     { lt[i].free=0;
       lt[i].r_locked=1;
       return transaction; }
       if (lt[i].r_locked)
       { lt[i].r-locked++;
         return transaction; }
     // the resource is w_locked

     queue_element q=new queue_element;
     q->transaction=transaction;
     q->type=0;
     in_queue(q,lt[i].queue);
     return 0; }

int lock_w (int transaction, char *resource) {
     int i=0;
     int end=0;
```

```
     while ((i<N) && (!end))
       if (strcmp(resource, lt[i])!=0) i++; else end=1;
     if (!end) return -1; // resource not found
     if (lt[i].free)
          { lt[i].free=0:
            lt[i].w_locked=1;
            return transaction;
          }
     queue_element q=new queue_element;
     q->transaction=transaction;
     q->type=1;
     in_queue(q,lt[i].queue);
     return 0;
}

int unlock (int transaction, char *resource) {
     int i=0;
     int end=0;
     while ((i<N) && (!end))
       if (strcmp(resource, lt[i])!=0) i++; else end=1;
     if (!end) return -1; // resource not found

     if (lt[i].r_locked)
          { lt[i].r_locked--;
          if (lt[i].r_locked) return 0;
          if (is_empty(lt[i].queue))
               { lt[i].free=1;
               return 0;
          }
          queue_element q=out_queue(lt[i].queue);
          lt[i].w_locked=1;

          return q.transaction;
          }
     if (is_empty(lt[i].queue))
          { lt[i].free=1;
            return 0;
          }
     queue_element q=out_queue(lt[i].queue);
     if (!(q.type)) // read transaction in queue
          { lt[i].r_locked=1;
            lt[i].w_locked=0; }
          // now unlock extracts all the read-transaction from
          //the queue;
     while (!(first_element(lt[i].queue)).type))
          { out_queue (lt[i].queue);
            r_locked++;
          }
     return q.transaction; }
```

Esercizio 12.9

Facendo riferimento all'esercizio precedente, aggiungere un meccanismo di timeout; si suppongano disponibili le funzioni di sistema per impostare il timeout, per verificare (a controllo di programma) se il tempo fissato è trascorso, e per estrarre uno specifico elemento da una coda.

Soluzione:

La struttura queue_element deve essere modificata nel seguente modo:

```
typedef struct queue_element {
      int transaction;
      int type;
      int time;
}
```

Il campo time rappresenta l'istante in cui la transazione viene messa in attesa.
Le funzioni lock_r e lock_w devono riempire anche questo campo prima di mettere le transazioni in coda, usando l'istruzione:

```
q.time=get_system_time();
```

Il meccanismo di timeout è realizzato da una nuova funzione, check_time, che è chiamata periodicamente dal sistema.

```
void check_time() {
      int now=get_system_time();
      for (int i=0:; i<N; i++)
      if (!is_empty(lt[i].queue))
      { int l=queue_length(lt[i].queue);
      for (int j=0; j<l; j++)
            { queue_element q=get_queue_element(lt[i].queue,j);
                  if ((q.time+MAX_TIME)<now)
                  remove_from_queue(lt[i].queue,j);
            }
      }
}
```

Esercizio 12.10

Se gli schedule descritti nell'esercizio 2.6 si presentassero a uno scheduler basato su timestamp, quali transazioni verrebbero abortite?

Soluzione:

1) r1(x), w1(x), r2(z), r1(y), w1(y), r2(x), w2(x), w2(z)

Operazione	**Risposta**	**Nuovo Valore**
read(x,1)	Ok	RTM(x)=1
write(x,1)	Ok	WTM(x)=1
read(z,1)	Ok	RTM(z)=1
read(y,1)	Ok	RTM(y)=1
write(y,1)	Ok	WTM(y)=1
read(x,2)	Ok	RTM(x)=2
write(z,2)	Ok	WTM(z)=2

Non viene abortita nessuna transazione.

2) r1(x), w1(x), w3(x), r2(y), r3(y), w3(y), w1(y), r2(x)

Operazione	**Risposta**	**Nuovo Valore**
read(x,1)	Ok	RTM(x)=1
write(x,1)	Ok	WTM(x)=1
write(x,3)	Ok	WTM(x)=3
read(y,2)	Ok	RTM(y)=2
read(y,3)	Ok	RTM(y)=3
write(y,3)	Ok	WTM(y)=3
write(y,1)	t_1 aborted	
read(x,2)	t_2 aborted	

3) r1(x), r2(x), w2(x), r3(x), r4(z), w1(x), w3(y), w3(x), w1(y), w5(x), w1(z), w5(y), r5(z)

Operazione	**Risposta**	**Nuovo Valore**
read(x,1)	Ok	RTM(x)=1
read(x,2)	Ok	RTM(x)=2
write(x,2)	Ok	WTM(x)=2
read(x,3)	Ok	RTM(x)=3
read(z,4)	Ok	RTM(z)=4
write(x,1)	t_1 aborted	
write(y,3)	Ok	WTM(y)=3
write(x,5)	Ok	WTM(x)=5
write(y,5)	Ok	WTM(y)=5
read(z,5)	Ok	RTM(z)=5

4) r1(x), r3(y), w1(y), w4(x), w1(t), w5(x), r2(z), r3(z), w2(z), w5(z), r4(t), r5(t)

Operazione	**Risposta**	**Nuovo Valore**
read(x,1)	Ok	RTM(x)=1
read(y,3)	Ok	RTM(y)=3
write(y,1)	t_1 aborted	
write(x,4)	Ok	WTM(x)=4
write(x,5)	Ok	WTM(x)=5
read(z,2)	Ok	RTM(z)=2
read(z,3)	Ok	RTM(z)=3
write(z,2)	t_2 aborted	
write(z,5)	Ok	WTM(z)=5
read(t,4)	Ok	RTM(t)=4
read(t,5)	Ok	RTM(t)=5

5) r1(x), r2(x), w2(x), r3(x), r4(z), w1(x), r3(y), r3(x), w1(y), w5(x), w1(z), r5(y), r5(z)

Operazione	**Risposta**	**Nuovo Valore**
read(x,1)	Ok	RTM(x)=1
read(x,2)	Ok	RTM(x)=2
write(x,2)	Ok	WTM(x)=2
read(x,3)	Ok	RTM(x)=3
read(z,4)	Ok	RTM(z)=4
write(x,1)	t_1 aborted	
read(y,3)	Ok	RTM(y)=3
read(x,3)	Ok	RTM(x)=3
write(x,5)	Ok	WTM(x)=5
read(y,5)	Ok	RTM(y)=5
read(z,5)	Ok	RTM(z)=5

6) r1(x), r1(t), r3(z), r4(z), w2(z), r4(x), r3(x), w4(x), w4(y), w3(y), w1(y), w2(t)

Operazione	**Risposta**	**Nuovo Valore**
read(x,1)	Ok	RTM(x)=1
read(t,1)	Ok	RTM(t)=1
read(z,3)	Ok	RTM(z)=3
read(z,4)	Ok	RTM(z)=4
write(z,2)	t_2 aborted	
read(x,4)	Ok	RTM(x)=4
read(x,3)	Ok	RTM(x)=4
write(x,4)	Ok	WTM(x)=4
write(y,4)	Ok	WTM(y)=4
write(y,3)	t_3 aborted	
write(y,1)	t_1 aborted	

7) r1(x), r4(x), w4(x), r1(y), r4(z), w4(z), w3(y), w3(z), w1(t), w2(z), w2(t)

Operazione	**Risposta**	**Nuovo Valore**
read(x,1)	Ok	RTM(x)=1
read(x,4)	Ok	RTM(x)=4
write(x,4)	Ok	WTM(x)=4
read(y,1)	Ok	RTM(y)=1
read(z,4)	Ok	RTM(z)=4
write(z,4)	Ok	WTM(z)=4
write(y,3)	Ok	WTM(y)=3
write(z,3)	t_3 aborted	
write(t,1)	Ok	WTM(t)=1
write(z,2)	t_2 aborted	

Esercizio 12.11

Si consideri un oggetto X sul quale opera un controller di concorrenza basato su timestamp, con WTM(X) = 5, RTM(X) = 7. Indicare le azioni dello scheduler a fronte del seguente input nel caso mono-versione e in quello multi-versione:
r(x,8), r(x,17), w(x,16), w(x,18), w(x,23), w(x,29), r(x,20), r(x,30), r(x,25)

Soluzione:

Mono-versione:

Operazione	**Risposta**	**Nuovo Valore**
		WTM(x)=5
		RTM(x)=7
read(x,8)	Ok	RTM(x)=8
read(x,17)	Ok	RTM(x)=17
read(x, 16)	Ok	RTM(x)=17
write(x,18)	Ok	WTM(x)=18
write(x,23)	Ok	WTM(x)=23
write(x,29)	Ok	WTM(x)=29
read(x,20)	t_{20} aborted	
read(x,30)	Ok	RTM(x)=30
read(x,25)	t_{25} aborted	

Multi-versione:

Operazione	**Risposta**	**Nuovo Valore**
		$WTM_1(x)$=5
		RTM(x)=7
read(x,8)	Ok	RTM(x)=8
read(x,17)	Ok	RTM(x)=17
read(x, 16)	Ok	RTM(x)=17
write(x,18)	Ok	$WTM_2(x)$=18
write(x,23)	Ok	$WTM_3(x)$=23
write(x,29)	Ok	$WTM_4(x)$=29
read(x,20)	Ok	RTM(x)=20 reads from x_2
read(x,30)	Ok	RTM(x)=30 reads from x_4
read(x,25)	Ok	RTM(x)=30 reads from x_3

Esercizio 12.12

Spiegare perché il livello più alto di isolamento previsto nello standard SQL:1999 (“`serializable`”) può avere effetti molto più pesanti anche solo del livello immediatamente inferiore (“`repeatable read`”).

Soluzione:

Il livello `serializable` può avere effetti molto più pesanti rispetto al livello immediatamente inferiore, in quanto applica i lock di predicato che, in assenza di indici, possono bloccare gli accessi ad intere relazioni.