



IS210 – Chương 4

Điều khiển đồng thời

Concurrency Control- Deadlock

Trương Thu Thủy



Nội dung

- Khả tuần tự và khả phục hồi
- Deadlock

Vấn đề dữ liệu rác - Dirty-data problem

	T ₁	T ₂	A	B
S	Lock(A) ; Read(A); A:=A+100 Write(A); Lock(B) ; Unlock(A) Read(B) Abort ; Unlock(B)	Lock(A) ; Read(A); A:=A*2 Write(A); Lock(B) ; Từ chối Lock(B) ; Unlock(A) ; Read(B) B:=B*2 Write(B); Unlock(B)	25 125 250	25 50

T₁ ghi dirty data và sau đó hủy bỏ

Vấn đề dữ liệu rác - Dirty-data problem

- T_1 đọc dữ liệu rác từ T_2 và phải hủy bỏ nếu T_2 hủy

T_1	T_2	T_3
$r_1(B)$	$w_2(B)$	
	$r_2(A)$	
	$w_2(C)$	$r_3(C)$
	Abort	
		$w_3(A)$

Quay lui dây chuyền - Cascading Rollback

- Nếu dữ liệu rác có mặt trong các giao tác, đôi khi phải thực hiện quay lui theo dây chuyền
 - Hủy bỏ đệ quy tất cả giao tác đã đọc dữ liệu “được ghi bởi giao tác bị hủy bỏ”

Lịch khả phục hồi - Recoverable Schedule

- Lịch khả phục hồi nếu
 - Mỗi giao tác chỉ commit sau khi tất cả các giao tác, mà nó đọc dữ liệu, đã commit
- Ví dụ:
 - T_2 đọc giá trị B được ghi bởi T_1
 - Do đó T_2 phải commit sau T_1 để lịch bên là khả phục hồi

T_1	T_2
$w_1(A)$	
$w_1(B)$	
	$w_2(A)$
	$r_2(B)$
commit	
	commit

Lịch khả phục hồi - Recoverable Schedule

- Lịch bên thì khả phục hồi, nhưng không khả tuần tự

T_1	T_2
	$w_2(A)$
$w_1(A)$	
$w_1(B)$	
	$r_2(B)$
commit	
	commit

T_2 phải thực hiện trước T_1 theo tuần tự bởi vì hành động ghi đơn vị dữ liệu A, nhưng T_1 phải thực hiện trước T_2 bởi vì hành động ghi và đọc đơn vị dữ liệu B. Lịch trên không khả tuần tự
Lịch trên khả phục hồi vì T_2 (đọc dữ liệu của T_1) commit sau khi T_1 đã commit

Lịch khả phục hồi - Recoverable Schedule

- Lịch bên khả tuần tự nhưng không khả phục hồi
 - T_1 thực hiện trước T_2 nhưng thứ tự commit xảy ra không theo thứ tự.
 - Trước khi có sự cố, giá trị đã được commit bởi T_2 được ghi vào ổ đĩa nhưng T_1 thì chưa
- T_2 đọc dữ liệu rác nhưng không thể bị hủy (vì đã commit)


T_1	T_2
$w_1(A)$	
$w_1(B)$	
	$w_2(A)$
	$r_2(B)$
	commit
sự cố → commit	

Lịch chống quay lui dây chuyền

Schedules That Avoid Cascading Rollback

- Lịch khả phục hồi thỉnh thoảng yêu cầu quay lui theo dây chuyền
 - Nếu sau bước 4 có sự cố xảy ra $\rightarrow T_1$ phải rollback
 - Thì T_2 cũng phải rollback theo.
- Để tránh quay lui dây chuyền thì cần phải
 - Có một điều kiện mạnh hơn khả phục hồi
- \rightarrow Lịch chống quay lui dây chuyền nếu
 - Giao tác chỉ đọc những giá trị được ghi bởi các giao tác đã commit

	T_1	T_2
1	$w_1(A)$	
2	$w_1(B)$	
3		$w_2(A)$
4		$r_2(B)$
5	commit	
6		commit

sự cố 

Lịch chống quay lui dây chuyền

- T_2 chỉ đọc B sau khi T_1 (giao tác cuối cùng ghi B) đã commit, và giá trị đã được ghi trên ổ đĩa. Do đó, lịch bên thỏa chống quay lui dây chuyền

T_1	T_2
$w_1(A)$	
$w_1(B)$	
	$w_2(A)$
commit	
	$r_2(B)$
	commit

Deadlock

- Định nghĩa
 - Là tình trạng hai hay nhiều giao tác đang tranh chấp tài nguyên và mỗi giao tác đang đợi tài nguyên bị nắm giữ bởi các giao tác khác và không một giao tác nào có thể thực hiện tiếp được

Phát hiện deadlock dùng đồ thị chờ

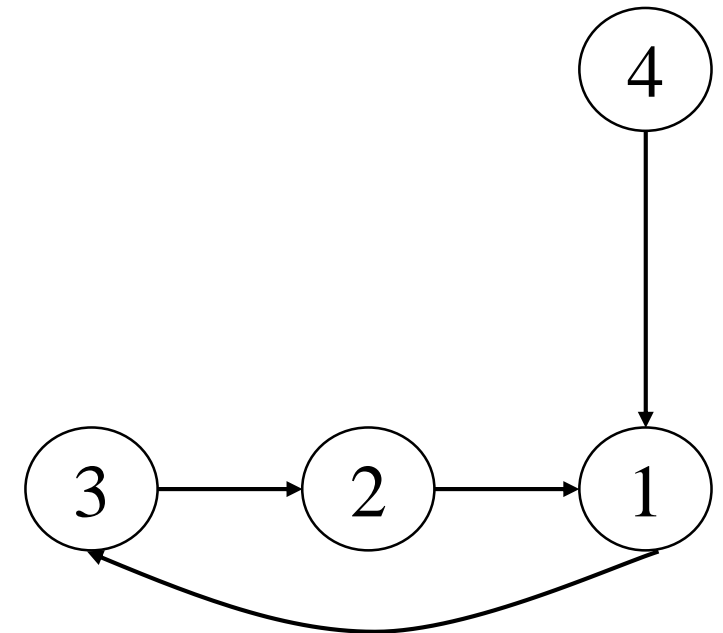
Waits-for graph

- Vẽ một cung từ T tới U nếu có một đơn vị dữ liệu A sao cho
 1. U nắm giữ một lock trên A
 2. T đang đợi để lock A,
 3. T không lock A trừ khi U nhả lock này
- Nếu không có chu trình trong đồ thị chờ thì từng giao tác có thể kết thúc
- Nếu có chu trình, không giao tác nào trong chu trình có thể tiếp tục, do đó có deadlock.

Ví dụ

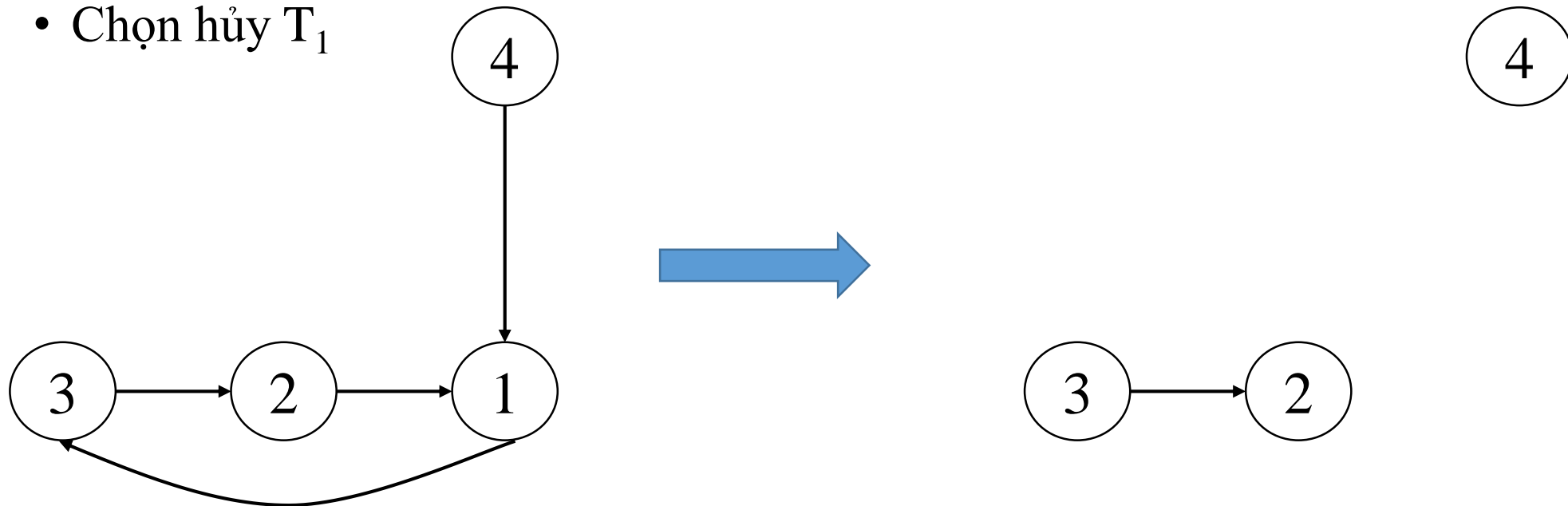
	T ₁	T ₂	T ₃	T ₄
1	Lock(A); Read(A)			
2		Lock(C); Read(C)		
3			Lock(B); Read(B)	
4				Lock(D); Read(D)
5		Lock(A)		
6		Denied	Lock(C)	
7			Denied	Lock(A)
8	Lock(B)			Denied

Denied



Phương pháp giải quyết deadlock (Prevention)

- Phương pháp:
 - Hủy đi đỉnh (giao tác) nằm trong chu trình (chọn đỉnh có số cung đi vào đi ra nhiều nhất)
- Trở lại ví dụ trên
 - Chọn hủy T_1

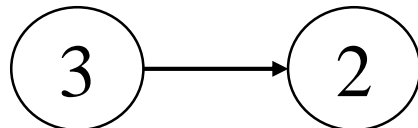


Phương pháp giải quyết deadlock (Prevention)

- Trở lại ví dụ trên
 - Chọn hủy $T_1 \rightarrow T_1$ giải phóng lock trên A và lock này sẽ giao cho T_2 hoặc T_4
 - Giải sử lock(A) được giao cho T_2 , sau đó T_2 hoàn thành và giải phóng lock trên A và C.
 - Sau đó T_3 , và T_4 hoàn thành.
 - Tại một thời điểm nào đó, T_1 khởi động lại nhưng không thể lấy được lock trên A và B cho đến khi T_2 , T_3 và T_4 hoàn tất

	T_1	T_2	T_3	T_4
1	Lock(A); Read(A)			
2		Lock(C); Read(C)		
3			Lock(B); Read(B)	
4				Lock(D); Read(D)
5		Lock(A)		
6		Denied	Lock(C)	
7			Denied	Lock(A)
8	Lock(B)			Denied

Denied



Thảo luận

- Đồ thị chờ (waits-for graph) có thể sẽ rất lớn
- Phân tích để tìm ra chu trình mỗi khi giao tác đợi chờ lock có thể tốn thời gian.
- Một cách để cải tiến đồ thị chờ là sử dụng thêm nhãn thời gian

Phương pháp ngăn ngừa deadlock

- Mỗi giao tác liên kết với một nhãn thời gian. Nhãn thời gian này dành riêng cho deadlock (không liên quan đến nhãn thời gian trong xử lý đồng thời)
- Nhãn thời gian trong deadlock không bao giờ thay đổi
- Khi giao tác T đợi lock đang được nắm giữ bởi giao tác U. Dựa vào T hay U già hơn (older - có nhãn thời gian sớm hơn), mà có hai chính sách khác nhau được sử dụng để quản lý giao tác và phát hiện deadlock
 - *Wait-Die*
 - *Wound-Wait*

Phương pháp ngăn ngừa deadlock

T has to wait for a lock that is held by another transaction U

The Wait-Die Scheme:

If T is older than U ($TS(T) < TS(U)$)

T waits for the lock(s) held by U

If U is older than T

T “dies” -- it is rolled back.

The Wound-Wait Scheme:

If T is older than U

T “wounds” U .

U must roll back

If U is older than T

T waits for the lock(s) held by U .

Ví dụ

- Cho bốn giao tác:
 - $T_1: l(A); r(A); l(B); w(B); u(A); u(B)$
 - $T_2: l(A); l(C); r(C); w(A); u(A); u(C)$
 - $T_3: l(B); r(B); l(C); w(C); u(B); u(C);$
 - $T_4: l(A); l(D); r(D); w(A); u(A); u(D)$
- T_1, T_2, T_3, T_4 thực hiện theo thứ tự thời gian. Giả sử T_1 là oldest
- Khi một giao tác rollback, nó không restart quá sớm trước khi các giao tác khác kết thúc

	T ₁	T ₂	T ₃	T ₄
1	l(A); r(A)			
2		l(A) Dies		
3			l(B); r(B)	
4				l(A) Dies
5			l(C);w(C);	
6			u(B);u(C);	
7	l(B);w(B)			
8	u(A);u(B)			
9				l(A);l(D)
10		l(A) Wait		
11				r(D);w(A)
12				u(A);u(D)
13		l(A);l(C)		
14		r(C);w(A)		
15		u(A);u(C)		

Wait-die

- T₂ yêu cầu lock trên A và dies
- T₃ lấy lock trên B
- T₄ yêu cầu lock trên A và dies
- T₃: lấy lock trên C và hoàn thành
- Khi T₁ tiếp tục, nó thấy B không còn bị lock và hoàn thành bước 8
- T₂, T₄ bị rollback và bắt đầu lại. T₂ vẫn older hơn T₄
- **Giả sử** T₄ bắt đầu trước, T₂ yêu cầu lock trên A ở bước 10, và phải đợi
- T₄ hoàn thành ở bước 12
- T₂ hoàn thành sau đó 3 bước

	T ₁	T ₂	T ₃	T ₄
1	l(A); r(A)			
2		l(A) waits		
3			l(B); r(B)	
4				l(A) waits
5	l(B);w(B)		wounded	
6	u(A);u(B)			
7		l(A);l(C)		
8		r(C);w(A)		
9		u(A);u(C)		l(A);l(D)
				r(D);w(A)
				u(A);u(D)
			l(B); r(B)	
			l(C);w(C);	
			u(B);u(C);	

Wound-wait

- T₂ yêu cầu lock trên A và wait
- T₃ lấy lock trên B
- T₄ yêu cầu lock trên A và wait
- T₁: yêu cầu lock trên B ở bước 5, T₁ “wounds” T₃ → T₃ rollback, và T₁ hoàn thành
- Giả sử T₂ lấy được lock trên A và hoàn thành
- Sau T₂ kết thúc, lock trên A được trao cho T₄ và hoàn thành
- Cuối cùng, T₃ restart và hoàn thành

Bài tập

- Đối với mỗi lịch, giả sử shared lock được yêu cầu ngay lập tức trước mỗi hành động đọc, và exclusive lock được yêu cầu ngay lập tức sau mỗi hành động ghi. Unlock xảy ra ngay lập tức sau khi giao tác kết thúc. Vẽ đồ thị chờ (waits-for graph). Nếu xảy ra deadlock, nêu cách giải quyết.
 - a. $r_1(A); r_2(B); w_1(C); r_3(D); r_4(E); w_3(B); w_2(C); w_4(A); w_1(D);$
 - b. $r_1(A); r_2(B); r_3(C); w_1(B); w_2(C); w_3(D);$
 - c. $r_1(A); r_2(B); r_3(C); w_1(B); w_2(C); w_3(A);$
 - d. $r_1(A); r_2(B); w_1(C); w_2(D); r_3(C); w_1(B); w_4(D); w_2(A);$
- Giả sử thứ tự deadlock-timestamps là T_1, T_2, T_3, T_4 . Giả sử giao tác cần restart cũng theo thứ tự mà chúng rollback.
 - Làm lại câu trên sử dụng wound-wait
 - Làm lại câu trên sử dụng wait-die