Chương II. QUẢN LÝ GIAO TÁC

2.1 Giới thiệu

Hai yêu cầu cơ bản của ứng dụng khai thác CSDL trong thực tế:

– Cho phép **nhiều người dùng đồng thời khai thác** CSDL nhưng phải **giải quyết được các tranh chấp.**

– Sự cố kỹ thuật có thể luôn luôn xảy ra nhưng phải **giải quyết được vấn đề về nhất quán dữ liệu.**

Một số tình huống

Hệ thống đặt vé máy bay: – "Khi hành khách mua vé" **– "Khi hai/ nhiều hành khách cùng đặt một ghế trống"** → Lỗi: Có thể có **nhiều hành khách đều đặt được dù chỉ còn 1 ghế** → **Phải giải quyết được tranh chấp để đảm bảo được nhất quán dữ liệu.**	GHẾ (**Mã ghế**, Mã CB, Trạng thái) CHUYẾN BAY(**Mã CB**, Ngày giờ, Số ghế còn) **Thao tác của người dùng:** 1. Tìm thấy một ghế trống 2. Đặt ghế
Hệ thống ngân hàng: – "Khi chuyển tiền từ tài khoản A sang tài khoản B" → Lỗi: Có thể đã rút tiền từ A nhưng chưa cập nhật số dư của B. → Phải đảm bảo được nhất quán dữ liệu khi có sự cố – "Khi rút tiền của một tài khoản"	TÀI KHOẢN(**Mã TK**, Số dư) GIAO DỊCH(**Mã GD**, Loại, Số tiền) 1. update TAIKHOAN set SoDu=SoDu-50 where MATK=A 2. update TAIKHOAN set SoDu=SoDu+50 where MATK=B
– "Nhiều người cùng rút tiền trên một tài khoản" → **Lỗi:** Có thể rút **nhiều hơn số tiền thực** có. → Phải giải quyết được tranh chấp để đảm bảo được nhất quán dữ liệu.	1. Đọc số dư của tài khoản A vào X 2. Cập nhật số dư mới của tài khoản A bằng X – Số tiền
Hệ thống quản lý học sinh: – Thêm một học sinh mới → **Lỗi:** Có thể xảy ra trường hợp **học sinh đã được thêm** nhưng **sĩ số không được cập nhật.** → Phải đảm bảo được nhất quán dữ liệu khi có sự cố.	Lớp học(**Mã lớp**, Tên, Sĩ số) Học sinh (**Mã HS**, Họ tên, Mã lớp) 1. Thêm vào một học sinh của lớp 2. Cập nhật sĩ số lớp tăng lên 1

Nhận xét: Thường xuyên xảy ra vấn đề **nhất quán dữ liệu** nếu **một xử lý gặp sự cố** hoặc khi **các xử lý được gọi *truy xuất đồng thời*.**

Cần 1 khái niệm biểu diễn một đơn vị xử lý với các tính chất: **Nguyên tố – Cô lập – Nhất quán – Bền vững** → **Giao tác:** Là một khái niệm nền tảng của **điều khiển truy xuất đồng thời và khôi phục khi có sự cố.**

2.2 Giao tác

2.2.1 Định nghĩa

- Giao tác (Transaction) là **một đơn vị xử lý nguyên tố** gồm **một chuỗi các hành động đọc / ghi trên các đối tượng CSDL**

– Nguyên tố: Không thể phân chia được nữa → Giao tác là một đơn vị công việc **không thể chia nhỏ được nữa.**

- Trong kiến trúc hệ quản trị CSDL: **Bộ phận Điều khiển đồng thời đóng vai trò quản lý giao tác**

2.2.2 Tính chất ACID của giao tác

- **Tính nguyên tố (Atomicity):** Hoặc là toàn bộ hoạt động được phản ánh đúng đắn *(thực thi)* trong CSDL, hoặc không có hoạt động nào cả *(không có hoạt động nào được thực thi)*

Nếu xảy ra lỗi trong quá trình thực thi, tất cả các thao tác trong giao tác sẽ được **hoàn tác về ban đầu (rollback)**.

- **Tính nhất quán (Consistency):** Khi một giao tác kết thúc (thành công hay thất bại), CSDL phải ở trạng thái nhất quán (Đảm bảo mọi RBTV). Một giao tác đưa CSDL từ trạng thái nhất quán này sang trạng thái nhất quán khác.

Ví dụ: Nếu một ràng buộc rằng Balance >= 0 được thiết lập cho tài khoản ngân hàng, giao tác phải đảm bảo rằng không có tài khoản nào có số dư âm sau khi giao tác hoàn thành.

- **Cô lập (Isolation):** Một giao tác khi thực hiện sẽ không bị ảnh hưởng bởi các giao tác khác thực hiện **đồng thời** với nó.

- **Bền vững (Durability):** Sau khi giao tác commit thành công, tất cả những thay đổi trên CSDL mà giao tác đã thực hiện phải được ghi nhận chắc chắn (vào ổ cứng).

Mọi thay đổi trên CSDL được ghi nhận bền vững vào thiết bị lưu trữ dù có sự cố có thể xảy ra. SQL Server đảm bảo tính bền vững thông qua **transaction log**.

Ví dụ: Khi bạn chạy một giao tác và COMMIT, thay đổi của nó sẽ được ghi vào log và lưu trên đĩa. Ngay cả khi hệ thống bị tắt đột ngột, cơ sở dữ liệu vẫn có thể khôi phục từ log.

Ví dụ:

Chuyển khoản tiền từ tài khoản A sang tài khoản B

Giao tác Chuyển khoản
1. update TAIKHOAN set SoDu=SoDu-50 where MATK=A
2. update TAIKHOAN set SoDu=SoDu+50 where MATK=B

Cuối giao tác

Atomicity: Hoặc cả 2 bước đều thực hiện hoặc không bước nào được thực hiện.

Nếu có sự cố thì HQT CSDL có cơ chế khôi phục lại dữ liệu như lúc ban đầu.

Consitency: Với giao tác chuyển tiền, tổng số dư của A và B luôn luôn không đổi.

Thêm học sinh mới vào một lớp

Giao tác Thêm học sinh mới
1. Thêm một học sinh vào bảng học sinh
2. Cập nhật sĩ số của lớp tăng lên 1

Cuối giao tác

Atomicity: Hoặc cả 2 bước đều thực hiện hoặc không bước nào được thực hiện.

Nếu có sự cố thì HQT CSDL có cơ chế khôi phục lại dữ liệu như lúc ban đầu.

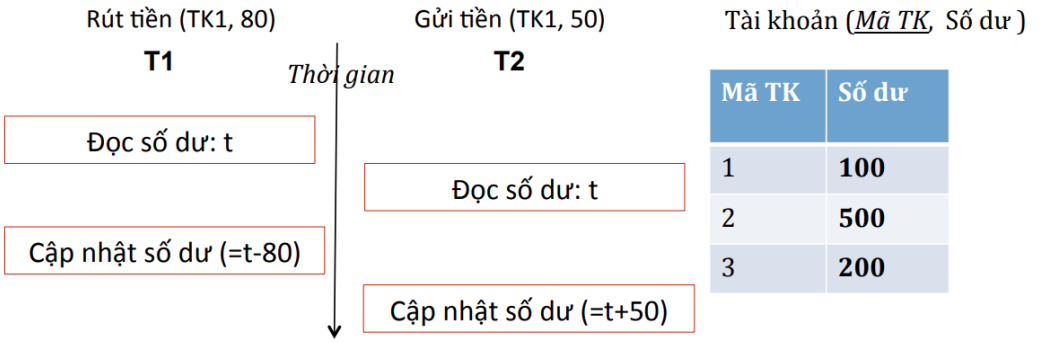
Lớp học(Mã lớp, Tên, Sĩ số)

Mã lớp	Tên	Sĩ số
1	10A	3

Học sinh (*Mã HS*, Họ tên, Mã lớp)

Mã HS	Họ tên	Mã lớp
1	An	1
2	Thảo	1
3	Bình	1

Consistency: Sĩ số của lớp phải luôn bằng số học sinh thực sự và **không quá 3.**



2 hành động xảy ra trên cùng 1 thời gian (đồng thời), nhưng thực hiện 2 tác vụ hoàn toàn độc lập.

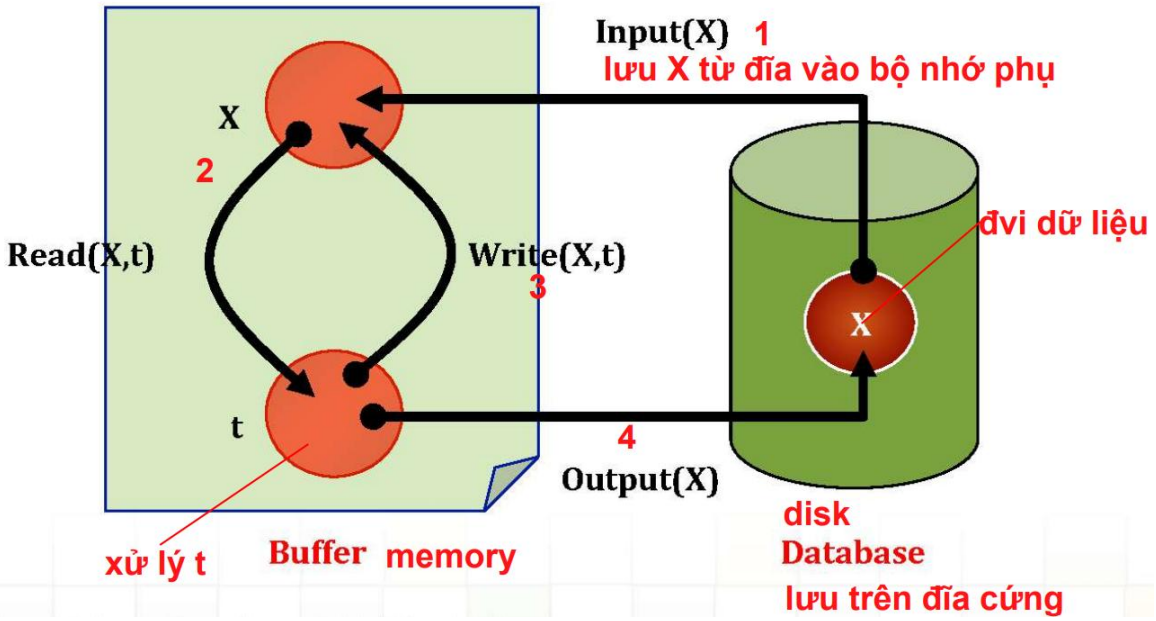
Isolation: Tính chất cô lập đảm bảo **mặc dù các giao tác có thể đan xen nhau nhưng kết quả của chúng tương tự với một kết quả tuần tự nào đó**

→ Các giao tác không bị ảnh hưởng bởi các giao tác khác khi thực thi.

2.2.3 Đơn vị dữ liệu

- *Đối tượng CSDL mà giao tác thực hiện các xử lý đọc/ghi* còn được gọi là **đơn vị dữ liệu.**
- Một đơn vị dữ liệu (element) có thể là các thành phần:
  - Quan hệ (Relations)
  - Bảng (Tables)
  - Khối dữ liệu trên đĩa (Blocks)
  - Bộ (Tuples)
- **Một CSDL bao gồm nhiều đơn vị dữ liệu.**

2.2.4 Các thao tác của giao tác



- **Read (A, t):** Đọc đơn vị dữ liệu A vào t (Đọc đơn vị dữ liệu A, lưu vào biến cục bộ t)
- **Write (A, t):** Ghi t vào đơn vị dữ liệu A (Lấy dữ liệu từ biến t, ghi vào đơn vị dữ liệu A)

Ví dụ:

Giả sử có 2 đơn vị dữ liệu A và B với ràng buộc **A = B** (nếu có một trạng thái nào đó mà A ≠ B thì sẽ mất tính nhất quán)

Giao tác T thực hiện 2 bước: **A = A * 2 ; B = B * 2**

Biểu diễn T:
Cách 1: ➔
Cách 2:
T: Read(A, t); t =t*2; Write(A, t); Read(B, t); t =t*2; Write(B, t)

T
Read(A, t);
t =t*2;
Write(A, t)
Read(B, t);
t =t*2;
Write(B, t)

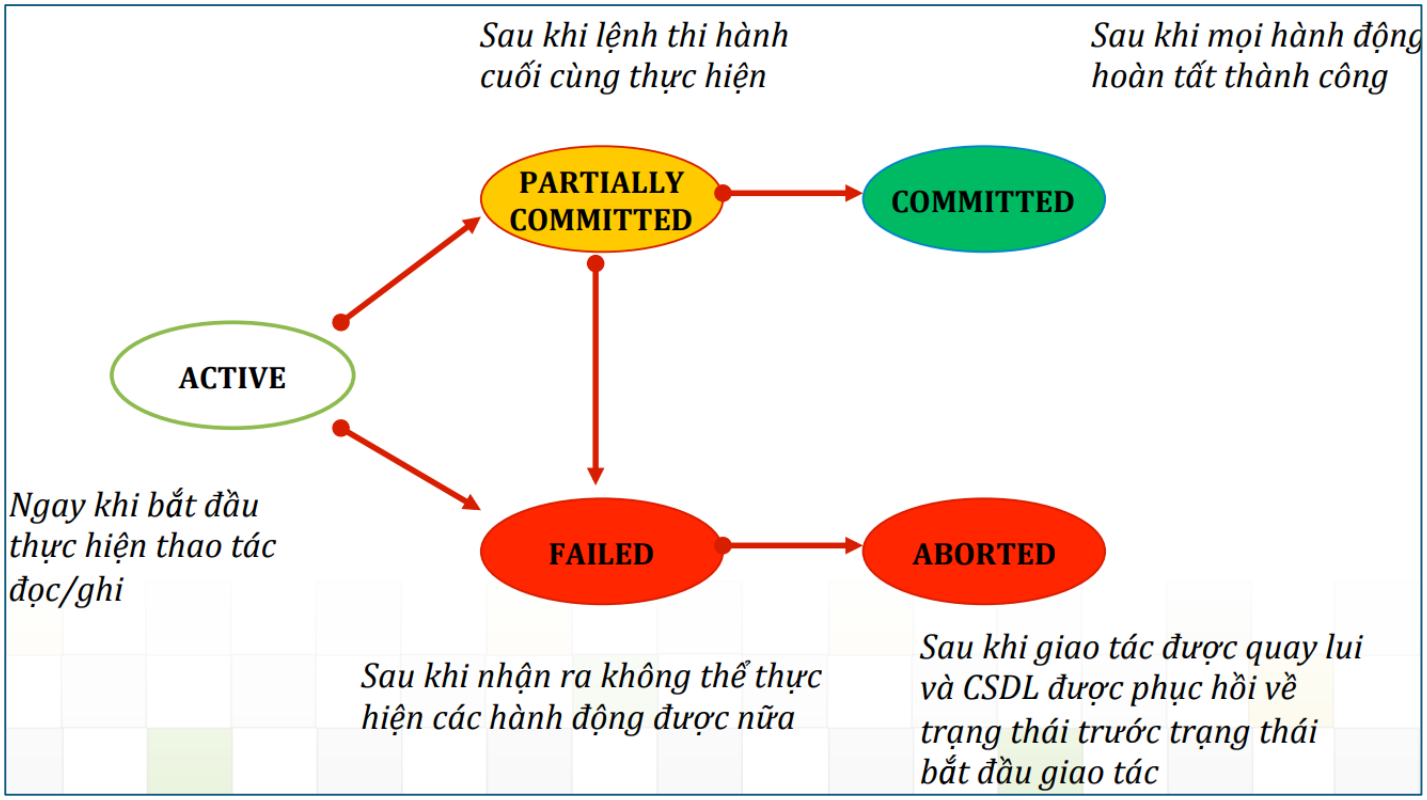
Ví dụ a/

Hành động	t	Mem A	Mem B	Disk A	Disk B
Input (A)		8		8	8
Read (A, t)	8	8		8	8
t := t * 2	16	8		8	8
Write (A, t)	16	16		8	8
Input (B)	16	16	8	8	8
Read (B, t)	8	16	8	8	8
t := t * 2	16	16	8	8	8
Write (B, t)	16	16	16	8	8
Output (A)	16	16	16	16	8
Output (B)	16	16	16	16	16

Ví dụ b/

	T	Mem A	Mem B	Disk A	Disk B
Input (A)		6		6	6
Read (A, t)	6	6		6	6
t = t*3	18	6		6	6
Write (A, t)	18	18		6	6
Input(B)		18	6	6	6
Read (B, t)	6	18	6	6	6
t = t*3	18	18	6	6	6
Write (B, t)	18	18	18	6	6
Output (A)	18	18	18	18	6
Output(B)	18	18	18	18	18

2.2.5 Các trạng thái của giao tác



2.2.6 Khai báo giao tác trong T-SQL

- **BEGIN TRANSACTION** Bắt đầu giao tác
- **COMMIT TRANSACTION** Kết thúc giao tác thành công
- **ROLLBACK TRANSACTION** Kết thúc giao tác không thành công. CSDL được đưa về tình trạng **trước khi thực hiện giao tác.**

Ví dụ: giao tác Chuyển khoản

```
create proc sp_ChuyenKhoan
@matk_chuyen char(10), @matk_nhan char(10), @sotien float
as
        declare @sodu float
        begin transaction
        select @sodu=SoDu from TAIKHOAN
        where MATK=@matk_chuyen
        if (@sodu < @sotien)
        begin
                rollback tran
                return
        end
        update TAIKHOAN set SoDu=SoDu-500000 where MATK=@matk_chuyen
        if (@@error <> 0) --có lỗi
        begin
                rollback tran
                return
        end
        update TAIKHOAN set SoDu=SoDu+500000 where MATK=@matk_nhan
        if (@@error <> 0) --nếu có lỗi từ hệ thống
        begin
```

```
            rollback tran
            return
        end
commit transaction
```

- Để đảm bảo tính chất A của giao tác, Người sử dụng phải tường minh điều khiển sự ***rollback*** của giao tác.
- Cần kiểm tra lỗi sau khi thực hiện mỗi thao tác trong giao tác để có thể xử lý rollback kịp thời.
- Trong SQL Server, dùng biến toàn cục **@@error và @@rowcount.**

Ví dụ: Cài đặt thêm sinh viên

```
Dùng Stored:
Create proc….
Begin
        /*Goi lệnh insert thêm mới vào SinhVien*/
        /* Đọc và kiểm tra SiSo lớp */
        /* Nếu SiSo >= Max , báo lỗi */
        rollback transaction

        /*Cập nhật tăng sĩ số*/
        /* Nếu có lỗi khi thực hiện thao tác :*/
        rollback transaction
End
```

```
Create proc sp_ThemSV @MaSV int, @MaLop int
As
        begin transaction
                declare @SiSo int
                select @SiSo = SiSo
                        from LOP
                        where MaLop = @MaLop
                if(@SiSo>=Max)
                begin
                        rollback transaction
                        return
                end

                Insert into SINH_VIEN(MaSV,MaLop)
                values(@MaSV,@MaLop)
                if(@@error<>0)
                begin
                        rollback transaction
                        return
                end

                update LOP
                        set SiSo = SiSo+1
                        where MaLop = @MaLop
                if(@@error<>0)
                begin
```

```
                rollback transaction
                return
            end
        commit transaction /* hoàn tất giao tác*/
/*end stored proc*/
```

2.3 Lịch giao tác

2.3.1 Các cách thực hiện của các giao tác

- **Thực hiện tuần tự:** Các thao tác khi thực hiện mà **không giao nhau về mặt thời gian.**
  - **Ưu:** Nếu thao tác **đúng đắn** thì **luôn luôn đảm bảo nhất quán dữ liệu.**
  - **Khuyết: Không tối ưu** về việc sử dụng tài nguyên và tốc độ.
- **Thực hiện đồng thời:** ***Các lệnh của các giao tác khác nhau*** **xen kẽ nhau trên trục thời gian.**
  - **Khuyết:** Gây ra nhiều phức tạp về nhất quán dữ liệu
  - **Ưu:**
    - **Tận dụng tài nguyên và thông lượng (throughput).** Ví dụ: Trong khi một giao tác đang thực hiện việc đọc / ghi trên đĩa, một giao tác khác đang xử lý tính toán trên CPU.
    - **Giảm thời gian chờ.** Ví dụ: Chia sẽ chu kỳ CPU và truy cập đĩa để làm giảm sự trì hoãn trong các giao tác thực thi.

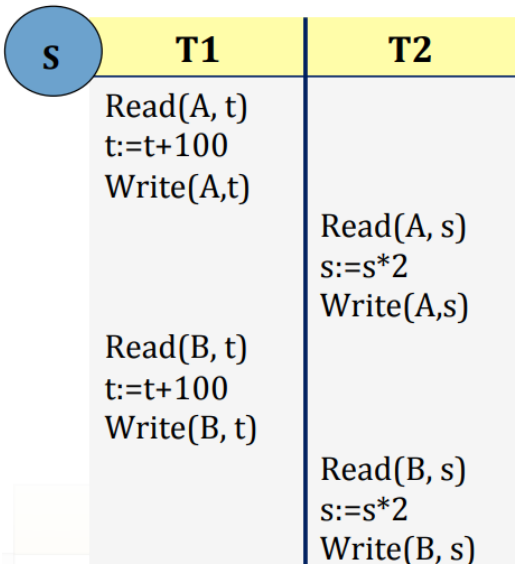
2.3.2 Lịch thao tác là gì?

- **Đặt vấn đề:** *Để xử lý một vấn đề gồm có nhiều hành động cần thực hiện, các hành động cần được thực hiện đồng thời, vì thế DBMS cần một cơ chế để quản lý – lập lịch (schedule).*
- **Định nghĩa: Một lịch** thao tác S được *lập từ n giao tác* $T_1, T_2, \ldots, T_n$ được **xử lý đồng thời** là *một **thứ tự** thực hiện **xen kẽ các hành động của n giao tác*** này.
- *Thứ tự xuất hiện của các thao tác trong lịch phải giống với thứ tự xuất hiện của chúng trong giao tác.*
- **Bộ lập lịch (Scheduler):** Là một thành phần của DBMS có nhiệm vụ *lập một lịch để thực hiện n giao tác xử lý đồng thời.*
- **Các loại lịch thao tác:**
  - **Lịch tuần tự (Serial)**
  - **Lịch khả tuần tự (Serializable):** Conflict – Serializability ; View – Serializability

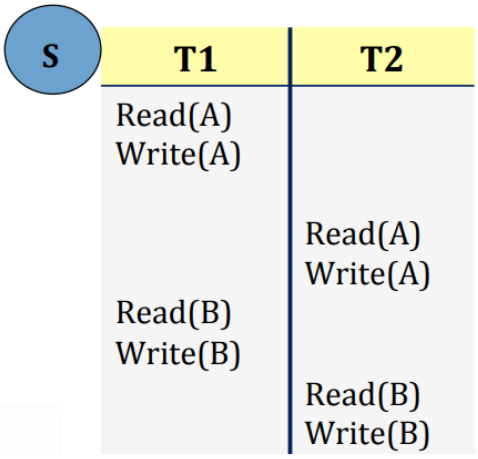
Feature	Serial Schedule	Serializable Schedule
Execution	Transactions execute one after the other.	Transactions can execute **concurrently**.
Concurrency	No concurrency *(đồng thời).*	**Allows** concurrency.
Performance	Lower performance due to no concurrency.	**Higher** performance due to concurrency.
Complexity	Simpler to implement.	**More complex** to implement due to concurrency control.
Final Outcome	Depends on the order of execution.	**Equivalent** to some **serial execution.**
Data Consistency	Guarantees *(bảo đảm)* data consistency.	Guarantees data consistency.

2.3.2.1 Biểu diễn lịch thao tác

Cách 1:



Cách 2: (chỉ quan tâm đọc/ ghi)

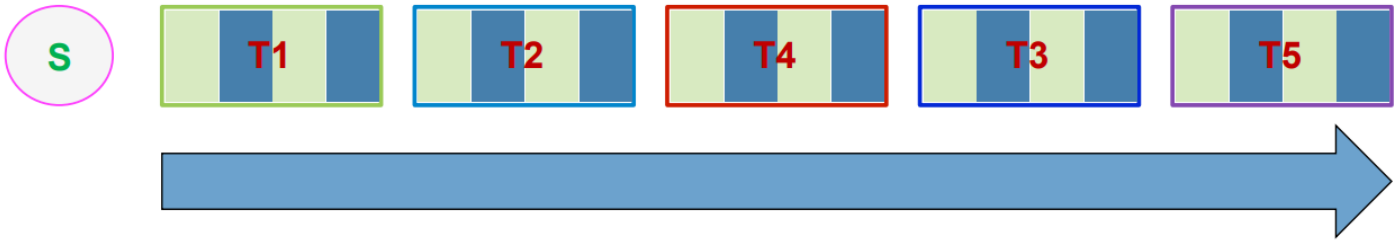


Cách 3: S: R1(A); W1(A); R2(A); W2(A); R1(B); W1(B); R2(B); W2 (B)

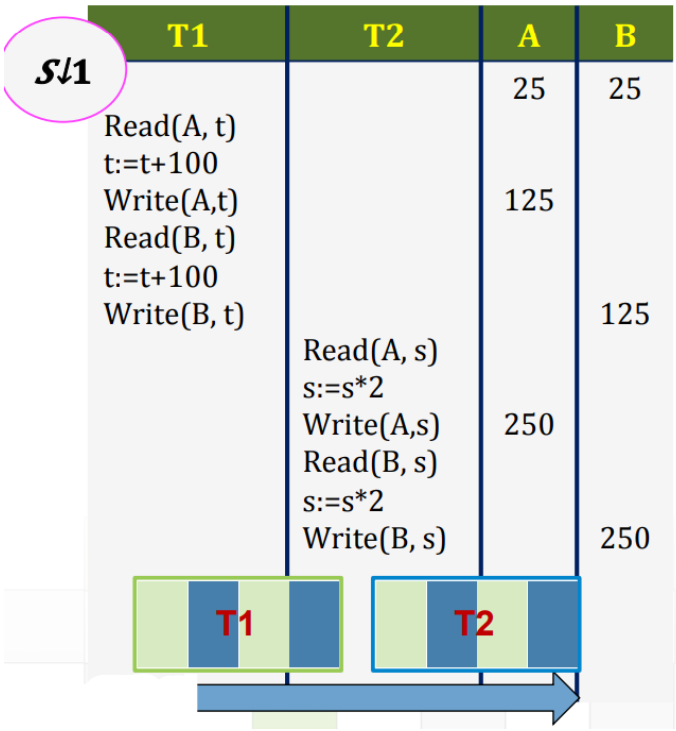
→ Thông thường người ta biểu diễn cách 3, ta cần chuyển về cách 2 cho dễ quan sát.

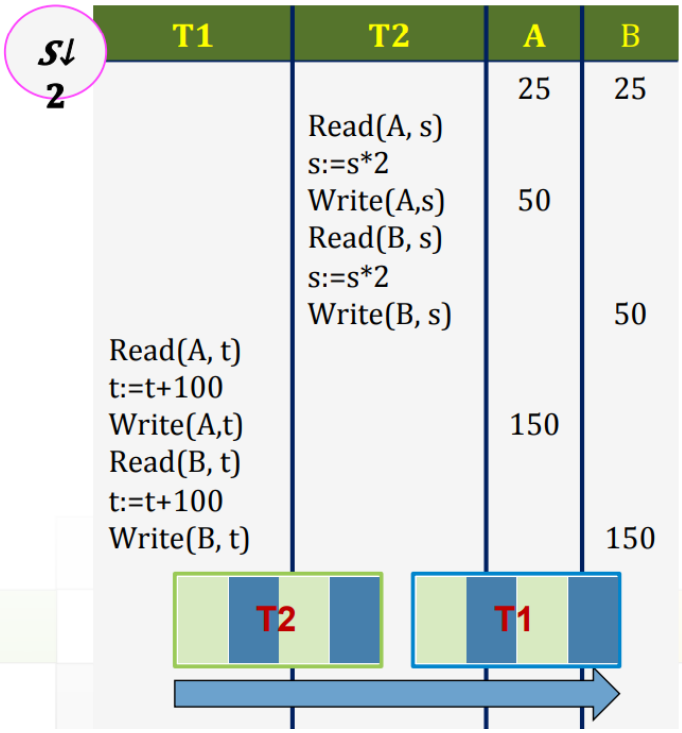
2.3.2.2 Lịch tuần tự (serial schedule)

- Một lịch S được gọi là **tuần tự** nếu **các hành động của các giao tác T_i được thực hiện liên tiếp nhau**, không có sự giao nhau về mặt thời gian.
- Lịch tuần tự **luôn luôn đảm bảo được tính nhất quán** của CSDL



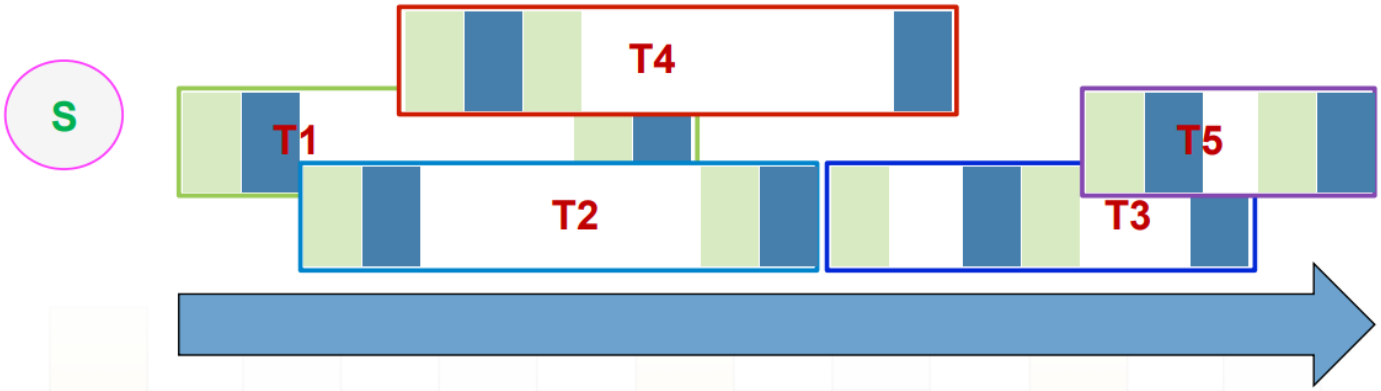
Ví dụ:





2.3.2.3 Lịch xử lý đồng thời (Non-serial Schedule) (Simultaneous Schedule)

- Lịch xử lý đồng thời là lịch mà **các giao tác** trong đó **giao nhau về mặt thời gian.**
- Luôn luôn tiềm ẩn khả năng làm cho CSDL **mất tính nhất quán**



Ví dụ:

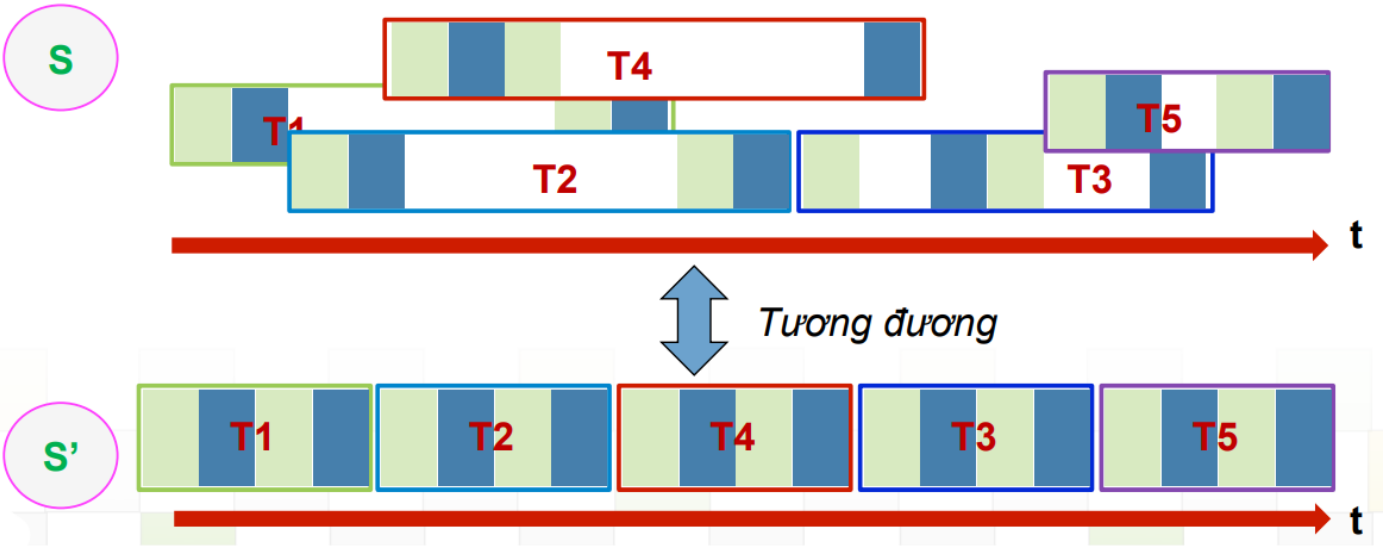
- S_3 là một lịch xử lý đồng thời vì **các giao tác giao thoa với nhau**
- Lịch xử lý đồng thời S_3 gây ra sự **mất nhất quán dữ liệu**
  - Trước S khi thực hiện: **A=B**
  - Sau khi S kết thúc: **A ≠ B**

S3

T1	T2	A	B
		25	25
Read(A, t)			
t:=t+100			
Write(A,t)		125	
	Read(A, s)		
	s:=s*2		
	Write(A,s)	250	
	Read(B, s)		
	s:=s*2		
	Write(B, s)		50
Read(B, t)			
t:=t+100			
Write(B, t)			150

2.3.2.4 Lịch khả tuần tự (Serializable Schedule)

- Một lịch S được lập ra từ n giao tác T_1, T_2, …, T_n ***xử lý đồng thời*** được gọi là lịch khả tuần tự **nếu** nó **cho cùng kết quả với một lịch tuần tự nào đó** được ***lập ra từ n giao tác này.***
- Lịch khả tuần tự cũng không gây nên tình trạng mất nhất quán dữ liệu



Ví dụ:

S_4

T1	T2	A	B
		25	25
Read(A, t)			
t:=t+100			
Write(A,t)		125	
	Read(A, s)		
	s:=s*2		
	Write(A,s)	250	
Read(B, t)			
t:=t+100			
Write(B, t)			125
	Read(B, s)		
	s:=s*2		
	Write(B, s)		250

- Trước S_4 khi thực hiện: **A = B = c, với c là hằng số** (trong ví dụ này c = 25).
- Sau khi S_4 kết thúc: **A=2*(c+100); B=2*(c+100)**
- Trạng thái CSDL nhất quán **(A = B).**
- **Kết quả của lịch S_4 <T1, T2> tương tự với kết quả của lịch tuần tự S_1<T1, T2>**
- Vậy S_4 là khả tuần tự.

S_5

T1	T2	A	B
		25	25
Read(A, t)			
t:=t+100			
Write(A,t)		125	
	Read(A, s)		
	s:=s*2		
	Write(A,s)	250	
	Read(B, s)		
	s:=s*2		
	Write(B, s)		50
Read(B, t)			
t:=t+100			
Write(B, t)			150

- Trước S_5 khi thực hiện: A = B = c, với c là hằng số
- Sau khi S_5 kết thúc: **A = 2*(c+100); B = 2*c + 100**
- **Trạng thái CSDL không nhất quán (A ≠ B)**
- S_5 **không khả tuần tự**

S_{5b}

T1	T2	A	B
		25	25
Read(A, t)			
t:=t+100			
Write(A,t)		125	
	Read(A, s)		
	s:=s*1		
	Write(A,s)	125	
	Read(B, s)		
	s:=s*1		
	Write(B, s)		25
Read(B, t)			
t:=t+100			
Write(B, t)			125

- Trước S_{5b} khi thực hiện: A = B = c, với c là hằng số
- Sau khi S_{5b} kết thúc: A = 1*(c+100); B = 1*c + 100
- **Trạng thái CSDL vẫn nhất quán**

- Để xem xét tính **mất nhất quán** một cách kỹ lưỡng, nếu **phải hiểu rõ ngữ nghĩa** của từng giao tác → **không khả thi**
- **Thực tế chỉ xem xét các lệnh của giao tác là đọc hay ghi**

Có 2 loại lịch khả tuần tự:

- **Conflict Serializable (Khả tuần tự xung đột):** Dựa trên ý tưởng **hoán vị các hành động không xung đột** để **chuyển một lịch đồng thời S về một lịch tuần tự S'**. Nếu có **một cách biến đổi như vậy** thì **S là một lịch conflict serializable.**

- **View Serializable (Khả tuần tự view):** Dựa trên ý tưởng **lịch đồng thời S và lịch tuần tự S' đọc và ghi** những **giá trị dữ liệu giống nhau.** Nếu có **một lịch S' như vậy** thì **S là một lịch view serializable.** ***(S' có sẵn, và so sánh với S).***

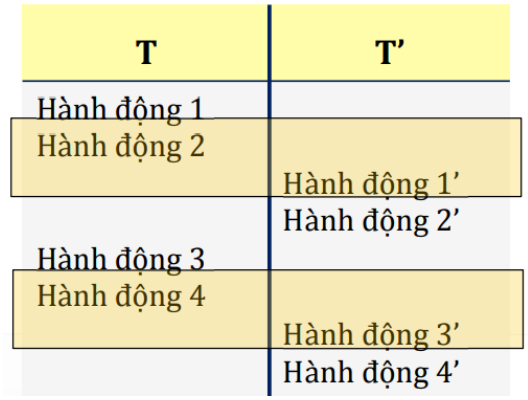
Ta có: Conflict Serializable ⊂ View Serializable ⊂ Serializable

2.3.2.4.1 Conflict Serializability

Ý tưởng: Xét **2 hành động** liên tiếp nhau của **2 giao tác khác nhau** trong một lịch thao tác, ***khi 2 hành động đó được đảo thứ tự*** có thể dẫn đến 1 trong 2 hệ quả:

– **Hoạt động của cả hai giao tác** chứa 2 hành động ấy **không bị ảnh hưởng** gì → **2 hành động đó không xung đột với nhau.**

– **Hoạt động của ít nhất một trong 2 giao tác** chứa 2 hành động ấy **bị ảnh hưởng** → **2 hành động xung đột.**



Cho lịch S có **2 giao tác T_i và T_j,** xét các trường hợp:

– $r_i(X)$; $r_j(Y)$

(Đọc trên đơn vị dữ liệu X của giao tác thứ i, đọc trên đơn vị dữ liệu Y của giao tác thứ j)

- **Không bao giờ có xung đột, ngay cả khi X = Y**
- Cả 2 thao tác *không làm thay đổi giá trị của X và Y*

– $r_i(X)$; $w_j(Y)$ **$w_i(X)$; $r_j(Y)$** **$w_i(X)$; $w_j(Y)$**

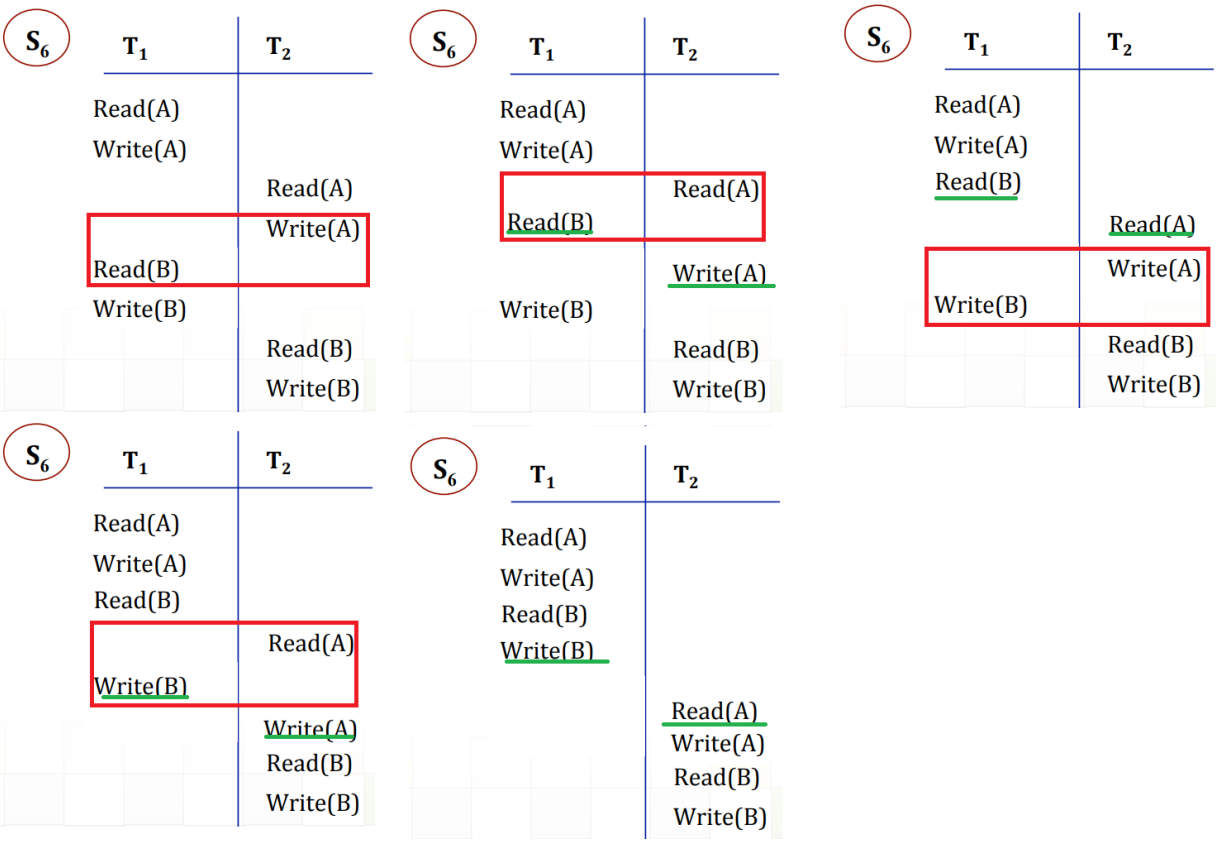
- **Không xung đột khi X ≠ Y**
  - – T_j không thay đổi dữ liệu đọc của T_i
  - – T_i không sử dụng dữ liệu ghi của T_j
- **Xung đột khi X = Y**

→ Tóm lại, hai hành động xung đột nếu chúng:

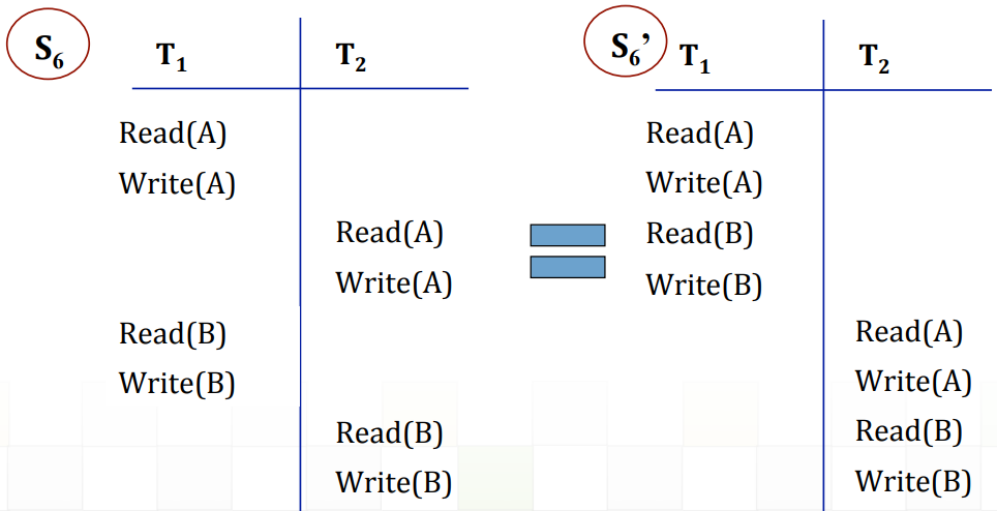
- Thuộc **2 giao tác khác nhau**
- Truy xuất đến **1 đơn vị dữ liệu**
- Trong chúng có **ít nhất một hành động ghi** (write)

→ Hai hành động **xung đột** thì **không thể nào đảo thứ tự của chúng** trong một lịch thao tác.

Ví dụ:



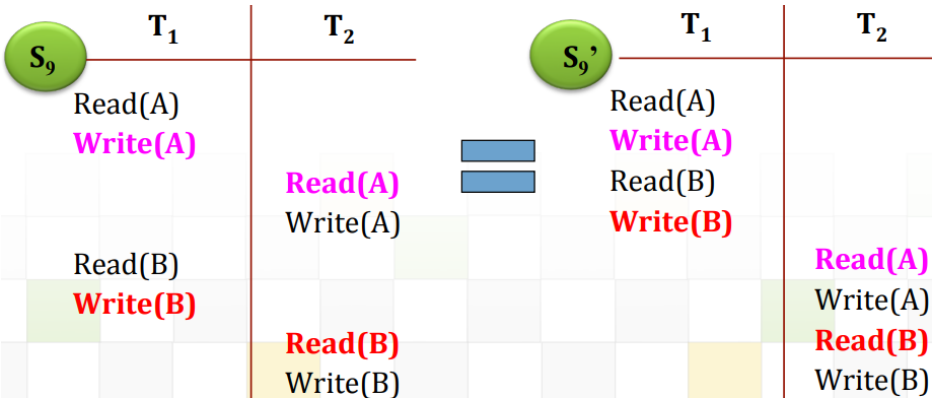
Vậy:



– **S và S'** là những lịch thao tác **conflict equivalent** (tương đương xung đột) **nếu S có thể chuyển được thành S' thông qua một chuỗi các hoán vị những thao tác không xung đột.**
– Một lịch thao tác **S là conflict serializable** nếu **S là conflict equivalent** với một lịch thao tác **tuần tự S'** nào đó.
– S là **conflict serializable** thì **S khả tuần tự.**
– S là khả tuần tự thì không chắc S conflict serializable.

2.3.2.4.2 Kiểm tra Conflict Serializability

Cho lịch S_9: S_9 có conflict serializability hay không?



Ý tưởng: Các hành động **xung đột** trong lịch S được thực hiện theo thứ tự nào thì **các giao tác thực hiện chúng trong S'** (kết quả sau hoán vị) **cũng theo thứ tự đó**.

Cho lịch S có **2 giao tác T1 và T2:**
– T1 thực hiện hành động A1
– T2 thực hiện hành động A2
– Ta nói **T1 thực hiện trước hành động T2** trong S, ký hiệu **T1 <s T2**, **khi:**
• **A1 được thực hiện trước A2** trong S, **A1 không nhất thiết phải liên tiếp A2**
• **A1 và A2 là 2 hành động xung đột** *(A1 và A2 **cùng thao tác lên 1 đơn vị dữ liệu** và có **ít nhất 1 hành động là ghi** trong A1 và A2)*

Phương pháp Precedence Graph (Đồ thị ưu tiên):

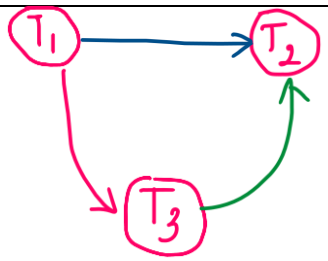
- Cho lịch S bao gồm các thao tác $T_1, T_2, \ldots, T_n$
- Đồ thị trình tự của S **(Precedence graph)** của S ký hiệu là **P(S)** có:
  – **Đỉnh** là các giao tác T_i **(vẽ trước đầy đủ các đỉnh)**
  – **Cung** đi từ T_i đến T_j nếu $T_i <_S T_j$

Nếu có nhiều hơn 1 cặp xung đột (nhiều hơn 1 cung giữa 2 đỉnh), ta chỉ vẽ một cung để đỡ rối.

- S Conflict Serializable khi và chỉ khi P(S) không có chu trình, điều này có nghĩa là chúng ta có thể xây dựng một lịch trình tuần tự S' tương đương với lịch xung đột S. Lịch tuần tự S' có thể được tìm thấy **bằng cách sắp xếp tô pô của đồ thị P(S). Lịch trình như vậy có thể nhiều hơn 1.**
- Với 2 lịch S và S' được lập từ cùng các giao tác, **S và S' conflict equivalent khi và chỉ khi P(S) = P(S')**
- Thứ tự hình học các đỉnh là **thứ tự của các giao tác** trong lịch tuần tự tương đương với S.

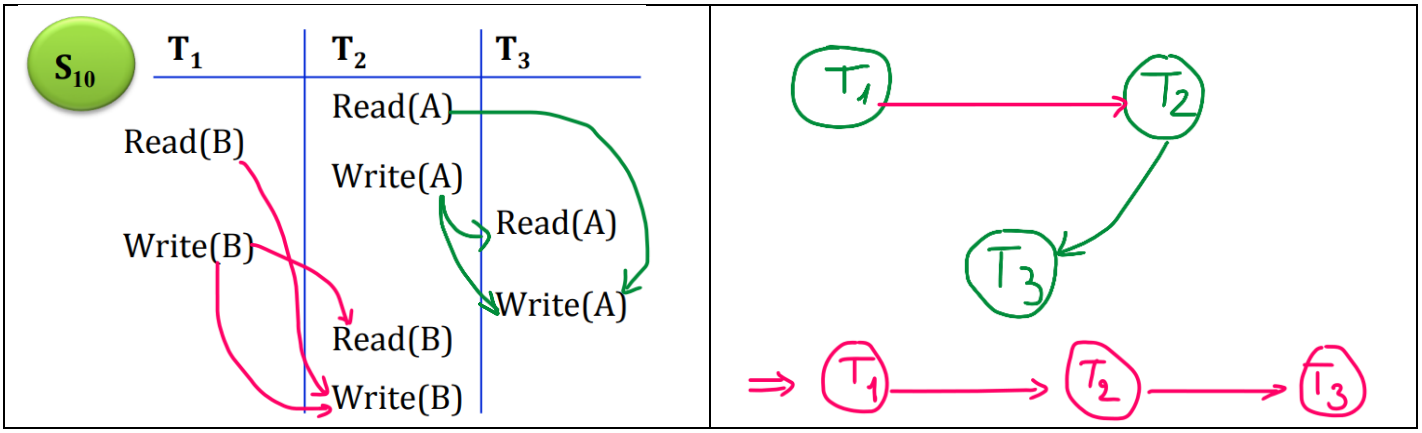
Ví dụ 1: S_1: r1(x);r3(y);w1(x);w2(y);r3(x);w2(x)

T1	T2	T3
R(X)		
		R(Y)
W(X)		
	W(Y)	
		R(X)
	W(X)	

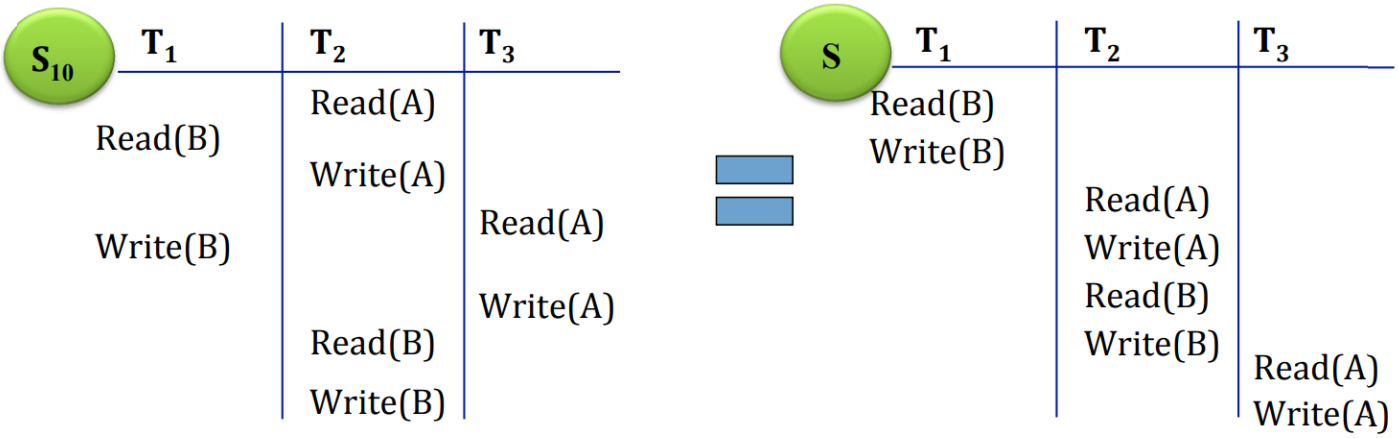


Lịch tuần tự là 1→ 3 → 2

Ví dụ 2: S_{10} có Conflict Serializability hay không?

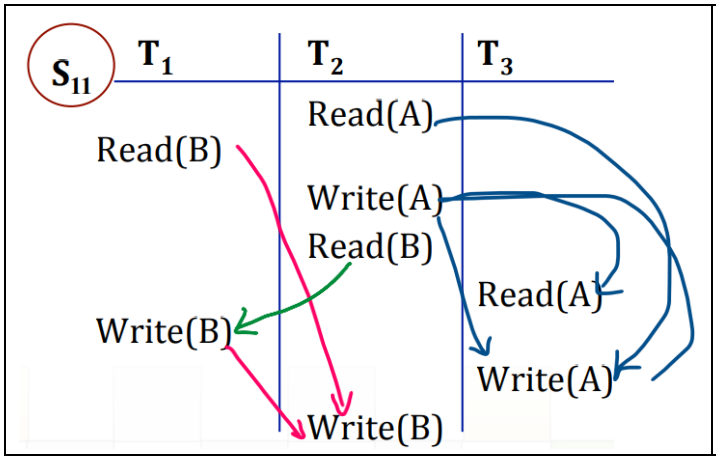


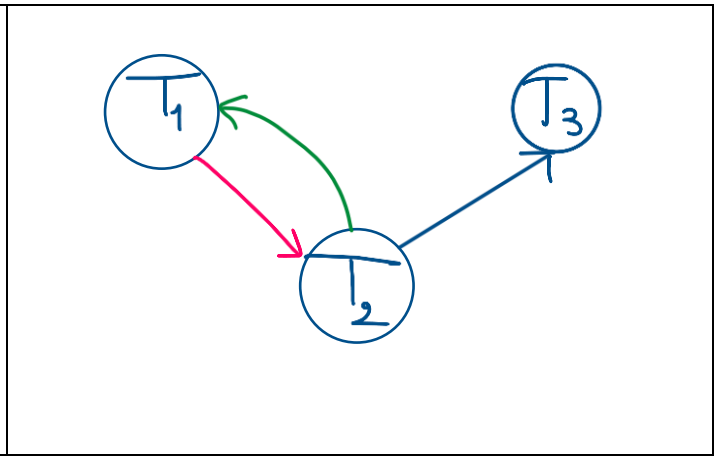
P(S10) không có chu trình → S10 conflict-serializable theo thứ tự T1 → T2 → T3.



Ví dụ 3: S_{11} có Conflict Serializability hay không?

S_{11}: r2(A) r1(B) w2(A) r2(B) r3(A) w1(B) w3(A) w2(B)

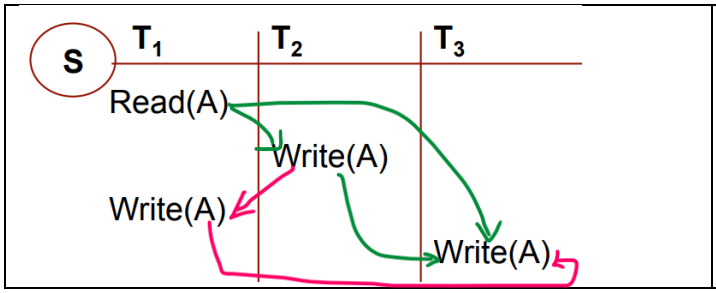


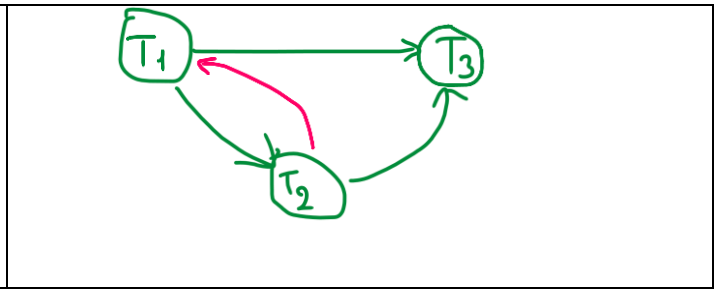


P(S_{11}) **có chu trình → S_{11} không conflict-serializable.**

2.3.2.4.3 View Serializability

Xét lịch S

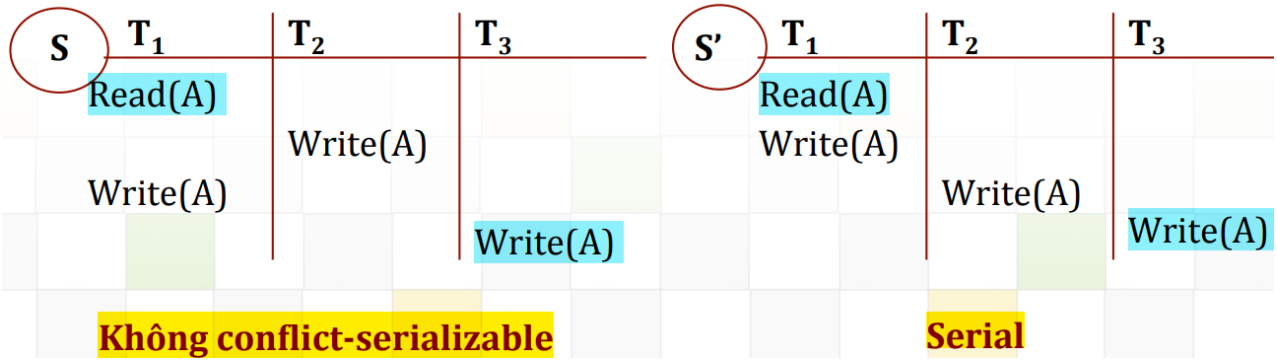




P(S) có chu trình → **S không conflict-serializable.**

Vậy S khả tuần tự hay không?

So sánh lịch S và 1 lịch tuần tự S'



– Giả sử **trước khi lịch S thực hiện**, có **giao tác Tb thực hiện việc ghi A** và **sau khi S thực hiện có giao tác Tf thực hiện việc đọc A.** *Ta cũng giả sử các lần ghi A thì A không có thao tác cập nhật giá trị.*

– Nhận xét lịch S và S':

- **Đều có T1 thực hiện read(A)** từ giao tác Tb → **Kết quả đọc luôn giống nhau**
- **Đều có T3 thực hiện việc ghi cuối cùng lên A.** T2, T3 không có lệnh đọc A

→ Dù S hay S' được thực hiện thì kết quả đọc A của Tf luôn giống nhau

→ Kết quả của S và S' giống nhau → **S vẫn khả tuần tự.**

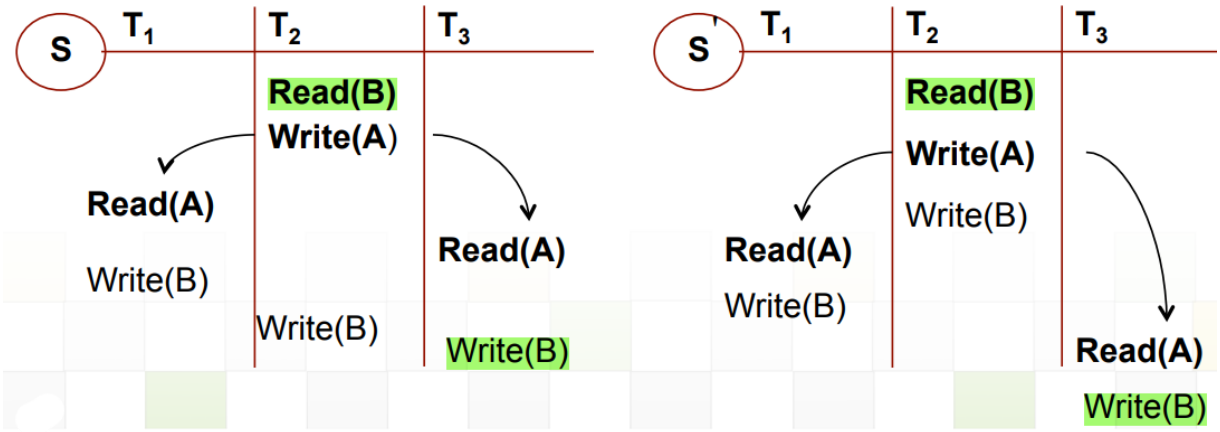
Khả tuần tự View (View-serializability)

– Một lịch S được gọi là khả tuần tự view nếu **tồn tại một lịch tuần tự S'** được tạo từ các giao tác của S sao cho **S và S' đọc và ghi những giá trị giống nhau.**

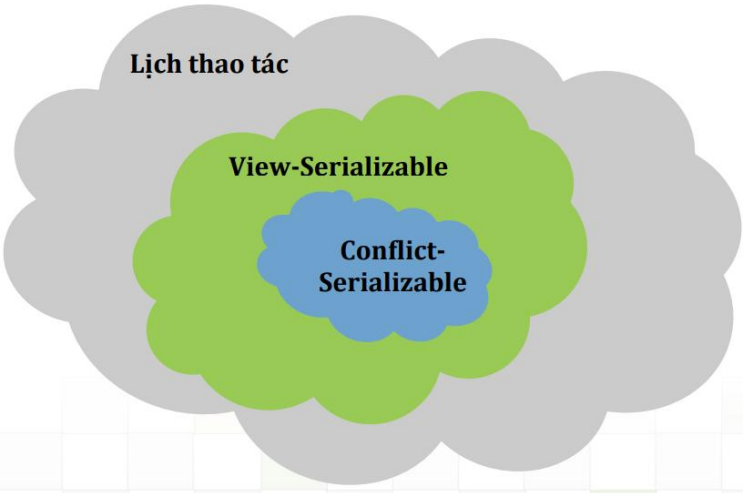
Ví dụ: Cho lịch S và S' như sau:

S : r2(B) w2(A) r1(A) r3(A) w1(B) w2(B) w3(B)

S': r2(B) w2(A) w2(B) r1(A) w1(B) r3(A) w3(B)



→ S khả tuần tự view.



View-equivalent: S và S' là những lịch view-equivalent nếu **thỏa các điều kiện sau:**

- ❖ Nếu trong **S** có **Ti đọc giá trị ban đầu của A** thì **nó cũng đọc giá trị ban đầu của A** trong **S'.**
- ❖ Nếu **Ti đọc giá trị của A được ghi bởi Tj trong S** thì **Ti cũng phải đọc giá trị của A được ghi bởi Tj trong S'.**
- ❖ Với mỗi dvdl A, **giao tác thực hiện lệnh ghi cuối cùng lên A (nếu có) trong S thì giao tác đó cũng phải thực hiện lệnh ghi cuối cùng lên A trong S'**.

– Lịch S được gọi là khả tuần tự view khi và chỉ khi nó **tương đương view (view-equivalent)** với một lịch tuần tự S'.

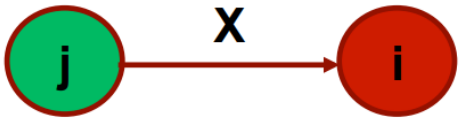
- Nếu S là conflict-serializable → S view-serializable. Không có chiều ngược lại.

2.3.2.4.4 Kiểm tra View Serializability

Phương pháp 1: Đồ thi phức (PolyGraph)

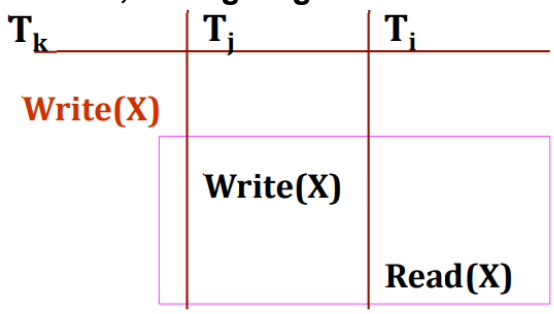
- Cho 1 Lịch giao tác S
- **Thêm 1 giao tác cuối Tf** vào trong S sao cho Tf thực hiện việc **đọc hết tất cả đơn vị dữ liệu** ở trong S S = … w1(A)……w2(A) **rf(A)**
- **Thêm 1 giao tác đầu tiên Tb** vào trong S sao cho Tb thực hiện việc **ghi các giá trị ban đầu cho tất cả đơn vị dữ liệu** S = **wb(A)**… w1(A)……w2(A)…

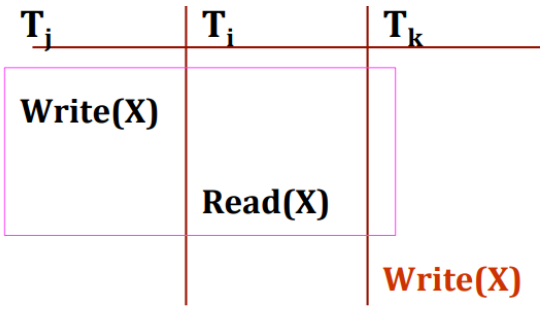
- Vẽ đồ thị phức (PolyGraph) cho S, ký hiệu G(S) với
  – Đỉnh là các giao tác Ti (**bao gồm cả Tb và Tf**)
  – Cung:
  • (1) Nếu giá trị mà ri(X) đọc được là do Tj ghi **(ri(X) có gốc là wj(X))** thì **vẽ cung đi từ Tj đến Ti.** *(mỗi thao tác đọc đơn vị dữ liệu X ứng với thao tác ghi X gần nhất với thao tác đọc đang xét).*

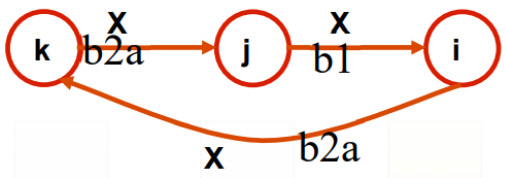


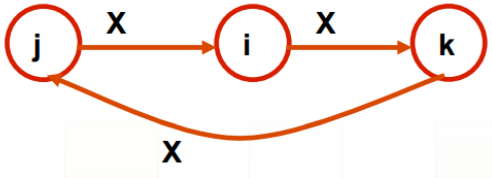
• (2) Với mỗi **wj(X) … ri(X),** xét **wk(X) khác Tb** sao cho **Tk không chèn vào giữa Tj và Ti.**

• **(2a)** Nếu **Tj ≠ Tb và Ti ≠ Tf** thì vẽ cung Tk → Tj và Ti → Tk **(1 cung từ k đến giao tác chứa write, 1 cung từ giao tác chứa read đến k).**

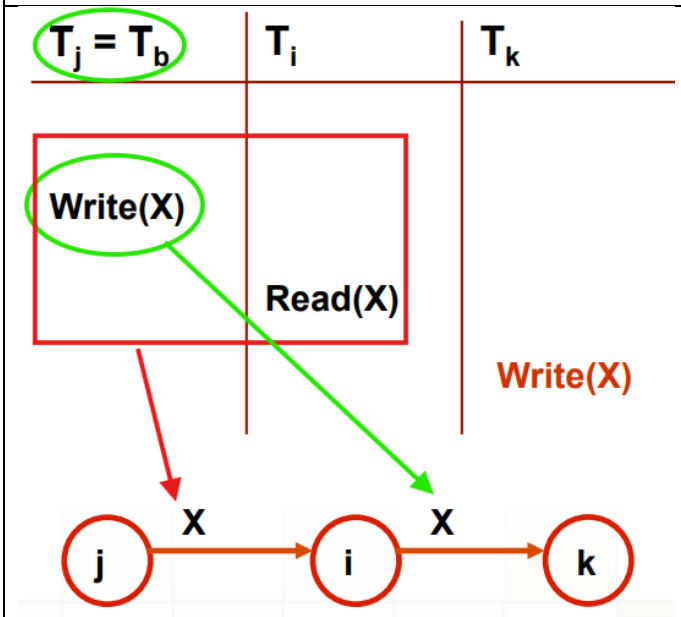
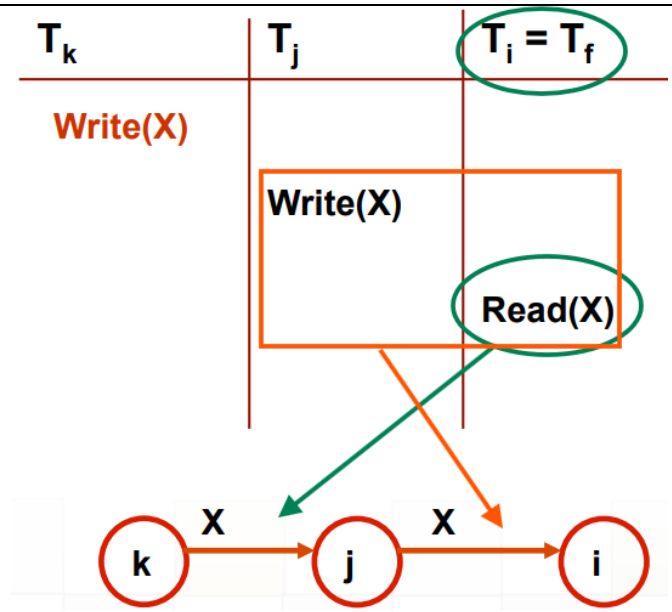




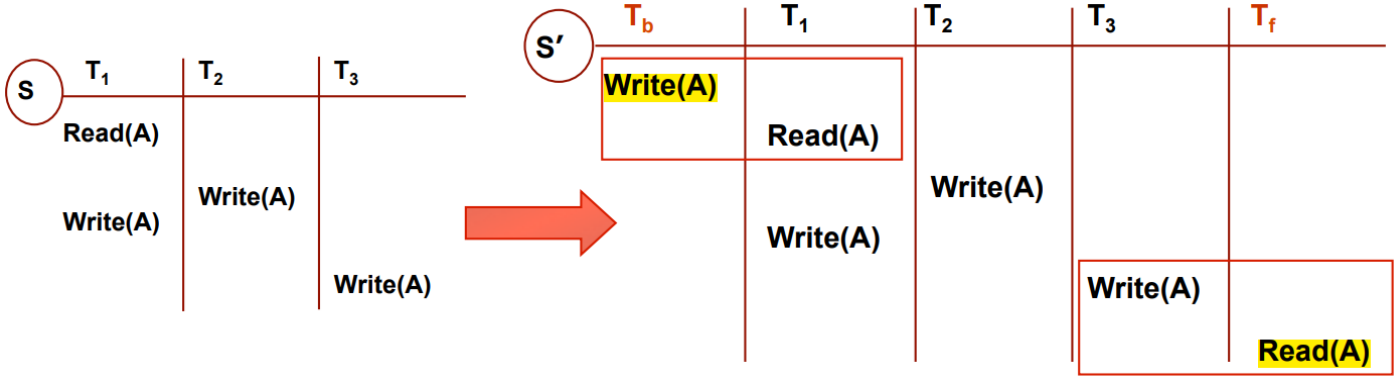


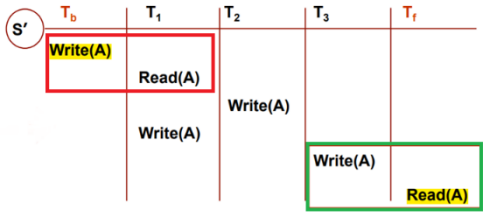


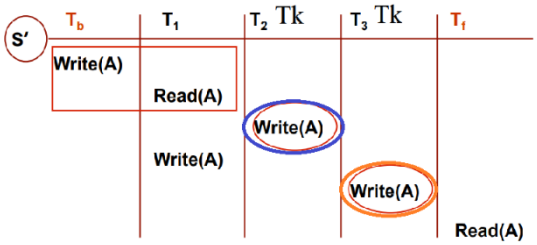
Tk có thể nằm <u>trước Tj</u> hoặc <u>sau Ti.</u>

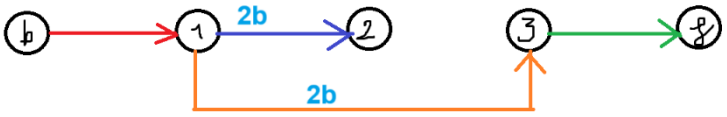
• **(2b)** Nếu **Tj = Tb** (trùng với giao tác bắt đầu là hành động write) thì vẽ cung **Ti → Tk (vẽ cung từ giao tác chứa read đến k).** *(Do bắt buộc không có cung nào trước Tb)*	• **(2c)** Nếu **Ti = Tf** (trùng với giao tác kết thúc là giao tác đọc) thì vẽ cung **Tk → Tj (vẽ cung từ k đến giao tác chứa write).** *(Do bắt buộc không có cung nào sau Tf)*
	

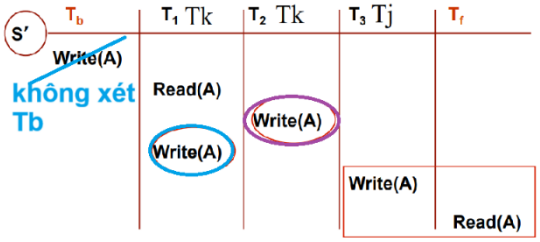
Ví dụ 1:

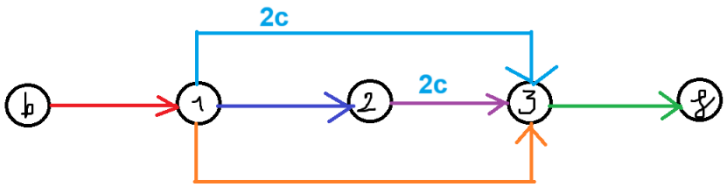


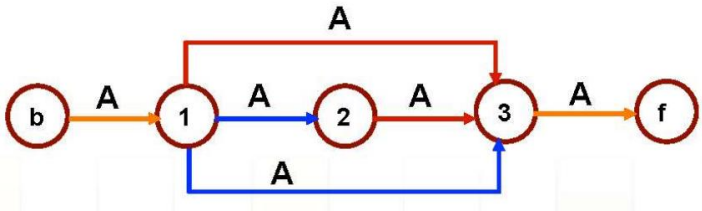








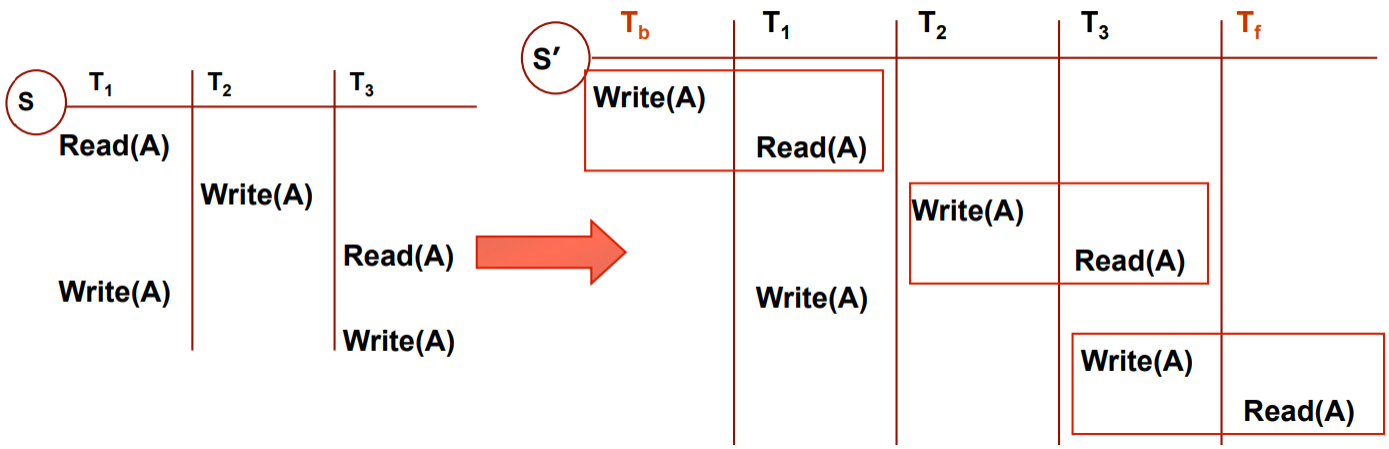


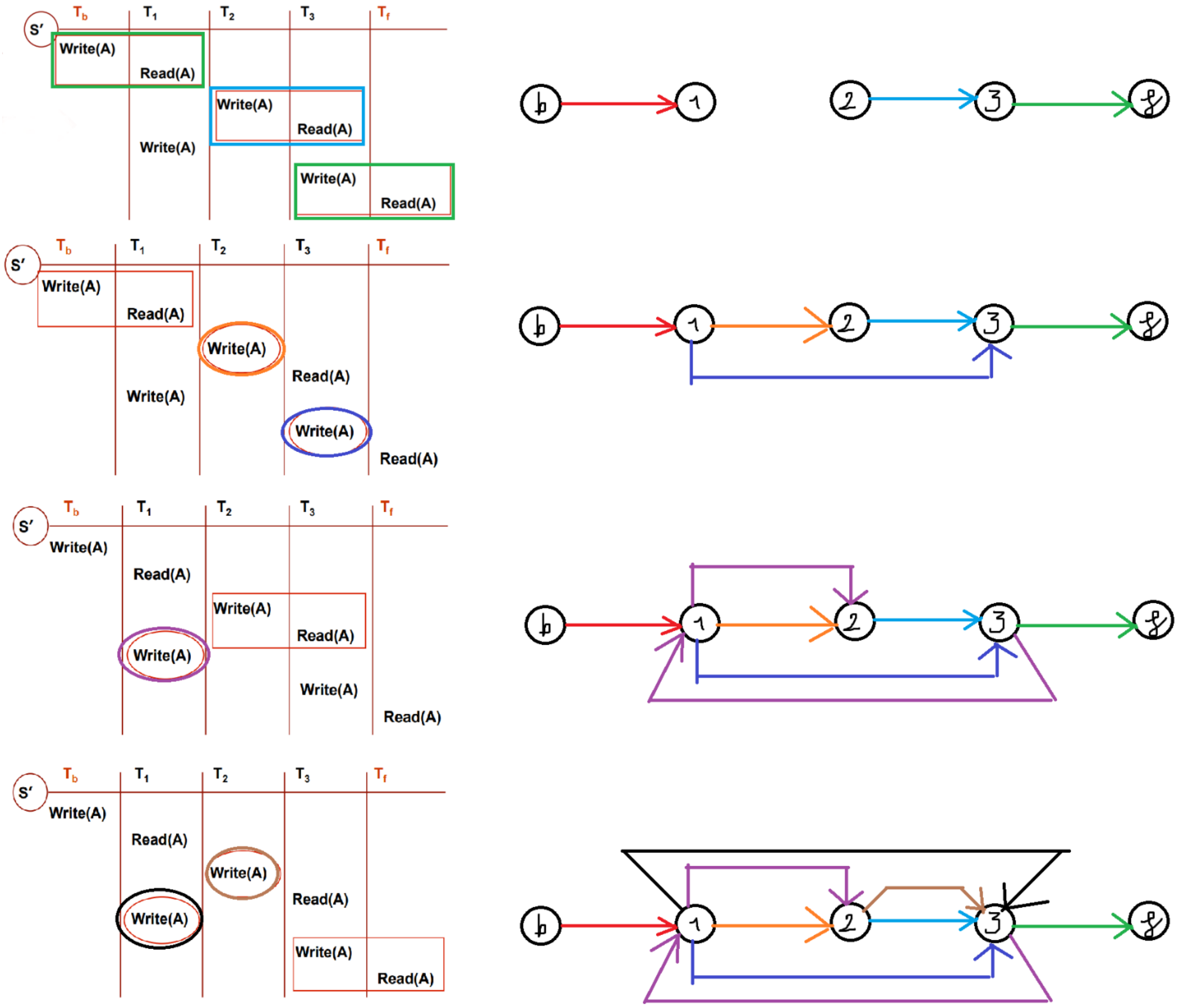


G(S) không có chu trình

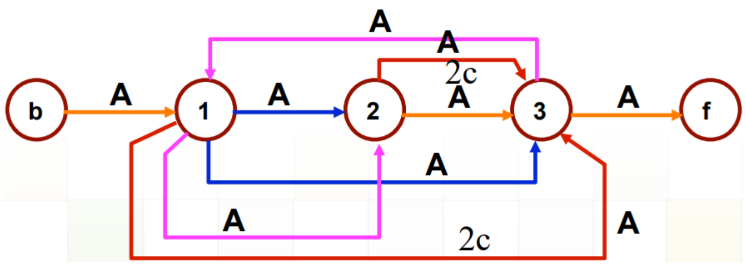
→ S view-serializable theo thứ tự T_b, T_1, T_2, T_3, T_f

Ví dụ 2:



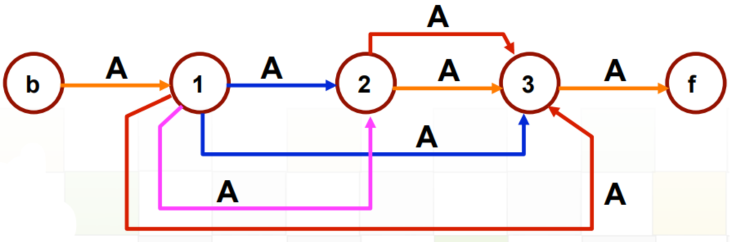


Vẽ lại:



G(S) có chu trình.

G(S) không có chu trình sau khi bỏ cung → S view-serializable



Phương pháp 2: Đồ thị phụ thuộc (dependency graph)

There's another method to check the view-serializability of a schedule.

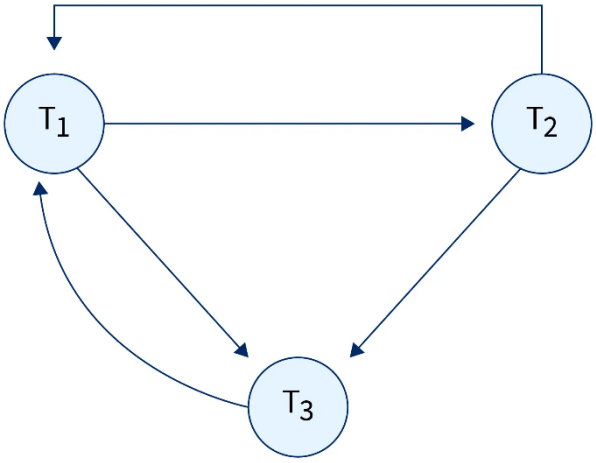
The first step of this method is the same as the previous method i.e. **checking the conflict serializability** of the schedule **by creating the precedence graph.**

T1	T2	T3
R(X)		
	W(X)	
		R(X)
W(X)		
		W(X)

Writing all the conflicting operations:

- R1(X), W2(X) (T1 → T2)
- R1(X), W3(X) (T1 → T3)
- W2(X), R3(X) (T2 → T3)
- W2(X), W1(X) (T2 → T1)
- W2(X), W3(X) (T2 → T3)
- R3(X), W1(X) (T3 → T1)
- W1(X), W3(X) (T1 → T3)

The precedence graph for the above schedule is as follows:



If the precedence graph doesn't contain a loop/cycle, then it is conflict serializable, and thus concludes that the given schedule is consistent.

As the above graph **contains a loop/cycle,** it **does not conflict with serializable.** Thus we need to perform the following steps.

Next, we will check for **blind writes.** If the blind writes don't exist, then the schedule is non-view Serializable.

We can simply conclude that the given schedule is inconsistent.

If there exist any blind writes in the schedule, then it may or may not be view serializable.

*** Ghi mù (Blind writes):** If a **write action** is performed on a data item **by a Transaction** (updation), **without performing the reading operation** then it is known as **blind write.**

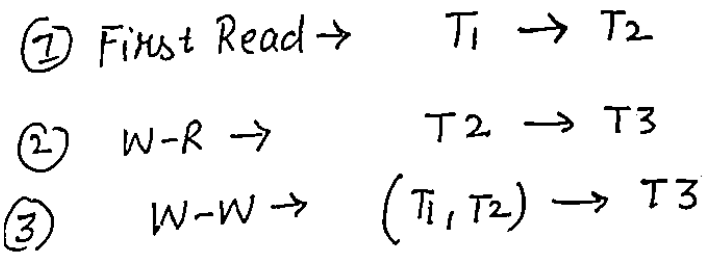
In the above example, **transaction T2 contains a blind write,** thus we are **not sure** if the given schedule **is view-serializable or not.**

Lastly, we will draw a **dependency graph (đồ thị phụ thuộc)** for the schedule.

Note: The dependency graph is different from the precedence graph.

→ In Dependence Graph method, we find the serial schedule corresponding to Non-conflict Serializable by checking for three conditions

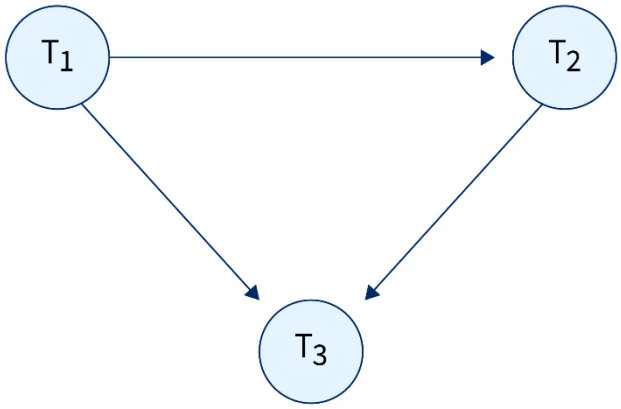
* First Read (R-W)
* First updated Read (W-R)
* Last Write (W-W)



Steps for dependency graph:

- Firstly, T1 reads X, and T2 first updates X. So, **T1 must execute before the T2 operation**. (T1 → T2)
- **T2 updates X before T3**, so we get the dependency as (T2 → T3)
- **T3 performs the final updation** on X thus, **T3 must execute after T1 and T2.** (T1 → T3 and T2 → T3)

The dependency graph is formed as follows:



As **no cycles** exist in the dependency graph for the example, we can say that **the schedule is view serializable.**

If any cycle/ loop exists, then it is not view-serializable and the schedule is inconsistent.

If cycle/loop doesn't exist, then it is view-serializable.

PHỤ LỤC: Khả tuần tự View (View Serializability)

Mục tiêu thực sự của chúng ta trong thiết kế một trình lập lịch là *chỉ cho phép các lịch có thể tuần tự hóa.* Chúng ta cũng thấy cách các khác biệt trong các giao dịch hoạt động áp dụng cho dữ liệu có thể ảnh hưởng đến việc một lịch nhất định có thể tuần tự hóa hay không. Các trình lập lịch thường áp dụng lịch **khả tuần tự xung đột** (conflict serializability), đảm bảo khả năng tuần tự hóa bất kể các giao dịch làm gì với dữ liệu của chúng.

Tuy nhiên, có những điều kiện yếu hơn conflict-serializability cũng đảm bảo khả năng tuần tự hóa, được gọi là **khả tuần tự View (View-serializability).** View-serializability xem xét tất cả các kết nối giữa các giao tác T và U sao cho T ghi (write) một phần tử cơ sở dữ liệu *(đơn vị dữ liệu)* có giá trị U đọc được.

Sự khác biệt chính giữa View-serializability và conflict serializability xuất hiện khi *một giao tác T **ghi** một giá trị A mà không có giao tác nào khác đọc được* (vì một số giao tác khác sau đó ghi giá trị của riêng nó cho A) *(~ ghi mù – blind write).* Trong trường hợp đó, hành động $w_T(A)$ có thể được đặt ở một số vị trí khác của lịch (nơi A cũng không bao giờ được đọc) mà sẽ không được phép theo định nghĩa về khả năng tuần tự hóa xung đột.

Trong nội dung này, chúng ta sẽ định nghĩa khả tuần tự View một cách đúng nhất và kiểm tra nó.

1. Tương đương View (View Equivalence)

Giả sử chúng ta có hai lịch S_1 và S_2 của cùng một tập hợp các giao dịch.

Có một giao dịch giả định T_0 (T_b) đã *ghi các giá trị khởi tạo cho mỗi phần tử* cơ sở dữ liệu *(đơn vị dữ liệu)* được *đọc bởi bất kỳ giao dịch nào* trong các lịch và

Một giao dịch giả định khác T_f *đọc mọi phần tử được ghi bởi một hoặc nhiều giao dịch* sau khi mỗi lịch kết thúc.

Sau đó, đối với mọi hành động đọc $r_i(A)$ trong một trong các lịch trình, chúng ta có thể tìm thấy hành động ghi $w_j(A)$ **gần nhất trước** hành động đọc đang đề cập. Ta nói **T_j là nguồn của hành động đọc $r_i(A)$.**

Lưu ý rằng giao dịch T_j có thể là giao dịch giả định ban đầu T_0 và Ti có thể là T_f.

Nếu đối với mọi hành động đọc trong một trong các lịch trình, nguồn của nó giống nhau trong lịch trình kia, chúng ta nói rằng **S_1 và S_2 tương đương View.** Chắc chắn, các lịch tương đương View thực sự tương đương; mỗi lịch trình đều thực hiện giống nhau khi được thực hiện trên bất kỳ trạng thái cơ sở dữ liệu nào.

Nếu một lịch S tương đương View với một lịch tuần tự, chúng ta nói S là khả tuần tự View.

Ví dụ 1. Giả sử lịch S được định nghĩa như sau:

T_1	T_2	T_3
	r(B)	
	w(A)	
r(A)		
		r(A)
w(B)		
	w(B)	
		w(B)

Hoặc trình bày như dưới đây:

$$\begin{array}{lccccccc} T_1: & & & r_1(A) & & w_1(B) & & \\ T_2: & r_2(B) & w_2(A) & & & & w_2(B) & \\ T_3: & & & & r_3(A) & & & w_3(B) \end{array}$$

Ta tách các hành động của từng giao dịch theo chiều dọc, để *chỉ ra rõ hơn giao dịch nào thực hiện gì*; bạn nên đọc lịch trình từ trái sang phải.

Trong S, cả T_1 và T_2 đều ghi các giá trị của B bị mất; chỉ giá trị của B do T_3 ghi mới tồn tại đến cuối lịch trình và được đọc bởi giao dịch giả định T_f.

S không thể khả tuần tự xung đột. Để xem lý do, trước tiên hãy lưu ý rằng

T_2 ghi A trước khi T_1 đọc A, do đó **T_2 phải đứng trước T_1** trong một lịch tuần tự tương đương xung đột giả định.

Thực tế là hành động $w_1(B)$ đứng trước $w_2(B)$ cũng buộc **T_1 phải đứng trước T_2** trong bất kỳ lịch tuần tự tương đương xung đột nào.

(Hai điều kiện trên mâu thuẫn với nhau, vì không thể có thứ tự tuần tự tương đương vừa thỏa mãn T_2 đứng trước T_1 vừa T_1 đứng trước T_2. Do đó, S không khả tuần tự xung đột).

Tuy nhiên, *cả $w_1(B)$ và $w_2(B)$ đều không có tác động dài hạn nào đến cơ sở dữ liệu.* **Những loại ghi không liên quan này khả tuần tự View có thể bỏ qua** khi xác định các ràng buộc thực sự trên một lịch trình tuần tự tương đương.

Chính xác hơn, chúng ta hãy **xem xét các nguồn của tất cả các lần đọc** trong S:

1. Nguồn của $r_2(B)$ là T_0, vì không có lần ghi trước nào của B trong S.
2. Nguồn của $r_1(A)$ là T_2, vì T_2 gần đây nhất đã ghi A trước khi đọc.
3. Tương tự, nguồn của $r_3(A)$ là T_2.
4. Nguồn của lần đọc giả định của A bởi T_f là T_2.
5. Nguồn của lần đọc giả định của B bởi T_f là T_3, giao tác thực hiện thao tác ghi B cuối cùng.

Tất nhiên, *T_0 xuất hiện trước tất cả các giao dịch thực trong bất kỳ lịch trình nào và T_f xuất hiện sau tất cả các giao dịch.*

Nếu chúng ta sắp xếp các giao dịch thực (T_2, T_1, T_3), thì nguồn của tất cả các lần đọc đều giống như trong lịch S. *(Giả sử có một lịch tuần tự S' tuân theo thứ tự thực hiện các giao*

dịch như trên, ta sẽ tìm xem lịch S' có tương đương với S không, nếu có, thì S và S' là tương đương View, dẫn đến S là lịch khả tuần tự).

Nghĩa là, T_2 đọc B và chắc chắn T_0 thực hiện thao tác ghi trước đó. T_1 đọc A, nhưng T_2 đã ghi A, do đó nguồn của $r_1(A)$ là T_2, như trong S.

T_3 cũng đọc A, nhưng vì T_2 trước đó đã ghi A, đó là nguồn của $r_3(A)$, như trong S.

Cuối cùng, T_f giả định đọc A và B, nhưng giao tác thực hiện thao tác ghi cuối cùng của A và B trong lịch (T_2; T_1; T_3) lần lượt là T_2 và T_3, cũng như trong S.

Ta kết luận rằng S là một lịch **khả tuần tự View** và lịch được biểu diễn bằng thứ tự (T_2; T_1; T_3) là một **lịch tương đương View.** □

2. Đồ thị phức cho lịch khả tuần tự View (Polygraphs for View-Serializability)

Có một phương pháp khái quát của **đồ thị thứ tự ưu tiên (precedence graph)** *(phương pháp mà ta đã sử dụng để kiểm tra khả tuần tự xung đột)* phản ánh tất cả các ràng buộc thứ tự ưu tiên được yêu cầu theo định nghĩa về khả tuần tự View. Ta định nghĩa **đồ thị phức (polygraph)** cho một lịch bao gồm những điều sau:

1. **Một nút (node) cho mỗi giao tác** và các nút bổ sung **cho các giao dịch giả định T_0 (T_b) và T_f.**

2. Đối với **mỗi hành động $r_i(X)$ với nguồn T_j, vẽ một cung từ T_j đến T_i.**

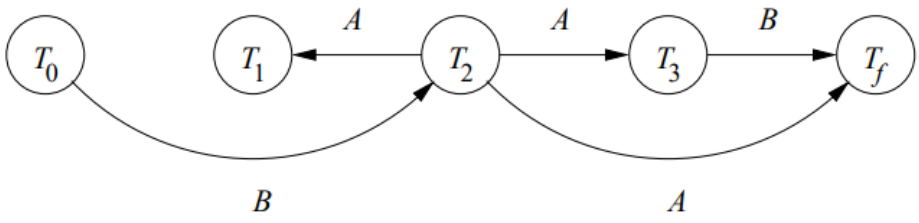
3. Giả sử **T_j là nguồn của lệnh đọc $r_i(X)$,** và **T_k là một giao tác khác thực hiện thao tác ghi của X. T_k không được phép chen** *(can thiệp – intervene)* **giữa T_j và T_i,** vì vậy nó phải xuất hiện trước T_j hoặc sau T_i. Ta biểu diễn điều kiện này bằng **một cặp cung** (được vẽ bằng nét đứt) từ **T_k đến T_j và từ T_i đến T_k.**

Theo trực giác (Intuitively), **một trong hai cung là "thực",** nhưng ta không quan tâm là cung nào, và khi **ta cố gắng làm cho đồ thị phức trở thành phi chu trình (không có chu trình),** chúng ta có thể **chọn bất kỳ cặp nào giúp làm cho nó phi chu trình (loại bỏ một trong hai cạnh tạo chu trình).**

Tuy nhiên, có những **trường hợp đặc biệt** quan trọng mà cặp cung trở thành một cung duy nhất:

(a) Nếu **T_j là T_0 (T_b),** *thì T_k không thể xuất hiện trước T_j,* vì vậy chúng ta sử dụng cung Ti → Tk thay cho cặp cung.

(b) Nếu **T_i là T_f,** *thì T_k không thể theo sau T_i,* vì vậy chúng ta sử dụng cung Tk → Tj thay cho cặp cung.



Hình 1. *Bước đầu tiên của phương pháp đồ thị phức minh họa Ví dụ 1.*

Ví dụ 2. Xem xét lịch trình S từ **Ví dụ 1**. Hình 1 biểu diễn phần đầu của đồ thị phức cho S, trong đó chỉ có **các nút và các cung từ quy tắc (2) được đặt.** Đồng thời cũng đã chỉ ra phần tử cơ sở dữ liệu *(đơn vị dữ liệu)* lập nên mỗi cung.

Nghĩa là, A được truyền từ T_2 đến T_1, T_3 và T_f, trong khi B được truyền từ T_0 đến T_2 và từ T_3 đến T_f.

Bây giờ, ta phải xem xét những giao tác nào có thể can thiệp vào từng "kết nối" này bằng cách *ghi cùng một đơn vị dữ liệu (element)* giữa chúng. Những can thiệp tiềm ẩn này bị loại trừ bởi các cặp cung từ quy tắc (3), mặc dù như chúng ta sẽ thấy, trong ví dụ này, *mỗi cặp cung liên quan đến một trường hợp đặc biệt và trở thành một cung duy nhất.*

Hãy xem xét cung $\mathbf{T_2 \rightarrow T_1}$ dựa trên đơn vị dữ liệu A. *Các giao tác thực hiện duy nhất việc ghi A là T_0 và T_2,* và không giao tác nào trong số chúng có thể chen giữa cung này *(theo quy tắc 3)*, vì T_0 không thể di chuyển vị trí của nó và T_2 đã là điểm kết thúc của cung. Do đó, không cần thêm cung (đỉnh) nào.

Một lập luận tương tự cho chúng ta biết, không cần thêm bất kỳ cung (đỉnh) nào để đảm bảo rằng **những giao tác khác thực hiện việc ghi A nằm ngoài các cung $\mathbf{T_2 \rightarrow T_3}$ và $\mathbf{T_2 \rightarrow T_f}$.**

(T_3 đọc A, và T_2 là giao tác gần nhất ghi A trước đó. T_f, giao tác giả định cuối cùng, đọc A, và T_2 là giao tác cuối cùng ghi A trước khi T_f đọc.

Giao tác khác thực hiện việc ghi A chỉ là T_0, thực hiện ghi A trước T_2.

Tuy nhiên, T_0 luôn đứng trước tất cả các giao tác khác trong lịch. Vì thế, T_0 không thể can thiệp hay làm ảnh hưởng đến thứ tự của các cung $T_2 \rightarrow T_3$ và $T_2 \rightarrow T_f$.)

Bây giờ hãy xem xét các cung dựa trên đơn vị dữ liệu B. Lưu ý rằng T_0, T_1, T_2 và T_3 đều ghi B.

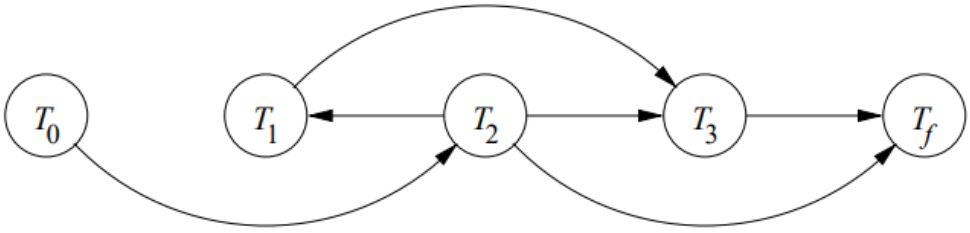
Trước tiên hãy xem xét cung $\mathbf{T_0 \rightarrow T_2}$. T_1 và T_3 là các giao tác khác thực hiện ghi B; T_0 và T_2 cũng ghi B, nhưng như chúng ta đã thấy, các đỉnh của cung không thể gây "nhiễu" (interference), vì vậy chúng ta không cần xem xét chúng.

Vì chúng ta không thể đặt T_1 giữa T_0 và T_2 theo nguyên tắc (3), chúng ta cần cặp cung ($T_1 \rightarrow T_0$; $T_2 \rightarrow T_1$). Tuy nhiên, không có gì có thể đứng trước T_0, vì vậy lựa chọn $\mathbf{T_1 \rightarrow T_0}$ **là không khả thi.** Trong trường hợp đặc biệt này, chúng ta chỉ cần thêm cung $T_2 \rightarrow T_1$ vào đồ thị phức. Nhưng cung này đã có sẵn của đơn vị dữ liệu A, vì vậy, chúng ta không thực hiện bất kỳ thay đổi nào đối với đồ thị phức để giữ T_1 bên ngoài cung $T_0 \rightarrow T_2$.

Chúng ta cũng không thể đặt T_3 giữa T_0 và T_2. Lý luận tương tự cho chúng ta biết rằng thêm cung $T_2 \rightarrow T_3$, thay vì một cặp cung. Tuy nhiên, cung này cũng đã có trong đồ thị do thuộc đơn vị dữ liệu A, vì vậy chúng ta không thực hiện bất kỳ thay đổi nào.

Tiếp theo, hãy xem xét cung $\mathbf{T_3 \rightarrow T_f}$. Vì T_0, T_1 và T_2 là những giao tác khác thực hiện viết B, chúng ta phải giữ chúng bên ngoài cung này. T_0 không thể nằm giữa T_3 và T_f, nhưng T_1 hoặc T_2 thì có thể. Vì không ai trong số chúng có thể nằm sau T_f, chúng ta phải buộc T_1 và T_2 xuất hiện trước T_3.

Đã có một cung $T_2 \rightarrow T_3$, nhưng chúng ta phải **thêm cung $\mathbf{T_1 \rightarrow T_3}$** vào đồ thị. Sự thay đổi này là cung duy nhất chúng ta phải thêm vào đồ thị, và tập hợp các cung của đồ thị được biểu diễn trong Hình 2. □



Hình 2. *Đồ thị phức hoàn chỉnh cho ví dụ 2.*

Ví dụ 3. Trong Ví dụ 2, tất cả các cặp cung đều là các cung đơn lẻ như một trường hợp đặc biệt. Hình 3 là một ví dụ về lịch trình gồm bốn giao dịch trong đó có một cặp cung thực sự trong đồ thị phức.

T_1	T_2	T_3	T_4
	$r_2(A);$		
$r_1(A);\ w_1(C);$			
		$r_3(C);$	
$w_1(B);$			
			$r_4(B);$
		$w_3(A);$	
			$r_4(C);$
	$w_2(D);\ r_2(B);$		
			$w_4(A);\ w_4(B);$

Hình 3. *Ví dụ về các giao tác làm cho đồ thị phức cần một cặp cung.*

Hình 4 cho thấy đồ thị phức chỉ có các cung xuất phát từ các kết nối từ nguồn đến thao tác đọc tương ứng. Như trong Hình 1, chúng ta gắn nhãn *(label)* mỗi cung theo (các) đon vị dữ liệu thuộc cung đó.

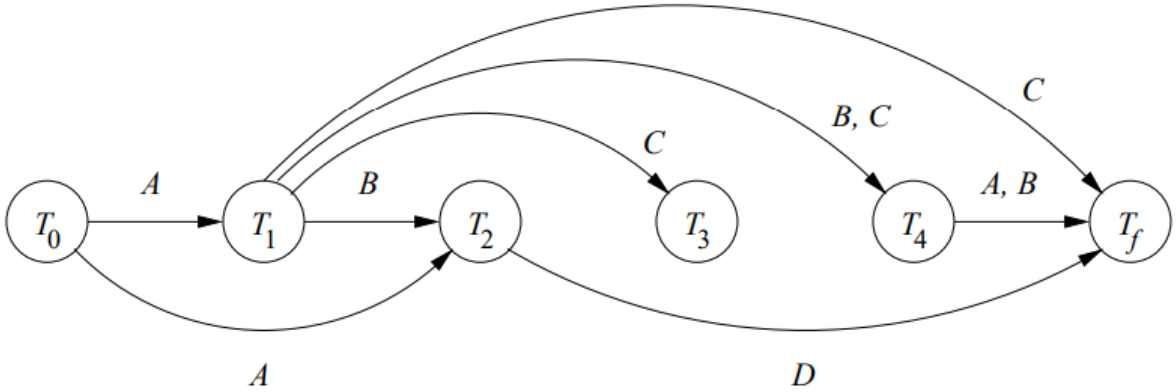
Sau đó, chúng ta phải xem xét các cách có thể để thêm cặp cung. Như chúng ta đã thấy trong **Ví dụ 2**, có một số cách đơn giản hóa mà chúng ta có thể thực hiện. Khi tránh sự can thiệp vào cung $T_j \rightarrow T_i$, duy nhất các giao tác cần được coi là T_k (giao dịch không thể ở giữa) là những giao tác thỏa:

- **Thao tác ghi của một đơn vị dữ liệu** đã tạo ra cung này Tj → Ti,
- **Nhưng không phải T_0** (T_b) **hoặc T_f** – những giao tác không bao giờ có thể là Tk, và
- **Không phải T_i hoặc T_j,** là các đỉnh của chính cung đó.

Với các quy tắc này, chúng ta hãy xem xét các cung thực hiện trên đơn vị dữ liệu A, được viết bởi T_0, T_3 và T_4.

Chúng ta không cần phải xem xét T_0. T_3 không được nằm giữa $T_4 \rightarrow T_f$, vì vậy chúng ta **thêm cung $T_3 \rightarrow T_4$;** hãy nhớ rằng cung còn lại trong cặp, *$T_f \rightarrow T_3$ không phải là một lựa chọn đúng.*

Tương tự như vậy, T_3 không được nằm giữa $T_0 \rightarrow T_1$ hoặc $T_0 \rightarrow T_2$, dẫn đến các cung **$T_1 \rightarrow T_3$ và $T_2 \rightarrow T_3$.**



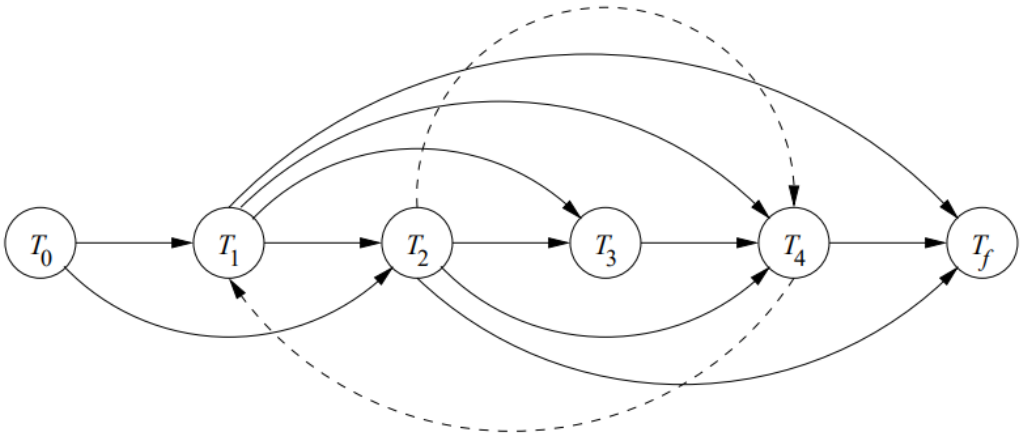
Hình 4. *Bước đầu cho việc vẽ đồ thị phức minh họa Ví dụ 3.*

Bây giờ, hãy xem xét, thực tế là T_4 cũng không được nằm giữa một cung do A. Đó là đỉnh của $T_4 \rightarrow T_f$, do đó cung đó không liên quan. T_4 không được nằm giữa $T_0 \rightarrow T_1$ hoặc $T_0 \rightarrow T_2$, do đó tạo ra các **cung $T_1 \rightarrow T_4$ và $T_2 \rightarrow T_4$.**

Tiếp theo, chúng ta hãy xem xét các cung thực hiện đơn vị dữ liệu B, được viết bởi T_0, T_1 và T_4.

Một lần nữa, chúng ta không cần xem xét T_0. Các cung duy nhất do B là $T_1 \rightarrow T_2$, $T_1 \rightarrow T_4$, và $T_4 \rightarrow T_f$. **T_1** không thể nằm giữa hai cung đầu tiên, *nhưng cung thứ ba yêu cầu thêm cung* ***$T_1 \rightarrow T_4$.***

T_4 chỉ có thể nằm giữa $T_1 \rightarrow T_2$. Cung này không có đỉnh nào ở T_0 hoặc T_f, do đó thực sự cần một cặp cung **($T_4 \rightarrow T_1$, $T_2 \rightarrow T_4$)**. Ta vẽ cặp cung này, cũng như tất cả các cung khác được thêm vào, như trong Hình 5.



Hình 5. *Đồ thị phức hoàn chỉnh của Ví dụ 3*

Tiếp theo, hãy xem xét các giao tác thực hiện ghi C: T_0 và T_1.

T_0 ta không cần quan tâm như giải thích trên. Ngoài ra, T_1 là một phần của mọi cung thuộc đơn vị dữ liệu C, vì vậy nó *không thể nằm ở giữa.*

Tương tự, D chỉ được ghi bởi T_0 và T_2, vì vậy chúng ta có thể xác định rằng *không cần thêm cung nào nữa.*

Do đó, đồ thị phức cuối cùng là máy trong Hình 5. □

3. Kiểm tra khả tuần tự View

Vì chúng ta chỉ phải chọn một cung trong mỗi cặp cung, nên chúng ta có thể tìm thấy một thứ tự tuần tự tương đương cho lịch trình S nếu và chỉ nếu có một số lựa chọn từ mỗi cặp cung biến đồ thị phức của S thành một **đồ thị phi chu trình.**

Để xem lý do tại sao, hãy lưu ý rằng nếu có một đồ thị phi chu trình như vậy, thì bất kỳ phép sắp xếp tôpô nào của đồ thị đều đưa ra một thứ tự mà *không có thao tác ghi nào có thể xuất hiện giữa thao tác đọc và nguồn của nó, và mọi thao ghi xuất hiện trước các thao tác đọc tương ứng của nó.* Do đó, các kết nối thao tác đọc-nguồn trong thứ tự tuần tự hoàn toàn giống như trong S; hai lịch tương đương View, và do đó **S có thể khả tuần tự View.**

Ngược lại (Conversely), nếu S có thể khả tuần tự View, thì có một thứ tự tuần tự tương đương View S'. Mỗi cặp cung ($T_k \rightarrow T_j$, $T_i \rightarrow T_k$) trong đồ thị phức của S phải có T_k trước T_j hoặc sau T_i trong S'; nếu không, việc viết bởi T_k sẽ phá vỡ kết nối từ T_j đến T_i, nghĩa là **S và S' không tương đương View.**

Tương tự như vậy, mọi cung trong đồ thị phức phải tuân theo thứ tự giao tác của S'. Chúng ta kết luận rằng có một lựa chọn cung từ mỗi cặp cung khiến đồ thị phức thành một đồ thị mà thứ tự tuần tự S' nhất quán với mỗi cung của đồ thị. Do đó, đồ thị này là phi chu trình.

Ví dụ 4. Xem xét đồ thị phức của Hình 2. Nó đã là một đồ thị, và nó là phi chu trình. Thứ tự tôpô duy nhất là (T_2; T_1; T_3), do đó là thứ tự tuần tự tương đương View cho lịch của Ví dụ 2.

Bây giờ hãy xem xét đồ thị phức của Hình 5. Chúng ta phải xem xét từng cung từ một cặp cung để lựa chọn xem có nên giữ lại không.

- Nếu chúng ta chọn $T_4 \rightarrow T_1$, thì sẽ có một chu trình.
- Tuy nhiên, nếu chúng ta chọn **$T_2 \rightarrow T_4$, thì kết quả là một đồ thị phi chu trình.**

Thứ tự tôpô duy nhất cho đồ thị này là (T_1; T_2; T_3; T_4). Thứ tự này tạo ra thứ tự tuần tự tương đương View và cho thấy lịch ban đầu có thể khả tuần tự View.

Tại sao tồn tại chu trình trong đồ thị thì không khả tuần tự View và cũng không khả tuần tự xung đột?

Khi tồn tại **chu trình trong đồ thị**, không thể tìm được một thứ tự tuần tự hóa S' tương đương với S.

- **Chu trình biểu thị mâu thuẫn thứ tự:**
  - Ví dụ: Nếu có chu trình $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$:
    - T_1 phải xảy ra trước T_2 (cung $T_1 \rightarrow T_2$).
    - T_2 phải xảy ra trước T_3 (cung $T_2 \rightarrow T_3$).
    - T_3 phải xảy ra trước T_1 (cung $T_3 \rightarrow T_1$).
  - Điều này là **mâu thuẫn logic**, vì không thể sắp xếp tuần tự hóa sao cho T_1, T_2, và T_3 thỏa mãn tất cả các cung.
- **Vi phạm View Serializable:**
  - Khi không thể sắp xếp thứ tự giao tác, sẽ xảy ra tình trạng thao tác ghi hoặc đọc trong S không tương đương với bất kỳ lịch trình tuần tự hóa nào S'.
  - **Ví dụ:** Một thao tác đọc có thể đọc giá trị từ một giao tác bị xung đột trong chu trình (đọc từ một giá trị "chưa ổn định"), gây ra kết quả không tương thích với bất kỳ lịch trình tuần tự nào.
- **Vi phạm Conflict Serializable:**
  - Khi không thể giải quyết các xung đột theo thứ tự hợp lệ, lịch trình không thể tương đương với bất kỳ lịch trình tuần tự nào.
  - Chu trình trong đồ thị xung đột là minh chứng cho việc không có thứ tự hợp lệ.

Tại sao lại có thể loại bỏ 1 trong 2 cung bất kỳ thuộc một cặp cung mà nó tạo chu trình trong đồ thị phức? Cơ chế nào loại bỏ hay chỉ cần chọn bất kỳ 1 trong 2 cung?

Tại sao có thể loại bỏ một cạnh trong cặp cạnh để phá chu trình?

Trong đồ thị phức (Serialization Graph), các cặp cạnh xuất hiện trong trường hợp có **xung đột về thứ tự giao tác** liên quan đến một thao tác trung gian. Ví dụ:

- Giả sử có hai cạnh $T_k \rightarrow T_j$ và $T_i \rightarrow T_k$, nhưng T_k xuất hiện trong cả hai cạnh, tạo nên một chu trình khi kết hợp với các cạnh khác.
- Loại bỏ **một trong hai cạnh** nghĩa là quyết định "không xem xét một mối quan hệ phụ thuộc cụ thể" để tránh chu trình.

Bằng cách này:

- Chỉ cần **một trong hai mối quan hệ phụ thuộc** được duy trì, thứ tự tuần tự hóa vẫn có thể tồn tại nếu không còn chu trình.
- Miễn là ta chọn cẩn thận, đồ thị kết quả vẫn phản ánh các phụ thuộc cần thiết và có thể xác định được một lịch tuần tự hóa tương đương.

- Việc loại bỏ một cung trong chu trình tương đương với việc *ép buộc* một thứ tự giữa hai giao tác liên quan đến cặp cung đó.
- Ví dụ, nếu ta loại bỏ cung $T_1 \rightarrow T_2$, điều đó có nghĩa là ta đang giả định rằng T_2 *không* phụ thuộc vào T_1 theo cách mà cung đó biểu diễn. Nói cách khác, ta đang xét trường hợp mà T_1 được thực hiện *trước* T_2.
- Bằng cách loại bỏ *từng* cung trong chu trình và kiểm tra xem đồ thị kết quả có còn chu trình hay không, ta đang xét *tất cả* các khả năng thứ tự tương đối giữa các giao tác trong chu trình đó.
- Việc loại bỏ một cung không phải là một "phép màu" loại bỏ xung đột. Nó là một cách để *xét tất cả các khả năng thứ tự* giữa các giao tác trong chu trình. Nếu *tồn tại* một cách loại bỏ cung (tức là tồn tại một thứ tự giao tác) mà đồ thị không còn chu trình, thì lịch đó khả tuần tự view. Nếu *không có* cách nào loại bỏ cung mà đồ thị hết chu trình, thì lịch đó không khả tuần tự view.

Cơ chế loại bỏ và việc chọn cung:

- **Không có cơ chế tự động "loại bỏ" cung nào cả.** Việc này là một bước trong thuật toán kiểm tra tính tuần tự xem.
- **Cần xét tất cả các khả năng:** Để xác định chắc chắn lịch sử có tuần tự xem hay không, ta cần thử loại bỏ *từng* cung trong mỗi chu trình và kiểm tra xem đồ thị kết quả có còn chu trình hay không. Nếu *tồn tại* một cách loại bỏ cung mà đồ thị không còn chu trình, thì lịch sử đó tuần tự xem.
- **Chọn bất kỳ cung nào trong cặp cung:** Về mặt lý thuyết, bạn có thể chọn bất kỳ cung nào trong cặp cung tạo chu trình để loại bỏ. Tuy nhiên, việc này cần được thực hiện một cách có hệ thống để xét *tất cả* các khả năng.

Tính chất của khả tuần tự View và khả tuần tự xung đột

- **Khả tuần tự view:** Một lịch được coi là khả tuần tự view nếu nó tương đương với ***ít nhất một*** thứ tự tuần tự của các giao tác.
- **Khả tuần tự xung đột:** Một lịch được coi là khả tuần tự xung đột nếu nó có thể được chuyển đổi thành một thứ tự tuần tự bằng cách hoán đổi các thao tác *không xung đột*.

Chính xác là, khả tuần tự xung đột (conflict serializability) yêu cầu lịch phải tương đương với ***mọi*** thứ tự tuần tự **có thể có được bằng cách hoán đổi các thao tác không xung đột.**

Điều này mạnh hơn so với yêu cầu của khả tuần tự view.