

# Checkpointing, Undo, Redo, Undo/Redo Logging

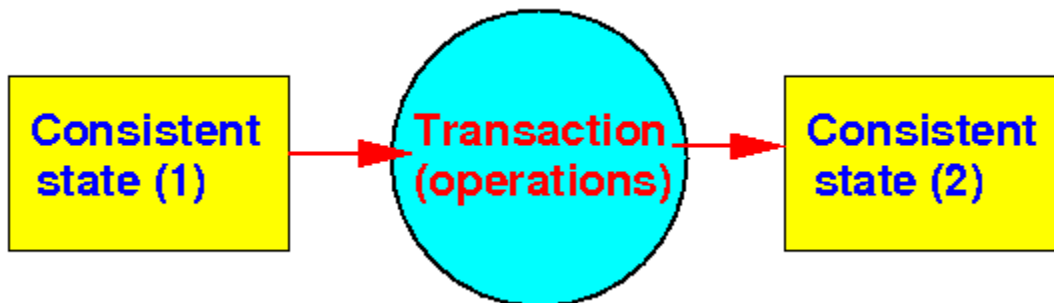
## 1. Dẫn nhập

Hệ quản trị cơ sở dữ liệu cần đảm bảo tính chất ACID của dữ liệu và các giao tác (transaction).

Giao tác phải có các thuộc tính sau:

- ✓ *Tính nguyên tử (Atomicity)*: TẤT CẢ các hoạt động trong một giao tác được thực hiện hoặc KHÔNG có hoạt động nào trong một giao tác được thực hiện.
- ✓ *Tính nhất quán (Consistency)*: Giao tác được đảm bảo để chuyển đổi cơ sở dữ liệu từ trạng thái nhất quán này sang trạng thái nhất quán khác.
- ✓ *Tính cô lập (Isolation)*: Nếu hai giao tác đang thực hiện đồng thời, mỗi giao tác sẽ thấy cơ sở dữ liệu như thể giao tác đang thực hiện tuần tự (= trong sự cô lập).
- ✓ *Tính bền vững (Durability)*: Khi giao tác được hoàn tất thành công, các thay đổi đối với trạng thái cơ sở dữ liệu được thực hiện bởi giao tác phải là vĩnh viễn.

Một cơ sở dữ liệu có thể ở một trong hai trạng thái là: Nhất quán hoặc Không nhất quán. Khi tất cả các thay đổi trong một giao tác được thực hiện, trạng thái cơ sở dữ liệu kết quả là một trạng thái nhất quán (nếu trạng thái ban đầu là nhất quán).

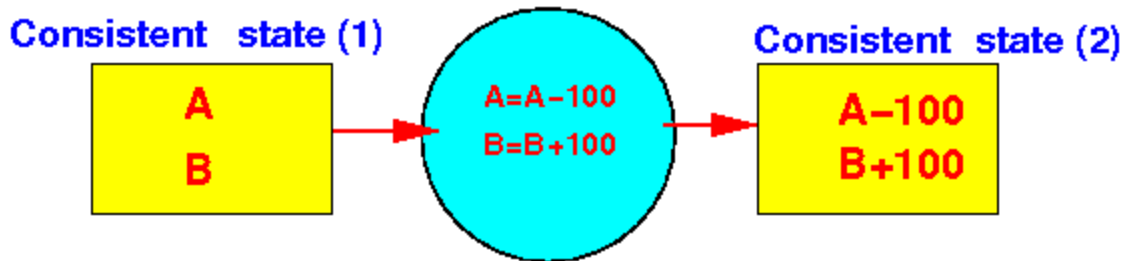


Có 2 nguyên nhân của trạng thái cơ sở dữ liệu không nhất quán:

- ✓ Lỗi hệ thống (hệ điều hành, hệ quản trị cơ sở dữ liệu hoặc nguồn điện bị hỏng).
- ✓ Thực thi đồng thời các giao tác.

Với nguyên nhân do Lỗi hệ thống:

Xem xét giao dịch chuyển 100 đô la từ A sang B



Có 2 trạng thái nhất quán có thể có:

- ✓ Trạng thái nhất quán 1: A B
- ✓ Trạng thái nhất quán 2: A - 100 B + 100

Xem xét trạng thái cơ sở dữ liệu do lỗi hệ thống sau

**Transaction:** transfer \$100 from A to B

read A

A.balance = A.balance - 100

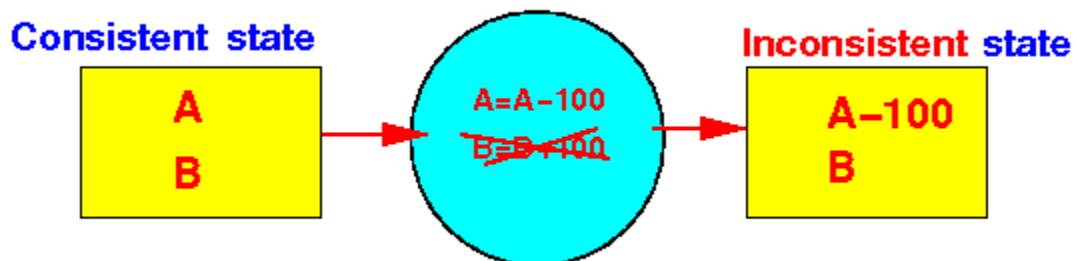
write A

===== system fails here

~~read B~~

~~B.balance = B.balance + 100~~

~~write B~~



Trạng thái cơ sở dữ liệu kết quả là:

Trạng thái cơ sở dữ liệu = A - 100 B

Chúng ta thấy rằng: Trạng thái cơ sở dữ liệu trên không phải là một trong 2 trạng thái nhất quán!!!

Thực hiện đồng thời (các giao tác) có thể gây ra các trạng thái không nhất quán

Xem xét 2 giao tác sau:

**T1 = Deposit \$1 to A:**

read A

A = A + 1

write A

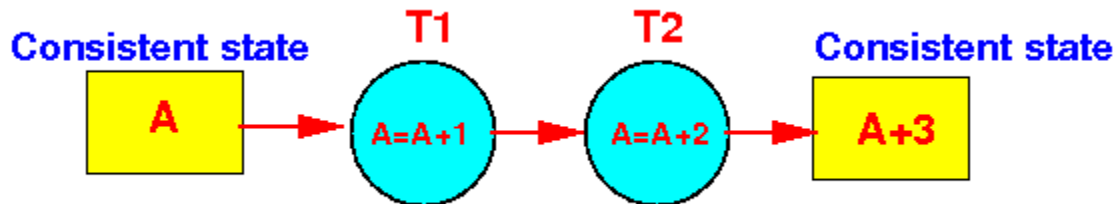
**T2 = Deposit \$2 to A:**

read A

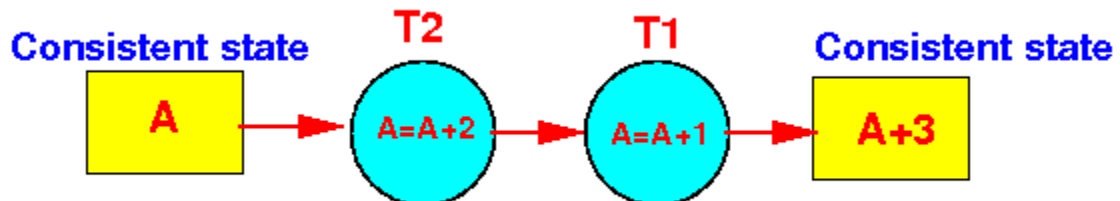
$A = A + 2$   
`write A`

Các trạng thái cơ sở dữ liệu nhất quán có thể thực hiện T1 và T2 là:

Trường hợp 1: T1 trước T2

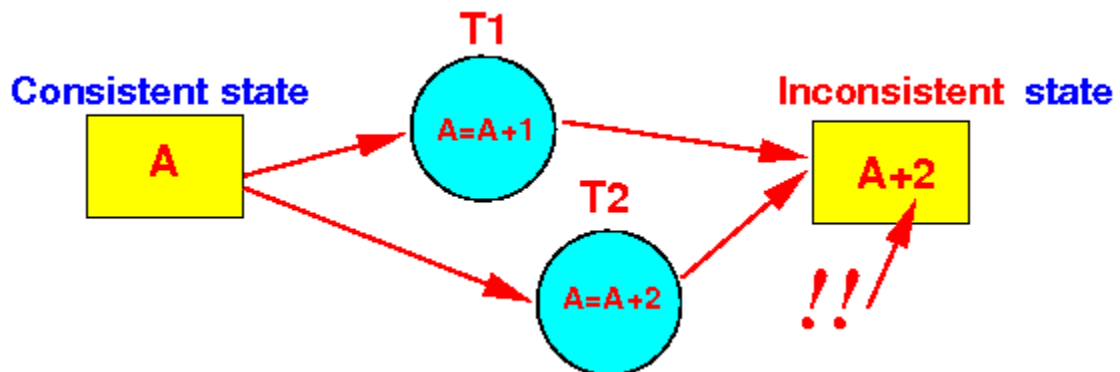


Trường hợp 2: T2 trước T1



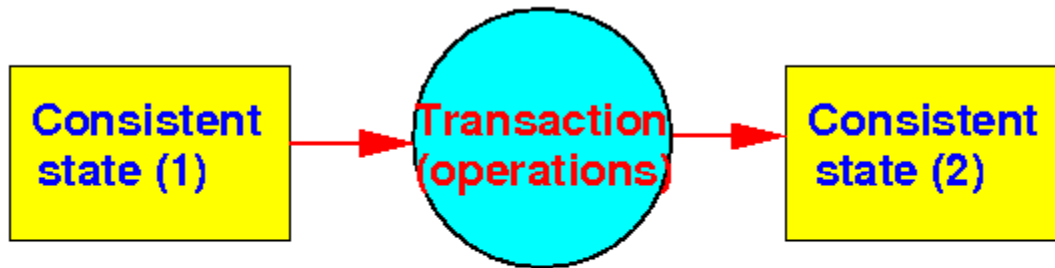
Xem xét việc thực thi đồng thời T1 và T2 sau đây:

T1	T2	
=====		
<code>read A</code>		Initially: $A = 10$
	<code>read A</code>	
<code>A = A + 1</code> ( $A = 11$ )		
	<code>A = A + 2</code> ( $A = 12$ )	
<code>write A</code>		Writes 11 to A
	<code>write A</code>	Writes 12 to A
<hr/>		
Final database state:		$A = 12 (= A + 2)$



Giả sử rằng cơ sở dữ liệu ở trạng thái nhất quán. Sau đó, một giao tác sẽ chuyển đổi cơ sở dữ liệu thành trạng thái nhất quán (khác) nếu:

- ✓ Không có lỗi hệ thống
- ✓ Không có giao tác nào khác thực hiện trong hệ thống cơ sở dữ liệu



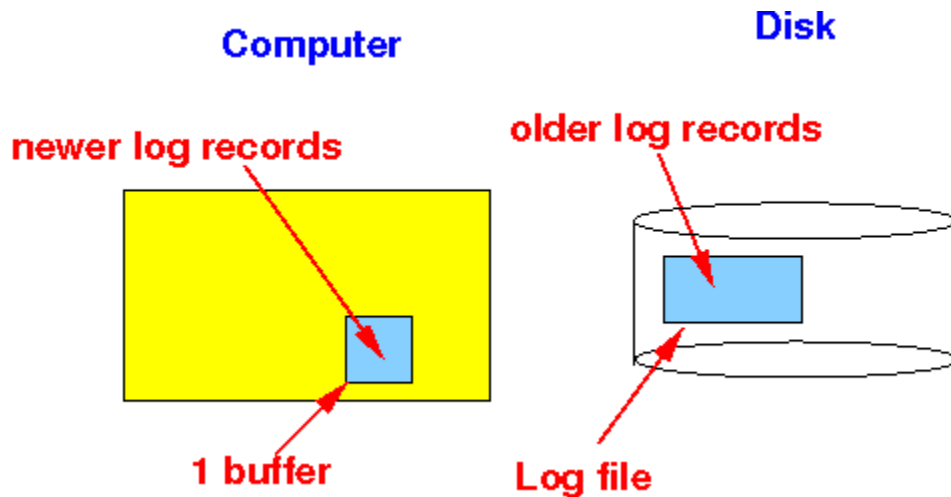
- 1. no system failures**
- 2. executed in isolation**

## 2. Logging

Có 3 kỹ thuật ghi nhật ký:

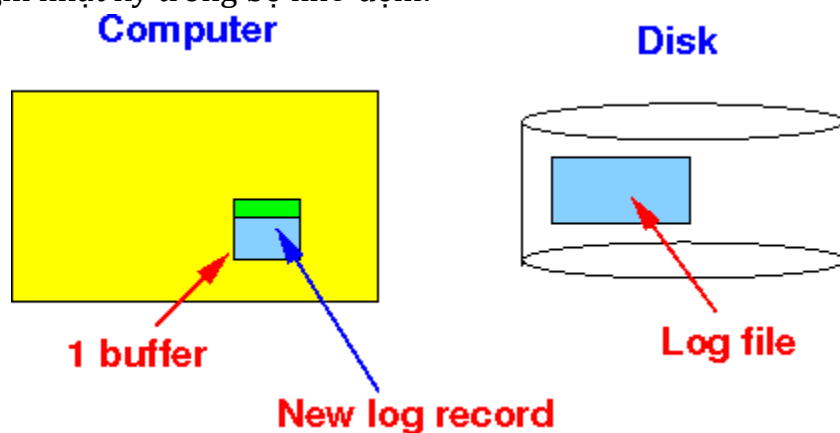
- ✓ Undo logging: Tập tin nhật ký chứa các bản ghi nhật ký cho phép hoàn tác (= roll back) các thay đổi được thực hiện bởi giao tác CHƯA hoàn tất.
- ✓ Redo logging: Tập tin nhật ký chứa các bản ghi nhật ký cho phép làm lại (= roll forward) các thay đổi được thực hiện bởi giao tác ĐÃ hoàn tất.
- ✓ Undo/redo logging: Tập tin nhật ký chứa cả hai lần hoàn tác và làm lại các bản ghi nhật ký và cho phép:
  - hoàn tác (= roll back) các thay đổi được thực hiện bởi giao tác chưa hoàn tất.
  - làm lại (= roll forward) các thay đổi được thực hiện bởi giao tác đã hoàn tất.

Tập tin nhật ký bao gồm 2 phần:

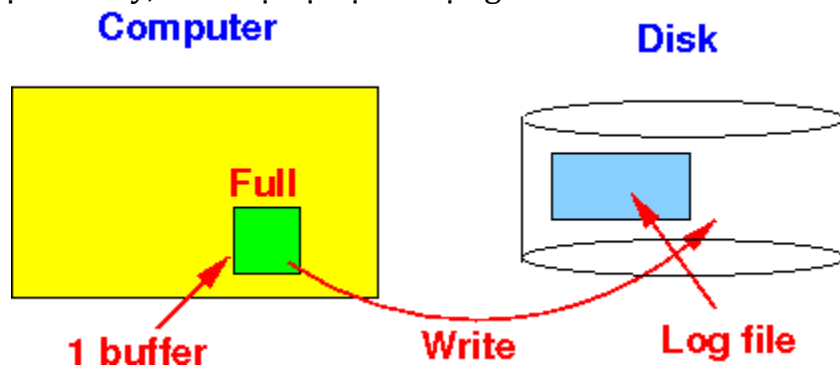


- ✓ Các bản ghi cũ của tập tin nhật ký được lưu trữ trên đĩa.
- ✓ Các bản ghi nhật ký mới hơn của tập tin nhật ký được lưu trữ trong bộ nhớ.

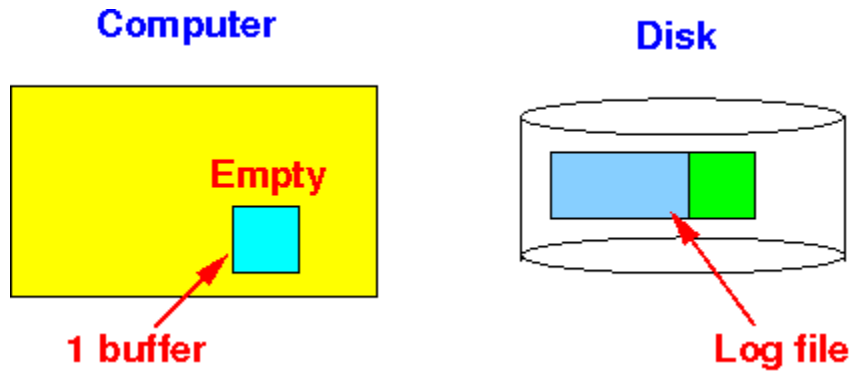
Khi người quản lý giao tác viết một bản ghi nhật ký mới, bản ghi nhật ký mới được nối vào các bản ghi nhật ký trong bộ nhớ đệm:



Khi bộ đệm bộ nhớ đầy, toàn bộ bộ đệm được ghi vào đĩa:

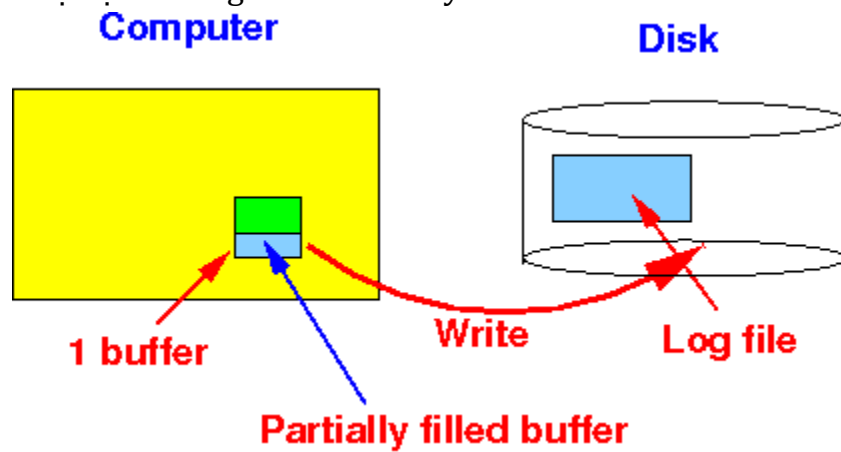


Nội dung bộ đệm bây giờ là một khối đĩa được nối vào tập tin nhật ký:

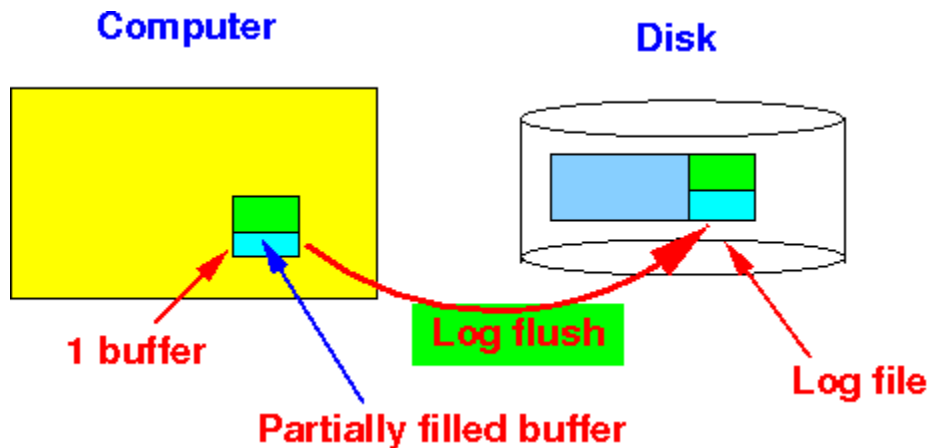


Thao tác ghi nhật ký thực tế:

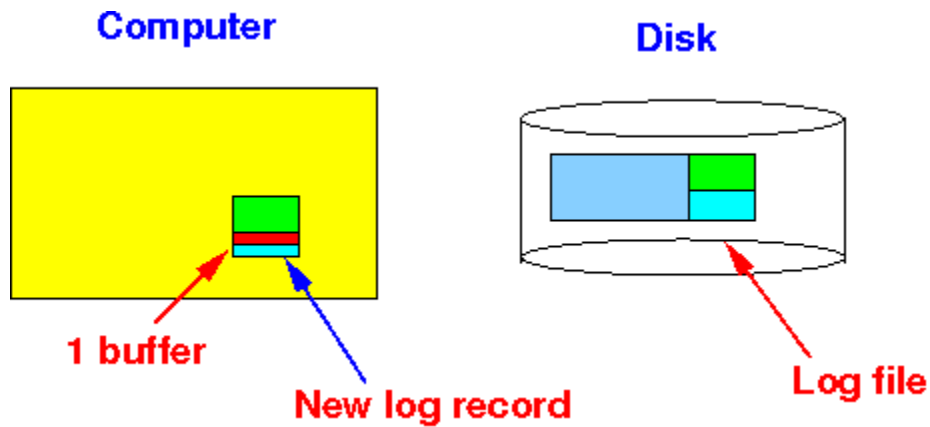
Do các quy tắc ghi nhật ký trong giao thức ghi nhật ký, hệ quản trị phải ghi nhật ký vào đĩa ngay cả khi bộ đệm không hoàn toàn đầy:



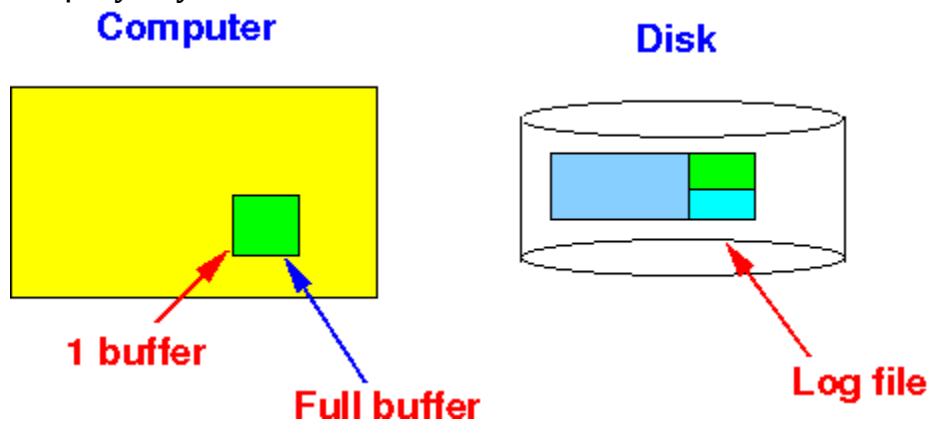
Thao tác *log flush*: Thao tác ghi nhật ký sẽ buộc các bản ghi nhật ký trong bộ nhớ được ghi vào đĩa.



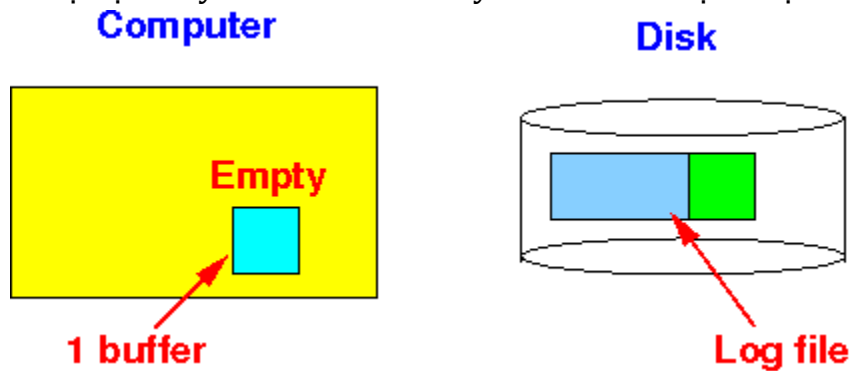
Sau khi *log flush*, các bản ghi nhật ký mới có thể được thêm vào bộ đệm nhật ký trong bộ nhớ:



Khi bộ đệm nhật ký đầy:



Hệ quản trị viết bộ đệm đầy đủ vào đĩa và thay thế khối dữ liệu được điền một phần:



### 3. Giao tác

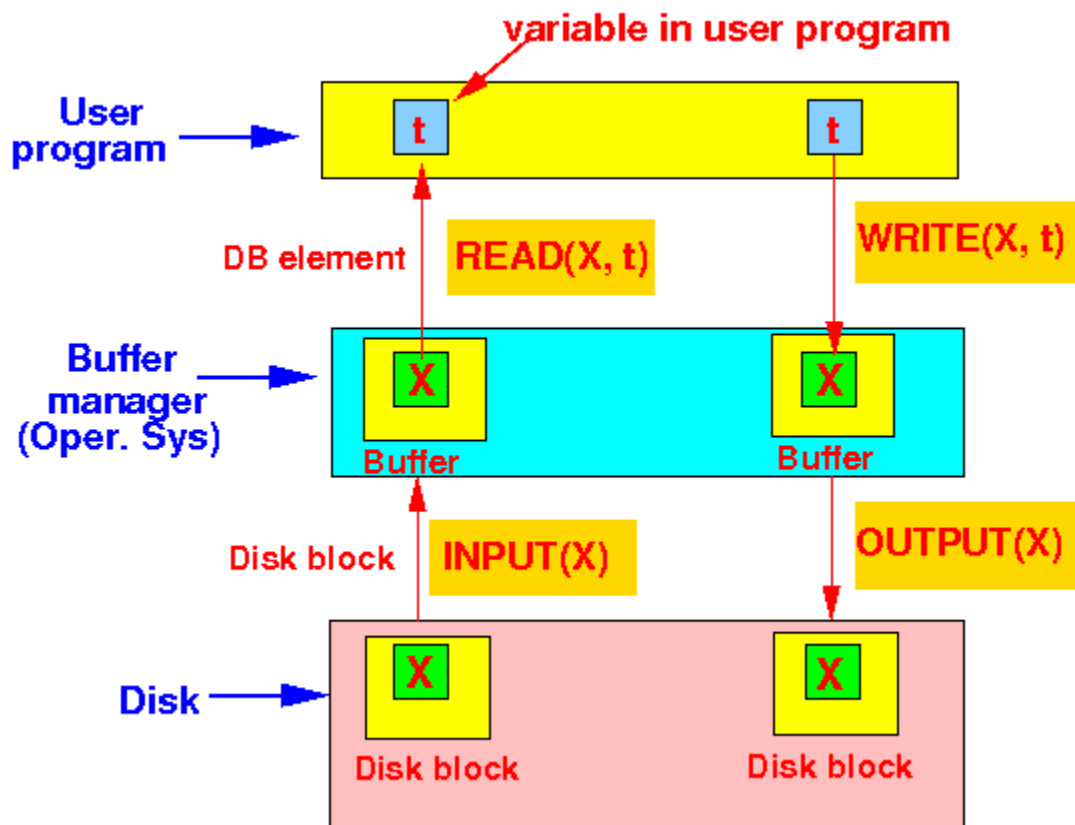
Những hoạt động cơ bản được sử dụng bởi các giao tác

Các giao tác tương tác với cơ sở dữ liệu bằng cách sử dụng các hoạt động cơ bản sau đây:

INPUT (X)	(X là phần tử cơ sở dữ liệu)
OUTPUT (X)	(X là phần tử cơ sở dữ liệu)
READ (X, t)	(X là phần tử cơ sở dữ liệu, t là một biến chương trình)

**WRITE (X, t)** (X là phần tử cơ sở dữ liệu, t là một biến chương trình)

- ✓ **INPUT (X)**: Sao chép/chuyển khối đĩa chứa phần tử cơ sở dữ liệu X vào bộ đệm trong trình quản lý bộ đệm (Trình quản lý bộ đệm duy trì bộ nhớ đệm có sẵn trong bộ nhớ).
- ✓ **READ (X, t)**: Sao chép / chuyển phần tử cơ sở dữ liệu X vào biến cục bộ của giao tác t. Nếu phần tử cơ sở dữ liệu X đã có trong bộ đệm trình quản lý bộ đệm, thì giá trị được sao chép vào biến cục bộ t. Nếu phần tử cơ sở dữ liệu X không nằm trong bộ đệm của trình quản lý bộ đệm, thì một **INPUT (X)** được thực thi và sau đó giá trị được sao chép vào biến cục bộ t.
- ✓ **WRITE (X, t)**: Sao chép/chuyển giá trị trong biến cục bộ của giao tác đến phần tử cơ sở dữ liệu X. Nếu phần tử cơ sở dữ liệu X đã có trong bộ đệm của trình quản lý bộ đệm, thì giá trị của biến cục bộ t được sao chép vào phần tử cơ sở dữ liệu X. Nếu phần tử cơ sở dữ liệu X không nằm trong bộ đệm của trình quản lý bộ đệm, thì một **INPUT (X)** được thực hiện và sau đó giá trị được sao chép vào phần tử cơ sở dữ liệu X.
- ✓ **OUTPUT (X)**: Chuyển khối chứa phần tử cơ sở dữ liệu X từ bộ nhớ đệm (trong trình quản lý bộ đệm) sang đĩa.



Ta có giao tác sau:



**Database elements:**

A = 1000

B = 2000

**Operations in T:**

READ (A, x)

x = x + 100

WRITE (A, x)

READ (B, x)

x = x + 100

WRITE (B, x)

Mô phỏng những hoạt động xảy ra trong hệ quản trị cơ sở dữ liệu:

Action	<i>Local variable</i> x	<i>Copy of A in memory buffer</i> Mem A	Mem B	<i>Copy of A on disk</i> Disk A	Disk B
READ(A, x)	1000	1000		1000	2000
x = x + 100	1100	1000			
WRITE(A, x)	1100	1100			
READ(B, x)	2000		2000		2000
x = x - 100	1900				
WRITE(B, x)	1900		1900		
OUTPUT(A)		1100		1100	
OUTPUT(B)			1900		1900

Viết bản ghi nhật ký vào nhật ký (tập tin) khi giao tác thực hiện một trong các thao tác sau:

- ✓ Bắt đầu giao tác: Viết: **<START T>** vào nhật ký
- ✓ Đọc dữ liệu: Viết: **<READ ...>** vào nhật ký
- ✓ Viết (cập nhật) dữ liệu: Viết: **<WRITE ...>** vào nhật ký

- ✓ Kết thúc: Viết: **<COMMIT T>** hoặc **<ABORT T>** vào nhật ký

## 4. Undo logging

### 4.1. Ý tưởng

Ý tưởng này xuất phát từ truyện cổ Grimm, Hansel và Gretel, một gia đình nghèo quyết định bỏ con trong rừng sâu để qua được nạn đói. Hai anh em ghi nhận các bước cha mẹ họ thực hiện, đánh dấu bằng sỏi trắng, bánh mỳ và theo đó để có thể trở về nhà.

Tương tự trong cơ sở dữ liệu, chúng ta có thể ghi nhận lại trong nhật ký (log) những điều chúng ta đã làm, nhưng phải tổ chức lưu trữ một cách hợp lý.

### 4.2. Định nghĩa

Undo Logging hay còn gọi là Immediate Modification log hoạt động theo nguyên tắc: trước khi viết bất cứ thứ gì vào đĩa, hệ quản trị sẽ ghi lại giá trị **CŨ** vào nhật ký và chỉ sau đó giá trị mới được ghi xuống đĩa.

Vì vậy, cách hệ quản trị làm điều đó là:

- ✓ Cho phép sửa đổi mọi thứ trong bộ nhớ.
- ✓ Trong khi sửa đổi chúng, hệ quản trị tạo các bản ghi tương ứng trong nhật ký giữ giá trị **CŨ**.
- ✓ Tất cả các bản ghi nhật ký phải được ghi lên đĩa trước khi ghi các mục cơ sở dữ liệu đã được thay đổi trở lại đĩa.

Ví dụ:

Transaction T1	Log	Comment
	$\langle T1, \text{start} \rangle$	Khi giao tác bắt đầu
<code>read(A, t); t ← t × 2;</code>		
<code>write(A, t)</code>	$\langle T1, A, 8 \rangle$	Ngay lúc này cho phép output (A)
<code>read(B, t); t ← t × 2;</code>		
<code>write(B, t)</code>	$\langle T1, B, 8 \rangle$	Ngay lúc này cho phép output (B)
<code>output(A)</code>		
<code>output(B)</code>		Ngay lúc này tất cả thay đổi đều được ghi xuống đĩa cứng
	$\langle T1, \text{commit} \rangle$	Giao tác đã hoàn tất

Biểu mẫu của bản ghi nhật ký là:

$\langle \text{id của giao tác}, \text{mục dữ liệu}, \text{giá trị cũ} \rangle$

$\langle T1, B, 8 \rangle$  có nghĩa là giao tác T1 đã sửa đổi mục B và giá trị cũ là 8

$\langle T1, \text{commit} \rangle$  nghĩa là T1 đã hoàn tất thành công và mọi thứ đã được ghi vào đĩa

### 4.3. Đặc điểm

Nhật ký của phương pháp Undo logging có các đặc điểm:

- ✓ Nhật ký đầu tiên được ghi vào bộ nhớ và sau đó vào đĩa
- ✓ Không thể flush log lên đĩa lúc thực thi từng câu lệnh - điều đó sẽ dẫn đến quá nhiều I/O.

Các tình huống xấu cần tránh:

- ✓ Một mục dữ liệu được sửa trên đĩa, nhưng không có bản ghi nhật ký tương ứng nào được ghi lại.
- ✓ Toàn bộ nhật ký nằm trên đĩa (bao gồm cả bản ghi  $\langle T, \text{commit} \rangle$ ) nhưng bản thân các giá trị mới chưa được ghi xuống.

### 4.4. Quy tắc

#### *Quy tắc ghi nhật ký*

- ✓ Với mỗi hành động tạo bản ghi nhật ký với giá trị cũ
- ✓ Trước khi phần tử  $X$  được sửa đổi trên đĩa cứng, hệ quản trị ghi tất cả các bản ghi nhật ký thuộc về  $X$  vào đĩa
  - Đây được gọi là *Write-Ahead Logging*
  - Trước khi ghi một giá trị mới, ghi tất cả các bản ghi nhật ký tương ứng
- ✓ Trước khi bạn ghi lệnh commit vào nhật ký, tất cả các sửa đổi phải được ghi lên đĩa.

#### *Quy tắc khôi phục từ bản ghi nhật ký*

Cách khôi phục từ bản ghi nhật ký như sau:

Hệ quản trị cơ sở dữ liệu hoàn tác các giao tác không thành công, điều này mang ý nghĩa là đặt cơ sở dữ liệu vào trạng thái trước giao tác này

Recover(log  $L$ )

```
for every transaction  $T_i$  that has a  $\langle T_i, \text{start} \rangle$  record in the log
  if there's already  $\langle T_i, \text{commit} \rangle$  or  $\langle T_i, \text{abort} \rangle$ 
    nothing
  otherwise - rollback:
    for all  $\langle T_i, X, v \rangle \in L$ 
      write( $X, v$ )
      output( $X$ )
    write  $\langle T_i, \text{abort} \rangle$  to  $L$ 
```

$\langle T_i, \text{abort} \rangle$  ghi nhận việc hủy bỏ giao tác

Để tránh tình trạng khi đang ở giữa quá trình hủy bỏ thì nguồn điện bị ngắt, hệ quản trị sẽ quy định nếu hủy một giao tác thành công, chúng ta không bao giờ phải làm điều đó một lần nữa.

Nếu trong thời gian rollback nguồn điện bị ngắt một lần nữa - lúc này điều đó không là vấn đề với hệ quản trị

- ✓ Hệ quản trị sẽ chỉ ghi đè lên giá trị cũ một lần nữa. Ghi giá trị cũ hai lần sẽ giống như chỉ ghi một lần (nó sẽ không thay đổi)

- ✓ Theo cách này, bạn được đảm bảo quay trở lại trạng thái nhất quán

Vấn đề: điều gì sẽ xảy ra nếu một giao tác thay đổi giá trị của một số biến số nhiều lần? Trong trường hợp này, chúng ta chỉ nên khôi phục cái đầu tiên và bỏ qua phần còn lại.

Recover(log **L**)

let **S\*** be all set of transactions **T<sub>i</sub>** with  $\langle T_i, \text{start} \rangle \in L$

(1) let **S** be all transaction **T<sub>i</sub>**  $\in S^*$  without  $\langle T_i, \text{commit} \rangle \in L$  or  $\langle T_i, \text{abort} \rangle \in L$

(2) for each  $\langle T_i, X, v \rangle \in \text{reverse}(L)$  (reverse order: latest  $\rightarrow$  earliest)

    if **T<sub>i</sub>**  $\in S$ :

        write(**X**, **v**)

        output(**X**)

(3) for each **T<sub>i</sub>**  $\in S$

    write  $\langle T_i, \text{abort} \rangle$  to **L**

Lưu ý rằng có thể có một số giao tác đang diễn ra cùng một lúc, như vậy có thể viết  $\langle T_i, \text{abort} \rangle$  theo thứ tự nào?

Ví dụ: **T1** và **T2** đều ghi **A**, **T1** trước **T2**, giả sử rằng cả **T1** và **T2** đều được khôi phục. Giả sử chúng ta hoàn tác cả hai, nhưng chỉ viết  $\langle T1, \text{abort} \rangle$  (nguồn điện bị cắt khi viết  $\langle T2, \text{abort} \rangle$ )

Hoàn tác một cái gì đó 2 lần không phải là một vấn đề, nhưng ở đây chúng ta có hai giao tác, nhớ lại rằng  $\langle T1, \text{abort} \rangle$  có nghĩa là giá trị trên đĩa là giá trị **A** xảy ra trước **T1**, chúng ta đã hoàn tác **T1** và hiện đang cố gắng hoàn tác **T2**, điều này sẽ quay trở lại giá trị đã có trước **T2**, ghi đè giá trị trước **T1**. (Thực tế có thể **A** nhận giá trị được ghi bởi **T1** mà chúng ta rollback)  $\Rightarrow$  **BAD STATE**

Nếu chúng ta ghi  $\langle T2, \text{abort} \rangle$ , không ghi  $\langle T1, \text{abort} \rangle$  thì sẽ không có vấn đề gì trong trường hợp này  $\Rightarrow$  chúng ta phải viết bản ghi **abort** theo thứ tự đảo ngược thời gian bắt đầu của các giao tác. Tức là bắt đầu với bản ghi mới nhất – nó sẽ được hoàn tác đầu tiên và bản ghi  $\langle \text{abort} \rangle$  của nó sẽ xuất hiện đầu tiên trong nhật ký.

## 4.5. Checkpoints

Nếu cứ theo dõi mọi thứ hệ quản trị làm, máy tính sẽ nhanh chóng hết dung lượng để ghi nhật ký. Lúc này, hệ quản trị có thể giải phóng một số không gian bằng cách cắt bớt nhật ký. Liệu có phần nào của nhật ký mà chúng ta biết chắc chắn không cần thiết nữa và có thể loại bỏ an toàn không?

Trong lúc checkpoint nhật ký, hệ quản trị sẽ rút ngắn tập tin nhật ký bằng cách "xóa" bản ghi nhật ký khỏi các giao tác đã hoàn tất, giao tác hoàn tất bao gồm: Giao tác đã *commit* (= đã hoàn thành thành công) và Giao tác bị *abort* (= hoàn thành không thành công)

Từ "xóa" được đặt trong dấu ngoặc kép vì thao tác checkpoint nhật ký sẽ xóa các bản ghi nhật ký một cách hợp lý. Hệ quản trị sẽ viết bản ghi LOG CHECK POINT vào tập tin nhật ký. Một số bản ghi nhật ký trước bản ghi LOG CHECK POINT này sẽ không được kiểm tra trong quá trình khôi phục.

Có 2 cách xóa bản ghi từ tập tin nhật ký:

- ✓ Cách vật lý: Các bản ghi nhật ký thực sự bị xóa khỏi tập tin nhật ký
- ✓ Cách logic: Tập tin nhật ký được đánh dấu bằng bản ghi "check point" đặc biệt. Một số phần của tập tin nhật ký sẽ bị bỏ qua (= bị bỏ qua khi hệ quản trị sử dụng nó trong khôi phục)

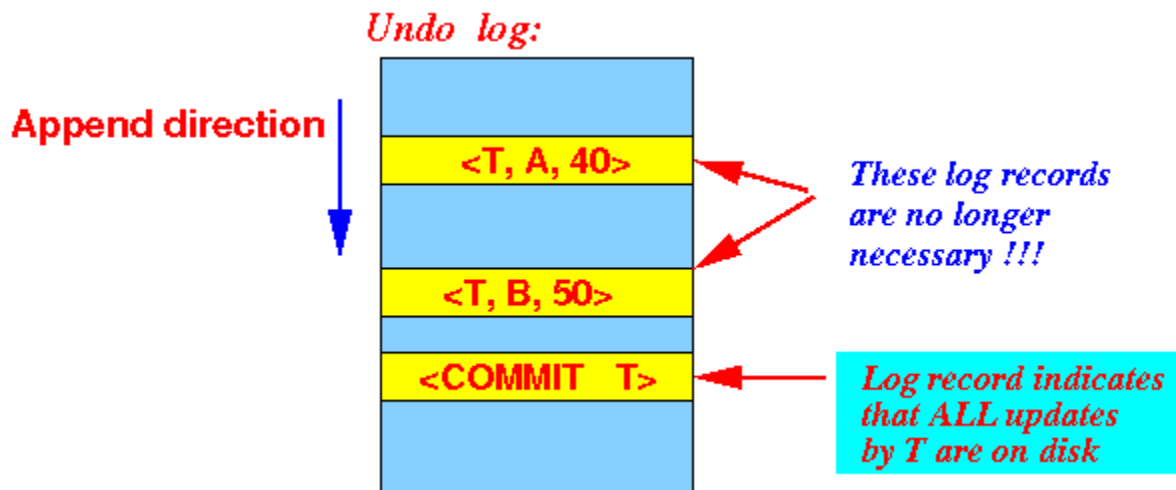
Trong thực tế, đặc biệt trong giao dịch ngân hàng, tất cả các bản ghi nhật ký phải được lưu trữ!!!

Để cài đặt checkpoint nhật ký, đầu tiên, một bản ghi nhật ký đặc biệt (log) được ghi vào tập tin nhật ký. Tiếp theo, các hoạt động phục hồi sẽ chủ yếu sử dụng phần của tập tin nhật ký được viết sau bản ghi LOG CHECK POINT. Viết là "chủ yếu sử dụng" bởi vì bạn sẽ thấy sau đó hệ quản trị cũng sẽ cần phải sử dụng một phần nhỏ của tập tin nhật ký trước bản ghi LOG CHECK POINT).

## 4.6. Thuật toán sử dụng Quiescent checkpoint (đối với undo log)

### 4.6.1. Các bước cài đặt checkpoint

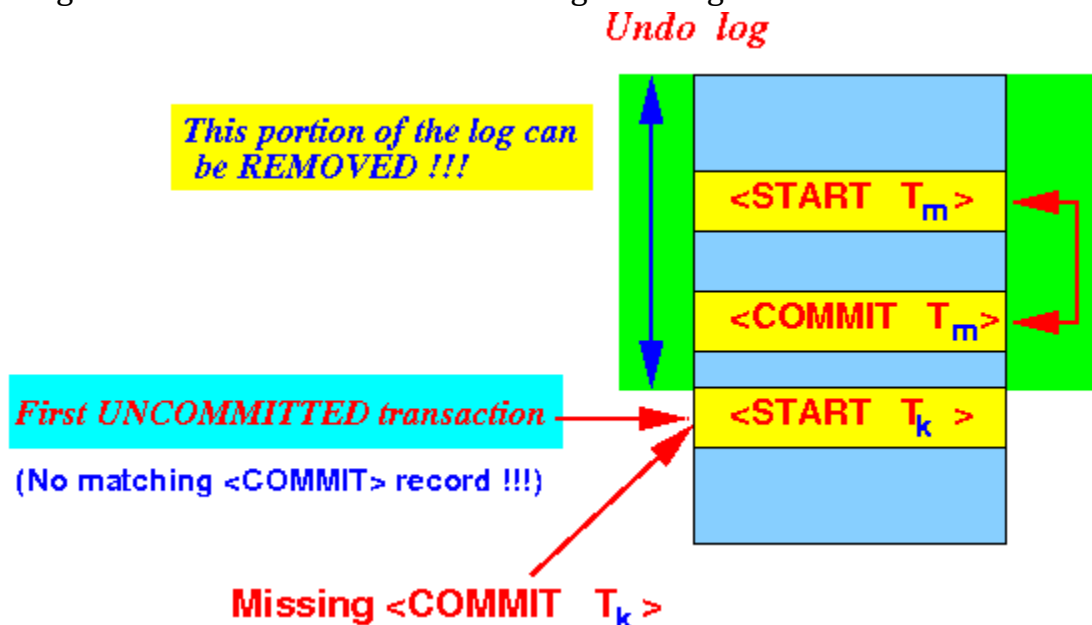
Khi bạn tìm thấy bản ghi <COMMIT T> cho giao tác T, thì các bản ghi undo log được ghi nhận của giao tác T hiện không còn cần thiết nữa



Lý do là các bản ghi undo log không được sử dụng để khôi phục các hành động trong giao tác T trong trường hợp có lỗi hệ thống xảy ra. (Các undo log này giống như bảo hiểm cho một tai nạn). Dòng nhật ký <COMMIT T> là bằng chứng cho thấy giao tác T đã hoàn tất thành công. Bản ghi nhật ký (bảo hiểm) không còn cần thiết vì sự kiện đã trôi qua.

Cách cắt ngắn Undo log:

(1) Tìm giao tác đầu tiên chưa commit trong undo log:



(2) Bạn có thể xóa mọi thứ trước bản ghi <START T<sub>k</sub>> này, vì tất cả các bản ghi nhật ký trước giao tác chưa commit đầu tiên thuộc về một giao tác đã commit !!!

Có 2 kỹ thuật cắt bỏ undo log là Quiescent check pointing và Non-quiescent check pointing.

Thực tế là tập tin nhật ký được sử dụng để khôi phục lỗi giao tác trong khi một số giao tác khác vẫn đang chạy (thực thi). Vì thế, nếu không có việc thực thi giao tác, chúng ta

sẽ không cần tập tin nhật ký!!!! Nội dung của tập tin nhật ký không cần thiết cho mục đích khôi phục! (Vì không có giao tác nào cần được "thu hồi").

Cách dễ dàng (nhưng không thực tế) để checkpoint một undo log là chờ cho đến khi tất cả các giao tác đã hoàn tất/hủy bỏ. Sau đó, toàn bộ undo log có thể bị cắt bỏ!!!

Quiescent = không hoạt động (= không có giao tác nào đang chạy)

Thuật toán sử dụng Quiescent checkpoint trên undo log:

- (1) Làm cho DBMS ngừng chấp nhận giao tác mới
- (2) Chờ cho đến khi tất cả các giao tác hiện đang hoạt động được commit hoặc abort (và đã viết bản ghi nhật ký <COMMIT> hoặc <ABORT>)
- (3) Ghi nhật ký vào đĩa
- (4) Ghi dòng <CKPT> (checkpoint) vào nhật ký ---- Đánh dấu vị trí chúng ta cho là "hữu ích"
- (5) Làm gọn nhật ký
- (6) Tiếp tục chấp nhận giao tác mới

Ví dụ:

Hiện tại: T1 và T2 đang hoạt động:

Undo log:

```
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
```

Bây giờ chúng ta sẽ thực hiện đặt một checkpoint bằng cách đợi cho đến khi T1 và T2 commit hoặc abort, ghi dòng <CKPT> vào nhật ký sau đó làm gọn nhật ký, như sau:

Undo log:

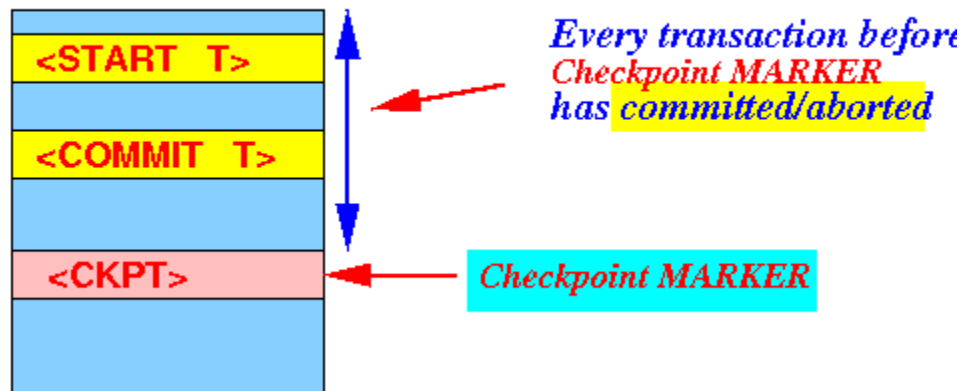
```
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<T2, C, 15>
<T1, D, 20>
<COMMIT T1>
<COMMIT T2>
```

<CKPT> ----- Useful "boundary"

```
<START T3>
<T3, E, 25>
<T3, F, 30>
```

Nếu muốn làm gọn undo log, chúng ta có thể xóa tất cả dòng nhật ký trước dòng <CKPT>

*Undo log:*



#### 4.6.2. Thao tác khôi phục với checkpointing

Sự khác biệt chính so với khi chưa có checkpoint đó là chúng ta không phải quét toàn bộ tập tin undo log. Quá trình quét (lùi lại) có thể dừng lại khi tìm thấy dòng <CKPT>

*Thuật toán khôi phục trên undo log sử dụng quiescent checkpointing*

```

/* =====
Step 1: identify the uncommitted transactions
===== */
Scan the undo log backwards until first <CKPT> record:
{
    identify the committed
    identify the uncommitted/aborted transactions
}
/* =====
Step 2: undo the uncommitted transactions
===== */
Scan the undo log backwards until first <CKPT> record:
{
    For ( each < T, A, v > in log file )
    {
        if ( T is uncommitted )
        {
            Update A with the (before) value v;    // Undo the
action
        }
    }
}
/* =====
Step 3: mark the uncommitted transactions as failed....
===== */
For ( each T that is uncommitted ) do
{
    Write <ABORT T> to log;

```



```
}
```

### Flush-Log

## 4.7. Thuật toán sử dụng Non-Quiescent checkpoint (đối với undo log)

Đây là một kỹ thuật phức tạp hơn, thực hiện checkpointing mà không bắt DBMS ngừng thực thi các giao tác mới.

### 4.7.1. Các bước đặt checkpoint

1. Write a start checkpoint log record:  
`<START CKPT(T1, T2, ..., Tk)>` to log file  
where T<sub>1</sub>, T<sub>2</sub>, ..., T<sub>k</sub> are the currently active transactions
2. Flush-Log (optional)
3. Wait until all of T<sub>1</sub>, T<sub>2</sub>, ..., T<sub>k</sub> to commit or abort.  
(DBMS can accept new transactions !!!)
4. When all T<sub>1</sub>, T<sub>2</sub>, ..., T<sub>k</sub> have completed:  
Write  
`<END CKPT>` to log file
5. Flush-Log (essential to keep the log file short)

Ví dụ:

T1 và T2 đang hoạt động:

```
Undo log:
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
```

Bây giờ chúng ta thực hiện đặt một Non-Quiescent checkpoint.

Ghi vào nhật ký dòng sau:

```
<START CKPT (T1, T2)>
```

Ghi nhật ký xuống đĩa cứng

Chờ cho đến khi T1 và T2 commit hoặc abort

Ghi dòng sau vào nhật ký

```
<END CKPT>
```

Ghi nhật ký xuống đĩa cứng lần nữa.

```
Undo log:
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT(T1, T2)> // Flush Log
<T2, C, 15>
<START T3> === New transactions can start !!
<T1, D, 20>
<COMMIT T1> ===== T1 done
<T3, E, 25>
<COMMIT T2> ===== T2 done
```

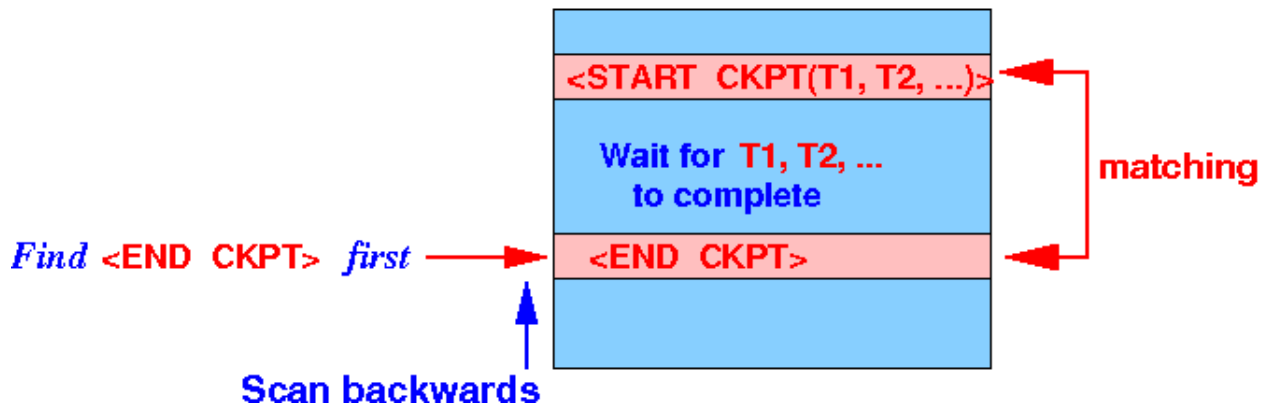
**<END CKPT> // Flush Log**  
**<T<sub>3</sub>, F, 30>**

#### 4.7.2. Khôi phục cơ sở dữ liệu sử dụng nonquiescent checkpointing

Khi quét tập tin nhật ký theo chiều ngược từ cuối lên đầu, có thể xảy ra một trong 2 khả năng sau:

Bạn tìm thấy bản ghi nhật ký <END CKPT> trước:

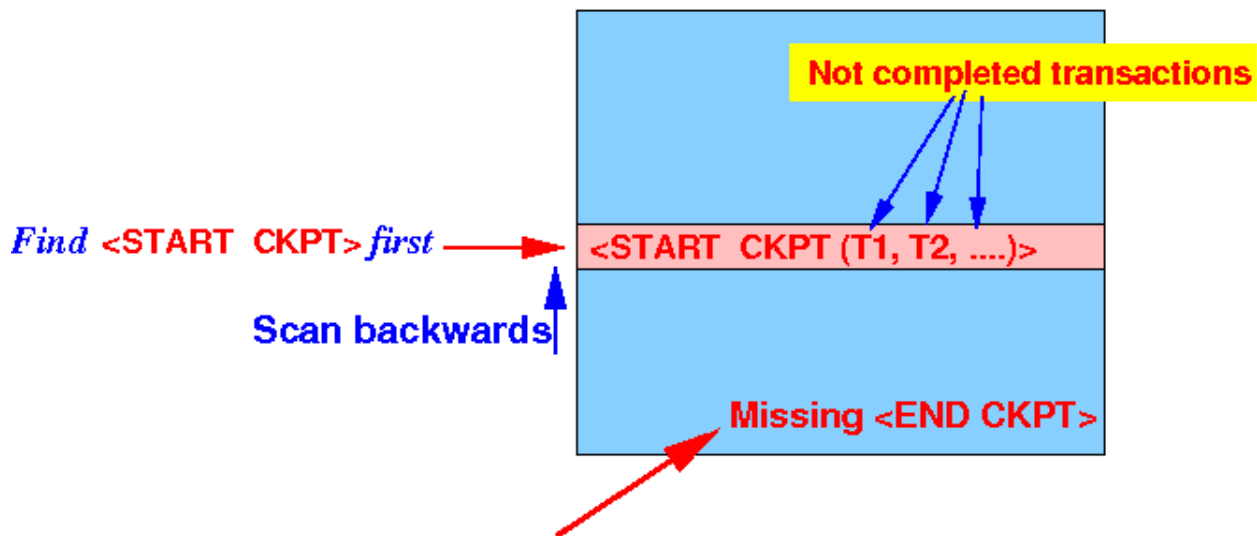
*Undo log:*



Đây là trường hợp khi, hoạt động đặt <END CKPT> (cuối cùng) đã hoàn tất thành công.

Bạn tìm thấy bản ghi nhật ký <START CKPT (T1, T2, ..., Tk)> trước tiên:

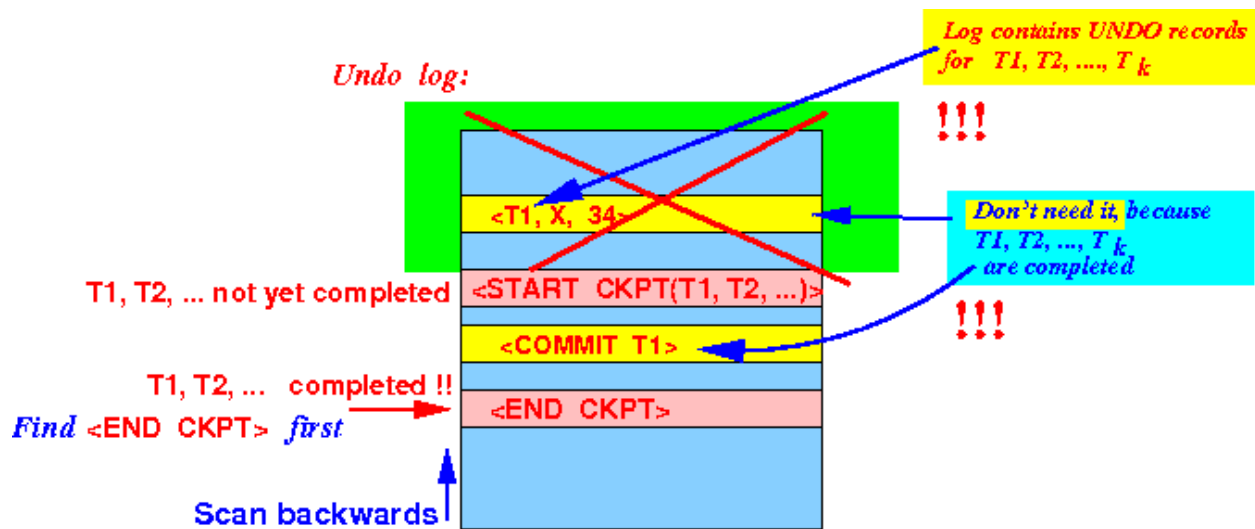
*Undo log:*



Trường hợp này xảy ra khi hệ thống đã gặp sự cố trong lúc đợi kết thúc giao tác để đặt <END CKPT>!!!!

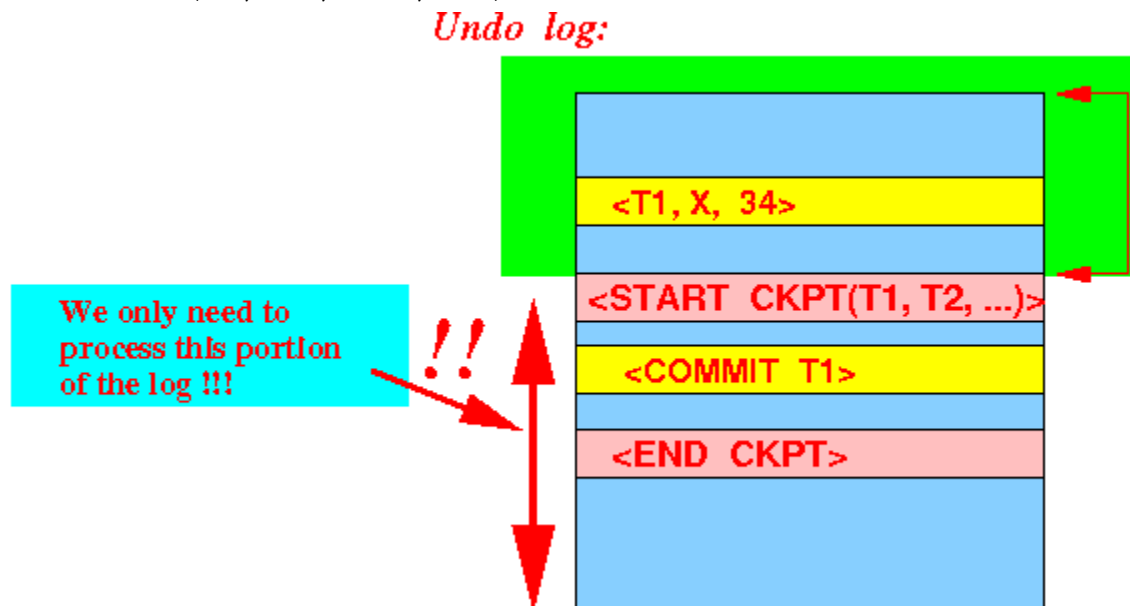
##### **Đối với trường hợp 1**

Chúng ta biết (chắc chắn) rằng tất cả các giao tác T1, T2, ..., Tk đã hoàn tất. Do đó, phần của undo log trước dòng nhật ký <START CKPT ...> là không cần thiết:



Cách khôi phục:

Hệ quản trị phải hoàn tác tất cả các giao tác chưa được commit bắt đầu sau dòng  
`<START CKPT (T1, T2, ..., Tk)>`



Thuật toán khôi phục cho trường hợp 1

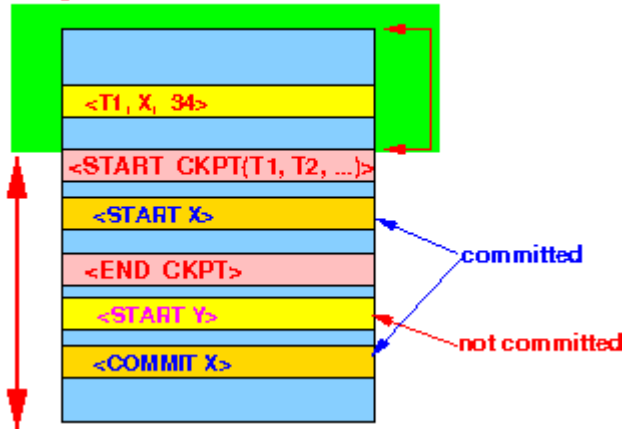
```

/* =====
Step 1: scan the appropriate log portion
       and identify the uncommitted transactions
===== */
Scan the undo log backwards until first <START CKPT(T1, T2,
..., Tk) record:
{
    identify the committed transactions

```

identify the uncommitted/aborted transactions

Undo log:



}

/\* =====

Step 2: undo the uncommitted transactions

===== \*/

Scan the undo log backwards until first <START CKPT(T<sub>1</sub>, T<sub>2</sub>, ..., T<sub>k</sub>)> record:

{

For ( each < T, A, v > in undo log ) do

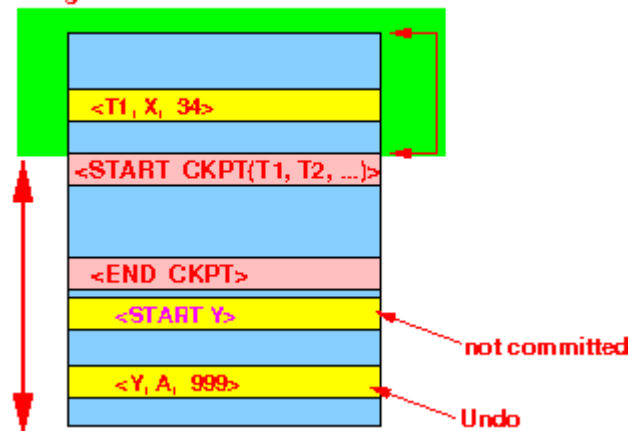
{

if ( T is uncommitted )

{

Update A with the (before) value v; // Undo the action !!!

Undo log:



}

}

/\* =====

Step 3: mark the uncommitted transactions as failed....

=====

\*/

For ( each T that is uncommitted ) do

```

{
    Write <ABORT T> to log;
}
Flush-Log

```

Ví dụ: Giả sử hệ thống bị hỏng sau khi hoàn thành một checkpoint

```

Undo log:
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT (T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>          ===== T1 done
<T3, E, 25>
<COMMIT T2>          ===== T2 done
<END CKPT>           // Flush Log
<T3, F, 30>
+++++++ System crashes

```

Tiến hành quét lùi và tìm thấy bản ghi <END CKPT> trước:

```

Undo log:
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT (T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>          ===== T1 done
<T3, E, 25>
<COMMIT T2>          ===== T2 done
<END CKPT>           // Flush Log
<T3, F, 30>
+++++++ System crashes

```

Tiếp tục quét ngược lên tới bản ghi <START CKPT (...)> tương ứng

```

Undo log:
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT (T1, T2)>  ++++++ Stop scanning here
<T2, C, 15>
<START T3>          ***** T3 uncommitted *****
<T1, D, 20>

```

```

<COMMIT T1>          ===== T1 done
<T3, E, 25>
<COMMIT T2>          ===== T2 done
<END CKPT>           // Flush Log
<T3, F, 30>
+++++++ System crashes

```

Hệ thống tìm thấy các giao tác có thể có là:

- T<sub>1</sub>, T<sub>2</sub> (Chúng nằm trong bản ghi <START CKPT (T<sub>1</sub>, T<sub>2</sub>)>)
- T<sub>3</sub> (Bản ghi nhật ký <START T<sub>3</sub>>)

Trong số những giao tác có thể có, những giao tác sau đã hoàn thành:

- T<sub>1</sub>, T<sub>2</sub> (Tìm thấy bản ghi <COMMIT T<sub>i</sub>> !!!)

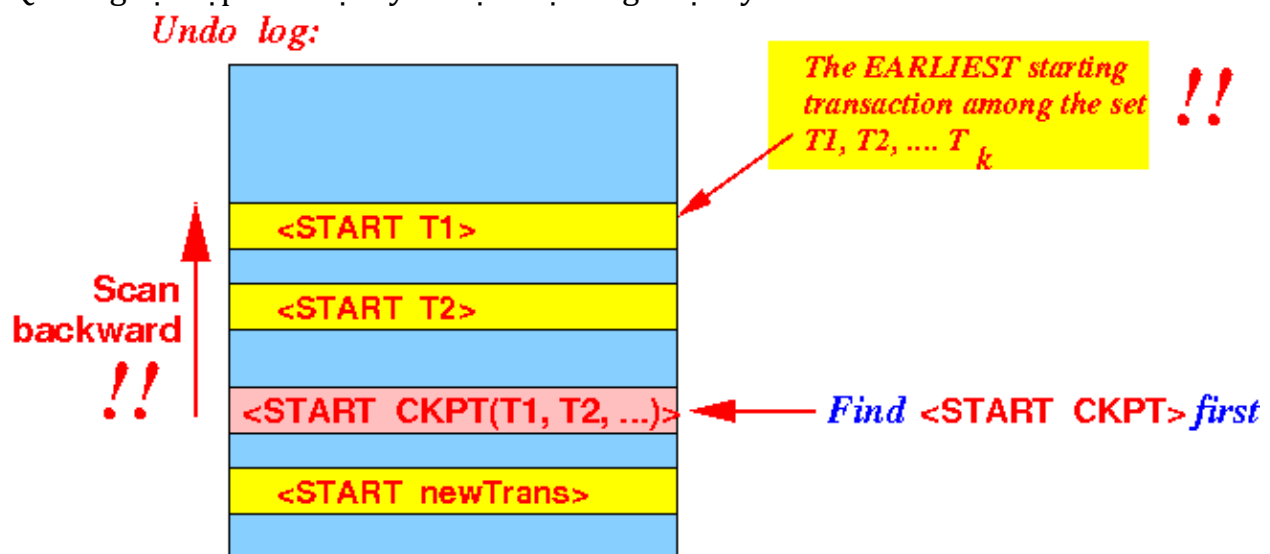
Vì thế, hệ thống phải hoàn tác giao tác T<sub>3</sub> (chưa hoàn thành)

### Đối với trường hợp 2

Chúng ta biết (chắc chắn) rằng các giao tác duy nhất chưa hoàn thành ở đầu checkpoint là T<sub>1</sub>, T<sub>2</sub>, ..., T<sub>k</sub>

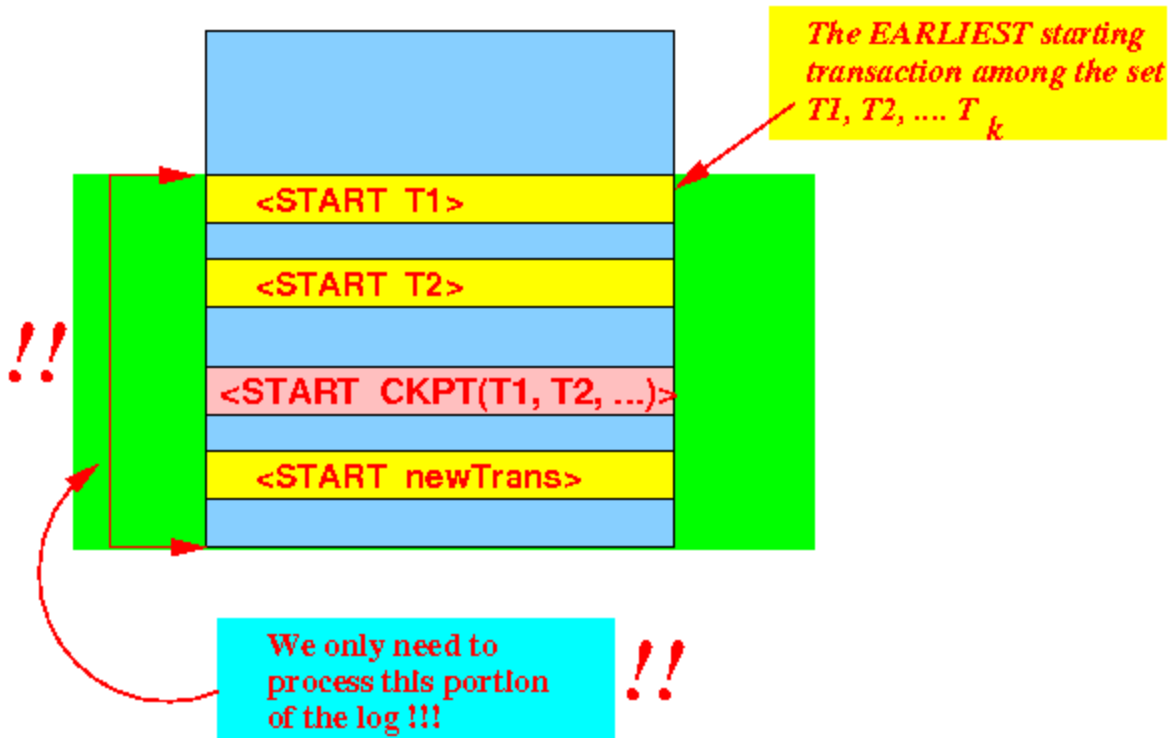
Chúng ta cần tra lại trong tập tin nhật ký để tìm tất cả các giao tác chưa hoàn thành.

Quét ngược tập tin nhật ký và định vị dòng nhật ký <START T<sub>i</sub>> sớm nhất



Hệ thống sẽ xác định được phần nào của undo log là cần thiết (= chứa thông tin) để phục hồi

*Undo log:*



### Thuật toán khôi phục cho trường hợp 2

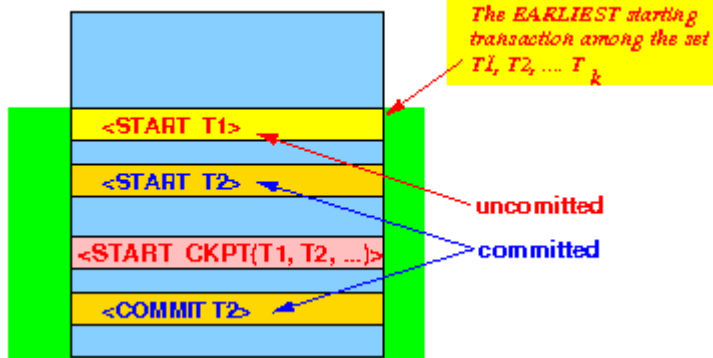
```

/* =====
Step 1: scan the appropriate log portion
      and identify the uncommitted transactions
===== */
Scan the undo log backwards
      until we found all <START T1>, <START T2>, ..., <START Tk>
records:
{

```

- identify the committed
- identify the uncommitted/aborted transactions

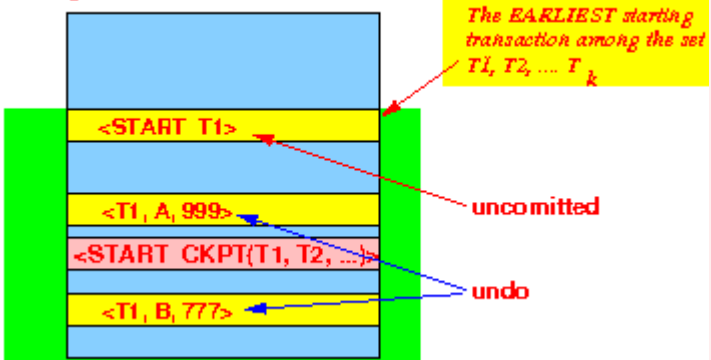
**Undo log:**



}

```
/* =====  
Step 2: undo the uncommitted transactions
```

```

===== */
Scan the undo log backwards
until we found the earliest uncommitted <START Tk> records:
{
  For ( each < T, A, v > in undo log ) do
  {
    if ( T is uncommitted )
    {
      Update A with the (before) value v;    // Undo the
action !!!
      Undo log:
      
    }
  }
}
}
/* =====
Step 3: mark the uncommitted transactions as failed....
=====
*/
For ( each T that is uncommitted ) do
{
  Write <ABORT T> to log;
}
Flush-Log

```

Ví dụ: Giả sử hệ thống bị hỏng trong quá trình hoạt động của checkpoint:

```

Undo log:
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT (T1, T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>      ===== T1 done
<T3, E, 25>
+++++++ System crashes

```

Tiến hành quét lùi lại và tìm thấy dòng <START CKPT (T<sub>1</sub>, T<sub>2</sub>)> trước:



Undo log:

```
<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT (T1, T2)>      ++++ Find <START CKPT> first
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>                ===== T1 done
<T3, E, 25>
+++++ System crashes
```

Tiếp tục quét ngược để tìm tất cả các bản ghi <START T1> và <START T2>

Undo log:

```
<START T1>                ++++ Stop scanning here
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT (T1, T2)>      ++++ Find <START CKPT> first
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>                ===== T1 done
<T3, E, 25>
+++++ System crashes
```

Hệ thống tìm thấy các giao tác có thể có là:

- T1, T2 (Chúng nằm trong bản ghi <START CKPT (T1, T2)>)
- T3 (Bản ghi nhật ký <START T3>)

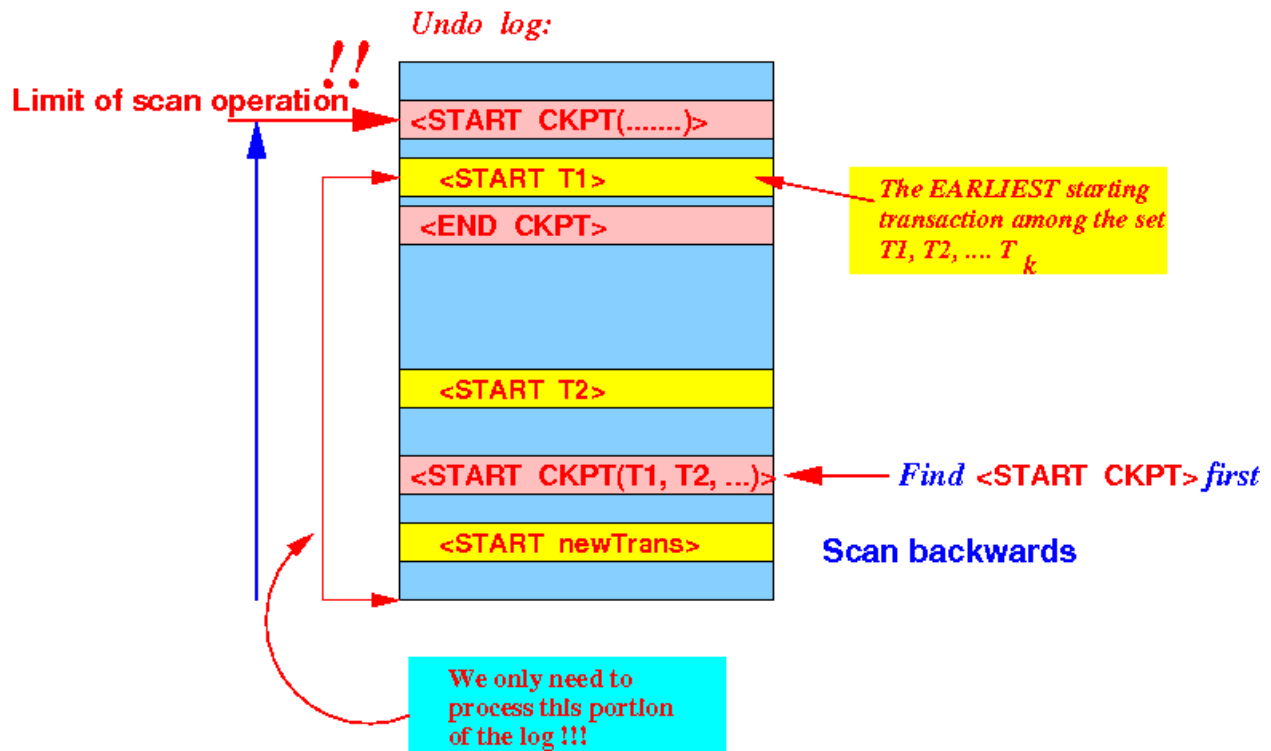
Trong số những giao tác có thể có, những giao tác sau đã hoàn thành:

- T1 (Tìm thấy bản ghi <COMMIT T1> !!!)

Vì thế, hệ thống phải hoàn tác giao tác T2 và T3 (chưa hoàn thành)

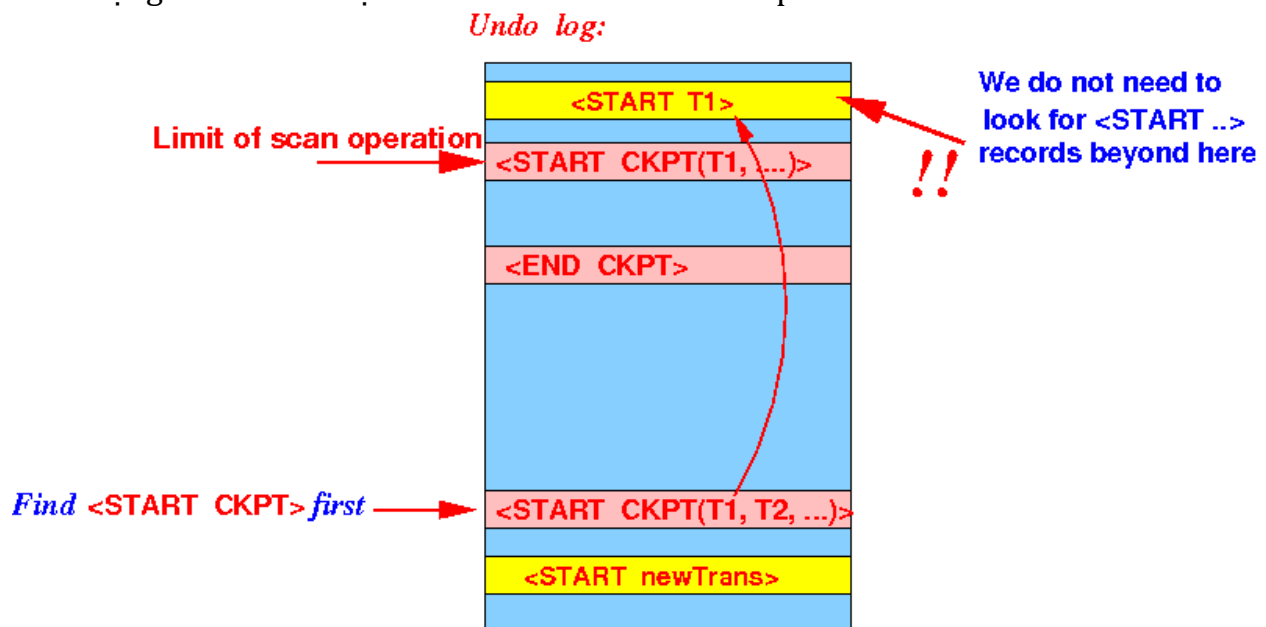
Chúng ta cần quét bao xa để tìm tất cả các dòng <START CPT>?

Điểm quay trở lại xa nhất mà chúng ta cần quét tập tin nhật ký để tìm tất cả các bản ghi <START T1>, <START T2>, ..., <START Tk> đó chính là dòng <START CKPT> liền trước đó.

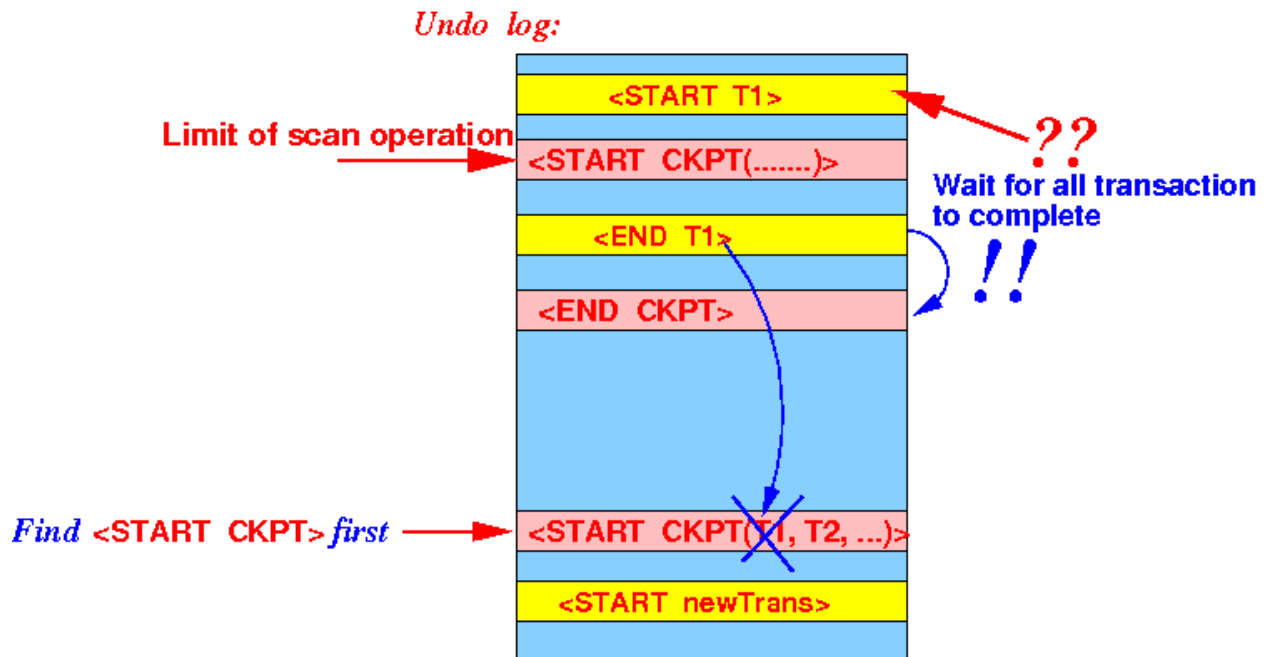


Giải thích:

Nếu một giao tác T1 được bắt đầu trước điểm checkpoint trước đó:



<COMMIT T1> phải được viết trước dòng <END CKPT> (theo quy ước về checkpoint):



Giao tác  $T_i$  phải hoàn thành !!!

## 5. Redo Logging

### 5.1. Nhược điểm của Undo log

Nhược điểm của undo log chính là hệ quản trị cơ sở dữ liệu buộc phải ghi tất cả các cập nhật trên dữ liệu vào đĩa cứng khi một giao tác commit. Các hoạt động ghi đĩa bắt buộc này tạo ra nhiều thao tác truy cập đĩa đồng thời. Điều này có thể tạo ra một nút thắt cổ chai về hiệu suất.

Cách tránh tình trạng nút thắt cổ chai chính là việc viết các phần tử dữ liệu đã cập nhật vào đĩa tuân theo sự sắp xếp của hệ quản trị cơ sở dữ liệu chứ không phụ thuộc vào thời điểm commit nữa.

### 5.2. Định nghĩa

Phương thức redo logging không yêu cầu các thao tác ghi đĩa khi một giao tác commit, mà redo log cho phép hệ quản trị cơ sở dữ liệu ghi dữ liệu cập nhật vào lúc thuận tiện. Chính vì thế phương pháp này còn được gọi là phương pháp trì hoãn sửa đổi (deferred modification).

Hệ quản trị không ghi lại giá trị cũ, nhưng sẽ ghi lại giá trị mới. Thay vì hoàn tác các hành động, hệ quản trị sẽ thực hiện lại chúng. Dòng nhật ký **<Ti, commit>** có thể được ghi sớm hơn so với sửa đổi dữ liệu thực tế được ghi xuống đĩa cứng. Nhưng ngay sau khi dữ liệu sửa đổi được chép lên đĩa, hệ quản trị sẽ ghi xuống dòng nhật ký **<Ti, end>**.

Ví dụ

Transaction T1	Log	Comment
----------------	-----	---------

	$\langle T1, \text{start} \rangle$	Khi giao tác bắt đầu
$\text{read}(A, t);$ $t \leftarrow t \times 2;$		
$\text{write}(A, t)$	$\langle T1, A, 16 \rangle$	Giá trị mới của A là 16
$\text{read}(B, t);$ $t \leftarrow t \times 2;$		
$\text{write}(B, t)$	$\langle T1, B, 16 \rangle$	Giá trị mới của B là 16
	$\langle T1, \text{commit} \rangle$	Ghi nhận trên nhật ký sẽ sớm hơn so với thay đổi dữ liệu thực tế
$\text{output}(A)$		
$\text{output}(B)$		Tất cả những thay đổi được chép lên đĩa cứng ngay lúc này
	$\langle T1, \text{end} \rangle$	Giao tác kết thúc

### ***So sánh giữa hai phương pháp undo và redo logging:***

Đối với undo log

- ✓ Được thiết kế để quay ngược (hủy) tác dụng của các giao tác chưa hoàn thành.
- ✓ Quy trình khôi phục sẽ bỏ qua các giao tác đã hoàn thành.

Đối với redo log

- ✓ Được thiết kế để làm lại (lặp lại) tác dụng của các giao tác đã hoàn thành.
- ✓ Thủ tục phục hồi sẽ bỏ qua các giao tác chưa hoàn thành.

### **5.3. Quy tắc**

#### ***Quy tắc ghi Redo log***

- ✓ Đối với mọi hành động hệ thống sẽ ghi nhận một dòng nhật ký với giá trị **MỚI**.
- ✓ Trước khi một dữ liệu **X** được ghi vào đĩa cứng, tất cả các dòng nhật ký về các giao tác **Ti** đã thực hiện sửa đổi **X** (bao gồm cả  $\langle \mathbf{Ti}, \text{commit} \rangle$ ) phải nằm trên đĩa cứng.
- ✓ Ghi nhật ký lên đĩa tại thời điểm commit
- ✓ Ghi  $\langle \mathbf{Ti}, \text{end} \rangle$  khi tất cả các dữ liệu sửa đổi đều đã ghi xuống đĩa.

Lưu ý rằng chúng ta không thể đi đến trạng thái trước đó với cách tiếp cận này: không quay lui. Nếu muốn làm điều đó, chúng ta phải áp dụng phương pháp Undo logging hoặc Undo/Redo logging.

#### ***Quy tắc khôi phục Redo log***

$\langle \mathbf{Ti}, \text{commit} \rangle$  cho người dùng biết rằng giao tác đã được thực thi chính xác. Ngay cả khi một số lỗi xảy ra, hệ quản trị phải đảm bảo rằng trạng thái của cơ sở dữ liệu là trạng thái mà người dùng mong đợi sau khi giao tác xảy ra.

$\langle \mathbf{Ti}, \text{end} \rangle$  thông báo rằng kết quả đã ở trên đĩa cứng - không cần phải làm lại bất cứ điều gì.

Redo(log **L**)

Let **S** be set of all transactions **T<sub>i</sub>** with  $\langle T_i, \text{commit} \rangle \in L$  but without  $\langle T_i, \text{end} \rangle$

for each **T<sub>i</sub>**  $\in S$  and for each  $\langle T_i, \text{commit} \rangle \in L$  in forward order (earliest  $\rightarrow$  latest)

write(**X**,v)

output(**X**) (write and ensure the modifications appear on disk)

## 5.4. Thuật toán sử dụng Non-Quiescent checkpoint (đối với redo log)

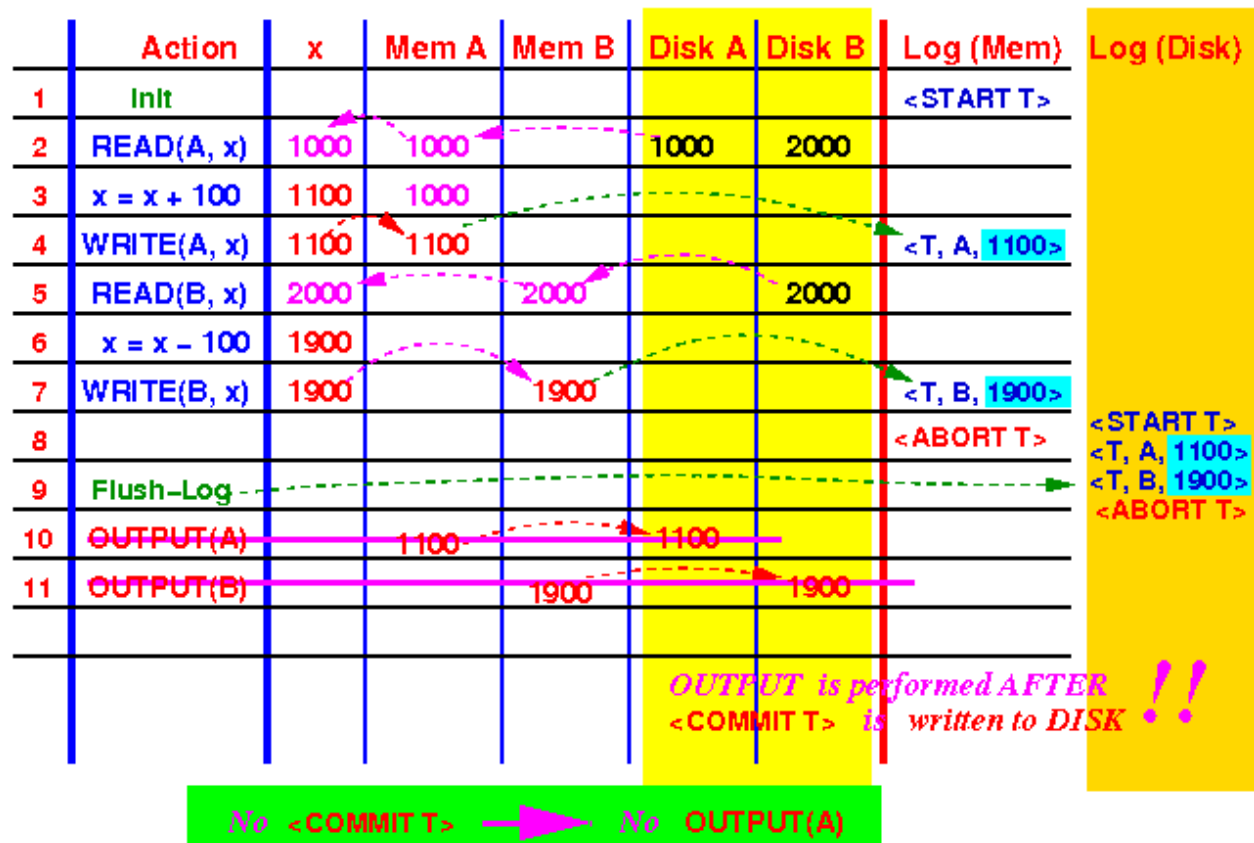
### 5.4.1. Các bước cài đặt checkpoint

Chúng ta dùng checkpoint với mong muốn rút ngắn nhật ký bằng cách “xóa” các giao tác đã hoàn tất (giao tác đã commit và giao tác bị abort).

Dưới đây là mô tả thực tế về redo log để giúp chúng ta hiểu về cách sử dụng checkpoint với redo log.

(1) Giao tác bị hủy (không thành công) sẽ không bao giờ thực hiện bất kỳ thao tác

OUTPUT () nào trong redo log:



Như vậy, chúng ta có thể bỏ qua (= loại bỏ) bản ghi của các giao tác bị hủy bỏ trong redo log mà không cần làm bất cứ điều gì.

(2) Các hoạt động OUTPUT () của các giao tác đã commit trong redo log có thể được trì hoãn:

	Action	x	Mem A	Mem B	Disk A	Disk B	Log (Mem)	Log (Disk)
1	Init						<START T>	
2	READ(A, x)	1000	1000		1000	2000		
3	x = x + 100	1100	1000					
4	WRITE(A, x)	1100	1100				<T, A, 1100>	
5	READ(B, x)	2000		2000		2000		
6	x = x - 100	1900						
7	WRITE(B, x)	1900		1900			<T, B, 1900>	
8							<COMMIT T>	
9	Flush-Log							<START T> <T, A, 1100> <T, B, 1900> <COMMIT T>
10								
11								
	OUTPUT(A) OUTPUT(B)		1100 1900		1100 1900			OUTPUT can be performed later !!!

**Việc trì hoãn thao tác OUTPUT (ghi dữ liệu lên đĩa cứng) là điểm mạnh (= lợi thế) của redo log so với undo log!!!**

Tóm lại, chúng ta có thể loại bỏ (= xóa) các bản ghi nhật ký thuộc về các giao tác không commit. Để loại bỏ các bản ghi redo log (một cách logic) thuộc về các giao tác đã commit, đầu tiên, chúng ta phải kết hợp tất cả các dòng nhật ký thực hiện cập nhật dữ liệu  $\langle T, x, v \rangle$  được thực hiện bởi các giao tác đã commit lên đĩa. Bởi vì chúng ta không thể redo các thao tác cập nhật này sau khi xóa chúng.

### Thuật toán đặt checkpoint

1. Write a start checkpoint log record:  
 $\langle \text{START CKPT}(T_1, T_2, \dots, T_k) \rangle$   
 where  $T_1, T_2, \dots, T_k$  are the currently active transactions
2. Flush-Log (This will ALSO write ALL  $\langle T, x, v, w \rangle$  records to disk !!!)
3. Incorporate updates from committed transactions:  
 Output all DB elements that were updated by committed transactions to disk
4. Write  $\langle \text{END CKPT} \rangle$  to log
5. Flush-Log

Ví dụ: khởi tạo giao tác T1 và T2

Redo log:  
 $\langle \text{START } T_1 \rangle$   
 $\langle T_1, A, 5 \rangle$

```
<START T2>
<COMMIT T1>
<T2, B, 10>
```

Bây giờ chúng ta thực hiện đặt một Non-Quiescent checkpoint.

Ghi vào nhật ký dòng sau:

```
<START CKPT>
```

Redo log:

```
<START T1>
<T1, A, 5>
<START T2>
<COMMIT T1>
<T2, B, 10>
<START CKPT(T2)>
```

Ghi dữ liệu A (do giao tác T<sub>1</sub> đã commit) xuống đĩa cứng.

Redo log:

```
<START T1>
<T1, A, 5> <----- update by a committed transaction !!!>
<START T2>
<COMMIT T1>
<T2, B, 10>
<START CKPT(T2)>
....
```

```
-----> Write (A,5) to disk
```

Ghi dòng sau vào nhật ký

```
<END CKPT>
```

Redo log:

```
<START T1>
<T1, A, 5>
<START T2>
<COMMIT T1>
<T2, B, 10>
<START CKPT(T2)>
....
```

```
-----> Write (A,5) to disk
```

```
....
```

```
<END CKPT>
```

Ghi nhật ký xuống đĩa cứng lần nữa.

Redo log lúc này có thể có nội dung như sau:

Redo log:

```
<START T1>
<T1, A, 5>
<START T2>
<COMMIT T1> ===== T1 done
<T2, B, 10>
=====
```

```

<START CKPT (T2)>
<T2, C, 15>      <--+ Between here:
<START T3>        | ***** OUTPUT (A, 5)
<T3, D, 20>      <--+
<END CKPT>        // Flush Log
<COMMIT T2>       ===== T2 done
<COMMIT T3>       ===== T3 done
(OUTPUT for T2 and T3 happens later)

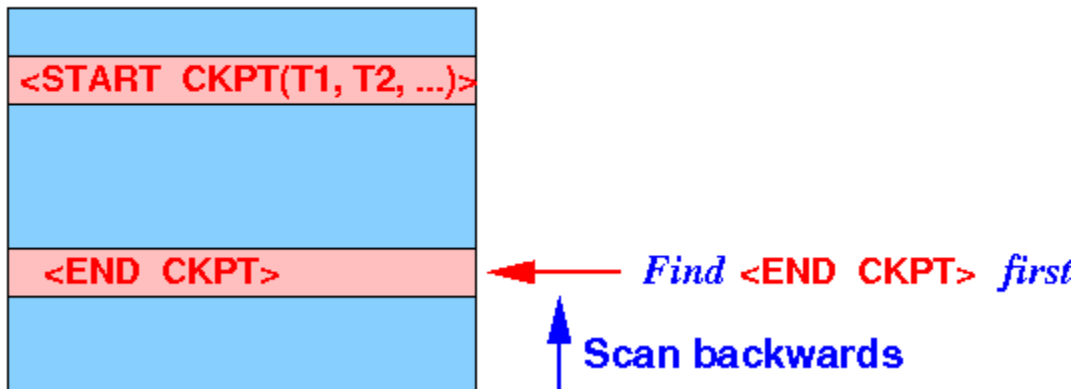
```

#### 5.4.2. Khôi phục cơ sở dữ liệu sử dụng nonquiescent checkpointing

Khi quét tập tin nhật ký theo chiều ngược từ cuối lên đầu, có thể xảy ra một trong 2 khả năng sau:

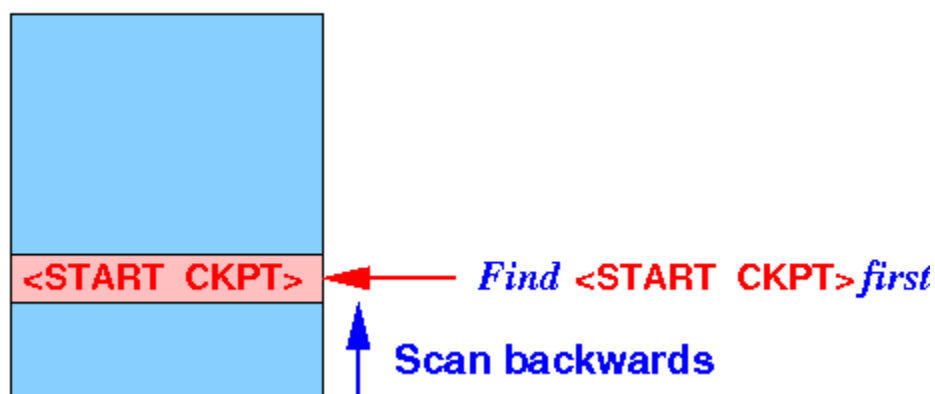
Bạn tìm thấy bản ghi nhật ký <END CKPT> trước:

*Redo log:*



Đây là trường hợp khi, hoạt động đặt <END CKPT> (cuối cùng) đã hoàn tất thành công. Bạn tìm thấy bản ghi nhật ký <START CKPT (T<sub>1</sub>, T<sub>2</sub>, ..., T<sub>k</sub>)> trước tiên:

*Redo log:*



Trường hợp này xảy ra khi hệ thống đã gặp sự cố trong lúc đợi kết thúc giao tác để đặt <END CKPT>!!!!

Đối với thuật toán khôi phục trong redo log, hệ quản trị loại bỏ các cập nhật được thực hiện bởi các giao tác chưa commit. Và bắt buộc phải làm lại các cập nhật được thực



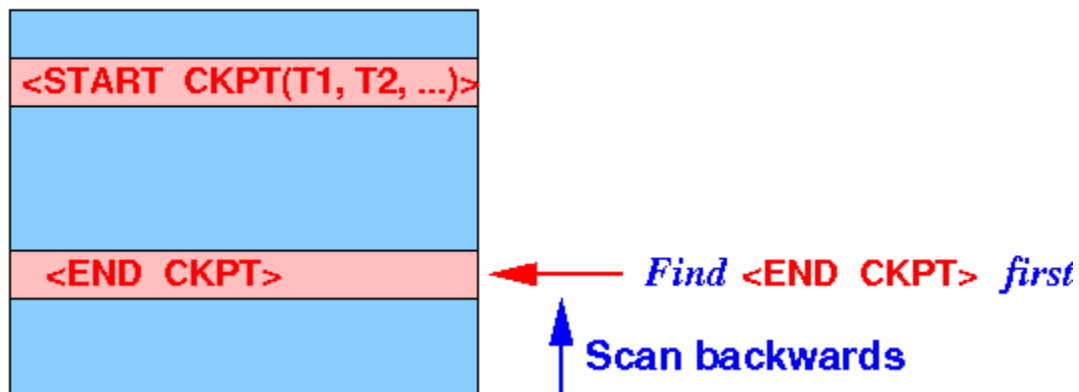
hiện bởi những giao tác đã commit. Sự khác biệt đối với sự cố xảy ra trong trường hợp và trường hợp 2 là việc xác định phần nào của redo log chúng ta cần tìm kiếm để lấy ra tất cả các giao tác đã commit.

### Khôi phục dữ liệu dựa vào redo log trong trường hợp 1

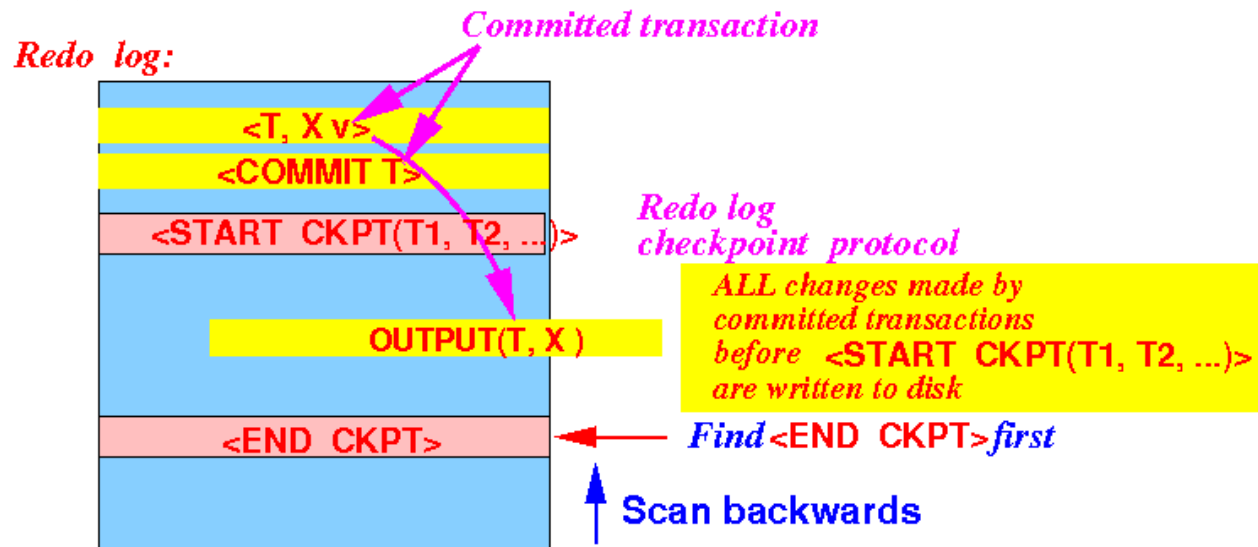
Trong trường hợp 1, muốn khôi phục chúng ta phải tìm thấy các bản ghi redo log cần thiết.

Với việc tìm thấy dòng <END CKPT> trước tiên

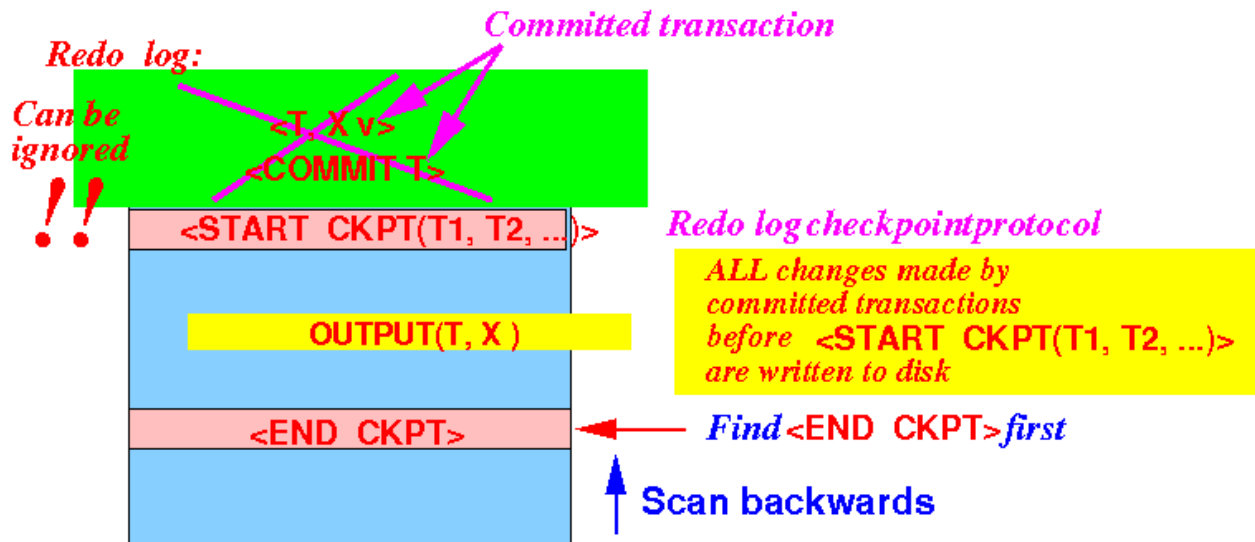
*Redo log:*



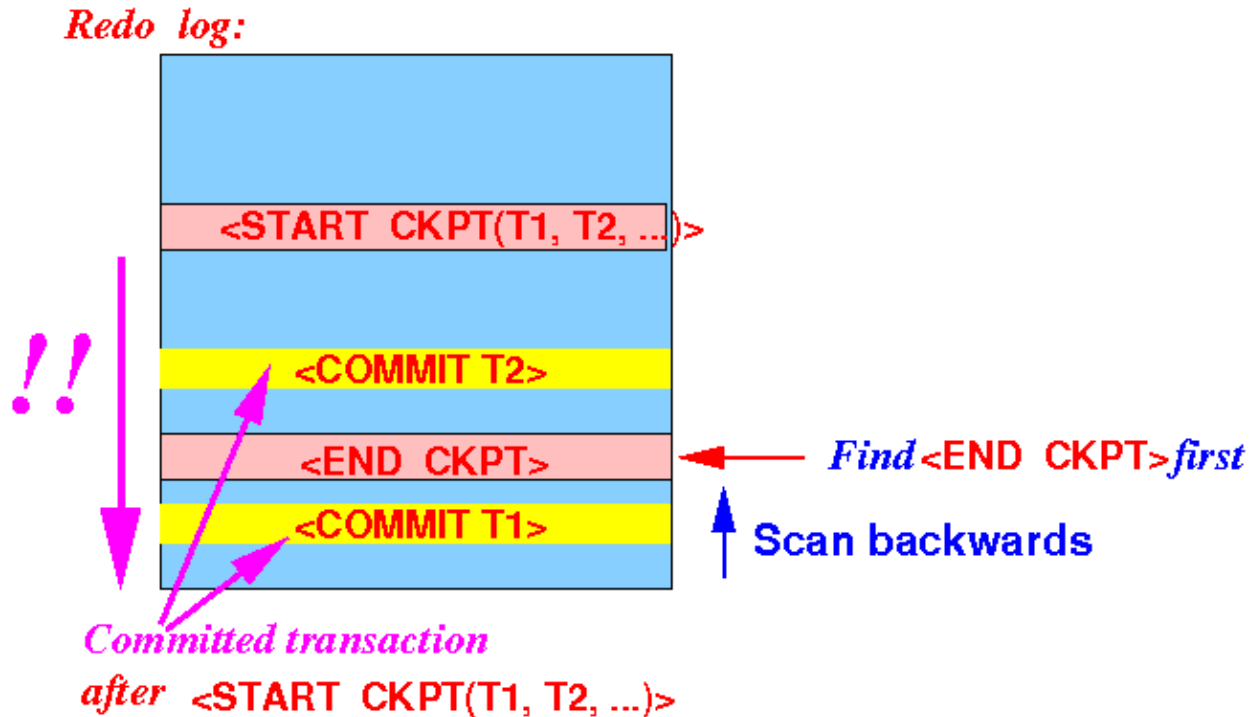
Bằng thao tác đặt checkpoint, chúng ta biết được tất cả các thay đổi được thực hiện bởi giao tác T đã commit trước <START CKPT (•)> đã được ghi vào đĩa:



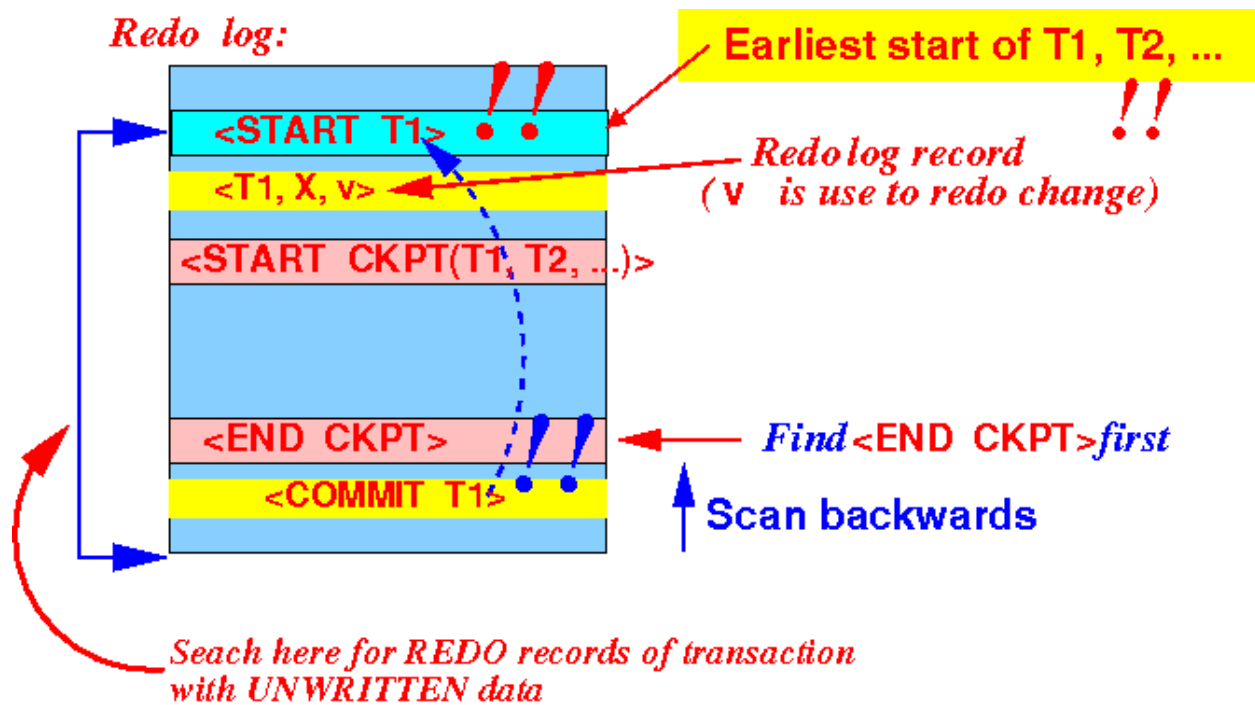
Do đó, trong quá trình khôi phục, chúng ta không cần phải làm lại các thay đổi được thực hiện bởi các giao tác đã commit trong phần nhật ký này.



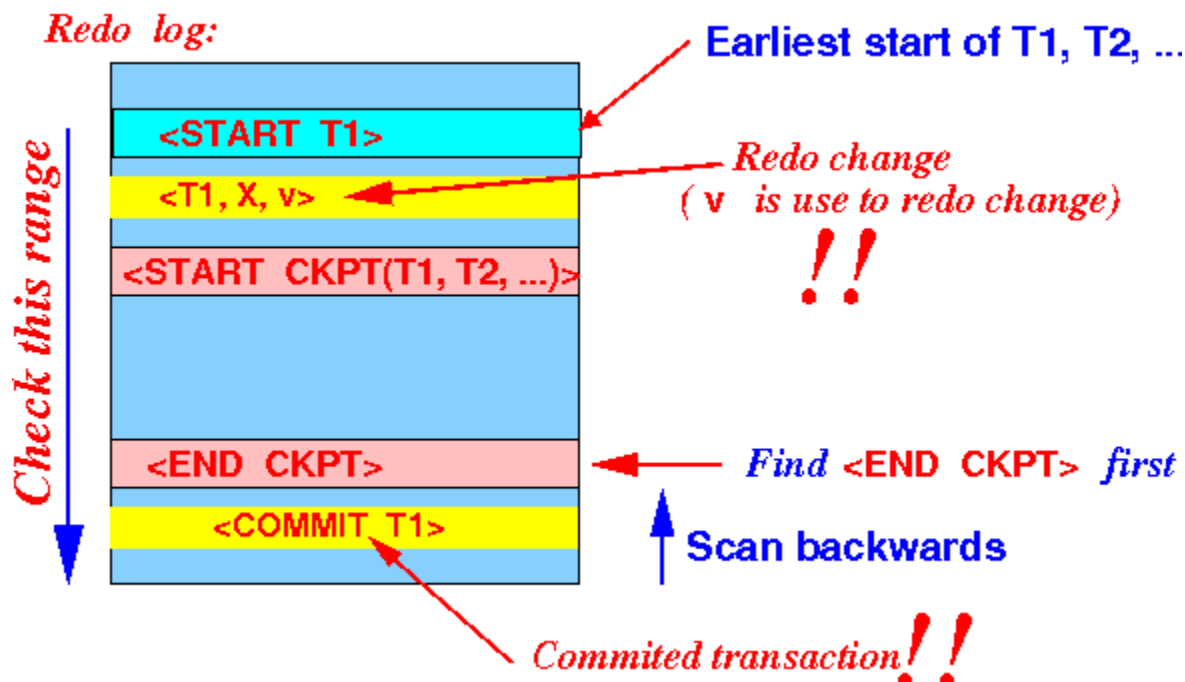
Chúng ta (vẫn) phải làm lại các bản cập nhật được thực hiện bởi các giao tác đã commit sau bản ghi  $\langle \text{START CKPT} \rangle$ :



Phần của redo log mà chúng ta cần tìm kiếm các bản ghi nhật ký là:



Tóm lại, cách khôi phục bằng redo log trong trường hợp 1 là tìm bản ghi nhật ký `<START Ti>` sớm nhất trong đó  $T_i$  nằm trong danh sách giao tác của checkpoint. Làm lại các cập nhật được thực hiện bởi những giao tác đã commit sau dòng `<START CKPT>`



Thuật toán khôi phục dựa theo redo log cho trường hợp 1:

```
/* =====
Step 1: identify the committed transactions
===== */
```

```

Scan the redo log backwards until the <START CKPT(T1, T2, ...,
Tk) record:
{
    identify the committed
    identify the uncommitted/aborted transactions
}
/* =====
Step 2: redo the committed transactions
===== */
Scan the redo log backwards
until we found all <START T1>, <START T2>, ..., <START Tk>
records
Starting from this point, scan the redo log forewards
{
    For ( each < T, A, v > )
    {
        if ( T is committed )
        {
            Update A with the (after) value v;    // Redo the
action !!!
        }
    }
}
/* =====
Step 3: mark the uncommitted transactions as failed....
===== */
For ( each T that is uncommitted ) do
{
    Write <ABORT T> to log;
}
Flush-Log

```

Ví dụ khôi phục dữ liệu dựa vào redo log trong trường hợp 1  
 Giả sử hệ thống gặp sự cố sau khi ghi cả hai bản ghi <COMMIT>:

```

Redo log:
<START T1>
<T1, A, 5>
<START T2>
    <COMMIT T1>      ==== T1 done
<T2, B, 10>
=====
<START CKPT(T2)>
<T2, C, 15>      <--+ Between here:
<START T3>        | **** OUTPUT(A)
<T3, D, 20>      <--+
<END CKPT>        // Flush Log
<COMMIT T2>      ==== T2 done

```

```

    <COMMIT T3>          ===== T3 done
    ++++++ System
crashed....
    (OUTPUT for T2 and T3 happens later)

```

Tiến hành quét lùi và tìm thấy bản ghi <END CKPT> trước:

```

Redo log:
    <START T1>
    <T1, A, 5>
    <START T2>
    <COMMIT T1>          ===== T1 done
    <T2, B, 10>
    =====
    <START CKPT (T2)>
    <T2, C, 15>          <--+ Between here:
    <START T3>            | **** OUTPUT (A)
    <T3, D, 20>          <--+
    <END CKPT>           // Flush Log
    <COMMIT T2>          ===== T2 done
    <COMMIT T3>          ===== T3 done
    ++++++ System
crashed....
    (OUTPUT for T2 and T3 happens later)

```

Tiếp tục quét ngược lên tới <START CKPT (T2)> để tìm tất cả dòng commit:

```

Redo log:
    <START T1>
    <T1, A, 5>
    <START T2>
    <COMMIT T1>          ===== T1 done
    <T2, B, 10>
    =====
    <START CKPT (T2)>
    <T2, C, 15>          <--+ Between here:
    <START T3>            | **** OUTPUT (A)
    <T3, D, 20>          <--+
    <END CKPT>           // Flush Log
    <COMMIT T2>          ===== T2 done
    <COMMIT T3>          ===== T3 done
    ++++++ System
crashed....
    (OUTPUT for T2 and T3 happens later)

```

Kết quả là hệ quản trị cần làm lại các giao tác T2 và T3.

Tiến hành quét ngược để tìm các bản ghi <START Tx> của tất cả các giao tác đã commit (T2, T3):

```

Redo log:
    <START T1>
    <T1, A, 5>

```

```

<START T2>                                     <----- Here !!!!!
  <COMMIT T1>                                ==== T1 done
<T2, B, 10>
=====
<START CKPT(T2)>
<T2, C, 15>      <--+ Between here:
<START T3>       | **** OUTPUT(A)
<T3, D, 20>      <--+
<END CKPT>       // Flush Log
<COMMIT T2>      ===== T2 done
<COMMIT T3>      ===== T3 done
+++++++ System
crashed....
  (OUTPUT for T2 and T3 happens later)

```

Từ dòng được đánh dấu bởi **Here**, quét từ trên xuống và làm lại tất cả các dòng của T2 và T3

```

Redo log:
  <START T1>
  <T1, A, 5>
  <START T2>                                     ***** Here !!!!!
    <COMMIT T1>                                ==== T1 done
  <T2, B, 10>
  =====
  <START CKPT(T2)>
  <T2, C, 15>      <--+ Between here:
  <START T3>       | **** OUTPUT(A)
  <T3, D, 20>      <--+
  <END CKPT>       // Flush Log
  <COMMIT T2>      ===== T2 done
  <COMMIT T3>      ===== T3 done
  ++++++ System
crashed....
  (OUTPUT for T2 and T3 happens later)

```

Ví dụ 2: (mất giao tác đã commit)

Giả sử hệ thống bị hỏng sau khi viết chỉ một bản ghi <COMMIT>

```

Redo log:
  <START T1>
  <T1, A, 5>
  <START T2>
    <COMMIT T1>                                ==== T1 done
  <T2, B, 10>
  =====
  <START CKPT(T2)>
  <T2, C, 15>      <--+ Between here:
  <START T3>       | **** OUTPUT(A)
  <T3, D, 20>      <--+

```

```

        <END CKPT>                // Flush Log
        <COMMIT T2>                ===== T2 done
        ++++++ System
crashed....
<COMMIT T3>                    ===== T3 is not comitted !!! <-----
----- *****
        (OUTPUT for T2 and T3 happens later)

```

Quét lùi lại và tìm thấy bản ghi <END CKPT> trước:

```

Redo log:
    <START T1>
    <T1, A, 5>
    <START T2>
        <COMMIT T1>                ===== T1 done
    <T2, B, 10>
    =====
    <START CKPT (T2)>
    <T2, C, 15>                    <--+ Between here:
    <START T3>                      | ***** OUTPUT (A)
    <T3, D, 20>                    <--+
    <END CKPT>                      // Flush Log
    <COMMIT T2>                      ===== T2 done
    ++++++ System
crashed....
<COMMIT T3>                    ===== T3 done
        (OUTPUT for T2 and T3 happens later)

```

Tiếp tục quét ngược lên tới <START CKPT (T2)> để tìm tất cả các dòng commit:

```

Redo log:
    <START T1>
    <T1, A, 5>
    <START T2>
        <COMMIT T1>                ===== T1 done
    <T2, B, 10>
    =====
    <START CKPT (T2)>
    <T2, C, 15>                    <--+ Between here:
    <START T3>                      | ***** OUTPUT (A)
    <T3, D, 20>                    <--+
    <END CKPT>                      // Flush Log
    <COMMIT T2>                      ===== T2 done
    ++++++ System
crashed....
<COMMIT T3>                    ===== T3 done
        (OUTPUT for T2 and T3 happens later)

```

Kết quả là chúng ta chỉ cần làm lại giao tác T2.

Chúng ta quét ngược để tìm các bản ghi <START Tx> của tất cả các giao tác đã commit (T2)

```

Redo log:
  <START T1>
  <T1, A, 5>
  <START T2> <----- Here !!!!!>
  <COMMIT T1> ===== T1 done
  <T2, B, 10>
  =====
  <START CKPT (T2)>
  <T2, C, 15> <--+ Between here:
  <START T3> | **** OUTPUT (A)
  <T3, D, 20> <--+
  <END CKPT> // Flush Log
  <COMMIT T2> ===== T2 done
  ++++++ System
crashed....
  <COMMIT T3> ===== T3 done
  (OUTPUT for T2 and T3 happens later)

```

Từ dòng được đánh dấu bởi **Here**, quét từ trên xuống và làm lại tất cả các dòng của T<sub>2</sub>

```

Redo log:
  <START T1>
  <T1, A, 5>
  <START T2> ***** Here !!!!!
  <COMMIT T1> ===== T1 done
  <T2, B, 10>
  =====
  <START CKPT (T2)>
  <T2, C, 15> <--+ Between here:
  <START T3> | **** OUTPUT (A)
  <T3, D, 20> <--+
  <END CKPT> // Flush Log
  <COMMIT T2> ===== T2 done
  <COMMIT T3> ===== T3 done
  ++++++ System
crashed....
  (OUTPUT for T2 and T3 happens later)

```

Lưu ý những thay đổi trong giao tác T<sub>3</sub> sẽ không được làm lại!

Cuối cùng hệ quản trị sẽ viết một dòng nhật ký là <ABORT T<sub>3</sub>>

Lưu ý: Giao tác "đã commit" (T<sub>3</sub>) có thể bị hủy nếu bản ghi <COMMIT> của nó không được ghi vào redo log! Trong redo log, giao tác chỉ được thực hiện nếu:

Log của giao tác đã commit hiện đã được ghi vào đĩa !!!

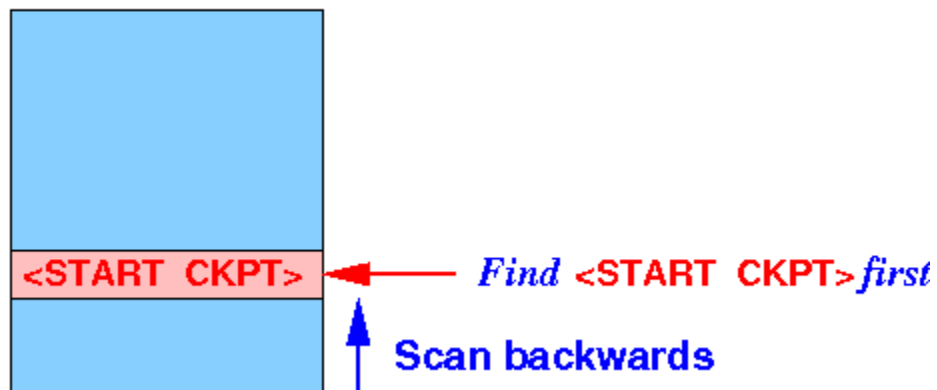
Để giảm thiểu hiện tượng giao tác "đã commit" bị mất này, redo log sẽ được xóa khi một dòng <COMMIT> được ghi.

**Khôi phục dữ liệu dựa vào redo log trong trường hợp 2**



Chúng ta tìm thấy dòng  $\langle \text{START CKPT } (T_1, T_2, \dots, T_k) \rangle$  đầu tiên

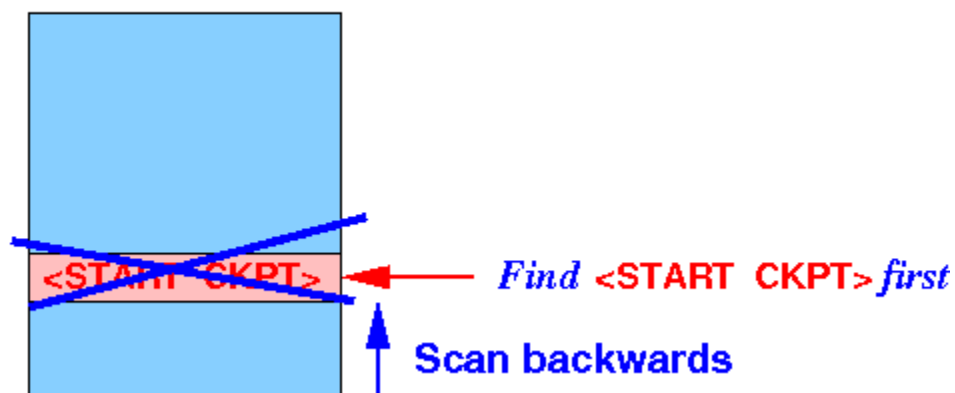
*Redo log:*



Trong quá trình checkpoint, hệ quản trị sẽ ghi các thay đổi được thực hiện bởi giao tác đã commit lên đĩa. Không có bản ghi  $\langle \text{END CKPT} \rangle$ , hệ quản trị sẽ không biết những dòng cập nhật nào đã được ghi vào đĩa (và cái nào chưa !!). Trường hợp xấu nhất là không có bản cập nhật nào được thực hiện bởi các giao tác đã commit được ghi vào đĩa !!!

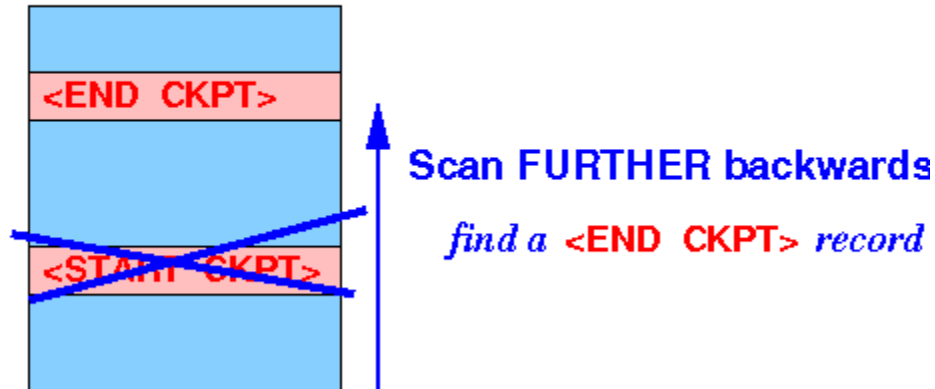
Trong trường hợp xấu nhất, hệ quản trị không thể có thêm thông tin gì từ dòng nhật ký  $\langle \text{START CKPT} \rangle$ , trường hợp này giống như không có bản ghi  $\langle \text{START CKPT} \rangle$

*Redo log:*



Tất cả những gì chúng ta có thể làm là rà soát và khôi phục lại toàn bộ redo log

*Redo log:*



Ví dụ: Giả sử hệ thống bị sự cố trước khi checkpoint kết thúc

```

Redo log:
<START T1>
<T1, A, 5>
<START T2>
  <COMMIT T1>          ===== T1 done
  <T2, B, 10>
  =====
  <START CKPT (T2)>
  <T2, C, 15>          <--+ Between here:
  <START T3>            | ***** OUTPUT (A)
  <T3, D, 20>          <--+
  ++++++ System
crashed....

  <END CKPT>          // Flush Log
  <COMMIT T2>          ===== T2 done
  <COMMIT T3>          ===== T3 done
  (OUTPUT for T2 and T3 happens later)

```

Chúng ta quét lùi và tìm thấy dòng <START CKPT> trước

```

Redo log:
<START T1>
<T1, A, 5>
<START T2>
  <COMMIT T1>          ===== T1 done
  <T2, B, 10>
  =====
  <START CKPT (T2)>          *****
  <T2, C, 15>          <--+ Between here:
  <START T3>            | ***** OUTPUT (A)
  <T3, D, 20>          <--+
  ++++++ System
crashed....

```

Tiếp tục tìm kiếm dòng <END CKPT> cuối cùng (hoặc tới đầu tập tin nhật ký):

Redo log:

```
<START T1> <----- HERE....
<T1, A, 5>
<START T2>
  <COMMIT T1> ==== T1 done
<T2, B, 10>
=====
<START CKPT (T2)> *****
<T2, C, 15> <--+ Between here:
<START T3> | **** OUTPUT (A)
<T3, D, 20> <--+
+++++++ System
crashed....
```

Chỉ có giao tác T<sub>1</sub> được commit. Như vậy, hệ quản trị chỉ làm lại các thay đổi được thực hiện bởi các giao tác T<sub>1</sub>.

Redo log:

```
<START T1> <----- HERE....
<T1, A, 5>
<START T2>
  <COMMIT T1> ==== T1 done
<T2, B, 10>
=====
<START CKPT (T2)> *****
<T2, C, 15> <--+ Between here:
<START T3> | **** OUTPUT (A)
<T3, D, 20> <--+
+++++++ System
crashed....
```

Chúng ta phải làm lại T<sub>1</sub> vì chưa biết chính xác được lệnh OUTPUT (A) của T<sub>1</sub> đã được thực hiện hay chưa.

Nhắc lại thuật toán khôi phục cho redo log

```
/* =====
Step 1: identify the committed transactions
===== */
Scan the redo log backwards until first <START CKPT (T1, T2,
..., Tk) record:
{
  identify the committed
  identify the uncommitted/aborted transactions
}
/* =====
Step 2: redo the committed transactions
===== */
Scan the redo log backwards
until we found all <START T1>, <START T2>, ..., <START Tk>
records
```

```

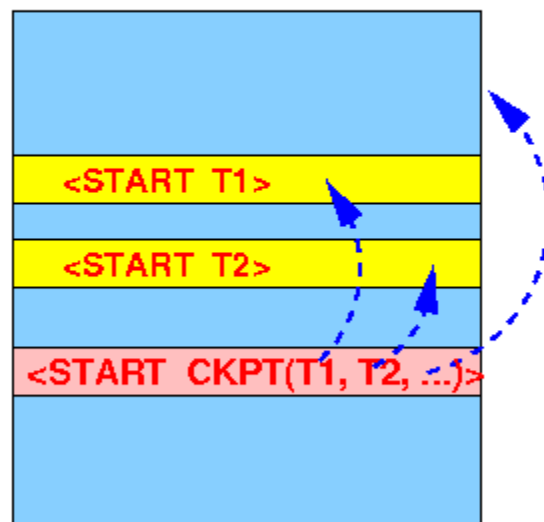
Starting from this point, scan the redo log forewards
{
  For ( each < T, A, v > where T is committed ) do
  {
    Update A with the (after) value v;    // Redo the
action !!!
  }
}
/* =====
Step 3: mark the uncommitted trasactions as failed....
=====
*/
For ( each T that is uncommitted ) do
{
  Write <ABORT T> to log;
}
Flush-Log

```

Chúng ta có thể tìm thấy tất cả các dòng <START T1>, <START T2>, ..., <START Tk>

Quá trình quét có thể được giới hạn lại nếu chúng ta đã ghi nhận tất cả các bản ghi <START T1>, <START T2>, ..., <START Tk> trong bản ghi <START CKPT>:

*Log file:*



## 6. Undo/Redo Logging

Phương pháp redo logging sẽ thực hiện câu lệnh `OUTPUT()` tại thời điểm thuận lợi, điều này làm cho nó trở nên phức tạp hơn so với undo logging. Tuy nhiên chúng ta có

thể nâng cao mức độ phức tạp lên hơn nữa bằng cách kết hợp cả hai phương pháp undo và redo logging.

### 6.1. Định nghĩa

Mỗi bản ghi nhật ký có dạng sau:

$\langle T_i, X, V_{new}, V_{old} \rangle$

$T_i$  - id của giao tác

$X$  - id của dữ liệu trong cơ sở dữ liệu.

$V_{new}$  - giá trị mới của  $X$  (như trong Redo Logging)

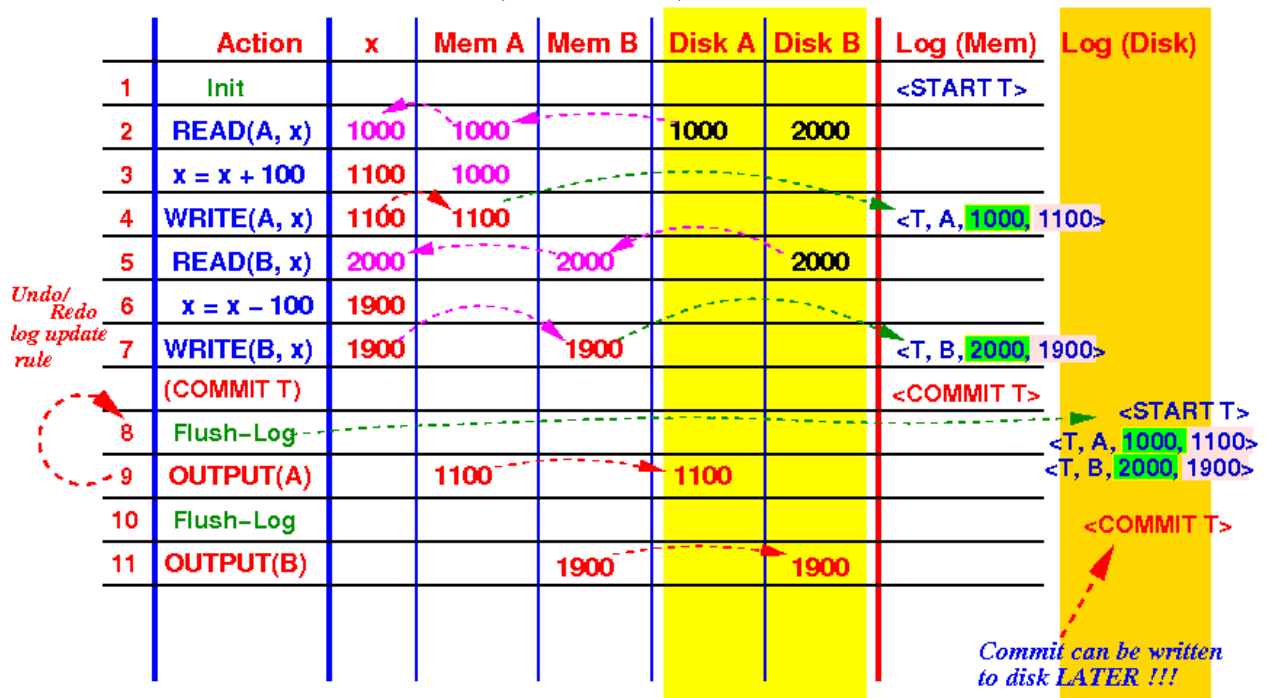
$V_{old}$  - giá trị cũ của  $X$  (như trong Undo Logging)

### 6.2. Quy tắc

Đối tượng  $X$  có thể được ghi xuống đĩa trước hoặc sau  $\langle T_i, \text{commit} \rangle$  - điều này không thành vấn đề.

Tất cả các bản ghi nhật ký phải được ghi xuống trước khi các sửa đổi tương ứng trên dữ liệu được ghi vào đĩa (write-ahead logging, WAL) (trước khi thực hiện lệnh  $\text{OUTPUT}()$ ).

Ghi ngay khi commit: một khi có  $\langle T_i, \text{commit} \rangle$ , ghi ngay nhật ký xuống đĩa cứng.



### Quy định chung để phục hồi từ tập tin nhật ký

Quy trình sau đây là thuật toán chung để khôi phục từ lỗi hệ thống:

- (1) Tìm tất cả các giao tác chưa commit.
- (2) Tìm tất cả các giao tác đã commit.

(3) Quay lui tất cả các cập nhật được thực hiện bởi các giao tác chưa commit.

(4) Làm lại tất cả các cập nhật được thực hiện bởi các giao tác đã commit.

Thuật toán khôi phục cho undo/redo logging:

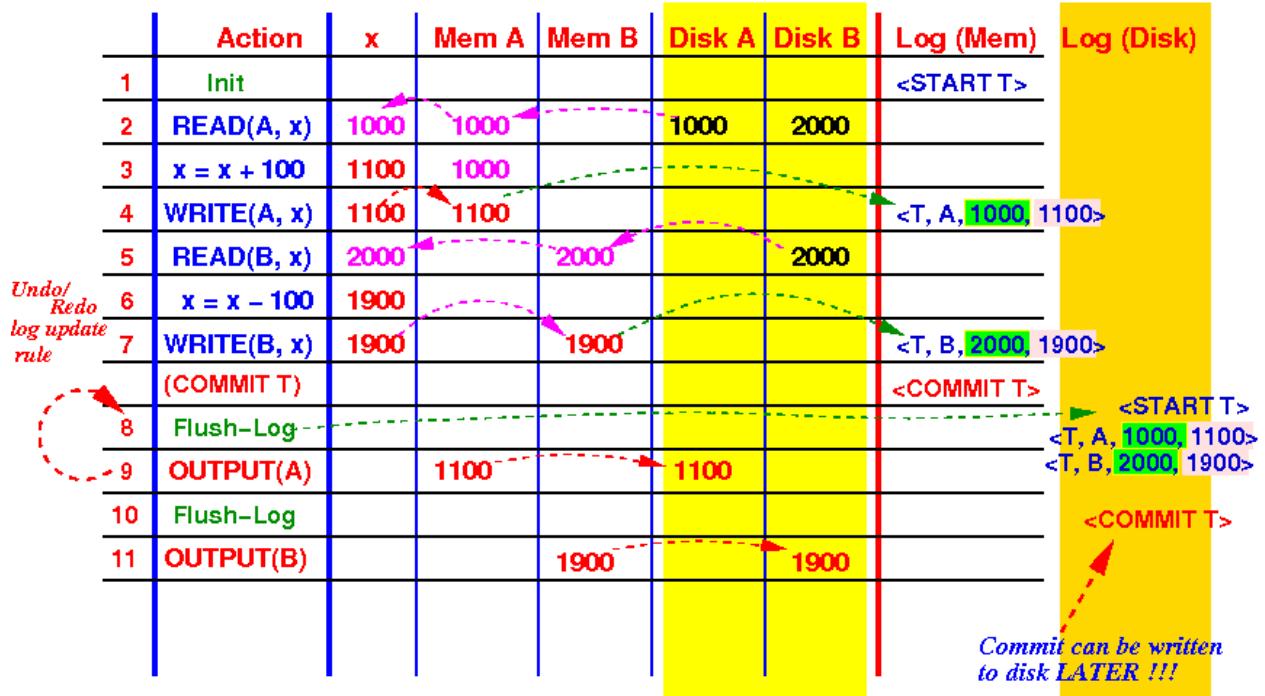
```
/* =====
Step 1: identify the committed/uncommitted transactions
===== */
Scan the redo log:
{
    identify the committed transaction // There is a COMMIT
record
    identify the uncommitted/aborted transaction // No COMMIT
record or ABORT
}
/* =====
Step 2a: undo the uncommitted/aborted transactions
===== */
Scan the undo log backwards:
{
    For ( each < T, A, v, w > in the undo/redo log )
    {
        if ( T is uncommitted )
        {
            Update A with the (before) value v; // Undo the
action !!!
        }
    }
}
/* =====
Step 2b: redo the committed transactions
===== */
Scan the undo log forwards:
{
    For ( each < T, A, v, w > in the undo/redo log )
    {
        if ( T is committed )
        {
            Update A with the (after) value w; // Redo the
action !!!
        }
    }
}
/* =====
Step 3: mark the uncommitted transactions as failed....
=====
*/
For ( each T that is uncommitted ) do
```

```

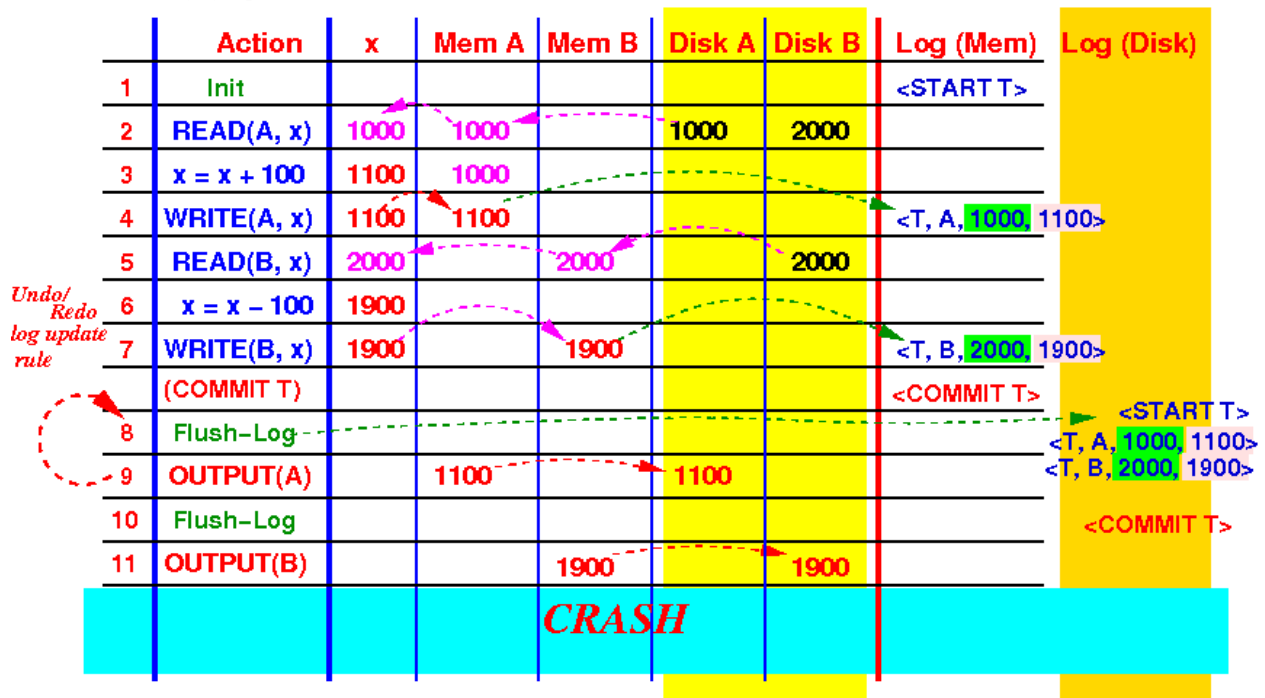
{
  Write <ABORT T> to log;
}
Flush-Log

```

Ví dụ về undo/redo logging:



Nếu hệ thống gặp sự cố sau bước 11:



Chúng ta tìm thấy:

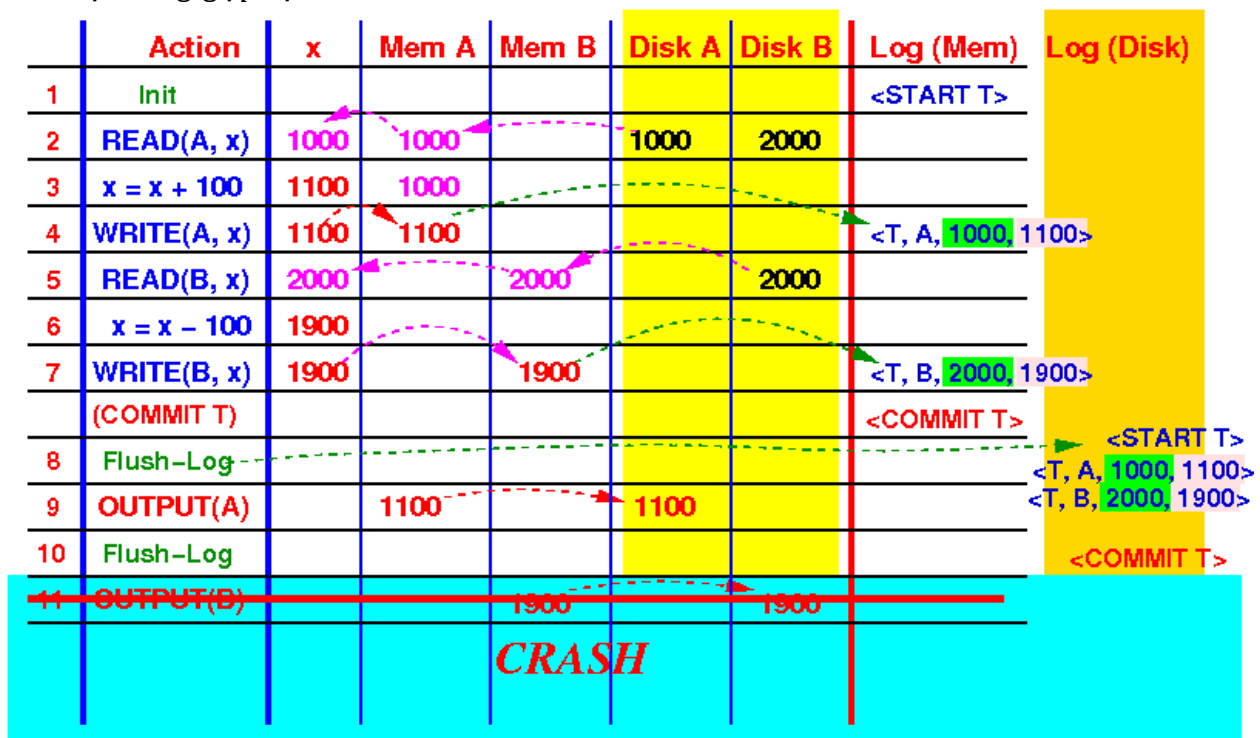
- ✓ Giao tác đã commit: T
- ✓ Giao tác chưa commit: không có

Tiến hành khôi phục:

- ✓ Cập nhật A thành (giá trị mới) 1100
- ✓ Cập nhật B thành (giá trị mới) 1900

Như vậy hệ quản trị đã thực hiện các cập nhật không cần thiết vì A và B đã sẵn chứa các giá trị mới. Tuy nhiên, chúng ta đảm bảo được trạng thái cơ sở dữ liệu là nhất quán.

Nếu hệ thống gặp sự cố sau bước 10:



Chúng ta tìm thấy:

Giao tác đã commit (tìm thấy trong nhật ký): T

Giao tác chưa commit: không có

Tiến hành khôi phục

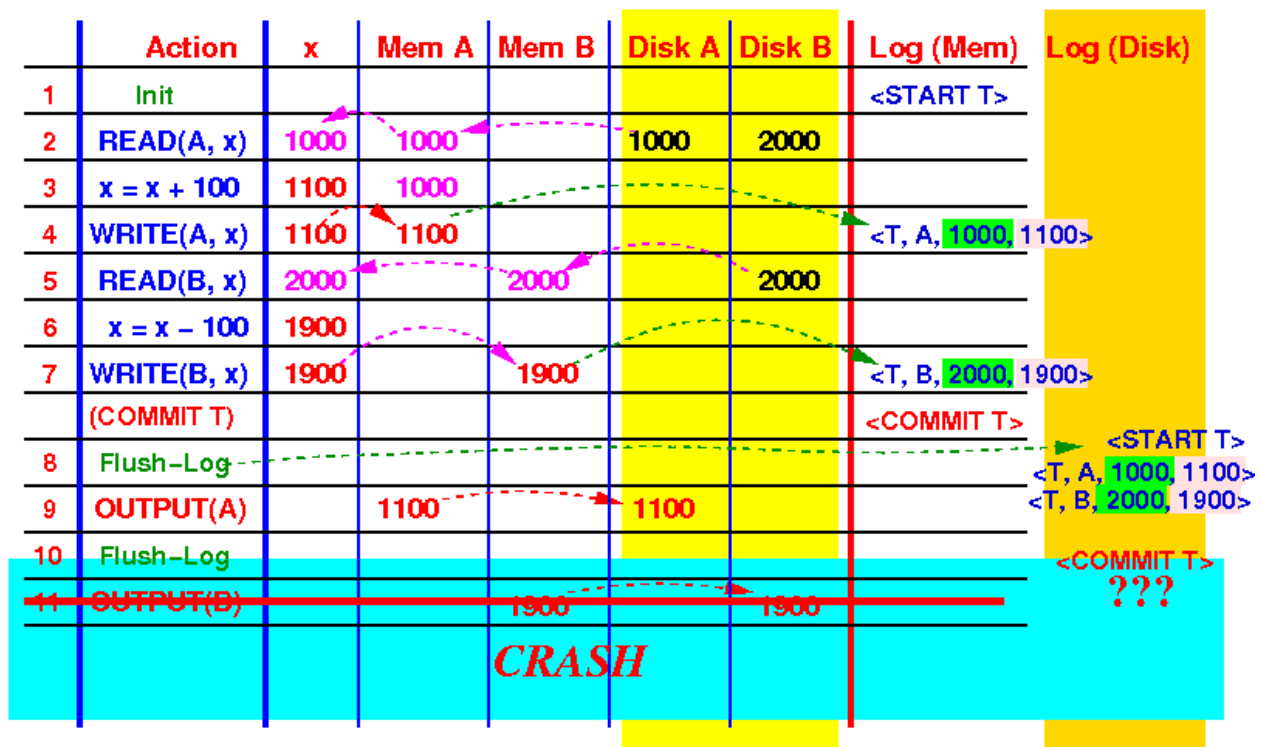
Cập nhật A thành (giá trị mới) 1100

Cập nhật B thành (giá trị mới) 1900

Nhận xét: Khi sự cố xảy ra A có giá trị mới, B có giá trị cũ, như vậy, cơ sở dữ liệu ở trong trạng thái không nhất quán!!! Việc phục hồi làm cho cơ sở dữ liệu trở nên nhất quán.

Nếu hệ thống gặp sự cố sau bước 9:





Có 2 khả năng xảy ra:

- ✓ <COMMIT T> đã được viết
- ✓ <COMMIT T> chưa được viết

Với khả năng thứ nhất: dòng <COMMIT T> đã được viết:

- ✓ Giao tác đã commit: T
- ✓ Giao tác chưa commit: không có

Tiến hành khôi phục

- ✓ Cập nhật A thành (giá trị mới) 1100
- ✓ Cập nhật B thành (giá trị mới) 1900

Như vậy khi sự cố xảy ra, A có giá trị mới, B có giá trị cũ → cơ sở dữ liệu không nhất quán → hệ quản trị quay lui chính xác các cập nhật. Do đó, cơ sở dữ liệu sẽ ở trạng thái nhất quán.

Với khả năng thứ hai: dòng <COMMIT T> chưa được viết:

- ✓ Giao tác đã commit: không có.
- ✓ Giao tác chưa commit: T

Tiến hành quay lui:

- ✓ Cập nhật A thành (giá trị cũ) 1000
- ✓ Cập nhật B thành (giá trị cũ) 2000

Như vậy khi sự cố xảy ra, A có giá trị mới, B có giá trị cũ  $\Rightarrow$  cơ sở dữ liệu không nhất quán  $\Rightarrow$  hệ quản trị quay lui chính xác các cập nhật. Do đó, cơ sở dữ liệu sẽ ở trạng thái nhất quán.

Nếu hệ thống gặp sự cố trước bước 10:

	Action	x	Mem A	Mem B	Disk A	Disk B	Log (Mem)	Log (Disk)
1	Init						<START T>	
2	READ(A, x)	1000	1000		1000	2000		
3	x = x + 100	1100	1000					
4	WRITE(A, x)	1100	1100				<T, A, 1000, 1100>	
5	READ(B, x)	2000		2000		2000		
6	x = x - 100	1900						
7	WRITE(B, x)	1900		1900			<T, B, 2000, 1900>	
	(COMMIT T)						<COMMIT T>	
8	Flush-Log							<START T> <T, A, 1000, 1100> <T, B, 2000, 1900>
9	OUTPUT(A)		1100		1100			
10	Flush-Log							
11	OUTPUT(B)			1900		1900		!!!
<b>CRASH</b>								

Chúng ta tìm thấy:

- ✓ Giao tác đã commit (tìm thấy trong nhật ký): không có.
- ✓ Giao tác chưa commit: T

Tiến hành hoàn tác:

- ✓ Cập nhật A thành (giá trị cũ) 1000.
- ✓ Cập nhật B thành (giá trị cũ) 2000.

Chúng ta có thể thấy rằng, giống như trường hợp redo log, giao tác đã commit (hoàn thành công việc) không có bản ghi <COMMIT> sẽ được quay lui. Những trường hợp này không thể tránh hoàn toàn, chúng ta chỉ có thể đảm bảo rằng trạng thái cơ sở dữ liệu sẽ nhất quán. Hệ quản trị sẽ giảm bớt vấn đề này bằng cách ghi nhật ký càng sớm càng tốt khi nhật ký vừa ghi dòng <COMMIT>.

### 6.3. Thuật toán sử dụng Non-Quiescent checkpoint (đối với undo/redo log)

#### 6.3.1. Các bước đặt checkpoint

1. Write a start checkpoint log record:  
`<START CKPT(T1, T2, ..., Tk)>`  
 where T<sub>1</sub>, T<sub>2</sub>, ..., T<sub>k</sub> are the currently active transactions
2. Flush-Log (This will write ALL <T, x, v, w> records to disk !!!)

3. Write:

All DB elements that were updated by ALL transactions  
that are still in memory buffers  
(I.e.: write all "dirty" (updated) buffers to disk)

4. Write <END CKPT> to log  
Flush-Log

Thuật toán checkpoint trong undo/redo log tương tự với thuật toán trong redo log (5.4.1). Sự khác biệt duy nhất giữa chúng là:

Redo log:

2. Write:

All DB elements that were updated by committed  
transactions  
that are still in memory buffers

---

Undo/redo log:

2. Write:

All DB elements that were updated by ALL transactions  
that are still in memory buffers

Ví dụ checkpoint trong undo/redo log: giả sử chúng ta có giao tác T1 và T2 đang hoạt động.

Undo/redo log:

<START T<sub>1</sub>>  
<T<sub>1</sub>, A, 4, 5>  
<START T<sub>2</sub>>  
<COMMIT T<sub>1</sub>>  
<T<sub>2</sub>, B, 9, 10>

Bây giờ chúng ta tiến hành đặt checkpoint.

Ghi <START CKPT (T<sub>2</sub>)> (Không bao gồm T<sub>1</sub> vì T<sub>1</sub> đã commit)

Ghi A và B xuống đĩa cứng (Ngay cả khi T<sub>1</sub> đã commit, dữ liệu được viết bởi T<sub>1</sub> có thể chưa được ghi vào đĩa).

Ghi <END CKPT>

Cuối cùng là ghi nhật ký vào đĩa cứng.

Undo/redo log:

<START T<sub>1</sub>>  
<T<sub>1</sub>, A, 4, 5>  
<START T<sub>2</sub>>  
<COMMIT T<sub>1</sub>> ===== T<sub>1</sub> done  
<T<sub>2</sub>, B, 9, 10>  
=====

<START CKPT (T <sub>2</sub> )>	
<T <sub>2</sub> , C, 14, 15>	<--+ Between here:
<START T <sub>3</sub> >	***** OUTPUT (A) and OUTPUT (B)
<T <sub>3</sub> , D, 19, 20>	<--+
<END CKPT>	// Flush Log

```
<COMMIT T2>          ===== T2 done
<COMMIT T3>          ===== T3 done
(OUTPUT for T2 and T3 happens later)
```

Thủ tục checkpoint sẽ ghi tất cả các phần tử dữ liệu đã cập nhật trước khi ghi bản ghi <END CKPT>. Điều này sẽ đơn giản hóa việc khôi phục các giao tác đã hoàn tất!

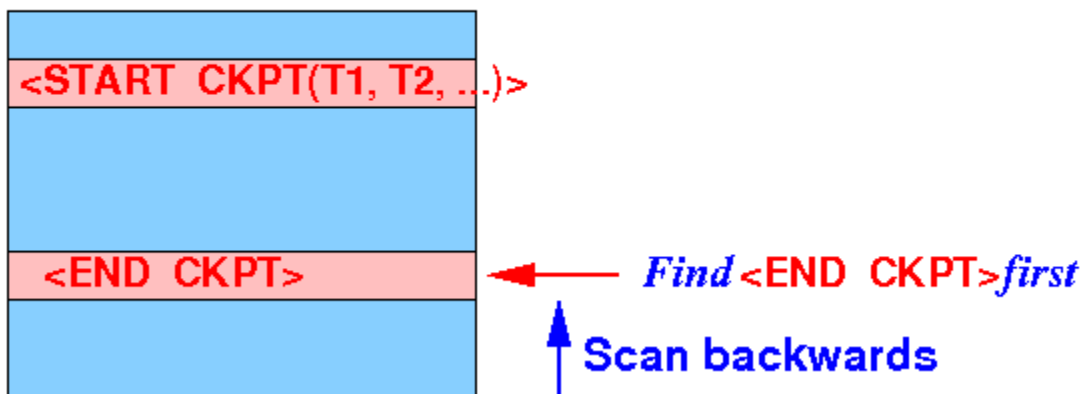
Ví dụ: Nếu trong quá trình phục hồi, hệ quản trị thấy T<sub>2</sub> đã commit (cùng với bản ghi <END CKPT>), thì chúng ta biết chắc chắn rằng tất cả các bản cập nhật do T<sub>2</sub> thực hiện trước khi bản ghi <START CKPT ( . . )> đã được ghi xuống đĩa cứng. Vì thế, để làm lại các hành động của một giao tác đã commit, chúng ta có thể bắt đầu làm lại hành động bắt đầu từ bản ghi <START CKPT ( . . . )>. Ngược lại, trong redo log, chúng ta phải bắt đầu từ bản ghi <START T<sub>i</sub>> sớm nhất.

### 6.3.2. Khôi phục cơ sở dữ liệu sử dụng nonquiescent checkpointing

Khi quét tập tin nhật ký theo chiều ngược từ cuối lên đầu, có thể xảy ra một trong 2 khả năng sau:

Bạn tìm thấy bản ghi nhật ký <END CKPT> trước:

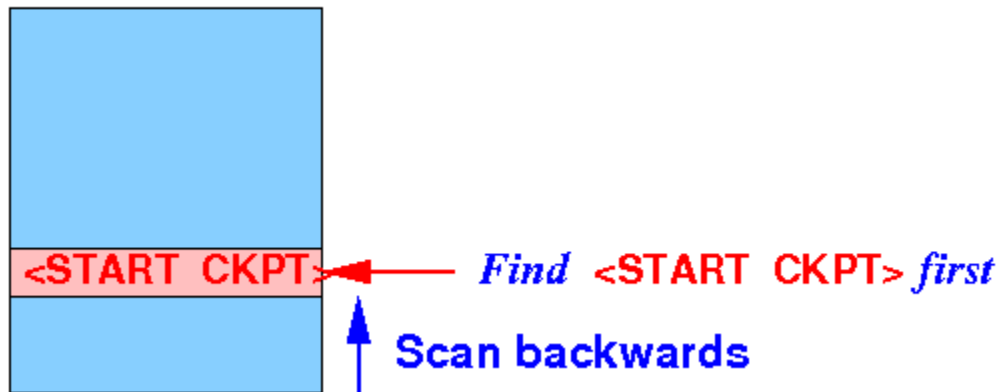
***Undo/redo log:***



Đây là trường hợp khi, hoạt động đặt <END CKPT> (cuối cùng) đã hoàn tất thành công. Như vậy, tất cả các khối dữ liệu cập nhật hoàn chỉnh và chưa hoàn chỉnh khi bắt đầu checkpoint đã được ghi vào đĩa.

Bạn tìm thấy bản ghi nhật ký <START CKPT (T<sub>1</sub>, T<sub>2</sub>, . . . , T<sub>k</sub>)> trước tiên:

### *Undo/redo log:*



Trường hợp này xảy ra khi hệ thống đã gặp sự cố trong lúc đợi đặt **<END CKPT>**! Trong trường hợp xấu nhất, không có dữ liệu nào khi tiến hành checkpoint kịp ghi được vào đĩa.

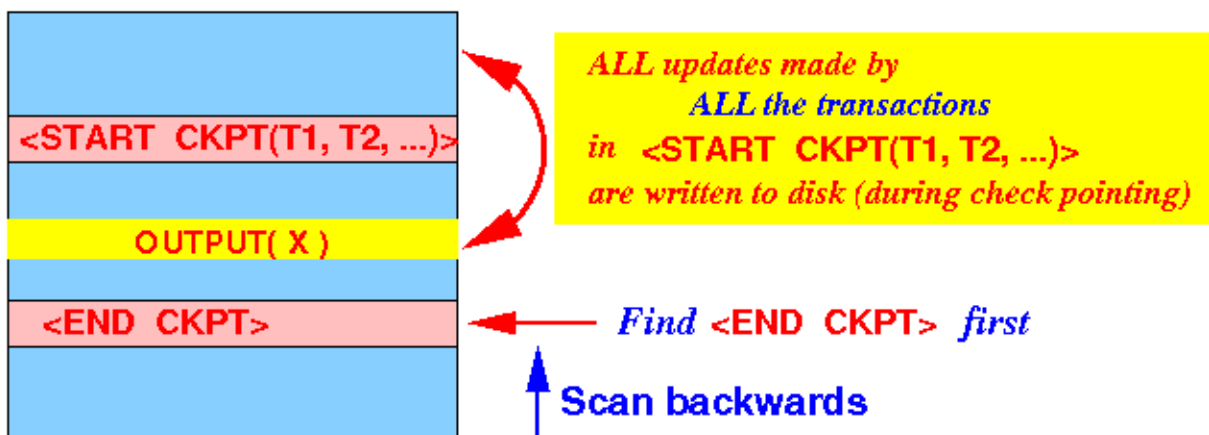
Chúng ta cần hoàn tác các giao tác chưa commit và làm lại các giao tác đã commit. Câu hỏi đặt ra là, phạm vi tìm kiếm các giao tác commit/chưa commit là như thế nào trong tập tin nhật ký?

### *Thuật toán làm lại (redo) cho những giao tác đã commit trong trường hợp 1 (undo/redo logging)*

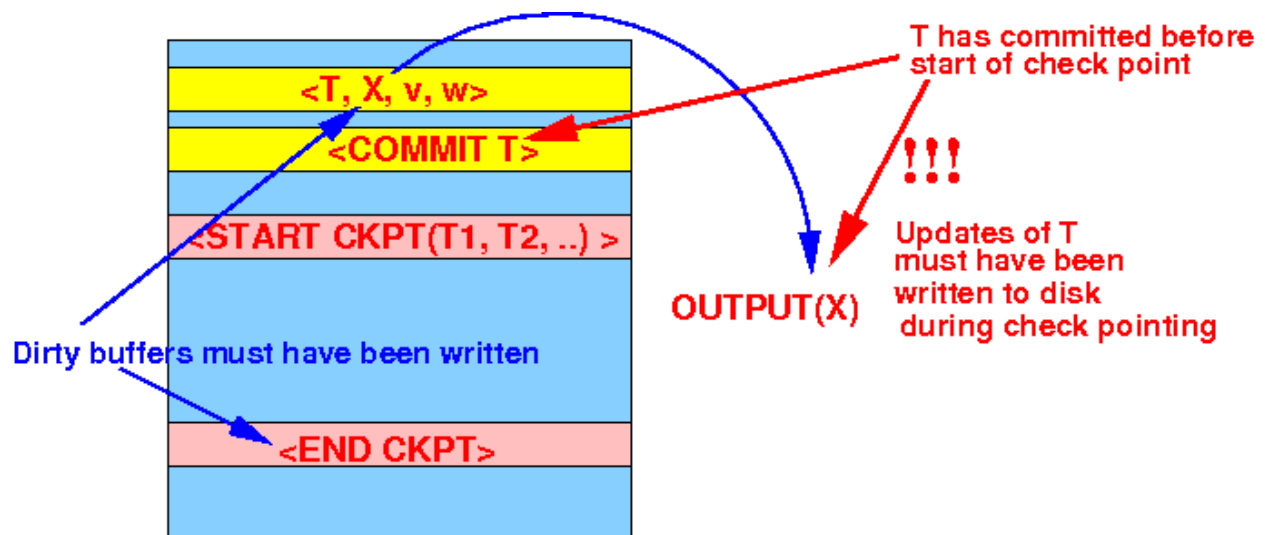
Xác định những giao tác đã commit cần làm lại trong trường hợp 1

Với việc tìm thấy dòng **<END CKPT>** trước:

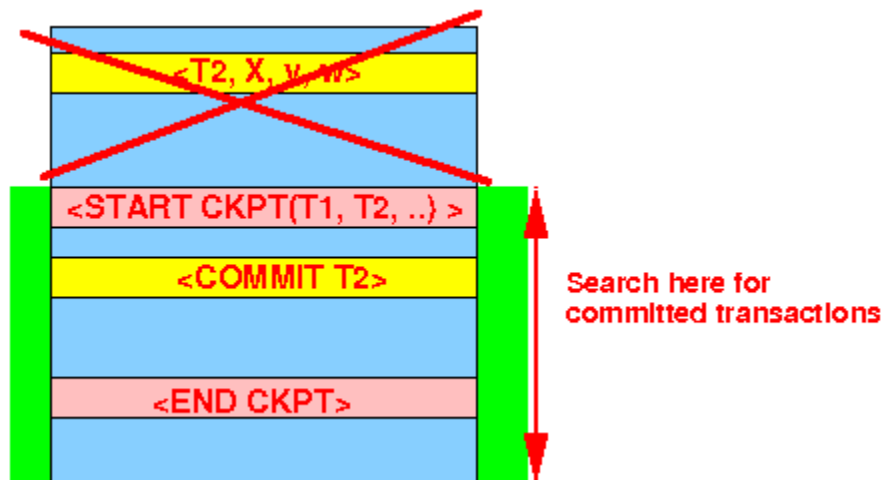
### *Undo/redo log:*



chúng ta đã biết chắc chắn tất cả các cập nhật được thực hiện bởi các giao tác đã commit trước **<START CKPT ( . . . )>** đã được ghi vào đĩa trong lúc checkpoint:

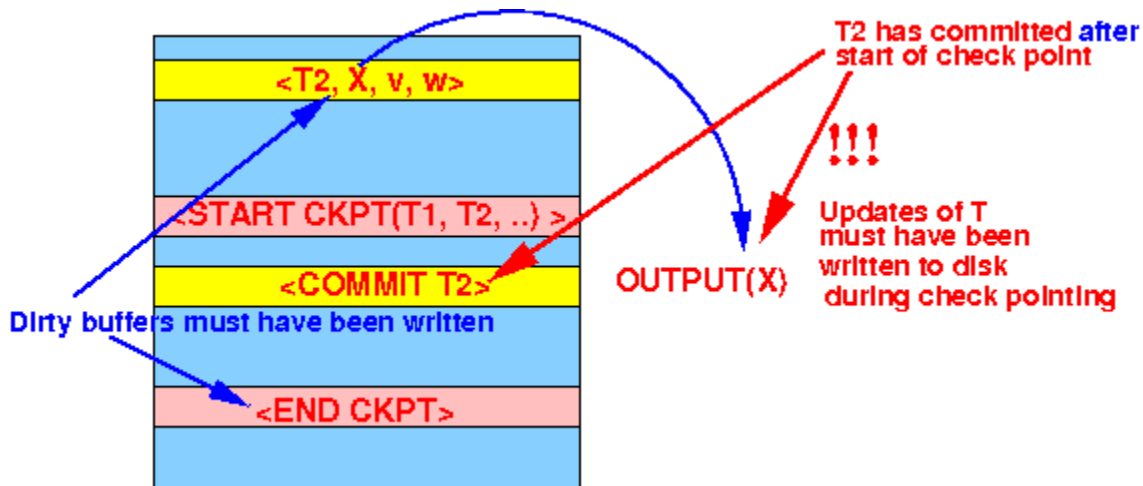


Vì thế, các dòng (trong bản ghi nhật ký) được thực hiện bởi các giao tác đã commit trước dòng `<START CKPT (...)>` có thể bỏ qua. Tức là, chúng ta chỉ cần tìm kiếm phần của undo/redo log cho các giao tác đã commit như trong hình vẽ dưới đây:

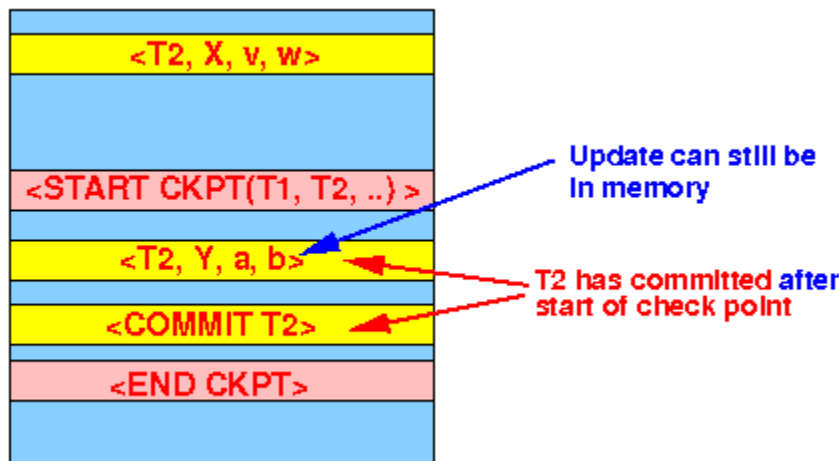


*Cách thực hiện lại các giao tác đã commit trong trường hợp 1:*

Các cập nhật được thực hiện bởi các giao tác đã commit trước dòng `<START CKPT (...)>` được ghi vào đĩa trong lúc checkpoint.



Các cập nhật được thực hiện sau khi checkpoint bắt đầu vẫn có thể nằm trong bộ nhớ:



Vì thế, giao tác đã thực hiện sau `<START CKPT>` chỉ cần được bắt đầu lại từ dòng `<START CKPT>`.

Tóm tắt: cách khôi phục các giao tác đã commit bằng undo/redo log:

- (1) Quét nhật ký ngược lên tới dòng `<START CKPT (T1, T2, ..., Tk)>`, xác định tất cả các giao tác đã commit.
- (2) Làm lại tất cả các thay đổi được thực hiện bởi các giao tác đã commit bắt đầu từ checkpoint.

```
<START CKPT (T1, T2, ..., Tk)>
...
...
...
```

**Thuật toán khôi phục (undo) cho những giao tác đã commit trong trường hợp 1 (undo/redo logging)**

Định vị các giao tác chưa commit trong trường hợp 1

Tìm thấy dòng `<END CKPT>` trước

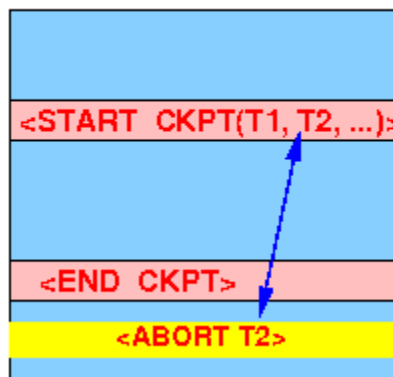
*Undo/redo log:*



Các giao tác chưa commit có thể là

Một số T1, T2, ... trong dòng `<START CKPT>`

*Undo/redo log:*



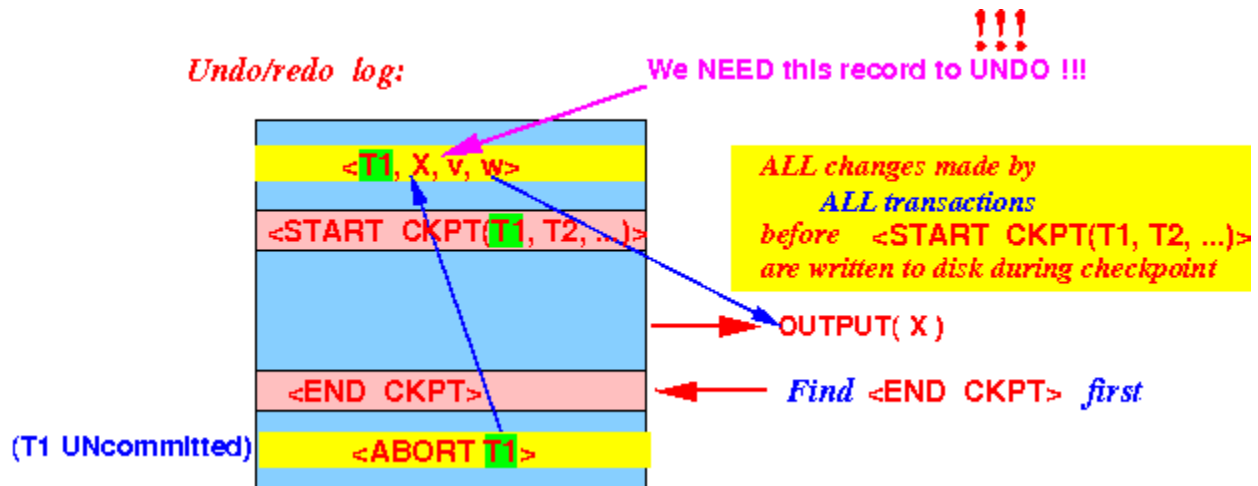
Các giao tác bắt đầu sau dòng `<START CKPT>`

*Undo/redo log:*



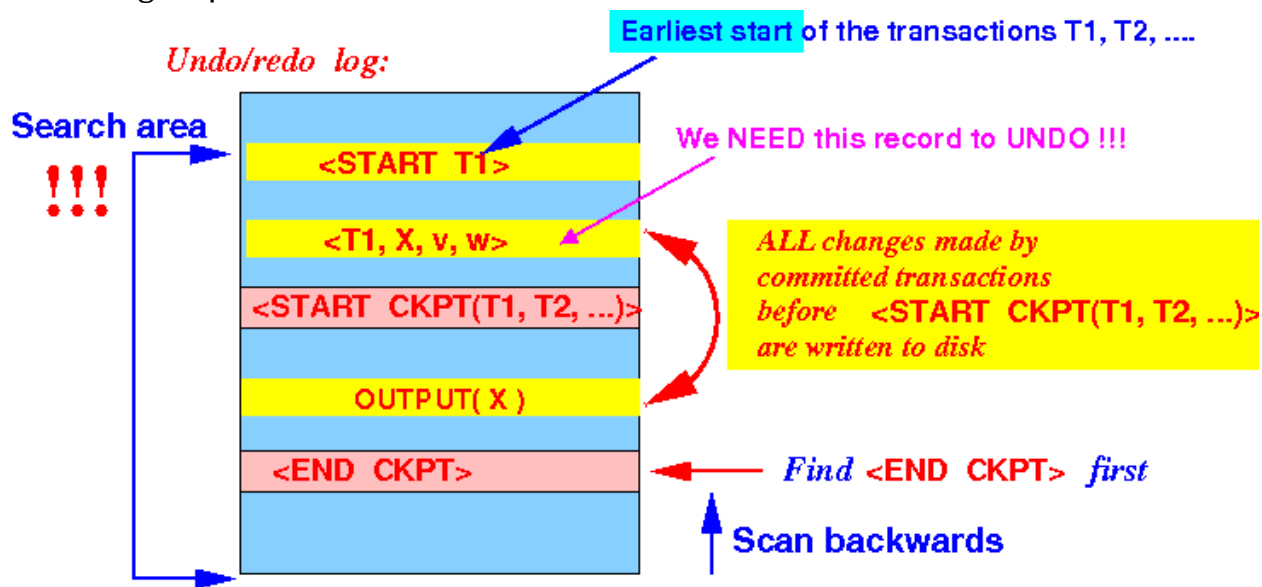
Thực tế quan trọng của thuật toán undo/redo logging đó là thuật toán checkpoint trong undo/redo log đã viết khối dữ liệu “xấu” được thực hiện bởi các giao tác không được commit trước checkpoint vào đĩa:





Vì thế, chúng ta sẽ cần các bản ghi nhật ký từ  $T1, T2, \dots$  trước bản ghi  $\langle \text{START CKPT} \rangle$  để hoàn tác một giao tác không được commit !!!

Do đó, phần của tập tin undo/redo log mà chúng ta cần sử dụng là để hoàn tác các giao tác không được commit là:



Cách hoàn tác các giao tác không được commit:

Quét ngược lại để tìm bản ghi  $\langle \text{START } T_x \rangle$  sớm nhất trong đó

$T_x$  là một trong các giao tác trong bản ghi checkpoint  $\langle \text{START CKPT}(T1, T1, \dots) \rangle$

$T_x$  là giao tác chưa commit

Sử dụng các bản ghi undo/redo log phần giữa kết thúc checkpoint và bản ghi  $\langle \text{START } T_x \rangle$  đầu tiên để hoàn tác các bản cập nhật của các giao tác không được commit:

$\langle \text{START } T_x \rangle$  log record

**Thuật toán khôi phục cho trường hợp 1 bằng cách sử dụng undo/redo log**

Bây giờ chúng ta có thể trình bày thuật toán khôi phục cho trường hợp 1 bằng cách sử dụng undo/redo log:

```

/*
=====
==
    Step 1: identify the committed and uncommitted
    transactions

=====
== */
    Scan the redo log backwards until first <START CKPT(T1, T2,
..., Tk) record:
    {
        identify the committed transaction // There is a COMMIT
record
        identify the UNcommitted transaction // No COMMIT record
    }
/*
=====
=
    Step 2a: undo the UNcommitted transactions

*****
***
    **** Tx = the earliest uncommitted transaction T1, T2, ...,
Tk ****

=====
== */
    Scan the redo log backwards until (earliest uncommitted)
<START Tx> record
    {
        For ( each < T, A, v, w > ) do
        {
            if ( T is UNcommitted )
            {
                Update A with the (before) value v; // Undo the
action !!!
            }
        }
    }
/* =====
    Step 2b: redo the committed transactions
===== */
    Scan the redo log backwards until first <START CKPT(T1, T2,
..., Tk) record
    Starting from this location, scan the redo log forewards
    {
        For ( each < T, A, v, w > ) do

```

```

    {
        if ( T is committed )
        {
            Update A with the (after) value w;    // Redo the
action !!!
        }
    }
}
/* =====
Step 3: mark the uncommitted transactions as failed....
=====
*/
For ( each T that is uncommitted ) do
{
    Write <ABORT T> to log;
}
Flush-Log

```

### Ví dụ 1

Giả sử hệ thống bị sự cố sau khi ghi cả hai dòng <COMMIT>:

```

Undo/redo log:
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>          ===== T1 done
<T2, B, 9, 10>
=====
<START CKPT(T2)>
<T2, C, 14, 15>      <--+ Between here:
<START T3>            | ***** OUTPUT(A) and OUTPUT(B)
<T3, D, 19, 20>      <--+
<END CKPT>           // Flush Log
<COMMIT T2>          ===== T2 done
<COMMIT T3>          ===== T3 done
+++++ System
crashed....
(OUTPUT for T2 and T3 happens later)

```

Hệ thống quét ngược lên và tìm thấy dòng <END CKPT> trước

```

Undo/redo log:
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>          ===== T1 done
<T2, B, 9, 10>
=====
<START CKPT(T2)>
<T2, C, 14, 15>      <--+ Between here:

```

```

<START T3> | **** OUTPUT (A) and OUTPUT (B)
<T3, D, 19, 20> <--+
<END CKPT> // Flush Log
<COMMIT T2> ===== T2 done
<COMMIT T3> ===== T3 done
+++++ System
crashed....
(OUTPUT for T2 and T3 happens later)

```

Tiếp tục quét cho đến khi tìm thấy dòng <START CKPT(..)>

```

Undo/redo log:
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1> ===== T1 done
<T2, B, 9, 10>
=====
<START CKPT(T2)>
<T2, C, 14, 15> <--+ Between here:
<START T3> | **** OUTPUT (A) and OUTPUT (B)
<T3, D, 19, 20> <--+
<END CKPT> // Flush Log
<COMMIT T2> ===== T2 done
<COMMIT T3> ===== T3 done
+++++ System
crashed....
(OUTPUT for T2 and T3 happens later)

```

Tìm thấy: T2 và T3 đã commit, không tìm thấy giao tác chưa commit.

Tiến hành làm lại tất cả các hành động của T2 và T3 bắt đầu từ dòng <START CKPT>

```

Undo/redo log:
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1> ===== T1 done
<T2, B, 9, 10>
=====
<START CKPT(T2)>
<T2, C, 14, 15> <--+ Between here:
<START T3> | **** OUTPUT (A) and OUTPUT (B)
<T3, D, 19, 20> <--+
<END CKPT> // Flush Log
<COMMIT T2> ===== T2 done
<COMMIT T3> ===== T3 done
+++++ System
crashed....
(OUTPUT for T2 and T3 happens later)

```

Kết quả: cập nhật C = 15, D = 20.

T2 cũng đã cập nhật B, về mặt kỹ thuật, chúng ta cũng phải làm lại bản cập nhật này. Nhưng, B đã được cập nhật bởi checkpoint, đó là lý do tại sao chúng ta có thể bắt đầu thao tác làm lại tại bản ghi <START CKPT>.

### Ví dụ 2:

Giả sử hệ thống gặp lỗi sau khi viết một trong 2 dòng <COMMIT>

```
Undo/redo log:
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>          ===== T1 done
<T2, B, 9, 10>
=====
<START CKPT (T2)>
<T2, C, 14, 15>      <--+ Between here:
<START T3>           | ***** OUTPUT (A) and OUTPUT (B)
<T3, D, 19, 20>      <--+
<END CKPT>           // Flush Log
<COMMIT T2>          ===== T2 done
+++++++ System
crashed....
-----<COMMIT T3>----- ===== T3 done
      (OUTPUT for T2 and T3 happens later)
```

Hệ thống quét ngược lên và tìm thấy dòng <END CKPT> trước

```
Undo/redo log:
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>          ===== T1 done
<T2, B, 9, 10>
=====
<START CKPT (T2)>
<T2, C, 14, 15>      <--+ Between here:
<START T3>           | ***** OUTPUT (A) and OUTPUT (B)
<T3, D, 19, 20>      <--+
<END CKPT>           // Flush Log
<COMMIT T2>          ===== T2 done
+++++++ System
crashed....
-----<COMMIT T3>----- ===== T3 done
      (OUTPUT for T2 and T3 happens later)
```

Tiếp tục quét cho đến khi tìm thấy dòng <START CKPT(..)>

```
Undo/redo log:
<START T1>
<T1, A, 4, 5>
<START T2>
```

```

<COMMIT T1>          ===== T1 done
<T2, B, 9, 10>
=====
<START CKPT (T2)>
<T2, C, 14, 15>      <--+ Between here:
<START T3>            | ***** OUTPUT (A) and OUTPUT (B)
<T3, D, 19, 20>      <--+
<END CKPT>           // Flush Log
<COMMIT T2>          ===== T2 done
+++++++ System
crashed....
-----<COMMIT T3>-----<T3 done
      (OUTPUT for T2 and T3 happens later)

```

Tìm thấy giao tác T<sub>2</sub> đã commit và T<sub>3</sub> chưa commit.

Tiến hành hai giai đoạn làm lại và khôi phục

*Phần làm lại*

Bắt đầu từ dòng <START CKPT>, làm lại tất cả các hành động của T<sub>2</sub>

```

Undo/redo log:
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>          ===== T1 done
<T2, B, 9, 10>
=====
<START CKPT (T2)>
<T2, C, 14, 15>      <--+ Between here:
<START T3>            | ***** OUTPUT (A) and OUTPUT (B)
<T3, D, 19, 20>      <--+
<END CKPT>           // Flush Log
<COMMIT T2>          ===== T2 done
+++++++ System
crashed....
-----<COMMIT T3>-----<T3 done
      (OUTPUT for T2 and T3 happens later)

```

Cập nhật C = 15, B được cập nhật bởi thao tác checkpoint.

*Phần khôi phục*

Vì T<sub>3</sub> không phải là một giao tác được đề cập trong nội dung dòng <START CKPT (T<sub>2</sub>)>, chúng ta có thể dừng lại ở dòng <START CKPT (T<sub>2</sub>)>. Dừng từ dòng <START CKPT (T<sub>2</sub>)>, hoàn tác tất cả tác vụ cho T<sub>3</sub>

```

Undo/redo log:
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>          ===== T1 done

```

```

<T2, B, 9, 10>
=====
<START CKPT(T2)>
<T2, C, 14, 15>      <--+ Between here:
<START T3>           | **** OUTPUT(A) and OUTPUT(B)
<T3, D, 19, 20>      <--+
<END CKPT>           // Flush Log
<COMMIT T2>          ===== T2 done
+++++++ System
crashed....
-----<COMMIT T3>-----<T3 done
      (OUTPUT for T2 and T3 happens later)

```

Kết quả, cập nhật D = 19

### Ví dụ 3:

Giả sử hệ thống bị hư sau dòng <END CKPT>

```

Undo/redo log:
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>          ===== T1 done
<T2, B, 9, 10>
=====
<START CKPT(T2)>
<T2, C, 14, 15>      <--+ Between here:
<START T3>           | **** OUTPUT(A) and OUTPUT(B)
<T3, D, 19, 20>      <--+
<END CKPT>           // Flush Log
+++++++ System
crashed....
-----<COMMIT T2>-----<T2 done
-----<COMMIT T3>-----<T3 done
      (OUTPUT for T2 and T3 happens later)

```

Hệ thống quét ngược lên và tìm thấy dòng <END CKPT> trước

```

Undo/redo log:
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>          ===== T1 done
<T2, B, 9, 10>
=====
<START CKPT(T2)>
<T2, C, 14, 15>      <--+ Between here:
<START T3>           | **** OUTPUT(A) and OUTPUT(B)
<T3, D, 19, 20>      <--+
<END CKPT>           // Flush Log

```

```

+++++++ System
crashed....

```

```

-----<COMMIT T2>-----===== T2 done
-----<COMMIT T3>-----===== T3 done
      (OUTPUT for T2 and T3 happens later)

```

Tiếp tục quét cho đến khi tìm thấy dòng <START CKPT (..) >

```

Undo/redo log:
  <START T1>
  <T1, A, 4, 5>
  <START T2>
  <COMMIT T1>          ===== T1 done
  <T2, B, 9, 10>
  =====
  <START CKPT (T2)>
  <T2, C, 14, 15>      <--+ Between here:
  <START T3>           | ***** OUTPUT (A) and OUTPUT (B)
  <T3, D, 19, 20>      <--+
  <END CKPT>           // Flush Log
+++++++ System
crashed....

```

```

-----<COMMIT T2>-----===== T2 done
-----<COMMIT T3>-----===== T3 done
      (OUTPUT for T2 and T3 happens later)

```

Kết luận: tìm thấy giao tác T2 và T3 chưa commit, không có giao tác nào đã commit. Do đó, không thực hiện làm lại mà chỉ thực hiện khôi phục.

Vì T2 là một giao tác đề cập trong nội dung dòng <START CKPT (T2)>, chúng ta cần tìm dòng <START T2>. Dừng từ dòng <START T2>, tiến hành hoàn tác tất cả tác vụ đối với T2 và T3

```

Undo/redo log:
  <START T1>
  <T1, A, 4, 5>
  <START T2>
  <COMMIT T1>          ===== T1 done
  <T2, B, 9, 10>
  =====
  <START CKPT (T2)>
  <T2, C, 14, 15>      <--+ Between here:
  <START T3>           | ***** OUTPUT (A) and OUTPUT (B)
  <T3, D, 19, 20>      <--+
  <END CKPT>           // Flush Log
+++++++ System
crashed....

```

```

-----<COMMIT T2>-----===== T2 done
-----<COMMIT T3>-----===== T3 done
      (OUTPUT for T2 and T3 happens later)

```



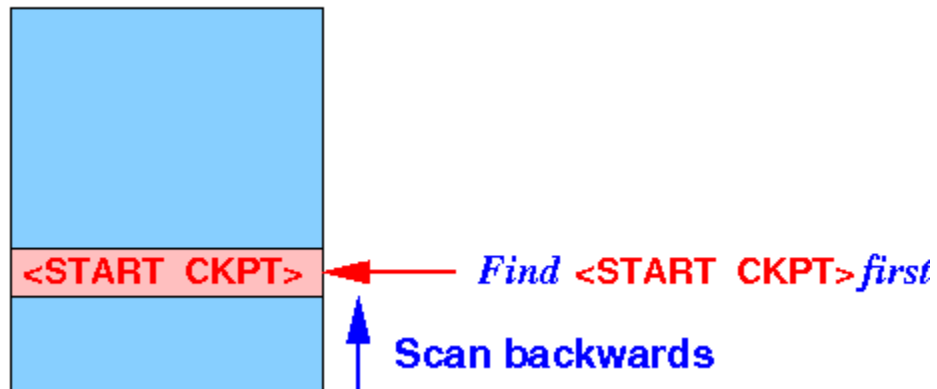
Kết quả: cập nhật B = 9, C = 14, D = 19

**Thuật toán phục hồi trong trường hợp 2 (giống như redo log)**

Thuật toán khôi phục cho trường hợp 2

Hệ thống tìm thấy bản ghi <START CKPT (T1, T2, ..., Tk)> trước:

*Redo log:*

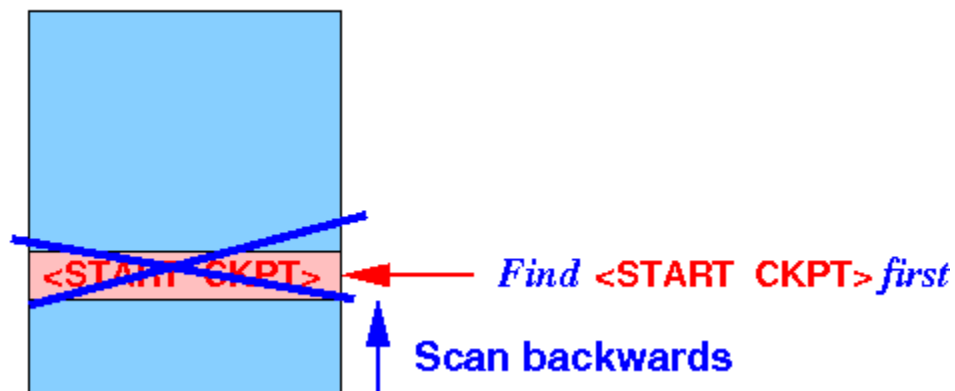


Trong trường hợp này:

Chúng ta không có thêm thông tin

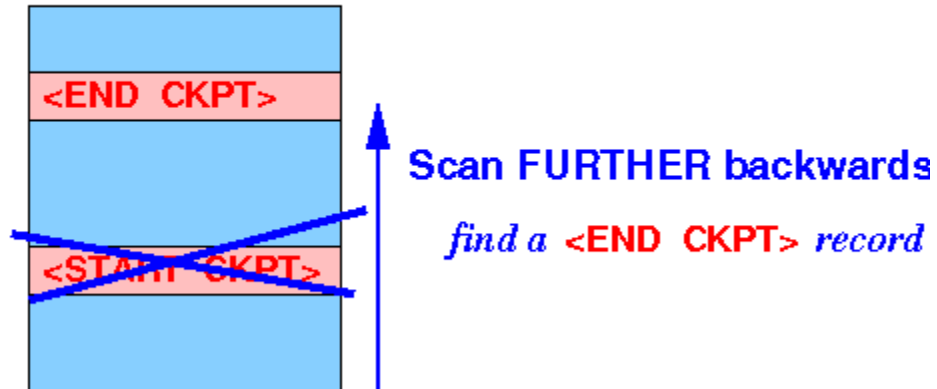
Nó giống như không có bản ghi <START CKPT>:

*Redo log:*



Tất cả những gì chúng ta có thể làm là:

*Redo log:*



và sử dụng quy trình khôi phục giống trường hợp 1.

## 7. References

- <http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/6-logging/>
- [http://mlwiki.org/index.php/Undo\\_Logging](http://mlwiki.org/index.php/Undo_Logging)
- [http://mlwiki.org/index.php/Redo\\_Logging](http://mlwiki.org/index.php/Redo_Logging)
- [http://mlwiki.org/index.php/Undo/Redo\\_Logging](http://mlwiki.org/index.php/Undo/Redo_Logging)