

## Chương 4. QUẢN LÝ GIAO DỊCH

### Nội Dung

- Giao dịch
  - ♦ Định nghĩa
  - ♦ Trạng thái của một GD
  - ♦ Thuộc tính của một GD
- Điều khiển cạnh tranh
  - ♦ Sự cần thiết phải quản lý cạnh tranh
  - ♦ Tính khả tuần tự, khả phục hồi của lịch trình
  - ♦ Các kỹ thuật quản lý cạnh tranh: bi quan & lạc quan
  - ♦ Độ mịn của mức DL
- Phục hồi CSDL
  - ♦ Sự cần thiết phải phục hồi dữ liệu
  - ♦ Các GD và sự phục hồi
  - ♦ Các tiện ích để phục hồi

7.2

### Định nghĩa Giao Dịch

- Giao dịch (GD): là một hành động hay một chuỗi các hành động được thực hiện bởi 1 người dùng hoặc 1 chương trình ứng dụng, trong đó có truy cập hoặc thay đổi nội dung của một CSDL.
- Một GD là một *đơn vị luận lý* của công việc trên CSDL. Nó có thể là:
  - ♦ Toàn bộ chương trình,
  - ♦ Một phần của chương trình,
  - ♦ Hoặc một lệnh đơn lẻ như INSERT hay UPDATE,
  - ♦ Hoặc nó có thể bao gồm nhiều thao tác trên CSDL

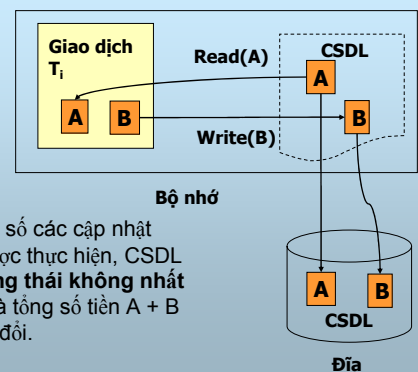
7.3

### Ví dụ một GD

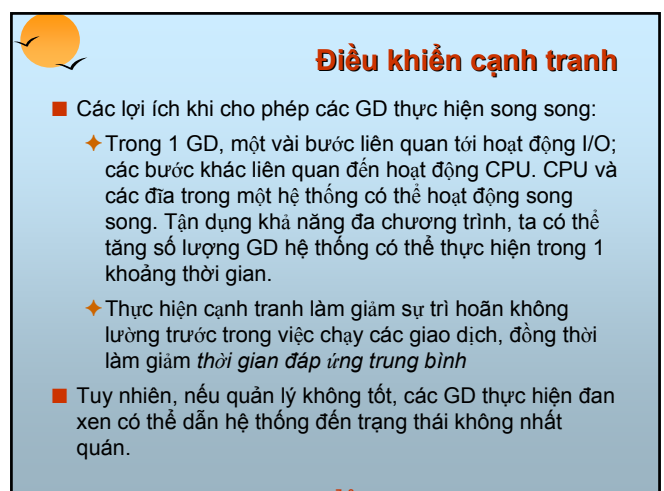
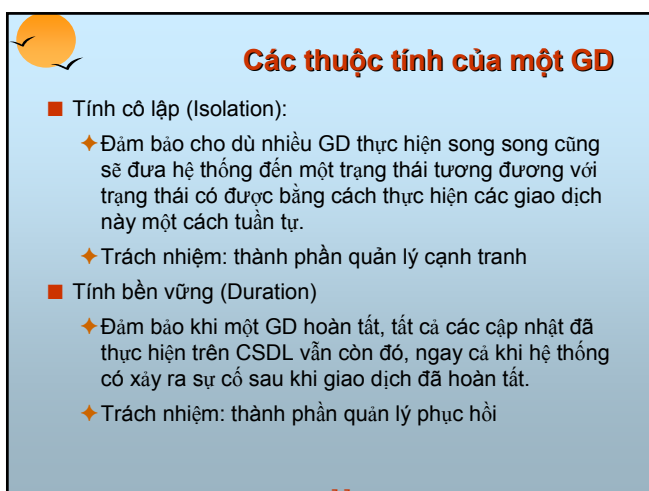
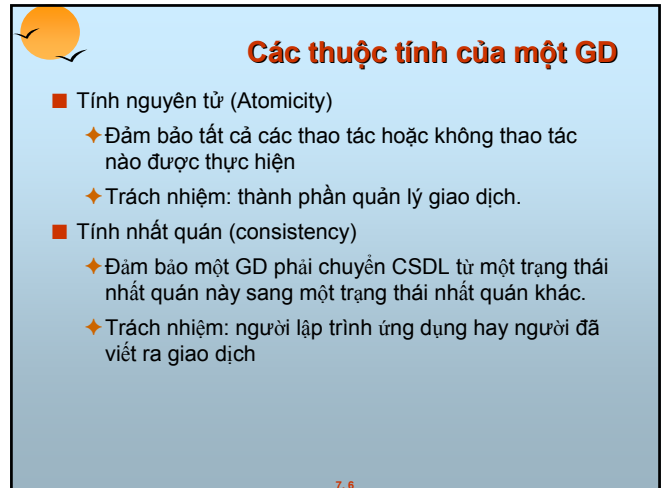
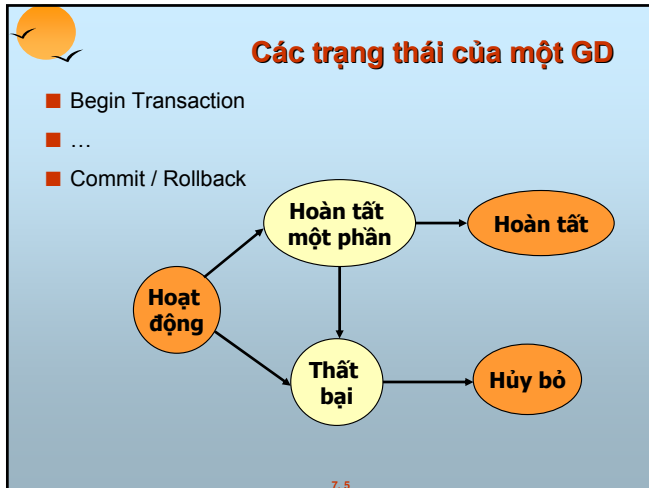
#### ■ Ví dụ GD $T_i$ :

```
READ(A);  
A:=A - 50;  
WRITE(A)  
READ(B);  
B:=B + 50;  
WRITE(B);
```

- Nếu một trong số các cập nhật này không được thực hiện, CSDL sẽ ở trong **trạng thái không nhất quán**: nghĩa là tổng số tiền A + B có thể bị thay đổi.



7.4



## Lịch trình (Schedule)

- **Lịch trình** là một chuỗi các thao tác thực hiện bởi một tập hợp các GD cạnh tranh mà vẫn đảm bảo thứ tự của các thao tác trong từng GD đơn lẻ.
- **Lịch trình tuần tự** (serial schedule) là một lịch trình trong đó các thao tác của một GD được thực hiện liên tiếp nhau.
  - ✦ Với một tập các giao dịch cho trước, không có sự đảm bảo là kết quả của tất cả các lịch trình tuần tự đều sẽ giống nhau.
- **Lịch trình không tuần tự** (nonserial schedule) là một lịch trình trong đó các thao tác từ một tập hợp các giao dịch cạnh tranh đan xen lẫn nhau.

T <sub>1</sub>	T <sub>2</sub>
Read(A);	
A:=A-50;	
Write(A);	
Read(B);	
B:=B+50;	
Write(B);	
	Read(A);
	Temp:=A*0.1;
	A:=A-temp;
	Write(A);
	Read(B);
	B:=B+temp;
	Write(B);

Hình 3. Lịch trình 1

7.9

## Vấn đề 1: Mất dữ liệu đã cập nhật

- Kết quả của một thao tác cập nhật hoàn tất một cách thành công bởi một người dùng có thể bị ghi đè lên bởi một người dùng khác
- $A = 100 + 100 - 10 = 190$

T <sub>1</sub>	T <sub>2</sub>	TK A
	Begin Transaction	100
Begin Transaction	<b>Read(A);</b>	100
<b>Read(A);</b>	A = A + 100 ;	100
A:=A-10;	<b>Write(A);</b>	200
<b>Write(A);</b>	Commit	90
Commit		90

7.10

## Vấn đề 2: Phụ thuộc GD không hoàn tất

- Vấn đề về sự phụ thuộc vào các GD không hoàn tất xảy ra khi một GD cho phép các GD khác nhìn thấy các kết quả tạm thời trước khi nó hoàn tất.
- $A = 100 - 10 = 90$

T <sub>3</sub>	T <sub>4</sub>	TK A
	Begin Transaction	100
	<b>Read(A);</b>	100
	A = A + 100 ;	100
Begin Transaction	<b>Write(A);</b>	200
<b>Read(A);</b>	:	200
A:=A-10;	<b>Rollback</b>	100
<b>Write(A);</b>		190
Commit		190

7.11

## Vấn đề 3: Phân tích không nhất quán

- GD chỉ đọc được phép đọc một phần kết quả của các GD chưa hoàn tất mà các GD này cùng lúc cũng đang cập nhật CSDL. Điều này gọi là dirty read hay unrepeatable read.

T <sub>5</sub>	T <sub>6</sub>	X	Y	Z	Sum
	Begin Transaction	100	50	25	
Begin Transaction	Sum = 0	100	50	25	0
<b>Read(X);</b>	<b>Read(X);</b>	100	50	25	0
X = X - 10 ;	Sum = Sum + X	100	50	25	100
<b>Write(X);</b>	<b>Read(Y);</b>	90	50	25	100
Read(Z)	Sum = Sum + Y	90	50	25	150
Z = Z + 10		90	50	25	150
<b>Write(Z);</b>		90	50	35	150
Commit	<b>Read(Z)</b>	90	50	35	150
	Sum = Sum + Z	90	50	35	185
	Commit	90	50	35	185

7.12

## Tính khả tuần tự-Thảo luận

### ■ Sự khả tuần tự (serializability):

- ♦ Mục tiêu: tìm ra các lịch trình không tuần tự cho phép các GD thực hiện cạnh tranh mà vẫn không can thiệp lẫn nhau, và vì vậy dẫn đến một tình trạng CSDL giống như sự thực thi tuần tự có thể dẫn đến.

### ■ Lịch trình khả tuần tự (serializable schedule): là các lịch trình không tuần tự cho phép các GD thực hiện cạnh tranh nhưng vẫn cho kết quả giống như lịch trình tuần tự.

### ■ Có 2 kiểu khả tuần tự:

- ♦ Khả tuần tự xung đột
- ♦ Khả tuần tự hướng nhìn

7.13

## Khả tuần tự xung đột

### ■ Tương đương xung đột (conflict equivalent): Hai lịch trình S và S' được gọi là tương đương xung đột nếu S có thể biến đổi thành S' bằng cách trao đổi thứ tự của các thao tác không xung đột

- ♦ Thao tác xung đột: trên một mục dữ liệu x, một GD ghi còn GD kia đọc hoặc ghi.

### ■ Lịch trình khả tuần tự xung đột: Một lịch trình S được gọi là khả tuần tự xung đột nếu nó tương đương xung đột với một lịch trình tuần tự.

### ■ Kiểm tra tính khả tuần tự xung đột của 1 lịch trình bằng Đồ thị ưu tiên:

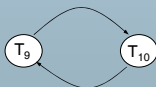
- ♦ Một nút cho mỗi GD
- ♦ Một cạnh trực tiếp đi từ Ti đến Tj nếu Tj đọc giá trị của mục được ghi bởi Ti
- ♦ Một cạnh trực tiếp đi từ Ti đến Tj nếu Tj ghi giá trị vào mục mà sau đó được đọc bởi Ti
- ♦ Nếu đồ thị này chứa một chu trình thì lịch trình tương ứng là không khả tuần tự xung đột. Ngược lại, lịch trình tương ứng là khả tuần tự xung đột.

7.14

## Ví dụ: Lịch trình không khả TT xung đột

T <sub>9</sub>	T <sub>10</sub>
Begin Transaction	
Read(X)	
$X = X + 100$	
Write(X)	Begin Transaction
	Read(X)
	$X = X * 1.1$
	Write(X)
	Read(Y)
	$Y = Y * 1.1$
	Write(Y)
Read(Y);	Commit
$Y = Y - 100$	
Write(Y);	
Commit	

Đồ thị ưu tiên:



7.15

## Khả tuần tự hướng nhìn (view)

### ■ Tương đương view (view equivalent): Hai lịch trình S1 và S2 là tương đương view nếu chúng thỏa 3 điều kiện sau:

- ♦ Với mỗi mục dữ liệu x, nếu GD Ti đọc giá trị khởi đầu của x trong lịch trình S1, thì trong S2 cũng vậy.
- ♦ Với mỗi thao tác đọc trên mục dữ liệu x bởi GD Ti trong lịch trình S1, nếu giá trị x đọc bởi Ti được ghi bởi GD Tj, thì trong S2 cũng vậy.
- ♦ Với mỗi mục dữ liệu x, nếu thao tác ghi cuối cùng trên x được thực hiện bởi Ti trong lịch trình S1, thì trong S2 cũng vậy.

### ■ Lịch trình khả tuần tự view: Một lịch trình được gọi là khả tuần tự view nếu nó là tương đương view với một lịch trình tuần tự.

- Mọi lịch trình khả tuần tự xung đột đều là khả tuần tự view, mặc dù điều ngược lại là không đúng.

7.16

## Tính Khả Phục Hồi

- Ví dụ lịch trình không thể phục hồi:

$T_9$	$T_{10}$
Begin Transaction	
Read(X)	
$X = X + 100$	
Write(X)	Begin Transaction
	Read(X)
	$X = X * 1.1$
	Write(X)
	Read(Y)
	.
Read(Y);	Commit
Rollback	

- **Lịch trình khả phục hồi** (recoverable schedule) là một lịch trình trong đó với mỗi cặp GD  $T_i$  và  $T_j$ , nếu  $T_j$  đọc một mục dữ liệu mà trước đó đã được ghi bởi  $T_i$ , thì thao tác commit của  $T_i$  phải thực hiện trước thao tác commit của  $T_j$ .

7.17

## Các kỹ thuật quản lý cạnh tranh

- Các kỹ thuật bị quan
  - ♦ Khóa chốt
  - ♦ Nhãn thời gian
- Các kỹ thuật lạc quan
- Độ mịn của mục dữ liệu
  - ♦ Khóa đa hạt

7.18

## Khóa chốt

- **Khóa đọc** (read lock): Nếu một GD có một khóa đọc trên một mục dữ liệu, nó có thể đọc nhưng không thể cập nhật mục dữ liệu đó.
- **Khóa ghi** (write lock) Nếu một GD có một khóa ghi trên một mục dữ liệu, nó có thể đọc và cập nhật mục dữ liệu đó.
- Cách sử dụng:
  - ♦ Bất kỳ một GD nào cần truy cập một mục dữ liệu, trước hết nó phải khóa mục đó lại, bằng cách yêu cầu một khóa đọc cho truy cập chỉ đọc hoặc khóa ghi cho truy cập có cả đọc và ghi.
  - ♦ Nếu mục dữ liệu này chưa bị khóa bởi một GD nào khác, thì khóa sẽ được cấp.
  - ♦ Nếu mục dữ liệu này đang bị khóa bởi 1 khóa tương thích (khóa đọc với khóa đọc), thì yêu cầu này sẽ được đáp ứng; bằng không thì GD buộc phải **đợi** cho đến khi khóa đã có được giải phóng.
  - ♦ Một GD tiếp tục giữ một khóa cho đến khi:
    - ✓ Nó giải phóng khóa một cách tường minh lúc đang thực thi
    - ✓ Hoặc lúc nó kết thúc (hủy bỏ hoặc hoàn tất).
  - ♦ Chỉ khi nào khóa ghi được giải phóng thì hiệu ứng của thao tác ghi mới được nhìn thấy bởi các giao dịch khác.

7.19

## Ví dụ: Lịch trình không khả TT xung đột

$T_9$	$T_{10}$
Begin Transaction	
Read(X)	
$X = X + 100$	
Write(X)	Begin Transaction
	Read(X)
	$X = X * 1.1$
	Write(X)
	Read(Y)
	$Y = Y * 1.1$
	Write(Y)
Read(Y);	Commit
$Y = Y - 100$	
Write(Y);	
Commit	

7.20

## Khóa chốt (tt.)

- Một số HQTCSDDL hỗ trợ nâng cấp / giáng cấp khóa nhằm tăng khả năng cạnh tranh cho hệ thống.
- Nếu chỉ sử dụng khóa, thì không đủ để đảm bảo tính khả tuần tự của các lịch trình:

S = {write\_lock(T9, X), read(T9, X), write(T9, X), unlock(T9, X),  
write\_lock(T10, X), read(T10, X), write(T10, X), unlock(T10, X),  
write\_lock(T10, Y), read(T10, Y), write(T10, Y), unlock(T10, Y),  
commit(T10),  
write\_lock(T9, Y), read(T9, Y), write(T9, Y), unlock(T9, Y), commit (T9)}

- ♦ Nếu ban đầu, X = 100, Y = 400, thì kết quả sẽ là:
  - ✓ X = 220, Y = 330, nếu T9 thực hiện trước T10;
  - ✓ hoặc X = 210, Y = 340, nếu T10 thực hiện trước T9.
- ♦ Kết quả đúng của S phải là X = 220 và Y = 340. (chứng tỏ S không phải là một lịch trình khả tuần tự.)

7.21

## Giao thức khóa 2 kỳ (2PL)

- Một GD tuân theo giao thức khóa hai kỳ nếu như tất cả các thao tác khóa đều tiến hành trước thao tác mở khóa trong GD đó.
- Mỗi GD có thể chia ra thành 2 kỳ:
  - ♦ **Kỳ phát triển** (growing phase) yêu cầu tất cả các khóa cần thiết nhưng không giải phóng khóa nào,
  - ♦ **Kỳ co lại** (shrinking phase) giải phóng các khóa của nó và không thể yêu cầu thêm bất kỳ khóa mới nào.

7.22

## Giao thức 2PL giải quyết vấn đề mất dữ liệu đã cập nhật

Time	T <sub>1</sub>	T <sub>2</sub>	TK A
t <sub>1</sub>		Begin Transaction	100
t <sub>2</sub>	Begin Transaction	Write_lock(A)	100
t <sub>3</sub>	Write_lock(A)	Read(A);	100
t <sub>4</sub>	<b>WAIT</b>	A = A + 100 ;	100
t <sub>5</sub>	<b>WAIT</b>	Write(A);	200
t <sub>6</sub>	<b>WAIT</b>	Commit/ unlock(A)	200
t <sub>7</sub>	Read(A);		200
t <sub>8</sub>	A:=A-10;		190
t <sub>9</sub>	Write(A);		190
t <sub>10</sub>	Commit/ Unlock(A)		190

7.23

## Giao thức 2PL giải quyết vấn đề phụ thuộc vào GD không hoàn tất

Time	T <sub>3</sub>	T <sub>4</sub>	TK A
t <sub>1</sub>		Begin Transaction	100
t <sub>2</sub>		Write_lock(A)	100
t <sub>3</sub>	Begin Transaction	Read(A);	100
t <sub>4</sub>	Write_lock(A)	A = A + 100 ;	100
t <sub>5</sub>	<b>WAIT</b>	Write(A);	200
t <sub>6</sub>	<b>WAIT</b>	Rollback/ unlock(A)	100
t <sub>7</sub>	Read(A);		100
t <sub>8</sub>	A:=A-10;		90
t <sub>9</sub>	Write(A);		90
t <sub>10</sub>	Commit/ Unlock(A)		90

7.24

## Giao thức 2PL giải quyết vấn đề phân tích không nhất quán

t	T <sub>s</sub>	T <sub>s</sub>	X	Y	Z	Sum
t <sub>1</sub>	Begin Transaction	Begin Transaction	100	50	25	0
t <sub>2</sub>	Write lock (X)	Sum = 0	100	50	25	0
t <sub>3</sub>	Read(X);	Read lock (X)	100	50	25	0
t <sub>4</sub>	X = X - 10 ;	WAIT	100	50	25	0
t <sub>5</sub>	Write (X);	WAIT	90	50	25	0
t <sub>6</sub>	Write lock (Z)	WAIT	90	50	25	0
t <sub>7</sub>	Read (Z)	WAIT	90	50	25	0
t <sub>8</sub>	Z = Z + 10	WAIT	90	50	25	0
t <sub>9</sub>	Write(Z);	WAIT	90	50	35	0
t <sub>10</sub>	Commit/ Unlock(X,Z)	WAIT	90	50	35	0
t <sub>11</sub>		Read(X);	90	50	35	0
t <sub>12</sub>		Sum = Sum + X	90	50	35	90
t <sub>13</sub>		Read lock(Y)	90	50	35	90
t <sub>14</sub>		Read(Y);	90	50	35	90
t <sub>15</sub>		Sum = Sum + Y	90	50	35	140
t <sub>16</sub>		Read lock(Z)	90	50	35	140
t <sub>17</sub>		Read(Z)	90	50	35	140
t <sub>18</sub>		Sum = Sum + Z	90	50	35	175
t <sub>19</sub>		Commit/ Unlock(X,Y,Z)	90	50	35	175

7.25

## Vấn đề nảy sinh khi sử dụng khóa hai kỳ

Time	T <sub>14</sub>	T <sub>15</sub>	T <sub>16</sub>
t <sub>1</sub>	Begin Transaction		
t <sub>2</sub>	Write lock(X)		
t <sub>3</sub>	Read(X)		
t <sub>4</sub>	Read lock(Y)		
t <sub>5</sub>	Read(Y)		
t <sub>6</sub>	X = X + Y		
t <sub>7</sub>	Write(X);		
t <sub>8</sub>	Unlock(X);	Begin Transaction	
t <sub>9</sub>	:	Write lock(X)	
t <sub>10</sub>	:	Read(X);	
t <sub>11</sub>	:	X = X + 100 ;	
t <sub>12</sub>	:	Write(X);	
t <sub>13</sub>	:	Unlock(X)	
t <sub>14</sub>	:	:	
t <sub>15</sub>	rollback	:	Begin Transaction
t <sub>16</sub>		Rollback	Read lock(X)
t <sub>17</sub>		:	
t <sub>18</sub>			Rollback

Cuộn nhiều tầng

7.26

## Vấn đề nảy sinh khi sử dụng khóa hai kỳ (tt.)

### ■ Để tránh vấn đề cuộn nhiều tầng, 2 giao thức khóa 2 kỳ biến đổi:

- ♦ **Khóa hai kỳ nghiêm ngặt** (rigorous 2PL): chỉ giải phóng tất cả khóa ở cuối GD.
- ♦ **Khóa 2 kỳ chặt chẽ** (strict 2PL): chỉ giữ các khóa ghi đến cuối GD.

### ■ Ngoài ra sử dụng 2PL, có thể dẫn đến:

- ♦ **Khóa chết** (deadlock): hai hay nhiều GD đang chờ lẫn nhau để có được các khóa đang giữ bởi đối phương.
- ♦ **Khóa sống** (livelock): GD bị bỏ ở tình trạng chờ mãi mãi, không thể đạt được thêm khóa mới nào. Nguyên nhân là do hệ thống sử dụng giải thuật chờ không công bằng và không xem xét đến thời gian mà GD đã đợi.
- ✓ Để tránh khóa sống: các GD càng chờ lâu thì độ ưu tiên của nó càng cao. Hoặc sử dụng một hàng đợi theo kiểu *FIFO*.

7.27

## Kỹ thuật xử lý khóa chết

### ■ Có hai kỹ thuật dùng để xử lý khóa chết là:

- ♦ Ngăn chặn khóa chết
- ♦ Phát hiện – phục hồi khóa chết.

### ■ Ngăn chặn khóa chết: sử dụng nhãn thời gian

- ♦ **Wait – Die**: Khi T<sub>i</sub> yêu cầu một mục DL đang bị chiếm bởi T<sub>j</sub>, T<sub>i</sub> chỉ được phép chờ nếu T<sub>i</sub> 'cũ' hơn T<sub>j</sub>, nếu không T<sub>i</sub> sẽ bị hủy và khởi động lại với cùng nhãn thời gian, để cuối cùng nó sẽ trở thành GD đang hoạt động cũ nhất và sẽ không chết nữa.
- ♦ **Wound – Die**: Nếu T<sub>i</sub> yêu cầu một mục DL đang bị chiếm bởi T<sub>j</sub>, T<sub>i</sub> được phép chờ nếu T<sub>i</sub> 'mới' hơn T<sub>j</sub>, nếu không T<sub>j</sub> bị hủy và khởi động lại với cùng nhãn thời gian, để cuối cùng nó sẽ trở thành GD đang hoạt động cũ nhất và sẽ không chết nữa.

### ■ Phát hiện – phục hồi khóa chết

- ♦ Một cách định kỳ, HQTCSDL xây dựng **Đồ thị chờ**:
  - ✓ Tạo một nút cho mỗi GD
  - ✓ Tạo một cạnh trực tiếp từ T<sub>i</sub> đến T<sub>j</sub>, nếu GD T<sub>i</sub> đang chờ để khóa một mục mà đang bị khóa bởi T<sub>j</sub>.
- ♦ Khóa chết tồn tại khi và chỉ khi đồ thị chờ chứa một chu trình

7.28



## Định nhãn thời gian

- **Nhãn thời gian** (timestamp) của một GD là một định danh duy nhất được tạo ra bởi HQTCSĐL cho thấy thời điểm bắt đầu tương đối của GD đó.
- Các nhãn thời gian (TG) có thể được tạo ra bằng cách:
  - ✦ Sử dụng đồng hồ của hệ thống ở thời điểm GD bắt đầu
  - ✦ Hoặc tăng một con số đếm luận lý mỗi khi một GD mới bắt đầu.
- **Định nhãn thời gian** (timestamping) là một giao thức điều khiển cạnh tranh mà trong đó mục tiêu cơ bản là sắp xếp các GD một cách toàn cục theo một cách mà các GD cũ hơn – GD với nhãn TG nhỏ hơn – sẽ được ưu tiên hơn khi có xung đột.
- Mỗi mục dữ liệu có:
  - ✦ Một **nhãn đọc** (read-timestamp): ghi nhận nhãn TG của GD cuối cùng đã đọc nó
  - ✦ Một **nhãn ghi** (write-timestamp), ghi nhận nhãn TG của GD cuối cùng đã ghi vào mục đó.

7.29



## Nguyên tắc của PP định nhãn thời gian

- **Trường hợp giao dịch T phát ra một lệnh read(x)**
  - ✦ Nếu GD T yêu cầu đọc 1 mục dữ liệu (x) đã được cập nhật bởi một GD mới hơn; nghĩa là  $ts(T) < write\_timestamp(x)$ ; thì T phải bị hủy và khởi động lại với một nhãn TG mới.
  - ✦ Ngược lại, thì thao tác đọc có thể tiến hành. Khi đó, ta đặt lại  $read\_timestamp(x) = \max(ts(T), read\_timestamp(x))$ .
- **Trường hợp giao dịch T phát ra lệnh write(x)**
  - ✦ Nếu GD T yêu cầu ghi trên một mục dữ liệu (x) mà giá trị của nó đã được đọc bởi một GD mới hơn;  $ts(T) < read\_timestamp(x)$ ; thì GD T phải bị hủy và khởi động lại với một nhãn TG mới hơn.
  - ✦ Hoặc nếu GD T yêu cầu ghi trên một mục dữ liệu (x) mà giá trị của nó đã được ghi bởi một GD mới hơn;  $ts(T) < write\_timestamp(x)$ ; thì GD T phải bị hủy và khởi động lại với một nhãn TG mới hơn.
  - ✦ Ngược lại, thao tác ghi có thể được tiến hành. Khi đó, chúng ta sẽ đặt lại  $write\_timestamp(x) = ts(T)$ .

7.30



## Giao thức điều khiển cạnh tranh lạc quan

- Một GD sẽ trải qua ba kỳ nếu là GD cập nhật, và trải qua 2 kỳ nếu là GD chỉ đọc:
  - ✦ **Kỳ đọc**: bắt đầu GD cho đến ngay trước khi commit. GD đọc các giá trị của tất cả các mục dữ liệu nó cần từ CSDL và lưu chúng vào các biến cục bộ. Các cập nhật sẽ được áp trên bản sao chép cục bộ của dữ liệu, không phải trên CSDL.
  - ✦ **Kỳ kiểm tra**: để đảm bảo tính khả tuần tự không bị vi phạm nếu các cập nhật của GD được đưa vào CSDL.
    - ✓ Đối với các GD chỉ đọc: kiểm tra xem các giá trị dữ liệu đã đọc vào các biến vẫn còn là các giá trị hiện hành trong các mục dữ liệu tương ứng.
    - ✓ Đối với một GD có cập nhật: kiểm tra liệu GD đó có đưa CSDL về một tình trạng nhất quán với tính khả tuần tự được duy trì hay không.
  - ✦ **Kỳ ghi**: Nếu kiểm tra thành công đối với các giao dịch cập nhật thì các cập nhật đã thực hiện trên biến cục bộ sẽ được chép vào CSDL.

7.31



## Giao thức điều khiển cạnh tranh lạc quan

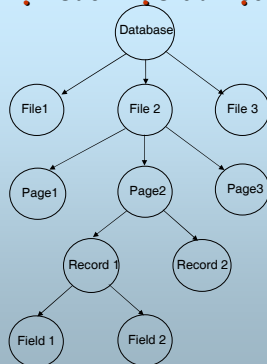
- Mỗi GD T được gán:
  - ✦ Một nhãn Start(T) ghi nhận thời điểm bắt đầu thực thi GD,
  - ✦ Một nhãn Validation(T) tại thời điểm bắt đầu vào kỳ kiểm tra,
  - ✦ Một nhãn Finish(T) ghi nhận thời điểm kết thúc.
- Để có thể thành công vượt qua kỳ kiểm tra, một GD T phải thỏa một trong các điều kiện sau:
  1. Tất cả các GD S với nhãn TG sớm hơn T phải hoàn thành trước khi GD T bắt đầu: nghĩa là  $Finish(S) < Start(T)$
  2. Nếu GD T bắt đầu trước khi một GD trước đó S kết thúc, thì:
    - a. Tập hợp các mục dữ liệu được ghi bởi GD trước đó không phải là các mục được đọc bởi giao dịch hiện tại; và,
    - b. GD trước đó phải hoàn tất kỳ ghi của nó trước khi GT hiện tại đi vào kỳ kiểm tra; nghĩa là,  $Start(T) < Finish(S) < Validation(T)$

7.32



## Độ mịn của mục dữ liệu

- Độ mịn (granularity) là kích cỡ của mục dữ liệu được chọn như là một đơn vị bảo vệ bởi giao thức điều khiển cạnh tranh.
- Mỗi khi một nút bị khóa thì tất cả các nút con cháu của nó cũng sẽ bị khóa.
- Nếu một GD khác yêu cầu một khóa không tương thích (incompatible) trên cùng một nút, thì HQTCSDL biết là không thể cấp khóa đó.



Sự phân cấp của độ mịn

7.33

## Khóa đa hạt (Multiple-granularity locking)

- Trước khi một nút nào đó bị khóa, thì một khóa Intention (I) sẽ được đặt lên tất cả các nút tổ tiên của nút đó.
- Khoá intention có thể là Shared (chia sẻ để đọc) hoặc eXclusive (độc quyền để ghi).
- Một khóa Intention Shared (IS) chỉ mâu thuẫn với một khóa độc quyền;
- Một khóa Intention eXclusive (IX) mâu thuẫn với cả khóa shared và exclusive.

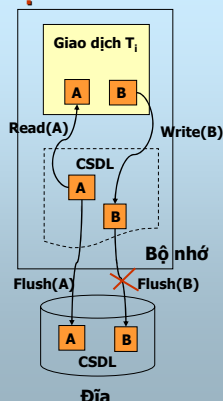
	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

Bảng thể hiện sự tương thích của các khóa

7.34

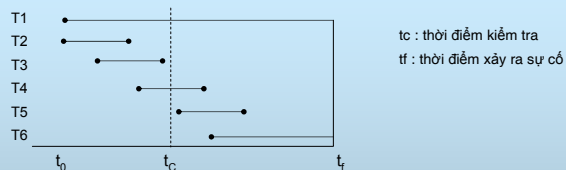
## Phục Hồi CSDL

- Chỉ khi nào các vùng đệm cho CSDL trong bộ nhớ được đẩy ra (flush) bộ lưu trữ thứ cấp thì các thao tác cập nhật mới được xem như là vĩnh cửu.
- Nếu có sự cố xảy ra giữa các lần ghi vùng đệm ra bộ lưu trữ thứ cấp, thì bộ quản lý phục hồi phải xác định tình trạng của GD đã thực hiện hành động ghi tại thời điểm xảy ra lỗi:
  - Nếu GD đã hoàn tất, thì nó phải **redo** các cập nhật của GD đó lên CSDL (điều này gọi là cuộn tiến **rollforward**).
  - Ngược lại, nếu GD chưa hoàn tất tại thời điểm xảy ra lỗi, thì nó phải **undo** (hay cuộn ngược lại **rollback**).




7.35

## Các tiện ích để phục hồi



- Cơ chế sao lưu, để tạo các bản sao chép định kỳ.
- Ghi nhật ký, để theo dõi tình trạng hiện tại của các GD và các thay đổi của CSDL.
- Điểm kiểm tra, để cho phép các cập nhật đang thực hiện trên CSDL có thể được chép ra vĩnh cửu.
- Bộ quản lý phục hồi, cho phép hệ thống phục hồi CSDL về tình trạng nhất quán sau sự cố.


7.36



### Một đoạn của tập tin nhật ký

$T_i$	Time	Operation	Object	Before Image	After Image	PPtr	NPTr
T1	10:12	START				0	2
T1	10:13	UPDATE	STAFF SL21	(old val.)	(new val.)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new val.)	3	5
T2	10:17	DELETE	STAFF SA9	(old val.)		4	6
T3	10:18	START				0	11
T1	10:18	COMMIT				2	0
	10:19	CHECK POINT	T2, T3				
T2	10:19	COMMIT				6	0
T3	10:20	INSERT	PROPERTY PG4		(new val.)	7	12
T3	10:21	COMMIT				11	0

7.37



### Điểm kiểm tra

- **Điểm kiểm tra (checkpoint)** là một điểm mà tại đó sự đồng bộ giữa CSDL và tập tin nhật ký GD được ghi nhận. Khi đó, tất cả các vùng đệm phải được ghi-ép-buộc ra bộ lưu trữ thứ cấp.
- Các điểm kiểm tra được lập lịch tại các khoảng thời gian định trước và tại điểm kiểm tra, các thao tác sau sẽ được tiến hành:
  - ✦ Ghi tất cả các mẫu tin nhật ký có trong bộ nhớ trong ra bộ lưu trữ thứ cấp
  - ✦ Ghi các khối đã được sửa đổi trong vùng đệm CSDL ra bộ lưu trữ thứ cấp.
  - ✦ Ghi một mẫu tin kiểm tra vào tập tin nhật ký. Mẫu tin này chứa các định danh của các GD đang hoạt động tại thời điểm kiểm tra.
- Khi có sự cố xảy ra:
  - ✦ Redo tất cả các GD đã hoàn tất kể từ sau điểm kiểm tra
  - ✦ Undo tất cả các GD đang hoạt động tại thời điểm xảy ra sự cố.

7.38