資訊管理學系 陳士杰老師

資料庫系統管理
Database System Management

並行控制與回復
Concurrency Control and Recovery

國立聯合大學
NATIONAL UNITED UNIVERSITY

Outlines

- 並行控制 (Concurrency Control)
- 復原 (Recovery)

【講義：Ch. 7, Section 2與Section 3】
【原文：Ch. 14】

❖注意❖

在本單元中，DML指令包含DQL指令

並行控制(Concurrency Control)

- 定義：
  - 讓多個交易可以在同一時間存取同一筆資料，稱之為並行。
  - 並行控制即讓多筆交易在並行的狀況下運作，而不會互相干擾，確保交易間的孤立性(Isolation)，以及提高交易的效率。
- 目的：
  - 多筆交易可同時進行，而不互相干擾
  - 減低交易回應時間(Response Time)
  - 增加交易產出(Throughput)

並行控制不佳可能產生的問題

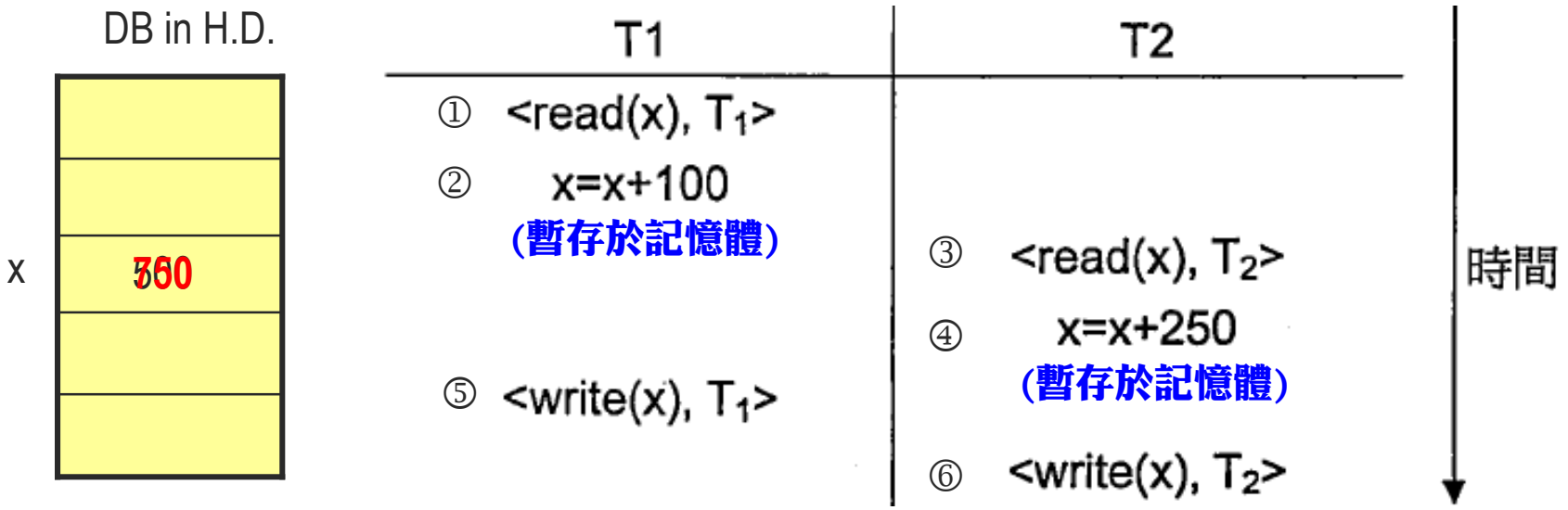
- 遺失更新(Lost Update)
- 不一致分析(Inconsistent Analysis)
- 未確認相依(Uncommitted Dependency)

遺失更新

- 當有多筆交易交錯執行，且這些交易針對**相同資料項目**作存取或異動時，可能會使此資料項目的內容值**不正確**(發生不可預期的錯誤)，稱為**遺失更新**。
  - 例：假設我有一個帳號x，其中x = 500

    (父) T1: x = x+100 ⇨ (操作：$r_1(x)$, x = x+100, $w_1(x)$)

    (母) T2: x = x+250 ⇨ (操作：$r_2(x)$, x = x+250, $w_2(x)$)

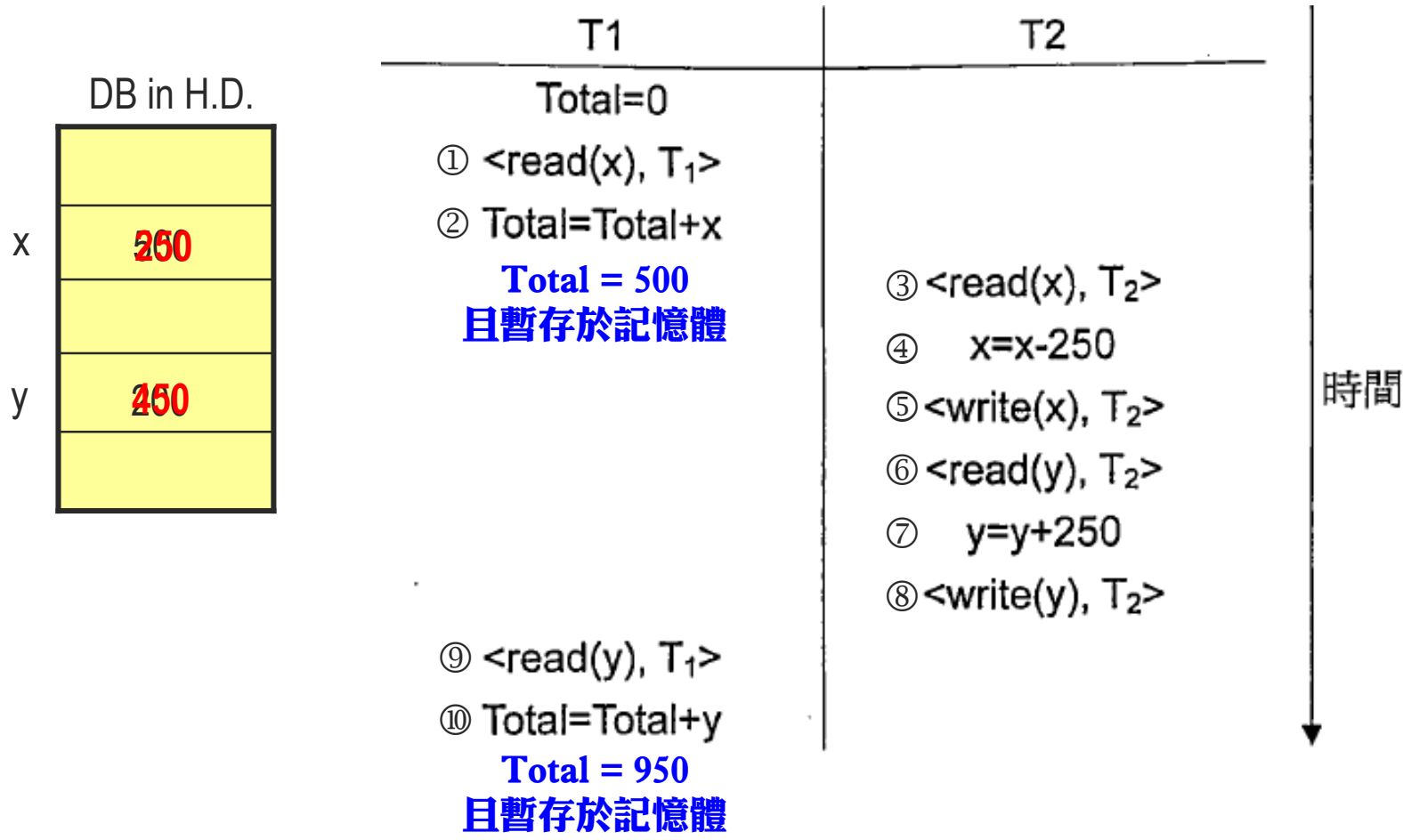


- T1對x的更新動作 (即：動作②) 遺失。
  - ∵程式執行過程中，同一個變數會被存放在相同的記憶體空間中。

- **會產生遺失更新的排程 “一定不是” 可序列化排程**
  - 可序列化排程一定要確保結果是正確的。

不一致分析

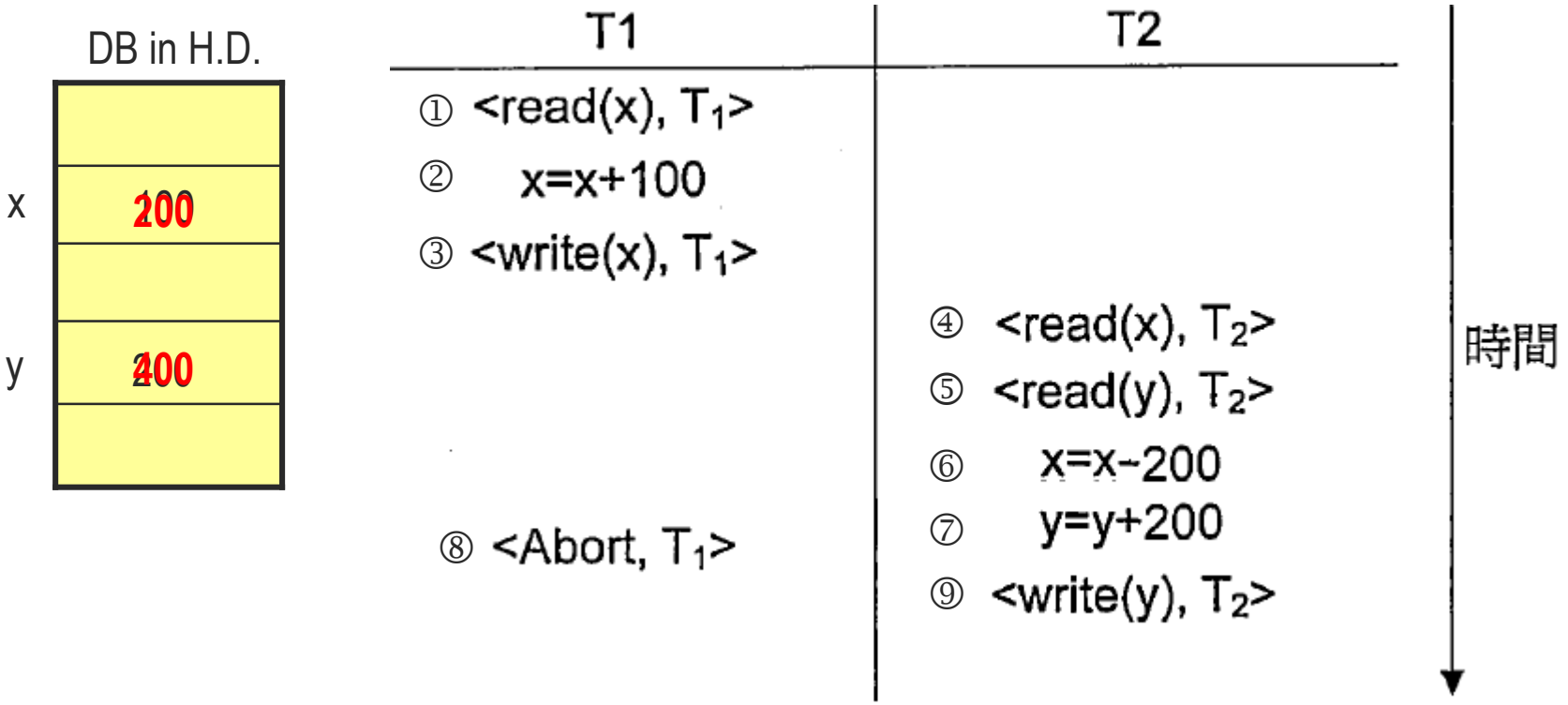
- 又稱不正確總合(Incorrect Summary)問題。
- 一筆交易正在針對某些資料項目作聚合函數(Aggregate Function)之計算，若此時其中一個資料項目被其它交易更改，造成此聚合函數計算出之結果不正確。
  - 例：假設我有兩個帳號x和y，其中x=500且y=200。現有兩個交易T1和T2：
    - T1: SUM(x, y) = x+y
      - ⇨ (操作：$r_1(x)$, T = T+x, $r_1(y)$, T=T+y) ，T初始值為0
    - T2: 轉250元從x到y
      - ⇨ (操作：$r_2(x)$, x = x-250, $w_2(x)$, $r_2(y)$, y=y+250, $w_2(y)$)



- **Total之加總動作擷取了不一致的資料(因時間點不一致所造成!!)。**

未確認相依

- 又稱Dirty Reads或Temporary Update 。
- 交易A已更新某一個資料項目，但尚未確認(Commit)時，此資料項目卻又被另一個交易B所存取，但是先前的交易A因故需要撤回(Abort/Rollback)，則交易B讀到了一個不應該讀的項目值。
  - 例：假設有兩個帳號x和y，其中x=100且y=200。現有兩個交易T1和T2：
    - (母) T1: x = x+100，但轉帳中途撤回
      - ⇨ (操作：$r_1(x)$, x = x+100, $w_1(x)$, Abort)
    - T2: 轉200元從x到y
      - ⇨ (操作：$r_2(x)$, $r_2(y)$, x = x-200, y=y+200, $w_2(x)$, $w_2(y)$)



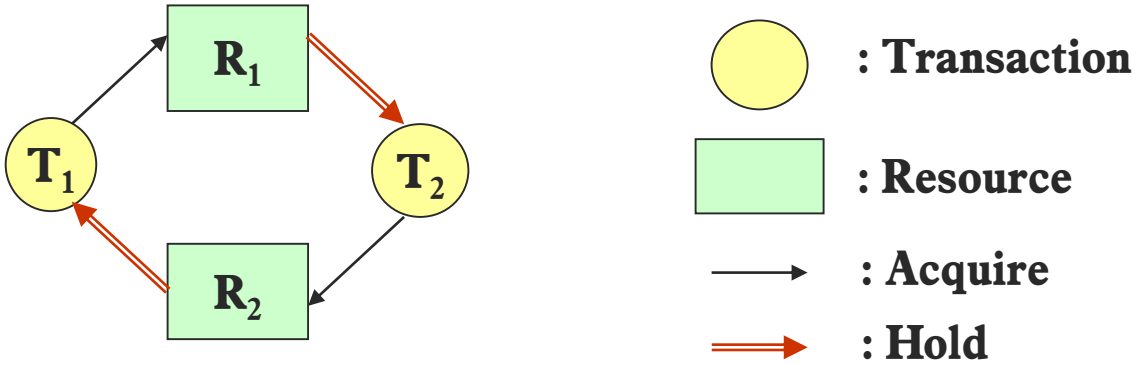
- **T_2讀取到錯誤的x值(尚未確認的x值，即Dirty data)。**

交易並行時可能會發生的問題

- 死結(Deadlock)
- 活結(Livelock)
- 餓死(Starvation)

死結(Deadlock)

- 當兩個以上的交易同時進行，彼此皆佔有某些資源，但亦企圖**奪取對方資源**時(即：陷入“**互相等待對方所擁有之資源**”)，便會發生死結的現象。
- Example
  - 假設系統中有兩個資源 R_1 與 R_2 ，而且該系統產生兩個交易 T_1 與 T_2



  - R_1 已配置給 T_2 ， R_2 已配置給 T_1 ，但這兩個交易在執行過程中又向對方要求對方正在使用中的資源，因此造成系統打結。

- 形成死結的四個條件 (缺一不可)：
  - Mutual exclusion (互斥)
    - 同一資源不可同時被兩個以上的交易共用，即資源被鎖定後，其它交易不可存取此資源。
  - Hold and wait (持有並等待)
    - 各交易皆佔有某些資源，並等候其它交易釋放其資源。
  - No preemption (不可搶奪)
    - 不可搶奪已被佔用的資源。
  - Circular waiting (循環式等候)
    - 各交易循環等待對方資源，即：優先順序圖(Precedence Graph)形成迴圈

活結(Livelock)與餓死(Starvation)

- 某交易在一段不確定的時間內無法進行，然而其它交易卻可以正常進行的情況，稱之為**活結**。
  - 舉例：
    - 若交易有**優先權**之概念，且優先權高的交易持續得到資源的授與，則優先權低的交易即可能進入活結的狀態 (一直拿不到資源)
- **餓死(Starvation)**
  - 為活結的衍生問題，特定交易可能永遠無法獲得所需資源的情況。
  - 優先權高的交易優先被執行，並不斷有新的高優先權交易進入，造成優先權最低的交易被選為Victim(犧牲者)，永遠無法執行。

餓死問題之解決

- **等-死方法** (wait-die):
  - 當交易Ti要求一個正被Tj持有的資源時，只要**Ti的時間戳記比Tj小**(即：Ti比Tj早進入資料庫系統)，Ti就被允許等待。否則的話，Ti就被撤回。
    - 較早進入的交易可等待，較晚進入的交易被撤回。
    - 撤回動作較多。
  - 例如，假設 T1、T2 與 T3 的時間戳記分別為 5、10 和 15。如果T1要求一個T2所持有的資源，T1將會等待。如果T3要求一個T1交易所持有的資源，T3將被撤回。
  - 以**不可搶奪(No preemption)**為基礎

- **傷-等方法** (wound-wait scheme):
  - 是等-死方法的相反。當交易Ti要求一個正被Tj持有的資源，如果Ti的時間戳記比Tj大(即：Ti比Tj晚進入資料庫系統)，那麼Ti就被允許等待。否則，Ti就把Tj的資源給搶奪過來，Tj就被撤回。
    - 較早進入的交易可搶較晚進入交易之資源，而較晚進入的交易則需等待。
    - 撤回動作較少。
  - 以可搶奪(Preemption)為基礎
- 上述兩種方法皆可避免飢餓情況，因為若撤回的交易繼續使用原本的時間戳記並重新要求資源，則其時間戳記終究會成為系統中最小的一個 。

並行控制的技術

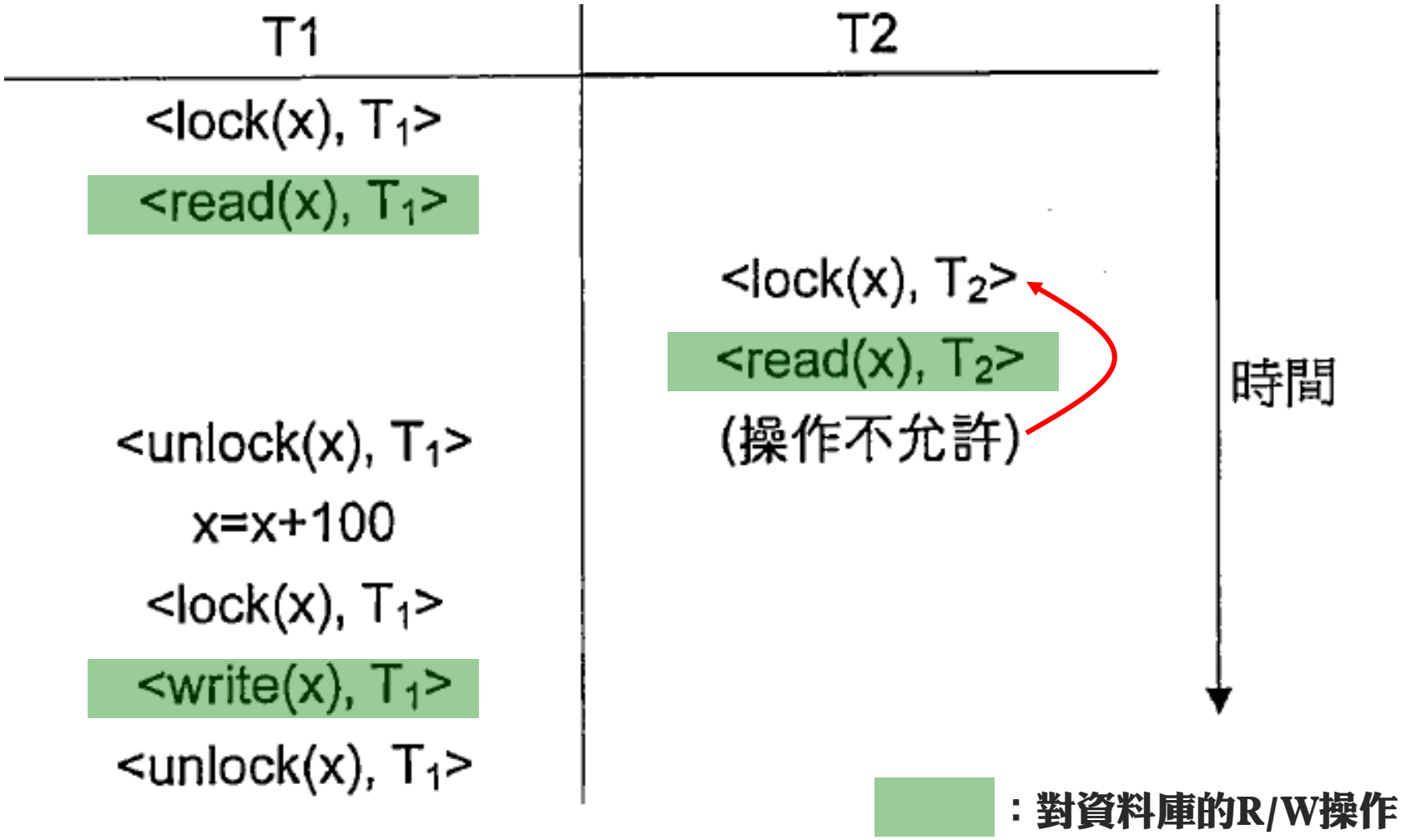
- **鎖定(Locking)**
  - 當某交易欲存取特定資料項目時，必須將此資料**鎖住(Lock)**，直到存取完畢才解除鎖定(Unlock)。
  - 若其它交易欲存取被鎖定的資料項目時，需等待至其解除鎖定。
- **時間戳記(Timestamp)**
  - 每個交易依其進入資料庫系統的時間先後，給予一個**時間戳記**，以時間戳記的大小來判交易之**優先權(Priority)**。
  - 若有交易欲存取某資料項目時，必須更新**存取此資料項目的時間戳記**。
- **多重版本並行控制(Multi-version Concurrency Control)**
  - 以時間為基礎，會在每次資料項目被修改時，**保留舊的版本**。所以在並行執行時，會自動選擇合適的版本資料以避免不一致的情況。
  - 較浪費空間。
- **樂觀並行控制(Optimistic Concurrency Control)**
  - 假設所有交易皆可順利且正確地進行，所以**不須在交易期間作任何的檢查**，直到交易完成才從事檢查。
  - 若交易違反可序列化(Serializable)的結果，則選擇**影響較小**之交易作撤回(Abort)。

鎖定(Locking)

- 鎖定是利用一個與資料庫中資料項目相結合的變數，以描述資料項目的狀況，並決定何種動作允許或不允許應用在此資料項目上。
- 並行控制的鎖定技術主要有：
  - 二元鎖定 (Binary Locking)
  - 共享互斥鎖定 (Shared and Exclusive Locking)
  - 兩階段鎖定 (Two Phase Locking)

二元鎖定 (Binary Locking)

- 概念：
  - 交易欲存取資料項目x時，即將x鎖定，直到存取完畢，再解除對x的鎖定。
- 所以，二元鎖定技術對於被交易使用到的資料項目 x 分成兩種狀態：
  - **鎖定(Lock)**：若資料項目x被鎖定，表示此資料項目x不能被其它交易所存取。
  - **解除鎖定(Unlock)**：若資料項目x解除鎖定，表示此資料項目x可以被其它交易所存取。


T1
T2
<lock(x), T1>
<read(x), T1>
<lock(x), T2>
<read(x), T2>
(操作不允許)
時間
<unlock(x), T1>
x=x+100
<lock(x), T1>
<write(x), T1>
<unlock(x), T1>
：對資料庫的R/W操作

- 二元鎖定的缺點：
  - 不能保証排程為可序列化 (Serializable) (∴所以不保証排程中所有交易執行後的結果一定是正確的。)
  - 可能產生死結 (Deadlock)
  - 可能產生活結(Live Lock)或餓死 (Starvation)狀態
  - 並行程度不佳

共享互斥鎖定 (Shared and Exclusive Locking)

- 由於二元鎖定過於嚴格，且並行程度不佳：
  - 事實上，若有多筆交易同時欲存取資料項目x，但皆只是從事**讀取(Read)**的動作時，沒有必要將此資料項目完全鎖定；只有在從事**寫入(Write)**動作時，才需要將此資料項目x獨佔式地鎖定。
- 共享互斥鎖定又稱**多元鎖定(Multi-Mode Locking)**，將二元鎖定中的Lock分為**共享鎖定 (Share Lock)**與**互斥鎖定 (Exclusive Lock)**兩種，較二元鎖定有彈性。

T1 \ T2	Read Lock	Write Lock
Read Lock	共享	互斥
Write Lock	互斥	互斥

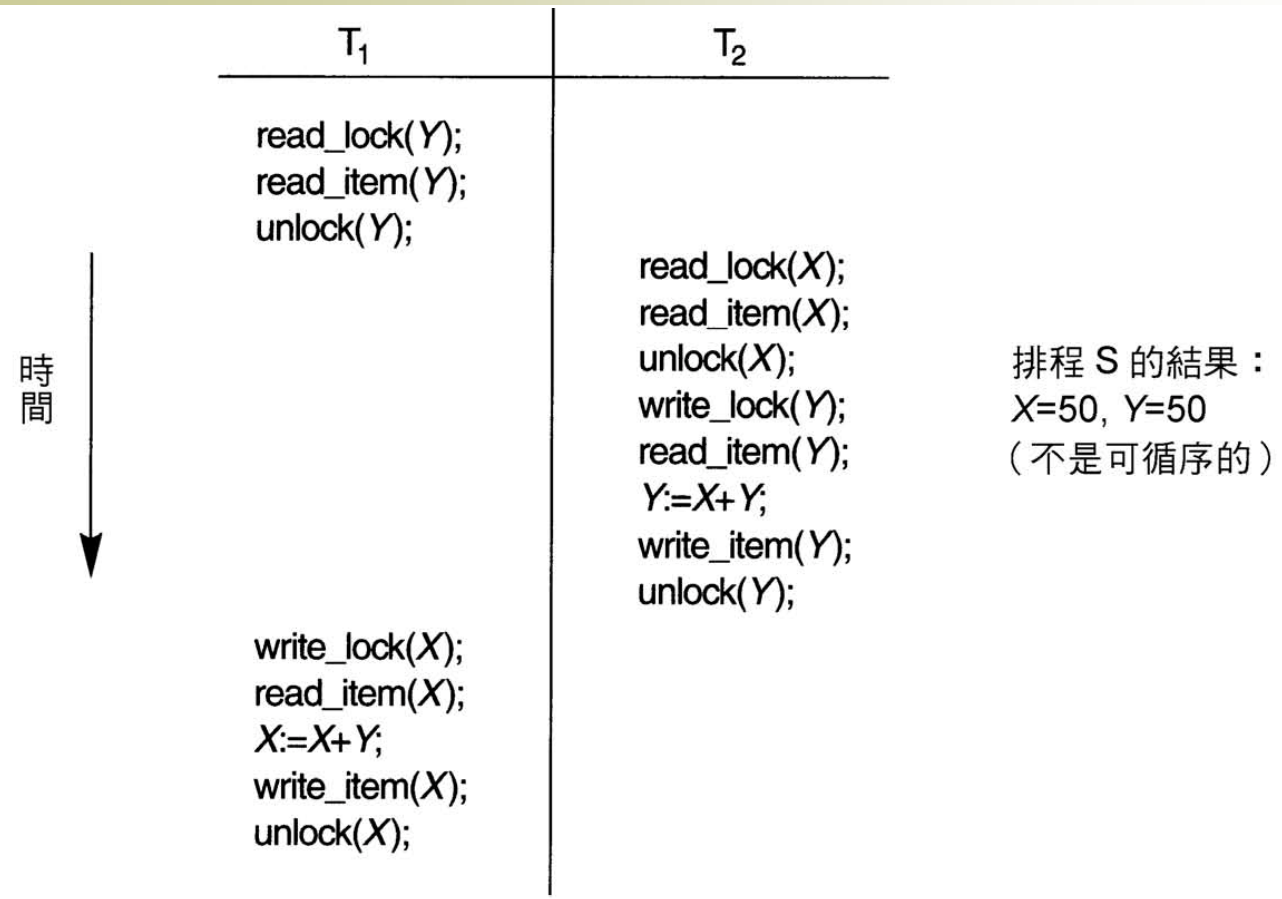
- 共享互斥鎖定技術對於被交易使用到的資料項目 x 分成三種狀態：
  - **共享鎖定(Shared-Lock)**：又稱**讀取鎖定(Read-Lock)**。若資料項目x被共享鎖定，表示此資料項目x仍可被其它交易所**讀取**。
  - **互斥鎖定(Exclusive-Lock)**：又稱**寫入鎖定(Write-Lock)**。若資料項目x被互斥鎖定，表示此資料項目x不能被其它交易讀取或寫入，只有該交易本身可獨佔使用此資料項目。
  - **解除鎖定(Unlock)**：若資料項目x解除鎖定，表示此資料項目x可以被其它交易所存取。

- **鎖定的升級與降級：**
  - **升級(Upgrade)**：Unlock→Read Lock→Write Lock
    - 即由Unlock→Read Lock， Unlock→Write Lock或Read Lock→Write Lock的狀態。
  - **降級(Downgrade)**：Write Lock → Read Lock→Unlock
    - 即由Read Lock→Unlock， Write Lock →Unlock或Write Lock → Read Lock的狀態。
- **共享互斥鎖定的缺點：**
  - 不能保証排程為可序列化 (Serializable) (∴所以不保証排程中所有交易執行後的結果一定是正確的。)
  - 可能產生死結 (Deadlock)
  - 可能產生活結(Live Lock)或餓死 (Starvation)狀態

共享互斥鎖定範例

- 假設有兩個交易T1與T2，若分開執行可得到正確之結果：

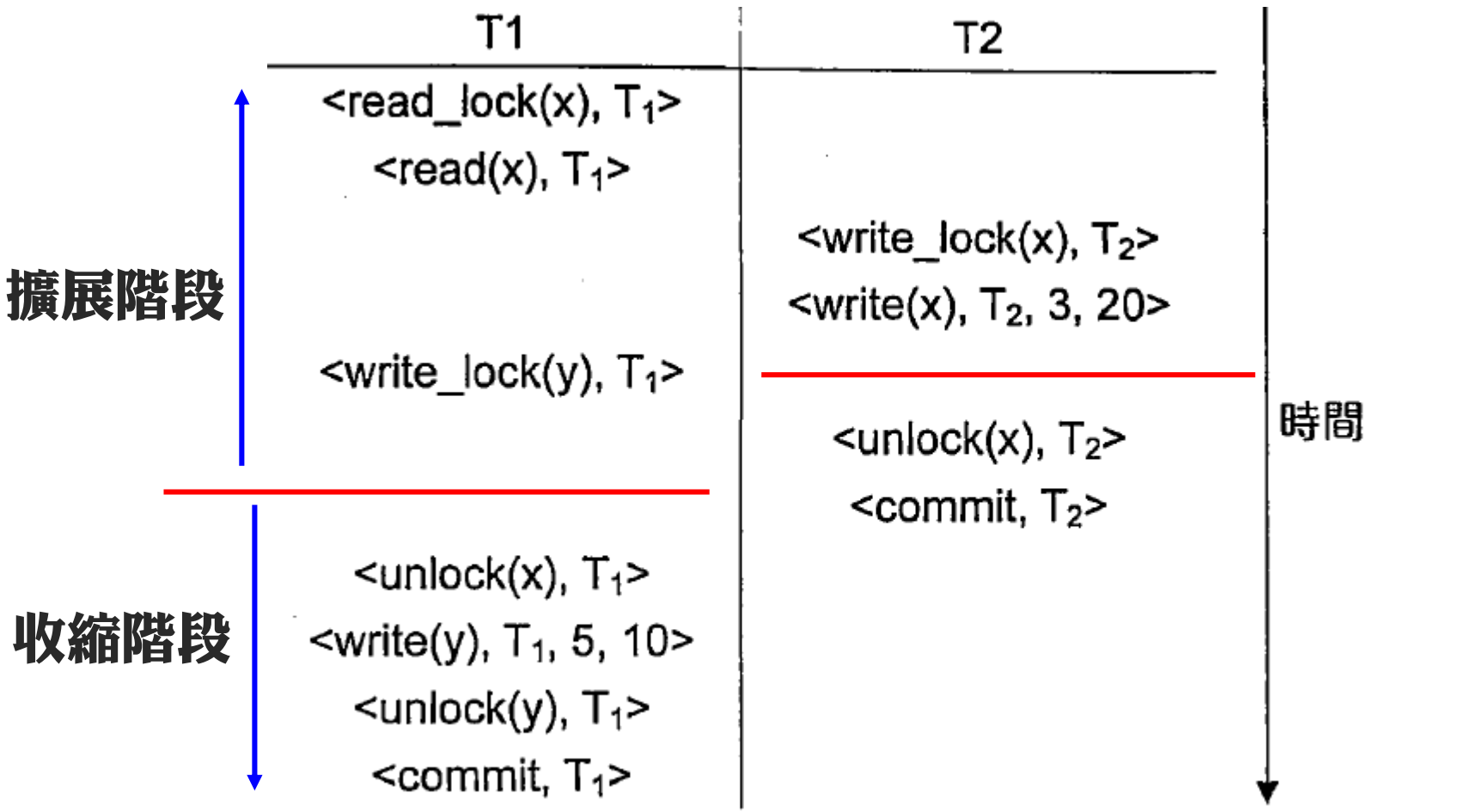
T1	T2	結果
read_lock (Y);	read_lock (X);	初值：X=20; Y=30
read_item (Y);	read_item (X);	
unlock (Y);	unlock (X);	循序排程先T1再T2 的結果：
write_lock (X);	Write_lock (Y);	X=50, Y=80
read_item (X);	read_item (Y);	
X:=X+Y;	Y:=X+Y;	循序排程先T2再T1 的結果：
write_item (X);	write_item (Y);	X=70, Y=50
unlock (X);	unlock (Y);	



(a) 兩個交易T_1和T_2；(b) T_1和T_2的循序排程結果；
(c) 使用鎖定機制的非循序排程 S

兩階段鎖定 (Two Phase Locking; 2PL)

- 2PL限定每一個交易中**所有的鎖定動作(含升級)，必須在所有解除鎖定(含降級)動作之前**。
- 因此，2PL分成兩個階段：
  - **擴展階段(Expanding)**：
    - 又稱**成長階段(Growing)**，擴展階段允許加入新的鎖定或升級動作，但不允許解除任何鎖定。
  - **收縮階段(Shrinking)**：
    - 又稱**釋放階段(Releasing)**，收縮階段允許解除現存鎖定或降級動作，但不允許任何新的鎖定加入。



註：此排程服從 2PL，所有鎖定動作皆在所有解除鎖定之前。

- 2PL的種類：
  - 基本型 (basic) 2PL：交易會逐漸鎖定資料項目
    - 可能會有死結
  - 保守型 (conservative)2PL：在交易執行前，先鎖定所有需要的資料項目。若有任何資料項目無法鎖定，則全部不鎖，直到可完全鎖定所有資料項目再開始交易。
    - 不太會有死結
    - 擴展階段短，收縮階段長
  - 嚴格型 (strict)2PL：交易過程中，不會解除任何鎖定，直到交易結束(Commit或Rollback)時，再一次釋放所有鎖定。
    - 可能會有死結
    - 擴展階段長，收縮階段短

- **2PL的優點：**
  - **遵守2PL要求而執行完畢的排程，保証皆為可序列化排程**
- **2PL的缺點：**
  - **可能產生死結 (Deadlock)**
  - **可能產生活結(Live Lock)或餓死 (Starvation)狀態**
  - **2PL可能保持某個鎖定過久，造成其它交易無法順利進行**

時間戳記排序法(Timestamp Ordering)

- 時間戳記是根據交易被送到資料庫系統的時間所指定出來的，所以可將時間戳記當作是交易開始時間。
  - 在此，將交易T的時間戳記表示為TS(T)。
  - 以時間戳記為基礎的並行控制技術不需要鎖定資料項目，所以不會造成死結。
- 定義：
  - 由資料庫系統指定給每一個交易一個獨特的唯一識別碼(Identifier)，以辨認各個交易的順序。
  - 通常時間戳記值會依照交易進入系統的順序排列，所以可以想像成交易開始的記號。
  - 時間戳記一般利用計數器 (Counter)產生流水號給各個交易，或是讀取系統時鐘 (Clock) 值給各個交易。

- 時間戳記並行控制技術，針對每一個資料項目會使用到兩種不同的時間戳記：
  - Read-TS(x)：
    - 資料項目x的讀取時間戳記，表示所有已成功讀取資料項目x的交易中，時間戳記最大的 。
    - 例如：Read-TS(x) = TS(T_i)，則代表交易T_i是所有成功讀取資料項目x的交易中，最年輕的交易。
  - Write-TS(x)：
    - 資料項目x的寫入時間戳記，表示所有已成功寫入資料項目x的交易中，時間戳記最大的 。
    - 例如： Write-TS(x) = TS(T_i)，則代表交易T_i是所有成功寫入資料項目x的交易中，最年輕的交易。

基本時間戳記排序法

- 一個交易T不論是執行read或是write時，均會產生一個時間戳記TS(T)。
- 交易T執行一個write x動作：
  - 若Read-TS(x) > TS(T) 或 Write-TS(x) > TS(T)，則write操作不允許，並將交易T取消並撤回 (Abort/Rollback)。
    - 理由：這是因為有一些比交易T晚進入系統的其它交易，在交易T進行寫入資料項目 x 的操作時，已對 x 做過讀取或寫入!!若允許交易T的寫入，將造成資料不一致的現象。
  - 若無上述狀況，則執行交易T的write操作，並設定write-TS(x) = TS(T)

- **交易T執行一個read x動作：**
  - **若 Write-TS(x) > TS(T)，則read操作不允許，並將交易T取消並撤回 (Abort/Rollback)。**
    - **理由：這是因為有一些比此交易T晚進入系統的其它交易，在交易T進行讀取資料項目 x 的操作時，已對 x 做過寫入!!**
  - **若Write-TS(x) ≤ TS(T) ，則執行交易T的read操作，並設定read-TS(x) = Max[TS(T), 原本 read-TS(x)]**

以時間戳記排序法避免遺失更新問題

- 例：假設我有一個帳號x，其中x = 500

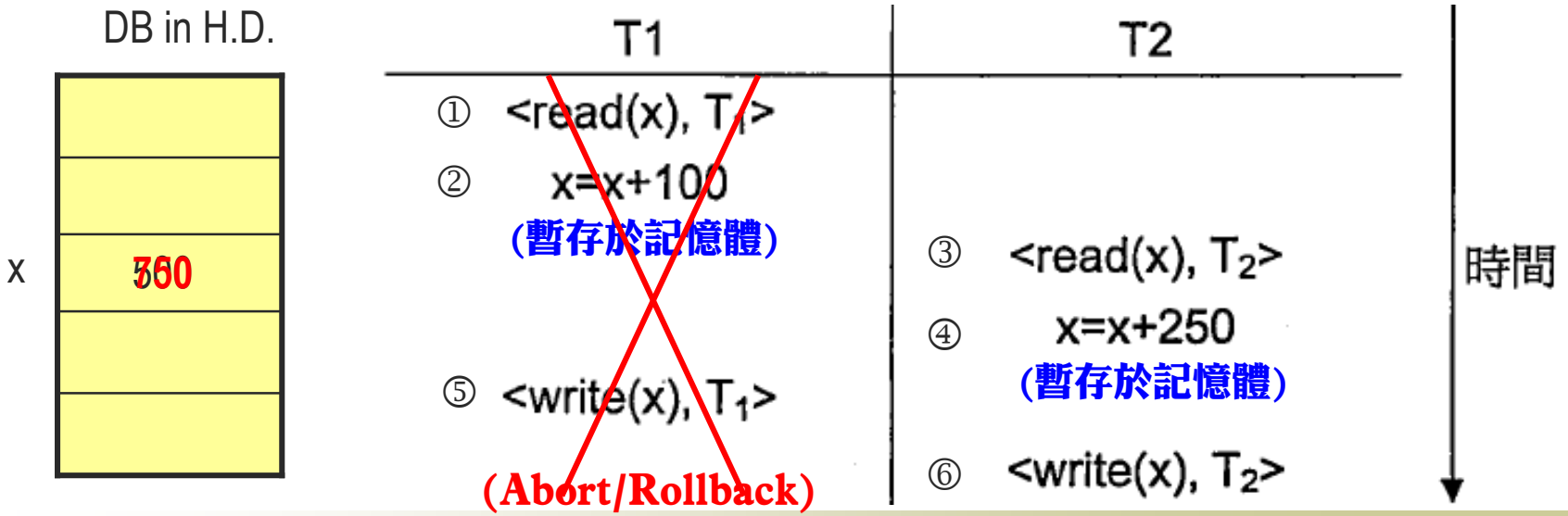
(父) T1: x = x+100 ⇨ (操作：$r_1(x)$, x = x+100, $w_1(x)$)，且TS(T1) = 3

(母) T2: x = x+250 ⇨ (操作：$r_2(x)$, x = x+250, $w_2(x)$)，且TS(T2) = 8

針對帳號 x，用到兩個時間戳記read-TS(x)與write-TS(x)：

操作時間點	初始						
read-TS(x)							
write-TS(x)							

交易T1在
時間點⑤被撤回



- **時間戳記並行控制優點：**
  - 保証排程皆為**可序列化(Serializable)**
  - 避免死結(Dead lock)發生
  - 結合wait-die或wound-wait方法可避免活結 (Live lock) 或餓死 (Starvation)發生。
- **時間戳記並行控制缺點：**
  - 可能產生**連鎖性撤回 (Cascading Rollback)**，降低執行效率。

樂觀並行控制(Optimistic Concurrency Control)

- 先前討論過的並行控制技術中，於資料庫動作執行之前都必須完成一定程度的檢查，例如：
  - **鎖定機制**需要先檢查資料項目 x 目前的鎖定狀態。
  - **時間戳記**作法必須檢查交易的時間戳記。
- 但是，過多的檢查步驟會使得**交易的執行效率降低**。
- 樂觀並行控制技術的觀念：
  - 假設所有交易皆可順利且正確地進行，因此在交易執行過程中**不需要作任何的檢查動作**。
  - 交易執行時，任何對資料項目的變更操作 (即：Write操作) 並非直接對資料庫進行修改，而是**對該資料項目的複製版本(Copy)進行修改，等到交易結束時再去檢查這些變更操作是否符合可序列化特性**。
  - 確認符合可序列化特性時，再從副本**變更到資料庫**中；反之，則交易將被**撤回(Abort)**

- 樂觀並行控制技術有3個階段：
  - 讀取 (Read)：交易可以從資料庫中讀取資料項目，並同時產生此資料項目的複本。而變更資料項目的操作只會修改到此資料項目的複本。
  - 確認 (Validation)：執行檢查，以確保交易修改到資料庫的操作沒有違反可序列化。
  - 寫入 (Write)：如果認階段成功，交易的更新資料會被寫入資料庫；否則將取消變更操作，且交易被撤回。
- 樂觀並行控制技術適用於交易甚少發生不一致的情況，可提升交易的執行效率。
  - 樂觀並行控制在確認階段一次做完所有的檢查，可以讓花在檢查上的時間最少。如果交易之間幾乎沒有不一致情況，那麼大部份的交易將會被成功確認。
  - 然而，若有太多的交易造成不一致的衝突，會造成許多執行完的交易必須放棄它們的結果而被撤回。

- **樂觀並行控制的優點：**
  - **保証排程皆為可序列化**
  - **避免死結發生**
  - **結合Wait-die或Wound-wait方法可避免活結或餓死發生**
  - **不一致情況甚少發生時，可加快系統執行效率**
- **缺點：**
  - **若經常發生不一致的狀況，樂觀並行控制會產生大量的撤回或復原工作。**

多重版本並行控制(Multi-version Concurrency Control)

- 以時間為基礎，當每回資料項目被修改時**保留舊的版本**，所以在多個交易並行執行時，會自動選擇合適的版本以避免不一致的情況。
- 優點：
  - 保証排程皆為可序列化
  - 避免死結發生
  - 結合Wait-die或Wound-wait方法可避免活結或餓死發生
  - 加快讀取及寫入速度
- 缺點：
  - 可能產生**連鎖性撤回(Cascading Rollback)**，降低執行效率
  - 需要較多**記憶體**，以維持每個資料項目的多個版本

並行控制技術的比較

	Serializable	Dead Lock	Live Lock	Starvation
二元鎖定	不一定	可能發生	可能發生	可能發生
共享互斥鎖定	不一定	可能發生	可能發生	可能發生
兩階段鎖定	是	可能發生	可能發生	可能發生
時間戳記	是	可避免	結合Wait-die、Wound-wait則可避免	結合Wait-die、Wound-wait則可避免
多重版本	是	可避免	結合Wait-die、Wound-wait則可避免	結合Wait-die、Wound-wait則可避免
樂觀並行控制	是	可避免	結合Wait-die、Wound-wait則可避免	結合Wait-die、Wound-wait則可避免

復原 (Recovery)

- 定義：
  - 復原是指在資料庫系統發生失敗(failure)後，重新回到一個已知的正確狀態的方法。
- 失敗的種類：
  - 非毀滅性故障
  - 毀滅性故障

- **非毀滅性故障：**
  - 一般是指導致主記憶體資料內容消失，但永久性儲存媒體中的資料仍完好的狀態。可能的情況如下：
    - **交易或系統錯誤 (transaction or system error)**：在交易執行時的部份操作可能導致故障，例如：溢位、除以零或是使用者強行中斷交易。
    - **交易偵測到例外狀況(exception condition)**：程式偵測到一些例外狀況，可能導致交易無法繼續執行，例如：找不到所需的值。
    - **強制實行並行控制(concurrency control enforcement)**：交易違反序列性(Serializability)或發生死結(Deadlock)時，系統可能會主動要求交易撤回(Abort/Rollback)。
    - **電腦失敗(Computer failure)**：電腦系統本身發生的硬體或軟體錯誤，造成程式中斷或當機等狀況，以致於記憶體內資料消失。
  - 復原方式：
    - 利用**系統日誌(System Journal, System Log)**從事復原。

- **毀滅性故障：**
  - 一般指硬體上的故障，又稱為媒體故障(Media failure)、硬體當機(Hard crash)。可能情況如下：
    - **磁碟失敗(Disk failure)**：交易中遇到磁碟軌損壞、讀寫頭損毀、磁碟錯誤讀寫等錯誤，造成硬體資料的損失。
    - **實體問題與大災難(Physical Problems and Catastrophes)**：交易中可能遭遇硬體被蓄意破壞、火災、水災...等嚴重故障導致失敗。
  - 復原方式：
    - 利用磁帶、磁碟、光碟等**定期備份的資料**從事復原動作。

復原的基本方法

- 針對非毀滅性故障：
  - 系統運作時，將所有改變資料項目的操作寫入系統日誌(System Journal)中，並將此系統日誌儲存於永久性儲存媒體。
  - 由於非毀滅性故障並未真正損毀實體磁碟上的系統日誌或資料庫，因此只要針對系統日誌中的紀錄進行復原的動作即可。
- 針對毀滅性故障：
  - 系統日常運作時，即定期將資料庫的內備份(Backup、Dump)至其它儲存媒體，並將此備份資料與日常資料庫放置於不同地點。
  - 發生毀滅性故障時，將備份資料回存至資料庫中，並將此期間的系統日誌進行重做(Redo)，以獲得正確完整的資料庫內容。

交易回復技術

- 延遲更新 (Deferred Update)
  - 採用No-Undo/Redo演算法
- 立即更新 (Immediate Update)
  - 採用Undo/Redo演算法
  - 採用Undo/No-Redo演算法
- 投影 (Shadowing/Shadow Paging)
  - 採用No-Undo/No-Redo演算法
- Undo/Redo回顧：
  - Undo：交易尚未Commit時，因某些因素將部份的資料變更寫入到H.D.中的資料庫，此時若系統失敗欲復原時，須將此部份的資料變更徹消並回復。
  - Redo：交易已Commit但尚未將資料變更寫入到H.D.中的資料庫時，若系統失敗欲復原，則須將此部份的資料變更動作重作，以確保變更可進入到資料庫中。

延遲更新 (Deferred Update)

- 觀念：延遲任何真正寫入資料庫的更新，直到交易成功的完成並且**到達確認點(Commit Point)**為止。
  - 當一個交易**尚未執行到確認點(Commit Point)之前**，其所有對資料庫資料項目之更改操作，皆紀錄在主記憶體的**工作區塊(Block)**或是**系統日誌(System Journal)**中，而不會真正修改到資料庫的內容。
  - 當交易**到達確認點時**，交易操作紀錄會強制寫入到磁碟上的系統日誌，並保証此交易**將會真正改變資料庫的內容**。

延遲更新下，不同時間點之交易處理

- **交易未Commit前：**

- **已Commit，但尚未到達Checkpoint：**

- **已Commit，且已到達Checkpoint：**

- **交易未Commit前：**

復原工作

No-Undo (無須處理)

- **已Commit，但尚未到達Checkpoint：**

Redo

- **已Commit，且已到達Checkpoint：**

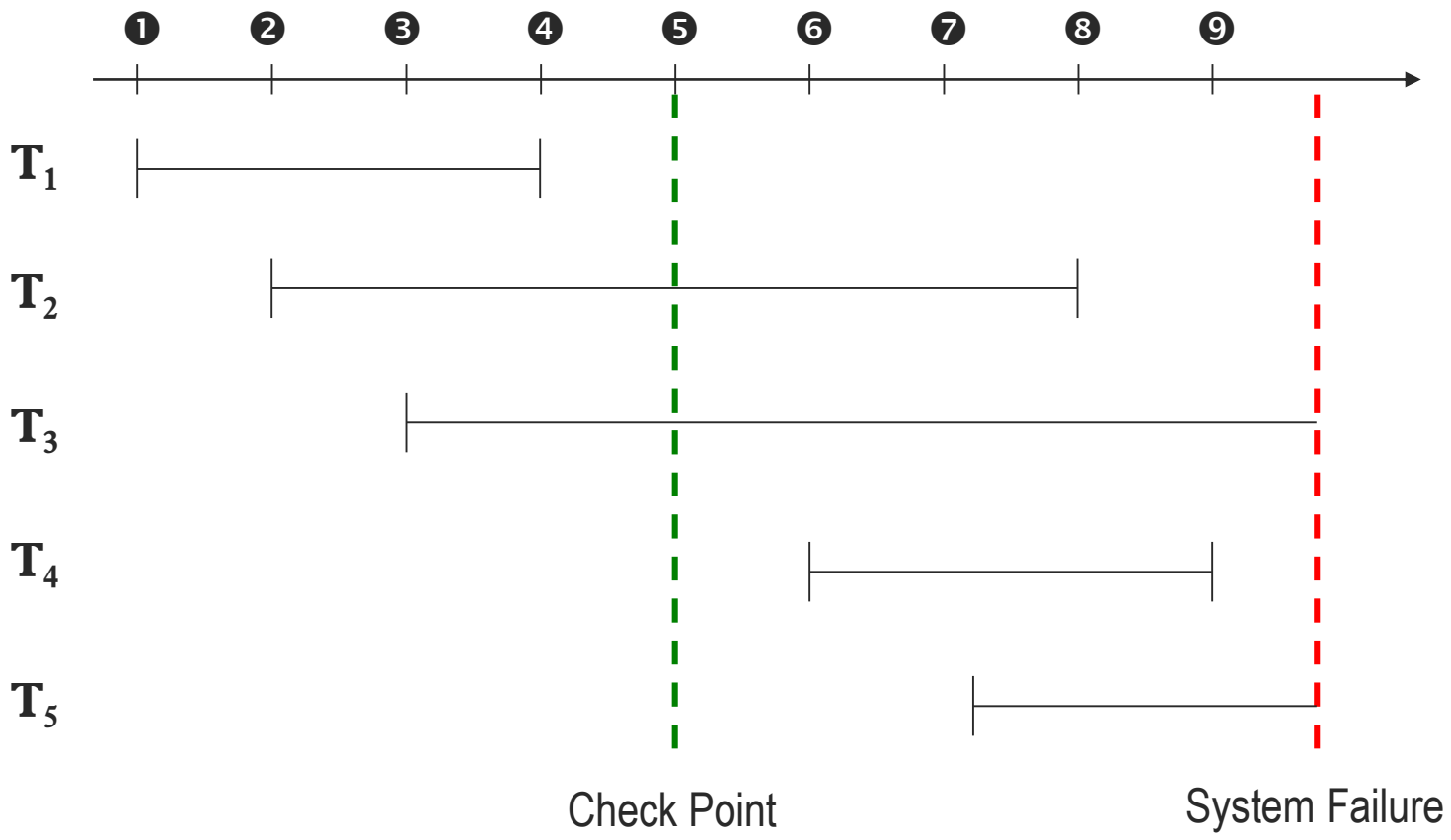
無須處理

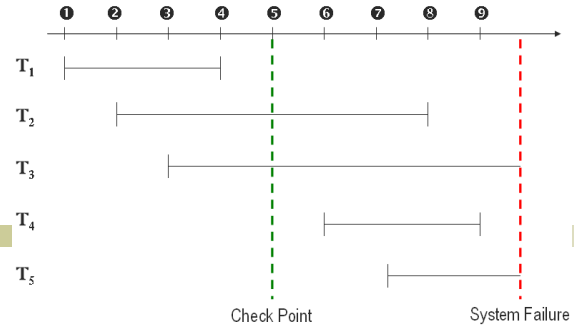
- 由於尚未到達確認點的交易，所有的更新操作都不會影響到資料庫的內容。因此若交易在確認點之前發生錯誤，則此交易**不需做Undo**的工作，可忽略此交易之復原動作。
- 到達確認點的交易若發生錯誤，需**進行Redo**的工作，以確認此交易真正改變資料庫的內容。
  - Redo工作必須是等冪(Idempotent)的，即：無論交易操作被執行幾次，其結果皆等同於只執行一次。
- 由於延遲更新在當系統發生非毀滅性故障時，不需進行Undo工作，只需進行Redo工作，因此這種技術又被稱為**No-Undo/Redo**演算法。

實作延遲更新的復原程序

- 建立作用中(Active)與未確認(Uncommitted)的**Undo串列**，以及建立最後一個檢查點之後已確認(Committed)的**Redo串列**。
  - **某交易開始**時，將此交易記載於**Undo串列**；
  - **某交易Commit**時，將此交易自Undo串列移到**Redo串列**；
  - **到達Check Point**時，**清空Redo串列**。
- 系統發生錯誤後，**Undo串列中的交易操作將被忽略取消**，不需從事Undo工作。
- 系統發生錯誤後，從Redo串列取出**已確認的交易操作**，並**重新執行（Redo）**此串列中的寫入(Write)動作。

- **範例：針對五個交易之延遲更新：**





- **Ans:**
  - **建立Undo/Redo串列：**
  - **系統失敗前 (時間 9)**
    - **Undo串列：{T_3, T_5}**
    - **Redo串列：{T_2, T_4}**

時間	Undo串列	Redo串列
1	T_1	
2	T_1, T_2	
3	T_1, T_2, T_3	
4	T_2, T_3	T_1
5	T_2, T_3	(清空)
6	T_2, T_3, T_4	
7	T_2, T_3, T_4, T_5	
8	T_3, T_4, T_5	T_2
9	T_3, T_5	T_2, T_4

  - **因此，系統失敗後：**
    - **Redo串列中的T_2, T_4交易之寫入操作，將被重新執行(Redo)**
    - **Undo串列中的T_3, T_5交易操作被忽略，不需執行任何動作(No-Undo)**
    - **在檢查點之前就已Committed的T_1交易，不需執行任何動作**

延遲更新的缺點

- 當系統日誌內的資料量龐大時，十分浪費時間。
  - ∵Redo動作費時。
- 經常進行一些不必要的動作，例如大部份的更新可能已被寫入資料庫中，卻**重複執行Redo**動作。
  - 已Committed但尚未到達Checkpoint的資料，有可能已寫入資料庫中。
  - 可結合**Checkpoint**以減少或解決此一問題。
    - ∵一旦發生Check point，表示此point之前已Committed的資料已確實寫入資料庫中，所以不用Redo。

立即更新(Immediate Update)

- 以立即更新為基礎的回復技術，主要是當交易發出一個資料更新(Write)命令時，位於硬碟上的**系統日誌**或**資料庫**可以立即被強制更新，不需要等待交易到達其確認點。
- 因此，交易若在未到達確認點前失敗，需進行**Undo**的動作，以復原到先前的狀態。
- 立即更新的回復技術分成兩種類型，視**立即更新到何種程度**來決定：
  - **Undo/Redo演算法**：交易的所有資料項目改變的操作，在確認點之前會強制寫入到**永久儲存媒體(系統日誌)**中。
  - **Undo/No-Redo演算法**：交易的所有資料項目改變的操作，在確認點之前會強制寫入到**資料庫**中。

立即更新下，不同時間點之交易處理(Undo/Redo)

- **交易未Commit前：**

- **已Commit，但尚未到達Checkpoint：**

- **已Commit，且已到達Checkpoint：**

- **交易未Commit前：**

復原工作

Undo

- **已Commit，但尚未到達Checkpoint：**

Redo

- **已Commit，且已到達Checkpoint：**

無須處理

- 由於尚未到達確認點的交易之更新操作，**有可能會影響到資料庫的內容**。因此若交易在確認點之前發生錯誤，則此交易**需要做Undo**的工作。
- 到達確認點的交易若發生錯誤，**需進行Redo**的工作，以**確認此交易真正改變資料庫**的內容。
  - Redo工作必須是等冪(Idempotent)的，即：無論交易操作被執行幾次，其結果皆等同於只執行一次。
- 由於此類立即更新在當系統發生非毀滅性故障時，需進行Undo與Redo工作，因此這種技術又被稱為**Undo/Redo演算法**。

立即更新下，不同時間點之交易處理 (Undo/No-Redo)

- **交易未Commit前：**

- **已Commit，但尚未到達Checkpoint：**

- **已Commit，且已到達Checkpoint：**

- **交易未Commit前：**

復原工作

Undo

- **已Commit，但尚未到達Checkpoint：**

No-Redo
(無須處理)

- **已Commit，且已到達Checkpoint：**

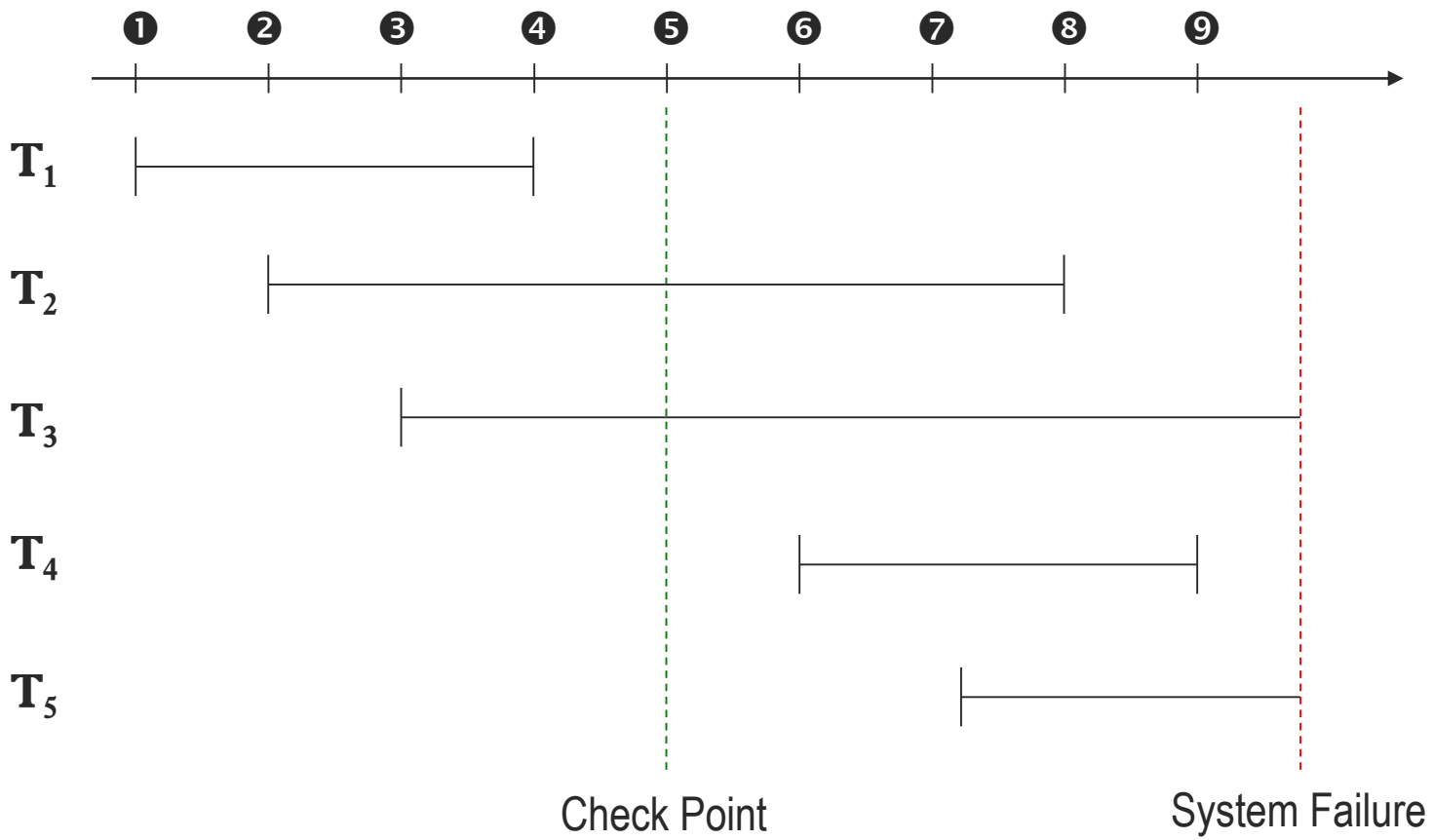
無須處理

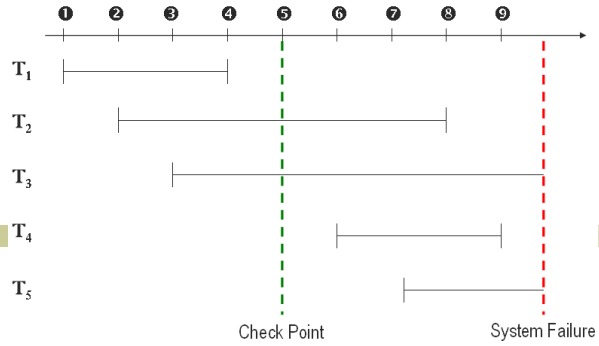
- 由於尚未到達確認點的交易之更新操作，會影響到**資料庫的內容**。因此若交易在確認點之前發生錯誤，則此交易需要做Undo的工作。
- 到達確認點的交易若發生錯誤，不需進行Redo的工作。(∵此交易的更新操作已經改變資料庫的內容。)
- 由於此類立即更新在當系統發生非毀滅性故障時，需進行Undo工作，但是不需要進行Redo工作，因此這種技術又被稱為**Undo/No-Redo**演算法。

立即更新的復原程序

- 建立作用中與未確認的Undo串列，以及建立最後一個檢查點之後已確認的Redo串列。
  - 某交易開始時，將此交易記載於Undo串列；
  - 某交易Commit時，將此交易自Undo串列移到Redo串列；
  - 到達Check Point時，清空Redo串列。
- 系統發生錯誤後，Undo串列中的交易操作將被反向進行以從事Undo工作。
- 系統發生錯誤後，
  - 從Redo串列取出已確認的交易操作，並重新執行(Redo)此串列中的寫入(Write)動作。(Undo/Redo演算法)
  - Redo串列中的交易操作將被忽略取消，不需從事Redo工作。(Undo/No-Redo演算法)

- **範例：針對五個交易之立即更新(Undo/Redo演算法)：**





- **Ans:**
  - **建立Undo/Redo串列：**
  - **系統失敗前 (時間 9)**
    - **Undo串列：{T_3, T_5}**
    - **Redo串列：{T_2, T_4}**

時間	Undo串列	Redo串列
1	T_1	
2	T_1, T_2	
3	T_1, T_2, T_3	
4	T_2, T_3	T_1
5	T_2, T_3	(清空)
6	T_2, T_3, T_4	
7	T_2, T_3, T_4, T_5	
8	T_3, T_4, T_5	T_2
9	T_3, T_5	T_2, T_4

  - **因此，系統失敗後：**
    - **Redo串列中的T_2, T_4交易之寫入操作，將被重新執行(Redo)**
    - **Undo串列中的T_3, T_5交易之寫入操作，將被執行Undo工作(Undo)**
    - **在檢查點之前就已Committed的T_1交易，不需執行任何動作**

立即更新的缺點

- 每項操作皆必須寫入永久性磁碟，十分浪費時間。
  - ∵H.D.是機械式運作。
- 經常進行一些不必要的動作。
  - 例如：大部份的更新可能已被寫入資料庫中，卻仍要對所有已Committed的交易重複執行Redo動作。(for Undo/Redo演算法)
- 尚未確認的交易亦必須進行Undo工作，浪費時間。

※練習範例※

- 有一系統日誌記錄記載五個交易如下，請分別以延遲更新與立即更新技術從事復原，並討論所需動作為何? (由左到右、由上到下)

① <starts, T1>
<read(x), T1>
<write(x), T1, 11, 5>
② <starts, T2>
<read(y), T2>
<write(y), T2, 1, 10>
③ <commit, T2>
④ <starts, T3>
⑤ <checkpoint>
⑥ <starts, T4>
<read(w), T3>
<read(z), T1>
<write(w), T3, 7, 3>
<read(x), T4>
⑦ <starts, T5>
⑧ <commit, T1>
<write(x), T4, 5, 20>
<read(y), T5>
<write(y), T5, 10, 8>
⑨ <commit, T4>
<read(v), T3>
<write(v), T3, 1, 11>
<system crash>

- **Ans:**
  - **建立Undo/Redo串列：**
  - **系統失敗前**
    - **Undo串列：{T_3, T_5}**
    - **Redo串列：{T_1, T_4}**
  - **先做Undo，再做Redo**
    - **Redo：由前往後做**
    - **Undo：由後往前做**

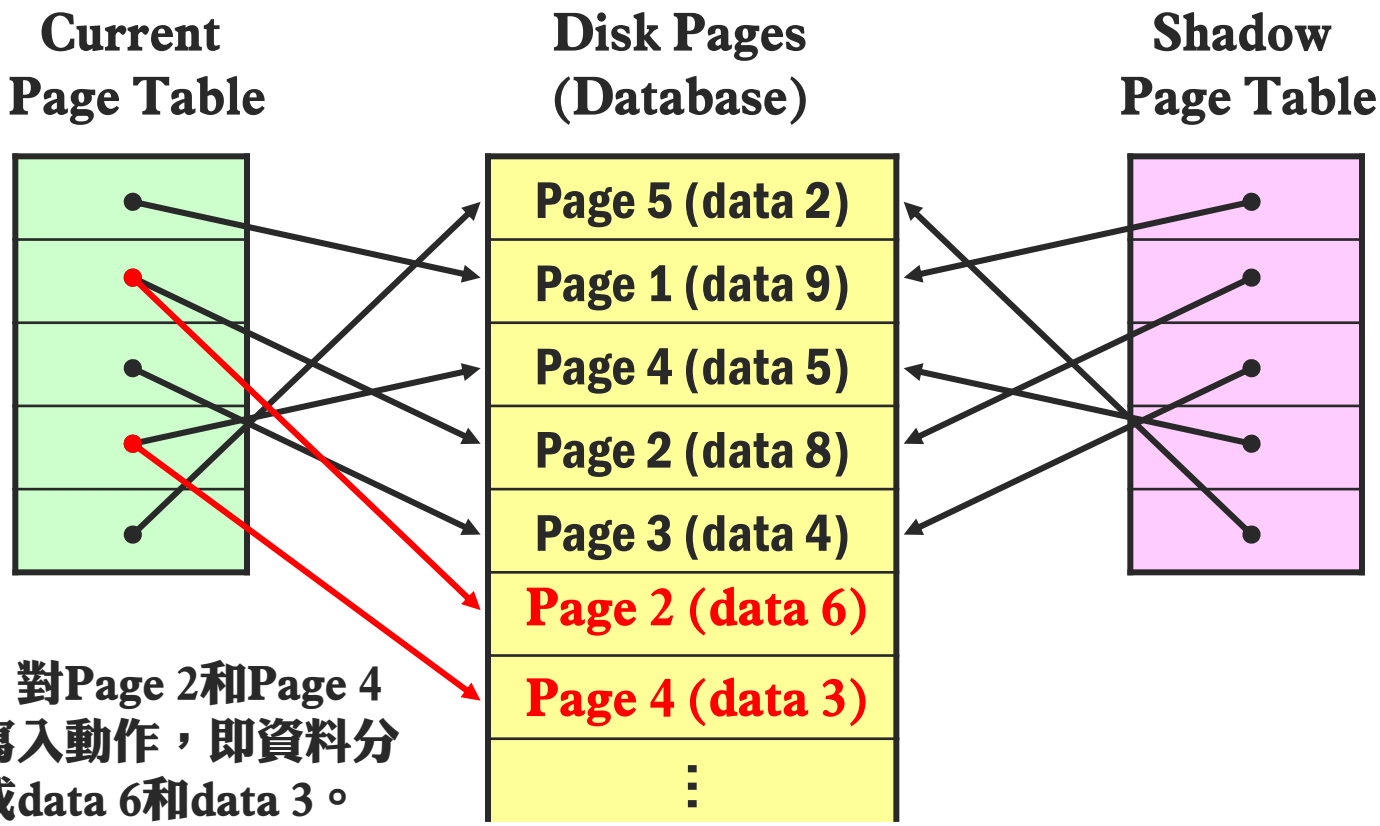
時間	Undo串列	Redo串列
1	T_1	
2	T_1, T_2	
3	T_1	T_2
4	T_1, T_3	T_2
5	T_1, T_3	(清空)
6	T_1, T_3, T_4	
7	T_1, T_3, T_4, T_5	
8	T_3, T_4, T_5	T_1
9	T_3, T_5	T_1, T_4

- 採用**延遲更新**技術：
  - **Undo串列**中的所有交易皆忽略。
  - **Redo串列**中的T1與T4交易之寫入操作重新執行並寫入至資料庫。即執行：<write(x), T1, 11, 5>與<write(x), T4, 5, 20>
- 採用**立即更新**技術 (**Undo/Redo**演算法)：
  - **Undo串列**中的T3與T5交易之寫入操作執行Undo動作。即執行：<write(v), T3, 11, 1>、<write(y), T5, 8, 10>與<write(w), T3, 3, 7>
  - **Redo串列**中的T1與T4交易之寫入操作重新執行並寫入至資料庫。即執行：<write(x), T1, 11, 5>與<write(x), T4, 5, 20>

投影 (Shadowing/投影分頁Shadow Paging)

- Shadowing是將資料庫視為由一些固定大小的磁碟分頁 (或磁碟區塊) 所組成。其中，有n個磁碟分頁存有資料。
- 需另外建構出具有n個項目的分頁表(Page Table，又稱目錄(Directory))，其中第i個項目是指向磁碟的第i個存有資料的磁碟分頁。
  - 當交易開始執行時，會先產生兩個不同類型的Page Table：
    - Shadow Page Table (投影分頁表)：用以指向交易前的原始磁碟分頁之資料，此分頁表不會被修改。
    - Current Page Table (目前分頁表)：用以指向交易執行期間的目前磁碟分頁之資料。
  - 當交易進行時，若欲改變資料庫中的某一磁碟分頁之資料時，會將改變的值另外存放至新的磁碟分頁，而不覆蓋舊磁碟分頁的資料。

- 交易開始時：
  - 一開始時Current Page Table和Shadow Page Table是一樣的。



交易的Commit與Recovery工作

- 交易Committed時：
  - 系統會拋棄 a. 被更新過的舊磁碟分頁，以及 b. 指向舊磁碟分頁所在的Shadow Page Table。
- 交易失敗，需要復原時：
  - 復原時，系統將拋棄 a. 新的磁碟分頁，以及 b. Current Page Table，採用Shadow Page Table。
- 由上可知，由於延遲更新在當系統發生非毀滅性故障時，不需進行Undo與Redo工作，因此這種技術又被稱為No-Undo/No-Redo演算法。

- **Shadowing/Shadow Paging特性：**
  - **在單一使用者的環境下，不需要使用到系統日誌 (System Log)即可從事復原，用2個不同類型的Page Table即可。**
  - **多使用者環境下，則需要用到系統日誌作並行控制。**
- **Shadowing/Shadow Paging優點：**
  - **Undo動作非常容易，留住Shadow Page Table即可。**
  - **不必進行Redo動作，留住Current Page Table即可。**
  - **在單一使用者的環境下，不需要使用到系統日誌即可從事復原。**

- **Shadowing/Shadow Paging缺點：**
  - **若分頁表(Page Table)太大，在Commit後會耗費較多的時間寫入分頁表。**
    - **∵Shadow Page Table位於H.D.，而Current Page Table位於M.M.。當Commit時，須將位於M.M.上的Current Page Table的資料寫到位於H.D.中的Shadow Page Table。**
  - **磁碟分頁邏輯上 (Logically) 連續，但時常更新磁碟分頁時，會造成磁碟分頁實際上 (Physically) 分散於磁碟各處，降低存取效率。**
    - **∴需定時作磁碟重組。**

補 充

兩階段確認協定(Two-Phase Commit Protocol)

- 定義：
  - 使用於**分散式資料庫系統環境**，讓分散於各處的資料庫可以同步執行交易的Commit與Rollback。
  - 交易隨時可以與其它各自獨立的的**資源管理者(Resource Manager)**溝通，而每個資源管理者皆各自管理自已可復原的資源，並擁有及維護自已的系統日誌(System Log)。
  - 此外，整個系統還有一個**總體協調者 (Coordinator)**，用以管理整個系統及各個管理者 (交易的參與者)。
- 執行步驟：
  - 分成**準備階段**與**行動階段**，共有5個步驟。

準備階段

① 協調者送出 “準備確認 (Prepare to commit)” 的訊息給所有參與者 (Participant/Cohost)。

② 交易程序的所有參與者收到準備確認的訊息後，將自已所擁有的局部資源(Local Resources)強制寫入位於永久性儲存媒體上的系統日誌。

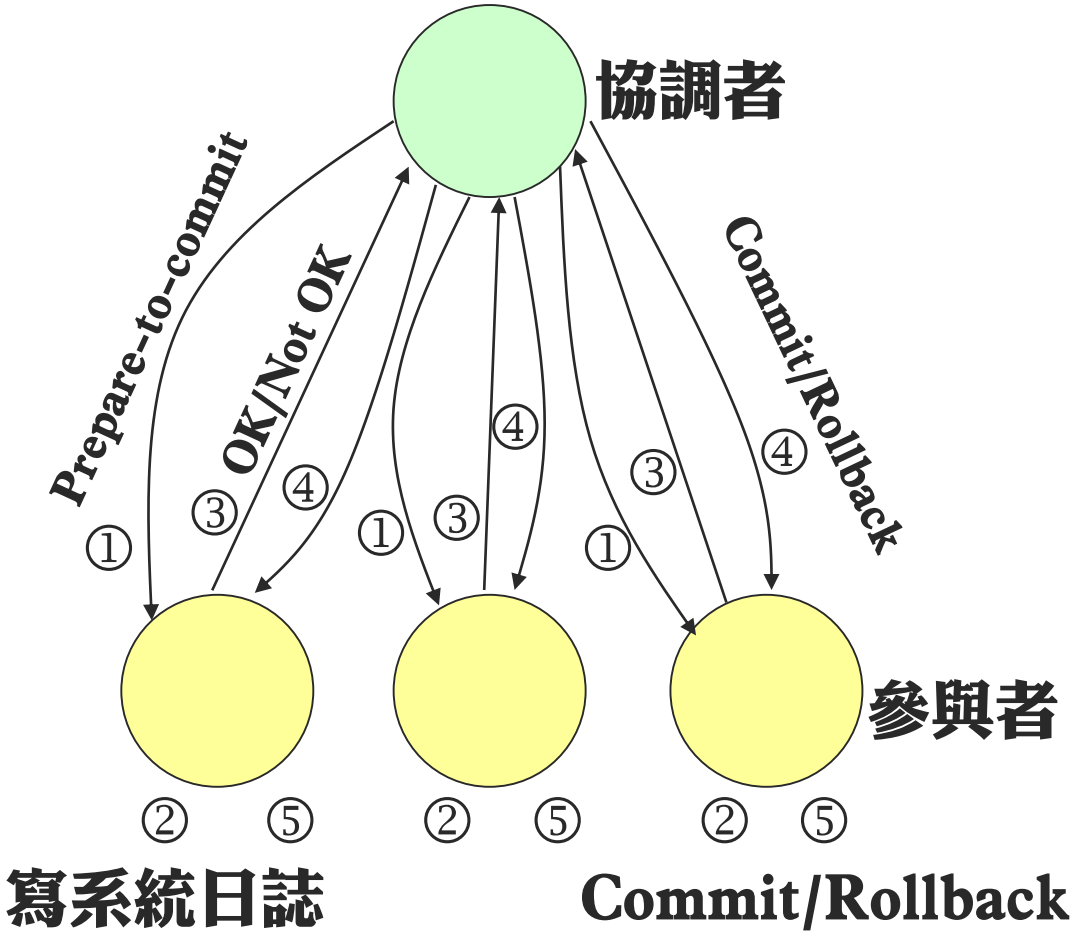
③ 參與者若成功寫入系統日誌，則發出一個 “OK” 的訊息給協調者；反之若寫入失敗，則發出一個 “Not OK”給協調者。

行動階段

④ 當協調者收到**所有參與者的 “OK” 訊息**，即送出一個 “**Commit**” 訊息給所有參與者；反之，若收到**任何一個參與者回覆的是 “Not OK”**，則送出 “**Rollback**” 給所有參與者。而自已也會從事相對應的操作(Commit/Rollback)。

⑤ 參與者收到 “Commit” 訊息後，則將Commit寫入系統日誌中，並真正更新資料庫的內容。若收到 “Rollback”，則針對自已的系統日誌從事回復 (Recovery)的動作。

圖示


協調者
Prepare-to-commit
OK/Not OK
Commit/Rollback
①
③
④
參與者
②
⑤
寫系統日誌
Commit/Rollback

- **失敗處理：**
  - 一般而言，系統於**準備階段**發生失敗，表示所有參與者尚未完全將紀錄寫入日誌，通常需要將交易**Rollback**。若於**行動階段**失敗，表示交易已成功執行完畢並寫入日誌，可被**復原 (Recovery)** 並**確認 (Commit)** 交易。
  - **系統協調者**會將每個階段所作的動作紀錄於協調者本身的系統日誌中。若系統(協調者)失敗後重新啟動，可搜尋**協調者的日誌**內容：
    - 找到失敗的地方以及狀態，並繼續進行。(如：收到所有參與者的OK/Not OK，但來不及發出訊息即掛掉時)
    - 若找不到協調者的日誌內容，則假設為Rollback。(如：未收到所有參與者的OK/Not OK即掛掉時)
- **一般採用延遲更新技術。**
  - ∵分散式環境風險高。