



資訊管理學系 陳士杰老師

資料庫系統管理

Database System Management

交易管理

Transaction



國立聯合大學
NATIONAL UNITED UNIVERSITY

[■ Outlines]

- 交易 (Transaction)
- 交易管理四大特性
- 交易狀態
- 系統日誌 (System Log, System Journal)
- 確認點 (Commit Point)
- 系統日誌強迫寫入 (System Log Force-writing)
- 檢查點 (Check Point)
- 交易的排程

【講義：Ch. 7, Section 1】

【原文：Ch. 13】

❖注意❖

在本單元中，DML指令包含DQL指令

交易管理

交易(Transaction)

- 將一或多個針對資料庫從事資料存取的動作 (即：DML指令，包括資料插入、刪除、修改或查詢...等)，包裝成**單一任務**來執行。
- 為資料庫處理的邏輯單位 (**不可分割**)。例：ATM提(轉)款，網路訂票
- 交易一定要整個完成，才能確保任務的正確性。

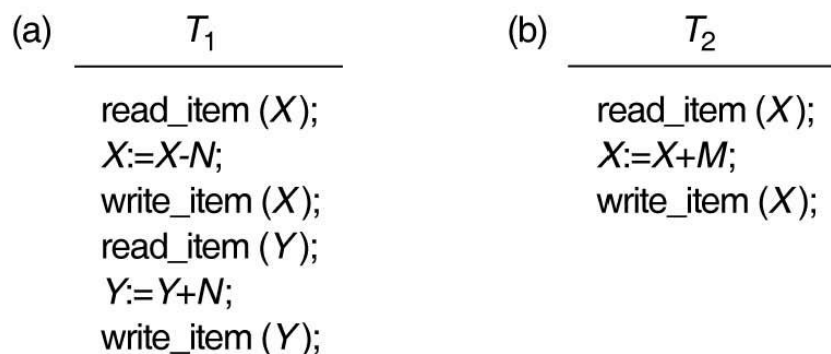


圖 13.2 兩個簡單的交易。(a) 交易 T_1 ；(b) 交易 T_2

- 在商業資料庫系統中，經常會遇到多筆交易同時對同一筆紀錄進行存取，如：訂位、股票交易、同一帳戶之金錢匯入(出)...等。若欲確保資料與交易的正確性，則交易管理是非常必要的工作。

- 交易管理的主要處理機制：

- 並行控制 (Concurrency Control)

- 讓多筆交易能在同一段時間內存取同一筆資料，而不會互相干擾。

- 失敗回復、復原 (Failure Recovery)

- 資料庫在執行某交易的過程中，若發生故障或執行失敗 (Failure) 的情況時，則必須要讓資料庫能夠重新回到一個已知的正確狀態。

■ 交易管理的目標：

- 確保交易可以**並行處理**
- 確保交易的**正確性 (Correctness)** 及**可靠性 (Reliability)**
- 提高資料庫系統異質性交易的**效率 (Efficiency)**
- 提高系統的**可用率 (Availability)**
- 降低系統**成本 (Cost)**

■ 交易管理的四大特性：ACID

■ ACID為交易管理必須注意的四大特性。

○ 單元性 (Atomicity；基元性)：

- 交易是一個**不可再分割的完整個體**，它不是全部執行，就是全部不執行。
 - **全部執行**：是指交易正確且正常完成，並透過確認(Commit)命令將交易結果存入永久性的資料庫中。
 - **全部不執行**：是指交易途中，若發生錯誤、毀損等因素，導致交易無法順利完成時，必須透過退回(Rollback)命令將交易回復到執行前的原點。
- 確保單元性是**回復 (Recovery)** 的責任。

○ 一致性 (Consistency) :

- 如果交易是全部執行，能讓資料庫從**某個一致狀態**，轉變到**另一個一致狀態**。我們則稱此次交易具有**一致性**。
 - 資料庫的**一致狀態 (Consistent State)**：是指資料庫所有被儲存的資料（不論是在交易前後），必須皆**滿足資料庫所設定的相關限制**，以及具有**正確的結果**。
- 確保一致性通常是**DBMS程式設計師**的責任。

○ 孤立性 (Isolation) :

- 某交易執行期間所用的資料或中間結果，**不容許其它交易讀取或寫入**，直到此交易被確認 (Commit，即：成功結束) 為止。也就是說，它不應被同時執行的其它交易所干擾。
- 某交易執行時，有可能會被**其它交易所查覺**。(如：處理同一筆資料)
- 確保孤立性是**並行控制 (Concurrency Control)** 的責任。可依需求定立**不同層級的限制**。

○ 永久性 (Durability, Permanency) :

- 一旦交易全部執行，且經過確認 (Commit) 後，其對資料庫所做的變更則**永遠有效**，即使未來系統當機或毀損。
- 一般是以**備份(Back Up)**、**硬碟映射(Disk Mirroring)**、**系統日誌(System Log、System Journal)**等數種方式來達成。
- 永久性是**回復 (Recovery)** 的責任。

■ 交易狀態

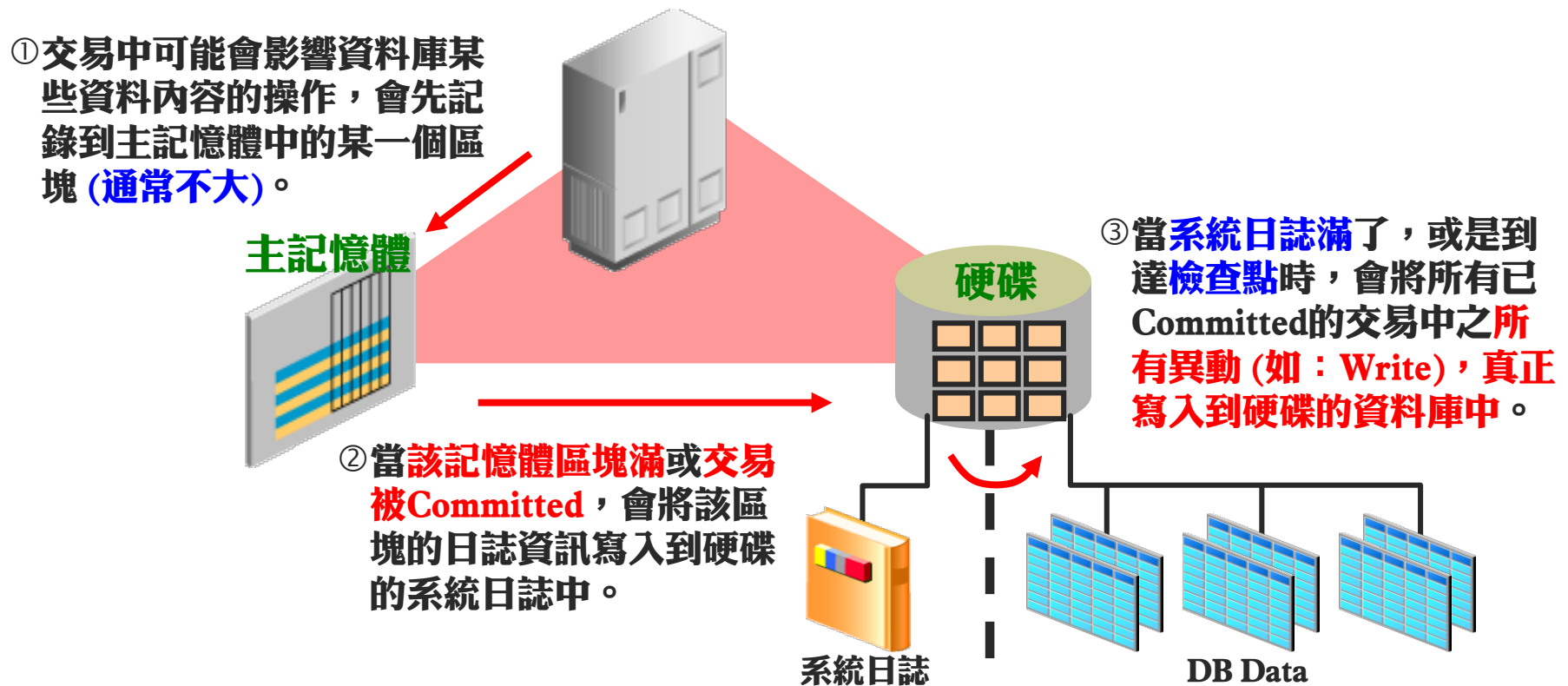
- 如先前所述，交易是將一或多個針對資料庫內的資料所做之存取動作（即：DML指令），包裝成單一任務的一個不可分割單元。通常會有下列動作：
 - **BEGIN_TRANSACTION**：交易執行**開始**。
 - **END_TRANSACTION**：交易執行**結束**，可能是Commit或Rollback。
 - **READ**或**WRITE**：指定某資料項的**讀寫動作**。
 - **Read_i(x)**表示在交易i中，對資料項x進行讀取動作； **Write_i(x)**表示對資料項x進行寫入動作。
 - 皆由 SQL 語法中的 DML指令所構成。
 - **COMMIT**：交易**全部完成**，且**更改的資料項已被確認進入到資料庫**。
 - **ROLLBACK** (或Abort)：交易**未完成**，此時需將交易對資料庫的所有改變做回復動作，即**退回到交易未執行前的原點**。

- 當交易進行中，若資料庫系統發生正常/不正常關閉時，DBMS可能會將交易Commit或是Rollback：
 - 以MySQL資料庫系統來說，不論是正常或不正常關閉，該交易皆會被Rollback。
 - 以Oracle資料庫系統來說，若是正常關閉，則該交易會被Commit；若是不正常關閉，則該交易會被Rollback。

■ 系統日誌 (System Log、System Journal)

- 為了能從各種故障回復，系統必須維護一個日誌 (Log)，以提供交易錯誤或故障時，所需的復原資訊。
- 系統日誌記錄了所有交易中可能會影響資料庫某些資料內容的操作(如：Write)。
- 系統日誌是儲存於永久性儲存媒體(如：硬碟)上，因此可預防非毀滅性故障。然而，系統日誌也必須定期備份到其它媒體，才能預防毀滅性故障。

- 雖然系統日誌是儲存在硬碟上，然而並非每新增一筆日誌記錄(即：可能會影響資料庫資料內容的操作)時，就立刻寫入硬碟中。
- 資料庫系統進行交易時，若有資料異動之運作方式：



- 在交易執行過程中，若遇到系統故障的情況，我們可以檢查**系統日誌**，並使用未來會講授的回復技術，有如下兩種動作：
 - **Redo**：若交易對資料的所有修改操作都已被記錄到系統日誌上，但尚未將所有的資料異動正式寫入資料庫中，此時可以回溯整個日誌，**重新執行交易的某些操作** (如：Write動作)，**以確認所有經commit的資料項目皆已真正更改了資料庫**。
 - **Undo**：**回復交易的某些操作** (如：Write動作)，當作沒發生過這些操作。
- Rollback與Undo的比較：
 - Rollback是回復整個交易，而Undo是回復交易中的單一或部份操作。

■ 系統日誌會記錄以下交易操作：（格式會因系統不同而不同）

- **[starts, Transaction_No]**：某一交易的開始
- **[write(x), Transaction_No, old_value, new_value]**：某一交易已將資料項目x所記錄的值，從原本的old_value改成new_value。
- **[read(x), Transaction_No]**：某一交易讀取了資料項目x所記錄的值。
- **[commit, Transaction_No]**：確認某交易已經成功完成，且其結果已交付於資料庫中。
- **[rollback, Transaction_No]**或**[abort, Transaction_No]**：代表某一交易已被中止、撤回。
- **[checkpoint]**：交易的檢查點，確認此點之前的資料已記錄至資料庫中。

■ 範例：

[starts, T1]

[read(x), T1]

[write(x), T1, 300, 500]

[read(y), T1]

[commit, T1]

[starts, T2]

[read(y), T2]

[write(y), T2, 200, 300]

[check point]

[write(x), T2, 500, 150]

[commit, T2]

■ 確認點 (Commit Point)

- 委任點、交付點
- 當某交易T裡所有對資料庫的存取動作都已成功執行，且交易動作的結果也已經寫入系統日誌時，此交易即到達了**確認點(Commit Point)**。
- 在達到確認點之後，此交易即稱之為**已確認的(Committed)**，且假設其結果會被永久記錄在資料庫中，接著交易便將確認記錄[commit, T]寫入**系統日誌**(在硬碟上)。
- 以**單元性(Atomicity)**的觀點來看，當系統發生故障時，若交易已經開始，但尚未到達確認點，則此交易必須被**退回**(Rollback、Abort)，以確保交易**全部不執行**。
- 反之，若交易到達確認點，代表此交易已成功完成，並將其所有可能影響資料庫之操作寫入永久性儲存媒體(即：系統日誌)。即使未來系統發生故障，此交易仍然有效。

■ 確認點之運作方式：

(in Block)

[starts, T1]
[read(x), T1]
[write(x), T1, 300, 500]
[read(y), T1]
[commit, T1]

主記憶體

(in H.D.)

[starts, T1]
[write(x), T1, 300, 500]
[commit, T1]

硬碟

③當系統日誌滿了，或是到達檢查點時，會將所有已Committed的交易中之所有異動(如：Write)，真正寫入到硬碟的資料庫中。

系統日誌

x=300

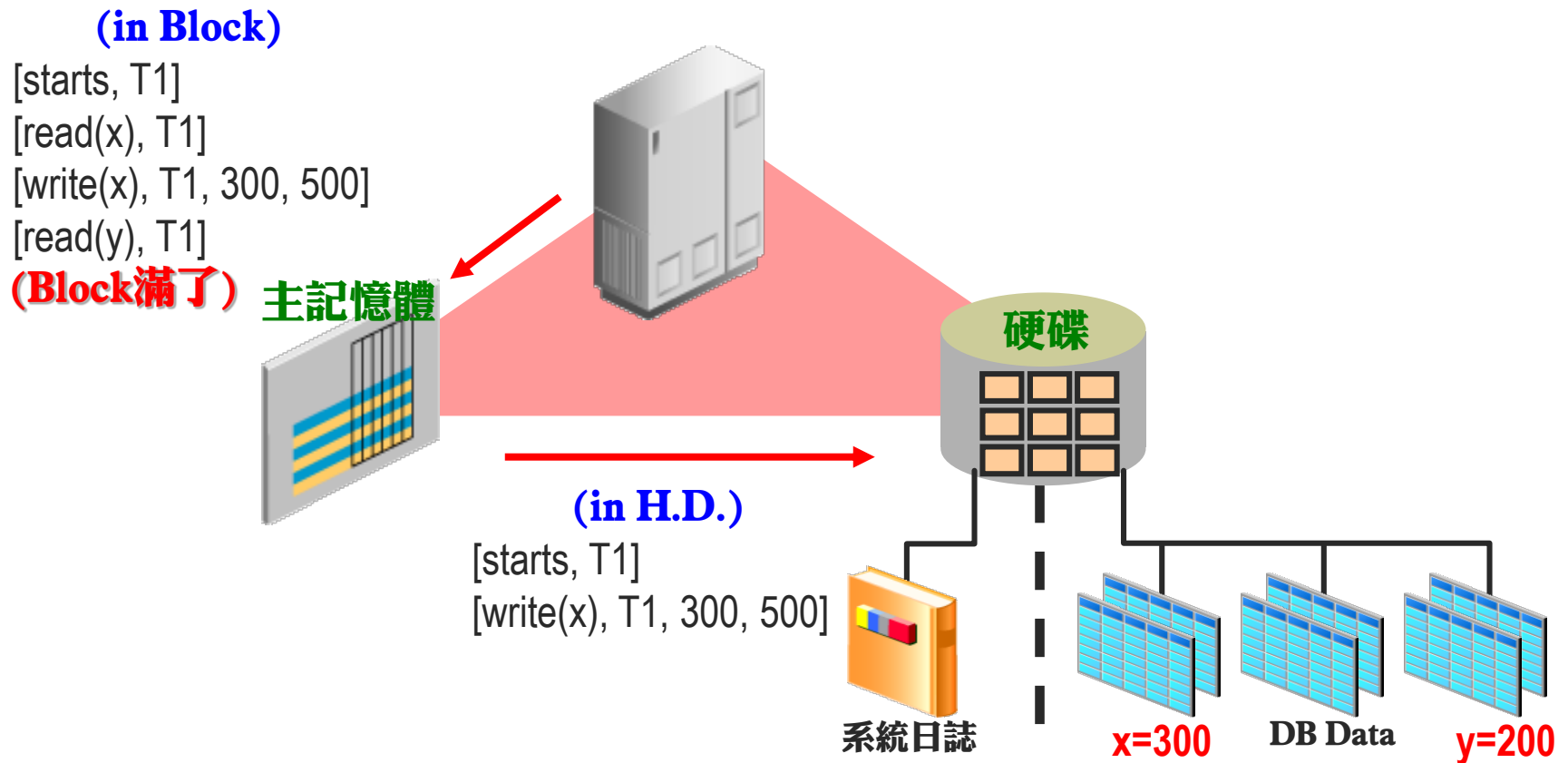
DB Data

y=200

■ 系統日誌強迫寫入 (System Log Force-writing)

- 正常而言，在主記憶體中會保留一個區塊(Block)，做為交易進行時記載資料異動操作的日誌記錄，並於**交易Committed (交易到達確認點)**時，將其寫入到硬碟內的系統日誌。
- 然而，當交易**尚未到達確認點**前，因某些原因而將位於主記憶體中的日誌寫入硬碟，即稱為**系統日誌強迫寫入**。
 - **區塊滿了**
 - 到達**檢查點(Check Point)**
- 當系統故障時，只需考慮那些被寫入硬碟的系統日誌記錄。

■ 系統日誌強迫寫入之運作方式：



Check Point (檢查點)

- 檢查點(Check Point)是由DBMS定期(週期性)發動的系統日誌強迫寫入，並將一個檢查點(Check Point)強迫寫入到系統日誌中。同時，會把系統日誌當中、於檢查點之前所有已確認 (Committed)的交易所產生之異動(如：寫入 (Write)動作)結果，真正寫入資料庫中。
- 檢查點的寫入，代表此點之前所有已確認(Committed)的交易，在系統發生錯誤或毀損時，不需要被重新執行(Redo)。
- 因此，當系統發生錯誤時，檢查點可確定哪些交易在發生錯誤前已確認(Committed)了。

■ 檢查點之運作方式：

(in Block)

[starts, T1]

[read(x), T1]

[write(x), T1, 300, 500]

[read(y), T1]

[commit, T1]

[starts, T2]

[read(y), T2]

[write(y), T2, 200, 600]

(Check Point發生)

Restart

主記憶體

(in H.D.)

[starts, T1]

[write(x), T1, 300, 500]

[commit, T1]

[starts, T2]

[write(y), T2, 200, 600]

[check point]

系統日誌

硬碟

x=300

↓
x=500

DB Data

y=200

■ 檢查點的運作：

- 暫停所有交易動作
- 將所有在主記憶體區塊中，已確認的交易操作強制寫入系統日誌(硬碟)
- 將主記憶體區塊上、尚未Commit之交易的日誌紀錄強制寫入到系統日誌(硬碟)中，並寫入一個check point 到系統日誌
- 繼續交易

■ 交易的排程

◆ 排程(Schedule)

- 多筆交易以**交錯方式**並行執行時，所構成的執行順序
- 若 n 個交易 T_1, T_2, \dots, T_n 構成一個排程 S ，則每一個交易之操作在排程 S 中的出現順序，必須與該操作於本身交易內之順序相同。

■ 範例：假設有兩筆交易 T_1 與 T_2 ， T_1 的交易操作為： $r_1(x), r_1(y), w_1(x), c_1$ ， T_2 的交易操作為： $r_2(y), w_2(y), r_2(x), w_2(x), c_2$ ，可能形成如下排程：

- Sa: $r_1(x), r_1(y), w_1(x), c_1, r_2(y), w_2(y), r_2(x), w_2(x), c_2$
- Sb: $r_1(x), r_1(y), w_1(x), r_2(y), w_2(y), c_1, r_2(x), w_2(x), c_2$
- ...

■ 序列排程、循序排程 (Serial Schedule)

- 一個具有n筆交易的排程為**序列排程**，若且唯若此n筆交易的操作皆連續不間斷地被執行，而**沒有任何相互交錯**的現象。
 - Sa: $r1(x), r1(y), w1(x), c1, r2(y), w2(y), r2(x), w2(x), c2$
- 優點：若各交易本身皆為正確，則序列排程可保證資料庫的正確性，無論交易間執行之順序為何，皆不會影響最終結果。
- 缺點：浪費時間和系統資源，且缺乏彈性。
 - 由於排程中的某一操作在執行時，無論其是否使用到CPU或其它資源，皆不可切換至其餘交易的操作去執行。

■ 可序列化排程、可循序性排程 (Serializable Schedule)

- 一個具有n筆交易的排程為**可序列化排程**，若此排程與**相同的n筆交易所構成之某一序列排程等價(Equivalent)**。
 - Sb: $r1(x), r1(y), w1(x), r2(y), w2(y), c1, r2(x), w2(x), c2$
- 可提供交易的**並行性(Concurrency)**，紓解序列排程的缺點，且保證交易的正確性。

■ 等價的種類：

○ 結果等價(Result Equivalent)

- 若兩個排程最後產生相同的資料庫狀態，則稱此兩排程為**結果等價**
- 結果等價可能在偶然中發生，因此排程的等價性不採用結果等價

○ 範例：

S_1
read_item(X); $X := X + 10$; write_item(X);

S_2
read_item(X); $X := X * 1.1$; write_item (X);

Figure 17.6

Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

○ 衝突等價(Conflict Equivalent)

■ 衝突：

T1 \ T2	讀(read)	寫(write)
	讀(read)	寫(write)
讀(read)	否	是
寫(write)	是	是

- 若兩個排程中，多個交易間**發生衝突的順序相同**，稱此兩排程為衝突等價
- 有衝突不見得會有錯!!
 - 有衝突，但不會影響多筆交易結果的正確性即OK!!
 - 有衝突，且會影響多筆交易結果的正確性即不OK!!

■ 例：下列3個排程各有兩個交易在進行：

- **序列排程 Sa**: $r1(x), r1(y), w1(x), c1, r2(y), w2(y), r2(x), w2(x), c2$ 的衝突有：
 - $r1(x) \rightarrow w2(x)$, $r1(y) \rightarrow w2(y)$, $w1(x) \rightarrow r2(x)$, $w1(x) \rightarrow w2(x)$
- **排程 Sb**: $r1(x), r1(y), w1(x), r2(y), w2(y), c1, r2(x), w2(x), c2$ 的衝突有：
 - $r1(x) \rightarrow w2(x)$, $r1(y) \rightarrow w2(y)$, $w1(x) \rightarrow r2(x)$, $w1(x) \rightarrow w2(x)$
- **排程 Sc**: $r1(x), r1(y), r2(x), w1(x), r2(y), w2(y), c1, w2(x), c2$ 的衝突有：
 - $r1(x) \rightarrow w2(x)$, $r1(y) \rightarrow w2(y)$, $r2(x) \rightarrow w1(x)$, $w1(x) \rightarrow w2(x)$
- 上述Sa與Sb兩個排程為衝突等價; Sa/Sb與Sc不符合衝突等價。

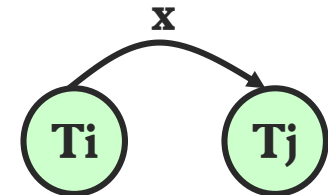
■ 衝突可序列化排程 (Conflict Serializable Schedule)

- 一個具有 n 筆交易的排程是**可序列化的**，假設此排程與相同的 n 筆交易之某個**序列排程**是衝突等價。(如前例的排程Sb)

■ 檢驗步驟：

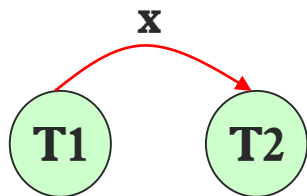
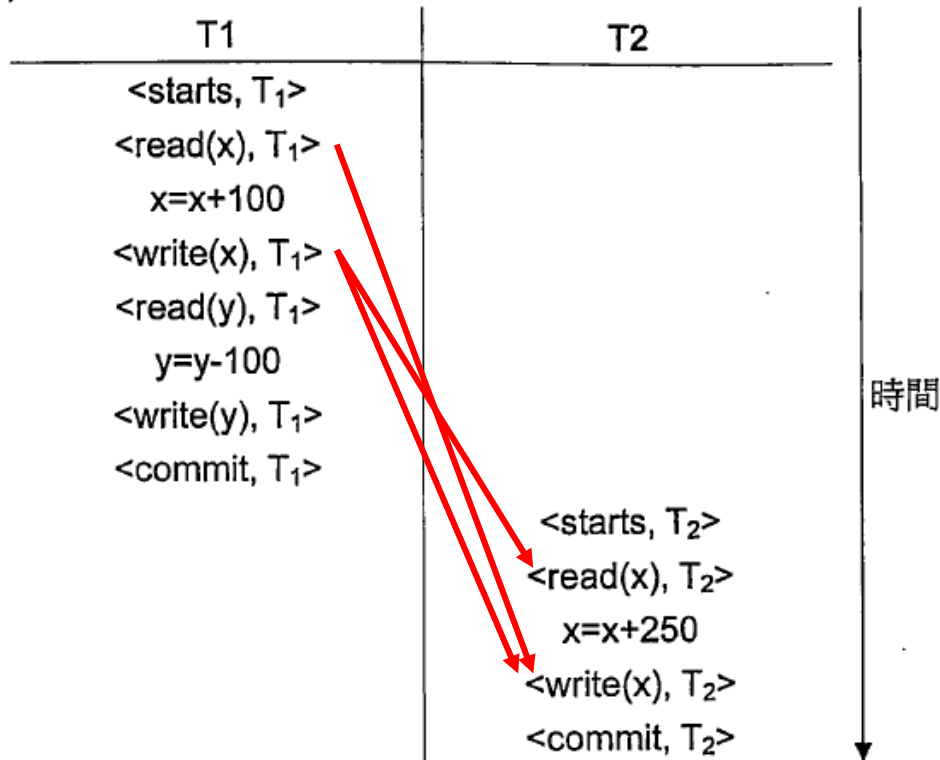
- 繪製**優先順序圖(Precedence Graph)**

- 對參與排程的每一筆交易 T_i ，建立一個**節點(Node)**
- 判斷排程中，不同交易操作所發生的每一個**衝突**之順序。如：先執行 $[read(x), T_i]$ 才執行 $[write(x), T_j]$ ，則建立 $T_i \rightarrow T_j$ 的射線。例如：



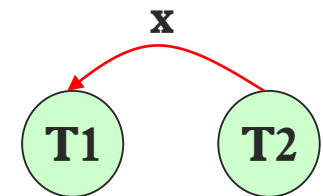
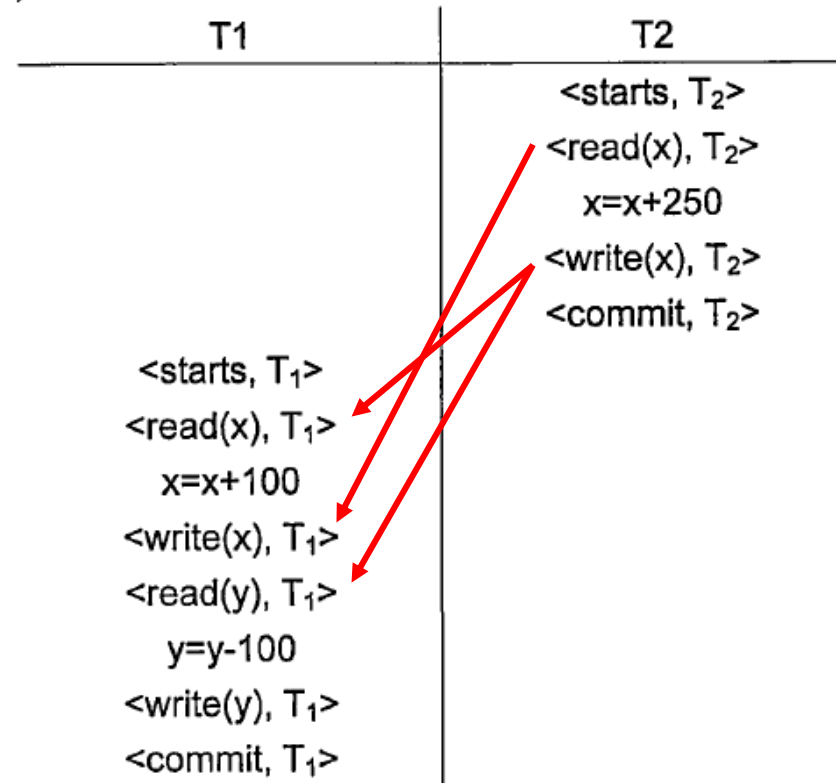
- 判斷優先順序圖是否含有**迴圈(Cycle)**
 - 無，則此排程為**可序列化(Serializable)**
 - 有，則此排程為**非可序列化**

(1)



序列(循序)排程
(T₁, T₂)

(2)



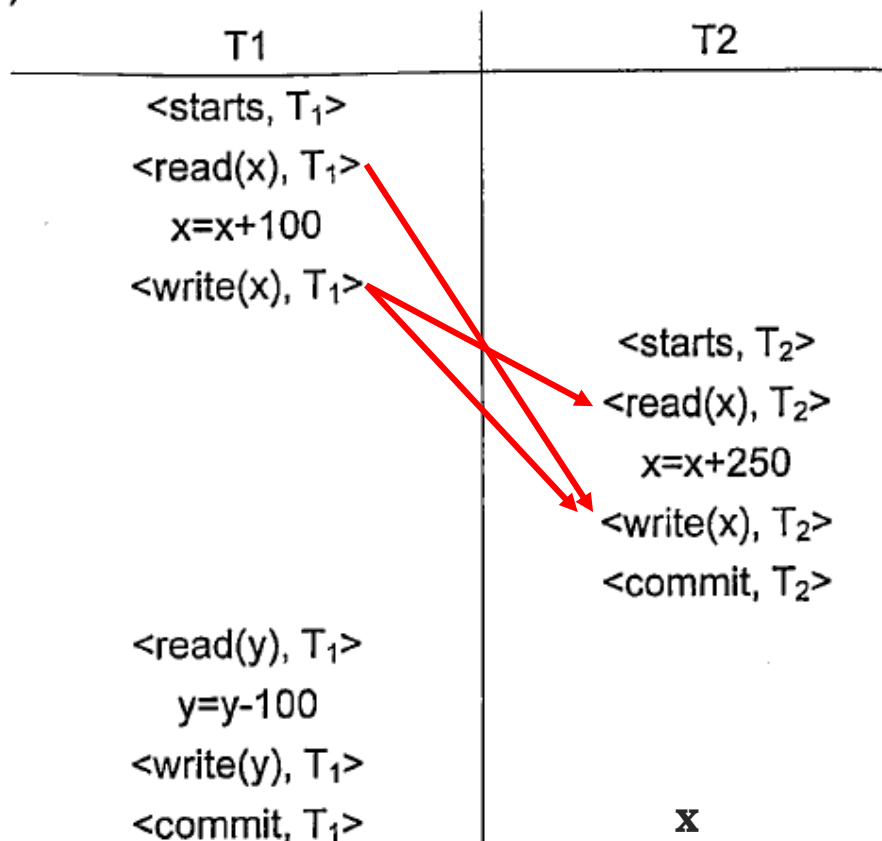
序列(循序)排程
(T₂, T₁)

[

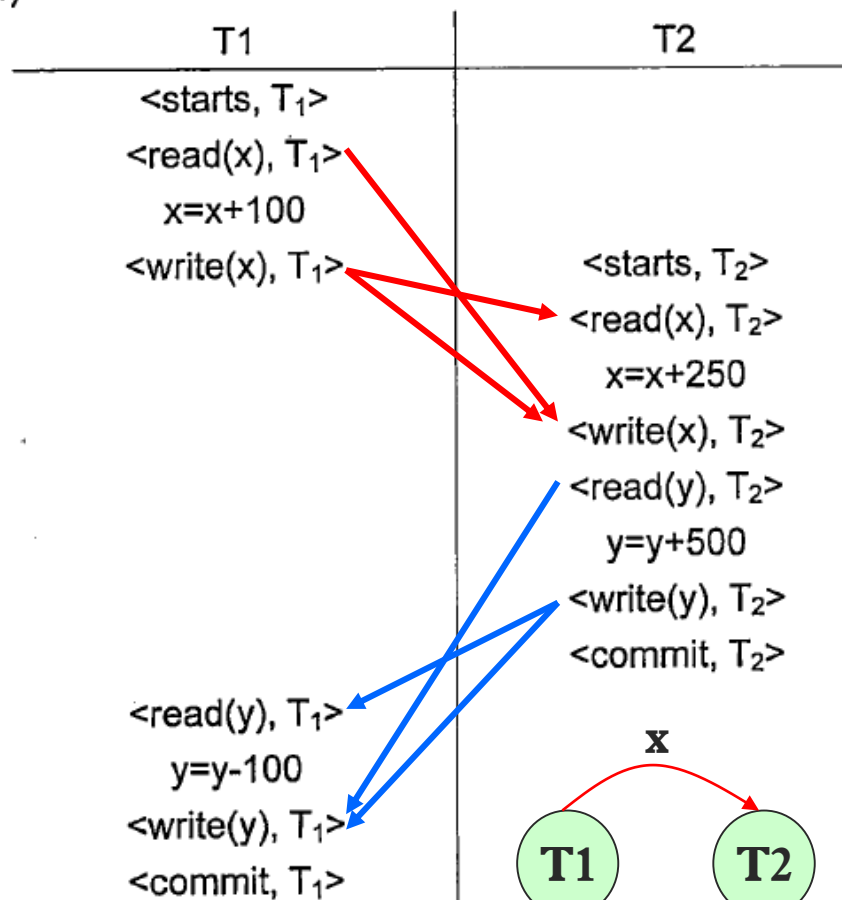
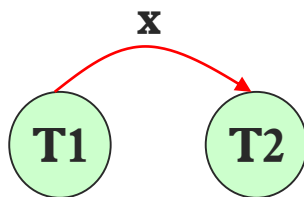
]

(3)

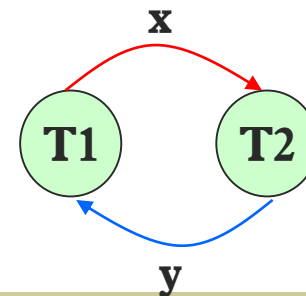
(4)



可序列(循序)化排程
(T₁, T₂)



非可序列化排程



【例題】下面排程是否為可序列化(Serializable)？若是，請列出一個與其等價(equivalent)之序列排程(serial schedule)；若否，為什麼？

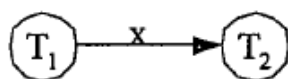
(1) $r_1(x), w_1(z), r_2(y), w_1(x), w_2(m), r_1(y), r_2(m), c_1, w_2(x), c_2$

(2) $r_1(x), w_1(x), r_3(y), w_2(y), w_3(z), w_1(y), w_2(z), r_1(y), r_1(z), w_2(m), c_1, c_2, c_3$

(3) $r_2(x), r_1(y), w_1(x), c_1, r_3(x), w_3(y), w_3(z), w_2(z), c_2, c_3$

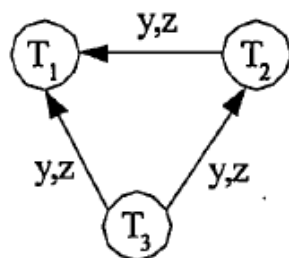
Ans:

(1)



無迴圈，故可序列化。 序列排程： T_1, T_2

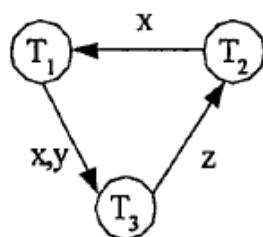
(2)



無迴圈，故可序列化。

序列排程： T_3, T_2, T_1

(3)

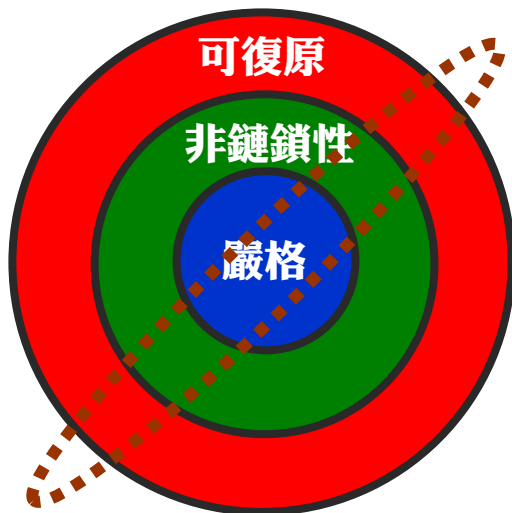


優先順序圖存在迴圈，故不可序列化。

迴圈： $T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_1$

其它類型的排程

- ① 可復原排程 (Recoverable Schedule)
 - ② 非連鎖性撤回排程 (Cascade-less Schedule)
 - ③ 嚴格排程 (Strict Schedule)
- 上述三種排程之特性一個比一個嚴格，因此排程②包含了排程①，排程③包含了排程①與排程②，反之則否。



可序列化排程 (Serializable Schedule)

- 可能是上述三種排程當中之，但也有可能都不是。

■ 可復原排程(Recoverable schedule)

- 如果排程S中的兩個交易T與T'，交易T有read到交易T'所write過的資料項目時，**交易T'必須比交易T先結束** (即：Commit或Rollback)，則排程S可被認為是Recoverable Schedule。

- 範例：假設有兩個交易T1和T2存在於下列排程S中

$S: w1(x) \rightarrow r2(x) \rightarrow c1 \rightarrow c2$

上述排程S為**可復原排程**。但若改為交易T2先被Commit的話，則當交易T1被Rollback或Abort時，交易T2的結果會不正確，且無法復原 (因Commit的交易不得再被取消)。

- **連鎖性撤回/連鎖性中止 (Cascading Rollback/Abort)**

- 若有未Commit的交易T，讀取了已失敗的其它交易T'曾經寫入之資料項目，因此必須被**連帶撤回**。
- 連鎖性撤回/中止非常耗時，因此應該盡量避免。

■ 非連鎖性撤回排程(Cascadeless Schedule)

- 如果排程S中的每個交易T都只讀取已Commit或Rollback之交易T'所寫入的資料項目，則排程S可被認為是Cascadeless Schedule，可避免連鎖性撤回。

- 範例：假設有兩個交易T1和T2存在於下列排程S中

$S: w1(x) \rightarrow c1 \rightarrow r2(x) \rightarrow c2$

上述排程S為非連鎖性撤回排程。雖然會延遲交易T2的執行，但可以確保如果交易T1中止時，不會發生連鎖性撤回的現象，使得交易的確定性較高。

此範例同時也合乎可復原排程的要求（∵交易T2讀取到交易T1寫入的資料項目，且交易T1比交易T2先結束）。

■ 嚴格排程(Strict Schedule)

- 如果排程S中的每個交易T都只讀取或寫入已Commit或Rollback之交易T'所寫入的資料項目，則排程S可被認為是Strict Schedule。
- 符合 Cascadeless Schedule的要求 (即： $w1(x) \rightarrow c1 \rightarrow r2(x)$)，且 $w1(x) \rightarrow c1 \rightarrow w2(x)$ 。

【例題】考慮下面 Sa、Sb、Sc 三個排程(schedules)：

Sa : r2(x)、r1(x)、w2(x)、w1(x)、c2、w1(y)、c1

Sb : r1(x)、w1(y)、c1、r2(y)、w2(x)、c2

Sc : r1(x)、w2(y)、w3(z)、r1(y)、r2(z)、c1、c2、c3

r 代表"read"，w 代表"write"，c 代表"commit"

- (1) 這些排程是否為可復原(recoverable)排程？
- (2) 這些排程是否為非連鎖性撤回(cascadeless)排程？
- (3) 這些排程是否為嚴格(strict)排程？

Ans:

1) 排程 S_a : $r_2(x)$ 、 $r_1(x)$ 、 $w_2(x)$ 、 $w_1(x)$ 、 c_2 、 $w_1(y)$ 、 c_1

- a. 無 read 到其它交易 write 過的資料項目，故為 Recoverable、Cascadeless Schedule。
- b. 交易 T_1 write 到交易 T_2 寫過的資料項目 x ，且交易 T_2 未 Commit，故不為 Strict Schedule。

2) 排程 S_b : $r_1(x)$ 、 $w_1(y)$ 、 c_1 、 $r_2(y)$ 、 $w_2(x)$ 、 c_2

- a. 交易 T_2 read 到交易 T_1 寫過的資料項目 y ，且交易 T_1 比交易 T_2 早 commit，故為 Recoverable Schedule。
- b. 交易 T_2 read 到交易 T_1 寫過的資料項目 y ，且交易 T_2 read 時，交易 T_1 已 commit，故為 Cascadeless Schedule
- c. 因為此排程為 Cascadeless Schedule，且無任何 write 到其它交易寫過的資料，故為 Strict Schedule。

3) 排程 S_c : $r_1(x)$ 、 $w_2(y)$ 、 $w_3(z)$ 、 $r_1(y)$ 、 $r_2(z)$ 、 c_1 、 c_2 、 c_3

- a. 交易T1 read 到交易T2寫過的資料項目 y ，但 T1 commit時，T2未 Commot；且交易T2 read 到交易T3寫過的資料項目 z ，但 T2 commit時，T3未Commot。故不為Recoverable Schedule，也非 Cascadeless Schedule，也一定不是Strict Schedule。

【解】

- (1) S_a 、 S_b ：可復原(recoverable)排程
 S_c ：不可復原(not recoverable)排程
- (2) S_a 、 S_b ：非連鎖性撤回(cascadeless)排程
 S_c ：連鎖性撤回(not cascadeless)排程
- (3) S_b ：嚴格(strict)排程
 S_a 、 S_c ：非嚴格(not strict)排程

[

]

補充

■ DDL與DCL指令對交易的影響

- 在資料庫系統實作中，DDL與DCL的每一個指令，本身皆是一個**完整的交易**!!
 - 即：執行一個DDL或DCL指令時，相當於隱藏了一組 Begin... Commit於該指令的前方與後方。
- 因此，若在執行一組交易的過程中，穿插了一個DDL或DCL的指令，則該執行中的交易會隨著DDL或DCL指令的結束而自動Commit。