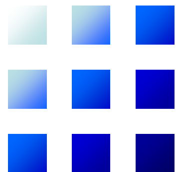
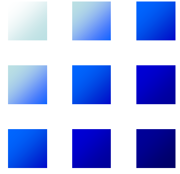


データベース
【12:同時実行制御】

石川　佳治



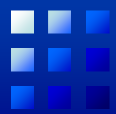
トランザクション

トランザクション（transaction）

- アプリケーションにおけるひとまとまりの処理を構成するデータベース操作の集まり
- 例：預金口座 A から預金口座 B へ10000円を送金するトランザクション

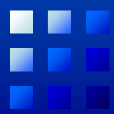
```
read(A, x)
read(B, y)
x := x – 10000
y := y + 10000
write(A, x)
write(B, y)
```

データベースに対する操作は，究極的には read, write の組合せで表現できるため，操作を read, writeに単純化



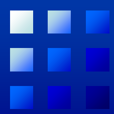
トランザクション処理

- トランザクション処理（transaction processing）
  - もしくはトランザクション管理（transaction management）
  - 並行して実行されるトランザクションの競合を解決
  - 各種障害への発生への対応（次章）
- ACID特性（ACID properties）
  - トランザクション処理において保障することが望ましい性質



ACID特性（1）

- 原子性（atomicity）
  - トランザクションがデータベース処理の単位
  - トランザクションの結果は，以下の二者択一
    - コミット（commit）：データ操作のすべてが確定されたものとしてデータベースに反映される
    - アボート（abort）：データ操作がすべて取り消される
  - 一部の操作のみの中途半端な実行は許されない
- 整合性（consistency）
  - 整合性がとれたデータベースに対して実行されたトランザクションの実行の結果は，整合性のとれたものとなる



ACID特性(2)

- 隔離性(isolation)
  - 複数のトランザクションを並行処理した場合でも,トランザクションは同時に処理されている他のトランザクションの影響を受けない
  - 複数のトランザクションの並行処理の結果は,トランザクションを何らかの順序で逐次処理した場合と一致しなければならない
- 耐久性(durability)
  - いったんコミットしたトランザクション中でのデータ操作は,その後の障害などで消滅してはならない

アプリケーションにおける命令

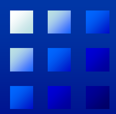
- 通常，アプリケーションプログラムには，read, write以外に以下の命令を提供
  - begin：トランザクションの開始を宣言
  - commit：トランザクションのコミットを要求
  - abort：トランザクションのアボートを要求
    - 処理の中断に利用

```
begin
read(A, x)
x := x – 10000
write(A, x)
commit
```

コミットの例

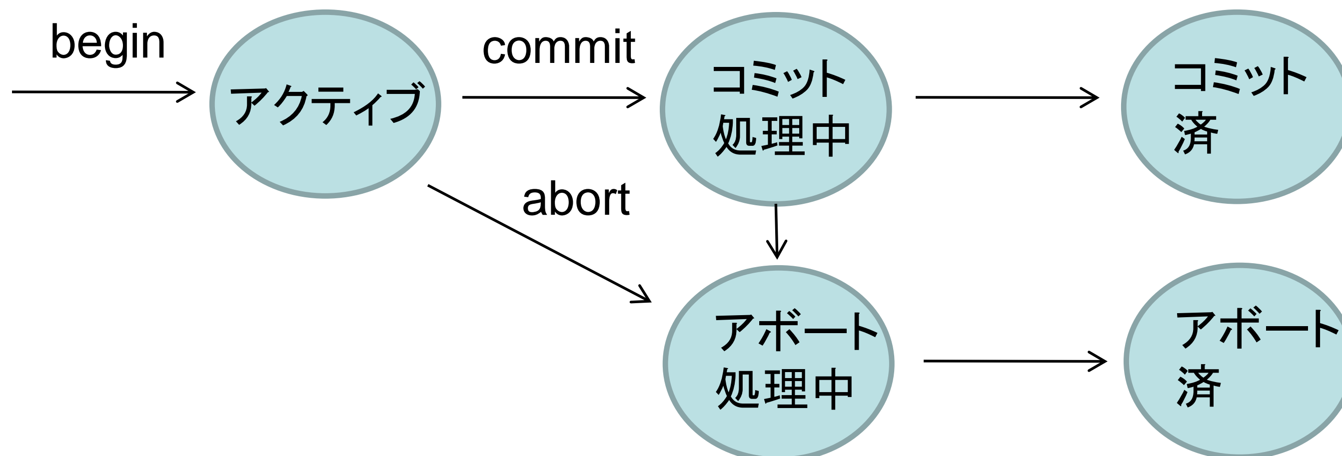
```
begin
read(A, x)
x := x – 10000
…
（何らかの問題を検知）
abort
```

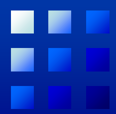
アボートの例



トランザクションの状態(1)

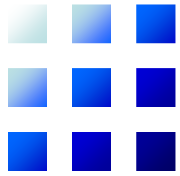
- 5つの状態
  - アクティブ:トランザクションを実行中
  - コミット処理中:commit命令後の状態
  - コミット済:コミットのための処理が終了
  - アボート処理中:abort命令後や，コミット処理が何らかの理由で正常に終了できないとき
  - アボート済み:アボート処理が終了





トランザクションの状態（2）

- ロールバック（rollback）
  - アボートされるトランザクションが，アクティブな状態の間に何らかのデータ更新を行っていた場合に，それらをすべて取り消す処理
  - トランザクション開始前の状態に「巻き戻す」



並行処理と直列可能性

並行処理における不整合（1）

- DBMSは複数のトランザクションを並行実行
  - 待ち時間（入出力，入力）を有効活用
- 一定の規約を設けないと不整合が発生

トランザクション T_1

```
read(A, x)
x := x – 10000



write(A, x)
```

トランザクション T_2

```
read(A, y)
y := y – 10000
write(A, y)
```

データ更新の喪失の例
トランザクションT_2が
書き込んだ値を
トランザクションT_1が
上書きしてしまうため，
トランザクションT_2の
更新は反映されない

並行処理における不整合(2)

トランザクション T_1

```
s := 0
read(A, x)
s := s + x
read(B, y)
s := s + y
```

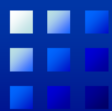
トランザクション T_2

```
read(A, z)
z := z – 10000
write(A, z)
read(B, w)
w := w + 10000
write(B, w)
```

整合性のないデータ読出しの例

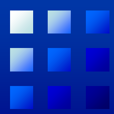
トランザクションT_1は誤った合計値を出力する

同時実行制御(concurrency control; 並行処理制御)により，ACID特性に反する不整合が生じないようにする



直列可能性

- 直列可能性（直列化可能性; serializability）
  - トランザクション $T_1, \ldots, T_n$ を並行処理したときの実行結果が，それらを何らかの順序で逐次処理したときの実行結果と一致すること
- 同時実行制御の役割
  - トランザクション $T_1, \ldots, T_n$ が並行処理された場合でも，各トランザクション T_i の直列可能性を保証する



スケジュール（1）

- データベースに対する基本操作の表記
  - R_i(A)：トランザクションT_iによる項目Aのread
  - W_i(A)：トランザクションT_iによる項目Aのwrite
  - C_i：トランザクションT_iのコミット
  - A_i：トランザクションT_iのアボート
- トランザクション$T_1, \ldots, T_n$に対する**スケジュール**（schedule）
  - $T_1, \ldots, T_n$の基本操作を，インターリーブして一列に並べたもの
  - T_i内の基本操作の順序関係は保存

スケジュール(2)

T_1	T_2
	read(A, z)
	z := z – 10000
	write(A, z)
s := 0	
read(A, x)	
s := s + x	
read(B, y)	
s := s + y	
	read(B, w)
	w := w + 10000
	write(B, w)

各トランザクションの表現

T_1: $R_1(A)\ R_1(B)$

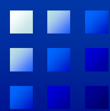
T_2: $R_2(A)\ W_2(A)\ R_2(B)\ W_2(B)$

上の実行順序に対応するスケジュール

$R_2(A)\ W_2(A)\ R_1(A)\ R_1(B)\ C_1\ R_2(B)\ W_2(B)\ C_2$

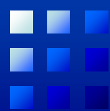
直列スケジュール

- 直列スケジュール(serial schedule)
  - 対象トランザクション群を何らかの順序で逐次処理する場合のスケジュール
- 非直列スケジュール(nonserial schedule)
  - 直列スケジュールでないスケジュール
- 直列可能スケジュール(serializable schedule)
  - 直列スケジュールと「等価」なもの
  - 並行処理における直列可能性の保証とは，スケジュールが直列可能となることを保証すること



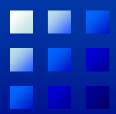
スケジュールの等価性(1)

- 異なる定義が存在
- 競合等価(conflict equivalent):同じトランザクション集合に対する二つのスケジュールS_1, S_2は，以下を満たすとき競合等価
  - ① S_1において $R_i(A)$($W_i(A)$)が $W_j(A)$($R_j(A)$)に先行するならば，S_2 においても同様の関係が成り立つ
  - ② S_1において$W_i(A)$が$W_j(A)$に先行するならば，S_2においても同様の関係が成り立つ
  - ※ write処理に関して不整合が発生するので，writeに関わる実行順序に着目



スケジュールの等価性(2)

- **ビュー等価**(view equivalent)
  - ① S_1において R_i(A)より読まれるA値が，W_j(A)によって書かれた値またはAの初期値ならば，S_2においても同様の関係が成り立つ
  - ② 各項目Aに関して，S_1において最後にA値を書くのがW_i(A) ならば，S_2においても同様のことが成り立つ
  - ※ 直観的には，各readが同じ値を読み，かつ最後のデータベースの状態が同じであること

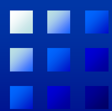


スケジュールの等価性(3)

- 競合等価の方がより厳しい条件
  - 競合等価なスケジュールはビュー等価
  - その逆は必ずしも成立しない
- 例
  - S_1 : $R_1(A)$ $W_2(A)$ C_2 $W_1(A)$ C_1 $W_3(A)$ C_3
  - S_2 : $R_1(A)$ $W_1(A)$ C_1 $W_2(A)$ C_2 $W_3(A)$ C_3
  - S_1, S_2はビュー等価
    - $R_1(A)$が読む値は両者において初期値
    - 両者において最後にAを書くのは$W_3(A)$
  - しかし，競合等価ではない
    - S_1では$W_2(A)$が$W_1(A)$に先行．S_2ではその逆．

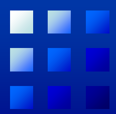
競合直列可能／ビュー直列可能

- 競合直列可能（conflict serializable）
  - そのスケジュールがある直列スケジュールと競合等価である場合
- ビュー直列可能（view serializable）
  - そのスケジュールがある直列スケジュールとビュー等価である場合
- 競合直列可能スケジュールはビュー直列可能：その逆は成立しない
  - 先の例のS_1は直列スケジュールであるS_2とビュー等価なのでビュー直列可能
  - しかし，S_1は競合直列可能ではない



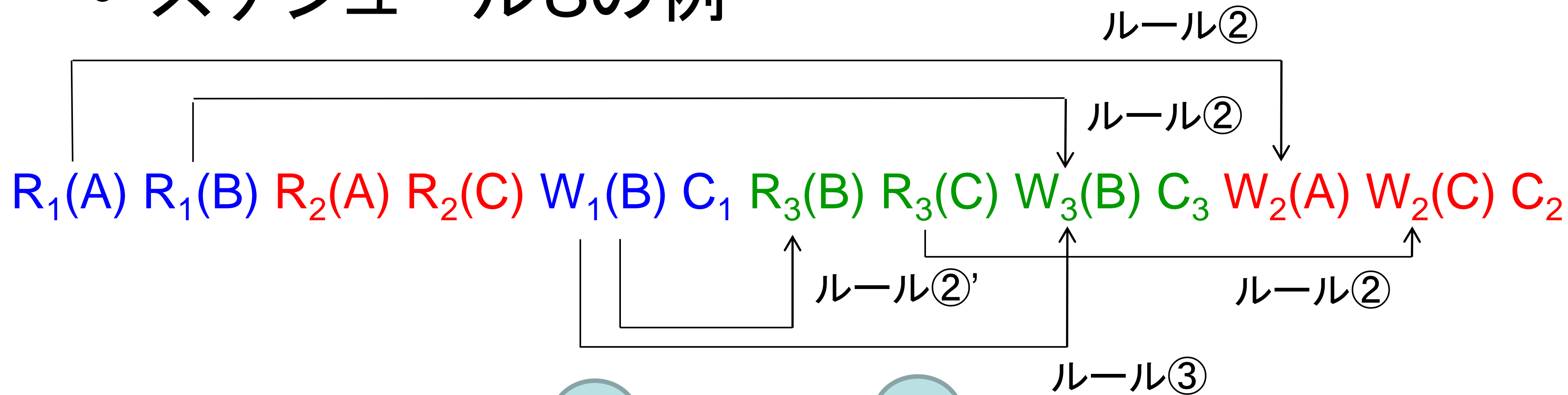
競合直列可能かの判定（1）

- スケジュールSに対する先行グラフ（precedence graph）を作成
  - ① Sに参加する各トランザクションT_iに対し，ノード$N(T_i)$を作成
  - ② $R_i(A)$（$W_i(A)$）が$W_j(A)$（$R_j(A)$）に先行するとき有向エッジ$N(T_i) \rightarrow N(T_j)$をひく
  - ③ $W_i(A)$が$W_j(A)$に先行するとき有向エッジ$N(T_i) \rightarrow N(T_j)$をひく
- 先行グラフにサイクル（閉路）がなければSは競合直列可能で，サイクルがあれば競合直列可能でない

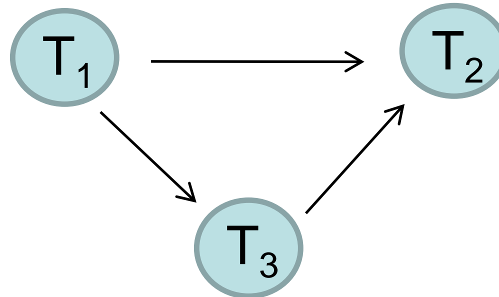


競合直列可能かの判定(2)

- スケジュールSの例



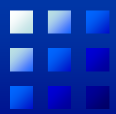
  - 先行グラフ



サイクルがないので
競合直列可能

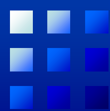
  - トポロジカルソートで
    等価な直列スケジュールが得られる

$R_1(A)$ $R_1(B)$ $W_1(B)$ C_1 $R_3(B)$ $R_3(C)$ $W_3(B)$ C_3 $W_2(A)$ $W_2(C)$ C_2 $R_2(A)$ $R_2(C)$



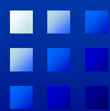
まとめ:競合等価とビュー等価の比較

- ビュー等価の方が条件が緩い
  - スケジュールがより柔軟に選択できる:先の例では両方のスケジュールを選択可能
  - 計算量がNP完全:実際の利用には適さない
- 競合等価は現実的な解
  - 競合直列可能かの判定が容易
  - 柔軟性にはやや劣る
    - ビュー等価の立場で直列可能なスケジュールが競合等価の立場で直列可能とならない場合があるため，可能なスケジュールの候補が少なくなりうる



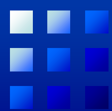
スケジュールの諸性質(1)

- これまでの議論はコミットされたトランザクションのみを対象：しかし，トランザクションはアボートされる可能性もある
- アボートに着目した場合の性質について
- A) 回復可能性(recoverable)
  - 「R_i(A)で読まれるA値がW_j(A)によって書かれた値であり，T_iがコミットするときにはC$_j$がC$_i$に先行する」という条件が常に成立
  - 回復可能でない例：$W_1(A)$ $R_2(A)$ C_2 A_1
    - T_1が書いたA値をT_2が読んでコミットしているので，T_1をアボートしても，T_2をもう取り消しできない



スケジュールの諸性質(2)

- 連鎖的アボート(cascading abort):あるトランザクションのアボートが他のトランザクションのアボートを連鎖的に引き起こす現象
  - $W_1(A)$ $R_2(A)$ A_1 C_2ならば回復可能だが，T_1をアボートするとT_2もアボートする必要あり
- B)連鎖的アボートの回避
  - 「$R_i(A)$で読まれるA値が$W_j(A)$によって書かれた値であるときにはC_jが$R_i(A)$に先行する」という条件が常に満たされるとき
  - トランザクションは取り消されうる値を読まない
  - 常に回復可能:その逆は成立しない

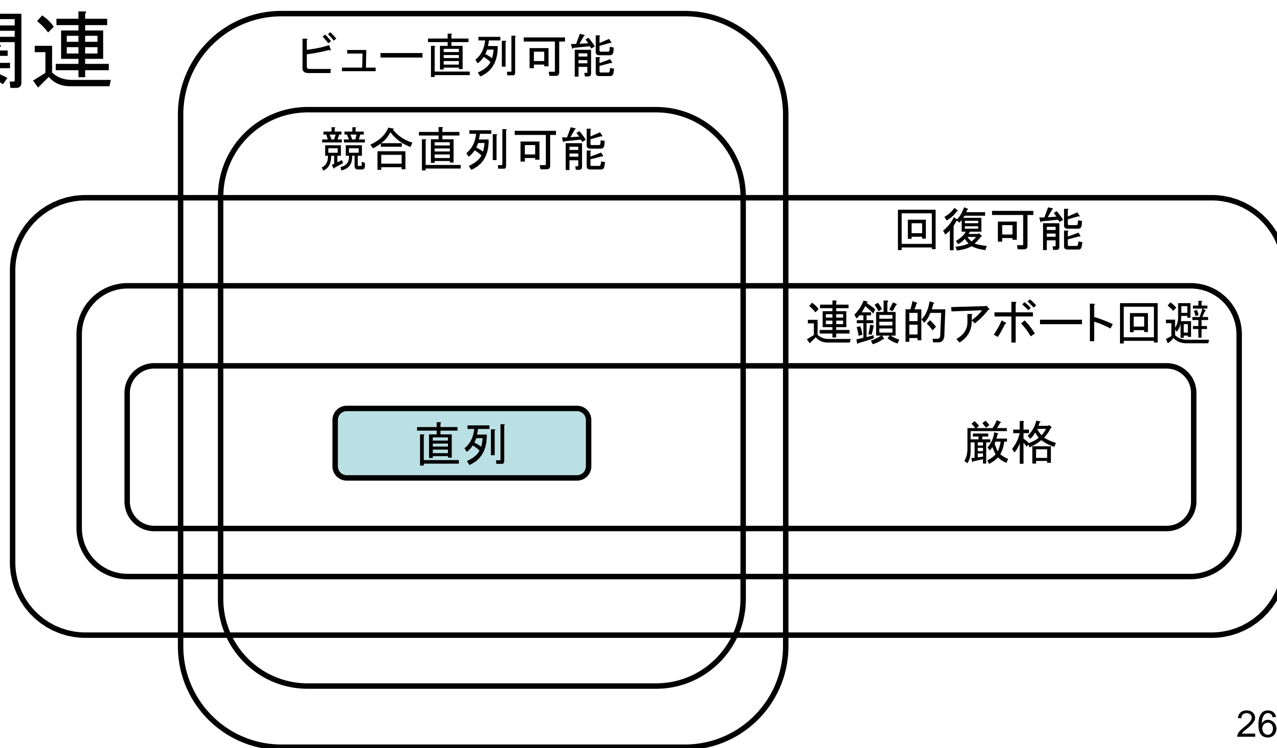


スケジュールの諸性質(3)

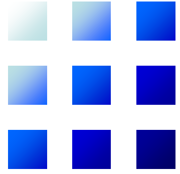
- C)厳格性(strictness)
  - 「R_i(A)またはW_i(A)よりもW_j(A)が先行するときには，C_jまたはA_jがそのR_i(A)またはW_i(A)に先行する」という条件が常に満たされるとき
  - 厳格なスケジュールは連鎖的アボートを回避：逆は成り立たない
  - 例：W_1(A) W_2(A) C_2 A_1は連鎖的アボートを回避するが，厳格ではない
  - 厳格なスケジュールではアボート処理が簡単
    - T_iのアボート時には，T_iがwriteした値をwrite前の値に戻せばよい
    - 厳格でない場合(上の例)では面倒

スケジュールの諸性質：まとめ

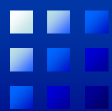
- 回復可能性，連鎖的アボートの回避，厳格性は直列可能性とは直交
  - 例：$R_1(A)\ W_2(A)\ W_2(B)\ C_2\ R_1(B)\ C_1$ は厳格であるが直列可能でない
- 各性質の関連



直列スケジュールは
常に直列可能で
厳格なスケジュール

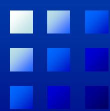


ロックを用いた同時実行制御



ロックの概念

- 実際のDBMSでは，何らかの機構・規約を用いて同時実行を制御
- ロック（lock）
  - もっとも一般的な機構
- 単純な例：各項目Aに対する排他的ロック
  - Aにロックをかけることができるのは，ある時点では1トランザクションに限る
  - すでにロックされている項目へのロック要求は，そのロック解除まで待ち状態となる
  - 並行性が低いという問題点

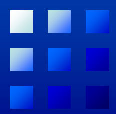


共有ロックと専有ロック（1）

- readのみの場合に排他的なロックをかけるのは，並行性を必要以上に落としてしまう
- 解決策：二つのロックに分ける
- 共有ロック（shared lock, S lock）
  - 読出しの場合に用いる
  - 共有ロック同士は両立
- 専有ロック（exclusive lock, X lock）
  - 書込み時に用いる排他的ロック

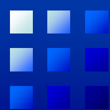
	共有（S）	専有（X）
共有（S）	Y	N
専有（X）	N	N

両立性行列
（compatibility matrix）



共有ロックと専有ロック(2)

- ロックの変換(conversion)
  - いったんデータを読み出した後，その値に応じて書込みを行うかを決定することも多い
  - ロックのアップグレード
    - 共有ロックを専有ロックに変換する処理
    - 他のトランザクションがその項目を共有ロックしていない場合のみ許可される
  - ロックのダウングレード
    - 専有ロックを共有ロックに変換

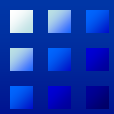


ロッキングプロトコル

- 例：図8.2(b)のトランザクション実行例に対するロック操作

$XL_2(A)$ $R_2(A)$ $W_2(A)$ $UL_2(A)$ $SL_1(A)$ $SL_1(B)$ $R_1(A)$ $R_1(B)$ $UL_1(A)$ $UL_1(B)$ C_1 $XL_2(B)$ $R_2(B)$ $W_2(B)$ $UL_2(B)$ C_2

  - XL / SLはX / Sロックをかける操作，UL はロックを解除する操作を表す
  - この操作例は両立性を満たすが，そもそも図8.2(b)の実行例は不整合な例
- ロッキングプロトコル（locking protocol）
  - ロックをかける操作と解く操作に関する規約
  - 合法（legal）：プロトコルに従ったスケジュール



デッドロック

- デッドロック（deadlock）：ロックを用いた場合に発生
- 例
  - T_1: ① $XL_1(A)$ ③ $XL_1(B)$ $W_1(A)$ $W_1(B)$ $UL_1(A)$ $UL_1(B)$
  - T_2: ② $XL_2(B)$ ④ $XL_2(A)$ $W_2(B)$ $W_2(A)$ $UL_2(B)$ $UL_2(A)$
  - 問題点：①の$XL_1(A)$の後に②の$XL_2(B)$を実行してしまうと，それ以降の③，④ともにロックをかけるのに失敗してしまう
- 対策については後述

二相ロッキングプロトコル(1)

- 二相ロッキングプロトコル(two phase locking protocol; 2PL)
  - 競合直列可能性を保証するロッキングプロトコル
- ロック操作を二つの部分に分離
  - 成長相(growing phase):ロックをかけるだけからなる
  - 縮退相(shrinking phase):ロックを解く操作だけ
  - いったんロックを解いた後に再びロックをかけてはいけない
  - 成長相ではロックのアップグレードが，縮退相ではダウングレードが許される

二相ロッキングプロトコル(2)

- 例(先と同じ)

T_1の成長相

$XL_2(A)\ R_2(A)\ W_2(A)\ UL_2(A)\ SL_1(A)\ SL_1(B)\ R_1(A)$
$R_1(B)\ UL_1(A)\ UL_1(B)\ C_1\ XL_2(B)\ R_2(B)\ W_2(B)\ UL_2(B)\ C_2$

T_1の縮退相

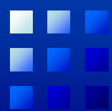
  - T_1は二相ロッキングプロトコルに従っている
  - T_2はそうでない
- すべてのトランザクションが二相ロッキングプロトコルに従うことが必要
- 二相ロッキングプロトコルではデッドロックが発生する可能性あり:先の例

厳格な二相ロッキングプロトコル

- アボート操作前に専有ロックを解くと回復可能性のないスケジュールが生じうる
- 例：二相ロッキングプロトコルに従うT_1, T_2, T_3

$XL_1(A)$ $R_1(A)$ $W_1(A)$ $UL_1(A)$ $SL_2(A)$ $R_2(A)$ $UL_2(A)$ C_2 $SL_3(A)$ $R_3(A)$ A_1

  - T_1がアボートするとT_3が連鎖的にアボートされる
  - T_2はすでにコミット済みなので取り消し不能
- 厳格な(strict)二相ロッキングプロトコル
  - 縮退相における最初のロックを解く操作はトランザクションのコミットまたはアボート操作の後
  - 競合直列可能であり厳格性も満たす

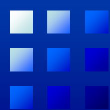


デッドロック

- デッドロックへの対処策
  - デッドロックの検出を行う方法
  - デッドロックを回避する方法（省略）
- デッドロックの検出
  - 待ちグラフ（wait-for graph）による方法
    - 各トランザクションT_iに対してノードN(T_i)を作成
    - T_iが他のトランザクションT_jのロックが解かれるのを待っているとき，有向エッジN(T_i) ⇒ N(T_j)を引く
    - 待ちグラフにサイクルがあればデッドロック：犠牲者（victim）を選びアボート
  - タイムアウトによる方法：一定時間以上待ち状態のトランザクションをアボートする

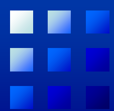
時刻印を用いた同時実行制御

- 時刻印順(timestamp ordering)方式
  - 各トランザクションに，発生順に一意な時刻印を与える
  - 時刻印の順にトランザクションを逐次実行する場合と等価なスケジュールが生じるように制御
- 各項目*A*に二種類の時刻印を持たせる
  - RTS(*A*):これまでに*A*のreadを行ったトランザクションの時刻印のうち最大値
  - WTS(*A*):これまでに*A*のwriteを行ったトランザクションの時刻印のうち最大値



基本時刻印方式(1)

- 時刻印TS(T_i)をもつトランザクションT_iの項目Aのread/write操作の規約は以下のとおり
- A) read
  ① TS(T_i) < WTS(A)のとき：本来T_iが読みだすべきAの値はwriteにより失われてしまっているので，T_iはアボートする
  ② それ以外のとき：T_iはAのreadを行うとともに，RTS(A) = max(RTS(A), TS(T_i))とする



基本時刻印方式(2)

- B) write
  - ① TS(T_i) < RTS(A)のとき: T_iがwriteを行った後のA値をreadで読むべきトランザクションが先にreadを行ってしまっているので，T_iはアボートする
  - ② TS(T_i) ≥ RTS(A)かつTS(T_i) < WTS(A)のとき: 同様に，T_iはwriteの時期を逸してしまっているのでアボートする
  - ③ それ以外のとき: T_iはAのwriteを行うとともに，WTS(A) = max(WTS(A), TS(T_i))とする

実行例①：readに関する規約

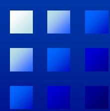
t	T_1	T_2	T_3	処理
0	begin			$\mathrm{TS}(T_1) \leftarrow 0$
1		begin		$\mathrm{TS}(T_2) \leftarrow 1$
2			begin	$\mathrm{TS}(T_3) \leftarrow 2$
3		read(A)		$\mathrm{RTS}(A) \leftarrow \max(\mathrm{RTS}(A), \mathrm{TS}(T_2)) = 1$
4		write(A)		$\mathrm{WTS}(A) \leftarrow \max(\mathrm{WTS}(A), \mathrm{TS}(T_2)) = 1$
5	read(A)			$\mathrm{TS}(T_1)$ (= 0) < $\mathrm{WTS}(A)$ (= 1) が成りたつので T_1をアボートする（規約①）
6			read(A)	$\mathrm{RTS}(A) \leftarrow \max(\mathrm{RTS}(A), \mathrm{TS}(T_3)) = 2$

- T_1をアボートする理由：時刻印が後であるT_2が書いた値をreadすることになるので，T_1 T_2 T_3 という直列実行の順序に違反
- 一方，T_3によるreadは直列実行の順序に違反しない
- なお，RTS(A), WTS(A)の初期値は $-\infty$とする

実行例②：writeに関する規約

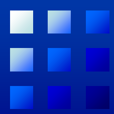
t	T_1	T_2	T_3	T_4	処理
0	begin				$TS(T_1) \leftarrow 0$
1		begin			$TS(T_2) \leftarrow 1$
2			begin		$TS(T_3) \leftarrow 2$
3				begin	$TS(T_4) \leftarrow 3$
3		read(A)			$RTS(A) \leftarrow \max(RTS(A), TS(T_2)) = 1$
4	write(A)				$TS(T_1)$ (= 0) < $RTS(A)$ (= 1) なので，T_1をアボートする（規約①）
5				write(A)	$WTS(A) \leftarrow \max(WTS(A), TS(T_4)) = 3$
6			write(A)		$TS(T_3) \geq RTS(A)$であるが$TS(T_3) < WTS(A)$なので，T_3をアボートする（規約②）

- トマスの書き込み規則（Thomas's write rule）：
  - writeにおける規約②は緩和でき，writeを許すことが可能
  - ただし，結果のスケジュールは必ずしも直列可能とならない



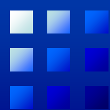
楽観的同時実行制御

- 楽観的同時実行制御（optimistic concurrency control）
- 適する状況
  - ほとんどのトランザクションがreadだけ
  - 複数トランザクションの同時発生がまれ
- アイデア
  - とりあえず他のトランザクションと競合しないと仮定してトランザクションを実行
  - 終了時に競合がなかったかを確認：競合していたらアボート処理などへ進む



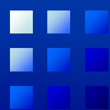
手法の概要

- A) 読出し(read)フェーズ
  - トランザクションの処理を実行
  - ただし，readではトランザクションの固有の作業領域に読み出し，writeでは作業領域上で更新
- B) 確認(validation)フェーズ
  - 他との競合がないかを確認
- C) 書込み(write)フェーズ
  - 確認処理をパスしたら，作業領域内で更新された項目の書き込みを行い，トランザクションをコミット
  - パスしなかったらトランザクションをアボート



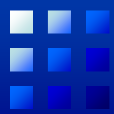
確認処理

- 各トランザクションT_iに以下の時刻印を与える
  - Start(T_i) : 読み出しフェーズの開始時刻
  - Validate(T_i) : 確認フェーズの開始時刻
- また，以下の集合を定める
  - rset(T_i) : T_iが読み出しフェーズでreadした項目
  - wset(T_i) : T_iが読み出しフェーズでwriteした項目
- T_iの確認フェーズでは，Start(T_i) < Validate(T_j) < Validate(T_i)なる時刻印を持つすべてのT_jについてrset(T_i) ∩ wset(T_j) = ∅ならば確認をパスしたとする



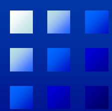
実行例

t	T_1	T_2	T_3	処理
0	読出し開始			Start(T_1) ← 0
1		読出し開始		Start(T_2) ← 1
2			読出し開始	Start(T_3) ← 2
3	read(A)			rset(T_1) ← {A}
4		read(B)		rset(T_2) ← {B}
5			read(A)	rset(T_3) ← {A}
6			write(B)	wset(T_3) ← {B}
7			確認開始	Validate(T_3) ← 7. 現時点で他と干渉しないのでコミットする
8	確認開始			Validate(T_1) ← 8. T_3に対しStart(T_1) < Validate(T_3) < Validate(T_1)で，rset(T_1) ∩ wset(T_3) = {A} ∩ {B} = ∅なのでコミットする
9		確認開始		Validate(T_1) ← 9. T_3に対しStart(T_2) < Validate(T_3) < Validate(T_2)で，rset(T_2) ∩ wset(T_3) = {B} ∩ {B} ≠ ∅なのでアボートする



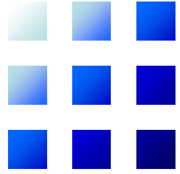
多版同時実行制御

- 多版同時実行制御（multiversion concurrency control; MVCC）
- 各項目Aに対しwriteがなされるたびに，Aに対する新しい版（version）を生成し維持管理
- read/writeの際は，時刻印などの情報を用いて版の新しさを考慮してアボート処理を制御
- 最近のDBMS（Oracle, PostgreSQLなど）では，ロックに基づく手法と多版同時実行制御の組合せがよく見られる

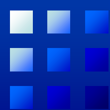


最近の動向

- 一般のDBMSでは多版同時実行制御(MVCC)をとるものが主流:Oracle等
- 新たな問題
  - マルチコア, マルチスレッド:多くの競合が発生
  - 不揮発性メモリ:入出力速度の向上 ⇒ トランザクション処理の速度が与える影響の割合が増大
  - HTAP(ハイブリッド型トランザクション/アナリティクス処理):入ってきたデータをどんどん分析 ⇒ トランザクション処理の効率化が重要に
- 競合によるアボート処理をどれだけ減らせるか:楽観的手法が再び注目

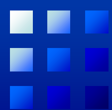


余談：SQLの同時実行制御



SQLの同時実行制御

- 隔離レベルに対する4つのオプション
  - 非コミット読取り(read uncommitted):コミットされていないデータを読むこと(ダーティーリード)がある
  - コミット済み読取り(read committed):以前readした項目Aの値を再度readしたとき，その値が他のトランザクションにより変更されていることがある
  - 再読込み可能読取り(repeatable read):ある条件で複数回検索したとき，新たな行(phantom; 幽霊)が追加されていることがある
  - 直列可能(serializable):直列可能性を保証
- 多くのDBMSではコミット読取りがデフォルト設定
  - 効率化のため:ACID特性のI(隔離性)は完全には保証されないため，開発者側で意識する必要あり



同時実行制御の指定

- データの挿入・更新・削除（SQLのINSERT / UPDATE / DELETE）の際に，明示的にロックをかける必要はない
  - DBMSが自動的にロックをかけ，不要になれば解除
- 明示的にロックをかける機能もあり
  - 例：OracleのFOR UPDATE句
    - 更新を前提としてデータの読取りを行う場合に使用
    - トランザクション終了まで検索結果にロックをかける
    - SELECT * FROM TABLE FOR UPDATE