データベースシステム

同時実行制御

トランザクションの同時実行

Transaction 1(T1)

```
begin
    read(x)
    write(x)
commit
```

Transaction 2(T2)

```
begin
    read(x)
    write(x)
    write(y)
commit
```

スケジュール例

S1

	T1	T2
t1	read(x)	
t2		read(x)
t3	write(x)	
t4		write(x)
t5		write(y)

S2

	T1	T2
t1	read(x)	
t2		read(x)
t3		write(x)
t4	write(x)	
t5		write(y)

S3

	T1	T2
t1	read(x)	
t2	write(x)	
t3		read(x)
t4		write(x)
t5		write(y)

$S=r_1(x)r_2(x)w_1(x)w_2(x)w_2(y)c_1c_2$

同時実行制御の必要性

複数のトランザクションを何も考えずに同時実行すると発生する異状

- 遺失更新異状(lost update anomaly)
  - あるトランザクションの更新が別の更新によって取り消されてしまう
- 不整合検索異状
  (inconsistent retrieval anomaly)
  - 一貫性のないデータを利用してしまう
- 汚読異状(dirty read anomaly)
  - 依存しているトランザクションがロールバックしてしまう

遺失更新異状を引き起こす例

- A子とB男が夫婦共通の口座から同時に現金を引き出すことを想定。
  - A子：30万引き出す
  - B男：20万引き出す

S1=r1(A)r2(A)w1(A)w2(A)c1c2

時刻	トランザクションT1 の実行	トランザクションT2の実行
t1	read(A)	
t2		read(A)
t3	write(A:=A-30)	
t4		write(A:=A-20)
t5	commit	
t6		commit

直列実行したとき、Aの口座額は？

上記のスケジュールの場合、Aの口座額は？

不整合検索異状を引き起こす同時実行例

- ユーザX:口座Aから口座Bに100万振り替え送金する
- ユーザY:口座Aと口座Bの額を合計する　A=200,B=300

S2=r1(A)w1(A)r2(A)r2(B)r1(B) w1(B)c1c2

時刻	トランザクションT1 の実行	トランザクションT2の実行
t1	read(A)	
t2	write(A:=A-100)	
t3		read(A)
t4		read(B)
t5	read(B)	
t6	write(B:=B+100)	
t7	commit	
t8		commit

直列実行したときのCの値は？

このスケジュールで実行したときのCの値は？

汚読異状を引き起こすスケジュールの例

- T2がコミットした後にT2がアボートしてしまう

S3=r1(A)w1(A)r2(A)r2(B)c2 a1

A=200

時刻	トランザクションT1 の実行	トランザクションT2の実行
t1	read(A)	
t2	write(A:=A+10)	
t3		read(A)
t4		write(A:=A-50)
t5		commit
t6	abort	

直列化可能の判定

- どのようなスケジュールなら異状が起きないのか

直列化可能であるスケジュール

- スケジュールAで実行されたトランザクション群が、直列化したスケジュールと等価になれば「直列化可能」であるという。
- 「等価」の種類
  - 最終状態等価 (Final State Equivalence)
  - ビュー等価 (View Equivalence)
  - 相反等価 (Conflict Equivalence)

最終状態等価(Final State Equivalence)

- 二つのスケジュールSとS'の実行結果が同じ
- S'が直列スケジュールならばSは最終状態直列化可能(Final State Serializable:FSR)であるという
- 問題:不整合検索異状を起こす場合がある

S

	T1	T2
t1	read(A)	
t2		read(A)
t3	write(A:=A-30)	
t4		write(A:=A-20)
t5	commit	
t6		commit

S'

	T1	T2
t1	read(A)	
t2	write(A:=A-30)	
t3		read(A)
t4		write(A:=A-20)
t5	commit	
t6		commit

ビュー等価（View Equivalence）

不整合検索異状を引き起こす例はFSRである

S=r1(A)w1(A)r2(A)r2(B)r1(B) w1(B)c1c2

書き込みの結果が同じか、だけでなく読み込んだ値も同じかを確認しなければならない

- ビュー等価（View Equivalence）
  - 二つのスケジュールSとS'が最終状態等価であり，さらにSのステップs（=$r_i(x)$）が、先行するステップs'（＝$w_j(x)$）の結果を読んでいるならば、S'でもそうでなければいけない
  - S'が直列スケジュールのときSはビュー直列化可能（View Serializable: VSR）であるという
  - 問題：スケジュールSとビュー等価なスケジュールを探すアルゴリズムがNP完全

相反等価（Conflict Equivalence）

- 相反関係（Conflict Relation）
  - 二つのオペレーションop1,op2が同じデータをアクセスして，少なくともどちらか一方がwriteであるときop1とop2は相反している

    S_1=.....$r_1(x)$..$r_1(y)$...$w_2(x)$...$r_1(x)$..$r_2(y)$...c_1c_2

- 相反等価（Conflict Equivalence）
  - スケジュールSとS’が全く同じ相反関係を持っているとき，相反等価であるという
  - S’が直列スケジュールのとき，Sを相反直列化可能（Conflict Serializable：CSR）であるという
  - 線形時間で解ける方法がある

3種類の直列化可能の関係

- スケジュールが直列化可能か確認する場合は相反直列化可能性をチェックする

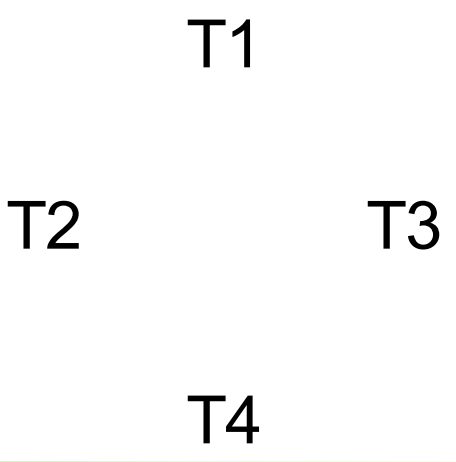
相反グラフ解析

- 正しいスケジュールかどうかを判定する、多項式時間で解ける手法
- 相反グラフ
  - トランザクション群{T1,T2,…,Tn}のスケジュールをSとし、Sの相反グラフをCG(S)で表す
    (1) CG(S)はn個のノードT1,T2,…,Tnからなる
    (2) あるデータ項目xにおいて次のいずれかが成立するとき
    - ステップTi:read(x)がステップTj:write(x)に先行する
    - ステップTi:write(x)がステップTj:write(x)に先行する
    - ステップTi:write(x)がステップTj:read(x)に先行する

$S=r_3(x)r_1(x)w_3(x)r_2(x)r_1(y)r_4(x)w_1(y)r_2(y)w_4(x)r_2(y)$

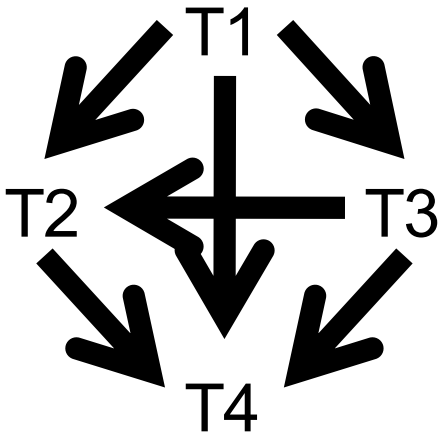
	T1	T2	T3	T4
t1			read(x)	
t2	read(x)			
t3			write(x)	
t4		read(x)		
t5	read(y)			
t6				read(x)
t7	write(y)			
t8		read(y)		
t9				write(x)
t10		read(y)		

相反直列化可能性



スケジュールSが相反直列化可能であるための必要かつ十分条件はSの相反グラフCG(S)が非巡回であること、またSに相反等価な直列スケジュールはCG(S)をトポロジカルソートすることにより得られる

	T1	T2	T3	T4
t1			read(x)	
t2	read(x)			
t3			write(x)	
t4		read(x)		
t5	read(y)			
t6				read(x)
t7	write(y)			
t8		read(y)		
t9				write(x)
t10		read(y)		



相反直列化可能性

スケジュールSが相反直列化可能であるための必要かつ十分条件はSの相反グラフCG(S)が非巡回であること、またSに相反等価な直列スケジュールはCG(S)をトポロジカルソートすることにより得られる

二つのスケジュールが相反等価であればビュー等価となる理由

- ビュー等価の条件
  - read-from関係（ $w_i(x) \rightarrow r_j(x)$ ）が等しい
- 相反等価の条件
  - 相反関係にread-from関係が含まれている

2相ロック法

- 直列化可能性判定、相反グラフ解析
  - まずトランザクション群に対して、非直列化スケジュールを作成し、それが直列化可能かどうかを判定する「静的」なアプローチ
  - 同時に実行するトランザクション群がわかってから直列化なスケジュールを立てる
- 「動的」なアプローチが必要
  - あるプロトコルでトランザクションを実行させることにより、結果的に直列化可能となっている

ロック法(locking)

ロッキングプロトコル

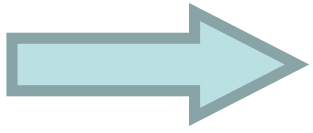
- 最も単純なやり方：排他ロック
  - 項目Aにアクセスをする前にロックをかける
  - ロックがかかっている間はほかのトランザクションはAにアクセスできない
- 排他ロックだと困ってしまう事例：
  - 多くのトランザクションが同時に項目Aを読み込む（read(A)）
  - 読み込み同士ならば排他制御をしないほうが効率的
- 以下のようにロックの種類を設定する
  - 共有ロック
    (Shared Lock, Read Lock)
    データ読み込みのための
    ロック
  - 専有ロック
    (eXclusive Lock, Write Lock)
    データ書き込みのための
    ロック

	共有 (rl)	専有 (wl)
共有 (rl)	Y	N
占有 (wl)	N	N

ロッキングプロトコル

```
begin
  read(y)
  write(y:=y+10)
commit
```

S=r(y)w(y)c



```
begin
  rl(y)
  read(y)
  wl(y)
  write(y:=y+10)
  wu(y)
commit
```

ロックのアップグレード（共有ロックから占有ロックへ）はアンロックなしにできる。

S=rl(y)r(y)wl(y)w(y)wu(y)c

ロッキングプロトコルだけではうまくいかない

- トランザクションt_1, t_2による以下のスケジュールSについて考えてみる
  - $t_1 = r_1(x) w_1(y)$
  - $t_2 = w_2(x) w_2(y)$

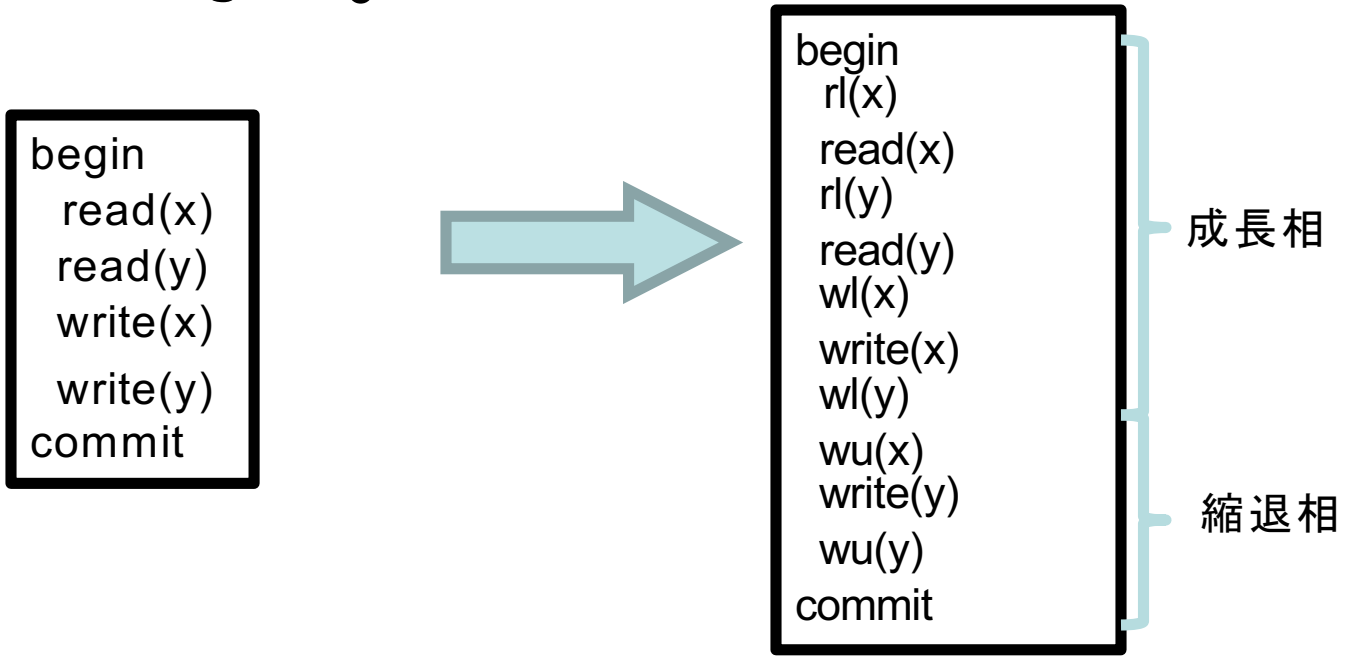
$S = rl_1(x) r_1(x) ru_1(x) wl_2(x) w_2(x) wl_2(y) w_2(y) wu_2(x) wu_2(y) c_2$
$wl_1(y) w_1(y) wu_1(y) c_1$

$DT(S) = r_1(x)\ w_2(x)\ w_2(y)\ w_1(y) c_1 c_2$

このスケジュールはCSRかな？

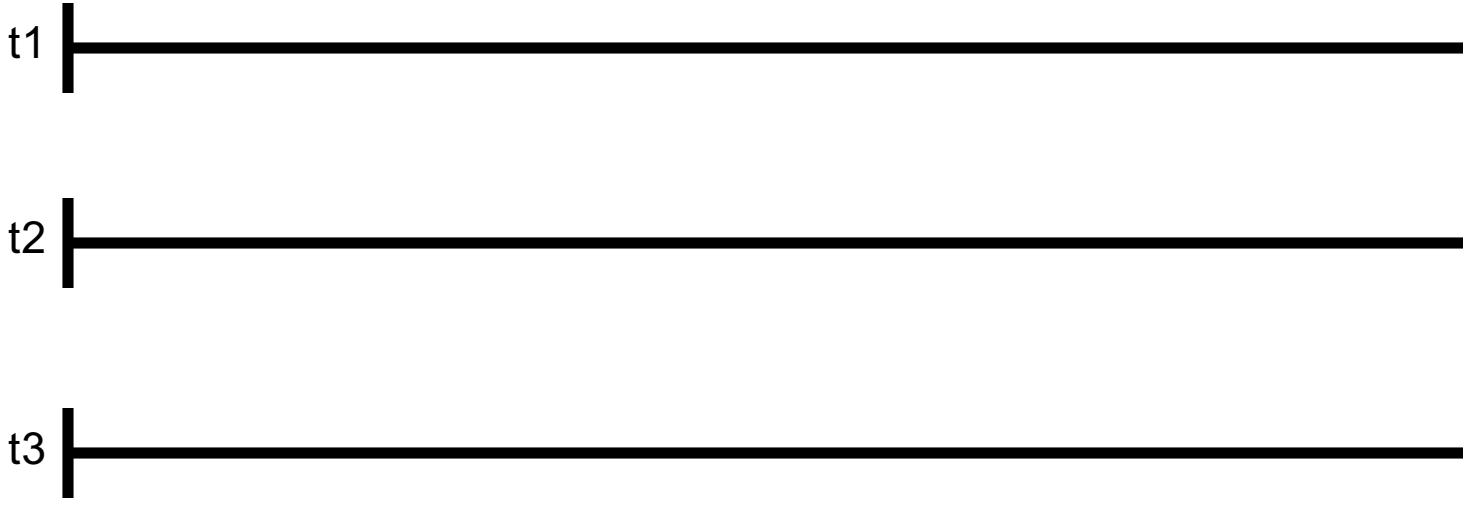
2相ロック(2 phase lock)

- トランザクションはデータ項目のロックが不必要となったらアンロックする。しかしトランザクションは、読み書きのために必要なすべてのロックが完了する前にそれらをアンロックすることはしない。



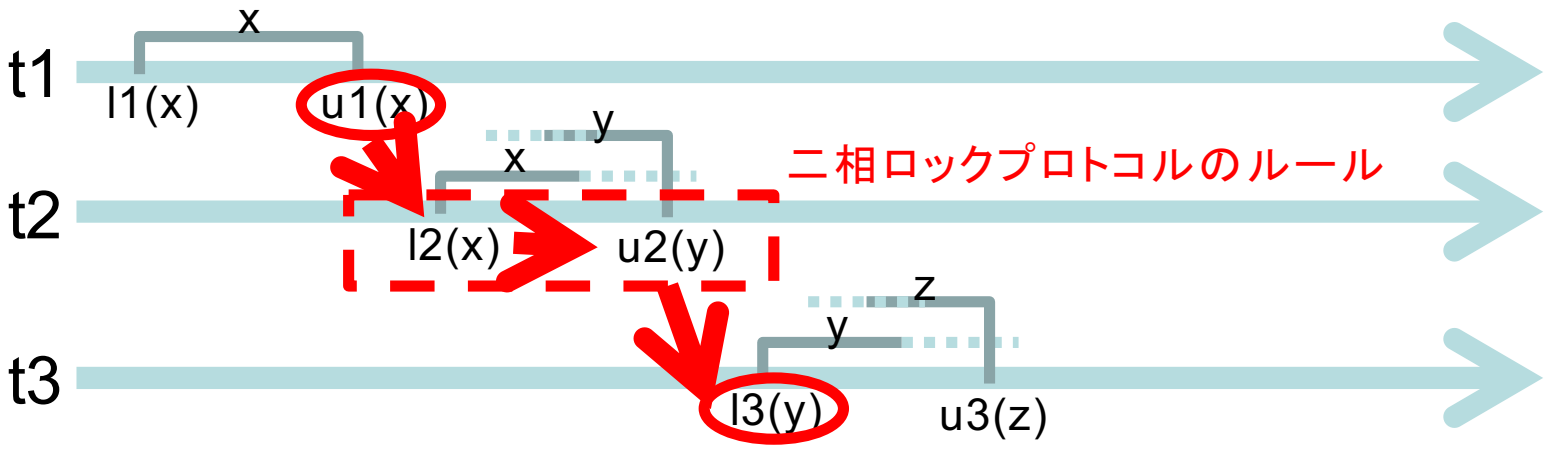
2相ロッキングを使ってスケジューリングせよ

- $S=w_1(x)r_2(x)w_1(y)w_1(z)r_3(z)c_1w_2(y)w_3(y)c_2w_3(z)c_3$



二相ロックの結果スケジュールが相反直列化可能である理由

- 二相ロックの結果実行されたスケジュールsの相反グラフを考える
  - 相反グラフに t1,…,tnのパスがあるとき
    u1(x) < ln(y)となる



  - 相反グラフに巡回パスが含まれていたら u1(x) < l1(y) となり二相ロックプロトコルに反するため、相反グラフには巡回パスは含まれない→相反直列化可能

デッドロック

- 複数のトランザクションが互いに別のトランザクションのロック待ち状態になってしまうこと

トランザクションT1

```
begin
 read(x)
 write(y)
end
```

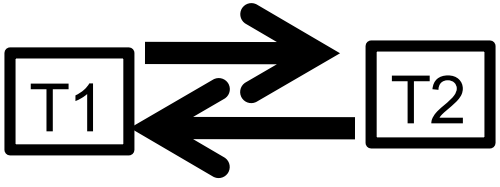
トランザクションT2

```
begin
   read(y)
   write(x)
end
```

	T1	T2
t1	S(x)	
t2		S(y)
t3	read(x)	
t4		read(y)
t5	X(y)待ち	X(x)待ち

デッドロックの検出

- 待ちグラフ(wait-for graph)
  - 各トランザクションTiをノードN(Ti)とする
  - トランザクションTiがほかのトランザクションTjのかけているロックがとかれるのを待っているとき有効辺 N(Ti)→N(Tj)
  - グラフがサイクルを持ったらデッドロック発生



- タイムアウトによる検出
  - ある一定以上待ち状態が発生しているトランザクションはデッドロックに陥っている可能性が高いとする

時刻印(Timestamp)によるデッドロック回避

- 各トランザクションTiの開始時刻TS(Ti)を記録しておく
- デッドロックを検出したとき以下のいずれかの方法でロックを解除する
  - wait-die方式
    - TS(Ti)<TS(Tj)のとき、TiはTjのロック解除を待つ
    - TS(Ti)>TS(Tj)のとき、Tiはアボートする
  - wound-wait方式
    - TS(Ti)>TS(Tj)のとき、TiはTjのロック解除を待つ
    - TS(Ti)<TS(Tj)のとき、Tiはアボートする

おまけ：分離レベル

二相ロックはめちゃくちゃ厳しい

- 二相ロックによって相反直列化可能性は確実に保証される
- しかし、二相ロックの結果はほとんど直列スケジュールに近くなってしまう

アプリケーションの状況により
ちょっと緩いロッキングプロトコルを採用する場合もある

分離レベル

複数のトランザクションが同時に実行されたときに、各トランザクションが他のトランザクションの影響からどの程度分離されているか

- 直列化可能
  - 全てのトランザクションは他のトランザクションの影響を全く受けない
- READ UNCOMMITTED
  - コミットされていないデータを読んでしまう場合がある
- READ COMMITTED
  - コミットしたデータしか読まないが、一度読んだデータが他のトランザクションによって変更される場合がある
- REPEATABLE READ
  - 一度読んだデータは他のトランザクションに変更されることはないが、読込をした時に存在しなかったデータが他のトランザクションによって追加される場合がある

READ UNCOMITTED

- もっとも緩いロッキングプロトコルを採用する
  - writeに対しては二相ロックに従うが readに関してはロックをかけない
  - ロックしないで読込を実行するので、ほかのトランザクションが処理中のデータを読んでしまう場合がある(dirty read)
- 役に立つケース
  - 更新が頻繁
  - 個々のデータを正確に読む必要がなく、データ全体の大まかな統計値だけが必要な場合
    - 常時500万人がログイン・ログアウトするSNSの利用者年齢層をリアルタイムに調べるために、年代ごとの人数をとる場合

READ COMMITTED

- dirty readを防ぐプロトコル
  - writeに対しては二相ロックに従う
  - readに関しては、readロックをかけようとしたときの他のトランザクションによってロック中だった場合は、そのトランザクションが実行される前、最後にコミットされた時の値を読み込む
    - マルチバージョン同時実行制御 (MVCC)
- 役に立つケース
  - 常時多くの利用者からのアクセスがある
  - 更新作業はデータの読込結果に依存しない
- 問題
  - 遺失更新異常の発生を防ぐことができない
    - 一度読んだ値を基に値を更新したとき、その途中で他のトランザクションによって更新があったら、その更新を上書きしてしまう
    - **MVCCではトランザクションのタイムスタンプを記録し，更新の衝突を管理→遺失更新異状をおこすトランザクションをアボートさせる**

REPEATABLE READ

- 基本的に直列化可能性とほとんど同じ程度の分離レベル
- 但し、トランザクション処理中に他のトランザクションによって新たなデータが追加されたりいくつかのデータが削除される場合がある（phantom read）
- 実現方法
  - ロックの単位をレコード単位にする
- 役に立つケース
  - データの挿入や削除がほとんどない場合

データベースサーバの基本分離レベル

- MySQL (InnoDB)
  - REPEATABLE READ
- PostgreSQL
  - READ COMMITTED
- Oracle
  - READ COMMITTED
- SQL Server
  - READ COMMITTED