データベースS

第13回　同時実行制御

システム創成情報工学科 尾下真樹

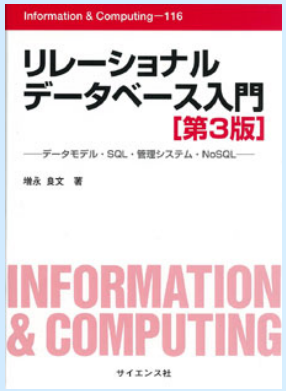
2018年度 Q2

今日の内容

- 前回の復習
- 同時実行制御
  - トランザクション
  - トランザクションの同時実行制御
  - 計画的な並行処理
  - 逐次的な並行処理

教科書

- 「リレーショナルデータベース入門［第3版］」
  増永良文 著、サイエンス社
  - 10章 10.2～10.3
  - 11章 11.1～11.4
- 「データベースシステム」
  北川 博之 著、昭晃堂 出版
  - 8章 147～168ページ

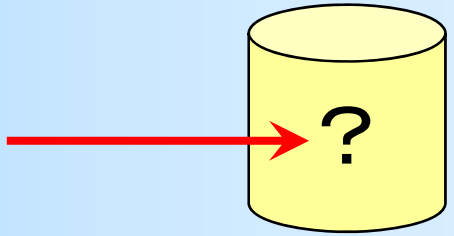


前回の復習

物理的データ格納方式

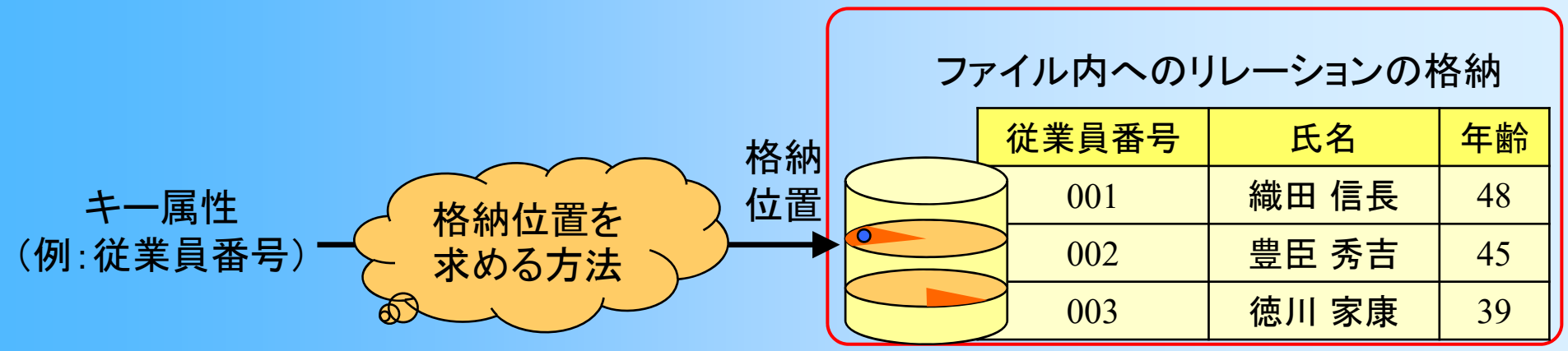
- リレーショナルデータモデルの特徴
  - 抽象的な「リレーショナルモデル」として定義
  - リレーションがメモリやハードディスクにどのように格納されるかは規定していない
    → 論理的モデルと物理的構造の分離
  - 各データベースシステムごとに、物理的なデータ構造を工夫することができる

従業員番号	部門番号	氏名	年齢
0001	01	織田 信長	48
0002	02	豊臣 秀吉	45
0003	03	徳川 家康	39



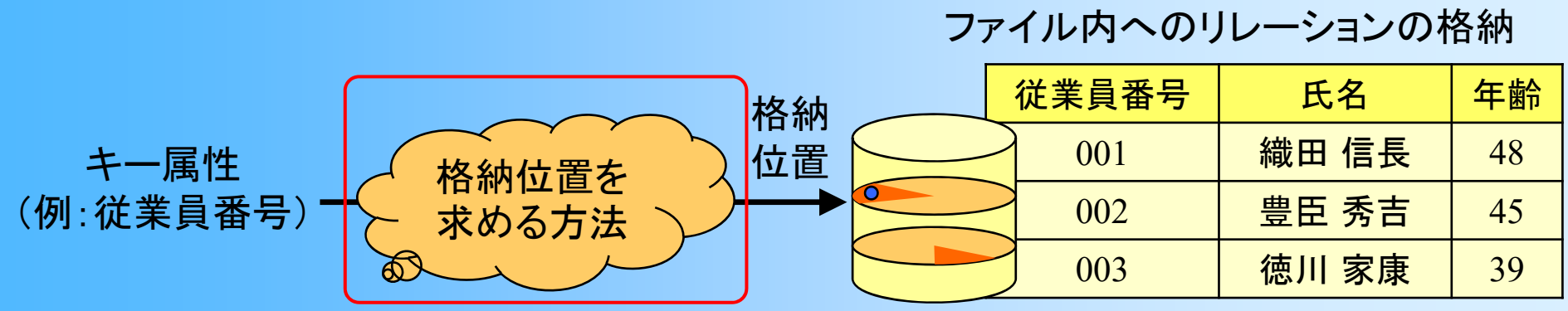
データの格納方法

- ファイル内のデータの格納方法の種類
  - 順序なしの格納
  - 順序付きの格納
  - ハッシュを使った格納



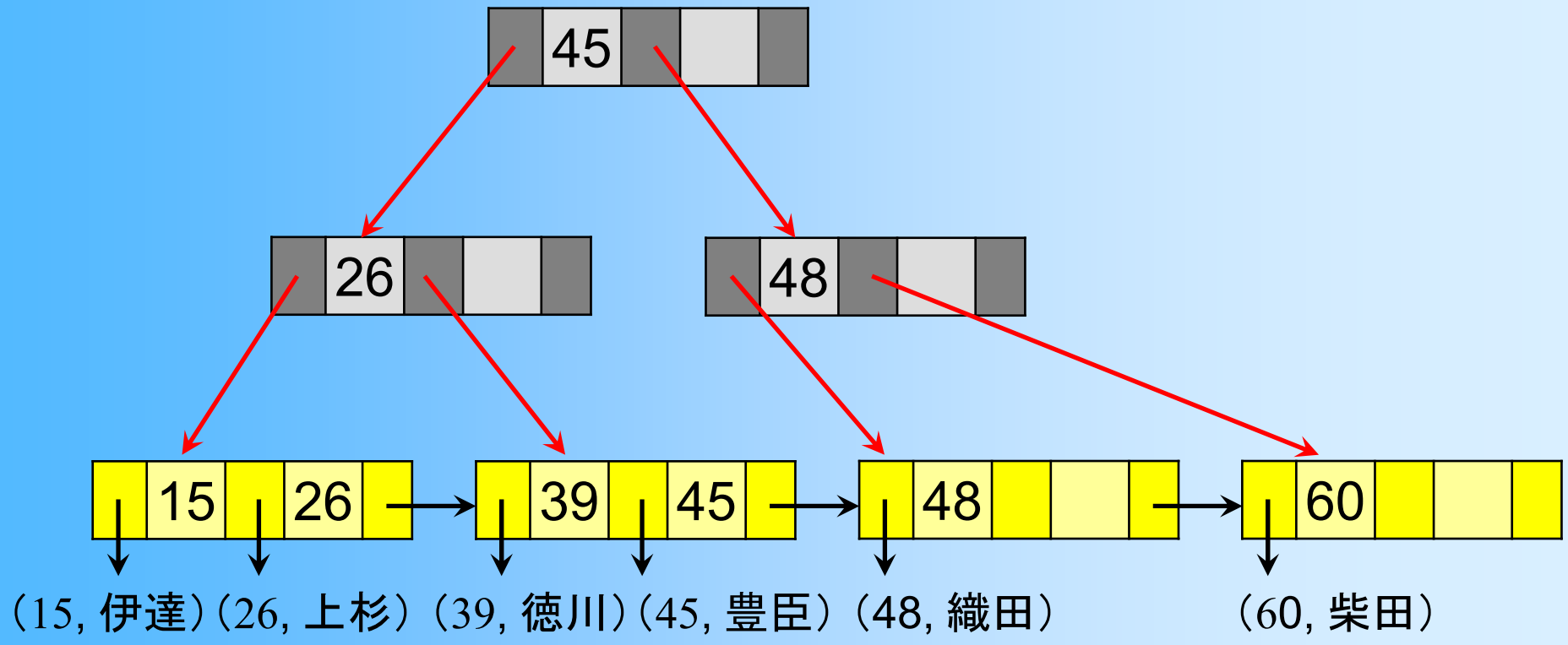
アクセス・メソッド

- データの格納位置の決定方法の種類
  - スキャン法
  - 探索法
  - インデックス法
  - ハッシュ法



B^+-ツリーの構成

- B^+-ツリーの例



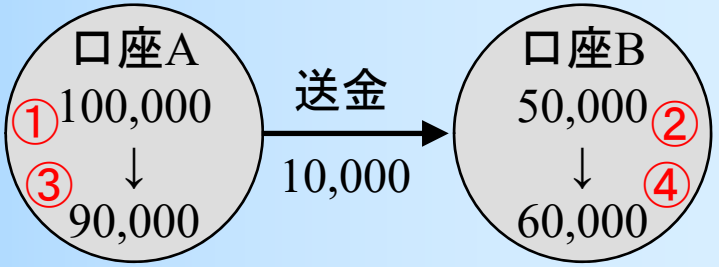
同時実行制御

同時実行制御

- トランザクション
- トランザクションの同時実行制御
- 計画的な並行処理
  - 直列可能性
  - ビュー等価と競合等価
  - 先行グラフによる競合等価の判定
- 逐次的な並行処理
  - ロックとデッドロック
  - デッドロックの回避方法

トランザクション

トランザクション



- トランザクション
  - １つの処理の単位
  - 例：銀行口座の送金（口座Aから口座Bへ1万円送金）
    1. 口座Aの現在の預金額を読み込む
    2. 口座Bの現在の預金額を読み込む
    3. 口座Aの預金額から1万円引いた値を口座Aに記録
    4. 口座Bの預金額に1万円加えた値を口座Bに記録
    - 全ての処理をまとめて、一つのトランザクションとする
    - 処理の途中で中断してしまったり、処理の途中で他の処理が口座A,Bの預金額を変更すると、整合性に問題が生じてしまうため

トランザクションの性質(1)

- 原子性
  - トランザクションは1つの処理の単位
  - 途中で中断することは許されない
  - 次のどちらかの結果のみを許す
    - トランザクションは完全に成功する(コミット)
    - トランザクションが中断した時は、途中の操作は全てキャンセルされて、開始前の状態に戻る(アボート)
- 一貫性・整合性
  - トランザクションの結果によってデータベースの整合性が崩れることはない

トランザクションの性質(2)

- 独立性・隔離性
  - あるトランザクションは、他のトランザクションに影響を与えない
- 耐久性
  - 一度完了したトランザクションの結果は何らかの理由で消滅してはならない
  - もし間違えてトランザクションを実行した場合は、別のトランザクションを発行して修正する（補償トランザクション）

トランザクションの性質(3)

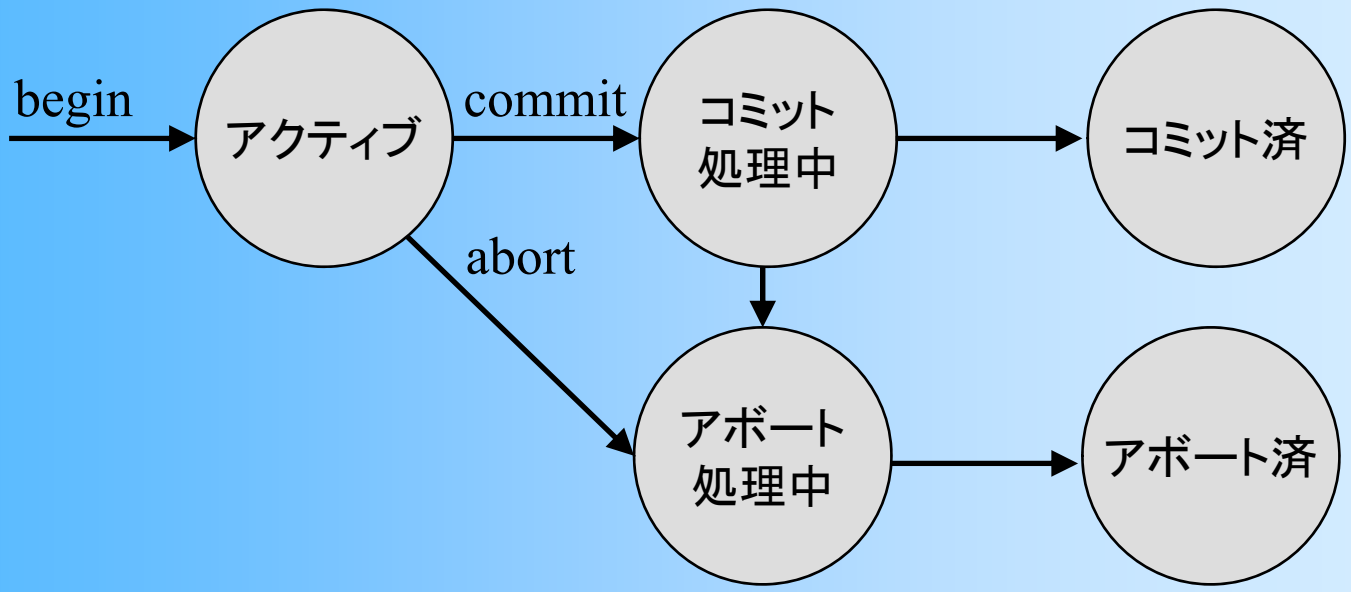
- トランザクションの4つの性質をまとめて、ACID性質と呼ぶ
  - 原子性(Atomicity)
  - 一貫性・整合性(Consistency)
  - 独立性・耐久性(Isolation)
  - 永続性(Durability)

トランザクションの操作

- トランザクションは、複数のデータベース直接操作のコマンドによって構成される
  - データの読み書きや条件分岐などの組み合わせ
  - アプリケーションからデータベースを直接操作する場合など
- データベースに対する操作のコマンド
  - 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完了（コミット）の要求
  - abort: トランザクションの中断（アボート）を要求

トランザクションの状態遷移

- アクティブ（実行中）
- コミット処理中 → コミット済
- アボート処理中 → アボート済



トランザクションの同時実行制御

同時実行制御

- トランザクション
- トランザクションの同時実行制御
- 計画的な並行処理
  - 直列可能性
  - ビュー等価と競合等価
  - 先行グラフによる競合等価の判定
- 逐次的な並行処理
  - ロックとデッドロック
  - デッドロックの回避方法

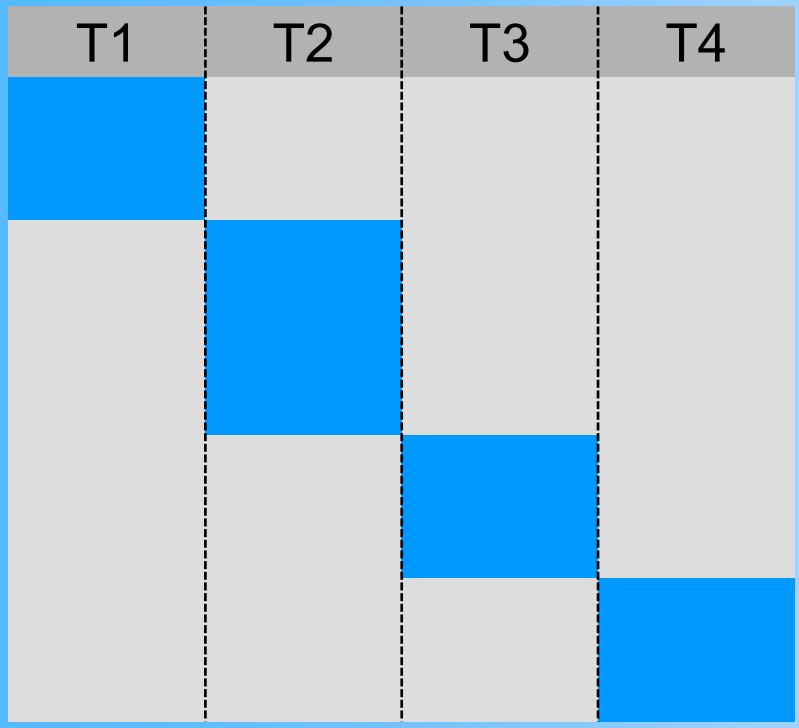
トランザクションの同時実行制御

- 実際のデータベースでは、複数のトランザクションを並列に実行する
  - トランザクションの途中でディスク読み書きなどが生じると、その入出力を待っている間、CPUの処理は停止する
  - トランザクションを並列に実行していれば、あるトランザクションが入出力待ちで停止している間も、別のトランザクションを実行できるので効率的
  - さらに、マルチCPUのコンピュータであれば、常に複数のトランザクションを同時に実行可能

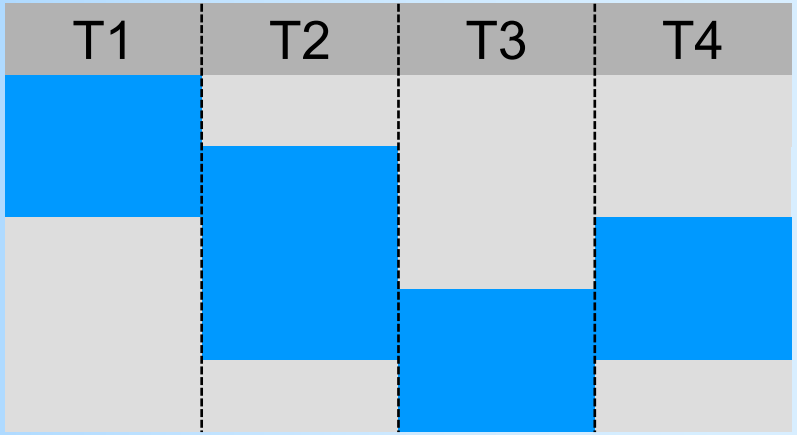
トランザクションの同時実行制御（続き）

- 結果が同じであれば、なるべく並列に実行した方が効率的

直列実行



並列実行（同時実行）



同時実行制御時の注意点

- あるトランザクションが、他のトランザクションに影響を与えないようにする必要がある
- 複数のトランザクションが同時に同じデータを読み書きしようとすると不具合が生じる

同時実行制御の方法

- 計画的な並行処理
  - トランザクションが並列に実行できるように、あらかじめ適切な順序やタイミングを決定して実行
    - 実行される一連のトランザクションが既知の場合
- 逐次的な並行処理
  - トランザクションが与えられた順番に処理を開始
  - 複数のトランザクションが競合しようとしても問題が起きないような仕組みを導入
    - 実際は、こちらの方法が一般に用いられる

計画的な並行処理

同時実行制御（説明の流れ）

- トランザクション
- トランザクションの同時実行制御
- 計画的な並行処理
  - 直列可能性
  - ビュー等価と競合等価
  - 先行グラフによる競合等価の判定
- 逐次的な並行処理
  - ロックとデッドロック
  - デッドロックの回避方法

同時実行制御の方法

- 計画的な並行処理
  - トランザクションが並列に実行できるように、あらかじめ適切な順序やタイミングを決定して実行
    - 実行される一連のトランザクションが既知の場合
- 逐次的な並行処理
  - トランザクションが与えられた順番に処理を開始
  - 複数のトランザクションが競合しようとしても問題が起きないような仕組みを導入
    - 実際は、こちらの方法が一般に用いられる

計画的な並行処理

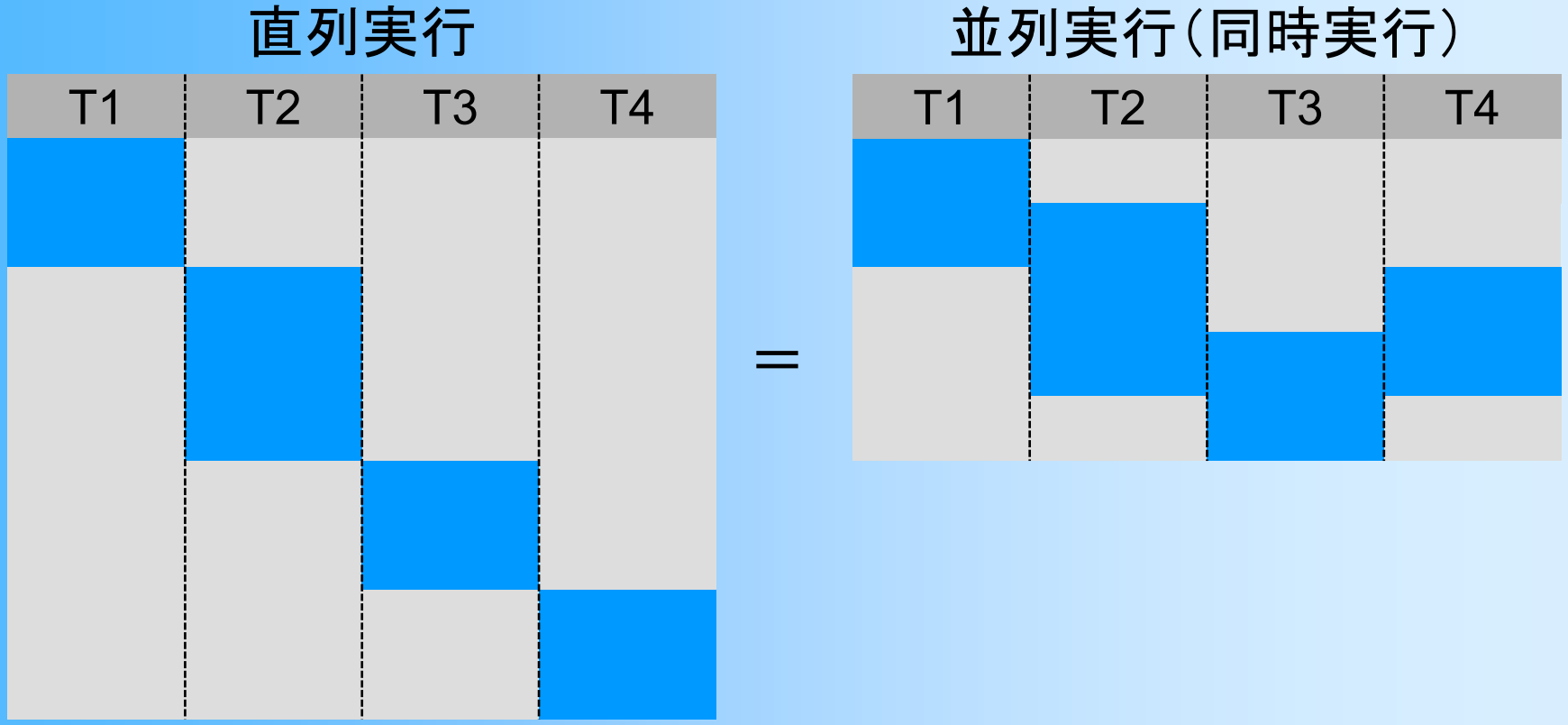
- 直列可能性
- ビュー等価
- 競合等価
- 先行グラフによる競合等価の判定
- ビュー等価と競合等価

直列可能性

- トランザクションの直列可能性
  1. トランザクションを何らかの順序で直列に実行
  2. トランザクション(の一部)を並列に実行
  - 両者の結果が等価になるとき、トランザクションは直列可能(直列化可能)であるという
    - たまたま並列に実行しているが、直列に実行したときと同じ結果になる、という意味で直列可能と言う

直列可能性（続き）

- 両者の実行結果が同じ（等価）であれば、この並列実行スケジュールは直列化可能



等価性の定義

- 等価性の定義が問題
  - どのような時にトランザクションの結果が等価であると言えるか
  - 並列処理のためには、トランザクションを実行する前に、トランザクションの内容をもとに、あるスケジュールが直列可能かどうかを判断したい
- 等価性の定義方法と判定方法
  - ビュー等価
  - 競合等価

等価の定義と判定方法

- ビュー等価
  - 各トランザクションが読み込む値が同じかどうかにより判定
  - 直列可能性の判定処理には時間がかかる
- 競合等価
  - 各トランザクションが競合するデータにアクセスする順番が同じかどうかにより判定
  - 等価なスケジュールも、等価でないと判定してしまうことがある（実用上は問題ない）
  - 直列可能性は比較的容易に判定できる

ビュー等価

- ビュー等価（view equivalent）
  - S_1S_2の2つのスケジュールがある
    - S_1・・・複数のトランザクションをある順序で直列に実行するスケジュール
    - S_2・・・同じトランザクションをある順序で非直列に実行
    - この2つが等価（ビュー等価）かどうかを調べたい
  - 2つのスケジュールがビュー等価である条件
    - 読み込み処理が、S_1S_2において、同じ書き込み処理の結果を読み込んでいる
    - 各データを最後に書き込んでいる処理が、S_1S_2で同一

ビュー等価の例

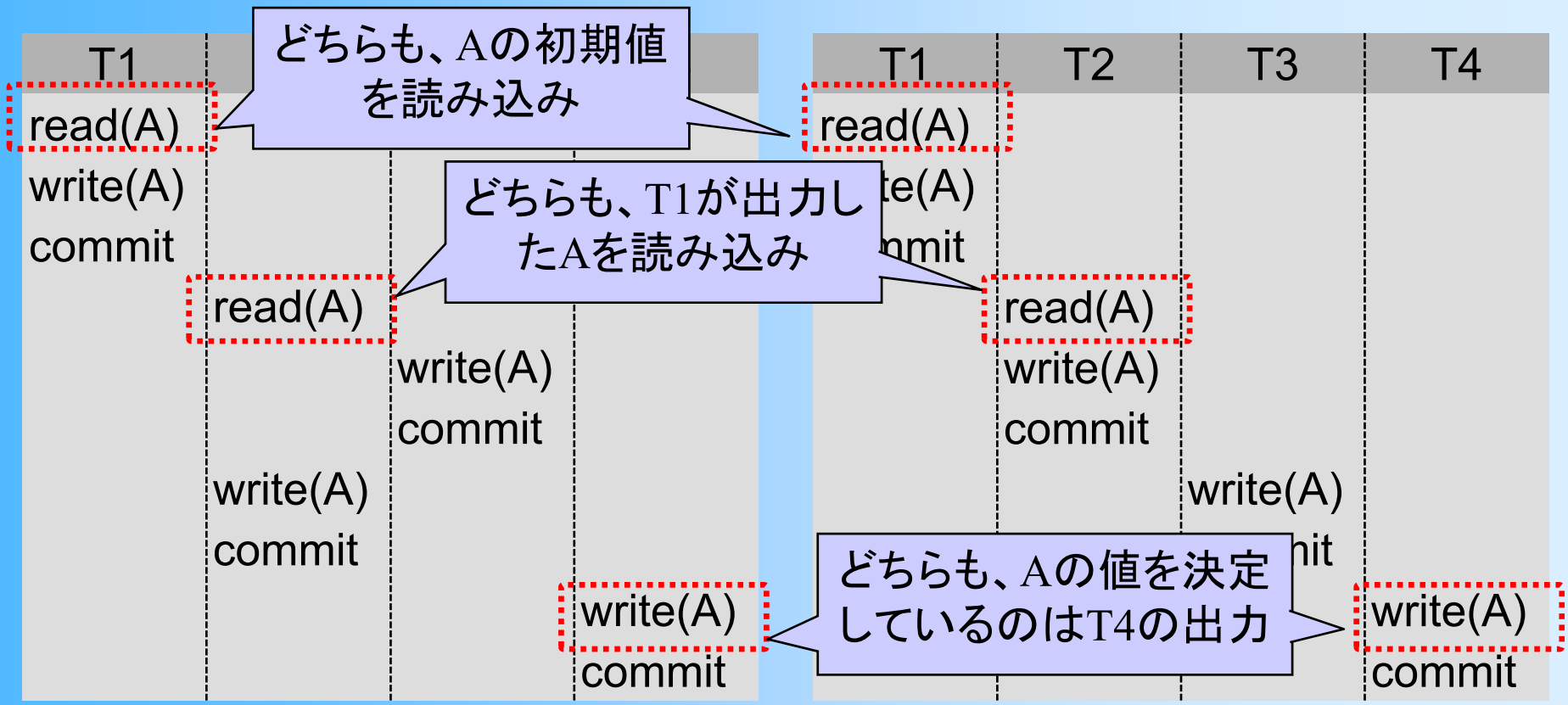
- ビュー（値を読み込む時点）で値が等価
  - 左のスケジュールは直列可能

T1	T2	T3	T4
read(A)			
write(A)			
commit			
	read(A)		
		write(A)	
		commit	
	write(A)		
	commit		
			write(A)
			commit

T1	T2	T3	T4
read(A)			
write(A)			
commit			
	read(A)		
	write(A)		
	commit		
		write(A)	
		commit	
			write(A)
			commit

ビュー等価の例

- ビュー（値を読み込む時点）で値が等価
  - 左のスケジュールは直列可能



ビュー等価の問題点

- ビュー等価による直接可能性を判定する問題はNP完全
  - 全ての直列実行の組み合わせを列挙して、ビュー等価である直列実行スケジュールが存在するかどうかを判定する必要がある
  - 非常に多くの処理がかかる
- より簡単に判定できる直列可能の判定方法が必要

等価の定義と判定方法

- ビュー等価
  - 各トランザクションが読み込む値が同じかどうかにより判定
  - 直列可能性の判定処理には時間がかかる
- 競合等価
  - 各トランザクションが競合するデータにアクセスする順番が同じかどうかにより判定
  - 等価なスケジュールも、等価でないと判定してしまうことがある（実用上は問題ない）
  - 直列可能性は比較的容易に判定できる

競合等価(相反等価)

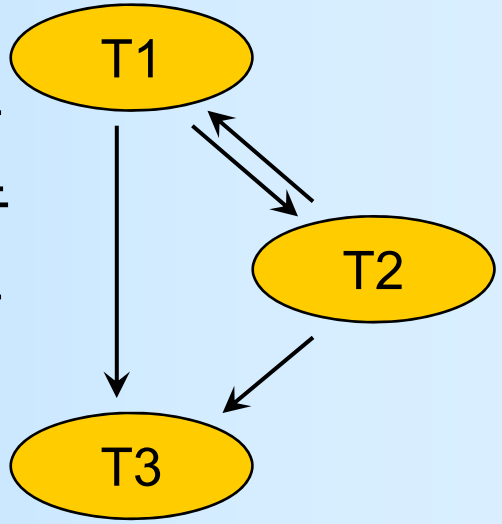
- 競合等価(相反等価、conflict equivalent)
  - 2つのスケジュールが競合等価である条件
    - 競合するデータに対する読み書きの順番が全て同じ
- 競合等価の特徴
  - 競合等価は多項式時間で判定可能
  - 競合等価であれば必ず直列可能
- 競合等価は先行グラフを作ることで判定できる

ビュー等価と競合等価

- ビュー等価
  - 各処理において同一のデータを参照することを条件
  - 正しく等価を判定できるが、計算に時間がかかる
- 競合等価
  - 競合するデータについて、各処理が同一の順番でアクセスすることを条件として範囲
  - 全ての等価は判定できない（等価なものを等価でないと判定する可能性がある）
  - 先行グラフを作成することで効率的に判定できる

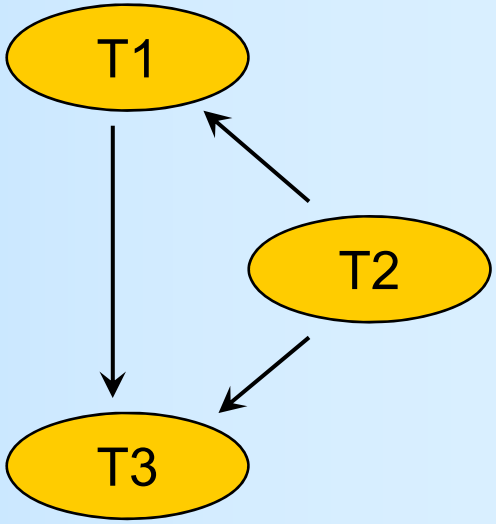
先行グラフ(相反グラフ)

- トランザクション同士の実行順序を有向グラフで表したもの
  - 各トランザクションをノードとする
  - あるデータXにアクセスするトランザクション$T_1$$T_2$について、次のいずれかのとき T_1 から T_2 に有向エッジを引く
    - T_1 のread(X) が T_2のwrite(X) に先行
    - T_1 のwrite(X) が T_2のwrite(X) に先行
    - T_1 のwrite(X) が T_2のread(X) に先行



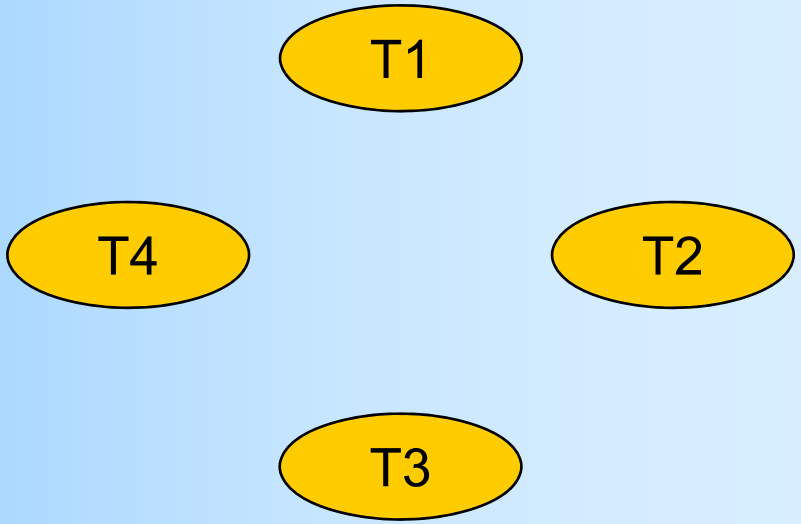
先行グラフと直列可能

- 先行グラフにループがなければ直列可能
  - あるノードから始めて、リンクを辿ってそのノードに戻ってくるようなルートが存在していれば、先行グラフにループがあるので直列可能ではない
  - ループがなければ、有向グラフにもとづく順序で直列に実行することが可能

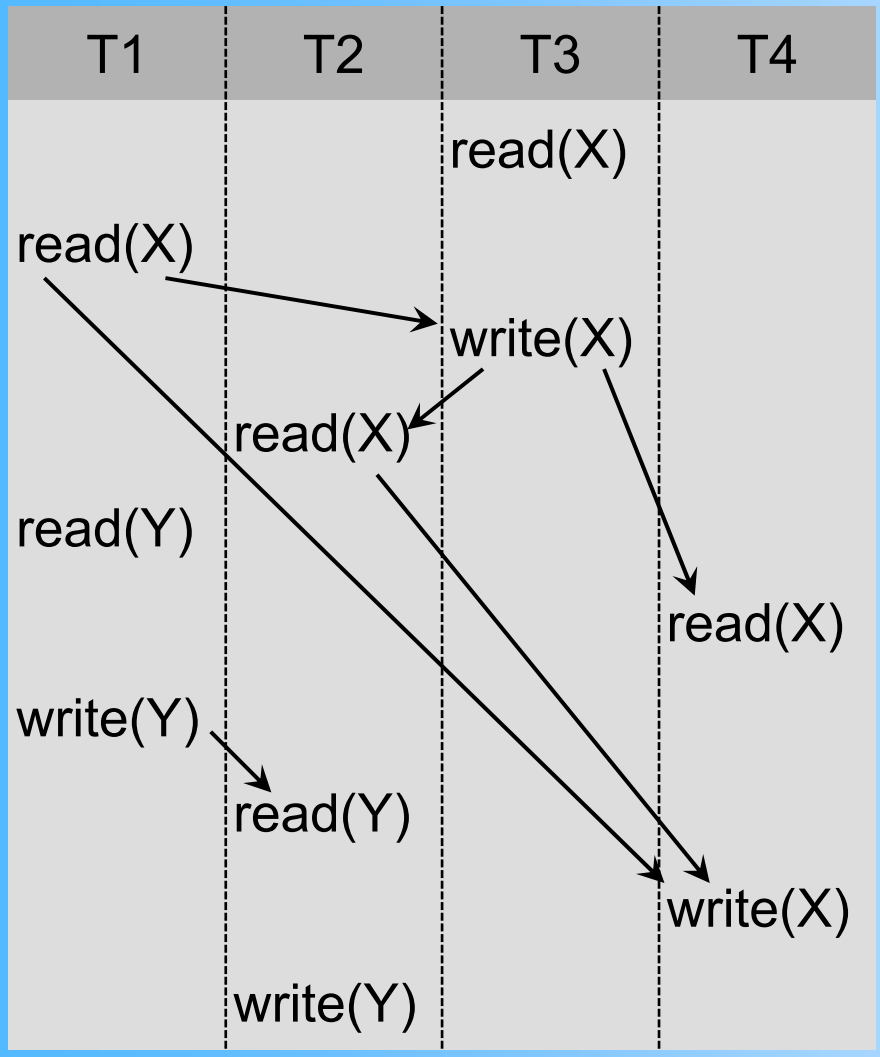


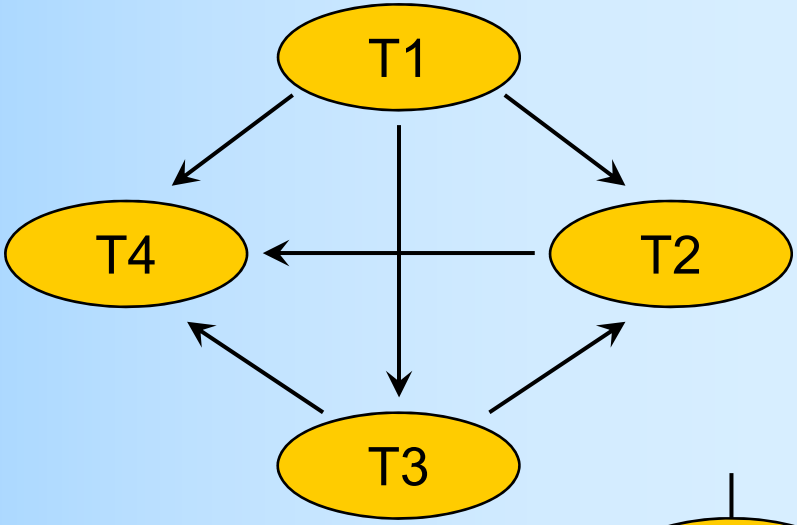
先行グラフの作成例

T1	T2	T3	T4
		read(X)	
read(X)			
		write(X)	
	read(X)		
read(Y)			
			read(X)
write(Y)			
	read(Y)		
			write(X)
	write(Y)		

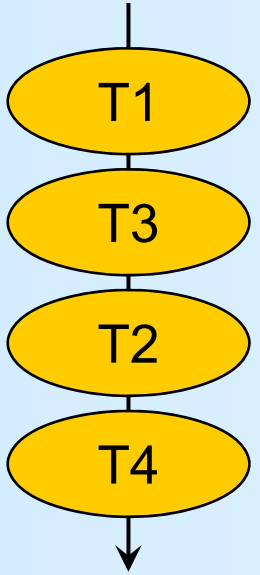


先行グラフの作成例





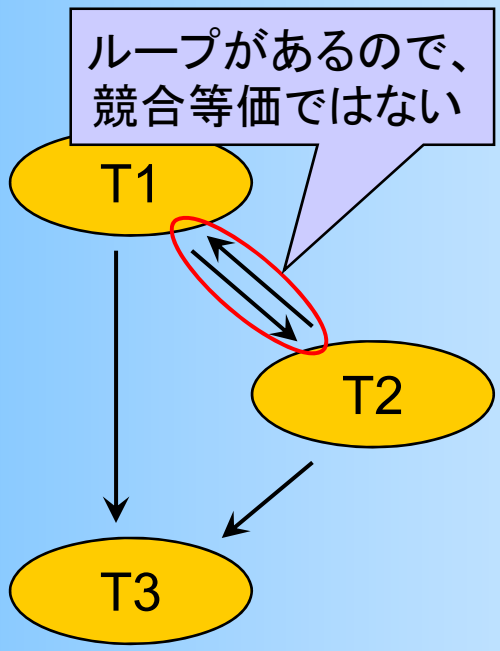
巡回はないので直列可能



ビュー等価と競合等価の関係

- 競合等価であれば、必ずビュー等価となる
  - 逆は必ずしも正しいわけではない
  - 以下は、ビュー等価であるが、競合等価でない例

T1	T2	T3
read(A)		
	write(A)	
	commit	
write(A)		
commit		
		write(A)
		commit



T1	T2	T3
read(A)		
write(A)		
commit		
	write(A)	
	commit	
		write(A)
		commit

等価の定義と判定方法（まとめ）

- ビュー等価
  - 各トランザクションが読み込む値が同じかどうかにより判定
  - 直列可能性の判定処理には時間がかかる
- 競合等価
  - 各トランザクションが競合するデータにアクセスする順番が同じかどうかにより判定
  - 等価なスケジュールも、等価でないと判定してしまうことがある（実用上は問題ない）
  - 直列可能性は比較的容易に判定できる

逐次的な並行処理

同時実行制御（説明の流れ）

- トランザクション
- トランザクションの同時実行制御
- 計画的な並行処理
  - 直列可能性
  - ビュー等価と競合等価
  - 先行グラフによる競合等価の判定
- 逐次的な並行処理
  - ロックとデッドロック
  - デッドロックの回避方法

同時実行制御の方法

- 計画的な並行処理
  - トランザクションが並列に実行できるように、あらかじめ適切な順序やタイミングを決定して実行
    - 実行される一連のトランザクションが既知の場合
- 逐次的な並行処理
  - トランザクションが与えられた順番に処理を開始
  - 複数のトランザクションが競合しようとしても問題が起きないような仕組みを導入
    - 実際は、こちらの方法が一般に用いられる

逐次的な並行処理

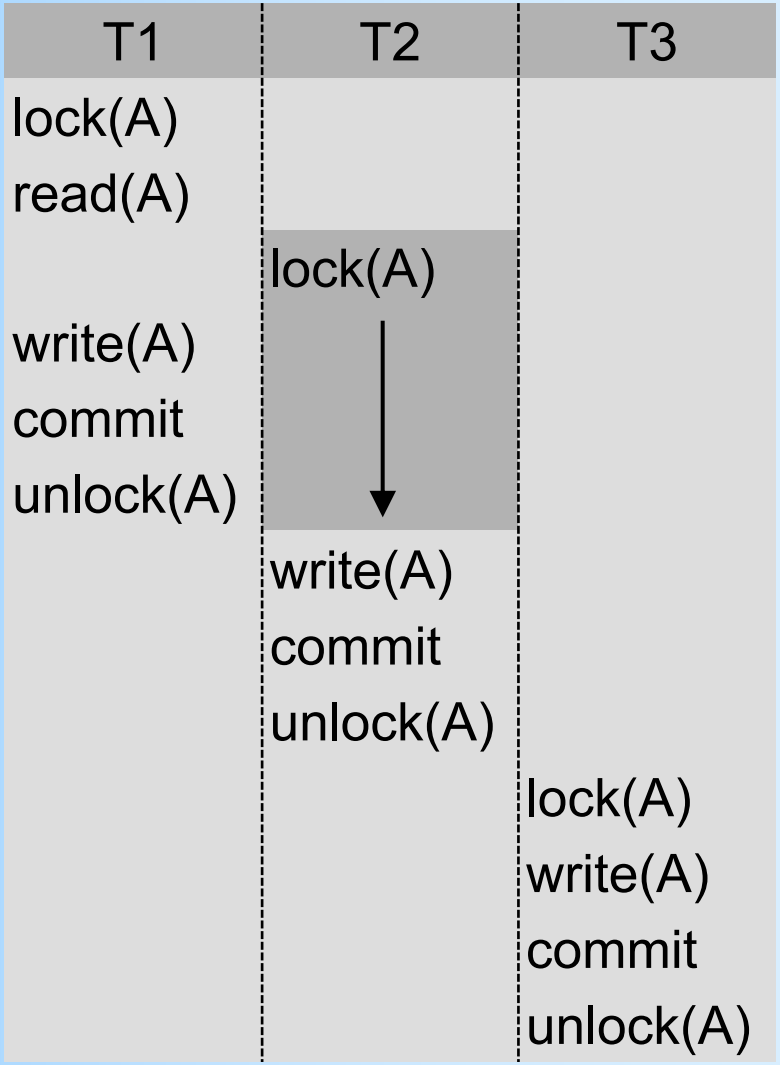
- ロック
- デッドロック
- デッドロックの回避方法

ロックを用いた同時実行制御

- ロック（Lock）
  - あるトランザクションだけが一部のデータを読み書きできるようにする仕組み
- ロックを使った処理の流れ
  - トランザクションがあるデータを読み書きする前に、そのデータに対して必ずロックを宣言
  - 他のトランザクションがそのデータを読み書きしようとした時には、後のトランザクションは、最初のトランザクションが完了するまで待たされる

ロックを使った処理の例

- ロックしようとしたデータがすでにロックされていたら、開放されるまで待つ

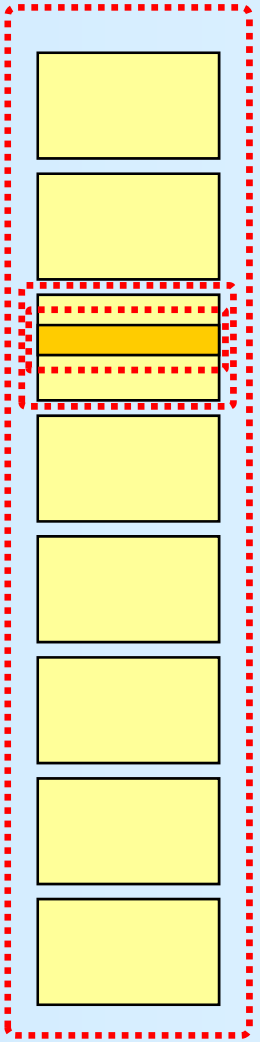


ロックの種類

- 共有ロック（読み出し専用ロック）
  - 複数のトランザクションが同時にロックできる
- 専有ロック（読み書き用ロック）
  - １つのトランザクションしかロックできない
  - 共有ロックとの共存も許さない
- ロックのアップグレード
  - 最初は共有ロックしておいて、書き込みの必要ができた時に、専有ロックに変更することも可能

ロックの粒度

- リレーション単位でのロック
  - ロックの管理は簡単
  - 実際は競合していないのに競合してしまう
- タプルや属性単位でのロック
  - ロックの管理が面倒、データ量が増える
  - 最小限の競合ですむ
- ページ単位でのロック
  - ディスク読み込みのページ単位でロック
  - 両者の中間

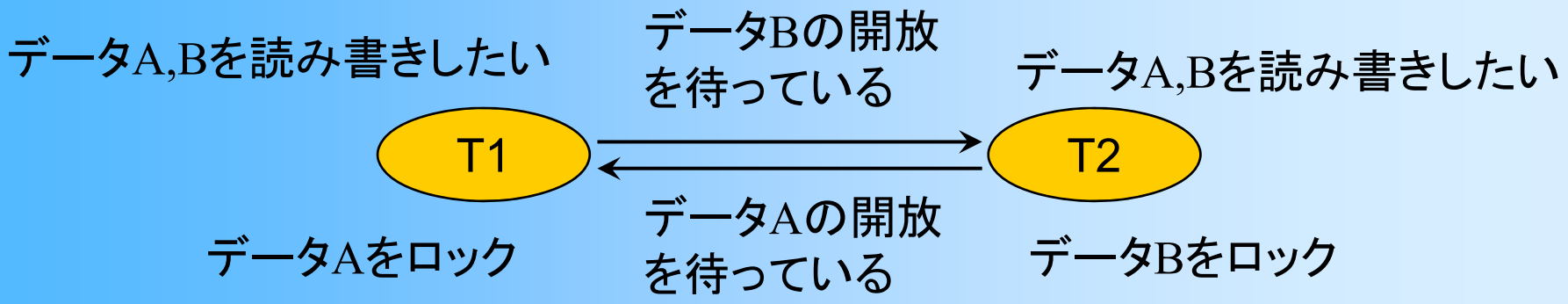


二相ロック

- 一度データをロックしたら、そのデータの処理が終わっても、トランザクションが完了（コミット）するまで開放しないというやり方
  - アボート処理の適用が容易になる
  - あるトランザクションで変更したデータが、そのトランザクションが終わる前に、別のトランザクションに読み書きされることがない
    - トランザクションがアボートされた時には、単純にロック時の値に戻すだけで良い

デッドロック

- デッドロック
  - 複数のトランザクションが互いに必要なリソースをロックし合っており、互いに処理を遂行できない状態
  - 基本的にはどちらか一方のトランザクションを一度アボートする必要がある



デッドロックへの対処方法

- いくつかの対処方法が考えられる
- デッドロックの検知
  - 何らかの方法で、デッドロックを起こしているトランザクションを特定し、終了させる
- デッドロックの防止
  - 何らかの方法で、デッドロックが発生しないよう防止する
- デッドロックの回避
  - 何らかの方法で、デッドロックが発生しても停止しないような処理を行う
  - この方法が一般的

デッドロックへの対処方法

- デッドロックの検知
  - 何らかの方法で、デッドロックを起こしているトランザクションを特定し、終了させる
- デッドロックの防止
  - 何らかの方法で、デッドロックが発生しないよう防止する
- デッドロックの回避
  - 何らかの方法で、デッドロックが発生しても停止しないような処理を行う

デッドロックの検知方法

- 待ちグラフを作成する方法
  - 各トランザクションがどのトランザクションの完了を待っているかどうかの有向グラフを作成
  - グラフにループがあれば、デッドロックが発生している
  - 待ちグラフを作成するタイミングが重要
- タイムアウトを行う方法
  - 一定時間トランザクションが進行しなかったら、デッドロックが発生している可能性があるとする
  - 正確な検出は行わない
  - タイムアウト時間の設定が重要
    - 設定によっては、いつまで待っても終了しない危険性がある

デッドロックへの対処方法

- デッドロックの検知
  - 何らかの方法で、デッドロックを起こしているトランザクションを特定し、終了させる
- デッドロックの防止
  - 何らかの方法で、デッドロックが発生しないよう防止する
- デッドロックの回避
  - 何らかの方法で、デッドロックが発生しても停止しないような処理を行う

デッドロックの防止方法

- 何らかの方法でデッドロックが生じないようにする
  - 例えば、トランザクションの最初に必要なデータを全部ロックする
    - 一度ロックが成功したら、途中でロックが必要になることはないので、必ずトランザクションは成功する
    - データが全部ロックできるまで待つ必要がある
    - 結果的に、並列処理がほとんどできなくなってしまう可能性がある

デッドロックへの対処方法

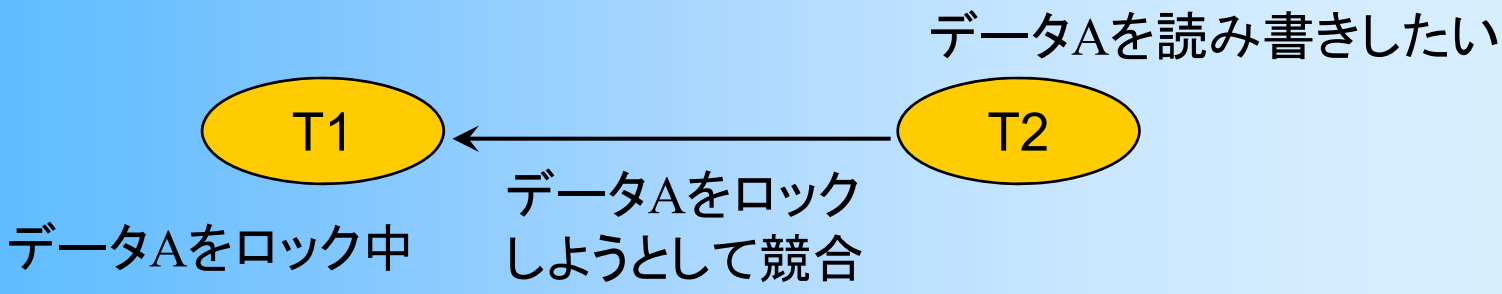
- デッドロックの検知
  - 何らかの方法で、デッドロックを起こしているトランザクションを特定し、終了させる
- デッドロックの防止
  - 何らかの方法で、デッドロックが発生しないよう防止する
- デッドロックの回避
  - 何らかの方法で、デッドロックが発生しても停止しないような処理を行う

デッドロックの回避方法

- デッドロックが生じないように、どちらかのトランザクションをアボートする
  - 同一のデータを複数のトランザクションがロックしようとした時、何らのルールに従って処理
  - トランザクションの実行時間にもとづいた方法が一般的
    - ウェイト・ダイ（wait-die）方式
    - ワウンド・ウェイト（wound-wait）方式
  - どちらの方法でも実行時間（実行開始からの経過時間）が長い方が優先されるので、アボートされたトランザクションも必ずいつかは実行される

デッドロックの回避方法

- ウェイト・ダイ（wait-die）方式
  - 最初にロックしていた方が古ければ、後からロックした方をアボート
- ワウンド・ウェイト（wound-wait）方式
  - 後からロックしようとした方が古ければ、最初にロックしていた方をアボート



ウェイト・ダイ方式

- ウェイト・ダイ（wait-die）方式
  - 最初にロックしていた方が古ければ、後からロックした方をアボート
    - ロックする際、待つ（wait）かアボートする（die）
- 判定方法
  - TS(T_i)をトランザクションの開始時刻とする
  - すでにT_2がロックしているデータを、T_1がロックしようとしたとき、
    - TS(T_1) ＜ TS(T_2)なら、T_1はT_2を待つ
    - TS(T_1) ＞ TS(T_2)なら、T_1をアボート

ワウンド・ウェイト方式

- ワウンド・ウェイト（wound-wait）方式
  - 後からロックしようとした方が古ければ、最初にロックした方をアボート
    - ロックする際、アボートさせる（wound）か待つ（wait）
- 判定方法
  - TS(T_i)をトランザクションの開始時刻とする
  - すでにT_2がロックしているデータを、T_1がロックしようとしたとき、
    - TS(T_1) ＜ TS(T_2)なら、T_2をアボート
    - TS(T_1) ＞ TS(T_2)なら、T_1はT_2を待つ

例題

時刻	T1	T2	T3
t0			read(Y)
t1	read(X)		
t2			write(Y)
t3	read(Y)		
t4		read(Y)	
t5			read(X)
t6			write(X)
t7			commit
t8	write(Y)		
t9	commit		
t10		write(Y)	
t11		commit	

- 各トランザクションの開始時刻は、最初に read または write の処理を実行するときとする
- トランザクションがアボートされたら次の時刻に再開するものとする

例題

時刻	T1	T2	T3
t0			read(Y)
t1	read(X)		
t2			write(Y)
t3	read(Y)		
t4		read(Y)	
t5			read(X)
t6			write(X)
t7			commit
t8	write(Y)		
t9	commit		
t10		write(Y)	
t11		commit	

- ワウンド･ウェイト方式での結果
- ウェイト･ダイ方式での結果

例題：ワウンド・ウェイト方式

時刻	T1	T2	T3	ワウンド・ウェイト方式
t0			read(Y)	T3がYをロック
t1	read(X)			T1がXをロック
t2			write(Y)	
t3	read(Y)			T1がYをロックしようとして停止（T1＞T3）
t4		read(Y)		T2がYをロックしようとして停止（T2＞T3）
t5			read(X)	T3がXをロックしようとしてT1をアボート
t6			write(X)	（T3＜T1）
t7			commit	T3コミット
t8	write(Y)			T1がYをロックしようとしてT2をアボート
t9	commit			（T1＜T2）
t10		write(Y)		
t11		commit		T1コミット → T2コミット

T1 よりも T3 の方が
開始時間が早いため

例題：ウェイト・ダイ方式

時刻	T1	T2	T3	ウェイト・ダイ方式
t0			read(Y)	T3がYをロック
t1	read(X)			T1がXをロック
t2			write(Y)	
t3	read(Y)			T1がYをロックしようとしアボート（T1＞T3）
t4		read(Y)		T2がYをロックしようとしアボート（T2＞T3）
t5			read(X)	T3がXをロックしようとし停止（T3＜T1）
t6			write(X)	（再開したT1が先にXをロックするため）
t7			commit	T2はYをロックしようとしてアボート繰返し
t8	write(Y)			T1がYをロックしようとしアボート（T1＞T3）
t9	commit			T3がYをロックでき、T3コミット
t10		write(Y)		T1がYをロックしようとして停止（T1＜T2）
t11		commit		T2コミット → T1コミット

同時実行制御のまとめ

- トランザクション
- トランザクションの同時実行制御
- 計画的な並行処理
  - 直列可能性
  - ビュー等価と競合等価
  - 先行グラフによる競合等価の判定
- 逐次的な並行処理
  - ロックとデッドロック
  - デッドロックの回避方法

次回予告

- 問い合わせ処理
  - 問い合わせ処理の最適化
  - 基本データ操作の実行方法
- 障害回復
- 授業のまとめ