データベースシステム（8）

スケジューリング
2相ロック
デッドロック管理

トランザクション（Transaction）

データベースから読み出したデータにもとづいて
ユーザのプログラムで様々な計算が実行される

DBMSから見ると，ユーザのプログラムは
（そのプログラムで必要となる）
一連のデータベース操作
（データベース内の，どのテーブルの，どのタップルに対して，
参照，挿入，削除，更新のうち何をするか）
の系列でしかない

SQL文で言えば
「select」「insert」「delete」「update」

1つのユーザプログラムが行う，
意味のある「一連のデータベース操作の系列」を
トランザクションと呼び，それを**1かたまりの仕事**とみなす

SQLにおけるトランザクション

注
トランザクションの開始部分には多くの方言がある

BEGIN TRANSACTION
.
. SELECT，
. UPDATE，INSERT，DELETE文
.

COMMITまたはROLLBACK

BEGIN．．．からCOMMIT/ROLLBACKまでが
ひとまとまりのトランザクションとなる．

トランザクションとしてまとめられた一連のSQL文は
すべて実行されるか，まったく実行されないかのどちらか
（一部だけが実行されることはない）

BEGIN．．．からCOMMIT/ROLLBACKまでが
ひとまとまりのトランザクションとなる．

トランザクションとしてまとめられた一連のSQL文は
すべて実行されるか，まったく実行されないかのどちらか
（一部だけが実行されることはない）

データベースおよびそのなかのデータを
安全に，かつ，効率的に管理・運用させるために
トランザクションが維持しなければならない4つの特性がある

ACID特性（Atomicity, Consistency, Isolation, Durability）
原始性　一貫性　隔離性　耐久性

DBMSが（トランザクションの）これらの4特性を維持している

同時実行制御

複数のユーザプログラムを同時実行できることが
DBMSのパフォーマンス向上に欠かせない

複数のプログラムで同時に同じデータに対する操作を
実行すると，（DBMSを使わなければ）矛盾や
不一致がおこりうる．（預金残高の例）

DBMSは，同時実行をしても，単一で（順番に）実行
された場合と同じ結果になることを保証

ユーザは自分の実行しているトランザクションしか見えない．
DBMSはそのようなユーザが
あたかも，DBMSを自分ひとりで実行しているように見える
ようにトランザクションを実行してゆくことを保証

トランザクションの隔離性（Isolation）

DBMSは
（多数の）ユーザの実行する
（多様な）データベース操作プログラムを
同時実行できなければならない

DBMSは，通常，多数のトランザクションを同時に処理している

DBMSは，多くの同時実行されているトランザクションの
ひとつひとつが単独で実行された場合と同じように振舞うよう制御
（これをトランザクションのIsolation性とよぶ）

ユーザは自分の実行しているトランザクションしか見えない
（あたかも，DBMSを自分ひとりで実行しているように見える）

トランザクションの一貫性 (Consistency)

複数のプログラムで同時に同じデータに対する操作を
実行すると，(DBMSを使わなければ) 矛盾や
不一致がおこりうる．(預金残高の例)

DBMSは，同時実行をしても，単一で (順番に) 実行
された場合と同じ結果になることを保証

この特性をトランザクションのConsistencyと呼ぶ．

トランザクションの一貫性と隔離性を保証するためにDBMSは
，競合を防止・解消する同時実行スケジュール
を行っている

直列スケジュール (Serial Schedule)

各トランザクションの実行中はそれが終わるまで，
他のトランザクションのDB操作を行わないスケジュール
(同時実行 (インターリーブ) なしのスケジュール)

トランザクションがn個ある場合，
直列スケジュールはnの順列個 ${}_nP_n$ ＝ n!個

トランザクションがT1，T2の2つの場合
T1，T2
T2，T1

トランザクションがT1，T2の3つの場合
T1，T2，T3
T1，T3，T2
T2，T1，T3
T2，T3，T1
T3，T1，T2
T3，T2，T1

直列化可能スケジュール (Serializable Schedule)

全トランザクションが終了した時点の状態が，
いずれかの直列スケジュールの結果と等価になるスケジュール

$R_1(A)$, $W_1(A)$,
$R_1(B)$, $W_1(B)$
$R_2(A)$, $W_2(A)$
$R_2(B)$, $W_2(B)$

このスケジュールは以下のように直列化できる

$R_1(A)$, $W_1(A)$,
$R_1(B)$, $W_1(B)$
$R_2(A)$, $W_2(A)$
$R_2(B)$, $W_2(B)$

実行結果を変えない移動だけで直列スケジュールとなる

データの一貫性・各トランザクションの隔離性
を保証できないスケジュールのパターン

(1) 汚読状態 ↕ より罪が大きい
(2) 反復不可能読み状態
(3) 幽霊状態 ↕ 比較的罪は軽い

DBMSは
スケジュールされた全トランザクションの
実行後のデータの一貫性と
各トランザクションへの隔離性を保証しなくてならない．
これらの3つの違反状態を起こさないように，かつ，
できるだけ早く実行する．

2相ロック (Two Phase Lock)

データの一貫性とトランザクションのIsolation性を
保ちつつ同時実行を実現する仕組み

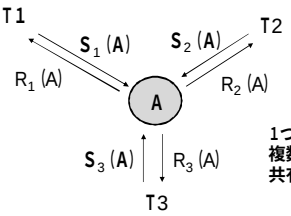
2段階のロックを利用する

共有ロック (Shared Lock)	Sロック
排他ロック (Exclusive Lock)	Xロック

各トランザクションは
・データを読む (SELECT) ときは (読む前に) そのデータを
「共有ロック (Shared Lock)」する
・データを書く (INSERT,UPDATE,DELETE) ときは
(書く前に) そのデータを 「排他ロック (Exclusive Lock)」する

共有ロック (Shared Lock)

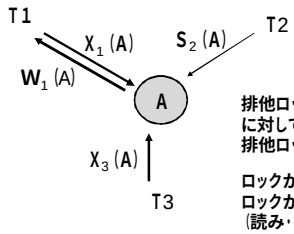
各トランザクションは
データを読む (SELECT) とき
(読む前に) そのデータを「共有ロック」する



1つのデータに対して
複数のトランザクションが
共有ロックをかけることができる

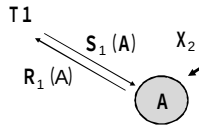
排他ロック (Exclusive Lock)

各トランザクションは
データを書く (INSERT, DELETE, UPDATE**する**) **とき**
(**書く前に**) **そのデータを**「**排他ロック**」**する**

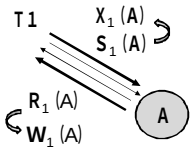


排他ロックがかかっているデータ
に対しては，他のトランザクションは
排他ロックも共有ロックもかけられない．

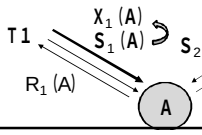
ロックがかけられない場合は
ロックがかけられるようになるまで
(**読み・書き**) **の操作を待つ．**



共有ロックがかかっているデータ
に対しては，他のトランザクションは
排他ロックをかけられない．



ただし自分が共有ロックしたデータに対しては
排他ロックをかけることができる．
(**共有ロックから排他ロックに変更できる**)



ただし自分が共有ロックした
データでも，他が共有ロックしていた
場合は排他ロックをかけられない．
(**共有ロックから排他ロックに**
変更できない)

2相ロック (Two Phase Lock)

各トランザクションは
・データを読む (SELECT) **ときは** (**読む前に**) **そのデータを**
「共有ロック (Shared Lock)」**する**
・データを書く (INSERT,UPDATE,DELETE) **ときは**
(**書く前に**) **そのデータを** 「**排他ロック** (Exclusive Lock)」**する**

各トランザクションは自分のかけたロックすべてを
トランザクション終了 (**コミットまたはロールバック**) **時に開放**
(**それまでは保持しておく**)

2相ロックをつかったスケジューリングの例

データの一貫性を保障できなかったスケジュールの例

$X_1(A)$**ロック** $X_1(B)$**ロック** $X_1(A)$, $X_1(B)$**開放**

T1:	$R_1(A), W_1(A),$	$R_1(B), W_1(B), R$
T2:	$R_2(A), W_2(A), C$	

他人の$X_1(A)$**が開放されるまで**
実行できない (**待たされる**)

T2**は**X(A)**を必要とするので**T1**が終わるまでは実行できない**

ある時点で，あるデータへの書込みは
ひとつのトランザクションのみが許される

同じデータに対する読み操作は，複数のトランザクション
で同時に起こっても問題はない．

2相ロックに従う限りは，実行中のすべてのトランザクションは
(**コミットされる順の**) **直列化可能スケジュールとなっている**

2相ロックをつかったスケジューリングの例

この時点までは待たされる

$S_1(A)$**ロック** $X_1(B)$**ロック** $S_1(A)$, $X_1(B)$**開放**

T1:	$R_1(A),$	$R_1(B), W_1(B),$	C
T2:	$R_2(A),$	$R_2(B), W_2(B),$ C	

$S_2(A)$**ロック** $X_2(B)$**ロック** $S_1(A)$, $X_1(B)$**開放**

A**に対する共有ロック**S(A)**は**T1，T2**が同時に取得できる**

2相ロックに従う限りは，実行中のすべてのトランザクションは
(**コミットされる順の**) **直列化可能スケジュールとなっている**

つまり，この2つのトランザクションの残す状態は
T2**のすべての作業を単独で行った後に**
T1**のすべての作業を単独で行った状態と同じになる**

汚読状態

T1**の**$W_1(A)$**はロールバックされたのに，**
T2**はその**$W_1(A)$**の値を読み，その後コミット**

T1:	$R_1(A), W_1(A),$	$R_1(B), W_1(B), R$
T2:	$R_2(A), W_2(A), C$	

↑ **これは待たせる**

反復不可能読み状態・幽霊状態

T1**の最初の**$R_1(A)$**と2回目の**$R_1(A)$**では**
T1**は自分ではA**に対する変更を行っていないのに**
値が変わっている

T1:	$R_1(A),$	$R_1(A), W_1(A), C$
T2:	$R_2(A), W_2(A), C$	

↑ **これは待たせる**

汚読状態（広義の汚読状態）

T1の$W_1(A)$がT2により上書き（広義の汚読．「汚書」）

↓ これは待たせる

T1:	$R_1(A)$,	$W_1(A)$,	C
T2:	$R_2(A)$,	$W_2(A)$,	C

↑ これは待たせる

この例は要注意
（お互いがお互いのロックの開放を待っている）

互いにロックの開放を待つ状態を「デッドロック」と呼ぶ

このままでは，（お互いが）永遠に待ち続ける

デッドロック（Deadlock）

互いにロックの開放を待つ状態を「デッドロック」と呼ぶ

2相ロックのしくみで直列化可能スケジュールを実現できる．
しかし，このしくみはデッドロックを起こしうる．

DBMSは2相ロックでトランザクションのACID特性を守りつつ
「デッドロック対策」を行っている

デッドロック検知（Deadlock Detection）
デッドロックが起こってないかをチェックし，
検知したらそのデッドロック状態を解消する．

デッドロック防止（Deadlock Prevention）
デッドロック状態が発生しないように2相ロックにさらに
制約をもたせる．

デッドロック検知（Deadlock Detection）

待ちグラフ（waits-for graph）の作成
各ノードがトランザクション．
もしTiがTjのロック開放を待っているなら，
Ti から Tj への有向辺をつくる．

T1 は
T2の X(C)開放を待っている

T1:	W(A),	W(B),	W(C)
T2:	W(C),	W(D),	W(B)

T2 は
T1のX(B)開放を
待っている

サイクルを検出したらそれはデッドロック
（3個以上のトランザクション間でも起こる）

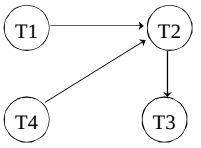
定期的に「待ちグラフ」のサイクルがないかをチェック

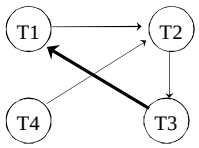
デッドロック検知（Deadlock Detection） つづき

待ちグラフにサイクルができたらそれはデッドロック状態．
サイクル内のいずれかのトランザクションを
Abort（強制的にRollback）する．

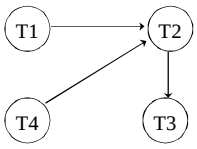
Example:

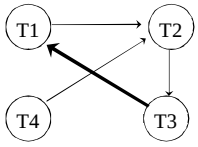
T1:	S(A),		S(B)	
T2:		X(B)		X(C)
T3:			S(C)	X(A)
T4:				X(B)





デッドロック検知（Deadlock Detection） つづき





定期的に「待ちグラフ」のサイクルがないかをチェック

各頂点から，深さ優先探索
・印をつけながら深い方向に進む
・浅い方向に戻るときには印を消す
・進む過程で印付の頂点にたどり着いたらサイクル

デッドロック防止（Deadlock Prevention）

各トランザクションにユニークなタイムスタンプを付ける

タイムスタンプを使ってトランザクションに優先順位をつける
（古いトランザクションが優先）

優先順位にもとづく2種類のデッドロック防止法

優先度が高い時　優先度が低い時

Wait-Die法
自分の優先度が高い時は待つが
低い時には，自らAbortして，
同じタイムスタンプで再実行

Wound-Wait法
自分の優先度が高い時は相手をAbortさせる．
自分の優先度が低い時は待つ．
Abortさせられたトランザクションは
同じタイムスタンプで再実行．

Wait-Die法

T1のタイムスタンプが「1」
自分が年上なので待つ

X(A)	X(B)	X(C)

T1: W(A),	W(B),	W(C)
T2: W(C),	W(D),	W(B)

X(C)	X(D)	X(B)

T2のタイムスタンプが「2」
自分が年下なので破棄

T2はそれまでの作業を元に戻し，
すべてのロックを開放する
（それによりT1は，作業を続けられるようになる）
T2は再びタイムスタンプ「2」で再実行

Wound-Wait法

T1のタイムスタンプが「1」
自分が年上なのでT2を破棄させる

X(A)	X(B)	X(C)

T1: W(A),	W(B),	W(C)
T2: W(C),	W(D),	W(B)

X(C)	X(D)

T2のタイムスタンプが「2」

T2はそれまでの作業を元に戻し，
すべてのロックを開放する
（それによりT1は，作業を続けられる）
T2は再びタイムスタンプ「2」で再実行

デッドロック防止法によりデッドロックが起こらないことの証明

- 同じタイムスタンプはない
- サイクルがある場合，必ず，そのサイクル内に
  （後輩⇒先輩）と
  （先輩⇒後輩）のいずれのリンクもある
- 「Wait-Die法」では，前者が存在しない
- 「Wound-Wait法」では，後者が存在しない
- よって，サイクル（デッドロック）は起こらない

すべての排他ロックをトランザクション終了時まで保持する
2相ロックでは，同時実行の効率は少し落ちてしまうが．．．

T1:	R(A), W(A),	R(B), W(B), R	
T2:		R(A), W(A),	

例えばT1のX(A)をW(A)の後ですぐに開放すると，
T2はあまり待たされずに実行できる．

T1がCなら問題ないがT1のW(A)が破棄されると，
T2はその破棄されたAに依存する作業を持つため，
以降の作業が異常となる

T1がR（Rollback）すると，T1の全操作を破棄するだけでなく
T2に対する全操作も破棄しなくてはならない（Rollbackの波及）

ロックを最後まで保持することにすればRollbackの波及は
なくなる
（ほとんどの商用DBMSはディフォルトでこのやりかたを採用）

SQLにおけるトランザクション

BEGIN TRANSACTION
．
．　SELECT文実行時には共有ロック
．　UPDATE，INSERT，DELETE文実行時には排他ロック

COMMITまたはROLLBACK

BEGIN．．．からCOMMIT/ROLLBACKまでが
ひとまとまりのトランザクションとなる

DBMSの隔離レベル（Isolation Level）

アプリケーションによっては，データの一貫性・隔離性よりも
応答速度が重要な場合もある．
トランザクションごとに隔離レベルを指定できる

隔離レベルはトランザクション開始後に
「SET TRANSACTION」文で指定する

SET TRANSACTION <レベル>

レベル
- SERIALIZABLE
- REPEATABLE READ
- READ COMMITED
- READ UNCOMMITED

↕ 高い（厳格）／低い（緩やか）

SET TRANSACTION <**レベル**>

レベル
- SERIALIZABLE
- REPEATABLE READ
- READ COMMITED
- READ UNCOMMITED

高い(**厳格**) **低速**

低い(**緩やか**) **高速**

	汚読状態	反復不可能読み状態	幽霊状態
SERIALIZABLE	起こらない	起こらない	起こらない
REPEATABLE READ	起こらない	起こらない	起こる
READ COMMITED	起こらない	起こる	起こる
READ UNCOMMITED	起こる	起こる	起こる

3つの違反 VS **実行効率**

(1) **汚読状態** (dirty read)

```
T1:   R(A), W(A),              R(B), W(B), R
T2:                 R(A), W(A), C
```

T1の$W_1(A)$**はロールバックされたのに,**
T2はその$W_1(A)$**の値を読み, その後コミット**

```
T1:   R(A),          W(A),          C
T2:          R(A),          W(A),     C
```

T1の$W_1(A)$**がT2により上書き** (**広義の汚読.「汚書」**)

(2) **反復不可能読み状態** (Unrepeatable Read)

```
T1:   R(A),                    R(A), W(A), C
T2:          R(A), W(A), C
```

T1の最初の$R_1(A)$**と2回目の**$R_1(A)$**では**
T1は自分ではAに対する変更を行っていないのに値が変わっている

(3) **幽霊状態** (Phantom)

3つの違反 VS **実行効率**

"ISOLATION LEVEL"**の選択肢**

SERIALIZABLE
= **厳格に2相ロックを適用**
= **3つの違反すべてを防ぐ**

REPEATABLE READ
= **共有ロックされているテーブルへの「Insert」は許す**
= (1) (2)**は防ぐ**

READ COMMITTED
=**共有ロックされているテーブルへの**
「Insert,Delete,Update」を許す
= (1)**は防ぐ**

READ UNCOMMITTED
= **ロックをしない**(file system**と同じ**)

(1) **汚読状態**
(2) **反復不可能読み状態**
(3) **幽霊状態**

ACID**特性** (Atomicity, Consistency, Isolation, Durability)

DBMS**はこの4つの特性を維持する**

原子性 (**Atomicity**)
トランザクション (**一連の操作系列**) **は**
すべて実行されるか, まったく実行されないかのどちらか
トランザクションの
一連のDB**操作** (**参照, 更新, 追加, 削除**) **のうち**
どこかで失敗したら, それまでの一連の操作を
(**ログの情報をもとに**) **全て元に戻せる.**

一貫性 (**Consistency**)
(**同時実行される複数の**) **トランザクションの実行後の状態も**
一貫性を保持している

ACID**特性** (Atomicity, Consistency, Isolation, Durability)

DBMS**はこの4つの特性を維持する**

隔離性 (**Isolation**)

DBMS**では, 通常, 複数のトランザクションが実行されているが**
トランザクションを実行しているユーザは
自分の実行しているトランザクションしか見えず,
あたかも, DBMS**を自分ひとりで実行しているように見える.**

耐久性 (**Durability**)

トランザクションの
一連のDB**操作** (**参照, 更新, 追加, 削除**) **が正しく実行され**
コミットされたら, その状態はDBMS**内に永続的に記録される**