データベースシステム（7）

トランザクション管理

データベース管理システム
（Database Management System）

——→ 「DBMS」と呼ぶ

一般的に，（巨大な）統合化されたデータ集合を扱う

データは

実世界（のビジネス）をモデル化したもの

実体　（「学生」，「科目」など）
関係　（「上戸彩」は「情報構造論」を履修）

DBMSはこうしたデータを保管し，管理するために
デザインされたソフトウエアパッケージ

なぜデータベースを学ぶのか？

データの多様性，サイズともに増加
そして情報を制するものが勝つ時代

デジタルライブラリ，
インタラクティブビデオ，
ヒトゲノム計画など
——→ DBMS による管理が必要

DBMSは巨大なデータを管理したり，
データを効率的に参照，変更，追加，削除するために
利用される

障害回復機能，同時実行制御，アクセス制御などで
有益な機能をもつ．（他にもデータ独立性，
データ一貫性の保障など重要な機能をもっている）

DBMSの仕組みを学ぶことで，情報の構造
情報管理，情報検索の真髄が理解できる

データを表形式で表現するデータモデル 「関係モデル」
関係モデルのデータを管理するためのソフトウエアが
「関係データベース」（問い合わせなどの機能をサポート）

データベース管理システム
（DBMS，Database Management System）

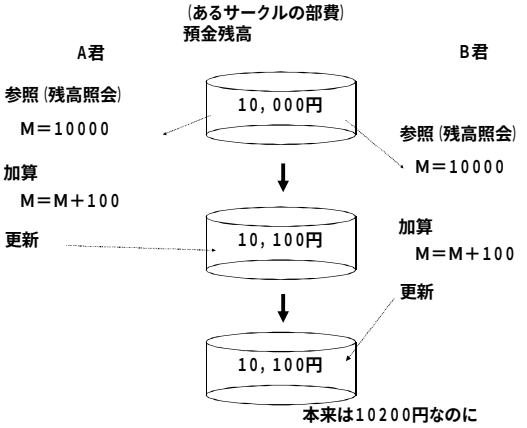
ファイル vs DBMS

（多数のユーザが利用する）データの
不一致や矛盾を防ぐ

「一貫性制約」
「同時実行制御」
「トランザクション管理」

クラッシュからの復元

大量データに対する高速アクセス，高速移動
（インデックス，問い合わせ最適化，バッファーリング，
ミラーリング，マテリアライズドビュー）



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

例：「銀行口座」のデータベース

Account

name	balance
広末	30000
倉木	50000
...	...

プログラム1
「倉木」の口座から
「広末」の口座に10000円振込

プログラム2
各口座の残高 (balance) に対し
6%の利息を振込

DBMSから見たプログラム1

```
UPDATE Account
  SET balance = balance-10000
  WHERE name = "倉木";
UPDATE Account
  SET balance = balance+10000
  WHERE name = "広末";
```

DBMSから見たプログラム2

```
UPDATE Account
  SET balance = balance*1.06
```

この四角で囲まれた
一連のDB操作がトランザクション

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
でないといけない．
(トランザクション内の操作の一部だけの実行ではいけない)

プログラム1
「倉木」の口座から
「広末」の口座に10000円振込

DBMSから見たプログラム1

```
UPDATE Account
  SET balance = balance-10000
  WHERE name = "倉木"
UPDATE Account
  SET balance = balance+10000
  WHERE name = "広末"
```

「倉木」の口座からの引落だけで終わってはならない
(たとえば残高不足等で)「倉木」の口座からの引落に失敗し，
「広末」の口座に振込むだけ実行もだめ

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

トランザクションの原子性 (Atomicity)

トランザクションは分割できない1かたまりの原子的DB操作

「すべて実行される」か，「まったく実行されない」か

```
Begin Transaction

UPDATE Account
  SET balance = balance-10000
  WHERE name = "倉木"
UPDATE Account
  SET balance = balance+10000
  WHERE name = "広末"

Commit
```

この特性を
トランザクションの「原始性 (Atomicity特性)」と呼ぶ

トランザクションの耐久性 (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
すべて順調に成功した場合は,
トランザクションの実行を完了 (「コミット (Commit)」命令を実行)
(コミットした状態がDBに記録され, その状態が確実に保存される.
これをトランザクションのDurability性とよぶ.)

確実な書き込み完了を保証するために
DBMSは特別な書き込みメカニズムを持つ

トランザクションの
一連のDB操作のうちどこかで失敗したら,
それまでの一連の操作を全てDBMSが自動的に破棄 (Abort)
(ユーザが明示的に取り消したい場合は
「ロールバック (Rollback)」命令を実行)

もとに戻すことを可能にするため,
DBMSはトランザクションの一連の操作を
ログ (log) に記録している

ログの例

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

	TransID	操作	TabID	TupID	ColID	OLD	NEW
...							
1054	T1	Begin					
1055	T1	Read	社員	007	給与		
1056	T2	Begin					
1057	T1	Write	社員	007	給与	50	100
1058	T2	Read	社員	002	給与		
1059	T2	Write	社員	002	給与	80	120
1060	T1	Commit					
...							

もしT2がRollbackされたらT2のログをもとに
T2が変更した値を元に戻す

ログのしくみ

ログにはDBMSの実行しているすべてのトランザクションの
操作記録 (READ,WRITE,ROLLBACK,COMMIT) が保存される

DBMSは実際にデータベースに対する操作を**実行する前**に
安定した記憶装置*注1にあるログに記録
(この仕組みをWAL(Write-Ahead Logging) Protocolと呼ぶ)
ログの記録に (システム障害などの理由で) 失敗すると
それに対応する操作はデータベースには残らない

注1 (企業などの) 大規模なDBMSでは, ログを記録するための (多重化され, 安定電源装置で動く) 専用のHWを装備している.

同一のマシン内にログをおく場合も, ログの記録には,
ファイルシステム内のファイルを使わず,
直接, ハードディスクを操作して記録する場合が多い.

(OSの (ファイルシステムなどの) 機能は安全なログの記録には不十分な面がある)

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

2つのトランザクション

```
T1:   BEGIN  A=A+100,   B=B-100  COMMIT
T2:   BEGIN  A=1.06*A,  B=1.06*B  COMMIT
```

この2つのトランザクションが同時に実行される場合も
この2つのトランザクションが
T1, T2の順で, あるいはT2, T1の順で
1つ1つを順に実行した結果と同じものとなる

```
A=300   A=400   A=400   A=424   A=424
B=500   B=500   B=400   B=400   B=424
```

T1, T2 →

```
T1:   A=A+100,  B=B-100
T2:                          A=1.06*A, B=1.06*B
```

```
A=300   A=318   A=318   A=418   A=418
B=500   B=500   B=530   B=530   B=430
```

T2, T1 →

```
T1:                          A=A+100, B=B-100
T2:   A=1.06*A, B=1.06*B
```

19

```
T1:   BEGIN  A=A+100,   B=B-100  COMMIT
T2:   BEGIN  A=1.06*A,  B=1.06*B  COMMIT
```

```
A=300   A=400   A=400   A=424   A=424
B=500   B=500   B=400   B=400   B=424
```

```
T1:   A=A+100,  B=B-100
T2:                          A=1.06*A, B=1.06*B
```

できれば, 同時実行 (インターリーブ) させて効率的に処理したい

正しい同時実行の「スケジュール」

```
T1:   A=A+100,   B=B-100
T2:              A=1.06*A,    B=1.06*B
```

```
A=300   A=400   A=424   A=424
B=500   B=500   B=400   B=424
```



```
T1:   BEGIN  A=A+100,   B=B-100  COMMIT
T2:   BEGIN  A=1.06*A,  B=1.06*B  COMMIT
```

```
A=300   A=400   A=400   A=424   A=424
B=500   B=500   B=400   B=400   B=424
```

```
T1:   A=A+100,  B=B-100
T2:                          A=1.06*A, B=1.06*B
```

正しくないスケジュール

```
T1:   A=A+100,                          B=B-100
T2:              A=1.06*A, B=1.06*B
```

```
A=300   A=400   A=424   A=424
B=500   B=500   B=530   B=430
```


スケジュール (Schedule)

DB**操作** (Read, Write, Rollback, Commit) **の実行順を示すリスト**

```
T1:   A=A+100,   B=B-100
```

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

```
T2:   A=1.06*A, B=1.06*B
```

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

Ri(A)**は**A**に対するトランザクション**i**の**Read**を**
Wi(A)**は**A**に対するトランザクション**i**の**Write**を意味する**

スケジュール (Schedule)

DB**操作** (Read, Write, Rollback, Commit) **の実行順を示すリスト**

正しい同時実行の例

```
T1:   A=A+100,   B=B-100
T2:              A=1.06*A,    B=1.06*B
```

このスケジュールのDBMS**の (概念的な) 内部表現**

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

スケジュール (Schedule)

DB**操作** (Read, Write, Rollback, Commit) **の実行順を示すリスト**

正しくない同時実行の例

```
T1:   A=A+100,                          B=B-100
T2:              A=1.06*A, B=1.06*B
```

このスケジュールのDBMS**の (概念的な) 内部表現**

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

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)

T1: A=A+100, B=B-100
T2: A=1.06*A, B=1.06*B

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

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

正しくない同時実行の例

T1: A=A+100, B=B-100
T2: A=1.06*A, B=1.06*B

このスケジュールのDBMSの (概念的な) 内部表現

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

このスケジュールは直列化できない
(T1のAの書き込みとBの書き込みの中間に，
他の書き込みがあるが，それを中間の位置から
その側の位置に移動させると，このスケジュールの答えが
違ったものになる可能性がある)

直列化可能スケジュールならば，
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，および，トランザクションの隔離性は保証される．

データの一貫性
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
(2) 反復不可能読み状態
(3) 幽霊状態

DBMSはこれらの違反状態が起こらないよう同時実行する．
(これらが起こらないスケジュールは直列化スケジュール
となっている)

汚読 (dirty read) **コミットされていないデータのRead**

T1: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$, R
T2: $R_2(A)$, $W_2(A)$, 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の$W_1(A)$**は破棄・ロールバックされるかもしれないし，**
さらに変更されるかもしれない．
T2は「まだ値が未確定のT1の$W_1(A)$**」に依存する作業を持つため，**
汚読以降の作業が一貫性を崩す可能性がある．

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

T1の$W_1(A)$**がT2により上書き**
(これも一種の汚読 (正確には汚書))

汚読状態

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

この汚読状態が一貫性を壊すシナリオ

T1君はサークルの部費を1000円支払い，部費残高を25000円としたが，今日，コンパがあることを思い出し，部費の支払いをキャンセル（25000円と変更したものを保存せずに終了した）．

T2君は（T1君の書き込みにより）部費残高が25000円となった状態を読み込み，（それがその直後，24000円にもどったことを知らずに）自分の部費を支払って26000円と書き込んだ．

汚読状態（広義の汚読状態）

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

T1の$W_1(A)$がT2により上書き（広義の汚読．「汚書」）

あるトランザクションが書き込んでコミットされていない状態を他人が読んだり，書いたりしてはいけない．

この汚読状態が一貫性を壊すシナリオ

T1君はサークルの部費を1000円支払おうと，24000円だった部費残高を25000円に変更した．

T2君はサークルの部費を1000円支払おうと，24000円だった（その直後T1により25000に変更された状態の）部費残高を25000円に変更した．

反復不可能読み状態（Unrepeatable Read）

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

Readの「反復不可能読み状態」

この反復不可能読み状態が隔離性を壊すシナリオ

T1君はコンサートチケットの残り枚数が2枚あるのを確認し，財布の中身を確認し，さて，買おうと思って再度読んでみたら残り枚数が1枚になっていた．

T2君はコンサートチケットの残り枚数が2枚あるのを確認し1枚購入した．

幽霊状態（Phantom）

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

T1の2回目の検索では，最初の検索には存在しなかったデータ（幽霊データ）がでる可能性がある
「幽霊状態」

この幽霊状態が隔離性を壊すシナリオ

T1君は日曜日のアルバイト求人状況を調べた．3件あったのでその3件を比較検討した．申し込み直前に再度調べると今度は別の魅力的なバイトが加わっており（あわせて4件となり），比較検討が徒労に終わった．

T2さんが日曜日の求人を書き込み．

・反復不可能読み状態では最初に読んだ値そのものが変わっている
・幽霊状態では最初に読んだ値そのものは変わらない

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．