Database

第13回：トランザクション処理
～同時実行制御～

上智大学理工学部情報理工学科
高岡詠子

分離レベル(isolation level)

- 3つの回避可能な現象という観点から，4つの分離レベルが定義
- トランザクションが別のトランザクションとの程度に分離されるかの程度を表す
- それぞれトランザクション処理の効率に与える影響の度合いが異なる

3つの回避可能な現象

1. 内容を保証しない読み取り(dirty read)
   - まだコミットされていないトランザクションが書き込んだデータを別のトランザクションが読んでしまう現象
2. non-repeatable read
   - あるトランザクションが以前に読みとったデータをもう一度読みとったときに別トランザクションによってそのデータが変更削除されたことが明らかになる現象
3. phantom read(仮読み取り)
   - あるトランザクションが検索条件を満たす一連の行を戻す問合せを2度実行する間に、コミットされた別のトランザクションによってその条件を満たす新しい行が挿入されたことが明らかになる現象

分離レベルと異常の関係

異常 / 分離レベル	lost update	dirty read 内容を保証しない読み取り	non-repeatable read	phantom read 仮読み取り
read uncommitted	×	○	○	○
read committed	×	×	○	○
repeatable read	×	×	×	○
serializable	×	×	×	×

○発生しえる
×発生しない

同時実行制御方式

- 直列可能なスケジュールを得るための手段
  - ロックに基づく方式
  - タイムスタンプに基づく方式
  - 多版制御方式
  - 楽観的制御方式

直列可能性

- トランザクションの分離モデル
- 複数のトランザクションが同時ではなく1つずつ(直列に)実行された結果と同じになること
  - あるトランザクションが問い合わせ実行中の表に対して他のトランザクションは挿入はできない
- 異常回避のために単純に順番に実行(直列実行)するか？

↓

- 結果は正しくても並列性は向上しない

↓

- 直列可能性を持つ(直列実行した場合の結果のどれかになる)ように複数のトランザクションを並列実行させる必要がある

直列可能か？

T1	T2
read_item(X);	
	write_item(X=1000);
write_item(X+20);	

時間経過

T1→T2，T2→T1のいずれとも異なる

直列可能か？

時間経過

T1	T2
read_item(X);	
write_item(X);	
	read_item(X);
	write_item(X);

直列可能か？

	T1	T2
	read_item(X);	
時間経過		read_item(X);
	write_item(X=X*2);	
		read_item(Y);
		write_item(Y=Y+X);

T1→T2，T2→T1のいずれかと等しければよい

直列可能か？

T1	T2
read_item(X);	
	read_item(X);
	write_item(X=X＋**1**);
write_item(X=X*2);	

時間経過

T1→T2，T2→T1のいずれとも異なる

直列可能か？

T1	T2
read_item(X);	
write_item(X=X+70);	write_item(X);
	read_item(X);
	write_item(X=X+30);
read_item(Y);	write_item(X=800);
write_item(Y=Y+10);	
	read_item(Y);
	write_item(Y=Y+50);

時間経過

直列可能かどうかの判定

- T2からT1に→を設定する
  - T1がreadを実行する前にT2がwriteを実行する
  - T1がwriteを実行する前にT2がreadを実行する
  - T1がwriteを実行する前にT2がwriteを実行する
- Conflict graphにループがなければ直列可能

ロック

- ロックとは
  - データを予約しておく(⇒lockをかける)
  - ただ一つのトランザクションだけが更新を行うことを許すための制御のしくみ
  - 他のトランザクションはロックが解放(unlock)されるまでそのデータの更新はできない

ロック方式

- 占有ロック(exclusive/write)
  - トランザクションが INSERT 文、UPDATE 文及び DELETE 文であるときのロックのモード
  - 他のトランザクションに参照も更新もさせない
  - データベース更新時に要求
- 共有ロック(shared/read)
  - トランザクションが SELECT 文によりデータを参照するものであるときのロックのモード
  - 他のトランザクションに参照のみを許す．更新は許さない
  - データベース参照時に要求する
- 共有ロック同士は両立
- 占有ロックと他のロック(共有でも占有でも)とは互いに排他的

ロックのイメージ

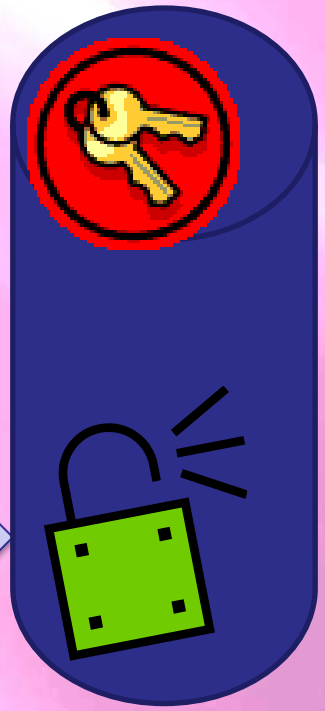
T1

- Start transaction
- Update文を実行

ロック
HOLD

T2

- Select文で問い合わせ



- Commit

ロック解除

ここまで結果表示を待つ

SQLでロックをかける例

Start transaction
Select * from テーブル名 with(ロックオプション)

- From句で指定したテーブルにwith句を使ってロックオプションを指定する
- ロックオプションがHOLDLOCKの場合
- トランザクションが完了するまで指定されたオブジェクトのロックが持続する

ロックのオプションと分離レベル

ロックのオプション	説明
NOLOCK READUNCOMMITED	•SELECT文にだけ適用される、ロックを無視するオプション •指定したテーブルには、共有ロックを掛けず、ほかのトランザクションが排他ロックを掛けている場合も関係なく読み込みを実行 •コミットされていないトランザクションを読み込んだり、読み取りの途中でページがロールバックされたりする可能性あり •ダーティリードが可能
READCOMMITTED	•READ COMMITTED の分離レベルで動作する •デフォルトのロックレベル
REPEATABLEREAD	•REPEATABLE READ の分離レベルで動作する
SERIALIZABLE HOLDLOCK	•ロックをすぐに解除するのではなく、トランザクションの完了時まで保持する

ロックの実現

- あるオブジェクトに対して
  - ロックモード（共有・占有）
  - そのオブジェクトにロックをもつトランザクションの数（共有なら1以上，占有なら？）
  - ロック要求待ち行列

ロック要求のアルゴリズム

- 共有ロックが要求されたとき
  - 待ち行列が空
  - 占有ロックがかけられていない

→ ロックをかけて
ロックテーブルを更新（トランザクションを1増やす）

- 占有ロックが要求されたとき
  - 待ち行列が空
  - いかなるロックもかけられていない

→ ロックをかけて
ロックテーブルを更新

- 条件が満たされない場合はロックの待ち行列に入れる

ロック解放のアルゴリズム

- ロックテーブルを更新(トランザクション数を1減らす)
- トランザクション数が0になったら
  - 待ち行列が空でなければ
  - ロック要求を実行(共有の場合は複数ロックが可能)

ロックと直列可能性

- ロックだけでは 直列可能性を保証するものではない。
- Unlock操作が早すぎると適正な結果を生まない。
- そこで通常はLock, Unlockの操作にプロトコルを設ける⇒ロッキングプロトコル

例：x=20，y=30の場合とうなるか

T1	T2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
X=X+Y;	Y=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

例：x=20，y=30の場合どうなるか

T1

```
read_lock(Y);
read_item(Y);
unlock(Y);
```

T2

```
read_lock(X);
read_item(X);
unlock(X);
write_lock(Y);
read_item(Y);
Y=X+Y;
write_item(Y);
unlock(Y);
```

Unlockが早すぎると
違う結果になってしまう

時間経過

```
write_lock(X);
read_item(X);
X=X+Y;
write_item(X);
unlock(X);
```

違う結果になったら困るので

Two Phase Locking Protocol (2PL・2相ロック)

- Two phase locking protocol
  - 全てのロック操作が終了した後、アンロックする
  - ロックフェーズ(growing phase)とアンロックフェーズ(shrinking phase)が分離
  - ロック獲得とロック解放に分離
  - いったんロックを解放したら追加ロックを行わない
  - トランザクション全部がこれを守れば直列可能

例の2相化

T1	T2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

2相ロックと直列可能性

- 全てのトランザクションが2相ロックプロトコルに従う場合、当該スケジュールは直列可能である。
- 2相ロックはある程度の並行性を犠牲にしている。しかし、直列可能性 チェック不要というメリットの代償と言える。

デッドロック

- データのロックにより発生する
- トランザクションがお互いの処理に必要なデータをロックし合って，処理不能の状態に陥ること

Deadlock を防ぐ手法(1)

- 必要なロックをそれぞれのトランザクションが実行前に一括して全て獲得する
- 一つでもロック出来ないデータがあると、何れのデータもロックしない。一定時間待ち、リトライする。
- この方法と2相ロックを組み合わせて「保守的2相ロッキングプロトコル」という
- 現実的でない為実際には利用されることは無い

Deadlock を防ぐ手法(2)

- データベースの全てのデータに順番を付けて、その順にロックする手法
  - プログラマが順序を気遣うのは非現実的

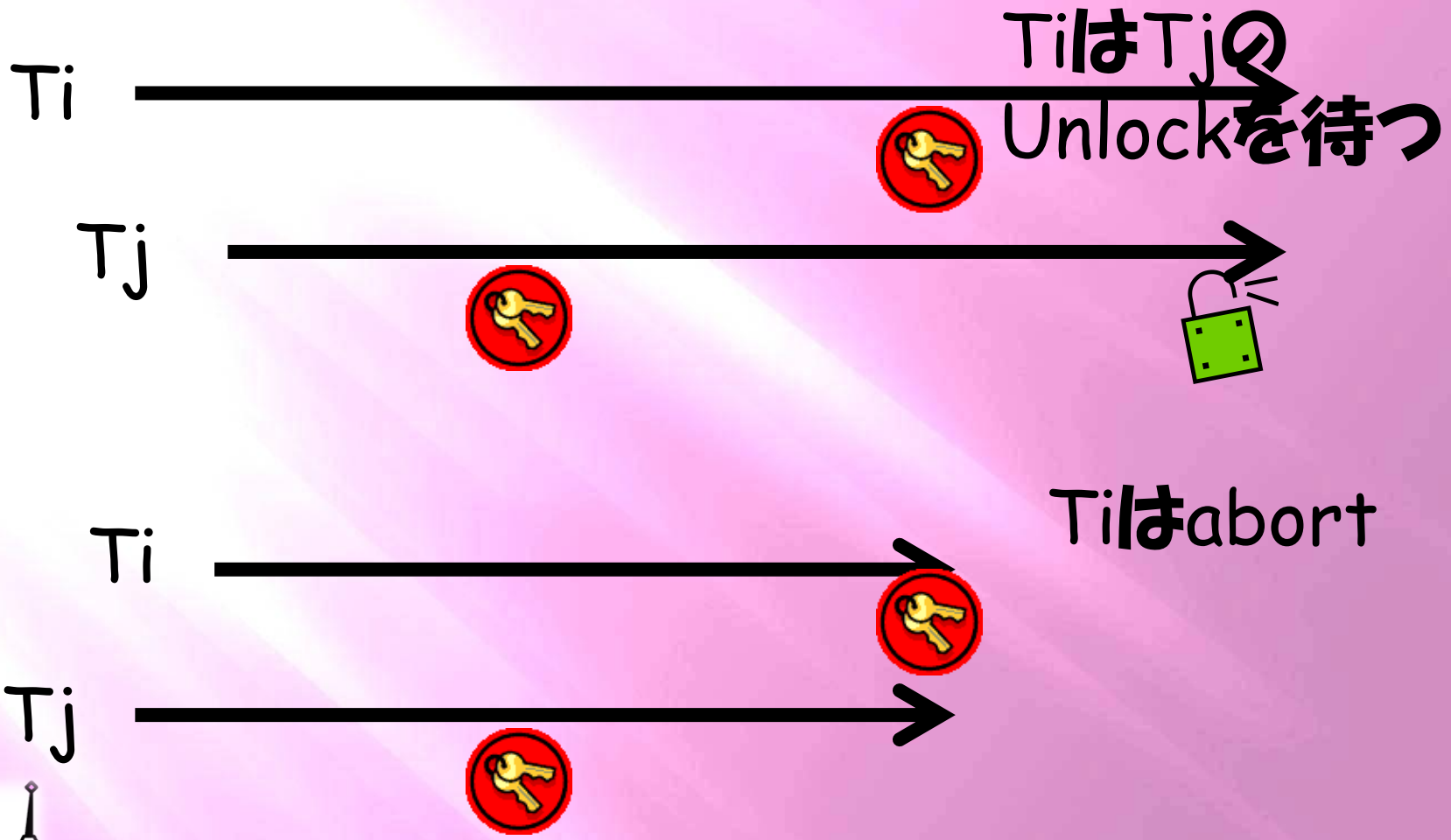
Deadlock を防ぐ手法(3)
トランザクションに優先度をつける

- トランザクションの優先度：Timestamp
  - トランザクションを識別するためにDBMSによって生成されたユニークな値
  - (通常)トランザクション投入時刻
  - TS(T)
  - 2つのポリシーのどちらかを守ればデッドロックフリー
- timestampの付け方
  - 時間が早いトランザクションの優先度が高い
  - トランザクション毎に整数を付与。1づつ増やす
  - 時刻を利用。完全に同時に2つのトランザクションが駆動されることが無いことを保証しておく。

Deadlock を防ぐ手法(3)
Timestampに基づく同時実行制御方式

- timestampポリシー(Ti,Tjが競合)
  - Wait-die:
    - Tiの優先度が高ければTiを待たせ，そうでなければTiをアボート
    - 自分が優先度が高いならば待つことができる
    - 優先度の低いトランザクションは優先度の高いトランザクションを待つことはない(アボートするから.自分より先に開始されたトランザクション(高い優先度)を待つことはない)
  - Wound-wait:
    - Tiの優先度が高ければ，Tjをアボートし，そうでなければTiを待たせる.
    - 自分の優先度が高いならば相手をアボートする。そうでなければ自分が待つ。

Wait-Die



Wound-Wait

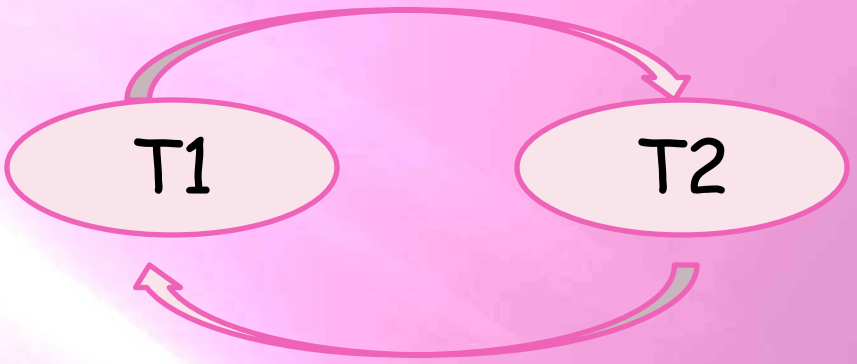
いずれにしても

- 実行中のトランザクションのうち最初に始まったトランザクションはアボートされない
- アボートされたトランザクション：前と同じtimestampで実行されるのでいつかは優先度があがる.

Deadlock の検出

- Deadlockが発生しているかどうかをチェック
- Wait-for グラフ(待ちグラフ)
  - Ti → Tj ： トランザクションTiがトランザクションTjの有するロックを待つことを示す
  - グラフがサイクルを持つと：デッドロック
- タイムアウト
  - 一定時間以上待ちが発生すると、デッドロックが実際に発生しているかどうかによらず、アボートする。簡単に実装可能なため良く利用される。
- 同時実行トランザクションが一定数を越える場合

デッドロックと待ちグラフ



時間経過

T1	T2
write_lock(a);	
write(a);	
	write_lock(b);
	write(b);
ロック待ち状態	
write_lock(b);	
write(b);	
	ロック待ち状態
	write_lock(a);
	write(a);

デッドロックの解消

- トランザクションの一方を強制的に終了させる
- ロックをかけているデータを強制的に終了させる
- 犠牲となるのは
  - ロックの数が少ないもの
  - 更新個数が最も少ないもの

楽観的・悲観的同時実行

- 楽観的な同時実行制御
  - データの競合は「原則として起こらない」ことを前提
  - トランザクション毎に，一時作業領域を確保し，read,writeの結果を一時保存
  - データ取得時にはロックは行わず、更新時にほかのユーザーによる同時更新を検出し，問題があればアボートして再実行
  - ほかのトランザクションで書き込みがあればアボート
  - 書き込みがなければ一時作業領域をDBに書き込む
  - readが多い，競合発生が少ない場合に使う
- 悲観的同時実行制御
  - 競合が起こることを前提に、データ取得時から厳密にロックを制御する方式

二相ロックのバリエーション

- 基本的な(basic)2PL:これまでの定義
  - 単に二相にわけるだけ
- 保守的な(Conservative)2PL(Static 2PL)
  - トランザクションは実行開始以前に全てのロックを取得
  - デッドロックフリー
  - 実際には全てを申告するのは現実的でない
- 厳密な(Strict)2PL
  - トランザクション終了前(コミット・アボート)には全ての排他的ロックを解放しない。
  - デッドロックフリーではない
- 厳格な(rigorous)2PL
  - トランザクション終了前(コミット・アボート)には全てのロックを解放しない。

多版型同時実行制御(MVCC)

- マルチユーザ環境におけるデータベースの性能を向上させる高度な技術
- データ読み込みの際，各トランザクションはデータの現在の状態に関知しない
- 現在から遡ったある時点におけるスナップショットを参照する
- readロックの獲得とwriteロックの獲得は競合しない