トランザクションと同時実行処理制御

内容

- トラントランザクション処理
  - トランザクションとその性質 (ACID)
- 同時実行と直列化可能性
  - 同時実行の目的
  - 同時実行したときの正しさの判定
- 同時実行制御
  - トランザクションを正しく同時に実行するための制御方式
  - 2相ロック方式
  - 時刻印方式

トランザクション処理

トランザクションとは

- **アプリケーションレベルでのデータベースに対する処理の単位**
- 原子的(atomic)な作用を行う
  - 原子的:それ以上分解できない
  - 作用:データに対する読み出し，書き込み操作
- なぜアプリケーションレベルに限定するのか
  - SQLの個々の命令単体 (SELECTやINSERTなど) でデータベースに対する処理が完結することはほとんどない
  - いくつか命令を組み合わせて初めて意味のある処理になる → これをトランザクションと呼ぶ

トランザクション例（振替送金）

A氏の口座からB氏の口座に以下の手順で100万円を振り込む時の銀行のデータベースに対する操作を考える (赤字はSQLの個々の命令に対応)

1. A氏の口座に100万円以上の残高があるか確認
   なければ残高不足を通知をして終了
2. A氏の口座から100万円引く
3. B氏の口座に100万円加える

続けて確実に実行しなければならない (原子的処理)

SQLでは BEGIN TRANSACTION と COMMIT で囲むことで原始的処理を実現する.
ROLLBACK によりトランザクションが実行されなかった状態で終了できる

トランザクション例（引出し）

A氏が口座から100万円を引き出す時の銀行のデータベースに対する操作を考える

1. A氏の口座に100万円以上の残高があるか確認
   なければ残高不足を通知をして終了
2. A氏の口座から100万円引く

データベースの一貫性

- データベースは実世界の写し絵である (p. 1より)
  - データベース中のデータの状態は，世の中と状態と矛盾してはなら ない
- 同じ操作を実行した後でも，アプリケーションの状態によっては，一貫性が取れていたり，取れていなかったりすることがある
  - 振替送金の場合は，A氏の口座から100万円減じた段階では一貫していない
  - 引き出しの場合は， A氏の口座から100万円減じた段階で一貫している

トランザクションが持つべき性質

ACID特性

- 原子性 (Atomicity)
- 一貫性 (Consistency)
- 隔離性 (Isolation)
- 耐久性 (Durability)

- 原子性　トランザクションは，それら全てが実行されるか，あるいは一切実行されない，の二者択一（all or nothing）でなければならない
- 一貫性　トランザクションは，一貫性のあるデータベースに対し，実行後も一貫性を保たなければならない
- 隔離性　複数のトランザクションを並行処理した場合でも，互いに影響を受けず，その結果はそれらを何らかの順序で逐次実行したものと一致しなければならない
- 耐久性　コミットしたトランザクションが行ったデータ操作の結果は決して消えない

トランザクションマネジャ

トランザクションの実行を管理する

- トランザクションに番号を付与して，同時実行スケジューラへ送付
- トランザクションのデータベースに対する操作をモニタリング (読み書き情報を観測)
- モニタリングして得られた情報をログマネジャに通知 (障害回復にログ法を用いる場合)

同時実行

データベースシステムに同時に複数のトランザクションの実行依頼が来た場合の処理として以下が考えられる

- 一つずつ順に実行 (直列実行)
  - データベースの一貫性は保持されるが，データベース（ディスク)へのアクセスとCPU処理が交互に実行され，処理効率が悪い
  - 時間がかかるトランザクションがあると，それ以降のトランザクションの応答が悪くなる.

- データベースに対するトランザクションの基本操作（(読み出し (*read*) / 書き込み (*write*)) を並行して実行 (同時実行)

複数のトランザクションの基本操作を並行して実行

↓

あるトランザクションの基本操作の処理中に，
別のトランザクションのCPU処理を行うことができる

↓

処理効率の向上

データベースの一貫性を保持するための仕組みが必要

同時実行を考えるときのトランザクションのとらえ方

- A さんの口座から B さんの口座に 10万円 を振り込む

```
read(A, x)                  // Aさんの口座の残高を読み出す
read(B, y)                  // Bさんの口座の残高を読み出す
x := x–100,000//            CPU処理
y := y+100,000//            CPU処理
write(A, x)          // Aさんの口座の残高を更新する
write(B, y)          // Bさんの口座の残高を更新する
```

- 他のトランザクションとの競合はデータベースへの読み書きで生じる

↓

トランザクションをデータの読み書きの系列で表す

(簡単のために *read* は r, *write* は w で表し，内部変数は省略)

例: T: r(A) r(B) w(A) w(B)

同時実行時の異状例

- 遺失更新異状 (pp. 151～152)
  - 更新が反映されない
- 汚読異状 (p. 152～153)
  - コミットしていないトランザクション (T_pとする) が更新したデータを更新しコミットする
  - そのあと T_pがロールバック (T_p が実行されなかった状態に戻す) する
- 反復不可能な読み異状を呈する同時実行 (p. 153)
  - 同じデータを再度読み出したときに以前と値が異なっている異状
  - 前の読出しとの間に更新されたことが原因

2つのトランザクションを仮定する

T_1: A さんの口座から B さんの口座に 10,000 円振り込む　$r_1(A)\ w_1(A)\ r_1(B)\ w_1(B)$

T_2: C さんの口座から B さんの口座に 5,000 円振り込む　$r_2(C)\ w_2(C)\ r_2(B)\ w_2(B)$

T_1 と T_2 の操作が並行して実行される系列を考える

- 系列1:　$r_1(A)\ r_2(C)\ w_1(A)\ r_1(B)\ w_2(C)\ w_1(B)\ r_2(B)\ w_2(B)$
  - B さんの口座は +15,000円となり，正しい結果になった
- 系列2:　$r_1(A)\ r_2(C)\ w_1(A)\ r_1(B)\ w_2(C)\ r_2(B)\ w_2(B)\ w_1(B)$
  - B さんの口座は ＋10,000円となり，正しくない結果になった
  - **T_2の操作が反映されていない !!!**
- 系列3:　$r_1(A)\ r_2(C)\ w_1(A)\ r_1(B)\ w_2(C)\ r_2(B)\ w_1(B)\ w_2(B)$
  - B さんの口座には +5,000円となり，正しくない結果になった
  - **T_1の操作が反映されていない !!!**