具体的な講義の項目

- イントロダクション
- データベースのモデル
- 関係データベース（関係代数、関係論理）
- 正規化
- 問い合わせ言語
- トランザクション処理
- データベースの内部構造
- データベースの問い合わせ処理
- その他（オブジェクト指向DB、アクティブDB等々）

本日の内容

- 時刻印方式
- 障害回復
  - 障害の分類
  - ログを用いた障害回復
  - 障害回復アルゴリズム
  - ログを用いない障害回復
  - メディア障害

時刻印 (Timestamp) 方式

- 各トランザクションが開始した順番で一意的な時刻印を与える
  - その時刻印に従って Serializability を実現
- 各項目 X に 2 種類の時刻印を持たせる
  - $ts_r(X)$: これまでに X に read を行なったトランザクションの時刻印の最大値
  - $ts_w(X)$: これまでに X に write を行なったトランザクションの時刻印の最大値

ルール

- トランザクション T_i の時刻印を $ts(T_i)$ とする:
  - T_i が X を Read する場合:
    - $ts(T_i) < ts_w(X)$:X は既に書き換えられしまっているので、T_i をアボート
    - $ts(T_i) \geqq ts_w(X)$:Read 処理をし、$(ts_r(X) = max(ts_r(X), ts(T_i))$とする。
  - T_i が X を Write する場合:
    - $ts(T_i) < ts_r(X)$:X が既に後ろのトランザクションに読まれてしまっているので、T_i をアボート
    - $ts(T_i) \geqq ts_r(X)$ かつ $ts(T_i) < ts_w(X)$: X が後ろのトランザクションに書き換えられてしまった後なので、T_i をアボート
    - $ts(T_i) \geqq ts_r(X)$ かつ $ts(T_i) \geqq ts_w(X)$: Write 処理をし、$(ts_w(X) = max(ts_w(X), ts(T_i))$ とする。

時刻印とロックの組み合わせ

- Serializability はロックによって実現
  - デッドロックを時刻印によって回避
- Wait-Die 方式 (T_i の視点から)
  - $ts(T_i) < ts(T_j)$ なら、T_i は T_j がロックを解放するまで待つ。
  - $ts(T_i) > ts(T_j)$ なら、T_i はアボートする。
- Wound-Wait 方式 (T_i の視点から)
  - $ts(T_i) < ts(T_j)$ なら、T_j をアボートさせるよう試みる。
    - その時点ではアボートしないかもしれないが、最終的なコミット時にはアボートする。
  - $ts(T_i) > ts(T_j)$ なら、T_i は T_j がロックを解放するまで待つ。
- 時刻印は一意なので、$ts(T_i) = ts(T_j)$ はあり得ない。

Wait-DieとWound-Waitの補足

- いずれの方式も後で開始されたトランザクションがアボートするが、
  - Wait-Die 方式では自分より先に開始されたトランザクションを待つことは無く、
  - Wound-Wait 方式では自分より後に開始されたトランザクションを待つことは無い

  ので、待ちのサイクルは生じず、デッドロックは発生しない。
- アボートさせられたトランザクションは、同じ時刻印を持って再実行すれば、いずれはアボートされること無く実行されることを保証できる。

ちょっと考えてみよう（8－1）

- ロック、時刻印、ロックと時刻印の組み合わせについて、それぞれの長所と短所をまとめてみましょう。
  - ロックの長所
  - ロックの短所
  - 時刻印の長所
  - 時刻印の短所
  - 組み合わせの長所
  - 組み合わせの短所

障害回復

- システムの前提
  - データベースバッファ（主メモリ）＋磁気ディスク
- 障害の分類
  - トランザクション障害
    - トランザクションがコミット前に異常終了する場合
      - 自主的なアボート、デッドロック解消のためのアボート等
  - システム障害
    - システムは停止するが、不揮発性記憶上のデータは残る場合
  - メディア障害
    - データを記憶した不揮発性記憶に障害が発生し、データが読み出せなくなる場合

障害の違い

- どこにデータが残っているかの違い
- トランザクション障害
  - 基本的にはトランザクションに関する全てのデータは保存される
- システム障害
  - 主メモリ上に置かれていたデータは消失する
- メディア障害
  - 他の不揮発性記憶にデータを保存して置かない限り全て消失
    - 部分的なメディア障害もありうる

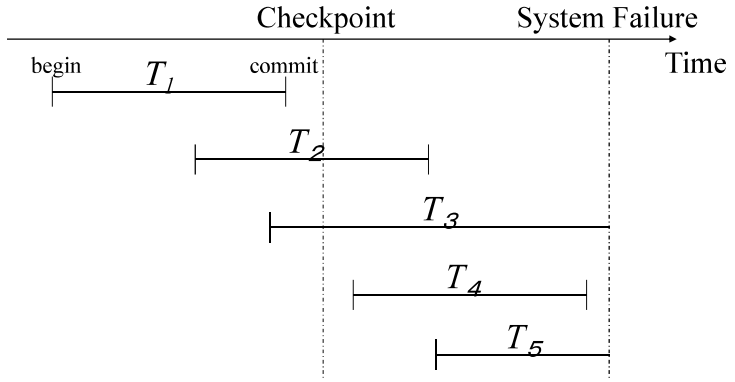
ログを用いた障害回復

- ログ (Log)
  - トランザクションの基本操作の履歴
- 記録内容
  - トランザクション T_iの開始: $begin(T_i)$
  - T_iによるデータ X の書き込み: $write(T_i, X, V_{old}, V_{new})$
  - T_iによるデータ X の読み出し: $read(T_i, X)$
  - T_iのコミット: $commit(T_i)$
  - T_iのアボート: $abort(T_i)$
  - チェックポイントの情報: $checkpoint([T_a, \ldots, T_b])$

ログの処理

- WAL (Write Ahead Log) プロトコル
  - 実際のデータベースに書き込む前にログに履歴を残す
    - データベース更新中の障害にも耐えられるように
- Redo と Undo
  - Do: Old State → New State + Log
  - Redo: Old State + Log → New State
  - Undo: New State + Log → Old State
- チェックポイント (Checkpoint)
  - 障害回復の時間を短縮させるための手段
    - データベースバッファの内容をディスクに反映
    - チェックポイント処理時のトランザクションのリストを格納

チェックポイントとシステム障害に対するトランザクションの種類



トランザクションの状態と回復処理

- T_1:チェックポイント(CP)前にコミット済み
  - データベースの状態はディスク上に反映 → 回復処理不要
- T_2 :CP前に開始、システム障害(SF)前にコミット済み
  - CP からコミット時点まで Redo が必要
- T_3:CP前に開始、SF 時にはまだコミットしていない
  - 原子性を保つためアボート → T_3開始時まで Undo
- T_4:CP後に開始、SF 前にコミット済み
  - 開始時点からコミット時点まで Redo が必要
- T_5:CP後に開始、SF 時にはまだコミットしていない
  - 原子性を保つためアボート → T_5開始時まで Undo

障害回復アルゴリズム(1)

1. Undo 用と Redo 用の2 つのトランザクションリスト L_{undo} と L_{redo} を用意する
2. 最新の *checkpoint*($[T_a, \ldots, T_b]$) の中のトランザクションを L_{undo} に入れ、L_{redo} を空にする
3. 最新のチェックポイントからログを時間軸に沿って読み込む
   - *begin*(T_i) 検出:T_i を L_{undo} に加える
   - *commit*(T_i) 検出:T_i をL_{undo}から取り出しL_{redo} に加える
   - *abort*(T_i) 検出:読み飛ばす
4. ログの最後まで行くとUndoとRedoのリストが出来る

障害回復アルゴリズム(2)

5. ログを時間軸と逆に読みながら、L_{undo} にあるトランザクション処理の取り消しを行う
6. 再度ログを時間軸方向に読みながら、L_{redo} にあるトランザクション処理の再実行を行う

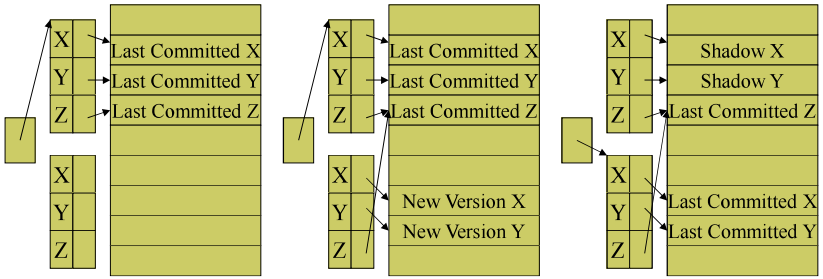
- 更新の取消しと再実行
  - ログの内容が *write*(T_i, X, V_{old}, V_{new}) なら
    - V_{old} (Before Image)にすることで取り消し
    - V_{new}(After Imange)にすることで再実行
  - 何度適用しても同じ:巾等 (Idempotent)
    - 差分適用だと、巾等にはならない

ちょっと考えてみよう(8－2)

- 今ログの中に以下のようなデータが残っていたとする。
  - *checkpoint*($[T_1, T_2, T_3]$), *commit*(T_1), *begin*(T_4), *read*(T_4, X), *write*(T_4, X, V_1, V_2), *abort*(T_2), *begin*(T_5), *read*(T_5, Y), *write*(T_5, Y, V_3, V_4), *commit*(T_4), *checkpoint*($[T_3, T_5]$), *begin*(T_6), *read*(T_6, X), *write*(T_6, X, V_5, V_6), *commit*(T_6), *abort*(T_5), *commit*(T_3), *begin*(T_7), *write*(T_7, Y, V_3, V_8)
- この後システム障害が発生したとすると、どのように回復すべきで、回復した後の X と Y の値はどうなるか

ログを用いない障害回復

- ログの必要な理由：上書きによる旧データの喪失
- Shadowing
  - 旧データと別な場所に新データを保存（元に戻すことが可能）



PostgreSQL:MVCC (Multi-Version Concurrency Control) →Read Committed

メディア障害対策

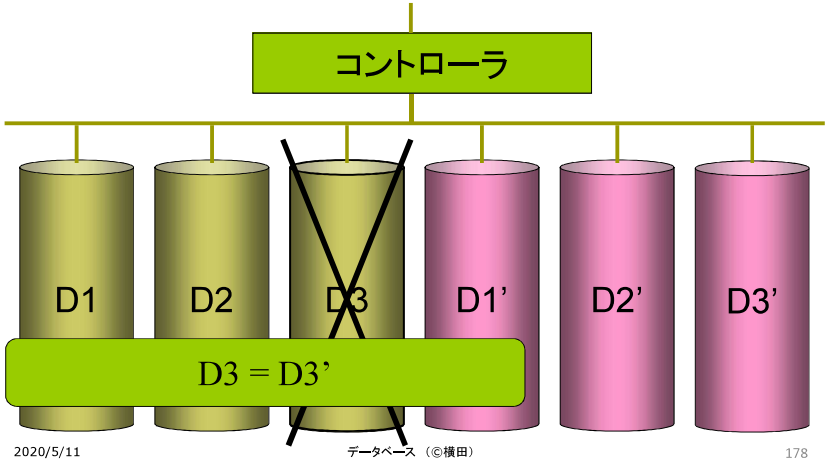
- ソフトウェア的対策（＋ハードウェアも必要）
  - ダンプ
    - ディスクの内容をテープ等に定期的に吸い上げておく
  - リプリケーション
    - 別のディスクに同じデータを格納しておく
- ハードウェア的対策
  - ディスクの耐故障化
    - RAID (Level 1 - 6)

RAID

- Redundant Array of Inexpensive Disks
  - RAID 1: ミラーリング
  - RAID 2: エラー訂正コード(ECC) [ハミングコード]
  - RAID 3: 同時更新パリティコード
  - RAID 4: 独立更新パリティコード(集中)
  - RAID 5: 独立更新パリティコード(分散)
  - RAID 6: ＋リードソロモン符号（耐多重故障）

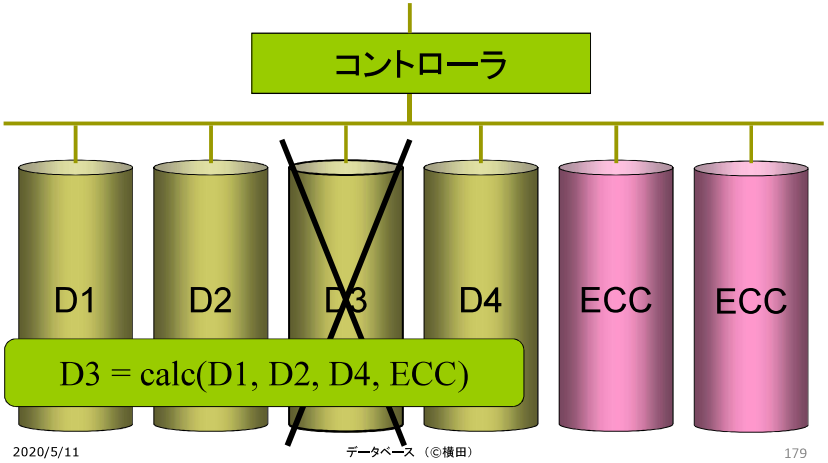
RAID1 の構成例

- ミラーリング
  - 同じ内容を Di と Di' に書く（一方が故障しても大丈夫）

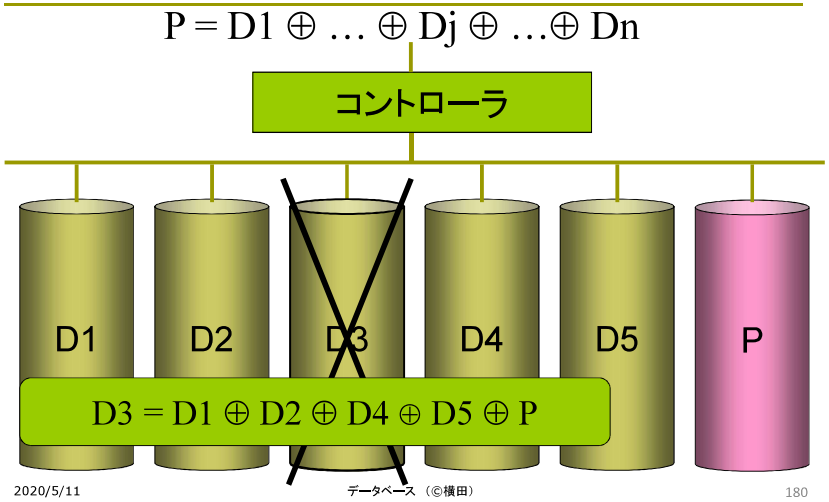


RAID2 の構成例

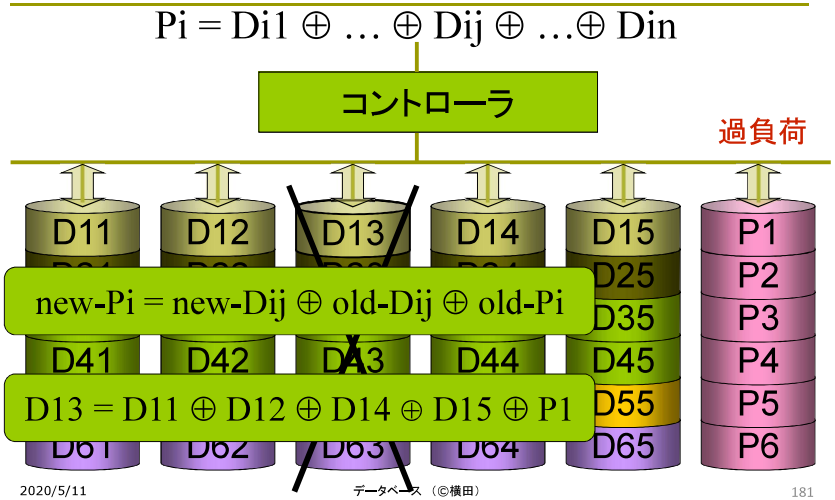
- エラー訂正コード(ECC)用のディスクを用意
  - 故障したディスクの内容は ECC で復元



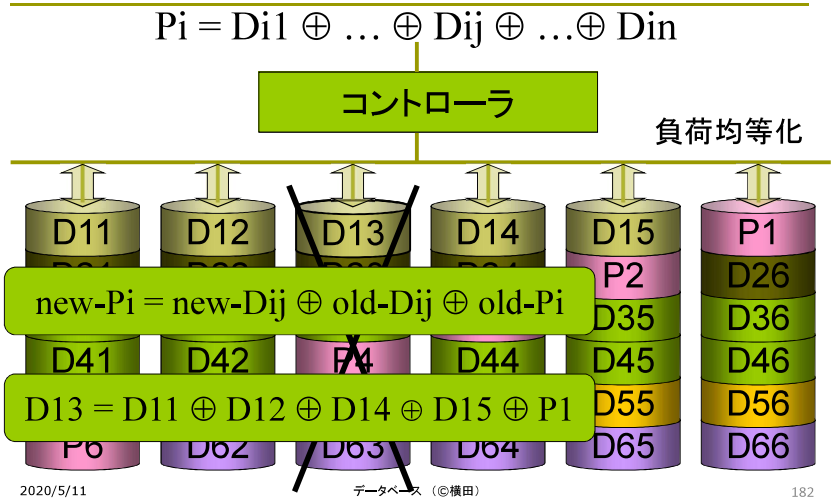
RAID3 の構成例



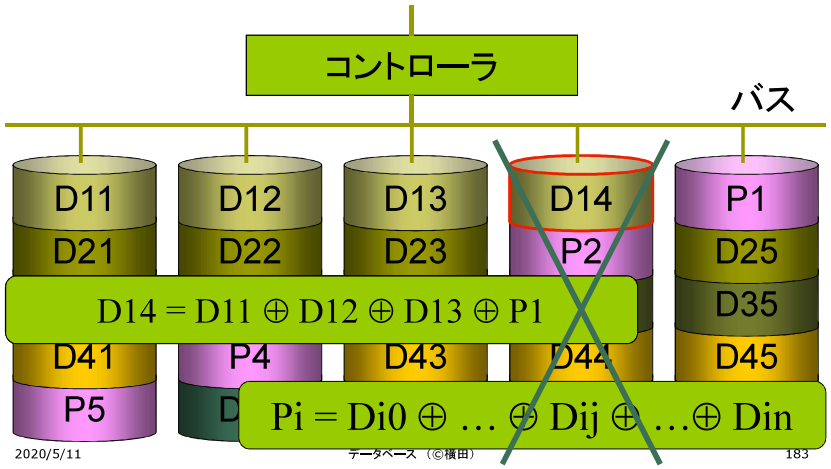
RAID4 の構成例



RAID5 の構成例



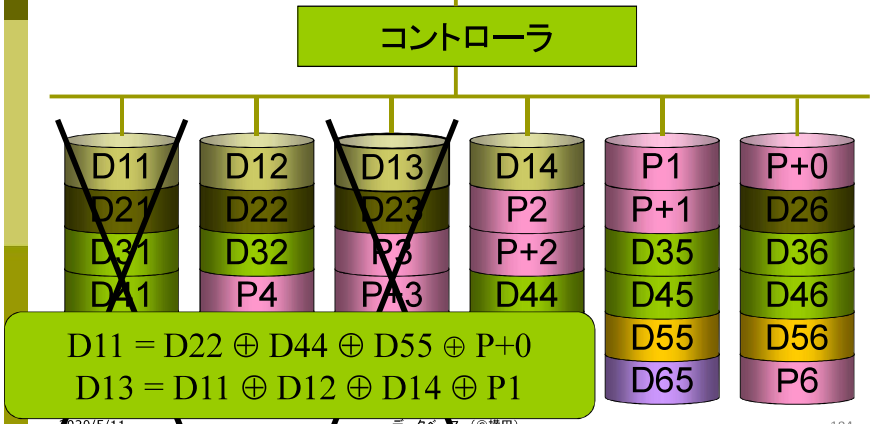
RAID5 の構成例



RAID6 の構成例

$$Pi = Di,1 \oplus \ldots \oplus Di,j \oplus \ldots \oplus Di,n$$

$$P+k = D1,1+k \oplus \ldots \oplus Di,i+k \oplus \ldots \oplus Dn,n+k$$



Reed-Solomon Coding や EVENODD Coding を用いた RAID 6 もある。

ちょっと考えてみよう（8－3）

- 最大8TBのデータを格納したいとする
- HDD(SSDでもOK)の記憶容量は1TBとする
- RAID1, RAID2, RAID3, RAID4, RAID5, RAID6 でそれぞれ何台のドライブが必要か。
  - なお、ECC はデータの $\log_2$ の容量が必要とする