SRA OSS, INC.

チュートリアル
同時実行とトランザクション設計

2011/02/25 PostgreSQLカンファレンス ［C-3］

15:30 - 16:20

(当日版)

SRA OSS, Inc. 日本支社 高塚 遙

データベース設計というと

概念設計
論理設計
物理設計

という分類がおなじみですが、

テーブル設計
トランザクション設計

という軸もありまして、

今日のテーマはこれです。
更にブレイクダウンすると・・・

- クエリ設計
- 同時実行設計
- etc..

テーブル設計やクエリ設計は書籍豊富

テーブル設計の本は
いろいろある。

クエリ設計／SQL書き方
の書籍も名著いろいろ

トランザクション設計参考文献は相対的に少ない

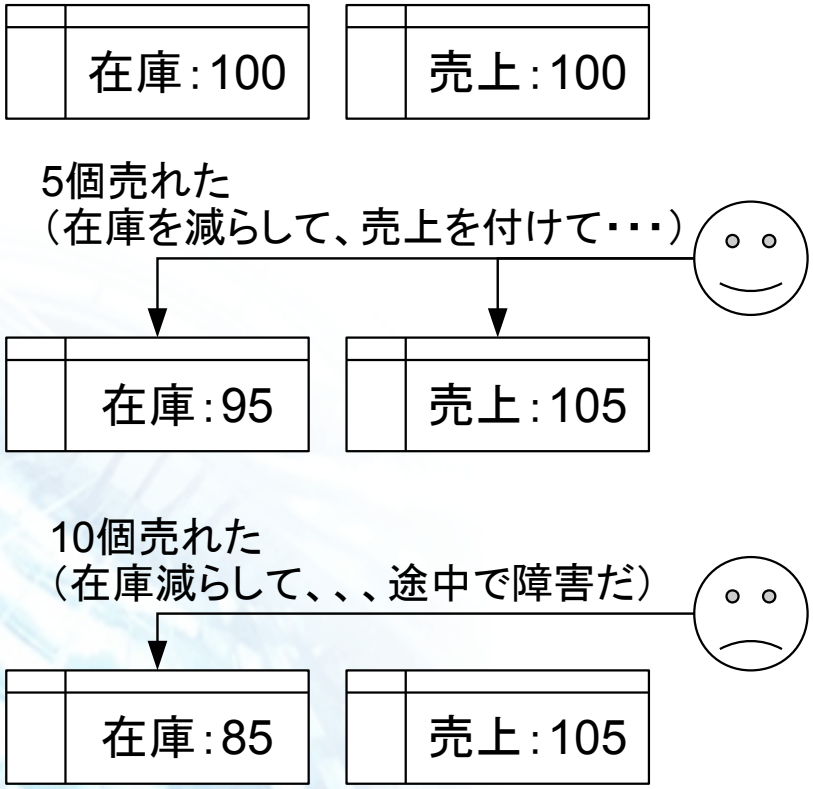
- たいした内容ではないから？
- データベースに限った技術じゃないから探しにくい？
- アカデミック方面や仕組み系の本なら
  - でも、実装手法を知りたいわけでもなかったり
- Web記事はそこそこある

まともなシステムには
間違いなく
絶対必要な技術

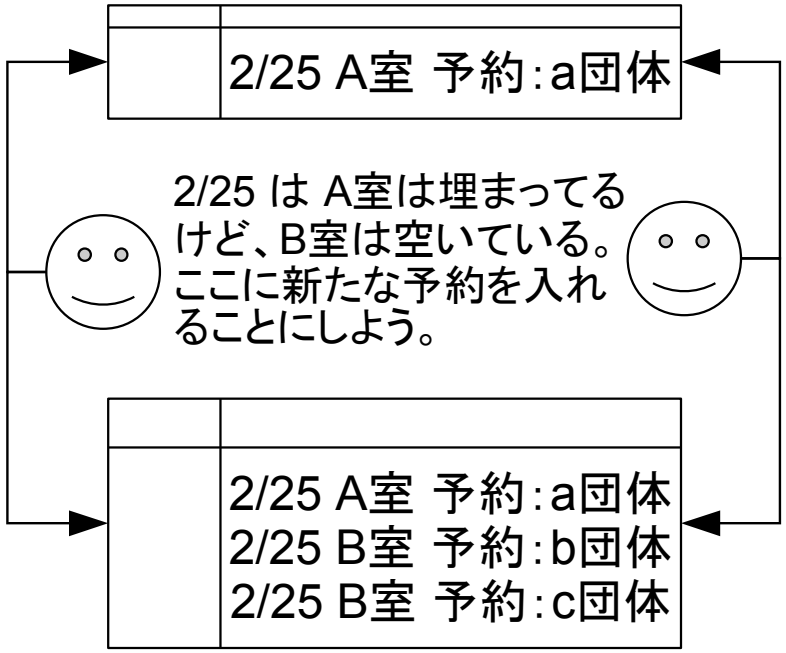
Table of Contents

トランザクション制御　なぜ必要

途中の障害の後始末



並列実行による問題



トランザクションって？

一般には

- 『一連のひとまとまりで行われるべき仕事』など

```
BEGIN;
SELECT ・・・
UPDATE ・・・
DELETE ・・・
COMMIT;
```

PostgreSQL的には

- BEGIN と COMMIT とでくくった一連の SQL
- 複数の処理のすべてが成功しなければすべて元に戻る

ACIDトランザクション

A	複数の処理のすべてが成功しなければすべて元に戻る
C	開始前、完了後には一貫性／整合性のある状態である
I	トランザクション中の過程が他の操作から隠蔽される
D	成功したトランザクションの結果は持続する

PostgreSQLが備える機能（1）
BEGIN～COMMIT

大昔（6.x 時代とか）からトランザクションサポート済

- ACID の AD は おのずと完備、C もアプリケーションプログラムがおかしくなければ問題ない
- 「途中障害の後始末」はトランザクション機能を素直に使えば解決
- デフォルトは自動COMMIT（1つのSQL毎にCOMMIT）なので、正しくトランザクション機能が働いているかチェック
  - DB接続するクライアント言語のライブラリやフレームワークで、startTransaction() などを実行していても、PostgreSQL向け実装がダメダメで実は働いていなかったりとか
  - SQLログ見て BEGIN や START TRANSACTION が在るのを確認

PostgreSQLが備える機能（2）
BEGIN～COMMIT

一括実行が必要な例

```
BEGIN;

UPDATE zaiko
 SET num = $1 WHERE zid = $2;

INSERT INTO haisou
 VALUES ($1, $2, $3, ･･･);

INSERT INTO uriage
 VALUES ($1, $2, $3, ･･･);

UPDATE c_point
 SET p1 = $1 WHERE cid = $2;

COMMIT;
```

最後に ROLLBACK; または途中で
エラーを出せばロールバック。

SAVEPOINT機能

```
BEGIN;
INSERT ･･･          ★A
UPDATE ･･･
SAVEPOINT sp1;

INSERT ･･･          ★B
UPDATE ･･･
       （なんらかのエラー）

ROLLBACK TO SAVEPOINT sp1;

INSERT ･･･          ★B'
UPDATE ･･･
COMMIT;
       （部分的にロールバックできる）
```

PostgreSQLが備える機能（3）

ACID の I はデフォルト動作では不十分~~ではない~~

```
BEGIN;
SELECT p1 FROM c_point
 WHERE cid = $1;

アプリ側で算出:
  新ポイント
    ＝ ビジネスロジック(現在ポイント)

UPDATE c_point
 SET p1 = $1 WHERE cid = $2;

COMMIT;
```

```
BEGIN;
UPDATE c_point
 SET p1 = $1 WHERE cid = $2;
COMMIT;
```

別のトランザクションがこのタイミングで
ポイント値を変更！

別トランザクションの効果を無視する結果。

データベース内でロジック処理するなら
UPDATE c_point SET p1 = f($1) ・・・
なら想定と違った結果になる。

PostgreSQLが備える機能(4)
SERIALIZABLE

トランザクション隔離レベルを指定する

マニュアル 13.2.
トランザクションの隔離

- READ COMMITTED
  - デフォルトの動作
  - トランザクション途中でも、他のトランザクションがコミットした内容は見えてしまう。トランザクション中の参照可能内容が一定でない。
- SERIALIZABLE
  - 常にトランザクション開始時のスナップショットが見える
  - **ときどき失敗する／アプリケーションでやり直す仕組みが要る**
  - 処理速度に不利はない

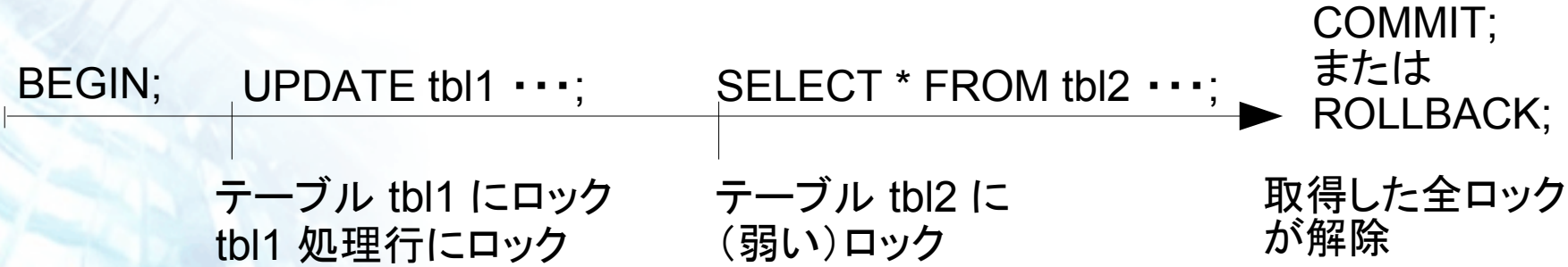
```
BEGIN;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
        （トランザクション本体処理）
COMMIT;
```

PostgreSQLが備える機能（5-1）
二相ロック

トランザクション単位でのロック機構がある

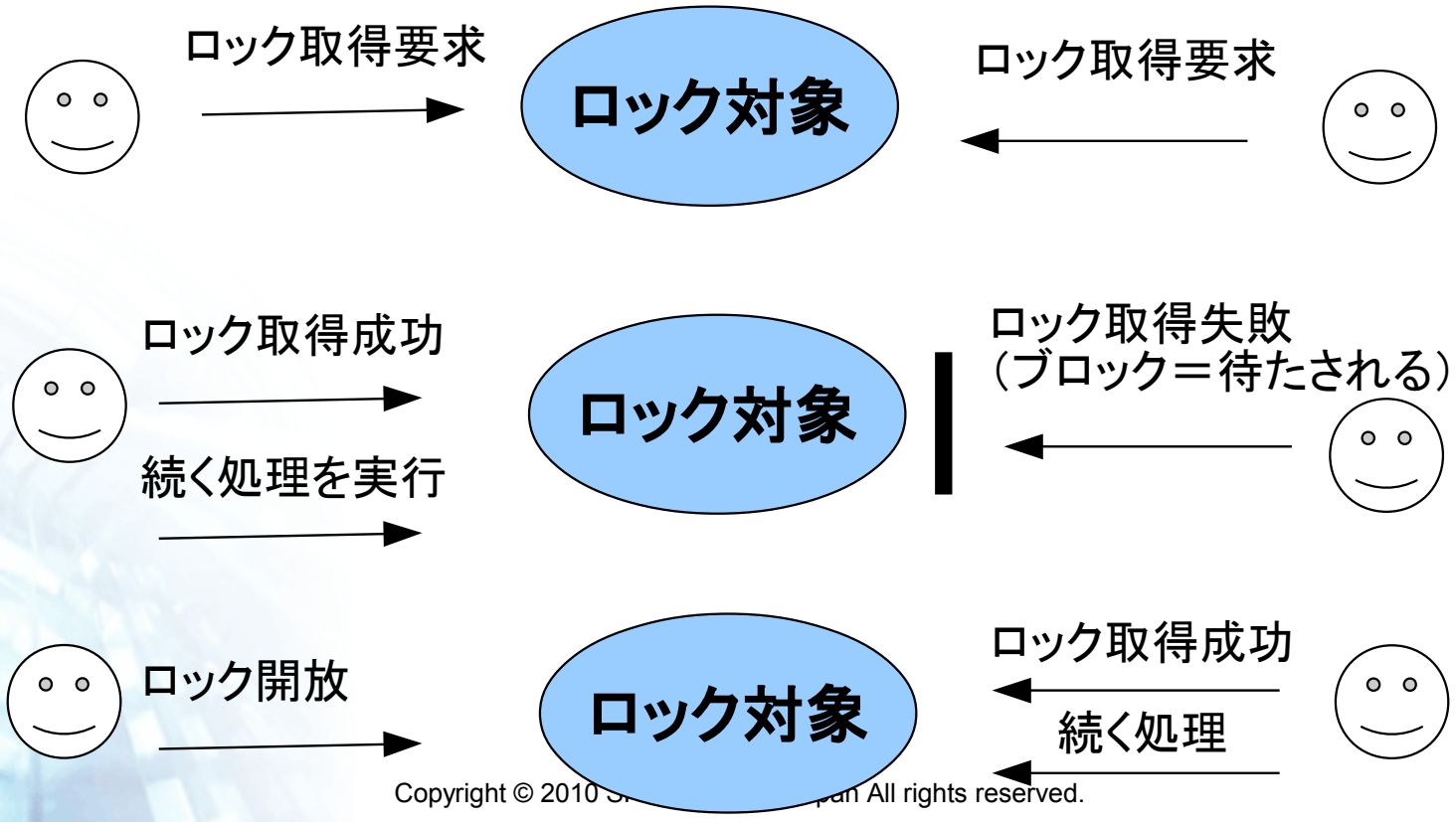
- ロックが追加されていき、トランザクション終了で解除
- ロックレベルに応じて、競合するロック取得をしようとするとそこで処理がブロックされる
- 各SQLで暗黙的に取得される
- LOCK命令等で明示的にもロックできる
- 行ロックとテーブルロックがある

マニュアル 13.3.
明示的ロック



PostgreSQLが備える機能（5-2）
ロックの基本　おさらい

「ロック」と「ブロックする・ブロックされる」の関係



PostgreSQLが備える機能（6）
行ロック

FOR UPDATE / FOR SHARE で明示的に、DELETE、UPDATE で暗黙的に

- データ書き換えパターンで FOR UPDATE 利用
- 存在する行しかロックできない／INSERT には無力

```
BEGIN;
SELECT p1 FROM c_point
 WHERE cid = $1 FOR UPDATE;

アプリ側で算出:
 新ポイント ＝ 現在ポイント + 10

UPDATE c_point
 SET p1 = $1 WHERE cid = $2;

COMMIT;
```

同時に同SQLが実行されても、
ここで待たされるので、
+10 が1個分消えてしまうケースを
防ぐことができる。

PostgreSQLが備える機能(7)
テーブルロック

LOCK TABLE ・・・ IN ・・・ MODE

- INSERT をブロックしたいとき
- テーブル全体を構造的に使っているとき(例:キューとして)
- 各SQLに付随する暗黙的テーブルロック
  - SELECT → ACCESS SHARE レベル
  - INSERT → ROW EXCLUSIVE レベル
  - REINDEX → ACCESS EXCLUSIVE レベル

INSERT をブロックするには、ROW EXCLUSIVE レベルと競合するロックを取れば良い。

NO WAIT

- テーブルロック、行ロックとも、ノンブロック方式も可能

PostgreSQLが備える機能(8)
アドバイザリロック

SELECT pg_advisory_lock(・・・)

- アプリケーション固有のロックに
  - 軽量な行ロックとして利用するケース
- bigint型の番号(またはint型2つ組)でキー指定
- トランザクションとは独立
  - 明示的なロック解除 pg_advisory_unlock(・・・)
  - 接続終了でロック解除
  - **ROLLBACK しても解除されない！**
- ノンブロック動作も可能 pg_try_advisory_lock(・・・)
- 共有ロックも可能 pg_advisory_lock_shared(・・・)

PostgreSQLが備える機能（8）
その他のロック

テーブル、行、以外のロック

- インデックスのロック
- テーブル拡張のためのロック

内部ロック

- LWLock （ライトウェイトロック）
  - SQLレベルでコントロール可能なロックは「ヘビーウェイトロック」
- 共有バッファへのアクセス
- トランザクションログへのアクセス

デッドロック

相互にロック待ちが起きればデッドロック

- PostgreSQLは自動でデッドロック検出してエラーにする
- ロック取得順序が互い違いになっていると起きる

```
BEGIN;
LOCK TABLE t1;
LOCK TABLE t2;
COMMIT;
```

典型的な

デッドロック

```
BEGIN;
LOCK TABLE t2;
LOCK TABLE t1;
COMMIT;
```

- 分かりにくいデッドロック発生SQL

```
UPDATE t3 SET v = v + 1;
```

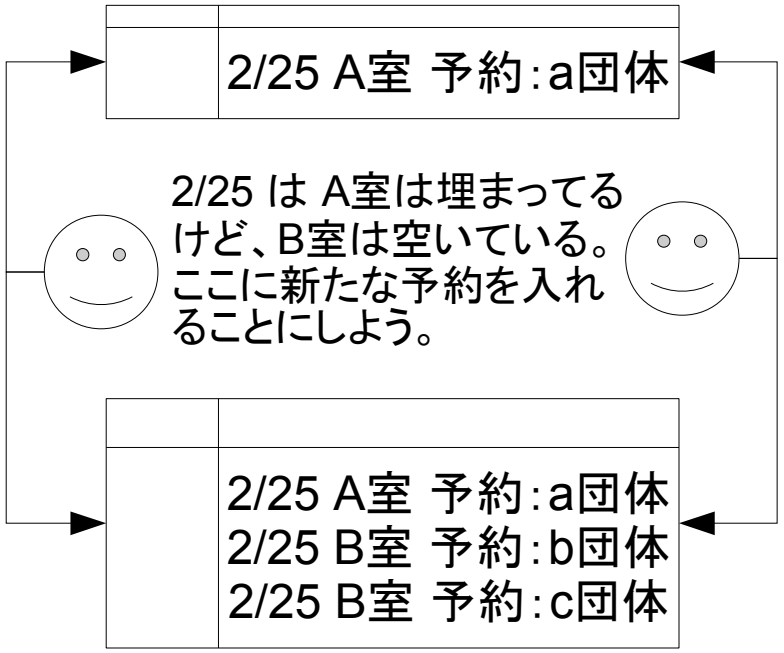
```
UPDATE t3 SET v = v + 1;
```

→ 行ロックの取得順序は保障されないため、互い違いになる可能性あり

短時間トランザクション vs 長時間トランザクション

予約問題再び

- 解決策A
  - テーブルロックをかける
- 解決策B
  - ユニーク制約を付けておいて、重複エラーになったら、「ごめんなさい、やっぱりダメでした、やり直しで」など
- 対話的アプリケーションなら B を使うことが多い（楽観的同時実行制御）



悲観同時制御 vs 楽観同時制御

悲観同時制御

- トランザクション冒頭で変更されては困るデータのロックを取っておく／あるいは SERIALIZE で実行
  - 競合する可能性が高いときに／短時間トランザクションのときに
- アプリケーションレベルで排他処理 （障害時の後始末に注意）

楽観同時制御

- ロック取らず、構わず処理を進める
- トランザクションの最後で他のトランザクションによる変更があって矛盾が生じるかチェック → ダメそうならロールバックやアプリケーションレベルでの取り消し処理
  - 「矛盾が生じるか？」判断はアプリケーション固有
  - 全ての更新処理が「想定内」でないといけない

データベース内 vs システム全体

システム全体から見たトランザクション

- データベース外にもトランザクション管理したいものはある
- データベースは分散システム全体から見たら1リソース

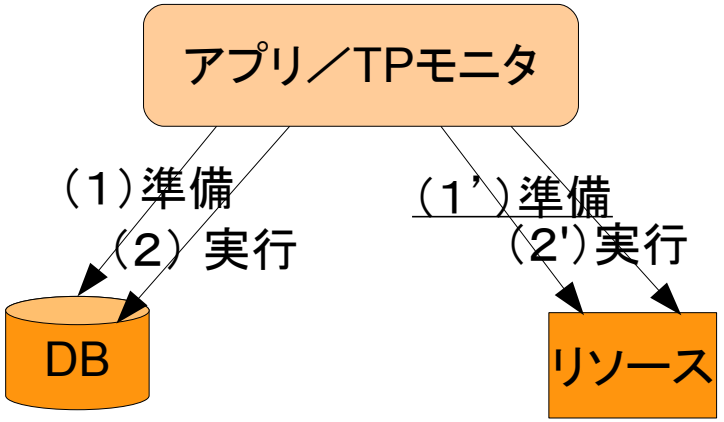
2相コミット（PREPARE TRANSACTION）

```
BEGIN;
INSERT ...
UPDATE ...
PREPARE TRANSCTION 'tx0123';
```

設定 max_prepared_transactions を想定同時利用数以上にしておく。

```
COMMIT PREPARED 'tx0123';
```

```
ROLLBACK PREPARED 'tx0123';
```



ACID vs BASE

疎結合な分散システムや長時間トランザクションでは2相コミットは必ずしも適合しない

CAP Theorem

- Consistency（整合性）
- Availability （可用性）
- Partition Tolerance （分散耐性）

3者を同時には満たせない
→ 形式的に証明された

Eric Brewer のBASE トランザクション

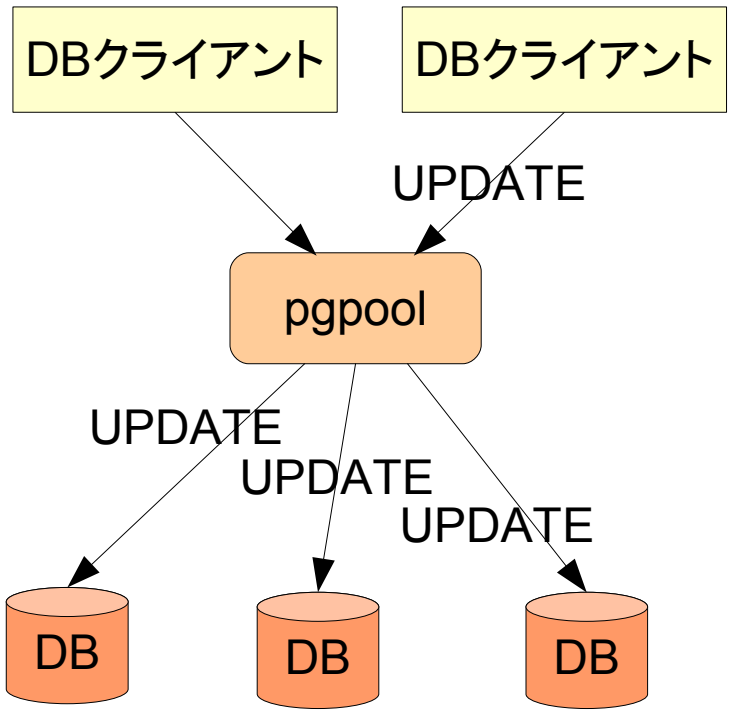
- Basically Available（基本的に可用性あり）
- Soft-State（緩い状態遷移）
- Eventual Consistency （結果的には整合する）

妥当な妥協

マルチノードDBクラスタの世界（1）

pgpool-II

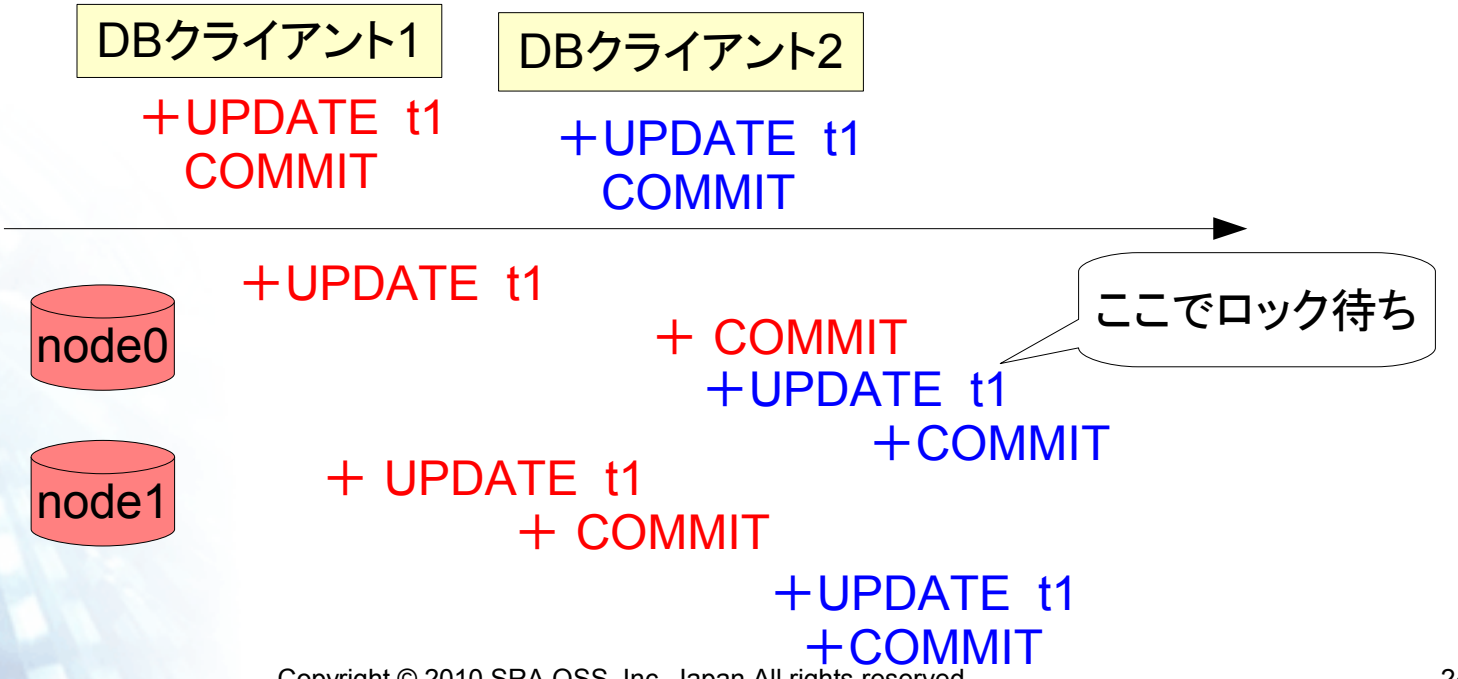
- クエリ複製による同期レプリケーション
  - マスター、その他ノード、の順で実行し、全てコミット後に、クライアントに結果を返す
  - ２相コミットを使っていない
- 参照負荷分散
  - SELECTはどれか１個で実行
- 非同期レプリケーションのフロントエンドとしても使える
  - HS/SR、Slony-I　等



マルチノードDBクラスタの世界（2-1）
pgpool-II

ーマスターノードのCOMMITを最後に実行してくれる

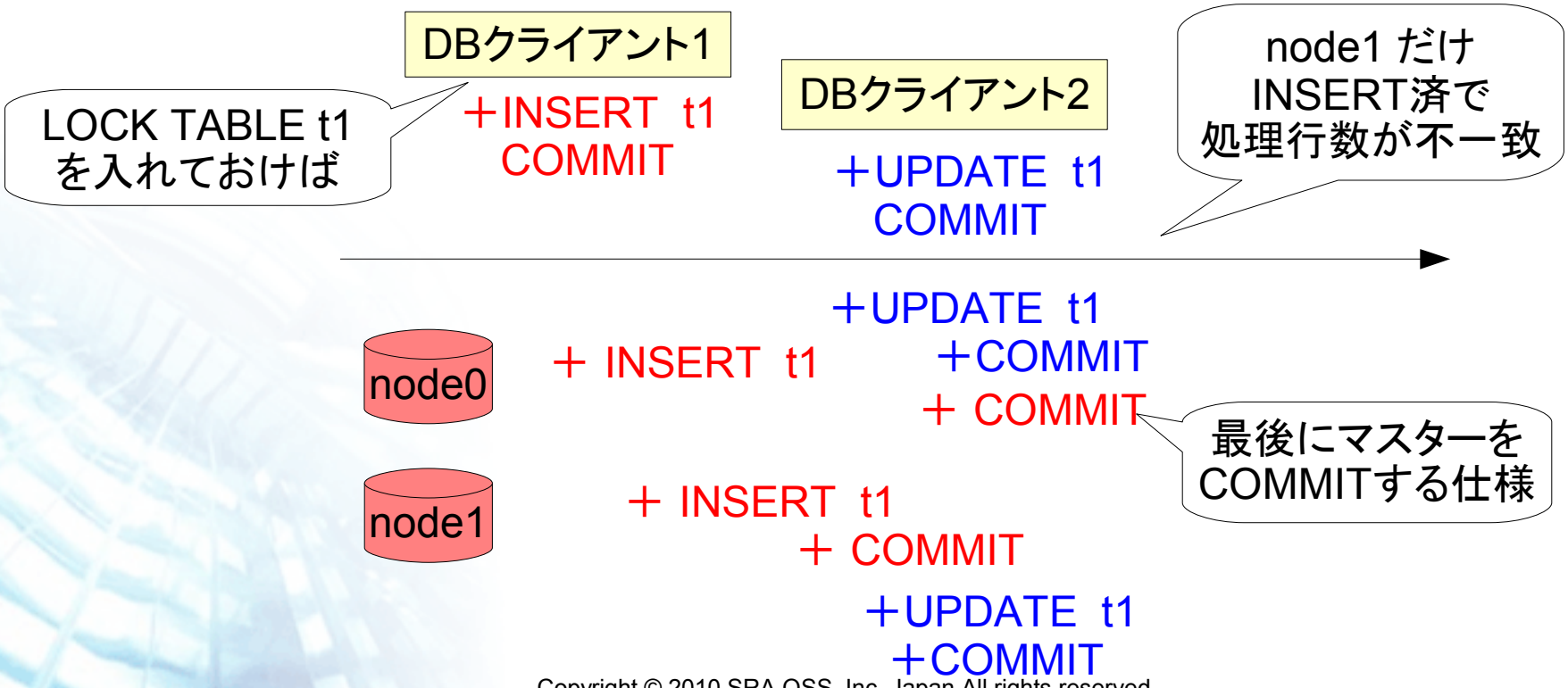
- ロック待ちが node0 内で行われるのでマルチノードデッドロックを回避



マルチノードDBクラスタの世界（2-2）
pgpool-II

一時的にデータが一致しない

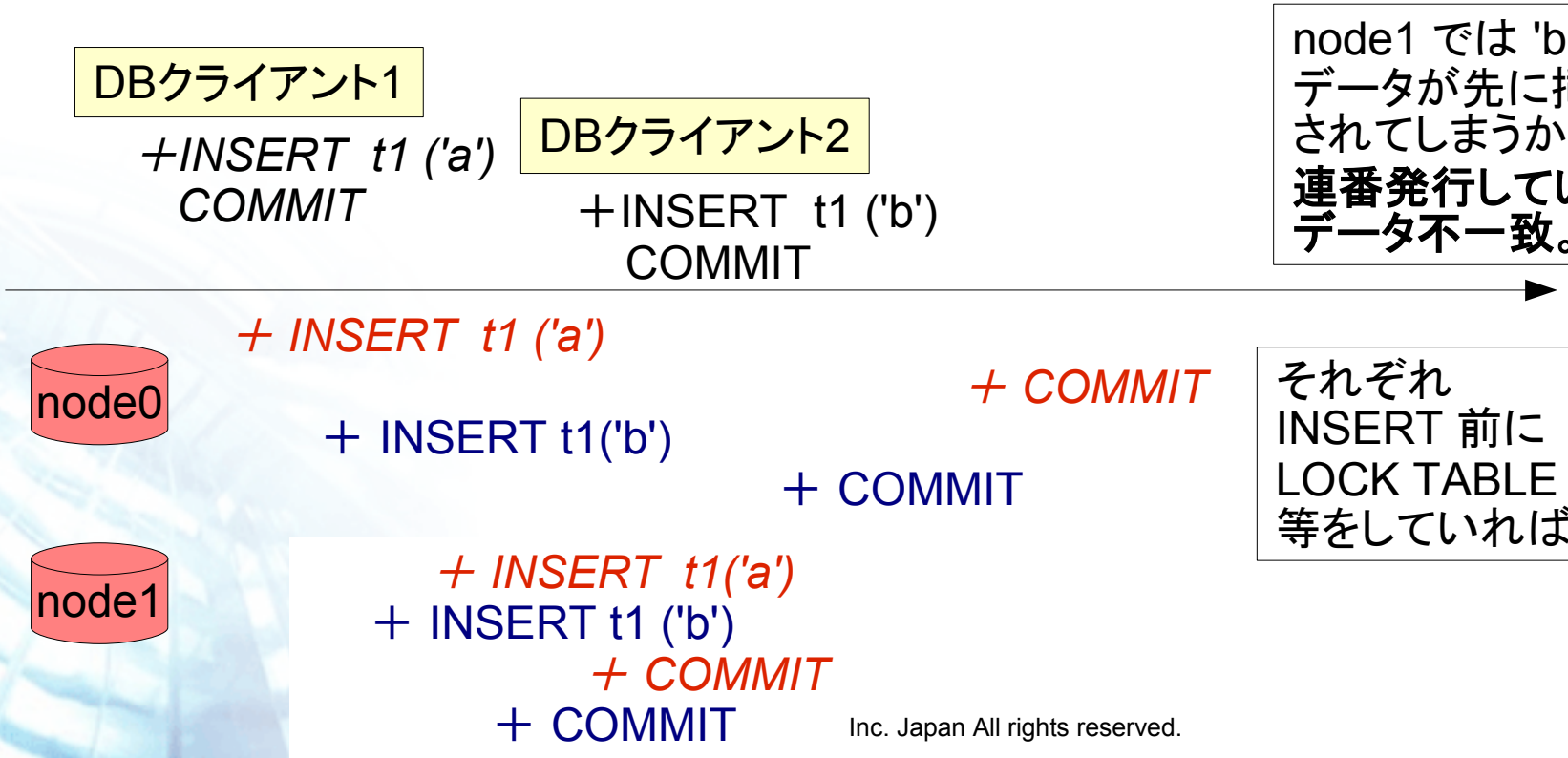
- ロックか、不一致エラーのとき「やり直し」、で回避



マルチノードDBクラスタの世界（3）
pgpool-II

永続的にデータが一致しなくなる

- プロセス実行順序は保障できない／テーブルロックで回避



マルチノードDBクラスタの世界（4）

pl/proxy など、さらに緩いSQLレプリケーション

- 不整合が出そうな処理をそもそも行うべきでない
  - 任意のトランザクションが流せると思ってはいけない

アプリレベル／フレームワークによるレプリケーション

- どう動作するかを正確に把握して、トランザクション設計
- マルチノードデッドロックに注意

非同期レプリケーション

- スレーブノードはマスターノードから遅延し、一致しないと思っておかなければいけない

まとめ

- 「トランザクション設計」は欠かせない
  - 昔は専門家がやってたことをどこでも要求される
  - 技法は大昔に確立しているはずなんだけど、いまひとつまとまった文献や勉強するモノがない
  - 「テーブル設計」とは脳の別の場所を使う
- 緩い分散クラスタ ～ クラウド
  - 新たな定番技法の確立が求められている