Transaction Processing in PostgreSQL 日本語訳版

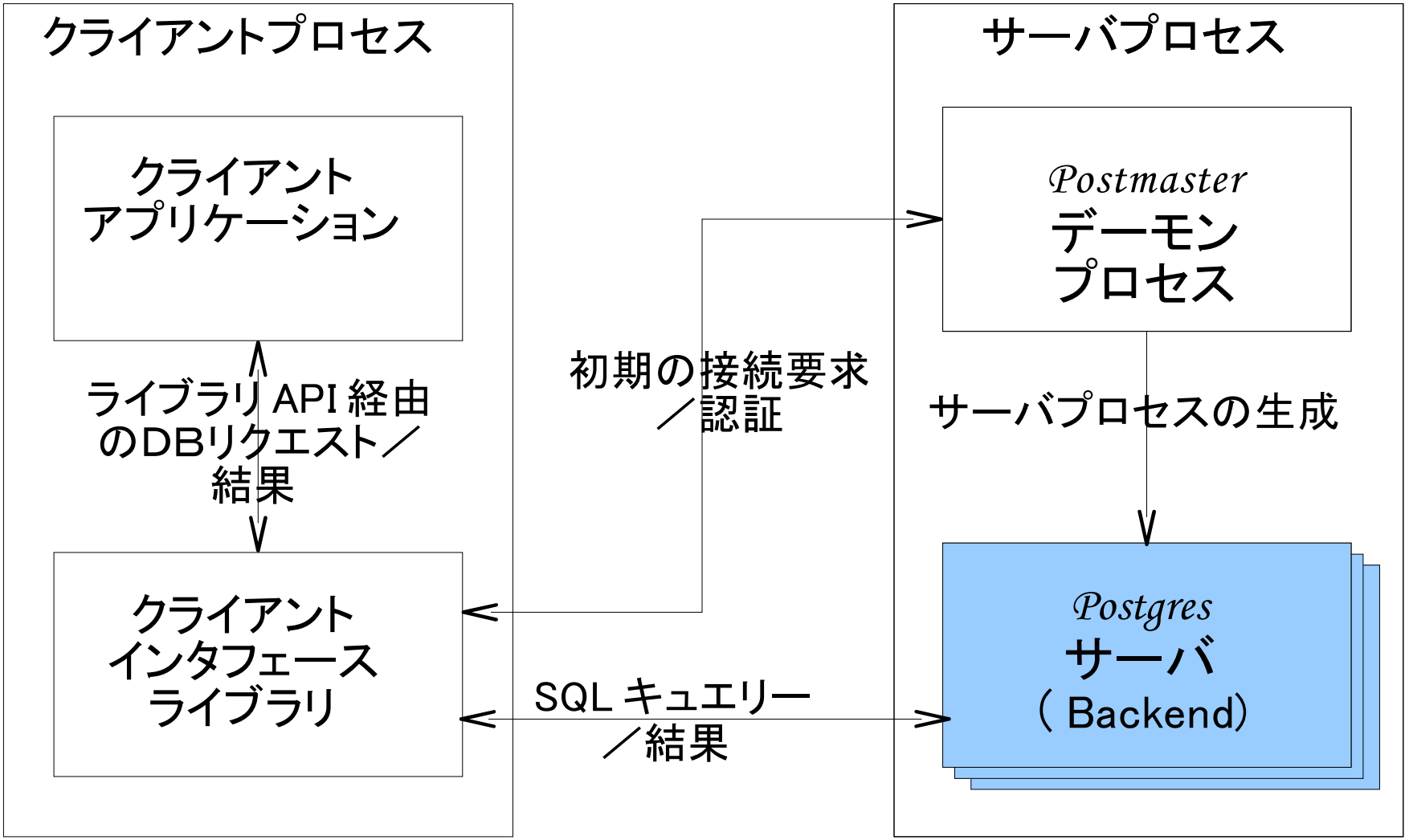
PostgreSQL におけるトランザクション処理

Tom Lane 著
岡田 敏 訳

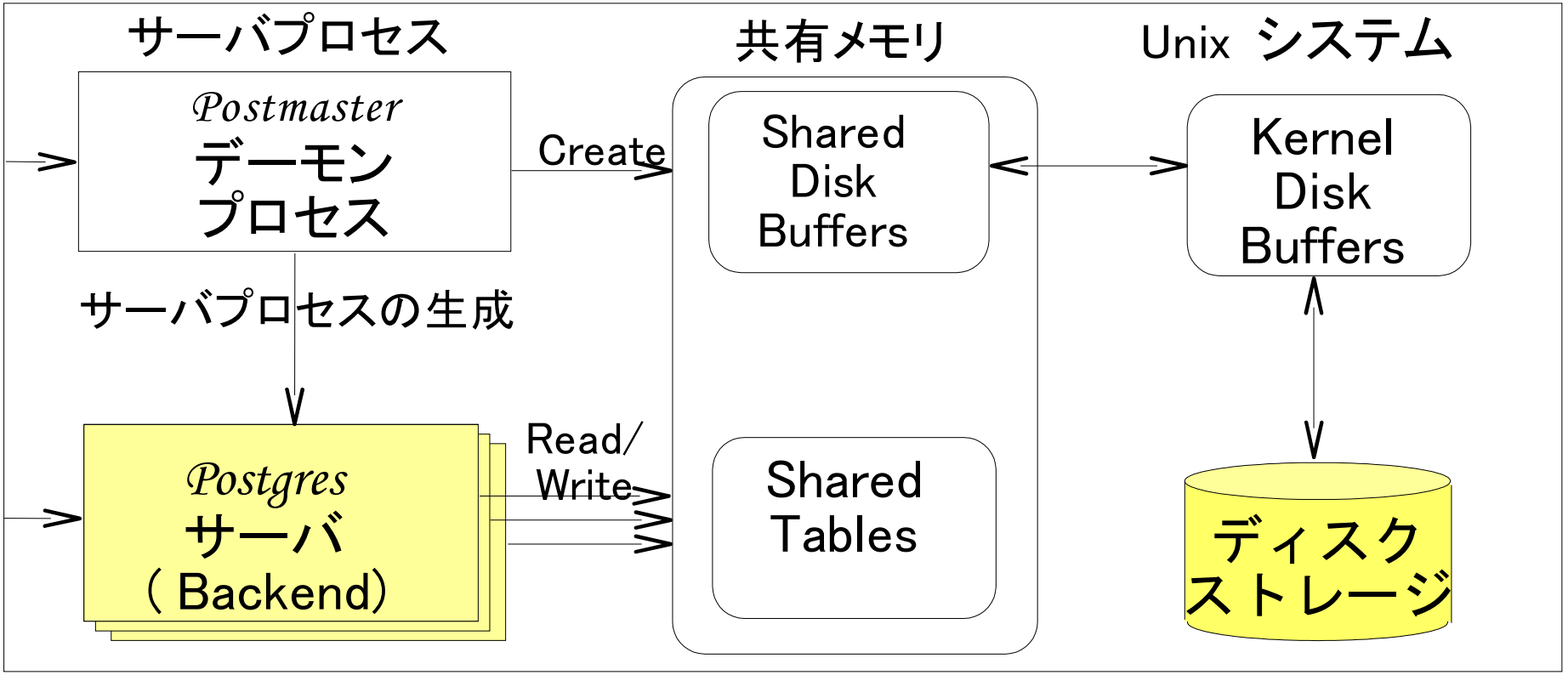
概要

- はじめに
  - トランザクションとは何か？
- ユーザの視点
  - マルチバージョン同時実行制御
- 実装
  - タプル可視性
  - ストレージマネージメント
  - ロック

PostgreSQL システム概観



PostgreSQL システム概観



- データベースファイルは共有バッファプールを介してアクセスされる。
  - そのため、二つのバックエンドがファイルの矛盾したビューを参照することはありえない。
- Unix のカーネルは通常、付加的なバッファを提供している。

トランザクションとは？

- 定義:トランザクションとはSQLコマンドの集まりであり、SQLコマンドの結果は、トランザクションがコミットした時、それらの操作が一まとまりとしてシステムの他の部分に見えるように（可視化される）なり、アボートした時には全く何事もなかったかのように見える。
- トランザクションは原子性（atomic）、整合性（consistent）、隔離性（isolated）、耐久性（durable）が求められる。
  - Postgresは分散トランザクションはサポートしていない。そのためトランザクションの全コマンドは1バックエンドで処理される。
  - 入れ子のトランザクションも現時点では扱わない

ACID テスト:
atomic, consistent, isolated,durable

- 原子性:トランザクションの結果は完全に参照されるものか、もしくは他のトランザクションの範囲内のものは参照されないかである。(トランザクションは自分自身に対する原子性を表す必要はない)
- 整合性:システム定義の一貫性制約は、トランザクションの結果に強要される。(ここでは制約のチェックの議論は行なわない)
- 隔離性:トランザクションは同時に走行しているトランザクションの振舞いから影響を与えられることはない。
  - 類義語:直列化可能。同時に実行したトランザクションのセットの結果が、ある一定の順序に従ってトランザクションを走らせた場合の結果と同じならば、その振舞いが直列化可能であるという。
- 耐久性:一度トランザクションがコミットしたら、その結果はそのあと発生する障害に関係なく、失われてはならない。

多数の更新が原子的であるには？

- タプルの挿入／削除／更新は、トランザクション N が行ったものとしてマークされる。同時実行中の複数のバックエンドはトランザクション N がまだ、コミットしていないことを知っているため、変更を無視する。トランザクションがコミットしたとき、これら全ての変更は直ちに可視化される。

- 制御ファイル pg_log はトランザクション ID 毎に 2 つの状態ビットを持っている。状態とは実行中 (in progress), コミット、アボートである。コミットを表す値へ 2 ビットを設定することは、トランザクションがコミット状態であるということをマークする原子的なアクションである。

- アボートするトランザクションは通常、pg_log 状態をアボートへセットする。しかし、プロセスがその作業を行なわないでクラッシュしても、安全である。次の機会に、いくつかのバックエンドがトランザクションの状態を調査し、″in progress″ とマークされているにもかかわらず、対応するバックエンドが実行中でないものを発見した場合には壊れているものと推論する。そして、pg_log エントリをアボートに更新する。アボート中は任意のテーブルデータファイルの更新を必要とすることはない。

システムクラッシュでも本当に原子的で、耐久的であるのか？

それは、カーネルやハードディスクがどの程度信頼できるかによる。

- ディスクページへの書き出しが原子的なアクションである場合に限り、Postgresトランザクションの原子性が保証される。1ページが物理セクタであるならば、通常のハードドライブは原子的なアクションが可能である。しかしながら、ほとんどの場合、ディスクページ8Kなどに設定しているため、原子的なアクションが絶対的なものであるとは言い切れない。

- pg_log は二つのビットを持つだけであり、トランザクションの状態を示すそれらのビットは同じセクタに存在するため、安全である。

- データページを含むタプルを移動する場合、例えば電源障害でページの書き出しが途中になってしまったときなど、データが壊れてしまう恐れがある（おそらく、コンポーネントセクタの一部のみが書き出されている状態であろう）。これが、ページサイズを小さく保っている理由である。　... そして、サーバにUPSを付けること！

Unix カーネルでの作業は何かの犠牲を伴う

pg_log を書き出す前に、トランザクションのデータページの変更をディスクに書き出すことは重要なことである。もし、ディスクへの書き出しが間違った順序で発生した時、電源障害は我々に pg_log にコミットとマークされたトランザクションを残すが、そのデータすべてがディスクに反映されているわけではない。-- これはアトミックテストに適合しない。

- Unix カーネルは、fsync(2) を用いることで、正しい順序で書き出しを行なうことができる。しかし、多くのファイルに対して fsync を行なうことによるパフォーマンスペナルティは非常に高い。
- 我々は多くの fsync() を多数、実行しないで良い方法を考えている。が、それは別の話とする。

ユーザの視点：
マルチバージョン同時実行制御

PostgreSQL アプリケーションは同時実行のトランザクションについて以下のような振舞いをみることとなる。

- それぞれのトランザクションは、その開始時点におけるスナップショット（データベースのバージョン）を参照する。他のトランザクションが今現在、走行しているかどうかは気にしない。
- 読み手（reader）は書き手（writer）をブロックせず、書き手は読み手をブロックしない。
- 複数の書き手が同一の行を更新しようとした時のみ互いにブロックを行なう。

同時実行の更新はトリッキーである

以下のような例を考えよう。
トランザクション A は

```
UPDATE foo SET x = x + 1 WHERE rowid = 42
```

を行い、これをコミットする前に、トランザクション B が実行され、B は A と同じ行に対し、同じ処理を行なう。

- 明らかに、B は A がコミットしたか否かを知るために待機しなくてはならない。
- A がアボートした時、B は既存の x の値を使って、処理を進めることができる。
- しかし、A がコミットした時、その時の処理はどうだろうか？
  - 古い x の値を使うことは、明らかに受け入れられない結果である。というのは、両方のトランザクションをコミットした時、2ではなく1だけ加算された状態となるからである。
  - しかし、B が x の新しい値をインクリメントすることが許されるならば、B が実行を開始してからコミットされたデータをBは読み込んでいる。これはトランザクションの隔離性の原則に違反するものである。

Read comitted vs. serializable transaction レベル

PostgreSQL は同時更新問題に対して二つの解を提供している（ISO SQL 標準の中で定義されている4つの分離性レベルによる）

Read committed レベル： 入力値として新しいタプルを使用することをBに許可する（新しいタプルがキュエリの WHERE 節を満足していることを確実にするために、後でチェックを行なう）。ゆえに、B は A の結果のタプルを参照することを許される。

Serializable レベル：“not serializable” エラーとしてアボートする。クライアントアプリケーションはトランザクション B 全てをやり直さなくてはならない。B は厳格な直列化を行なうためのルールのもとで新しくなった x の値に対しては参照することが許される。

- Serializable レベルは論理的に明確なものであるが、アプリケーションにより多くのコードが必要となる。そのため、デフォルトの設定では、read-committed レベルで走行する。そして、その結果は通常要求される振舞いを提供する。

- いずれにしても、純粋な SELECT トランザクションは、トランザクションが開始する前にコミットされたデータを参照するだけである。そのトランザクションは関心のある更新および削除結果を見ることになる。

どのようにして実装されているか

「そのタプルをあなたは参照できるか？」

最も基本的な実装コンセプトは”タプル可視性（tuple visibility）” である。すなわち、どのテーブル行のどのバージョンが、どのトランザクションから参照されるかということである。

参照可能でないタプルを無視することは、トランザクションが原子的に見えるようにするためのキーである。

定義：タプルはテーブル中に具体的に格納されたオブジェクトであり、論理的なテーブルに存在する行の1バージョンを表している。1つの行は同時に複数のバージョンとして存在していることがある。

非オーバーライトストレージ管理

我々は、すべての行の複数のバージョンを保持しなくてはならない。タプルは、削除としてコミットされた後、十分に時間がたち、そのタプルを参照するアクティブなトランザクションがなくなったときにはじめて移動（削除）できる。

PostgreSQL は、常に”non overwriting”ストレージ管理を実践してきた。そのため、更新されたタプルはテーブルに追加され、古いバージョンのタプルは十分に時間がたった後、取り除かれる。

現在、削除されて時間のたったタプルの除去は VACUUM というメンテナンス用のコマンドを用いている。このコマンドは定期的に発行されなくてはならない。我々は、削除されたタプルを実行中にリサイクルすることによって、VACUUM の必要性を少なくする方法を検討している。

タプルごとのステータス情報

タプルヘッダは以下のものを含んでいる。

- xmin： 挿入している（挿入した）トランザクションのトランザクション ID
- xmax: 置き換え／削除をしているトランザクションのトランザクション ID （初期値は NULL ）
- forward link ：存在していれば、同じ論理ローの次の（より新しい）バージョンへのリンク

基本的な考え方：xmin が有効（Valid）でかつ xmax がそうでない時、タプルは可視的である。”有効（Valid）” とは、”コミットされているかもしくはトランザクションの途中”である。

我々が、削除することよりも、更新することを計画するなら、我々は第一にテーブルへ行の新しいバージョンを追加し、その後、xmax と古いタプル中の forward link をセットする。 forward link はカレントのアップデーターによって必要とされるものである（リーダーが必要としているものではない）。

pg_log への繰返しの参照を避けるため、xmix と xmax に対するいくつかのステータスビットが存在しており、” known commited” もしくは” known aborted” を示す。これらは参照されたトランザクションがコミット／アボートした後、xmin もしくは xmax を検査する最初のバックエンドによってセットされる。

アクティブトランザクションを隔離する”Snapshots”フィルタ

トランザクションBが走行している間にトランザクションAがコミットした場合、我々はBがAの更新結果を途中から参照しはじめるようなことをやって欲しくない。

- それゆえ、我々はトランザクション開始時に、どのトランザクションが他のバックエンドによって走行しているいるかを示すリストを作成する。(簡単な共有メモリ通信がここでの要点である。というのは、我々は共有メモリテーブル中をまさに参照しており、その中には、それぞれのバックエンドが現在のトランザクション番号を記録している。)

- これらのトランザクションIDは決してカレントトランザクションによって有効(valid)とはみなされないであろう。それはたとえ、pg_logの中や行のステータスビット中でコミットされたと示されてもである。

- カレントトランザクションよりもより大きいIDを持ったトランザクションでも有効とはみなされないだろう。

- これらのルールはカレントトランザクションのスタートの後のトランザクションコミットをコミットされたとみなさないということを確実にさせる。

- 有効性(validity)は人それぞれである。

テーブルレベルロック：いくつか理由のためにもち続けている

MVCC の下では読み手（readers）と書き手（writers）は互いに妨害しないにもかかわらず、我々はテーブルレベルロックを必要としている。

これは、主に読み手（readers）もしくは書き手（writers）以外によって変更や削除がなされることから、テーブル全体を守るために存在している。

我々は、アプリケーションの使用のためにさまざまなロックレベルも提供している（主に、同時実行制御に伝統的なロックベースアプローチを採るポーティングアプリケーションのためである。

ロックのタイプ

ロックタイプ	取得される命令	何と衝突するか
1 AccessSharedLock	SELECT	7
2 RowSharedLock	SELECT FOR UPDATE	6, 7
3 RowExclusiveLock	UPDATE, INSERT, DELETE	4, 5, 6, 7
4 ShareLock	CREATE INDEX	3, 5, 6, 7
5 ShareRowExclusiveLock		3, 4, 5, 6, 7
6 ExclusiveLock		2, 3, 4, 5, 6, 7
7 AccessExclusiveLock	DROP TABLE, ALTER TABLE, VACUUM	all

全てのロックタイプはLOCK TABLEコマンドにより取得される。

ロックはトランザクションの終了時点まで保持される。あなたはロックを取得することは可能だが、トランザクションの終了によるものを除いて、ロックを解放することはできない。

ロックの実装

ロックは共有メモリハッシュテーブル中に記録される。ハッシュテーブルはロックを取得されたオブジェクトの ID と種類をキーとしている。個々のアイテムは、そのオブジェクトに関して保持済みおよび、取得待ちとなっているロックの種類と数を示している。既にロックがとられているオブジェクトに対して、衝突する関係のロックを取得する場合、取得しようとする者（プロセス）は、待たなくてはならない。

”待ち”はプロセス毎の IPC セマフォを待つことによって、実現される。他のプロセスがリリースすると、ロックを取得したいプロセスにシグナルが送付される。我々は同時実行するバックエンド毎にただ一つのセマフォを必要とするだけである。潜在的にロック可能なオブジェクト毎にセマフォを必要としているわけではない。

デッドロック検出

二つのトランザクションが異なった順序で、競合するロックを獲得しようとしたとき、デッドロックが発生する可能性がある。

ロックを取得しようとするプロセスが、要求されたロックを取得することなしに1秒以上に渡り、スリープ状態であったとき、デッドロックチェックアルゴリズムが走行する。デッドロックチェックアルゴリズムはロックハッシュテーブル中でロックの依存関係が循環していないかを探索する。循環したの依存関係が見つかったとき、ロックを取得することは不可能であり、エラーのリポートをだし、ロック取得をあきらめる。循環した依存関係が見つからなかった場合、スリープ状態に戻り、ロックの許可が出るまで待つ（もしくは、クライアントアプリケーションがあきらめ、トランザクションキャンセルのリクエストを出すまで）。

- デッドロックチェックアルゴリズムが走行する前の待ち時間は個々のサーバのワークロード中で典型的なトランザクションの時間に合わせる形で設定できる。この方法では、不必要なデッドロックチェックは滅多に実行されないが、実際のデッドロックは適度に早く検出される。

Short-term ロック

Short-term ロックは、これまで説明したロックハッシュテーブルのような共有メモリ中のデータ構造を保護する。

これらのロックは実行、かつ / もしくは共有されたアイテムを更新するのに十分な間だけ取得されるべきである。――特に、共有アイテムを保持している間、バックエンドは決して妨害すべきでない。

実装： spin locks はプラットフォームに特化した原子的な test-and-set 命令をベースにしている。ロックの競合がないという典型的なケースでは、非常に高速に実行を終了するというロックコードを許容する。もし、test-and-set が失敗したときには、我々は短期間（select(2) を使用して）スリープする。そして、再試行する。デッドロック検知といった機能はないが、何回も失敗した場合には処理をあきらめ、エラーを報告する。

まとめ

PostgreSQL は、ハードウェアや Unix カーネルを基礎とする振舞いを前提とにはしているが、トランザクションに対して真の ACID 特性を提供する。

マルチバージョン同時実行制御はテーブルの同時読み込み、書き出しを許容し、同じ行に対する同時更新のみをブロックする。

MVCC はバークレイ POSTQUEL プロジェクトから受け継いだ非上書きの(non-overwriting)ストレージ管理を用いているために実用的である。伝統的な行上書き(row-overwriting)ストレージ管理の場合には難しいものとなるだろう。