D-016

行ロックによる SQLite の同時実行性を向上させる手法の提案
Concurrency improvement of SQLite by row lock

片山 大河† 金松 基孝†
Taiga Katayama Mototaka Kanematsu

1. はじめに

SQLite[1]はリレーショナルデータベース管理システムの一つであり、多くの組込みシステムで使用されている。この SQLite は複数ユーザの更新による一貫性を保つためにテーブルロックを採用している。複数ユーザが同一テーブル内の異なる行を更新する場合、一方のトランザクション終了後でないともう一方の更新クエリはロックエラーが発生して実行できない。ロックエラーとなったクエリは再実行しなければならず、アプリケーションの速度性能に影響する。

そこで SQLite のロック範囲をテーブルロックから行ロックに緩和して同時実行性を向上させる手法を提案する。本手法では SQLite 内部を修正して行ロックを実現するために、ユーザごとに行ロック情報とトランザクション中の変更データを保持できるようにした。また、SQLite と使用方法を同じにするため API には変更を加えていない。本稿では SQLite の仕組み、提案手法の設計および実験について説明する。

2. SQLite のデータ操作の仕組み

本節では、SQLite が一貫性を実現するための排他制御の仕組みとデータ管理方法について説明する。

2.1 クエリ実行における排他制御の仕様

SQLite は、クエリの並列実行制御とデータの同時アクセス制御によってクエリの実行を排他制御している。

2.1.1 クエリの並列実行制御

アプリケーションがデータベース（DB）にアクセスするためには、まずは DB ハンドルを作成する。そのハンドルを用いてクエリを実行させる。複数の DB ハンドル（ユーザ）によって並列にクエリ処理要求があった場合、DB が異なるハンドルに対するクエリは並列に、同一の場合は逐次的に処理される（図 2）。

2.1.2 データの同時アクセス制御

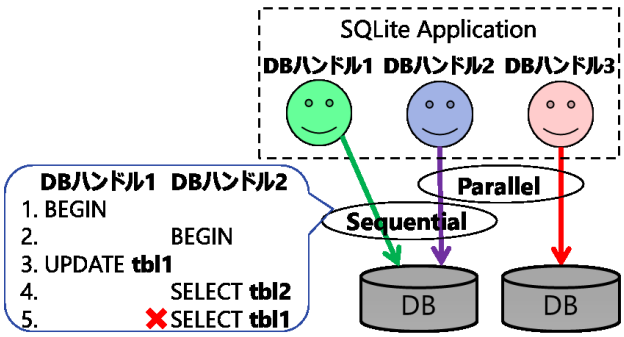

図 2 SQLite の排他制御

†株式会社東芝 ソフトウェア技術センター

SQLite のトランザクション分離レベルは Serializable である。WRITE トランザクションを行えるのは 1 人に限られる。WRITE トランザクションがある場合、他人は参照クエリしか実行できない。

さらに、SQLite ではテーブルロックが実装されている。SQLite における一番粒度が細かいデータロックの単位で、同一プロセス内のユーザ間で排他する。ちなみに、この機能を使うためには SQLite Shared-Cache Mode を有効にする必要がある。トランザクション中にあるテーブルに変更があった場合、他人はそのテーブルに対して参照することができない（図 2）。

2.2 レコードの管理構造

データ構造は Btree というモジュールで管理されており、レコードの追加・探索・削除や DB ファイルの読み書きといった Btree に対する操作関数が SQLite に実装されている。このモジュールはキャッシュ機構も備えていて、DB ファイルとの入出力のための高速化が図られている。また、Btree に蓄積した未コミットデータがキャッシュ溢れした場合、一時ファイルへ退避して消費メモリを抑制する機能も備えている。

図 1 のように、このモジュールのインスタンス中に DB ハンドル間で共有する領域があり、そこにテーブルおよびインデックスを格納している。エントリは Key と Value で構成され、Key の値で整列されている。すべてのレコードには、行を一意に識別する rowid が付与されていて、テーブルの場合はこれが Key となる。rowid はレコードをテーブルに登録する際に SQLite が自動で発行する。インデックスの場合はインデックス列の値が Key となる。

2.3 クエリの実行とカーソル機構

SQLite はクエリをコンパイルすると実行エンジン VDBE（Virtual Database Engine）が実行するための実行計画 VDBE コードを作成し、VDBE が VDBE コードに従って処理を行う。その実行計画に応じて Btree に対する操作関数が実行される。

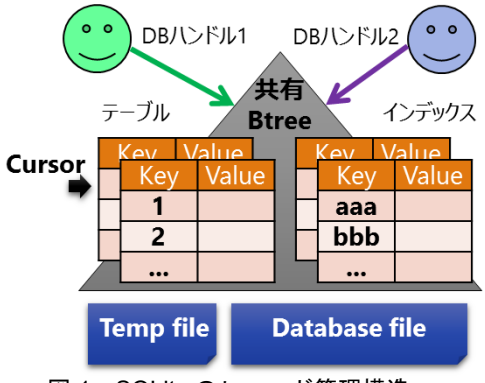

図 1 SQLite のレコード管理構造

Btree に対するレコード操作はカーソルを介して行うように実装されている。先頭へ移動、末尾へ移動、ひとつ次あるいは前の行へ移動、キーをもとに探索し移動、カーソルが指す行の参照や削除、レコード追加などの関数がある。

テーブルはページ番号（テーブル番号）という整数で識別されていて、まずアクセスしたいテーブルに対するカーソルを作成する。その後、クエリの実行計画に応じて操作関数を呼ぶ。

3. 行ロック機能を導入する手法の提案

データの同時アクセス制御の仕様を変更して、行ロック機能を導入することで同時実行性を向上させる。

今回、同一プロセス内のユーザ間で行ロックが行えるようにした。そのために、複数ユーザが同一の DB にアクセスするために必要な機能 SQLite Shared-Cache Mode を有効化したもとで行ロック機能を実現した。将来的にはプロセス化（異プロセスのユーザ間にも行ロックを適用させること）を見据えて設計を行っている。

提案手法ではトランザクション分離レベルを Read Committed とし、Multi Version Concurrency Control（MVCC）を実装した。これらの実現の仕組みを説明する。

3.1 ソースコード修正の方針

SQLite のバージョンアップに追従できるように、SQLite のソースコードは可能な限り修正箇所を局所化するようにし、新規実装コードは新たなファイルに追加した。方針としては、VDBE が Btree 関数を呼ぶ部分をフックし、行ロック用に今回実装した関数を呼ぶようにした（図 4）。その行ロック用関数は、これから説明する設計に基づいて、Btree 関数を使用するなどして実現している。

3.2 行ロックの仕様と排他制御の緩和

2.1.2 節で述べた SQLite のデータ同時アクセス制御を次のように緩和する。

3.2.1 複数ユーザの WRITE トランザクション

SQLite は更新系クエリの処理開始時に WRITE トランザクション用のロックを取得する。取得したロックはトランザクション終了時に解放される。提案手法では複数ユーザによる同時更新を可能にするために、その処理をコミット時まで遅延させ、コミット処理終了時にそのロックを解放するようにした。

3.2.2 テーブルロックの緩和

SQLite は更新系クエリの処理開始時に変更対象のテーブルのロックを取得する。提案手法ではこのロック機能を撤廃し、レコードの変更・削除時に行ロックを取得するようにした。

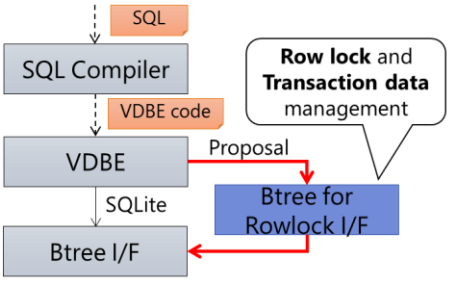

図 4　行ロック用のモジュール

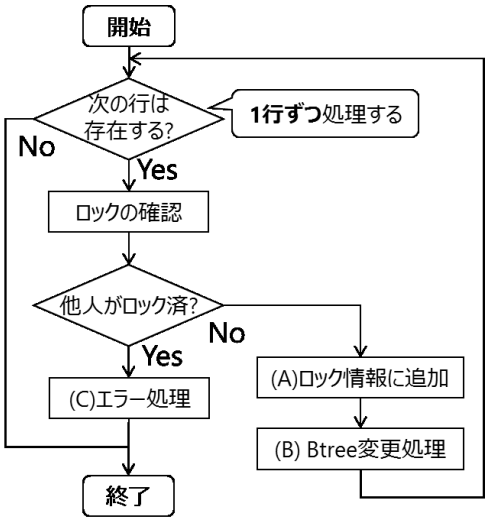

図 5　行ロックの処理の流れ

3.3 行ロック情報の管理

行ロック情報として、トランザクション中に変更があったレコードの rowid、ロックの所有者（ユーザ識別子）、プロセス番号とした。これらの情報は、プロセス化を見据えて、プロセス間で共有できる領域（Windows ではファイルマッピング）に格納するようにした。

図 5 のように、(A)ロック情報の追加および(B)Btree 変更処理が 1 行ずつ繰り返し行われる。(C)のエラー処理では、クエリ実施直前の状態に戻す必要がある。つまり、一つのクエリで複数行を変更あるいは削除している途中で行ロックエラーが発生した場合、そのクエリの処理の過程で行った変更をなかったことにしなければならない。(B)の操作に対する処理の取り消しは、SQLite に備わっているステートメントロールバック機能により行える。しかし(A)の操作に対しても同様にクエリ実行前の状態に戻す必要がある。そこで、行ロック情報の追加履歴を保存するようにした。エラー発生時には、クエリで変更されたレコードを元に戻し、この情報に基づいてクエリ処理開始後に取得された行ロックをアンロックする。

3.4 トランザクションデータ管理

MVCC を実現するためのデータ管理の仕組みについて説

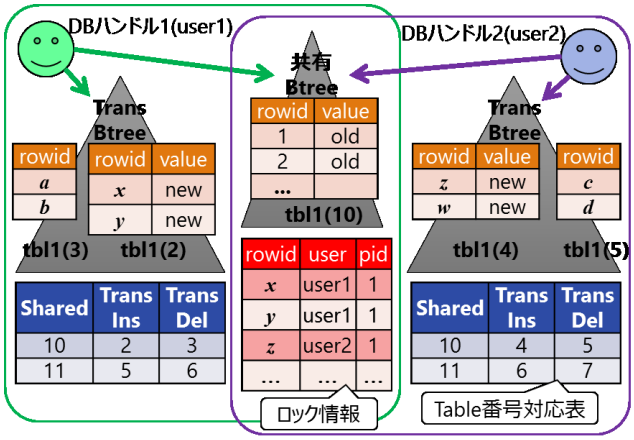

図 3　トランザクションデータ管理

明する。行ロックを導入しても ACID 特性の Isolation（独立性）を維持するために、各ユーザが自身の管理下に蓄積しておき、COMMIT 時に共有化することで、トランザクション中の変更データは他人から見えなくする。

3.4.1 データ構造

トランザクションを管理するためのデータ構造として Btree を利用した。従来の Btree に加えて自身のみアクセス可能な Btree をユーザごとに別途用意する（図 3）。以降では新たな Btree のことを TransBtree と呼び、従来からある Btree を共有 Btree と呼ぶ。

行ロック機能の実現のために、共有 Btree に対する操作の代わりに、トランザクション変更データに対して実施することになる。SQLite の Btree はキャッシュ機能やファイル退避機能（2.2 節）も利用でき、新たにデータ構造を定義するより、既存の仕組みを流用した方が良いと考えた。

3.4.2 テーブル管理

TransBtree 上にデータを蓄積するために、TransBtree 上にもテーブルを作成する。共有 Btree 上のテーブル 1 つに対して、TransBtree 上に 2 つテーブルを作成する。ひとつはレコードの追加および更新用に使用する。共有 Btree 上のテーブルあるいはインデックスと同様に、Key と Value を格納する。更新の場合でも更新されない列の値も含めて格納する。この理由は、共有 Btree へ更新処理を行う際、レコード更新のための Btree 関数の I/F が、未変更の列を含む Value 値を必要とするからである。

もう一つのテーブルは、削除したレコードを管理するために使用する。ここには Key として rowid を格納する。削除用テーブルに rowid を追加することは共有 Btree 上のレコードが無効であることを意味している。

テーブルはテーブル作成時に発行されるテーブル番号で識別されていて、アクセス時にパラメータとして Btree 関数に与えられる。VDBE が呼ぶ Btree 関数に与えられるテーブル番号は共有 Btree に対するものであるため、そのままでは TransBtree に対しては使用できない。そこで、共有 Btree 上と TransBtree 上とのテーブル番号の対応表を用意して、変換できるようにした（図 3）。

3.4.3 レコードの登録・削除・更新

表 1 に示すように、3 つのテーブルへの格納状況によってレコードの状態を管理する。例えば、3 つのテーブルすべてに同一の rowid を持つレコードが存在していた場合、そのレコードはトランザクション中に更新されたことを意味する。

表 1　レコードの状態

状態	Shared	TransIns	TransDel	意味
S0	×	×	×	初期状態：レコードなし
S1	×	○	×	追加されたレコード
S2	○	×	×	コミットされたレコード
S3	○	○	○	更新されたレコード
S4	○	×	○	削除されたレコード

クエリ処理を行う際の状態遷移図を図 6 に示す。この状態遷移に基づいて、各テーブル上の対象レコードに対して登録・削除・更新処理を行う。例えば、コミット済みのレコードは共有 Btree 上のテーブルにあり（状態 S2）、そのレコードに対して UPDATE を行った場合、状態は S3 となる。この状態遷移の処理では、TransBtree 上の追加用テーブルに更新後のレコードを登録するとともに、削除用テーブルに rowid を追加する。

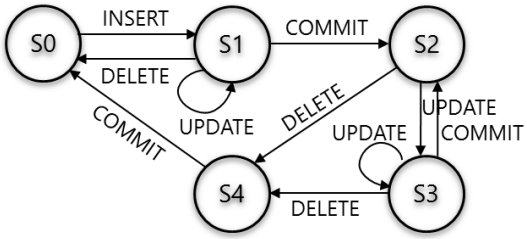

図 6　レコードの状態遷移

このように 3 つのテーブルを使ってトランザクションデータを管理する。テーブルにアクセスするためにはカーソルを作成する必要がある。共有 Btree 上のテーブルに対するカーソル作成時に、TransBtree 上の追加用テーブルおよび削除用テーブルに対するカーソルも作成することになる。

3.4.4 rowid の発行

INSERT 処理で行う rowid の発行方法に改造が必要である。SQLite ではテーブル内で最大の rowid を見つけ、その値+1 を追加レコード用の rowid として発行する。提案手法では各ユーザ持つすべての TransBtree 上の追加テーブルを参照して最大の rowid 値を探さなければならない。rowid 発行の度にその探索を行うことは無駄である。そこで、rowid をキャッシュしておくようにした。つまり初回は共有 Btree のみを探索して、その値をユーザ間で共有できる場所に覚えておく。以降、そのキャッシュ値をもとに rowid を発行するとともにキャッシュ値を更新する。

3.4.5 レコードの参照

VDBE は、レコードが Key で整列されている前提でカーソル移動とレコードのフェッチを行う。提案手法では、3 つのカーソルを使うため、カーソルが指すレコードの状態と Key 値の大小によって、移動すべきカーソルとフェッチするレコードを選択する。

例えば、カーソルを初期位置に移動させ、そのレコードをフェッチしたいケースを説明する。まず、3 つのカーソルを初期位置に移動させる。次に共有 Btree に対するカーソルが指すレコードの有効性を判定する。削除用テーブルに同一の rowid が存在するかどうかで判定できる。これを有効なレコードが存在するまで、共有 Btree 上のテーブルに対するカーソルを一つずつ進めて存在判定を繰り返す。そうすると共有 Btree と TransBtree 上のテーブルに対する 2 つのカーソルが有効なレコードを指すことになる。これらのカーソルが指す Key 値を比較して小さい方を選択する。これがフェッチするレコードである。

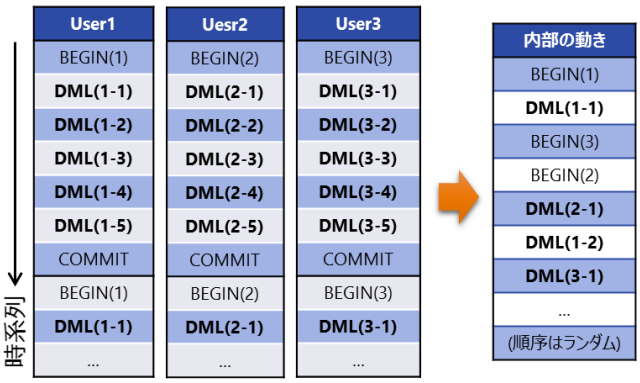

図 7　実験でのユーザとクエリの関係

4. 実験

行ロック機能による同時実行性の向上効果を確認するための実験を行った。

4.1 内容

ユーザがトランザクション処理を 10 回繰り返し、すべての処理が完了するまでの時間を SQLite と提案手法で比較した（図 7）。もし途中でロックによるエラーが発生した場合、該当のクエリをリトライして成功するまで繰り返す。SQLite および提案手法の内部ではクエリの実行は逐次処理される。

4.1.1 実験環境とデータ

実験環境は表 2 のとおりである。次のスキーマのテーブルを作成し、レコードを 1000 件格納した。

- CREATE TABLE tbl (col1 INTEGER, col2 INTEGER)
- INSERT INTO tbl VALUES(*n*, *n**100) ：*n*=0…999

表 2　実験環境

OS	Windows8.1 64bit
CPU	Core i7-4500U 1.80GHz
Memory	8GB RAM
Storage	215GB SSD
SQLite	3.21.0

4.1.2 実験パラメータ

ユーザ数は 1 から 3 の 3 パターン、トランザクション処理内容は次に挙げる 5 パターンの DML を測定した。

（A）更新行が競合しない UPDATE
次のように各ユーザが異なる行を UPDATE する。

```
DML1-1～1-5:
  UPDATE tbl SET col2 = col2+1 WHERE col1%3=0
DML2-1～2-5:
  UPDATE tbl SET col2 = col2+1 WHERE col1%3=1
DML3-1～3-5:
  UPDATE tbl SET col2 = col2+1 WHERE col1%3=2
```

（B）更新行が競合する UPDATE
各ユーザは異なる行を更新した後、最後に共通の行を更新する UPDATE クエリを実行する。つまり、（1）の全クエリの WHERE 条件に OR col1=999 を追加したクエリを実行する。

（C）ランダム行の UPDATE
（1）の WHERE 条件を「col1 = RANDOM() % 1000」としたクエリを実行する。

（D）DELETE
次のように各ユーザが異なる行を DELETE する。WHERE 条件の「col<?」を制御して、毎回のクエリで DELETE 対象が 6 行になるようにした。

```
DML1-1～1-5:
  DELETE FROM tbl WHERE col1%3=0 AND col1<?
DML2-1～2-5:
  DELETE FROM tbl WHERE col1%3=1 AND col1<?
DML3-1～3-5:
  DELETE FROM tbl WHERE col1%3=2 AND col1<?
```

（E）INSERT
各ユーザは初期状態が空のテーブルに INSERT する。

4.2 結果と考察

表 3 に実験結果を示す。ユーザ数 1 のケースでは、SQLite より性能が悪い。これは提案手法では行ロック機能を実装したことによるオーバヘッドだと考えられる。SQLite に比べて、変更データを一度 TransBtree に格納する処理が余分なため、変更行数が多い A および B は C よりオーバヘッドが大きく、今回の実験では 2 倍程度であった。

B のユーザ数 2 のケースに着目してみると、総リトライ数が同等である。SQLite ではレコードにアクセスする前に WRITE トランザクション開始のためのロックエラーが返る。一方で、提案手法では実際にレコードにアクセスして、更新データを TransBtree に入れている途中でエラーになる。そのため、いち早くロックエラーと判定される SQLite の方が高速だったと考えられる。

それ以外の複数ユーザによる UPDATE では、ロックエラーによるリトライが必要な SQLite よりも高速であり、行ロックによる同時実行性向上が確認できた。また、DELETE と INSERT のケースにおいても、ユーザ数 1 の場合では約 1 割オーバヘッドがかかるものの、複数ユーザでは SQLite よりも高速に実行できた。

このように SQLite はユーザ数が多いほどリトライ数が多く指数関数的に性能が劣化するのに対して、提案手法ではそれを抑制できることが実験で確認できた。

表 3　実験結果

処理	ユーザ数	時間（ミリ秒）		総リトライ数	
		提案手法	SQLite	提案手法	SQLite
(A)	1	55	29	0	0
	2	81	108	0	72
	3	100	720	0	2073
(B)	1	57	28	0	0
	2	150	86	60	58
	3	349	692	219	1902
(C)	1	26	20	0	0
	2	43	95	0	69
	3	74	302	0	341
(D)	1	29	26	0	0
	2	35	104	0	77
	3	44	206	0	389
(E)	1	28	24	0	0
	2	55	910	0	60
	3	77	476	0	997

5. おわりに

SQLite に行ロック機能を導入する方法を提案し、その実現方法を説明した。実験により SQLite よりも高速であることが確認でき、SQLite の弱点である更新の同時実行性問題を改善できたと考えている。

これにより開発者は軽量で手軽さを維持しつつ同時実行性が向上した SQLite を利用できるようになった。今後、単一ユーザでのオーバヘッド低減策を検討する。ソースコードは[2]にて公開予定である。

参考文献

[1] SQLite, https://www.sqlite.org/.
[2] ソースコード, https://github.com/t-kataym/sqlite_rowlock/.