2C-01

SQLite の行ロック機能導入におけるマルチプロセス対応

片山 大河† 金松 基孝†
株式会社 東芝 ソフトウェア技術センター†

1. はじめに

リレーショナルデータベース管理システムの一つである SQLite[1]は、ライブラリ型のデータベース（DB）で利用しやすく、多くの組込みシステムで使用されている。ソースコードがパブリックドメインライセンスで公開されており、誰でも自由に利用や改変が可能である。

しかし、同時実行性の問題が頻繁に挙がる。複数ユーザから同時に処理を要求されたとき、排他制御によりクエリが一方しか処理されないことにより、並列処理性能が求められるシステムでは問題となることがある。

2. SQLite の同時実行性の問題

SQLite の同時実行性が低下している原因は二つあり、一つは書き込み処理を行うとトランザクション終了までテーブルロックを取得し続けることにある(図 1(A))。この原因を解決するために、論文[2]では SQLite のトランザクション分離レベルを Serializable から Read Committed に変更し、排他制御を緩和して同時実行性を向上させる手法を提案している。この手法は、Multi-Version Concurrency Control を実現してユーザごとにトランザクション中の変更データを管理する機能とレコード単位でロック情報を管理する機能を追加して、複数ユーザによる同一テーブルの変更を可能としている(図 1(B))。

もう一つの原因は、SQLite がプロセス内部で DB データを共有しており、クエリ処理単位で DB のロックを必要とするため、複数スレッドによるクエリは逐次処理されることである。なお、トランザクション内であってもクエリ処理が完了すればロックが外される。この問題はプロセスが異なれば回避できる。しかし、論文[2]の手法は一つのプロセスから複数ユーザによって DB にアクセスすることを前提としているため解決できない。そこで、さらに同時実行性を向上させるために、行ロック機能をマルチプロセス環境下でも動作できるようにした(図 1(C))。

3. マルチプロセス対応

マルチプロセスに対応するためにはまずプロセス間で行ロック管理情報のデータ共有化が必要である。それに加えて、2章で述べたようにシングルプロセスとマルチプロセスとではクエリ同時実行制御の挙動が違うため、より厳格なデータの整合性管理および排他制御が必要になる。以降で修正内容を説明する。

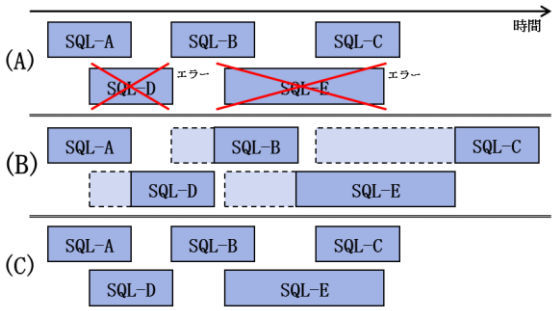

図1 同時実行性によるクエリ処理の違い
(A)SQLite, (B)行ロック機能（スレッド），
(C)行ロック機能（プロセス）

3.1. プロセス間でデータの共有

プロセス間で共有するデータは、行ロック機能導入のために追加した情報、つまりロック情報およびキャッシュする rowid 値である。これらを Memory Mapped File 上に格納するようにした。この共有領域は揮発的なデータとして扱ってよく、ファイルに同期する必要はない。アクセスしていた最後のプロセスが SQLite の使用終了時に削除するようにした。

3.2. 共有 rowid 値の同時使用への対処

rowid は一意に識別可能な各レコードに付与される整数値である。論文[2]に記載されているように、テーブル内の最大 rowid をキャッシュして、レコード追加時に新 rowid 値の発行を高速化するために利用している。

従来のシングルプロセス環境下では、クエリは逐次処理されるのでキャッシュ値の同時参照に気をつけなくてよかった。しかし、マルチプロセス環境下では、キャッシュ値を参照してから行ロックを取得するまでの間に時間差があり、同時に複数プロセスが同じ rowid 値を発行する可能性がある。それを防ぐために、同じ rowid が発行されたことにより行ロックを取得できなかった場合、自動的にクエリを再実行することで新 rowid を再発行するようにした。

3.3. 強制終了対策

マルチプロセスの場合、一つのプロセスが強

Multiprocess support of row lock feature on SQLite
†Taiga Katayama, Mototaka Kanematsu
†Toshiba Corporation, Corporate Software Engineering & Technology Center

制終了した場合、そのプロセスが共有領域に保持していたロック情報が残ってしまい、共有領域が破棄されるまで他のユーザはロックが取得できなくなる。

この問題を解消するためにツールを作成した。プロセス番号をパラメータとして与えてこのツールを実行すると、そのプロセスが保持していた共有領域上のロック情報を開放する。

3.4. 更新情報の取り込み

SQLite はページという単位で DB ファイルからデータを読み書きしている。ページは必要となったときに読み込まれ、メモリ上にキャッシュされる。そのキャッシュはトランザクション終了時にクリアされ、次に参照する際に DB ファイルから読み直される。SQLite では書き込みトランザクションは一人しか開始できないこと、また他人に参照されているテーブルに対して変更をコミットできないため、キャッシュクリアはこのタイミングで問題ない。

しかし、行ロック機能のマルチプロセス対応では、他人の更新をクエリ実行ごとに反映する必要がある。まず、DB ファイルに格納されているバージョン値がメモリ上に持っている値を比較することで変更を検知する。変更があったらキャッシュをクリアして、以降の処理でページを読み込んだときにキャッシュを最新にするようにした。

3.5. ページ間の関係変更への対処

各ページにはページ番号が付与されており、ページ間は親子関係を持つ木構造で管理されている。コミットによって DB ファイルの中身が変更され、ページ間の関係が変わることがある（図 2）。

もし SELECT クエリの処理時間が長いと、実行中に他のプロセスによってページ間関係が変更される可能性がある。その場合、SELECT クエリの処理では正しくページを辿ってレコードにアクセスできなくなる。この問題を回避するために、SELECT 処理が行われているテーブルかを区別できるフラグを導入することでそのテーブルにはコミットできないようにした。SELECT 処理が終了すれば、トランザクション中であっても、他のプロセスはコミットできるようになる。

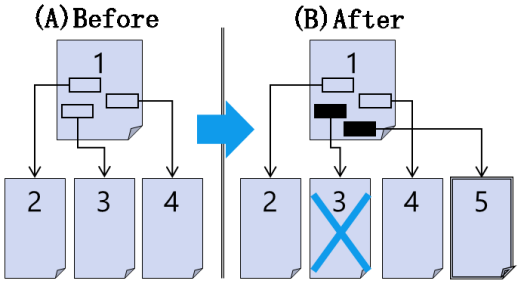

図2 ページ間の関係性変化例

4. 動作確認

プロセス対応した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に動作確認を行った。環境は下記のとおりである。

OS	CentOS7 64bit on VM
CPU	Core i7-4800MQ 2.7GHz 4CPU 割当
Memory	10GB RAM
Storage	SSD 16GB 割当
SQLite	3.25.0

4.1. 内容

三つのプロセスの処理にかかった時間を提案手法とオリジナルの SQLite で比較した。各プロセスは異なるテーブルに約 0.4 秒の書き込みトランザクションを 1 秒間隔で実行する。もしロックエラーが発生した場合はロールバックしてトランザクションを再実行する。処理が 1 秒以上かかった場合はすぐに次の処理を開始する。

4.2. 結果

1 分間動作させたときの 3 プロセスの 1 トランザクションあたりの平均処理時間を測定したところ、SQLite が 1.21 秒、提案手法が 0.46 秒であった。SQLite では常に競合が発生して、いずれかひとつのプロセスしか処理を行えない状況となり、処理時間は単体処理時 0.4 秒に対して 3 倍（プロセス数）の時間がかかった。提案手法が 0.4 秒とならない原因は、並列に処理を行えないコミットでロックエラーが返ったためである。その試行では処理時間が 2 倍程度になった。しかし、SQLite に比べてロックによる競合は大幅に削減でき、高速化を確認できた。

5. おわりに

マルチプロセス環境で SQLite への行ロック機能を導入するための修正内容を説明した。実際に動作確認を行い、SQLite よりも 2.6 倍高速であることが確認でき、同時実行性をさらに改善できた。今後、単一ユーザでのオーバヘッド低減策を検討する。この改良版 SQLite は SQLumDash として、そのソースコードを GitHub[3]にて公開している。

参考文献

[1] SQLite, https://www.sqlite.org/.
[2] 片山大河, 金松基孝, “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”, FIT2018, D-016, (2018).
[3] SQLumDash, https://github.com/sqlumdash/sqlumdash/.