第 6 章

並行実行制御

並行実行制御は英語では concurrency control (CC) といいます．並行実行制御とは，並行/並列にトランザクションを実行することを前提として，isolation (ACID の I) を担保するために必要な処理です．Isolation は独立性とか分離性などと訳されますが，要はトランザクション同士の実行が混ざらない性質を言います．混ざらないとは何かについて厳密に考え始めると，serializability の話になります．Serializability については後述します．

並行実行制御を行う手法を CC protocol と呼びます．CC protocol の仕事は，各トランザクションが要求する read/write 実行の排他制御したり，(multi-version DBMS の場合は) 指定されたレコードのどの version を読むかを決めたり，commit 処理時に isolation が守られているかチェックをしてダメなら abort させたりする処理などです．具体的にどのようなデータ構造とアルゴリズムを用いるかは CC protocol によって異なります．

6.1 直列実行ではもったいない

トランザクションを直列にひとつずつ実行すれば isolation は完璧です．昔は直列で良かったんじゃないかと思われそうですが，CPU がひとつしかなくても HDD が遅かったため，IO 実行中に CPU がヒマしているのはリソースがもったいないわけで，並行実行したいというモチベーションはありました．現代はひとつのサーバだけ見ても CPU コアがたくさんあり，メモリも潤沢で，永続ストレージも flash memory やら NVRAM やら高速な選択肢が豊富になりました．現代のアーキテクチャにおいては昔にも増して，直列実行しかできないのではリソースがものすごくもったいないということになります．

6.2 並行実行制御の難しさ

直感的には，まったく異なるレコードにアクセスするなら並列に実行しても何の問題もなさそうじゃないか，と思います．はい，そのとおりです．では，アクセスするレコード集合 (ここでは read/write set と呼びます) がトランザクション実行前に把握できるでしょうか．実は，限られたものを除けば，難しいです．仮に one-shot トランザクションのように決定的な挙動をするトランザクションに限ったとしても，アクセスするレコード集合データベースの状態に依存します．その場合，トランザクションロジックは，database 状態と入力を引数にとり，変更後の database

状態と出力を返り値とする副作用のない関数，`function do_transaction(before_database, input) -> (after_database, output)` と解釈できます．`do_transaction` の中身を静的解析 (`before_database` なしで分かる情報を得るという意味です) できたとしても，限られたケース以外では read/write set を決定するのは無理ですね．限られたケースとは，`before_database` の状態に依存せずに read/write set が確定することです．例えば `do_transaction` 内に条件分岐があり，それによって read/write set が変わるとしたら，実行前に確定させるのは無理ですね．

もし `before_database` も使えれば read/write set はトランザクション実行開始前にほぼ把握できるはずだ，と思ったあなた，それは正しいです．その仮定を置いた DBMS を deterministic DBMS と呼びます．私は deterministic の仮定が世の中で広く使われている DBMS の用途全てに適応できるとは思っていませんが限られたアプリケーションであれば適用できるかも知れません．Deterministic DBMS ではない，すなわち，トランザクション開始時には read/write set が不明であるとの立場をとる DBMS を non-deterministic DBMS と呼びます．静的解析だとか過去のワークロードから read/write set を推定などする手法も，広い意味での non-derministic DBMS に分類することにしましょう．

我々は non-deterministic DBMS を想定します．Non-deterministic DBMS の CC protocol は `do_transaction` を実行しながら isolation を担保するために頑張る必要があります．

6.3 Serializability

Serializablility とは日本語では「直列化可能性」と訳されます．トランザクションの理論では，各トランザクションは有限の長さの read/write オペレーション列を持っています．CC を処理する機構を scheduler とよび，scheudler は複数のトランザクションのオペレーション列を適切に順序づけたり，multi-version の場合は read-from 関係を決定します．(multi-version でない，すなわち mono-version の場合は，オペレーション列から reads-from が一意に決まるモデルで，具体的には直前に同一レコードを書いたオペレーションが対象となります．) Scheduler が生成したオペレーション列 (や read-from 関係) を schedule といいます．ある schedule が serializable であるとは，同じトランザクション集合を直列に実行した schedule (serial mono-version scheule といいます) と「等価」であることと定義されます．トランザクションの数が N 個あれば，serial mono-version scheudle はトランザクションの並べ方の数すなわち $N!$ 個存在しますが，そのなかに与えられた scheudle と「等価」であるものが存在すれば良いわけです．

「等価」とは何でしょうか．これには複数の考え方があります．

Final State Serializability

結果として得られるデータベースの状態 (final state) が同じなら「等価」とみなす考え方です．ある schedule について final state が同じ serial mono-version schedule が存在するとき，その schedule は final-state-serializable であるといい，final-state-serializable な schedule からなる集合を FSR といいます．実は FSR だけでは我々が良しとする等価性にはなりません．各トランザクションが一貫性のない値を読んでしまったりする問題があります．

View Serializability

一貫性のある値を読めるとはどういうことでしょうか．それは view が同じなら「等価」とみなす考え方です．View とは，各トランザクションが，どのトランザクションの書いた値を読んだか，すなわち reads-from 関係の集合を指します．ある schedule について，view が同じ serial mono-version schedule が存在するときその schedule は view-serializable であるといい，view-serializable な schedule からなる集合を VSR といいます (multi-version schedule の場合は MVSR)．VSR に含まれる schedule は FSR にも含まれます．VSR は我々が満足する等価性を持っていますが，難点があります．それは view が等価である serial monoversion schedule を探すのが難しいということです．理論では，与えられた schedule が view serializable であるかどうかを決定する問題は NP-complete です．

Conflict Serializability

我々が満足する性質をもっている等価性で，かつ現実的な CC protocol が作れるものはないだろうかという話になります．はい，あります．それは，競合関係 (conflicts) が同じなら「等価」とみなす考え方です．具体的には，同一レコードにアクセスするトランザクションの関係のうち，片方が write をするものを競合関係と定義します．競合関係は，オペレーション列における順序を考慮して，write-read (w-r)，write-write (w-w)，read-write (r-w) の 3 つです．Read-read (r-r) は競合とはみなしません．ある schedule と競合関係が同じ serial mono-version schedule が存在するとき，その schedule は conflict serializable であるといい，conflict serializable な schedule からなる集合を CSR といいます．CSR に含まれる schedule は VSR に含まれます．

CSR は mono-version schedule を前提とします．(競合関係は view についてなにも言及していないですが mono-version であれば競合等価性を満たすならば view 等価性を満たすことが導けるのです．) Multi-version schedule で近いものがあるとすれば，MVSR の等価性に加えて w-w の競合関係のみについて等価性を要求する schedule 空間があります．これは過去の論文[*1]では DMVSR とか，私の呼び方では MWWCSR とか呼んでいて，schedule 空間に属するかどうかの決定問題が NP-complete ではなく P に属します．

Anomaly

Anomaly は view の異常 (すなわち view 等価な serial mono-version schedule が存在しない) の典型的なものに名前をつけて分類したものです．全ての anomaly に名前がついているかどうかは示されていないようです．詳細が気になる人は，「いろんな Anomaly」[*2] という記事がありますので参考にしてください．

[*1] On Concurrency Control by Multiple Versions. Christos H. Papadimitriou and Paris C. Kanellakis. 1984. https://dl.acm.org/citation.cfm?id=318588

[*2] いろんな Anomaly: https://qiita.com/kumagi/items/5ef5e404546736ebac49

6.4 S2PL プロトコル

CC protocol の具体例として，長らくデファクトスタンダードとして使われてきた S2PL (Strict two-phase locking) について紹介します．

S2PL プロトコルは 2PL (two-phase locking) プロトコルの亜種なので，まずは 2PL について説明します．2PL およびその亜種はレコード毎に mutex object を用意し，reader-writer lock を用いて排他制御を行います．アクセスする record は必ず read または write ロックを取って，他のトランザクションが触れないようにします．Read ロック同士は共存できます．つまり，競合関係にあるトランザクション同士は排他制御されるというわけです．2PL のルールは 1 つだけです．トランザクションの実行はロックの成長 (growing) フェーズと縮退 (shrinking) フェーズがそれぞれひとつだけ存在することが求められます．トランザクション実行中にひとつでも unlock したらそれ以降 lock はできません，ということです．2PL を使って生成された schedule は CSR に含まれます．

S2PL プロトコルは，write lock の解放を commit 完了後に行う制約を追加で守る必要のあるプロトコルです．これにより，S2PL は (適切に WAL 手法と連携する必要はありますが) strictness も満たします．SS2PL (strong strict two-phase locking) は write lock だけでなく read lock の解放を commit 完了後に行う制約を守る必要のあるプロトコルです．SS2PL は rigorousness も満たします．

6.5 Serializable ではないプロトコル

世の中の DBMS 実装においては，性能が出ないなどの理由で，isolation の性質を完全には満たさない，すなわち serializable とはいえず，ワークロードによっては anomaly が発生してしまう CC protocol が多くの場所で使われています．例えば，read committed とか snapshot isolation とよばれるプロトコルです．Read committed は read lock について 2PL のルールを満たさない S2PL を指すことがほとんどです．Snapshot isolation は multi-version を前提にしたプロトコルで，2PL とは異なる性質を持っています．アプリケーションによってはそれらのプロトコルでも問題とならないかも知れませんが，設計者が問題が起きないように注意深く選択しなければならないことは言うまでもありません．

6.6 CC protocol の実装について

CC protocol はトランザクションの並行/並列 実行を前提としますので，インデクスも並列アクセスに対応しているデータ構造を使う必要があります．2PL で使われる reader-writer lock は CC のためのレコードアクセスの排他制御であって，インデクスを構成するデータ構造への並列アクセスには専用の排他制御が必要になります．

CC プロトコル以外の実装を極力サボってプロトタイプを作りたい場合は，ごくごく単純なデータベースとしてレコードの配列を用意して配列インデクスを key と見做せばデータ構造専用の排他制御が不要で，とりあえず並列に動かすことはできます．ただし，key は $\{0, 1, ..., N-1\}$ で固定ですし，insert や delete 操作にも対応できませんので，ごくごく単純なベンチマーク (YCSB など)

しか実行できません.

6.7 その他の話題

ここにはキーワードのみを書いておきます.

- Early lock release
- Multi-version Concurrency Control
- Optimistic Concurrency Control
- Deadlock prevension
- Starvation