第 5 章

ログ先行書き込み

英語だと Write-ahead Logging (WAL，ワル) と言います．その名の通り，log を (data より) 先に書く方式を意味します．

5.1 そもそも log って何でしょうか？

Log はトランザクションによるデータの変更後のもしくは変更前の差分データのことです．変更後のデータを redo log，変更前のデータを undo log といいます．より厳密にいえば，トランザクション実行前の状態から実行後の状態を作れるだけの情報を持っているものが redo log，実行後の状態から実行前の状態を作れるだけの情報を持っているものが undo log です．ごく稀に，read log，すなわちトランザクションが何を読んだかという log，を考えることがありますが，効率の面からあまり現実的ではありません．通常，log はデータベース本体のデータとは別に記録されます．

5.2 何のために log はあるの？

Redo log はその名の通り redo するための log，トランザクションがあったことにするためのものです．Undo log は undo するための log，トランザクションをなかったことにするためのものです．WAL は ACID 特性の，A と D を実現するための仕組みです．トランザクションが複数の操作からなり，それらの操作を crash が発生し得る世界で atomic に実行したのと同じ結果を得るためです．Crash が発生したときは，データベース本体のファイルが中途半端な状態になってしまうことは避けられません．というのも atomic にしたい複数操作が実際には atomic に永続化を伴って実行できないからです．Log があれば，crash 直後のデータベース本体ファイルが中途半端な状態から，全てのトランザクションについて，あった/なかったのどちらかの状態に回復できます．Commit できたとアプリケーションに返事済みのトランザクションは，必ずあったことにしないといけません．

5.3 何故「先行」する必要があるの？

トランザクションを後からあったことにする/なかったことにすることが出来る方法であればどんな方法を採用しても良いのですが，「先行」して書く方法が 1991 年の ARIES 論文が発表された前後から主流となっています．(NVRAM の台頭で変わるかも知れません...) 先行して log を書いておけば，いつ crash が発生したとしても，データベース本体の中途半端な状態から回復するため

に必要な log が永続化されている状態を保つことが可能です．WAL ファイルに追記していけば良いことからシーケンシャルアクセスがランダムアクセスよりも高速であり，永続化にも追加の操作 (fsync 相当) を必要とする HDD とは相性が良いと思われます．並列に WAL を書く方法も最近は (少なくとも研究レベルでは) 当たり前になりました．(並列じゃない WAL をシングル WAL，並列のものを パラレル WAL と区別して呼ぶことにします．) 似たような方法として，ファイルシステムのジャーナルも WAL の考え方を採用しています．例えば Linux OS で広く使われている ext4 ファイルシステムは，メタデータ操作を atomic 実行の対象として，crash しても fsck (という名の全メタデータチェックツール) の実行が不要な (crash recovery 相当のジャーナルリプレイ操作のみ必要) システムを採用しています．

5.4 Redo/undo log は両方必要？

データベース本体ファイルへの変更の反映についての制約を一番緩いものにしたければ両方必要ですが，必要な制約を守れば片方だけでも事足ります．あるトランザクションの時系列による状態の変化に注目して考えてみましょう．

1. トランザクション開始から commit/abort 命令発行直前まで
2. commit/abort 処理の実行中
3. commit/abort 処理完了後

ここでは，話を簡単にするため redo log，undo log，commit/abort log しかないものとしましょう．commit/abort log は log の中で最後に書きます．これは，commit log が永続化されていれば，そのトランザクションの全ての log は永続化されているという性質を満たすためです．commit log が永続化されていないトランザクションは abort 扱い (なかったこと) になります．ひとつひとつの log の適用 (redo もしくは undo 操作のこと) は atomic に実行できるものとします[*1]．また，同じ log を複数回適用しても結果は変わらないものとします (べき等性)．それぞれの log を適用する順序には制約があることには意識しておいてください．同一の record に対する redo log 適用の順序は log を書いた順[*2]，undo log 適用の順序は log を書いた逆順となります．

Redo log しか書かない (undo log がない) システム

Undo log がないということは undo できないので，一部でもデータベース本体に変更を反映したトランザクションは，必ずあったことにしないとけません．ということは，少なくとも (1) の間にデータベース本体のファイルに変更を反映できないという制約が発生します．さらに，commit が確定する (当該トランザクションの log が全て永続化する，かつ，それが依存している全てのトランザクションの commit が完了している) まで，データベース本体に変更を反映できないという制約も発生します．つまり，(3) になって始めてデータベース本体のファイルに変更を反映しても良いことになります．(あくまでディスク上のファイルについての話であって，メインメモリ上では通常もっと前に反映されています．) すなわち，commit log の永続化が終わってからデータベース本体

[*1] 実際は atomic に操作できなかったりするので，double write するなどの工夫が必要です．
[*2] 厳密には serialization order の順となりますが，S2PL プロトコルを前提とすれば log を書いた順です．パラレル WAL 方式では何らかの方法で順序を担保する必要があります．

に反映を開始することになります．

Undo log しか書かない (redo log がない) システム

Redo log がないということは redo できないので，データベース本体に変更を反映するまで commit したことにできません．undo はできるので，(1) の間でも undo log が永続化済みの操作は，どんどんデータベース本体に反映して問題ありません．(2) において，commit が確定するためには，当該トランザクションの log が全て永続化する，かつ，データベース本体への反映が永続化も含めて終わっている，かつ，依存しているトランザクションが commit 完了している，必要があります．つまり，commit log はデータベース本体の永続化が完了してから書くことになります．(3) においてそのトランザクションについてやるべきことはありません．

Redo log と undo log の両方を書くシステム

片方しかない場合に比べて制約が大幅に減ります．必要な制約は，redo/undo log を永続化してから対応する操作をデータベース本体ファイルに反映開始することのみです．Commit に必要な条件は，commit log まで永続化が完了する，かつ，依存しているトランザクションが commit 完了していることです．途中までデータベース本体に反映したところで crash しても，undo log は永続化されているのでトランザクションをなかったことにできますし，commit log が永続化していれば redo log は全て永続化しているのでトランザクションをあったことにできます．

図にするとより分かりやすいと思いますが，自分で書いてみてください (ぉぃ)

5.5 Crash recovery

Crash が発生した直後のデータベース本体ファイルは中途半端な状態になっています．これを log を使って各トランザクションがあった/なかったのどちらかに確定させ，データベースの永続データを一貫性のある状態に修復します．まず，log を先頭からなめて，トランザクション毎にあった/なかったのどちらにするか決定します．あるトランザクションを「あった」ことにする条件は，

1. トランザクションの commit log が記録されている
2. トランザクションが依存していた全てのトランザクションが commit 扱いとなっている

です．

(2) をきちんと考え始めると，込み入った話になります．トランザクションの「依存」とはなにかということです．トランザクション B が トランザクション A に依存する関係は，A が書いた record を B が読んだときに成立します[*3]．A が abort 扱いになるのであれば，A の書いた record データはなかったことになるので，それを読んでしまった B もまた commit できないということです．このケースで，B を先に commit してしまうと，A が abort してはいけませんが，実際には crash によって A が abort する可能性を排除できないので，破綻します．破綻を避けるには A を commit してから B を commit する必要があります．この制約を recoverability といいます．

[*3] Reads-from 関係ともいいます．B reads from A とか B reads x written by A といったりします．

Recoverability よりも強い制約として strictness があり，それを DBMS が満たせば (1) が満たされたとき (2) も自動的に満たされるようになるので，ここでは strictness を前提として (1) のみを気にすることにします．これなら簡単ですね．詳細はコラムに書いておきました．

Commit log を永続化した後に commit 成否はユーザ/アプリケーションに通知されているはずですから，commit 成功したと通知されているトランザクションは全て commit log が永続化されているはずです．そのようなトランザクションは (1) を満たすわけなので，必ずあったことになります．

トランザクションをあった/なかったのどちらにするかを決定した後は，あったことにするトランザクションを redo します (redo log がないシステムでは不要です)．最後に，なかったことにするトランザクションを undo します (undo log がないシステムでは不要です)．Log 適用の順番には気をつける必要があります．シングル WAL であれば，最も単純な方法として，log の書かれた順番を守って適用すれば問題ありません．必要とされる性質は，DBMS が並行実行制御で決定したトランザクションの順序 (一般に全順序ではなく半順序です) に矛盾しないように適用順序を制御します．その制御のために log に必要な情報を含める必要があります．どんな情報が必要かは並行実行制御プロトコルや WAL 方式に依存します．半順序関係を満たせば良いということは，理論上は並列に log 適用することも可能です．

■コラム: Recoverability や strictness について

DBMS は最低でも recoverability (RC) を満たす必要がありますが，RC のみを満たそうとすると，cascading aborts (abort 処理の連鎖) 機構を DBMS に持たせる必要があるので，設計実装の複雑さやそれに伴うオーバーヘッドを考慮するとオススメできません．昔はあったようですが，現代の DBMS プロダクトで cascading aborts 機構を供えているものは私の知る限りないです．

Cascading aborts を防ぐには，トランザクション A が書いた record をトランザクション B が読もうとするとき，A が commit するまで待てば良いです．この制約を avoiding cascading aborts (ACA) といいます．ACA を満たせば RC も満たします．

ACA だけだと crash recovery 時に処理が複雑になります．具体的には，トランザクション A がある record 書いた後，トランザクション B が同一 record を書き，A のみ abort 扱いで B が commit 扱いになった場合です．この場合，A の undo 処理の後に B の redo 処理を行う必要があります．B の redo よりも A の undo を後で行うと，B の書いたデータが A の undo 処理によって消えてしまう可能性があるからです．まとめて redo した後，まとめて undo するというより単純な crash recovery を実現するには，A が commit するまで B が書くのを待てば良いです．ACA にこれを加えた制約を strictness (ST) といいます．

Crash recovery が目的であれば ST で十分です．しかし，もっと強い制約が必要なケースがあります．それは log を他のホストにレプリケーションして，適用し，read-only トランザクションを実行するような構成です．問題が起きるのはトランザクション A がある record を読んだ後，トランザクション B が同一 record を上書きし，しかし commit 順は B,A となってしまった場合です．このときレプリケーション先で B までの log を適用し A の log がまだ適用されていないデータベースは一貫性のある状態とはいえません．レプリケーション元では A の後に B が実行されたことになっていますが，この状態は元の世界では存在しないからです．

この問題を防ぐためには，やはり A の commit を待ってから B は上書きする必要があります．ST にこれを加えた制約を rigorousness (RG) といいます．

詳しくは，Transactional Information Systems 本の 11.4 節 Sufficient Syntactic Conditions を参照ください．

最近のインメモリ DBMS では commit 操作が pre-commit と log の永続化に分離されている設計が多く，commit を待って読み書きする，という言葉が言葉通りに解釈できないことがありますのでご注意ください．

5.6 Checkpointing

Log はいつまで残しておけば良いでしょうか？ それは，log に対応するトランザクションが完了していて，当該トランザクションの全ての log の操作がデータベース本体ファイルに反映され終わるまでです．それらの条件が満たされた log はもう不要なので，消すことができます．メインメモリが大量にあったり，そもそもインメモリデータベースなどでは，データベース本体ファイルへの変更の反映をずっとずっと先延ばしにすることが可能です．先延ばしし続けると crash 時に適用しなければならない log が増え，crash recovery にかかる時間が長くなってしまうだけでなく，いつまでも log を消せないということになります．それを防ぐために行う操作が checkpointing です．Checkpointing とは，先延ばししていたデータベース本体ファイルへの反映を実行して，log を消す作業です．プロトタイプを作る場合は実装が後回しになる機能だと思いますが，長時間 DBMS を動かし続けるためにはなくてはならない機能です．