第 10 章

DBMS を学ぶためのリファレンス設計 応用

前章「DBMS を学ぶためのリファレンス設計 基本」に従ってプロトタイプを作り，動かしたりテストしたりして満足し，次のステップに進みたいと思った方は，楽しい並行実行制御やその他の改善を実装するための準備をしていきましょう．本資料では，向かうべき方向について概要のみを述べます．詳しくは，キーワードを頼りにご自分で調べてみてください．場合によっては専門の教科書や学術論文を読まないといけないこともあるでしょう．

10.1 設計の選択肢

シングルスレッドで並行実行制御ナシ，マルチスレッドで並行実行制御アリ，という設計の間に，シングルスレッドで並行実行制御アリ，という設計が考えられます．CPU コアはひとつしか使わないけれど，IO や他の処理待ちの間は別のトランザクションなどを実行することを意味します．具体的にはいわゆるユーザースレッドを使って並行処理を行うシステムです．並列 (parallel) ではなくて並行 (concurrent) という言葉を使っているところが，昔はまさにそうしていたことの状況証拠ではないかと思っています．

CPU コアがたくさん使える現代において，並行実行制御をするけれど，CPU コアはひとつしか使わない縛りをする意味が，性能を目的とした場合はあまりありません．CPU コアをたくさん使った方たくさん処理できるはずですから．. 皆さんには，最終的にはマルチスレッド動作する，スケーラブルなシステムを目指して欲しいなと思っています．

シングルスレッドで並行実行制御アリの設計を採用する利点があるとすれば，それは皆さんのような学習者にとってのものです．マルチスレッドに比べて設計実装の難易度が低いからです．シングルスレッド，すなわち DBMS が CPU コアをひとつしか使わずに動作するため，メモリに存在するデータ構造についての排他制御が必要なくなります．一方，トランザクションは並行に動くため，並行実行制御は必要となります．この設計でも，例えば素朴な 2PL を実装し，デッドロックなどを引き起こすことはできます．

マルチスレッドで並行実行制御アリの場合，データ構造の排他が必要になってきますので，これがひとつのハードルになると思います．

10.2 データ構造の排他制御

DBMS をマルチスレッド動作させようとすると，データ構造に排他制御が必要になります．以下，具体的なポイントをいくつか列挙します:

- シングルスレッドでアクセスすることが前提の thread-unsafe なインデクス構造は動きません．
- もちろんインデクスまるごと排他制御すれば動きますが，全然スケールしないので，マルチスレッドで頑張る意味が激減します．
- インデクス構造上で細粒度の排他をする方法は色々と提案されていますが，枯れているものとして，intention lock があります．
- 最近は，mutex-free なインデクス構造を使うことも珍しくなくなってきたと思いますが，GC (garbage collection) の仕組みとセットになるので，実装難易度は高めだと思います．具体的な設計を挙げると，Bw-tree とか，Masstree などでしょうか．Java などの GC が前提の言語で開発している場合，自分で GC を実装する必要がないので楽できます．メモリ管理を自分で行う必要がある C++ などでは，QSBR (quiescent state based reclamation) や EBR (epoch-based reclamation) という手法があります．
- ごくごく単純なプロトタイプレベルでよければ，insert も delete もなし，update と retrieve のみを前提に，インデクス構造もなし，ただの固定配列 (fixed array，key は array index) をインメモリ DB として並行実行制御をマルチスレッド DBMS として実装して動かしてみることはできます．

10.3 並行実行制御

並行実行制御には色々な方式があります．理論から勉強するには，現状，

- Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.
  - 著者: Gerhard Weikum and Gottfried Vossen
  - 出版社: Morgan Kaufmann
  - 出版年: 2001 年

という本を読むしかないと思います．また，ひとつひとつの CC プロトコルをきちんと理解したいのであれば，この本の参考文献に挙げられている元論文もあたった方が良いです．他の本だと 2PL のみが紹介されていることが多いと思います．

2001 年以降の並行実行制御の研究について，特に many-core アーキテクチャでのインメモリ DBMS 向けのものについてのキーワードをいくつか挙げておきます:

- Silo (2013)
- FOEDUS (2015)
- SSN (Sefial Safety Net, 2015)

- TicToc (2016)
- MOCC (Mostly Optimistic Concurrency Control, 2016)
- Cicada (2017)

それぞれ対応する論文がありますので，興味のある方は読んでみてください．

まずは S2PL を実装してみましょう．ロッキングプロトコルを用意します．ごくごく単純な reader-writer lock で構いませんし，reader (shared) lock をサポートしていないただの lock でも構いません．並行実行できる機会は減りますが，reader lock を writer lock で代用できます．

Record 毎に mutex object を用意します．ロックテーブル (ロックを管理するためのデータ構造) を用意するアーキテクチャもありますが，簡単なのは，record 毎に mutex object を用意するアプローチだと思います．Record を格納する構造体の中に確保しても良いですし，ポインタのみ保持して，別途確保された mutex object を指しても良いです．

S2PL は元々ページモデルという理論を前提として serializability の議論がなされています．そしてページモデルは record の insert/delete を考慮していないモデルです．S2PL はページモデルでは serializable (CSR) スケジューラなのですが，insert/delete 操作が存在する世界では，phantom problem と呼ばれる一貫性の問題が存在します．Phantom problem は具体的には range scan と insert/delete の順序が前後することで読めるべき record が読めていなかったり，読むべきでない record が読めてしまったりする問題です．Mono-version を前提とする場合，多くの設計では追加の排他制御を行うことで対処します．Range (範囲) を排他するためには，predicate locking (条件) という仕組みを使う場合と，インデクス構造を使って range 相当のオブジェクト (たとえば B+tree の leaf node) を lock する方法があります．MySQL InnoDB が採用している next key lock という手段もあります．

素朴な S2PL には deadlock という問題があります．例えば record A -> record B という順番で lock するトランザクションと record B --> record A という順番で lock するトランザクションを同時に動かすと，簡単にお互いがお互いをロック解放を永遠に待ち続けてしまう現象が発生します．これが deadlock です．Deadlock の発生は困るのでなんとかしたいと思った方は，Deadlock avoidance とか deadlock prevension というキーワードで調べてみてください．

ここでは，S2PL しか紹介しませんでしたが，他の並行実行制御手法について調べて，探求しましょう．キーワードとしては，OCC (Optimistic concurrency control)，MVCC (Multi-version concurency control) などです．既存の deadlock を解決する方法の多くは，starvation という別の問題を引き起こしますので，それについても探求してみてください．

10.4 WAL

WAL はまずシングル WAL を実装することを考えてみましょう．マルチスレッド設計の場合，複数のトランザクションが同時にログを書こうとするわけなので，直列化のための排他が必要になります．通常は，thread-safe な queue 構造を用意して，それを経由して WAL に書き込むことで直列化を実現します．

WAL がボトルネックになると思ったら，パラレル WAL にも挑戦してみてください．並列に複数の WAL ファイルに書くとしたら，どうやってトランザクションの依存関係 (recoverability など) を担保すれば良いでしょうか... これも奥深いテーマです．Redo log についていえば，CC プロト

コルが決定したトランザクションの serialized order と矛盾しない順序を保持できれば，rigorous といえます．

ハードウェアアーキテクチャの進化によっては，WAL という仕組みそのものが効率などの面で過去のものになる可能性もあります．あくまで WAL は atomicity と durability を実現するためのいち手段ということを忘れないでください．

10.5 おわりに

駆け足で説明していまいましたが，これらひとつひとつが大きなテーマだと思います．トランザクション処理だけでこれだけのことを考えなければならないのです．OS ほどではないかも知れませんが，かなり入り組んだシステムであることが分かっていただけましたでしょうか．

ここではトランザクション処理に注目しましたが，データを扱う処理として，データ分析というテーマが別にあります．SQL はどちらかというと，データ分析のための言語と言っても良いでしょう．大量のデータを複雑な条件で選択したり，変換したり，ソート，集約，結合，などなど様々な処理を行えます．実装としては，トランザクションをサポートしている DBMS (MySQL や postgreSQL など) でも出来ますが，大規模なデータに対して動かしたけれど，例えば Hadoop や Spark などがそれを担ってくれます．それとは別に，特殊なデータベースとして，全文検索エンジンがあります．こちらも特殊なインデクスを提供し，キーワードなどによる文書検索を担う DBMS と考えることが出来るでしょう．トランザクションシステムとこれらのデータ分析等のシステムの統合や連携については，様々な試みがなされています．個人的な感想ですが，データ分析やその基盤を探求している方は，トランザクション処理を探求している方々よりも人口が多い気がします．

新しい用途が生まれたり，ハードウェアアーキテクチャの進化に合わせて DBMS の良いアーキテクチャも変わっていきます．探究心を忘れずに，今後も楽しい DBMS 開発人生を過ごしてください．