第 9 章

DBMS を学ぶためのリファレンス設計基本

これまで，トランザクションを処理を目的とした DBMS が備えているべき機能や性質について説明してきました．では，どんな機能があればトランザクション処理できると言えるでしょうか．本章では，トランザクション処理ができるといえる最小限の機能セットについて考えます．習うより慣れろの精神で，作ってみよう，という人はこれを読んで実際に動くプログラムとしての DBMS を作ってみてください．

9.1 並行実行制御

並行実行制御は，今なおより良いものを求めて研究が行われている奥深いテーマですが，必要最小限のトランザクション処理システムに必要か，と言われると，なくても成立すると思いますので，ここでは涙を飲んでバッサリ削ろうと思います．並行実行制御をせずに，きちんと動かす，すなわち serializable に実行するためにはどうするか．そうです，本当に serial (直列) 実行をすれば良いのです．直列実行するスケジューラを trivial scheduler といいます．また，データ構造の排他は大変＆設計と実装の難易度が上がりますので，思いきってシングルスレッドで動かす DBMS を作ることにしましょう．

ということで皆さんはまずシングルスレッドで動くトランザクション処理システムを作ることを目標にしてもらおうと思います．Trivial scheduler には並行実行制御は不要なので，インデクスと WAL 機能があれば最小限の DBMS ができることになります．

9.2 インデクスとスキーマ

インデクスを実現するデータ構造について，Tree map と Hash table を紹介しました．より単純な DBMS は，どちらかのみサポートしているでしょう．どちらでも良いですが，迷ったら tree map をオススメしておきます．Tree map なら range query が出来ますからね．

インデクスを使わずに record の配列をデータベースだ，と主張することも可能です．Table full scan はできますからね．ただ，お目当ての record に低コストでアクセスできる機能はさすがに必須としたくはなりますが... 皆さんにおまかせします．

メモリ上とディスク上のデータフォーマットを無理に共通化したくありませんし，ページ単位で atomic に書いたりするのが面倒そうなので，インメモリ DBMS を作ることにしましょう．メモリに格納できない大きなデータベースは扱わないという割り切りをします．これで，メモリ上でインデクスを実装すれば良くなりました．シングルスレッド前提なので，排他制御も不要です．あれ，多くのプログラミング言語ではほぼ標準ライブラリでこの要件を満たす tree map や hash table を用意しているようですよ．．それらのライブラリを使えばインデクスが何故高速か，そのメカニズムを知らなくてもインデクスを実装できてしまいます．私としては学びのために自分でインデクス構造を実装することも検討して欲しいのですが，最小限,,, という意味ではサボっても構いません．

DDL (data definition language) 等を用意すると手間なので，設定ファイルを読み込む形で定義させるか，もっと簡単に作るために，ハードコーディングしてしまいましょう．動的なスキーマ変更もサポートしないこととします．

Record の仕様を決めましょう．もちろん，たくさんの primitive 型を用意して，任意の column を組み合わせて record を定義できると便利ですが，実装を簡単にするために，サボりましょう．巷の key-value store と呼ばれるものには，key 型も value 型もバイト列しか選べないものもあるようです．複数の型が扱えるようにしたいとしても，とりあえずは文字列 (バイト列) と整数，くらいでいいのではないでしょうか．

セカンダリインデクスは欲しくなるかも知れませんが，まずは，プライマリインデクスだけあれば良いでしょう．同じデータ構造を流用すれば良いので，セカンダリインデクスは比較的簡単に実装できると思います．

9.3 ログ先行書き込み

インメモリ DBMS とはいっても，ACID を満たすために永続化はしないといけません．ログ先行書き込み (WAL) 機能は commit 条件を満たすために実装する必要があります．ログファイルのフォーマットを決めましょう．DBMS がどんなデータ操作をサポートするかを決めて，そのオペレーションの redo log の仕様を決めます．例えば update/insert/delete の三種類をサポートするとします．

Redo/undo log のどれを採用するかについて，trivial scheduler を使う場合，トランザクションロジックによる明示的な abort 命令と crash による abort 以外は abort しませんので，実装が簡単な redo log のみを使う実装をオススメします．Commit が確定するまでデータベース本体に変更を反映するのを待つ処理を入れれば undo log は不要です．Redo log は 後述する write set から作れます．

9.4 Read set と write set

DBMS 側のトランザクション実行エンジンはどのように振る舞えば良いでしょうか．我々は non-deterministic DBMS の立場を取るといいましたね．実行エンジンは次にどんな命令が来るのか分からないので，トランザクションの状態管理をする必要があります．

同じ record に対する複数回のアクセスをうまく吸収するための仕組みとして read set と write set があります．Read set はトランザクションの中で，過去に読んだことのある record の参照とそ

の内容を保持しておきます．Write set はトランザクションが書いた内容で，やはり record の参照とともに保持しておきます．Trivial scheduler を採用するとき read set は不要です．なぜなら自分以外にトランザクションは並行に実行されていないので，同じ record を何回読んでも自分が変更しない限り同じだからです．

Trivial scheduler を採用していても write set は必要です．何故なら redo log のみ保持する設計にしているからです．Redo log しか保持しない場合 commit が確定するまでデータベース本体に変更を反映してはいけないです．もし commit 前にデータベース本体に反映してしまい，その後トランザクションロジックから abort 命令が来たら undo できなくて詰みます．Commit が確定するということは undo する必要がなくなるということです．トランザクション実行中はデータベース本体と write set は別々に管理して，トランザクションロジックからはあたかもそれまでの変更がデータベース本体に反映されているかのように振る舞います．すなわち，write set に存在する record の read 要求に対しては write set に保持されている内容を返すということです．別の見方をすれば write set はキャッシュデータとして振る舞います．

Write set はトランザクションによるデータベース変更への変更データそのものなので，write set から redo log が作れます．自分で決めたフォーマットに従って変換するだけです．

9.5 Checkpointing

Checkpointing は出来るだけ簡単なものにしたいので，トランザクションが動いていないときのみ，もっと極端には正常終了時と起動時のみにやることにしましょう．起動時には，前回のデータベースファイルをメモリに読み出し，WAL ファイルの中身を redo してメモリ上に正しいデータベース構造を構築し，そのイメージをディスクに書き出し，WAL ファイルを空にしてからトランザクションを受けつけます．正常終了時には，トランザクションの受付を停止し，メモリ上の正しいデータベースをディスクに書き出して WAL ファイルを空にしてから終了します．これなら，排他処理不要の dump/load 機能を用意するだけで済み，ちまちまと細粒度で checkpointing するコードを書くのを避けられます．その代わり，起動時と正常終了時に皺寄せがきますが，諦めましょう．また，ファイル操作を適切に atomic に実行するのを忘れないようにしましょう．

9.6 トランザクション

Read/insert/update/delete そして commit/abort を実行する API を用意して，簡単な動作確認をするためのトランザクションを実装してください．トランザクションロジックが呼び出すデータベース操作 API はライブラリとして実装しましょう．それがネットワークなどを介した専用プロトコルを用意するより簡単です．それを組み込み DBMS というのでしたね．トランザクションやそれを呼び出すワークロードも同じプログラミング言語で書いて DBMS 実装に組み込んでしまいましょう．DBMS が起動し，初期化が終わってトランザクションを受けつけられる状態になったら，トランザクションを有限個または無限個実行するようなコード (ワークロード実装) を用意しておいて動かしてください．Crash test がしたい場合は，外部から強制的に DBMS が動いているホストを落とすので，トランザクションを無限に実行し続けるようにしておく必要がありますね．性能測定がしたい場合は，時間を測ったり実行できたトランザクションの個数を数えたりするコード

もアプリケーションとして一緒に書いてしまいましょう．

9.7 作って動かしてみる

以上で，大体のアーキテクチャは固まってきました．細かいところは自分で考えてみてください．また，ご自分の興味に従って設計を変更しても大丈夫です．どんな設計をすればどんな制約が出てくるかを考えるのは大切なことです．考えながら設計しましょう．ただし，まずは出来るだけ簡単なものを作ることが大事です．そして，動く状態を保ちながら機能を追加していくこともまた大事です．どこから手をつけるべきかも含めて考えましょう．

9.8 テスト

作ったプログラムが正しく動いていることを確認するにはテストをすることが欠かせません．正常系として，insert/update/delete したらそれが反映されているか，などの基本動作についてテストします．プロダクションで動かすことを考えていくならエラー処理 (そして異常系テスト) がとても重要ですが，学習用のプログラムなので，ある程度は目をつぶりましょう．. しかし，異常系の中で DBMS としてひとつだけ絶対に押さえておかなければならないテストがあります．それが crash test です．トランザクション実行中に電源を落として，crash recovery できることを確認してください．Virtual Machine を使って仮想的に電源を落とすのが良いでしょう．Commit の返事をしたのに反映されていないことがないかどうか，中途半端な状態になっていないか，データとして壊れていないかどうかを確認してください．

■コラム: テストについて

コードを書いたり修正したとき，とりあえず動作確認をする行為は，スモークテストというそうですが，それはそれでとても大事なテストです．私がここで言っているテストは，リグレッションテストと呼ばれているものです．リグレッションテストとは，コード修正をしたことで，それまで期待通りに動いていたはずの機能が壊れたとき，それに出来るだけ早く気付くためのテストです．ですから，

1. コードを変更したときにすぐに実行してパスするかどうかを確認できること
2. CI などで自動化できること (テストの準備や実行をするスクリプト等の用意，テスト結果もコマンドの返り値などで判別できるようになっていること)

が必要になります．今回，CI で動かす必要が必ずしもあるかは分かりませんが，たとえば make utest などと実行したらすぐにテストが走るようにしておくことが重要です．

私は，C++ でプログラムを書く場合，同僚の作ってくれた簡易なヘルパ関数を使ってテストを作り，それを make utest で動かせるようにしています．

- https://github.com/herumi/cybozulib/blob/master/include/cybozu/test.hpp

これを使ってもいいですが，原則としては，終了コードでパスしたかどうかを判別できるよ

うなテストプログラムをテストしたい項目毎に書けば良いと思います．もう少し複雑なテストをしたい場合は，環境や入力データの作成，テスト実行，結果のチェックなどをする専用のスクリプトを書いたりします．これもコマンド一発で動かせて，成功失敗を返り値などでチェックできるようにしておけば CI で動かせます．

実運用で使う，バグを踏むとものすごく困ったことになる，メンテナンスを少なくとも数年続ける必要があるようなプログラムと違って，今回のような学習用，プロトタイピング的なプログラムを作る場合に，このようなテストにたくさん時間をかけたくないでしょう．ただ，何回も実行するのが面倒くさいと人間はサボるようになりますから，何回も実行する必要があるテストは自動化しておきましょう．Crash recovery は事ある度にテストして欲しいです．また，B+tree などもそれなりに複雑なロジックで，色々な状態，入力に対して動くことを確認したくなるはずですから，テストコードを書きましょう．目安として，自分が複雑だと感じるコードや機能については，まず間違いなくテストが必要です．経験則として，そのようなコードは大抵の場合バグっています．もし大丈夫だと安心していたコードがバグっていたことが判明したら，あなたの安心を感じるセンサーを修正する必要があります．大前提としてバグをなくすことはできませんから，バグが入りにくい設計，実装を心がけることが重要なのはいうまでもありません．