第 3 章

トランザクション

本章では，トランザクションをより具体的にイメージしてもらうことを目標に説明します．

3.1 トランザクションが行う操作

DBMS におけるトランザクションが実際にデータベースに対して行う処理は，データの読み書き操作を複数回実行するだけです．それだけです．

データを読む場合，クエリ (典型的には SQL の select 文) を実行して，その結果を得ます．Select 文は読んだデータを色々と加工できますが，データベースに対して行っていることは，データを読んでるだけです．SQL をサポートしない，より基本的な機能のみを持つ DBMS では，ある table において key とその値を指定して絞り込んだ record 集合を読むという操作が可能です．このような基本的な DBMS は join や group by などの高度な演算はサポートしておらず，必要ならアプリケーション側でそれらの機能を実装する必要があるだろうということです．読むだけでは書けませんので，DBMS においては select 文や key による絞り込みによって得た record 集合に対して更新 (update) や削除 (delete) 操作が可能です．また，挿入 (insert) 操作は，record を生成し，table に追加することができます．

SQL の世界では select, update, delete, insert がデータベースを読み書きする基本的な操作群で，これらをまとめて DML (Data Manipulation Language) と呼びます．ここでは SQL を使わない世界にも配慮して select の代わりに read という言葉を使うことにしましょう．また，update, delete, insert はデータベースに変更を加えるという意味で write と呼ぶことにします．

Read, update, delete, insert の 4 つの操作ができるのが汎用的な DBMS だとすると，より低機能な DBMS は何を供えているべきでしょうか．Read ができないと DBMS の意味がないでしょうね．実際，ほぼ read-only の DBMS というのは存在します．Hadoop など主にデータ分析に特化したものです．実際はデータ投入や追記など制限的な write 操作が可能ですが，例えば read 操作と write 操作は同時に実行されないことを前提として作られていたりします．私はこれをトランザクションシステムと呼ぶのには抵抗があります．何故なら本来トランザクションシステムが制御しなければならない難しさをほとんど排除しているからです．もちろん read-only システムには目的に特化した特有の難しさはあります．例えば，Hadoop では，如何にタスクを分割して複数ノードおよびプロセッサに割り振るか，どのアルゴリズムを使って集約処理するか，などは難しい問題だと思います．ただ，これはトランザクションシステムを実現する難しさとは違う難しさですよという

主張です．Read に続いて次に，データ投入を実現したければ insert が必要でしょう．初期データロードという特殊なタスクは必ずしもトランザクショナルに行わなくても良いのですが，ここでは，insert 操作で代用することにしましょう．データが増える一方というのも困るでしょうから，次に delete が欲しくなるでしょうか．Update は delete と insert で代用できますので，単純に機能だけで見ると，update 操作は一番優先順位が低いといえるかも知れません．別の視点で見ると，初期データロードは別の手段で用意するとして，データは増えもせず減りもしないというデータベースを考えることもできます．その場合は，read と update だけあれば事は足ります．このように，より制限的な操作しかトランザクションに許さない DBMS も read-only でなければトランザクション処理システムと言えるでしょう．

どの record をどのように読んで，どんな計算をし，どのように書くかはトランザクションの内容次第です．それをトランザクションロジックと呼びます．トランザクションロジックはどう表現されているでしょうか？ 一般にトランザクションロジックは，アプリケーション内に実装されたトランザクションを実行するプログラムコード断片や，DBMS 内に記録されたストアドプロシージャなどに記録されています．

3.2 トランザクション処理の流れ

トランザクションの実行は，ユーザやアプリケーション側からは，begin コマンドで始まり，先に説明したデータベースの読み書き操作を複数回行った後，commit または abort コマンドの実行で終わります．Abort コマンドは rollback コマンドと呼ばれることもあります．Begin, commit や abort は DML とは区別され，トランザクション制御文と呼ばれるようです．トランザクションが開始されることが明らかであるときは begin が省略できるインターフェースもあります．Commit コマンドはトランザクションの正常終了を試みる操作です．成功した場合，その結果すなわちトランザクションによる書き込み操作がデータベースに反映されます．Abort コマンドは，トランザクションを意図的に失敗させて，なかったことにする操作です．Commit 操作が失敗した場合，abort 操作をしたのと同じ結果となります．DBMS はトランザクションの commit 要求が来て，そのままでは ACID の性質を担保できないと判断したとき，commit 操作を失敗させて abort 扱いにすることがあります．私が知る限り，commit 操作が失敗するのは，並行にトランザクションが実行されていて，全てのトランザクションを同時に commit 扱いすることができない場合です．アプリケーションは，commit 操作を DBMS に要求し，その返事が返ってくるまでは成功したかどうか分かりません．また，突然の電源断などの故障 (以後 crash と呼びます) が起き，commit 成否の返事を受けとれなかったとき，commit できたかどうかは，再起動時のデータベース復旧操作 (Crash recovery といいます) が終わるまで分かりません[*1]．

3.3 トランザクションが満たすべき性質

理想的には，commit に成功したトランザクション全てが ACID の性質を満たすように，そして，出来るだけ commit を成功させるように，DBMS は頑張る必要があります．ACID についてはもう知ってると思いますので，省略します．

[*1] 分散 DBMS においては故障の概念が違うのでその限りではありませんが，ここでは議論しません．

実装によっては ACID のうち I すなわち isolation について設定で制約を緩くできるものがありますが，DBMS 側で性能面の制約が減る代わりに，isolation が一部のケースで保証されない場合のケアをする責任がアプリケーション側に負わされます．Isolation については並行実行制御の章でも述べます．A，C，および D を緩められるシステムというのはほぼ有り得ません．それすらできないシステムは少なくともトランザクションを処理できると言わないと思います．トランザクションという言葉が拡大解釈して宣伝に使われるケースがありますが，「それ本当にトランザクション実行できるんですか？？？」という疑問は常に持つようにしましょう．また，ACD は大丈夫そうだとしても「Isolation はどのくらい担保されるんですか？？？」という疑問も持つようにしましょう．

3.4 トランザクションシステムの分類

トランザクションは，one-shot トランザクションと interactive トランザクションに分けることが出来ます．One-shot トランザクションとは，トランザクションの開始前に外部から入力データを与えて，トランザクションの完了後に外部に出力データを，少なくとも commit 成功か失敗かを返すトランザクションのことで，トランザクションの実行中に，外部とのデータのやりとりを行わないものです．Interactive トランザクションとは，トランザクションロジックがアプリケーション側に存在していて，トランザクション実行中にも外部とのデータやりとりができるものです．

Interactive トランザクションをサポートする場合，アプリケーション側にトランザクションロジックが実装されますから，DBMS とやりとりするために DML などのコマンドをデータと共にネットワーク等を経由して送り合う必要が出てきます．当然，そのプロトコルを設計実装しなければ動きません．一方，one-shot トランザクションはストアドプロシージャで利用することが想定され，アプリケーションはストアドプロシージャを指定し，その入力データを DBMS に送り，最後にトランザクションの出力のみを受けとるというより簡単なプロトコルで済みます．また，one-shot トランザクションをサポートしようと考えたとき，ストアドプロシージャを記述する専用言語と実行ランタイムを用意する方法もあり得るでしょうし，もう少し楽な方法としてプラグインで実現する方法もあるでしょうが，らに楽な方法として，ストアドプロシージャを DBMS と同じコードレポジトリ内に定義してしまい，一緒にコンパイルしてしまう方法も考えられます．他にも，one-shot トランザクションのみ想定するならアプリケーションとのやりとりの遅延を隠蔽できるので，より性能を確保したい場合 one-shot トランザクションに特化したシステムは魅力的だと思います．2019 年現在，多くの実用的な DBMS (特に RDBMS) は interactive トランザクションを実行できるように作られています．ただ，より性能を求めたアカデミアでの研究の多くは one-shot トランザクションを前提にしたものが多く，高速さやスケーラビリティを求めた一部のプロダクトでも one-shot トランザクションに特化したものもあります (VoltDB など)．

一方，アーキテクチャの視点でトランザクションシステムを区別することもできます．一般に，アプリケーションと DBMS は別プロセスで動きます．同一コンピュータで動くこともありますが，別のコンピュータで動いていて，やりとりはネットワーク経由ということも珍しくありません．これには例外があり，それが組み込み (embedded) DBMS です．BerkeleyDB などの組み込み DBMS は，トランザクションロジックコードから DBMS の機能をライブラリ関数経由で呼び出し，それが直接 DBMS 側のコードを呼び出す仕組みになっています．つまり，アプリケーションと DBMS のコードが同一のプロセス内で動作します．当然，データベースファイルはローカルに保存されてい

る前提です．BerkeleyDB は interactive トランザクションを実行するために作られていますが，組み込み DBMS であるが故にアプリケーションと DBMS 間でのネットワーク等を介したプロトコルが不要です．組み込み DBMS では interactive と one-shot の区別をする意味があまりありません．

非組み込み DBMS よりも組み込み DBMS の方が簡単に設計実装できます．ライブラリとして作るよりも，DBMS のコードに必要なトランザクションロジック等を追加で記述してしまい，一緒にコンパイル/ビルド/リンク等してしまうのが一番お手軽かと思います．不要であれば入出力の機能すら省略してしまうことができます．テストやデバッグ，ベンチマークのみを差し当っての目的にするのであれば入出力はほぼ不要でしょう．疑似乱数を用いてワークロードを生成するとか，パラメータをハードコーディングするとか，使い勝手という点で色々と制約はありますが，トランザクションシステムとしては機能します．