［改訂新版］

SQL実践入門

高速でわかりやすいクエリの書き方

ミック
［著］

技術評論社

本書は、小社刊『WEB+DB PRESS』の下記の記事をもとに、
大幅に加筆と修正を行い書籍化したものです。

- **Vol.44 特集 2「SQL アタマ養成講座」**
- **Vol.57 特集 2「リレーショナルデータベース & SQL 入門」**
- **Vol.45〜55 連載「SQL アタマアカデミー」**
- **Vol.56〜61 連載「DB アタマアカデミー」**
- **Vol.62〜67 連載「SQL 緊急救命室」**

はじめに——クラウドネイティブ時代にパフォーマンスチューニングはどのような意味を持つか

本書の初版が刊行されてから、10年が経過しました。ありがたいことに、初版は長い期間にわたって読み継いでいただき、ちょっとしたロングセラーとなりました。

初版の目的は、パフォーマンスと読みやすさの両立を目指したSQLの書き方(必ずしもSQL単独のリファクタリングだけではなく、モデルの修正にも踏み込んでいましたが)を学ぶことでした。これは、当時としては挑戦的な目的で、実行計画の読解にまで踏み込んだ書籍というのは日本語では初めてだったと思います。その後、日本でも実行計画の読解をテーマにした書籍も何冊か出て、雑誌でも特集されるなど「実行計画」(Execution Plan)という語もかなり市民権を得たと思いますし、「実行計画を意識してSQL文を書く」という文化の醸成に、本書の初版は一定の貢献をなしたと思います。

それから10年間の間に、データベースとパフォーマンスの分野においても大きな地殻変動が起きました。まず1つ目は、SSDに代表されるフラッシュストレージの普及、およびメモリの大容量化によってストレージI/Oが劇的に改善されたことです。これはハードウェアの進化によってエンジニアリングが大きく変わった記念すべき事例の一つとなりました。2つ目は、データベースについてもパブリッククラウドのマネージドサービスを利用することが一般化し、アーキテクチャ変更によるチューニング手段が多く出現したことです。スループットを向上させたければ、リードレプリカを簡単に複製できますし、それでも足りなければキャッシュをデータベースの前段に置くという選択を採ることができるようになりました。このようなアーキテクチャの変更を気軽に行うことは、オンプレミス主体だった時代には難しかったことです。

こうしたハードウェアとアーキテクチャの進歩によって、データベースのパフォーマンスチューニングは大きく様変わりしました。筆者が駆け出しエンジニアのころはちょっと重いSQL文が何時間も返ってこないなどということは日常茶飯事でしたが、そんな時代ももう終わったのだなと実感します。では、もはやデータベースにおいてパフォーマンスチューニングや性能を意識した設計というのは過去のものになったのでしょうか。必ずしもそうではない、というのが筆者の考えです。というのは、上に挙げたようなチューニング手段というのは、リソースへ投資することによってパフォーマンスを向上させようというアプローチです。現在では、それがかえって高コスト体質へ結び付く原因ともなっています。「あれ、クラウドに移行したら安くなると思っていたのに思ったようにコスト削減できていないな」と感じる場合の多くは、非効率なリソース割り当てが関係しています。

その状況に対するカウンターとして、現在、クラウド環境でのコスト最適化を行う取り組みとして**FinOps**というコンセプトが登場してきています。データベースのチューニングも今後はその一手段として位置付けられていくと思います。なぜなら、現在でもデータベースが最もリソースを消費するコンポーネントであることに

変わりはないからです。したがって、そこを改善してやることはユーザ体験のみならずコスト削減にもダイレクトに寄与します。従量制が一般化したクラウドネイティブ・アーキテクチャにおいては、なおさらその傾向は強まっています。そのため、純粋にパフォーマンスを良くしたいと思っているエンジニア以外にも、コスト削減に頭を悩ませているエンジニアにとっても本書は有用であると思います。

本書では、さまざまなSQLの例題をもとにどのような書き方がリソース的に効率が良く、かつエレガントで読みやすいコードになるかを考えていきます。興味深いことに、SQLにおいてはその両者は並び立つのです。初版ではまだ多くの実装でサポートされていなかった機能も取り入れてコードをリバイスしました。そして、初版と同様、必ず**実行計画**のレベルでパフォーマンスを検討していきます。その理由は、SQLというのは実行計画を見ないとパフォーマンスの良し悪しは判断できないからです。そのため本書ではほとんど全てのSQL文について実行計画を見ていきます。実行計画は、初めて見ると複雑に見えて敬遠する人もいますが、読み方の基礎さえ覚えてしまえば意外にスラスラ読めるようになります。実装依存の部分もあるのですが、実は実行計画の骨子、いわば「アーキテクチャ」は全てのDBMSに共通しています。本書の目的の一つは、そのエッセンスを学ぶことで「実行計画アレルギー」を克服することにあります。初版ではOracleとPostgreSQLの実行計画をサンプルに使いましたが、第2版ではMySQLも追加しました。クラウド上で提供されるDBaaSもまたMySQLやPostgreSQLをベースにしていることが多いため、その実行計画を読めるようになることはクラウドネイティブ時代を生きるエンジニアとしての実力の涵養(かんよう)にも寄与するでしょう。

データベースの理想に照らせば、本当は、人間が実行計画を見るというのは良くないことなのです。そうした低レイヤの処理をなるべく隠蔽(いんぺい)しようというのがSQLの目指した理念の一つだったからです。しかし、その理想はいまだ実現されていません。いずれはAI(*Artificial Intelligence*、人工知能)を搭載したデータベースが自動的に最適な実行計画を選んでくれる時代が来ると思いますが、それでもなお問題のすべてが解決するわけではありません。本書の中で詳しく見ていきますが、せっかくの豊富なリソースを使い切れない無駄の多いコーディングスタイルを取ると、データベースとSQLの進化も台無しにしてしまう場合もあるからです。そのような意味において、本書の価値はかなり普遍的なものであると思います。本書が皆さんの直面する、あるいは今後向き合うであろう問題の解決に役に立つことを祈っています。

なお、本書のレビューにあたって関口裕士氏、小林隆浩氏の両名にご協力をいただきました。また、本書の企画から一貫して技術評論社の池田大樹氏にお世話になりました。ここに謝辞を記します。

2025年8月　ミック

動作環境

本書に登場するSQL文は、主に以下の環境で動作確認および実行計画の取得を行いました。特定の実装でしか動作しないSQL文については適宜本文中で言及しています。

- **PostgreSQL 17.5**
- **Oracle Database 21c Express Edition**
- **MySQL 8.4.5**

JavaおよびPythonのサンプルコードを実行する際は、以下のサイトより最新版のJDKやドライバをダウンロードしてください。

- **Java**
  - Java SE Development Kit（64bit）
    https://www.oracle.com/jp/java/technologies/downloads/
  - JDBCドライバ
    本文ではPostgreSQL向けのJDBCドライバを使用しています。これは下記よりダウンロードできます。
    https://jdbc.postgresql.org/
- **Python**
  https://www.python.org/downloads/

本文ではPostgreSQL向けのドライバとしてpsycopg2を使用しています。これは下記よりダウンロードできます。

https://pypi.org/project/psycopg2/

サンプルコードのダウンロード

本書で利用しているサンプルコードはWebで公開しています。詳細は本書サポートページを参照してください。補足情報や正誤情報なども掲載しています。

https://gihyo.jp/book/2025/978-4-297-15190-4

本書の構成

本書は全10章と3つのAppendixにより構成されています。

第1章：DBMSのアーキテクチャ――この世にただ飯はあるか

本書の導入として、RDBの内部的な動作に関するモデルを理解します。データキャッシュやワーキングメモリといったメモリ機構とストレージのしくみ、そして何より、SQLのパフォーマンスを理解するためのキー概念である実行計画とそれを構築するオプティマイザの概念を理解します。

第2章：SQLの基礎――母国語を話すがごとく

SQLの基本構文を理解します。検索と更新、分岐、集約、行間比較といったSQLでデータ操作を行うための手段を確認します。本章で学ぶSQL文のさまざまな道具が、第3章以降でパフォーマンスを向上させる際の強力な武器となります。

第3章：SQLにおける条件分岐――文から式へ

SQLにおいて条件分岐を表現する強力な武器であるCASE式が、パフォーマンス改善においても重要な役割を持っていることを、実行計画を読み解くことで明らかにします。

第4章：集約とカット――集合の世界

SQLが体現する集合指向というパラダイムがもたらす考え方の変化を、GROUP BY句や集約関数の使い方を通して体感します。同時に、前章で学習したCASE式と集合指向との組み合わせが、どのようにパフォーマンスへ貢献するのかを理解します。

第5章：ループ――手続き型の呪縛

RDBのパフォーマンスを劣化させる理由の一つが、SQLの集合指向の世界に無理に手続き型のパラダイムを持ち込むことにあります。本章では、その典型的な症状である「ループ依存症」、別名「ぐるぐる系」の怖ろしさについて考察します。

第6章：結合――結合を制する者はSQLを制す

SQLのパフォーマンスが遅いとき、そこにはほぼ必ずと言ってよいほど結合が関係しています。Nested Loops、Hash、Sort Mergeといった結合アルゴリズムの実行計画を読み解くことで、RDBがどのように結合を最適化しようとするかを取り上げます。

第7章：サブクエリ——困難は分割するべきか

問題を小さな規模に分割し、ステップ・バイ・ステップで解決へと至るサブクエリのアプローチは、手続き型に近いものです。これに頼りすぎた場合に引き起こされる性能問題——サブクエリ・パラノイアについて考察し、いかにしてそれを解消するかを論じます。

第8章：SQLにおける順序——甦る手続き型

伝統的に手続き型と相容れないパラダイムを持っていると思われていたSQLですが、近年、再び手続き型の考え方を取り入れる変化が起きています。その動きを象徴するのがウィンドウ関数です。この強力な表現力を持つ関数がSQLにもたらした革命的な進歩を中心に、SQLにおける行の順序を意識したプログラミングを考えます。

第9章：更新とデータモデル——盲目のスーパーソルジャー

パフォーマンスを改善する一番良い手段——それは実は、SQL文を変えることではなく、データモデルを変えることです。ときに忘れられがちな「コロンブスの卵」とも言える選択肢が持つ利点と欠点を明らかにします。

第10章：インデックスを使いこなす——秀才の弱点

パフォーマンスを語るうえで避けて通ることのできないインデックスの利用方法について論じます。どのような条件下で有効に作用するかをインデックスの構造から理解し、そのために必要なデータモデルおよびユーザインタフェースの設計についても取り上げます。

Appendix A：PostgreSQLのインストールと初期設定

学習用の実行環境として、PostgreSQL 17.5のインストールおよび初期設定手順を解説します。

Appendix B：MySQLのインストールと初期設定

学習用の実行環境として、MySQL 8.4のインストールおよび初期設定手順を解説します。

Appendix C：演習問題の解答

各章末の演習問題の解答および解説を行います。

[改訂新版]SQL実践入門——高速でわかりやすいクエリの書き方●目次

第1章

DBMSのアーキテクチャ

この世にただ飯はあるか

> 意思決定に関する最初の原理は，「無料の昼食(フリーランチ)といったものはどこにもない」ということわざに言い尽くされている。自分の好きな何かを得るためには，たいてい別の何かを手放さなければならない。意思決定は，一つの目標と別の目標の間のトレードオフを必要とするのである。
>
> ——Nicholas Gregory Mankiw

本章ではまず、SQLのパフォーマンスを議論するうえで最低限必要になるDBMS(*Database Management System*、データベース管理システム)のアーキテクチャについての知識を解説します。SQLの書き方については次章以降で詳しく見ていきますが、そのベースとして、まずはDBMSと記憶装置の関係、オプティマイザのしくみ、メモリ機構の動き方などを知ってもらいたいと思います。それを通して、DBMSが、多くのトレードオフ(二者択一)のバランスを取るために努力を重ねているソフトウェアであり、私たちもDBMSを利用する際には何を優先して何を捨てるべきか考える必要があることを理解します。

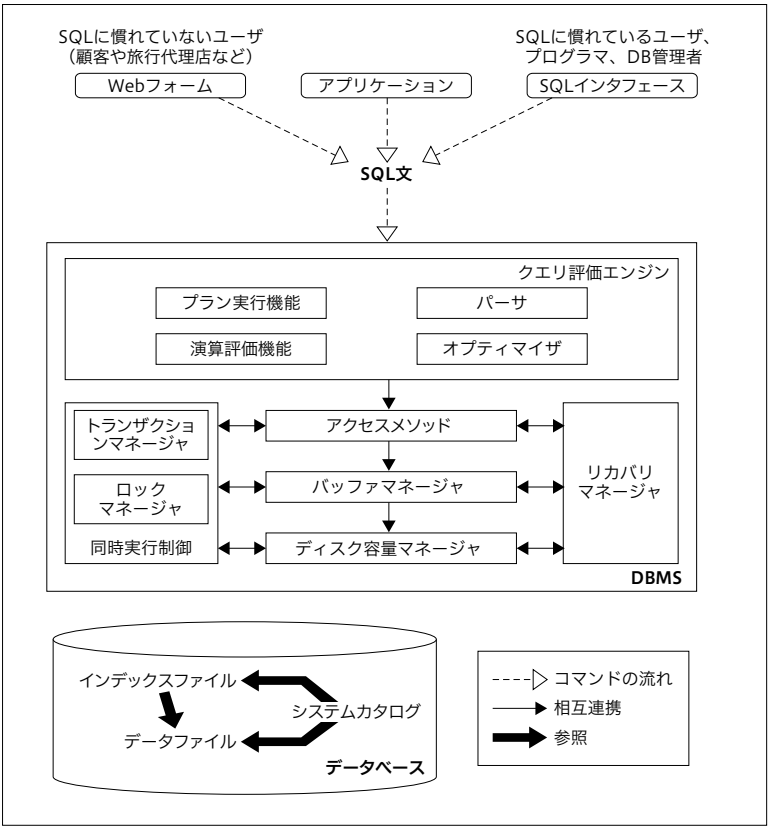
1.1 DBMSのアーキテクチャ概要

現在商用で使われているRDB(*Relational Database*)製品には、数多くの種類があります。日本ではOracle Database(以下Oracleと略します)、PostgreSQL、MySQL、Microsoft SQL Server、Db2といったあたりがよく利用されています。これらの製品はそれぞれ特徴を持っており、内部のアーキテクチャも完全に同じというわけではありません。

しかし、RDBとしての機能を提供するという共通の目的を持っているうえ、リレーショナルモデルの数学的理論を基礎としているわけですから、基本的なしくみはそれほど異なるものではありません。したがって、その共通部分を頭に入れておけば、個々のDBMSの特色は変奏のようなものです。

図1.1は、DBMSの一般的なアーキテクチャの概要を示したものです。

図1.1 DBMSのアーキテクチャ



出典：Raghu Ramakrishnan, Johannes Gehrke, Database Management Systems 3rd ed., McGraw-Hill, 2002. p.20

上段の層が、ユーザやプログラマなど、データベース使用者とのインタフェースを表します。ここから実行されたSQL文が、中段の層であるDBMSに届き、さまざまな処理が実行され、下段の記憶装置に蓄えられたデータに(参照にせよ更新にせよ)アクセスが行われる、という流れです。

私たちの関心は、中段の層であるDBMS内部で行われる「さまざまな処理」にあります。以下、DBMS内の各機能を簡単に見ておきましょう。

クエリ評価エンジン

クエリ評価エンジンは、ユーザから受け取ったSQLを解釈し、どのような手順で記憶装置のデータへアクセスに行くかを決定します。ここで決定された計画を「実行計画」（または「実行プラン」）と呼びます。この実行計画に基づいたデータへのアクセス方法が「アクセスメソッド」です。すなわちクエリ評価エンジンは、プランを立てそれを実行するというDBMSの脳に当たる重要な機能を担っているのです。本書の主題であるパフォーマンスとも非常につながりの深いモジュールです。

なお、「クエリ」（*query*）とは「問い合わせ」という意味の英語で、狭義にはSELECT文のことなのですが、広義にはSQL文全体を指して使います。本書では前者のSELECT文と同じ意味の言葉として使います。

バッファマネージャ

DBMSは、バッファという特別な用途に使うメモリ領域を確保します。そのメモリ領域の使い方を管理するのがバッファマネージャです。ディスクの使い方を管理するディスク容量マネージャ（後述）と連携しながら動きます。このメカニズムもまた、パフォーマンスにとっては非常に重要な役割を果たしています。

ディスク容量マネージャ

データベースはシステムを構成するコンポーネントの中で最大のデータを保存する必要があります。これは、Webサーバやアプリケーションサーバが処理を実行する間だけデータを保持すればよいのに対して、データベースは永続的にデータを保持しなければならないからです。ディスク容量マネージャは、どこにどのようなデータを保存するかを管理し、それに対する読み出し／書き込みを制御します。

トランザクションマネージャとロックマネージャ

商用システムにおいてデータベースを使う人は、普通は一人ではありま

せん。何百人、何千人もの大勢でいっせいにアクセスしています。そうした個々の処理は、DBMS内部では「トランザクション」という単位で管理されます。このトランザクション同士をうまくデータの整合性を保ちながら実行させ、必要とあらばデータにロックをかけて誰かを待機させるといった仕事をするのが、この2つの機能です。

リカバリマネージャ

DBMSが保存するデータには、絶対に失われてはいけない大切なデータが多く含まれています。そうは言っても、システムは使い続けていればいつか障害に見舞われるタイミングがあるものです。そのような有事に備えて、定期的にバックアップを取得し、いざというときにデータを復旧(リカバリ)できる必要があります。この機能を司るのがリカバリマネージャです。

以上はあくまで大雑把な説明です。これだけで理解しろと言うほうが無理ですが、今はすべての機能についてイメージが湧かなくてもかまいません。本書の主題であるSQLのパフォーマンスという観点から見ると、最も重要なのは「クエリ評価エンジン」およびそれが立てる「実行計画」です。本書では、この実行計画のサンプルをいくつも見ながら、SQLがなぜ遅い(または速い)のかを解明していきます。パフォーマンスにとってその次に重要なのが「バッファマネージャ」ですが、これについては本章「DBMSと記憶装置の関係」(7ページ)で詳しく取り上げます。

それ以外の機構は、とりあえず忘れてもらってかまいません。本当はSQLのパフォーマンスという点では「トランザクションマネージャ」と「ロックマネージャ」も重要なのですが、これらはSQL単体というよりも多くのSQLを同時実行する際のパフォーマンスに関係するメカニズムです。本書では1つのSQLを単独で実行した際のパフォーマンスにフォーカスするため、SQLを同時実行した際の競合という観点は取り上げません。

I.2
DBMSとバッファ

まずは、DBMSのバッファマネージャの役割について見ていきます。先述のように、バッファはパフォーマンスに対して重要な役割を担っています。それは、メモリという希少資源に対してデータベースが保存するデータ量は圧倒的に多いため、どのようなデータをバッファに確保するべきかに対するトレードオフを発生させるからです。そのことを理解するため、最初に、システムがデータを保存する記憶装置(ストレージ)について、基本的なことをおさらいしておきましょう。というのも、バッファとストレージは表裏一体、一方を理解するにはもう一方についての知識も必要になるからです。「今さら言われなくても知ってるよ」という人もいると思いますが、復習も兼ねてお付き合いください。

この世にただ飯はあるか

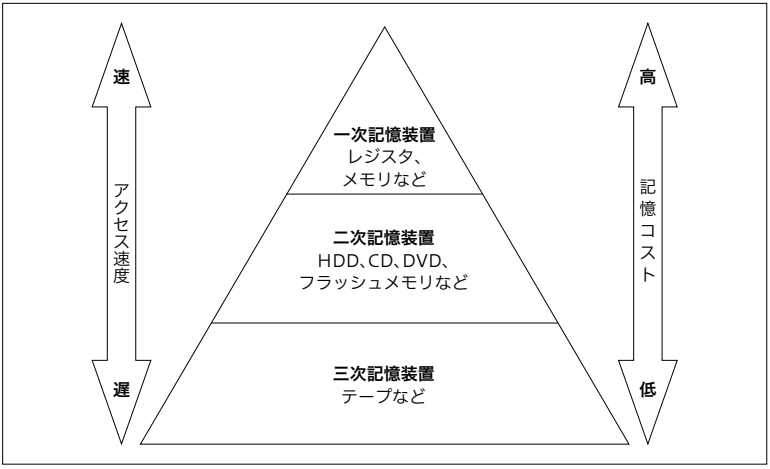
図1.2は、記憶装置の分類を階層化したものです[注1]。

一般的に、記憶装置は記憶コストに応じて一次から三次まで3つの階層に分類されます。「記憶コスト」というのは、思い切って単純化すると、同じデータ量を保存するのにかかるオカネのことです。私たちは普段、PCのHDD(*Hard Disk Drive*)は平気で何TBでも増設しますが、メモリは数GB買うだけでもけっこう悩みます。これは、それだけHDDが安価で大容量データを保存できる(つまり記憶コストが低い)ことを意味します。ピラミッドの下位ほど面積が大きいのは、この「同じコストで保存できるデータ容量の大きさ」を表しています。

それなら、下位階層のHDDやテープが上位層のメモリより優れた記憶装置かと言うと、そういう単純な話ではありません。たしかにこれらの媒体は大量データを永続的に保持するには向いているのですが、データへのアクセス速度という点でメモリに遠く及ばないからです。みなさんも、自

注1　図1.2ではHDDを二次記憶装置と位置付けていますが、このポジションもここ数年で変わりつつあり、三次記憶装置へ移動しつつあります。つまり、HDDは現在ではテープなどのアーカイブ用途のメディアと競合しています。これは、HDDの高速化に限界が見えたため、近年は高密度化・大容量化を目指す傾向があるからです。

図1.2　記憶装置の階層



分のPCで大きなファイルを操作したときなど、「ガリガリガリ」というあのディスクアクセスの特徴的な音とともにマシンがうんともすんとも言わなくなり、長時間待たされた(酷いときはそのまま永遠に近い時間待たされる)というストレスフルな経験をしたことがあると思います(近年はPCにもSSDが普及したことでそれも少なくなりましたが)。

つまりここには、容量と永続性をとれば速度が犠牲になり、速度をとれば容量と永続性が犠牲になる、というトレードオフ(二者択一)の関係が成立しているわけです。良いとこ取りはできません。システムの世界にフリーランチ(ただ飯)は存在しないのです。これが、ストレージについてまず知っていただきたい第一のトレードオフです。

DBMSと記憶装置の関係

DBMSは、重要なデータを保存することを主目的としたソフトウェアですから、記憶装置とは切っても切れない関係にあります。DBMSが使う代表的な記憶装置は次の2つです。

HDD

DBMSがデータを保存する媒体(ストレージ)のかつての主流はHDDでした。ディスク以外の選択肢がまったくない、というわけではないのです

が[注2]、かつてはディスクが第一選択肢でした。最近では、次に述べるSSD（*Solid State Drive*）という高速かつ永続性のある記憶装置が普及したことで主役の座を降りましたが、SSDと組み合わせて利用されることもあり、容量、コスト、パフォーマンスなどの総合的な観点からHDDはまだ現役です。

▪―― SSD

初版刊行（2015年）から現在（2025年）に至るまでにデータベースの世界で起きた最も重要な革命と言えば、SSDに代表されるフラッシュストレージがデータベースのストレージとして普及したことでしょう。SSDは不揮発性メモリの一種に分類され、HDDに比べてデータの読み書き速度が大幅に改善されました（数十倍から数百倍）。SSDには大きく分けてSSD SATAとNVMeという2種類があり、両者は通信ドライバとインターフェイスが完全に異なります。SSD SATAはHDD用に設計されたドライバを使用するのに対して、NVMeドライバはフラッシュ技術を使用するSSD専用に設計されており、転送速度が大きく異なります。SSD SATAの読み出し速度が500MB/s程度で頭打ちになるのに対して、NVMe（現行規格のPCIe 5.0）では10,000MB/s程度まで速度が出せます。

現在では多くのデータベースのストレージはフラッシュストレージです（前述の通り、HDDと組み合わせて使うこともあります）。これによって、従来SQLの最大の難所であったストレージというボトルネックポイントが軽減されました。それは非常に喜ばしいことなのですが、一方でSSDはHDDに比べて高価で大容量の保存に向いていないため、ここでもトレードオフが発生しています。利用の際にはこの点を頭に入れておく必要があります。**表1.1**に両者の比較を掲示しますので参考にしてください。

表1.1　SSDとHDDの比較

項目	SSD（PCIe 5.0想定）	HDD
データの読み出し速度	非常に速い（10,000MB/s程度）	遅い（100MB/s〜200MB/s程度）
アクセス時間	非常に短い（0.1ms未満）	比較的長い（5〜10ms程度）
耐衝撃性	高い（可動部なし）	低い（回転ディスクやアームが衝撃に弱い）
消費電力	低い	高め（回転が必要なため）
価格（GB単価）	高価	安価

注2　たとえばインメモリデータベースは、名前のとおりメモリにデータを保持しますし、データのバックアップをテープなどのメディアに取ることは一般的な運用です。

それでは、DBMSがデータをHDDとSSD以外に保持しないかというと、そうではありません。むしろ、通常のDBMSは、常にストレージ以外の場所にもデータを持つようにしています。それが一次記憶装置である(揮発性の)メモリです。以後、本書で単に「メモリ」と言う場合はこの揮発性メモリを指すものとします。

■――メモリ

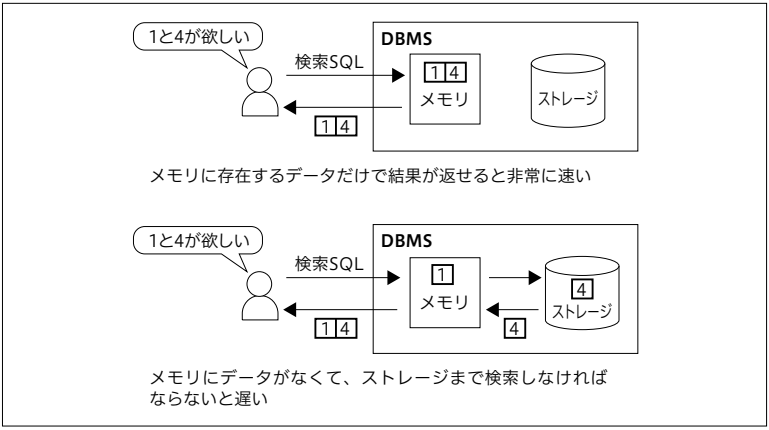
メモリはストレージに比べると記憶コストが高いため、1台のハードウェアに搭載できる量は多くありません。しかし近年、メモリの低価格化も進み、サーバに搭載されるメモリの大容量化が進みました。データベースサーバの場合、搭載されるメモリは一般的な業務システムであれば数十～数百GB、大規模向けシステムであればテラバイト級になるでしょう。以前に比べれば潤沢なメモリが利用できるようになりましたが、それでもHDDやSSDに比べれば相対的に小さいサイズです。このため、ある程度の規模を持つ商用システムでは、データベース内のデータすべてをメモリに載せることは、原則できません(すべてのデータを原則メモリ上に保持するインメモリデータベースのような例外は除きます)。

■――バッファの活用による速度向上

それでも、DBMSが一部でもよいからデータをメモリに載せている理由は、パフォーマンス向上、つまりSQL文の実行速度を速くするためです。図1.2からわかるように、メモリは最も高速な一次記憶装置に該当します[注3]。そのため、頻繁にアクセスされるデータをうまくメモリ上に保持しておくことができれば、同じSQL文を実行するにしても、ストレージからデータを読み出すことなくメモリへのアクセスだけで処理を返すことが可能になるわけです(**図1.3**)。

注3　メモリとHDDの性能差は、大雑把な数値で数十万～百万倍程度と考えてください。

図1.3　メモリにデータがあれば高速に処理できる



ストレージへのアクセスを回避できれば、大きなパフォーマンス改善が可能です。その理由は、一般的にSQL文の実行時間の大半はストレージに対するI/Oに費やされるからです[注4]。

このようにパフォーマンスの向上を目的としてデータを保持する領域を、バッファ(*buffer*)とかキャッシュ(*cache*)と呼びます。バッファとは「緩衝材」という意味です。ユーザとストレージとの間に割って入ることでSQL文のディスクアクセスを減らす役割を果たすわけですから、緩衝材という言葉はぴったりのイメージです。一方キャッシュとは、やはりユーザとストレージの中間に位置することでデータの転送遅延を緩和するための機構です。どちらも物理的な媒体としてはメモリが利用されることが多いため、ストレージ上のデータにアクセスするよりもバッファ(またはキャッシュ)にアクセスするほうが高速です。データベースにおいては、Oracleのデータベースバッファキャッシュ、PostgreSQLの共有バッファ、そしてMySQLのバッファプールをデータキャッシュに分類しましたが(後述の表1.2参照)、実際にはこれらはバッファとキャッシュの役割を兼ね備えています。本書では、便宜的にバッファとキャッシュは互換可能な言葉として使います。

こうした高速アクセス可能なバッファに、どのようなデータをどの程度の期間載せておくかといったことを管理する機能が、DBMSのバッファマネージャ

注4　これはもちろん、すべてのSQLがというわけではなく、全体の傾向としてという意味です。実際、非常に小さなデータにしかアクセスしないSQL文であれば、相対的にストレージのI/OよりもCPUによる演算に時間を要することになります。

というわけです。このように考えると、バッファマネージャがデータベースのパフォーマンスにおいて非常に重要な役割を担っていることがわかると思います。

メモリ上の2つのバッファ

DBMSがデータを保持するために使うメモリには、大きく次の2種類があります。

- **データキャッシュ**
- **ログバッファ**

ほとんどのDBMSが、この2つに該当するメモリ領域を持っています。また、これらのバッファは、ユーザが用途に応じてサイズを変えることもできます。これらのメモリサイズを決めるパラメータを、Oracle、PostgreSQL、MySQLを例に整理したので参考にしてください(**表1.2**)。

表1.2　DBMSのバッファメモリの制御パラメータ

項目		Oracle 23ai	PostgreSQL 17.5	MySQL 8.4.4 (InnoDB)
データキャッシュ	名称	データベースバッファキャッシュ	共有バッファ	バッファプール
	パラメータ	DB_CACHE_SIZE	shared_buffers	innodb_buffer_pool_size
	初期値	**SGA_TARGET が設定されている場合(自動共有メモリ管理が有効)** DB_CACHE_SIZE を明示的に指定しない場合、デフォルト値は0となり、Oracleが内部的に適切なサイズを自動的に決定する。DB_CACHE_SIZE を指定した場合、その値はバッファキャッシュの 最小サイズ として扱われる **SGA_TARGET が設定されていない場合(手動メモリ管理)** 48MB または 4MB × CPU 数のいずれか大きい方	128Mバイト	128Mバイト
	設定値の確認コマンド例	`SELECT value FROM v$parameter WHERE name ='db_cache_size';`	`show shared_buffers;`	`SHOW VARIABLES LIKE 'innodb_buffer_pool_size';`
	備考	SGA 内部に確保される	システムの総メモリの 25% 程度が推奨	システムの総メモリの 80% 以内が推奨
ログバッファ	名称	REDO ログバッファ	トランザクションログバッファ	REDOログバッファ
	パラメータ	LOG_BUFFER	wal_buffers	innodb_log_buffer_size
	初期値	自動的に決定される(通常は数MB)	自動的に決定される(通常は数MB)	16MB
	設定値の確認コマンド例	`SELECT value FROM v$parameter WHERE name ='log_buffer';`	`show wal_buffers;`	`SHOW VARIABLES LIKE 'innodb_log_buffer_size';`
	備考	SGA 内部に確保される	-	InnoDB エンジン使用時のみ有効

■――データキャッシュ

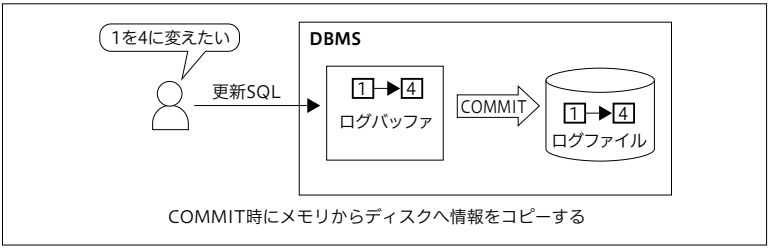
データキャッシュは、まさにディスクにあるデータの一部を保持するためのメモリ領域です。もし、みなさんの実行するSELECT文で選択したいデータが、運良くすべてこのデータキャッシュの中に存在した場合、低速なストレージからデータを読み出すことなく処理が実行されるので、非常に高速なレスポンスが期待できます。

反対に、運悪くバッファ上にデータが見つからなかった場合、はるばるストレージまでデータを取りにいかなければならないため、SQL文のレスポンスが遅くなります。データベースの世界には「**ストレージに触る者は不幸になる**」という古い格言がありますが、その呪いを受けたSQL文のパフォーマンスは、とても待っていられないほど遅くなることも珍しくありません。

■――ログバッファ

ログバッファは更新処理（INSERT、DELETE、UPDATE、MERGE）の実行に関係します。というのも、DBMSはこうした更新SQL文をユーザから受け取ったとき、即座にストレージ上のデータを変更しているわけではないからです。実は、一度このログバッファ上に更新情報を溜めて、ストレージへの更新はあとでまとめて行っています（**図1.4**）[注5]。

図1.4　更新処理はコミットのタイミングで同期処理になる



このようにデータベースの更新処理は、SQL文の実行タイミングとストレージへの更新タイミングにずれがある非同期処理なのです。

単純にSQL文の実行時にストレージ上のファイルを更新するほうが話は

注5　ログファイルへ出力されるタイミングとしてはコミットが一般的ですが、それ以外のタイミングで出力されることもあります。詳細は製品マニュアルなどを参照してください。

単純なのに、DBMSが一度ログバッファで受けてタイミングをずらしている理由は、これも結局パフォーマンスを良くしたいからです。つまり、ストレージは検索だけでなく更新にも相応の時間がかかるため、ストレージの更新が終わるまで待っていると、ユーザを長時間待たせることになるからです。そのため、一度メモリで更新情報を受けた時点で、ユーザにはその更新SQL文は「終わった」と通知しているのです。

この2つのバッファの説明を読んでおわかりかもしれませんが、つまるところ、DBMSというのは「**ストレージの遅さをどうカバーするか**」ということをずっと考え続けてきたソフトウェアです。DBMSは、昔から低速なストレージによるパフォーマンス問題に悩まされてきました。それを克服するために、こうした複雑なバッファのメカニズムを搭載するに至ったのです。逆に言うと、ストレージが速かったならこんな面倒なしくみを考えなくたってよかったのです。いまさら言っても始まらないのですけど。

メモリの性質がもたらすトレードオフ

先ほど、メモリの欠点は高価なので保持できるデータ量が少ないことだ、と言いました。これはもちろん大きな欠点なのですが、もう少し正確を期すと、欠点はほかにもいくつかあります。

■――揮発性とは

メモリにはデータの永続性がありません。ハードウェアの電源を落とせば、メモリ上に載っていたすべてのデータは消えてなくなります。この性質を揮発性と呼びます[注6]。

DBMSを再起動しただけでも、バッファ上のデータはすべてクリアされてしまいます。これはつまり、DBMSに何らかの障害が発生してプロセスダウンが起きた(いわゆる「落ちた」)場合には、メモリ上のデータも消えてしまう、ということです。だから、たとえメモリが非常に安価になったとしても、永続性がない以上、機能的に完全にストレージの代替ができるわけではないのです。

注6　中には電源供給がなくてもデータの失われないメモリ、いわゆる不揮発性メモリもあり、期待されている技術ですが、まだ一般的なサーバには使われません。

■――揮発性の問題点

揮発性の一番困るところは、障害時にメモリ上のデータが消失してしまうことで、データ不整合の原因となることです。データキャッシュであれば、障害によってメモリ上のデータが失われたとしても、オリジナルのデータはストレージ上には残っています。もう一度ストレージから読み出せばよいだけなので、データ不整合の問題は起きません。キャッシュ用のメモリが空っぽの状態であっても、SELECT文はストレージを直接読みにいくだけなので、時間がかかるだけで結果不正は起きません。

しかし、ログバッファ上に存在するデータが、もしストレージ上のログファイルへ反映される前に障害によって消えてしまった場合、そのデータは完全になくなり復旧できません。これは、ユーザが行ったはずの更新情報が消えることを意味します。この障害はビジネス的な観点からは深刻です。完了したはずの銀行振り込みやカードの引き落としが行われていない、なんていうことが発生したら、社会は大混乱に陥ってしまいます。

しかし、ログバッファに溜めた更新情報がDBMSのダウン時に消えてしまう、という現象は、DBMSが更新を非同期処理として行っている以上、起きる可能性はゼロではありません。これを回避するため、DBMSはコミットのタイミングで必ず更新情報をログファイル(これは永続的なストレージ上に存在しています)へ書き込むことによって、障害時のデータ整合性を担保するようにしています。コミットとは、更新処理の「確定」を行うことで、DBMSにはこのコミットされたデータを永続化することが求められます。

逆に言うと、コミットの際は必ずストレージへの同期アクセスが必要になるため、ここで遅延が発生する可能性があるのです[注7]。ここにもまたトレードオフが顔を出しています(**表1.3**)。ストレージへの同期処理をすればデータ整合性と耐障害性は高まるけれど、パフォーマンスは低くなる。パフォーマンスを追求するとデータ整合性と耐障害性は低くなる。悩ましい二者択一に挟まれて、今日もデータベースエンジニアは悩むのです。

注7　ログバッファからディスクへの書き出しが行われるタイミングとしては、コミット以外にもありますが、コミット時には例外なく書き出しが行われます。このルールに対する例外として、PostgreSQLのUNLOGGEDオプションやOracleのNOLOGGINGモードがあります。これらは更新SQLの実行時にREDOログファイルへの書き出しを行わないことで高速化を実現します。障害時に外部データから復旧させられる手段がある場合に、大量データのロード高速化などの目的で利用されます。

表1.3　データ整合性とパフォーマンスはトレードオフ

名前	データの整合性	パフォーマンス
同期処理	○	×
非同期処理	×	○

なお、データベースは、更新SQLによる変更を処理するにあたって、ログバッファからログファイルへの書き込み以外にも、データキャッシュへ変更分を書き溜めておいて、一定間隔でストレージ上のデータファイルへ書き出すという動作も行っています。データキャッシュ内の更新済みデータ(更新後のデータは、ダーティブロックやダーティページなどと呼ばれます)のストレージへの書き出しは、コミットのタイミングとは別に非同期に行われます。この処理を**チェックポイント**(checkpoint)と呼びます。チェックポイントが非同期なのも、ストレージ負荷を考慮してパフォーマンスへ配慮しているからです。これに対し、ログバッファからログファイルへの書き出しは、チェックポイントに先行して行われることから、**WAL**(*Write-Ahead-Log*：ログ先行書き込み)と呼ばれます。

システムの特性によるトレードオフ

■――データキャッシュとログバッファのサイズ

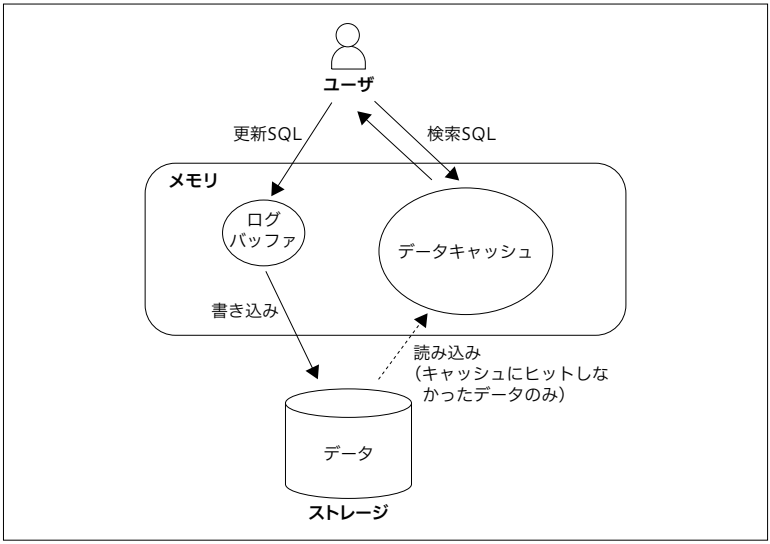
ところで、前出の表1.2のデータキャッシュとログバッファを比較してみると、3つのDBMSに共通して、データキャッシュに比べてログバッファのデフォルト設定値が相対的に小さいことに気付きます。OracleやPostgreSQLなど、ログバッファは10MBにも達しません。こんなに小さくて大丈夫なのでしょうか？

大丈夫かそうでないかは、性能試験をやってみるまでわかりません。ただ、データベースが2つのバッファに対してこのように極端に非対称なサイズ割り当てを行っているのには、明確な理由があります。それは、データベースが、基本的に検索をメインの処理と想定しているソフトウェアだということです。

検索処理においては、検索対象のレコードが数千万件、数億件というオーダーになることも珍しくありません。他方、更新処理において変更対象とされるデータは、せいぜいトランザクションあたり1～数万件です(もちろんトランザクションの規模によって差はありますが)。そのため、更新に貴重なメモリを多く割くよりは、可能な限り多く検索処理でヒットしそう

なデータをキャッシュに載せておくほうが得策だ、というのがデータベースの基本精神になっているのです(**図1.5**)。

図1.5　データベースは検索を重視したメモリ配分をしている



実際多くのDBMSが、物理メモリに余裕があればデータキャッシュになるべく多く割り当てることを推奨しています[注8]。

もちろんこれは、データベースの作り手側が(言葉は悪いですが)勝手にそう決めてかかっているだけなので、もしみなさんが作るシステムが、検索に比べて更新量の多い業務特性を持っているならば(たとえばバッチ処理がメインの場合)、デフォルト設定のままでは更新処理のパフォーマンスが出ないということもあり得ます。そのときは、ログバッファにより多くのメモリを割り当てるといったチューニング(最適化)が必要になることは言うまでもありません。著者自身、バッチ処理による大量のデータ更新を行う特性を持ったシステムに対するチューニングでは、ログバッファを拡張

注8　たとえば、MySQLはマニュアルで「サーバがデータベース専用ならば、物理メモリの80%をバッファプールに割り当ててもよい」と示唆しています。「データベース専用ならば」という条件付きなのは、もちろん同じサーバでほかのアプリケーションが動いている場合は、そちらのメモリ使用量も考慮する必要があるからです。

・「MySQL 8.4 Reference Manual :: 17.14 InnoDB Startup Options and System Variables」
https://dev.mysql.com/doc/refman/8.4/en/innodb-parameters.html

したことがあります。

■――検索と更新、大事なのはどっち

つまり、ここで私たちは、**検索と更新のどちらを優先すべきか**というトレードオフに直面しているのです。メモリという高価で希少な資源は、すべてのデータをカバーするには足りません。したがって、何を優先して守り、何を捨てるかという判断が必要になるのです。

もちろん、システムにかかる負荷に対して、相対的にメモリが潤沢に余っているならばこういう悩みは生じません。データキャッシュとログバッファの両方に十分なメモリを割り当てればよいでしょう。また最近のDBMSはかなり進歩していて、リソースを自動調整する機能が充実してきています。メモリ割り当ても自動判断してくれるDBMSもあります。ただ、現状それにもやはり限界はあります。厳しいリソース配分の計算が必要な局面では、何も考えずに自動設定に頼ることは危険な行為です。

そうしたとき適切な判断を下すためにも、そのデータベースがどのような思想に基づいてリソース配分が行われたのかを理解することは大切なことです。もしログバッファが大きく取られていれば、それは高負荷の更新処理を想定した設計を行っているということがわかりますし、反対にデータキャッシュが大きく取られていれば、検索処理のレスポンスを重視していることがわかるわけです。

もう一つのメモリ領域「ワーキングメモリ」

■――いつ使われるか

DBMSは、上で説明した2つのバッファ以外に、通常はもう一つのメモリ領域を持っています。それが、ソートやハッシュなど特定の処理に利用される作業用の領域、ワーキングメモリです。ソートは、ORDER BY句、あるいは集合演算やウィンドウ関数などの機能を使用する際に実行されます。一方、ハッシュは主にテーブル同士の結合でハッシュ結合が使用された場合に実行されます[注9]。

注9　最近はGROUP BYでもハッシュのアルゴリズムが使われることがあります。またハッシュ結合については第6章で詳しく見ます。

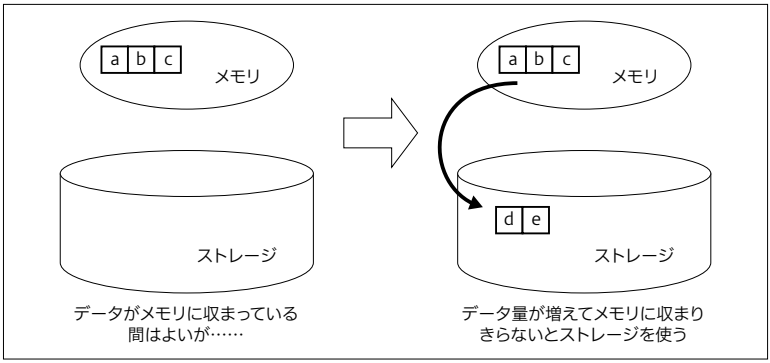
このメモリ領域の名称や管理方法はDBMSによって異なります。たとえばOracle、PostgreSQL、MySQLでは、それぞれ**表1.4**のような名称で呼ばれています[注10]。

表1.4　ワーキングメモリの各DBMSでの呼び方と設定

DBMS	名称	パラメータ	デフォルト値
Oracle 23ai	PGA（Program Global Area）	PGA_AGGREGATE_TARGET	10MB、またはSGAサイズの20%のいずれか大きいほう
PostgreSQL 17.5	ワークバッファ	work_mem	4MB
MySQL 8.4	ソートバッファ	sort_buffer_size	256KB
	ジョインバッファ	join_buffer_size	256KB

この作業用のメモリ領域は、SQLでソートやハッシュが必要になったときに使用され、終われば解放される文字どおり一時的な領域で、通常はデータキャッシュやログバッファとは別の領域として管理されていることが多いです。この領域が性能的に重要な理由は、もしこの領域が扱うデータ量に対して小さく不足した場合は、多くのDBMSがストレージを使用するためです（**図1.6**）。これは、OSの動作で言うところのスワップ（*swap*）のようなものです。スワップ動作は、物理メモリ（RAM）が不足した際に、使用

図1.6　ワーキングメモリが不足するとストレージが使われる



注10　詳細はそれぞれ次のマニュアルを参照してください。
・「Oracle Database データベース・リファレンス Release 23」
https://docs.oracle.com/cd/G11854_01/refrn/PGA_AGGREGATE_TARGET.html
・「PostgreSQL PostgreSQL 17.5 文書 19.4. 資源の消費」
https://www.postgresql.jp/document/17/html/runtime-config-resource.html
・「MySQL 8.4 Reference Manual 17.14 InnoDB Startup Options and System Variables」
https://dev.mysql.com/doc/refman/8.4/en/innodb-parameters.html

頻度の低いデータを一時的にストレージ上のスワップ領域に退避させる仕組みです。これにより、メモリの空き領域を確保し、システムの安定性を維持します。

多くのDBMSが、ワーキングメモリが溢れたときに使用する一時領域をストレージ上に持っています。これはたとえば次のような名前で呼ばれています。

- **Oracle：一時表領域（TEMP表領域）**
- **Microsoft SQL Server：TEMPDB**
- **PostgreSQL：一時領域（pgsql_tmp）**
- **MySQL：一時ファイル**

こうした領域が使用されることを、通称「**TEMP落ち**」と言います。この一時領域はストレージ上に確保されるため、当然アクセス速度はメモリに比べてはるかに低速です。

■――ワーキングメモリが不足すると何が起きるのか

ストレージが使われると何が起きるのでしょうか？ 前述のとおり、ストレージはメモリに比べて非常に低速です。そこにアクセスすると……そう、スローダウンが起きてしまうわけです。もちろん、メモリが不足することで処理が止まったりエラーが起きたりすることに比べれば、まだスローダウンのほうがマシではあるのですが、この種のスローダウンのやっかいなところは、データがメモリに収まっている間は非常に高速なのに、メモリから溢れた瞬間に遅くなる、という劣化が（突然）起きてしまうことです。このメモリ領域は各SQL文の実行により必要とされたサイズが確保されます。したがって、MySQLやPostgreSQLで多数のSQL文が同領域を要求した場合、確保される領域の総量が物理メモリ以上となり、スワップによって全体的に性能への影響が発生します。Oracleのようにワーキングメモリ（PGA）が自動管理されている場合には、各SQL文が利用できる領域サイズが縮小（自動調整）されTEMP落ちしやすくなります。この問題は、複数のSQLが競合する状態を再現する試験（負荷試験）を実施しないと判明しないという難しさがあります。したがって、よく注意しなければ、カットオーバー後に本番環境で性能劣化が発生することになりかねません。

DBMSのこうした複雑なメカニズムを、みなさんはやっかいなものだと

考えるかもしれません。しかし、これは逆に考えることもできます。つまり、DBMSはたとえメモリが不足しようとも何とか処理を継続しようと努力するソフトウェアだ、ということです。実際、DBMSとしては、ワーキングメモリが不足したとき、さくっとそのSQL文をエラーにして処理を中断してしまうという選択肢もあり得るのです。たとえば、JVMのヒープサイズが不足したときに、Javaアプリケーションがメモリ不足(*Out of Memory*)エラーによって処理を異常終了させるように。しかしながら、データベースはそのような選択をしません。SQL文をエラーにするぐらいなら、たとえ遅くなってもよいから何とかして処理を完了させるよう努力します。これは、DBMSが重要なデータを保管し、それを処理するがゆえに、OSに準じるレベルで処理継続性を担保しようとしているからです。

このワーキングメモリの機構は、第4章でGROUP BY句、第6章でハッシュ結合を取り上げたとき再び登場することになるので、覚えておいてください。

1.3 DBMSと実行計画

Web画面の入力フォームであれコマンドラインツールであれ、ユーザインタフェースにかかわらず、RDBに対する操作はSQLという専用の言語で行われます。ユーザや開発者が意識的に記述するのは通常このSQLレベルまでで、あとはSQL文を受け取ったDBMSが処理を行い、結果が返却されるのを待つのみです。ユーザはデータのありかを知る必要もなければ、そこへのアクセス方法も考えません。そういう仕事は、全部DBMSに任せています。

このプロセスは、通常「プログラミング」と呼ばれるものとはかなり異なります。普通、データの検索や更新をプログラミング言語によって行う場合、どこにあるデータをどのように探すか、という手続きを細部に渡って記述しなければいけません。しかし、SQLにおいてはそのような手続きは一切現れません。

権限委譲の功罪

この態度の違いは良いとか悪いとかいうものではなく、言語の設計思想の違いです。C言語、JavaからPythonに至るまで、手続き型を基礎とする言語においては、ユーザがデータアクセスのための手段(How)を責任持って記述することが前提です。他方、非手続き型であるSQLは、その仕事をユーザからシステム側に移管しました。その結果、ユーザのすることは対象(What)の記述だけに限定されたのです。

RDBがこのような大胆な権限委譲を断行したことには、もちろん正当な理由があります。それは、「そのほうがビジネス全体の生産性は上がるから」です。現在の状況を眺めてみると、この言葉は半面正しく、半面間違っていました。正しかったことは、RDBがシステムの世界の隅々にまで浸透したことからわかります。間違っていたことは、それでもやはり私たちはRDBを扱うのに苦労していることからわかります。SQLは思ったほど簡単な言語ではなかったし、Howを意識しないことによるSQL文のパフォーマンスに悩まされることもしばしばです。

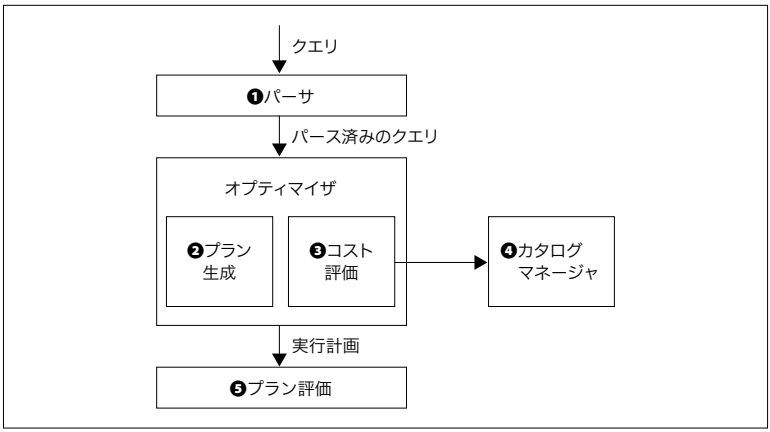
私たちが(不本意ながらも)RDBがいったんは隠蔽したはずの内部の手続きを再度のぞき見なければならないのは、このような理由によるのです。

データへのアクセス方法はどう決まるのか

先述のとおり、RDBにおいてデータアクセスの手続きを決めるモジュールは、クエリ評価エンジンと呼ばれます。これは、ユーザから送信されたSQL文(クエリ)を最初に受け取るモジュールでもあります。クエリ評価エンジンは、さらにパーサやオプティマイザといった複数のサブモジュールから構成されています。

クエリがどのように処理されて、実際にデータアクセスが実行されるのかを大まかに図示すると、**図1.7**のようになります。

図1.7　DBMSのクエリ処理の流れ



出典：Raghu Ramakrishnan, Johannes Gehrke, *Database Management Systems 3rd ed.*, McGraw-Hill, 2002, p.405.

■――パーサ(*parser*)

パーサ(❶)の役割は、名前のとおりパース(構文解析)です。つまり受け取ったSQL文を一度バラバラの要素に分解し、それをDBMSが処理しやすい形式に変換することです。

なぜこの処理が最初に必要かと言えば、第1の理由は、受け取ったSQL文が常に構文的に適正である保証がないため、整合性チェックが必要だからです。ユーザがカンマを書き忘れたり、FROM句に存在しないテーブル名を書いたりしてきたときには、「書類審査」で落第させる必要があります。第2の理由は、SQL文を定型的な形式に変換することで、DBMS内部での後続の処理が効率化されるからです。構文解析は、SQLに限らず一般のプログラミング言語のコンパイル時にも同様に実行されるものです。

■――オプティマイザ(*optimizer*)

書類審査をパスしたクエリは、次にオプティマイザに送られます。オプティマイズの和訳に「最適化」という語が当てられているとおり、ここで「最適」なデータアクセスの方法(実行計画)が決定されます。この処理がDBMSの頭脳におけるコアです。

オプティマイザはルールベースとコストベースという2つの動作モードがありますが、現在ではほとんどのDBMSでコストベースがデフォルトで採

用されているので、以下コストベースを前提に話を進めます。コストベースでは、オプティマイザは、「インデックスの有無」「データの分散や偏りの度合い」「DBMSの内部パラメータなどの条件」を考慮して、選択可能な多くの実行計画を作成し(❷)、それらのコストを計算して(❸)、最も低コストな1つに絞り込みます。オプティマイザがどのようなSQLに対してどのような実行計画を立てるのかについては、本書の後半で多くの実例を取り上げます。

RDBがデータアクセスの手続き決定を自動化している理由は、アクセスパスの候補の数が多いうえに、それら個々のプランについてしらみつぶしにコスト計算をして、互いを比較考慮しなければならないためです。このような計算は、人間よりコンピュータのほうが高速に処理可能なので、一般的にはオプティマイザに任せるのが得策です。

■――カタログマネージャ(*catalog manager*)

オプティマイザが実行計画を立てる際、オプティマイザに重要な情報を提供するのがカタログマネージャ(❹)です。カタログとはDBMSの内部情報を集めたテーブル群で、テーブルやインデックスの統計情報が格納されています。そのため、このカタログの情報を単に「統計情報」とも呼びます。本書でもこの呼び名を使います。

■――プラン評価(*plan evaluation*)

オプティマイザがSQL文から複数の実行計画を立てたあと、それを受け取って最適な実行計画を選択するのがプラン評価(❺)です。あとで実際にいくつかのサンプルを見ていきますが、実行計画というのはまだそのままDBMSが実行できるようなコードにはなっていません。むしろ人間が読むことのできる、文字どおり「計画書」です。したがって、パフォーマンスの悪いSQL文については、この実行計画をエンジニアが読むことによって補正案を考えることもできるのです。

こうして一つの実行計画に絞り込まれたあと、DBMSは実行計画を手続き型のコードに変換してデータアクセスを実行することになります。

オプティマイザとうまく付き合う

以上が、DBMSがクエリを受け取ってから実際のデータアクセスを行う

までの流れです。オプティマイザ内部の処理については、このエンジンそのものを実装するエンジニア以外には関係しないため、本書では扱いません。むしろデータベースのユーザとしては、このオプティマイザをうまく使ってやることのほうが大事です。というのも、オプティマイザは放っておけば万事よろしくやってくれるほど万能ではないからです。特に、カタログマネージャ(❹)が管理する統計情報については、データベースエンジニアは常に神経を使う必要があります。

というのも、プラン選択をオプティマイザ任せにしている場合、現実には最適なプランが選ばれないことが多々あるからです。オプティマイザが失敗する代表的なパターンはいくつかありますが、中でも最も初歩的かつありがちなのが、統計情報が不適切なケースです。

実装によって差はありますが、カタログに含まれている統計情報は、例えば次のようなものです。

- **各テーブルのレコード数**
- **各テーブルの列数と列のサイズ**
- **列値のカーディナリティ(値の個数)**
- **列値のヒストグラム(どの値がいくつあるかの分布)**
- **列内にあるNULLの数**
- **インデックス情報**

これらの情報を入力として、オプティマイザは実行計画を作ります。問題が起こるのは、このカタログ情報がテーブルやインデックスの実体と一致しない場合です。テーブルに対してデータの挿入／更新／削除が行われたのにカタログ情報が更新されていないと、オプティマイザは古い情報をもとに実行計画を作ろうとします。オプティマイザの手元にはそれしか情報がないのだから、しかたありません[注11]。

たとえば極端な例ですが、テーブルを作ったばかりのレコード0件の状態でカタログ情報が保存され、その後レコードを1億件ロードしたのにカ

注11　統計情報が一度も収集されていなかったり失効したりしていた場合、クエリ実行時に統計情報をリアルタイムに収集する機能を持つDBMSもあります。これをJIT(*Just In Time*)統計と呼びます。JITは、鮮度の高い情報が得られるのがメリットですが、欠点もあります。1つ目は、統計情報収集はそれ自身かなり時間のかかる処理なので、JIT自身が遅延するという本末転倒になる危険があること、2つ目が、だからと言ってJITを高速化するために集める情報を制限すると、精度の低い統計情報しか集められないことです。お気付きのように、ここにおいても、問題はまたしてもトレードオフなのです。

タログ情報を更新しなかった場合、オプティマイザはデータ0件を前提してプラン生成をしようとします。これでは最適なプランは到底期待できません。「Garbage In, Garbage Out」（ゴミのような入力からはゴミのような結果しか生まれない）というやつです。それでSQL文が遅かったからといって、オプティマイザのせいにするのは酷です。

適切な実行計画が作成されるようにするには

正しい統計情報を集めることは、SQLのパフォーマンスにとって死活問題なので、テーブルのデータが大きく更新されたらカタログの統計情報もセットで更新することは、データベースエンジニアの間では常識です。マニュアルで更新するだけでなく、データを大きく更新するバッチ処理の場合はジョブネット[注12]に組み込む場合も多いですし、Oracleのようにデフォルト設定で定期的に統計情報更新のジョブが動いたり、Microsoft SQL Serverのように更新処理が行われたタイミングで自動的に統計情報を更新するDBMSもあります。

統計情報の更新は、対象のテーブルやインデックスのサイズと数によっては数十分〜数時間を要することがあり、実行コストの高い作業ではあります。しかし、DBMSが適切なプランを選択するための必要条件ですので、更新タイミングは手を抜かずに検討する必要があります。

代表的なDBMSの統計情報更新コマンドの一覧を**表1.5**に掲載します。ここに掲載するのは基本構文のみで、オプションのパラメータによってテーブル単位ではなくスキーマ全体で取得したり、サンプリングレートを指定したり、テーブルに付与されているインデックスの統計情報もあわせて取得したりなど、さまざまな制御が可能です。詳細は各DBMSのマニュアルを参照してください。

注12 個々のジョブの実行シーケンスのことです。業務的な前後関係やパラレル実行可能かどうかなどを考慮して組み上げます。

表1.5　代表的なDBMSの統計情報更新コマンド

名前	コマンド
Oracle	EXEC DBMS_STATS.GATHER_TABLE_STATS(OWNNAME => スキーマ名, TABNAME =>テーブル名, CASCADE=>true, NO_INVALIDATE=>false);
Microsoft SQL Server	UPDATE STATISTICS テーブル名;
Db2	RUNSTATS ON TABLE スキーマ名.テーブル名;
PostgreSQL	ANALYZE スキーマ名.テーブル名;
MySQL	ANALYZE TABLE スキーマ名.テーブル名;

I.4 実行計画がSQL文のパフォーマンスを決める

実行計画が作られると、DBMSはそれをもとにデータアクセスを行います。しかし、データ量の多いテーブルへアクセスしたり複雑なSQL文を実行したりする場合、レスポンスの遅延に遭遇することがよくあります。この理由には、先述のように統計情報が不正確であるというケースもありますが、現状最適なアクセスパスが選択されているのに遅い、ということもあります。また、統計情報は最新でも、SQL文が複雑過ぎてオプティマイザが最適なアクセスパスを生成できないこともあります。

実行計画の確認方法

こうした理由から、SQL文の遅延が発生したとき、最初に調べるべき対象は実行計画となります。どんなDBMSも、実行計画を調べる手段を提供しています。実装によって違いますが、多くのDBMSがコマンドラインのインタフェースから確認する手段を用意しています(**表1.6**)。

表1.6　実行計画を確認するコマンド

名前	コマンド
Oracle※	set autotrace traceonly
Microsoft SQL Server※	SET SHOWPLAN_TEXT ON
Db2	EXPLAIN ALL WITH SNAPSHOT FOR SQL文
PostgreSQL	EXPLAIN SQL文
MySQL	EXPLAIN FORMAT=TREE SQL文

※OracleおよびMicrosoft SQL Serverでは、上記コマンドのあとに対象のSQL文を実行します。いずれのDBMSにおいても、確認コマンドとSQL文の間には改行を入れてもかまいません。また、Oracleではこれから、次の3つの基本的なSQL文に対する実行計画を見てみましょう。

❶テーブルフルスキャンの実行計画

❷インデックススキャンの実行計画

❸簡単なテーブル結合の実行計画

具体的に、**図1.8**のようなサンプルテーブルを使うことにします。ある業種の店舗についての評価や所在地域のデータを保存するテーブルだと考えてください。主キーは店舗IDで、テーブルには60行のデータを入れ、そのあとに統計情報を取得済みだと仮定します。

図1.8　店舗テーブル

Shops（店舗）

shop_id（店舗ID）	shop_name（店舗名）	rating（評価）	area（地域）
00001	○○商店	3	北海道
00002	△△商店	5	青森県
00003	××商店	4	岩手県
00004	□□商店	5	宮城県
略			
00060	☆☆商店	1	東京都

※以降テーブルのデータを図示する場合には、表の左上にテーブル名を表記しています。また、列名においてアンダーラインが引かれている列は、主キー（プライマリキー）であることを示します。主キーは、テーブル内のレコードを一意に特定することのできる列の組み合わせです。

テーブルフルスキャンの実行計画

まずは、レコードを全件取得する単純なSQL文の実行計画を見てみましょう。

```
SELECT *
  FROM Shops;
```

PostgreSQLとOracleおよびMySQLで取得した実行計画を掲載します(**図1.9**、**図1.10**、**図1.11**)。なお、本書では以降、実行計画の読みやすいこの3つのDBMSの実行計画をサンプルとして使います。

図1.9　テーブルフルスキャンの実行計画(PostgreSQL)

```
EXPLAIN
SQL文を実行

------------------------------------------------------
Seq Scan on shops (cost=0.00..1.60 rows=60 width=22)
```

図1.10　テーブルフルスキャンの実行計画(Oracle)

```
set autotrace traceonly
SQL文を実行

------------------------------------------------------------------------
| Id | Operation         | Name  | Rows | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------
| 0  | SELECT STATEMENT  |       | 60   | 1260  |      3  (0)| 00:00:01 |
| 1  | TABLE ACCESS FULL| SHOPS | 60   | 1260  |      3  (0)| 00:00:01 |
------------------------------------------------------------------------
```

図1.11　テーブルフルスキャンの実行計画(MySQL)

```
EXPLAIN FORMAT=TREE
SQL文を実行

| -> Table scan on Shops  (cost=6.25 rows=60)
```

※以降で実行計画を掲載するときは、実行計画を確認するコマンドとSQL文は省略します。

実行計画の出力フォーマットは完全に同じではありませんが、どのDBMSにも共通する項目があります。それは、次の3つです。

❶操作対象のオブジェクト
❷オブジェクトに対する操作の種類
❸操作の対象となるレコード数

この3つは全てのDBMSの実行計画に含まれています。それだけ重要な項目だということです。

■――操作対象のオブジェクト

1つ目の対象オブジェクトについて見ると、PostgreSQLとMySQLは「on」のあとに、Oracleは「Name」列にSHOPSテーブルが出力されています。サンプルのSQL文がこの1つのテーブルしか使用していないため今は迷うことはありませんが、複数のテーブルを使用するSQL文では、どのオブジェクトに対する操作なのか混同しないよう注意が必要です。

また、この項目にはテーブル以外にも、インデックスやパーティション、シーケンスなど、SQL文でアクセス対象となるオブジェクトなら何でも現れる可能性があります[注13]。

■――オブジェクトに対する操作の種類

2つ目のオブジェクトに対する操作は、実行計画で最も重要な項目です。PostgreSQLとMySQLは文頭の単語、Oracleでは「Operation」列が示します。PostgreSQLの「Seq Scan」は「シーケンシャルスキャン」の略で、ファイルを順次(シーケンシャルに)アクセスして当該テーブルのデータを全部読み出す、という意味です。Oracleの「TABLE ACCESS FULL」は、「テーブルのデータを全部読み込む」という意味です。MySQLの「Table scan」も当該テーブルのデータを全部読み出すという意味です。

これらの記述は、厳密には同じレベルの出力にはなっていません。テーブルのデータを全部読み込む方法として、必ずしもシーケンシャルスキャンを選択しなければならない、というわけではないからです。つまり、PostgreSQLの出力のほうが残りの2つより物理レベルに近い出力です。しかし実際のところ、OracleにおいてもMySQLにおいても、テーブルへのフルアクセスを行う場合は内部的にシーケンシャルスキャンが実施されるので、この2つの操作はほぼ同義と考えてかまいません。このタイプのテーブルへのフルアクセスを、本書では「テーブルフルスキャン」と呼ぶことにします。

■――操作の対象となるレコード数

3つ目の重要な項目は、操作の対象となるレコード数です。これらはい

注13　もちろん一番多く現れる可能性があるのがテーブル、その次にインデックスであることは言うまでもありません。

Column

実行計画の「実行コスト」と「実行時間」

オブジェクト名やレコード数といった指標に比べて、出力に含まれる実行コスト（Cost）という指標は、評価の難しい項目です。これはデータベースがクエリを実行する際にかかると予想されるリソース使用量を数値化したものです。一見するとこの数値を減らすことが良いことのように思えますし、それは大筋において間違いではないのですが、値の大小を絶対評価することは困難です[注a]。あくまで相対評価において、ある程度の目安にしかなりません。

また、Oracleが出力している「Time」列も、あくまで推計の実行時間なのであまりあてになりません。このように、実行計画に出力されるコストや実行時間、処理行数は推計値なのであまり鵜呑みにできないのですが、こうした値について、実測値を取得する方法を用意している実装もあります。

たとえばOracleでは、SQL文の実際の実行計画（Actual Plan）を取得する方法（DBMS_XPLAN.DISPLAY_CURSOR）もあるので、その場合には本当にかかった処理時間を操作ごとに出力できます。たとえば、インデックスによるアクセスのSQL文では、**図1.a**のような実行計画が得られます[注b]。

図1.a　DBMS_XPLAN.DISPLAY_CURSORによる実行計画の取得

```
set serveroutput off
alter session set statistics_level=all;

SELECT *
  FROM Shops;

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(format=>'ALL ALLSTATS LAST' ));

-----------------------------------------------------------------------------------------------------------
| Id  | Operation         | Name  | Starts | E-Rows |E-Bytes| Cost (%CPU)| E-Time   | A-Rows |   A-Time   | Buffers |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |       |      1 |        |       |     2 (100)|          |     60 |00:00:00.01 |       7 |
|   1 |  TABLE ACCESS FULL| SHOPS |      1 |     60 |  5280 |     2   (0)| 00:00:01 |     60 |00:00:00.01 |       7 |
-----------------------------------------------------------------------------------------------------------
```

注a　たとえば「このSQL文はCostが5,000しかないので1秒以内で終わるでしょう」という推測はできません。

注b　もちろん、SQL文を実際に実行する必要があるため、あまり実行時間の長いSQL文には不向きです。OracleやMySQLでは、Actual Plan取得中にキャンセルした場合、その時点まで実行したActual Planを出力する仕様であるため、途中でキャンセルしても遅延部分の特定に有用な情報が得られることがあります。一方、PostgreSQLのexplain（analyzeを含む）は、キャンセルの仕様で途中結果を返せないので、なにも出力されません。

列の説明は次のとおりです。

- **E-Rows：推計の操作行数**
- **A-Rows：実際の操作行数**
- **A-Time：実際の実行時間**

Eは「Estimate」、Aは「Actual」の略です。本書は特定の実装の解説は目的としていないため、これ以上の詳細については製品のマニュアルを参照してください。

ずれも「Rows」の項目に出力されます。結合や集約が入ってくると、1つのSQL文を実行するだけでも複数の操作が行われます。そうすると、各操作でどれだけのレコードが処理されるかが、SQL文全体の実行コストを把握するために重要になります。

なお、この件数に関して1つ誤解してほしくないことがあります。それは、この数値が取得されている情報源です。先ほど「データへのアクセス方法はどう決まるのか」(21ページ)でオプティマイザが実行計画を作る過程を解説した際にも説明したように、オプティマイザはテーブル情報をカタログマネージャから受け取ります。つまり、ここに出ている件数は、統計情報として取得された値がもとになっています。そのため、SQL文を実行した時点のテーブル件数とは乖離(かいり)がある場合もあります(JITの場合は別ですが)。

ためしに、このShopsテーブルから全レコードを削除して(もちろんコミットもして)、それから再度実行計画を取得したらどうなるでしょう。実際にやっていただければわかりますが、OracleでもPostgreSQLでも、やはりRowsの項目には変わらず「60」という値が出力されます。これは、オプティマイザが、あくまで統計というメタ情報を頼りにしていて、実テーブルは見ていない証拠です[注14]。

注14　「そんな不正確なことしなくても、SQL文実行時にテーブル件数を数えるJIT処理をすればよいのに」と思うかもしれませんが、もし1億件オーダーのレコードが入っているテーブルにJITを行う場合、Rowsの値を厳密に取得しようとすれば、それだけでも数分を要することもあります。注11 (24ページ)も参照してください。

インデックススキャンの実行計画

今度は、先ほど実行した簡単なSQL文にWHERE条件を付けてみましょう。

```
SELECT *
  FROM Shops
  WHERE shop_id = '00050';
```

再度実行計画を取得すると、**図1.12**、**図1.13**、**図1.14**のようになります。

図1.12　インデックススキャンの実行計画（PostgreSQL）

```
-----------------------------------------------------
Index Scan using pk_shops on shops  (cost=0.00..8.27 rows=1 width=320)
   Index Cond: (shop_id = '00050'::bpchar)
```

図1.13　インデックススキャンの実行計画（Oracle）

```
-------------------------------------------------------------------------------------
| Id  | Operation                    | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |          |     1 |    28 |     1   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| SHOPS    |     1 |    28 |     1   (0)| 00:00:01 |
|*  2 |   INDEX UNIQUE SCAN          | PK_SHOPS |     1 |       |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("SHOP_ID"='00050')
```

図1.14　インデックススキャンの実行計画（MySQL）

```
 -> Rows fetched before execution  (cost=0..0 rows=1)
```

今度の実行計画には、おもしろい変化が見られます。先ほどと同様、3つの項目に注目して見てみましょう。

■――操作対象のオブジェクトと操作

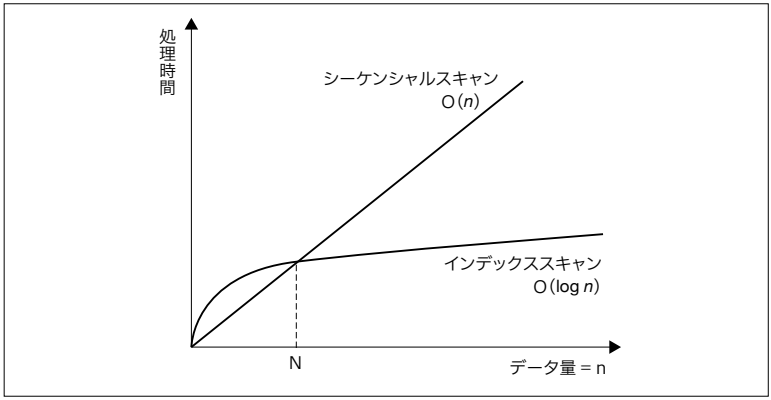
オブジェクトと操作についてはどうでしょう。こちらは興味深い変化が見られます。PostgreSQLでは「Index Scan」、Oracleでは「INDEX UNIQUE SCAN」という操作が現れています。これは、インデックスを使ったスキャンが行われるようになったことを示しています。

Oracleでは「TABLE ACCESS FULL」の代わりに「TABLE ACCESS BY

INDEX ROWID」と表示され、さらにその内訳としてId=2の行に「INDEX UNIQUE SCAN」、Name（対象オブジェクト）に「PK_SHOPS」が出力されています。この「PK_SHOPS」は主キーのインデックスの名前です。

インデックスについては第10章で詳しく解説しますが、一般的に、スキャンする母集合のレコード数に対して選択されるレコード数が小規模な場合に、テーブルに対するフルスキャンよりも高速なアクセスを実現します。これは、フルスキャンが母集合のデータ量に比例して処理コストが増大するのに対し、インデックスの中で一般的なB-treeインデックスでは、母集合のデータ量に対数関数的に増大するからです。これは、インデックスのほうが処理コストの増大のしかたが緩やかだということを意味するので、あるデータ量（N）を損益分岐点として、インデックススキャンのほうがフルスキャンよりも効率的にアクセスできるのです（**図1.15**）[注15]。

図1.15　母集合のデータ量が増えるとインデックススキャンが有利



今は60行しかデータが存在しないので、テーブルを順次読み込むのもインデックスでランダムアクセスするのも実行時間は大差ありませんが[注16]、この差はレコード数が増えたときに効いてきます。

なお、MySQLの実行計画は少し分かりにくいものです。「Rows fetched before execution」というのは、「クエリを実行する前に行が取得された」と

注15　あくまで一般論なので、諸条件によってそうでないこともあるのですが、そうした細かい話は第10章でします。

注16　実際60行程度のデータ量であれば、たとえWHERE句で主キーを指定しても、インデックススキャンではなくフルスキャンが選択されることもあります。

いう意味であり、最適化が行われた結果です。MySQLでは、クエリ実行の最適化や準備段階で、部分的に行を取得したりインデックス情報を収集したりする処理があります。そこでレコード数を計算しているのです。これは非常に分かりにくい表示ですが、ただの「EXPLAIN」コマンドを使ってツリー形式の表示をやめると、**図1.16**のように主キーのインデックスが使われていることが明示されます(実行計画が横に長いため、一部項目を省略しています)。

図1.16　インデックススキャンの実行計画(MySQL)

```
+----+-------------+-------+------------+-------+---------------+---------+
| id | select_type | table | partitions | type  | possible_keys | key     |
+----+-------------+-------+------------+-------+---------------+---------+
|  1 | SIMPLE      | Shops | NULL       | const | PRIMARY       | PRIMARY |
+----+-------------+-------+------------+-------+---------------+---------+
```

「key」列に「PRIMARY」と表示されていることから、主キーのインデックスが利用されていることが分かります。「select_type」列が「SIMPLE」と表示されているのは、結合やサブクエリなど複雑な操作をしていないという意味です。

■――操作の対象となるレコード数

どのDBMSもRowsが1になっています。WHERE句で主キーが「00050」の店舗を指定しているわけですから、アクセス対象が必ず1行になるためです。これは当然の変化です。

簡単なテーブル結合の実行計画

最後に、結合を行う実行計画を見てみましょう。SQLが遅いケースの十中八九は結合が関係します。かつ、結合が利用される場合は実行計画が複雑になりがちなため、オプティマイザも最適な実行計画を立てるのが難しくなります。したがって、結合時の実行計画の特性を学ぶことには重要な意味があります。結合の実行計画を理解することは、本書の主眼の一つです。

結合を行うにはテーブルが2つ以上必要ですので、図1.8の店舗テーブルのほかに**図1.17**の予約管理テーブルを追加しましょう。データ件数は10件登録するとします。

図1.17　予約管理テーブル

Reservations（予約管理）

reserve_id（予約ID）	shop_id（店舗ID）	reserve_name（予約者）
1	00001	Aさん
2	00002	Bさん
3	00003	Cさん
4	00004	Dさん
略		
10	00010	Jさん

実行計画を取得する対象のSQL文は、次のような予約の存在する店舗を選択するSELECT文です。

```
SELECT shop_name
  FROM Shops S INNER JOIN Reservations R
    ON S.shop_id = R.shop_id;
```

詳細は第6章で解説しますが、一般的に、DBMSが結合を行うアルゴリズムは3種類あります。

最も基本的でシンプルなのがNested Loops[注17]です。最初に片方のテーブルを読み込み、その1行のレコードに対して、結合条件に合致するレコードをもう一方のテーブルから探します。手続き型言語で書くと二重ループで実装するので、「入れ子ループ」という名前が付いています[注18]。

2つ目はSort Mergeです。結合キー（今のケースでは店舗ID）でレコードをソートしてから、順次アクセスを行って2つのテーブルを結合します。結合の前処理として（原則として）ソートを行うので、そのためのメモリ領域を必要とします。この作業用の領域が「もう一つのメモリ領域『ワーキングメモリ』」（17ページ）で触れたワーキングメモリです。

3つ目はHashです。名前のとおり、結合キーの値をハッシュ値にマッピングします。これもハッシュテーブルを確保するための作業用のメモリ領域を必要とします。

では、3つのDBMSがどのような結合アルゴリズムを採用するか見てみましょう（**図1.18**、**図1.19**、**図1.20**）。

注17　Oracleは「Nested Loops」で、PostgreSQLとMySQLは「Nested Loop」と表記していますが、本書では便宜上「Nested Loops」に統一します。

注18　このネーミングからわかるように、DBMSも内部では手続き型の方法でデータアクセスしているのです。

図1.18　結合の実行計画（PostgreSQL）

```
------------------------------------------------------------------------------
Nested Loop  (cost=0.14..14.80 rows=10 width=2)
  ->  Seq Scan on reservations r  (cost=0.00..1.10 rows=10 width=6)
  ->  Index Scan using pk_shops on shops s  (cost=0.14..1.36 rows=1 width=8)
        Index Cond: (shop_id = r.shop_id)
```

図1.19　結合の実行計画（Oracle）

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |              |    10 |   210 |     4   (0)| 00:00:01 |
|*  1 |  HASH JOIN         |              |    10 |   210 |     4   (0)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| RESERVATIONS |    10 |    60 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| SHOPS        |    60 |   900 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("S"."SHOP_ID"="R"."SHOP_ID")
```

図1.20　結合の実行計画（MySQL）

```
-> Nested loop inner join  (cost=4.75 rows=10)
    -> Filter: (r.shop_id is not null)  (cost=1.25 rows=10)
        -> Table scan on R  (cost=1.25 rows=10)
    -> Single-row index lookup on S using PRIMARY
          (shop_id=r.shop_id)  (cost=0.26 rows=1)
```

■――オブジェクトに対する操作の種類

OracleのOperation列には「HASH」と出ているため、使われているアルゴリズムについて迷うことはありません。また、PostgreSQLとMySQLでは「Nested Loop」と出ていることから、両者で同じアルゴリズムが選択されていることがわかります[注19]。

ここでちょっと、実行計画の読み方のワンポイントを教えましょう。実行計画は一般的にこのようにツリー構造で出力されるのですが、基本的に以下の2つのルールに従って読みます（このルールには例外もありますが、

注19　実際に使われる結合のアルゴリズムは、環境によって変化します。そのため、例えばPostgreSQLにおいてSort Merge結合が現れたり、Oracleにおいて Nested Loops結合が現れたりすることもあります。あくまで上記は実行計画のサンプルだと考えてください。

まずはこれが原則です)。

- **ルール1：インデントが深い操作ほど先に行われる**
- **ルール2：同じインデントの深さにある操作の場合、上に記述されている操作が先に行われる**

今、どの実行計画も、結合アルゴリズムの指定が最初に来ていますが、よりインデントの深い操作があるので先にそちらを見ます。すると、以下のような操作が実行されています。

```
PostgreSQL
  ->  Seq Scan on reservations r  (cost=0.00..1.10 rows=10 width=6)
```

```
Oracle
|   2 |   TABLE ACCESS FULL| RESERVATIONS |  10 |  60 |   2   (0)| 00:00:01 |
```

```
MySQL
        -> Table scan on R  (cost=1.25 rows=10)
```

これは、どのDBMSもまずはReservationsテーブルへアクセスしていることを示しています。Shopsテーブルへのアクセスも同じインデントの深さに記述されているのですが、ルール2に従って同じ深さであれば上に書いてある操作が優先して実行されます。結合の場合、どちらのテーブルを先にアクセスするかが重要な意味を持ってくるのですが[注20]、これは同じインデントの階層において上に位置するテーブルが先に使われるということです。今は、3つのDBMS全てでShopsテーブルへのアクセスが行われています。ここで、インデックスを使うかテーブルフルスキャンが使われるか選択肢が分かれています。PostgreSQLはインデックスを使い、OracleとMySQLはテーブルフルスキャンを実行しています。後二者がインデックスを使わないのは、レコード件数が少ない場合はインデックスを使っても使わなくても性能に大差が出ないからです。結合アルゴリズムとインデックスの重要性については、第6章で詳しく取り上げます。今は、実行計画のフォーマットに目を慣らしておいてもらえれば十分です。

注20　この先にアクセスされるテーブルを駆動表(*driving table*)と言います。

1.5 実行計画の重要性

最近のオプティマイザは、DBMSのバージョンを経るごとに日進月歩で優秀になってきていますが、それでも完全ではありません。前節でも解説したように、良かれと思って選んだ実行計画が惨憺(さんたん)たるパフォーマンスを生むこともあります。また、そうした複雑な問題以前に、絶対に使ったほうが速いはずのインデックスを使ってくれない、テーブルの結合順序が明らかにおかしい、といったポカもやります。オプティマイザもしょせん人間が作ったプログラムなので、絶対はありません。

そうした場合、最後のチューニング手段は、実行計画を手動で変えてしまうことです。たとえば、Oracle、PostgreSQL、MySQL、SQL Serverはヒント句というツールを持っています。これをSQL文の中に埋め込むことでオプティマイザに強制的に命令を出すことができます。ヒント句は結果には中立で、あくまでデータへのアクセスパスだけを変更する手段です。ヒント句以外にも、OracleのSPM(*SQL Plan Management*)、Amazon Aurora PostgreSQLのQPM(*Query Plan Management*)のように、クエリの実行計画を制御する機能を持つDBMSもあります。現実問題、オプティマイザが選ぶ実行計画が最適でない場合、どうにかして人間が実行計画を修正してやる必要に迫られるわけで、ヒント句以外にもSQL文の構文変更やテーブル設計、アプリケーションの修正といった大規模な対応に迫られることも少なくありません。

そうしたケースにおいて、われわれデータベースエンジニアとしてどのような選択肢があり得るかは、本書の中で一つずつ検討していきますが、まずは現状の実行計画を確認し、当該のSQL文がどのようなアクセスパスでデータを取得しているかを知ることが、チューニングの第一歩です。そしてそれ以前に何よりも、効率的なテーブル設計を行い、無駄のないSQL文を記述するためには、データベースエンジニアはSQL文の実行計画を机上である程度予測できなければいけません。これは、物理層を隠蔽しようというRDBが目指した目標には逆行するのですが、いまだ理想の達成されない現実に生きる者には、理想的ではない手段も必要とされるのです。

第1章のまとめ

- **データベースはさまざまなトレードオフのバランスを取ることを目的としたソフトウェアである**
- **特にパフォーマンスの観点では、データを低速なストレージと高速なメモリのどちらに配置するかのトレードオフが重要**
- **データベースは更新よりも検索を重視した設計とデフォルト設定になっているが、それが本当に適切かは判断が必要**
- **データベースはSQLを実行可能な手続きに変換するために実行計画を作っている**
- **本当はユーザが実行計画を読むのは本末転倒だが、人生すべてが思いどおりにはいくわけではない**

演習問題1

DBMSのデータキャッシュは、容量の限られたメモリ上になるべく効率的にデータを保持できるような工夫がなされています。どのようなルール（アルゴリズム）でデータを保持するのが効率的か、考えてみてください。そのあとに、自分の使っているDBMSにどのようなアルゴリズムが採用されているか、マニュアルなどを使って調べてください。

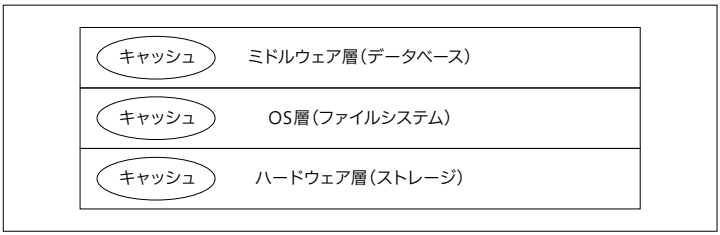
➡解答は366ページ

Column

いろいろなキャッシュ

キャッシュというしくみは、データベース層だけでなく、いろいろなレイヤで利用されています。たとえば、Linux OSは「ファイルキャッシュ」と呼ばれるキャッシュを持っています。このキャッシュは、OS上で動作するアプリケーションなどが使っていない「余った」メモリを利用して、同じファイルを繰り返し読み込むときに、再度ストレージへアクセスせずに済むようにしています。一方、ハードウェアレベルにおいても、ストレージは(グレードにもよりますが)やはりキャッシュを持っています(**図1.a**)。

図1.a　キャッシュの階層



また、アプリケーション側でデータベースの結果セットをキャッシュに保持し、データベースまでSQL文を発行しなくてもユーザに結果を返せるようにするしくみを採用することもあります(**クエリキャッシュ**と呼びます)。最新のデータが必要ない場合は、こうしたシンプルなパフォーマンス改善方法もあります。

これらさまざまなレイヤにおけるキャッシュは、データベースにおけるデータキャッシュの役割と一部重複します。しかし、OSのファイルキャッシュに割り当てるよりは、サーバの物理メモリを圧迫しない程度にデータベースのデータキャッシュに割り当てたほうが、データベースのパフォーマンス向上が期待できます。これは、OSがキャッシュする対象はデータベースのデータ以外に、OS上のほかのプロセスが使用するデータも含まれるので、(データベースから見ると)キャッシュ効率が悪いためです。

かといって、データベースにメモリをあまり多く割り当てすぎると、物理メモリの枯渇を招き、スワップが起きてスローダウンが発生します。これでは本末転倒な話になるので、データベースにどの程度のメモリを割り当てるかを判断するときは、「あくまで物理メモリの範囲内でなるべく多く」というのが原則です。

なかにはSQL Serverのように、データベースに対する割り当てメモリ量を自動調整可能なDBMSもあります。OSや他のアプリケーションがメモリを必要とすると、SQL Serverは自身のキャッシュを自動的に縮小して解放します。これはOS(Windows)とDBMS(SQL Server)を同一ベンダー(Microsoft)が開発しているため、両者を密接に連携させられるからこそ可能な機能です。