達人に学ぶDB設計徹底指南書

ミック●著

初級者で終わりたくないあなたへ

アプリケーション開発者・DBエンジニア必携!

データベース設計の正しい考え方と実践ノウハウが身につく

- パフォーマンス／性能設計
- 正規化／非正規化のケーススタディ
- やってはいけないバッドノウハウ
- 注意すべきグレーノウハウ
- 基礎知識とつまづきやすいポイント　など

67の勘どころ

21問の演習問題

SE SHOEISHA

達人に学ぶDB設計 徹底指南書

ミック●著

初級者で終わりたくないあなたへ

SE
SHOEISHA

本書内容に関するお問い合わせについて

このたびは翔泳社の書籍をお買い上げいただき、誠にありがとうございます。弊社では、読者の皆様からのお問い合わせに適切に対応させていただくため、以下のガイドラインへのご協力をお願い致しております。下記項目をお読みいただき、手順に従ってお問い合わせください。

ご質問される前に弊社Webサイトの「正誤表」をご参照ください。これまでに判明した正誤や追加情報を掲載しています。

また、ご質問いただく際には、弊社Webサイトの「刊行物 Q＆A」をご利用ください。

正誤表　https://www.seshop.com/book/errata/

刊行物 Q＆A　https://www.seshop.com/book/qa/

インターネットをご利用でない場合は、FAXまたは郵便にて、下記“翔泳社愛読者サービスセンター”までお問い合わせください。電話でのご質問は、お受けしておりません。

〒160-0006　東京都新宿区舟町5

（株）翔泳社 愛読者サービスセンター

FAX番号：03-5362-3818

回答は、ご質問いただいた手段によってご返事申し上げます。ご質問の内容によっては、回答に数日ないしはそれ以上の期間を要する場合があります。

本書の対象を越えるもの、記述個所を特定されないもの、また読者固有

の環境に起因するご質問等にはお答えできませんので、予めご了承ください。

※本書に記載されたURL 等は予告なく変更される場合があります。

※本書の出版にあたっては正確な記述につとめましたが、著者や出版社などのいずれも、本書の内容に対してなんらかの保証をするものではなく、内容やサンプルに基づくいかなる運用結果に関してもいっさいの責任を負いません。

※本書に掲載されているサンプルプログラムやスクリプト、および実行結果を記した画面イメージなどは、特定の設定に基づいた環境にて再現される一例です。

はじめに

本書は、リレーショナルデータベースにおける設計についての書籍です。「データベース設計」と一口に言っても、その内容は多岐にわたります。このジャンルの書籍は、大きく以下の3種類に分類できます。

❶ 論理設計
❷ 物理設計
❸ 実装設計

❶ は、いわゆる正規化やER図といった道具を使ったデータのモデル設計です。一般的に「データベース設計」と聞いて、多くの人が思い浮かべる分野がこれかもしれません。

❷ は、サーバーやストレージといった物理的なハードウェアレベルの設計です。データベースも究極的にはこうしたハードウェアの上で動作するため、物理設計も重要です。

❸ は、特定のデータベース製品を前提に、具体的な構築の手順や方法を解説したものです。データベース製品によってアーキテクチャに違いがあるため、実際にシステムを構築する段には、製品に寄りそって考える必要があります。「○○で作るデータベースサーバー」とか「○○実践パフォーマンスチューニング」といったタイトルの本（○○はデータベース製品）は、このカテゴリに属します。

上記の分類に従えば、本書がカバーするのは、❶ 論理設計と❷ 物理設計です。書籍によっては、❶ と❷ を分離して、どちらか一方にフォーカスするものも珍しくありません。ですが、本書ではこの両者を同時に考えていくスタイルを取っています。章立ての便宜として、論理設計と物理設計に分ける構成にはなっていますが、片方が主題の章にも必ずもう一方が顔を出してきます。

その理由は、論理と物理、二つのレベルの設計を同時に考えたほうが理解しやすいから——ではありません。むしろ、片方を学ぶ間はもう一方のことは忘れたほうがわかりやすいぐらいです。本書が両者を同時に考える理由は、この二つのレベルの間に、強いトレードオフの関係が成立しているからです。

> トレードオフ。日本語に訳すと「あちらを立てればこちらが立たず」。すなわち、論理設計をきれいに行なおうとすると物理設計が犠牲になり、物理設計を優先すると論理設計が犠牲になる。

実のところ、トレードオフは別にデータベース設計だけに使われる言葉ではなく、システム開発全般、ひいては仕事からプライベートから、私たちの生活すべてにおいて当てはまる原理です。米国の経済学者マンキューは次のように言っています。

> 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トレードオフを必要とするのである。［※］

みなさんは本書を通じて、望ましい論理設計とは何か、望ましい物理設計とは何かを学んでいくことになります。しかし本書の目的は、単にそれを理解してもらうだけにはとどまりません。さらに、望ましい論理設計を達成しようとするとき、犠牲になるものは何か、望ましい物理設計を諦めなければならないのは、どのような理由によるのか、といったトレードオフを学びます。

エンジニアの本当の仕事は、それを知った後に始まるのです。

それでは、始めましょう！

N・グレゴリー・マンキュー『マンキュー入門経済学』（東洋経済新報社、2008）p.5

目 次

エンティティの抽出

エンティティの定義

正規化

ER図の作成

2-2　内部スキーマと物理設計

物理設計のステップ

テーブル定義

インデックス定義

ハードウェアのサイジング

ストレージの冗長構成

ファイルの物理配置

2-3　バックアップ設計

バックアップの基本分類

完全／差分／増分

フルバックアップ

差分バックアップ

増分バックアップ

バックアップ方式にもトレードオフがある

どんなバックアップ方式を採用すべきか？

2-4　リカバリ設計

リカバリとリストア

リストアとロールフォワード

演習問題

第3章 論理設計と正規化～なぜテーブルは分割する必要があるのか?

索引

■本書を読むに当たっての注意事項

・本書に記載されているSQLは、可能な限り標準SQL（SQL-92/99/2003）に準拠するよう配慮しています。

・テーブルの相関名を指定するキーワード「AS」は、標準SQLでは要請されていますが、本書では省略しています。これは、Oracleでエラーが生じることを回避するためです（他のDBでも、省略した場合にエラーは生じません）。

・本書サンプルコード（SQL）の動作確認は、PostgreSQL 9.1.1で行ないました。

・以下のサンプルダウンロードサイトから、本書サンプルコードの一部をダウンロードできます。本書内のSQLでは、読みやすさを考えてテーブル名、列名に日本語を用いている箇所がありますが、ダウンロードできるサンプルコードではすべてローマ字表記を使用しています。

https://www.shoeisha.co.jp/book/download/9784798124704

第1章

データベースを制する者はシステムを制す

データベースは、私たちを取り囲むあらゆるシステムで利用されています。システム開発においても、データベース設計は中心的な役割を持ちます。本章では、データベースがシステムおよび開発工程において持つ重要性を、システムにおける位置づけや開発プロセスとの関連から学びます。

学習のポイント

- 私たちの生活は、意識していなくても様々なデータベースに囲まれて成り立っている。
- データベースにはデータを保持する形式によって多くの種類がある。本書で取り上げるのは、現在主流の「リレーショナルデータベース」とその設計技法である。
- 「データベース」と「DBMS」は異なる。「データベース」はデータの集積を指す論理的概念であるのに対し、「DBMS」は「データベース」を実装したソフトウェアを指す。
- データベース設計を制する者はシステム開発を制す。それはシステムがデータのフォーマットに合わせて作られるから（システムに合わせてデータを作るのではない）。
- データベース設計を考える際は、データベースを外部スキーマ、概念スキーマ、内部スキーマの3層に分けて考える。

1-1 システムとデータベース

今日、私たちの生活とシステムは切っても切り離せないほど密接に結びついています。これほどに多くの人がシステムを利用して生活を送るようになった時代は、今までにありませんでした。

一口に「システムを使う」と言っても、そこには多面的なとらえ方があります。これまで人間が行なっていた作業をシステムが肩代わりしてくれるという意味では、システムとは優れたサービス代行業者のようなものです。また、これまでになかったサービスを創造するという点では、人々の生活を完全に変えてしまう革命的な一面も持っています。20世紀末に登場した検索エンジン（Google）に加えて、21世紀に入って隆盛を見ているSNSサービス（Facebook、Twitter）やスマートフォン（iPhone、Android）、電子書籍（Amazon Kindle）といったシステムは、私たちの生活を二度と戻らない形で変えてしまう力を持っています。

データ処理としてのシステム

そうしたシステムは、システムが稼働するハードウェアもソフトウェアも異なっていながら、一つだけ共通点を持っています。それは、すべてのシステムが「データ」を取り扱

っていることです。たとえば、Googleは世界中のWebサイトを巡回するプログラムを作り、各サイトの一時的なコピー（キャッシュ）を自社のサーバーに保存して、それをもとにページランキングを算出しています。FacebookやTwitterのサービス上を流れる友人間のメッセージもまた、データとして保存され、処理されることでサービスが成立します。Amazonのような小売業にとっては、顧客の購買履歴データを使って「おすすめ商品」をプッシュするのは、もはや常識になっています。

こうした「データ」を整合的に保持して、いつでも簡単に利用可能な状態にしておくためのシステムが、私たちが本書でその設計技法を学ぶ「データベース（Database）」です（略して「DB」とも呼びます）。厳密に言うと、「データベース」とは「データの集まり」を指すために使われる言葉で、そのデータベースを管理するためのシステムを「DBMS（Database Management System）」と呼びます。本書では、このような意味で「データベース」と「DBMS」を使い分けます。

話をシステムに戻すと、今日のシステムの中で、データベースを持っていないシステムは存在しません。もちろん、システムを構成する部品的なプログラム［※1］の中にはデータベースを持たないものもありますが、一つのサービスという全体的な観点から見れば、そこには必ずデータベースが存在しています。ただ、データベースは通常、個人情報など重要なデータを多く含んでいるため、利用者に見えるようにはなっていません。注意深く隠されています。そのため、私たちがシステムを利用する際に、データベースを意識することがないだけです。私たちは普段、「空気」の存在を意識しないからといって、周りに空気が存在しないわけではありません。それと同じで、私たちはデータベースに囲まれて生活しているのですが、その存在を意識していないだけなのです。

勘どころ 1

データベースを使わないシステムは、この世に存在しない。

データと情報

さて、先ほど「データ」という言葉が出てきました。この「データ」は、「情報」という言葉と対になって使われることが多く、その定義も様々ですが、本書では以下のように定義します。

データとは、**ある形式（フォーマット）に揃えられた事実**のことである

たとえば、ある人物をデータ化しようとしたら、まず何の属性（年齢、性別、身長、体重、好きな食べ物……）を切り出すかを考える必要があります。その次に、そうした属性の値をあるフォーマットに従って並べます。一番わかりやすい形式は、おそらく二次元表の形でしょう。データベースは、こうした一定の形式に従ったデータの集積です。

■二次元の表（人物のデータ化）

名前	年齢	性別	身長	体重	好きな食べ物
佐藤	36	男	181	68	カレー
鈴木	29	女	162	60	チョコレート
田中	32	男	178	65	ポテトチップス

一方、情報という概念は、こうしたデータとは異なります。というのも、情報とは、データから、ある文脈なり観点なりに従って集約したり加工したりしたものを指すからです。

先ほどのAmazonのレコメンド機能を例に説明するなら、ある顧客の購買履歴のデータは、それ自身は「誰が何月何日にこういう商品を何個買った」という事実を語っているだけです。しかし、小売業者にとって欲しいのは、「このお客はどういう商品なら買うだろうか？」という問いに対する答えです。この答えが、すなわち「情報」と呼ばれるわけですが、これは膨大なデータをただ眺めていても出てきません。「どのような時期によく買っているか」「どのような種類の商品をよく買っているか」といった観点（文脈）による分析が必要になります。したがって、情報とはデータと文脈を合成して生まれてくるものだ、と言っても良いでしょう。

勘どころ 2

情報はデータと文脈を合成して生まれる。

システムとは、このデータと情報の区別という観点から見た場合（図1-1）、ユーザーがシステムにデータを登録し[※2]、さらにそれを引き出して情報を作り出す、という一連のサイクル（流れ）の中に位置づけることもできるのです。

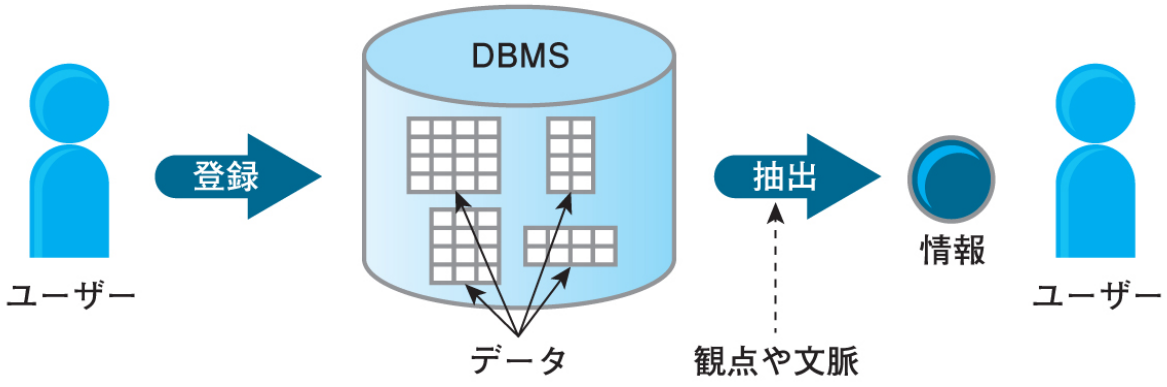


図1-1●システムのサイクル

このような部品を「モジュール」と呼びます。

もちろん、ユーザーは自分が意識的にデータを登録しているとは思っていません。彼らはただサービスを利用しているだけです。

本書のテーマは、こうしたシステムの流れの中で中核をなすと言っていいデータベースにおいて、どのようにデータを保持するべきか、その設計についてです。しかしその話をする前に、データベースおよびDBMS、そしてシステム開発そのものについて少し予備知識を持っておく必要があります。本節と次節ではデータベースについて本当に何の知識も持っていない人に向け、必要な予備知識を解説します。そのため、すでに仕事である程度データベースを利用している人は、この1-2節および1-3節は飛ばして、1-4節へ進んでいただいてかまいません。

データベースの代表的なモデル

現在、商用で利用されているデータベースには、何種類かありますが、その分類は、データを保持するフォーマットを基準にして行なわれています。代表的なモデル（構造／仕様）を挙げると以下のようになります。

リレーショナルデータベース（Relational Database：RDB）

関係データベースとも呼ばれ、現在最も広く利用されているデータベースです。そのため、普通IT業界では、特に断りなく「データベース」と言えば、リレーショナルデータベースを指します。リレーショナルデータベースは、すでに長い歴史を持ち、誕生は1969年までさかのぼります。

特徴としては、データを人間が理解しやすい二次元表の形式で管理するため、データの取り扱いが他のデータベースに比べると直観的で簡単です（図1-2）[※3]。本書で取り上げるのが、このリレーショナルデータベースについての設計技法です。

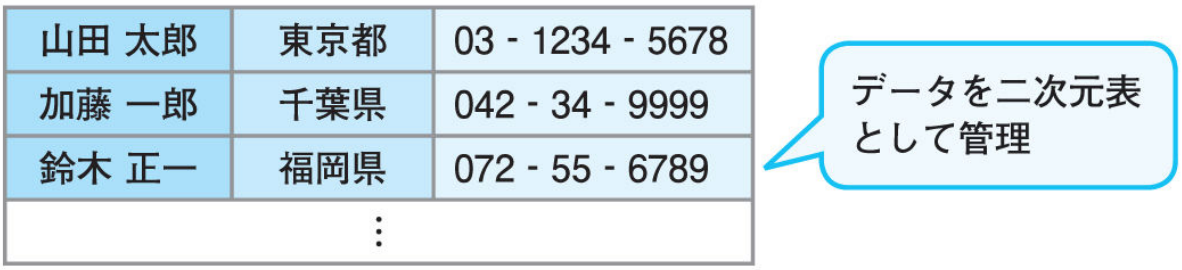


図1-2●リレーショナルデータベース

オブジェクト指向データベース（Object Oriented Database：OODB）

プログラミング言語の中には、JavaやC++など、オブジェクト指向言語と呼ばれるタイプの言語があります。データと操作をまとめて「オブジェクト」という単位で管理するためにそう呼ばれます。このオブジェクトをデータベースに保存するために作られたのが、オブジェクト指向データベース（図1-3）です。

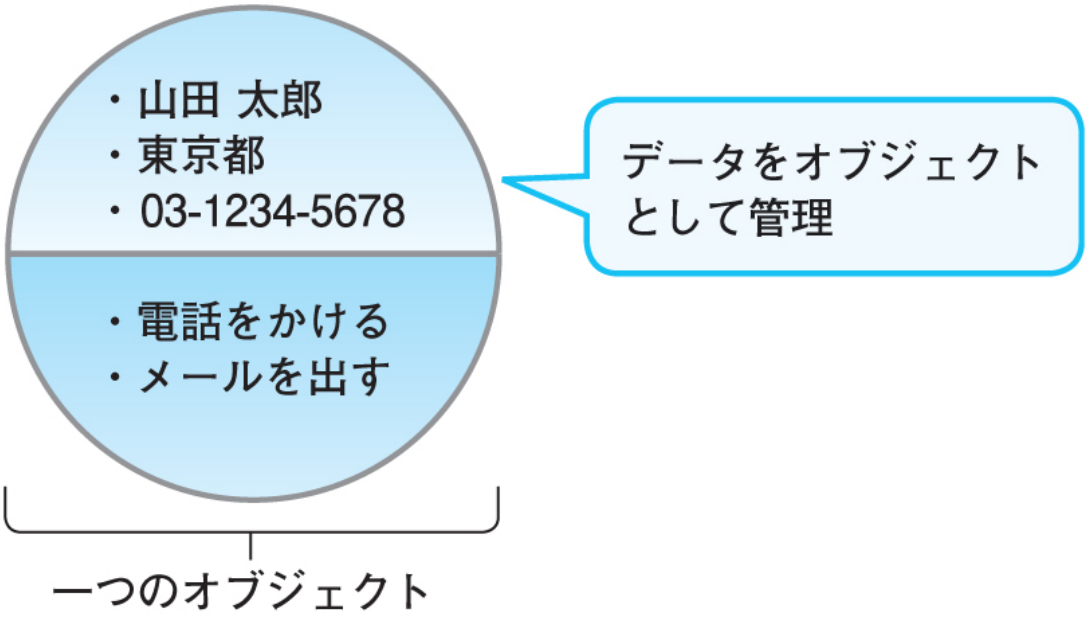


図1-3●オブジェクト指向データベース

XMLデータベース（XML Database：XMLDB）

近年、Web上でやり取りされるデータの形式にXML（HTMLのようにタグでデータ管理を行なう言語）が普及しています。このXML形式のデータを扱うことのできるデータベースとして開発されたのが、XMLデータベース（図1-4）です。リレーショナルデータベースが苦手とする階層構造のデータの扱いを得意とします［※4］。

```
<address>
  <name>山田 太郎</name>
  <prefecture>東京都</prefecture>
      ⋮
</address>
```

データをXMLとして管理

図1-4●XMLデータベース

キー・バリュー型ストア（Key-Value Store：KVS）

データをKey（識別キー）とValue（値）の組み合わせだけの単純なデータ型で表現するデータベースです（図1-5）。単純なデータ問い合わせを高速化することを目的としており、大量データを高速に処理する必要のあるWebサービスで多く利用されます。その反面、複雑なデータ操作などは苦手です。

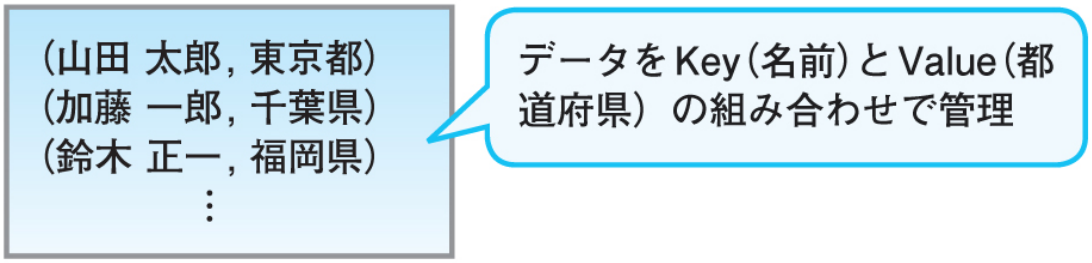


図1-5●キー・バリュー型ストア

階層型データベース（Hierarchical Database：略称は特になし）

データを階層構造（木構造）で表現するデータベースです（図1-6）。リレーショナルデータベースより一世代前の主流データベースでしたが、現在はRDBの普及に伴い、あまり一般的には使用されなくなっています。

図1-6●階層型データベース

こうしたデータベースのモデルが異なるということは、データのフォーマットが異なることを意味するため、設計の方法も異なります。したがって、一つのモデルで覚えた設計技法は、基本的に別のモデルでは応用できない、ということになります。たとえば、本書で取り上げるリレーショナルデータベースの重要な設計技法に「正規化」というものがありますが、これは他のモデルのデータベースでは、通常行ないません。

しかし、こうした設計技法は、データの整合性の保持や冗長性排除といった、業務上必要な目的を達するために考えられたものです。この目的は、モデルが変わったからといって変わるわけではありません。したがって、同様の観点に注意して設計を行なう、という点は同じです［※5］。

勘どころ 3

データベースのモデルが異なれば、データフォーマットも異なる。モデルが異なれば設計技法も異なる。

DBMSの違いは設計に影響するか？

主なDBMS

リレーショナルデータベースでもDBMSにはいくつか種類があります。代表的なものを挙げると以下のとおりです。

- Oracle Database
- SQL Server
- DB2
- PostgreSQL
- MySQL

Oracle Databaseはオラクル社の製品で、リレーショナルデータベースのDBMSとしては最大のシェアを誇ります（Windows、Linuxなどマルチプラットフォーム対応）。省略して「Oracle」と呼ばれることが多いため、本書でも以後「Oracle」という呼称を用います。

SQL Serverは、マイクロソフト社の製品で、Windowsプラットフォームに特化したDBMSです。DB2はIBM社の製品で、UNIX、Windows、同社のメインフレームOSといったマルチプラットフォーム対応のDBMSです。

ここまでの3つが大手のITベンダーによって作られているDBMSです。したがって、これらは基本的に商用利用するときは有料のDBMSです。

対して、PostgreSQLは米バークレイ大学で開発されたオープンソースのDBMS です。MySQLも同様にオープンソースですが、サン・マイクロシステムズの所有を経て、2010年から上述のオラクル社の所有するDBMSとなりました［※6］。

DBMSの違いと設計技法の関係

先ほど、データベースのモデルが異なる場合、その設計技法も異なる、と述べましたが、こうしたDBMSの違いもまた、設計技法に影響を与えるのでしょうか？ これに対する答えは、「No」です。

データベースの設計技法は、モデルに影響を受けますが、DBMSは、そのモデルを具体的に表現したものに過ぎません。そのため、本書で紹介する設計の方法論は、DBMSの違いによらず適用可能なものです。なお、「モデルを実際に表現する」ことを実装（implementation）と呼びます。

ただし、この実装に少し留保が必要なのも事実です。実際には、それぞれのDBMSの持っている機能に応じて設計が少し影響を受けることはあります。そのような実装間の機能差異の多くは、各DBMSが独自機能を発展させてきたことに起因しますが、また一方で、そのような事態が生じるということは、DBMS側がモデルを十分に表現できていないという力不足に起因することもあります［※7］。

勘どころ 4

DBMSが異なっても、（基本的には）設計の方法は影響を受けない。

なお、ここで紹介したような、リレーショナルデータベースを管理するシステム（DBMS）は、「RDBMS（Relational Database Management System）」と呼びます。本書では、特に断らない限り、「DBMS」と「RDBMS」を同義の言葉として使います

このフォーマットの詳細については3-1節であらためて解説します。

一方、リレーショナルデータベースにおいて階層データを取り扱うための方法論もあります。これについては第9章で解説します。

ただし、目的を実現するための手段が異なるため、リレーショナルデータベース以外のモデルの設計技法を学びたい場合は、そのモデルにフォーカスした専門の書籍を読む必要があります。

つまり、オラクル社は主要なDBMS製品を二つ持っているわけです。

こうした実装に依存する注意点については、本書では基本的に取り上げませんが、解説の要点に関係する場合は、随時説明します。ただ、詳しくは、各DBMSのマニュアルやそれに特化した書籍などを参照してください。

前節でも述べたように、本書は、リレーショナルデータベースにおける設計の方法論を取り上げます。ここでもう一つ予備知識として理解しておく必要があるのが、今まで「設計」と大ざっぱなくくりで呼んでいた作業が、システム開発全体においてどのような位置を占めているか、という点についてです。読者のみなさんの中には、すでに何年かのシステム開発のキャリアを積んできた方もいるでしょう。そうした方は、設計について具体的なイメージを持っていると思いますが、そうではない方のために、ここで設計の基本的な流れを整理しておきます。

システム開発の設計工程

システム開発は、よく建物や橋など建造物を造る行為にたとえられます。もともと比較的歴史の浅いIT業界は、その仕事の進め方を他の業界から借りている部分が多いのです。

開発は、建築と同様、いくつかの工程（ステップ）に分解することができます。大きな分類としては以下のとおりです。

❶ 要件定義
❷ 設計
❸ 開発（実装）
❹ テスト

❶ 要件定義

システムが満たすべき機能やサービスの水準、すなわち要件を決める工程です。通常は何らかの形で外部の顧客と要件を合意しますが、パッケージソフトやWebサービスの場合は、自社内のみで要件を定義することもあります。

❷ 設計

定義された要件を満たすために必要なシステムを作るための設計（デザイン）を行なう工程です。建築で言えば、実際に作る前に図面を引く作業に相当します。本書で焦点を当てる工程はこの設計工程なのですが、この工程はさらに細分化して考える必要があります。これについて、1-4節で詳細を検討します。

❸ 開発（実装）

設計書に従ってシステムを実際に作る工程です。なお、「システムを実際に作る」ことをIT業界では「実装する（implement）」と言います（DBMSをデータベースの「実装」と呼ぶのと同じ意味です）。

この工程は、一般的にはプログラマによるコーディングを指すことが多いのですが、それ以外にもサーバーやネットワーク機器、ストレージといったハードウェアの調達や環境の物理的構築といった作業も含まれます。

❹ テスト

実装によって組み上がったシステムが、本当に実用にたえる品質であるかを試験（テスト）する工程です。目的に応じてテストの種類とレベルにも何種類かありますが、大きくは機能的な品質に対するテストと、性能や信頼性といった非機能品質に対するテストの二種類に大別できます。

上記の四つの工程は、システム開発の最も基本的な分類であり、各工程はさらにいくつかのサブ工程に細分化されます。このうち、本書では、「設計」工程について焦点を当てていくことになります。

設計工程と開発モデル

システム開発の進め方（開発モデル）には、大きく分けて二つの方法があります。一つが**ウォーターフォールモデル**、もう一つが**プロトタイピングモデル**です。

ウォーターフォールモデル

「滝（waterfall）」という名前が示すとおり、要件定義→設計→開発→テスト、というように一つずつ工程を踏んで、段階的にシステムを作っていきます（図1-7）。滝が逆流することがないように、基本的に工程が逆戻りすることはありません。これは、建物や橋など建築物を造るときの方法でもあります。「家を作っていてあまりうまくいかなそうなので、もう一度図面から引きなおしましょう」というのは、時

間とお金の無駄が大きすぎるので、まずありえません。つまり、手戻りが難しく、改修が高コストになりがちです。

ウォーターフォールモデルは、大規模なシステム開発において採用されることが多いのですが、手戻りがきかないということは、各工程をきっちり仕上げる精密さが要求される、ということでもあります。

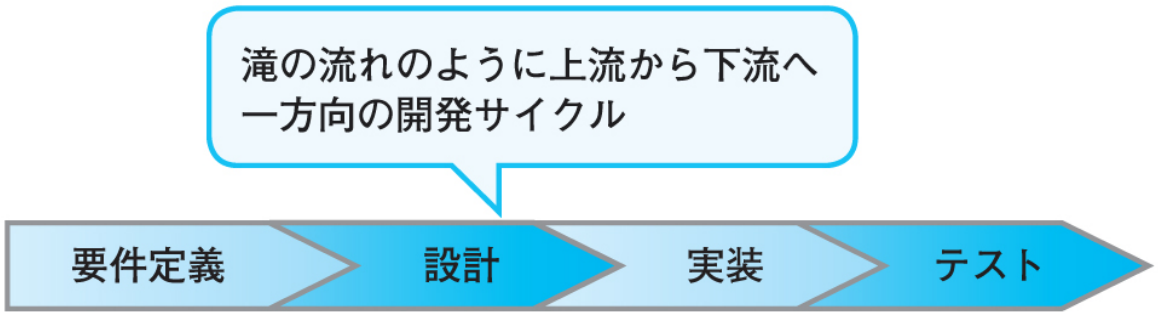


図1-7●ウォーターフォールモデルの開発イメージ

プロトタイピングモデル

もう一方のプロトタイピングも、「試作品（prototype）」という名前が内容を表わしています。最初に小さな試作品を作って顧客やユーザーに見せてフィードバックをもらい、それを取り入れた形で改良版を作って顧客やユーザーに見せて……というサイクルを繰り返す**循環的**な開発手段です（図1-8）。

早い段階からシステムの具体的なイメージを開発者や顧客と共有できるため、要件定義の取りこぼしや意思疎通の齟齬（そご）を防げるというメリットがある一方で、何度も同じ工程を繰り返す必要があり、変更を繰り返すうちに発散して収拾がつかなくなる、というリスクもあります。こうした特性から、普通は小規模のシステムに対して適用されます。

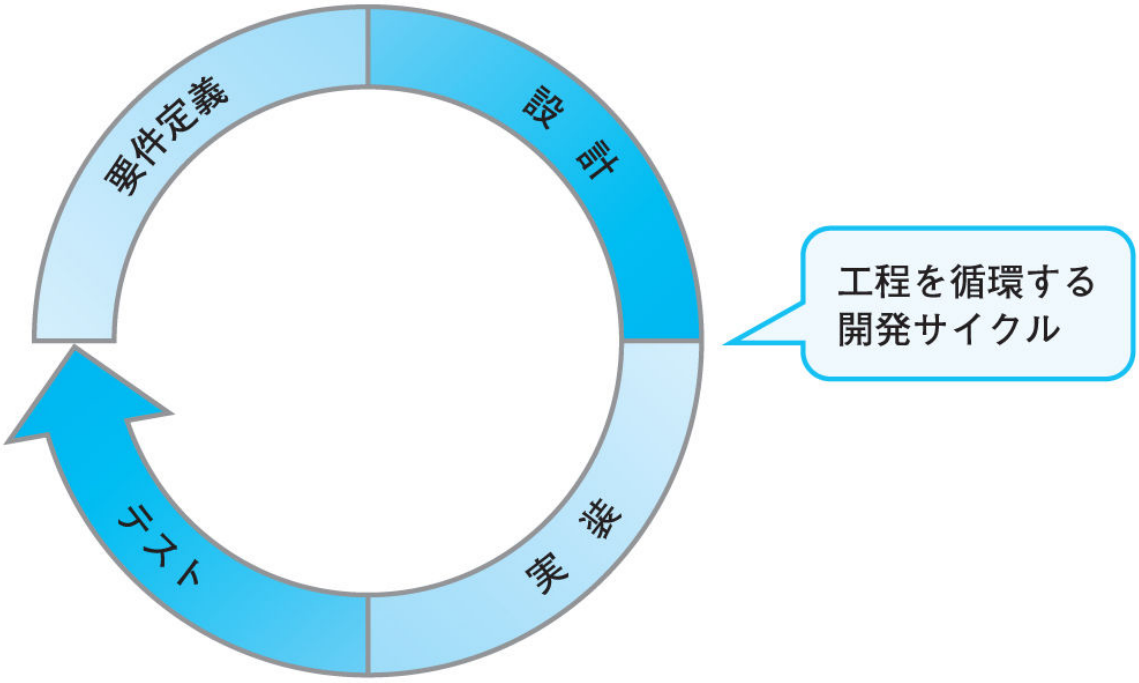


図1-8●プロトタイピングモデルの開発イメージ

また、このモデルを応用した「**アジャイル**」と呼ばれるモデルもあります。これは特にサービス提供を迅速に行なうことが死活的に重要なWeb系システムの開発で利用されます。

四つの基本工程のうち、本書がフォーカスを当てるのは設計工程です。この工程には、さらに多くのサブ工程が含まれます。たとえば、ソフトウェアの提供する機能を決めるアプリケーション（AP）設計、ユーザーが使用する画面などのユーザーインタフェース設計などが該当します。

本書で取り上げるのは、こうしたサブ工程の一つである「データ設計」、特にデータベースに保持するデータの設計です［※8］。この設計を以降「データベース設計（DB設計）」と呼びますが、なぜデータベース設計が重要かと言えば、次の二つの理由があります。

理由1 システムにおいて大半のデータ（少なくとも永続的に使用されるデータ）はデータベース内に保持される。そのため、普通、データ設計とはデータベース設計とほぼ同義である。

理由2 データ設計がシステムの品質を最も大きく左右する。ソフトウェアというのは、言ってみれば「データの流通機構」であって、どのようなプログラムが必要になるかは、どのようなデータをどういうフォーマットで設計するかに左右される［※9］。

DOAとPOA

このうち特に重要なのが**理由1**です。近年のソフトウェア開発では、**データ中心アプローチ**（Data Oriented Approach：**DOA**）という考え方が主流です。これは、文字通りシステムを作る際に、プログラムよりも前にデータの設計から始める方法論です。スローガン的に言えば「最初にデータありき」です。

少し歴史的な話になりますが、かつてのシステム開発の主流の考え方は、これと正反対でした。つまり、**プロセス中心アプローチ**（Process Oriented Approach：POA）です。「プロセス」というのは「処理」のことですから、プログラムと同義だと考えてください。しかし、現在ではこの考え方はすでに時代遅れと見なされており、普通はPOAが採用されることはありません。

- DOA：データ ➡ プログラム
- POA：プログラム ➡ データ

これは素朴な直観で考えると、少し不思議な印象を受けます。システムというのは何よりプログラムによって作られているのですから、プログラムから作り始めるというのは、それほどおかしいことではありません。というよりむしろ自然なような気がします。事実、システム開発の素人に、予備知識を何も与えないままで開発を行なわせると、ほぼ十中八九がプログラムから設計を始めます。新入社員研修のカリキュラムの一つに、新人同士でチームを組ませてミニシステムを開発させる、という演習がありますが、こういう場合など、データ設計から始めるチームは少数派です。

しかし実際にやってみるとわかるのですが、POAは最初に抱くイメージよりも非効率な方法です。プロセス、つまり業務処理というのは短期間で大きく変わっていくことが頻繁にありますし、プロセス単位でデータ設計を行なうことになるため、複数のプロセスで同じデータを別個に持つという冗長性が生じるなど、不都合が多いのです（図1-9）。

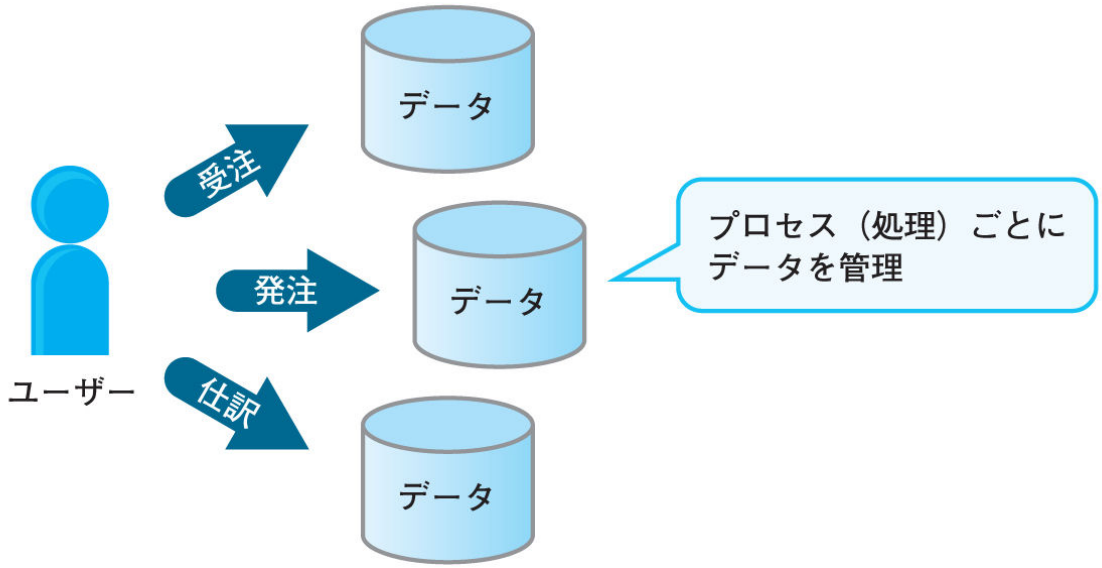


図1-9●POA では処理ごとにデータが分散して管理が大変

DOAは、こうしたPOAの欠点を克服するために登場しました。その着眼のポイントは、データがあまり変化しない（永続的）という点です。したがって、データの意味や形式が先に決まっていれば、複数のプログラムで共用することも容易で、業務要件の仕様変更にも柔軟に対応できるというメリットが得られます（図1-10）。

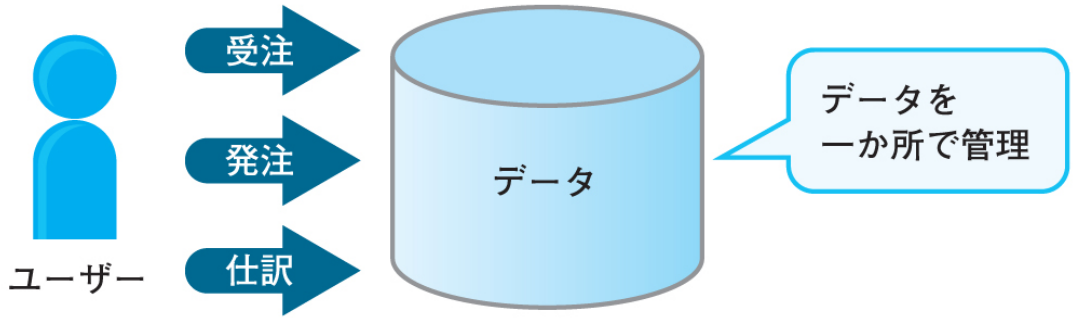


図1-10●DOA ならばデータを一元管理できる

こうした理由から、システム開発においては、プログラム設計に先立ってまずデータ設計が優先されるというわけです。

勘どころ 5

最初にデータがある。プログラムはその次にできる。

したがって、データ設計――それは現在ではほぼデータベース設計と同義なのですが――システムの品質を決める最も重要な要因と言っても過言ではないのです。このデータ設計においてセオリーを踏み外した設計（バッドノウハウ、またはアンチパターンと呼びます）を行なってしまうと、システムの機能／非機能の品質を致命的に損なうことになります。そうしたバッドノウハウ／アンチパターンについては、第7章および第8章で具体的に一つ一つ検証していきます。また、DOA の重要性については、コラム「バッドノウハウのどこが悪いのか？」（219ページ）でも触れているので、参考にしてください。

勘どころ 6

データベースを制する者がシステムを制す。データベースは、システムの中心であると同時に、システム開発の中心でもある。

3層スキーマ

さて、ここまでで、データベース設計がシステム開発の最も重要な要素であることを解説しました。以降では、データベース設計を、具体的にどのような段階を踏んで行なっていくのかについて整理します。

そこで重要な概念として登場するのが、「**スキーマ**（schema）」という概念です。「枠組み」や「構図」という意味の単語ですが、データベース設計においては、データベースのデータ構造やフォーマットという意味で使います。データベース設計のステップは、このスキーマのレベルと密接に結びついています。スキーマは、一般的に三つのレベルに分けられます［※10］。

1. **外部スキーマ**（外部モデル）＝ビューの世界
2. **概念スキーマ**（論理データモデル）＝テーブルの世界
3. **内部スキーマ**（物理データモデル）＝ファイルの世界

この三つのスキーマに基づいてシステムを記述したモデルを、「**3層スキーマモデル**」と呼びます（図1-11）。

❶ 外部スキーマ

外部スキーマとは、システムの利用者であるユーザーから見て、データベースがどのような機能とインタフェースを持っているかを定義するスキーマです。いわば「**ユーザーから見たデータベース**」の姿です。データベースのオブジェクトとしては、ビューが相当します（ビューについては、第8章で詳しく取り上げます）。このスキーマは、データベースだけでなく、画面のユーザーインタフェースや入力データなど、ユーザーから見える「システムの姿」の一部である、と言うこともできます。

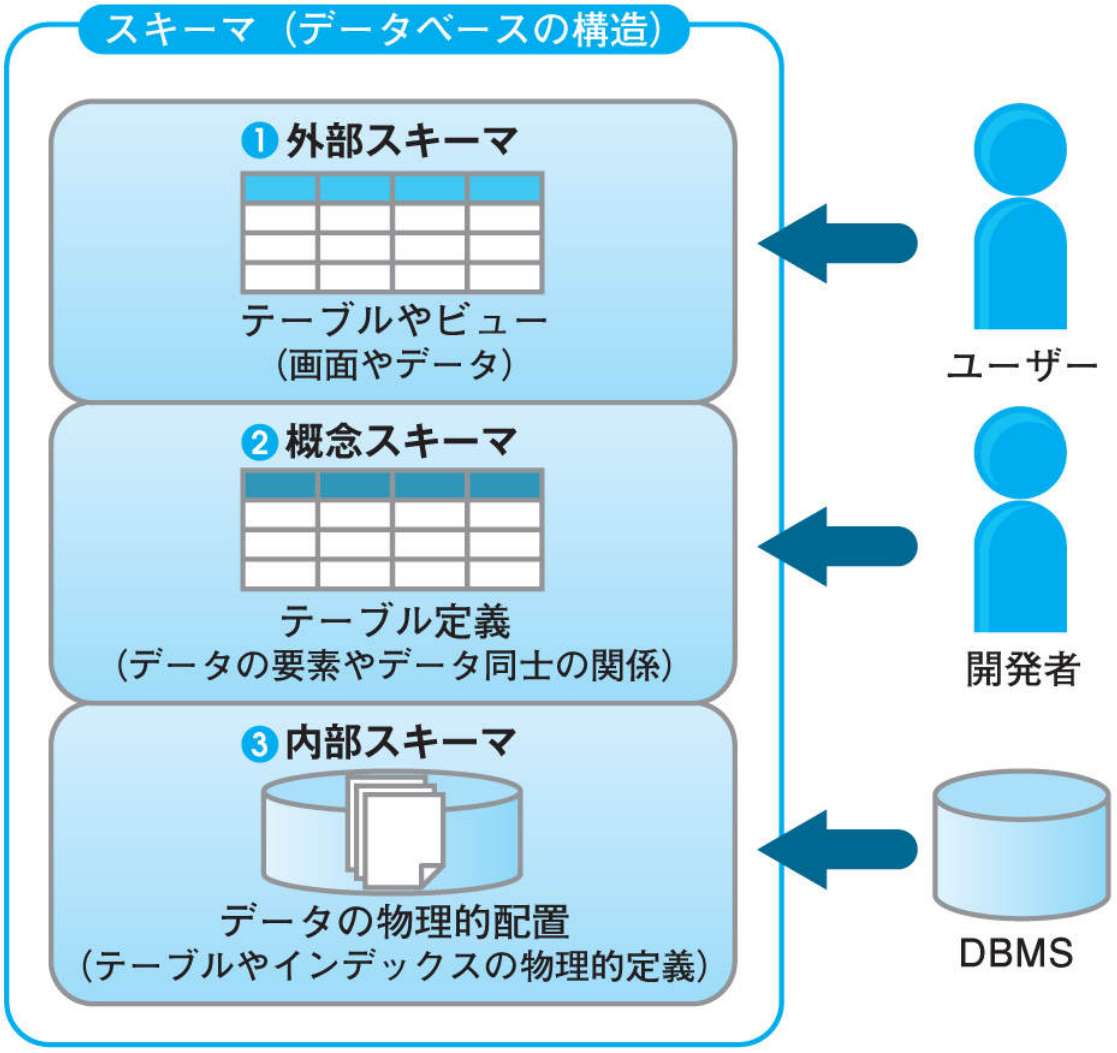


図1-11●データベースの3 層スキーマモデル

② 概念スキーマ

データベースに保持するデータの要素および、データ同士の関係を記述するスキーマです。外部スキーマがユーザーから見たデータベースだとすれば、概念スキーマは「**開発者から見たデータベース**」です。必然的に、データベース設計において重要な位置を占めることになります。概念スキーマの設計を「論理設計」とも呼びます（「論理」の意味については次章で説明します）。データベース設計の中心となるスキーマであるため、本書でもこのスキーマについては詳細に見ていきます。主に第3章（正規化）、第4章（ER図）で解説するほか、後続の章も基本的に概念スキーマに関係します。

③ 内部スキーマ

概念スキーマで定義された論理データモデルを、具体的にどのようにDBMS内部に格納するかを定義するスキーマです。いわば「**DBMSから見たデータベース**」です。テーブルやインデックス（索引）の物理的定義を含みます。リレーショナルデータベースもコンピュータ上で動く以上は、あらゆるデータは最終的に「ファイル」の形で管理されるわけで、その「ファイル」で表現される世界だ、と考えていただければ良いでしょう。内部スキーマの設計を、論理設計との対比で「物理設計」と呼びます。本書では第2章（物理設計の概要）および第6章（パフォーマンス）で主に取り上げます。

概念スキーマとデータ独立性

この3層スキーマを初めて見た人が感じる共通の疑問があります。それは、

概念スキーマは何のためにあるのか？

という疑問です。外部スキーマと内部スキーマの必要性は、わかりやすいものです。ユーザーにどのようにデータを見せるか、というのは、システムの機能や使い勝手にダイレクトに関係することですから、外部スキーマの定義は必須ですし、内部スキーマも、データを実際にDBMS内部に格納するための形式を考える時点で、おのずと定義することになります。

しかし、その中間に存在する概念スキーマは、なぜ必要なのか最初は理由がよくわからない、という人も少なくありません。この理由を理解するには、「もし概念スキーマがなかったらどうなるか？」という問いを立ててみることです。実は、小規模のシステム開発だと、あえて概念スキーマを定義せず、外部スキーマや内部スキーマに吸収させてしまうこともあります。名づけるなら「2層スキーマ」でしょうか。しかし、システムの規模が大きくなると、この「2層スキーマ」には非効率な部分が多いことがわかります。その理由は、**変更に対する柔軟性がない**ことです。

もし概念スキーマがなかった場合、ユーザーが「データの見え方を変えたい」と思った場合、もちろん外部スキーマは変更が必要になるのですが、それだけでなく、内部スキーマまで変更の必要が出てくることがあります。これと反対に、内部スキーマに変更を行なう場合にも、外部スキーマが影響を受けてしまうことが起こります。2層スキーマでは、スキーマ同士の独立性が低く（＝依存性が高く）なり、変更に弱いシステムができあがってしまうのです［※11］。

つまり、概念スキーマというのは、外部スキーマと内部スキーマの間に位置することで、両者の変更が互いに影響し合わないようにするための、**緩衝材**（かんしょうざい）の役割を果たしているのです（図1-12）。

このようなスキーマの独立性のことを、**データ独立性**と呼びます。外部スキーマからの独立性を**論理的データ独立性**、内部スキーマからの独立性を**物理的データ独立性**と呼びます。

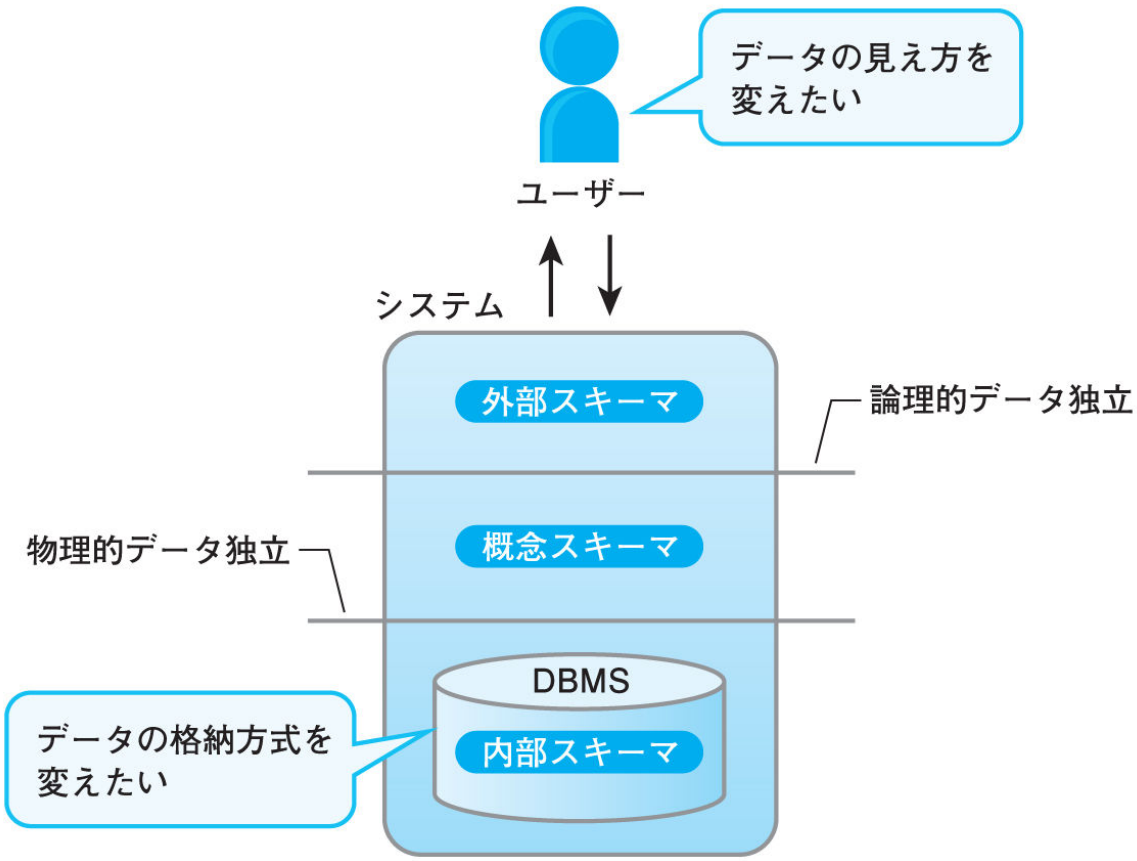


図1-12●概念スキーマは緩衝材の役割を果たす

勘どころ 7

概念スキーマはデータ独立性を保証するためにある。

システム開発の概念や方法論には、一見すると「なぜこれが必要なのだろう？

何の役に立っているのだろう？」という疑問が浮かぶようなものがあります。実は、そうした直観的に必然性の理解できない概念は、長年の試行錯誤によって生み出された工夫であることがしばしばです。私たちは、言ってみれば「後から」結果だけを学ぼうとするために、違和感を感じることになるわけです。そういう場合の学習のコツは、

これがなかったらどういう不都合が生じるだろうか？

と考えてみることです。そうすることで、直観では理解できなかった有用性が明らかになります。

勘どころ 8

概念の有用性がわからなかったら、「それがなかったらどうなるか」を考えてみよう。

また、これは余談ですが、システム開発においては、データベースとは別に、もう一つの3層スキーマが登場します。それは、Webシステムの構成モデルである「**Web3層モデル**」です。このモデルは、Webアプリケーションを用いるシステムでは、ごく標準的なモデルなので、仕事で利用する人は多いでしょう。しかし、このモデルを最初に学ぶときにその役割がイメージできなくて困るのが、やはり真ん中のスキーマなのです。これについては、コラム「Web3層モデル」で取り上げますので、興味のある方は参照してください。

Web3層モデル COLUMN

システム開発の世界には、多層的にレイヤーを重ねるアーキテクチャ、すなわち多層アーキテクチャがいろいろなところに登場します。通信プロトコルを7層に分類した「OSI参照モデル」（図A）や仮想化を用いた多層モデルはその一例です。

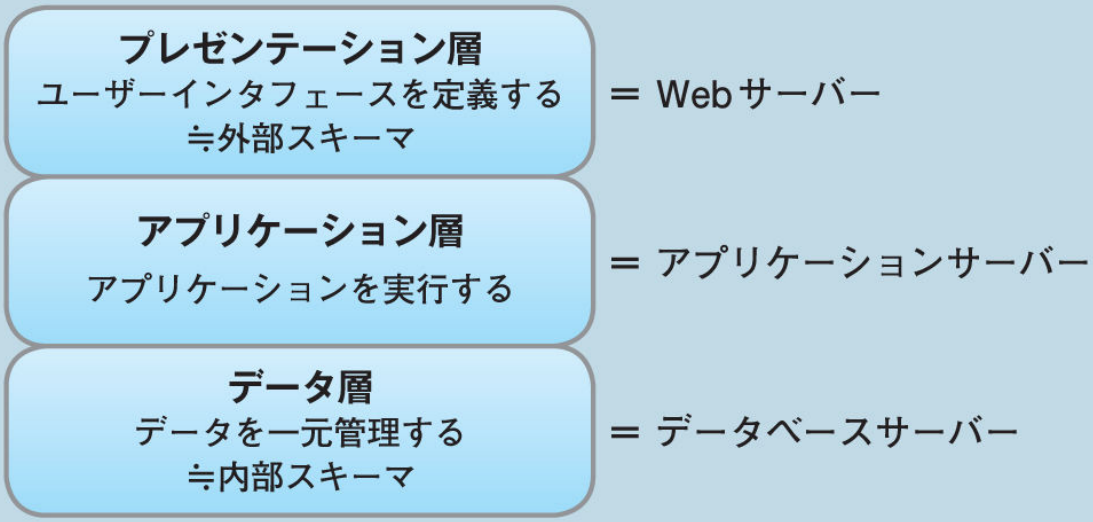
システムにおいてこのように多くのレイヤーのレベルを定義する理由は、システムが究極的にはハードウェアという物理層のレベルで動作するのに対して、そのような物理層の表現は、人間が認識しやすいものではないからです。よく知られているように、物理層では情報は0/1のビット形式に変換されていますが、それを直接人間が認識しようとするのは非効率です。したがって、より抽象度の高い表現形式を可能にするために上位のレイヤーを重ねていくことになります。

図A●OSI参照モデル

図A●OSI参照モデル

その観点から見れば、本文でも触れたリレーショナルデータベースの3層スキーマは、内部スキーマという物理層に近い表現形式を、ユーザーから隠蔽するためにある、という解釈も可能です（実際、ほとんどのユーザーは、直接テーブルのファイルを見せられてもチンプンカンプンです）。

3層スキーマと同様、一般的にWebシステムでよく利用される3層モデルに、「Web3層モデル」というものがあります。これは、システムを図Bの3つのレイヤーに分割する構成です。



図B●Web3 層モデル

「プレゼンテーション層」とは、ユーザーインタフェースを定義する層で、つまりユーザーへのシステムの見え方を決定します。「データ層」というのが、本書で私たちが学んでいるデータベースの層、すなわちデータを一元管理する層です。この二つの役割は、非常に理解しやすいものです。そして、リレーショナルデータベースの3層スキーマにおける「外部スキーマ」と「内部スキーマ」に、それぞれ対応する役割であるこ

ともに理解していただけるでしょう。

問題は、中間層である「アプリケーション層」です。代表的なアプリケーションサーバーには、Oracle WebLogic Server（オラクル）、WebSphere Application Server（IBM）、JBoss（レッドハット）などがあります。

しかし、アプリケーションサーバーの役割というのは、少しわかりにくいものです。実際、この層の役割を疑問に感じる初心者は、少なくありません。それはちょうど「概念スキーマ」の役割が理解しにくいのと同じです。ありていに言えば

「アプリケーションサーバー」というのは何のためにあるのだろう？

という疑問が生じるわけです。

実は歴史的に振り返ると、かつてはこのアプリケーション層が存在しないモデルが主流でした。クライアントとサーバーだけからなる「2層モデル」が主流だったのです。このモデルでは、プレゼンテーション層とアプリケーション層がクライアント側に位置し、データ層だけがサーバー側にありました。年輩のエンジニアの方は覚えていると思いますが、当時はクライアントにVisual Studioなどの製品でGUIアプリケーションを作っていたものです。かなりクライアントに多くの仕事をさせる、リッチクライアントなシステムでした。

このモデルには、主に以下のような欠点がありました。

- **業務を実現するためのビジネスロジックに変更が生じた場合、いちいちすべてのクライアントのプログラムを更新しなければならないのが煩雑。**
- **性能の貧弱なクライアントでは複雑な処理が実現できない。**
- **サーバーとクライアントの通信量が多く、ネットワーク帯域の狭かった昔は**

- **サーバとクライアントの通信量が多く、ネットワーク帯域の狭かった昔は回線がボトルネックになった。**

こうした理由から、ビジネスロジックとユーザーインタフェースを分離し、前者をサーバー側に持ってくることが、3層モデルの着想でした。最終的には、ユーザーインタフェースを担うプレゼンテーション層も、サーバー側に移されたため、最近のクライアントにはWebブラウザさえ入っていれば他には何のアプリケーションもインストールする必要はなくなりました（Adobe Flashなど、まだリッチクライアント思想のシステムも残ってはいますが）。

このように、Web3層モデルにおいても、中間層は非常に重要な役割を持っていますが、その存在意義は見えにくいものです。しかし、「もし中間層がなかったら」と考えてみると（事実昔はなかった）、その重要性が理解していただけるのではないでしょうか。

次章では、データベース設計において中心的なタスクである、概念スキーマおよび内部スキーマの設計、すなわち論理設計と物理設計について見ていきたいと思います。

演習問題

演習 1-1 DBMSの情報確認

みなさんが現在使っている、またはこれまでに使ってきたリレーショナルデータベースのDBMSについて、以下の情報を確認してください。

❶ 使用しているDBMSのバージョン、およびそのDBMSの最新バージョン

❷ エディション

❸ マニュアルの所在

DBMSの実装の違いは、特にアプリケーションの物理的な実装方法を検討する際に影響してきます。それぞれにサポートしている機能に差があったり、また独自拡張の機能を用意していることがあります。そうした点についての調査は、まずマニュアルを読み込むことが大事なのですが、その前提となるのがバージョンとエディションです。

演習 1-2 アプリケーション改修のタイプとコスト

12ページの「設計工程と開発モデル」において、開発モデルの主流であるウォーターフォールモデルでは、後半の工程に入ってからの手戻りが難しく、改修が高コストになりがちだ、と説明しました。

今、みなさんが参加しているシステム開発のプロジェクトがテスト工程に差し掛かっているとします。すでにプロジェクトも佳境、カットオーバーも近づいてきています。そんな中、テストでシステムに次のような問題が発見されました。

問題1 **性能試験において、ある夜間バッチ処理のSQLの性能が非常に悪く、要件上は1時間で終了するべきSQLが10時間かかることがわかった。**

問題2 **画面からオンラインで出力する帳票において、現在のレイアウトでは情報が不足していることがわかった。顧客との要件調整に漏れがあったことが原因だった。**

上記の問題に対して、実装および設計に対してどのような変更によって解決が可能か、思いつくだけの方法を提案してください。もっとも、まだデータベース設計について詳細を学んでいない現段階では、具体的な手段にまで落とし込むことは難しいでしょう。ここでは、大ざっぱな方針を考えてもらうだけでかまいません。そして、少なくとも第6章まで終えた後に、再度この演習問題に取り組んでみてください。

システムでは、データベース以外にもテキストやCSV、HTMLといった様々な形式のファイルという形でデータを保持するのが一般的であり、そうした「データベースの外」のデータ設計も、もちろん重要ですが、本書ではそうしたデータの設計は扱いません。

前節でデータベースのモデルが異なると設計技法が異なると説明しましたが、もちろんモデルが異なればプログラムも同じものにはなりえません。

この分類は、米国の標準規格を定める委員会ANSIによる定義です。これ以外にも、「概念／論理／物理」という三つで分類する方法など、3層スキーマの定義はいくつかあります。ただ、内容的にはほぼ重なるので、まずはANSIの定義を覚えておけば、あとは応用がききます。

そしてみなさんも現場で開発を行なえば実感するように、システムの開発や運用という仕事は変更対応の連続です。

第2章

論理設計と物理設計

データベース設計は、大きく論理設計（概念スキーマ）と物理設計（内部スキーマ）に分けられます。本章では、それぞれのポイントを示すとともに、信頼性（可用性）、性能、キャパシティといったデータベースに求められる要件を満たすため、どのような観点に注意して設計を行なうべきかを学びます。

学習のポイント

- 論理設計は物理設計に先立つ。少なくともそう意識していなければいけない。
- DBMSはユーザーに「ファイル」を極力意識させないようにしているが、設計者はファイルレベルで考えるのが物理設計のポイント。
- サイジングは難しい。
- RAIDは信頼性、性能、そして“財布”を考慮して決める。
- バックアップおよびリカバリの設計は地味なタスクだが、おろそかにすると“新聞に載る”。

2-1 概念スキーマと論理設計

概念スキーマを定義する設計を、**論理設計**と呼びます。システムの世界では「論理」という言葉がよく登場しますが、それは通常の日本語の意味である「整合的で筋道が通っている」という意味ではなく、「**物理層の制約にとらわれない**」という意味で使います。物理層の制約とは、たとえばデータベースサーバーのCPUパワーや、ストレージ（一般的にはハードディスク）のデータ格納場所や、もう少し上位のレベルで言うと、読者のみなさんが使用しているDBMSで使えるデータ型やSQLの構文、といった、より具体的で実装レベルの条件のことです。最初にデータベース設計を行なう際は、ひとまずこうした物理層の制約はいったん脇において話を進めます。

システム開発におけるデータベース設計は、図2-1の手順で行ないます。

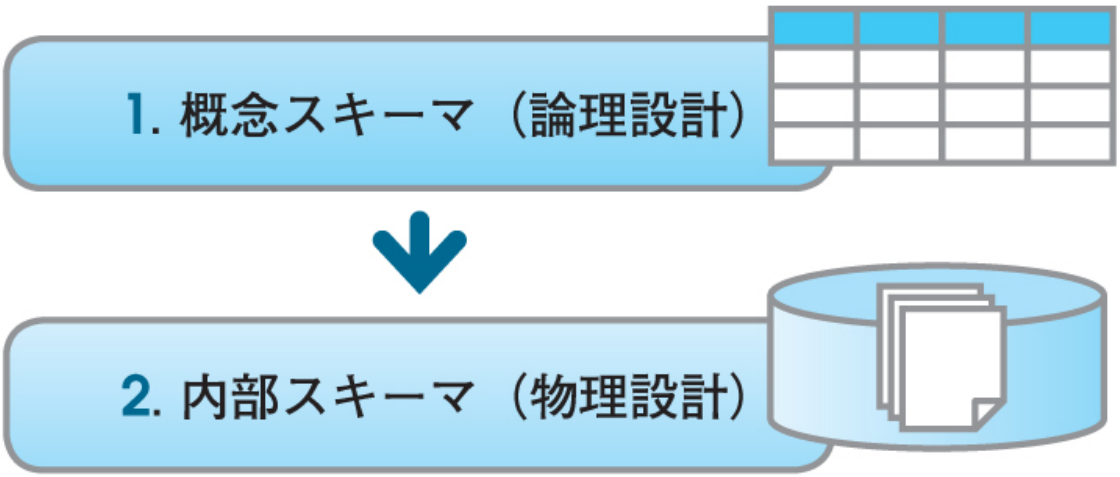


図2-1●データベース設計は原則として論理設計が物理設計に先立つ

論理設計が物理設計より前に位置しているのは、この設計が物理的制約には、原則として依存しないことを示しています。これは料理にたとえるなら、器を用意してから何の料理を作るか決めるのではなく、料理に合わせて器を決める、ということです［※1］。

論理設計のステップ

では、その論理設計で行なうことは何かですが、それは現実世界に存在する数多くのデータから、リレーショナルデータベースにおいて、何を、どのようなフォーマットで保存するかを決めることです。具体的には図2-2のようなタスク（作業）が含まれます。1 ～ 4はこの順序で行ないます。

図2-2●論理設計の四つのタスク

以降で、それぞれのタスクの内容と、行なうときに気をつけるべきポイントを見ていきましょう。

エンティティの抽出

エンティティ（entity）は、日本語で「**実体**」と訳します。現実世界に存在するデータの集合体を指す言葉で、具体的には「顧客」や「社員」「店舗」「車」といった、物理的実体を伴ったものもあれば、「税」や「会社」「注文履歴」のように物理的実体を伴わない、匂いもなければ触れもしない、単なる概念としてしか存在しないものも含まれます。「実体」という日本語には、何か「物理的な手触りのあるもの」というニュアンスがありますが、リレーショナルデータベースで扱うエンティティには、そのような含意は一切ありません［※2］。

勘どころ 10

「エンティティ（実体）」と言っても、物理的実体を伴う必要はない。

リレーショナルデータベースでは、こうした現実世界のエンティティを、最終的に「テーブル」という物理的単位で格納していくことになります。そのため、まずは、システムのためにどのようなエンティティ（＝データ）が必要になるかを抽出することが、論理設計の第 1 ステップです。

お気づきかもしれませんが、このタスクの一部は「要件定義」と重なっています。みなさんが作ろうとしているシステムでどのようなエンティティが必要になるか、という問いは、要するにどういうデータを扱いたいか、という問いの変形だからです。受発注システムを作ろうとしているのに顧客リストをデータ化（＝エンティティとして抽出）しないというのは考えられませんし、学生の講義履修登録システムを作ろうとしているのに、学生や教授をデータ化しない、というのも考えられません。したがっ

て、このエンティティ抽出は、ある程度、顧客やシステム利用者と要件を詰めていく中で実施することになります。

エンティティの定義

エンティティを抽出した後は、各エンティティがどのようなデータを保持するかを決める必要があります。エンティティは、データを「**属性**（attribute）」という形で保持します。難しい言葉だと思うかもしれませんが、これは二次元表における「列」と同義だと考えてください。本書でも、「属性」と「列」は互いに同義の言葉として使います［※3］。リレーショナルデータベースでは、二次元表に近い「テーブル」というフォーマットでエンティティを保持しますが（図2-3）、各表がどのような「列」を持つか、ということを定義するのがこの作業です。

ここで特に重要なのは、「**キー**（key）」という列を定義することです。キーとは、ある特定の列の値を決定するための列（複数列でも良い）のことです。こうした「テーブル」や「キー」といった、リレーショナルデータベースの基礎概念については、3-1節で詳細を解説します。

名前	住所	電話番号	メールアドレス
翔泳 太郎	東京都新宿区舟町 5	03-1234-5678	aaa@hoge.co.jp
出位多 米巣	春日部市粕壁東 3-20-50	048-9999-9999	bbb@test.or.jp
ミック	東京都江戸川区葛西 1-4	03-9834-1983	mick@hotmail
マック	静岡県浜松市中央 1-9-13	053-111-2222	mack@webmail.jp
⋮			

図2-3●リレーショナルデータベースのエンティティ

正規化

正規化（normalization）は、エンティティ（テーブル）について、システムでの利用がスムーズに行なえるよう整理する作業です。特に、正規化は更新（データの登録、変更、削除）が整合的に行なえるように、エンティティのフォーマットを整理することが重要な目的です。

裏を返すと、単にエンティティを抽出し、属性を定義しただけの状態では、まだそのエンティティはシステムでの利用にたえる状態にはなっていない、ということです。したがって、リレーショナルデータベースの論理設計においては、この正規化が最も重要な土台をなします。正規化は、データベースの論理設計を理解する鍵と言っても

過言ではありません。そのため、本書でも第3章で一章を割いて、ケーススタディを交えた解説を行ないます。

ER図の作成

ER図は、Entity-Relationship Diagramの略です。正規化を行なうと、エンティティの数が増えます。これは第3章で、正規化を学ぶとわかりますが、物理的な観点から見ると、正規化とは、エンティティ（＝テーブル）を細かく分割していく作業だからです。すると、そのままではエンティティ同士の関係が把握しにくくなるという問題が起こり、開発の効率を下げることになります。中小規模のシステムでもエンティティの数は数十個できるのが普通ですし、大規模システムになると、何百という数のエンティティが作られます。こうした大量のエンティティ同士の関係を、何の助けもなしに理解することは常人には困難です。

その問題を解決するために考案されたのが、エンティティ同士の関係を表現する図を作成する、という方法です（図2-4）。いわば「エンティティの見取り図」です。

このER図も、論理設計において非常に重要な役割を果たすことになります。ER図は、その記述の仕方にいくつかの流派があるのですが、本書ではスタンダードなER記法およびIDEF1Xの二つを取り上げます。ER図の記述の仕方については、第4章でケーススタディによって学びます。

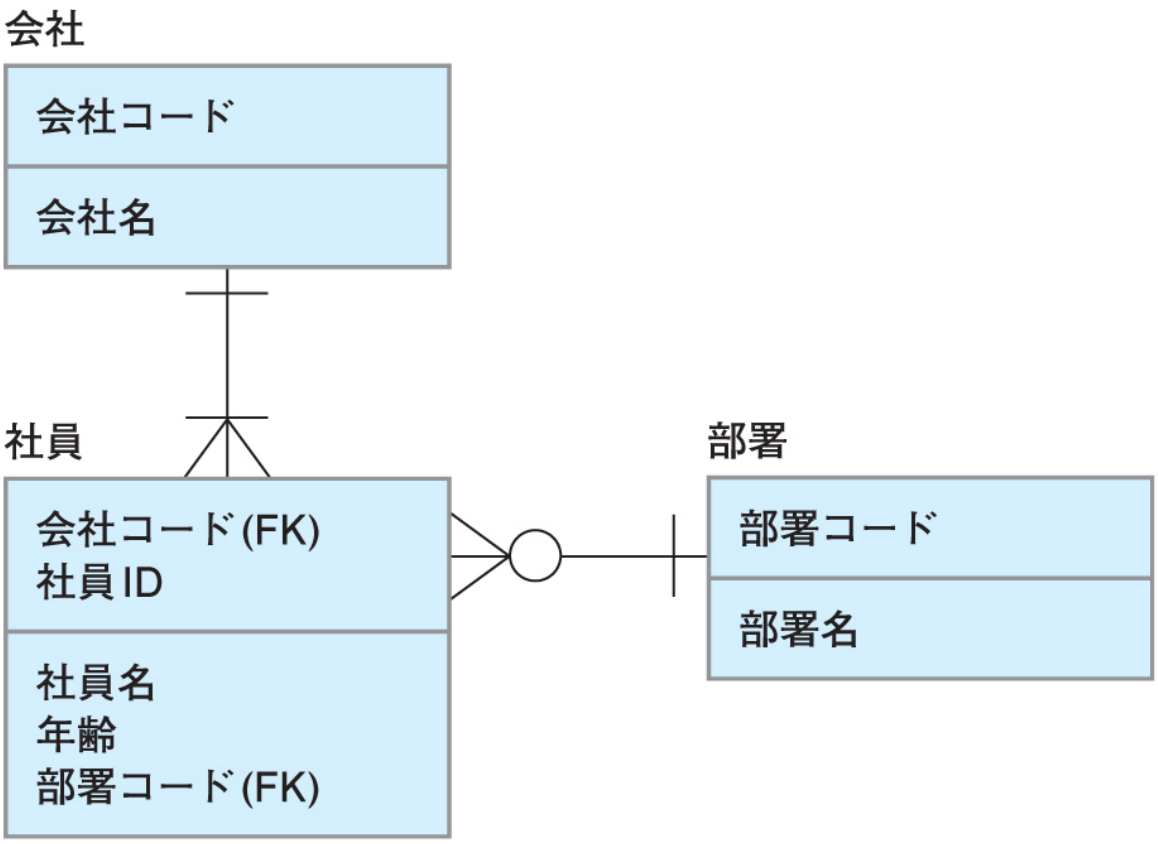


図2-4●ER 図

以上、論理設計の概要をざっくり説明してきました。こうした論理設計を行なう段階では、まだサーバーやストレージといった具体的なハードウェアを買ったり、DBMSのミドルウェアをインストールしたりする必要がありません。すべて**机上**で行なうことが可能です。その意味で、論理設計の「論理」は「机上で行なえる」というように考えてもらってもいいでしょう。この机上計算で作った見取り図をもとに、建築物の素材を用意していくのが、次に見る物理設計です。

もちろん、これは原則であって、実際には様々な要因（一番多いのは予算）によって、どうしても大きい器を用意することができず、小さな器に合わせて料理のメニューを決める、という順序で設計が進むこともあります。しかし、最初からそのような姿勢でシステムを作るのは、いわば

「戦わずして負ける」ということで、推奨はできません。あくまで原則は「論理設計が物理設計に先立つのだ」と覚えてください。

実際、第3章および第4章で詳しく見ますが、「関連エンティティ」という完全に理論的要請だけから導入される、それこそ「実体のない」エンティティもあるのです。しかしこの話はもう少し先に取っておきましょう。

「属性」のほうがより理論的な言葉なのですが、実務では違いを気にする必要はありませんし、そもそもあまりこの言葉自体を使いません。

物理設計は、論理設計の結果を受けて、データを格納するための物理的な領域や格納方法を決める工程です。物理層にはハードウェアを含むため、必然的にこの工程を行なうためには、ハードウェアやDBMSの個々の製品についての知識が必要になります。本節では、極力一般的に通用する大筋の流れを中心に解説します。

物理設計のステップ

物理設計には、大きく分類して以下の五つのタスクが含まれます。

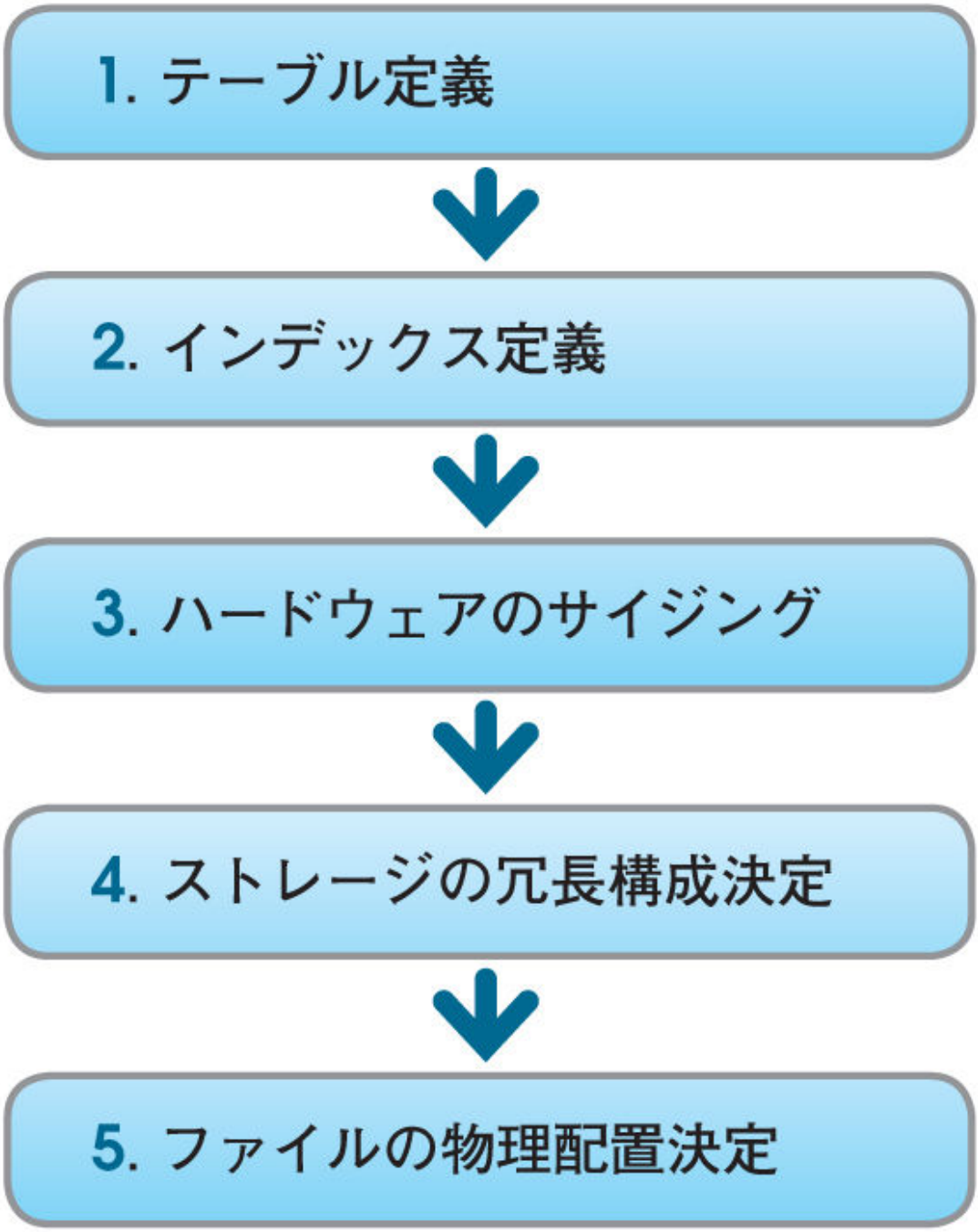


図2-5●物理設計の五つのタスク

以降で、それぞれのタスクの内容と行なうときに気をつけるべきポイントを見ていきたいと思います。

テーブル定義

論理設計で定義された概念スキーマをもとに、それをDBMS内部に格納するための「テーブル」の単位に変換していく作業です。論理設計で作られるERモデルを「論理モデル」と呼ぶのに対し、このフェーズで作られるモデルを「物理モデル」と呼びます。

テーブルとは、先ほどから何度か出てきているように、二次元表に近い形式を持っており、リレーショナルデータベースにおけるデータ管理の基本的な仕組みです。テーブルの詳細については、3-1節で解説します。

インデックス定義

インデックス（索引）は、リレーショナルデータベースにおいてテーブルと並んで重要な概念です。インデックスは、なくても機能的には何の問題もありません。データベースはきちんと機能します。インデックスが重要な役割を果たすのは、非機能の部分、つまりパフォーマンスです。インデックスを理解するには、ちょうど本の索引をイメージすると良いでしょう。特定の単語がどのページに存在するか見つけ出したいとき、1ページ目から順にページをめくっていくのは、ページ数が多くなればなるほど非効率で時間がかかります。本の索引は、この手間を省略して、ある単語が何ページ

目に書かれているかをダイレクトに示してくれるので、直接該当ページへ飛ぶことを可能にします。

インデックスについては、第6章でパフォーマンスについて解説する際に、詳しく述べます。

ハードウェアのサイジング

「サイジング」という言葉は、「サイズ（大きさ）」に由来しています。「大きさを決める」という意味ですが、システム開発では二種類の意味で使います。

一つは、データの規模を問題にする場合で、「システムで利用するデータサイズを見積もり、それに十分な容量の記憶装置（ストレージ）を選定する」という意味です。ここでは、文字通りデータの「サイズ（規模）」が重要です。「キャパシティの見積もり」と言い換えても良いでしょう。

サイジングのもう一つの意味は、パフォーマンスに関わる使い方です。こちらの場合、サイズを測る対象はサーバーのCPUやメモリです。つまり、システムが十分な性能を発揮できるだけのスペックのCPUやメモリを持ったサーバーを選定することです。実は、この意味でのサイジングの対象にはサーバーだけでなくストレージも含まれます。ストレージにもディスク回転数など性能を決めるスペックがあるからです。また実際のところ、性能問題のほとんどは**ストレージのI/Oネック**によって引き起こされます。

勘どころ 11

サイジングはキャパシティとパフォーマンスの2つの観点から行なう。

勘どころ 12

データベースの性能問題の8割はディスクI/Oによって起きる。

データベースにおいては、データの整合性とパフォーマンスの間に強いトレードオフが存在します。つまり、整合性を高くしようとするとパフォーマンスが犠牲になり、パフォーマンスを追求すると整合性を犠牲にする、という二律背反の原則です。データベース設計とは、限られた予算制約の中で、両端の間の平衡点を見つけ出そうとする努力だと言っていいでしょう。これは、本書全体を貫く一つのテーマと言って過言ではありません。したがって、本書でも今後、あらゆる章において「整合性か、それともパフォーマンスか」という二律背反の選択肢を常に問うことになります。ストレージのサイジングにおいて、パフォーマンスという観点を決しておろそかにできないのも、この理由によるのです。

サイジングを行なうには、以下のような入力情報が必要になります。

キャパシティのサイジング

► システムで利用するデータ量

データベース内に格納するデータ量は、物理的なテーブル定義およびインデックス定義が終わらなければ算出できません。したがって、この作業を実施するには、論理設計の終了が前提条件になります［※4］。また、ここでのデータ量には、データベース内に格納するテーブル以外にも、テキストや画像、HTMLといった様々な形式のファイルの分も加算する必要があります。

► サービス終了時のデータ増加率

データ量というのは、システムの運用開始から、基本的には増えていきます。中には、システムの特性上、ほとんどデータ量が増えないタイプのシステムもあります

（たとえば「直近n年」のデータだけ保持するような場合）。しかし、ほとんどのシステムでは、サービス終了時点では、サービス開始時点よりも増えるのが普通です（図2-6）。そのため、システムの運用終了時にデータ量がどの程度増えるかを見越しておかないと、途中でストレージの容量が足りなくなってしまう、という重大な問題が起きます。

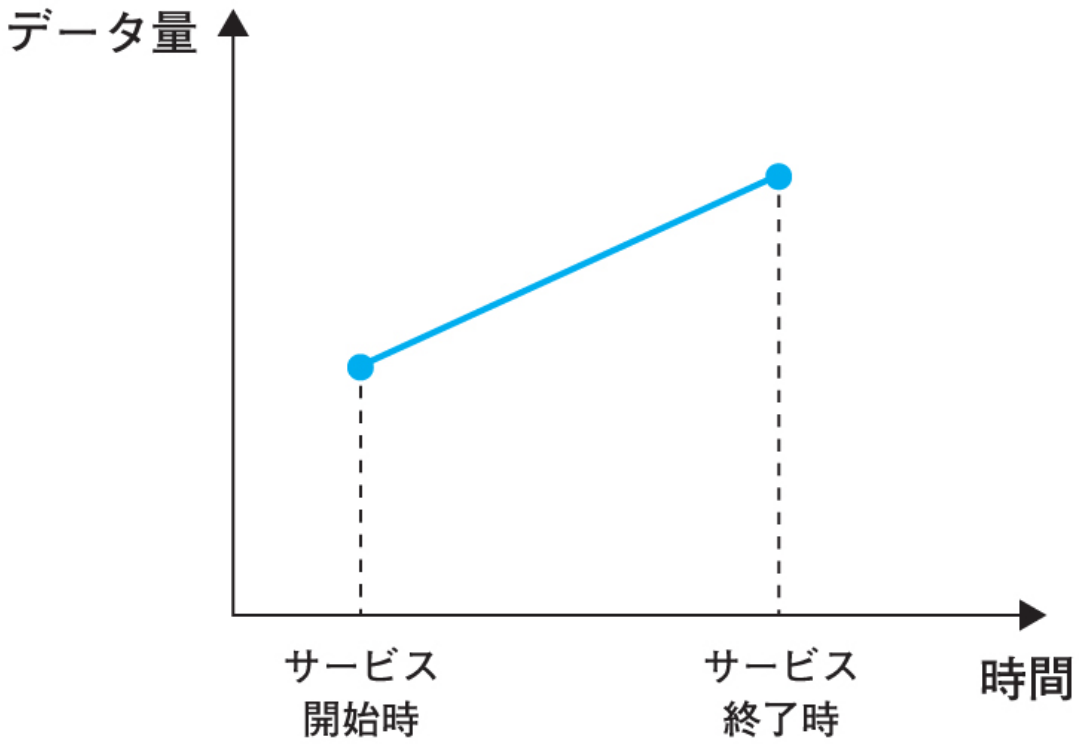


図2-6●サービス終了時のデータ量を見積もる必要があるが、なかなか難しい

ただ、実際の話、システムのサービス終了時のデータ量を正確に見積もることが難しい場合もあります。まったく新規のサービスを提供するシステムで、どれぐらいの需要があるのか見当もつかないような場合です。これに対しては、二つのアプローチがあります。

❶ 安全率を大きくとって、余裕を持たせたサイジングを行なう。

❷ 仮に後で容量が不足した場合に、簡単に記憶装置を追加できるような構成にしておく。

❷ のような後からの拡張が簡単な構成を、「スケーラビリティが高い」と表現します。「スケーラビリティ（scalability）」とは「拡張性」という意味です。負荷の増大に対処するため、柔軟にシステムを増強できる性質を表わします。このスケーラビリティ問題については、コラム「クラウドとスケーラビリティ」でもう少し掘り下げて解説します。

パフォーマンスのサイジング

性能要件

通常、システム開発では性能要件を二つの指標を使って定義します。

一つが**処理時間**。特定の処理について「何秒以内に終了すること」といった形で定義します。

もう一つの要件が**スループット**。これは単位時間当たりにどれだけの処理をシステムがこなせるかを示します。処理時間が「どれだけ速く処理できるか」の指標であるのに対して、スループットは「どれだけたくさん処理できるか」の指標です。こちらの単位は、「1秒当たり仕事量」を示す**TPS**（Transaction Per Second）という指標を使うのが一般的です。

この二つの要件を、要件定義の段階で決めておく必要があります。

勘どころ 13

性能要件の指標は二つ。「どれだけ速いか」と「どれだけ多いか」。

► リソース使用量の基礎数値

これは特に新規システムをゼロから構築する場合に問題になるのですが、どの程度の処理を行なうと、どの程度のハードウェアリソースを消費するのか、という相関関係は、机上では見えにくいものです。したがって、何か別の情報から類推するしかないのですが、それにしても根拠となる基礎数値が必要です。
この基礎数値を得る方法は二つあります。

方法1 類似の稼動中システムのデータを流用する。
方法2 開発の初期段階で**プロトタイプ**システムを構築して、性能検証を実施する。

方法1 は、安上がりですが、適当な類似システムが見つからない場合は精度が低くなります。一方、**方法2** は、きちんと実施すれば精度は高いのですが、プロトタイプの作成や検証実施に時間と人手がかかるため、予算とスケジュールに余裕があるプロジェクトでないと実施が難しいという問題があります［※5］。近年の開発プロジェクトは**どんどんスケジュールが短期化されている**という事情もあり、なかなかプロトタイプ検証を行なう余裕がない場合も多いです。
こうした理由から、パフォーマンスのサイジングにも、キャパシティのサイジングと同様の不確定要因（リスク）がついてまわります。そのため、同様に安全率およびスケーラビリティに留意したサイジングを行なう必要があります。

勘どころ 14

精度の高いサイジングは難しい。それゆえ、
・必ず実施時には安全率をかけること。
・スケーラビリティの高い構成を組むこと。

サイジングというのは、失敗すると被害が大きいわりに、精度を高く行なうのも難しいという、悩ましいタスクです。用意したストレージの容量が大きすぎるとか、リソースが余ってサーバーがスカスカ、というオーバーサイジングの場合ならまだ上司や顧客から「もっと安くあげられたんじゃないのかね」と嫌味を言われる程度で済みますが、用意したハードウェアスペックで不足した場合には目もあてられません。混雑時にパフォーマンス低下が発生するぐらいならまだしも、システム全体がダウンして巨額の損失を出してしまう、なんていう惨劇を招くこともあります。システムの中でも最も慎重なサイジングが求められるはずの金融機関のシステムでさえ、時折、処理集中による高負荷のためシステムダウンする、というニュースが報じられるぐらいです。それだけ、サイジングは物理設計の中でも難易度の高いタスクなのですが、これがこなせるようになればDBエンジニアとしてはエース級です。

クラウドとスケーラビリティ

COLUMN

読者のみなさんも、「クラウド」あるいは「クラウドコンピューティング」という言葉を聞いたことがあると思います。これは最近のIT業界の流行を表わす言葉（バズワード）です。

クラウドというのは、ネットワーク、特にインターネットの発達に密接に結びついたシステムの利用形態を指す言葉です。クラウドシステムでは、ユーザーの手元には非常に限られた機能だけを持ったクライアント端末（最近ではスマートフォンのような携帯端末や、タブレット端末であることも多い）だけを残して、実質的なサービスを提供する機能はネットワーク越しのサーバーに配置します。つまり、機能の所在という観点から見ると、クライアントを極限まで薄くして、サーバーを厚くするような形態を指します。

このクラウドは、すでに日常生活の様々なシーンで応用されており、みなさんの

このクラウドは、すでに日常生活の様々なシーンで応用されており、みなさんの中にも、DropboxやGoogle Appsといったクラウドサービスを利用している人は少なくないと思います。

このように、すでに私たちの生活に大きな変化をもたらしつつあるクラウドですが、システム開発に与える影響という観点から見ると、その影響はハードウェアの外部化と表現することができます。つまり、開発主体が自前でサーバーやストレージ、ネットワーク機器といったハードウェアを調達するのではなく、クラウドによって提供されているハードウェアの上にシステムを構築していく、ということです。このようなクラウドの利用形態を**HaaS**（Hardware as a Service）または**IaaS**（Infrastructure as a Service）と呼びます。

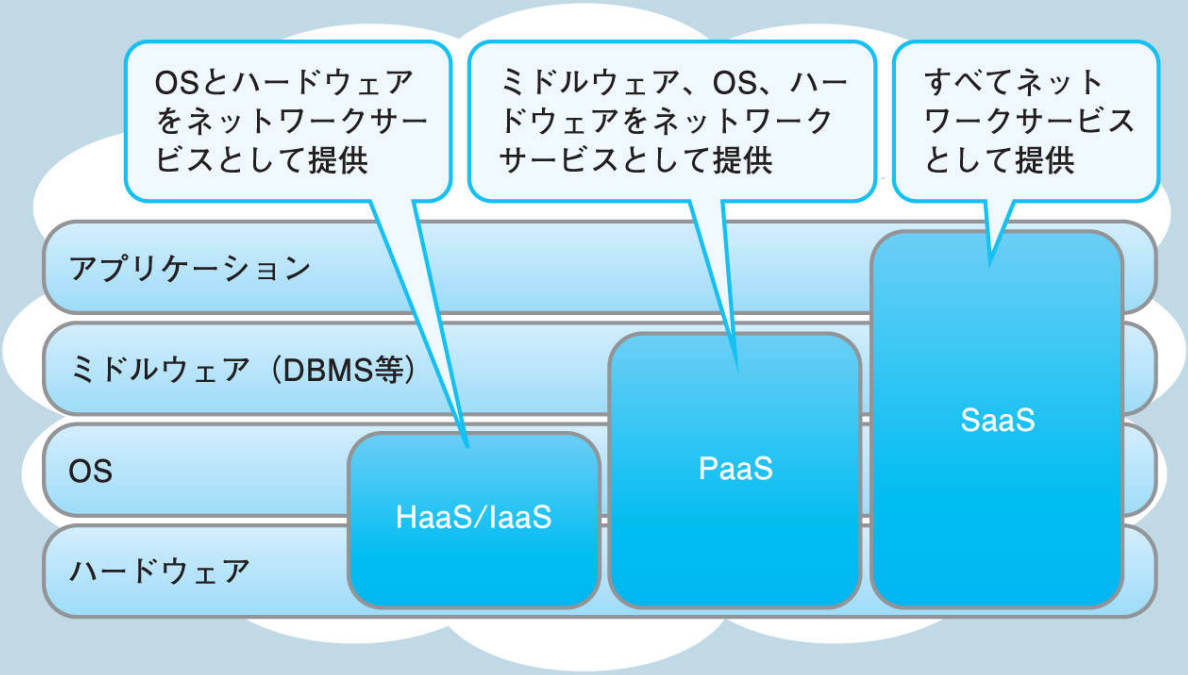
このHaaS/IaaSのメリットは、構成変更が柔軟に行なえることです。従来、ハードウェアというのは後から追加したり減らしたりすることが難しいものでした。足りないからといって（予算の面でも技術的な面でも）簡単に追加はできませんし、余ったからといって返品することはできません。

しかし、ハードウェアがクラウドによって「サービス」として提供されているなら、ユーザー側はそれを必要なときに必要とする単位だけ借りれば良い、ということになります。したがって、当面必要とする分だけ借りたけど、当初の予定より利用者が増えてリソース不足に陥りそうだ、となればもっとたくさん借りれば良いですし、逆にリソースが余ったなら、余計なリソース分を解約すれば良い、ということになります。これは、従来のシステム開発において頭痛の種の一つだった「スケーラビリティ」の問題を解決する方法になります。いわばシステム開発におけるクラウドとは「プラットフォームの賃貸契約化」です。

もちろん、現実はそれほど単純ではないので、クラウドサービスを利用したからといって、スケーラビリティの問題から完全に自由になれるわけではありません。しかし、これが有力な対策の一つであることは確かですし、HaaS/IaaSの考え方を

かし、これが有力な対策の一つであることは確かですし、HaaS/IaaSの考え方をさらに発展させるなら、DBMSなど上位層のレイヤーまでクラウドで提供する、という考えも必然的に生まれてきます。これを**PaaS**（Platform as a Service）や**SaaS**（Software as a Service）と呼びます。

　このように、クラウドの利用シーンはシステム開発においてもどんどん広がっており、多くのベンダーがサービスを拡充しています。こうしたキーワードも、覚えておいて損はありません。



図A●クラウドの形態

ストレージの冗長構成

次に行なう物理設計のステップは、ストレージの冗長構成の決定です。ストレージは、データベースのデータを保持する媒体で、一般的にはHDD（ハードディスクドライブ）を使用します。データベースに保管されるデータは、業務の基幹データなので、これを失うことは絶対に許されません。そのため、可能な限り高い耐障害性を持つようにシステムを構築する必要があります。ここで登場する技術が「**RAID**（レイド）」です。

RAIDは**R**edundant **A**rray of **I**ndependent **D**isksの略で［※6］、日本語に訳すと「独立したディスクの冗長配列」になります。複数のディスクを束ねて仮想的に一つのストレージとする技術で、この単位でまとめられたディスクを**RAIDグループ**と呼びます。RAIDのもともとの目的は、十分な信頼性の得られない安価なディスクで、なんとかシステムの保持するデータの保全を図ろうとすることでした（RAID のIに「**I**nexpensive（安い）」という単語が当てられることがあるのは、そのためです）。RAIDには何段階かレベルがあるのですが、基本的な考え方は、複数のディスクに同じデータを書き込んで**冗長化**することで、そのうちの一本が壊れても残りのディスクが生きていればデータを保全できるようにする、というものです。「冗長」とは「同じものを複数の場所に持つ」という意味です［※7］。

このように、RAIDは本来システムの信頼性（可用性）を高めるための技術なのですが、実はもう一つの利点が存在します。それが、**性能向上**です。

今から詳しく見ていきますが、ほとんどのレベルのRAIDでは、複数のディスクにデータを**分散**して保持します。そのため、システムにおいて最も性能的にボトルネックと

なるディスクI/Oを分散することで、パフォーマンス向上を図ることができるのです。

勘どころ 15

RAIDはシステムの信頼性と性能を共に改善できる技術。

データベースの物理設計においては、RAIDについて以下のことを考える必要があります。

- 当該データには、信頼性が求められるのか、それとも性能が求められるのか
- どのようなレベルのRAIDを採用するか
- 何本のディスクでRAIDを構成するか

最初の問いが最も重要で、これによって、RAIDのレベルやディスク本数が決まってきます。以降では、代表的な4種類のRAIDレベルを解説します。ここに挙げる以外に「RAID2」などのレベルもあるのですが、あまり実務で利用されないため省略します。

❶ RAID0

別名**ストライピング**。ストライプとは「縞々模様」のことです。データを異なるディスクに分散して保持することからこの名前がつきました（図2-7）。I/O性能はディスク本数が増えるほど向上しますが、ディスクのうち1本でも故障したらデータが失われるため、冗長性はまったくありません。そのため、人によってはこれを**RAIDとは認めない**人もいます。

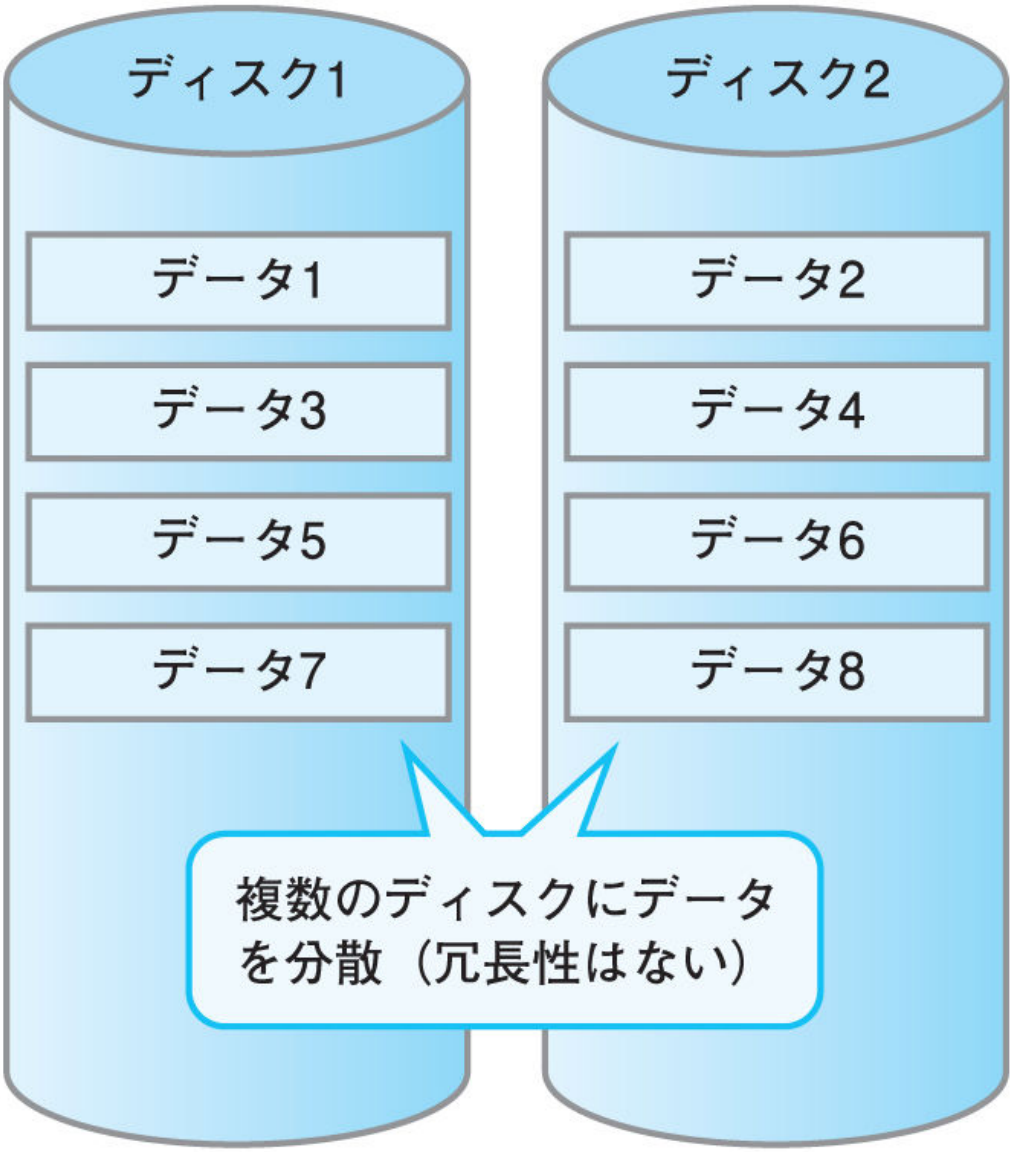


図2-7●RAID0 のイメージ

❷ RAID1

別名**ミラーリング**。「鏡」という言葉のとおり、2本のディスクにまったく同じデータを持ちます（図2-8）。そのため、冗長性は1本だけの場合に比べて2倍になり、2本のディスクが同時に壊れない限り、データは保全されます。信頼性は1本のときより

上がります。ただし、データは分散されないため、性能は1本の場合と変わりません。また、2本で一つのデータを保持するので、ディスクの使用効率もよくありません。

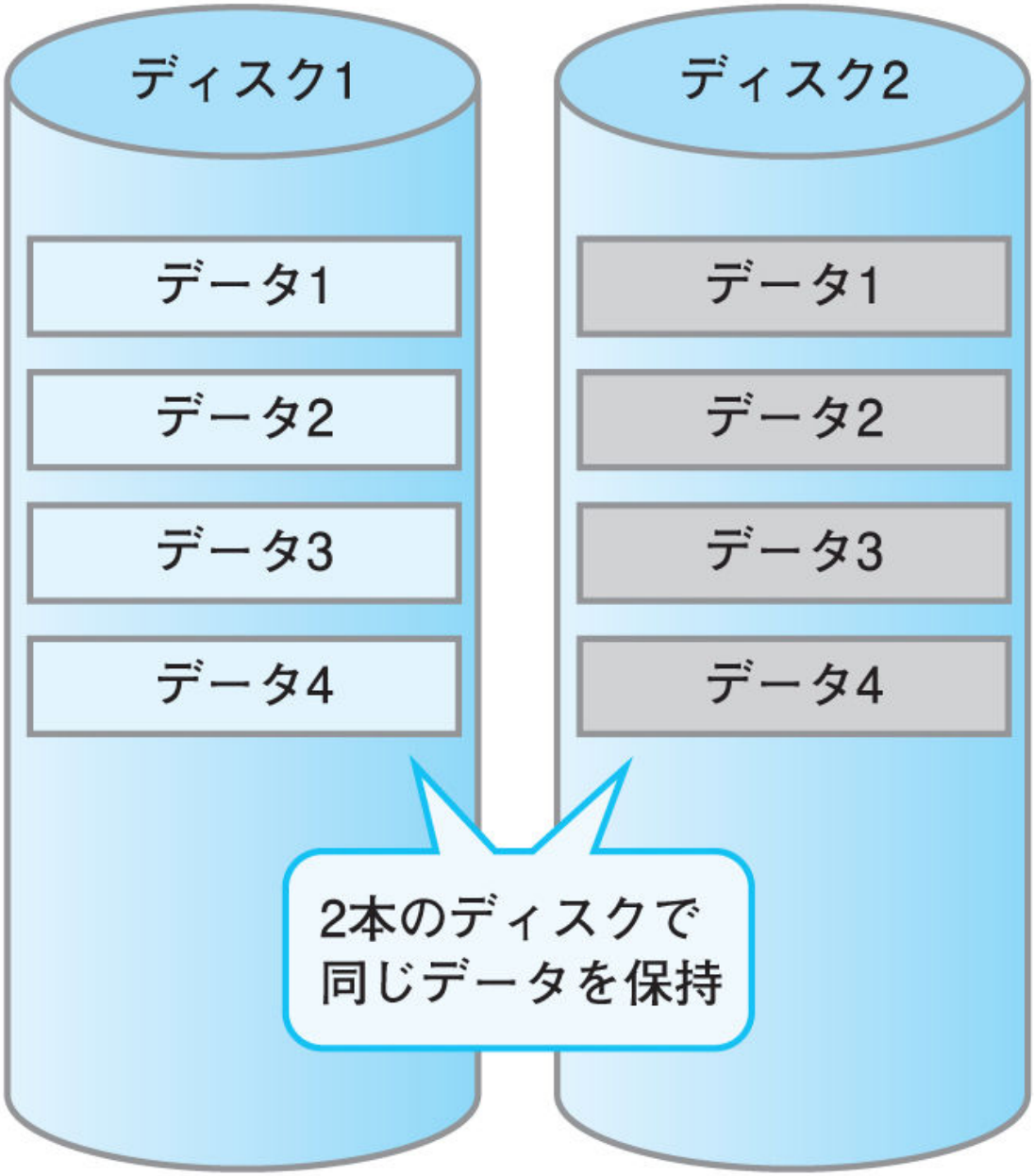


図2-8●RAID1 のイメージ

❸ RAID5

パリティ分散と呼ばれる方式です。最低3本で構成し、データとともに「**パリティ**」と呼ばれる誤り符号訂正符号を分散して格納します（図2-9）。ディスクが壊れたとしても、パリティから実データを復元することが可能です。このため、1本までならばどのディスクが壊れてもデータを保全できます（2本のディスクが同時に壊れると、データが失われます）。

また、データを分散できるためI/O性能（読み出し）の向上も期待できます。ディスク本数が増えるほど読み出し性能が向上します。パリティの計算を行なうため、書き込み性能は高くありませんが、通常、データベースは書き込みより読み出しのデータ量が多いため、読み出し性能が重視されます。

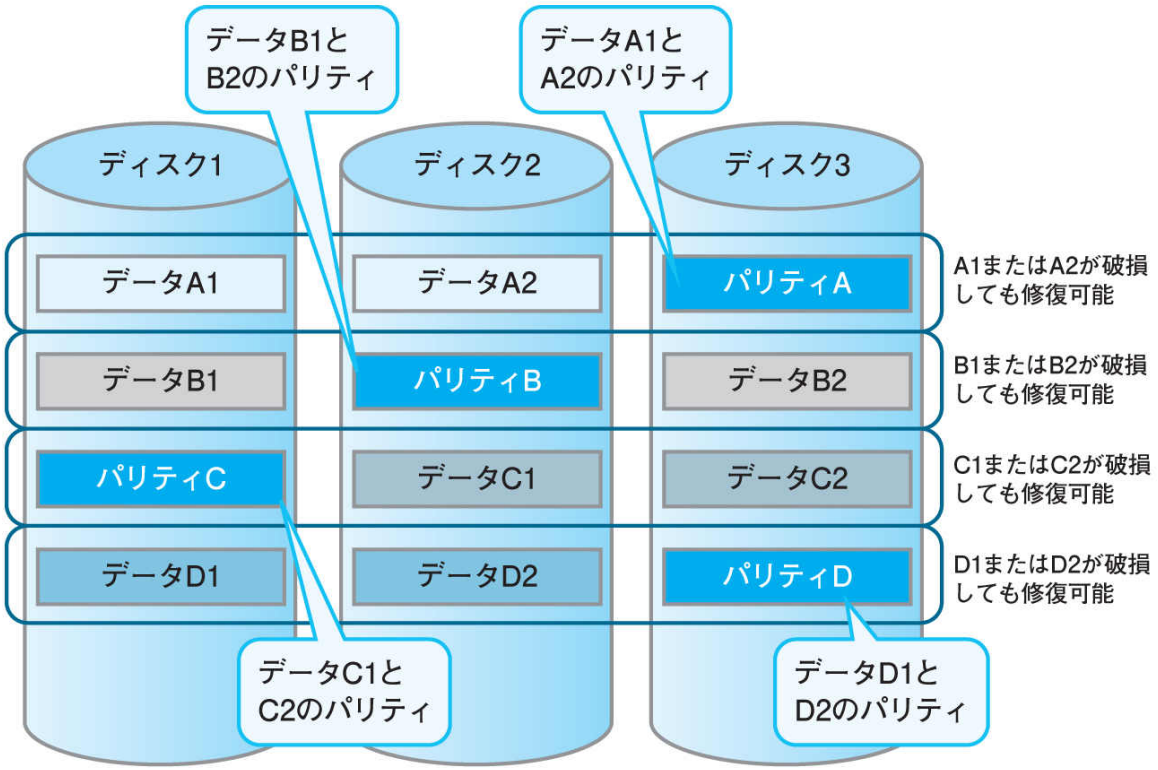


図2-9●RAID5 のイメージ

❹ RAID10

別名を「RAID1＋0」とも呼びます。その名前が意味するとおり、RAID1とRAID0を組み合わせたものです（図2-10）。最初にRAID1のグループを二つ作り、そのグループを使ってRIAD0を作ります。これは、いわばRAID1とRAID0の「**いいとこどり**」を目論んだ方法で、RAID1の高信頼性とRAID0の高速性を両立させます。

この方法の欠点は、必要になるディスクの本数が多いため、**コストが高い**ことです（最低でも4本が必要になります）。

これと似た構成で、ミラーリングとストライピングの順序を入れ替えた「RAID01（またはRAID0＋1）」という構成もあるのですが、同じ本数のディスクを使うわりに、信頼性はRAID10より若干劣るため、通常はあまり利用されません。

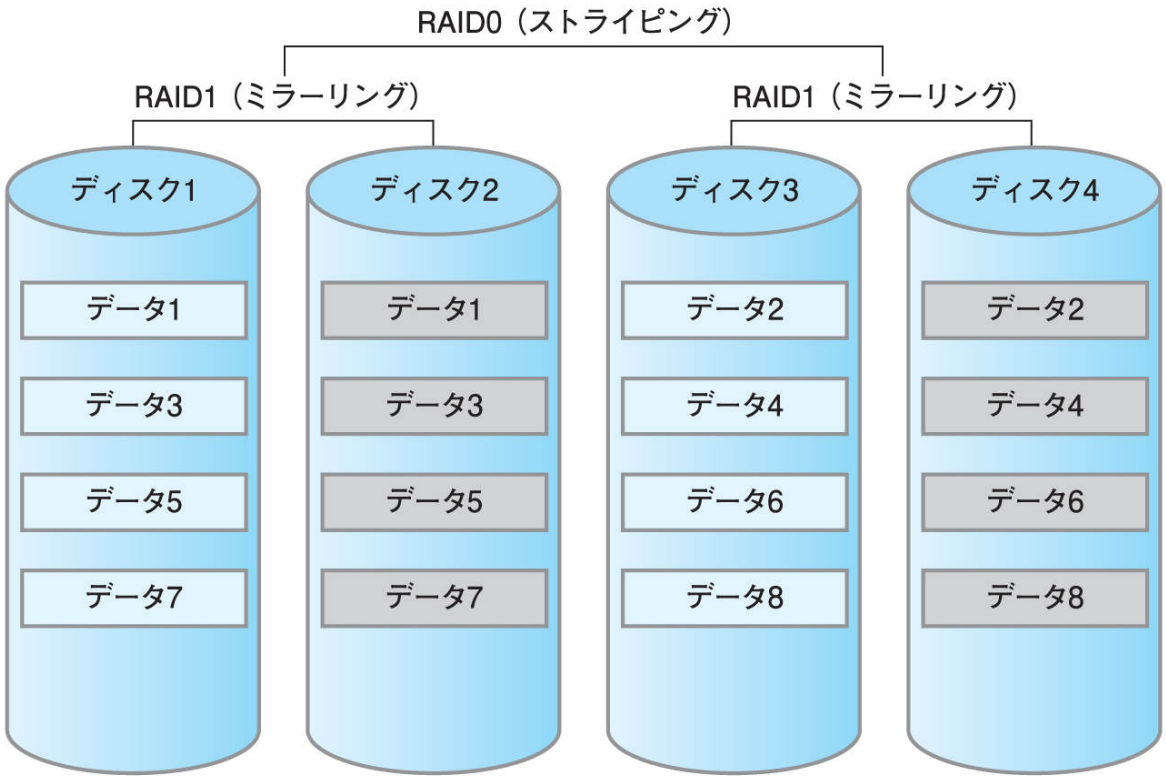


図2-10●RAID10 のイメージ

どのRAIDのパターンを採用するか？

これで基本的なRAIDのパターンが網羅できました。それでは、具体的に私たちはどのレベルのRAIDをデータベースサーバーのストレージとして利用すれば良いのでしょうか？

これは信頼性や性能の要件、およびコストをトータルに勘案して決める必要があるので一概に正答はないのですが、まず採用できるものならば **RAID10が望ましい** でしょう。これはストライピングとミラーリングのいいとこどり方式だからです。しかし、最低構成に必要なディスクが多く最もコストが高いため、財布に余裕がないと採用できません。

逆に、データベースとして絶対に採用してはならないのは、RAID0です。これは耐障害性が低いので、ディスクが1本壊れた瞬間に障害発生です。自宅のパソコンを組み立てているのではないのですから、間違ってもこのようなセーフティネットのない方式を使ってはいけません。

では、両極の中間に位置するRAID1とRAID5についてはどうでしょうか？

まず、RAID1は、信頼性は強固なので、その点では重要なデータを格納するデータベースにふさわしいのですが、パフォーマンス向上が期待できないのが難点です［※8］。一方、RAID5は、信頼性と性能を両方ともある程度向上させられます。性能に関しては、データ領域に使用するディスク本数を増やすことでより一層のパフォーマンス向上も期待できます。したがって、データベースサーバーのストレージにおけるRAIDの推奨構成は次のようになります。

勘どころ 16

データベースのRAIDは少なくともRAID5で構成する。お金に余裕があればRAID10。RAID0は論外。

RAID6

COLUMN

本文では、RAIDのレベルとして、0、1、5、10（および01）を紹介しました。ここでは、もう1種類のRAIDレベルについて触れておきたいと思います。それがRAID6です。

RAID6は、言ってみればRAID5の上位版です。RAID5が1本のパリティディスクを使用することで耐障害性を上げていたのに対し、RAID6は、パリティ以外にもう一つの冗長データをディスクに分散して持つ方式です（図B）。

図B●RAID6（P ＋Q 方式）のイメージ

これによって、RAID5ではディスクが2本故障した際にデータが失われる懸念が

これによって、RAID5ではディスクが2本故障した際にデータが失われる懸念があったのに対し、RAID6では二重障害時においてもデータを保全できます。つまり、RAID5よりもさらに信頼性が高い方式というわけです。

ただし、RAID5よりも最低必要なディスク本数が多く、二重に冗長データを生成するためにI/O性能が悪いなど、短所もRAID5より大きくなります。

ディスクの価格は低下の一途をたどっているため、最近ではディスク本数を多く使用する分信頼性の高いRAID6が、RAID5の代わりに利用されるケースも増えてきています。応用編として覚えておきたい方式の一つです。

ファイルの物理配置

さて、データベースのストレージの冗長構成が決まったら、物理設計の最終ステップは、データベースのファイルをどのディスク（またはRAIDグループ）に配置するかを考えます。実は、このファイルの配置については、最近のDBMSでは自動化が進んでおり、エンジニアが意識しなくても、ある程度はDBMSが自動的に配置してくれることもあります。しかし、やはり基本的な考え方を押さえておかなければ、思わぬ落とし穴にはまったり、問題が起きたときの対処が困難です。

まず、データベースに格納されるファイルは、用途別に以下の5種類に大別できます。DBMSによって細かな差異はあるのですが、基本はそれほど変わりません。

❶ データファイル
❷ インデックスファイル
❸ システムファイル
❹ 一時ファイル
❺ ログファイル

このうち、開発者がその存在を意識するのは、❶ データファイルと❷ インデックスファイルだけです。このファイルは、いわゆる「テーブル」のデータと、テーブルに付与されたインデックスのデータが格納されることになります。残りの三つはいずれも、DBMSの内部処理で使用されるファイルのため、通常は**DBA（DB Administrator）**と呼ばれるデータベース管理者以外は意識しません。以下、それぞれの用途と特性を説明します。

❶ データファイル

ユーザーがデータベースに格納するデータを保持するためのファイルです。つまりはテーブルのデータを格納するファイルであるため、業務アプリケーションがSQLを通じて参照および更新を行なうファイルでもあります。ただし、アプリケーションから見えるのはあくまで「テーブル」という論理的単位であって、「ファイル」が直接見えることはありません。

❷ インデックスファイル

テーブルに作成されたインデックスが格納されるファイルです。DBMSではテーブルとインデックスは普通異なるファイルとして管理されます。このファイルも、開発者が意識することはありません。というのも、SQLではテーブルへのアクセスを記述すること

はあっても、特定のインデックスに対するアクセスを記述することはないからです［※9］。インデックスを使うかどうかは、DBMSが内部で勝手に判断するため、ユーザーもインデックスの存在を意識することはないのです。

❸ システムファイル

システムファイルは、DBMSの内部管理用に使われるデータを格納します。基本的に業務アプリケーションやユーザーがアクセスすることはありません。

❹ 一時ファイル

一時ファイルは、名前のとおり、DBMS内部での一時的なデータを格納するために使われます。一時的なデータとは、たとえば、SQLで使われたサブクエリを展開したデータや、GROUP BY句やDISTINCTを利用したときのソートデータなどです。こうした一時データは、処理が終了すれば削除されてなくなるため、❶ ～ ❸ までのファイルと異なり、継続的にサイズが増加することはありません。

❺ ログファイル

DBMSは、テーブルのデータに対する変更を受け付けた場合、即座に❶ データファイルを更新しているわけではありません。いったん、このログファイルに変更分を溜め込んだ後に、一括してデータファイルに変更を反映しています［※10］。

このログファイルは、Oracleでは「REDOログ」、PostgreSQL、SQL Server、DB2では「トランザクションログ」、MySQLでは「バイナリログ」など、DBMSによって呼び方が異なります（本書では、次節以降、バックアップの説明をする際に「トランザクションログ」、または単に「ログファイル」という呼び名を使います）。また、このファイル

も一時ファイルと同じで、データファイルに反映が終われば不要になるため、継続的にサイズが増加するタイプのものではありません［※11］。

以上、5種類のファイルについて説明しました。一つ、注意点を補足しておくと、こうしたファイル群を、DBMSはテーブルや、さらにその上位の論理的な概念（たとえば、Oracleならば「表領域」、SQL Serverならば「データベース」）でラップして、ユーザーからできるだけ見えないようにしています。DBMSを操作する、という点ではそうした上位レイヤーの概念を扱っていれば良いのですが、容量と性能を設計するには、ファイルという物理層に近いところまで降りて意識しておく必要があります。

時々、表領域やデータベースといった論理レベルで分離すれば、I/Oが分散されると考えている人を見かけますが、これは物理層を意識しない人に典型的な勘違いです。

各ファイルの特徴

これら五つのファイルの特徴を表にまとめると、表2-1のようになります。

表2-1●DBMS ファイルの特徴

	用途	ユーザーからのアクセスは？	データ量の継続的な増加は？	性能の考慮レベル
テーブルデータの	データファイル格納	有（テーブル経由）	有［※］	高
インデックスファイル	インデックスの格納	有（インデックス経由）	有	高

システムファイル	管理用データの格納	原則として無（DBA のみ有）	有	低
一時ファイル	一時データの格納	無	無	高
ログファイル	更新ログの格納	無	無	中

※ もしテーブルのデータが一定である場合は、増加しません。

これら5種類のファイルの物理配置を考える際、最も重要なことは、（サイジングのときもそうであったように）サイズと性能です。サイズとは、もちろん容量の足りないディスクに配置してはいけない、ということです。これはほとんどの人が注意を払うところですが、もう一方の性能に関しては、あまり意識されません。

5つのファイルのうち、最もファイルI/O量が多いのは❶ データファイルです。業務データのテーブルを保持するのですから当然です。したがって、基本的な考え方としては、❶ と残りの❷～❺ を異なるディスク（RAIDグループ）に配置する、ということになります。

❶ の次にI/O量が多いのは、❷ インデックスファイルおよび❹ 一時ファイルです。この二つのファイルも、できれば独立したディスク（RAIDグループ）に配置することが望ましいです。❺ ログファイルは、データ更新が多いシステムではI/O 量が大きくなりますが、一般的には❶ データファイルの読み込み量に比べればそれほどではありません。❸ システムファイルは、DBMS内部の管理用にしか使われないため、I/O量は非常に低くなります。

以上から、最も望ましいファイル配置の例を挙げると、図2-11のようになります。

図2-11●性能的には理想的な、しかし高コストのパターン

すべてのファイルを異なるディスク（RAIDグループ）に配置します。これはファイルI/Oの分散という点では極めて望ましいのですが、ここまで潤沢にディスクを用意できることは滅多にないため、現実的ではありません。

もう少し妥協した案として考えられるのは、性能的に分散優先度の高いファイルを分離して、I/Oコストの低いファイルを一つにまとめるパターンです（図2-12）。

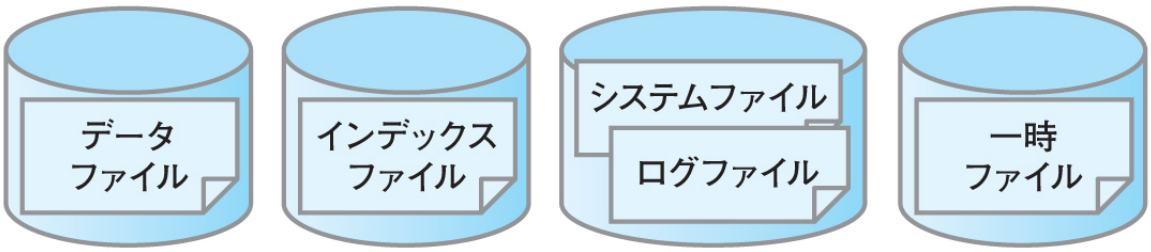


図2-12●性能的に少し妥協してコストを下げたパターン

I/Oコストの低いシステムファイルとログファイルを同じディスク（RAIDグループ）にまとめています。これでもまだコストが高いと言われるようであれば、あとはインデックスファイルや一時ファイルも格納するディスクを統合していくしかなくなりますが、それ

だけ性能への影響は大きくなっていきます。「あちらを立てればこちらが立たず」のトレードオフの関係が成立するのです。

なお、最近は上記5種類のファイルのほか、ストレージの大容量化に伴い、画像データなどのバイナリファイルをデータベースに格納することも増えています。このようなデータを「ラージオブジェクト」と呼ぶこともあります。データファイルの一つと見なすこともできるのですが、サイズが非常に大きくI/Oコストが高いため、このようなバイナリファイルの格納先も、他のファイル群（少なくともデータファイル）と分離することが望ましいです。

ただし、現実にはスケジュール上の制約から、ある程度並行で実施されることもしばしばです。

こういうときに、一般的なベンチマーク指標――有名なものとしてはSPEC（Standard Performance Evaluation Corporation）が公表しているもの――を基礎数値として使う、という代替手段もありますが、あまり推奨できません。ベンチマークで使用される処理モデルと、みなさんが開発しようとしているシステムの処理モデルが近いという保証がどこにもないからです。これはほとんど当て推量と精度は変わりません。

Redundant **A**rray of **I**nexpensive **D**isksの略とも言われており、この場合だと「安いディスクの冗長配列」という意味です。

実は「冗長」という言葉は、次章で見る「正規化」においてもキーワードです。ただし、RAID では冗長性を良いことだととらえるのに対し、正規化ではむしろ冗長性を極力なくすことを目指します。

逆に言えば、RAID1が適しているのは「性能はそれほど求められないが、高い信頼性が求められる」データ領域に対してです。そのため、OSなどのファイルを格納する領域に採用されることが多い構成です。

例外的に、「ヒント」と呼ばれる機能などを使って特定インデックスへのアクセスを指定できますが、これはあくまで例外的手段です。SQLのデータへのアクセスパスがどうやって決まるか、という問題については第6章を参照。

こういう回りくどい手順を踏む理由は、ユーザーから更新処理を受け付けるたびにチマチマとデータファイルを変更するより、一括処理したほうがパフォーマンスが良いためです。

ただし、障害復旧のためこのファイルをバックアップとして残す場合は、バックアップされるまでの一定期間データが増加を続けます。しかし、通常はバックアップが終了した後に削除するため、

単純増加はしません。

データベースの物理設計と隣接する領域に、データのバックアップおよびリストアの設計があります。データベースは、システムにとって極めて重要なデータが一元管理されており、いわばシステムの心臓です。それゆえ、万が一にでもデータベース内のデータが失われるようなことがあってはなりません。もしそのようなことが起きた場合は、多額の損害を覚悟する必要があります。大規模なシステムでこうしたデータ損失事故が発生したときには、新聞やテレビなどのメディアでも大々的に報道されます［※12］。

こうした「事件」を引き起こさないために必要な設計には、二通りの方針があります。一つは、極力データを失わないような設計にすること。これは先ほど解説したRAID設計が該当します。二つ目が、それでもなお、障害によってデータが失われたときに、復旧できるようにしておくことです。これが、本節と次節で解説するバックアップとリカバリです。

バックアップの基本分類

みなさんも、普段、自宅や会社でパソコンを使っていると思います。それゆえ、ほとんどの人が、バックアップを取った経験（または、バックアップを取っていなくて大事なデータを失ったという、泣きたくなるような経験）があるでしょう。ここでのバックアップというのは、基本的には**ファイルのコピー**で行ないます。

システムにおけるバックアップも、基本的にはそのイメージの延長で考えれば良いのですが、システムにおけるバックアップに、「わかりにくい」というイメージを持っている人も少なくありません。この大きな理由は、バックアップにたくさんの種類があるからです。システム関係の書籍や製品のマニュアルを読むと「～バックアップ」という用語がたくさん出てきて、混乱してしまうことがあります。最初はそれらの意味や区別を理解するだけでも大変な思いをします［※13］。

しかし実は、バックアップについては、主要な分類基準は一つしかありません。その基準にそって分類される三つのバックアップ方式を覚えてしまえば、あとは補足的なものだけです。そこでまずは、主要な三つの方式について見ていくことにしましょう。

完全／差分／増分

主要な三つのバックアップ方式は、以下のとおりです。

1. フルバックアップ（完全バックアップ）
2. 差分バックアップ
3. 増分バックアップ

データベースのバックアップ設計においては、この三つの方式を組み合わせていきます（すぐ後で説明しますが、どれか一つだけ、ということは普通ありません）。この区分は、バックアップデータをどのような単位で分割するか、という基準に基づいており、「完全」とか「差分」は、その単位を示しています。それでは、具体的に詳細を見ていきましょう。

フルバックアップ

フルバックアップ（full backup）、または完全バックアップは、あらゆるバックアップ方式の基本で、最も単純で理解しやすい方法です。「フル（全部）」という言葉のとおり、ある時点でそのシステムで保持されているすべてのデータをバックアップする方式です。これは、通常私たちが自分のPCデータをバックアップする方法と同じなので、イメージも湧きやすいと思います。あるタイミングにおけるスナップショット（静止画）を取るようなものです。

フルバックアップで保存されたファイルには、バックアップ時点のデータがすべて含まれているわけですから、そのファイルさえあれば、バックアップ時点のデータをすべて復旧することが可能になります。

フルバックアップのイメージを図示してみましょう。図2-13のように、月～土曜まで毎日あるタイミング、たとえば22:00にフルバックアップを取得していたとします。そして、日曜の13:00に障害が発生したとします。すると、最新のデータを戻すのに必要なバックアップファイルは「❻」だけです。逆に、もし何かの理由でバックアップしたその❻のファイルが壊れてしまった場合には、最新の状態に戻すことはできなくなります

（これは悲劇ですが、決してありえない話ではありません）。次善策として、❺のファイルを使って金曜日の状態に戻すしかありません（もし❺も壊れていたら……）。

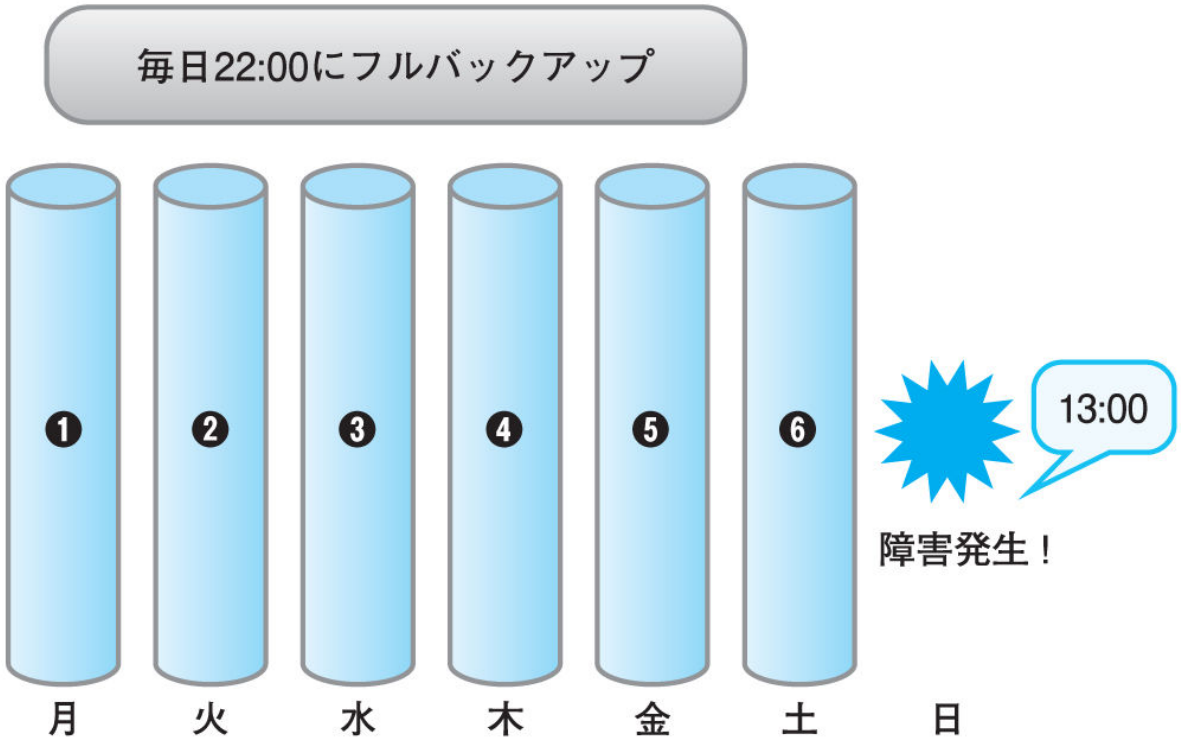


図2-13●フルバックアップの運用イメージ

このフルバックアップは、非常にシンプルでわかりやすく、また残りの二つのバックアップ方式の基礎にもなります。その意味で非常に重要なのですが、この方式だけではシステム運用ができないことが多いのです。それは、以下のような欠点も抱えているからです。

欠点1 バックアップの時間が長い

「長い」というのは、あくまで他のバックアップ方式と比べて、という相対的な意味においてです。フルバックアップは常にすべてのデータのバックアップを取得するため、三つのバックアップ方式のうち最長の時間がかかります。もちろん、具体的にどの程度の時間がかかるかは、システムで扱うデータ規模やハードウェア性能に依存するため、一概には言えません。

欠点2 ハードウェアリソースへの負荷が高い

この「高い」というのも、あくまで他のバックアップ方式と比べて、という相対的な意味においてです。バックアップするデータ量が多いということは、ストレージにおけるディスクI/O、サーバーのCPUおよびメモリの使用量も増える、ということを意味します。

欠点3 サービス停止が必要

フルバックアップを取得するには、一般的にはDBMSなどのソフトウェアを停止し、オンライン処理も閉塞した状態で実施します。これは、データの整合性を保った状態でバックアップを取得しなければならないためです。バックアップ中にデータが変更されてしまっては、データ整合性が取れません。Webシステムなどを利用していると、時々「ただいまシステムのメンテナンス中です」という画面が表示されて使えないことがありますが、あれは裏でフルバックアップなどを取得しているのです（もちろんそれ以外のメンテナンス作業も実施されています）。

このような特性から、フルバックアップには運用上の厳しい制限が課せられます。最近のシステムは24時間365日稼動も当たり前になってきており、サービスを停止

できる時間は極力短くすることが、要件上求められるからです。半年や一年単位でしかフルバックアップを取らない、というシステムも珍しくありません。

差分バックアップ

フルバックアップは、非常にシンプルでわかりやすい反面、いろいろと欠点も抱えた方式でした。そこで、フルバックアップの欠点を補うための方式を考えましょう。それが、ここで見る差分バックアップ（differential backup）と、次項の増分バックアップです。

先ほどと同様、図で表現してみましょう（図2-14）。今度は、毎日フルバックアップを取得するのではなく、月曜日だけにフルバックアップを取得するようにします。火～土の間は、月曜からの**変更分**だけのバックアップを取得することになります。この差分管理はどのようにして可能なのでしょうか？

実はここで、「ファイルの物理配置」の項（48ページ）で登場した「ログファイル」を使います。差分バックアップは、このログファイル（トランザクションログ）をバックアップすることで実現するのです。

もう一度おさらいですが、DBMSは、ユーザーから受け付けた変更を、すぐにデータファイルに反映するのではなく、いったんトランザクションログに溜め込むのでした（これはどんなDBMSでもそうです）。したがって、トランザクションログには、データベース内のデータに対するあらゆる変更操作の履歴が残っているのです。これをバックアップしておけば、いわば、**データベースに対する変更操作をもう一度再現（リプレイ）する**ことが可能になるわけです。

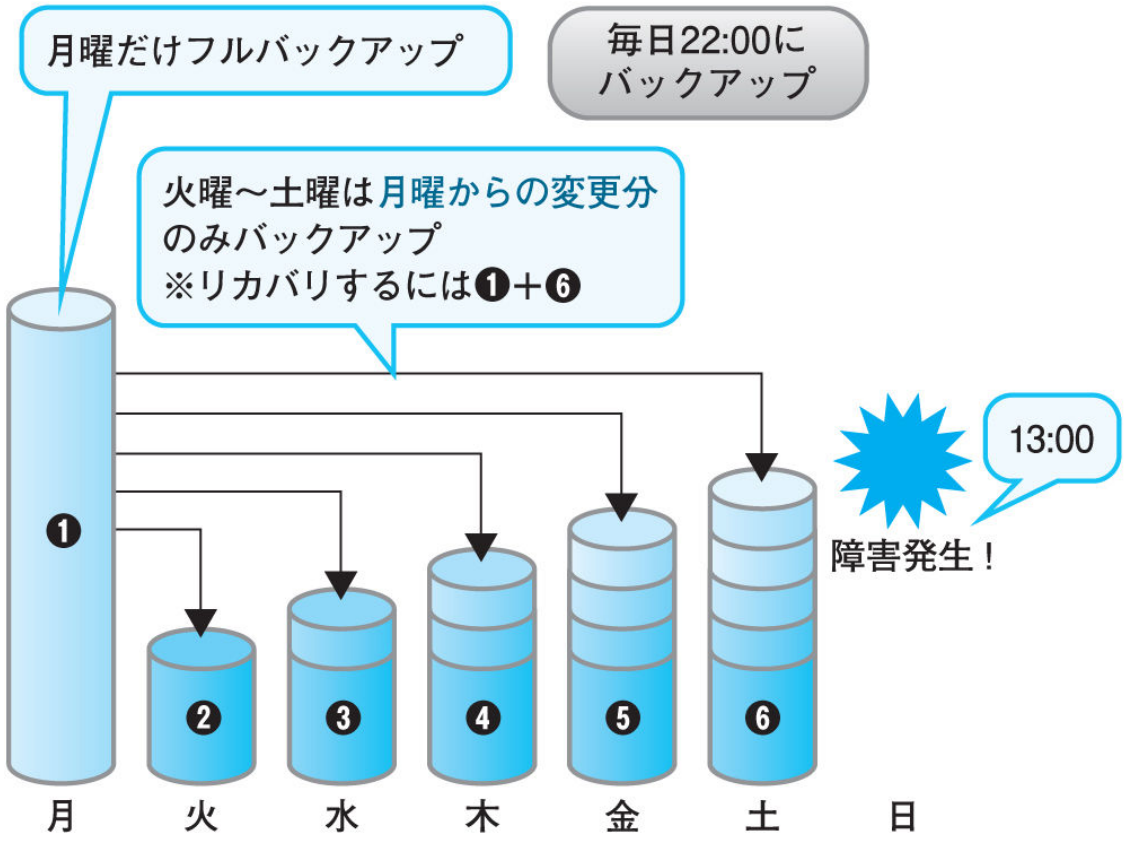


図2-14●フルバックアップ＋差分アップの運用イメージ

フルバックアップ＋差分バックアップの運用で、月曜日から運用を始めて日曜日に障害が起きたとします。このとき、最新データを復旧するのに必要なファイルは、「❶＋❻」となります。❶のフルバックアップのファイルは必ず必要です。変更履歴のログだけあっても、変更を適用するべき大元のファイルがなくては意味がないからです。一方、❷～❻のファイルは、いずれも差分バックアップのファイルではあるのですが、必要なのは最新の「❻」だけです。❷～❺の内容は、❻に含まれているからです（ここが、次に解説する増分バックアップとの違いです）。

さて、差分バックアップの利点と欠点は何でしょうか？

利点は、バックアップデータ量が減ることです。フルバックアップを毎日繰り返していたのに対し、差分バックアップを併用すれば火～土のバックアップ対象は変更分のログファイルだけで済みます。これによって、バックアップ時間も短くなるほか、バックアップファイルを保管しておく媒体の容量も小さくして節約できます。

一方、欠点は、リカバリのときに、フルバックアップのファイル「❶」だけでなく差分ログの「❻」も適用する必要があるため、リカバリの手順が増えて時間も長くなることです。リカバリというのは実施する時は相当せっぱ詰まっていて担当者も通常の精神状態ではないので、手順が複雑だったり時間がかかったりするといろいろとよろしくないことが起きます。

また、リカバリのためには「❶」と「❻」の両方のファイルが正常に使用できる必要があります。もし仮にどちらか一方でも壊れていたら、復旧はできません[※14]。

増分バックアップ

最後に紹介する方式は、増分バックアップ（incremental backup）です。これは、差分バックアップと基本的な考え方は同じです。それをある意味でより「効率的」にしたものです。

実は、差分バックアップには「無駄」があったことにお気づきでしょうか？ 何のことかというと、❷～❻のトランザクションログのファイルです。このファイルは、❷は❸に含まれ、❸は❹に含まれ、❹は❺に……という調子で、最新のログファイルが古いログファイルの内容も包含するという関係にあります。だからこそ、リカバリの際には最新ファイルの❻だけで良かったのです。

しかしこれは、同じデータを何度も何度もバックアップしていることになるので、バックアップ処理の効率性という観点からは無駄（冗長）です。そこで、トランザクションログから一切の無駄を省いたバックアップ方式が、この増分バックアップです。図2-15に示すように、常に必要なログしかバックアップしません。

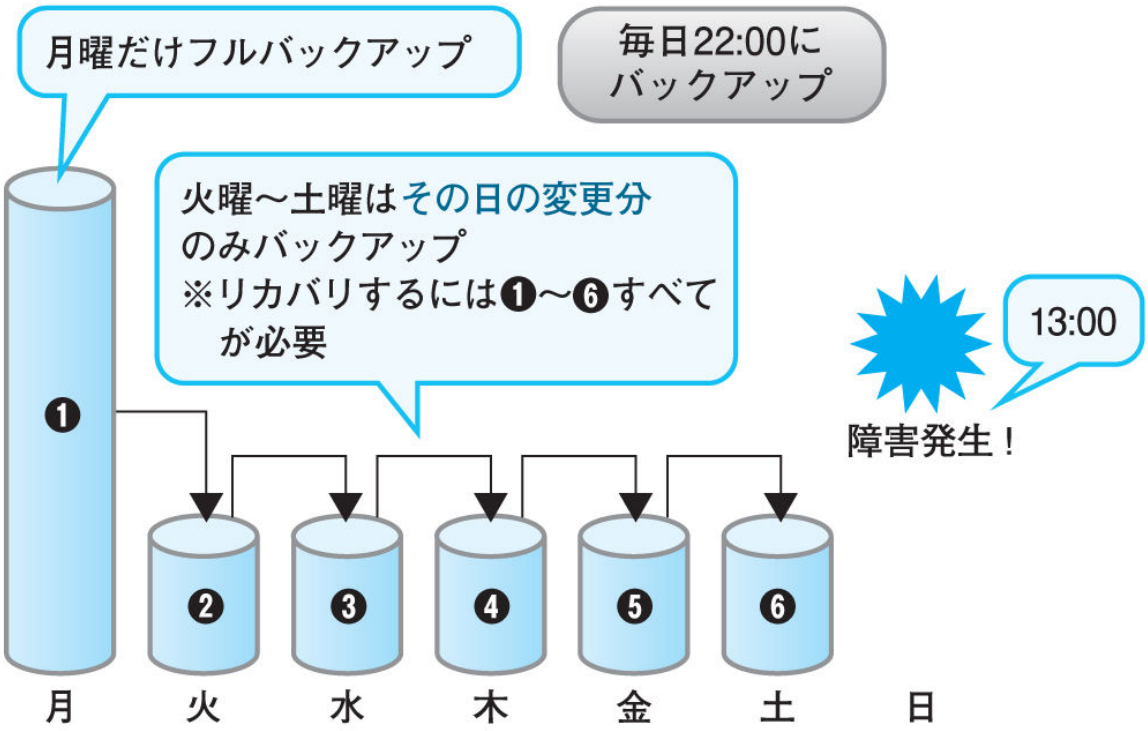


図2-15●増分バックアップの運用イメージ

増分バックアップの利点と欠点は何でしょうか？

まず利点は、バックアップデータ量が三つの方式のうちで最小になることです。必然的に、バックアップに要する時間も最短となります。バックアップファイルを保管するメディアの容量も最小で済みます。コスト的には最も優れた方式だと言えるでしょう。

欠点は、リカバリ手順が最も複雑になることです。なにしろリカバリには❶～❻のすべてのファイルが必要になります。リカバリに要する時間も長くなり、エンジニアはサービス復旧を今か今かと待ち望むユーザーからのプレッシャーと闘いながらリカバリを実施しなければならないでしょう。

また、復旧に必要なファイルも増えるため、完全にデータを復旧できる可能性が最も低くなります（なにしろ❶～❻のすべてのファイルが必要になるのです）。

バックアップ方式にもトレードオフがある

以上、主要三つのバックアップ方式を見てきました。フルバックアップを基本として、その欠点を補うために、差分バックアップと増分バックアップがあります。また、差分バックアップと増分バックアップは、ともに変更分のログファイルをバックアップする、という点で、同じカテゴリに属すると言って良いでしょう。

このため、差分バックアップと増分バックアップの区別があいまいな書籍や製品マニュアルにも存在します。両者を区別しないままどちらも「差分バックアップ」と呼んでいたりして、混乱を招く書き方をしているものもあります。しかし、すでに見てきたように、厳密には差分バックアップは、前回バックアップからのデータを**累積的**に保持する、という点が増分バックアップと違います（図2-16）［※15］。

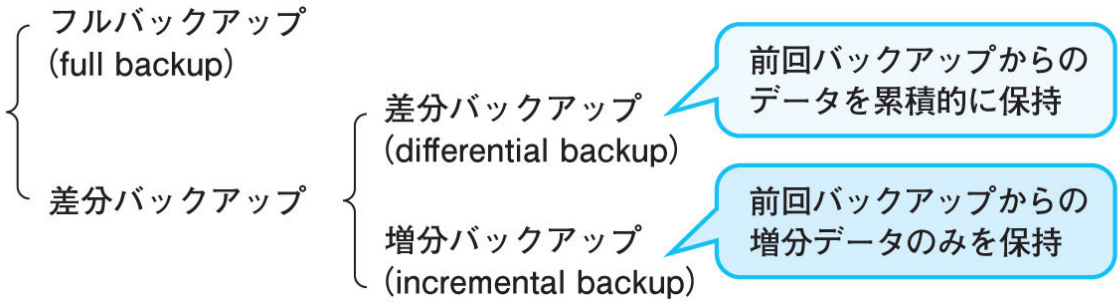


図2-16●バックアップの基本分類

三つの方式の特徴を表にまとめると表2-2のようになります。

表2-2●三つのバックアップ方式の特徴

この表からもわかるように、「バックアップコストが低いほどリカバリコストは高い」というトレードオフの関係があります（図2-17）。

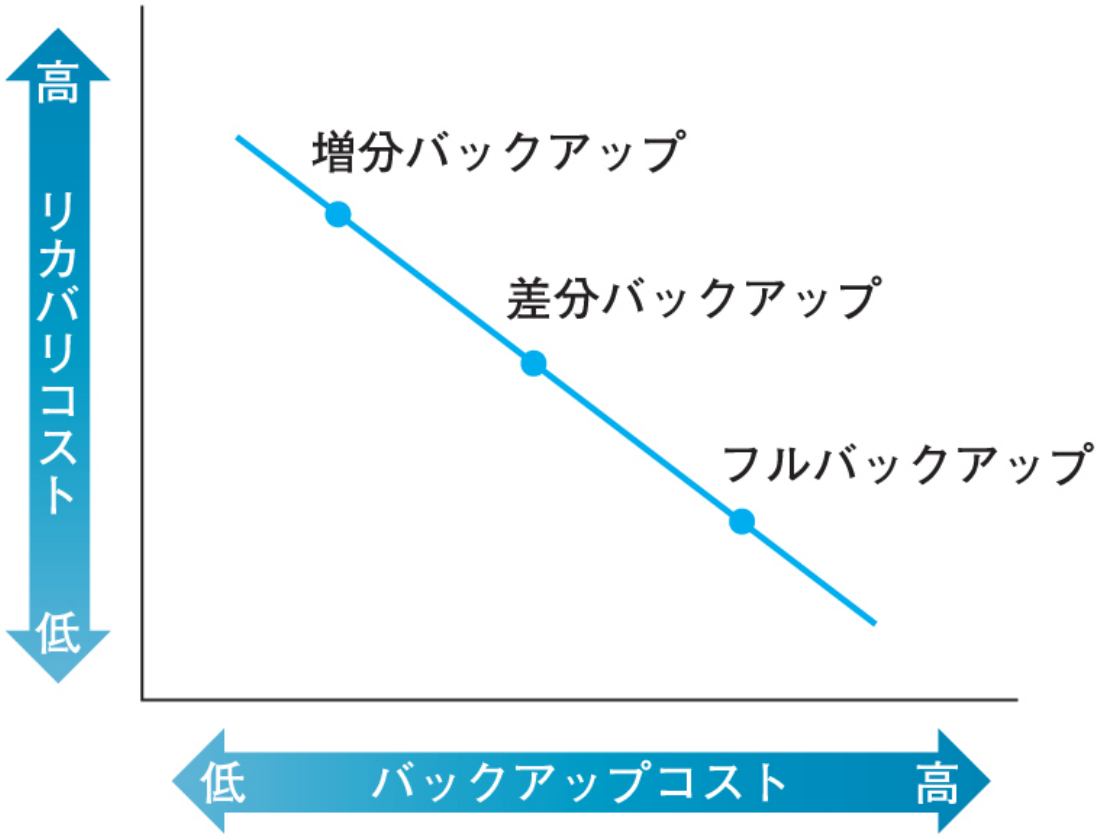


図2-17●バックアップとリカバリは二律背反

どんなバックアップ方式を採用すべきか？

さて、それでは実際にバックアップ設計を行なうとき、どのような方式を選択するべきでしょうか？

教科書的な回答としては、「それぞれの利点と欠点を比較検討して、システムの特性に応じて選択してください」となるのですが、それではちょっと無味乾燥なの

で、もう少し踏み込んで考えてみましょう。バックアップ設計において考慮すべきポイントは以下の4点です。

ポイント1 いつ時点の状態に復旧させる必要があるか。そもそも復旧の必要があるか

ポイント2 バックアップに使用できる時間（バックアップウィンドウ）

ポイント3 リカバリに使用できる時間（リカバリウィンドウ）

ポイント4 何世代までのデータを残す必要があるか（保管用の媒体サイズに影響）

一方、選択肢を網羅すると、次の4パターンになります。

❶ バックアップしない

❷ フルバックアップのみ

❸ フルバックアップ＋差分バックアップ

❹ フルバックアップ＋増分バックアップ

「バックアップをしないなんて、❶ は論外ではないか？」と思うかもしれませんが、一概にそういうわけでもなく、真剣な検討の対象になるケースがあります。それはどんなケースかわかるでしょうか？

答えは、バックアップファイル以外からデータを復旧させる手段がある場合です。たとえば、入力ファイルをシステムからもう一度登録することで、データベースのデータが復活するような場合です。

残りの三つの選択肢はどうでしょうか？

まず、フルバックアップのみというのは、バックアップの頻度にもよりますが、毎日やる場合は相当バックアップデータのサイズが大きくなることを覚悟しなければなりませんし、バックアップにかかる時間（**バックアップウィンドウ**）も大きくなります。また、最新の状態には戻せなくなることを覚悟しなければなりません。

したがって、現実的に選択されることが多いのは❸ および❹ の選択肢です［※16］。著者の開発経験に照らしても、この二つの方式でほぼ9割を占めます。ただし、この二つの方法はリカバリウィンドウが長くなる傾向があるため、その点を検討する必要があります。

勘どころ 17

バックアップ方式は、「フルバックアップ＋差分バックアップ」または「フルバックアップ＋増分バックアップ」が一般的。

バックアップの基本は上記の3種類を押さえることで理解できますが、実はもう一つ、バックアップ設計において検討の候補に加えておきたい手段として、レプリケーションがあります。これはバックアップと呼ぶには少し違和感を感じるかもしれませんが、重要な選択肢の一つであるため、後ほどコラム（260ページ）で取り上げます。

それにもかかわらず、システムのデータが失われたというニュースは毎年必ず数件は報道されます。気をつけてニュースをチェックしていれば、みなさんも目にすると思いますが、どうぞ当事者にだけはならないようにしてください。

また悪いことに、DBMSごとに独自の名称が使われていることがあり、それも理解を妨げる一因になっています。

正確には、「❻」が壊れている場合は、最悪「❶」だけで月曜の時点には戻せますが、「❶」が壊れていて「❻」だけ生きていた場合は、すべてのデータが永遠に失われます。

DBMSベンダーによっても用語に揺れがあることがあり、たとえばDB2は、一般的に「差分バックアップ」と呼ばれているものを「増分バックアップ（＝インクリメンタルバックアップ）」と呼んでいるため、間違えやすいので注意してください。代わりに一般的な「増分バックアップ」に相当する用語として「デルタバックアップ」が採用されています。「デルタ」は微積分を習った方はご存知のとおり、ギリシア語で「増分」を意味します。

差分バックアップも増分バックアップも、フルバックアップと組み合わせることが前提になるので、これら単独ということはありえません。

バックアップ設計とリカバリ設計はセットで実施することが一般的です。これは、バックアップ方式によってリカバリ手順が左右されるためです。そのため、実はリカバリ設計は、バックアップ方式が決まれば、自動的に決まってきます。

ところで、先ほどバックアップ方式の説明を読んだところで、次のような疑問を持った人はいないでしょうか？

> 「バックアップファイルだけからでは、データの状態を障害の直前に戻すことは不可能ではないのか？」

実は、この疑問は正しいのです。リカバリについて理解するには、この疑問点をまずは解消しておかねばなりません。

リカバリとリストア

どのようなバックアップ方式でもかまわないのですが、話を単純にするために、フルバックアップのみで実施する方式を仮定しましょう。

フルバックアップは月曜日から毎日22:00に開始され、23:00に終わるとします。障害が発生してデータが失われたのは、日曜日の13:00です。すると、土曜日にバックアップを取り終わってから、日曜の13:00までの間にも、当然、ユーザーがシステムを利用することによってデータがどんどん変更されていっています（図2-18）。しかし、土曜日に取得した「❻」のバックアップファイルには、その変更分が含まれていません。つまり、バックアップファイルだけでは、「障害の直前」の状態にデータを戻すことはできないのです。

これは、PCのファイルをバックアップする場合でも同様です。三日前に仕事用のドキュメントのファイルをバックアップしていたとしても、今日データが失われた場合は、ここ二日間の努力は水の泡になります。それと同じです。

つまり、障害直前の状態にデータを復旧させるためには、単にバックアップファイルをデータベースに戻しただけではダメなのです。そこからさらに、ユーザーの変更分を再反映させなければ、リカバリは完結しません。

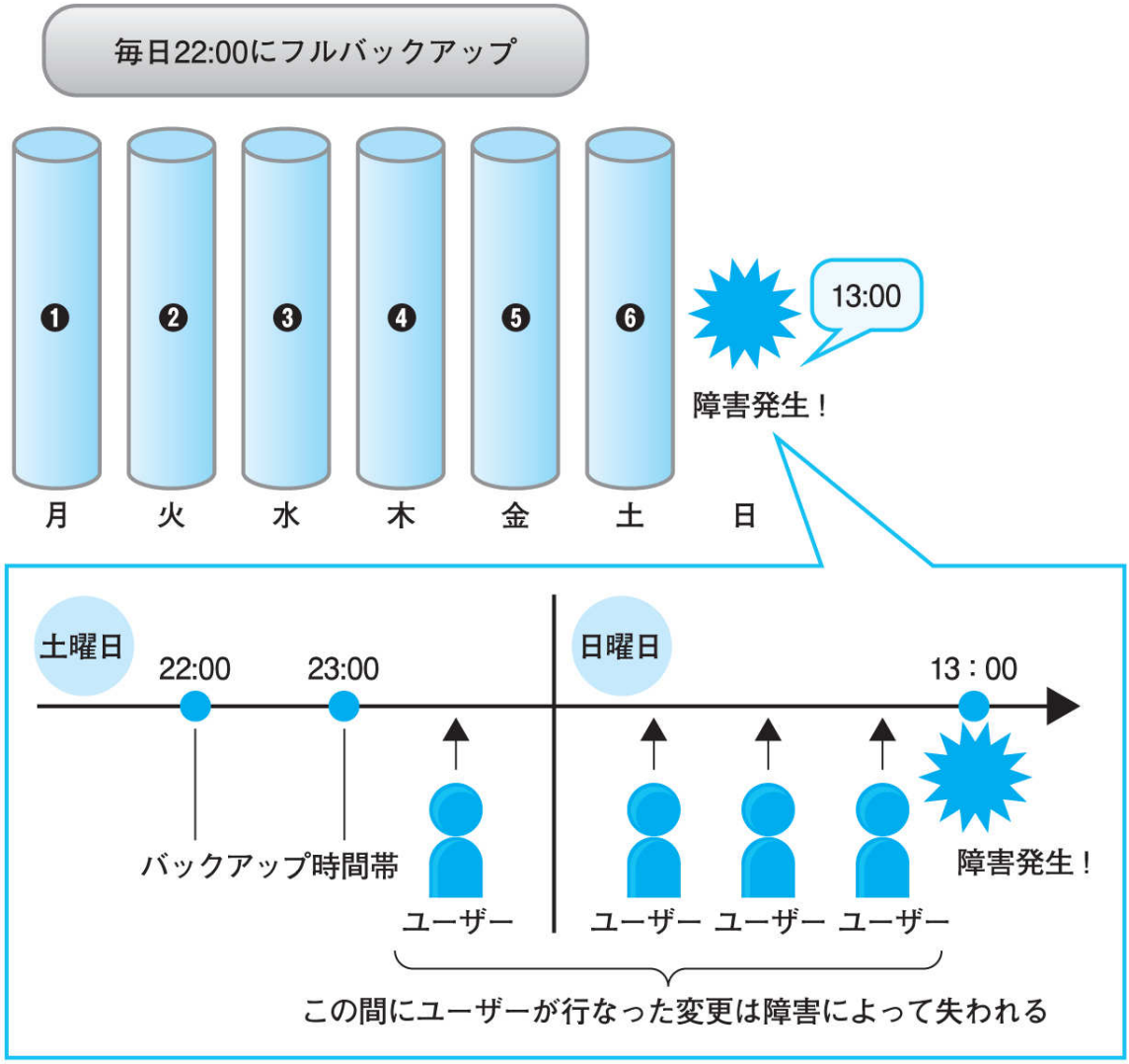


図2-18●フルバックアップ方式で障害が発生した場合

ここで、障害復旧の手順を厳密に二つに分ける必要があります。「バックアップファイルを戻す」作業を「**リストア**」[※17]、そのファイルに対して、トランザクションログを適用して変更分を反映する作業を「**リカバリ**」と呼びます（図2-19）。今まで、この両者を区別せず一緒にして「リカバリ」と呼んでいましたが、以降は分けて使います。

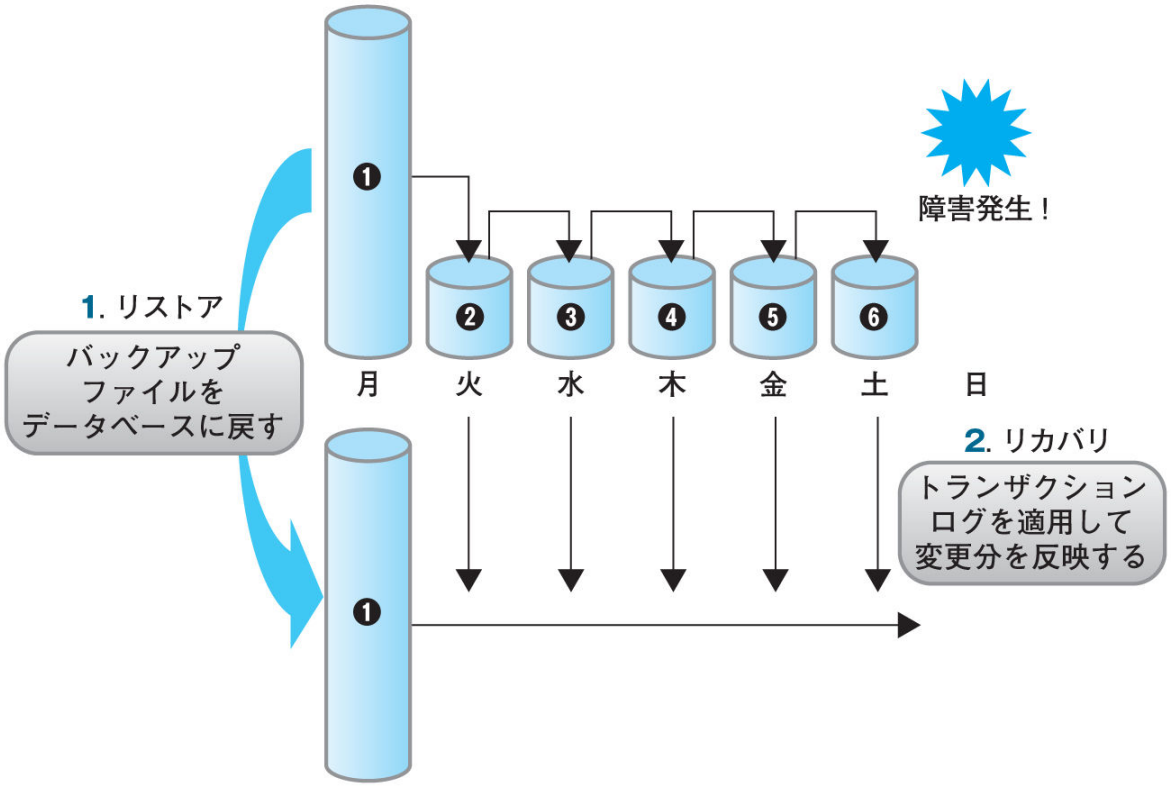


図2-19●リストアとリカバリ

リストアとロールフォワード

重要なのはこの意味での「リカバリ」です。バックアップしておいたトランザクションログを適用することもリカバリなのですが、実はトランザクションログは、DBMS 内部にも残っているのです。その中に何が含まれているかと言えば、最後のバックアップ後に実施されたユーザーからの変更分にほかなりません。まだバックアップされていないだけで、データベースサーバーにはファイルとして残っています［※18］。それゆえ、こ

の未バックアップのトランザクションログまで適用することで、初めてデータは障害直前の状態に復旧するのです。まとめると、リストアおよびリカバリの手順は以下の三段階となります。

1. フルバックアップのファイルをデータベースに戻す。➡リストア
2. 差分（または増分）バックアップしていたトランザクションログを適用する。➡リカバリ
3. データベースサーバーに残っているトランザクションログを適用する。➡ロールフォワード

これで、めでたく障害発生直前のデータが復旧されることになります。

演習問題

演習 2-1 データベースサーバーのクラスタリング構成

データベースの信頼性と性能を高めるための物理設計として、本章ではRAIDについて学びました。このほかにも、システム全体としての信頼性と性能を向上させる方法として「クラスタリング」という構成が存在します。一般的に、サーバーのクラスタリング構成にはどのような形態がありうるか、調べてください。

演習 2-2 ハードウェアリソースの情報取得

ハードウェアのサイジングを行なうには、現行システムのハードウェアリソースの情報が欠かせません。これを取得するためのOSコマンドを、OS別に調べて整理してください。

演習 2-3 サーバーCPUの机上サイジング

ハードウェア、特にサーバーCPUのサイジングについてケーススタディをしてみましょう。

みなさんは、今度行なわれるシステム更改（つまりシステムのリプレース）プロジェクトにおいて、サイジングを担当することになりました。

話を単純にするために、単純なハードウェアリプレースのみで、アプリケーションには改修が入らず、現行通りと仮定しましょう。変更されるのは、サーバーやストレージなどハードウェアのみです。また、現行システムのアクセスログやリソースログは取得することができるため、レスポンスとスループットは算出できます。

しかし、プロジェクトマネージャからは、次のように言われています。

> 「このシステム、今はお客さんからピーク時間帯になると遅いと苦情が来るんだ。今度はこんなことがないよう、適切なハードウェアを選んでくれよ」

プロジェクトの予算は少ないため、プロトタイプ検証を行なう余裕はありません。さて、みなさんはどんな手順でサーバーのCPUについて机上サイジングを実施すれば良いでしょうか？ なるべく具体的に、手順を考えてください。

日本語に訳すと「（ファイルの）再配置」です。

もちろんこのファイルが障害によって破損してしまった場合は、障害直前の状態にはリカバリ不能です。バックアップしていたトランザクションログを適用して終わりです。

第3章

論理設計と正規化
～なぜテーブルは分割する必要があるのか？

本章では、論理設計の中心である正規化について学びます。正規化が何を目的に、どのような理論に従って行なわれるかを理解することで、データベースに格納されるデータを整合的に保持するための方法論を身につけます。

学習のポイント

- リレーショナルデータベースにおけるテーブルとは、「同じ種類の物の集合」のことです。
- 「キー」とは、ある情報を引き出すための鍵です。特に、テーブルで最も重要なのが「主キー」です。
- 「正規化」は、データの冗長性をなくしていく作業です。その目的は更新時のデータ不整合を防止することにあります。
- 正規化を理解するためには、関数従属の概念を理解する必要がありますが、これの基礎になっているのが「キー」の概念です。
- 正規化を進めていくほどデータ整合性は高まりますが、検索性能が劣化します。通常は第3正規形までを考えれば十分です。

3-1 テーブルとは何か？

リレーショナルデータベースでは、あらゆるデータを「テーブル」という単位で扱います。テーブルは、見た目上は「表」と良く似ています。というより、英語のテーブルにはそのままずばり「表」という意味があります。しかし、テーブルと「表」は、厳密には同じものではありません。まずは、その違いから話を始めましょう。

二次元表≠テーブル

最初にちょっとしたクイズから入りましょう。別にひっかけなどはないので、感じたままに、10秒以内で答えてください。

問題

以下の二次元表はテーブルか？

項目1	項目2	項目3	項目4
千代大海	10 勝5 敗	大関	幕内
ドラえもん	ネコ型ロボット	四次元ポケット	スモールライト

レディ・ガガ	歌手	ファッション	
アル・カポネ		マフィア	禁酒法
あんぱんまん	バイキンマン	カレーパンマン	食パンマン
ピカソ		クレー	マティス

いかがでしょう。心の中で、答えは決まったでしょうか？

正解は、Noです。これは、リレーショナルデータベースにおけるテーブルではありません。確かに上記のようなデタラメなデータを含む表を、DBMSの中に作ることはできます。しかし、それはリレーショナルデータベースにおけるテーブルではないのです。

その理由は、これがテーブルに必要とされる要件を満たしていないからです。では、テーブルの要件とは何でしょう？ それを解説するために、まず本物のテーブルのサンプルを見て、違いを比較してみましょう。

■本物のテーブル（「社員」テーブル）

社員

社員ID	社員名	年齢	部署
000A	加藤	40	開発
000B	藤本	32	人事
001F	三島	50	営業

001D	斉藤	47	営業
009F	田島	25	開発
010A	渋谷	33	総務

先ほどの「偽テーブル」と本物のテーブルを比較すると、どこが同じで、どこが異なるでしょうか？

それは、偽テーブルが特に共通点のないレコードを寄せ集めているのに対し、上記の本物のテーブルは「社員」という共通点を持ったレコードの集合になっていることです。この「社員」テーブルは、一つのレコードが一人の社員を表現しています。偽テーブルの場合、各列は、第1列の人名に関係する何かの情報を意味しているようですが、それがレコード間でバラバラで、統一性がありません。

すなわち、テーブルとは、ただ**形だけ**二次元表であるだけではダメで、「ある共通点を持ったレコードを集めたもの」である必要があるのです。偽テーブルと本物のテーブルは、どちらもレイアウトは4列×6行で、構造的には同じ二次元表です。しかし、意味的にはまったく似ても似つかないものです。

勘どころ 18

テーブルとは、共通点を持ったレコードの集合である。

別の言い方をすれば、「テーブルとは同じ種類の物の集合である」とも言えます。このことを、アメリカの優れたDBエンジニアであるジョー・セルコは「**テーブル名はすべて複数形または複数名詞で書ける**」と表現しています。たとえば、先ほどのテーブルの例なら社員の集合ですから「Employees」、商品を集めたテーブルなら

「Items」、注文履歴であれば「Orders」など、すべて複数形で書ける、ということです。

勘どころ 19

テーブル名は英語ならば複数形／複数名詞で書ける。そうでなければそのテーブルにはどこか間違いがある。

テーブルというのは、形だけ二次元表を満たしていれば良い、というものではなく、それ自身が現実世界と結びついた意味を持っていなければなりません［※1］。これは、ともすると当たり前のことに思えるかもしれません。しかし、間違った設計というのは、往々にしてこの当たり前の基本を踏み外したときに生じるのです。その基本を踏み外したテーブルが、どのような問題を引き起こすか、という点については、第7章と第8章で見ることになります。ひとまず今は、テーブルが「物の集合」だという点だけ押さえておいてください。

二次元表とテーブルの定義上の違いは、実は他にもあります。詳しくはコラム「関係（リレーショナル）とは何か？」（81ページ）も参照。

それではテーブルについて詳しく見ていこうと思います。まずいくつか用語を整理しておきましょう。

行と列

通常の表と同様に、テーブルにおいても横と縦のデータの組を「行」と「列」と呼びます（図3-1）。あるいは「レコード」と「カラム」という呼び方もします。これは特に違和感のない呼び名だと思います。第1章でエンティティについて解説したときに、列を「属性」という呼び方もする、と言いましたが、実務ではほとんど使わない表現なので、本書でも基本的に「列」という呼び名を採用します。

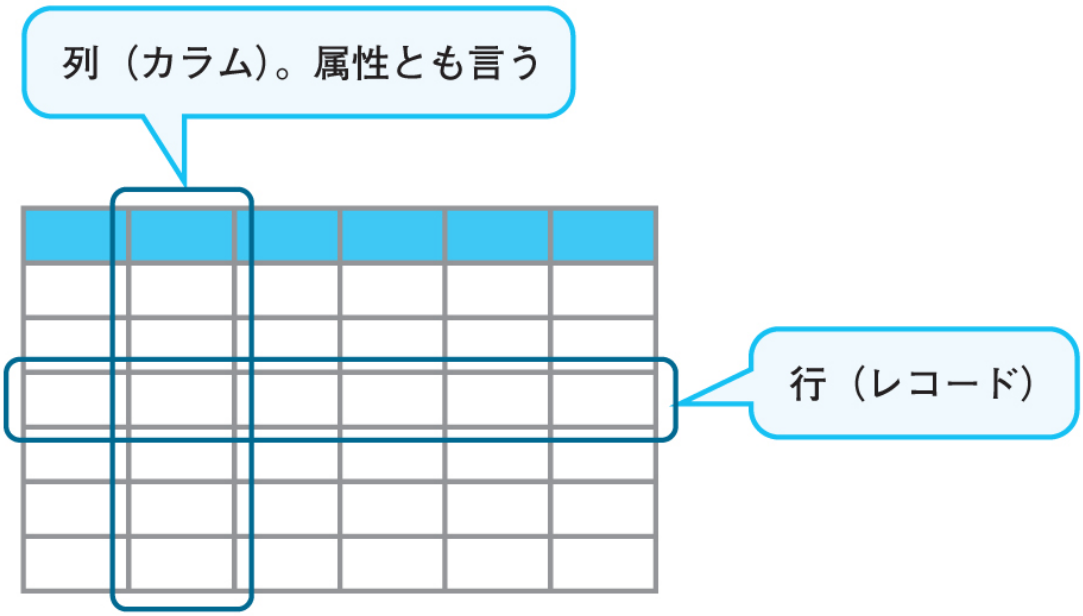


図3-1●テーブルの構成

キー

表になくて（も良くて）、リレーショナルデータベースのテーブルになくてはならないものがいくつかあります。そのうちの一つが「**キー**（key）」です。何の「鍵」かというと、ある特定のデータを引き出すための鍵です。特にリレーショナルデータベースでは、「あるレコード（1行とは限らない）を特定するための列の組み合わせ」という意味で使います。キーの種類にも複数ありますが、特に重要なものは、以下の二つです。

❶ 主キー

❷ 外部キー

❶ 主キー

「**主キー**」、または「**プライマリキー**（primary key）」と呼びます。主キーは、テーブルにおいて必ず一つ存在しなければならず、かつ一つしか存在しません。主キーとは、その値を指定すれば、必ず1行のレコードを特定できるような列の組み合わせのことです。この「1行に特定する」ことを「**一意に識別する**」という言い方をします。「一意」は「unique」の訳語です。

例として、先ほどの「社員」テーブルを見ましょう。このテーブルにおいて、値を決めれば必ず一つのレコードを特定できる列の組み合わせは何でしょうか？

社員

社員ID	社員名	年齢	部署
000A	加藤	40	開発
000B	藤本	32	人事
001F	三島	50	営業
001D	斉藤	47	営業
009F	田島	25	開発
010A	渋谷	33	総務

「年齢」と「部署」が候補から除外されることは、すぐにわかります。たとえば、「部署＝開発」という値を決めたとして、レコードは複数（加藤さん、田島さん）該当してしまうからです。年齢も、今このテーブルはたまたま重複する年齢のレコードがないだけで、**原理的**には複数行存在しえます。

「社員名」ならばレコードを一つだけ特定できそうな気がしますが、これも「年齢」と同様の理由で不適切なことがわかります。同姓同名の人間というのは、存在しうるからです（ある程度大きな規模の会社なら、実際に同姓同名の社員がいてもおかしくありません）。

残る候補は「社員ID」です。この値ならば、同姓同名の社員にも異なるIDを割り当てることで、常に一意にレコードを識別することができます。

主キーがテーブルに必ず存在しなければならない、というルールは、理論的に次のことも導きます。

勘どころ 20

テーブルには重複行は存在できない。

たとえば、社員を管理するテーブルには、同じ社員が二人登録されていてはいけませんし、それは商品を登録するテーブルの場合でも同じです。私たちが表計算ソフトなどで作る二次元表では、やろうと思えば重複データを持たせることも可能です。しかし、リレーショナルデータベースにおいては、どのような理由があっても重複するデータ、すなわち重複するレコードを持つことは許されません。たとえ入力データに重複が存在する場合でも、（新しいキーを追加するなど）重複しないよう工夫する必要があります［※2］。

■テーブルは重複レコードを持ってはならない

社員 ID	社員名	年齢	部署
000A	加藤	40	開発
000B	藤本	32	人事
001F	三島	50	営業
001D	斉藤	47	営業
009F	田島	25	開発
009F	田島	25	開発
010A	渋谷	33	総務

重複レコード

ここで一つ注意しておいていただきたいことがあります。主キーのない（＝重複行を許容する）テーブルは、作ろうと思えば多くのDBMSにおいても作ることが可能です。ただし、設計上、そのようなテーブルは許されない、ということです［※3］。

さて、「社員」テーブルのサンプルでは、たまたま「社員ID」一列だけで主キーとすることが可能でした。しかし場合によっては、複数列を組み合わせなければ主キーが作れないこともあります。こうした複数列を組み合わせて作るキーを**複合キー**と呼びます。

また、テーブルのサンプルなどを表記するとき、主キーとなる列の見出しに下線を引いて示す表記がよく使われます。本書でも以下、この表記を使用します。

■列名（社員ID）の下線は主キーであることを示す

社員ID	社員名	年齢	部署
000A	加藤	40	開発

000B	藤本	32	人事
001F	三島	50	営業
001D	斉藤	47	営業
009F	田島	25	開発
010A	渋谷	33	総務

なお、主キーに関連する概念として、「**候補キー**（candidate key）」と「**スーパーキー**（super key）」と呼ばれるキーもあります。候補キーとは、主キーとして利用可能なキーが複数存在した場合、それら「候補」となるキーのことを言います。したがって候補キーは定義上、複数存在しえます。主キーは一つのテーブルに一つしか設定できないため、候補キーのうちから一つを主キーとして選択します。スーパーキーは、主キーに、非キー列を付加した場合のキーの組み合わせです。たとえば、（A, B）という2列で主キーになる組み合わせがあった場合に、無関係の「C」という列も加えて（A, B, C）をキーとすることです。これでも十分に主キーの役割は果たしますが、Cは不要な列であるため、通常はこれを付加することはありません。

この候補キーおよびスーパーキーは、設計理論上は概念に名前が付けられていますが、実際の業務ではほとんど重要な役割は果たさないので、「そういうものもある」くらいに覚えておいていただければ大丈夫です。

❷ 外部キー

主キーの次に重要なキーが、「**外部キー**（foreign key）」です。これは、二つのテーブルの間の列同士で設定するものです。たとえば、次のような形です。

■外部キーは二つのテーブル間の列同士で設定する

社員

社員 ID	社員名	年齢	部署
000A	加藤	40	開発
000B	藤本	32	人事
001F	三島	50	営業
001D	斉藤	47	営業
009F	田島	25	開発
010A	渋谷	33	総務

部署

部署
開発
人事
営業
総務

この例では「社員」テーブルの「部署」列が「外部キー」です。この列は、部署の一覧を保持する「部署」テーブルの「部署」列を参照しています。この外部キーの役割は、「部署」テーブルに存在しないような部署のデータが、間違って「社員」テーブルに登録されないよう防止することです。いわば外部キーの役割は、「社員」テーブルに

対して一種の「**制約**（constraint）」を課すことなのです。この制約を「**参照整合性制約**」と呼びます。

　たとえば、この状態の「社員」テーブルに、次のようなレコードを新たに登録することはできません。

社員ID	社員名	年齢	部署
111Q	島田	22	広報

　この「島田」社員の所属部署は「広報」ですが、このような部署は「部署」テーブルに登録されていないからです。このレコードを登録しようとするSQL文はエラーになります。

　一方で、「部署」テーブルに次のようなレコードを登録することは問題なく可能です。

部署
研究

「部署」テーブルはいわば「社員」テーブルの「親」に当たる存在であるため、「子」の状態を気にすることなく変更することが可能です。これは人間の親子関係と同じです。子は親が存在しないと存在できませんが、子のいない親は存在しうる、ということです。

勘どころ 21

外部キーは人間の親子関係と同じ。

それでは、親である「部署」テーブルのレコードが変更されたり削除されたりした場合は、子の「社員」テーブルはどのような影響を受けるでしょうか？

これについては、「社員」テーブルを作成する際に、要件に応じて動作を選ぶことが可能です。たとえば、「部署」テーブルの「開発」レコードが削除された場合、「社員」テーブルの加藤さんや田島さんは、“親がいない子”になってしまいます。こうした“親がいない子”もあわせて削除するか、それとも削除SQL文をエラーにするかを選択可能です。このあわせて削除する動作を「**カスケード**」と呼びます。更新の場合も同じで、“親がいない子”のデータをあわせて変更するか、更新SQL をエラーにするかを選択可能です。

ただし、一番良いのはこうした厄介な問題を考えないようにテーブルを操作することです。つまり、常に子のテーブルを先に削除なり変更して、後から親のテーブルを更新していれば、こうした問題は起きないのです。

勘どころ 22

外部キーが設定されている場合、データの削除は子から順に操作するのが吉。

さて、これで主要なキーについては理解できましたが、ここで一つ注意しておきたいことがあります。「どのような列をキーとするか」という点です。

ここでは、「部署」列に外部キーを設定しましたが、これはあくまで説明のためのサンプルとしてこうしただけで、本当は良くない設計です。どこが良くないかというと、この「部署」列は、テーブルの物理定義において**可変長文字列**として定義されること

が多いからです。こうした何らかの「名前」というのは、何文字で構成されるかわかりません。そのため、テーブル作成する際においても、上限値だけが決まっていて、その上限以内であれば何文字でもかまわない可変長文字列というデータ型で定義します。

このような可変長文字列をキーに使うことの危険は、こうした文字列は、同じ名前であっても微妙に異なる表記になることがあるため、ヒットするはずの文字列でキーがヒットしなかったり、キーが重複しているように見えてしまったりすることがあるからです。たとえば、社員名を例に考えれば、「山田太郎」のように姓と名の間に空白を含まない表記もあれば、「山田太郎」のように空白を挟む表記もあります。この両者は、ともに同じ人物を表わしているにもかかわらず、データベース上では異なる値として扱われます。そうすると、たとえば外部キーで結ばれている二つのテーブルのうち、片方が「山田太郎」で、もう一方が「山田太郎」だった場合に、両者は不一致とされてしまうのです。

こうした理由から、キーに使用する列は、必ず何らかのコードやIDといった表記体系のきっちり定まったデータを、固定長文字列のデータ型に格納して使います。これは、リレーショナルデータベースにおけるテーブル設計の鉄則です。

勘どころ 23

キーとなる列には、コードやIDなど表記体系の定まった固定長文字列を用いる。

制約

外部キーの説明において、「参照整合性制約」について触れましたが、テーブルには、他にもいくつかの種類の制約を付けることができます［※4］。代表的なものは以下の三つです。

❶ NOT NULL制約
❷ 一意制約
❸ CHECK制約

❶ NOT NULL制約

データベースにデータを登録するとき、その値がわからないことがあります。現時点でまだ不明だからという場合もあれば、原理的にそこにデータの値が定まらない場合もあります。こういう場合、リレーショナルデータベースはそのデータをNULLという扱いにすることができます。つまり「空欄」にすることができるのです（NULL自身は、データの値ではありません）。

■NULLの例

社員ID	社員名	年齢	部署
000A	加藤	40	開発
000B	藤本		人事
001F	三島	50	営業
001D	斉藤	47	営業
009F	田島	25	開発
010A	渋谷	33	総務

藤本さんの年齢が不明なのでNULLに。

しかし、NULLというのはSQL上で扱うにはいろいろな問題を引き起こす厄介なものです。したがって、可能な限りデータはNULLにしない、というのがデータベース設計

における大方針です。そこで、「この列は絶対にNULLにはならない」ということがわかっている列については、NULLを禁止することができます。

この制約は、列単位で設定することができます。NOT NULL制約が設定された列にNULLのデータを登録しようとしたり、NULLに更新しようとしたりした場合、そのSQL文はエラーとなります。

勘どころ 24

テーブル定義において、列には可能な限りNOT NULL制約を付加する。

ちなみに、主キーとなる列には、DBMS側で暗黙にNOT NULL制約が付加されます。主キーは重複が許されないため、NULLも当然許されないわけです。

❷ 一意制約

一意制約は、ある列の組について一意性を求める制約です。その点で、主キーと似ていますが、主キーがテーブルにつき一つしか設定できないのに対し、一意制約は何個でも設定できます。

❸ CHECK制約

ある列の取りうる値の範囲を制限するための制約です。たとえば、「年齢」列についてなら、「20 ～ 65までの整数」や、「部署」列ならば「'開発'、'人事'、'営業'のいずれかの文字列」といった具合です。CHECK制約も一つのテーブルにつき何個でも設定できますが、複数列にまたがった制約は、現在のところ設定できません（たとえばA列B列について「A > B」のような関係は設定できません）。

テーブルと列の名前

テーブルについて最後に解説するポイントは、名前についてです。まず、テーブルとテーブルの物理定義における列の名前に関するルールから話をしましょう。

ルール1 名前に使える文字集合

（a）半角のアルファベット

（b）半角の数字

（c）アンダーバー（_）

これ以外の文字、たとえば日本語や$、#といった特殊文字を名前として使うことを許しているDBMSもありますが、それは標準SQLで定められたルールを破っている独自拡張です。したがって、DBMSを変えたときにも使えるという保証がありません。

アンダーバーの変わりにハイフン（-）を使いたいと思う方もいるでしょうが、これも標準SQLで認められていないため、アンダーバーを使うようにしましょう。

なお、本書ではテーブルの名前や列名を日本語で表記していますが、これはあくまで論理設計について語るうえで、日本語のほうがわかりやすいための措置です。**テーブルの物理的な定義において日本語を使って良いわけではありません**ので、注意してください。

勘どころ 25

テーブルや列の名前に日本語はご法度。

ルール2 最初はアルファベット

2009_uriageや_namaeのように、名前の先頭をアルファベット以外で始めてはいけません。uriage_2009のように先頭はアルファベットで始めてください。

ルール3 名前は重複してはならない

同じ名前を持つテーブル、同じ名前を持つ列は存在してはいけません（もし作ろうとしても、エラーになります）。この場合、問題なのはその範囲です。列の場合は、同じテーブルの中で同じ名前を二つの列に使うことはできません。これはわかりやすいと思います。

一方、テーブルの場合は、DBMSが設定する範囲内においては、同じ名前を使うことができません。この範囲を**ドメイン**と呼びます。DBMSによっては**スキーマ**と呼ぶこともあります。逆に言うと、この範囲をまたがっていれば、同じ名前のテーブルを作ることも可能です。

さて、これでテーブルについての基礎知識を一通りカバーできました。それでは、次節からいよいよ論理設計の中核である正規化について学んでいきましょう。

関係（リレーショナル）とは何か？ COLUMN

「どうして関係モデルと呼ぶのですか」という質問がときどきある。どうして表形式（tabular）モデルと呼ばないのか、理由は２つある。（1）関係モデルを考えたころ、データ処理に携わる人たちの間では、複数の対象の間の関係（あるいは関連）は、つなぎデータ構造で表現されなければならないと考える傾向があった。この誤解を避け討つために、関係モデルという名前を選ん

> る傾向があった。この誤解を避け許すために、関係モデルという名前を選んだ。（2）関係よりも表の方が、抽象水準が低い。表は、配列と同様に位置による呼出しが可能だという印象を与えるが、n 項関係ではそうではない。また表の情報内容が行の順番と無関係であるという点についても、表は誤解を招きやすい。しかし、こうした小さな欠点はあるにしても、関係の概念を表現するもっとも重要な手段は、依然として表である。表といえば、誰にでもわかる。
>
> ——エドガー・F・コッド［※5］

3-1節で、

テーブルは、見た目上は「表」とよく似ている

と書きました。開発の現場で実務を遂行する分には、おおよそ、そういう理解でかまわないのですが、しかし、だったら、最初から「表データベース」とか「表モデル」という名前にしておけば混乱も少ないだろうに、なんで「リレーショナルデータベース」とか「関係モデル」なんていうわかりにくい名前をつけてしまったのでしょうか？

この疑問は、気にならない人にとっては「そういうものだ」と受け流してまったく気にしない反面、気になる人にとっては夜も眠れなくなってしまうほど考えてしまうものです（ちょっとオーバーかもしれませんが）。そこでこのコラムでは、なぜ「リレーショナルデータベース」が「表データベース」というもっとキャッチーな（？）名前で呼ばれないのか、という問いに答えたいと思います。

このコラムの冒頭に、エドガー・F・コッド（リレーショナルデータベースの理論を作った偉い人です）の言葉を引用しましたが、彼もこの質問を何度も受けていたようです。「なんでまた"リレーショナルデータベース"なんていうわかりにくい名前をつけたんですか？」と。

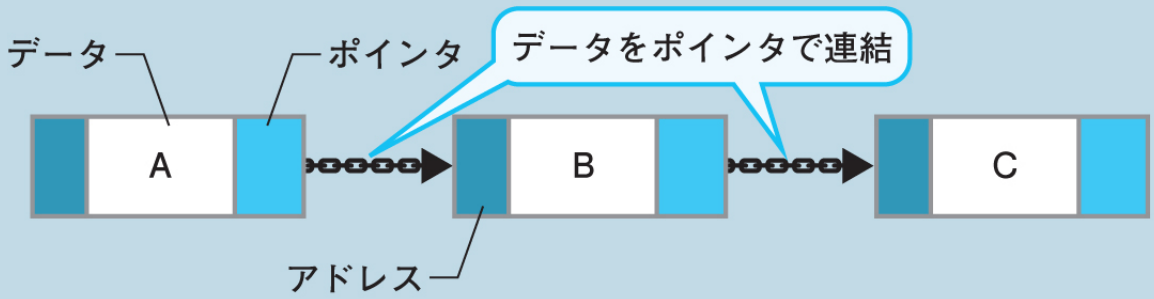
コッドは、こういう素朴な質問もおろそかにせず丁寧に答えてくれています。彼

は以下の二つの理由を挙げています。

理由1 当時のエンジニアは、「データの関連」を表現する手段といえばポインタしか頭になかった。

理由2 「関係」と「表」は厳密には異なるものである。

理由1について、コッドは「つなぎデータ構造」と言っていますが、これはC 言語で言うところの「ポインタチェイン」のことです。もっとも、最近では若い人たちはC言語を知らない層も増えているので、図を使って説明しましょう（図A）。「ポインタ」とは、あるデータの物理的な格納場所（アドレス、番地）を示す情報です。あるデータとデータ（たとえば名前と生年月日）を関連づけるためには、二つのデータを、アドレスを指定することで結びつける必要があります。このようにして、データを次々と数珠つなぎにしていった構造が、「ポインタチェイン（ポインタの鎖）」です。



図A●ポインタチェインによるデータ連結構造

これは、リレーショナルデータベースが登場する以前の階層型データベースなどで採用されていた方式ですし、実はリレーショナルデータベースも、ユーザーにこういう見せ方をしていないだけで、内部では今でもデータを同じやり方で管理してい

見せ方をしていないだけで、内部では今でもデータを同じやり方で管理しています。

この方法の欠点は、実装が複雑で面倒になることです。実際、ポインタというのは、C言語を学習する際に多くの人にとって躓(つまづ)きの石となる箇所として有名です。そのため、最近の言語（Javaなど）は、ユーザーにポインタが見えないよう配慮しています。

コッドは、データベースにおいても、ユーザーがデータとデータの関連を扱うに当たり、いちいちポインタやアドレスという物理的な概念まで考慮しなければならないのは、不便だと考えて、データベースからアドレスとポインタを追放しました。「そんなものに頼らなくても、データの関連を表現することはできる」というメッセージが、「リレーショナルデータベース」という名前には込められています。

続いて、**理由2** は、もっと直接明快です。ずばり、「関係」と「表」は、決して同じものではないのです。たとえば、3-3節以降で「正規形」について学びますが、そこで最も基本的な正規形（第1正規形）では、

テーブルのセル（一つのマス目）の中に複数の値が入ってはならない

というルールがあります。「表」であれば、セルの中に複数の値を入れ込むのも、入れ子の表を作るのも自由です。「関係」と「表」の区別は、たとえば他にも以下のようなものがあります。

- **関係には重複するレコードは存在してはならないが、表には存在しても良い。**
- **関係のレコードは上下の順序を持たないが、表の行は上から下へ順序づけられている。**

- **関係の列は左右の順序を持たないが、表の列は左から右へ順序づけら**

- **関係の列は左右の順序を持たないが、表の列は左から右へ順序づけられている。**

こうした違いがあることから、コッドは「関係」と「表」を厳密には区別する必要を感じていました。もっとも、最後に彼も認めているように、「関係って何？」と聞かれたときに最も近いイメージを持てる答えが「表」である、というのも間違いのないところです。

こうした汚れたデータの「洗浄」については、第8章（データクレンジング）で取り上げます。

なぜ重複行を許可するテーブルが許されないのか、という理由は複数あるのですが、最も重大な理由としては、SQLでデータを利用する際の不都合が挙げられます。詳細はコラム「主キーはなぜ必要か？」（239ページ）を参照。

主キーも制約の一種なのですが、付与して当然であるため、あまり制約とは意識しません。

E.F.コッド「関係データベース：生産性向上のための実用的基礎」『ACMチューリング賞講演集』（共立出版、1989）pp.457-8

3-3 正規化とは何か？

本節からは、具体的なテーブルを使いながら、リレーショナルデータベースにおける論理設計の技法について学んでいきましょう。論理設計とは、前章でも述べたように、データの形式、すなわちテーブルのレイアウトを決める設計であると言えます。論理設計において、特に理解を深めておくべき概念が、「**正規化**（normalization）」およびそれによって作られる「**正規形**（normal form）」です［※6］。聞きなれない言葉かもしれませんが、リレーショナルデータベースの論理設計をマスターするための重要概念ですので、本章でしっかり理解しましょう。

「正規化」といういかめしい語感から、難しい印象を持つかもしれませんが、実際には、慣れてしまえばある程度機械的に正規化の作業を行なえるようになります。むしろ、実務での論理設計において本当に頭を悩ませることになるのは、一通りの正規化が「終わった後」なのだ、ということが、後続の章を読むとわかってきます。

ともあれ、まずは第一の関門である正規化をクリアすることにしましょう。

正規形の定義

最初に、正規形について概略を説明します。正規形とは、一言で言うと、データベースで保持するデータの冗長性を排除し、一貫性と効率性を保持するためのデータ形式です。

データベース設計の知識がなく、行き当たりばったりでテーブルを作って、データを格納していくと、いろいろと困ることが起きます。たとえば、一つの情報が複数のテーブルに存在して無駄なデータ領域と面倒な更新処理を発生させてしまう、ということがあります（**冗長性**）。また、こうした冗長なデータ保持をしていると、更新処理のタイムラグによってデータの不整合が発生したり、そもそもデータを登録することができないようなテーブルを作ってしまうことがあります（**非一貫性**）。

こうした冗長性や非一貫性の問題を解決するために考案された方法論が、正規化です。正規化には何段階かレベルがあって、一般的には第1 ～第5正規形がよく知られています。これは数字が増えるほど正規化のレベル（次数）が上がっていくことを意味しています。

正規化の中でも、通常の業務で使用するレベルとして、第3正規形までを考えることが多いため、本書でも第3正規形までを中心に解説を行ないます。第4以降の正規形についても取り上げていますが、実際にみなさんが仕事で正規化を行なう際には、第3正規形まで達成したら、その時点で以降の正規形の条件も満たしていることがほとんどです。そのため、いきなり全部は覚えきれないと思ったら、最初は第3正規形までを理解していただけば十分です［※7］。

勘どころ 26

正規形のレベルは第5まであるが、普通は第3正規形まで理解すれば十分。

正規化という言葉自体は、リレーショナルデータベース以外の分野でも使われ、一般的には「あるルールに従った形に整理する」という意味を持ちます。リレーショナルデータベースにおいても、この意味に従っています。

この後ですぐに説明しますが、第1正規形はすごく簡単なので、実質的に勉強する必要があるのは、第2、第3正規形です。

それでは、正規形についての説明を始めましょう。まずは最もレベルの低い第1正規形からですが、実は、リレーショナルデータベースのテーブルは、いずれもすでに第1正規形を満たす形になっています。拍子抜けするかもしれませんが、これは第1正規形の定義を見ればわかります。

第１正規形の定義～スカラ値の原則

ここで、便宜的にテーブルの行と列が交差する特定の一マスのことを、Excelなどスプレッドシートの用語にならって「セル」と呼ぶことにしましょう［※8］。すると、第1正規形の定義とは「**一つのセルの中には一つの値しか含まない**」というものです。

■第１正規形の例

社員

会社コード	会社名	社員 ID	社員名	年齢	部署コード	部署名
C0001	A 商事	000A	加藤	40	D01	開発
C0001	A 商事	000B	藤本	32	D02	人事
C0001	A 商事	001F	三島	50	D03	営業
C0002	B 化学	000A	斉藤	47	D03	営業
C0002	B 化学	009F	田島	25	D01	開発
C0002	B 化学	010A	渋谷	33	D04	総務

このように、一つのセルに一つだけの値が含まれているとき、この値のことを「**スカラ値（scalar value）」と呼びます。scalarは「単一の」という意味の英単語です。**

第１正規形を作ろう

まず第1正規形の定義を確認しましたが、「そんなの当たり前のことじゃないか」と思うかもしれません。しかし、現実に私たちが日常的に使う二次元表では、こうやってきれいに「一つのセルに一つの値」という原則が守られているわけではありません。

たとえば、こんな表を見たことがないでしょうか？

■非正規形（正規化が行なわれていないテーブル）

扶養者

社員ID	社員名	子

000A	加藤	達夫 信二
000B	藤本	
001F	三島	敦 陽子 清美

一人の社員は、複数の子を養っていることがあります。そうすると、当然、一人の社員につき複数の子が対応することになるわけですが、それを上記の表のように一つのセルに複数の値を入れ込む形で表現することは、私たちも普通にやることです。

しかし、これはリレーショナルデータベースでは重大な規則違反と見なされます。上記のような表は、リレーショナルデータベースの世界では、次のどちらかの形に置き換えることで、第1正規形に変換してやる必要があります。

■第1 正規形 その1

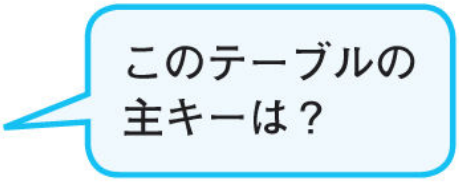
扶養者

社員ID	社員名	子1	子2	子3
000A	加藤	達夫	信二	
000B	藤本			
001F	三島	敦	陽子	清美

■第１正規形 その２

扶養者

社員 ID	社員名	子
000A	加藤	達夫
000A	加藤	信二
000B	藤本	
001F	三島	敦
001F	三島	陽子
001F	三島	清美



その1 は、子の数だけ列を増やしてやる方法、これに対し **その2** は、子の数だけ行を増やしてやる方法です。どちらの方法でも第1正規形は達成できます［※9］。

どちらでも第1正規形になるのなら、どっちを使っても良いのか、というと、そうではありません。様々な理由から、基本的には **その2** （行持ち）を採用することが望ましい場合が多いのです。詳しくは第7章で説明するので、今は頭の隅にとめておいてもらえればかまいません。

第1正規形 その2 の問題と解決方法

ただし、**その2** のテーブルについても、このままだと二つの問題があります。まず一つ目は、このテーブルには**主キーを決められない**ことです。

子を行持ちの形で展開したこのテーブルでは、1レコードの列をすべて特定しようとすると、{社員ID, 社員名, 子}の3列（要するに全部の列）を指定せざるをえませ

ん。{社員ID, 社員名}だけだと、「社員ID＝000A」「社員名＝加藤」のような場合に、2レコードが該当してしまうからです。

しかし、問題はすべての社員が子を持っているわけではない、ということです。社員ID＝000Bの藤本さんは、子を持っていません。そのため、「子」列はNULL になります。ところが、主キーというのは定義上、その一部の列であっても、NULLにすることが許されません。ここで矛盾が生じてしまうわけです。

勘どころ 27

（主）キーは一部であってもNULLを含んではならない。

この問題を解決する方法として、子を持っていない社員の「子」列には、特定の文字列、たとえば'子なし'を入れる、という業務ルールを作る方法もあります。これならば、テーブル構成を変更せずとも主キーを決められます。

しかし、この方法では、二つ目の問題を解決することができません。それは、このテーブルが意味的に「社員」と「扶養者」という二つのエンティティの情報を含んでしまっており、**テーブルの意味やレコードの単位をすぐに理解できない**ことです。

上記二つの問題をともに解決するには、次のようにテーブルを分ける必要があります。

■第1 正規形 その3 （テーブル分割）

社員

社員ID	社員名
000A	加藤

000B	藤本
001F	三島

扶養者

社員ID	子
000A	達夫
000A	信二
001F	敦
001F	陽子
001F	清美

「扶養者」テーブルについては子を持っている社員のレコードのみ限定して保持することで、主キーがNULLになる事態を防止できます。また、それだけでは子を持たない社員である藤本さん（000B）の情報が欠落しますが、これについては、別に「社員」テーブルで管理するようにすれば、子の有無にかかわらず、社員全員の情報を保持することが可能になります。また、社員の名前と子の名前を結びつけたいときも、「社員」テーブルと「扶養者」テーブルとを社員IDをキーに結合すれば、難なく得ることが可能です。

SQLのイメージは次のようなものです。

■「社員」テーブルと「扶養者」テーブルを内部結合する

```
SELECT 社員.社員ID,
       社員.社員名,
       扶養者.子
  FROM 社員 INNER JOIN 扶養者
    ON 社員.社員ID = 扶養者.社員ID;
```

結果

```
社員ID   社員名   子
-----   ----   ---
000A    加藤     達夫
000A    加藤     信二
001F    三島     敦
001F    三島     陽子
001F    三島     清美
```

「社員」テーブルと「扶養者」テーブルを内部結合し、その結合キーとして社員IDを使う、というごく平凡なSELECT文です。

これだと、子がいない藤本さんが結果に現われませんが、もし藤本さんを含めたければ、内部結合を外部結合に変えた以下のSELECT文を使用すれば大丈夫です。

■「社員」テーブルと「扶養者」テーブルを外部結合する

```
SELECT 社員.社員ID,
       社員.社員名,
       扶養者.子
  FROM 社員 LEFT OUTER JOIN 扶養者
    ON 社員.社員ID = 扶養者.社員ID;
```

結果

```
社員ID   社員名   子
-----   ----   ---
000A    加藤     達夫
000A    加藤     信二
000B    藤本
001F    三島     敦
001F    三島     陽子
001F    三島     清美
```

なぜ一つのセルに複数の値を入れてはダメなのか？～関数従属性

第1正規形の定義についてはこれで理解いただけたと思いますが、ところでよくよく考えてみると、なぜ一つのセルに複数の値が含まれていてはいけないのでしょうか？ そういう表は日常、私たちは頻繁に使うものなのに、なぜリレーショナルデータベースではスカラ値のテーブルしか認められないのでしょうか？

面と向かってこう質問されると、経験を積んだデータベースエンジニアでも意外に返答に窮することがあります。この理由は、正規形の意味に深く関わっているので、ぜひ理解していただきたいものです。

なぜ一つのセルに複数の値を入れることがリレーショナルデータベースでは認められないか、その理由を端的に言うと、セルに複数の値を許せば、主キーが各列の値を一意に決定できないからです。これは主キーの定義に反してしまいます。一方、スカラ値だけのテーブルならば、主キーは、各列を一意に決めることができます。したがって、リレーショナルデータベースは、必ず第1正規形（スカラ値だけから構成されるテーブル）を満たすテーブルだけから作られなければならないのです。

実はこれは、正規形全体を理解するための鍵になる概念と結びついています。それを、**関数従属性**（functional dependency）と呼びます。

関数とは、私たちが学校で習ったあのY = f(X)という関数です。この関数は、「入力（X）に対して出力（Y）を一つに決めるための箱」です。どんな関数でも、X＝5と定めれば、Y＝10のように、Xに対してYが一つに決まります。これを、「**YはXに従属する**」と表現します。リレーショナルデータベースでは、このXとYの関数を次のように表現します。

{X} → { Y}

これは、「X列の値を決めれば、Y列の値が一つに決まる」という、数学の関数と同じことを意味しています（XおよびYは一つの列ではなく、複数の列の組み合わせであってもかまいません）。

実は正規化とは、テーブルのすべての列が、関数従属性を満たすように整理していくことなのです。実際、第1正規形以前の非正規形「扶養者」テーブル（87ページ）を考えると、

{社員ID} → {社員名}

という関数従属は成立しています。しかし、

{社員ID} → {子}

という関数従属は不成立です。社員ID「000A」に対して、子は「達夫、信二」という二人が該当してしまうからです。

一方、正規化した「扶養者」テーブル（87ページ）では、以下のような関数従属が成立しています。

{社員ID} → {社員名}
{社員ID} → {子1}
{社員ID} → {子2}
{社員ID} → {子3}

また同様に、正規化した「社員」テーブルおよび「扶養者」テーブル（89ページ）についても、関数従属が成立します。

「扶養者」テーブルの場合、{社員ID, 子}という主キー列しか存在しませんが、主キーは当然、主キーに従属するのです。

リレーショナルデータベースでは、この「セル」を指すための正式な用語は特にありません。あくまで本書では便宜的にこの名前を使います。

時々、「その1 （列持ち）の形式は第1正規形ではない」と書いているテキストがありますが、これは間違いです。「一つのセルに一つの値」というスカラ値の原則を満たしているのですから、これも立派な第1正規形です。

さて、それでは第2正規形へ話を進めましょう。3-4節で第1正規形の例として取り上げた「社員」テーブルをサンプルに使います。もう一度、登場させましょう。

■第1 正規形（第2 正規形ではないテーブル）

社員

会社コード	会社名	社員ID	社員名	年齢	部署コード	部署名
C0001	A 商事	000A	加藤	40	D01	開発
C0001	A 商事	000B	藤本	32	D02	人事
C0001	A 商事	001F	三島	50	D03	営業
C0002	B 化学	000A	斉藤	47	D03	営業
C0002	B 化学	009F	田島	25	D01	開発
C0002	B 化学	010A	渋谷	33	D04	総務

まず、このテーブルが第1正規形を満たしていることを確認しておきましょう。このテーブルは、いずれのセルもスカラ値から構成されているので、問題なく第1 正規形であると言えます。

しかし、このテーブルはまだ、第2正規形ではありません。その理由は、このテーブルにおける関数従属が「完全」ではないからです。このテーブルの主キーは{会社コード, 社員ID }です。したがって、他のすべての列はこのキーに従属するのですが、よく見ると、「会社名」列だけは、主キーの一部である「会社コード」に従属しています。つまり、以下のような関数従属性があるのです。

{会社コード} → {会社名}

このように、主キーの一部の列に対して従属する列がある場合、この関係を**部分関数従属**と呼びます。これに対して、主キーを構成するすべての列に従属性がある場合を、**完全関数従属**と呼びます。第2正規形とは、テーブル内で部分関数従属を解消し、完全関数従属のみのテーブルを作ることなのです。

勘どころ 28

第2正規形は、部分関数従属を解消することで得られる。

第２正規化を行なう

部分関数従属を解消する手段も決まっており、その方法はやはりテーブルの分割です［※10］。

第2正規形にするには、次のように部分関数従属の関係にあるキー列と従属列だけ独立のテーブルにすれば良いのです。平たく言えば、元のテーブルで邪魔だった「会社名」という列を外に追放した、ということです。

■第2 正規形

社員

会社コード	社員ID	社員名	年齢	部署コード	部署名
C0001	000A	加藤	40	D01	開発
C0001	000B	藤本	32	D02	人事
C0001	001F	三島	50	D03	営業
C0002	000A	斉藤	47	D03	営業
C0002	009F	田島	25	D01	開発
C0002	010A	渋谷	33	D04	総務

会社

会社コード	会社名
C0001	A 商事
C0002	B 化学

これによって、「社員」テーブルも、新しく作った「会社」テーブルも、すべての列が主キーに完全関数従属することになりました。「社員」テーブルは、もともと「会社名」列以外は完全関数従属していましたし、「会社」テーブルのほうも、もともと完全関数従属の関係にあった列の組み合わせでテーブルを作っているわけですから、こちらも完全関数従属するのは当然です。非常にシンプルな解決と言えます。

第２正規形でないと何が悪いのか？

これで第2正規形を作るための手順についての解説は終わりです。ところで、第2正規形というのは、何のために作るのでしょうか？ 第2正規形にすることの**メリット**は何なのでしょう？ 肝心のこの点を理解しないまま、機械的に正規化の手順だけ暗記しているエンジニアもたまにいるのですが、これは本末転倒な態度です。ここで、第2正規形のメリットをきちんと理解しておきましょう。

正規形のメリットは、正規化する前と、した後のテーブルを比較してみると、はっきりします。

まず、正規化する前の「社員」テーブル（93ページ）から考えましょう。このテーブルを使って実際の業務を運用しようとすると、困ったことが起こります。社員の情報が不明の会社（C建設）があった場合、この会社をテーブルに登録することができません。主キーの一部に社員IDが含まれているので、これが不明（NULL）の状態ではレコードを登録できないのです。社員IDに一時的なダミー値を入れる、という逃げの手もあり、実際にこういう運用をしているシステムも世の中にはあるのです

が、根本的な解決にはなっていません。みなさんも、いきなりこういう非正統的なやり方を覚えないでください。

　また、このテーブルにはもう一つの欠点があります。それは、運用を誤ると、会社コードと会社名の対応がレコードによってマチマチになってしまう危険があることです。たとえば、{ C0001, **A商事** }というレコードの他に、{C0001, **A商社** }という間違ったデータが登録されてしまう危険があります。今、この間違いを防止するための方法はありません。データベースではなく、アプリケーション側でチェックロジックを実装しない限り、この間違いには誰も気付きませんし、そもそもアプリケーションを介さずに直接SQLでテーブルを更新されたらお手上げです。

　一方、第2正規形にした後の「社員」テーブルと「会社」テーブルを考えると、「会社」テーブルにC建設のレコードを登録することで、この問題を簡単に解決することが可能になります。

■第2 正規形のメリット

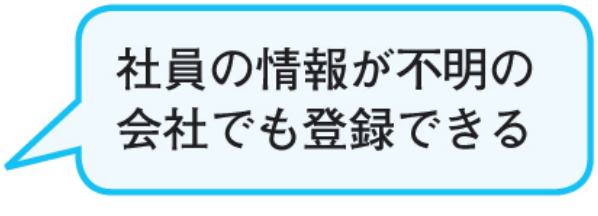
社員

会社コード	社員ID	社員名	年齢	部署コード	部署名
C0001	000A	加藤	40	D01	開発
C0001	000B	藤本	32	D02	人事
C0001	001F	三島	50	D03	営業
C0002	000A	斉藤	47	D03	営業
C0002	009F	田島	25	D01	開発

C0002	010A	渋谷	33	D04	総務

会社

会社コード	会社名
C0001	A 商事
C0002	B 化学
C0003	C 建設



社員の情報が不明であっても、「会社」テーブルに会社の情報を登録することには何の不都合もありません。また、会社コードと会社名の対応は1レコードだけで示されるため、データ誤登録の危険も格段に減ります。冗長性を解消したことによるメリットです。

このような観点から見ると、第2正規化というのは、言ってみれば「会社」と「社員」という、それぞれに**異なるレベルの実体（エンティティ）を、きちんとテーブルとしても分離してやる**作業だ、という見方もできます。現実の世界では、会社があってその下に社員が存在しているわけですから、その会社という実体と社員という実体を、リレーショナルデータベースの世界でも異なるテーブルとして表現する、というのは自然な発想だとも言えます。

勘どころ 29

正規化とは、現実世界の実体間にある階層の差を反映する手段でもある。

無損失分解と情報の保存

さて、ここまで正規形の解説を読んで、もしかすると次のような疑問を持った方もいるかもしれません。

> 「テーブルを分割することで更新異常を防ぐなど、正規化のメリットは理解できた。でも、正規化でテーブル分割することによる不都合はないのだろうか？ つまり、**分割前のテーブルに戻せなくなってしまう**ことはないのだろうか？」

この疑問はもっともなものです。しかし、心配はありません。第2正規化は、必ず正規化する前の状態にテーブルを戻すことができるようになっています。こういう操作を「可逆的（reversible）」と呼びますが、第2正規化は**可逆的**な操作なのです（図3-2）。第2正規化に可逆性があるのは、正規化によって失われる情報がないからです。このように情報を完全に保存したままテーブルを分割する操作のことを、**無損失分解**と呼びます。

図3-2●正規化とは元に戻すことができる可逆的操作

では、正規化されたテーブルから、非正規化状態のテーブルへどうやって戻すのでしょうか？ その方法が、結合です。第2正規形の「社員」テーブルと「会社」テーブル（96ページ）に対して、次のような結合を使ったSELECT文を実行することで、非正規化状態の「社員」テーブル（第1正規形）を作ることが可能です。

■第１正規形に戻す（「社員」テーブルと「会社」テーブルを内部結合する）

```
SELECT 社員.会社コード,
       会社.会社名,
       社員.社員ID,
       社員.社員名,
       社員.年齢,
       社員.部署コード,
       社員.部署名
  FROM 社員 INNER JOIN 会社
    ON 社員.会社コード = 会社.会社コード;
```

結果 第２正規化を行なう前の「社員」テーブル（第１正規形）

会社コード	会社名	社員ID	社員名	年齢	部署コード	部署名
C0001	A商事	000A	加藤	40	D01	開発
C0001	A商事	000B	藤本	32	D02	人事
C0001	A商事	001F	三島	50	D03	営業
C0002	B化学	000A	斉藤	47	D03	営業
C0002	B化学	009F	田島	25	D01	開発
C0002	B化学	010A	渋谷	33	D04	総務

「社員」テーブルと「会社」テーブルを内部結合し、結合キーとして会社コードを使っています。正規形から非正規形へ戻す操作は、常に結合SQLです。

勘どころ 30

正規化の逆操作は結合。

実は、第5正規形に至るまでの正規化はすべてテーブル分割です。このため、正規化をテーブル分割と同義だと思い込んでいる人もいるぐらいです。ですが、章末のコラム「損失分解」（120ページ）で述べるように、単にテーブル分割することと正規化は同義ではありません。

　第2正規形化した「社員」テーブル（と「会社」テーブル）は、かなりデータ登録および更新時の不都合を防止することができる形になりました。しかし、まだ完全に防止できるわけではありません。不都合が起きるのはどのようなケースか、調べてみましょう。

■第2 正規形（再掲）

社員

会社コード	社員ID	社員名	年齢	部署コード	部署名
C0001	000A	加藤	40	D01	開発
C0001	000B	藤本	32	D02	人事
C0001	001F	三島	50	D03	営業
C0002	000A	斉藤	47	D03	営業
C0002	009F	田島	25	D01	開発
C0002	010A	渋谷	33	D04	総務

会社

会社コード	会社名
C0001	A 商事
C0002	B 化学

推移的関数従属

今度は「部署コード」と「部署名」のペアに注目します。今、「会社コード」がC0001のA商事について考えてみると、「社員」テーブルからは、開発、人事、営業という三つの部署が存在することがわかります。しかし、A商事の部署がこの三つだけであるという保証はありません。実は、「広報」という部署も持っているかもしれません。ただ、現在は欠員のため、たまたま社員が**一人もいない**だけです。

こういう社員が一人もいない部署を、現在の第2正規形化した状態の「社員」テーブルには登録できません。この理由は、先ほど社員が一人もいない会社を登録できなかったのと同じで、社員IDが主キーの一部である以上、そこをNULLのままレコードを登録できないからです。

このような不都合が発生する理由は、「社員」テーブルの中に、まだ隠れた関数従属性が残っているからです。今、明らかにこの二つの列の間には、

{部署コード} → {部署名}

という関数従属が成立します。一方、社員IDと部署コードの間にも、当然のことながら、

{会社コード, 社員ID} → {部署コード}

という関数従属が存在しています。つまり、全体としては、

{会社コード, 社員ID} → {部署コード} → {部署名}

という二段階の関数従属があるわけです。このように、テーブル内部に存在する段階的な従属関係のことを、**推移的関数従属**と呼びます。

第３正規化を行なう

この推移的関数従属によるデータ登録時の不都合を解消するには、第2正規化のときと同じように、テーブルを分割することで、それぞれの関数従属の関係を独立させます。

■第3 正規形

社員

会社コード	社員ID	社員名	年齢	部署コード
C0001	000A	加藤	40	D01

C0001	000B	藤本	32	D02
C0001	001F	三島	50	D03
C0002	000A	斉藤	47	D03
C0002	009F	田島	25	D01
C0002	010A	渋谷	33	D04

会社

会社コード	会社名
C0001	A 商事
C0002	B 化学
C0003	C 建設

部署

部署コード	部署名
D01	開発
D02	人事
D03	営業
D04	総務

このように、新しく部署を管理するための「部署」テーブルを独立させることで、すべてのテーブルについて、非キー列はキー列に対してのみ従属するようになり、推移的関数従属もなくなりました。それにより、先ほど問題だった、一人もいない部署を登録できない、という問題は解消されます。「部署」テーブルに登録するときには、社員の情報が必要ないからです。

この第3正規化も、社員と部署という、これまたレベルの異なる実体を異なるテーブルとして切り分けてやる作業として見れば、第2正規形と同じ意味を持っています。そしてまた、第3正規化も無損失分解です。以下のようなSQL文によって、第3正規化を行なう前の状態にテーブルを戻すことが可能です。

■第２正規形に戻す（「社員」テーブルと「部署」テーブルを内部結合する）

```
SELECT 社員.会社コード,
       社員.社員ID,
       社員.社員名,
       社員.年齢,
       社員.部署コード,
       部署.部署名
  FROM 社員 INNER JOIN 部署
    ON 社員.部署コード = 部署.部署コード;
```

結果

```
会社コード   社員ID   社員名   年齢   部署コード   部署名
--------   -----   ----   ---   --------   ----
C0001      000A    加藤     40    D01        開発
C0001      000B    藤本     32    D02        人事
C0001      001F    三島     50    D03        営業
C0002      000A    斉藤     47    D03        営業
C0002      009F    田島     25    D01        開発
C0002      010A    渋谷     33    D04        総務
```

ここでもやはり、正規化の逆操作は結合SQLであることが確認できます。

本章の冒頭でも述べたように、一般的な業務を設計する場合、ほとんどのケースでは第3正規形までを意識しておけば事足ります。したがって、理論的な話に興味のない方は、ここから数節は飛ばして、3-10節のまとめを読み、演習問題へ取り組んでいただいてもかまいません。この3-7節から3-9節は、必要になったときに再度読み返していただけば十分です。

一方、複雑な業務設計においては、より高次の正規形を必要とするケースも、たまにですが存在することも事実です。正規形のレベルは第5まで定義されており、こうした第3を超える正規形に留意する必要がある場合があります。第3を超える正規形のことを「高次正規形」と呼びますが、本節以降の数節では、そのような高次正規形について解説します。

3次と4次の狭間

正規形において、3次のすぐ上位に位置するのが、ボイス－コッド正規形です。これは考案者の二人の名前から取られており、BCNF（Boyce–Codd normal form）という略称でも呼ばれます［※11］。

「第3正規形の次は第4正規形では？」と思った方もいるでしょう。しかし、正規形の理論では第3と第4の間に、このボイス－コッド正規形が定義されています。というのも、実はこの正規形はある意味で第3正規形をより厳密にしたものと考えられるからです。そのため、ボイス－コッド正規形は、非公式にですが「第3.5正規形」という呼び方もされます。

第3正規形を満たしていて、かつ、ボイス－コッド正規形を満たしていないテーブルというのは、あまりお目にかからないのですが、たとえば以下のようなテーブルです。

■ボイス－コッド正規形を満たしていないテーブル

社員-チーム-リーダー

社員ID	チームコード	チーム補佐
000A	001	123W
000B	001	456Z
000B	002	003O
001F	001	123W
001F	002	003O
003O	002	999Y

このテーブルの意味は次のとおりです。テーブルの主キーは、{社員ID, チームコード}です。その意味で、このテーブルは社員とその属するチームの関係を示しています。

社員は、複数のチームに同時に所属することができます（社員「000B」のように）。それゆえ、以下のような主キーから非キーへの関数従属性があります。

{社員ID, チームコード} → {チーム補佐}

一方、このテーブルには、もう一つの業務ルールが存在すると仮定します。それは、「チーム補佐」から「チームコード」に対する関数従属がある、とするという仮定です。チーム補佐は、社員を各チームにおいて補佐する役割です。同じチームに複数人いることもありますが、一人が複数のチームを兼任することはできません。つまり、このテーブルには以下のような関数従属も存在するのです。

{チーム補佐} → {チームコード}

このテーブルには、部分関数従属および推移的関数従属はありません。したがって、文句なく第3正規形です［※12］。しかし、ボイス－コッド正規形ではありません。というのも、ボイス－コッド正規形とは、「社員-チーム-リーダー」テーブルに見られるような**非キーからキーへの関数従属をなくした状態**を指すからです（図3-3）。

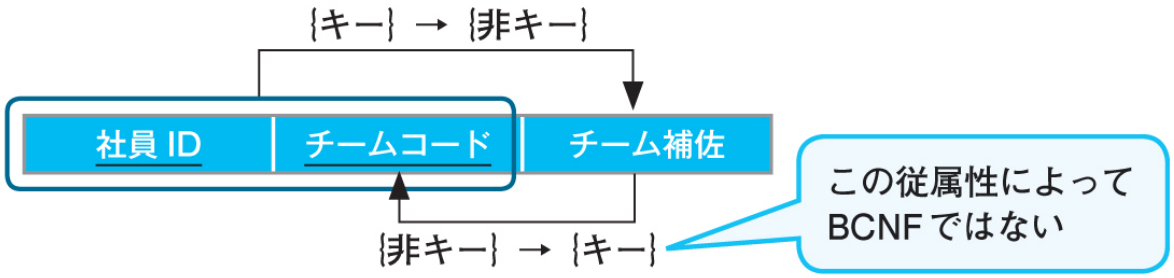


図3-3●非キーからキーへの関数従属をなくした状態

ボイス－コッド正規形でないと、更新時に次のような問題が発生します。

問題1 チーム補佐が担当チームを変える場合に複数行の更新が発生する（データの冗長性）。

問題2 社員がチームに参加するまで、チーム補佐とチームの関連を登録できない。

問題3 社員がチームから外れたときにレコードを削除すると、チーム補佐とチームの関連も削除される危険がある。

こうした不整合を解消する方法が、ボイス－コッド正規形です。

これまでの正規化の手順にならうと、「｛チーム補佐｝ → ｛チームコード｝」に関数従属が存在する以上、チーム補佐を主キーとするテーブルを独立させるのが良いように思えます。すると、次のような形にテーブルが分解されます。

■「社員-チーム-リーダー」テーブルを分解

社員-チーム

社員ID	チームコード
000A	001
000B	001
000B	002
001F	001
001F	002

003O	002

チーム補佐-チーム

チーム補佐	チームコード
123W	001
456Z	001
003O	002
999Y	002

　このように分解すると、どちらのテーブルにも、もう非キーからキーへの関数従属はなくなっていることがわかります。したがって、ボイス－コッド正規形を満たしているように思われます。ところが、この分解には重大な問題があります。この二つのテーブルからは、**元のテーブルに戻すことができない**のです。試しにやってみればわかりますが、結合のSQL文は、次のようにチームコードを結合キーとして二つのテーブルを結合させるものになります。

■「社員-チーム」テーブルと「チーム補佐-チーム」テーブルを内部結合する

```
SELECT 社員-チーム.社員ID,
       社員-チーム.チームコード,
       チーム補佐-チーム.チーム補佐
  FROM 社員-チーム INNER JOIN チーム補佐-チーム
    ON 社員-チーム.チームコード = チーム補佐-チーム.チームコード;
```

結果

```
社員ID   チームコード   チーム補佐
-----   ---------   -------
000A    001         123W
000A    001         456Z
000B    001         123W
000B    001         456Z
000B    002         003O
000B    002         999Y
001F    001         123W
001F    001         456Z
001F    002         003O
001F    002         999Y
003O    002         003O
003O    002         999Y
```

SQLの結果をよく見てください。

「あれ、元のテーブルよりレコード数が増えている……」

そう、この結果には、存在してはならないレコード（枠で囲った部分）まで含まれてしまっているのです。これはつまり、「社員-チーム-補佐」の間に、現実には存在しない関連が表現されてしまっている、ということです。社員「000B」のチーム「001」における補佐は、「456Z」ただ一人のはずが、「123W」まで補佐することになっています。これはおかしな話です。

このようなおかしな事態になってしまった理由は、上記で行なったテーブルの分解が、可逆的ではなかった（＝非可逆）からです。3-5節の「無損失分解と情報の保存」（96ページ）で説明したように、正規化は必ず可逆的な操作でなければなりません。ボイス－コッド正規形への変換時は、気をつけないとこうした非可逆な分解を行なってしまう危険があります。

勘どころ 31

ボイス－コッド正規形への分解時には気をつけないと非可逆な分解を行なってしまうことがある。

この原因は、「チームコード」をキーにした結合が、「多対多」の関連になってしまったからです。正規化は常に「1対多」の関連を生むようにテーブルを分割する必要があります。この「1対多」「多対多」という関連の種類については、次章で詳しく見ます。

ボイスーコッド正規化を行なう

■ボイス－コッド正規形を満たしていないテーブル（再掲）

社員-チーム-リーダー

社員ID	チームコード	チーム補佐
000A	001	123W

000B	001	456Z
000B	002	003O
001F	001	123W
001F	002	003O
003O	002	999Y

それでは、正しい分解の仕方とはどのようなものでしょうか？

先ほどの失敗は、元のテーブル（上記）に存在していた以下の関数従属性を失ったために起きたことでした。

{社員ID, チームコード} → {チーム補佐}

したがって、この関数従属性を保存することで、可逆的な分解が可能になります。

■ボイス－コッド正規形

社員-チーム補佐

社員ID	チーム補佐
000A	123W
000B	456Z

000B	003O
001F	123W
001F	003O
003O	999Y

チーム補佐-チーム

チーム補佐	チームコード
123W	001
456Z	001
003O	002
999Y	002

　この二つのテーブルは、次の結合SQLによってオリジナルのテーブルへ戻すことが可能です。

■「社員-チーム補佐」テーブルと「チーム補佐-チーム」テーブルを内部結合する

```
SELECT 社員-チーム補佐.社員ID,
       チーム補佐-チーム.チームコード,
       チーム補佐-チーム.チーム補佐
  FROM 社員-チーム補佐 INNER JOIN チーム補佐-チーム
    ON 社員-チーム補佐.チーム補佐 = チーム補佐-チーム.チーム補佐;
```

結果

```
社員ID   チームコード  チーム補佐
-----   ---------- -------
000A    001        123W
000B    001        456Z
000B    002        003O
001F    001        123W
001F    002        003O
003O    002        999Y
```

今度は、「チーム補佐」をキーにすることで、結合が「1対多」になりました。これが正しいボイス－コッド正規形なのですが、実はまだ問題は完全に解決したわけではありません。というのも、この形のテーブルでは、本来登録できてはいけないはずのデータが登録できるようになってしまう可能性があるからです。たとえば、「社員-チーム補佐」テーブルに、社員「000A」と補佐「003O」のペアを登録することは、何の問題もなく可能です。

しかし、この組み合わせは「チーム補佐-チーム」テーブルを見るとおかしいことに気づきます。補佐「003O」が担当できるチームは「002」だけなので、これはつまり、現在はチーム「001」にしか所属していない社員「000A」が、チーム「002」にも所属するようになることを、**暗黙に**意味してしまう、ということになります［※13］。

もちろん、それが許される場合もあります。たとえば、「003O」さんの補佐を受けるということは、すなわちチーム「002」への参加を意味するのだ、という**ルールが業務**

的に決められている場合もあるでしょう。

しかし、すべての業務でそのようなルールが存在するとは限りません。そして問題は、そのようなルールが存在しない場合です。そのときは、このような「ありえない」組み合わせのレコードを登録することがないよう、アプリケーション側で制御する必要が出てきます。

このように、ボイス－コッド正規形というのは、なかなか難しい問題を含んだ正規形です。

1974年に、レイモンド・ボイスとエドガー・F・コッドが定義しました。コッドは、リレーショナルデータベースの基礎理論そのものを提示した人でもあります。実際には、二人に先立って1971 年に、イアン・ヒースが同等の概念を提出しており、そのため「この正規形は本当はヒース正規形と呼ばれるべきだ」と主張する理論家もいます（C.J.Date『データベース実践講義』、p.154）。

100ページの「社員」テーブルにおける、推移的関数従属は「部署コード」→「部署名」の部分です（非キー→非キーへの関数従属のため）。{会社コード, 社員ID} → {部署コード} は、キー→非キーの関数従属であるため、推移的関数従属にはなっていません。

これは、オリジナルのテーブルには存在していた社員とチームの関係を、正規化によって壊してしまったからです。

次に、第4正規形がどのような形式かを見ていくことにします。例として、先ほどとよく似た、社員と、社員が所属するチームの対応テーブルを考えます。

■第3 正規形

社員-チーム

社員ID	チームコード
000A	001
000B	001
000B	002
001F	001
001F	002
003O	002

社員「000A」のように、一つのチームにしか所属していない社員もいれば、「000B」のように複数のチームに所属している社員もいます。そのため、社員とチームの所属関係を適切に表現するには、{社員, チームコード}という二つの列によって主キーを構成する必要があります。つまり、このテーブルは、主キーしか持たないテーブルということです。

このように、「社員」や「チーム」といったエンティティ同士の関連を表現するエンティティを、文字通り「**関連エンティティ**」と呼びます。関連エンティティについては、次章で詳しく取り上げますが、実は第4正規形および第5正規形は、この関連エンティティに対して発生するもので、第3正規形までで考えてきたような、「社員」や「会社」といった、具体的な実体を伴ったエンティティの正規化では発生しません。

さて、話を元に戻すと、この「社員-チーム」テーブルは、第3正規形であることは自明です。このテーブルには、部分関数従属も推移的関数従属もありません。非キー列がないため、非キーからキーへの関数従属性もありません（ゆえにボイス－コッド正規形でもあります）。ここまでは、特に問題ないでしょう。

では次に、このテーブルにもう一つ、「製品コード」という列を追加してみましょう。これは、社員が開発に当たっている製品を表わすもので、一人の社員が複数の製品開発に携わることもあれば、一つの製品を複数の社員が共同開発していることもあります。

■「製品コード」列を追加（第３正規形）

社員 - チーム - 製品

社員 ID	チームコード	製品コード
000A	001	P1
000A	001	P2
000B	001	P1
000B	002	P1
001F	001	P2
001F	002	P2
003O	002	P2
003O	002	P3

社員「000A」は「P1」と「P2」の二つの製品を担当

この「社員-チーム-製品」テーブルにおいても、やはり第3正規形（およびボイス－コッド正規形）が成立していることを確認しておきましょう。{社員ID, チームコード, 製品コード}の3列で主キーが構成されるため、やはり主キーだけのテーブルということになります（このように主キーだけで構成されるのは、関連エンティティの特徴です）。

多値従属性～キーと集合の対応

しかし、「社員-チーム-製品」テーブルには、これまで見てきた下位の正規形において発生してきたのと同様、更新時の問題が発生します。たとえば、社員「000A」がチームを「001」から「002」へ異動することになった場合、複数行に対する更新が発生します。これはデータの保持が冗長であることを意味しており、不整合が発生する可能性が高まるほか、更新コストも高くなります。また、すべての列が主キーの

一部を構成するため、NULLを指定することができません。そのため、全列の情報が確定するまでレコードを登録できないという不便も発生します（所属チームは決まったけど、担当製品が決まっていない社員はどうすればいいでしょうか？）。

こうした不都合を解消する方法を考えます。まず、このテーブルにおける従属性には、関数従属性と異なる特徴があります。従属性が、キーと非キーの間ではなく、キーと集合の間に成立しています（関数従属性は、キーと非キー列の間に成立していました）。

たとえば、社員IDを一つ決めると、それに対してチームは「複数」定まります。同様に、製品コードも、社員ID一つに対して「複数」定まります。つまり、キーに対して、値の「集合」が対応するわけです（図3-4）。これは関数従属には見られなかった特徴です。

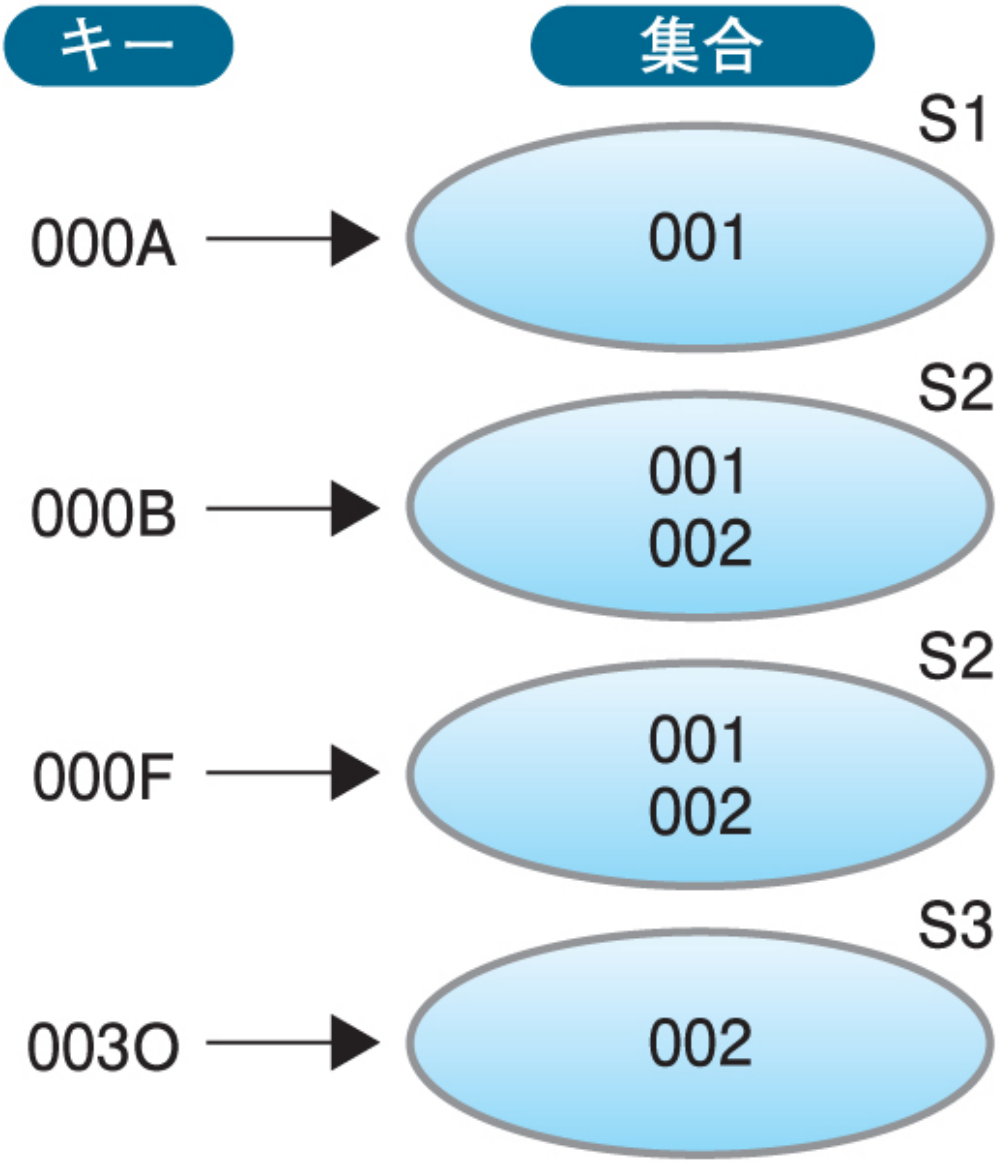


図3-4●社員とチームの従属性

このようなキーと集合との対応を**多値従属性**と呼びます。「→→」という記号で表わします。この例で言えば、以下の二つの多値従属性が成り立っています。

{社員ID}　→→　{チームコード}

{社員ID}　→→　{製品コード}

今、チームと製品の間には特に関係がないので、従属性はありません。このような場合、上記の二つの従属性をまとめて以下のように一つの式で表わします。

{社員ID}　→→　{チームコード|製品コード}

第４正規化を行なう

第4正規形は、上記のような独立な多値従属性が複数存在するテーブルを分割することで作られます。つまり、「｛社員ID｝　→→　｛チームコード｝」を表わすテーブルと、「｛社員ID｝　→→　｛製品コード｝」を表わすテーブルに分離すれば良いのです。

■第4 正規形

社員-チーム

社員ID	チームコード
000A	001
000B	001

000B	002
001F	001
001F	002
003O	002

社員-製品

社員ID	製品コード
000A	P1
000A	P2
000B	P1
001F	P2
003O	P2
003O	P3

見てわかるとおり、「社員-チーム」テーブルは社員とチームの間の多値従属性を、「社員-製品」テーブルは社員と製品の間の多値従属性を表わしています。この形式ならば、社員「000A」がチームを「001」から「002」に移ったとしても、1レコードだけの更新で済むため、更新不整合が発生する可能性はなく、更新コストも最小に

抑えられます。担当する製品を変える場合でも、話は同様です。また、この分解は可逆的であるため、いつでも元のテーブルに戻すことが可能です。

第４正規形の意義

この第4正規形は、ある意味で「自然な」正規形です。複数の多値従属性を一つのテーブルで表現しようとするのは、正味な話、かなり無理のある設計だからです。つまり、第2正規形どまりのテーブルがしばしば「自然な流れ」で作られる可能性があり、第3正規形を意識する意義があるのに対し、「社員-チーム-製品」テーブルのように、「第3正規形だが第4正規形でない」テーブルが「自然な流れ」で作られることは、現実にはあまりありません。

しかし、「社員-チーム-製品」テーブルは第3正規形を満たしていながら、冗長性を残したテーブルとして成立しているため、論理的には作ることができます。そのため、こうしたテーブル構成を明示的に禁止しておく、というのが第4正規形の意義です。これは要するに、「関連エンティティを作る場合は、そこに含まれる関連は、一つだけにすること」という設計上のルールが必要だ、ということです。それを守っていれば、みなさんの設計したテーブル群は「自然に」第4正規形になります。

勘どころ 32

関連エンティティに含まれる関連は一つだけにすること。

第5正規形は、第4正規形をもう少し発展させたものです。先ほどの「社員-チーム-製品」テーブル（110ページ）は、「｛社員｝ →→ ｛チーム｝」「｛社員｝ →→ ｛製品｝」という多値従属性がありました。ここにもし、さらに「｛チーム｝ →→ ｛製品｝」という多値従属性（図3-5）があったとしたらどうでしょう？ つまり、チームによっても扱う製品が異なる、という業務上のルールが存在する場合です。

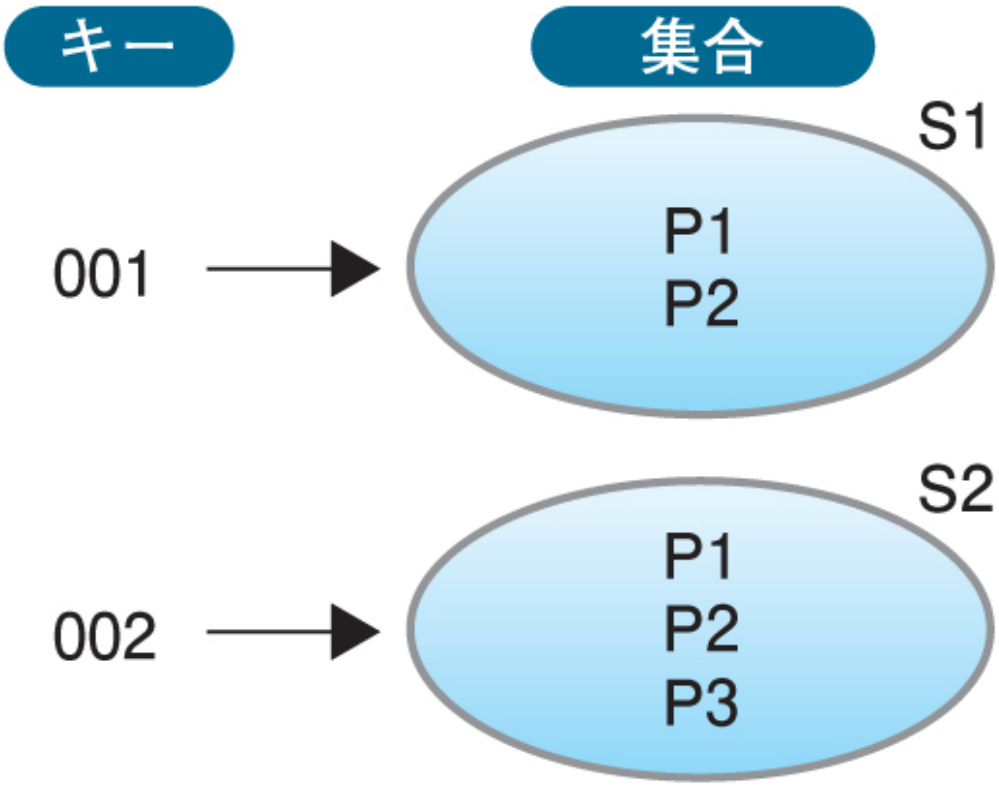


図3-5●チームと製品の従属性

こうしたケースでは、第4正規形で作ったような「社員-チーム」「社員-製品」の二つのテーブルだけからでは、「チーム-製品」間の従属性がわからないという欠点があります。

■第4 正規形（再掲）

社員-チーム

社員ID	チームコード
000A	001

000B	001
000B	002
001F	001
001F	002
003O	002

社員-製品

社員ID	製品コード
000A	P1
000B	P1
001F	P2
003O	P2
003O	P3

第５正規化を行なう

この問題を解決するには、「チーム-製品」の関連を表わす関連エンティティも作ってやれば良い、ということになります。

■第5 正規形

社員-チーム

社員ID	チームコード
000A	001
000B	001
000B	002
001F	001
001F	002
003O	002

社員-製品

社員ID	製品コード
000A	P1
000B	P1
001F	P2

003O	P2
003O	P3

チーム-製品

チームコード	製品コード
001	P1
001	P2
002	P1
002	P2
002	P3

三つのテーブルを見て、第4正規形のときと同じように「これは自然なテーブル設計ではないか」という感想を持つ人もいるでしょう。実際、「一つの関連につき一つのテーブル」という対応でテーブルを作れば良いのですから、シンプルな方針です。第5正規形を満たすために必要な設計上のルールは、ちょうど第4正規形の場合と逆で「関連がある場合は、それに対応する関連エンティティを作ること」というものです。つまり、関連と関連エンティティは一対一に対応させろ、ということです。このルールを守っていれば、みなさんの設計したテーブル群は「自然に」第5正規形になります。

さて、これで第5正規形まで到達しました。みなさんが開発に携わるほとんどの開発業務では、第3正規形まで到達できていれば、ほとんどカバーすることが可能です。ここで正規化について重要なポイントをおさらいしておきましょう。

正規化の三つのポイント

ポイント1 正規化とは更新時の不都合／不整合を排除するために行なう

正規化を行なう目的は、何よりも更新（データ登録＝INSERTも含む）時の不都合を防ぐためのものです。また、データの冗長性を排除して、人間のオペレーションミスによるデータ不整合を防ぐ目的もあります。

ポイント2 正規化は従属性を見抜くことで可能になる

正規化を行なうためには、テーブル内部の従属性の関係を見抜く必要があります。部分関数従属（第2正規形）、推移的関数従属（第3正規形）が存在

していれば、まず正規化の対象になります。また、多値従属性が存在していたら、自分が第4正規形に反する設計をしていないか注意しましょう。

ただし、こうした従属性はテーブルを形式だけ見ていてもわかりません。どの列がどのキーに従属しているか、ということは業務ロジック（ビジネスルール）で決まることなので、各列が業務上どのような意味と関係を持っているか、ということを調べなければなりません（業務分析の必要性）。

ポイント3 正規形はいつでも非正規形に戻せる

正規化によって分割されたテーブルは、いつでも非正規化テーブルに復元することができます。これは正規化が情報を完全に保存する**無損失分解**だからです。そしてこのことが示すのは、高次の正規形は、低次の正規形を含んでいる、ということです。正規形同士のレベルの関係を示すと図3-6のようになります［※14］。

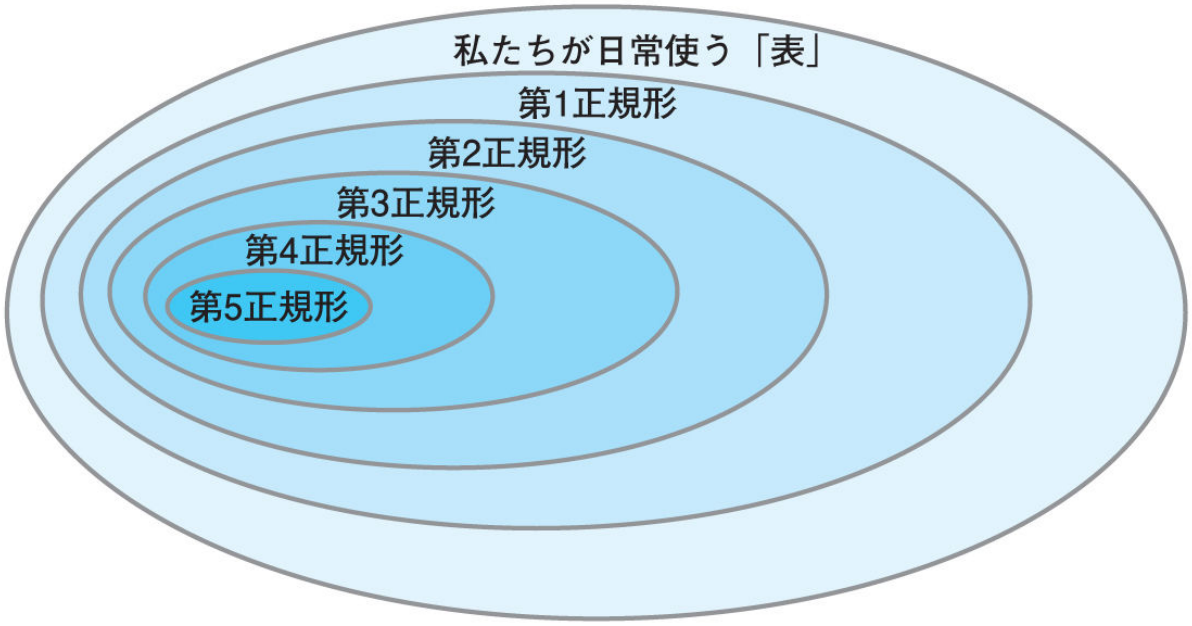


図3-6●正規形同士のレベルの関係

一番外側に、私たちが使うあらゆる形式の「表」があります。ここには、第1正規形すら満たしていない雑多なものがすべて含まれるカオスの世界です。その内側に、第1正規形の領土があります。この内側が、リレーショナルデータベースの世界です。あとは、第2正規形は第1正規形に含まれ、第3正規形は第2正規形に含まれ……という形で入れ子になっていきます。

正規化は常にするべきか？

この問題に対する回答は、以下のようなものです。

- 第3正規形までは、原則として行なう。
- 関連エンティティが存在する場合は関連とエンティティが1対1に対応するよう注意する。

一般に正規化を行なえば行なうほど、以下のような利点があります。

利点1 データの冗長性が排除され、更新時の不整合を防止できる。
利点2 テーブルの持つ意味が明確になり、開発者が理解しやすい。

データベースの第一目的はデータを整合的な状態で保持することにあるため、何よりもデータ不整合を防止することに主眼が置かれます。正規化はそのために考え出された方法論です。

一方、正規化の主な欠点は以下のようにパフォーマンスに関わるものです。

欠点■ テーブルの数が増えるため、SQL文で結合を多用することになり、パフォーマンスが悪化する。

この欠点はしばしば無視できないため、パフォーマンス向上のため、あえて正規形を低次なものにとどめる設計が採用されることはあります。この「非正規化」については、第5章で詳しく取り上げます。

正規化を学ぶことに対する的外れな批判について

COLUMN

正規化について学ぶと、程度に差はあれ、少なからぬ人がこう感じます。

「わざわざ理論として教えられなくても、そのぐらいのことは論理設計をやる人は常識的にやっていることではないか？」

実際、正規形についてまったく知らない人であっても、独力で更新不整合をなくそうとしたり、自分なりの美意識で「きれいな」テーブル構成を行なおうとしたり努力する過程を通して、図らずも正規化について学んだ人と同じ論理設計にたどりつくこともしばしばあります。そうした事例を引き合いに出して「正規化なんてわざわざ勉強する必要はない。だってそんなの当たり前のことなんだから」と言う人もいます。

このような批判に対しては、優れた理論家であるクリス・デイトが的確な反論をしています。以下、少し長いですが、重要なことを言っているので引用します。

前述のように、正規化理論は基本的に常識にすぎないという理由で批

前述のように、正規化理論は基本的に常識にすぎないという理由で批判されてきた。……優秀な設計者であれば、たとえBCNFの予備知識がなかったとしても、この関係変数を射影SPとSSに「自然に」分解するだろう。だが、「自然に」とはどういう意味なのか。設計者は「自然な」設計を選ぶうえで、どのような「原理」を用いているのだろうか。

答えは、正規化の原理とまったく同じである。つまり、優秀な設計者は、そうした原理を正式に学んだことがなくても、その名前を言い当てることができなくても、そうした原理がすでに頭の中にある。したがって、原理は確かに常識だが、それらは**正規化された**常識なのである……正規化を批判する人は、だいたいこの点を見逃している。それらの概念が実は常識にすぎないことを（実にもっともらしく）主張するが、総じて、常識の意味を正確かつ形式的に述べることが偉業であることに気付いていない。［※15］

正規化の結果得られる結論は、整合的なシステムを構築しようとすれば必然的に得られるものです。その意味で、正規形の理論はまったく意外性のあるものではありません。むしろ正規形の理論に対する批判者の言うとおり、ごく当たり前のことです。

しかし、正規化のメリットは、まさにその当然さにあるのだ、というのがデイトの反論です。常識を定式化することで誰もが利用可能なツールにすることができ、道を外れた設計を防ぐことができるようになります。

正規化について「常識的」と感じた方々には想像できないと思いますが、世の中には常識人にはまったく想定することすらできない、とんでもない論理設計を行なう「規格外」の人たちが一定数います。論理設計を免許制にして、そのような人々を事前にスクリーニングできれば良いのですが、まだエンジニアを医師や弁護士と同じように免許制にした国というのは聞きません。

本書の後半で、そのような「トンデモ設計」の一部を見ることになります。常識

本書の後半で、そのような「クソテ設計」の一部を見ることになります。常識ある人が見たら目を疑うようなものも含まれています。それらを一通り見た後で、再び正規化されたテーブルを見ると、また今までとは違った感想を持つのではないかと思います。

「常識は、多くの人に広めてこそ常識として価値を持つようになるのだ」と。

損失分解

COLUMN

本章では、正規化がすべて「無損失分解」、すなわち常に分解したテーブルは分解前の状態に戻せるような分解の仕方であることを説明しました。

裏を返すと、これは「無損失」ではない分解、言ってみれば「損失分解」がある、ということです。この「損失分解」という言葉はリレーショナルデータベースでも正規化理論でも使われず、著者が作った造語なのですが、数の上で言えば、分解の仕方としてはこの損失分解のほうが多くあります。それは、たとえば次のようにテーブルをいい加減に分割してしまえば、これは損失分解です。

■損失分解

社員

社員 ID	社員名	年齢	部署
000A	加藤	40	開発
000B	藤本	32	人事
001F	三島	50	営業
001D	斉藤	47	営業
009F	田島	25	開発
010A	渋谷	33	総務

分解 

社員

社員 ID	社員名
000A	加藤
000B	藤本
001F	三島
001D	斉藤
009F	田島
010A	渋谷

年齢部署

年齢	部署
40	開発
32	人事
50	営業
47	営業
25	開発
33	総務

この分解は、元のテーブルを復元不可能な分解です。両方のテーブルを結合するためのキーが存在しないからです。しかも、片方のテーブル（「年齢部署」）から主キーをなくしてしまうという点で、実際にこのように分解する価値はありません。すでに見てきたように、リレーショナルデータベースにおいて主キーを持たないテ

ル。すでに見てきたように、リレーショナルデータベースにおいて主キーを持たないテーブルを作ることは許されないからです。次のように、「年齢部署」テーブルの側にも「社員ID」を主キーとして持たせれば、これは無損失分解となります。

■無損失分解

社員

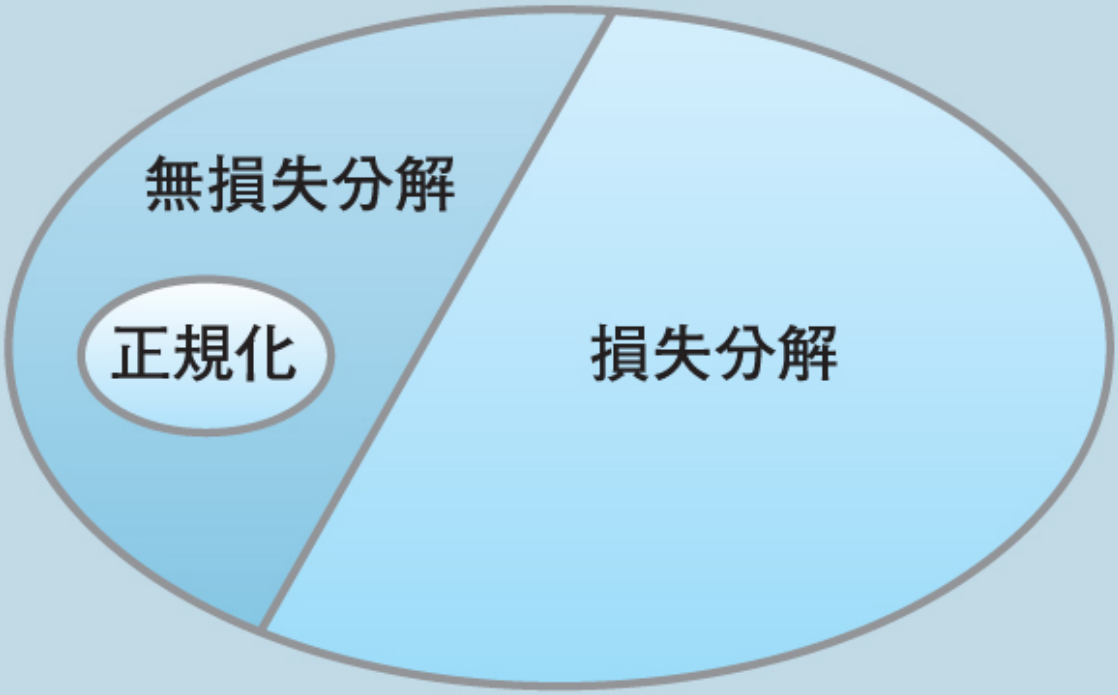
社員ID	社員名
000A	加藤
000B	藤本
001F	三島
001D	斉藤
009F	田島
010A	渋谷

年齢部署

社員ID	年齢	部署
000A	40	開発
000B	32	人事
001F	50	営業

001D	47	営業
009F	25	開発
010A	33	総務

これは正規化ではありませんが、元のテーブルを復元可能な無損失分解です（第7章でバッドノウハウの一つとして取り上げる「垂直分割」です）。つまり、無損失分解であれば必ず正規化になるわけでもないのです。損失分解、無損失分解、正規化の関係を図示すると図Bのようになります。



図B●損失分解、無損失分解、正規化の関係

このように、正規化（および非正規化）というのは、テーブルの操作方法と

このように、正規化（および非正規化）というのは、テーブルの操作方法としては非常に厳格なルールが課された操作です。こうしたルールを無視したテーブル操作を行なうことで、様々なバッドノウハウ、グレーノウハウが生まれてくることを、第7章と第8章で見ていきます。

演習問題

演習 3-1 正規形の次数

次のような「支社支店商品」テーブルについて考えます。主キーは{支社コード、支店コード、商品コード}です。このテーブルの正規形の次数を答えてください。

支社支店商品

支社コード	支社名	支店コード	支店名	商品コード	商品名	商品分類コード	分類名
001	東京	01	渋谷	001	石鹸	C1	水洗用品
001	東京	01	渋谷	002	タオル	C1	水洗用品
001	東京	01	渋谷	003	ハブラシ	C1	水洗用品

001	東京	02	八重洲	002	タオル	C1	水洗用品
001	東京	02	八重洲	003	ハブラシ	C1	水洗用品
001	東京	02	八重洲	004	コップ	C1	水洗用品
001	東京	02	八重洲	005	箸	C2	食器
001	東京	02	八重洲	006	スプーン	C2	食器
002	大阪	01	堺	001	石鹸	C1	水洗用品
002	大阪	01	堺	002	タオル	C1	水洗用品
002	大	02	豊	007	雑	C3	書籍

	阪		中		誌		
002	大阪	02	豊中	008	爪切り	C4	日用雑貨

演習 3-2 関数従属性

「支社支店商品」テーブルの関数従属性をすべて挙げてください。

演習 3-3 正規化

「支社支店商品」テーブルを可能な限り、高次に正規化してください。

この図では、BCNFは第3正規形の厳密化と考えて省略しています。

C.J.Date『データベース実践講義』（オライリー・ジャパン、2006）p.155

第4章

ER図～複数のテーブルの関係を表現する

エンティティ（テーブル）の数が増えると、相互の関連を効率的に把握するための手段が必要になります。それが ER図です。ER図の描き方には様々な「流派」がありますが、本章では比較的ポピュラーなIE表記法とIDEF1Xによる記述方法を学びます。

学習のポイント

- テーブル（エンティティ）の数が増えていくと、テーブル同士の関係がわかりにくくなり、設計に支障をきたします。この問題を解決するため、テーブル同士の関係を記述する道具がER図です。
- ER図の描き方には何種類かフォーマットがありますが、代表的なIDEF1XとIE表記法を学習します。
- リレーショナルデータベースにおけるテーブル間の関係は、基本的に「1対多」です。「多対多」の関係は、「1対多」の関連に分解します。
- 「多対多」を「1対多」に分解するときに必要になるエンティティを「関連実体」と呼びます。

前章で学んだ正規化によってテーブルを整理していくと、どんどんテーブルの数が増えていきます。実際の業務システムを開発すると、何百という数のテーブルが作られることも珍しくありません。

こうしてうじゃうじゃと増えたテーブルを、そのままの状態で管理することは、人間にはできない作業です。こうした多数のテーブルを管理するために、それぞれのテーブルがどういう意味を持っていて、テーブル同士が互いにどういう関係にあるのか、ということを明示するために作る図を**ER**図（Entity-Relationship Diagram：実体関連図）と呼びます。E（Entity：実体）とはテーブルのこと、R（Relationship：関連）は文字通りテーブル同士の関連のことだと考えてもらってかまいません。

ER図の描き方にはいくつか「流派」があるのですが、基本的な考え方は大きく変わるものではないため、代表的なフォーマット二つを覚えておけば、大体の場合には対処できます。その二つが、IE（Information Engineering）表記法とアイデフワンエックス
ＩＤＥＦ１Ｘです。

IE表記法は通称「鳥の足」という名前でも知られており、直観的にテーブル間の関連を理解しやすいため、著者としては初心者向けの記法だと思います。

IDEF1Xは、米国で規格化された表記法で、そのすべての機能を学ぼうとすると、かなり覚えることが多いため、本章では業務上、主に必要となる点に限定して解説します。

それでは、前章で第3正規形まで正規化した「社員」テーブルをサンプルに、ER図を描いてみましょう。

■第3 正規形

社員

会社コード	社員ID	社員名	年齢	部署コード
C0001	000A	加藤	40	D01
C0001	000B	藤本	32	D02
C0001	001F	三島	50	D03
C0002	000A	斉藤	47	D03
C0002	009F	田島	25	D01
C0002	010A	渋谷	33	D04

会社

会社コード	会社名
C0001	A 商事
C0002	B 化学
C0003	C 建設

部署

部署コード	部署名
D01	開発
D02	人事
D03	営業
D04	総務

ER図を描くとき、最初に着目するポイントは、あるテーブルの主キーが、他のテーブルに列として含まれているかどうか、という点です［※1］。なぜなら、もしその場合、二つのテーブルの間には意味的な関連があることになるからです。

たとえば、「会社」テーブルと「社員」テーブルの場合、「会社」テーブルの主キーである会社コードが、「社員」テーブルにも含まれています［※2］。また、「部署」テーブルの主キーである部署コードも、「社員」テーブルに含まれています。

このような場合、二つのテーブルの間には**1対多**の関係が成立している、と言います。たとえば、「会社」テーブルには、一つの会社は1行しか含まれませんが（会社コ

ードが主キーなのだから当然です）、「社員」テーブルには一つの会社が複数行に現われます。つまり、「会社：1」に対して「社員：多」という関係があるわけです。これは、普通の日本語で言えば「**一つの会社では複数の社員が働いている**」ということで、至極当然のことではあります。より厳密に言うなら、このテーブルの関係を見ると、「会社」テーブルには一人も社員のいない会社「C建設」も登録可能ですから、「一つの会社には0 ～ n人の社員が働いている」という言い方が適切です（nは適当な整数）。

もう一方の「部署」テーブルについても、やはり一つの部署が1行しか含まれませんが、「社員」テーブルには複数行の部署が含まれています。これも普通の日本語で言えば「一つの部署には複数の社員が所属している」という意味になります。このサンプルの場合、社員が所属していない部署、というのは存在しないことになっているので、社員の数は先ほどとは違って、1 ～ nと考えられます［※3］。

1対1、1対多、多対多

このように、同じ意味の列を持っているテーブル同士の間では、一般に次の3パターンの関連がありえます［※4］。

パターン1 1対1
パターン2 1対多
パターン3 多対多

このうち、**パターン1** の「1対1」というのは、あまり見かけません。というのも、二つのテーブルのレコードが1対1に対応するということは、要するに二つのテーブルの主キーが一致するケースであり、そうであれば普通は一つのテーブルにまとめてしまっても問題ないからです。少なくとも、正規化の過程でこのような1対1のテーブルが作られることはありません。

パターン2 の「1対多」は、最もよくある関連のタイプです。基本的に正規化によって生まれる関連はこのカテゴリに属します。「会社-社員」間の関連も、「部署-社員」間の関連もこのカテゴリです。先ほど注意点として述べたように、厳密には「1対多」と「0または1対多」に分かれますが、一般的にはこの二つのサブカテゴリをまとめて「1対多」と呼ぶことが多いです。また、「多」の側についても、「0 以上」と「1以上」の場合があるのですが、こうした細かい区別についても、ER図で表記することが可能です。

最後に、**パターン3** の「多対多」ですが、これは少し特殊なカテゴリです。というのも、最初に業務要件からテーブルを作っていくと、この多対多の関連を持ったテーブル群ができあがることがあります。ただし、リレーショナルデータベースの「お約束」として、この多対多の関連は作ってはならない、ということになっています。なぜそういうお約束があるのか、具体的にどうやってこのカテゴリの関連を解消するのか、という点については、4-4節で見ます。

名目上、互いに異なる列名が使われていても、実質的に同じ内容を表わしている列同士であれば同様です。たとえば、一方の列に「会社コード」、もう一方の列に「取引先コード」のような名称が使われている場合にも、中身が同じ会社の集合を表わしているならば、関連があると考えられます。

「社員」テーブルの主キーは、{会社コード, 社員ID }のため、会社コード単独では主キーにならない点に注意してください。

本当にそうであるかは、業務要件しだいです。ここではあくまでサンプルとしてそのような業務要件を仮定しています。実際には、社員が一人もいない部署を許すような要件もあるかもしれま

せん。

この3パターンの表記方法としては、「1：1」「1：N」「N：M」のようなものもあります。

4-3 ER図の描き方

さて、お待たせしました。それでは、社員、会社、部署の三つのテーブルの関係をER図で表現してみましょう。

テーブル（エンティティ）の表記方法

まず、ER図でのテーブルの表現方法ですが、これはIDEF1XもIE表記法も同じで、図4-1のような四角形で表わします。

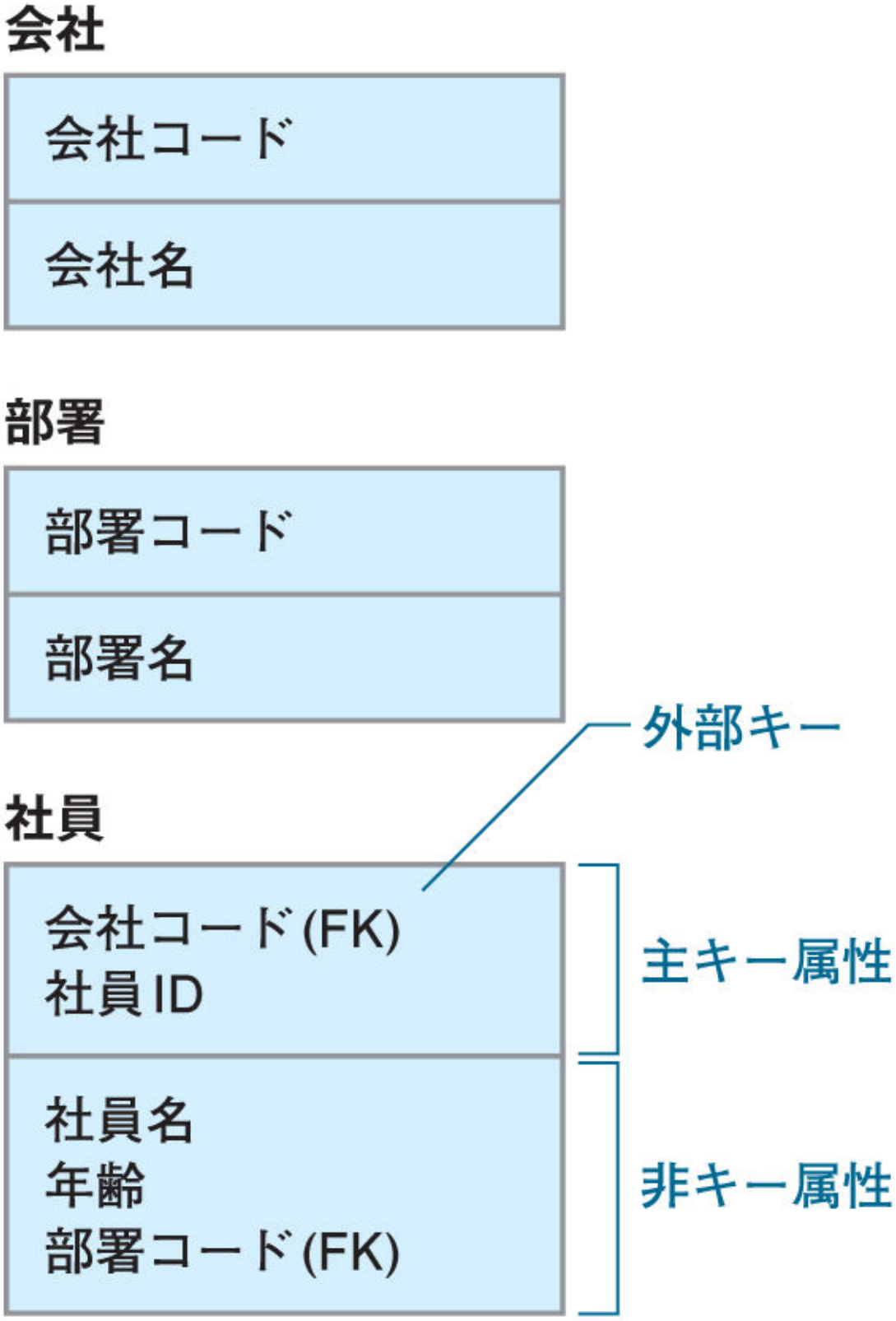

会社
会社コード
会社名
部署
部署コード
部署名
外部キー
社員
会社コード(FK)
社員ID
主キー属性
社員名
年齢
部署コード(FK)
非キー属性

図4-1●ER 図で描いたテーブル

四角の中を横線で二つのスペースに区切っていますが、上のスペースは主キー属性、下のスペースは非キー属性を記述する場所です（非キー属性は、数が多い場合は代表的なものだけを記述します）。また、他のテーブルの主キーを参照する**外部キー**（foreign key）については、属性名のとなりに略称の「FK」を記述します。

IE表記法でER図を描く

これだけではまだ、エンティティを並べただけで、互いの関連が描き込まれていません。いわばこれは「E図」です。これだけでは何の役にも立たないので、テーブル間の関連を記述して、完全なER図を完成させましょう。この関連の記述方法が、IDEF1XとIE表記法とで異なりますので、違いに注意して理解しましょう。

まずは、直観的に理解しやすいIE表記法から見ていきましょう。引き続き、会社、社員、部署の三つのエンティティをサンプルに利用します（図4-2）。

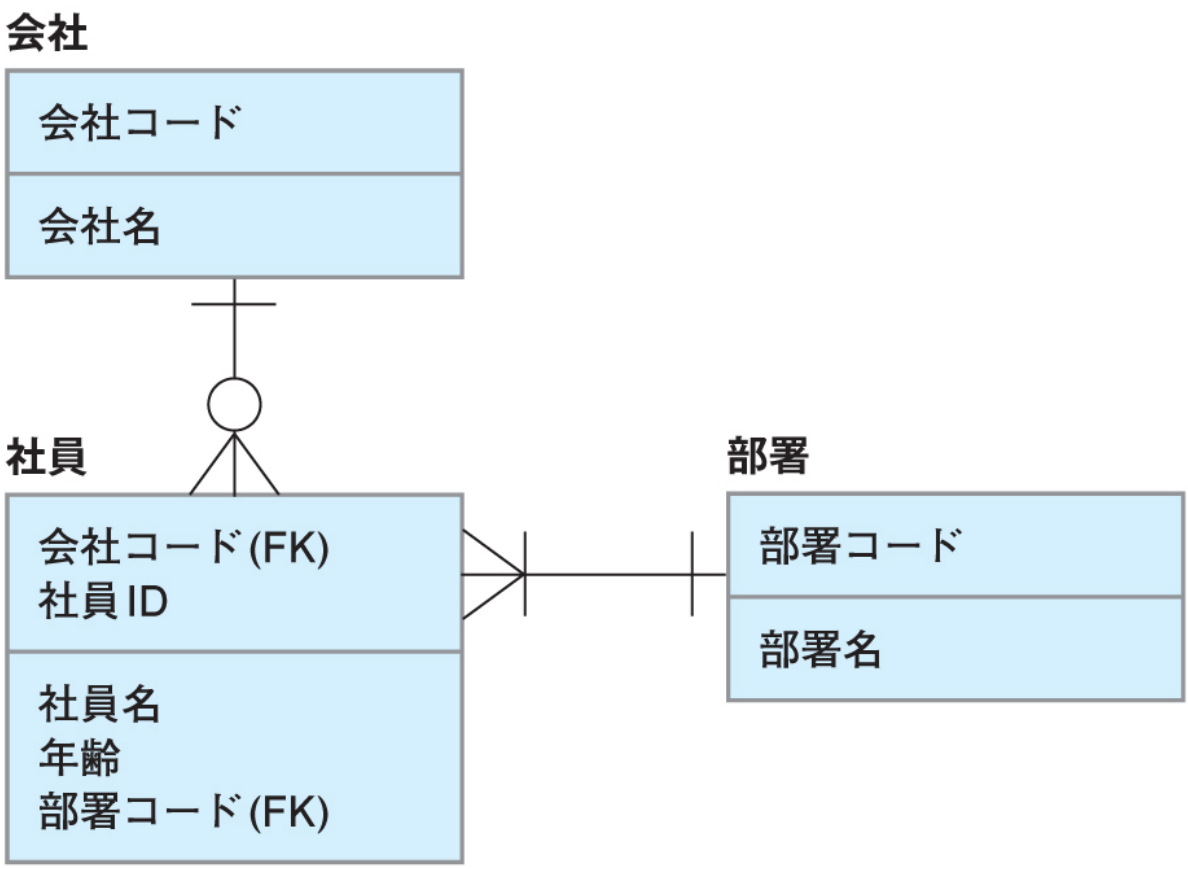


図4-2●IE表記法によるER 図

いくつか記号が出てきました。これらの意味と、ER図全体をどう解釈するか、という点を解説しましょう。

まず、会社と社員のエンティティから見ていきます。「会社」エンティティの側についている横棒「—」は、相手のエンティティと対応するレコード数（これを**カーディナリティ**と呼びます）が1であることを示しています。漢数字の「一」を連想させるので、覚えやすいと思います。一方、「社員」テーブル側の「○」「⩚」はそれぞれ、カーディナリティが「0」および「複数」を意味します。つまりこの二つを合わせて「0以上の複数」という意味になります。IE表記法が「鳥の足」と呼ばれるのは、複数を表わす「⩚」に由来しています。これは誰にでも理解しやすい、優れた記号です。

つまり、このER図は、会社と社員の関連が「1対多（ただし0も含む）」であることを表わしています。事実、「会社」テーブルから見れば、一つのレコード（＝一つの会社）を決めれば、「社員」テーブルは0 ～複数人が対応します（C建設のように今のところ、社員の登録がない会社は、「社員」テーブルのカーディナリティは0です）。一方、「社員」テーブルから見れば、一つのレコード（＝一人の社員）を決めれば、必ず一つの会社が対応します。**会社に属していない社員は存在しない**からです。

対して、部署と社員の関連はどうでしょう。部署と社員も「1対多」の関係であることは、会社と社員の場合と同じです。ただし、会社の場合と違って、部署の場合は、一人も社員が所属しない部署、というのは存在しません。したがって、社員の側のカーディナリティが0ということはありえないので、1を示す「―」と鳥の足が組み合わされています。これは「1以上」という意味になります。一方、社員を一人決めれば、部署は一つに決まるので、部署側は「―」だけが使われています。実際には、「社員」テーブルの部署コードはNULLがありうるため、必ずしも「部署」テーブルのレコードが決まるわけではないのですが、IE表記法ではそこまで厳密に表現はしません。

勘どころ 33

IE表記法のカーディナリティの記号は以下のとおり。

○：0

―：1

⩚：多（2以上）

この基本的な三つを組み合わせることで、先ほどの例のように複雑なカーディナリティのパターンを表現することができるわけです。

IDEF1XでER図を描く

次に、同じER図をIDEF1Xに変換してみましょう。カーディナリティを表記する記号など、いろいろ変わるポイントがあるので、間違い探しのつもりで注意してみてください（図4-3）。

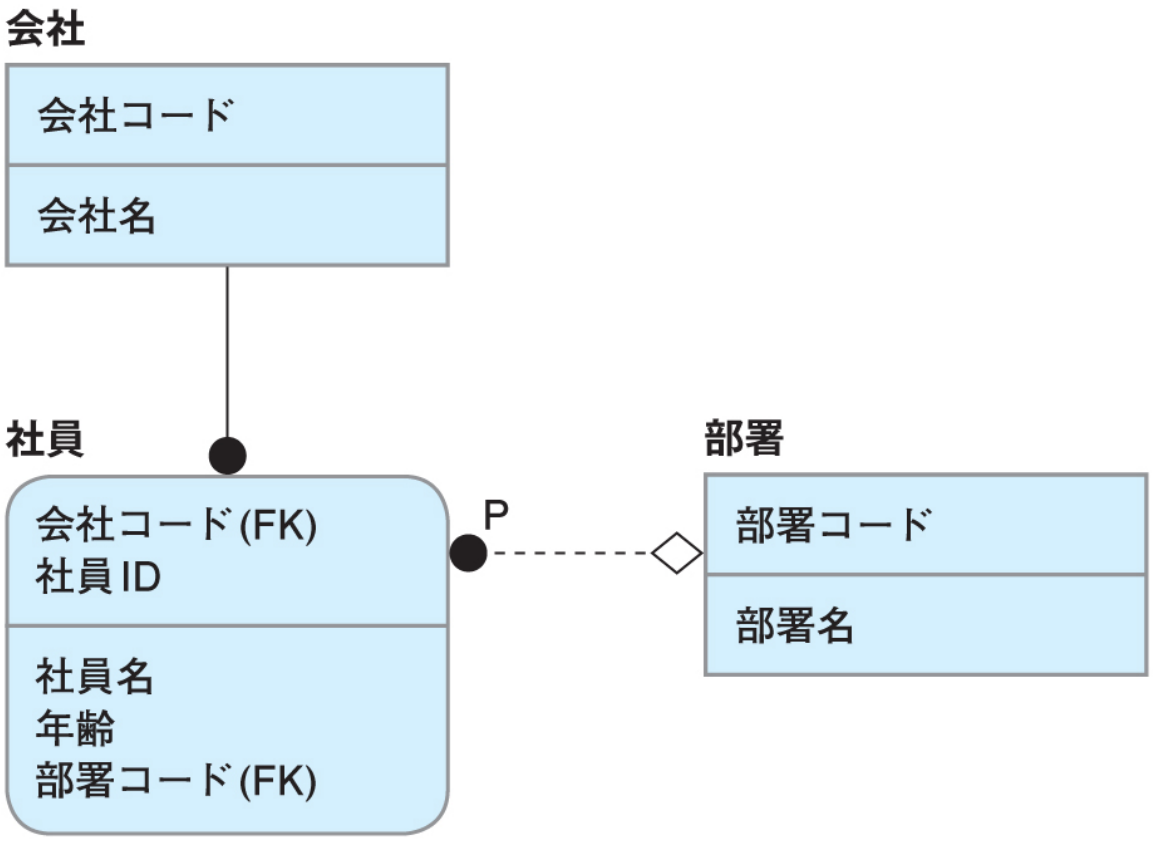


図4-3●IDEF1X によるER 図

IE表記法と比べると、記号の意味が少し抽象的で複雑になった印象を受けると思います。

まず、IE表記法と大きく違うのは、エンティティの表現として、角の尖った四角と角の丸い四角の二つに分かれたことです。

角の尖った四角で表記したエンティティである会社および部署は、**独立エンティティ**であることを意味しています。これは、他のテーブルのデータに依存することなく、データを保持することのできるエンティティです。

一方、角の丸い四角で表記したエンティティである社員は、**従属エンティティ**（または依存エンティティ）であることを意味しています。これは、他のテーブルにデータが存在しなければ、データを保持することのできないエンティティということです。実際、「社員」テーブルにレコードを登録するには、その前に「会社」テーブルにレコードが存在しなければなりません。このエンティティの特徴は、主キーに他テーブルを参照する外部キー（この例で言えば、会社コード）を含むことです。逆に、主キーに外部キーを含んでいなければ独立エンティティなので、この区別は機械的に行なうことができます。

勘どころ 34

独立エンティティと従属エンティティの区別は主キーに外部キーが含まれているかでわかる。

次に、エンティティ間の関連を見ていきましょう。まず、すぐに対応がわかるのは、IE表記法で「鳥の足」で表現されていた、カーディナリティの「多」が黒丸（●）で表現されていることです。会社と社員のように、「0以上の多」である場合はこれで良いのですが、部署と社員のように、「1以上の多」の場合は、●の横に「P」を記述します。このカーディナリティの記述パターンには、図4-4のような種類があります。

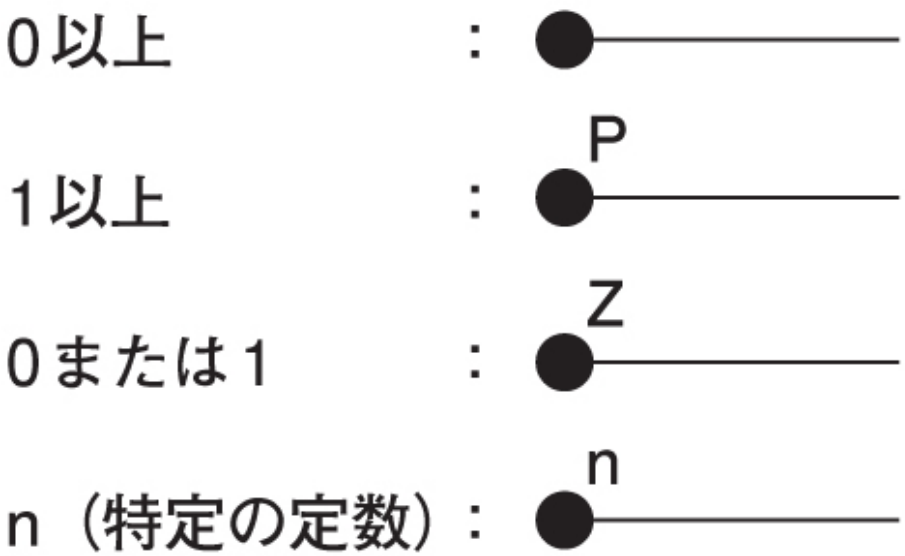


図4-4●カーディナリティの記述パターン

ここで、「会社」テーブルや「部署」テーブルの側にも同じように黒丸が使われていないことについて、疑問を持つ人がいるかもしれません。この理由は、IDEF1Xでは、「1対多」のうち、「1」の側のエンティティ、つまり会社や部署については、黒丸は使わない、と決められているからです。黒丸を必ず「多」の側にだけ付けることで、どちらが「1」でどちらが「多」であるかを一目でわかるようにしているのです。IE表記法の「鳥の足」の代わりだと思えば良いでしょう（ただし、「鳥の足」は「2以上の複数」を意味するので、厳密に対応するわけではありません）。

勘どころ 35

IDEF1Xでは、黒丸（●）がIE表記法の「鳥の足」にほぼ近い意味を持つ。

そのため、「1対多」の「1」の側である会社については、黒丸をつけずにただの棒線を引くことでカーディナリティが「1」であることを表現しています。ここで注意すべきは、「部署」テーブルの側についている「◇」です。これは、同じカーディナリティ「1」であっても、部署コードがNULLでありうることで、社員から部署が一つに決まらない可能性があることを示しています。同じ外部キーでも会社コードは「社員」テーブルの主キーの一部であるため、NULLにはなりえない、という違いがあることに注意してください。

また、会社と社員を結ぶ関連の線が実線であるのに対し、部署と社員を結ぶ線が点線であるのも、この外部キーにNULLを許すか否かによっています。会社と社員のように、社員が会社に必ず属さなければならない場合、これを**依存リレーションシップ**と呼んで、実線で表わします。一方、社員と部署のように、必ずしも社員が部署に所属していなくても良い場合、これを**非依存リレーションシップ**と呼んで、点線で表わします。これは先ほどの依存エンティティと独立エンティティの関係にも対応しています。

このようにIE表記法とIDEF1Xを比較してみると、IE表記法がかなり簡素で直観的に概略を把握しやすいことを目指しているのに対して、IDEF1Xはかなり細かいところまで記述しようとしていることがわかります。細かいところまで記述できるということは、情報量が多く正確であるという利点がありますが、反面、記号の意味に慣れないと理解が難しい、という欠点にもなります［※5］。

本書では、入門書であることから、読者の理解しやすさを考慮して、今後基本的にER図はIE表記法を使って記述します。ただし、IDEF1X特有の注意点などがある場合には、都度触れていきます。

もともとIDEF1Xは米空軍で、メーカーに部品や機器の仕様を伝えるために開発された技法です。その業務の性格から、高い厳密さが要求されたことは、想像にかたくありません。

4-4 「多対多」と関連実体

現実世界に存在するエンティティを、リレーショナルデータベース内のER図として記述しようとすると、多対多の関連を持ったエンティティができることは、珍しいことではありません。具体例を挙げると、図4-5のようなものがあります。

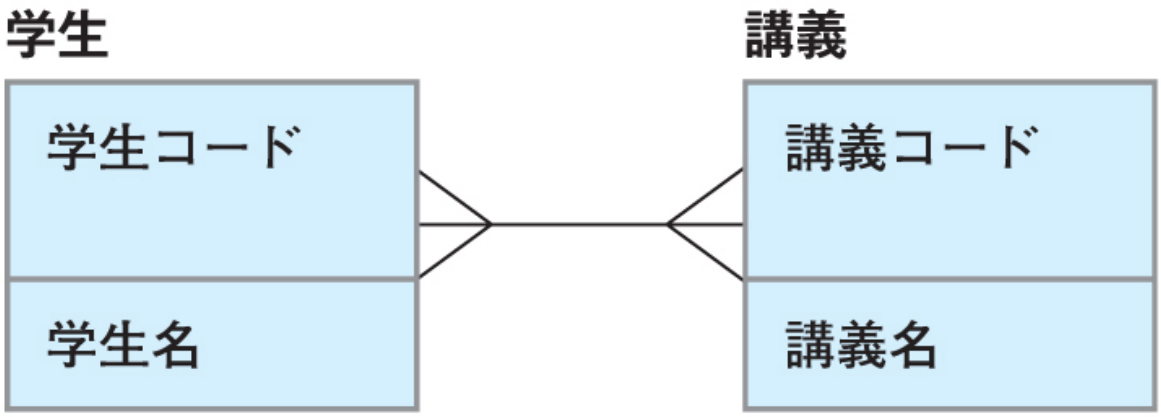


図4-5●多対多

一人の学生は、複数の講義に出席することができます。その点で、エンティティ「学生」から見たエンティティ「講義」のカーディナリティは「多」です。反対に、講義から見ても、一つの講義には複数人の学生が参加できます。こちらも、エンティティ「講義」から見たエンティティ「学生」のカーディナリティは「多」です。つまり、どちらか見ても「多」となるため、「多対多」の関連が成立するのです。

この「多対多」がリレーショナルデータベースの世界において問題となる理由は、両者のエンティティが共通のキーとなる列を保持していないため、両エンティティを結合した情報を得ることができないことです［※6］。もし、この多対多のエンティティ同士を無理矢理にでも関連づけようとすれば、「学生」テーブルに「講義コード」列を持つようにするしかありません。しかし、それでは「学生」テーブルに、一人の学生が複数行含まれるようになりますし、講義を未登録の学生は、「学生」テーブルにも登録できなくなってしまいます。これは、「講義」テーブルに「学生コード」列を追加したとしても同じ問題が発生します。多対多エンティティの問題は、「学生」と「講義」の二つのエンティティだけを前提としている限り、解決不可能なのです。

この問題を解決するための方法が、**関連実体**（associative entity）です。これは、図4-6のように二つのエンティティの間に作られる、第3のエンティティ（受講）です。

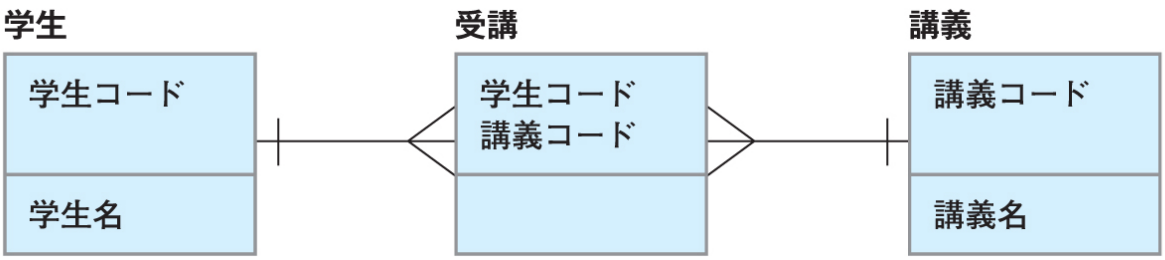


図4-6●関連実体

エンティティ「受講」は、エンティティ「学生」とエンティティ「講義」の主キーを組み合わせたキーを主キーに持ちます。非キー属性を持っても良いのですが、最低限、このように作られる主キーがあれば十分です。

すると、学生と受講の間の関連は「1対多」、講義と受講の間の関連も「1対多」になり、ER図から「多対多」の関係が消去される、という仕掛けです。

この関連実体には、人工的なエンティティという印象を受ける人も少なくないでしょう。実際、リレーショナルデータベース側の都合によって導入されるエンティティですから、このエンティティが人工的なのはそのとおりです。特に、現実世界を見回しても実体として存在しているわけではないものを、エンティティとして配置することに気持ち悪さを感じるかもしれません。

しかし、関連エンティティの作り方は上記のように単純ですし、実際にリレーショナルデータベースを使ったシステムを作るうえでは必ずといっていいほど使用する技術なので、ここできちんと理解するようにしましょう。

演習問題

演習 4-1 ER図

この演習で利用するテーブルは第3章の演習問題の解答を含むため、未解答の方はまず先に演習3-3（122ページ）に取り組んでください。

さて、この演習では、演習3-3で正規化した以下の五つのテーブルについて考えます。

支社

支社コード	支社名
001	東京
002	大阪

支店

支社コード	支店コード	支店名
001	01	渋谷
001	02	八重洲
002	01	堺
002	02	豊中

支店商品

支社コード	支店コード	商品コード
001	01	001
001	01	002
001	01	003
001	02	002
001	02	003
001	02	004
001	02	005

001	02	006
002	01	001
002	01	002
002	02	007
002	02	008

商品

商品コード	商品名	商品分類コード
001	石鹸	C1
002	タオル	C1
003	ハブラシ	C1
004	コップ	C1
005	箸	C2
006	スプーン	C2
007	雑誌	C3
008	爪切り	C4

商品分類

商品分類コード	分類名
C1	水洗用品
C2	食器
C3	書籍
C4	日用雑貨

これら五つのテーブルを、IE表記法およびIDEF1XによってER図を記述してください。なお、関連を表わすのに必要ない属性については適宜省略してかまいません。

演習 4-2 関連エンティティ

上記五つのテーブルの中には、関連エンティティが含まれています。それはどれでしょうか？

演習 4-3 多対多の関連

現実世界に存在するエンティティについて、多対多の関連があるものを三つ挙げてください。

また、それに対して多対多の関連を解消するための関連実体を考えてください。

SQLで結合を行なおうにも、結合キーを指定できないため、直積結合以外を選択できません。

第5章

論理設計とパフォーマンス
～正規化の欠点と非正規化

正規化はデータベース設計における基本です。しかし、正規化されたテーブルは、往々にして現場のDBエンジニアからは評判が良くありません。それは、正規化がパフォーマンス上の深刻な問題を引き起こしてしまうからです。本章では、正規化の功罪と対処法について考えます。

学習のポイント

- 正規化することは、データベースにおけるテーブル設計の基礎です。しかし、厳格に正規化すると、時として正規化の欠点が現われることがあります。その最大の欠点は、SQLのパフォーマンス問題です。
- 正規化がSQLのパフォーマンス問題を引き起こすのは、正規化するとSQL文の中で結合（join）が必要になるからです。結合は、SQLの処理の中でも高コストなため、多用するとSQLの速度が悪化します。
- この問題に対処する方法は二つあります。一つは、SQL文そのものに対するチューニング。もう一つは、非正規化です。
- 正規化以外にも、SQLのパフォーマンスを悪化させるテーブル設計があります。それもまた正規化と同様、過度に冗長性を排した結果生じるものです。相した場合には、あえて冗長なデータを保持する設計が有効になります。

前章までで、リレーショナルデータベースにおける論理設計の基本を学習しました。その中核となる概念が正規化でした。ここでもう一度、なぜ正規化する必要があるか、そのメリットは何か、というところをまとめておけば、それはひとえに、データの整合性を保持するためでした。データベースというのは、ユーザーの利用するデータを一元的に集約して保管する倉庫です。したがって、ここに含まれるデータが間違っている、ということは絶対に許されません。正規化は、この厳しい要求に応えるためデータ整合性を厳密に保つ方法論です。

しかし一方、正規化がもたらすのは良いことばかりではありません。世の中の大抵のことに表と裏があるように、正規化にも副作用があります。それもかなり強いものが。それは、正規化されたテーブル群に対するSQLが非常に遅くなってしまい、しばしばシステムとして実用にたえないぐらい、パフォーマンス劣化を招いてしまうことです。

本章では、そうした正規化の負の側面を明らかにするとともに、それに対して現場のエンジニアはどのような対処を取るべきか、という点を学習していきます。

正規化とSQL（検索）

まずは、第3正規化した状態のテーブル群から始めましょう。

会社

会社コード	会社名
C0001	A 商事
C0002	B 化学
C0003	C 建設

社員

会社コード	社員 ID	社員名	年齢	部署コード
C0001	000A	加藤	40	D01
C0001	000B	藤本	32	D02
C0001	001F	三島	50	D03
C0002	000A	斉藤	47	D03
C0002	009F	田島	25	D01
C0002	010A	渋谷	33	D04

部署

部署コード	部署名
D01	開発
D02	人事
D03	営業
D04	総務

図5-1●第3 正規形のテーブル群

会社、社員、部署、という三つのテーブル（エンティティ）に分かれた、第3正規形のテーブル群です。どのテーブルにも、部分関数従属も推移的関数従属も存在しないことを、もう一度確認してください。

内部結合を使うケース

さてある日、みなさんの席へ上司がやってきて

「田島さんが今勤めている会社を知りたいのだが……」

と言われたとしましょう。これを見つけるSQL文を考えます。SQLは、以下のようなステップで考えることになります。

1. 「社員名」を検索条件としてデータ検索することから、「社員」テーブルが検索対象になります。ここまでは特に問題ありません。
2. しかし、「社員」テーブルからは会社名がわかりません。「社員」テーブルに含まれている列は「会社コード」だけだからです。
3. 「会社名」列を持っているのは、「会社」テーブルだけです。したがって、「会社」テーブルも検索対象に含める必要があります。
4. すると「社員」テーブルと「会社」テーブルの二つのテーブルを検索対象とすることになり、両者を結合する必要が出てきます。このときの結合キーが「会社コード」になります。

以上のようなステップから、作られるSQLは次のようなものになります。

```
SELECT 会社.会社名,
       社員.社員名
  FROM 社員 INNER JOIN 会社
               ON 社員.会社コード = 会社.会社コード
 WHERE 社員.社員名 = '田島';
```

結果

```
会社名   社員名
----   ----
B化学    田島
```

さて、無事結果を得たみなさんのところに、もう一度上司がやってきて言いました。

「ごめんごめん、やっぱり田島さんについて、会社だけじゃなく部署名も知りたいんだ」

「まったくもう、だったら最初から言ってくれよ」とブツブツ言いながらも、上司の命令です。再びSQLを考えなければなりません。今度は、「部署名」の列を持っているのは、「部署」テーブルだけです。したがって、上記のSQLにさらに「部署」テーブルも追加する必要があります。このとき、「社員」テーブルとの結合キーになるのは、「部署コード」です。

■内部結合

```
SELECT 会社.会社名,
       社員.社員名,
       部署.部署名
  FROM 社員 INNER JOIN 会社
                  ON 社員.会社コード = 会社.会社コード
               INNER JOIN 部署
                  ON 社員.部署コード = 部署.部署コード
 WHERE 社員.社員名 = '田島';
```

結果

```
会社名   社員名   部署名
----   ----   ----
B化学    田島     開発
```

この二つの例からわかることは、正規化されたテーブルは、単独のテーブルでは必ずしもユーザーが欲するすべての情報をカバーできない、ということです。これは、正規化が、ある意味で、情報を複数のテーブルに分散させる行為だからです。

外部結合を使うケース

さてその翌日。またまた上司がみなさんのところにやってきて、面倒なことを言い出しました。

「会社ごとに社員が何人いるか集計してもらえないかな」

会社の一覧を得るには、「会社」テーブル、社員の人数をカウントするには「社員」テーブルへのアクセスが必要です。この点で、先ほどと同じく結合を利用することになるのですが、今度少し違うのは、会社については「すべての会社」を網羅しなければならないことです。

というのも、「社員」テーブルにはA商事とB化学の社員については登録されていますが、C建設の社員は未登録の状態です。したがって、C建設については「0 人」という結果を得たいのですが、内部結合を使うとC建設の情報は漏れてしまいます。このような場合は、外部結合を使うことで、片方のテーブルに存在しない情報であっても保存して得ることができます。

■外部結合

```
SELECT 会社.会社コード,
       COUNT(社員.社員名) AS 社員数
  FROM 会社 LEFT OUTER JOIN 社員
               ON 社員.会社コード = 会社.会社コード
 GROUP BY 会社.会社コード;
```

結果

```
会社コード   社員数
-------   ----
C0001        3
C0002        3
C0003        0
```

このように、正規化されたテーブル群であっても、SQL文で結合を使うことによって、得たい結果を自由に得ることができます。これは、正規化が無損失分解、つまり情報を一切失わない操作であるからです。しかし、他方でSQLにおける結合は非常にコストの高い操作であり、結合するテーブル数、およびテーブルのレコード数が増えれば増えるほど処理時間がかかります。正規化することでシステムパフォーマンスが劣化する原因の多くが、このSQLの結合操作にあるのです。

非正規化による解決

一方、上記でサンプルに取り上げた「上司からの依頼」を、結合によらず解く方法もあります。それはほかでもない、正規化を捨てること（非正規化）です。第2正規化する前の「社員」テーブルを、もう一度引っ張り出しましょう。

■第2 正規形以前の「社員」テーブル（非正規化）

社員

会社コード	会社名	社員ID	社員名	年齢	部署コード	部署名
C0001	A 商事	000A	加藤	40	D01	開発
C0001	A 商事	000B	藤本	32	D02	人事
C0001	A 商事	001F	三島	50	D03	営業
C0002	B 化学	000A	斉藤	47	D03	営業
C0002	B 化学	009F	田島	25	D01	開発
C0002	B 化学	010A	渋谷	33	D04	総務

このテーブルを使えば、たとえば、直前に上司から受けた依頼「田島さんの会社と所属の部署名」を得るに当たって、結合を使わずに済みます。

■非正規化（第2 正規形以前）の場合の検索SQL

```
SELECT 会社名,
       社員名,
       部署名
```

```
 FROM 社員
WHERE 社員名 = '田島';
```

正規化したテーブル群におけるSQLと比較すると、非常にすっきりシンプルで、簡単なSQLになることがわかります。しかも見た目が単純である以上に重要なポイントは、結合を利用しないためパフォーマンスが良い、というところです。

勘どころ 36

非正規化テーブルならば、SQLで結合を使わずに済む。

正規化とSQL（更新）

一方、正規化（非正規化）と更新処理の関係はどのようなものになるのでしょうか？ 実は、このケースにおいては、正規化のほうに軍配が上がります。たとえば、A商事が企業買収の対象になって、E物産という会社に吸収された場合を考えます。このとき、第2正規化以前の「社員」テーブルでは、A商事の会社名のレコードすべてをE物産に更新する必要があります。実際のシステムでは、社員の人数分のレコード数ですから、相当な数にのぼるでしょう。かつ、会社の社員数によって更新処理のパフォーマンスに大きなバラつきが生じます。

更新のSQL文は、以下のようになります。

■非正規化（第2 正規形以前）の場合の更新SQL

```
UPDATE 社員
   SET 会社名= 'E物産'
 WHERE 会社コード= 'C0001';
```

　これに対し、第3正規化された状態で同じ更新処理を行なうことを考えましょう。この場合、会社の情報はすでに「会社」テーブルへ切り出されていますので、「会社」テーブルが更新の対象になります。かつ、ここが重要なポイントですが、「会社」テーブルでは、一つの会社に対して**常に1レコード**が対応するため、更新処理のコストは常に低く、かつ一定です。

　こちらの更新のSQL文は、以下のようになります。形としては先ほどのUP DATEの「社員」テーブルが「会社」テーブルに変わっただけですが、その影響は非常に大きなものがあります。

■正規化した場合（第3 正規形）の更新SQL

```
UPDATE 会社
   SET 会社名= 'E物産'
 WHERE 会社コード= 'C0001';
```

正規化と非正規化、どちらが正解なのか？

　この問題に対する回答は、エンジニアや理論家によって意見の分かれるところです。正規化は必須であり、いかなる理由があろうとも非正規化するべきではない、

という原理主義的な立場から、最初から非正規化を論理設計の中に織り込む「現実主義的」な立場まで、正規化に対する態度は人によって様々です。

それだけ、正規化と検索SQLのパフォーマンスは強い**トレードオフ**の関係にある、ということなのです。厳しく正規化すればパフォーマンスが悪化し、パフォーマンスを求めて非正規化すれば、データ不整合が発生しやすくなります。データ整合性とパフォーマンスのトレードオフ関係を図示すると、図5-2のようになります。

正規化の次数が低いほど検索SQLのパフォーマンスは良いのですが、データ整合性は低く、正規化していくほどパフォーマンスが低下する代わりにデータ整合性が高くなります。

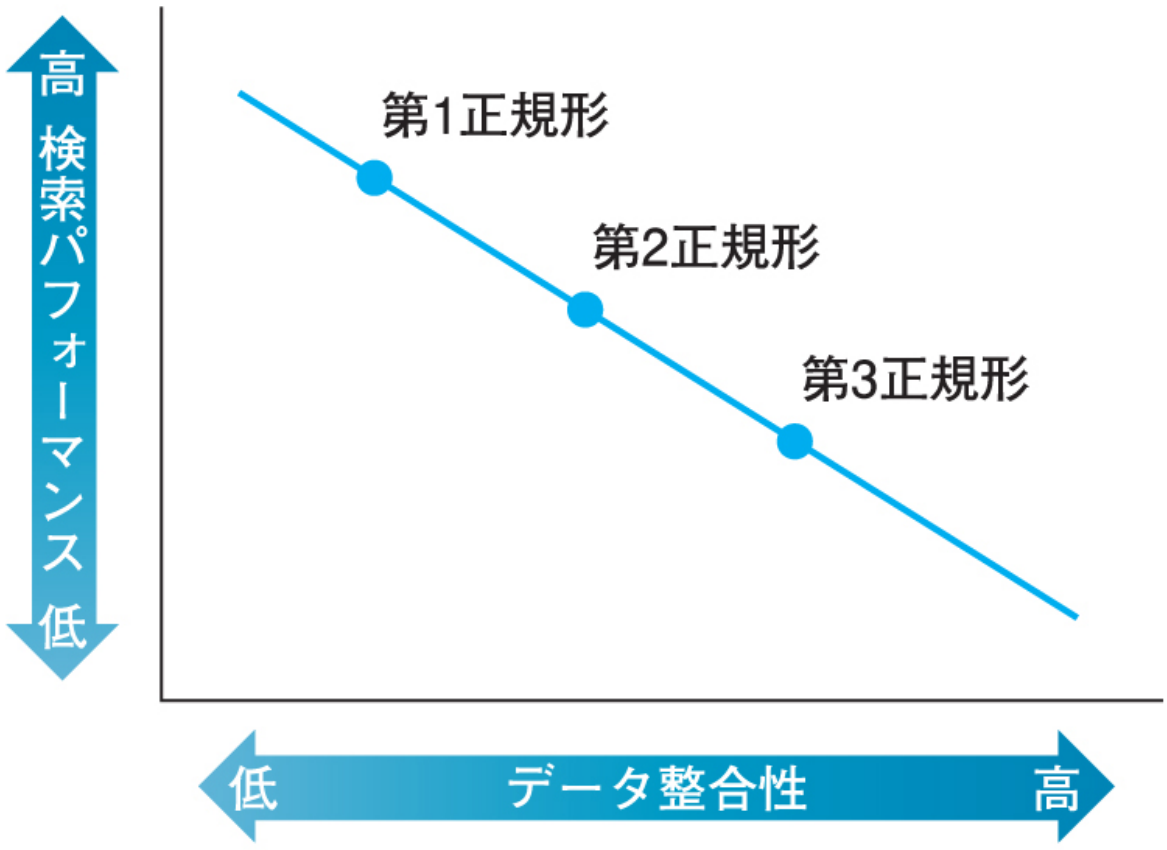


図5-2●データ整合性とパフォーマンスのトレードオフ関係

なお、非正規化について著者個人が支持する立場は、**原則として非正規化は許さない**、というものです。この点をはっきりと述べたクリス・デイトの言葉を紹介しましょう。

> 本節を次の**論理的**な意見で締めくくりたいと考えているが、それは今まで聞いたことがないものかもしれない。それは、「**非正規化**」**はあくまでも最後の手段である**という姿勢でのぞむ、というものだ。要するに、十分に正規化された設計をあきらめてもよいのは、パフォーマンスを向上させるためのその他すべての戦略が要件を満たさない場合だけである。[※1]

「非正規化は最後の手段」――逆に言うと、最初は必ず正規化するのだ、ということです。正規化の次数は高ければ高いほど良いことだと考えてください。第1正規形よりは第2正規形、第2正規形よりは第3正規形のほうが、設計という観点からは望ましいのです。

読者のみなさんは、今後リレーショナルデータベースの論理設計に携わることも多いでしょう。そのときは、上述のトレードオフの話を思い出してください。そして、他の手段によってパフォーマンス向上が図れないかを、最後の最後まで検討してください。非正規化は、いよいよ切羽詰まったときに取る、最後の手段（しかも劇薬）なのです。

C.J.Date『データベース実践講義』（オライリー・ジャパン、2006）p.164。文字の強調は原著者によるもの。

このように、正規化はしばしばSQLのパフォーマンスを悪化させる論理設計となります。この問題は重要であるため、もう少し違うサンプルと角度から「冗長性を排した結果、パフォーマンスが悪化する」というパターンについて解説しておきたいと思います。

サマリデータの冗長性とパフォーマンス

正規化、すなわち冗長性排除によって引き起こされる性能問題には、SQLの構文という観点から見ると、大きく2種類のパターンに分けられます。一つ目が、**サマリデータの冗長性排除**によるパターン。もう一つが、**選択条件の冗長性排除**によるパターンです。両方とも、必ず1対多の関連を持つ二つのテーブル間で発生します。そこで、新たなサンプルテーブルとして、図5-3のようなものを使って考えていきましょう。

この二つのテーブルは、たとえばお歳暮とか、結婚式の引き出物などの注文受付を管理するためのテーブルだと考えてください。「受注」テーブルにおいて1件ずつの注文を管理し、その注文に含まれる個々の商品を、「受注明細」テーブルで管理し

ます。したがって、「受注」テーブルと「受注明細」テーブルは、1対多の関連を持つことになります（そして、「受注」テーブルは関連エンティティです）。

　このテーブル構成は、かなり簡略化して記述しています。実務で利用するにはもっとたくさんの列を保持しなければなりませんし、実際には「商品名」や「注文者名義」をこれらのテーブルに持つよりも、「注文者ID」や「商品コード」を保持するほうが現実的です。しかし、そうした細部は、本節で考える問題とは関係がないことに注意してください。

受注

受注 ID	受注日	注文者名義
0001	2012-01-05	岡野　徹
0002	2012-01-05	浜田　健一
0003	2012-01-06	石井　慶子
0004	2012-01-07	若山　みどり
0005	2012-01-07	庄野　弘一
0006	2012-01-11	若山　みどり
0007	2012-01-12	岡野　徹

受注明細

受注 ID	受注明細連番	商品名
0001	1	マカロン
0001	2	紅茶
0001	3	オリーブオイル
0001	4	チョコ詰め合わせ
0002	1	紅茶
0002	2	日本茶
0002	3	ティーポット
0003	1	米
0004	1	アイロン
0004	2	ネクタイ
0005	1	チョコ詰め合わせ
0005	2	紅茶
0005	3	クッキーセット
0006	1	牛肉
0006	2	鍋セット
0007	1	米

図5-3●1 対多の関連を持つ二つのテーブル

再び、結合はパフォーマンスの敵

まず、この二つのテーブルが、第3正規形を満たしていることを確認しましょう。どちらのテーブルにも、部分関数従属および推移的関数従属は存在しません。したがって、十分に正規化された状態にあります。

さて、それではこのテーブルを使って、次のような処理を実現するSQLを考えてください。

問題	**受注日ごとに何個の商品が注文されているかを調べよ。**

これは、実際の業務においても、よくあるタイプの処理です。1日の注文につき何個の商品が注文されているかというのは、マーケティングをするためにも在庫調整を行なううえでも重要な情報です。

今、「受注日」列は「受注」テーブルにあります。一方、商品の数は、「受注明細」から数える必要があります。したがって、この問題を解くSQLは、二つのテーブルを結合した後に、「受注日」をGROUP BYのキーに使った集約を行なうものになります。

解答

```
SELECT 受注.受注日,
       COUNT(*) AS 商品数
  FROM 受注 INNER JOIN 受注明細
               ON 受注.受注ID = 受注明細.受注ID
 GROUP BY 受注.受注日;
```

結果

```
受注日          商品数
----------   ------
2012-01-05        7
2012-01-06        1
2012-01-07        5
2012-01-11        2
2012-01-12        1
```

これで意図したとおり、受注日ごとの商品の数が得られました。機能的な観点だけから見たら、このSQLは、問題に対する解としては完璧です。しかし、別の観点から見ると、このSQLには大きな問題があります。その観点というのが、パフォーマンスです。

今、このSQLは「受注」と「受注明細」という二つのテーブルを結合しています。どちらのテーブルも、今はサンプル的に用意した数行のレコードしか保持していませんが、実際の業務では、非常に多くのレコードが格納されます。特に「受注明細」テーブルのほうは、「受注件数☒n」件のレコード数を格納することになるため、レコード件数は膨大なものになります（nは注文1件当たりの平均商品数）。

したがって、このSQLは、現実には非常に大きなテーブル同士を結合する高コストな処理になり、パフォーマンス上の問題を引き起こす可能性が高いのです。

結合しないSQLを作るためのテーブル設計

では、先ほどの「問題」を、結合を利用せずに解くにはどうすれば良いでしょうか？

この問いに答えるには、今のままのテーブル構成では不可能です。次のように「受注」テーブルに新たな列を追加します。

■テーブル構成の変更

受注

受注 ID	受注日	注文者名義	商品数
0001	2012-01-05	岡野　徹	7
0002	2012-01-05	浜田　健一	7
0003	2012-01-06	石井　慶子	1
0004	2012-01-07	若山　みどり	5
0005	2012-01-07	庄野　弘一	5
0006	2012-01-11	若山　みどり	2
0007	2012-01-12	岡野　徹	1

新たな列「商品数」を追加

このように、「受注」テーブルにそもそも「商品数」の列が存在していれば、「受注明細」テーブルとの結合など必要ありません。以下のように「受注」テーブルだけを使ってSQLを記述できます。

■結合なしのSQL

```
SELECT DISTINCT 受注日,
                商品数
  FROM 受注;
```

結果

先ほどと同じ（151 ページ 結果 参照）。

このように、「注文」というエンティティに「商品数」というサマリデータ（集計データ）を持つことは、冗長性を持たせた設計の一種と言えます。事実、「受注」テーブルへのこのような変更は、非正規化でもあるからです。以下のように、「商品数」列は、「受注日」という非キー列に従属しています。

{受注日} → {商品数}

このような推移的関数従属があることから、この「受注」テーブルは第3正規形ではなくなります。したがって更新時における問題が発生することになるのですが、そのトレードオフとして検索処理が非常に簡単でハイパフォーマンスなものにできるのです。

勘どころ 37

サマリデータを冗長に保持すると正規形に違反するが、検索を高速化できる。

選択条件の冗長性とパフォーマンス

次に見るパターンは、選択条件の冗長性がパフォーマンスに与える影響についてです。再び、オリジナルの「受注」および「受注明細」テーブル（図5-3）を使って、以下の問いを考えましょう。

問題	**受注日が2012-01-06 ～ 2012-01-07の期間に注文された商品の一覧を出力せよ。**

受注日が2012-01-06と2012-01-07の二日間に受けた注文は、「受注ID」が「0003」「0004」「0005」の三つです。この三つの注文に含まれる商品は、「商品明細」テーブルからわかります。つまり、この問題も、「受注」テーブルと「受注明細」テーブルを結合しなければ解くことができない、ということです。解となるSQLは次のとおりです。

解答

```
SELECT 受注.受注ID,
       受注明細.商品名
  FROM 受注 INNER JOIN 受注明細
               ON 受注.受注ID = 受注明細.受注ID
 WHERE 受注.受注日 BETWEEN '2012-01-06' AND '2012-01-07';
```

結果

```
受注ID   商品名
-----   ------------
0003    米
0004    アイロン
0004    ネクタイ
0005    チョコ詰め合わせ
0005    紅茶
0005    クッキーセット
```

選択条件が冗長でないことの問題点

このSQLは、機能的には問題ありません。正しい結果を得られるものです。しかし、性能の観点から見ると、結合を必要とする点でコストの高いものになっています。この問題を解消するために、次のようにテーブル構成を変更しましょう。今度は、「受注明細」テーブルに新たな列を追加します。

■テーブル構成の変更

受注明細

新たな列「受注日」を追加

受注 ID	受注明細連番	商品名	受注日
0001	1	マカロン	2012-01-05
0001	2	紅茶	2012-01-05
0001	3	オリーブオイル	2012-01-05
0001	4	チョコ詰め合わせ	2012-01-05
0002	1	紅茶	2012-01-05
0002	2	日本茶	2012-01-05
0002	3	ティーポット	2012-01-05
0003	1	米	2012-01-06
0004	1	アイロン	2012-01-07
0004	2	ネクタイ	2012-01-07
0005	1	チョコ詰め合わせ	2012-01-07
0005	2	紅茶	2012-01-07
0005	3	クッキーセット	2012-01-07
0006	1	牛肉	2012-01-11
0006	2	鍋セット	2012-01-11
0007	1	米	2012-01-12

このように、「受注明細」テーブルに「受注日」列を追加することで、「受注」テーブルを見る必要はなくなります。SQLは次のようにシンプルなものになります。

■結合なしのSQL

```
SELECT 受注ID,
       商品名
  FROM 受注明細
 WHERE 受注日 BETWEEN '2012-01-06' AND '2012-01-07';
```

結果

先ほどと同じ（154 ページ 結果 参照）。

このように、列を一つ追加するだけでSQLから結合をなくすことができます。また、仮にまだ結合が必要なケースであっても、「受注明細」テーブルの「受注日」列を選択条件に使えるようなケースであれば、I/Oコストを大きく削減することが可能になります。

選択条件を冗長にすると第2正規形ではなくなる

変更後「受注明細」テーブルは、先ほどのサマリデータを追加した場合の「受注」テーブルと同じく、正規形に違反します。というのも、主キーの一部である「受注ID」から非キー列である「受注日」への部分関数従属が生まれてしまうからです。

{受注I D} → {受注日}

したがって、変更後の「受注明細」テーブルは、第2正規形ではなくなります。

勘どころ 38

選択条件を冗長に保持すると正規形に違反するが、検索を高速化できる。

いかがでしょう。冗長性を排除するという原則を突き詰めて正規化を推し進めていくと、検索のSQLにいかにしわ寄せが来るか、という点を理解していただけたでしょうか？

本章では、かなりくどいぐらい正規形の性能面での難点をクローズアップしてきたので、読者の中には著者が「非正規化を勧めている」と思った方もいるかもしれません。しかし、それはまったくの誤解です。**正規化は、可能な限り高次にすることが大原則**です。この点は、何度強調しても強調したりないぐらいです。

しかしそれでもなお、実務における論理設計では、性能のために非正規化が必要になるときがあります。それは、いかにSQLのパフォーマンスチューニングを行なおうとも、テーブル構成が正規化された状態では限界があるからです。本書の前半でも述べたとおり、データ構造がプログラムのコードを決定するのであって、その逆ではないのです。

しかし、実際の開発現場において非正規化まで考慮された論理設計が行なわれることは、まれとは言わないまでもそれほど多くありません。それは、正規化が教科書的／機械的に実施できるのに対して、非正規化はそうではないからです。非正規化は、様々なトレードオフを考慮しながら慎重に実施する必要のある、難しい仕事です。人々は一般にそのような仕事を嫌がります。しかし、それこそがエンジニアの本務だと著者は考えています。

次節では、非正規化を実施する際に考えるべきトレードオフを、もう少し掘り下げて解説したいと思います。

5-3 冗長性とパフォーマンスのトレードオフ

すでに、非正規化が更新不整合のリスクを増やすことについては解説したとおりです。ここでは、それ以外のリスクを整理しておきましょう。大きく三つに分類できます。

リスク1 非正規化は、検索のパフォーマンスは向上させるが更新のパフォーマンスを低下させる。
リスク2 データのリアルタイム性（鮮度）を低下させる。
リスク3 後続の工程で設計変更すると、手戻りが大きい。

更新時のパフォーマンス

たとえば、「商品数」というサマリデータを追加した「受注」テーブルを考えましょう。このデータをテーブルに持つためには、当然のことながら、「受注」テーブルに注文データを登録する際に商品数を計算しておく必要があります。それだけではなく、注文内容というのは一度登録されたら確定ではなく、しばらくの期間は変更が可能なものです。したがって、商品数にも増減がありえます。ということは、定期的に「受

注」テーブルへ「商品数」列の価を反映する更新処理が必要になる、ということです。こうした更新処理の負荷を考慮しなければなりません。

データのリアルタイム性

上記から導かれる結論として、データの**リアルタイム性**という問題が発生することもわかります。「商品数」が受注受付後に変更されるというのなら、その変更はどのようなタイミングで反映すべきなのでしょうか？　1日1回、夜間に反映すれば良いのか。それとも、30分に1回、たとえ日中でも反映しなければならないのか。はたまた商品数に変更が生じたら即座に反映しなければならないのか……。

こうした要件は、「商品数」のデータにどの程度の最新性が求められるか、という業務要件とセットで考える必要があります。当然のことですが、反映周期が短ければ短いほど、システムへかかる負荷は高くなり、性能問題も起きやすくなります。しかしユーザーとしては、短ければ短いほど嬉しいに決まっており、両者のバランスが取れる平衡点を見つけ出さなければなりません。

改修コストの大きさ

最後の問題が、システムの改修コストの大きさです。データモデルの変更は、コードベースの修正に比べて、非常に改修コストが大きくなります。これは、データのフォーマットがプログラムを決めるDOAの大原則です。したがって、性能試験をやったとこ

ろ性能が出なかったので、やっぱりテーブル構成を変更したい、と言い出しても、アプリケーションを大きく改修しなければならない、という理由でなかなか受け入れられなかったりします。

もっとも、それは正規化した状態から非正規化するときでも、非正規化の状態から正規化するときでも同じなのですが、ともあれ、論理設計をする際には「システムの品質は（ひいては開発が成功するかどうかは）今ここで決まる！」という気概を持って臨む必要があります。かつ、論理設計を担当する人間は、正規形の理論を理解しているだけでなく、それによって生じる様々なトレードオフを知り尽くしたうえで、あらゆる要件を同時に満たせる平衡点を探し出せる能力が必要とされるのです。

これは非常に高い要求水準だと思うかもしれません。そう、実際著者は高い条件を要求しています。論理設計を行なう人間は、開発チームのエースでなければなりませんし、もしみなさんがその担当であるとしたら、エースになるべく努力していただきたいと思うのです。

そのとき重要なことは、実は論理設計を行なうには、論理層、すなわち概念スキーマだけ考えていてはダメだ、ということです。というのも、パフォーマンスの問題について考えるときは、どうしてもファイルやハードウェアといった物理層に踏み込まざるをえないからです。

正規理論を熟知し、美しい論理モデルを考えられる人が、物理レベルについてはまったくの無知、ということもしばしばあります。しかし本章で見たように、本当の論理設計は、論理と物理のトレードオフを理解して初めて可能になるのです。

勘どころ 39

残念なことに、論理設計には物理設計の知識が必要である。

今すぐに論理層と物理層に精通しろ、というのは無理かもしれません。しかし、少なくともその理由を、本章を読んで理解いただけたなら幸いです。

演習問題

演習 5-1 正規化されたテーブルに対するSQL

この演習で利用するテーブルは第3章の演習問題の解答を含むため、未解答の方はまず先に演習3-3（122ページ）に取り組んでください。

さて、この演習では、演習3-3で正規化した以下の五つのテーブルについて考えます。

支社

支社コード	支社名
001	東京
002	大阪

支店

支社コード	支店コード	支店名
001	01	渋谷
001	02	八重洲
002	01	堺
002	02	豊中

支店商品

支社コード	支店コード	商品コード
001	01	001
001	01	002
001	01	003
001	02	002
001	02	003
001	02	004
001	02	005
001	02	006
002	01	001
002	01	002
002	02	007
002	02	008

商品

商品コード	商品名	商品分類コード
001	石鹸	C1
002	タオル	C1
003	ハブラシ	C1
004	コップ	C1
005	箸	C2
006	スプーン	C2
007	雑誌	C3
008	爪切り	C4

商品分類

商品分類コード	分類名
C1	水洗用品
C2	食器
C3	書籍
C4	日用雑貨

上記五つのテーブルから、以下の要件を満たす結果を得るためのSQL文を考えてください。なお、テーブル名、列名、データの値は上記のテーブルのものを使用するものとします。

SQL文1 **商品分類ごとの商品数。結果には分類名を含むものとする。**

SQL文2 **支社／支店別の取り扱い商品の一覧。結果には支社名、支店名、商品名を含むものとする。**

SQL文3 **最も取り扱い商品数が多い支店の支店コードと商品数。**

演習 5-2 非正規化によるSQLチューニング

演習5-1の解答のSQL文に対して、パフォーマンス向上を実施します。本章で学んだ非正規化を含むテーブル構成の変更による方法を考えてください。

第 6 章

データベースとパフォーマンス

データベース設計というと、論理設計にのみ目を奪われがちですが、それでは十分なパフォーマンスを確保するには不十分です。データベースにおけるパフォーマンスの設計は、DBMS の内部アーキテクチャに踏み込む物理レベルの知識が必要になります。本章では、インデックスと統計情報という二つの観点を通して、ハイパフォーマンスを実現する方法を考えます。

学習のポイント

- データベースのパフォーマンスを決める主な要因は、ディスクI/Oの分散（RAI D）、SQLにおける結合（正規化）、そしてインデックスと統計情報です。
- インデックスにはいくつかの種類がありますが、まずは B-treeインデックスを覚えておくことで多くのケースに対処できます。
- インデックスはパフォーマンス向上に有効な道具ですが、正しい使い方をしなければ効果を発揮しません。
- 統計情報はDBMSにとっての地図情報です。これが最新でなければ、DBMS は最短のアクセスパスを選ぶことができません。
- しかしDBMSも人間の作ったものなので、最新の地図を見ても道を間違えることがあります。

これまで本書では、物理設計（第2章）および非正規化（第5章）について解説する中で、データベースのパフォーマンスを良好に保つための方針についても触れてきました。本章では、データベースのパフォーマンス設計という観点において重要なポイントを、もう二つ紹介したいと思います。それが、インデックスと統計情報です。

インデックス

インデックスは、SQLチューニングの手段として非常にポピュラーで、これを利用しないシステムはない、というぐらいよく使います（図6-1）。

インデックスとは、プログラミング言語的な表現をすると（x, α）という形式の配列です。ここで、xはキー値、αはそれに結びつく情報——実データか、あるいはそれへのポインタ——を意味します。実際には、データベースにおいてαはデータへのポインタであることが多いです。巻末についている本書のインデックスも、キーとなる単語と、ページ番号（ポインタ）の配列になっています。

インデックスについては、第2章でも少し触れました。そこでは、「DBMS内にテーブルとは独立に保持されるオブジェクトである」と述べましたが、本章ではその内部構造やロジックにも踏み込んで、どのようなインデックス設計が望ましいかを考えていきます。

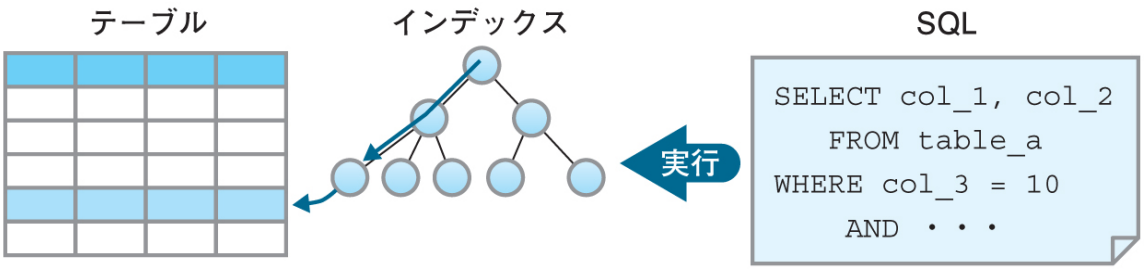


図6-1●SQL 文は、インデックスをたどることで、テーブルの特定レコードを狙い撃ちでアクセスできる

統計情報

統計情報が重要な理由は、これがSQLのアクセスパスを決める最大の要因だからです。DBMSは、ユーザーからSQLを受け取ると、どのような経路（パス）でデータを探しに行くのが最も効率的か、自分で判断します。というのも、SQLには、「こんなデータが欲しい」という条件は記述されていても、「こうやってデータを取りに行け」という方法までは書かれていません。その方法を多くの選択肢の中から見つけ出すのは、DBMSの仕事なのです。クリス・デイトはこの特徴を次のように表現しています。

> SQLのような関係型の言語は非手続き型の言語と呼ばれるが，これはユーザは「いかに」ではなく「何を」を指定する—つまり，獲得するための手続きを指定することなしにユーザは何が欲しいかをいう—からである。[※1]

これはちょうど、車のカーナビと同じです。ドライバーは「ここに行きたい」という目的地だけを入力して、どんな道を通るのが良いかはカーナビに任せています。カーナビは地図や渋滞などの情報をもとに、最短と判断した一つの経路を選びます。データベースの場合も同様で、統計情報とはいわば、SQLの最適なアクセスパスを見つけるために必要な地図情報なのです（図6-2）。かつては、ルールベースといって、ある程度エンジニアがデータアクセスの経路を選択する方法が主流でしたが、最近のDBMSではコストベースといって、DBMSに経路選択を一任するアーキテクチャが主流になっています。

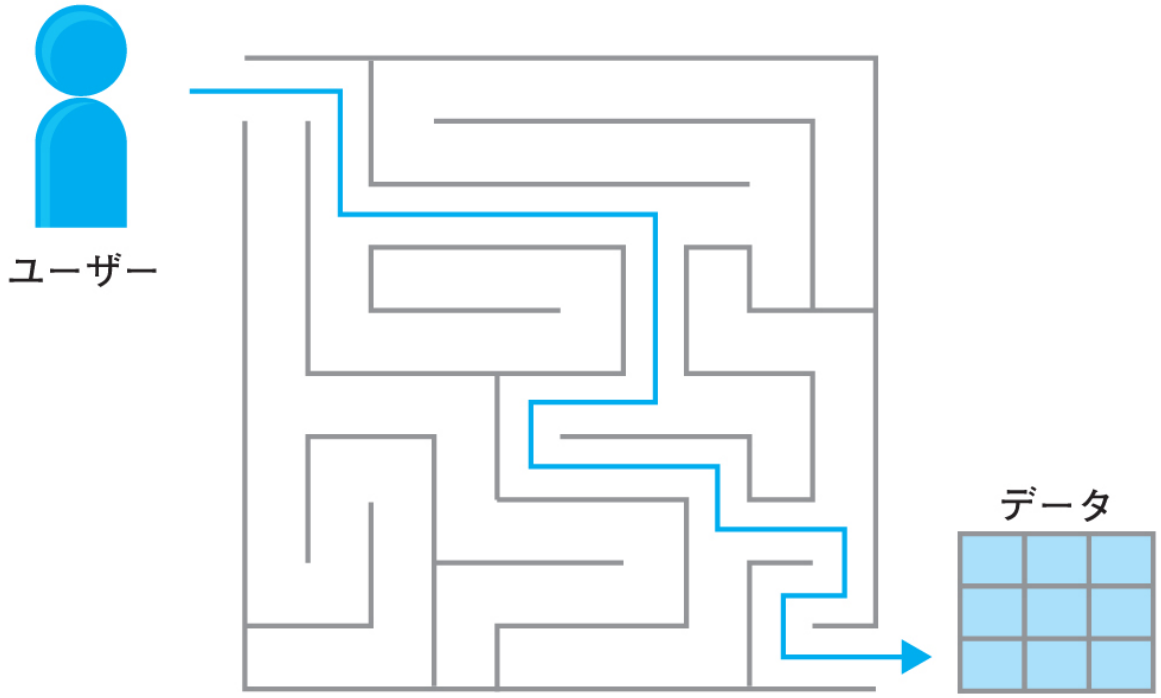


図6-2●統計情報は、データへの最短距離を知るための地図

C.J.Date『データベースシステム概論第6版』（丸善、1997） p.68

前述のように、インデックスはSQLのパフォーマンス改善のための非常にポピュラーな手段です。その理由を整理すると、以下のようなものを挙げられます。

❶ アプリケーションのコードに影響を与えない（アプリケーション透過的）。
❷ テーブルのデータに影響を与えない（データ透過的）。
❸ それでいて性能改善の効果が大きい。

❶ アプリケーション透過的

インデックスを使うかどうかは、DBMSが自動的に判断します。したがって、インデックスを使う場合、単純にデータベース側にインデックスを作成すれば良いだけで、アプリケーションプログラムの変更が必要ありません。これは使う際のハードルが非常に低いということです。同じチューニング手段でも、非正規化はアプリケーションを大きく改修しなければなりませんでした。

このように「存在を意識しなくて良い」という性質を「**透過性**（transparency）」と呼びます。もとは「透明」という意味です。空気のように透明でその存在を意識しなくて良い、というニュアンスの言葉です。アプリケーションから見れば、インデックスは**空気のように透明**なのです。

② データ透過的

インデックスはまた、データ透過的でもあります。すなわち、インデックスを作成することでテーブルに格納されているデータの中身が影響を受けることがありませんし、テーブルの構造も変化することはありません。したがって、インデックスを作成する際に論理設計を修正するような手戻りを心配する必要はありません。

③ 大きな性能改善効果

インデックスのもたらす性能改善の効果は、しばしば劇的です。これは、インデックスの性能が、データ量に対して線形よりも緩くしか劣化しないためです（詳細は後続の節で見ます）。したがって、インデックスによる性能改善は、多くの場合にデメリットをメリットが大きく上回る形で成果が出ます。といっても、インデックスをやみくもに作っても、効果が出るわけではないことは言うまでもありません。正しい指針を理解したうえで使って初めて効果が得られます。

以上のようなインデックスについての基礎知識を前提としたうえで、その詳しいメカニズムと設計の方法を学んでいきましょう。

まずはB-treeインデックスから

一口にインデックスと言っても、いくつかの種類があり、またDBMSによっても使用できる種類に差があります。しかし、実は頻繁に利用するインデックスは1種類しかないため、基本的には一つを覚えておけば十分です。そのインデックスが、**B-tree**イ

ンデックスです［※2］。この他にもビットマップインデックス、ハッシュインデックスなどがあるのですが、実際に使うことはまれです。B-treeは、非常にポピュラーであるため、通常、DBMSにおいて何のオプションもつけずにインデックスを作成すると、暗黙にB-treeインデックスが作成されるぐらいです。

勘どころ 40

インデックスにはいろいろあるが、まずはB-treeインデックスを覚えておこう。

B-treeインデックスの長所

しかし、B-treeインデックスのパフォーマンスが非常に優れている、というわけではありません。考案者のルドルフ・バイエルもそのことを認めていて、「もし世界が完全に静的で、データが変化しないなら、他のインデックス技術でも同程度のパフォーマンスは達成できるだろう」と認めています。要するに、「場合によっては他のタイプのインデックスのほうが優れていることもある」というのです。

それでは、B-treeの長所とは何でしょうか？

それは、**平均点の高さ**です。クリス・デイトは、B-treeを「幅広い名手」と呼びました。事実、B-treeをレーダーチャートで評価すると、「オール4」の秀才型になります（図6-3）。

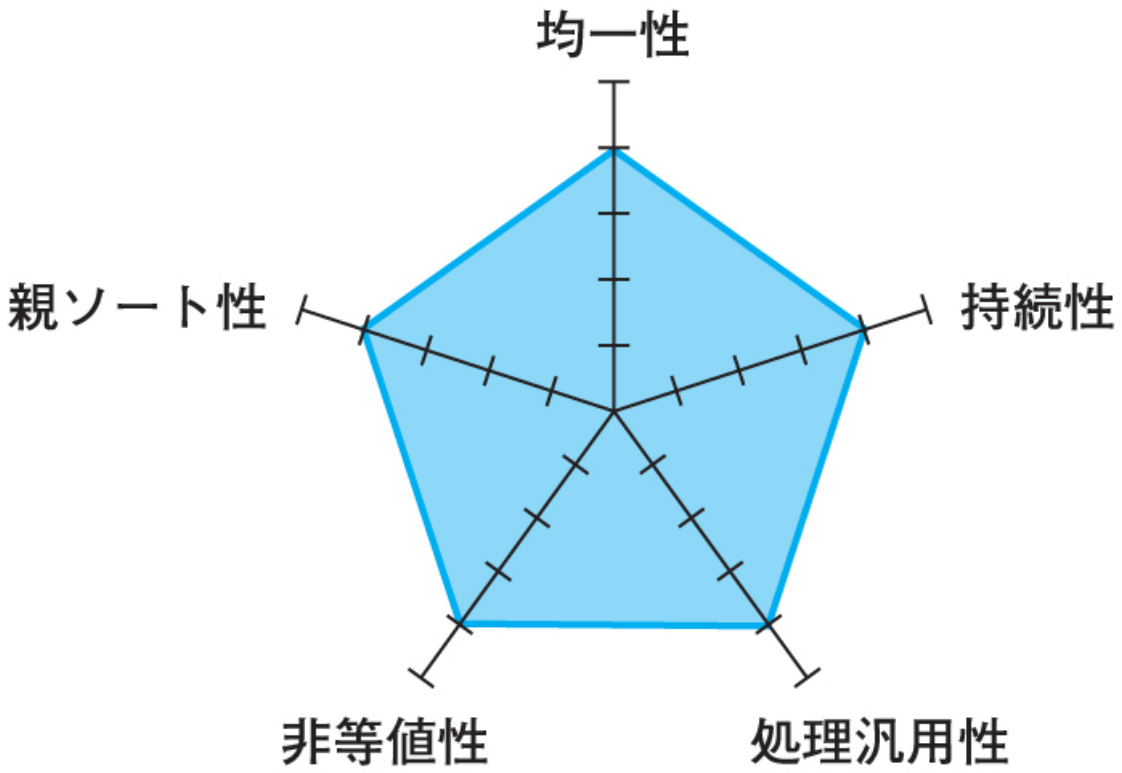


図6-3●B-tree の評価

❶ 均一性（4点）　　：各キー値の間で検索速度にバラツキが少ない。

❷ 持続性（4点）　　：データ量の増加に比してパフォーマンス低下が少ない

❸ 処理汎用性（4点）　：検索、挿入、更新、削除のいずれの処理もそこそこ速い。

❹ 非等値性（4点）　　：等号（=）に限らず、不等号（<、>、<=、>=）を使ってもそこそこ速い。

❺ 親ソート性（4点）　　：GROUP BY、ORDER BY、COUNT／MAX／MINなどなどソートが必要な処理を高速化できる。

これは、他のインデックスにはない特徴です。他のタイプのインデックスは、いずれも一長一短のものが多く、汎用性に欠けるのです。B-treeは、どの科目でも一番にはなれないのですが、総合評価で1位を勝ち取るタイプです。

勘どころ 41

B-treeインデックスはオール4の秀才。

B-treeインデックスの構造

なぜB-treeが秀才型なのか、その理由を、B-treeの構造を解説しながら見ていきましょう（図6-4）。

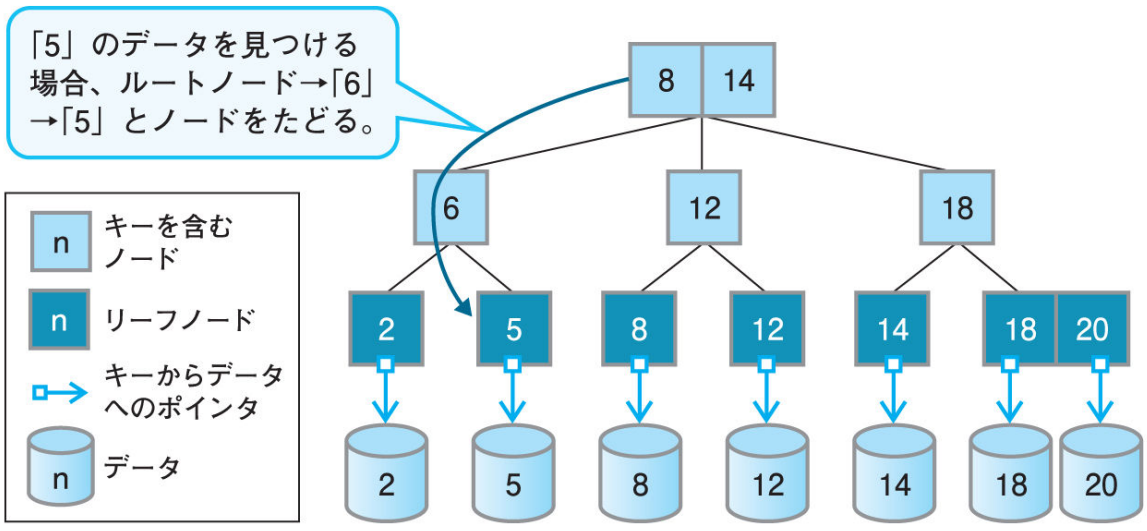


図6-4●B-tree の構造

B-treeは、「木」という名前のとおり、木構造でデータを保持します。最下層のリーフ（葉）と呼ばれるノードだけが、実データに対するポインタを保持しており［※3］、データベースは、最上位のノード（ルート）から順にノードをたどって、リーフから実データを見つけにいきます。

❶ 均一性

なぜB-treeインデックスは、キーとなる列については、どのような値を指定しても同じぐらいの速度で結果を得られるのでしょうか？

それは、B-treeが**平衡木**（balanced tree）であるためです。平衡木とは、どのリーフもルートからの距離（高さ）が一定の木のことを指します（図6-5）。これによって、どんなキー値を使っても、常にリーフまでの距離が一定になるため、探索を同じ計算量で行なえるのです。

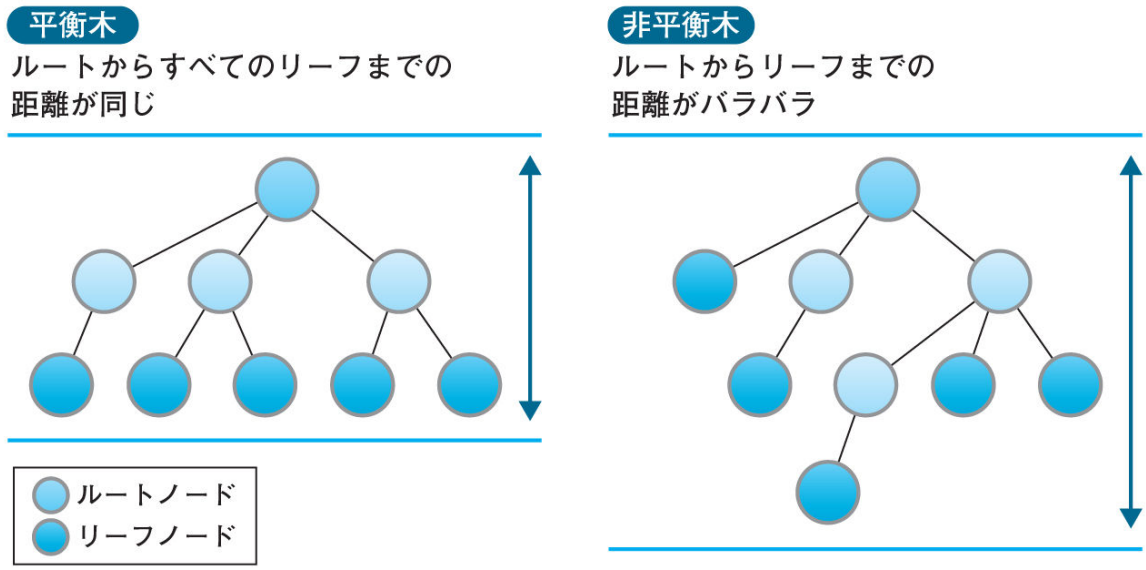


図6-5●平衡木と非平衡木

なお、B-treeは、最初に作られたときは、きれいな平衡木なのですが、テーブルへの挿入、更新、削除などが繰り返されることによって、インデックスの構造も崩れていき、非平衡木になっていくことがあります。B-treeは、極力このようなバランスの崩れを起こさないよう、自動的に修復する機能も備わっているのですが、それでも長期間の運用によって更新が重なれば、木のバランスは悪くなり、探索にかかるコストもバラつきが出るようになります。

❷ 持続性

先ほど、「B-treeといえども更新が繰り返されることによって性能劣化は発生する」と述べました。しかし、それでもなお、B-treeの性能劣化は長期的に見ても、非常に緩やかなのです。この「緩やか」をもう少し正確に言うと、テーブルのデータ量が増えても、B-treeの検索や更新にかかる時間はほとんど増えません。厳密には、データ量の対数に比例して増えます。このような計算量を示すとき、ランダウの記号Oで表現することがありますが、それを使うと、B-treeインデックスの性能は$O(\log n)$となります（nはデータ量）。

これは、テーブルをフルスキャンする処理よりもかなり高速です。フルスキャンは、ほぼテーブルのデータ量に比例して、$O(n)$で性能が劣化していくからです（図6-6）。

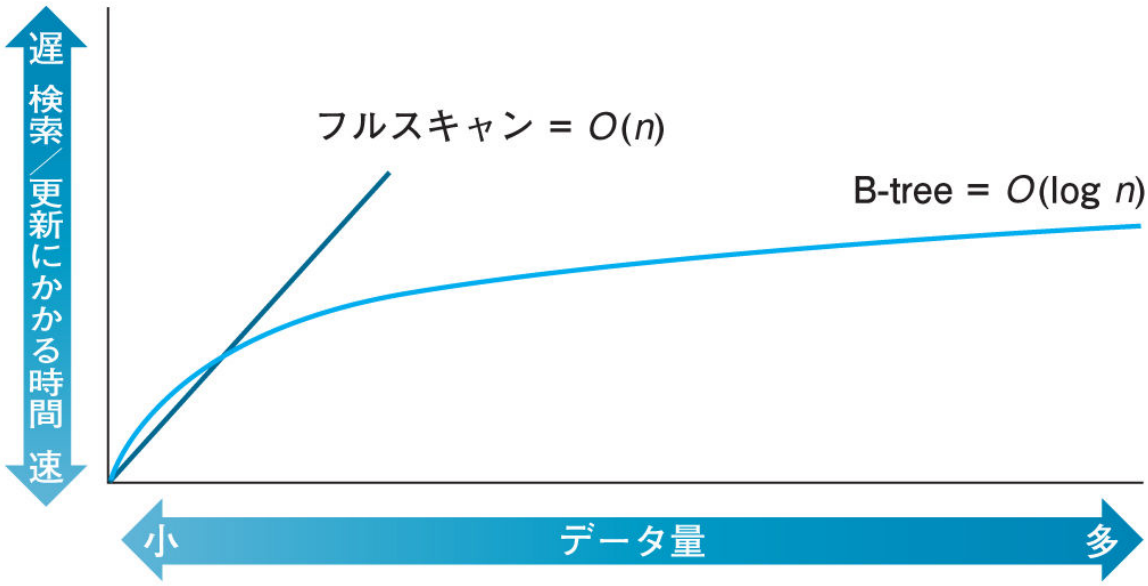


図6-6●データ量が多いほどB-tree はフルスキャンより有利

なぜB-treeインデックスがデータ量に対して緩やかにしか性能が劣化しないのでしょうか？

その秘密は、B-treeがかなり「平べったい（broad）」木だからです。具体的に言うと、B-treeの木の高さは、平均的には3 ～ 4程度です。このように背が低い木であることは、データ量が膨大に増えたとしても変わらない特性です。

❸ 処理汎用性

B-treeインデックスは、挿入、更新、削除のコストも、検索と同じくデータ量nに対してO(log n)です。したがって、いずれの処理も同じぐらいの探索速度となり、かつデータ量が増えても性能劣化の度合いが緩やかです。このような特徴は、たとえばビットマップインデックスにはありません。ビットマップインデックスは、検索性能はB-treeを凌駕することもありますが、更新に多大な時間を要します。

❹ 非等値性

B-treeは、等号（=）による検索のみならず、不等号（<、>、<=、>=）やBET WEENといった範囲検索の条件に対しても、高速化を可能とします。というのは、B-treeは、構築されるとき必ずキー値をソートするため、たとえリーフノードを一つに絞れないとしても、特定のノードよりも「左」とか「右」のノードだけに探索範囲を絞ることが可能になるからです。B-treeが効果を持たない検索条件は、否定条件（<>、!=）です。これは、特定のノード以外のすべてのノードが該当してしまうので、B-treeによる絞り込みが効かず、まったく役に立ちません。

❺ 親ソート性

SQLは一切の手続きを記述しないため、SELECT文やUPDATE文の中でも明示的にソートを記述することがありません。しかし、以下のような処理を記述したときには、暗黙にDBMS内部でソートが行なわれています。

- 集約関数（COUNT、SUM、AVG、MAX、MIN）
- ORDER BY句
- 集合演算（UNION、INTERSECT、EXCEPT）
- OLAP関数（RANK、ROW_NUMBERなど）

ソートというのは、かなりコストの高い演算です。ソートはDBMS内部で専用のメモリ領域が割り当てられており、その内部に一時的にデータを保持して実施されますが、大量データのソートが必要な場合、メモリに載りきらないためにあふれてしまうことがあります。その場合、DBMSは一時的にディスクへデータを書き出します。この場合のI/Oコストが非常に大きなものになるのです。したがって、SQL 文を記述する

際は、極力大きなソートを避けることがパフォーマンス上は望ましい、ということになります。

B-treeインデックスは、先述のように構築時にキー値をソートして保持します。そのため、B-treeインデックスが存在する列をORDER BY句のキーとして指定した場合、ソート処理をスキップすることが可能になるのです。これは、データベースのパフォーマンスにとって鬼門の一つであるソート処理をチューニングする大きな助けになります。

ちなみにB-treeの「B」がどういう意味かはわかっていません。考案者のルドルフ・バイエルとエドワルド・マクライトが明かしていないからです。「Balanced」や「Broad」という説が有力ですが、「Boeing」という珍説もあります。これは考案者の二人がボーイング社の研究所で働いていたからです。

細かい話ですが、正確にはリーフからのみ実データへのポインタを持つのは、B+treeというB-treeの亜種ですが、一般的にDBMSで利用されるインデックスはB+treeが多いため、以後もこのB+treeを前提に話を進めます。

それでは、B-treeインデックスの特性と仕組みを理解したところで、具体的にインデックスはどのように設計していけば良いのか、ということを見ていきましょう。

B-treeインデックスはどの列に作れば良いか？

B-treeインデックスは、もちろん、やみくもに作れば良いわけではなく、そこにはいくつかの指針があります。それらを列挙すると以下のようになります。

指針1 大規模なテーブルに対して作成する。
指針2 カーディナリティの高い列に作成する。
指針3 SQL文でWHERE句の選択条件、または結合条件に使用されている列に作成する。

以下、順に説明します。

B-treeインデックスとテーブルの規模

先ほどB-treeインデックスの特性「持続性」について解説したときの、データ量と処理時間のグラフを、もう一度見てみます。

すると、データ量が少ない場合、B-treeインデックスを使うよりもフルスキャンに任せたほうが高速な領域があることに気付きます（図6-7）。

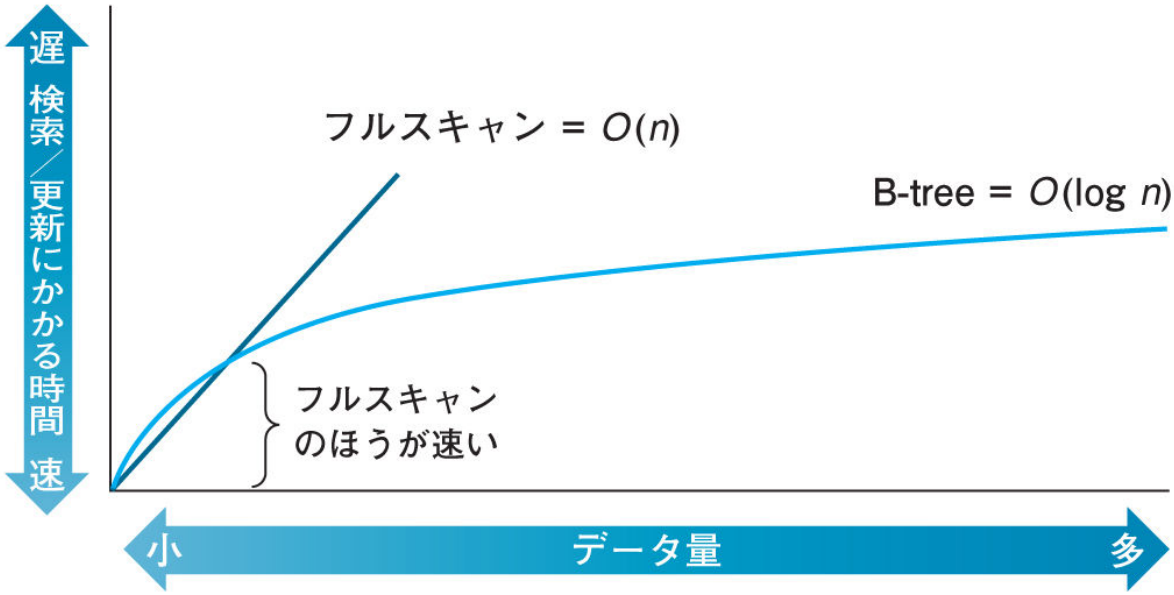


図6-7●フルスキャンのほうが高速な領域がある

この領域における処理時間の差はごくわずかであるため、実際のところB-treeでもフルスキャンでも大差はありません。しかし、それであればわざわざ無駄なインデックスを作る必要はありません。

では、その「データ量が少ない場合」というのがどの程度の閾値（しきいち）なのか、というところが気になるところです。これは、ストレージやサーバーの性能など環境要因によって変わるために、固定的な値は存在しないのですが、最近のハードウェアを考えるなら、目安としてはレコード数が1万件以下の場合はほぼ効果がないと考えてもらってかまいません。

勘どころ 42

データ量が少ない場合はインデックスの効果はない。

ただ、これはあくまで目安です。実際の閾値はシステムによって異なるため、あらかじめ簡単な実測を行なって感覚をつかむことをお勧めします。

B-treeインデックスとカーディナリティ

B-treeインデックスを作成する列として適不適を判断するための最も重要な情報が、**カーディナリティ**です。これは、特定の列の値が、どのぐらいの種類の多さを持つか、ということを表わす概念です。

たとえば簡単な例で、「性別」を表わす列を考えましょう。この列が持つ値としては、次のようなものが考えられます。

1. 男性
2. 女性
3. 不詳

したがって、この列のカーディナリティは「3」ということになります。これは非常に小さいカーディナリティの例です。

一方、たとえば「顧客の口座番号」とか「受付日」のカーディナリティはどうでしょうか？ こちらは非常にたくさんの種類があると想定されます。口座番号については、同一の銀行内では絶対にかぶることはないので非常に大きいですし、受付日も1年で365日（営業日だけに限定するなら200日程度）のカーディナリティがあるため、まずまず高いと言えます。

B-treeインデックスを作るときは、こうしたカーディナリティの高い列を選ぶことが基本です。この場合の目安は、特定のキー値を指定したときに、全体のレコード数の5％程度に絞り込めるだけのカーディナリティがあることです。365日のうちの1日を指定するSELECT文を考えるとすれば、0.3％に絞り込めるため、「受付日」列にB-treeインデックスを作る意味はある、と判断できます。

カーディナリティの注意点

ここで、カーディナリティについて二点、注意しておくことがあります。

まず一つ目は、複合列に対してインデックスを作成する場合、カーディナリティは対象の複合列の組み合わせで考える、ということです。たとえば、（a, b, c）という列にインデックスを作成するとして、カーディナリティがそれぞれaは2、b は10、Cは5だとします。この場合、どの一列をとっても、絞り込みの率は5％より大きくなります。しかし、(a, b, c)という組み合わせで見た場合のカーディナリティが100だとすれば、絞り込みの率は1％となり、B-treeインデックスは十分に有効性を発揮します（なお、なるべく先頭に近いキー、たとえばaやbのカーディナリティが高いほど、効率的に絞り込めるため性能的に有利です）。

そして二つ目は、たとえカーディナリティが高くても、特定の値にデータが集中しているような列は向いていない、ということです。たとえば、1 ～ 100までの値を取る列があったとします。この列のカーディナリティは100です。ところが、値の99％が100で、1 ～ 99の値は全体の1％しか取らない、というようなケースではB-treeによる検索性能は安定しません。これは、100を指定したSELECT文が非常に広範囲の検索を行なわなければならないのに対して、残りの1 ～ 99の値を指定したときはほぼピンポイントでヒットするからです［※4］。

勘どころ 43

カーディナリティが高い列ほどインデックスの効果が高い。ただし、値が平均的に分散しているのがベスト。

B-treeインデックスとSQL

これは当たり前といえば当たり前の話ではありますが、SQLで検索条件や結合条件として使用されない列にいくらインデックスを作っても無意味です。

ただし、インデックスが使用されるためには、SQLの記述方法としていくつか気をつけるべきポイントがあるので、代表的なものを挙げます。

今、SomeTableのcol_1という列にインデックスが存在するとします。この場合、以下のようなSQLはいずれもインデックスを利用できていません。

❶ インデックス列に演算を行なっている

```
SELECT *
  FROM SomeTable
 WHERE col_1 * 1.1 > 100;
```

インデックスを作成した列はSQLにおいて「裸」で用いるのが原則です。演算を行なってはいけません。というのも、B-treeインデックスの中で保持されているデータは、あくまで「col_1」に対してであって、「col_1 * 1.1」ではないからです。これは、次のように同値の式変形を行なうことで回避できます。

```
WHERE col_1 > 100/1.1
```

❷ 索引列に対してSQL関数を適用している

```
SELECT *
  FROM SomeTable
 WHERE SUBSTR(col_1, 1, 1)= 'a';
```

理由は❶と同じです。インデックスの中に存在する値は、あくまで「col_1」の値であって、「SUBSTR(col_1, 1, 1)」の値ではないのです。

❸ IS NULL 述語を使っている

```
SELECT *
  FROM SomeTable
 WHERE col_1 IS NULL;
```

B-treeインデックスは一般的にNULLについてはデータの値とは見なさず、保持していません。したがって、IS NULLまたはIS NOT NULL述語に対しては有効ではありません。ただし、これはDBMSの実装に依存するところもあり、一部のDBMSはIS NULLを用いた場合にもインデックスが有効に作用します。しかし、汎用性はありません。

❹ 否定形を用いている

```
SELECT *
  FROM SomeTable
 WHERE col_1 <> 100;
```

否定形はインデックスを利用できません。これは、たとえ利用したとしても検索範囲が広すぎて役に立たないからです。

❺ ORを用いている

```
SELECT *
  FROM SomeTable
 WHERE col_1 = 99 OR col_1 = 100;
```

ORを用いた場合はインデックスが利用されません。これは次のようにINで書き換えることで回避できます。

```
WHERE col_1 IN(99, 100);
```

❻ 後方一致、または中間一致のLIKE述語を用いている

× SELECT * FROM SomeTable WHERE col_1 LIKE '%a';

× SELECT * FROM SomeTable WHERE col_1 LIKE '%a%';

○ SELECT * FROM SomeTable WHERE col_1 LIKE 'a%';

LIKE述語を使うときは、前方一致検索の場合のみ索引が使用されます。

❼ 暗黙の型変換を行なっている

文字列型で定義されたcol_1に対する条件を書く場合の例を示します。

× SELECT * FROM SomeTable WHERE col_1 = 10;

○ SELECT * FROM SomeTable WHERE col_1 = '10';

○ SELECT * FROM SomeTable WHERE col_1 = CAST(10, AS CHAR(2));

データ型の異なる列値をSQLにおいて選択条件または結合条件として利用する場合、数値型⇔文字列型、文字列型⇔日付型のように、型変換を行なって型を統一する必要があります。これは、通常はSQL文の中で関数を使って明示的に行ないますが、明示的な型変換を行なわなくても、SQL文がエラーになるわけではありません。

列とデータ型の異なる値を条件に指定した場合、DBMSは内部的に暗黙の型変換を行ないます。しかし、その場合はインデックスは使用されなくなります。これを回避するためには、明示的に条件に使用する値のデータ型を列のデータ型に合わせてやる必要があります。

B-treeインデックスに関するその他の注意事項

これでB-treeインデックスの設計方針について一通りのことを学びました。あと何点か、ちょっとした注意事項を述べておきたいと思います。

主キーおよび一意制約の列には作成不要

実は、DBMSは主キー制約や一意制約を作成する際、内部的にはB-treeインデックスを作成しています。B-treeインデックスがデータをソートして保持するため、重複値チェックにこれを利用しているわけです。そのため、主キーや一意制約が存在する列に、二重にインデックスを作成する必要はありません。こうした列をSQL文の中で条件として利用する場合は、自動的にインデックスが使われます。

B-treeインデックスは更新性能を劣化させる

これはB-treeに限らず、インデックス全般の欠点としてしばしば言われることです。インデックスは一般的にテーブルとは独立のオブジェクトとしてDBMS内部に保持されています。したがって、インデックスが作成されている対象の列値が変更されると、インデックス内に保持している値も変更しなければなりません。つまり、B-tree

インデックスを作成すればするほど、当該テーブルに対する更新性能が劣化していくのです。このトレードオフにはよく注意して、極力無駄なインデックスを作成しないよう気をつける必要があります。

定期的なメンテナンスを行なうことが望ましい

「均一性」と「持続性」の特性を説明したときにも述べたように、インデックスは、テーブルのデータが更新されていくと、長期的には構造が崩れて性能が劣化していきます（その度合いは緩やかとはいえ）。そのため、運用において定期的なメンテナンスを行なう、具体的にはインデックスの再構築を行なうことが、性能を維持するためには望ましい方策です。

　具体的にどの程度の頻度で行なうべきかは、システムがどの程度の更新量を持つか、といったシステム特性に依存するので一概には言えませんが、DBMSごとに、インデックスの構造が崩れている場合の指標値（断片化率や木の高さなど）と、その調査方法が存在するため、マニュアルで調べてみることをお勧めします。また、インデックス再構築のコマンドについては、演習問題で取り上げます。

これは裏を返すと、100を指定した検索が要件的に行なわれない場合には、この列にインデックスを作るメリットがある、ということです。しかしそのような条件が満たされることはまれです。

本章の最初にも述べたように、統計情報はテーブルやインデックスなど「データ」についてのデータ、すなわち「メタデータ」です。DBMSはこのメタデータを頼りにSQLのアクセスパスを決定します。その具体的なプロセスには、ユーザーは基本的に関わらないため［※5］、SQLがどのようなアクセスパスで実行されるか、という問題には、ユーザーは統計情報を通してのみ関与することになります。

そこでまずは、統計情報に関する設計指針を理解するに当たり、DBMSがどのようにして内部的にSQLを扱っているか、ということから学習しましょう。

オプティマイザと実行計画

DBMSがSQL文――SELECT文やDELETE文などのあらゆる種類のSQL文――を受け取ると、テーブルにアクセスするまでに図6-8のような流れをたどります。

まず、ユーザーから何らかの形でSQL文がDBMSへ発行されます（❶）。すると、最初にこのSQLを受け取るのは、DBMS内の「パーサ（parser）」と呼ばれるモジュールです。これの役割は、SQL文が適法な構文であるかどうかチェックすることです。文法的に間違っているSQL文は実行できませんから、そういうSQLはエラーとしてユ

ーザーに突き返す必要があります（「parse」は「解析する」という意味です）。いわば、パーサはDBMSの門番に当たります。

パーサによるチェックが済むと、次にSQLは「オプティマイザ（optimizer）」というモジュールへ送られます（❷）。このオプティマイザは、DBMSの頭脳とも言うべき重要な役割を担っています。というのも、SQLのアクセスパス、すなわち実行計画を決めるのがこのオプティマイザだからです（「optimize」は「最適化する」という意味です）。

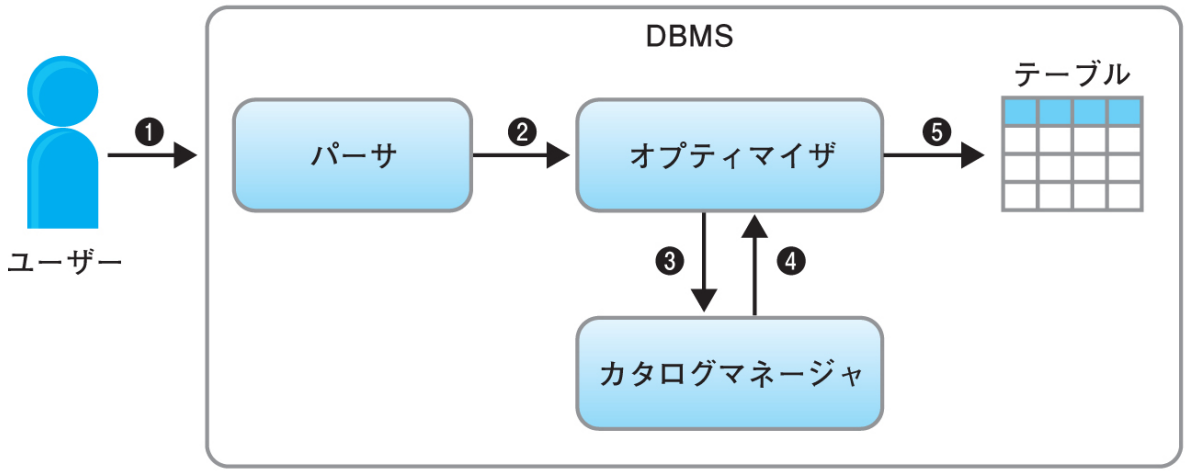


図6-8●SQL 文によってテーブルにアクセスする流れ

オプティマイザが実行計画を立てる際、必要になるのが統計情報でした。そこで、オプティマイザは「カタログマネージャ」というモジュールに、統計情報の照会をかけます（❸）。カタログマネージャは、統計情報を管理するモジュールで、たとえるなら図書館の司書です。

カタログマネージャから統計情報を受け取ると（❹）、オプティマイザはたくさんの経路の中から最短（と思われる）経路を選択し、SQLを手続きに変換します。そのとき得られた手続きの手順が、実行計画（「実行プラン」「アクセスプラン」という

呼び方もあります）です。それに従って、ようやく実データであるテーブルへとアクセスを行ないます（❺）。

このように、DBMSはSQLを受け取った後、直接データアクセスを行なっているのではなく、何段階もの前作業を行なっています。このように、DBMSが実行計画を決めるブラックボックス的な方法は、一見すると非効率で、どんなパスが選択されているか見えないことを不安に思うかもしれません［※6］。しかし、実際にはデータへのアクセス方法は非常にたくさんあるうえに、コストの計算も非常に複雑であるため、これを人間が行なうほうが非効率なのです（図6-9）。

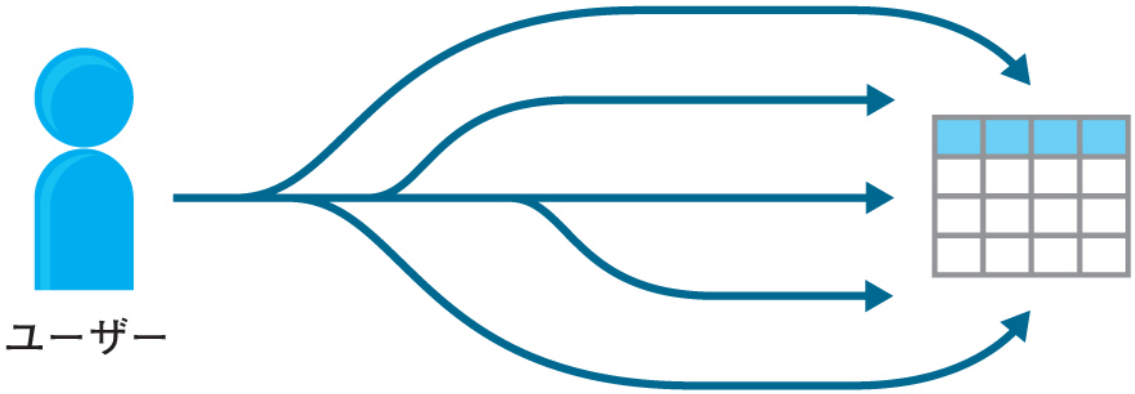


図6-9●データへの経路は複数ある。その中から最短経路を選ぶのがオプティマイザの仕事

勘どころ 44

SQLの実行計画は、DBMSが「お任せ」で選ぶ。

統計情報の設計指針

さてそうすると、繰り返しになりますが、一般的にエンジニアがSQLの実行計画立案に直接的に関与することはありません。それはあくまで、統計情報を通した間接的な関わり方になります。

したがって、設計において考慮することも、統計情報の収集をどのように行なうか、というポイントに絞られます。大きくは、以下の二点を考えます。

ポイント1 統計情報収集のタイミング
ポイント2 統計情報収集の対象（範囲）

統計情報収集のタイミング

統計情報を収集すべきタイミングについての基本的指針は以下の一点に尽きます。

勘どころ 45

データが大きく更新された後、なるべく早く。

この場合の「更新」には、INSERT ／ UPDATE ／ DELETEのいずれも該当します。レコード件数が増減するのはもちろんのこと、データの値の分布や偏りが変わることも、アクセスパスの選定に影響するためです。

更新処理によってテーブルのデータが大きく変われば［※7］、古い統計情報と最新のテーブルの状態に齟齬（そご）が生じることになります。カーナビでたとえるなら、「地図

が古くなる」わけです。間違った情報に基づいていては、オプティマイザがどれだけ賢くても、正しい道を選ぶことはできません。

逆に言うと、データの更新量が少ない場合は、統計情報を収集する意味はあまりありません。しかも、統計情報収集というのは、それなりにリソースを消費する、長時間かかる処理でもあります。テーブルの規模が大きい場合には、数時間に及ぶこともあります。そのような重い処理を、オンライン処理が行なわれる日中帯に実施することは、他の処理の性能をかえって劣化させるリスクを伴います。したがって、先ほどの勘どころ 45 と矛盾するようなことを言いますが、統計情報収集は、基本的にはシステムの使用者が少ない夜間帯に実施することが原則です。

勘どころ 46

統計情報収集は原則、夜間帯に実施する。

なお、DBMSによっては統計情報収集は、デフォルトの設定で定期的に実施されていることもあります［※8］。DBMS側が「気を利かせて」くれているわけですが、こうした動作は実装によって異なるため、使っているDBMSのマニュアルで統計情報収集の運用がどういう設定になっているか、確認することをお勧めします。

統計情報収集の対象（範囲）

統計情報を収集する範囲は、これまでの統計情報についての説明から自ずと導かれます。つまり、「大きな更新のあったテーブル（およびインデックス）」が対象です。

上述のように、統計情報収集というのは、わりと負荷の高い処理でもあります。したがって、データが変更されていないテーブルまで再収集の対象に含めるのは、収

集にかかる時間を無駄に長くするだけなので、本当に必要なテーブルに限定する必要があります。

統計情報収集は、通常のテーブルだけを考える場合、その対象に悩むことはあまりありません。どのテーブルがいつ更新されるか、ということは、テーブルを設計したエンジニアであれば知っていて当然ですし、CRUD表のような形で整理もするからです。

しかし、統計情報収集の対象として、一つ気をつけなければならないテーブルがあります。それが、一時テーブルです。これは、名前のとおり一時的に使うデータを保持するテーブルで、使う直前にデータを投入し、終わったら空っぽになります。このテーブルに対する統計情報収集の考え方については、第8章の演習問題で取り上げているので、詳細はそちらを参照してください。

統計情報の凍結について

統計情報収集の設計の基本は、上記で説明したとおり、なるべくそれを最新の状態に保つことです。オプティマイザが適切な実行計画を作るためには、地図が最新である必要がありますし、この点については、あらゆるDBMSベンダーの間で見解が一致しています。

しかし、実際のシステム開発においては、あえて統計情報の収集をまったく行なわない、という選択肢が有効な場合があります。逆説的に思えるかもしれませんが、そうした設計が合理性を持つ例外的なケースが存在します。

そのように、統計情報をあえて収集せず、特定の時点の統計情報で更新を止めることを「統計情報を凍結する」や「統計情報をロックする」と言います。では、どのような場合に、統計情報の凍結を行なうのが望ましいのでしょうか？

それは、**現状のものから実行計画を変化させたくない**場合です。現在使われている経路が、将来にわたってもデータへの最短ルートであり、現状維持が最適な選択だ、とわかっている、そういう場合に、統計情報を凍結します。具体的には、システムのサービス終了時のデータを想定した状態での統計情報が存在する場合です。

第2章でサイジングを説明した際にも述べましたが、多くのシステムでは、テーブルに含まれるデータは時間とともに増えていきます。したがって、データベースのパフォーマンスは、たとえまったく同じSQLであっても、サービス終了が近づくほど悪くなっていくのが普通です。しかし逆に言うと、そのサービス終了時を想定したデータの状態で、十分なパフォーマンスを実現できているならば、それよりも時間軸的に前のあらゆるポイントにおいても、（実行計画が同じならば）それよりも速い、十分なパフォーマンスが実現できることが保証されます。

このような統計情報の凍結を行なうのは、データ量の増加に伴って実行計画が変化しパフォーマンスの劣化が起こることは、決して珍しくないからです（図6-10）。DBMSも、しょせんは人間の作るソフトウェアです。いかに最新の地図を使おうとも、100％最適な実行計画を選択するとは、言い切れません。時には間違った（とあえて言ってしまいますが）実行計画を選ぶこともあります。統計情報の凍結は、いわばオプティマイザの仕事を完全には信じない、**悲観的**な考えに基づいた選択なのです。

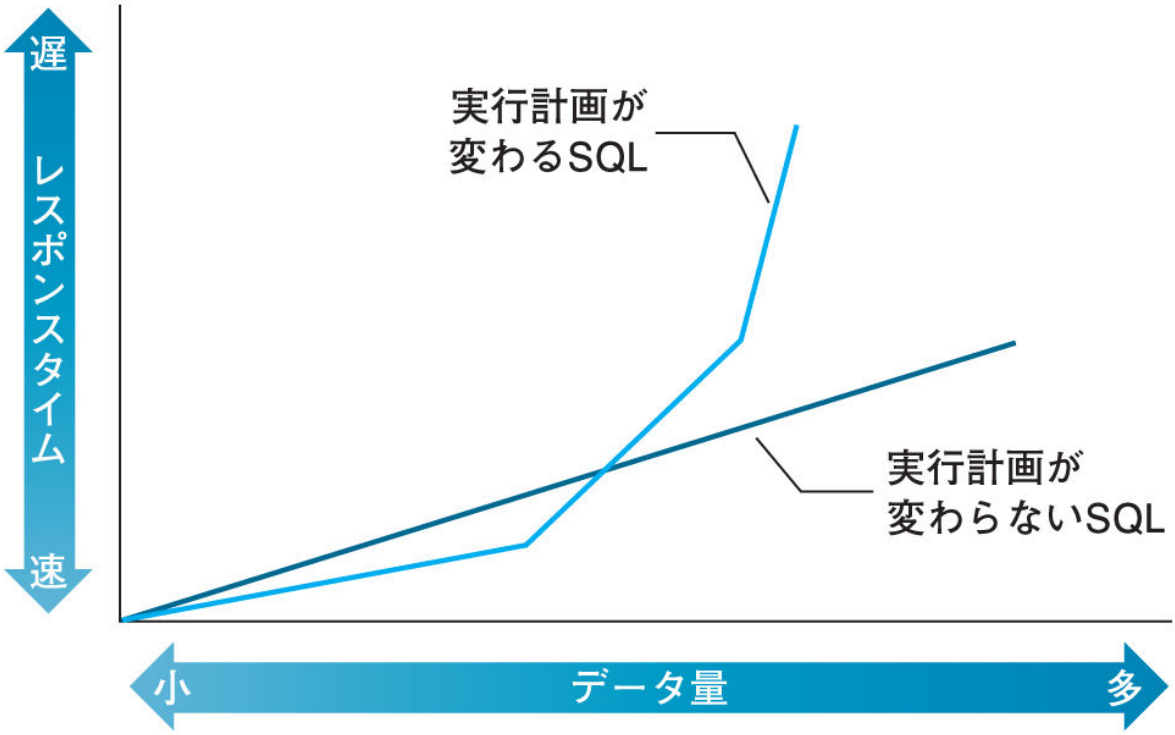


図6-10●実行計画の変動には性能リスクが伴う

統計情報を（ということはつまり実行計画を）凍結すること自体は、簡単にできます。統計情報の収集をストップすれば良いだけです。DBMSによってはさらに、統計情報にロックをかけて「読み取り専用」の状態にする機能もあります。

統計情報の凍結を行なうときに難しいのは、サービス終了時のデータに近いテストデータを、開発時に用意することです。これはデータ量や分布、分散を予想で作るしかないので精度に限界がありますし、データ量を増幅させる作業もなかなか地道で骨が折れます。「統計情報を凍結したい」と思っても、現実にそれを行なうための実作業の膨大さを前に断念することもしばしばあります。実際に統計情報を凍結できるのは、データ量がサービス開始時と終了時でほとんど変化しない特性を持っているシステムぐらいというのが現実です。

ともあれ、実行計画の変動リスクを極力回避しようとする場合には、このような悲観的設計も選択肢としてはあるのだ、ということは覚えておいてください。

勘どころ 47

統計情報の凍結は、オプティマイザを信じない悲観的設計。しかし現実に実施するのはかなり大変。

演習問題

演習 6-1 ビットマップインデックスとハッシュインデックス

本文ではインデックスの種類として、B-treeインデックスのみを取り上げました。これ以外にも、リレーショナルデータベースで利用可能なインデックスには「ビットマップインデックス」や「ハッシュインデックス」といった補助的な存在します。これらのインデックスの長所と短所を調べてください。

演習 6-2 インデックスの再編成

「B-treeインデックスに関するその他の注意事項」（178ページ）で、B-treeインデックスは更新が発生することで時間とともに構造が崩れ、性能が劣化することを説明しました。これを防止するためにインデックスの再編成が必要になります。再編成を実施するための具体的な方法を、DBMSごとに調べてください。

厳密に言うと、ユーザーがより直接的にアクセスパスを決定する手段（オプティマイザヒントなど）がないわけではありません。しかしこれはあくまでチューニングの一手段であって、設計の方法論ではありません。また、DBMSベンダーも実行計画の立案はDBMSに任せることを推奨しており、たとえばSQL Serverのマニュアルには以下のように記述されています。

通常、SQL Server クエリオプティマイザでは、クエリにとって最適な実行プランが選択されるため、<join_hint>、<query_hint>、および<table_hint>は、経験を積んだ開発者やデータベース管理者が最後の手段としてのみ使用することをお勧めします。

SQLの実行計画を見る手段は、すべてのDBMSで用意されています。オプティマイザの選択する実行計画が最適なものにならなかった場合、最終的にはこの実行計画を見ながらエンジニアが最適なパスを考えるという、ある意味、本末転倒なチューニングをすることになります。

それはもちろん、そのテーブルに付与されているインデックスのデータも変わるということです。

たとえばOracle 11gは夜22時など決まった時間帯に統計情報収集を行ないます。ここでも、やはり夜間帯が選ばれていることに注目してください。

第7章

論理設計のバッドノウハウ

この世界は、バッドノウハウ（アンチパターン）と呼ばれる、ろくでもない設計であふれています。システムの心臓であるデータベースの設計におけるバッドノウハウは、システム全体としての品質を致命的なレベルで損なってしまいます。本章では、そのような罪深いバッドノウハウの具体的な症例を検討し、なぜバッドノウハウは生まれるのか、そのどこが「悪」であるのかを明らかにします。

学習のポイント

- リレーショナルデータベースにおける論理設計の基本は、正規化です。しかし、世の中には、この原則に反した設計が多く存在します。これをバッドノウハウ、またはアンチパターンと呼びます。
- バッドノウハウは、正規化からは導かれません。また、非正規化によって生まれるわけでもありません（非正規化はトレードオフを理解している限りバッドノウハウではない）。
- バッドノウハウがダメなのは、これがシステムの品質を左右するうえに、後から変更することが容易でないからです。
- 論理設計の代表的なバッドノウハウには、非スカラ値、ダブルミーニング、単一参照テーブル、テーブル分割、不適切なキー、ダブルマスタがあります。

7-1 論理設計の「やってはいけない」

ここまでで、正規化、ER図、非正規化（あるいは非正規化はなぜすべきでないか）について学習してきました。これで論理設計の基礎を一通りカバーしたことになります。第6章までを基礎編とするなら、本章以降を応用編や実践編と名づけてもいいのですが、実際のところ、本章で紹介するのは、真似してはいけない設計のパターンです。こういうパターンをIT業界では「**アンチパターン**」や「**バッドノウハウ**」と呼んでいます。わざわざこのようなパターンに注目するのは、それだけ世の中にそういう設計が多く存在している、ということでもあります。こういう設計ばかりを集めた本（アンチパターン集）も出版されているぐらいです。

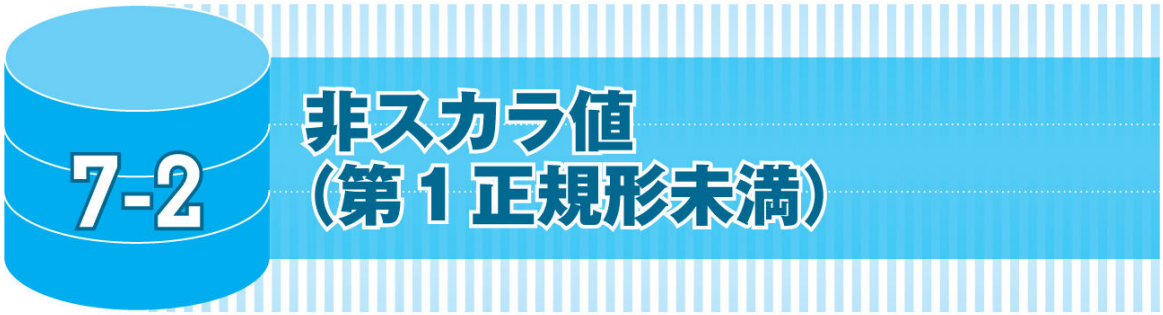
初心者の方の中には、まだイメージが湧かない人もいるかもしれませんが、システムの品質を決めるのは、設計です。プログラミングというのは、設計をプログラミング言語に翻訳する作業であって、プログラミング自身がシステムの品質を左右することは、相対的にはまれです。「**戦略の失敗を戦術で取り返すことはできない**」という有名な言葉がありますが、システム開発においては、設計が戦略、プログラミングが戦術に相当します。

が、しかし！ 思い切って予言してしまいますが、みなさんが実際に開発の現場に出て、既存の設計書を見ると、たびたび驚くことになるでしょう。そのときの気持ちを一言で表わすなら次のようになります。

「誰だ、この設計書を書いたのは！」

実は、世の中のシステムには、正規化も何もかも無視した論理設計があふれかえっているのです。ほとんど無法地帯と言いたくなるような開発現場もあります。もちろん、中にはやむをえぬ事情があって意識的に非オーソドクスな設計を行なっているケースも、ないわけではありません。しかし大半のケースにおいては、そうしたダメ設計が生まれる理由は、「**何も考えていない**」ことによるものです。

本章では、著者がこれまでに様々な開発現場で遭遇してきたバッドノウハウの実例を紹介していきます。読者のみなさんには、真似をしてほしくないのはもちろんのこと、みなさんの同僚、先輩、後輩がこのような悪の道に入り込みそうになったら止めてあげてください。そして、バッドノウハウ／アンチパターンの根絶に、ぜひ力を貸していただければと思います。



第3章において正規化について学習した際、最初に、最も低次の正規形である第1正規形を学びました。第1正規形の定義は、テーブルに含まれるすべての値がスカラ値、すなわち原子的な値である、というものでした（覚えているでしょうか？）。しかし、驚くかもしれませんが、世の中には、この最低限のルールすら守られていないシステムもしばしばあるのです。

配列型による非スカラ値

第3章で取り上げた、非スカラ値の表をもう一度見てみましょう。

■非正規形

扶養者

社員ID	社員名	子
000A	加藤	達夫 信二

000B	藤本	
001F	三島	敦 陽子 清美

この表は、「子」列が非スカラ値です。そのため、これは「表」ではあっても、リレーショナルデータベースにおける「テーブル」ではありません。そのように第2章では説明しました。原則としてそう考えていただいて良いのですが、しかし、今から少しややこしい話をします。

SQLには世界共通の標準規格があり、数年に一度改訂が行なわれています。最近だと1999年、2003年、2008年に改訂版が定義されました。その1999年の改訂で標準に盛り込まれた機能に「**配列型**」というものがあります。ここでの「配列」はJavaやC言語など通常のプログラミング言語における配列と同じ意味です。すなわち、これはテーブルの一つの列を配列として扱うことのできる機能なのです。それはつまり、**非スカラ値を含むテーブルを作ることができる**、ということですから、リレーショナルデータベースの掟破りとも言うべき機能です。先ほどの「扶養者」テーブルを例にとると、具体的なテーブル作成のDDL（テーブルの物理定義SQL）は、以下のようになります［※1］。

■「扶養者」テーブルを作成するSQL 文

```
CREATE TABLE huyosha (
    shain_id  varchar(4),
    shain_mei varchar(20),
```

```
    kodomo      varchar(20)[], ← 配列型の宣言
    PRIMARY KEY(shain_id));
```

これで配列型の列「kodomo」が宣言されました。レコードのINSERT文も、以下のように子を配列の形で入力します。

■「扶養者」テーブルにレコードを格納するSQL 文

```
INSERT INTO huyosha VALUES ('000A', '加藤', '{達夫, 信二}');
INSERT INTO huyosha VALUES ('000B', '藤本', NULL);
INSERT INTO huyosha VALUES ('001F', '三島', '{敦, 陽子, 清美}');
```

こうすることで、列「kodomo」に、複数の値を配列として格納することが可能になります［※2］。

この配列型の機能が標準SQLに取り込まれる際には、いろいろ議論があったようです。確かに、Javaのような一般的なプログラミング言語は、配列型を持っているため、リレーショナルデータベースだけが配列を扱えないと、データベースとアプリケーションとの間のデータ受け渡し（インタフェース）に齟齬が出てしまい、面倒が多いのは事実です（図7-1）。一方で、リレーショナルデータベースは長い間、厳密な正規化によってデータ整合性を保つことをポリシーとしてきたため、その原則を崩すことには抵抗があったようです。

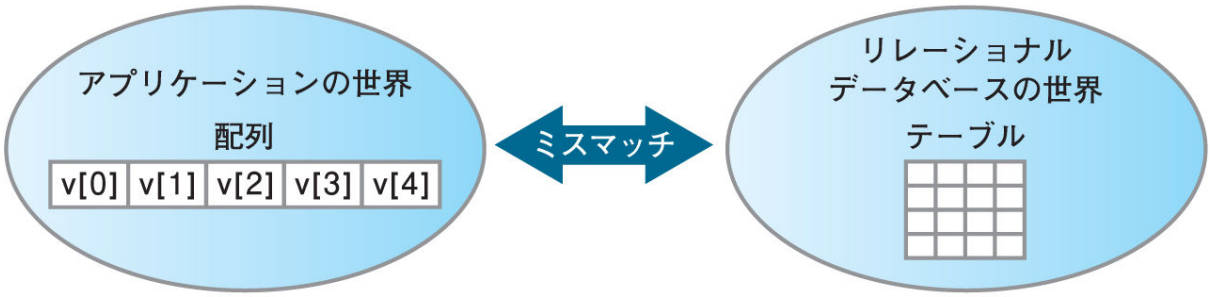


図7-1●アプリケーションとデータベースの間のミスマッチ

しかし結局のところ、この配列型は、機能としてはサポートされたものの、それほど一般的に普及するには至らなかった、というのがここ10年間の結果のようです。データベース側だけが配列型に対応したとしても、データベースと接続するアプリケーションやミドルウェアも対応しなければ意味がないため、まだ実用化へのハードルが高いのでしょう。この結果を見て、著者は深く安堵しました。

そのため現状としては、安易に配列型を使用するのではなく、第1正規形を確保することを優先するべきです。逆に言うと、配列型を採用する場合は、単にデータベース内部に閉じた設計だけを考えるだけでは不十分で、データベースに接続するアプリケーションやミドルウェアとの整合性を考慮しなければならないのです。これは、設計にかかるコストを増やすことにつながるため、採用する場合には細心の注意が必要です。

勘どころ 48

配列型は利用しない。第1正規形を守ろう。

ただ、配列型のような機能が、リレーショナルデータベースに要望される理由というのも、理解できないわけではありません。リレーショナルデータベースのテーブルはかな

り**固定的**な性格を持っていて、後になってから列を増減させるようなテーブル構成の変更コストが高くつきます。配列型のような、よく言えば「柔軟」なデータ型が欲しい、という気持ちもわかります。

しかし、原則として配列は、3-4節で見たように「列」ではなく「行」で表現するべきなのです。この「配列を表現する手段として列と行どちらがふさわしいか」という問題は、次章8-3節で再び詳しく取り上げます。

スカラ値の基準は何か？

ところでスカラ値と第1正規形について、次のような疑問を持った人がいるかもしれません。それは「**スカラ値の基準とは何か**」です。もちろん、問うまでもなく分解不可能な値と判断できるものも多いです。たとえば、会社名を意味する文字列「A商事」や年齢を意味する「29」という数値は、この単位でのみ意味を持つのであり、「A」と「商事」、「2」と「9」に分解してしまえば、単語の持っていた意味が完全に壊れてしまいます。

しかし一方、論理的には現状以上に分解可能な語もあります。たとえば、人名を意味する「鈴木健二」は、姓の「鈴木」と名の「健二」に分解することができます。あるいはメールアドレスを表わす「hoge@test.com」という文字列を、@を区切りとして「hoge」というアカウント名と「test.com」というドメイン名に分解することも可能です。こういうケースにおいては、スカラ値、つまりテーブルの列はどのような単位で作るべきなのでしょうか？

この問いには、一定の答えはありません。どのような単位でテーブルの列を作るか、という決定は、システムの要件次第です。「A商事」を「A」と「商事」のような明らかにナンセンスな分解をすることだけが禁じられていて、どのような単位でまとめるかという点についての指針は、設計理論からは出てきません。

先ほどの人名の例で考えるならば、人名をキーに検索を行なうようなシステムであれば、姓と名を別々のキーとして入力するようなインタフェースが好ましいかもしれません。メールアドレスの場合にも同様のことが言えそうです。

一般論としては上記のとおりですが、ここで著者自身の見解を述べておくと、著者は、**意味的に分割できる限り、なるべく分割して保持する**、という基本方針が良いと考えています。「鈴木健二」ではなく「鈴木」と「健二」、「hoge@test.com」ではなく「hoge」と「test.com」と分割して保持する、ということです。その理由は、分割したものを後で結合することは簡単にできるのに対し、結合された状態のものを後から分割するのは比較的難しいからです。

たとえば、メールアドレスの場合なら、「@」という確実な区切り文字（セパレータ）があるため、機械的な判別が可能ですが、名前の場合、どこからが姓でどこからが名かというのは、機械的に判別しにくいものです。「坂上田村麻呂」のように区切りがわかりにくい名前の人もいます。こういう場合、後になってデータベースから姓だけを切り出して利用したい、という場合に苦労します。一方、分解された状態で保存した文字列を結合するのは、SQLで簡単に行なえます［※3］。

上記の理由から、情報は可能な限り分解し、原子的な形でデータを保持することが望ましいと、著者は考えています［※4］。ただし、くれぐれも、「鈴木」を「鈴」と「木」に分解して、意味を破壊してしまうことだけはないよう気をつけてください。

勘どころ 49

情報は可能な限り分割して保存するのが良い。ただし意味を壊してはいけない。

このDDLはPostgreSQLで動作確認を行ないました。

2012年2月現在で配列型をサポートしているDBMSは、PostgreSQLなどごく一部にとどまります。

具体的には、「||」という縦棒を二本並べた文字列連結の演算子を使います。

こういう判断に迷うケースでは、分解した列と、結合した列の両方をテーブルに保持する、という「二正面作戦」を採用したい誘惑にかられることもあるでしょう。しかしこれは、データ容量を無駄に増やすうえ、冗長なデータを保持することになり更新コストが上がるため、基本的には採用するべきではありません。

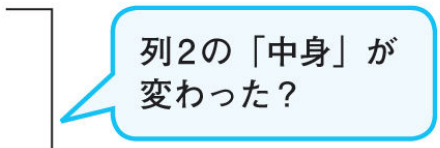
次に紹介するバッドノウハウは、人によっては「そんなやり方があるのか！」と驚くような衝撃的なものです。しかし世の中とは広いもので、人間とはいろいろなことを考えつく生き物なのです。それが、「ダブルミーニング」というバッドノウハウです。

この列の意味は何でしょう？

これがどういうものかは、言葉で説明するより、実際のテーブルを見ていただいたほうが早いでしょう。以下に、年度別に学生の情報を保持するテーブルを示します。ここで読者のみなさん、列1と列2が、それぞれ何を意味する列か考えてみてください。

学生

年度	学生名	列 1	列 2
2001	石井良治	170	62
2001	福原博	165	55
2001	大塚幸一	175	70
2001	佐藤孝司	178	65
2002	石井良治	170	19
2002	福原博	165	22
2002	大塚幸一	176	18
2002	佐藤孝司	179	20



さて、いかがでしょう。

列1については、すぐにわかったと思います。そう「身長」ですね。

一方、列2はどうでしょうか？　2001年の学生については50台後半から70までの分布……これだけ見ると「体重」のように見えますが、2002年のデータでは20前後に分布……こちらだけなら「年齢」……あれ、どっちだろう？

実は、このテーブルは、2001年と2002年で、列2に格納するデータを「体重」から「年齢」へ変えているのです。したがって、同一の列が二つの意味を持つ「ダブルミーニング」の状態になっているわけです［※5］。詳しい理由はわかりませんが、このテーブルを作成した人物が「2002年は体重はもう必要なくて、代わりに年齢だけわかればいい」と判断した結果、このような奇怪なテーブルが誕生したのです。

このテーブルを見た瞬間、「これは反則だろう」と思った読者の方もいると思います。そう、反則なのですが、やろうと思えばこのようなテーブルを作ることは可能です。しかし、こうしたダブルミーニングの列が許されない理由は、列の意味が不明確になることで、システムのバグを生み出す原因になるからです。

実際、このようなテーブル設計を行なうと、列名を見てもこの列が何のデータを格納しているのか、判別できません。ダブルミーニングの列は、運用途中で意味が変わってしまうため、「体重」とか「年齢」といった固定的な列名をつけることができないからです。これではプログラマも混乱します。

「ああ、この列、途中から意味が変わるから、コーディングのとき気をつけてね」
「えっ！？」

テーブルの列は「変数」ではない

このようなダブルミーニングの列が作られる背景として考えられるのが、プログラミング言語における「変数」の存在です。プログラミングを行なう際には、雑多な値を格納するための一時変数がよく使われます。そうした変数の名前も、やはり特定の意味を持たないため「tmp」や「var」という名前がつけられます。

こうしたプログラミングの習慣に慣れたエンジニアがリレーショナルデータベースのテーブルを見ると、「値を格納するわけだから、これも一つの変数なんだな」と考えるのは、ある意味で自然なことです（また事実、テーブルは形式的には変数と見なすことができます）。

しかし、リレーショナルデータベースにおけるテーブルは、意味的には現実の世界にあるいろいろな実体（エンティティ）の「写像」、つまり物を写し取ったものであるため、変数よりもずっと**静的**で固定的な存在です。テーブルや列の意味は、一度決

めたら容易に変更してはならないものである、というのがリレーショナルデータベースの世界における取り決めなのです［※6］。

勘どころ 50

列は変数ではない。一度意味を決めたら変更不可。

したがって、「2002年からは年齢の情報も格納したい」と考えたエンジニアは、次のように「年齢」列を追加することで対処するべきだったのです。

■2001年の状態

学生

年度	学生名	身長	体重
2001	石井良治	170	62
2001	福原博	165	55
2001	大塚幸一	175	70
2001	佐藤孝司	178	65

■2002年の状態

学生

年度	学生名	身長	体重	年齢
2001	石井良治	170	62	
2001	福原博	165	55	
2001	大塚幸一	175	70	
2001	佐藤孝司	178	65	
2002	石井良治	170		19
2002	福原博	165		22
2002	大塚幸一	176		18
2002	佐藤孝司	179		20

　これならば、列名を見ればどんなデータが格納されているか一目瞭然ですし、既存のアプリケーションに対しても改修を行なう必要がありません。

やろうと思えば、原理的には「トリプルミーニング」や、一般化した「マルチミーニング」も可能です。

一方、このようにテーブルの構成を柔軟に変更できないことは、しばしばリレーショナルデータベースの欠点としても指摘されています。しかし、この観点はより進んだテーマであるため、本書では踏み込みません。

次に紹介するのは、ダブルミーニングの応用版です。現実には、単純なダブルミーニングよりもこちらのほうが、見かける頻度が高いパターンです。

第3章において、社員情報を管理するテーブルをサンプルとして、第3正規化まで行ないました。その結果、最初は一つのテーブルにすべての情報を保持していたのですが、最終的に「会社」「社員」「部署」という三つのテーブルに分離することになりました。

多すぎるテーブルをまとめたい？

この三つのテーブルを眺めていると、あることに気付きます。それは、「会社」テーブルと「部署」テーブルが、構造的に同じ形をしている、ということです。この両テーブルは、ともに「識別ID」＋「名称」という形をしています。

■社員情報を管理するテーブル

社員

会社コード	社員ID	社員名	年齢	部署コード

C0001	000A	加藤	40	D01
C0001	000B	藤本	32	D02
C0001	001F	三島	50	D03
C0002	000A	斉藤	47	D03
C0002	009F	田島	25	D01
C0002	010A	渋谷	33	D04

■社員情報を管理するテーブル

社員

会社コード	社員 ID	社員名	年齢	部署コード
C0001	000A	加藤	40	D01
C0001	000B	藤本	32	D02
C0001	001F	三島	50	D03
C0002	000A	斉藤	47	D03
C0002	009F	田島	25	D01
C0002	010A	渋谷	33	D04

会社

会社コード	会社名
C0001	A 商事
C0002	B 化学
C0003	C 建設

部署

部署コード	部署名
D01	開発
D02	人事
D03	営業
D04	総務

二つのテーブルは同じ構造を持っている

ということは、この二つのテーブルは、あえて別々のテーブルにしておく必要はなく、**一つのテーブルにまとめる**ことが可能だということです。

まだ二テーブルぐらいなら、別々のテーブルとして管理してもまとめても管理コストに大きな差異は現われませんが、正規化は他のあらゆるテーブルに対しても行なう作業であるため、結果として、テーブル数がどんどん増えていきます。数百というテーブル数になることも珍しくありません。たとえば、上記の二つ以外にも「都道府県」や「性別」といったマスタ類のエンティティは、同様のテーブル構造を持つわけですから、こうしたマスタ群を一つにまとめてしまいたいという考えは、それほど不自然なものではありません。

こうした発想で作られたテーブルが、**単一参照テーブル**です［※7］。これは以下のようにあらゆるタイプのマスタテーブルを、一つのテーブルに放り込んでごった煮にしたものです。

■単一参照テーブル（雑多なコード体系の寄せ集め）

コードタイプ	コード値	コード内容	
comp_cd	C0001	A 商事	会社コード
comp_cd	C0002	B 化学	
comp_cd	C0003	C 建設	
	⋮		
comp_cd	C9999	Z 興業	
pref_cd	01	北海道	県コード
pref_cd	02	青森	
	⋮		
pref_cd	47	沖縄	
sex_cd	0	不明	性別コード
sex_cd	1	男	
sex_cd	2	女	
sex_cd	9	適用不能	

まず、何のコード体系であるかを識別するために、コードの名称が必要になります。これが「コードタイプ」列です。残りの「コード値」と「コード内容」は、個々のマスタテーブルが持っていた列と同じです。ただし、今まで「会社名」や「部署名」といった「名称」だった列が、何の名称であるか決められなくなるので「コード内容」という一般化した列名にせざるをえません。

単一参照テーブルの功罪

この単一参照テーブルが、前節で取り上げたダブルミーニングを一般化した手法であることが理解いただけたでしょう。ダブルミーニングでは、列の意味がレコードによって変化しましたが、今度はテーブル全体が、あるときは「会社」、あるときは「性別」へと七変化するわけです。

この考え方を、オブジェクト指向言語における「ポリモルフィズム」の機能に似ている、と指摘するエンジニアもいます。ポリモルフィズムは、多態性や多様性とも呼ばれ、オブジェクトや関数が複数の型に属することを意味する概念です。確かに考え方に共通するところはありますが、実際に両者の間に影響関係があったのかどうかはよくわかりません。単一参照テーブルについては、誰か一人が考え出した方法というわけではなく、開発現場で自然発生的に使われていた技法に名前がついた、というのが正しいようです。

さて、この単一参照テーブルの利点と欠点をまとめると、以下のようになります。

利点

- マスタテーブルの数が減るため、ER図やスキーマがシンプルになる。
- コード検索のSQLを共通化できる。

欠点

- 「コードタイプ」「コード値」「コード内容」の各列とも、必要とされる列長はコード体系によって異なるため、余裕を見てかなり大きめの可変長文字列型で宣言する必要がある。
- 一つのテーブルにレコードを集約するため、コード体系の種類と数の多さによっては、レコード数が多くなり、検索のパフォーマンスが悪化する。

- コード検索のSQL内でコードタイプやコード値を間違えて指定してもエラーになることがないため、バグに気付きにくい。
- ER図がすっきりするとはいっても、ERモデルとしては正確さを欠いており、かえってER図の可読性を下げることになる。

このように、利点に比べると欠点のほうが大きいことがわかります。一見すると良いアイデアのように思われるのですが、実際に使ってみると、なかなか大きな問題を抱えているのです。実際、著者が見た単一参照テーブルでは、「コード値」の列長がたった5バイト（！）で宣言されていたため、7バイトの列長を持つコード体系が登録できませんでした。様々な業務で共通に使うテーブルであることが災いして［※8］、列のサイズ拡張も許されず、結局、別途専用のマスタテーブルを作るという本末転倒な結果に終わりました。

勘どころ 51

テーブルにポリモルフィズムはいらない。

英語ではOne True Lookup Table、略してOTLTと言います。

なにしろ単一参照テーブルの定義上、すべての業務システムがこのテーブルを参照するのです。

次のバッドノウハウは、テーブル分割です。これは、一般的にパフォーマンス向上を目的として実施されることが多く、目的がはっきりしていることもあって、今までに紹介したものに比べると一般にそれほどバッドノウハウとは認識されていません。

実際、テーブル分割の方法には何種類かあり、ものによっては現実的な解であることもあります。しかし、中には絶対に許されないタイプの方法もあるため、注意する必要があります。

テーブル分割の種類

テーブル分割を分類すると、大きく以下の二つに分かれます。

1. 水平分割
2. 垂直分割

今から、それぞれのパターンについて詳しく見ていきますが、最初に簡単に説明しておきます（図7-2）。最初の「水平分割」とは、レコード単位にテーブルを分割す

ることです。テーブルを水平にカットするのでこう呼びます。次の「垂直分割」は、その反対で列単位にテーブルを分割します。

また、厳密には分割ではないのですが、「集約」という方法についてもあわせて本節で解説します。これは、元のテーブルのデータの一部をサマリ（集計）した結果を、新たなテーブルとして保持するパターンです。したがって、テーブルは追加されるだけで元のテーブルの構成が変更されるわけではありません。しかし、開発時にはパフォーマンス向上のための手段として、しばしばテーブル分割と一緒に考えるため、ここで扱います。

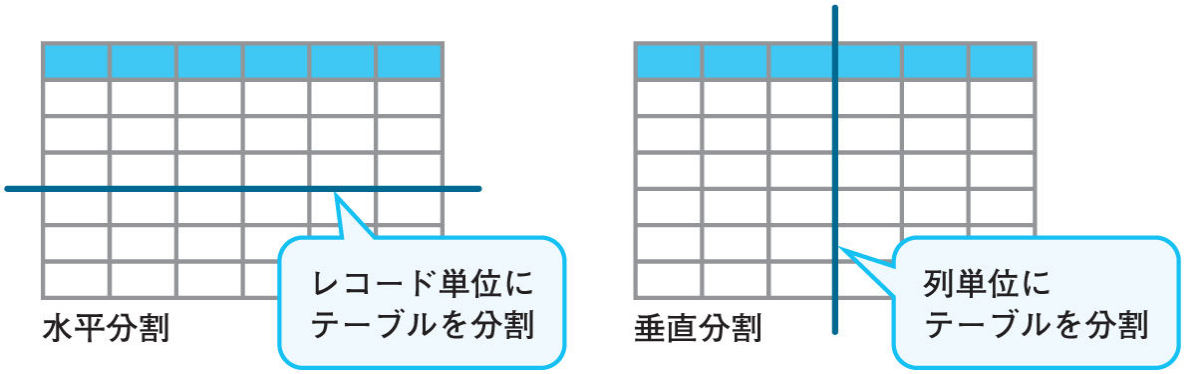


図7-2●テーブル分割の種類

水平分割

水平分割とは、レコード単位でテーブルを分割する手段です。たとえば、会社の1年ごとの売上げを保持する以下のようなテーブルを考えましょう。

売上げ

年度	会社コード	売上げ（億円）
2001	C0001	50
2001	C0002	52
2001	C0003	55
2001	C0004	46
2002	C0001	52
2002	C0002	55
2002	C0003	60
2002	C0004	47
2003	C0001	46
2003	C0002	52
2003	C0003	44
2003	C0004	60

　このテーブルは、サンプルなのでたかだか12行しかレコードを含んでいませんが、実際の業務システムではテーブルに何百万～何十億という数のレコードが含まれま

す。その結果、テーブルにアクセスするSQL文のパフォーマンスが悪化することがしばしば起きます。第2章で学んだように、SQL文のパフォーマンス悪化が起きる最大の原因（ボトルネック）は、ストレージ（多くの場合はディスク）に対する**I/Oコストの増大**でした。その場合、アクセスするデータ量を減らすことが、パフォーマンス改善の手段となります。

　そこで考えられる手段の一つが、SQL文がアクセスするテーブルのサイズを極力小さくしようとすることです。たとえば、SQLが常に1年ごとにしか「売上げ」テーブルにアクセスしないのであれば、次のように年度ごとにテーブルを分割することでパフォーマンスを改善できます。

■「売上げ」テーブルを水平分割（年度ごとに分割）

売上げ（2001）

年度	会社コード	売上げ（億円）
2001	C0001	50
2001	C0002	52
2001	C0003	55
2001	C0004	46

売上げ（2002）

年度	会社コード	売上げ（億円）
2002	C0001	52

2002	C0002	55
2002	C0003	60
2002	C0004	47

売上げ（2003）

年度	会社コード	売上げ（億円）
2003	C0001	46
2003	C0002	52
2003	C0003	44
2003	C0004	60

これは、わかりやすいといえばわかりやすい解決策です。しかし、以下のような重大な欠点があるため、リレーショナルデータベースでは原則禁止とされます。

欠点1 分割する意味的な理由がない

これはすべてのパターンのテーブル分割に該当することですが、このようにテーブルを分割する理由は、正規化の理論からは出てきません。純粋にパフォーマンスという物理レベルの要請によるものなので、逆に言うとその要請がなければ実施する必要のないものです。

欠点2 拡張性に乏しい

この分割がパフォーマンス改善に効果を発揮するのは、全年度のデータを総なめで検索することはない、という前提が成立する場合だけです。そして、その前提が常に成立するかどうかは保証の限りではありません。経年変化を分析したい、といった場合にはこの分割方法では意味がありません。

また、この分割を採用する場合、当然ながら2004年以降のデータもテーブルを分割して保持する必要があるため、テーブルが次々に増えてゆきます。そのたびにアプリケーションも改修が必要となり、拡張性に欠けます。データの形がプログラムを決めるというDOAの原則（第1章）を思い出してください。

欠点3 他の代替手段がある

こうした水平分割の手段として、多くのDBMSが「パーティション」という機能を持っています。パーティション機能を用いることで、テーブルを分割することなく、パーティションキー（「売上げ」テーブルの場合は「年度」）を軸として物理的に格納領域を分離することが可能になります。これによって、SQLがアクセスするデータ量を1/nに減らせます（nはパーティションの数）。

このパーティション機能は、DBMSによっては持っていなかったり、有料オプションだったりするのですが、利用可能な環境においては、水平分割を回避しつつパフォーマンス改善が可能になるため、積極的に利用しましょう。なお、インデックスとパーティションのどちらを使うかで迷うかもしれませんが、一般的にはパーティションはインデックスよりもカーディナリティが小さく、かつ値の変更があまり起きない列をキーにして利用します。典型的な例としては「年」や「都道府県」などカーディナリティが十～数十であるキーが対象になります。インデックスの場合は、もう少しカーディナリティが高くないと、あまり絞り込み効果が得られないことが多いのです。

垂直分割

水平分割がレコードを軸にした分割だったのに対して、垂直分割は列を軸に分割します。例として、第3正規化された状態の「社員」テーブルで考えてみましょう。

社員

会社コード	社員ID	社員名	年齢	部署コード
C0001	000A	加藤	40	D01
C0001	000B	藤本	32	D02
C0001	001F	三島	50	D03
C0002	000A	斉藤	47	D03
C0002	009F	田島	25	D01
C0002	010A	渋谷	33	D04

今、このテーブルに対する検索のSQL文に遅延が発生していて、改善の必要があるとします。かつ、その検索で利用する列は、常に「会社コード」「社員ID」および「年齢」だけであるとします。このとき、次のようにテーブルを分割することでSQL文がアクセスするデータ量を減らすことができます。

■「社員」テーブルを垂直分割

社員1

会社コード	社員ID	年齢
C0001	000A	40
C0001	000B	32
C0001	001F	50
C0002	000A	47
C0002	009F	25
C0002	010A	33

社員2

会社コード	社員ID	社員名	部署コード
C0001	000A	加藤	D01
C0001	000B	藤本	D02
C0001	001F	三島	D03
C0002	000A	斉藤	D03
C0002	009F	田島	D01

C0002	010A	渋谷	D04

このように必要な列だけに絞ってデータを保持した「社員1」テーブルを検索対象とすることで、SQL文のパフォーマンス改善が可能になります（もちろん、ボトルネックがストレージのI/Oコストだった場合ですが）。かつ、この分割は正規化でこそないものの、無損失分解ではあるので、結合によって元の「社員」テーブルを復元することもできます。

しかし、この垂直分割にも、分割することが論理的な意味を持たないという水平分割と同様の欠点があるため、原則利用するべきではありません。特に、垂直分割の場合は、次に紹介する「集約」で代替が可能です。

それでは、最後に第3の手段「集約」を解説しましょう。

集約

集約は、テーブル分割ではなく、むしろテーブル分割の代替案に位置づけられる方法です。その種類は、さらに細かく2種類に分けられます。

1. 列の絞り込み
2. サマリテーブル

列の絞り込み

まず一つ目が、単純に保持する列を絞ったテーブルを作成するタイプです。これは、先ほどの垂直分割に対する代替案に相当します。「社員」テーブルのうち「会社コード」「社員ID」および「年齢」が頻繁に参照される列であるならば、これらの列だけを持った新しいテーブルを追加作成するわけです。オリジナルの「社員」テーブルは残すため、分割ではありません。

■列の絞り込み（データマートの作成）

社員

会社コード	社員 ID	社員名	年齢	部署コード
C0001	000A	加藤	40	D01
C0001	000B	藤本	32	D02
C0001	001F	三島	50	D03
C0002	000A	斉藤	47	D03
C0002	009F	田島	25	D01
C0002	010A	渋谷	33	D04

定期的にデータ同期が必要

社員（年齢のみ）

会社コード	社員 ID	年齢
C0001	000A	40
C0001	000B	32
C0001	001F	50
C0002	000A	47
C0002	009F	25
C0002	010A	33

データマート（マート）

新たに作成した小規模テーブル

このようにして作られる（オリジナルのテーブルと比較すれば）小規模なテーブルを、**データマート**（Data Mart）、あるいは省略して単に**マート**と呼びます。これは、

もともとは大量データを扱うDWH（データウェアハウス）分野で使われていた用語です。

このマートは、非常に便利で、オリジナルのテーブルを意味的に破壊することなくパフォーマンスも向上させられるので、実際の開発においてもよく利用されています。便利すぎてマートばかりたくさん作られてしまい、ストレージの容量を圧迫するという本末転倒なケースも時折見ます。また、マートを利用する際に注意すべき点がもう一つあります。それが**データ同期**の問題です。

今、オリジナルのテーブルとマートは、部分的に同じ列を共有しています。このとき、たとえば「年齢」列は、当然のことながら社員が年齢を重ねるたびに値を更新していく必要があります。そのため、オリジナルの「社員」テーブルの「年齢」列が更新されたら、マートの「年齢」列も更新しなければなりません。

問題となるのは、マートの更新タイミングです。このタイミングが短いほど、オリジナルのテーブルと齟齬がある期間も短くなるので、データ精度が高く機能的には好ましいのですが、更新タイミングが短いほど更新処理の負荷が上がり、もともと解決するはずだった性能問題をかえって悪化させてしまう危険もあります。したがって、多くの場合、マートの更新は1日1回～数回程度の頻度で一括更新（**バッチ更新**）されています。しかしこの場合、オリジナルのテーブルとマートのデータ不整合の期間が長くなるため、機能的に問題がないか、要件と照らし合わせながら慎重に検討する必要があります。

サマリテーブル

サマリテーブル（summary table）も集約の一手段です。列の絞り込みと違うのは、サマリテーブルは集約関数によってレコードを集約した状態で保持することです。たとえば、会社別に社員の平均年齢が必要な業務があったとします。このと

き、もちろん毎回「社員」テーブルに対してSELECT文でアクセスして集約を行なっても良いのですが、テーブルの規模が大きくなると、その集約処理のコストが大きくなり、実行時間が長くなります［※9］。

そこで、次のような事前に集約を行なったテーブルを作っておくのです。社員の会社別平均年齢を求めたければ、このテーブルに対する単純なSELECT文で求められます。

■サマリテーブル

社員

会社コード	社員 ID	社員名	年齢	部署コード
C0001	000A	加藤	40	D01
C0001	000B	藤本	32	D02
C0001	001F	三島	50	D03
C0002	000A	斉藤	47	D03
C0002	009F	田島	25	D01
C0002	010A	渋谷	33	D04

集約

SQL

```
SELECT 会社コード, AVG(年齢) AS 平均年齢
  FROM 社員
GROUP BY 会社コード
```

社員平均年齢

会社コード	平均年齢
C0001	41
C0002	35

サマリテーブル

この「社員平均年齢」テーブルのサイズは、行列ともに元の「社員」テーブルより小さくなり、アクセスするときのI/Oコストを大きく削減する効果があります。オリジナルのテーブルを変更することもないので、分割によるデメリットもありません。

ただし、この方法も先ほどの列の絞り込みと同じデメリットを共有しています。つまり、更新のタイムラグによってデータの整合性がとれない時間帯が生じるデータ同期の問題は、サマリテーブルの場合も同様です。また、一般的にサマリテーブルはオリジナルのテーブルに比べてかなりサイズが小さくなるものの、新たにテーブルを追加するわけですから、ストレージの容量を消費するのも事実です。

shardingとカラムベース　COLUMN

本文において、テーブルの水平分割と垂直分割をバッドノウハウとして取り上げました。このコラムでは、この二つの発想とよく似た技術について説明したいと思います。それが、**sharding**（シャーディング）**とカラムベースデータベース**（column-base database）です。どちらも新しいデータベースの物理的なアーキテクチャとして期待されていますが、驚かないでください。なんとその発想は、まさに水平分割と垂直分割なのです。

● sharding

shardingとは、適当な和訳がない言葉ですが、これはまさに水平分割、つまりレコードを物理的に分離して格納することでI/O分散を図るパフォーマンス向上手段です。リレーショナルデータベースのパーティションと違うのは、パーティションによって分割されたテーブルは、論理的にはあくまで単一のテーブルとして扱うことができるのに対し、shardingによって分割されたテーブルは、論理的にも物理的にも異なるテーブルとして扱われることです。つまりshardingは、基本的にシェアードナッシング型のアーキテクチャ（第2章の演習参照）を前提としている、というこ

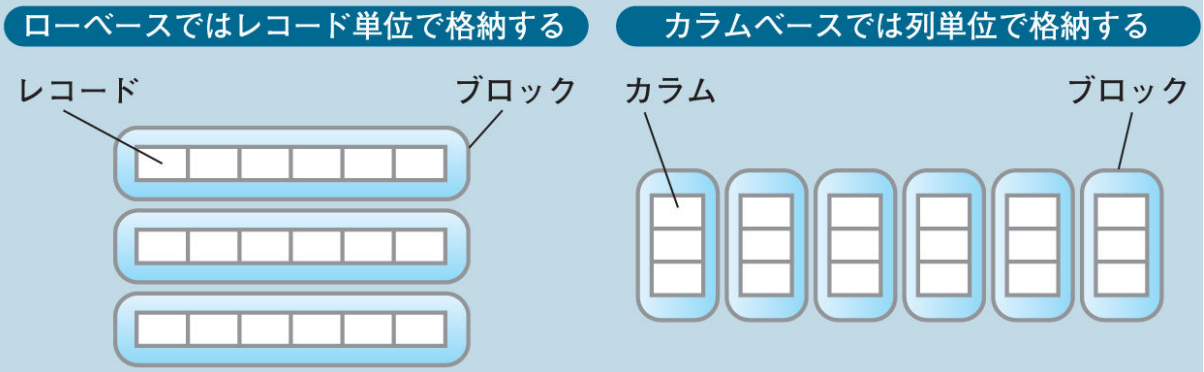
とです。I/Oを完全に分離できるため、性能のスケーラビリティが高く、Googleの

ような巨大検索エンジンのデータベースに使われています。

● カラムベースデータベース

もう一方のカラムベースデータベースについて説明するには、それと対になるローベースデータベース（row-base database）について説明する必要があります。

ほとんどのDBMSは、ローベースデータベースです。それはつまり、「ロー（行）」を一つの物理的な格納単位［※10］としている、ということです（図A）。



図A●ローベースとカラムベース

ローベースでは、たとえ一つのSQL文で利用する列が一列だけであっても、レコード全体を読み出す必要があります。これは、DBMSが読み出す単位が「ブロック」だからです。器用にブロック内部の特定の一列だけを読み出すことはできません。

これに対し、カラムベースの場合、一つのSQL文で利用する列が限られている場合、その特定の列だけを読み出すことで、I/O量を削減できるのです。これは、パフォーマンス上大きなメリットです。

このカラムベースのアイデアは、「テーブルは多くの列を持っていても、一つのSQL

このカラムベースのアイデアは、「テーブルは多くの列を持っていても、一つのSQL文で利用する列数は限られている」という洞察に基づいています。したがって、「SELECT *」のように選択する列数が多い場合、カラムベースの恩恵を受けることはできません。

ここまで読めば、このカラムベースが、垂直分割の発想であることを理解していただけるでしょう。カラムベース／ローベースというのは、あくまでDBMSの物理的なアーキテクチャレベルの構造であるため、テーブルを利用するユーザーが意識することはありません。カラムベースストアのDBMSは、DWH/BI用途を除いて、まだあまり普及していませんが、これが一般的になってくれば、データベースの物理設計方針も大きく変わることになるでしょう。

本書はSQLについては深く取り上げませんが、一般にSUMやAVGなど集約関数を使うと、対象レコードのソート処理が発生するため、処理時間が長くなります。

DBMSによって呼び方が違いますが、この単位を「ブロック」や「ページ」と呼びます。図Aでは簡略化して、一つのブロックに一つのレコードを対応させましたが、実際には一つのブロックに複数レコードが格納されることもあります。

データベースでテーブルを作る際に考える要素の一つとして、どの列にどのようなデータ型を採用するか、という問題があります。代表的なデータ型には、以下のようなものがあります。

- 文字列（CHAR、VARCHAR）
- 数値（INTEGER、LONG、REAL）
- 日時（DATE、TIMESTAMP）

これ以外にもDBMSは、標準的なものから独自仕様のものまで、多くのデータ型を持っています。中には、7-2節で取り上げた配列型のように、標準的なデータ型でありながら、使用を推奨できないものもあります。

それ以外のデータ型については、データの属性に応じて適切なものを採用していくことになりますが、中には特定の目的のためには利用するのが望ましくないデータ型があります。その点について、本節で注意を喚起しておきたいと思います。

問題となるのは、キーに使用される列に対するデータ型です。ここで言う「キー」とは、以下の二つを指しています。

- 主キー、外部キーなどデータベースの機能で設定されるもの
- テーブルの結合条件で使用される列（結合キー）

この2種類のグループは、互いに重なり合っていることがほとんどです。結合で使われるキーというのは、当該テーブルの主キーや外部キーであるのが普通です。

この二つの用途で使われる列には、明らかに使ってはいけないデータ型があります。それが、**可変長文字列**（VARCHAR）です。理由は、二つあります。一つは、可変長文字列の列は、キーが満たすべき条件である**不変性**（Stability）を備えていないことです。そして、もう一つが、固定長文字列（CHAR）との混同です。

キーは永遠に不変です！

可変長文字列が使用されるのは、具体的にどのような種類のデータでしょうか？ 本書で取り上げてきた例で言えば、会社名や社員名、部署名などが挙げられます。つまりは、何かの「名前」に使われることが多いということです。名前は文字数が固定ではありませんから、名前の列に可変長文字列が採用されるのは自然です。

しかし、「名前」をキーに使うことは非常に危険です。というのも、名前は長期的なスパンで考えれば、変動する可能性が高いからです。人名の場合が特に顕著ですが、結婚すれば姓が変わりますし、まれに成人してから自分で改名する人もいます。会社名はそれほど短いスパンで変わることはないかもしれませんが、やはり可能性はありますし、部署名はもっと頻繁に、組織変更のたびに変わります。

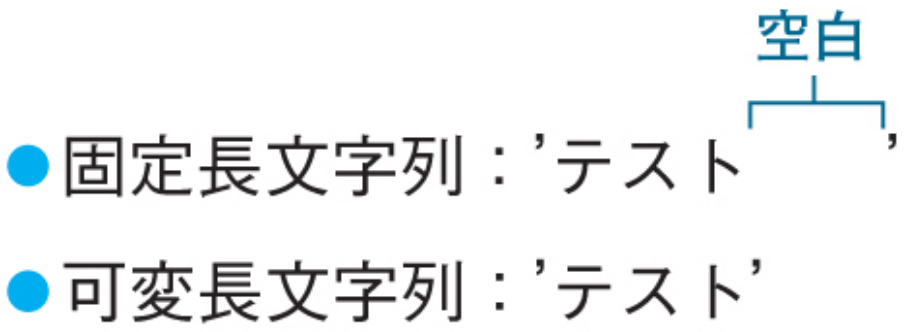
このようにコロコロ値が変わる列をキーにすると、データの更新処理が多く発生することになるため、システムの安定的な運用とパフォーマンスの両面でマイナスです。

勘どころ 52

可変長文字列は不変性がないためキーには不向き。

同じデータを意味するキーは同じデータ型にすべし

有名な話ですが、固定長文字列と可変長文字列は、同じ文字列を保持しようとしても、物理的には同じ「値」にならないことがほとんどです。それは、固定長文字列が空白による穴埋め（パディング）をするからです。



そのため、ある結合キーについて、一方のテーブルが固定長文字列で宣言されていて、他方のテーブルでは可変長文字列で宣言されていると、単純に列同士を比較するとアンマッチになってしまいます。

この問題は、文字列型と数値型など異種のデータ型の間でも起きます。DBMSによっては暗黙のデータ変換を行なってくれることもありますが、それでも、固定長文字列の「001」と数値型の「1」は、型変換だけでは不十分で、数値型のほうを「0埋め」しなければマッチしません。こうした厄介な問題にSQLコーディングの時点で頭

を悩ませるぐらいなら、最初からデータ型を固定長文字列型に揃えておくべきなのです。

勘どころ 53

キーには固定長文字列の「コード」列が望ましい。

バッドノウハウ最後の一つは、ダブルマスタです。これは他のバッドノウハウのように一般的な名称ではなく、著者が名づけた造語です。

ダブルマスタとは、その名のとおり、同じ役割を果たすはずのマスタテーブルが二つ存在するようなケースです。たとえば、顧客を管理するためのテーブル「顧客マスタ」が、以下のように二つ存在するような場合です。

■ダブルマスタ

顧客マスタA

顧客コード	顧客名
C001	山田太郎
C002	中島義人
C003	新藤啓太
C004	中村佳織

顧客マスタB

顧客コード	顧客名

C001	山田太郎
C002	中島義人
C003	新藤啓太
K001	小嶋雅夫

ダブルマスタはSQLを複雑にし、パフォーマンスを悪化させる

顧客マスタAと顧客マスタBは、同じ顧客の情報も重複して持っていますが、片方のマスタにしかない顧客の情報もあります。顧客マスタAの「C004 中村佳織」と顧客マスタBの「K001　小嶋雅夫」がそうです。このように必要な情報が複数のテーブルに分散されていると、結局のところ、顧客全員の情報を得るには二つのマスタを結合しなければなりません。

この結合は、SQLレベルで言うならば完全外部結合またはUNIONによって実現できるのですが、いずれもコストの高い処理であり、パフォーマンスはマスタが一つだけの場合よりも悪くなります。

■完全外部結合の例

```
SELECT COALESCE(A.顧客コード, B.顧客コード),
       COALESCE(A.顧客名, B.顧客名)
```

```
  FROM 顧客マスタA A FULL OUTER JOIN 顧客マスタB B
    ON A.顧客コード = B.顧客コード;
```

　完全外部結合は、通常の外部結合がマスタにする片方だけのテーブルのレコードを保存するのに対し、両方のテーブルのレコードを保存します（いわばレコードを「完全」に保存するため、完全外部結合と呼ばれています）。COALESCE関数は、引数を左からスキャンして、最初に見つけたNULLではない値を返します。これは、どちらかのテーブルにしか存在しない顧客のレコードにNULLが生じることを防ぐための措置です。

　一方、UNIONを使った例は以下のようになります。こちらは単純に、両方のテーブルを足し合わせるだけです。

■UNION の例

```
SELECT A.顧客コード,
       A.顧客名
  FROM 顧客マスタA A
UNION
SELECT B.顧客コード,
       B.顧客名
  FROM 顧客マスタB B;
```

　いずれも結果は次のようになります。

結果

```
顧客コード 顧客名
-------- --------
C001     山田 太郎
```

C002　中島 義人
C003　新藤 啓太
C004　中村 佳織
K001　小嶋 雅夫

ダブルマスタはなぜ生じるのか

このダブルマスタは、最初に見たときに、むしろどうやったらこのようにマスタが二つに分かれるのか、といぶかしく思うかもしれません。実際、普通に設計をしている限り、このような不可解な状態には陥りません。実は、ダブルマスタが生じる直接的な理由は一つで、それは、もともと別のシステムで利用されていたマスタ同士が、システム統合によって同じドメインに存在するようになった場合なのです。

たとえば、金融機関や小売業の企業は、それぞれ独自に顧客マスタを管理しているでしょう。そしてその企業の内部に限れば、顧客の一元管理は実現できていたわけです。しかし、近年盛んな企業同士の統廃合によってシステム統合が行なわれると、それぞれの企業が管理していたデータも統合の必要が出てきます。

勘どころ 54

ダブルマスタはシステム統廃合で起きることが多い。

そうすると、互いに同じような役割を持っているテーブルが複数存在することになります。そのとき、きちんとデータを精査してエンティティの統廃合を行なえば、ダブルマスタが生じることはないのですが、それには時間と人手がかかります。データモデル

を一新すれば、データベースに接続している既存アプリケーションにも改修が及ぶでしょう。そのようなわけで、往々にしてこの部分のタスクに手を抜いて、既存のエンティティを並べただけ、という名ばかりの統合が行なわれることも珍しくないのです。

このようなデータ精査の作業を、データクレンジングと呼びます。この作業は、システム統合や既存システムの改修時において重要な役割を果たすため、8-6節で詳しく取り上げます。

バッドノウハウのどこが悪いのか？ COLUMN

本章ではバッドノウハウを見てきましたが、読者のみなさんはどのような感想を持ったでしょうか？

まず大多数の方の反応は、「正気の沙汰とは思えない」というものでしょう（そうであることを著者は切に願っています）。しかし中には、「このぐらい別にいいじゃないか。だって、実際にこういうデータ構造の設計で動いているシステムだって、たくさんあるんでしょう？」と思った人もいるかもしれません。「バッドノウハウ、アンチパターンと世間では言うけれど、そんなに悪いことなのか」と。この理由は一理あるように感じるかもしれません。確かに、世の中には、お世辞にも「きちんとしている」とは言いがたいデータ構造を持ったシステムはたくさんあります。ということはつまり、設計が汚くてもシステムを作り、それを稼動させ、長期にわたって運用していくことは、不可能ではないということです。だったらバッドノウハウに何の問題があるでしょうか？

このコラムでは、その疑問に答えるため、バッドノウハウのどこが「バッド」であるのかを、整理しておきたいと思います。

バッドノウハウがバッドである理由❶

バッドノウハウがバッドである理由❶

可読性——エンジニアはそんなに頭が良くない

非オーソドクスなデータモデルを利用しても、確かにシステムを作ることは不可能ではありません。しかし、この主張はコストの問題を度外視しないと成り立ちません。バッドノウハウを採用した場合の開発および運用にかかるコストは、そうでない場合に比べて何十倍にも高くなります。その最大の理由は、バッドノウハウが人間の直観に反するものであるため、エンジニアやプログラマの設計に対する理解を妨げるからです。

たとえば、ダブルミーニングや単一参照テーブルは、非常にトリッキーで、最初に設計した人間以外には意図を理解することが困難です。テーブル分割のように、システムのパフォーマンスを追求するあまり可読性を下げるタイプの設計も同様です。こうした人間にとって「わかりにくい」システムは、結局のところ開発するエンジニアやプログラマの理解やコミュニケーションを阻害することにつながり、バグを生み出し開発効率を落とす温床になるのです。

バッドノウハウがバッドである理由❷

設計変更の難しさ

第二の理由は、本文でも触れたことですが、リレーショナルデータベースのモデルを、開発の後工程の段階（具体的にはテストなど、アプリケーションの開発終了後）になってから修正することが難しいことです。いざテーブル構成に後から手を加えようとすると、せっかく作ったアプリケーションにまで修正が及ぶことは必至です。ある程度アプリケーションの側で柔軟な作りをすることで吸収は可能ですが、原則として「ERモデルの手戻り」はかなり代償が高くつきます。

三途の川で石を積むに等しい——というのは少し自虐的すぎる表現かもしれませんが、実際、ERモデルまで修正が必要な設計変更が繰り返されると、アプリケーションを作っては直し作っては直し…を延々続ける破目に陥ったりして、文

リファクタリングを行っては直し行っては直し、を延々続ける破目に陥ったりして、文字通り地獄を味わえます（その地獄を、IT業界では「**デスマーチ**（死の行軍）」と呼んでいます）。

この問題は、時としてリレーショナルデータベースそのものの欠点として語られることもあります。リレーショナルデータベースがデータ構造の変更コストが高い、というのは事実で、もっと柔軟にデータ構造を変更可能なデータベースの研究開発も行なわれています。しかし、そういう問題と、変更コストを計算しないこととは、無関係の話です。

バッドノウハウがバッドである理由❸

データ構造がコードを決めるのであって、その逆ではない

システム開発の本質的な問題を明晰（めいせき）に論じた名著『人月の神話』において、フレデリック・ブルックスはこう言っています。

> 私にフローチャートだけを見せて、テーブルは見せないとしたら、私はずっと煙に巻かれたままになるだろう。逆にテーブルを見せてもらえるなら、フローチャートはたいてい必要なくなる。それだけで、みんな明白に分かってしまうからだ。［※11］

フローチャートというのは、文字通りデータの「流れ」を記した処理設計書に相当します。ブルックスがこの本を書いた1975年当時、リレーショナルデータベースはまだアイデアが登場したばかりで、あまり一般的ではありませんでした。つまり、ブルックスは別にリレーショナルデータベースを念頭においてこの文章を書いたわけではないのです。

にもかかわらず、彼の言葉は現在のリレーショナルデータベースの設計にもまった

にも分かりますが、彼の言葉は現在のリレーショナルデータベースの設計にもまったく当てはまります。それは、プログラミングおよびシステム開発という仕事全体が、データによって規定されるからです。下手くそなデータ構造に対して、エレガントなプログラミングを行なうことは不可能なのです。データ構造がダメな状態で、プログラミングによって挽回することはできません。システム開発において最も重要なのは、コーディングではなくデータ設計なのです。7-7節では、ダブルマスタを採用した場合に必要になる、非効率なSQLを紹介しました。こうした簡潔でないコードは、例外なくパフォーマンスが悪く、保守にも苦労する、ということの好例です。

以上の三つの理由から、バッドノウハウは決して採用すべきでないことがわかります。納得いただけたでしょうか？

さて、本章では論理設計における「やってはいけない」バッドノウハウを紹介してきました。

著者も普段から、こうしたバッドノウハウの非を鳴らす啓蒙活動は行なっているのですが、みなさんも、是非バッドノウハウの根絶に力を貸してください。そしてともに、世の中のデータベースをきれいな設計で埋め尽くしましょう。

演習問題

演習 7-1 パーティションの特徴

7-5節ではバッドノウハウ「水平分割」の代替手段として、「パーティション」機能の存在を取り上げました。このパーティションには、（ちょうどインデックスがそうであったように）いくつかの種類があります。以下に代表的なパーティシ

ョンを挙げます。これらがどのような利点と欠点を持っているかを整理してください。

- **レンジパーティション**
- **リストパーティション**
- **ハッシュパーティション**

演習 7-2 マテリアライズドビューの機能

7-5節ではバッドノウハウ「水平分割」の代替手段として、集約、特にデータマートを取り上げました。このデータマートを実現する手段として、DBMSは「マテリアライズドビュー」という機能を提供しています。これがどのような機能であるか、調べてください。

フレデリック・P・ブルックス, Jr.『人月の神話』（ピアソン桐原、2010） p.95

第 8 章

論理設計のグレーノウハウ

データベースの論理設計には、明確にバッドノウハウとまでは言えないものの、きわどいライン上に位置するグレーゾーンが存在します。グレーゾーンは、利点と欠点のバランスが拮抗しており、使用の際は慎重な判断が求められます。

学習のポイント

- バッドノウハウ以外にも、違法スレスレの「グレーゾーン」の設計、グレーノウハウがあります。
- グレーノウハウは、節度ある利用にとどめる限り、有効に作用する「良薬」になりえますが、利点と欠点のバランスを慎重に考える必要があります。
- グレーノウハウには、代理キー、列持ちテーブル、アドホックな集計キー、多段ビューがあります。

法律の世界には、「グレーゾーン」と呼ばれる領域があります。明らかな違法行為（ブラック）ではないのだけど、大手を振って合法（ホワイト）とも断言しにくい、法のボーダーライン上に位置するような行為を指す言葉、それが「グレー」です。

システムの世界にも、やはりグレーゾーンが存在します。バッドノウハウとはっきり断定することこそできないものの、無神経に使うと開発や運用に支障をきたすような、そんな毒を含んだ設計のことです。これを**グレーノウハウ**と呼びます（この言葉は著者の造語なので、バッドノウハウやアンチパターンほど一般的ではありません）。

こうしたグレーノウハウは、バッドノウハウ以上に広く利用されています。そのうえ、バッドノウハウ以上に使う側にあまり「悪いことをやっている」という罪の意識がないため、その危険性が意識されないまま使われていることもしばしばです。バッドノウハウを劇薬とするなら、グレーノウハウは用法用量を守れば役に立つけれど、間違った使い方をすると副作用の強い薬のようなものです。

本章では、こうしたグレーノウハウを取り上げ、その利点と欠点を整理します。読者のみなさんは、まず正確な知識を持って、利点と欠点を理解したうえで、正しい用法で利用するようにしていただきたいと思います。

8-2 代理キー～主キーが役に立たないとき

リレーショナルデータベースのテーブルにおいて、原則主キーが必ず必要であることは、みなさんもすでにご承知でしょう［※1］。しかし、その主キーを決めるのが容易ではないケースがあります。

主キーが決められない、または主キーとして不十分なケース

それはどんなケースかというと、大きく以下の三つのパターンが考えられます。

パターン1 そもそも入力データに主キーにできるような一意キーが存在しない。
パターン2 一意キーはあるが、サイクリック［※2］に使いまわされる。
パターン3 一意キーはあるが、途中で指す対象が変化する。

パターン1 そもそも入力データに主キーにできるような一意キーが存在しない

これは最も単純ですが、同時に言語道断なケースです。テーブルを作ろうにも、一意キーが存在しないのでは主キーが決められません。信じがたいことかもしれませんが、業務システムによっては、実際にこういうデータを許容しているケースがありま

す。たとえば、入力インタフェースではレコードの二重登録を許容していて、後でレコードの重複を排除する処理が行なわれるような場合です。

こういうケースでは、原理原則を言えば、そもそもそんなデータをテーブルに投入するべきではありません。前段階のアプリケーションできちんとレコードを一意にして、「きれいな」状態にしてからデータベースへ投入するべきです。このように、そのままではデータベースに登録できない状態のデータを、「きれいに」する処理を、その名もずばり「データクレンジング（データの掃除）」と呼びます。この処理にどのようなものがあるかについては、8-6節で取り上げますので、そちらを参照してください。

パターン2 一意キーはあるが、サイクリックに使いまわされる

これは、主キーの値がすべて使われてしまった場合に、既存の値が使われるケースで起こります。たとえば、市町村の情報を管理するテーブルを考えましょう。このテーブルの主キーとして、「市町村コード」という「A＋3桁の数字」からなるコードを使うとします。すると、A000 ～ A999までの1000個の市町村を登録してしまうと、そのままでは次に1001個目の市町村を登録することができません。

■市長村コードに重複が発生するケース

市町村

市町村コード	市町村名	人口（人）
A000	A 市	100,000
A001	B 市	120,000
A002	C 町	30,000
A003	D 村	10,000
⋮		
A999	Z 市	15,000

B市が廃止

Q市の情報を追加

市町村コード	市町村名	人口（人）
A000	A 市	100,000
A001	Q 市	180,000
A002	C 町	30,000
A003	D 村	10,000
⋮		
A999	Z 市	15,000

B市の市町村コードを再利用

このとき、「市町村コードの桁数を拡張する」という選択をとることができれば問題ないのですが、こうしたコードは業務ルール（＝ビジネスロジック）によってシステムとは無関係に体系が決まっていることがあり、「すでに廃止された市町村の番号を使いまわす」という選択を余儀なくされることもあります。これでは、市町村コードを使った履歴管理を行なうことができず、主キーの役目を果たしません。同じ「A001」という市町村コードが、ある年度までは「B市」を意味していたのが、その年を境に「Q市」になってしまう、ということが発生します。常に最新のコードしか使用し

ないのならこの運用でも問題はありませんが、時系列で過去のデータを取得したい場合などには、履歴情報を残さなければ対応できません。

パターン3 一意キーはあるが、途中で指す対象が変化する

このケースは、主キーの指す対象が途中で変わってしまうという点で、パターン2とよく似ています。ただ、その理由が値の枯渇ではないだけです。たとえば、C 町とD村が合併することになったとします。実質的にはD村がC町に吸収される形をとり、合併後の名前は引き続き「C町」が使われることになったとします。

■一意キーの指す対象が変わるケース

市町村

市町村コード	市町村名	人口（人）
A000	A 市	100,000
A001	B 市	120,000
A002	C 町	30,000
A003	D 村	10,000
⋮		
A999	Z 市	15,000

C町にD村が吸収合併に

↓

市町村コード	市町村名	人口（人）
A000	A 市	100,000
A001	B 市	120,000
A002	C 町	40,000
⋮		
A999	Z 市	15,000

この場合、「A002」というコードは、表向きは継続的に「C町」を指しています。しかし、その内実は合併を境に大きく変わっています。その証拠に、合併後の人口は合併前のC町（30, 000人）とD村（10, 000人）を合計した40, 000人になっています。本来であれば、やはりこの場合も合併前と合併後の情報は分けて履歴管理することが望ましいのですが、このコード体系ではそれができません。

代理キーによる解決

上記のような問題を解決する手段として利用されるのが、**代理キー**（surrogate key）です。「サロゲートキー」とも呼びます。これは名前のとおり、入力データに最初から存在しているキー［※3］の「代理」として新たに追加するキーです。ためしに、先ほどの「市町村」テーブルに、代理キーとして「市町村管理コード」という整数型の代理キーを追加してみましょう。

すると、たとえば**パターン2**のケースは、以下のように市町村管理コードを主キーとして使うことで、市町村コードに重複が発生しても問題がなくなります。

■代理キーの利用

市町村

市町村管理コード	市町村コード	市町村名	人口（人）
0	A000	A市	100,000
1	A001	B市	120,000
2	A002	C町	30,000
3	A003	D村	10,000
⋮			
999	A999	Z市	15,000

代理キー

Q市の情報を追加

市町村管理コード	市町村コード	市町村名	人口（人）
0	A000	A市	100,000
1	A001	B市	120,000
2	A002	C町	30,000
3	A003	D村	10,000
⋮			
999	A999	Z市	15,000
1000	A001	Q市	180,000

市町村コード「A001」が再利用されて重複が発生する場合にも、一意な連番を割り振ることに決めた市町村管理コードが一意に保たれています。同様のことが、パターン3のケースに対しても当てはまります。このように、**人工的なキー**をシステム側で付与してやることで、自然キーを主キーに選んだ場合に発生する不都合を解消することができるのです。

これまでの話を読んだ限りでは、代理キーが非常に便利な機能であるように思ったかもしれません。実際、入力データの仕様と独立に、データベース側で主キーを設定できるため、面倒な業務上の仕様調整を省略できます。また、代理キーを使ったことによる余禄（主目的とは違う恩恵）として、主キーが多くのキーからなる複合キーの場合に、SQLのWHERE句における条件記述をシンプルにできる、という効果もあります。

しかし、一般的な原則としては、極力代理キーの使用は避けて、自然キーによる解決を図るべきです。その主な理由は、代理キーがそもそも論理的には不要なキーのため、論理モデルをわかりにくくしてしまうからです。業務的には不要なキーであるため、代理キーが何の役割を果たしているのかは、ER図を一読してもわかりません。つまり、代理キーは本来「使わなくても何とかなる」道具なのです。実際、まともな理論家ならば、サロゲートキーを推奨することはしません。ジョー・セルコは「オートナンバーを主キーに使うことは、データモデルを欠いている証拠だ」と、自然キー以外を主キーに使うことを厳しく批判しています。

自然キーによる解決

それでは、代理キーを使わず、自然キーだけで解決するにはどのようにすれば良いか、という方法を解説します。先ほどの自然キーを主キーとする場合に問題が発生する三つのパターンを再度取り上げましょう。

パターン1 については、残念ながら自然キーに主キーが存在しないケースなので、データベース側で打つ手はありません。業務仕様を調整するか、データベースに投入

される前のアプリケーションでデータが一意になるよう整形する対策を行なうのみです。

一方、**パターン2** と**パターン3** のケースについては、基本的に自然キーによる解決が可能です。その方法は、履歴管理のための時間を表わす列を追加する方法です。この時間を示す列には、**タイムスタンプ**と**インターバル**の2種類があります。**パターン2** のケースを例に考えてみましょう。

タイムスタンプ

ここで、「B市」が廃止されたのが、2007年だとします。2007年以降は、それまで「B市」を意味していた「A001」という市町村コードは、新たに「Q市」を意味するようになります。2005 ～ 2007年の3年間のデータを「市町村」テーブルに登録すると、以下のようになります。簡略化のため、市町村の集合に変化はなく、人口も3年間変わらなかったとします。

■タイムスタンプの利用

市町村

年度	市町村コード	市町村名	人口（人）
2005	A000	A 市	100,000
2005	A001	B 市	120,000
2005	A002	C 町	30,000
2005	A003	D 村	10,000
2006	A000	A 市	100,000
2006	A001	B 市	120,000
2006	A002	C 町	30,000
2006	A003	D 村	10,000
2007	A000	A 市	100,000
2007	A001	Q 市	120,000
2007	A002	C 町	30,000
2007	A003	D 村	10,000

B市からQ市に変わった

タイムスタンプを表わす列

ご覧のように、2005年、2006年、2007年と、それぞれのタイミングにおける市町村のレコード集合を用意してやるわけです。データのスナップショット（断面）を必要な数だけ用意する方法と言ってもいいでしょう。それぞれの断面が履歴の役割を果たします。ここで付け加えた「年度」列は、連番という無味乾燥な代理キーと違って、きちんと業務的に意味を持つ列（自然キー）です。

このようにいつ時点のデータであるかを示すタイムスタンプがあれば、直近の情報だけでなく、過去にさかのぼってデータを調べることも可能になります。2007 年には「Q市」を意味する市町村コード「A001」が、2006年以前は「B市」を意味していたことが、非常にわかりやすい方式です。

このタイムスタンプ方式の利点は、データの形式が単純なので利用する際も、SQLの条件が簡単になるなど、そのシンプルさにあります。一方、必ず一つのスナップショットに含まれる市町村をフルセットで保持しなければならないので、レコード数が増えます。特に、1年に1回程度の変更ならまだ良いのですが、毎月や毎週変更されるようなデータの場合は、レコード数が膨大になってしまうでしょう。

インターバル

時間の情報を保持するもう一つの方法は、インターバルです。これは、先ほどのタイムスタンプがある「時点」を意味したのに対し、データの有効な「期間」を表わす方法です。インターバルを使った「市町村」テーブルは以下のようになります。

■インターバルの利用

市町村

開始年度	終了年度	市町村コード	市町村名	人口（人）
1945	9999	A000	A市	100,000
1998	2006	A001	B市	120,000
1955	9999	A002	C町	30,000
2005	9999	A003	D村	10,000
2007	9999	A001	Q市	120,000

B市は
1998 ～ 2006年

Q市は
2007年以降

インターバル（有効期間）を表わす列

今度は、「開始年度」と「終了年度」という二つの時間列を持つことで、データの有効期間を示しています。終了年度に「9999」という最大値を入力している市町

村は、まだ存続していることを示しています。この場合、NULLを使っても良いのですが、以下に見るように、SQLでの使い勝手を考えると、入力上許される最大値を利用するほうが便利です。

　このインターバル方式は、タイムスタンプ方式の欠点を克服できます。スナップショット別にデータを持つ必要はなく、データの有効期間が切れたら既存のレコードの「終了年度」に終了期限を入力するだけなので、全体としてレコード数を少なく抑えられます。これはデータ変更が頻繁に発生する場合には大きな利点です。

　一方の欠点は、データを利用する際に、SQLで必ず範囲指定の条件を入れる必要があるので、SQLが少し複雑になることです。しかし、これもあくまで相対的には、という話で、それほど難しいことではありません。サンプルのSQLを示すと、2007年時点で存在している市町村を選択するSELECT文は、以下のようになります［※4］。

■タイムスタンプ方式

```
SELECT 市町村コード,
       市町村名
  FROM 市町村
 WHERE 年度 = 2007;
```

■インターバル方式

```
SELECT 市町村コード,
       市町村名
  FROM 市町村
 WHERE 2007 BETWEEN 開始年度 AND 終了年度;
```

もし終了年度の最大値がNULLで格納されていた場合は、上記のSQLにおいて「終了年度」列のNULLを適当な最大値に変換する面倒が発生します。それならば、最初から最大値を登録しておくほうが簡単です。

どちらも結果は、次のようになります。

結果

```
市町村コード 市町村名
---------- ------
A000       A市
A002       C町
A003       D村
A001       Q市
```

2007年度の市町村コード「A001」は、すでにB市からQ市に変わっています。その結果が、正しく表示されていることがわかります。

オートナンバリングの是非

代理キーと自然キーに関する議論の結論としては、前節までで見たように、「可能な限り自然キーを使う」という一点に尽きます。しかし、現場の開発においてはやむをえず代理キーを使わざるをえない場合もあるでしょう。本節では、いざ代理キーを使うとなった場合の、具体的な実装方法について考えてみたいと思います。

代理キーとして「市町村管理コード」を採用した、先ほどの「市町村」テーブルを例にとります。

■代理キーを利用した「市町村」テーブル

市町村

市町村管理コード	市町村コード	市町村名	人口（人）
0	A000	A 市	100,000
1	A001	B 市	120,000
2	A002	C 町	30,000
3	A003	D 村	10,000
	⋮		
999	A999	Z 市	15,000

代理キーに求められる要件は、基本的に一意性のみです。キーの値自身に意味が求められることは、ほとんどありません。このような、システム側の裁量で体系を決めることができるという自由度の高さが、代理キーの便利さの一つでもあります。

代理キーを実装する際、しばしば利用される方法が「オートナンバリング」です。これは、1レコードに一意な（通常は整数型の）数値を自動的に割り振る方法です。「市町村」テーブルにおいても、「市町村管理コード」には1から始まる整数値を割り当てています。

このやり方は名案のように思えるかもしれませんが、問題はどうやって数値を払い出すか、です。主キーとして使うことから、次の二つの要件が守られなければいけません。

要件1 重複値が生じないこと（一意性の保証）

要件2 歯抜けが生じないこと（連続性の保証）

二つの要件のうち、一意性が保証されなければならないのは、当然のことです。「50」という値が重複して払い出されては、主キーの一意性違反によってレコード追加時にエラーになっていまいます。連続性の保証については、もう少し微妙です。もしかすると、一意的でありさえすれば、連続的な値でなくても用が足りる場合もあるでしょう。厳密に連番である必要があるかどうかは、業務要件に依存します。

ともあれ、上記二つの要件（少なくとも一意性の要件）をクリアするには、二つの方法が考えられます。それが、データベースの機能を利用する方法と、アプリケーションを利用する方法です。それぞれ、どのような方法で、どのような利点と欠点があるかを見ていきましょう。

オートナンバリングの実現方法❶ ～データベース機能

まずデータベースの機能を利用する方法から説明すると、実は、標準SQLには、このオートナンバリングを実現するために、以下の機能が定義されています（どちらもSQL:2003で追加）。

- シーケンスオブジェクト
- ID列

► シーケンスオブジェクト

シーケンスオブジェクトは、「順序」という名前の意味するとおり、これにアクセスすることで一意な連番を払い出すオブジェクトです。SELECT文でアクセスすることによって重複のない連番を取得することが可能です。開始値、最大値、カウントアップの増分（一つずつ増えるか、飛び石で増やすか）、最大値まで達したときにサイク

リックに開始値から再び採番するか、といった様々な条件をオプションで指定できるので、柔軟性に富みます。

　たとえば、OracleとPostgreSQLで、シーケンスオブジェクトを使うSQL文はそれぞれ次のようになります。どちらも、CREATE SEQUENCEでオブジェクトを作り、それにSELECT文でアクセスします。

■Oracleの場合

```
CREATE SEQUENCE test_seq
 START WITH     1
 INCREMENT BY   1
 NOCYCLE;

SELECT test_seq.nextval
  FROM DUAL;
```

結果

```
test_seq.nextval
----------------
               1
```

■PostgreSQLの場合 [※5]

```
CREATE SEQUENCE test_seq
START 1;

SELECT nextval('test_seq');
```

結果

```
nextval
-------
      1
```

※5　PostgreSQLではデフォルトでノンサイクリックであるため、サイクリックにしたい場合のみ「CYCLE」オプションを付けます。

この方法は、次に見るID列に比べて細かい制御ができるという利点があります。ただし、比較的新しい機能であることから、まだすべてのDBMSがサポートしているわけではありません［※6］。

► ID列

データベースの機能でオートナンバリングを実現するもう一つの方法は、ID列です。これは、一意な連番を払い出すデータ型です。テーブルで使用する主キーとして、このデータ型を採用します（別にID列そのものは、主キー以外の列として使うこともできます）。こちらは、シーケンスオブジェクトに比べると指定できるオプションが少なく、開始値と増分ぐらいです。

また、ID列にはもう一つ難点があります。それは、DBMSごとに実装が統一されておらず、**移植性が低い**ことです。たとえば、SQL ServerではIDENTITY型、PostgreSQLではSERIAL型、MySQLではAUTO_INCREMENT型と、名前もバラバラで機能も微妙に異なります（Oracleは未サポート）。こうした理由から、利用できる場合は可能な限りシーケンスオブジェクトを使用することが推奨です。

勘どころ 55

シーケンスとID列では、シーケンスのほうがより柔軟で拡張性に富む。

オートナンバリングの実現方法❷ ～アプリケーション側で実装

オートナンバリングを実現するもう一つの方法は、データベースの機能に頼らずアプリケーション側で実装する方法です。最もポピュラーな方法は、「採番テーブル」を利

用するものです。

1行1列の整数型のデータを持つ採番テーブルを用意し、他のテーブルにデータを登録するたびにインクリメントしていく、という方法です。

■採番テーブルの利用

採番テーブル

シーケンス
21

新しい番号を採番

市町村

市町村管理コード	市町村コード	市町村名	人口（人）
0	A000	A市	100,000
1	A001	B市	120,000
2	A002	C町	30,000
3	A003	D村	10,000
⋮			
20	A999	Z市	15,000

この方法には、いくつか欠点があります。第一に、こういう機能を持ったプログラムの開発とテストが必要なため、コストがかかります。また、採番テーブルと連番を使用するテーブルとは密接に関連していて、片方が欠けたらシステムは動かないのですが、ER図からはそのような依存関係がわかりません。そのため、環境作成や移行時に、採番テーブルを含め忘れてシステムが動かない、というトラブルがよく起きます（これはシーケンスオブジェクトの場合にも言えることですが）。

しかしおそらく最大の問題は、排他制御の仕組みをきちんと作らないと機能的なバグを生み出してしまうことです。考えてみればわかることですが、二人以上の人間が同時に採番テーブルにアクセスして、同じ番号を引き当ててしまったら、「市町村」テーブルへのINSERT時に一意性制約違反が起きます。また、逆に番号が飛び石になってしまうこともあります。これを防止するには、一人が採番テーブルにアクセスした時点でテーブルをロックし、他の人間は参照も更新もできないようにすることで

す。こうした排他制御を意識したプログラム作成は、複雑になりがちですし、DBMS間でトランザクション分離レベルが異なることも多いため、異なるDBMSにプログラム移植をするにも、簡単にはいきません。シーケンスオブジェクトであれば、最初からそのようなロックメカニズムが実装されており、これを利用することができます。

以上の理由から、オートナンバリング機能をあえてアプリケーション側で作り込む、というのはお勧めできる選択肢ではありません。実際、データベースがシーケンスオブジェクトやID列といった機能をサポートしていなかった過去の時代ならばともかく、データベースの機能が充実してきた今の時代に、あえて「車輪の再発明」をする必要はないのです。

勘どころ 56

オートナンバリングをアプリケーションで実装するのは「車輪の再発明」。

最後にシーケンスオブジェクトと採番テーブルの共通の、そして巨大な欠点を挙げておきましょう。それは、上記に説明したロックが起きることで、同時アクセスが多数集中したときに、この連番の払い出し処理がボトルネックとなり、性能遅延が発生する可能性があることです（これは、比較的頻繁に見かける事例です）。

ロックは連番の一意性を保証するために必要なロジックなので、原理的にこの現象を防ぐことはできません。オートナンバリングが便利に見えて少なからぬ危険をはらむ機能であることが、理解いただけたでしょう。

主キーはなぜ必要か？ COLUMN

リレーショナルデータベースにおいては、テーブルが主キーを持つことは必須です。言い換えれば、重複レコードの存在を認めません。これは、レコードを一意に特定することが一般的な業務システムでは要求されるから、というのが最大の理由ですが、では仮に、業務的にレコードの一意性が求められなかった場合には、テーブルに主キーを設定しないことは許されるのでしょうか？

実際、たまにあるのですが、何らかの記録を保管するだけの機能（「ジャーナルファイル」と呼ばれたりします）が求められる場合、レコードの一意性がなくても良い、という業務要件も存在するのです。こういう場合、リレーショナルデータベースのテーブルは、主キーを作らなくても良いのでしょうか？

その答えは、やはり「No」です。たとえ業務要件の上で主キーが不要であったとしても、リレーショナルデータベースのテーブルには主キーが必要です。ここでは、重要な一つの理由を説明します。

まず、実際に「主キーがない」テーブルを許すと、どのようなことが起きるかを考えてみましょう。

社員

会社コード	社員ID	社員名	年齢	部署コード
C0001	000A	加藤	40	D01
C0001	000B	藤本	32	D02
C0001	001F	三島	50	D03
C0002	000A	斉藤	47	D03
C0002	009F	田島	25	D01

C0002	010A	渋谷	33	D04

今、部署テーブルに主キーがなく、重複レコードを許しています。それでは、この二つのテーブルを使って、加藤さんが所属する部署の名前を取得することを考えましょう。SQLは、以下のようになります。

```
SELECT 社員.社員名,
       社員.部署コード,
       部署.部署名
  FROM 社員 INNER JOIN 部署
    ON 社員.部署コード = 部署.部署コード
 WHERE 社員.社員名 = '加藤';
```

結果

```
社員名   部署コード   部署名
----   -------   ----
加藤     D01       開発
加藤     D01       開発
加藤     D01       開発
```

結果は3行出力されます。これは、社員テーブルの1レコードに対して、部署テーブルの3レコードが対応するためです。このSQLには、以下の二つの問題があります。

問題1 **結果の重複を排除するためにDISTINCTを付ける必要がある。**
問題2 **結合において多数のレコードを対象にする必要がある。**

どちらも、パフォーマンスに悪影響を与える要因です。**問題1** は結果に含まれるレコード数が少ない場合には大きな問題にはなりませんが、**問題2** が与える影響は甚大です。結合はSQLの行なう処理の中でもコストが高く、結合レコード数を極力減らすことがSQLチューニングの基本です。冗長なレコードを結合対象にすることは、その基本に真っ向から反対する行為なのです。

これはほんの一例にすぎず、重複レコードを許すことによるパフォーマンスへの悪影響は他にもあります。こうした理由からも、テーブルに主キーを付けることはリレーショナルデータベースにとって非常に重要な意味を持っているのだ、ということがお

リレーショナルデータベースにとって非常に重要な意味を持っているのだ、ということがお
わかりいただけるでしょう。

主キーがなぜ必要か、という点についてはコラム「主キーはなぜ必要か？」（239ページ）を参照。

たとえば1～100を値の範囲とするデータにおいて、1, 2, …… 100までの値を使い切ったときに、再度1から順に循環的に使うことを指します。

このようなキーを**自然キー**（natural key）、または「ナチュラルキー」と呼びます。

このサンプルでは、「開始年度」と「終了年度」は、ともに市町村の存続期間に含むと仮定しました。そうではなく、両端のどちらかを存続期間に含まない場合は、BETWEENではなく不等号で記述することになります。

具体的には、2012年2月時点では、Oracle、PostgreSQL、DB2の最新版はサポートしていますが、SQL Server、MySQLは未サポートです（SQL ServerはSQL Server 2012でサポート予定）。

列持ちテーブル

本節は、3-4節の「第1正規形」と7-2節の「非スカラ値（第1正規形未満）」の続きです。「やれやれ、またスカラ値と配列型の話か」と思うかもしれませんが、またその話です。このポイントは、それだけ難しく、誰もが設計に悩む箇所なのです。それだけに、本節で重点的に解説しておきます。

配列型は使えない、でも配列を表現したい

7-2節では、非スカラ値を認める配列型を紹介しました。驚いた方もいるかもしれませんが、リレーショナルデータベースは、この配列型を原則許可してしまったのです。現時点において配列型の使用は、バッドノウハウに位置づけて良いのですが、一方で配列型を使わずに配列を模倣する論理設計があります。それが本節で紹介する「列持ちテーブル」です。別名「繰り返し項目テーブル」とも呼ばれます。

この列持ちテーブルは、3-4節の「第1正規形を作ろう」でも一度紹介したことがあります。再び「扶養者」テーブルをサンプルに使うと、以下のような構造のテーブルでした。

■列持ちテーブル

扶養者（列持ち）

社員ID	社員名	子1	子2	子3
000A	加藤	達夫	信二	
000B	藤本			
001F	三島	敦	陽子	清美

JavaやCといったプログラミング言語であれば、kodomo[0]、kodomo[1]、kodomo[2]……のような形で子のデータを配列に保持します。それをテーブルの「列」を使って模したのが、この列持ちテーブルです。この構造のテーブルは、わりと頻繁に見かけます。ある意味「素直」な設計で、直観的に理解しやすいのが、人気の理由でしょう。

この列持ちテーブルを、著者としてはバッドノウハウではなくグレーノウハウに分類したいと思います。データベースエンジニアの中には、このテーブルですらバッドノウハウに分類するべきだと主張する人もいますが、著者はそこまで厳しく使用を制限する必要はないと考えています。それは、以下に示すように、このテーブルには確かに無視できない利点があるからです。

列持ちテーブルの利点と欠点

列持ちテーブルの利点を挙げると以下の二つになります。

利点1 シンプルな設計

列持ちテーブルの利点は、そのシンプルさにあります。正規化とかスカラ値とか、小難しいことを知らない人でも、このテーブルを見れば一目で、子1、子2、子3の各列が配列を表現しているのだということがわかります。

利点2 入出力のフォーマットと合わせやすい

そのシンプルさのために、このテーブルは、アプリケーションサイドとのインタフェース設計も非常に簡単になります。たとえば、実際に業務で出力される帳票などでは、このように被扶養者の情報を列として並べるフォーマットを持つものがあります。そのようなときは、この列持ちテーブルであれば、単純にSELECTした結果をダイレクトに帳票へ流し込むことができます。

一方で、このテーブルには欠点もあります。それは主に保守性と拡張性に関わるものです。

欠点1 列の増減が難しい

リレーショナルデータベースのテーブルは、一度作ると、後になってから構成を変更することが難しいという特徴があります。つまり拡張性に乏しいのです。もちろん技術的にはできるのですが、アプリケーション側もあわせて変更する必要が出るなど、変更コストが高くなりがちです。

テーブルの列も同じで、配列ほど柔軟に要素数を変化させることができません。たとえば、四人の子を持つ社員が入社してきたので、この扶養者テーブルに「子4」という列を追加しようと思った場合、変更対象はこのテーブルだけでなく、SQLやデータを受け取るアプリケーション側にも及びます。もちろん、ある程度そういった変更

を見越して動的にアプリケーションを作り込んでおくことも可能なのですが、なかなか大変なことです。

欠点2 無用のNULLを使わなくてはならない

列持ちテーブルのもう一つの欠点は、子がいない、または子の数の少ない社員について、子の列にNULLを使わなくてはならないことです。NULLというのはリレーショナルデータベースにおいて非常に厄介な性質を持っていて、NULLが演算に含まれているとSQL文の結果を混乱させる原因になります［※7］。

行持ちテーブル

このように、列持ちテーブルは、利点と欠点のバランスが拮抗しているため、使いどころによっては有用です。逆に言うと、特殊な状況でない限り、原則として列持ちテーブルは使うべきではありません。基本的には「行持ち」のテーブル構成を採用するべきです。行持ちテーブルは、列持ちテーブルに比べて、欠点の少ない定石です。

■行持ちテーブル

扶養者（行持ち）

社員ID	枝番	子
000A	1	達夫
000A	2	信二

001F	1	敦
001F	2	陽子
001F	3	清美

　この二つのテーブルは、SQLによって簡単に変換が可能なので、いったんどちらかのテーブルにデータを蓄積した後でも、行持ち⇔列持ち間のデータ移行は簡単です。それぞれの変換SQLは以下のようになります。

■列持ち⇒行持ちへの変換

```
SELECT 社員ID,
       1,
       子1
  FROM 扶養者（列持ち）
 WHERE 子1 IS NOT NULL
UNION ALL
SELECT 社員ID,
       2,
       子2
  FROM 扶養者（列持ち）
 WHERE 子2 IS NOT NULL
UNION ALL
SELECT 社員ID,
       3,
       子3
  FROM 扶養者（列持ち）
 WHERE 子3 IS NOT NULL;
```

■行持ち⇒列持ちへの変換

```
SELECT 社員ID,
       MAX(CASE WHEN 枝番 = 1 THEN 子 ELSE NULL END),
       MAX(CASE WHEN 枝番 = 2 THEN 子 ELSE NULL END),
       MAX(CASE WHEN 枝番 = 3 THEN 子 ELSE NULL END)
  FROM 扶養者（行持ち）
 GROUP BY 社員ID;
```

まず、列持ちから行持ちへの変換ですが、このロジックはとても単純です。子1 ～子3までの列について、NULLではない（つまり子が存在する）レコードだけをそれぞれ選択して、UNION ALLによってレコードをマージすればできあがりです［※8］。「枝番」列だけは、元のテーブルからは選択できないので、定数で埋め込みます。

一方、行持ちから列持ちへの変換は、少し「ワザ」を使います。まず、行を列に展開するときの常套手段として、CASE式を使います。これはSQLにおいて分岐を表現するための重要な技術で、条件をWHEN句に記述し、それに合致する場合の値をTHENの後に記述します。これによって、子1、子2、子3という列へ展開することが可能になります。

しかし、これだけだとNULLを含むレコードも含む冗長な結果が出力されてしまうので、これを社員ID単位で集約するために、社員IDをGROUP BY句のキーとして使います。MAX関数を使っているのは、GROUP BY句を使うと列を裸でSELECT句に記述できないための方便で、別に最大値を選択する働きはしていません。社員ID単位ならば枝番は必ず一意になるため、実質的に、MAX関数の引数は常に1行に定まるからです。

このように行持ちと列持ちの変換は保証されているうえにSQL一発で可能なので、最初は拡張性の高い行持ちテーブルでデータを保持しておき、パフォーマンス上の問題でどうしても列持ちテーブルが必要になった場合に、列持ちテーブルを作る、という方針が良いでしょう。

NULLがもたらす数々の問題について興味のある方は、拙著『達人に学ぶSQL徹底指南書』（翔泳社、2008）の「3値論理とNULL」と「NULL撲滅委員会」を参照してください。

UNION ALLの代わりにUNIONを使っても結果は同じです。ただ、今回はマージするレコード同士に重複がないことが明らかなので、UNION ALLのほうがソートを行なわない分、効率的で高速です。

次のグレーノウハウは、アドホックな集計キーです。これは、DWH/BI分野の論理設計を行なう際によく問題になります。例として、こんなケースを考えましょう。

今、都道府県別の人口を保持するテーブルがあるとします。このテーブルから、県別ではなく地方別の人口の合計を求めるとします。つまり、東北＝北海道＋青森＋岩手、東海＝静岡＋愛知＋三重、といったようにです。

■都道府県別の人口を保持するテーブル

都道府県

県コード	県名	人口（万人）
01	北海道	550
02	青森	130
03	岩手	133
22	静岡	370
23	愛知	740

24	三重	185
36	徳島	78
37	香川	99

こうした集計をSQLで行なうには、集約関数とGROUP BY句を組み合わせれば良いのですが、肝心の集計キーにすべき「地方コード」が存在していません。こういう場合、一番安直な解としては、次のように「地方コード」列を追加する、ということになります。便宜的に地方コード01：東北、02：東海、03：四国とすると、次のようになります。

■集計キーを追加

都道府県（地方コード付き）

県コード	県名	人口（万人）	地方コード
01	北海道	550	01
02	青森	130	01
03	岩手	133	01
22	静岡	370	02
23	愛知	740	02
24	三重	185	02
36	徳島	78	03
37	香川	99	03

アドホックな（場当たり的な）集計キー

これならば苦もなくSQLで集計ができます。非常に簡単な解決策ですが、一方で問題もあります。こうしたアドホックな（場当たり的な）キーは、名前のとおり、コード体系が短いスパンで変わったり、別のコード体系が必要になったりしま

す。そのたびに、サイズの大きいテーブルにアドホックキーを次から次へと追加すると、ただでさえ規模の大きなテーブルがますます巨大になって、パフォーマンスを劣化させます。

この問題点を解決する手段は大きく三つあります。

解決策1

一つ目の解決策は、キーを別テーブルに分離することです。この場合であれば、都道府県と地方の変換テーブルを作ることになります。

■都道府県と地方の変換テーブル

都道府県-地方

県コード	地方コード
01	01
02	01
03	01
22	02
23	02
24	02
36	03

37	03

この変換テーブルのレコード数はトランザクションテーブルに比べてかなり小さくなることが期待でき、メンテナンスも容易です。一方でSQLでは結合処理が必要になるため、パフォーマンス問題の解決にはあまり寄与しません。

解決策2

二つ目の解決策は、ビューを使うことです。オリジナルのテーブルには手を加えずに、地方コードを追加したビューを用意すれば、そのビューへのアクセスによって簡単に地方単位の集計を行なうSQLが記述できます。これは、実質的にオリジナルのテーブルへアクセスしているのとコストは変わらないため、必要なコードの数だけビューを作っても、パフォーマンスを劣化させることがありません（ただし、次節で見る「多段ビュー」には注意が必要です）。

解決策3

最後の選択肢は、GROUP BY句の中でアドホックキーを、それこそ「アドホック」に作る方法です。具体的には、CASE式で次のようにコードの読み替えを行ないます。

■GROUP BY 句の中でアドホックキーを作る

```
SELECT CASE WHEN 県コード IN ('01', '02', '03') THEN '01'
            WHEN 県コード IN ('22', '23', '24') THEN '02'
            WHEN 県コード IN ('36', '37') THEN '03'
            ELSE NULL END AS 地方コード,
```

```
      SUM(人口（万人）)
  FROM 都道府県
GROUP BY CASE WHEN 県コード IN ('01', '02', '03') THEN '01'
              WHEN 県コード IN ('22', '23', '24') THEN '02'
              WHEN 県コード IN ('36', '37') THEN '03'
              ELSE NULL END;
```

CASE式は「1＋1」のような「式」の仲間なので、式が書ける場所にはどこでも書けます。具体的に言えば、SELECT句だけでなく、WHERE句、GROUP BY句、HAVING句、ORDER BY句に書くことが可能です。この万能ぶりから、数あるSQLの機能の中でも重宝します。

ビューは、リレーショナルデータベースを用いた開発においては非常に有用な道具です。SELECT文を保存して、一種のテーブルとして扱うことができるため、正規化されたテーブルの結合など、複雑な操作を要するSELECT文をビュー化しておくことで、上位レイヤーのアプリケーションのプログラミングを簡潔に行なえるという利点があります。テーブルを隠蔽してユーザーに必要なデータだけを必要な形で見せられることから、外部スキーマに属します。

本書でも何度か名前の出てきたDBエンジニアのクリス・デイトは、ビューを「**クエリの缶詰**」と表現しました。保存がきくうえに、開ければ常に新鮮なデータを取り出せる、というわけで、なかなか巧みな比喩です。

しかし、何事につけて「いいとこどりは存在しない」というのがシステム開発における第一原則ですので、ビューも長所ばかりではありません。ビューの短所は、パフォーマンスへ悪影響を与えることと、濫用するとかえって設計と実装を複雑なものにしてしまうことです。

ビューへのアクセスは「2段階」で行なわれる

　というのも、先述のように、ビューというのはテーブルと違って、データを保持しません。

　ビューは利用者からの見た目上、テーブルと同じように扱うことができます。SQL文の構文上でも、テーブルとまったく同じ構文で扱います。そのため、初級者はビューとテーブルの違いに無頓着になりがちなのですが、物理的なレベルで見れば、ビューとテーブルは「月とスッポン」というやつです。ビューは実データを保持しないという点で、物理的にはSELECT文が書かれたファイルにすぎないのです。

　では、ビューに対してSQL文によってアクセスが行なわれたとき、ビューはどうやってデータを取り出しているのかといえば、結局のところ、ビュー定義のSELECT文を実行して、**オリジナルのテーブルにアクセスしている**のです（このオリジナルのテーブルを「**基底テーブル**」と呼ぶこともあります）。

　つまり、ビューへアクセスするときは、2段階のSQLが発行されているようなイメージを持ってもらえば良いでしょう（図8-1）。

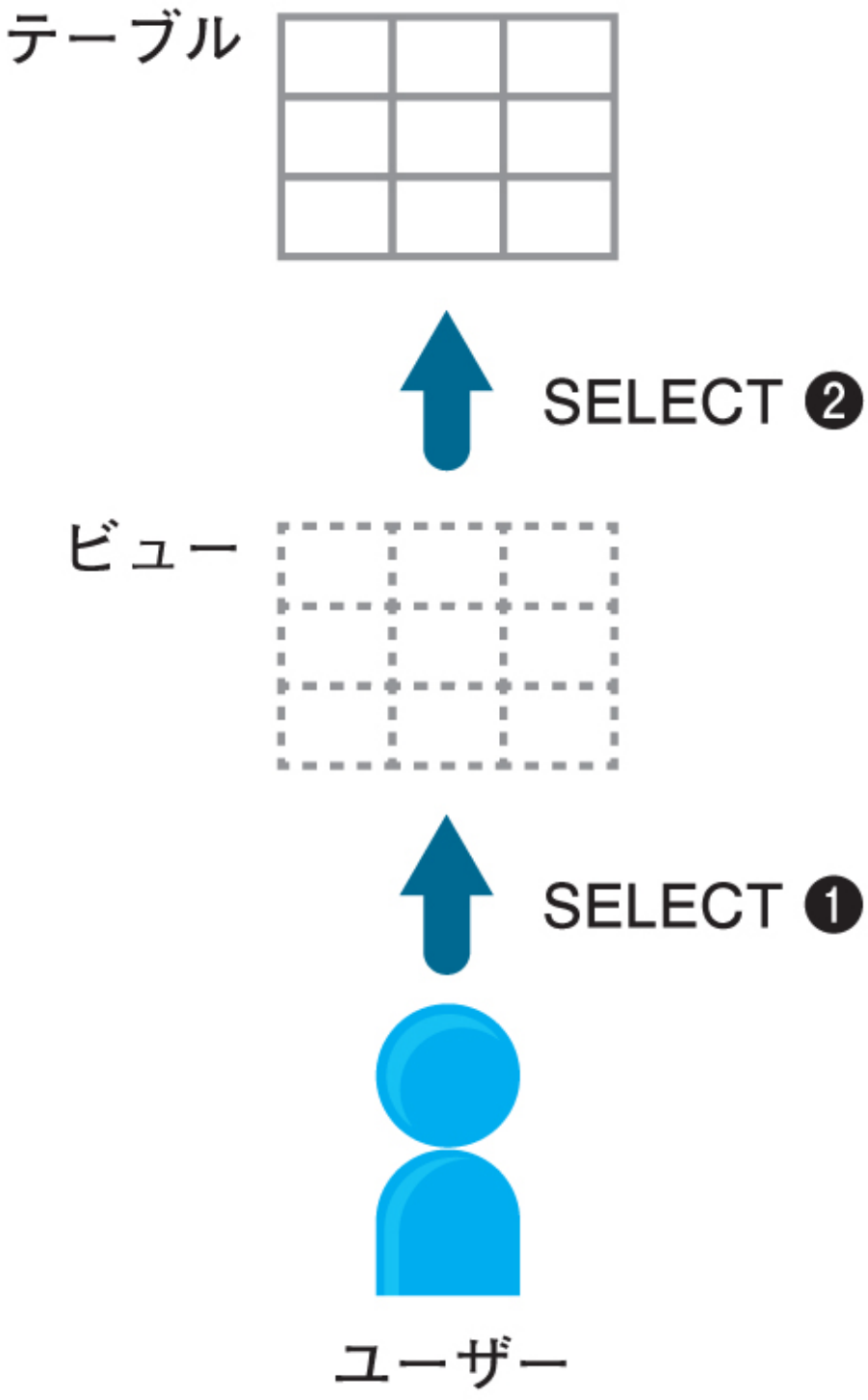


図8-1●ビューへのアクセスイメージ

少し細かい補足をすると、現実にはDBMS内部でこのようなSQL文が処理されるときは、本当に2段階に分けて実行されるとは限りません。むしろDBMSは、可能な限り❶と❷のSELECT文を合体（マージ）して、効率の良い形で実行しようとします。ただ、いずれにせよその場合も、純粋に❶のSELECT文が実行されるのではなく、「❶＋❷」によって生まれる新たなSELECT文（❸）を実行することになり、❶よりは複雑で高コストな処理になります。

多段ビューの危険性

このように、ビューへのSQL文によるアクセスにおいては、常に背後に存在する基底テーブルの存在を意識しておく必要があります。もともとビューは基底テーブルを隠蔽する技術でもあるのですが、本当に基底テーブルを意識しなくていいのはエンドユーザーだけであって、エンジニアは常に意識しなければなりません。エンジニアは、決して物理層から自由になれないのです。

勘どころ 57

ビューの背後にあるテーブルの存在を、常に意識せよ。

この警告を無視して、図8-2のようにビューを多段で構成すると、バッドノウハウ「多段ビュー」が生まれます。

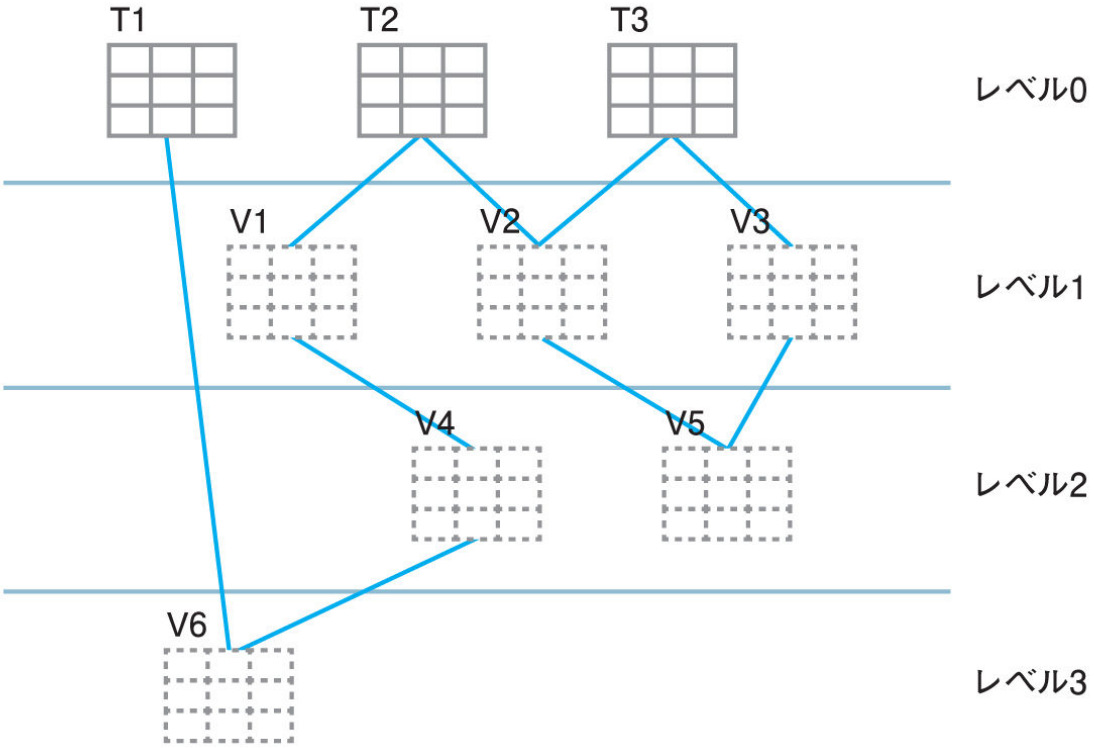


図8-2●多段ビュー

T1 ～ T3をテーブル、V1 ～ V6をビューとします。このとき、基底テーブルのレベルを0とすると、そこからどのレベルのテーブルまたはビューを基底として導出されるかによって、ビューのレベルが決まります。

すると、たとえば最もレベルの高いV6のビューへアクセスするSELECT文を考えると、内部的な処理はどうなるでしょう。まずは、T1およびV4へのSELECT文を実行することが必要になります。しかし、V4はビューなので、さらに展開されて、V1へのアクセスするSELECT文も実行されます。V1もまたまたビューなのでT2へアクセスするSELECT文も実行されて、最終的には合計4つのSELECT文が内部的には必要になります。これは、V6へアクセスする処理全体として考えた場合、パフォーマンスを悪くします。

かつ、このように階層の深いビュー定義というのは、パフォーマンスを別としても、テーブルとビューの依存関係をわかりにくくするため、仕様が複雑になり管理が困難です。ひどい場合には、設計したエンジニアですら何のために作ったビューなのかわからなくなってしまい、同じような機能のビューが乱立する、という最悪の事態にまで発展します（これはちょうど、データマートが乱立するケースと類比的です）。

以上から、多段ビューがどれほど危険な設計であるか理解していただけたと思います。ビューの使用は、原則として1段にとどめておくようにしましょう。システムの世界には、「過度に複雑な作りはシステムをダメにする」という思想があります。この思想を表現するスローガンを「KISSの原則」と言います。それは次のようなものです。

勘どころ 58

Keep It Simple, Stupid.（単純にしておけ、このバカ）

みなさんも、後になってから意味不明の仕様に悩みたくなければ、設計は極力シンプルにしておくことです。

なお一点補足しておくと、第7章の演習問題で取り上げたマテリアライズドビューには、本節で取り上げた多段ビューの問題は発生しないため、マテリアライズドビューを多段で構築することで、SELECT文のパフォーマンスを悪化させることはありません。これは、マテリアライズドビューが実データを保持するからです。しかし、それによって新たな問題点も発生することを、すでにみなさんは演習問題を通じて学んでいると思います。「この世にいいとこどりは存在しない」（トレードオフ）でしたね。

8-6 データクレンジングの重要性

リレーショナルデータベースを利用する／しないにかかわらず、システムで扱うデータをデータベースによって一元管理することは、システム化による業務効率化の第一歩です。そのために、データベースを構築するに当たっては、それまでの業務で利用されていたデータをデータベースに登録できる状態にすることが必要になります。この作業を**データクレンジング**（Data Cleansing）と呼びます。

データクレンジングが終了しないと、せっかく本書で学んだ正規化のような方法論も利用することができませんし、バッドノウハウやグレーノウハウが発生する理由の一つも、データクレンジングをおろそかにすることにあります。本節では、このデータクレンジングについて取り上げます。

データクレンジングは設計に先立って行なう

「効率的なシステムにはデータクレンジングが必要」ということを逆からとらえると、通常の業務データは、掃除が必要な「汚れている」状態ということです。これは、そのデータがあらゆる意味で「使えない」ということではありません。それまでも既存のデータを利用して業務が行なわれてきた以上、既存業務の目的には合致したフォーマ

ットではあったはずです。ただ、新たに構築されるシステムにとっては、使えなかったり非効率的なフォーマットであったりすることを「汚い」と表現しています。

　このような場合、データベースの論理設計にとりかかる前に、入力データのフォーマットが適切なものか調査し、必要であればデータそのものの変更やフォーマット変換が必要になります（図8-3）。つまり、データクレンジングはデータベース設計に先立って行なう必要があるのです。

　特に厄介なのが、もともとシステム化されていなかった業務を初めてシステム化する場合や、システムは利用されていても、リレーショナルデータベース以外のフォーマットのデータベースが利用されていた業務をリレーショナルデータベースに移し替えるような場合です。

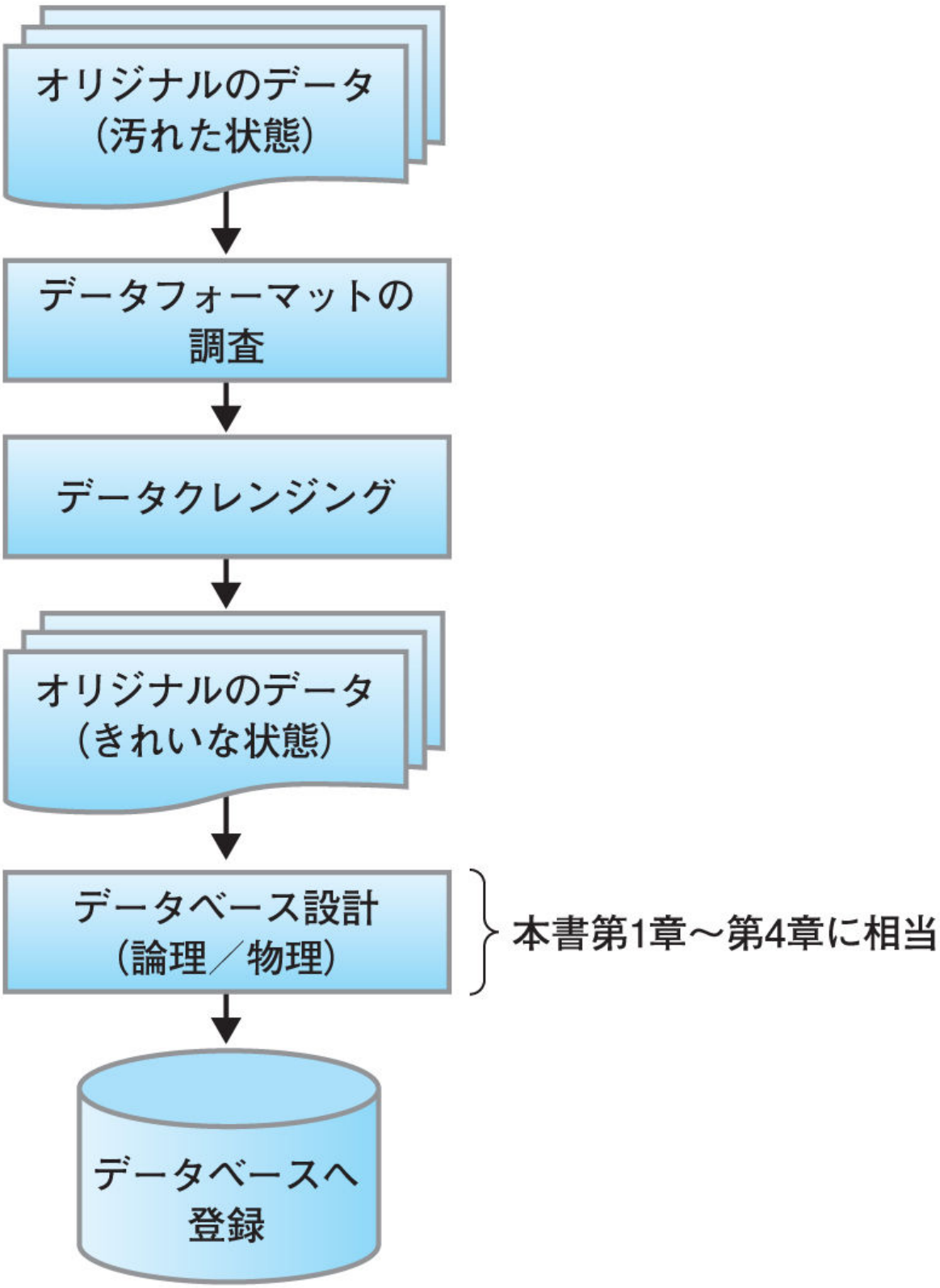

オリジナルのデータ
(汚れた状態)
データフォーマットの
調査
データクレンジング
オリジナルのデータ
(きれいな状態)
データベース設計
(論理／物理)
本書第1章～第4章に相当
データベースへ
登録

図8-3●データクレンジングはデータベース設計の前段階の作業

代表的なデータクレンジングの内容

それでは、データクレンジングとは、具体的にどのような処理を行なうのか、代表的なものを紹介しましょう。この内容を理解すると、データクレンジングを怠った場合にどのようなバッドノウハウやグレーノウハウが生じるのかもわかるようになります。

一意キーの特定

リレーショナルデータベースの考え方に慣れてくればくるほど、レコードを一意に特定することのできる一意キー（ユニークキー）の存在は、当たり前のように思えてきます。しかし、意外に世の中で行なわれている多くの業務では、データの一意性については注意が払われていないことも多くあります。

たとえば、みなさんがホテルや旅館の利用者の帳簿をつける仕事をしていて、昔ながらの紙で記録するやり方をしているとしましょう。この場合、日々の宿泊の記録だけをとっているだけなら、宿泊客について一意なデータを作る必要は感じないかもしれません。また、それをやろうとしても、宿泊客の中には、泊まるたびに「渡邊」「渡辺」と新字と旧字で異なる表記をしたり、それどころか「ワタナベ」とカタカナ表記をしたりする人もいるかもしれません。この場合、宿泊記録のどのレコードとどのレコードが同一の客のものなのか、後から正確に識別することは困難です。

■宿泊代金の計算だけを念頭においたデータ形式

宿泊日	宿泊者（代表）	部屋	日数	宿泊料	朝食
2011/11/20	川田　光一	1101	2	53,000	無
2011/11/20	渡辺　光代	1230	3	48,000	無
2011/11/20	新藤　修司	306	2	120,000	有
2011/11/21	武藤　明	207	1	32,000	無
2011/11/21	峯　恵一	208	1	22,000	有
同一人物？			同一人物？	⋮	
2011/12/24	渡邊　光代	1103	1	30,000	無
2011/12/24	武藤　あきら	1101	1	32,000	無
2011/12/24	加山　鏡子	1203	2	42,000	無

このような宿泊記録からは、月々の宿泊料や宿泊人数を計算することはできます。そのため、そうした業務しか行なわないのであれば、このデータは十分に「きれいな」データと言ってかまいません。しかし、最近ではこうしたデータを経営分析に使うことがよくあります。たとえば、リピーター率がどのぐらいか、どのようなタイプの部屋が客受けが良いのか……といった観点から、過去のデータの集積を使って分析するのです。

こうした目的でデータベースを構築しようと考えた場合、宿泊客を一意に識別する情報（＝宿泊客コード）がないのは致命的です。この帳簿からは、よく似た名前の顧客が本当に同一人物なのか確認する手段がありません。また、厳密に見るとこの宿泊記録には、主キーと呼べる列の組み合わせもありません。近いのは{宿泊日, 宿泊者（代表）}ですが、宿泊者（代表）は、同姓同名の人物がいた場合に重複が生じます。これは第7章のバッドノウハウ「不適切なキー」でも指摘したことです。つまり、一意キーの確定をおろそかにしたデータは、バッドノウハウを生み出す原因でもあるのです。

勘どころ 59

一意キーの存在しないデータは、バッドノウハウ「不適切なキー」をも生み出す。

以上から、この宿泊記録をリレーショナルデータベースのテーブルとして扱おうと考えるならば、正規化以前に少なくとも以下の二つの一意キーが必要になることがわかります。

- 宿泊客コード：顧客を一意に識別
- 宿泊ID　　：宿泊を一意に識別

しかし、宿泊IDは後から追加することができるとしても、宿泊客コードについては、そもそもよく似た名前の宿泊客が同一かどうかを判別できない限り、完全には付与することができません。この問題を解決するために必要になるのが、次に紹介する「名寄せ」です。

名寄せ

名寄せとは、名前のとおり「似通った名前を寄せ集めて統合する」という意味です。人名や企業名の表記揺れを解消して名称を統一します。その意味で、標準化の一種に数えられる作業です。

もともと、この名寄せという作業は金融機関で古くから行なわれてきました。同一の金融機関で複数の口座を所有する個人や法人を一元管理するには、契約主体を一意に特定する必要があります。特に最近は、金融機関の統廃合が盛んに行なわれているという日本経済の事情もあって、別個の金融機関にあった口座を統合する大規模な名寄せも行なわれるようになってきました。

名寄せの対象になるデータとしては、個人名、法人名、住所、電話番号など多岐にわたります。たとえば、法人名の場合だと、次のようにいろいろな表記の方法があります。いずれも間違いではなく、ただ「フォーマット」（見た目）が異なるだけです。

㈱　武田産業
株式会社　武田産業
Co. Takeda Industry

住所なども定型的なフォーマットがあまりかちっと決まっておらず、郵便番号を書くか、マンションの建物名まで書くか、といったところは記入者の裁量に任されていることがしばしばです。

つまり、名寄せが発生する根本的な原因は、上記のような情報を記録する際に標準的なフォーマットを決めず「**フリーハンド**」の入力を許していることにあります。フリーハンドは、複数のフォーマットが混在するという問題はもちろん、単純な誤記を許してしまうという欠点も持っています。最近では、名寄せにかかるコストを抑えるため、事前に統一された入力フォーマットを用意して、最初から名寄せが発生しないデータを作ることも意識されています。しかし、そうした取り組みが始まったのは最近のことで、まだまだ過去の「負の遺産」（＝汚いデータ）を洗浄してやる必要が残っているのです。

ちなみに、金融機関の統合などのケースで典型的なのですが、名寄せをサボって複数のマスタテーブルを統合しないまま残すと、どういうことになるか、おわかりでしょうか？ そう、これはバッドノウハウ「ダブルマスタ」を生み出すことになるのです（図8-4）。

勘どころ 60

名寄せをサボると、バッドノウハウ「ダブルマスタ」を生み出す。

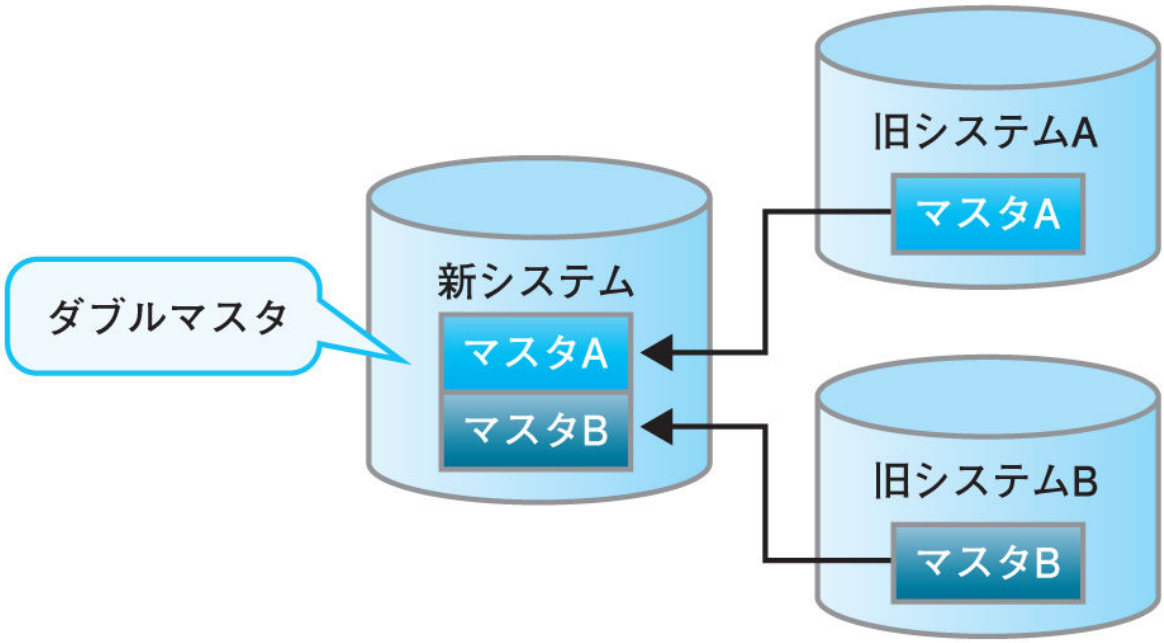


図8-4●旧システムのマスタは名寄せによって一元化すべき

名寄せの判断方法

それでは具体的にどうやって名寄せを行なうのでしょう？「渡辺光代」さんと「渡邊光代」さんは同一人物であると（あるいは同姓同名の別人であると）どうやったらわかるのでしょうか？

一般的には二つの方法があります。

一つは、別の情報と組み合わせて確度を高めるというもの。たとえば、登録されている住所が同じであれば「渡辺光代」＝「渡邊光代」である可能性はかなり高いと考えていいでしょう（その住所にも表記揺れがあるとまた話がややこしくなるの

ですが……）。さらに年齢まで一致したら、これは同一人物と断言して間違いなさそうです。

もう一つの方法は、情報の出現頻度です。たとえば、会社名というのは覚えてもらいやすくするために、他社と似ていない特徴的な名前をつけるものです。かつ、会社名は所定の役所に法人登録を行なうため、普通は重複がありません（国が違うとまた別でしょうが）。したがって、珍しい名前の会社であれば、異なるフォーマットで記載されていても同一であると判断できることは多いでしょう。

このような二つの判断方法は、ある程度機械的に行なうことができるため、システム化が可能です。事実、データクレンジングに特化したソフトウェアも市販されており、けっこうな規模の市場を形成しています。ただ、どれだけ確度が高くなったとしても、機械では100％確実という精度までは実現できないこともあります。そうした場合は、最終的に人間の手による確認が必要になります。

データクレンジングは、行なわずに済むならばそれに越したことはありません。最初からシステムで扱うことを意識したデータフォーマットの設計を行なっていれば、理論的にはデータクレンジングは不要なのです。今後みなさんが新規のシステム構築に携わる機会があったときには、データ入力のユーザーインタフェースに注意していただきたいと思います。しかしまた、システムでデータを扱うことをまるで意識しないまま蓄積されてきたデータが非常に多いことも事実ですし、システム統合においてはほぼ例外なく必要になります。そうした際には、データ設計の前にその「洗浄」が重要な鍵を握るということを、是非覚えておいてください。

以上、本章ではグレーノウハウの数々を紹介してきました。バッドノウハウと比べた場合のグレーノウハウの特徴は、バッドノウハウが文句なしに劇薬として「禁じ手」にできるのに対し、グレーノウハウは使い方次第では有効な良薬になりうることです。そのため、グレーノウハウは「用量用法をよく守って」使う限りはシステムに良い作用

を及ぼしますが、そのためには利点と欠点の比較考量（トレードオフ）を検討しなければなりません。

しかし、実際の開発現場では、そのような考慮がなされないまま、軽い気持ちで処方されていることもしばしばです。このようなグレーノウハウの濫用は、ある意味でバッドノウハウよりもタチの悪いものです。読者のみなさんは、本章を通してグレーノウハウの危険性を十分に理解されたと思います。今後、みなさんがデータベースの論理設計を中心となって組み立てる機会も数多いでしょう。そのようなときには、本章で学んだ知識を活かして、システムと、そして開発チームにとって最善の設計を行なっていただくことを期待します。

演習問題

演習 8-1 ビジネスロジックの実装方法の検討

本文で代理キーについて解説した際、テーブルに主キーを設定せず、重複行データを登録後に削除する、という間違った設計について触れました。しかし、主キーによる一意性制約に限らず、データが満たすべき様々な条件（ビジネスロジック）を、テーブルの制約ではなく、アプリケーションコードによって実装することは一般的に行なわれています。

さて、このビジネスロジックをアプリケーションコードで実装することについて、以下の問いについて考えてください。

問い1 **ビジネスロジックをアプリケーションコードで実装することの是非**

問い2 **ビジネスロジックをデータベースの「トリガー」で実装することの是非**

演習 8-2 一時テーブル

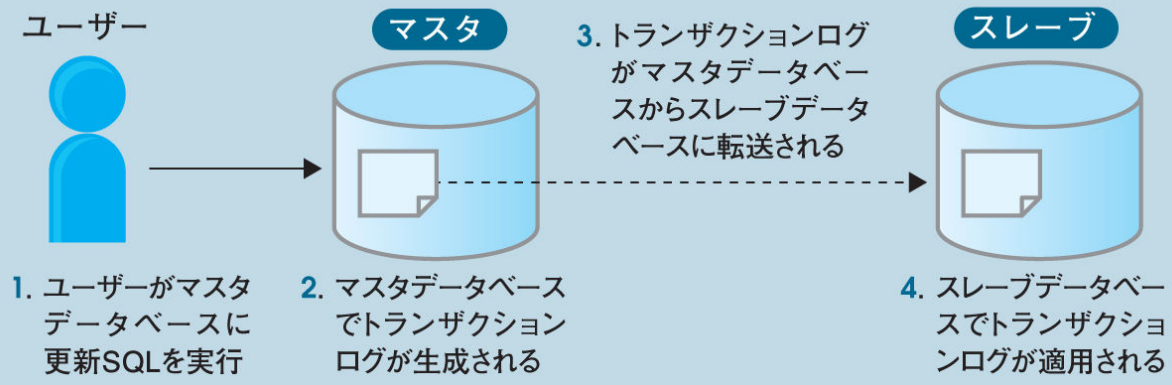
8-4節で紹介したようなアドホックなデータ、つまり一時的にしか使わず、処理が終わればなくなってかまわないようなデータを用意する手段として、DBMSは「一時テーブル」という機能を用意しています。この機能について調べ、使うことの是非を考えてください。

バックアップとレプリケーション COLUMN

第2章において、バックアップの方式としてフルバックアップ、差分バックアップ、増分バックアップを紹介しました。基本的な方式としてはこの三つを押さえておけば良いのですが、このコラムでは、しばしば利用されるもう一つのバックアップ方法を紹介しておきたいと思います。それがレプリケーションです。

実際のところ、レプリケーションを「バックアップ」の一種と見なすのは、違和感を感じる人もいるかもしれません。というのも、これはデータを冗長に保持する方法の一種なので、どちらかというとRAIDに近いところがあるからです。

レプリケーションとは、「複製」「複写」という意味です。つまり、まったく同じデータを持つデータベースを2セット用意して、常に一方の更新差分をもう一方に反映することで同期を取る、という仕組みです。このとき、主としてユーザーが利用するほうをマスタ、もう一方の更新差分で更新していくほうをスレーブと呼びます。レプリケーションは図Aのステップを踏んで実施されます。



図A●レプリケーションのステップ

4.における「トランザクションログの適用」は、2-4節で学んだ「ロールフォワード」と同様の処理だと考えてください。要するに、スレーブ側は、マスタ側で更新が発生するたびにロールフォワードを実施しているようなもの、とも言えます。

レプリケーションの利点は、何といっても障害時の復旧に要する時間が短いことです。なにしろ、同じデータを持ったデータベースがもう一つ存在しているのですから、マスタが障害でダウンしたら、スレーブにつなぎかえればサービスを継続できます。マスタの復旧は裏でゆっくり実施すれば良いのです。

一方、欠点はコストです。データベースを最低2セット用意する必要があるため、それだけ費用がかかります。

つまり、レプリケーションは、極力ダウンタイムを短くしたいシステムで、データベースを2セット用意するコストが許容できる場合に利用される、ということです。これも是非、選択肢の一つとして覚えておいていただきたい方式です。

第9章

一歩進んだ論理設計 ～ SQLで木構造を扱う

リレーショナルデータベースは強力な表現力を持ったデータベースですが、昔から扱うのが苦手なデータ構造があります。それが木構造です。これは、長らくリレーショナルデータベースのアキレス腱でした。本章では、この弱点をリレーショナルデータベースがどのように克服するか、その方法を明らかにします。

学習のポイント

- リレーショナルデータベースには、表現することが苦手なデータ形式があります。その代表格が、木（tree）構造です。
- 木構造を表現する伝統的な手段として、隣接リストモデルという方法がありますが、これはSQL文を複雑にする難しい方法です。
- 近年、隣接リストモデルに代わる新しい方法論が考えられています。本章では、その方法論である、「入れ子集合モデル」およびそれを発展させた「入れ子区間モデル」、そして「経路列挙モデル」を紹介します。

二次元表によく似た「テーブル」という形式（もともとtableという単語には「表」という意味もありますが）を採用したリレーショナルデータベースは、その強力で便利な表現力を駆使して、ありとあらゆるシステムの基幹データベースとして採用されるようになりました。今では、特に前置きなく「データベース」といえば、暗黙にリレーショナルデータベースを意味するほどです。二次元表は、生活の中で最もよく使う馴染み深い表現形式の一つですから、人間にとっても直観的に理解しやすく便利なものです。

しかし、誰にでも得意不得意というのはあるものです。リレーショナルデータベースは、幅広いオールラウンダーではあるのですが、それでもいくつかの苦手な相手を持っています。その一つが、本章で取り上げる「木構造」の取り扱いについてです。

本章では、この木構造をリレーショナルデータベースで扱う方法にどのようなものがあるか、その選択肢を一つ一つ検討していきます。その中には、まだ現在のリレーショナルデータベースの実装では条件が整っていないために現実的な利用が難しいものも含まれています。しかし、いずれもよく考えられた面白いものばかりなので、「未来のリレーショナルデータベース」における論理設計のモデルを、見ていくことにしましょう。

木構造とは？

最初に、木構造のデータがそもそもどのようなものか、という点をはっきりさせておきましょう。木構造というのは、名前のとおり、データが「木」の形をしているような階層状の構造です。この構造は私たちの周りにもたくさんあって、たとえば会社などの組織や、製品の部品同士の関係などは、階層構造を持っています。第6章で見たインデックスもこの構造でした。ある会社の組織を例に図示してみましょう（図9-1）。

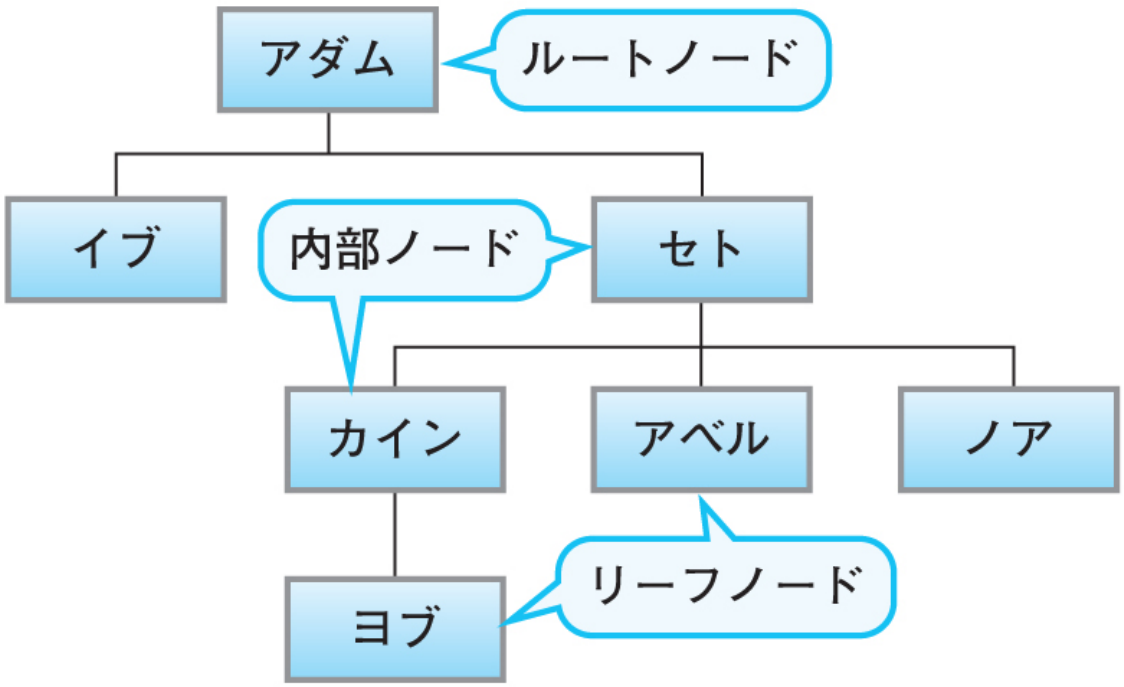


図9-1●木構造の組織図

アダム氏がトップ（社長）で、その下に部下が次々ぶら下がっていく構造になっています。この見た目が木に似ているので、そのまま木構造と呼ばれています。この木構造を扱うに当たり、最初にいくつか用語を定義しておきましょう。

- ノード（node）
  木の結節点のことです。今は組織図なので一人一人の社員がノードに相当します。
- ルートノード（root node)
  木が始まるトップのノードです。図9-1ではアダム社長がそうです。「根っこ」という意味です。木は、このルートノードを定義上一つしか持ちません。
- リーフノード（leaf node）
  自分よりも下位のノードを持たない「終着点」のノードです。この図では、ヒラ社員のイブ、ヨブ、アベル、ノアの四氏が相当します。「葉っぱ」という意味です。
- 内部ノード（inner node）
  ルートでもリーフでもない中間のノードのことです。
- 経路（path）
  あるノードから別のノードへたどる道筋のことです。「パス」と片仮名表記もします。たとえば、アダム社長からアベル氏ならば、「アダム→セト→アベル」が経路（パス）になります。なお、途中で他のノードへ寄り道をする経路（たとえば「アダム→イブ→アダム→セト→アベル」）も考えられます。

さて、これで準備が整いました。それでは、リレーショナルデータベースが木構造をどのように扱うか見ていきましょう。

リレーショナルデータベースで木構造を扱う方法として、最も古くから知られている方法が、**隣接リストモデル**（Adjacency List Model）です。これは、ノードのレコードに親ノードの情報（ポインタ）も持たせようとするものです。これはすなわち、コラム「関係（リレーショナル）とは何か？」（81ページ）で見た、ポインタチェインの構造をリレーショナルデータベースで表現したテーブルです。

■隣接リストモデル

組織図

社員	上司
アダム	
イブ	アダム
セト	アダム
カイン	セト
アベル	セト

ノア	セト
ヨブ	カイン

社員を1レコードとして保持するため、主キーは社員名です［※1］。社員は一人の上司にしか所属しないため、「上司」列によって誰が親ノードであるかがわかります。ルートの社長だけは上司がいませんので、アダム社長の「上司」列だけはNULLになります。

このモデルは、SQL で木構造を表現する最も古く、かつポピュラーな方法です。そのため、DBMSベンダーが木構造を扱うための機能開発を行なう際も、基本的にこのモデルを前提にしています。しかし、このモデルには、実用的な問題も指摘されています。本書では詳細を省きますが、更新や検索のクエリが（ベンダー独自拡張を使わないと）極めて複雑になり、パフォーマンスも悪いという欠点があります［※2］。

こうした欠点に対処し、なおかつ標準SQLのみを使用して階層データの取り扱いを実現するために考案された方法が、これから紹介する入れ子集合モデルです。このモデルは、隣接リストモデルとは対極的なアプローチを採用します。

同姓同名がありえるため、本来は「社員コード」列を作るのが望ましいのですが、ここでは簡略化します。

隣接リストモデルにおける検索や更新SQLの詳細については、ジョー・セルコ『プログラマのためのSQL 第2版』（ピアソンエデュケーション、2001）の「第28章 SQLの木構造の隣接リストモデル」を参照。

9-3 新しい解法 ～入れ子集合モデル

隣接リストモデルにおいては、各ノード（ここでは社員）は、文字通り結節「点」として考えられていました。この「点」は、直径も面積も持ちません。一方、**入れ子集合モデル**（Nested Sets Model）は、ノードを点ではなく面積を持った「円」としてとらえます。そして、ノード間の階層関係を円の包含関係によって表わします。

勘どころ 63

入れ子集合モデルでは、ノードを円と見なす。

論より証拠、実際に見たほうがイメージをつかみやすいでしょう。テーブルの構成と入れ子集合のイメージ図を図9-2に示します。

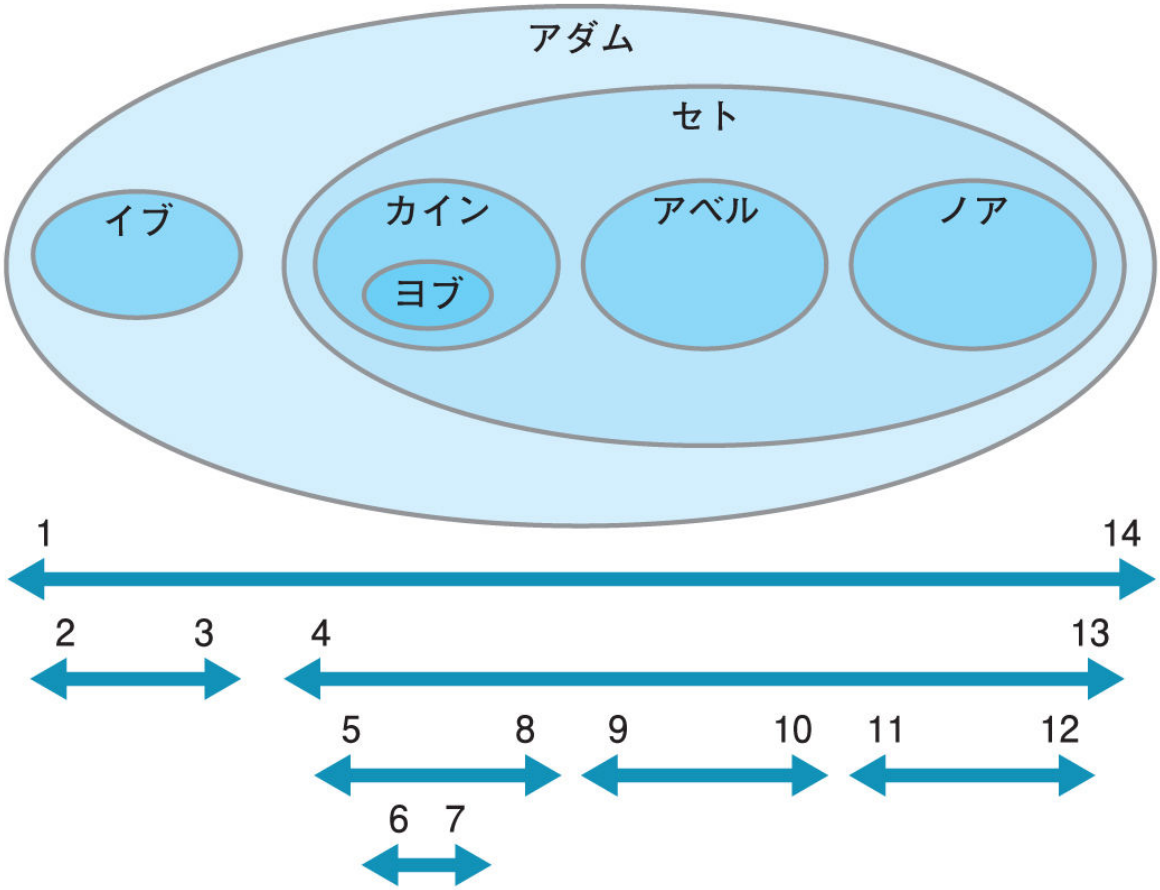


図9-2●入れ子集合で木構造を表現する

■入れ子集合モデル

組織図

社員	左端	右端
アダム	1	14
イブ	2	3
セト	4	13

カイン	5	8
ヨブ	6	7
アベル	9	10
ノア	11	12

1行が一人の社員を表わすことは、隣接リストモデルと変わりありません。「上司」列の変わりに追加された列「左端」と「右端」が入れ子集合モデルのかなめです。これは円の左端と右端の座標を表現しています。したがって「右端 − 左端」という引き算で円の直径が求まります。

このモデルに立って考えた場合、上司は、自分の腹の中に部下を抱え込む格好になります。読んで字のごとく「腹心」です。したがって、上司が部下の円をきちんと包含できるように各円の座標を割り当てる必要があります。実際のところ、座標は一続きの連番である必要はないのですが、最初はわかりやすくするために、歯抜けのない連番で考えます。

入れ子集合モデルを使った検索

この入れ子集合モデルの大きな利点は、木構造を操作するためのSQL文が、隣接リストモデルに比べて非常にシンプルになることです。まずは、検索から見ていきましょう。

ルートとリーフを求める

木の操作で最も基本となるのは、ルートとリーフのノードを探すことです。入れ子集合モデルでこれらを求めることは非常に簡単です。まず、ルートは必ず左端の座標が1になるので、こうなります。

■ルートを求める

```
SELECT *
  FROM 組織図
 WHERE 左端 = 1;
```

結果

```
社員     左端   右端
----   ---   ---
アダム     1    14
```

一方、リーフとは自分の中に他の円を一つも含まない円のことです。これは、次のように「組織図」テーブルを上司と部下に見立てて自己結合を行ないます。

■リーフを求める

```
SELECT *
  FROM 組織図 上司
 WHERE NOT EXISTS
       (SELECT *
          FROM 組織図 部下
         WHERE 部下.左端 > 上司.左端
           AND 部下.左端 < 上司.右端);
```

結果

```
社員    左端   右端
----   ---   ---
イブ      2     3
ヨブ      6     7
アベル    9    10
ノア     11    12
```

また、これと同じ考え方を使えば、ルートの左端の座標が1でない場合でも、ルートを求めることが可能です。ルートとはリーフの裏返しで、他のどんな円にも含まれない円、ということですから、次のように記述できます。

■ルートを求める（左端が１でなくてもOK）

```
SELECT *
  FROM 組織図 部下
 WHERE NOT EXISTS
       (SELECT *
          FROM 組織図 上司
         WHERE 部下.左端 >上司.左端
           AND 部下.右端 < 上司.右端);
```

結果

```
社員     左端   右端
----   ---   ---
アダム     1    14
```

木の深さを求める

残りの木の操作も、すべてこの「包含関係」の応用で記述することができます。たとえば、あるノードの「深さ」（階層の位置）を求める場合は、「自分を包含する円が何個あるか」という風に「包含関係」に翻訳してやれば良いのです。

■ノードの深さを計算する

```
SELECT 部下.社員 , COUNT(上司.社員) AS 深さ
  FROM 組織図 上司 INNER JOIN 組織図 部下
    ON 部下.左端 BETWEEN 上司.左端 AND 上司.右端
GROUP BY 部下.社員;
```

結果

```
社員    深さ
----    ---
アダム     1
セト       2
イブ       2
カイン     3
ノア       3
アベル     3
ヨブ       4
```

このSQLでは、BETWEENを使っているので、自分も数えています。もし深さを0から始めたければ、不等号の条件に変えて自分を除外するか、あるいは単純にSELECT句の深さを計算している箇所を「COUNT(上司.社員)-1」としてもいいでしょう。木の「高さ」を求める場合も、この結果から最大の深さを求めてやれば、「4」という結果がすぐに導き出せます。

入れ子集合モデルを使った更新

このように入れ子集合モデルは、木構造の検索においては大きな力を発揮しますが、しかし一方で、更新については問題を抱えています。今から、その問題を見て

いきます。

ノードの追加

組織である以上、恒常的に人事異動が発生します。そのため、木の形も常に変化していきます。新たに雇われた社員がいればノードを追加する必要がありますし、退職した社員がいればノードを削除する必要があります。また昇進／降格といった人事異動によって、社員同士の関係が変化することもあるでしょう。

まず、ノードを追加する場合は、リーフとして追加するのか、親として追加するのかによって処理が分かれます。基本はリーフを追加する場合を考えれば良いため、こちらを取り上げます。

たとえば、イブ氏の配下に新たにイサク氏を加えることを考えましょう。この場合、イブ氏の座標は（2, 3）ですから、**整数値を使う以上**、部下のノードを抱え込むことができません。そこでまずは、イブ氏の円を広げてやることから始めます。

■［第 1 段階］追加するノードの席を空ける

```
UPDATE 組織図
   SET 左端 = CASE WHEN 左端 > 3
                   THEN 左端 + 2
                   ELSE 左端 END,
       右端 = CASE WHEN 右端 >= 3
                   THEN 右端 + 2
                   ELSE 右端 END
 WHERE 右端 >= 3;
```

結果 イブ氏より右のノードが右へずれ、隙間が生まれる

```
社員     左端   右端
-----    ---    ---
アダム     1     16
イブ       2      5
セト       6     15
カイン     7     10
ヨブ       8      9
アベル    11     12
ノア      13     14
```

部下（3, 4）を持つ余裕が生まれた

これでイブ氏の円が広がりました。後は、イサク氏を追加するだけです。

■［第2 段階］イサク氏を追加する

```
INSERT INTO 組織図 VALUES ('イサク', 3, 4);
```

イサク氏を追加した後の組織図の入れ子集合のイメージは図9-3のようになります。イブ氏の円が広がり、イブ氏より右側の円たちが右にずれていることがわかります。

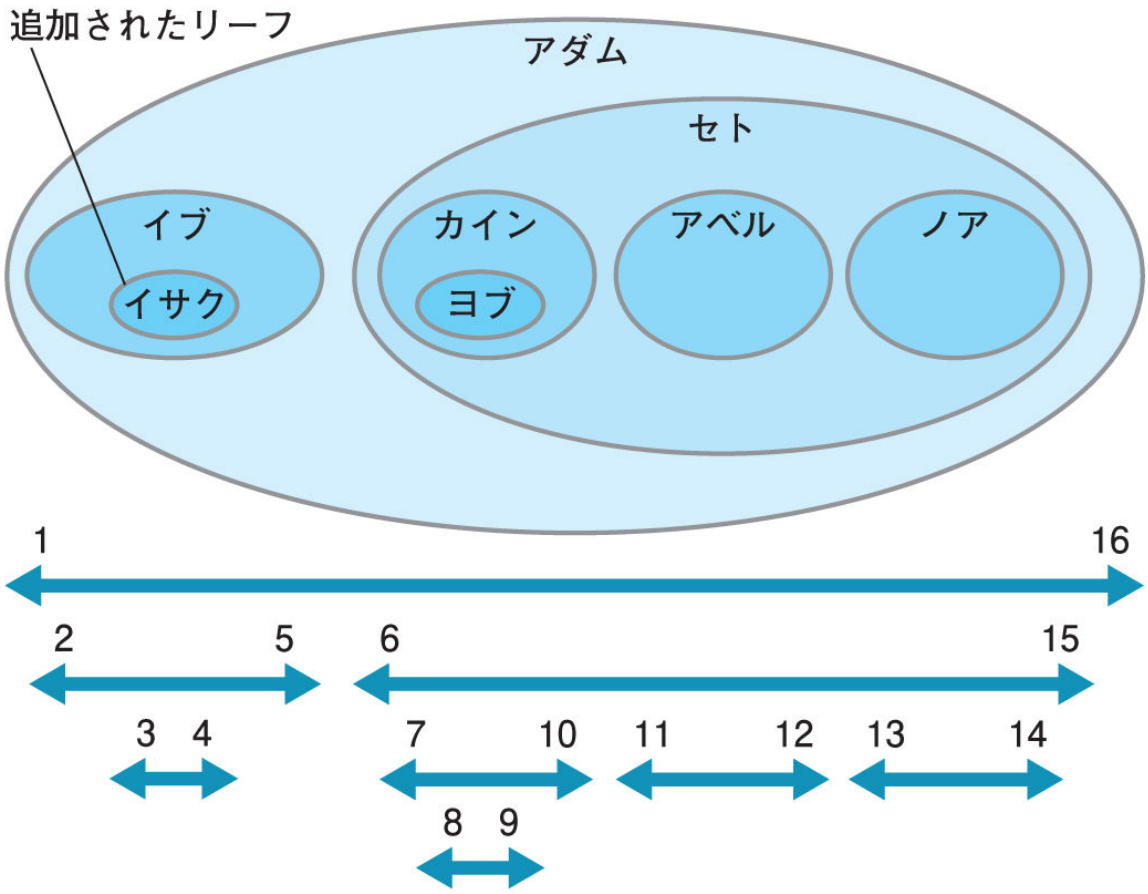


図9-3●リーフ追加後の入れ子集合のイメージ

実はこの例にも見られるような「更新対象と無関係な円の座標も連動して更新しなければならない」という点が、入れ子集合モデルの最大の弱点です。これは動かす円の数が増えれば増えるほど、更新対象のレコードも増えるということですから、データ量が多い場合には更新負荷とロック競合によって深刻なパフォーマンス問題を引き起こします。

勘どころ 64

入れ子集合モデルの弱点は更新時のパフォーマンス。

この問題にどう対処するかという点については次節に譲るとして、ここはひとまず、ノードの削除のケースも見ておきましょう。

ノードの削除

リーフを削除する場合は、単純にそのレコードを削除するだけなので非常に簡単です。末端社員の首を切るのはいつでもやりやすいものです。対して、部下を抱えている上司の首を切るときは、いろいろ考えることがあります。まず、その上司一人だけを削除するのか、それとも部下も連座して削除するのかで処理が分かれます。上司一人だけを削除する場合も、その上司のレコードを削除するだけなので、話は簡単です（後で後任の人間を設定する場合はまた別途処理が必要ですが）。一方、部下も連座して削除する場合は、そのノードに含まれるノードも一括で削除します。たとえば、カイン氏を解雇すると、部下のヨブ氏も連座して解雇されます。これを実現するには、次のようにノードの座標を範囲指定して削除します。

■カイン氏を解雇（ヨブ氏も連座）

```
DELETE FROM 組織図
 WHERE 左端 BETWEEN (SELECT 左端 FROM 組織図 WHERE 社員 = 'カイン')
                AND (SELECT 右端 FROM 組織図 WHERE 社員 = 'カイン');
```

このように、下位のノードも含めたノードを**部分木**と呼びます。木の内部で新たにルートを見出して、小さな木と見なすイメージです。ここでは、カイン氏をルートと見なした場合の部分木を削除したわけです（図9-4）。

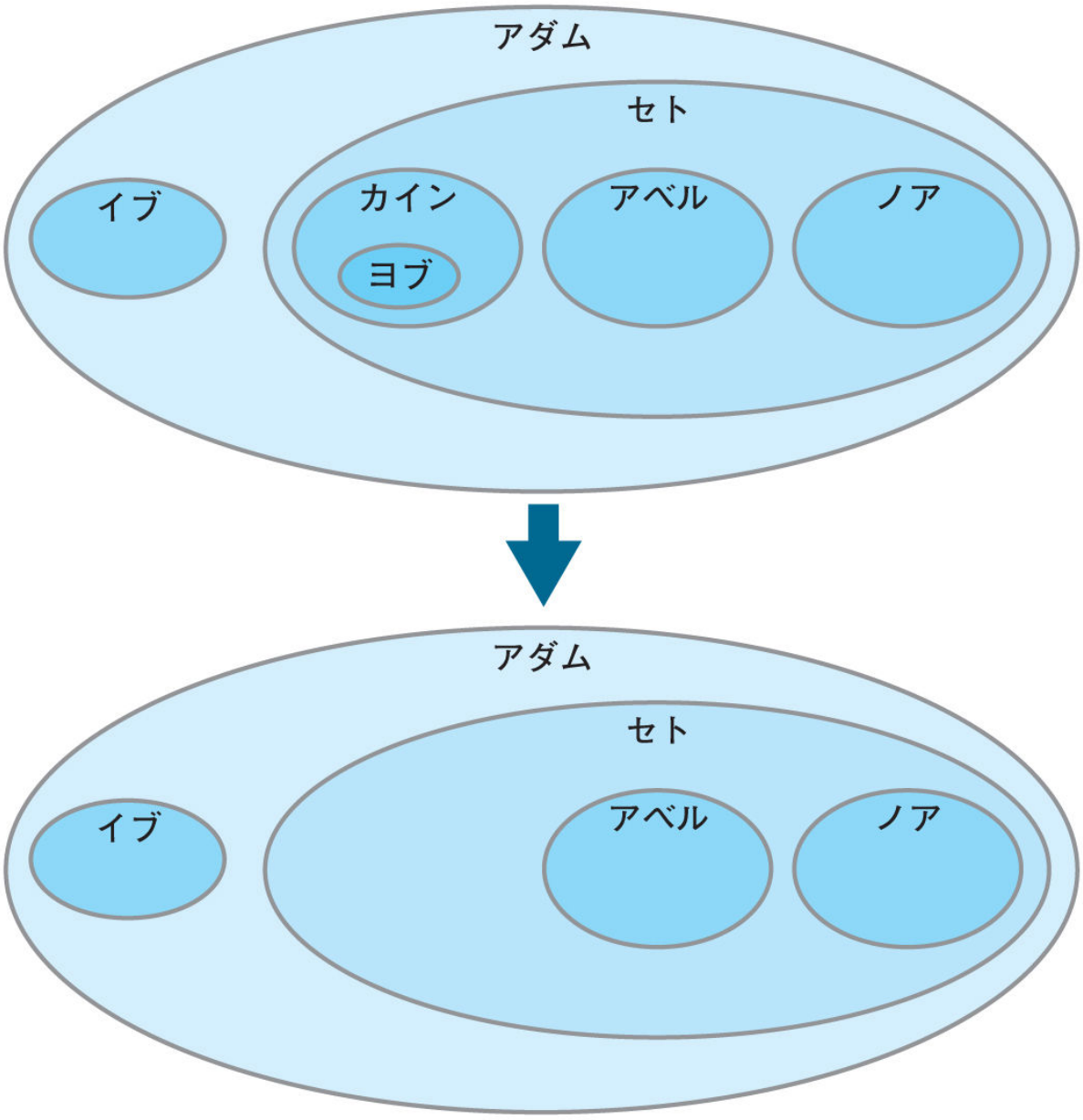


図9-4●部分木の削除イメージ

この削除によって、座標には歯抜けの値が生まれたわけですが、これは特に更新しなくても問題はありません。入れ子集合では、座標の絶対値が重要なわけではなく、包含関係という相対的な関係が保持されていれば十分だからです。そして、

この「相対関係さえ維持されていればOK」という考え方が、入れ子集合モデルの更新時のパフォーマンス問題を解決する鍵になるのです。それを次節で見ることにします。

9-4 もしも無限の資源があったなら～入れ子区間モデル

前節で確認したように、入れ子集合モデルの欠点は、ノードを挿入（追加）するときに、自分より「右側」にある無関係なノードをもっと右へずらさなければならないことでした。あらかじめノード間の座標に隙間を作っておいて初期データを登録するという方法もありますが、根本的解決ではありません。このリソース枯渇の問題は、座標に整数を使う以上、原理的に不可避なものです。

使っても使っても尽きない資源

整数――と著者は今言いました。ここが本節のポイントです。実際のところ、入れ子集合の左端／右端の座標に整数を使う必要はあるのでしょうか？

この答えは「別にない」となります。確かにキリのいい整数値を使えば、図示したときに簡単に互いの円の関係がわかりやすい、というだけで、実用的な観点から考えれば整数を使う必要は別にないのです。

そこで、座標のとれる範囲を整数から実数にまで広げることが、有効な解決策になります（図9-5）。

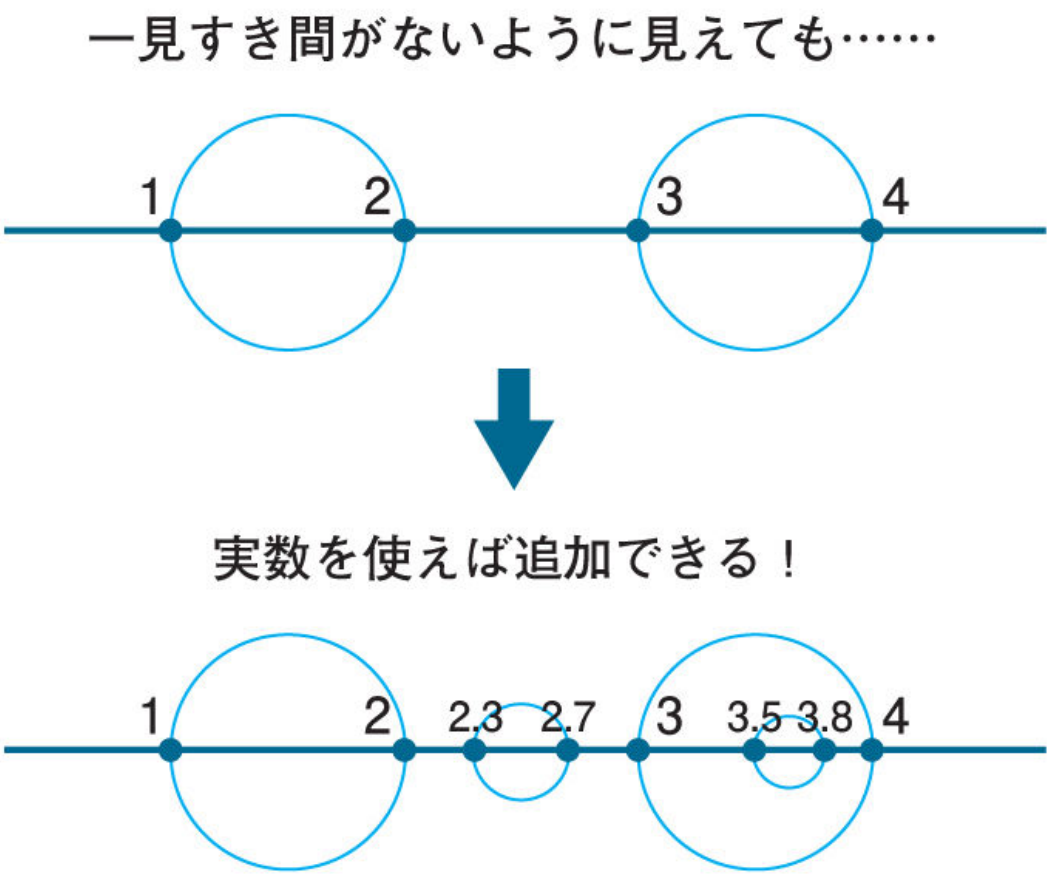


図9-5●座標のとれる範囲を整数から実数にまで広げたら

このように、円の左端／右端の座標値として、とれる範囲を実数まで広げた、入れ子集合モデルの拡張版が、**入れ子区間モデル**（Nested Intervals Model）です。

実数は、その定義上、無限にあります。かつ、どんな二つの実数の間にも無限にギッシリ詰まっています（この性質を稠密性（ちゅうみつせい）と呼びます）。したがって、どんなにすき間がないように見える二つの円の間であっても、新たに何個でも円を追加できてしまうのです。実数とは、使っても使っても減らない、夢のような**リソース無限**の世界なのです。

勘どころ 65

実数は汲めども尽きぬ無限の泉。

もちろん、お気づきのとおり、これはあくまで理論上の話です。実際には実数型といえどもDBMSにおいて定義された有効桁数が限界になるので、小さな隙間に円を追加することを繰り返せば、遠からず枯渇するときが来ます。その意味で、これは「未来のモデル」と言えるでしょう。現状の有効桁数が不十分であったとしても、将来、これが拡張されて条件が整えば、リレーショナルデータベースで木構造を扱う「本命」の手段になるでしょう。

入れ子区間モデルを使った更新

それでは、前節で懸案だったノード追加が、入れ子区間モデルではどのようになるかを見てみましょう［※3］。

前節の「入れ子集合モデルを使った更新」（271ページ）と同様に、イブ氏の下にイサク氏を部下としてつけることを考えます。入れ子区間モデルでは、イブ氏の（2, 3）の円の中にもノードの座標をとることが可能です。その座標を決める方法は難しくありません。挿入対象としたい区間の左端座標をplft、右端座標をprgtとすると、次の数式によって自動的に追加ノードの座標を計算できます。

追加ノードの左端座標 ➡ (plft * 2 + prgt)/ 3
追加ノードの右端座標 ➡ (plft + prgt * 2)/ 3

なぜこれでうまくいくかというと、上記4つの座標について、必ず次の関係が成立するからです（図9-6）。

plft <(plft * 2 + prgt)/ 3 <(plft + prgt * 2)/ 3 < prgt

この理由は、各辺を3倍してみればわかります。

plft * 3 < plft * 2 + prgt < plft + prgt * 2 < prgt * 3

pleftは区間の左端、prgtは区間の右端であるという定義から、明らかにplft <prgtです。したがって、plft * 3 < plft * 2 + prgtが成り立ちます。同様に、plft *2 + prgt < plft + prgt * 2とplft + prgt * 2 < prgt * 3も成り立ちます。

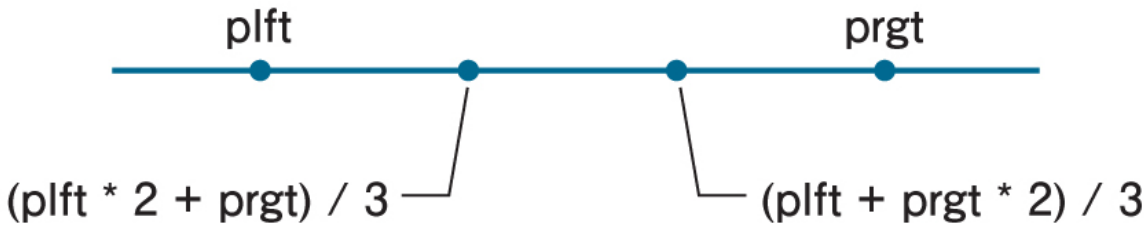


図9-6●4点の大小関係は常に成立する

この式に従えば、新たに追加するイサク氏の円の左端と右端の座標は以下のように求められます。

イサク氏の左端 ➡ (2 * 2 + 3)/ 3 = 2.3333....
イサク氏の右端 ➡ (2 + 3 * 2)/ 3 = 2.6666....

後は、求めた座標の円を挿入すればできあがりです。

図9-7に示すように、無関係なノードの座標をずらすことなく、新規ノードを追加することができました。

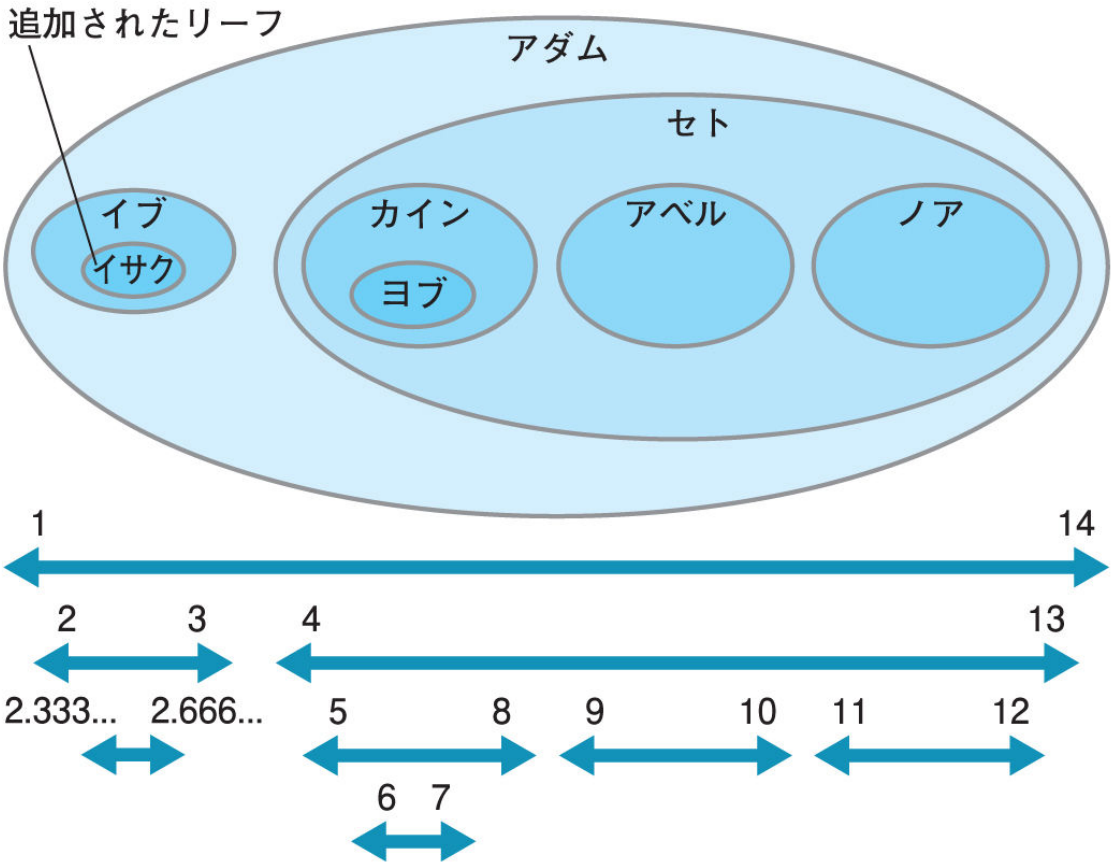


図9-7●実数を使えば、ノードをずらさなくても追加が可能

このように、入れ子区間モデルは、入れ子集合モデルの基本的な発想を踏襲しつつ、欠点を克服した進化版のモデルと言うことができます。これまでリレーショナルデータベースが苦手と言われていた木構造を柔軟に表現する方法であるため、今後の応用が期待されるところです。ただし、そのためには十分な実数の有効桁数が

必要になるため、現行のリレーショナルデータベースの実装で、このモデルを採用できるかどうかは未知数です。

検索のSQLは入れ子集合モデルと変わりません。整数が実数になっても、ノードの位置関係が変わるわけではないからです。

9-5 ノードをフォルダだと思え ～経路列挙モデル

ここまでで、リレーショナルデータベースで木構造を扱う方法論として、隣接リストモデルと入れ子集合モデル（およびその進化版である入れ子区間モデル）について解説しました。本節では、四つ目のモデルを紹介しましょう。名前は、「**経路列挙モデル**（Path Enumeration Model）」または「**経路実体化モデル**（Materialized Path Model）」です。以下、便宜上「経路列挙モデル」と呼ぶことにします。

ファイルシステムとしての階層

このモデルの発想は、ある意味でこれまでに紹介したモデルのどれよりも、私たちにとって馴染み深いものなので、考え方はすぐに理解できると思います。というのも、私たち（多分本書を読んでいるみなさん全員）が、この発想に基づいて階層構造を管理しているからです。その発想とは次のようなものです。

勘どころ 66

ノードをディレクトリ（フォルダ）と見なし、各ノードまでの経路（path）を記述する。

ディレクトリ（フォルダ）は、私たちがパソコンを使うときにファイルを入れておくあの仮想的な整理箱です。入れ子集合モデルでは、ノードを「円」に見立てました。今度はノードをディレクトリに見立てるのです。論より証拠、組織図をディレクトリで表現してみましょう（図9-8）。

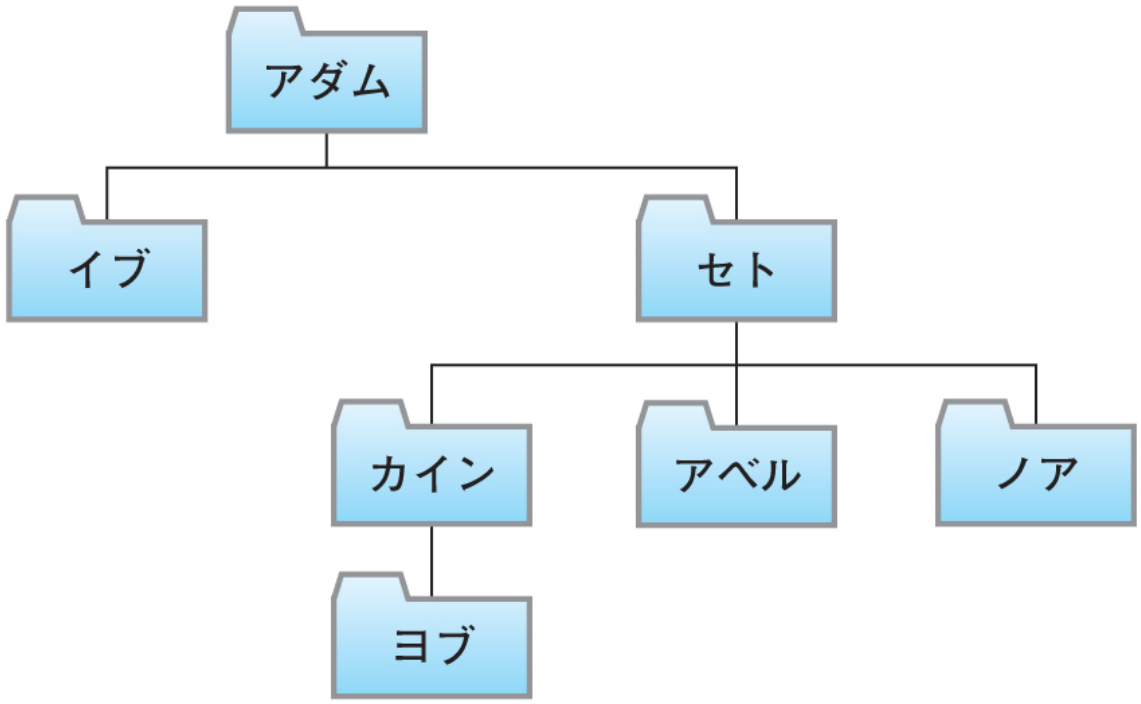


図9-8●組織図をディレクトリで表現する

いかがでしょう、ノードがディレクトリに見えてきたでしょうか？

それでは、このモデルを表現するテーブルも作ってみましょう。具体的には、方法が二通り存在します。

■経路列挙モデル1（主キーを使う場合）

組織図

社員	経路
アダム	/アダム/
イブ	/アダム/イブ/
セト	/アダム/セト/
カイン	/アダム/セト/カイン/
ヨブ	/アダム/セト/カイン/ヨブ/
アベル	/アダム/セト/アベル/
ノア	/アダム/セト/ノア/

■経路列挙モデル2（番号を使う場合）

組織図

社員	経路
アダム	.1.
イブ	.1.1.
セト	.1.2.

カイン	.1.2.1.
ヨブ	.1.2.1.1.
アベル	.1.2.2.
ノア	.1.2.3.

どちらの方法でも、1行が一人の社員（ノード）を表わすことは、入れ子集合モデルや隣接リストモデルと変わりません。このモデル最大の特徴は、「経路」列にあります。この列は、各ノードまでの**絶対パス**を保持します。
「経路列挙モデル1」の方法は、主キーを経路として使うもので、「経路列挙モデル2」の方法は、経路の階層を数値化して利用します（よく書籍の章立てなどで見る形式です）。

このモデルの特性を説明するには、「経路列挙モデル1」のテーブルを見てもらうのが良いでしょう。区切り文字に「/」を使っていますが、これはまさにUNIX/Linuxのファイルシステムの構造そのものです（Windowsに慣れている人は「¥」に置き換えて考えてください）。OSのファイルシステムも木構造ですから、同じ表現ができるのは当然といえば当然のことですが、まさに「コロンブスの卵」です。

入れ子集合モデルでは、円の包含関係で階層を表現したのに対し、今度はパス文字列の包含関係で表現しています。面白いことに経路列挙モデルの場合、子のパスが親のパスを含むという、入れ子集合とは反対の包含関係になっています。たとえば、ノア氏の「アダム/セト/ノア」というパスは、その親であるセト氏の「アダム/セト」やアダム氏の「アダム」といったパスを含みます。ルートのパスは全ノードのパ

スの部分集合になっています。子は親たちの膨大な情報を受け継いでいく存在なのです。

経路列挙モデルの利点と欠点

この経路列挙モデルの利点は、**検索のパフォーマンスが良い**ことです。ノード自身のレコードに親子関係が含まれているので、他のモデルに比べて経路探索のSQL文が圧倒的に簡単になるからです。特に、PostgreSQL やOracleなど正規表現を利用できるデータベースと、高い親和性を示します。パフォーマンス面においても、パスもテーブル上で一意になるため、**ユニークインデックスによる高速検索が可能**です。

反対に欠点は、以下に挙げるとおりです。

- 経路に主キーを使うと、経路の文字列が非常に長大になる危険がある（番号を使う場合にはこの心配は少ない）。DBMSの可変長文字列の上限値を超えると、現実的な利用が難しい。
- 経路に主キーを使うと、同じ階層内のノード同士の順序が一般的に把握できない（入れ子集合モデルでは、ノードの座標位置で示すことができた）。ただし、番号を使うことで、この欠点は補える。
- しかし、パスに番号を使うと、ノードの削除、追加などの更新が複雑になる。
- 検索SQLで文字列操作を多用するため、文字列関数の標準化が遅れているSQLでは、かなり実装依存のコードになってしまう［※4］。

こうした利点と欠点を総合して判断すると、更新が少なく、大量データの高速な検索が必要なケース（たとえばDWH/BI）に向いているモデルと言えます。

勘どころ 67

経路列挙モデルは検索が簡単で更新が複雑。

なお、主キーを使う場合と番号を使う場合、どちらの方法をとっても、経路の区切り文字が大変重要な役割を果たします。この区切り文字がないと、たとえば「エレミヤノア」というパスが「エレミヤ/ノア」なのか「エレミ/ヤノア」なのか、あるいは「111」が「1.11」なのか「1.1.1」なのか区別がつきません。また、パスの先頭と終端は必ず区切り文字で囲む必要があります。これは一見、些細なことに思えるかもしれませんが、これを囲むと囲まないとでは、更新クエリの効率に大きな違いが出るのです［※5］。

経路列挙モデルによる検索

それでは、入れ子集合モデルのときと同じように、経路列挙モデルにおける検索と更新の方法を見てみましょう。まず、基本となる操作は、やはりルートとリーフを探すことです。

ルートとリーフを探す

経路列挙モデルでのこうした検索は、とても簡単です。経路に主キーを使っている場合は「区切り文字を削除した文字列がキーと同じになる」という条件で検索すれば良いのです。

■ルートを求める［経路が主キーの場合］

```
SELECT *
  FROM 組織図
 WHERE 社員 = REPLACE(経路, '/', '');
```

結果

```
社員    経路
----    ------
アダム  /アダム/
```

このSQL文は、主キーである「社員」列のユニークインデックスの一意検索であるため、非常に高速です。

REPLACEはほとんどのDBMSで利用できる標準関数で、第1引数の文字列の中から、第2引数の文字列を見つけ出し、第3引数の文字列で置換します。今は、第3引数が空文字（''）なので、結局第2引数の文字列を削除する意味になります。

また、番号を使う場合であれば、ずばり経路が「1」のノードを探すだけです。

■ルートを求める［経路が番号の場合］

```
SELECT *
  FROM 組織図
 WHERE 経路 = '.1.';
```

結果

```
社員    経路
----    ---
アダム  .1.
```

一方、リーフを求めるクエリは、ちょっと考える必要があります。リーフであるイブ、ヨブ、アベル、ノアの四氏と、そうでないアダム、セト氏らを比べてみてください。リーフノードが共通して持ち、そうでないノードが持たない性質が見えてこないでしょうか？

それは、以下のような性質です。

自分のパスの後ろに追加のパスを持つような他のノードが存在しない

たとえば、リーフでないセト氏のパス「/アダム/セト/」について見ると、「/アダム/セト/カイン/」（カイン）、「/アダム/セト/ノア/」（ノア）のように、「自分のパス＋追加のパス」を自らのパスとするようなノードが他に存在します。リーフの人々は、こういうノードを持ちません。自分で打ち止めです（それがリーフの定義なのだから当然ですが）。そこで、この条件を NOT EXISTSで表わしましょう。

■リーフを求める

```
SELECT *
  FROM 組織図 親
 WHERE NOT EXISTS
       (SELECT *
          FROM 組織図 子
         WHERE 子.経路 LIKE 親.経路 || '_%');
```

結果

```
社員    経路
----    --------------------
イブ    /アダム/イブ/
ヨブ    /アダム/セト/カイン/ヨブ/
アベル  /アダム/セト/アベル/
ノア    /アダム/セト/ノア/
```

「'_%'」は、「（一文字以上の）任意の文字列」を意味する文字列のパターンです。LIKEは前方一致検索においてはインデックスを使用するので、「経路」列のユニークインデックスが利用できて、やはりパフォーマンスは良好です。文字列を結合する||演算子はれっきとした標準SQLの関数ですが、使えるかどうかは実装に左右されます。Oracle、PostgreSQL、DB2では問題なく使えますが、SQL Serverでは代わりに「+」、MySQLではCONCATを使う必要があります。

このように、SQLの文字列操作の標準化が遅れているため、実装ごとに異なるコードを使わなければならないのが、経路列挙モデルの欠点の一つです。しかし、これは先述のようにモデルそのものの欠点ではありませんし、時間がたてば標準SQLは多くの実装がサポートしていくので、いずれは解決される問題です。

木の深さを求める

では今度は、各ノードの「深さ」を計算してみましょう。ここでも、経路を直接データとして保持している利点が活きてきます。「経路」列に含まれる区切り文字を数えれば、0を開始値とする深さが求められます。

区切り文字の数を数える方法は、「（元の文字列長）－（区切り文字を削除した文字列長）」という引き算で可能です。たとえば、「/アダム/セト/アベル/」（文字数＝11）から「アダムセトアベル」（文字数＝8）を引けば、区切り文字の数「3」が求められます。後は「両端に植木がある場合」の簡単な植木算です。

■ノードの深さを計算する

```
SELECT 社員, LENGTH(経路) - LENGTH(REPLACE(経路, '/', '')) -1 AS 深さ
  FROM 組織図;
```

結果

```
社員     深さ
----   ---
アダム     1
セト       2
イブ       2
カイン     3
ノア       3
アベル     3
ヨブ       4
```

入れ子集合モデルでは、この深さを円の包含関係として定義して求めたため、自己結合が必要になりました。それに比べれば、結合を一切必要としないこのSQL文は、「組織図」テーブルに対するフルスキャン1回のみですから、パフォーマンスは高速なものが期待できます。

経路列挙モデルを使った更新

このように検索に関してはいいことずくめの経路列挙モデルですが、その対価として更新処理は複雑になります。どのように難しくなるのか、具体例を通して見ていきましょう

ノードの追加

木にノードを追加するときは、リーフとして追加するのか、それとも親として追加するのかによって処理が分かれます。リーフを追加するには、どの親ノードの配下に付けるかさえわかれば、その親の経路に自分を追加すれば、自分の経路ができあがります。パスに番号を使う場合も、基本的には同じです。

一方、親ノードとして追加するのはちょっと大変です。木の「中間」にぐいっと新しいノードを押し込むようなイメージですが。これは、単純に木の下端に追加するだけで良かったリーフよりも骨が折れます。新たに持つことになる子の数だけ処理を繰り返す必要があるため、単発のSQLでは解決できません。

たとえば、アベル氏とノア氏の上司としてイサク氏を追加するなら、次のようになります（図9-9）。

■親ノードを追加する

```
--1. 親の新規追加
INSERT INTO 組織図
VALUES ('イサク', (SELECT 経路 FROM 組織図 WHERE 社員 = 'セト') ||
'イサク' || '/');

--2-1. アベル氏の部分木のパス更新
UPDATE 組織図
   SET 経路 = REPLACE(経路, '/' || 'アベル' || '/', '/' ||
'イサク' || '/' || 'アベル' || '/')
 WHERE 経路 LIKE '%/' || 'アベル' || '/%';

--2-2. ノア氏の部分木のパス更新
UPDATE 組織図
   SET 経路 = REPLACE(経路, '/' || 'ノア' || '/', '/' || 'イサク'
|| '/' || 'ノア' || '/')
 WHERE 経路 LIKE '%/' || 'ノア' || '/%';
```

※誌面の都合上で折り返しています。

結果

```
社員    経路
----    ----------------------
アダム  /アダム/
イブ    /アダム/イブ/
セト    /アダム/セト/
カイン  /アダム/セト/カイン/
ヨブ    /アダム/セト/カイン/ヨブ/
イサク  /アダム/セト/イサク/        ←― 親として追加
アベル  /アダム/セト/イサク/アベル/
ノア    /アダム/セト/イサク/ノア/
```

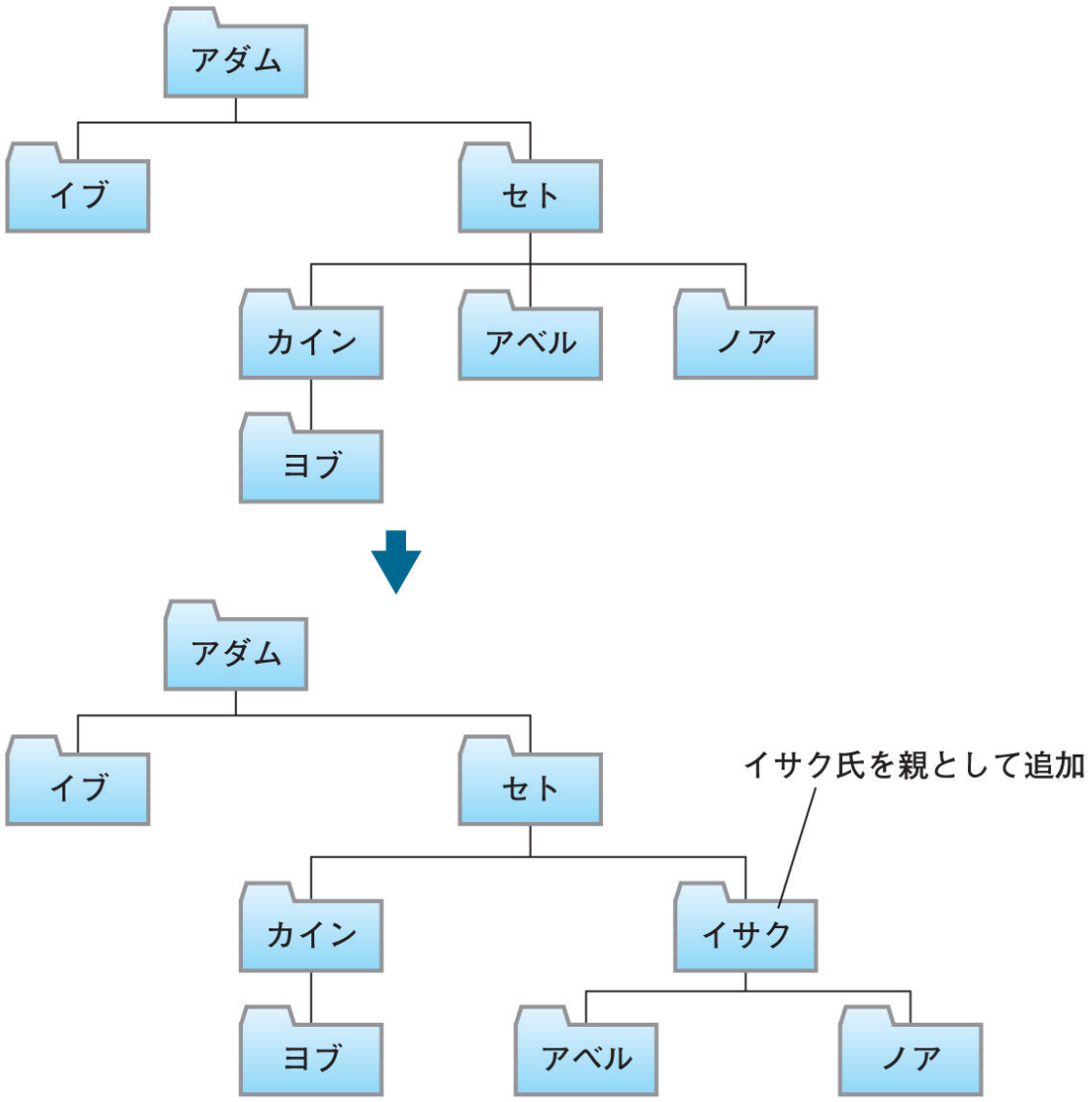


図9-9●親としてノードを追加するイメージ（経路列挙モデル）

新たに抱える子ノードの数だけ、経路更新のUPDATE文をループで繰り返す必要があります。今回は二人なので、列挙してもたいしたことはありませんが、一般的にはプロシージャ内のループで処理することになるでしょう。

入れ子区間モデルでは、ノードの追加が非常に簡単で、既存のノードに対して影響を与えなかったことと比べると、経路列挙モデルの更新の複雑さは対照的です。

部分木の削除

部分木を削除するとは、あるノードをルートに見立て、それにぶら下がるノードも一括して削除することでした。経路列挙モデルでは、あるノードが誰かの部下であれば、その上司の経路を必ず含んでいます。したがって、解雇対象の社員の経路を含むノードをすべて削除します。これは単純に考えれば、次のようにLIKEによる文字列の中間一致検索で可能です（図9-10）。

■部分木を削除する

```
--カイン氏を解雇（ヨブ氏も連座）
DELETE FROM 組織図
 WHERE 経路 LIKE '%/' || 'カイン' || '/%';
```

結果 解雇後の組織図

```
社員    経路
----    ----------------
アダム  /アダム/
イブ    /アダム/イブ/
セト    /アダム/セト/
アベル  /アダム/セト/アベル/
ノア    /アダム/セト/ノア/
```

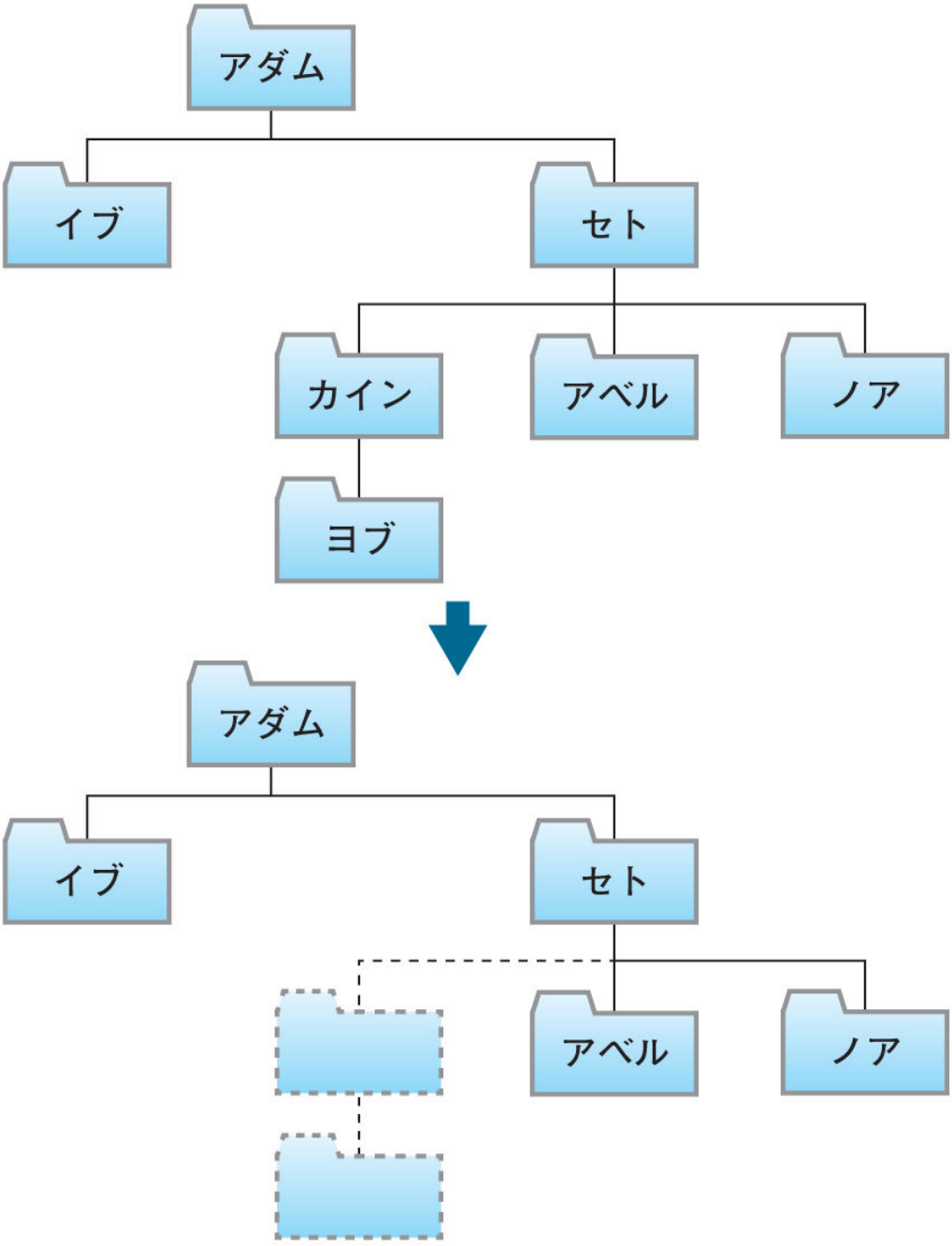


図9-10●部分木の削除（経路列挙モデルのイメージ）

しかしこれはあまり良いSQL文ではありません。まず、経路に主キーを使う場合にしか使えません。また、LIKEの中間一致はインデックスを利用できないため、「組織

図」テーブルに対するフルスキャンを余儀なくされ、パフォーマンスもよくありません（第6章参照）。

この二つの欠点を改良するために、以下のようなサブクエリを利用する方法が考えられます。

■部分木を削除する［パフォーマンス改良版］

```
DELETE FROM 組織図
 WHERE 経路 LIKE (SELECT 経路
                    FROM 組織図
                   WHERE 社員 = 'カイン') || '%';
```

まずサブクエリ内のSELECT文によって、削除対象となる部分木のルートの経路を求めます。残りの子ノード（自分も含めて）たちは、いずれもこの経路を前半に含む経路を持っているので、LIKEの前方一致検索が使えるようになります。経路が主キーでも番号でも使えて、かつサブクエリの内と外、どちらのSELECT文でもインデックスが有効になるため、パフォーマンスも良好です。

もっとも、これをモデルの欠点と言うのはかわいそうで、むしろ SQLの欠点と言うべきです。

UNIXのパスが/home/usr/bin/のように区切り文字で囲まれているのも、きっと同じ理由によるのだと思います。

本章では、木構造をリレーショナルデータベースのテーブルでどのように扱うか、その方法論を紹介してきました。各モデルを最後にまとめると、以下のようになります。

- 隣接リストモデル
  最も古典的なモデル。検索／更新ともに複雑な処理を必要とします。
  DBMS依存の機能を使うことでこの欠点を軽減できますが、あくまで独自機能に頼ることになるため、汎用性がありません。
- 入れ子集合モデル
  木を円の包含関係によって表現するモデル。検索のSQLが簡単になりますが、更新処理のパフォーマンスに問題を抱えています。
- 入れ子区間モデル
  入れ子集合モデルの拡張版。入れ子集合モデルの欠点だった更新処理のパフォーマンス問題を克服できます。ただし、実数型の有効桁数が十分に確保されていなければ実用的でありません。
- 経路列挙モデル
  ルートから各ノードまでの経路を保持するモデル。検索に強く更新に弱いのが特徴。

リレーショナルデータベースの論理設計といえば正規化と非正規化のせめぎあい、という印象が一般的です。しかしそれだけでは決着のつかない分野もある、ということです。そもそも、木構造とフラットな二次元表の相性が悪く、リレーショナルデータベースでは表現するに向かない構造なのです。それゆえ、木構造を表現するための専用のデータベース（XMLデータベース）も研究されています。しかし、リレーショナルデータベースも黙って指をくわえていたわけではなく、このように斬新なモデルによって従来の苦手を克服しつつあります。こうした分野では、今でも最適なモデリング方法についての研究が世界中で行なわれており、どんどん新しい成果が登場しているのです。

実際、隣接リストモデル以外の三つのモデルは、いずれも本格的な応用が考え始められたのは、2000年代に入ってからで、いずれも非常に新しい方法論です。特に入れ子区間モデルは、データ型の精度という物理的制約さえなければ、リレーショナルデータベースで階層構造を扱う方法論としては、検索／更新のパフォーマンスとモデルの簡潔さの双方において最も優れており、今すぐにでも採用して良いものです。著者は、これが近い将来において主流となるモデルだと予想しています。「リレーショナルデータベースは木構造が苦手だ」という定番の批判も、過去のものになる日は近いでしょう。

なお、リレーショナルデータベースにおける木構造の扱い方についてより深く勉強したいと思った方は、本書「おわりに」において参考文献として紹介している書籍を参考にしてください。本書では紹介しきれなかった応用方法が、まだまだ存在するのです。

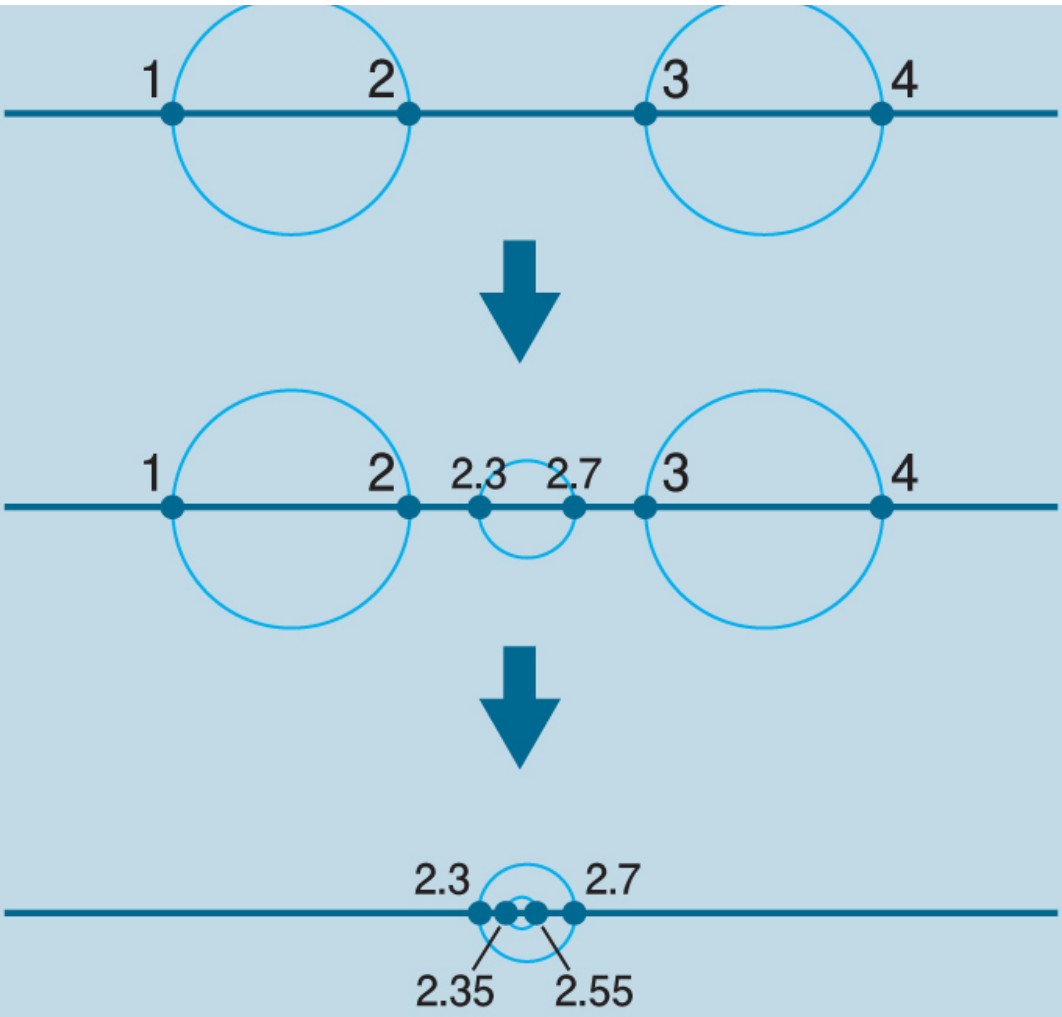
入れ子集合とフラクタル　COLUMN

入れ子区間モデルでノードを挿入するとき使った公式をもう一度思い出してく

入れ子区間モデルでノードを挿入するとき使った公式をもう一度思い出してください。

追加ノードの左端座標 ➡ (plft * 2 + prgt)/ 3
追加ノードの右端座標 ➡ (plft + prgt * 2)/ 3

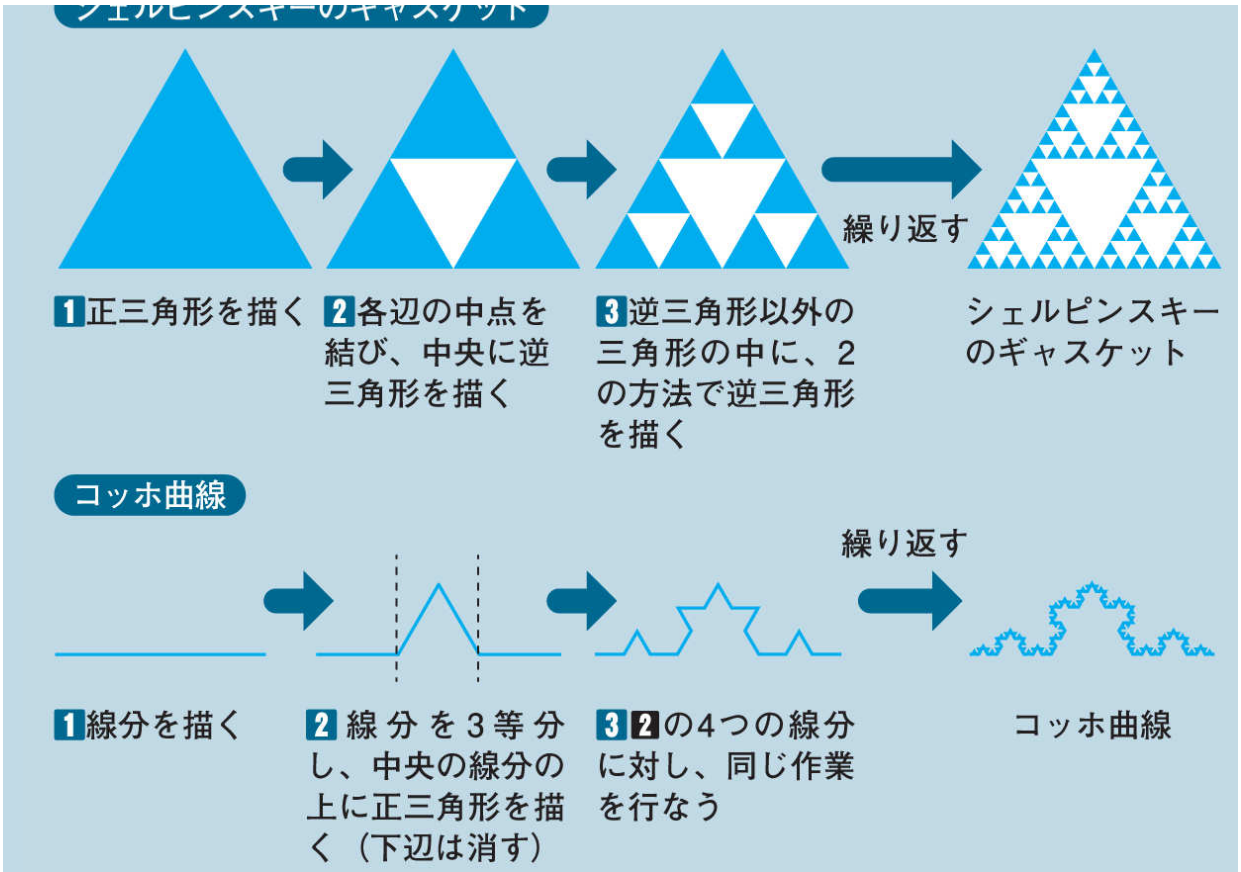
この公式は、区間（線分）を3分するための2点を与えるものでした。その2点を新たに追加するノードの左端と右端の座標に使ったわけです。そして、実数の稠密性から、理論的にはこの公式を繰り返し適用することによって、いくらでも小さな入れ子集合を作っていくことが可能になります（図A）。



図A●入れ子集合は無限に作ることができる

こういう自己相似的な図形を**フラクタル**と呼びます。フランスの数学者マンデルブロ（1924-2010）が1977年に論文で定義した概念で、海岸線や樹木の形など自然界にも多く存在することが知られています。入れ子集合はフラクタルな図形の一種です。他にも、シェルピンスキーのギャスケットやコッホ曲線など、様々な種類のフラクタル図形が知られています（図B）。

シェルピンスキーのギャスケット



図B●様々なフラクタクル図形

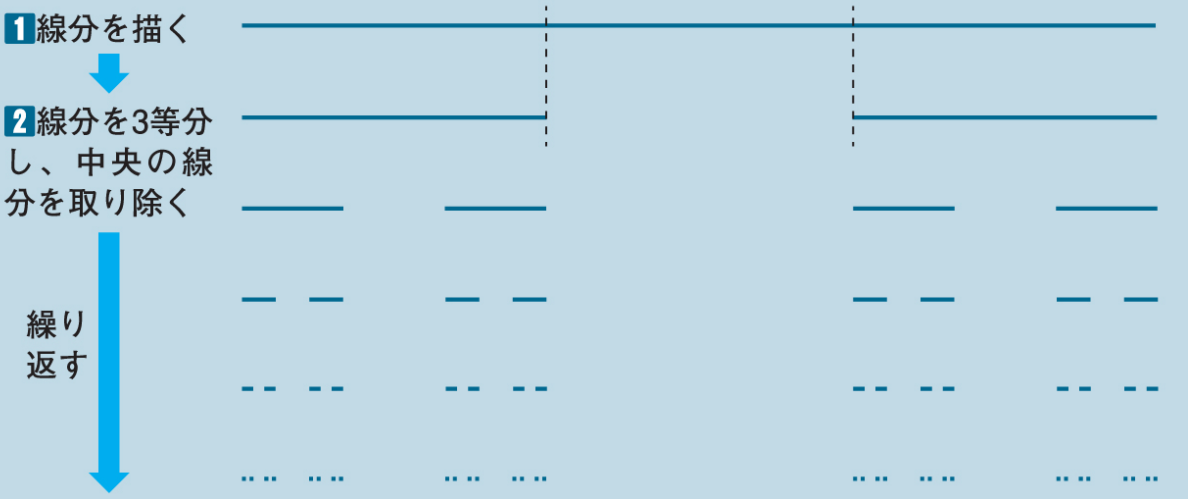
フラクタルは、入れ子集合モデルや入れ子区間モデルがそうであるように、定義上、大きさを無限に小さくしていくことが可能です。

マンデルブロは、このフラクタルを株価のチャートの中にも発見したといいます。自然界だけでなく、人間のランダムな行動の集積の中にまでフラクタルが存在するとなると、神秘的な気持ちになると同時に、少し頭がクラクラしてきます。

特に、入れ子集合モデル（および入れ子区間モデル）と関係の深いフラクタルを挙げるならば、それは**カントール集合**です。ドイツの数学者カントール（1845-1918）が考案したもので、線分を3等分し、得られた真ん中の線分

を取り除く、という操作を繰り返すことで作られる線分の集合です（図C）。お

を取り除く、という操作を繰り返すことで作られる線分の集合です（図C）。お気づきのようにこの操作は、入れ子区間モデルで新たな円を追加するときの操作と原理的に同じです。入れ子区間モデルのアイデアの源泉は、このカントール集合にあります。



図C●フラクタクル図形「カントール集合」

カントールは集合論の創始者として知られる大数学者ですが、その集合論はリレーショナルデータベースの基礎にもなっている理論です。彼は20世紀前半に亡くなっているので、リレーショナルデータベースの登場を見ることはありませんでしたが、その彼が、モデリング技法においても再び関係しているというのも、また面白いところです。

演習問題

演習 9-1 木構造を扱うモデルの正規形

本章では、木構造を扱う四つのモデルが登場しました。

- **隣接リストモデル**

  組織図{社員, 上司}
- **入れ子集合モデル**

  組織図{社員, 左端, 右端}
- **入れ子区間モデル**

  組織図{社員, 左端, 右端}
- **経路列挙モデル**

  組織図{社員, 経路}

これらの正規形の次数は、それぞれいくつか答えてください。

演習 9-2 実数のデータ型

入れ子区間モデルの「左端」および「右端」の列を実装するためのデータ型を、以下のDBMSの最新バージョンについて調べなさい。

- **Oracle**
- **SQLServer**
- **DB2**
- **PostgreSQL**
- **MySQL**

付録

演習問題の解答

演習 1-1　DBMSの情報確認

おそらく読者のみなさんが利用するリレーショナルデータベースは、9ページの「主なDBMS」で紹介した5つのDBMSだと思います。これ以外にも、最近は組み込みシステムでよく利用されるSQLiteなども普及してきていますが、一般的な業務システムでは使われることが少ないため、ここでは省略します。

❶ バージョン

各DBMSの最新バージョンは、調べるタイミングによって異なるため、解答は割愛します。以下、バージョンを調べるコマンドをDBMSごとに挙げておきます。

■Oracle

```
select * from v$version;
```

■SQL Server

```
SELECT SERVERPROPERTY('productversion'), SERVERPROPERTY
('productlevel'),
SERVERPROPERTY ('edition');
```

※紙面の都合上で折り返しています。

■DB2（OS コマンド）

```
db2level
```

■PostgreSQL

```
select version();
```

■MySQL

```
select version();
```

❷ エディション

多くのDBMSは、利用形態やシステム規模に合わせて複数のエディションを用意しています。基本的な区分としては、小中規模向けの「Standard Edition」と大規模向けの「Enterprise Edition」の二つがあると考えてもらえば良いのですが、さらに細かい分類を設けているDBMSもあります。

エディションの違いによって、利用できる機能や構成可能なサーバー台数（またはCPU数や利用人数）などの条件が異なってきます。当然、ライセンス費用も違ってくるので、システムに導入する際にはエディションの差異は重要な検討ポイントになります。

❸ マニュアル

何であれ、マニュアルというものを読むのが好きな人はいないと思います。百歩譲って、いたとしてもよほどの少数派でしょう。家電が故障したときに分厚いマニュアルを1ペ

ージ1ページたぐっては、欲しい情報がどこにあるのか見つけられなくて怒りを爆発させた経験は、きっと誰にでもあることでしょう。

しかし、家電ごときで音をあげていたらIT業界では生きていけません。DBMSのマニュアルは、家電など及びもつかないほどの情報量と複雑さです。しかも、最近はどのDBMSにおいても、マニュアルは非常によく整備されており、われわれ開発者が疑問に思うことの多くが、実はすでにマニュアルに書かれていたりします。したがって、みなさんもエンジニアの端くれになろうというのなら、マニュアルを熟読することを面倒に思ってはいけません。

勘どころ 9

DBMSのマニュアルは熟読せよ。

かつ、DBMSのマニュアルはいずれもWeb上に無償公開されているものばかりです。今すぐ自分の使っているDBMSのマニュアルのURLをブックマークしてください。新バージョンが登場するたびにマニュアルも更新されるので、自分の使っているバージョン（前問でちゃんと確認しましたね？）にあったマニュアルの所在を把握しておきましょう。

- **Oracle**
  ⇒Oracle Database Documentation Library 11g リリース2 (11.2)
  http://docs.oracle.com/cd/E16338_01/index.htm
- **SQL Server**
  ⇒Microsoft MSDNライブラリ SQL Server
  http://msdn.microsoft.com/ja-jp/library/bb418431%28v=sql.10%29.aspx
- **DB2**
  ⇒IBM DeveloperWorks DB2オンライン・マニュアル

http://www.ibm.com/developerworks/jp/data/library/ds/manual/db2online/

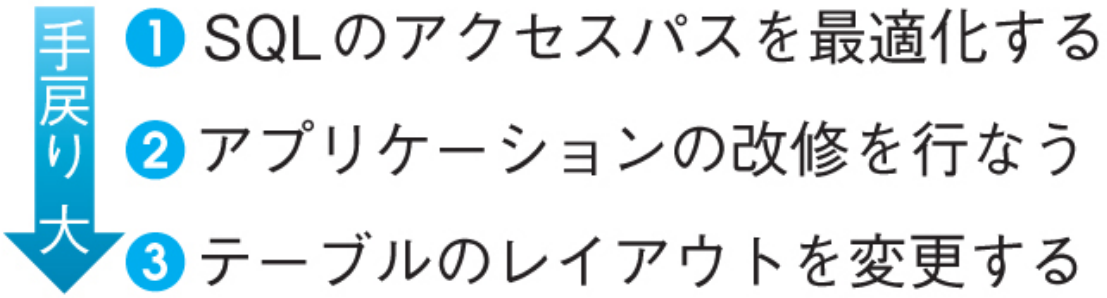
- **PostgreSQL**
  ⇒日本PostgreSQLユーザ会 PostgreSQL日本語ドキュメント
  http://www.postgresql.jp/document/
- **MySQL**
  ⇒MySQL Documentation：MySQL Reference Manuals
  http://dev.mysql.com/doc/

演習 1-2　アプリケーション改修のタイプとコスト

問題1 の対処方法

近年、データベースに蓄積されるデータ量は増加の一途をたどっており、それを原因とする性能問題も頻発しています。バッチ処理のように大量データを一括更新するSQLで遅延が発生することも珍しくありません。

こういう場合の対処には、大きく以下のような選択肢が考えられます。並んでいる順に手戻りが大きく（＝改修コストが大きくて大変に）なっていきます。



❶ SQLのアクセスパスを最適化する

SQL文はDBMS内部で手続き型のコードに変換されて実行されます。この変換は自動的に行なわれるのですが、その際、必ずしもパフォーマンス的に最適な変換が行なわれるとは限りません（このメカニズムの詳細は第6章参照）。その結果、SQLの性能問題が発生することになります。

このようなケースでは、DBMSに最適なアクセスパスを選択させるよう、アクセスプラン（実行計画）を変えさせることが解となります。具体的な方法としては、

- 適切な列に対するインデックスを付与する（第6章）
- 実行計画をユーザーが直接指定する機能を使う（実装依存のためDBMSによって利用制限がある）

などがあります。

この方法の良いところは、システムの設計に対する変更が少ないことです。アプリケーションもそのままで、テーブルのレイアウトにも変更が必要ありません。その点で、非常に手戻りを小さく抑えることの可能な手段です。

❷ アプリケーションの改修を行なう

アクセスパスの最適化だけでは解決しなかった場合、次の選択肢はアプリケーションの実装を変更する方法を考えます。すなわち、もっと効率的なアクセスが可能なようにプログラムを作り変えるということです。ここには、「非効率なコードを排除する」という単純なものから、「取得するデータ量を減らす」というような要件調整まで必要な方法まで含まれます。

これは、システムの実装と設計に変更が必要な改修方法ですから、かなり手戻りは大きくなります。実際にテスト工程でこの判断をしなければならないとしたら、相当大きな決断になるでしょう。

❸ テーブルのレイアウトを変更する

アプリケーションの改修では解決しなかった場合、三つ目の選択肢は、さらに大きな犠牲を伴う方法です。すなわち、テーブルの物理定義の変更です。これは、具体的には非正規化（第5章）、データマート（第7章）といった手段を含みます。

❷ で言及したアプリケーションの改修という手段は、実はテーブル設計の変更なくして実施できる場合はまれです。DOAの「プログラムがデータを決めるのではない、データがプログラムを決めるのだ」という基本原則が示すとおり、アプリケーションの実装を左右するのはデータのフォーマットだからです。

実際のシステム開発において、このシナリオに踏み切ることは、実に「最後の手段」です。テーブルのレイアウトを変更するということは、改善対象のアプリケーション以外の多くのアプリケーションにも影響が波及するため、下手をすると「ほとんど全部作り直し」になることすらあります。データマートを追加するぐらいならば、まだ改修コストは軽いのですが、非正規化のインパクトは甚大です。

ウォーターフォールモデルにおいて、いかに最初の要件定義と設計が重要であるか、このようにちょっとシミュレーションしてみただけでも理解いただけるのではないかと思います。

問題2 の対処方法

問題1 のような性能要件（非機能要件）が問題となるケースでは、やり方次第によっては改修コストを低く抑える解決策も選択肢としてはありえました。一方、問題2 のように、システムの機能要件上の問題を解決する手段は、基本的にアプリケーションそのものの改修に踏み込まざるをえません。それだけにコストも高くなりがちです。

ここでの問題は帳票に出力される情報が足りないということでした。これを解決するには、帳票出力のアプリケーションのコードを修正する必要があることは明白ですが、問題はそれだけで済むか、ということです。

単純に、テーブルに存在するデータ（列）を出力項目に含め忘れていた、というだけなら、その列も出力用のSQL文に埋め込めば良いため、話は簡単です。困るのは、そもそも必要なデータがテーブルに存在していなかった場合です。このケースでは、結局のところテーブルレイアウトの変更が必要になります。

この場合でも、なるべく既存のテーブル群を変更することなく、

必要な情報を付加したデータマートを追加する

というアドホックな修正によって、改修コストを下げることは可能かもしれません。実際のシステム開発でも、こういう用途で後から追加されたデータマートが作られることがよくあります。著者は、百以上のデータマートが作られたシステムを見たこともあるぐらいです。ER図を見せてもらっても「人間の理解力の限界を超えている」と感じたものでした（そもそも、こういう乱発されたマートはER図にきちんと反映されていないこともしばしばで、そうなるともう、地図なしで極地探検に出かけるようなものです）。

演習2-1　データベースサーバーのクラスタリング構成

データベースサーバー（以下、DBサーバー）の物理構成としては、まずごくシンプルに、**スタンドアロン**という形式があります。ポツンと1台だけサーバーを用意する方式で、最も単純かつ安上がりですが、サーバーに障害が起きたら即座にシステム停止、かつ性能もほとんどスケーラビリティがないため、よほどの小規模システムでないと採用されません。

そこで、信頼性と性能を向上させるために、DBサーバーを複数台で構成する**クラスタリング**が採用されるわけですが［※1］、これには多くの種類があります。

Active-Standby方式

まず、信頼性向上だけを念頭においた方式として、**Active-Standby**方式があります。通常、動いているのは現用系（Active）だけで、もう一方の待機系（Standby）はまったく処理をしません。ただ、現用系に障害が発生した場合には待機系に切り替わり、少ない停止時間でシステムが復旧できるという利点があります。このときの復旧の速度が速い方式として**ホットスタンバイ**方式、復旧の速

度が遅い方式として**コールドスタンバイ**方式があります。「そんなの復旧時間が短いほうがいいに決まってるんだから、コールドスタンバイなんて使うの？」と思うかもしれませんが、コールドスタンバイは実際に利用されることもあります。というのも、ここにはもう一つ「コスト」という観点があって、コールドスタンバイ方式のほうがサーバーの光熱費も少なくなりますし、ライセンス料が安くなるDBMS製品もあるからです。

Active-Active方式

いずれにせよ、Active-Standby方式というのは、基本的にサーバー2台で構成し、しかも動いているのはそのうち1台だけなので、性能はスタンドアロンと変わりません。これにプラスして性能の向上も目指した方式が、**Active-Active**方式です。名前の示すとおり、この構成では「待機系」というサーバーは存在せず、すべてのサーバーが「現用系」です。したがって、処理を各サーバーに分散することができ、かつ台数も（理論的には）特に限定せず増やしていくことができるので、信頼性のみならず性能的にも有利な方式と言えます。

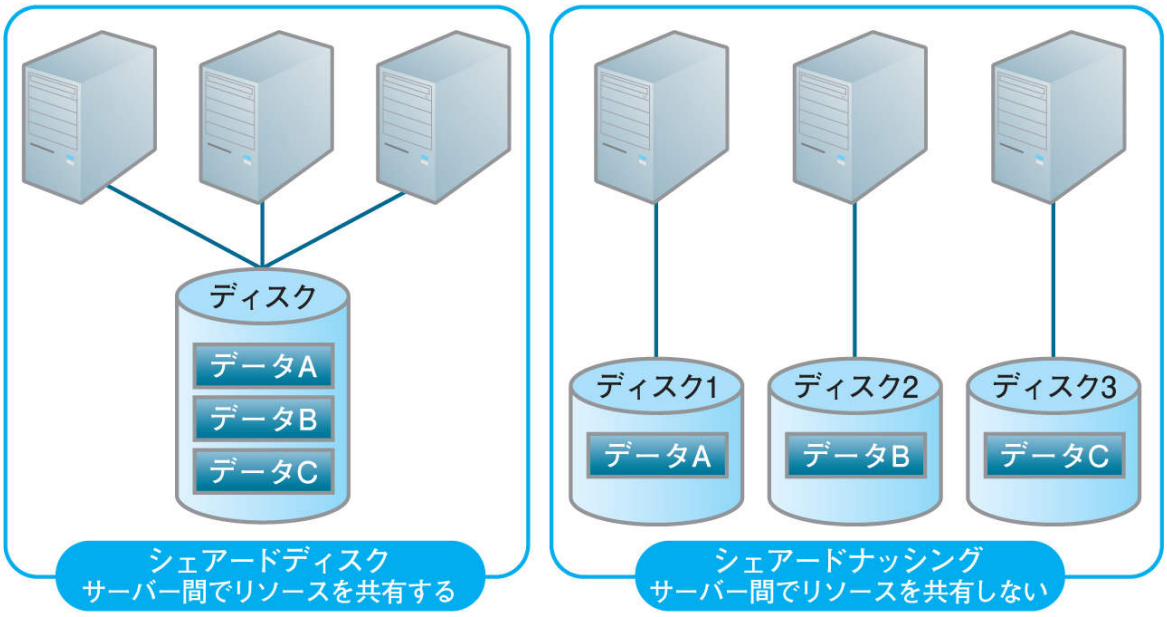
Active-Active方式にも、詳しく見ると二種類の方式があります。

まず一つ目が**シェアードディスク**方式です（図A左側）。これは、ディスクだけをサーバー間で共有して、CPU、メモリ、ネットワークといった資源は分離する構成です。この方式の利点は、どのサーバーからも共通のデータを参照／更新することができるため、どのサーバーに障害が発生しても残りのサーバーで処理を継続することができる、という柔軟性にあります。一方で、共有資源となるディスクがボトルネックポイントになりやすく、それが原因でサーバー台数を増やしても性能向上が頭打ちになってしまう、という欠点があります。

また、このシェアードディスク方式をサポートしているDBMSは、本書の執筆時点（2012年2月）ではOracle（Oracle RAC）とDB2（pureScale）のみです。

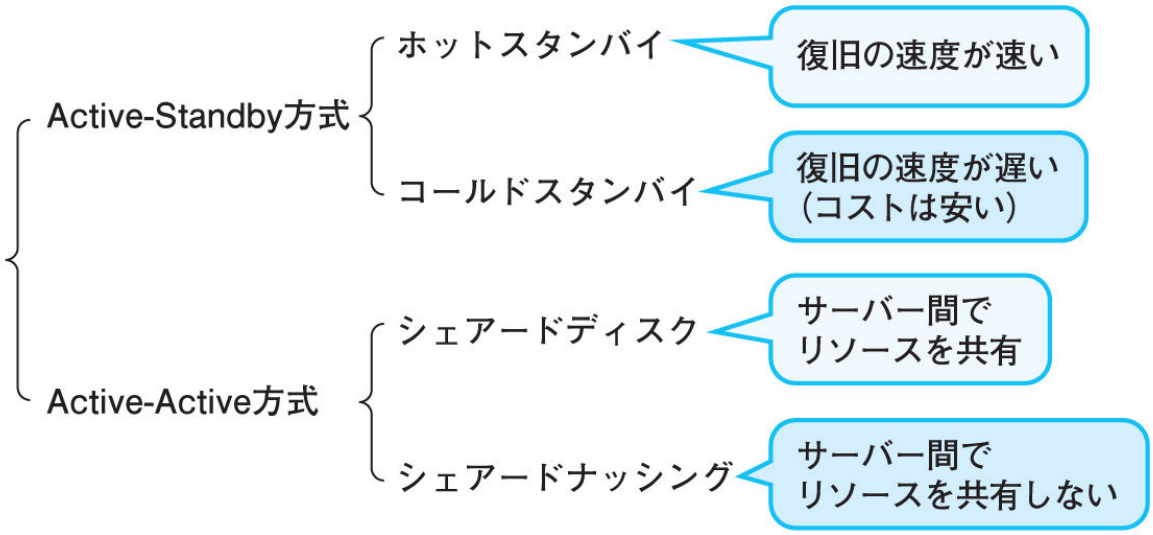
「え、二つだけなの？」と思うでしょう。このActive-Active方式は、Web ／アプリケーションサーバーのクラスタリングとしては非常にポピュラーなのですが、DBサーバーでは難しいのです（この「難しい」というのは、ベンダーとシステム開発者の両方にとってです）。それは、基本的にWeb ／アプリケーションサーバーが次に見るシェアードナッシング方式で十分要件を満たせるのに対し、DBサーバーはどうしてもシェアードディスク方式でないと要件を満たせないことが多いからです。

Active-Active方式のもう一つの選択肢は、**シェアードナッシング**方式です（図A右側）。これは、ディスクを含むあらゆるリソースをサーバー間で**共有しない**方式です。構造的にボトルネックポイントを発生させないため、サーバー台数を並べることで性能がどんどん上がっていく、という利点があります。一方で、データの含まれるディスクを共有しないわけなので、各サーバーのアクセスできるデータは限られます（これを「局所化」と呼びます）。そのため、特定のサーバーがダウンすると、残りのサーバーが稼動していても、一部のデータにアクセスできない状態が発生します。たとえば、都道府県別にデータを分割したシェアードナッシング方式では、「北海道」のサーバーがダウンすると、残りの46のサーバーが稼動していても、北海道のデータへはアクセスできなくなります［※2］。



図A●シェアードディスクとシェアードナッシング

まとめると、データベースサーバーのクラスタリング構成は、図Bのように分類できます。



図B●DB サーバーのクラスタリング構成の分類

演習2-2　ハードウェアリソースの情報取得

- Windows　　：パフォーマンスモニタ
- Linux/Unix系：sar、vmstat、iostat、netstat

商用Unixの中には、HP-UXのglanceのように、独自のリソース監視ツールを用意しているものもあります。こうした独自機能については、ベンダーから提供される製品マニュアルを参照してください。

演習2-3　サーバー CPUの机上サイジング

まず、現行システムにおいて（それほどひどくはないにせよ）すでに性能問題が発生している、ということは、現行のハードウェアリソースが不足している可能性を示唆しています。

したがって、まずは現行システムのピーク時間帯におけるアクセスログとハードウェアリソースのログを確認する必要があります。

1.アクセスログを確認する

アクセスログからは、ピーク時間帯のレスポンスタイムとスループットを調べて、性能要件で定義されている目標を達成しているかどうかチェックします。たとえば、性能要件が以下のように決められていたとします。

- レスポンスタイム：3秒（90パーセンタイル［※3］）
- スループット　　：100TPS

一方、ピーク時間帯の性能を調査したところ、以下のとおりでした。

- レスポンスタイム：5秒（90パーセンタイル）
- スループット　　：50TPS

これはレスポンスタイム、スループットともに目標未達ということです。それはそれで由々しき事態なのですが、顧客は文句を言いながらも使ってくれているようなので、ここはこれ以上騒ぎ立てるのは得策ではありません。みなさんの仕事は、あくまで新システムのサーバーサイジングなのです。

2.ハードウェアリソースのログを確認する

さて次に、現行システムのハードウェアリソースのログを調査して、ピーク時間帯にリソースのキャパシティに限界が来ていないかを確認します。リソースログの取得方法は、演習2-2で見たとおりです。ここで確認する項目は、以下の四つです。

- CPU使用率
- メモリ

- ディスク
- ネットワーク

サーバーの資源は、大きくこの四つから構成されています。このうちのどれか一つでも頭打ちになれば、そこがボトルネックとなって、他の資源がどれだけ余っていても有効活用することができなくなります。

さて、今CPUがピーク時間帯に定常的に90％程度と高止まりすることがわかったとします。他のリソースには余裕があり、CPUがボトルネックとなっています。新しいサーバーでCPUを増強することに意味がありそうです。

3.新旧サーバーの性能比を算出する

次に行なうのは、現行のサーバーと新サーバーのCPUの性能比を算出することです。新サーバーではピーク時に50％までにCPU使用率を抑えるとすれば、大ざっぱに言って、

80 / 50 ＝ 1.6

となり、現行の1.6倍の性能を持つCPUが必要となります。安全率を見て、2倍の性能を持つCPUとしましょう。ではCPU同士の間で「性能がx倍」というのはどんな指標によって判断できるのでしょうか？ 何らかの共通的な尺度が必要になります。

ここで利用するのが、ベンチマーク指標です。各ハードウェアベンダーもいろいろな指標を公表していますが、異なるベンダーのハードウェアを比較するには、本文でも触れた非営利業界団体のSPEC（Standard Performance Evaluation

Corporation）の指標を使うのが一般的です。SPECの行なったベンチマーク結果は、すべてWeb上で無償公開されています［※4］。

SPECは様々な指標を提供していますが、特にCPUを評価する場合は、整数演算能力を示す**SPECint**を用います。SPECintには、1コア当たりの計算能力を示すSPECintと、マルチコアの計算能力を示すSPECint_rateがあります。最近のサーバーはすべてマルチコア対応しており、またDBMSもマルチコア対応したミドルウェアであるため、普通はSPECint_rateを使います。

このSPECint_rateについて、現行システムのサーバーが50であれば、少なくとも100のサーバーを新サーバーとして選らなければならない、ということがわかるわけです［※5］。

クラスタは「かたまり」とか「（ブドウのような植物の）房」という意味です。

その場合に、たとえば「新潟」のデータベースサーバーが「北海道」のサーバーの代わりをする、というようにシステムを作ることは可能ですが、そうすると「新潟」のサーバーには「北海道」と「新潟」の処理が集中することになりサイジングのハードルが上がるなど、なかなか難しいところです。

「nパーセンタイル」とは、レスポンスタイムの遅い順に並べて、上位n％を除外したときに一番遅いレスポンスタイム、という意味です。つまり、「90パーセンタイルで3秒」という要件を満たすシステムにおいては、「全レスポンスのうち、9割が3秒以内に収まる」という意味になります。

http://www.spec.org/なお、こういう場合に使うベンチマーク指標を公開している機関としては、SPECのほかに同じく業界団体TPC（http://www.tpc.org/）などもあります。

実際にはさらに、演習2-1で見たクラスタリングの影響を考慮する必要があります。たとえば、現行システムが3台のクラスタ構成だったとすれば、現行システムのSPECint_rateの計算能力は50×3=150と見積もれます（本当はオーバーヘッドが入るので、150よりも小さくなりますが）。すると、新サーバーは、もし1台でまかなうならば、150の性能を持つサーバーが必要ですし、2台で構成するなら、1台当たり75の性能が必要、ということになります。いずれにせよ、この計算はかなり大ざっぱですし、机上でやる限りこの精度が限界なので、安全率を積んでおくことは必須です。

第3章 解答（演習問題➡122ページ）

演習3-1　正規形の次数

第1正規形。理由は演習3-2の解答を参照してください。

演習3-2　関数従属性

まず、以下の部分関数従属が存在します。

{支社コード} → {支社名}
{支社コード, 支店コード} → {支店名}
{商品コード} → {商品名, 商品分類コード}

次に、以下の推移的関数従属が存在します。

{商品コード} → {商品分類コード} → {分類名}

部分関数従属が存在することから、この「支社支店商品」テーブルは、第2正規形を満たしていません。しかし、すべてのセルがスカラ値によって構成されていることから、第1正規形は満たしています。

演習問題3-3　正規化

三つの部分関数従属を解消するために、次のような三つのテーブルを作ることができます。

支社

支社コード	支社名
001	東京
002	大阪

商品

商品コード	商品名	商品分類コード
001	石鹸	C1
002	タオル	C1
003	ハブラシ	C1

004	コップ	C1
005	箸	C2
006	スプーン	C2
007	雑誌	C3
008	爪切り	C4

支店

支社コード	支店コード	支店名
001	01	渋谷
001	02	八重洲
002	01	堺
002	02	豊中

次に、推移的関数従属を解消するために、以下のようなテーブルを作成します。

商品分類

商品分類コード	分類名
C1	水洗用品

C2	食器
C3	書籍
C4	日用雑貨

そうすると最終的に、「支店商品」テーブルに残るのは、次の三つの列ということになります。なおここでは、問題文で最初に考えたテーブル名「支社支店商品」から「支店商品」に名前を変更しました。このほうが、「支店」と「商品」間の関連を表現している、ということがわかりやすくなります。データの粒度に合わせたテーブル名を付けるということも、基本的ですが重要なポイントです。

支店商品

支社コード	支店コード	商品コード
001	01	001
001	01	002
001	01	003
001	02	002
001	02	003
001	02	004
001	02	005

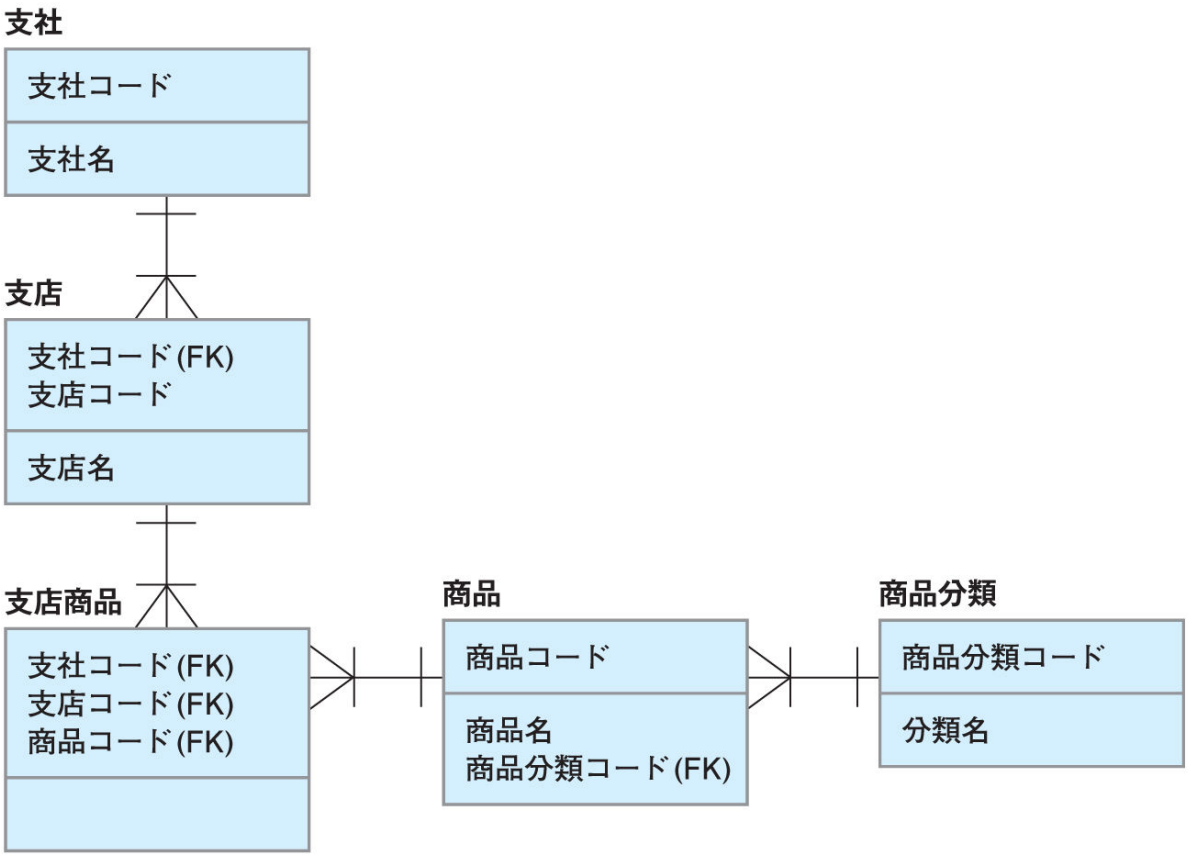
001	02	006
002	01	001
002	01	002
002	02	007
002	02	008

この五つのテーブル群が解答です。

第4章 解答（演習問題➡137・138ページ）

演習4-1　ER図

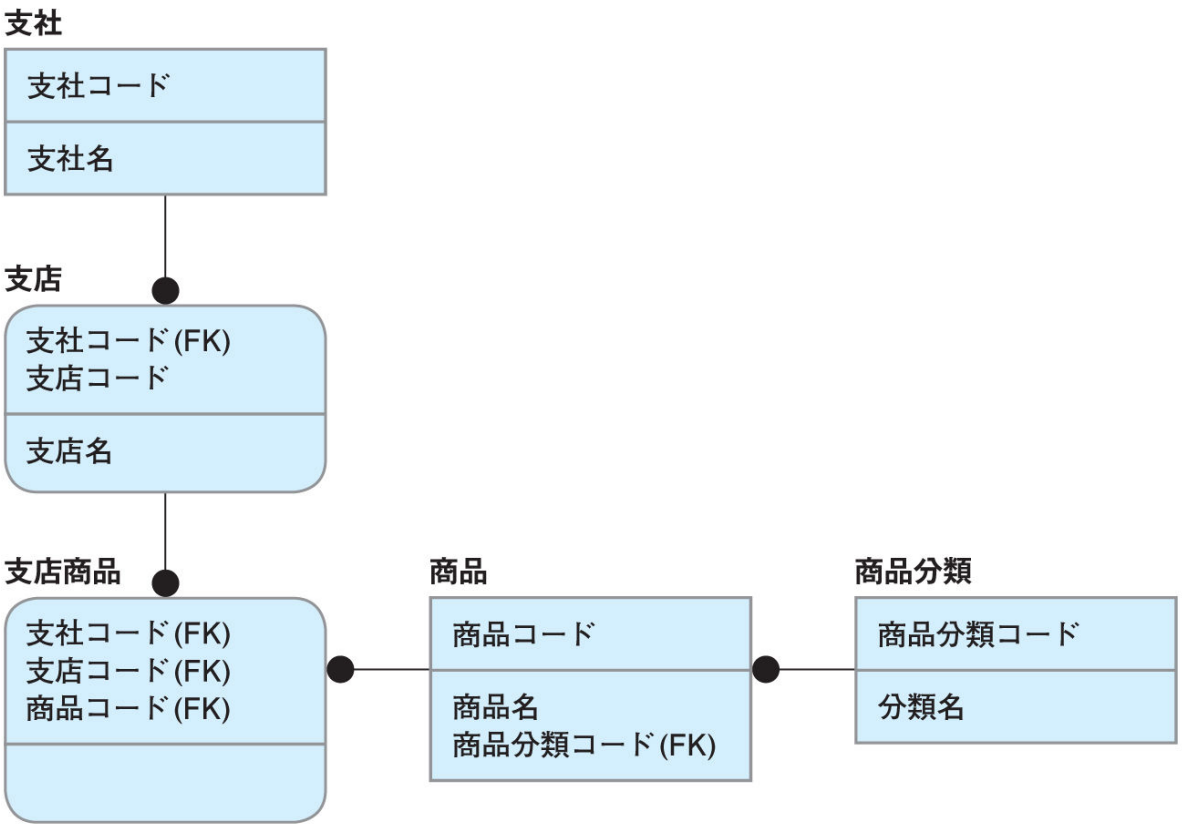
IE記法



「支社」と「支店」の間には、「支社は複数の支店を統括する」という1対多の関係が成立します。ここで「支店を一つも持たない支社はあるか」という問題が生じますが、少なくともテーブルのデータからは、そのような支社は確認できないため、ここでは「支社は少なくとも一つの支店を管理する」という解釈を採用しました。あるいは、「支店を一つも管理しない支社もありうる」と解釈して、「支点」側の記号を鳥足ではなく「○」にした人もいるかもしれませんが、それも間違いではありません。

同じ問題は、「商品」と「商品分類」の間にも当てはまります。ここでは、「何の商品も属さない商品分類はない」という解釈を採用しましたが、そのような空の分類もありうると考えて、「商品」側の鳥足を「○」にすることも、間違いではありません。いずれにせよ、その判断は業務ロジックに依存します。

IDEF1X



IDEF1Xにおける考え方も、IE記法と大きな相違はありません。唯一、特有の考慮が必要な箇所は、「支店」および「支店商品」を従属エンティティとして表現することです。「支店」エンティティには、支社が決まらない限り支店を登録できませんし、「支店商品」のレコードは、「支店」および「商品」の二つのエンティティのレコードが登録されない限り、やはり登録できません。一方、「商品」エンティティは、商品分類コードを主キーの一部として持たないため、商品分類が決まらなくても商品を登録することは可能です。

演習 4-2　関連エンティティ

以下が関連エンティティです。

支店商品

「支店」エンティティと「商品」エンティティは、多対多の関連を持っています。一つの支店は複数の商品を扱いますし、一つの商品は複数の支店で売られているからです。したがって、この二つのエンティティを直接ER図において関連させることはできません。そこで両者の関連エンティティである「支店商品」を導入して、多対多の関連を二つの1対多の関連に分解しています。

演習 4-3　多対多の関連

たとえば以下のようなエンティティが考えられます。

❶ 消費者-商品

消費者は、一人が複数の商品を買うことができます。また、一つの商品は複数の消費者に買われるので、多対多の関連になります。

これを解消するための関連実体は、「購入」または「注文」です。

❷ 学生-部活

学生は、一人が複数の部活をかけもちできます。また、一つの部活には複数の学生が所属します。

これを解消するための関連実体は、「所属」です。

❸ 著者-書籍

著者は、一人が複数の書籍を書くことができます。書籍も、複数の著者（共著）を持つことができます。

これを解消するための関連実体は、「著述」です。

演習5-1　正規化されたテーブルに対するSQL

SQL文1

```
SELECT 商品分類.商品分類コード,
       商品分類.分類名,
       COUNT(*) AS 商品数
  FROM 商品分類 INNER JOIN 商品
    ON 商品分類.商品分類コード = 商品.商品分類コード
 GROUP BY 商品分類コード, 商品分類.分類名;
```

結果

```
商品分類コード   分類名     商品数
----------   ------   ----
C1           水洗用品       4
C2           食器           2
C3           書籍           1
C4           日用雑貨       1
```

SQL、特に複数のテーブルを結合するような複雑なSQLを考えるときのコツは、「FROM句から考える」ことです。これはつまり、「このSQLを書くために必要なテーブ

ルは何だろう」と考えることです。SQLの初心者は、SQL文の先頭、つまりSELECT句から考えがちです。しかしSELECT句というのは、最後に決まる要素であるため、実はFROM句→WHERE句→GROUP BY句……という順序で考えたほうが簡単なのです。

そのやり方に従って考えてみましょう。今回の問題を解くのに必要なテーブルは何でしょうか。まず、商品分類名が必要とされることから、「商品分類」テーブルが必要です。次に、商品の個数を数える必要があることから、「商品」テーブルも必要です。この二者がFROM句で必要になるテーブルです。

その両者を「商品分類コード」によって結合した後に、商品分類ごとのレコード数のカウントを取れば良いのです。GROUP BYのキーは、実質的には「商品分類コード」だけで良いのですが、SQLではGROUP BY句を使った場合、SELECT句に記述できるテーブルの列は、GROUP BY句に含まれている必要があるため、「分類名」列もGROUP BY句に含めています［※6］。

SQL文2

```
SELECT 支社.支社名,
       支店.支店名,
       商品.商品名
  FROM 支社 INNER JOIN 支店
               ON 支社.支社コード = 支店.支社コード
                  INNER JOIN 支店商品
                     ON 支店.支社コード = 支店商品.支社コード
                    AND 支店.支店コード = 支店商品.支店コード
                        INNER JOIN 商品
                           ON 支店商品.商品コード = 商品.商品コード;
```

結果

支社名	支店名	商品名
東京	渋谷	石鹸
東京	渋谷	タオル
東京	渋谷	ハブラシ
東京	八重洲	タオル
東京	八重洲	ハブラシ
東京	八重洲	コップ
東京	八重洲	箸
東京	八重洲	スプーン
大阪	堺	石鹸
大阪	堺	タオル
大阪	豊中	雑誌
大阪	豊中	爪切り

先ほどと同様に、FROM句から考えていきましょう。まず、結果に必要な列として、「支社名」「支店名」「商品名」が指定されています。このことから、それぞれの列を保持している「支社」「支店」「商品」の三つのテーブルが必要なことがわかります。また、それ以外にも、各支店別の商品の取り扱いを知るには、関連エンティティである「支店商品」テーブルが必要です。この四つのテーブルが、今回使うテーブルです。

あとは、それぞれのテーブルを主キーによって結合していけば、「1対多」の結合が完成します。問題文では、特に「支社コード」や「支店コード」などコード列を結果に含めることは要求されていなかったため、名前だけをSELECT句に含めました（コード列を含めても間違いではありません）。

SELECT句を見ると、FROM句で使ったテーブルのうち、「支店商品」テーブルだけは列を結果に含めていないことがわかります。これは、関連エンティティの役割が、結局のところは「支店」と「商品」の関連を示すためだけなので、結合によってその関連が表示されたら役割を終えるからです。

SQL文3

```
SELECT 支店.支社コード,                                          ┐
       支店.支店コード,                                          │
       COUNT(*) AS 最大商品数                                    │
  FROM 支店 INNER JOIN 支店商品                                  │ パート2
    ON 支店.支社コード = 支店商品.支社コード                     │
   AND 支店.支店コード = 支店商品.支店コード                     │
 GROUP BY 支店.支社コード, 支店.支店コード                       ┘
HAVING COUNT(*) >=
        ( SELECT MAX(商品数)                                     ┐
            FROM ( SELECT 支社コード,                            │
                          支店コード,                            │
                          COUNT(*) AS 商品数                     │ パート1
                     FROM 支店 INNER JOIN 支店商品               │
                       ON 支店.支社コード = 支店商品.支社コード  │
                      AND 支店.支店コード = 支店商品.支店コード  │
                    GROUP BY 支店.支社コード, 支店.支店コード) TMP );┘
```

結果

```
支社コード   支店コード   最大商品数
-------   -------   -------
001            02         5
```

要件自体はありふれているのに、解答のSQLのコードが思ったよりも複雑になることに驚いた人もいるかもしれません。

解答のSQLは、大きく二つのパートに分かれています。一つ目が、HAVING COUNT(*)>=よりも後のサブクエリ。このパートでは、取り扱い商品数が最も多い支社の商品数を選択しています。その結果を受けて、HAVING句より前のパートで、その最大商品数を持つ支店を選択しています。

演習5-2　非正規化によるSQLチューニング

SQL文1

解答のSQLの複雑な点は、以下の2点です。

- 結合を行なっている。
- 集約を行なっている。

したがって、方針としてはこの二つを行なわないSQLを可能にするようなテーブル構成の変更です。これを実現するには、次のように「商品分類」テーブルに「商品数」列を追加することです。

■「商品分類」テーブルに「商品数」列を追加

商品分類

商品分類コード	分類名	商品数

C1	水洗用品	4
C2	食器	2
C3	書籍	1
C4	日用雑貨	1

このテーブルを前提にすれば、SQL文は極限まで簡略化されます。

■テーブル構成変更後のSQL文1

```
SELECT *
  FROM 商品分類;
```

このテーブル変形は、本章の5-2節で見たサマリデータをテーブルに保持する方法と同一です。しかも、本文のサンプルで見たときには第3正規形を崩す形でサマリデータを保持していたのに対し、この例では正規形を維持しています。「商品数」列は、{商品分類コード}という「商品分類」テーブルの主キーに従属しているからです。

問題は、この新しい「商品分類」テーブルにおいても、データのリアルタイム性と、更新パフォーマンスという、サマリデータに伴う問題点はやはり考慮しなければならないことです。実際、この「商品数」列はかなりの頻度で更新されることが予想されるので、更新のタイムラグによってデータ鮮度が落ちたり、更新負荷集中による性能劣化のリスクは高いと言えるでしょう。

SQL文2

解答のSQL文では三つの結合を行なっています。この回数を減らすようなテーブル構成の変更を行なうことになります。つまり、あえて冗長データを保持することで結合を減らすのです。

そもそも、解答のSQL文でなぜ結合を行なっているかといえば、「名前」を結果に含めたいからです。支社、支店、商品のいずれも「コード」で良いならば、「支店商品」テーブルにすべて揃っています。ということは、逆を言うと、「支店商品」テーブルにすべての名前を保持してしまえば良いのです。

■「支店商品」テーブルに名前の列を追加

支店商品

支社コード	支店コード	商品コード	支社名	支店名	商品名
001	01	001	東京	渋谷	石鹸
001	01	002	東京	渋谷	タオル
001	01	003	東京	渋谷	ハブラシ
001	02	002	東京	八重洲	タオル
001	02	003	東京	八重洲	ハブラシ
001	02	004	東京	八重洲	コップ
001	02	005	東京	八重洲	箸
001	02	006	東京	八重洲	スプーン

002	01	001	大阪	堺	石鹸
002	01	002	大阪	堺	タオル
002	02	007	大阪	豊中	雑誌
002	02	008	大阪	豊中	爪切り

このテーブルを前提にすれば、SQL文は極限まで簡略化されます。

■テーブル構成変更後のSQL文2

```
SELECT 支社名, 支店名, 商品名
  FROM 支店商品;
```

しかし、このテーブルはすでに第2正規形ですらありません。名前の各列は、主キーの一部に従属しているため、部分関数従属が存在するためです。

{支社コード} → {支社名}
{支店コード} → {支店名}
{商品コード} → {商品名}

そのため、非正規化によるデメリット（詳細は本文参照）をおもいきり被ることになります。

SQL文3

SQL文1 と同様、次のように「支店」テーブルに最初からサマリデータ（商品数）の列を持っておくことが考えられます。

■「支店」テーブルに「商品数」列を追加

支店

支社コード	支店コード	支店名	商品数
001	01	渋谷	3
001	02	八重洲	5
002	01	堺	2
002	02	豊中	2

このテーブルを出発点に考えれば、結合もHAVING句も使う必要なく、以下のような簡単なSQLで答えを求められます。

■テーブル構成変更後の SQL文3

```
SELECT 支社コード,
       支店コード,
       商品数
FROM 支店
WHERE 商品数 = (SELECT MAX(商品数)
FROM 支店);
```

このSQLは見た目が単純であるだけでなく、「支店商品」テーブルとの結合がなくなる分、パフォーマンスも向上します。

このテーブル変形も、やはりサマリデータを保持する方法です。かつ、第3正規形を維持しています。「商品数」列は、{支社コード, 支店コード}という「支店」テーブルの主キーに従属しているからです。

とはいえ、この「支店」テーブルにおいても、データのリアルタイム性と、更新パフォーマンスという、サマリデータに伴う問題点はやはり考慮しなければならないことは、言うまでもありません。

この列は、あってもなくても結果のレコード数を変えることはありません。主キーと1対1に対応するからです。

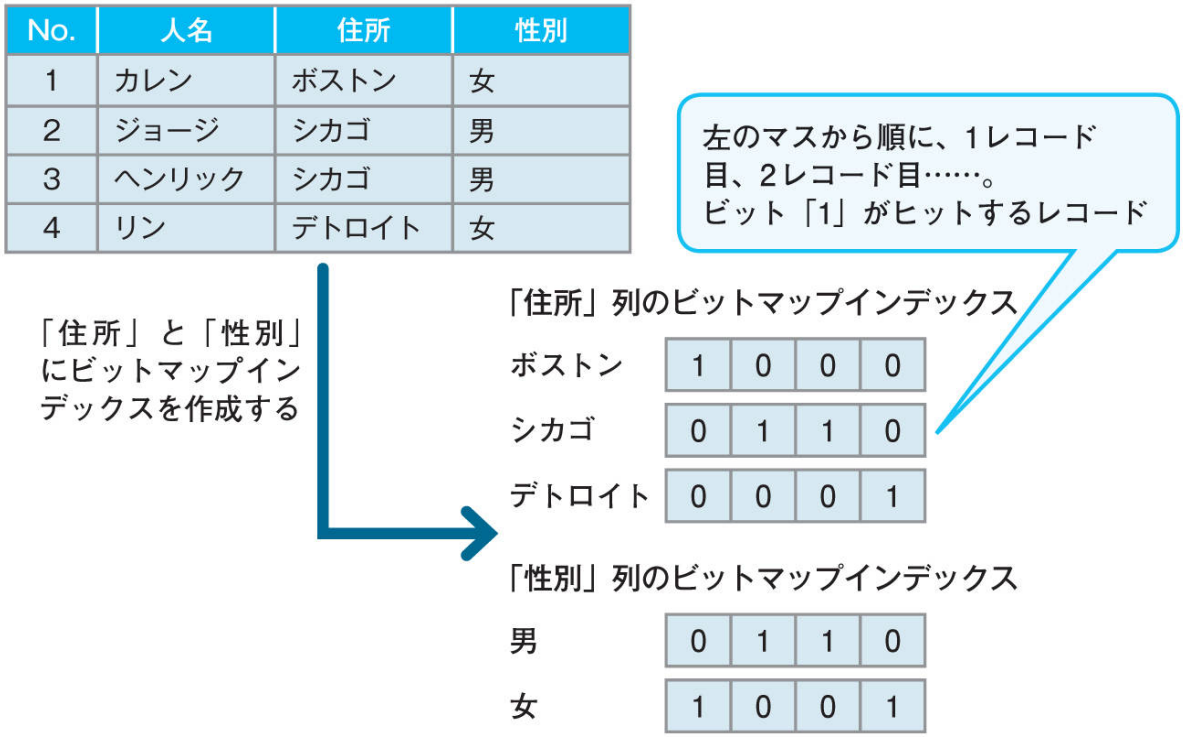
第6章

解答 （演習問題➡186ページ）

演習6-1　ビットマップインデックスとハッシュインデックス

ビットマップインデックス

　ビットマップインデックスは、データの値からビットデータを作成して、それをインデックスとして保持します（図C）。



図C●ビットマップインデックス

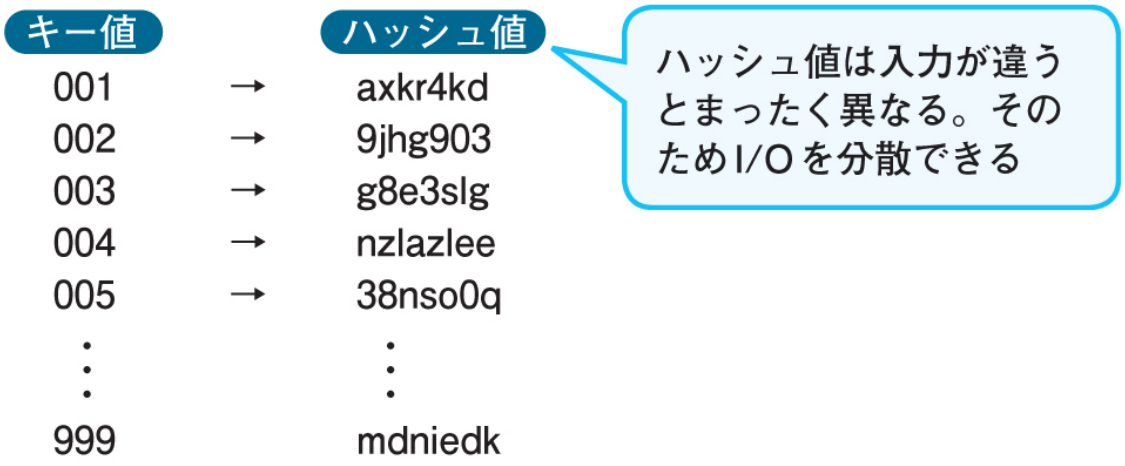
それによって、検索条件に対してもビットデータの演算で検索を行なうことができるため、B-treeが苦手だったカーディナリティの低い列に対しても検索性能が良いという利点を持ちます。また、B-treeインデックスが有効にならない「OR」条件に対しても利用できるという利点があります。この他、ビットデータはとてもサイズが小さいため、インデックスのサイズそのものも小さくなる、という利点もあります。

ビットマップインデックスの欠点は、テーブルのデータが更新されるたびにインデックスのビット値も更新する処理を行なわなければならないため、**更新時の性能が悪い**ことです。そのため、頻繁にデータ更新が行なわれるタイプのシステム（OLTP）で利用するには、性能的なリスクの大きなインデックスと言えます。

ハッシュインデックス

ハッシュインデックスは、ハッシュ関数という関数を使ってデータの値をハッシュ値に変換してインデックスを保持します（図D）。ハッシュ関数とは、入力値が異なれば出力値も異なるような関数のことで、データの分散に利用します。

このため、ハッシュインデックスは、等値検索、つまり「=」を使った検索条件においては非常に高速な検索を実現します。B-treeインデックスでは、目的のリーフを見つけるまでに数回の読み込みが必要なのに対して、ハッシュインデックスはいきなり1回の読み込みで目的地にたどり着けるからです。



図D●ハッシュインデックス

ハッシュインデックスの欠点は、利点の裏返しです。それは**等値検索以外ではこのインデックスが利用できない**ことです。範囲検索には、ハッシュインデックスは使えません。その理由は、ハッシュ関数で変換されたハッシュ値は、元の値の順序関係を維持していないからです。このため、ハッシュ値を使って範囲検索というのは不可能なのです。LIKE述語の部分一致検索で使えないのも同様の理由です。

このように、ビットマップインデックスやハッシュインデックスは、使う条件によってはB-treeインデックスを凌駕する性能を見せるのですが、いかんせんその条件が狭すぎて、使いどころが難しいのです。B-treeインデックスが「オールラウンダー」なのに対して、ビットマップインデックスやハッシュインデックスは自分の得意分野でだけ力を発揮する「スペシャリスト」なのです。

演習6-2　インデックスの再編成

まず、どのDBMSでも共通の最も簡単な方法は、一度インデックスを削除して、もう一度作成することです。この方法は非常に明快ですが、リスクもあります。それは、インデックスを削除した後、作成するコマンドが何らかの理由でエラーになって失敗してしまうと、問題が解消されるまでそのデータベースは「インデックスがない」状態が続いてしまうことです。この間に実行されるSQL文は、もちろんインデックスを利用できず、性能問題を引き起こすでしょう。

この欠点を補うために、DBMSはインデックスを削除しなくても再編成できるための手段を提供しています。以下、DBMSごとに紹介します。

Oracle

```
ALTER INDEX インデックス名 COALESCE;
```

```
ALTER INDEX インデックス名 REBUILD;
```

上記の二つのコマンドによって、すでに存在するインデックスに対して実行することで、インデックスを削除することなく再編成できます。両者の違いは、「COALESCE」がリーフブロックの結合だけを実行するため高速ですが、簡易な再編しか行なわないのに対して、「REBUILD」は本格的な再編を行なうため時間がかかることです。

SQL Server

```
ALTER INDEX インデックス名 REORGANIZE;
```

```
ALTER INDEX インデックス名 REBUILD;
```

SQL Serverにも、Oracleと同様、二つのレベルで再編成コマンドを用意しています。「REORGANIZE」が簡略版、「REBUILD」が本格的なものです。

DB2

```
REORG INDEX インデックス名;
```

DB2では「REORG」というコマンドを用います。コマンドとしては一つですが、オプションが非常に豊富で、細かい指定を行なうことが可能です。

PostgreSQL

```
REINDEX インデックス名;
```

PostgreSQLでは「REINDEX」というコマンドを用います。やはりオプションで細かい制御ができるので、詳細はマニュアルを参照してください。

MySQL

```
OPTIMIZE TABLE テーブル名;
```

MySQLでは、不思議なことにインデックスの再編成を行なうためにテーブルの再編成を行ないます。この「OPTIMIZE」コマンドは、もちろん第一義的にはテーブルの再編成（デフラグ）を行なうためのコマンドなのですが、実行すると、あわせて当該テーブルに作られているインデックスの再編成も行なわれます。それはそれでかまわないのですが、なぜMySQLがインデックス単独の再編成コマンドを持っていないのかは謎です。

以上、簡単にDBMSごとのインデックス再編成の手段を列挙しましたが、実際はオプションを付加することによって機能を細かく制御することができるため、詳細については各DBMSのマニュアルを参照してください。マニュアルは、演習1-1 （297・298ページ）で見たように、いずれもWeb上で無償公開されています。

演習7-1　パーティションの特徴

三つのパーティションの特徴は、以下のとおりです。

❶ レンジパーティション

レンジパーティションは、パーティション機能の中で最も基本的なもので、頻繁に利用します。インデックスで言えばB-treeに相当します。

レンジパーティションは「レンジ（範囲）」という名称のとおり、ある列をキーにして、その値の範囲によってパーティションを分割します。したがって必然的に、順序を持ったデータにしか適用できません。一般的には、年月や時間といった時系列をキーにすることが多いです。

たとえば、本文でも挙げたように「年度」をパーティションキーとすれば、「2008」「2009」「2010」……というように1年単位でパーティションを区切れば、SQL文で特定の1年のデータにアクセスする際にも、残りの年度のデータには一切触れないことで、I/O量を大きく削減することができます。

❷ リストパーティション

リストパーティションは、原理的にはレンジパーティションと同じです。こちらも、特定のキーの値を使ってレコードをパーティションごとに振り分けます。違うのは、リストパーティションが連続的なキーではなく、離散的なキーに利用されることです。たとえば、都道府県や顧客コードなどです。

以上のことからもわかるように、レンジパーティションとリストパーティションを利用するときの基本方針は、以下のとおりです。

方針1 なるべく各パーティションに含まれるレコード数が同じぐらいになるキーを選ぶこと。そうでないと、キーによってI/O量が大きく変わってしまい、性能が安定しない。

方針2 SQL文で選択条件／結合条件に利用されるキーを選ぶこと。せっかくパーティションでデータを区切っても、SQL文で利用されなければ意味がない。

方針3 値があまり変更されないキーを使うこと。パーティションにレコードが格納された後に、キー値が変更されると、パーティション間のレコード移動が発生する。それ自身も性能問題を起こすが、パーティション間のレコード件数の分布（ヒストグラム）も崩れてしまうため、あまり変動要素の大きいキーを使うことは推奨できない。

❸ ハッシュパーティション

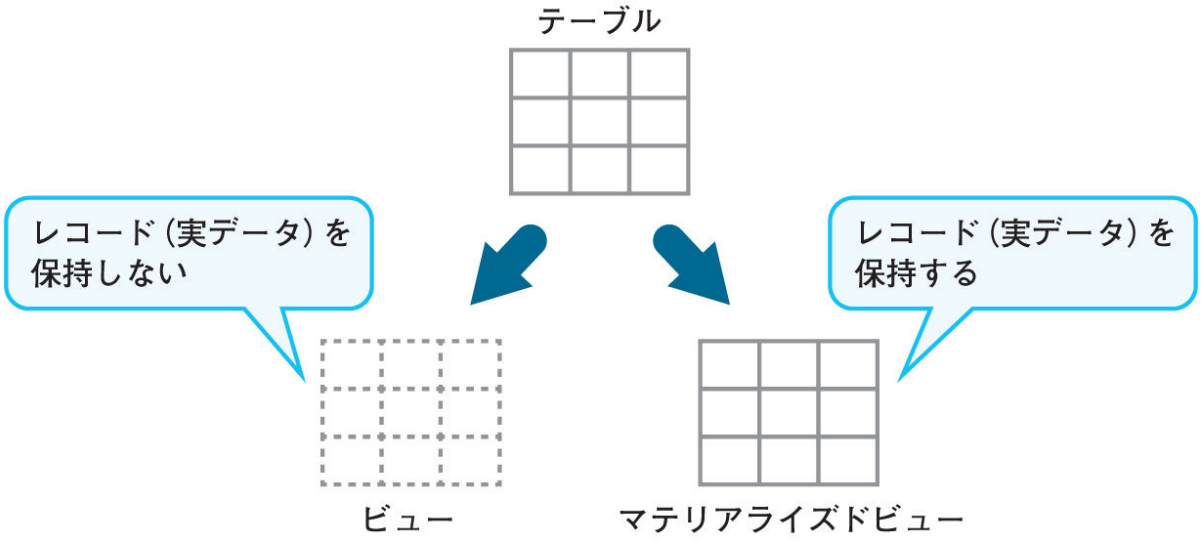
一方、ハッシュパーティションは、前二者のパーティションとは毛色が違います。これは名前のとおり、ハッシュ関数でパーティションキーの値を分散させます。したがって、一般的にはパーティションの数がいくつになるかを事前にユーザー側が指定しな

い使い方をします［※7］。原理的にはハッシュインデックスと同じと考えて良いでしょう。

もう想像がついていると思いますが、ハッシュ関数を使っているという特性上、ハッシュパーティションは、パーティションキーを使った範囲検索では意味を持ちません。あくまで一意検索でしか効力を発揮しないのがハッシュパーティションの弱みです。ハッシュインデックスも、同じ特性を持っていたのでしたね。

演習7-2　マテリアライズドビューの機能

マテリアライズドビュー（Materialized View）は、日本語に訳すと「実体化されたビュー」で、文字通り、実データ（レコード）を保持するビューのことです（図E）。



図E●ビューとマテリアライズドビューの違い

これはもう、ビューというよりほとんどテーブルに近い存在です（実際、Oracle などでは、内部的にビューではなくテーブルのカテゴリに分類されています）。クリス・デイトも、マテリアライズドビューは本当の意味でのビューではない、と次のように述べています。

> 少なくともリレーショナルモデルに関する限り、実体化されないことこそビューの真髄なのである。・・・・・・ したがって、「実体化されたビュー」とは、何とも矛盾した表現である。[※8]

デイトは、マテリアライズドビューは、むしろ「スナップショット」、つまりテーブルのある瞬間における断面を実装した機能である、と述べており、これをビューと呼ぶのは誤解を招くと警告しています。この批判自体は妥当なものだと著者も思いますが、すでに「マテリアライズドビュー」という名前は普及してしまっているので、今から名称が変わることはないでしょう。

マテリアライズドビューがパフォーマンス面で有利な理由は、以下の二つです。

利点1 実データを保持するので、通常のビューと異なり、アクセス時にビュー定義のSELECT文が実行されない。
利点2 主キーをはじめとしてインデックスを作成することができる。

要するに、普通のテーブルと同じ扱いができる、というのが大きな利点なのです。一方、欠点は以下の二つです。

欠点1 リフレッシュの管理が必要である。
欠点2 普通のテーブルと同様にストレージを消費する。

どちらも、普通のビューを使っているときには起こらない問題でした。リフレッシュを怠れば、元のテーブルとデータが食い違ってしまいます。そのため、定期的にリフレッシュを行なって元のテーブルとデータの同期を取る必要があります。これは、通常のテーブルでデータマートを作る場合と同じです。また、実データを保持するため、ストレージの容量も消費することになります。

何事もいいことばかりではない（トレードオフの原則）というのはここでも当てはまる真理です。

これに対し、レンジパーティションとリストパーティションは、逆にユーザーが明示的にパーティション数を決めます。

C.J.Date『データベース実践講義』（オライリー・ジャパン、2006）p.76

解答（演習問題→259ページ）

演習8-1　ビジネスロジックの実装方法の検討

問い1　ビジネスロジックをアプリケーションコードで実装することの是非

結論から言うと、ビジネスロジックをアプリケーションコードで実装すること自体は、認められるべきだ、というのが著者の立場です。

この問題については、エンジニアの間でも意見の振れ幅が大きく、リレーショナル原理主義的な人の間では「ビジネスロジックはテーブルの制約として実装するべきであって、アプリケーションで実装するのは邪道である」という主張もあります。具体的には、あるテーブルの列のデータが「100以下の数値である」という条件を満たさなければならない、というビジネスロジックがあった場合、データの入力時にアプリケーションでチェックするのではなく、テーブルにCHECK制約を付与することで実現するべきだ、という立場です。

この意見には、著者も基本的に賛成します。手続き（コード）を宣言（定義）で置き換えていくというリレーショナルデータベースの基本思想に合致する、合理的な主張だからです。

しかし、実際問題として、テーブルの制約だけでビジネスロジックを実現するのは、現状いろいろと困難がつきまといます。大きくは以下の二つです。

困難1 複雑なビジネスロジックを実装できない

先ほど「ある列は100以下の数値である」というビジネスロジックの例を挙げました。これは非常に単純なので、CHECK制約でも実装できます。では、次のようなロジックはどうでしょう？

「テーブルAの列aは、テーブルBの列bより小さい」（a、bともに数値型の列）

数式にすれば「A.a < B.b」です。これでもまだ十分に単純なロジックだと思うでしょう。しかし、現在のところ、こんな単純な条件ですら、テーブルの制約として実現することが困難です。というのも、CHECK制約は複数のテーブル間の関係を記述することはできないからです。CHECK制約はまた、同一テーブル内であっても、レコードをまたいだ条件を記述することもできません。したがって、ビジネスロジックを実装しようにも、現在のDBMSの機能では不足しているのです。

こういう複雑なビジネスロジックをデータベース側で実装するための機能を「**表明**（Assertion）」といって、標準SQLにも含まれているのですが、まだまだDBMS 側のサポートは遅れています。

「ビジネスロジックのうち、データベース側で実装できるものはデータベース側で、できないものはアプリケーション側で」という折衷案を採用するシステムも見かけますが、これをやると結局、ビジネスロジックの実装箇所が分散されて、どのビジネスロジックがどこで実装されているかわからなくなり、管理が面倒です。こういう中途半端な対応はかえって「ヤブヘビ」を招くので、データベース側かアプリケーション側かの二者択一を迫られます。

困難2 エラーハンドリングが難しい

ビジネスロジックをデータベース側に実装することのもう一つの困難は、エラー時に発生します。先ほどの「100以下」という制約のついている列を考えると、仮に「101」というデータを含むINSERT文が実行されるとどうなるでしょう？ 当然、このINSERT文はエラーになります。問題は、そのときに返却されるエラーコードやメッセージです。

このエラーはDBMSが生成するため、フォーマットや含まれる情報は限られており、かつフォーマットもユーザー側が定義できるものではありません。したがって、アプリケーション側で必要な情報が得られなくなったり、ユーザーに対して「不親切」なエラーメッセージを返すことになりがちです。このような「ユーザーアンフレンドリー」なシステムの戯画的なたとえが、あらゆるエラーに対して「システム管理者に連絡してください」の一点張りで、どこがどう間違っていたのかを一切教えてくれないシステムです。みなさんも、一度はこういう意味不明のエラーメッセージにいら立ちを覚えた経験があるかもしれません。

ここで、次のような疑問を持った人もいるかもしれません。

> 「ビジネスロジックをデータベース側で実装するのが難しいのなら、**全部アプリケーション側でやればいい**。主キー制約も参照性整合性制約も全部、データベース側で実装しなくていいのではないか？」

この意見を全面的に認めるならば、テーブルにはそもそも主キーや外部キーなんて設定しなくてもいい、という結論になります。これは容認できる主張でしょうか？

著者は、この主張は認められないと考えています。その理由は、主キーや参照整合性制約のようなビジネスロジックは、簡単には変化しない持続性の高いルールであり、そのような永続的ルールは永続層であるデータベース層に位置することは、シ

ステムの構成を考えたときに合理的だと考えるからです。また、主キーやNOT NULL制約は、データベース側に実装することで、データベースがSQLを実行する際に、その情報を使って効率的で高速なアクセスパスを選択できるようになる、というパフォーマンス上の利点もあります。

したがって、結論としては、以下のようになります。

結論 主キー、参照整合性制約、NOT NULL制約というリレーショナルデータベースにとって基本的なルールはデータベース側で実装する。それ以外のビジネスロジックはアプリケーション側で実装する。

問い2 ビジネスロジックをデータベースの「トリガー」で実装することの是非

問い1 で、ビジネスロジックの多くはアプリケーションコードで実装する、という結論に達しました。実は、リレーショナルデータベースも、それを実装するための手段を二つ持っています。一つが「ストアドプロシージャ」、そしてもう一つが、この問題の主題である「トリガー」です。

両者について簡単に説明すると、「ストアドプロシージャ」は、「保存されたプロシージャ」という名前のとおり、アプリケーションではなく、DBMS内にコードを保存する手続き型言語です。略して単に「プロシージャ」とも呼びます。Oracleの「PL/SQL」、PostgreSQLの「PL/pgSQL」など、多くのDBMSがこのプロシージャ機能をサポートしています。

一方の「トリガー」は、手続き型であるとか、DBMS内にコードが保存される、という点ではストアドプロシージャに似ていますが、大きな違いがあります。それは、ストアドプロシージャが、ユーザーが（SQLのように）明示的に実行しないと実行されない

のに対して、トリガーは、テーブルへの更新（INSERT、UPDATE、DELETE）を契機として自動的に実行（発射）されることです。トリガーとは「引き金」という意味ですが、テーブルに銃の「引き金」を引っ掛けておいて、更新されたら自動的に銃が発射されるイメージです。

このトリガーは、一見すると非常に便利なような気がします。特に、いろいろな処理で同じようなデータベースの更新を行なわなければならない場合、処理を一か所にまとめる「モジュール化」と同じ効用を期待できます。

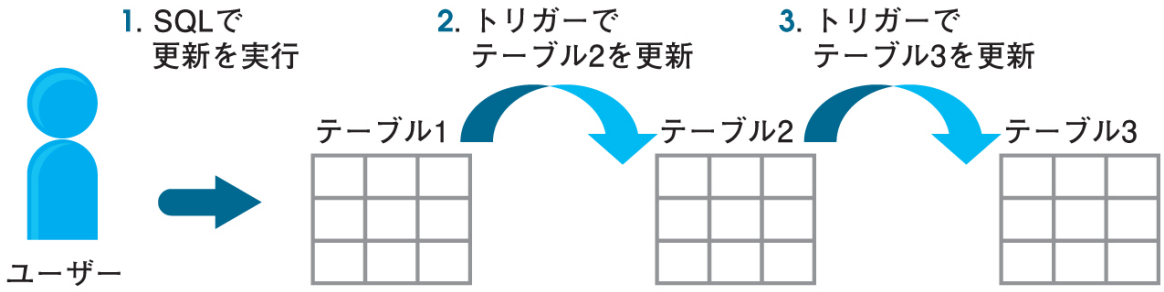
しかし、トリガーは非常に危険なので、使うべきではありません。代わりにストアドプロシージャを使うべきです。この理由は、以下の四つです。

理由1 実装ごとに文法がバラバラで統一性がない。
理由2 オプティマイザによる最適化を受けられない。
理由3 エラーハンドリングが難しい。
理由4 アプリケーション以外の更新でも発射されるため、開発が難しくなる。

理由1 は、程度に差はあれストアドプロシージャについても当てはまるので、「トリガーをプロシージャで代用せよ」と説く理由としては少し弱いものですが、残りの三つの理由は極めて重大です。

トリガーで実行されるコードは、DBMS内部のブラックボックス処理であるため、通常のSQLチューニングができません。また、トリガーによるINSERTやUPDATEが失敗したときのエラーハンドリングも難しくなります。特にその苦労は、連鎖的にトリガーが設定されていると余計大きくなります。一つのトリガーによって行なわれた更新が別のトリガーを引いて、それがさらにまた別のトリガーを引いて……という**連鎖トリガー**は、システムの挙動を非常に複雑なものにしてしまうため、「KISSの原則」（252ページ）にも反します（図F）。

連鎖トリガーによってビジネスロジックを実装する方法を、著者は**連鎖的設計**や**誘爆的設計**と呼んでいますが（いずれも著者の造語です）、これを行なわれると、データの変遷を追うことはもちろん、エラー状況を再現するだけでも一苦労です［※9］。しかも、自分のあずかり知らぬところでテーブルが間違った更新を受ける危険も生じます。反対にプロシージャなら、完全に自分の統制下に置くことが可能です。



図F●連鎖トリガーは危険

実際、トリガーについては、DBMSを作っているベンダーですら否定的です。Oracleのドキュメントには、トリガーについて以下のような注意事項が述べられています［※10］。

- トリガーは、データベースのカスタマイズに役立ちますが、必要な場合のみ使用してください。トリガーを過剰に使用すると相互依存関係が複雑になる可能性があり、大規模なアプリケーションでは管理が困難になります。
- 32KBを超えないようにトリガーのサイズを制限します。トリガーが多くのコードの行を必要とする場合、トリガーから起動されるストアド・プロシージャへのビ

ジネス・ロジックの移行を検討してください。

「そこまで否定的なことを言うぐらいなら、なぜトリガーなんて機能を用意したのさ？」という疑問が浮かぶかもしれませんが、これは標準SQLで定められている機能なので、DBMSとしては一応サポートせざるをえないのです。

トリガーは、確かに使いどころを間違わなければ強力な武器です。しかし殺傷能力が高いので、開発においては厳密な銃規制をして臨むことを推奨します。

勘どころ 61

銃規制で安全な開発を実現しよう。

演習8-2　一時テーブル

一時テーブルとは、その名のとおり「一時的なデータ」を保持するためのテーブルです。「テンポラリテーブル」や「一時表」とも呼びます。標準SQLに準拠した機能であるため、DBMSによらず利用することができます。

このタイプのテーブルは、決められたスパンにおいてしかデータを保持せず、そのスパンが終了するとデータは（ユーザーがDELETE文を実行しなくても）自動的に削除されます。そのスパンは、トランザクションであったりセッションであったり、複数から選べます。必要なときだけデータを保持して、使い終われば自動的に空になる、名前のとおり「使い捨ての」テーブルというわけです。ストレージ容量の節約になることもあって、比較的頻繁に使われている機能です。

この一時テーブルは、「データマート」（7-5節参照）を作るために使われることもしばしばです。ビューと違って実データを保持するため、SELECT文のコストが上昇せず、かといってマテリアライズドビューのように永続的にデータを保持するわけではないので、ストレージの節約になるなど、利点は大きいように思えます。

しかし、この一時テーブルにも欠点はあります。しかも、かなり重大な欠点であるにもかかわらず、少し気付きにくいため、一時テーブルのせいで苦しむことになるシステムも少なくありません。

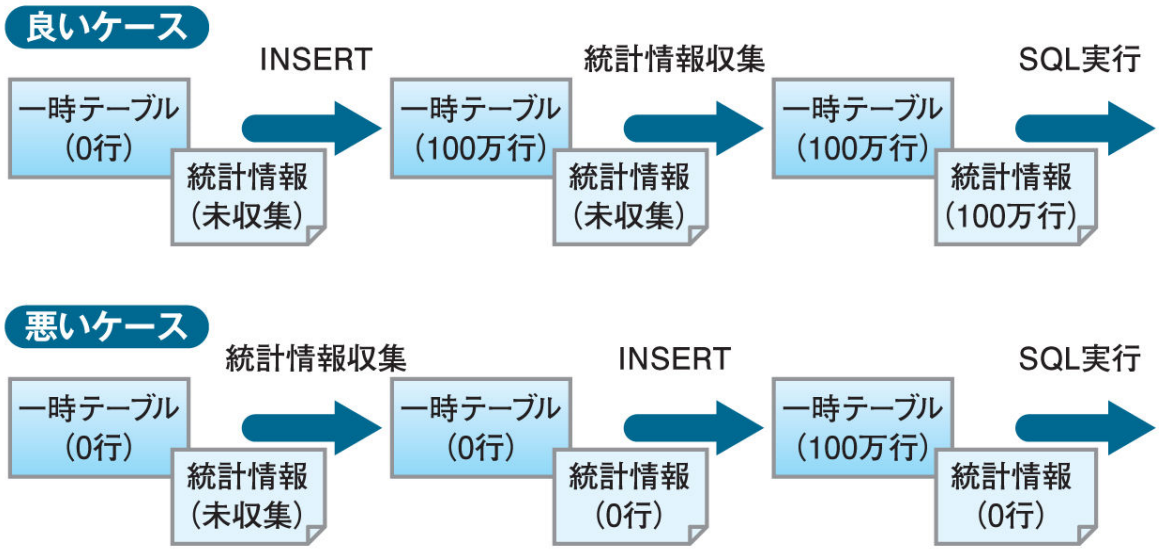
一時テーブルの欠点は次の二つです。どちらもパフォーマンスに関係します。

欠点1 統計情報の収集タイミングが難しい。
欠点2 物理的なI/O分散が難しい。

欠点1 統計情報収集のタイミング

リレーショナルデータベースは、SQL文を実行するときに、テーブルやインデックスに含まれるデータの統計情報をもとに実行計画（アクセスパス）を決定します。したがって、この統計情報が古く、最新のデータ状態と乖離がある場合、最適な実行計画が作成されず、性能問題の原因になります（第6章参照）。

一時テーブルは、通常は空っぽで1行のレコードも持っていません。したがってこの状態で統計情報を収集しても無意味です。それどころか、大量のレコードが投入されたにもかかわらず、DBMSが0行と判断して実行計画を組み立てることは、はっきりと有害です。このことから、**一時テーブルの統計情報は、必要なデータがINSERTされた後に収集する必要がある**、ということになります（図G）。



図G●統計情報収集のタイミング

これだけなら「一時テーブルにデータがINSERTされた後に統計情報を収集するように注意すれば良いだけでしょ？」と思うかもしれませんが、その統計情報収集も、テーブルの件数が巨大な場合にはそれなりに時間のかかる処理であるところが悩ましいのです。一時テーブルにデータを入れるときは、すぐにそのデータを使いたいときが多いものですが、入れてすぐ使えるわけではなく、統計情報収集という処理を一段間にはさむことを前提に、全体としての処理時間を計算しなければなりません。

欠点2 一時テーブルの物理的なI/O分散

一時テーブルのもう一つの問題は、このテーブルのI/O分散設計が難しいことです。第2章で学習したように、リレーショナルデータベースの情報は5種類のファイルによって管理されています。おさらいも兼ねて、もう一度列挙しておきます。

❶ データファイル

❷ インデックスファイル

❸ システムファイル

❹ 一時ファイル

❺ ログファイル

さて、ここで質問です。一時テーブルは上記五つのファイルのうち、どこに保存されるのでしょうか？

これはDBMSの実装にも依存するので一概に断言することはできませんが、ほぼ間違いなく❹ の一時ファイルです。つまり、通常の永続的なテーブルが作成される❶ のデータファイルではない、ということです。

第2章では、データファイルと一時ファイルは、I/O負荷を分散するために、ディスク（RAIDグループ）を分離すべきであることを学びました。しかし、一時テーブルだけは、問答無用に一時ファイルに割り当てられてしまうのです。通常は、データファイルに最も高性能なスペックのストレージを割り当てますが、一時テーブルはその恩恵にあずかれない、ということになるわけです。

以上のように、一時テーブルは二重の意味で性能的な問題を抱えています。著者は、一時テーブルの使用そのものはグレーノウハウだとは思いませんが、「一時テーブルを使うときには、かなり性能的に厳しい条件下での設計が求められる」ということを覚えておいてください。

勘どころ 62

一時テーブルは性能的に不利な条件での戦いを強いられる。

なお、データマートを作成する手段であるビュー、マテリアライズドビュー、一時テーブルの特徴を一覧表にまとめておきますので、設計の際の参考にしてください。

■ビュー、マテリアライズドビュー、一時テーブルの特徴

	データ鮮度	パフォーマンス	データを格納領域	データの保持するか
ビュー	高（リアルタイム）	低	しない	-
マテリアライズドビュー	中（更新タイミング次第）	高	永続的に保持する	任意（通常はデータファイル）
一時テーブル	中（更新タイミング次第）	中（物理設計と統計情報に注意）	一時的に保持する	一時ファイル

なにしろ、最初のトリガーを契機に最後のテーブルまで一気にトリガーが実行されるので、トリガーを解除しないと、途中のデータ断面を作ることができません。

『Oracle Database 2日で開発者ガイド 11g リリース1（11.1）』「5 トリガーの使用」。

解答（演習問題➡294ページ）

演習9-1　木構造を扱うモデルの正規形

解答は以下のとおりです。

- 隣接リストモデル　：第5正規形
- 入れ子集合モデル：第3正規形
- 入れ子区間モデル：第3正規形
- 経路列挙モデル　：第3正規形

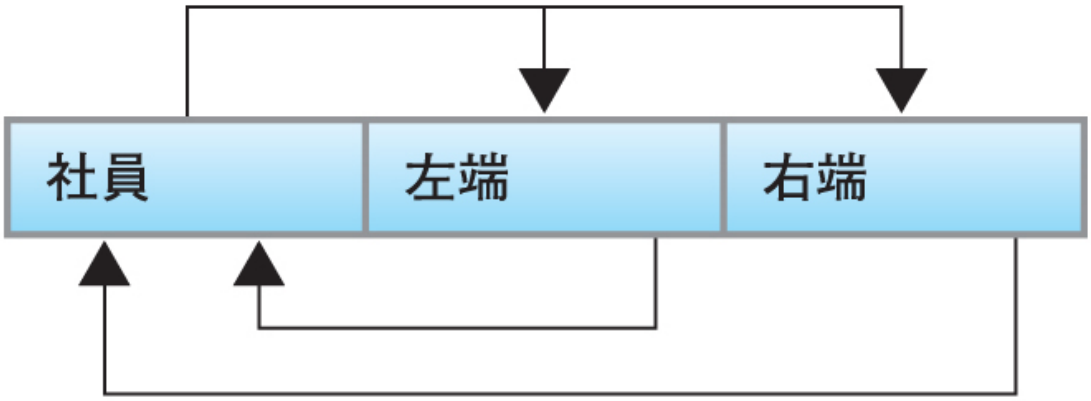
解説

どのモデルのテーブルにおいても、部分関数従属も推移的関数従属もないため、第3正規形は満たしています。

隣接リストモデルは、非キー列からキー列への関数従属はないため、ボイス－コッド正規形も満たしています。また、そもそも多値従属も存在しないため、第4 と第5正規形も満たしていることがわかります。

入れ子集合モデルと入れ子区間モデルは、座標列（左端、右端）のデータ型が異なるだけで、テーブルの論理的構造は同じため、正規形については同じものと

して考えられます。この二つのモデルの場合、「社員」列だけでなく、「左端」列や、「右端」列も主キーとして利用することが可能です（図H）。つまり、**候補キー**です。二つの異なる円は、決して同じ左端（または右端）の座標を使いません。したがって、理論的には「左端」列や「右端」列を主キーとして使うことも可能なのです。



図H●「左端」や「右端」の列も、「社員」列を規定する

これが意味することは、このモデルのテーブルはボイス－コッド正規形ではない、ということです。しかし、このテーブルを分解する必要はありません。このままの形式でも、更新時の不整合が発生することは、業務ロジック上考えにくいからです。このテーブルに社員を登録する際には、左端や右端の座標は決まっているのが普通ですし、もし仮に決まっていなかったとしても、ひとまず座標をNULLとして登録することはできます。また、異動によって社員の組織内での位置が変わった場合にも、複数行に対する更新は論理的には不要です［※11］。

逆の方向で考えてみても良いでしょう。「このテーブルをもし分解するとして、一体どのように分解するのか」ということです。元のテーブルは3列で構成されているので、

分解の仕方を網羅すると、以下のとおりです。

❶ {社員}、{左端, 右端}
❷ {社員}、{左端}、{右端}
❸ {社員, 左端}、{社員, 右端}
❹ {社員, 左端}、{右端}
❺ {社員, 右端}、{左端}

しかし、このいずれの分解方法にも問題があります。まず、❶、❷、❹、❺に共通しているのが、元のテーブルに復元できないことです。これは結合キーに使える列がないためで、つまりこれらの分解には**結合従属性**がありません。実際の使用にもたえません。

唯一、元のテーブルに復元できるのが❸ ですが、分解してできた{社員, 左端}も{社員, 右端}もいまだ**ボイス－コッド正規形になっていない**ため、分解する意味がありません。

以上の理由から、このテーブルには適当な分解方法はない、ということがわかります。裏を返すと、入れ子集合モデルおよび入れ子区間モデルのテーブルは、分解してはならない、ということなのです。

経路列挙モデルについても、同じ理由でやはりボイス－コッド正規形ではありません。主キーである「社員」列は、非キー列である「経路」列に関数従属するからです。しかし、やはり元のテーブルを分解する必要はなく、また分解するべきでもありません。

演習9-2　実数のデータ型

いずれのDBMSもREAL型という実数を格納するためのデータ型を持っています。もちろん、厳密な実数値を格納できるわけではなく（実数は無限小数を含むのでそれは不可能）、あくまで近似的な概数型です。論理は、物理の制約から自由ではないのです。

入れ子集合モデルの場合、座標に整数を使うというモデル上の制約によって、無関係の社員に対して更新が発生してしまいますが、これは正規形とは別の問題です。

おわりに

本書では、データベースの論理設計と物理設計のせめぎあい（トレードオフ）を軸に、様々な観点から設計について考察を行なってきました。特にデータベースの場合、問題になるのが、広い意味でのデータの整合性とパフォーマンスのトレードオフです。正規化を厳密に行なって整合性を高くすればするほど、検索SQLのパフォーマンスは悪化します。かといって、非正規化やバッドノウハウで示した数々の方法によってパフォーマンスを最大化しようとすれば、データのリアルタイム性は低下し、更新異常のリスクが増えていきます。

近年では、「ビッグデータ」という言葉に象徴されるように、データベースで扱うデータ量は加速度的に増えてきています。かつてはデータベースのデータ量の単位は、GB（ギガバイト）が当たり前でしたが、現在ではTB（テラバイト）クラスのデータベースも珍しくありません。さらにその上の単位であるPB（ペタバイト）を基本に話をする日も、決して遠い話ではありません。そのとき、DBエンジニアはますますシビアなトレードオフの間で頭を悩ませる可能性もあります。

DBエンジニアの（少なくとも現在における）仕事は、こうした相反するベクトルの間の均衡点を探し出すことにあります。ただDBサーバーを構築すれば終わり、ではないのです。しかも実際の開発においては、ベクトルはこの二つだけではありません。「予算」とか「ユーザーからの要求」とか「使っているDBMSの機能上の制限」とか、第3、第4の方向を持ったベクトルまで考慮しなければなりません。「あちらとこちら」どころではなく、そちらもどちらも考えて、多次元ベクトルの方

程式を解くことが、エンジニアには求められます（一般にエンジニアはこういう仕事を「調整」とか「すり合わせ」と一言で表現しますが、その内実はこういうことです）。

これを聞いて、「面倒だなあ」と思う人も多いでしょう。「複雑な物理レベルのことなんか考えたくない。自分は論理レベルのきれいな世界のことだけ考えて生きていきたいんだ」と。

この考え自体は、決して間違っていないと、著者は思います。こういう姿勢を「怠惰」と見なす人もいますが、人間はみな、その本性として怠惰です。本当なら、誰も物理層のことなんか考えたくない。その思いは、システムのユーザーのみならず開発するエンジニアが持ったとしても、決しておかしくも恥ずかしくもないものです。DBエンジニアだって、許されるならば論理レベルより上のことだけを考えて生きていきたいのです。

「最後の最後になんてことを言うんだ、それは本書が説いてきた主張と正反対ではないか！」という声が聞こえてきそうです。しかし、もしそのように思ったなら、それは誤解です。本書のメッセージを正確に理解できていません。著者は、条件次第で整合性を犠牲にする設計の必要性は認めます。しかし、それはあくまで特例であって、特別な事情のない限り、論理設計は物理設計に優先するべきです。それは、正規化と非正規化について語った章をはじめとして、本書で繰り返し述べてきたことです。人生において妥協が必要な局面があることを認めつつ、問題に当たったとき、いきなり妥協を考える精神を身につけてほしくないのです。

またここには、論理設計を物理設計に優先すべき、もう一つの理由があります。それは、パフォーマンスをはじめとする物理層に起因する問題は、いずれハードウェアによる解決が可能になっていくことが予想されるからです。先ほど、データベースに保持するデータ量は爆発的に増えるであろうという予測を述べました。この予測は、ほとんど外れようがありません。したがって、データベースに求められる性能要件もどんどんシビアになっていくこと自体は間違いありません。

著者が言いたいのは、その要件に対して、論理設計を犠牲にして応える必要はなくなるはずだ、ということです。現在のリレーショナルデータベースとハードウェアは、論理と物理のレベルをきれいに分離できていません。そのため、論理での設計が物理レベルの構成にまで影響してしまっています。これは本来あってはならないことで、3層スキーマによる独立性も何もあったものではありません。

そして同時に、現在のDBMSは概念スキーマと内部スキーマの分離を完全には実現できていない、ということでもあります。見た目上、利用者からは内部スキーマを隠蔽したとしても、パフォーマンス悪化という形で、物理層の「ファイル」という武骨な岩肌が顔を覗かせる。理想を言えば、そうした問題は物理層において解決すべき問題であって、論理層を汚すべきではないのです。

しかし、ハードウェアやDBMSのベンダーも状況を座視しているわけではありません。ご承知のとおり、IT業界におけるハードウェアの性能改善は日進月歩ならぬ秒進分歩であり、DBMSも次々に性能改善の機能拡充が行なわれています。こうした人々の努力によって、将来的には論理設計と物理設計をきれいに切り分けて考えられる日がきっと来るでしょう。著者は、その点については楽観的で

「最後は物量作戦で勝てる」と考えています。そのとき、私たちエンジニアの悩むトレードオフは、少し小さくなっているはずです。

さて、未来のことにまで及んで、少し話の規模が大きくなりました。ここで、本書を読み終わった後、さらにみなさんが上級のデータベース設計について学んでいくための書籍案内をしたいと思います。

【参考文献案内】

C.J.Date『データベース実践講義』（オライリー・ジャパン、2006）

人呼んでリレーショナル原理主義者、クリス・デイト。本書でも何度もデイトの文章を引用したので、彼の主張の一端はお伝えできたと思いますが、この本は一切の妥協を認めることなく論理設計の優位性を説いた一冊です。本書において「論理設計が物理設計に優先する」という著者の主張を原理主義的に感じた人もいるかもしれませんが、それはとんでもない話で、この本を読めば本当の原理主義に出会うことができます。

デイトはリレーショナルデータベースの生みの親であるエドガー・F・コッドと長年の親交を持つエンジニアで、リレーショナルデータベースの誕生から現在までを知る生き字引でもあります。したがって、リレーショナルモデルとその設計理論に関する記述は極めて正確。やや理論的で硬いことと、その偏った（というと大先達に怒られるでしょうが）思想がなければ素晴らしい論理設計の教科書なのですが、本書を読み終わったみなさんならば、臆することなく挑戦できると思います。

Joe Celko『Joe Celko’ s SQL Programming Style』（Morgan Kaufmann、2005）

デイトの本が理論寄りの教科書であるのに対して、より実践的な内容を含んでいるのがこの本です。ジョー・セルコもまた優れたDBエンジニアとして世界的に知られており、特にSQLの秀逸なテクニックを解説する書き手として一家をなしています。

この本は、そんな彼がデータベース設計とSQLのコーディングスタイルについて論じた一冊です。セルコもまた、「非正規形は認めない」「サロゲートキーはリレーショナルモデルのキーにはなりえない」など、論理を物理に優先させる、デイトと共通の姿勢を持っています（というより、まともなリレーショナルデータベースの理論家でそのような姿勢を持たない人は存在しません）。

タイトルに「SQL」とついていることからもわかるとおり、どちらかというと、データベースの設計というよりもSQLのコーディングスタイルについての注意点に主眼が置かれています。そのため「正規化／非正規化」のような「ザ・論理設計」な話には紙数が割かれていません。カーニハン＆パイクの往年の名著『プログラミング作法』のSQL版、と言えばイメージが湧くでしょうか？ 若い読者には、たとえが古すぎるかもしれませんが……。

非常に良い本で、本書を書くうえでも大いに参考にさせてもらった一冊ですが、残念ながら邦訳はまだありません。

Bill Karwin『SQL Antipatterns』（Pragmatic Bookshelf、2010）

「アンチパターン集」というタイトルがすべてを物語っているとおり、SQLコーディングおよびデータベース設計におけるアンチパターンを集めた本です。全編が、本書第7章と第8章だけで構成されている、と考えてもらえば良いでしょう。

本のタイトルに「SQL」とついていますが、セルコの本よりかなり論理設計に踏み込んだ内容にもなっています。「木構造を扱うときは入れ子集モデルや経路列挙モデルを使え！」と主張するなど、進取の精神にあふれた米国人らしくかなり先進的なことを言っていて、「私はそこまで言い切れないなあ」という感想も持ちました（なぜ著者がそこまで言い切れないのかは、本書第9章を

読んだみなさんはご存知でしょう。著者もまだ、物理層の制約から自由になれていないのです）。ともあれ記述も平明で手軽に読める良い本です。

惜しむらくは、この本も邦訳がありません。

Joe Celko『Joe Celko's Trees and Hierarchies in SQL for Smarties, 2nd Edition』（Morgan Kaufmann、2012）

本書第9章で学んだ入れ子集合モデル（≒入れ子区間モデル）と経路列挙モデルについて詳細に解説した本です。特に、入れ子集合モデルと入れ子区間モデルはセルコ自身が考案したモデルであることもあって、非常に力の入った解説を読むことができます。世界には、読むと感動を覚える技術書というのがまれにありますが、この本はその数少ない一つです。

この本も邦訳はないのですが、手前味噌ながら著者のサイトでも上記モデルの解説を行なっていますので、そちらも参考にしていただければ幸いです。

- SQLで木と階層構造のデータを扱う（1）——入れ子集合モデル
  http://www.geocities.jp/mickindex/database/db_tree_ns.html

- SQLで木と階層構造のデータを扱う（2）——経路列挙モデル
  http://www.geocities.jp/mickindex/database/db_tree_pe.html

ミック『達人に学ぶ SQL徹底指南書』（翔泳社、2008）

さらに手前味噌を続けます。この本は、本書のSQL版として書いたもので、SQL初級者が中級者へステップアップするためのコツをテーマにしています。本書では、SQL の詳細解説を行なわなかったので、もしかすると、例に挙げた

SQLを難しいと感じた方もいるかもしれません。そのような場合は、この本を参照していただけば疑問が解けるでしょう。本書で取り上げたSQLはすべて、この『指南書』を読むことで理解できるものばかりです。

　また、この本は翔泳社のWebマガジン「CodeZine」で連載していた記事「達人に学ぶSQL」がもとになっているので、以下のサイトで一部を無料で読むこともできます。

- 達人に学ぶSQL（CodeZine）
  http://codezine.jp/article/corner/51

　以上、デイトと著者の執筆したもの以外はすべて英語の本ですが、何の分野につけても上級レベルのテキストは、まずは英語で著わされるというのが21世紀の常識です。みなさんにおかれましても、英語の研鑽は基礎トレーニングとしてたゆまず続けていただければと思います。

　最後になりましたが、本書の執筆に当たり尽力をいただいた翔泳社の編集者、片岡さんにお礼を申し上げます。遅筆なうえにたびたび説明不足に陥る著者に付き合いながらの編集は骨の折れる仕事だったと思いますが、粘り強い仕事ぶりに深く感謝いたします。

索 引

数字

A

B

D

FK 129, 130

HaaS 40

I

MySQL 10

R

TPS 38

UNION 216, 217

UNION ALL 244

Web3層モデル 23

Webサーバー 23

XMLDB 7

XMLデータベース 7

あ

い

ウォーターフォールモデル 12

エンティティ 30

お

か

関連エンティティ 109

関連実体 136

き

け

こ

さ

し

す

せ

た

ち

稠密性 276

て

と

な

に

二次元表4, 70

ノード 169, 264

は

ひ

ふ

へ

ほ

ま

み

ミラーリング 43

無損失分解 97, 116

モジュール 4

モデル 6

ら

ラージオブジェクト 52

り

る

れ

ろ

著者紹介

ミック

SI企業に勤務するDBエンジニア。主にDWH/BI分野のデータベース構築とパフォーマンスチューニングを専門としている。自身のサイト「リレーショナル・データベースの世界」でデータベースとSQLについての技術情報を公開している。

著書 **『達人に学ぶSQL徹底指南書』（翔泳社、2008）**
『SQL ゼロからはじめるデータベース操作』（翔泳社、2010）

訳書 **ジョー・セルコ『SQLパズル第2版』（翔泳社、2007）**

●装丁　　轟木亜紀子（株式会社トップスタジオ）

たつじん　まな　ディービィーせっけいてっていしなんしょ
達人に学ぶDB設計徹底指南書

～初級者で終わりたくないあなたへ

2012年 3月15日　初版 第1刷発行
2022年 6月10日　初版 第15刷発行

著　者　　ミック

発行人　　佐々木 幹夫
発行所　　株式会社 翔泳社（https://www.shoeisha.co.jp）

ISBN978-4-7981-2893-1
2022年10月26日　電子書籍版発行

表2-2●三つのバックアップ方式の特徴

名称	フルバックアップ	差分バックアップ	増分バックアップ
バックアップ対象データ	すべて	前回のフルバックアップからの差分	前回の任意のバックアップからの差分
バックアップのトータル処理時間	大	中	小
リカバリのトータル処理時間	小	中	大
メリット	・バックアップリカバリ運用が簡単 ・リカバリ時には一つのファイルが必要	・何かにつけ中間的 ・リカバリ時は二つのファイルが必要	バックアップファイルのサイズが小さい
デメリット	・リソースへの負荷が大きい ・サービス停止が必要		リカバリ時にすべてのファイルが必要

各DBMSでの呼び名	Oracle	全体バックアップ	差分増分バックアップ	累積増分バックアップ
	SQL Server	完全バックアップ	差分バックアップ	トランザクションログバックアップ
	DB2	フルバックアップ	増分バックアップ	デルタバックアップ
	PostgreSQL	ベースバックアップ	特になし	
	MySQL	フルバックアップ	増分バックアップ	